BeAvis Car Rental System

By: Brianna Garcia, Jose Arroyo Redondo, Nathan Moreno, Gabriel Magana

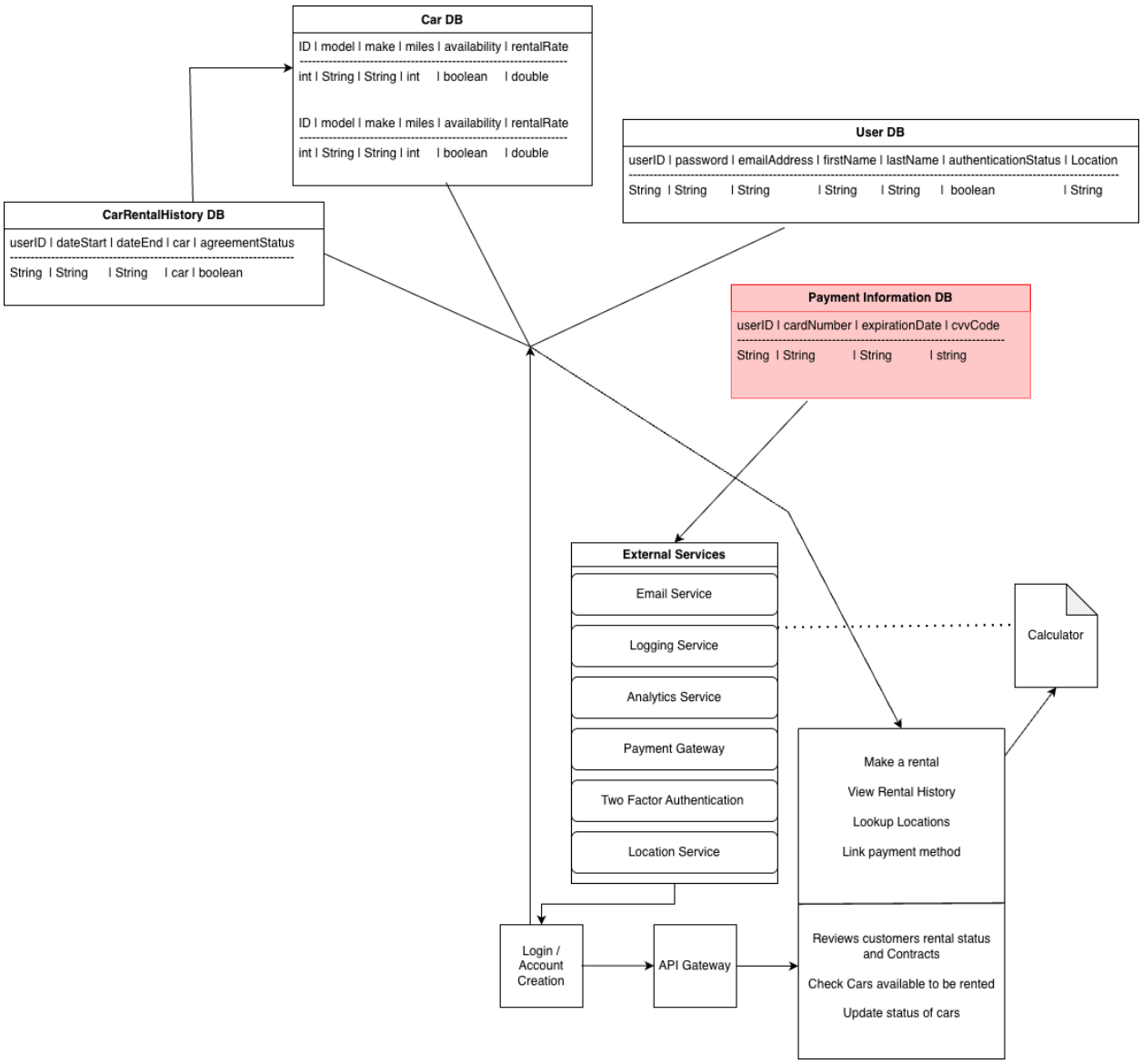
System Description

The car rental system software is aimed to be a substitute for the traditional car rental process that is made manually on paper and requires both people (contractor and client) to be present physically at the car rental office. This software will be efficient as contracts can be signed from anywhere with a connection to the network, with no need to visit the rental place.

