Software Architecture Diagram Revisions:
Compared to the previous version of our software architecture diagram, numerous improvements and additions have been made to adjust and reflect our data management strategy. First, we added an additional database to store manager information such as manager name, username, password, unique ID, work schedule, and work location. This will be useful when we create relationships between various tables of data (like creating a relationship between users and managers) in such a way that will follow data normalization, where we will specifically isolate independent multiple relationships since many managers can help and assist many customers/users. We will discuss the various strategies for this in the tradeoff discussion below. The second improvement we made to the diagram was adding two additional pages: one page that stores the directory of cars available and another that serves as a transaction/payment page where users can edit/add payment methods and check out selected vehicles in their cart. We also added more use cases in the web pages such as an option to return a car, get directions to the nearest facility where the selected car is stored, view shopping cart, and updating user log in information. Finally, we distinguished between interactions between the webpages and the interactions with the databases: the interactions and flow of the website were signified with solid arrows while interactions with the databases and other 3rd party apps/systems were illustrated in dashed arrows. This makes it easier to visualize the relationships between the different users, use cases, and elements of our website application. We also attached additional text boxes next to each arrow relationship to show the inputs and outputs of each use case in the SWA diagram as well as how this information is used to edit and/or make additional entries in our databases. For example: searching a vehicle by location first requires a “user location”, which is seen in a text book as an output of the “Search Vehicle By Location” box in the diagram. This location is found by interacting with the 3rd party GPS app connected to the software system. This “user location” is returned and used by the vehicle directory page to filter vehicles from the car rental database based on their nearest location to the user. These vehicles are found by passing “car information” from the vehicle directory page to the car rental database. These vehicles are essentially filtered based on the user location because each car entity in the database is sorted based on their location attribute, where cars with a closer distance to the user’s location will be towards the top of the vehicle directory page and cars farther away from the user will be towards the bottom.

Diagram Description And Justification For Attributes:
Our diagram utilizes a combination of SQL schema and a key-value NoSQL JSON format. The table begins with an SQL table representing Rental Car Lots in the Rental Car Lot database. The table contains 6 fields: Car Lot ID, Street, City, State, Zipcode,

and Capacity. The Car Lot ID is a unique 5 digit integer that distinguishes rental car lots from each other. Therefore each rental car lot will be given a different car lot ID from one another. The street is a string that represents the address of the car lot. The city field is a string that describes the city of where the car lot is located. The state field is a string that represents which state the car lot is located. The zipcode is a 5 digit integer where each car lot is located. It can be assumed that some car lots may be located in the same city, state, and/or zipcode, which is why each car lot should have a unique ID. Finally, the capacity filed in the Rental Car Lot SQL table is an integer value that represents the maximum number of cars available for storage. The reason why we chose these fields to represent a car lot in our database management system is because it is important for distinguishing car lots from one another as well as useful for getting directions based on where the location is located. Specifically, the unique car lot ID serves as a way to differentiate one car lot from another since each ID will be unique: no car lot can have the same ID. This will also help in normalizing our data and reducing redundancy between our SQL tables, as discussed in the tradeoff section. Storing the address, city, state, and zipcode of each car lot is necessary so that cars can be tracked and filtered by location. These fields are useful in assigning locations to cars since every car lot may store many cars, all of which require a specific location in order to be filtered by the customer based on their location. Connecting the Rental Car Lot and Cars SQL tables is a 1 mandatory to many optional relationship. This is accomplished by the Car Lot ID field stored in the Cars table in the Rental Car Lot database. This means that a rental car lot can store from 0 up to n number of cars, where n is equal to the capacity of the rental car lot. The word “stores” indicates that each rental car lot will store a certain number of cars. This relationship means that a Car Lot can store from 0 to many cars. The Cars SQL table has many fields: the VIN field stores the unique serial number string for each vehicle to be properly distinguished and identified, the Make field represents the car’s manufacturer name that stores it in the form of a string, the Car Capacity field stores an integer for the maximum number of occupants that can fit in each vehicle, the Model field is the string name of the vehicle model (e.g. Prius) by the manufacturer, the Type field stores the string corresponding to the type of vehicle (e.g. truck, SUV, sedan, etc.) for each vehicle entry, the Mileage field stores the integer number corresponding to each vehicle’s current mileage, the Status field corresponds to a string for each vehicle that indicates if the vehicle is not available, is already being driven, or is available at the specified car lot ID, the Car Lot ID field stores the unique integer that is associated with the Rental Car Lot ID (this is so that we can reduce redundancy for the number of fields in these datatables). Finally, the Color fields store the string color of each associated vehicle in each row of the table. The justification for incorporating these fields in the Car SQL table is because based on the software architecture diagram, the user will be able to filter what cars they wish to add to their shopping car based on a variety of specifications; these specifications could include the