The software is accessible either from a mobile application or a website. This means it has to be compatible with the newest version of the host OS. In addition, location services are required. This software will also require credentials from the user to rent a car, and will be as intuitive as possible to avoid users reaching a deadlock. For instance, it will keep a basic login/register process with an option of two-step authentication activation. Users can enter their payment information, which will be kept secure from other information. Employees can check rental history, view car status, and update car status if needed. Security is key, which is why the software will be enforced with security methods that assure the protection of all the client's data. Finally, employees will also have access to the software to monitor the requests and manage the system.

Software Architecture Overview

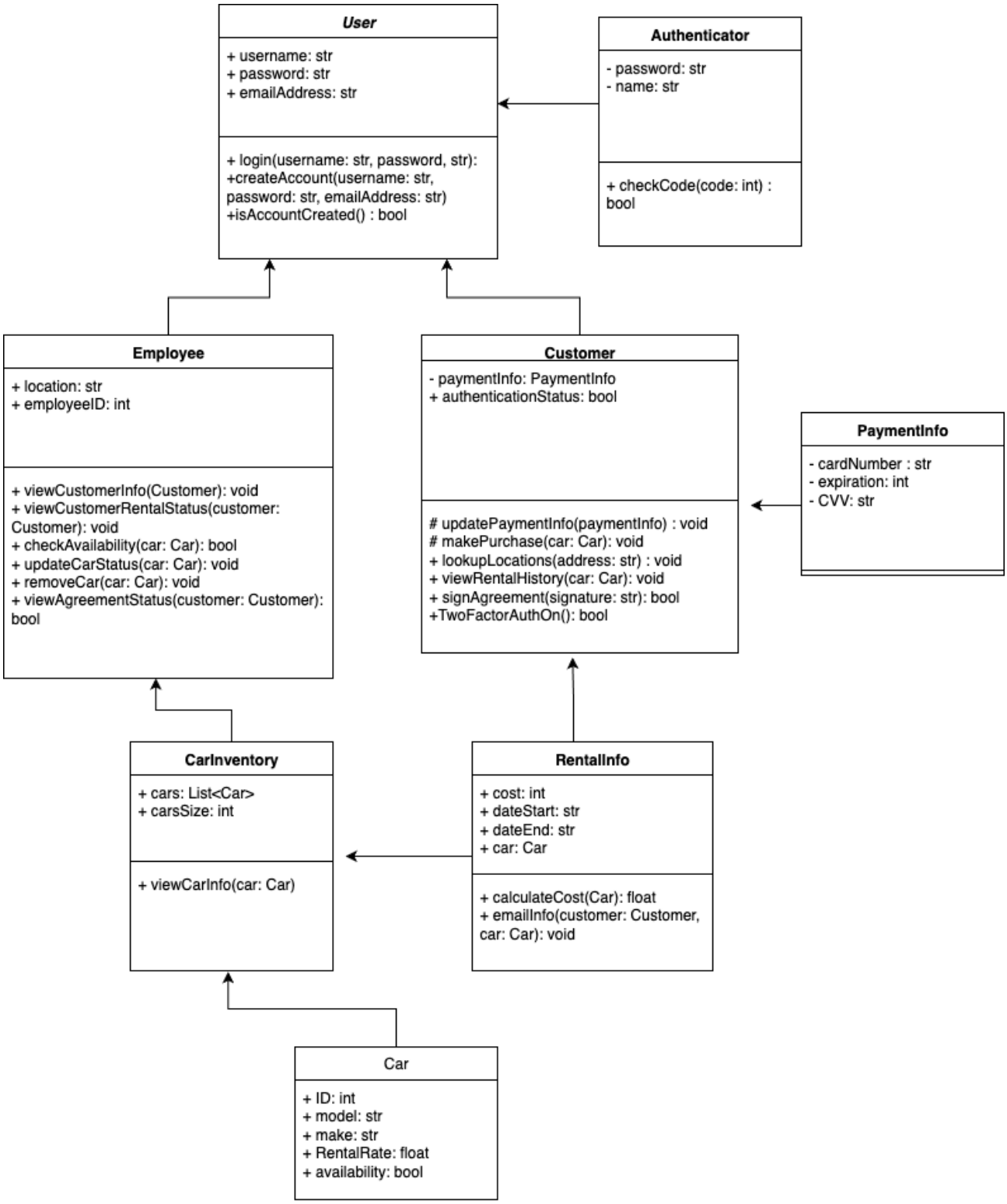
- Architectural Diagram of all major components