make, model, seat capacity, type, mileage, status, and the color of the vehicle. Therefore, all of these fields must be stored in a database so that the directory of cars stored can be filtered by the user. It also helps differentiate vehicles from one another by adding more descriptive elements that may be useful to the manager in finding vehicles in the car lot. A 0 to many optional relationship arrow exists from the Cars SQL table to the Transaction SQL Table stored in the User Payment Database: this means that a Car can be involved in 0 to many (n number of) transactions according to how many transactions have been made with a specified vehicle. The Transaction SQL table has 6 fields: Transaction ID field stores the unique ID for each transaction in the form of a 6 digit integer, Flat Day Fee field stores a double value that represents the price for renting the car for one day, Extra Mileage Fee field represents the cost in dollars of type double to rent a car and drive over a certain predetermined mileage set by the manager, the VIN field provides a 17 character long string that acts as a unique ID for each vehicle, the Total Amount field describes the cost in dollars in the form of a double of the total price of the transaction after fees, taxes, and other user expenses; and the User ID is a unique integer identifier that is used to show which user was involved in the transaction. It is also useful in linking the Transaction SQL table with the Payment SQL table since every transaction uses some sort of payment information set by the user. The reason why these fields are critical to the Transaction SQL table and the database management system as a whole is because every transaction should involve a car, a user, fees, and a total amount that is due. Every car is differentiated by a unique car ID, therefore storing the car's VIN will allow us to avoid creating redundant data such as storing other information about the car in the same table. The user ID is required to distinguish the user of the transaction since it is very important for the managers to know who is purchasing which vehicle and for how much at a specific time in case of issues that arise in the rental process of a vehicle. Fees are necessary to store separately since fees may change depending on the type, make, and model of the vehicle; this would affect the total amount differently for each vehicle. Finally, a total amount field is required because it displays the cost of the entire transaction in dollars (stored in the form of a double) that will be displayed to the user on the web page; storing and displaying this information is critical to both the user (since they should know how much it costs to rent the vehicle) and other employees of the company. Multiple relationships exist from transaction: a 1 to 1 arrow connects from the Transaction SQL table to the Timestamp table; this means that one transaction should create one timestamp that will be used to indicate when a transaction has occurred. It will also be helpful for the user because, through the 3rd party Bank API based on the user's credit card, it will send this timestamp and transaction information to the databases of the user's respective bank. The Timestamp SQL table has 5 fields: Transaction ID, Day, Month, Year, and Time. The Transaction ID field stores unique IDs for each transaction in the form of a 6 digit integer. Since many vehicles may be

purchased in one transaction, storing redundant information about vehicles and fees would be redundant since it is already stored in the Transaction table in the User Payment Database. Therefore, a Transaction ID will limit the redundant information between the two SQL tables. The Day field will store the corresponding day of the week for each transaction. The Month field will store the month that the transaction took place in the form of a string. The Year field will be used to store what year the transaction occurred also in the form of a string. These fields were chosen to represent a Timestamp database object because every transaction needs some sort of data and time to indicate when a car was purchased for rental. Storing this in a separate table allows more flexibility in the database by allowing for more simpler queries to access information about dates. It also is easier to send this information without the use of search queries by breaking the Transaction table into manageable portions. Finally, the Time field will store string entries that signify the approximate time (e.g. 9:30PM) that a vehicle was rented. A 1-to-1 relationship also exists between the SQL Transaction table and the NoSQL JSON file Rental Agreement. This 1-to-1 relationship indicates that a Transaction develops 1 Rental Agreement JSON file. It is important that Rental Agreement is stored as a key-value NOSQL JSON file because Rental Agreements are usually in the form of documents, so using a key-value type of NoSQL representation allows us to create this requirement. The document form of NoSQL representation allows managers to easily visualize and send Rental Agreements to customer accounts. The Rental Agreement document contains 5 fields: Date Signed, VIN, Mileage at Rental, Total Cost, and Car Lot ID. The reason why we chose these attributes for this document NoSQL representation is because all rental agreements outline the contract that the user makes and signs when purchasing a vehicle; information such as the date the document was signed (in the form of a string) is used to update the status of vehicles as well as ensure that vehicles are returned in a timely manner; the VIN is used to store vehicles by using unique information such as the VIN that act as an identifier in the database of vehicles; the mileage at rental is used to calculate fees and the total price (double value in dollars) which is necessary in the normal operations of a business; the total cost provides both the user and managers with critical information that affect both parties, and the car lot ID which indicates where the vehicle is purchased from and should be stored to ensure proper protocols are met. Finally, a 1-to-many optional relationship exists between the Payment Info SQL table and the Transaction SQL table. This means that a user’s payment information (credit card) can create from 0 to many transactions: one user’s payment information stored on their account may have been used to rent 4 vehicles over a period of time. Meanwhile another user’s payment information may not have been used to rent any vehicles thus far. The Payment Info SQL table contains 6 fields: User ID stores integer representations of the User ID of each payment method. This is relevant to include because users are linked to transactions, so in order to indicate which user was

involved in a purchase of a rental vehicle then a user ID is required. This unique ID also ensures that transactions are properly identified since every user has a unique ID so there will be no confusion between multiple users for a certain transaction. The second field is the Card Number that stores the string representation of each customer's card number. The card number is important to store since it will be used to verify with the bank that the customer has sufficient funds to purchase a rental vehicle. The CVV field stores the 3 digit integer on the back of the customer's card to verify that the customer is the rightful owner of the card. The Type field indicates which type of payment (debit card, credit card, check, etc.) in the form of a field for each entry in the table. The Card Network stores a string for each user payment information, indicating if the card is a Visa or Mastercard. Finally, the Bank Name field stores a string for each user to provide more information about where the user's funds are located or where the transaction should be sent to (Chase Bank, Schoolsfirst, etc…). The Bank Name field is significant because it ensures that the transaction is properly sent to the right bank, and it eases communication between the third parties involved. A many mandatory to many optional relationship exists between the User JSON document collection NoSQL representation and the SQL Payment Info data table. This means that many users (either managers or customers) could have from 0 to many (up to n) payment information objects. We used a document NoSQL database to represent the users of our system. These two users are the customers (the people purchasing the rentable vehicles) and the managers (the employees of the company responsible for updating and managing transaction and inventory across car lots). Each user and manager will inherit attributes such as User ID (data type is a string), User Type (data type is a string), Name (data type is a string), Password (data type is a string), Email (data type is a string), and Phone Number (data type is a string). Every manager and user JSON in the User JSON document collection will share these attributes because the software system requires storing contact information and log in information in order to ensure that user accounts are properly established. As a result, this saves space in our NoSQL database since redundancy in these two groups can be eliminated by inheritance. The Manager JSON of the User JSON collection has extra attributes including Work Schedule and Car Lot ID. Work schedule, which is a PNG image, is critical because there will be more accountability for transactions and it eases the burden for finding the managers working at a current Car Lot ID on a specific day. Car Lot ID is also important because it allows the database management system to keep track of the current managers at a particular car lot location. This helps mitigate problems that arise with either transactions or car inventory in the future. Finally, the Customer object of the User JSON has a Credit Score (Integer), Billing Address (String), Insurance Policy (String), and GPS Current Location (Pair<double,double) attributes. The reason why there are attributes like credit score and insurance policy is to ensure that the use case for checking eligibility requirements for renting a vehicle is met. Therefore this information must be stored in our database

system. A GPS location of type Pair<double, double> is required because the user's location is tracked by the software system to determine the miles they have driven thus far and calculate the total based on how far they have driven and the location they are currently at (since prices may vary depending on where the customer is driving to).