- SWA Description

The architectural diagram illustrates how the major components are interconnected. Starting from the Login/Account Creation, it has access to the database, which includes customer records, user information, and contracts. Such privileges and access are granted to those who are employees of BeAvis. The Login also connects to the API Gateway, which is a service that verifies users' authorization to access the software. The API Gateway connects to the services the software provides: Make a rental, view rental history, look up locations, and link payment methods. The listed services are those accessible to customers. The other services shown are only accessible by employees, such as reviewing customers' rental status and contracts, checking car availability to be rented, and updating the status of cars. All these services are then connected to the calculator, which calculates the total amount the customer will be charged for the rental car. The external services are: email service, logging service, analytics service, payment gateway, two-factor authentication, and location service. The external services are connected to Log-in/Account Creation because they'll be catered to the particular users' settings and previous activity. The payment information database is extra secured, and has access to the external services to access the payment gateway. The external services should have a calculator to determine the total cost the customer will be charged for the car rental. The database is only accessible to employees and is in a separate database to prevent the chances of a data breach. The other database, which holds three different databases, connects to the services that are accessible to customers and/or employees because it retrieves customer information from the services they use, and it is stored in the appropriate database.

- UML Diagram


User
+ username: str
+ password: str
+ emailAddress: str
+ login(username: str, password, str):
+createAccount(username: str,
password: str, emailAddress: str)
+isAccountCreated() : bool
Authenticator
- password: str
- name: str
+ checkCode(code: int) :
bool
Employee
+ location: str
+ employeeID: int
+ viewCustomerInfo(Customer): void
+ viewCustomerRentalStatus(customer:
Customer): void
+ checkAvailability(car: Car): bool
+ updateCarStatus(car: Car): void
+ removeCar(car: Car): void
+ viewAgreementStatus(customer: Customer):
bool
Customer
- paymentInfo: PaymentInfo
+ authenticationStatus: bool
# updatePaymentInfo(paymentInfo) : void
# makePurchase(car: Car): void
+ lookupLocations(address: str) : void
+ viewRentalHistory(car: Car): void
+ signAgreement(signature: str): bool
+TwoFactorAuthOn(): bool
PaymentInfo
- cardNumber : str
- expiration: int
- CVV: str
CarInventory
+ cars: List<Car>
+ carsSize: int
+ viewCarInfo(car: Car)
RentalInfo
+ cost: int
+ dateStart: str
+ dateEnd: str
+ car: Car
+ calculateCost(Car): float
+ emailInfo(customer: Customer,
car: Car): void
Car
+ ID: int
+ model: str
+ make: str
+ RentalRate: float
+ availability: bool

- **Description of classes**

- **User**: Is the main base class for all users in the system. It stores general information such as username, password, and email address. This class also manages the basic account functions, including creating an account and logging into the system.
- **Authenticator:** a class that handles the security part of the system. It is responsible for checking authentication codes and managing two-factor authentication when users choose to enable it. This class helps make the login process more secure.
- **Employee**: It is a class that represents the workers of the car rental company. It inherits from the User class, meaning it has the same basic account details but also adds employee-specific information, like their location and employee ID. Employees can view customer information, check car availability, and update car or rental statuses.
- **Customer**: represents people who rent cars from the company. It also inherits from the User class and adds extra features such as storing payment information and managing authentication status. Customers can make rentals, view their rental history, and update their payment details.
- **PaymentInfo**: stores all the customers' payment details. This includes the card number, expiration date, and CVV code. It ensures that sensitive payment data is handled and stored securely.
- **CarInventory**: class keeps track of all cars available in the system. It stores a list of car objects and the total number of cars. Employees and customers can use this class to check which cars are available for rent.

- **Car**: describes each vehicle in the system. It includes details such as the car's ID, model, make, rental rate, and whether it is available. This class helps the system identify and manage each car.
- **RentalInfo** is the last class, and it stores all the details related to a specific rental transaction. It keeps the rental cost, the start and end dates, and the car being rented. It also includes methods for calculating rental costs and emailing rental details to the customer.

● Description of attributes

The User class has three main attributes: "username", "password", and "emailAddress". These store the basic login information for any person using the system, whether they are a customer or an employee. They make it possible for users to create an account and access their personal data securely.

The Authenticator class includes attributes like "password" and "name". These are used to identify the authentication method and to verify codes during the login process. This helps improve the security of the system by allowing two-step verification.

The Employee class adds two extra attributes: "location" and "employeeID". The location represents the branch or office where the employee works, while "employeeID" is a unique number that identifies each staff member. These attributes help organize and track employees within the system.

The Customer class includes "paymentInfo" and "authenticationStatus". The "paymentInfo" attribute links to the customer's stored payment details, while "authenticationStatus" shows whether two-factor authentication is turned on. These help ensure smooth and secure transactions.

The PaymentInfo class contains the attributes “cardNumber”, expiration, and “CVV”. These store the customer’s card details safely. The system keeps this information separate from other user data to reduce the risk of security issues.

The CarInventory class has two attributes: “cars” and “carsSize”. The “cars” attribute is a list that stores all the cars in the system, and “carsSize” shows how many cars are in the inventory. These attributes make it easy to manage and update the list of vehicles.

The Car class includes several important attributes such as “ID, model, make, rentalRate, and availability”. These describe each car in detail — what brand and model it is, how much it costs per day, and whether it is currently available for rent.

Finally, the RentalInfo class has attributes like “cost”, “dateStart”, “dateEnd”, and car. These keep track of how much the rental costs, when it starts and ends, and which car is being rented. They help record and manage all rental transactions in the system.

- **Description of operations**

In the User class, there is the function login, which takes the username and password, both strings, as parameters. This function is used for employees and the manager to log into the system, and if their logins are recognized, they can access the system. The create Account function takes the parameters username, password, and email address, all strings, that allow a new user to use their information to create an account with the system. Their login information will then be stored in the system so that they do not have to create an account again. The function isAccountCreated() is used to demonstrate whether the account has been created successfully or not. All

functions in this class are public.

In the Authenticator class, there is the function called check code, which takes in an integer code as a parameter and ensures that the 2-factor authentication code that the user inputs matches the one that the system generated. This adds another layer of security to the user's account. This function is public since the Authenticator class is used by the User class.

The Employee class has a function called viewCustomerInfo(), which takes a Customer as a parameter to retrieve information about the customer, which includes their payment details and authentication status. The viewCustomerRentalStatus() that takes in Customer as a parameter is used to check whether they are renting a car or returning a car. The checkAvailability() function takes in a car as a parameter, which checks to see what cars are available to rent to a customer, ones that are not already rented. The updateCarStatus() takes in a car as a parameter and is used to mark the car as either being rented or available. The removeCar() function takes in a car as a parameter and is used to remove the car from the list of available cars to be rented out to customers. The viewAgreementStatus() function takes in the customer as a parameter, which is used to view the documents signed by the customer before renting a car. All functions are made public.

The Customer class has a protected function called updatePaymentInfo() which takes in a paymentinfo instance, which allows a customer to update their card number, expiration date, and ccv. The function makePurchase() is a protected class and takes in a car instance that the customer wishes to buy and uses their stored information to allow the customer to complete the process. lookupLocations() is a public function takes in the user's location using the

external location service and shows the user a map of all nearby car rental locations. viewRentalHistory() is a public function that takes in a car instance that the user wishes to see past uses from other customers and reviews. signAgreement() is a public function that takes in a string as the customers signature and uses it as a signature to agree to the terms and conditions of the purchase. TwoFactorAuthOn is a public function that checks if the customer has 2-factor authentication enabled before allowing them to be logged in from just their username and password.

The Car Inventory class has a public function called viewCar() that takes in a car as a parameter. This is used to check the attributes of the car, such as the ID number given to the car, model, make, the rental rate, and availability.