Tradeoffs for Data Management:

Design Decisions For Databases and Justification:

We chose to utilize 7 databases to separately store critical information to our software car rental system. 5 of these databases/tables are SQL while the other 2 are a form of document JSON NoSQL database. SQL was a straightforward decision for our data management system because the types of data that we needed to store for the car lots, rental vehicles, transactions, timestamps, and payment info can be generalized in data types that are covered by the SQL syntax. The data we store is primarily relational; each car we have in the databases is tied to a car lot which is tied to a specific location, each of which has a myriad of employees. Transactions are made with payments related to a customer, etc. This makes the decision to go with SQL straightforward since the syntax is inherently relational. There are a few aspects of our system where SQL falls short of what we need. For example, when we're storing document based data, like information concerning user profiles. We define two types: manager and customer, they each have similar attributes but each have their own unique attributes. We decided to organize this type of data using JSON files and collections since SQL isn't capable of unique attributes to the same type. It also makes conceptual sense to represent users as documents. On a similar level, rental agreements work best in a NoSQL structure because that type of document often has blocks of text that wouldn't make sense to make it a String type in an SQL database. It would also make organizational sense as we're storing literal documents in this case, representing data that is already in an SQL, but with the customer in mind. We decided to use SQL databases for the car lots, the rental vehicles, the transaction, the timestamp, and and the payment info because they each encapsulate different and distinct use cases from the software architecture diagram while ensuring data normalization. First of all, we decided that car lots and cars should be in their own separate database because cars and car lots follow a distinct relationship where one car lot can store many cars. Also, a directory page is required for our software system, so it should be easily accessible for the user to browse cars without the need for long querying times if they were in the same database as a car lot. We realized that because cars and car lots will frequently be used for complex queries, it would be more effective and less time-consuming to keep them separate by indexing the Car Lot IDs in the Cars SQL database. Since many transactions are occurring at a time, using SQL over NoSQL is more beneficial due to the Atomicity and Isolation properties of SQL. If one part of our system is flawed, then the entire system should shut down because private information such as users' payment information, contact

information, and more could be vulnerable to malicious people. Also, the Durability property of SQL in this situation would ensure that these key components of customer data are preserved in the event of failure because users would have to create new accounts everytime the database system experienced a failure. Storing important information such as car capacity is necessary in the operation of our software system because the storage, retrieval, and available cars for customers all depend upon the capacity of the car lot. For example, if a car lot is full and a customer is trying to return a vehicle they just rented, then it would be more efficient for the capacity of the car lot to be integrated into the web application and alert the user that they must travel to the next closest car rental lot; this would be better for both parties since if this value was not stored for each car lot, efficiency would be lost since managers would have to manually update the current capacity of the car lot while having to physically deal with customers. For storing the various vehicles in our software system, we opted for an SQL table database because complex queries like filtering transactions and searching databases on a specific input from the user (like model of car, color, type of car, etc). We chose to integrate these fields into the table of our database because our software architecture diagram outlines a use case for searching and filtering vehicles from a wide variety of vehicles. Thus, SQL would be more efficient for performing queries on already normalized data. Our data is normalized since we have already reduced repeating groups and we have reduced redundancy within the tables themselves. We believe that the attributes in the Cars SQL table provides concise enough information to perform queries based on user input while ensuring that vehicles can be grouped together based on shared attributes. These list of attributes provide more characteristics for the user to select and filter from.

Alternatives and Tradeoffs:

One alternative could be to use a key value NoSQL database in place of the Transaction SQL database. Implementing a key value NoSQL database here would provide greater flexibility in making adjustments to the data; unlike NoSQL, making changes to the Transaction database would require altering the schema and using SQL commands to link the changes. The reason why we chose to utilize a SQL database for Transactions is because SQL is more suited for higher volumes of data and allows for more complex queries. SQL's property of Atomicity under ACID ensures that either all of the transactions are valid or none of them are. This helps mitigate risks of unauthorized or unintended purchases of rental vehicles. Finally, since the website is expected to receive large amounts of traffic, an SQL database for storing transactions would ensure robustness, consistency, and flexibility across platforms.

Another alternative could be to implement a column-wide/column store for the Transaction SQL table. The benefits of using a column-wide/column store would be so

that car data entries would be expanded horizontally; this ensures that data can be accessed much quicker and more efficiently by reducing query time. The justification for implementing a NoSQL column-wide/column store database for cars would be to ensure that users can filter and access cars in the directory page at a much faster rate. NoSQL would also be a great alternative since NoSQL's CAP properties (specifically Consistency) would ensure that data is available even during outages; this would be helpful if cars had null values or if other issues arose that affected the characteristics of these vehicles on the application. We instead opted for an SQL database design for representing cars because NoSQL doesn't guarantee durability under ACID; this is more beneficial since we want our car data to be visible even if our system malfunctions because we want customers to purchase vehicles despite failures in our system. Without this, customers would lose the ability to browse and purchase vehicles which would result in the loss of revenue.