The RentalInfo class has a public function calculateCost() which takes in a car instance and uses that data to show the user the cost of renting that car. emailInfo() is a public function that takes in a customer instance and an instance of a car that the customer is purchasing, and emails them their confirmation details and receipt.

Development plan and timeline

Overall, the Car Rental System is used to provide an efficient and faster way for customers to rent a car, and employees to keep track of records, along with other services.

Tasks:

- Write system overview: **Jose**
    - Estimated timeline to complete: 1 day
- Create Software Architecture Diagram (SAW): **Gabe/Brianna**
    - Estimated timeline to complete: 1 day
- Write an overview of the SAW Diagram: **Brianna**
    - Estimated timeline to complete: 1 day
- Create UML Diagram: **Nathan**
    - Estimated timeline to complete: 1 day
- Write a description of classes, attributes: **Jose**
    - Estimated timeline to complete: 2 days
- Write a description of operations: **Brianna/Gabe**
    - Estimated timeline to complete: 2 days

Verification Test Plan

Function: viewCustomerInfo(customer)

Unit test:

Input: Customer object with username, password, email, payment information, and authentication status fields
Expected output: Should display customer Name, Email, Payment information, and Authentication status
Test: This test attempts to display the information of a customer when an employee needs to view their information. The username, password, and email will be strings, payment information will be split into a 16-digit integer card number, 2-digit expiration month, 2-digit expiration year, and 3-digit CCV

Input: Customer that does not exist
Expected output: Error, “Customer not found. Please enter a valid customer,” and the system should prompt the user to enter another customer.
Test: This unit test ensures that the system does not try to access an instance of a Customer that does not exist, possibly creating an issue.

Test: This test ensures that the system is able to output available information, even without all information present, instead of the function failing altogether when one or more pieces of information are missing.
Input: Customer with a Username (Bob2), password (BobCar2), email (bob@gmail.com), authentication status (True), but no payment information |

Expected output: Username: Bob2, Password: BobCar2, Email: bob@gmail.com, Authentication Status: True, Payment Information: Not Available.

Integration Test:

Successfully retrieving customer information

Input: Customer Object that includes their first and last name, Jose Arroyo Redondo, and their email address, jarroyoredondo2467@sdsu.edu. The systems database is then expected to have records for Jose that include his payment details, with a Visa ending in 3288, authentication status set to verified, username, and password. The employee calls employee.viewCustomerInfo(customerJose), and then Jose's information is retrieved.
Test: This is testing the feature viewCustomerInfo with a customer who is in the system's customer database, using the Employee class and Customer data. The test vectors in this case are Jose Arroyo Redondo, which then their information, such as their email, password, payment information, and authentication status, is displayed. How I selected this test to cover this feature is by selecting a customer name that is in the system, and I know that their information will be retrieved.

Invalid Customer

Input: The Customer object John Johnson, and email jjohnson5431@sdsu.edu. The system will then have no record of Johnson's information. Their

information will not be retrieved when the employee class employee.viewCustomerInfo(customerJohn). The method attempts to retrieve the information, but there is no record found in the customer database. Then the system will display “Customer information not found.”

- The targeted feature in this case is handling system errors when the customer's information is not found in the database. The test vectors are John Johnson and email jjohnson5431@sdsu.edu. The system remains stable even when errors occur.

System Test:

Successfully retrieving and displaying customers' information.

The employee logs into the system by using their username, bgarcia6421, and password, cs250sdsu. They look up a customer by their first and last name, Gabriel Magana. They then see the customer's payment information to make sure it is in the system. Other information is displayed, including: username, gmagana7398, password, cs2025, email, gmagana7398@sdsu.edu, and their authentication status, which is off. Employee clicks out of the customer's information and logs out of the system.

- This test begins with the employee logging into the system, given access that is limited to employees. They can then look up customers, which in this case was Gabriel Magana, and were able to view their information. The system correctly displayed the customer's information, including their payment information, which is a 16-digit number, with the

expiration date being 4 integers separated by a dash, and CVV being 3 digits.

Handling no user information found for Customer

The employee logs into the system by using their username, bgarcia6421, and password, cs250sdsu. They enter valid credentials and are given employee access. They look up a customer by their first and last name, Bob Smith. The system then searches in the database and finds no record of the customer. It then displays the error message “Customer information not found.”

- The targeted feature is to see how the system handles errors, which in this case was invalid customer information. The test set vectors are the name Bob Smith, which is not in the customer database. The goal was to confirm that the system displays an appropriate message when an invalid name is entered.

Data Management Strategy - **SQL**

We chose to use SQL for our databases. SQL allows for a much more intuitive implementation for a program of our scale and its needs. Because of SQL's structure around predefined schemas, such as in our system of managing cars, where we know every car and customer will follow a well-defined structure (manufacturer, model, year, MPG, list of damage, gas/electric, etc.), SQL ends up being a good fit for us. In the event of our system gaining more users than initially expected, we are able to upgrade the hardware components in our servers, such as storage and processing power, which SQL handles well.

We have chosen to utilize 4 databases: CarDB, CarRentalHistoryDB, UserDB, and PaymentInformationDB. We've done this for resource allocation as well as security.
For example, our customers' payment information is stored on a separate, much more secure database to ensure the safety of our customers' most valuable data in the event of an attack on our system.
Similarly, customer user data is stored in a secure database, albeit less secure than the payment information database, to keep customers' personal information secure.
Our car database is separate because it will receive the most traffic as people view our selection of available cars, so we will allocate much more computing resources to this database.
Lastly, we have the Car Rental History database. This is separate because, while the data does get viewed, it is mainly for the use of administrators and management to keep track of the history of each car, so we are allocating much less computing resources to it than compared to the Car database

Method we are using, Alternatives, Advantages, and Trade-offs

SQL

We will use a SQL database for our main app. SQL gives us safe transactions, so data can stay correct, also has rules and links to keep data clean, like constraints or
foreign keys, and clear queries with joins that match our work. The tools are very
common, so we can build faster and with less risk.

Advantages:

- Safe transactions: many-row updates are reliable, that way we can roll back on errors
- Strong data rules: constraints and foreign keys protect data quality
- Flexible queries: joins let us answer new questions without redesigning data
- Mature ecosystem: good tooling for migrations, backups, and monitoring
- Easy to learn: many developers know SQL because it's simpler

Disadvantages:

- Schema changes need migrations
- It is harder to scale across many servers at extreme write rates
- Mostly time-series data needs partitions or extensions
- May need a cache later for very low latency

Alternatives:

Document Stores (MongoDB, Firestore)

The first alternative is a document database. It stores data as JSON-like documents
and lets us change fields quickly without heavy migrations. This fits when each record can have a different shape.

Advantages:

- Flexible schema: add or remove fields without big migrations
- Natural JSON & nested data: great for hierarchical objects
- Fast to start: simple setup and high developer speed
- Horizontal scale options: managed services offer autoscaling and global replicas

Disadvantages:

- Hard joins and multidocument transactions, this means, cross-document work is limited or slower
- Data duplication risk because denormalization can create copies and drift
- Strong consistency can be tricky or expensive, depending on the product and setup
- Harder to learn and manage

KeyValue / WideColumn (DynamoDB, Cassandra)

We could also choose a key-value or wide-column database. It is built fora huge scale
and low latency when we know our access patterns.

Advantages:

- Massive horizontal scaling and high availability
- Predictable performance, especially at large traffic levels
- Low latency for key lookups

Disadvantages:

- Rigid data model
- Limited adhoc queries and joins
- Refactors are painful and may need expensive backfills
- Strong consistency or global tables can add costs

Graph Databases (Neo4j)

We could also choose a graph database. It stores nodes and edges to model relationships and runs multihop queries quickly.

Advantages:

- Great for relationship queries
- Expressive graph query languages, for example, Cypher
- Natural modeling for networks and permissions

Disadvantages:

- Niche skills and operations
- Not ideal for general CRUD/OLTP
- Joining with non-graph data can add complexity

TimeSeries Databases (TimescaleDB, InfluxDB)

Finally, we could also choose a timeseries database for metrics or events that are

written often and read by time windows.

Advantages:

- Fast writes and high ingestion rates
- Efficient time window queries with retention policies

Disadvantages:

- Weak for nontime data and more complicated joins
- Fewer transactional features than SQL OLTP
- Different query models that don’t fit all app reads