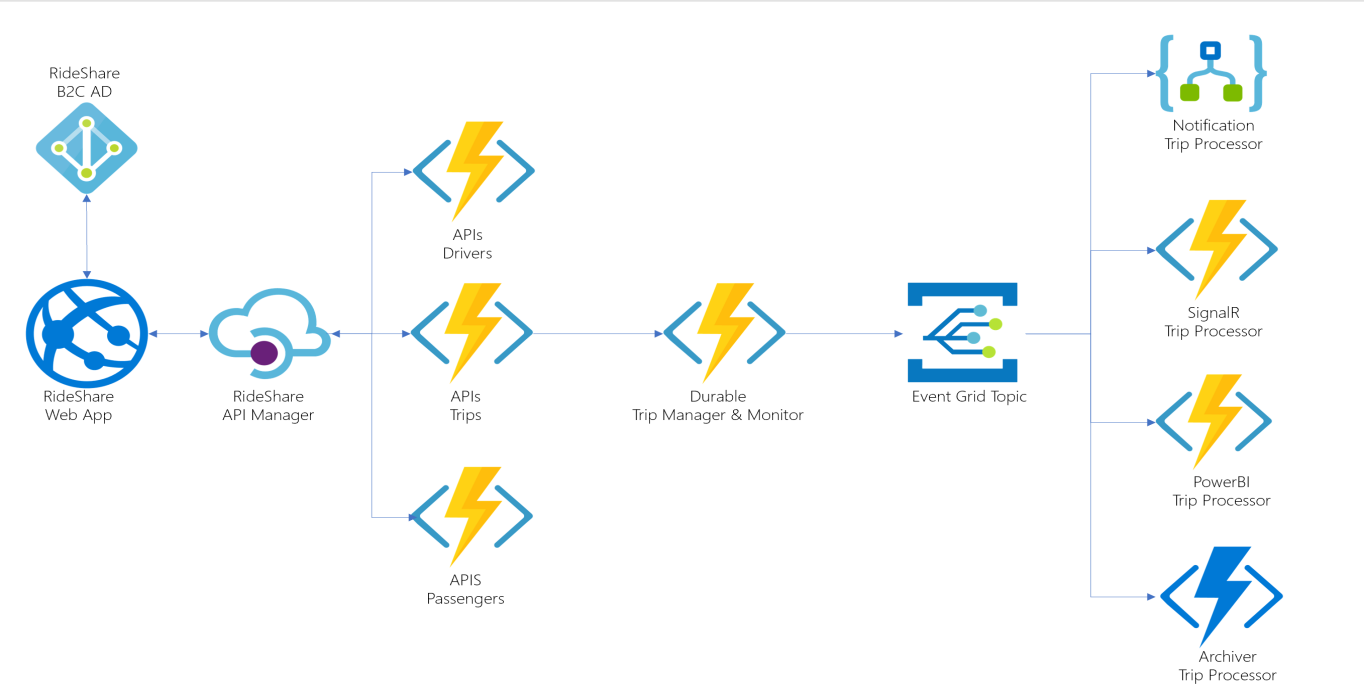
# Architecture overview



| **Component** | **Technology** | **Description** |
| --- | --- | --- |
| RideShare Web App | Vue.js SPA | A user-friendly website where people can sign up and log in. It serves different users with varying permissions. Passengers can request rides and receive real-time updates on their rides. Executive users can view reports on rides and system performance. |
| API Manager | Azure API Manager | Acts like a gatekeeper for the RideShare APIs. It ensures that the APIs are secure and accessible only to authorized users. It also collects usage data and provides documentation for developers to use the APIs correctly. It also limits the rate of API requests to prevent overloading the system. |
| RideShare APIs | C# Azure Functions | These are like small applications that provide specific functionalities for RideShare. They handle creating, reading, updating, and deleting data related to drivers, trips, and passengers. They are used by the Web App to perform various actions. |
| Durable Orchestrators | C# Durable Functions | Special assistants that manage trips from start to finish. They constantly keep track of the trip's progress and provide real-time updates to users. They play a vital role in ensuring the whole system works smoothly during the entire trip. |
| Event Emitter | Event Grid Topic | A way to communicate important events related to trips. It broadcasts messages about trip stages to the outside world. |
| Event Subscribers | Functions & Logic Apps | Different parts of the system that listen to the Event Emitter. They respond to the messages and perform specific tasks based on the trip events received. |
| Key Vault | Azure Key Vault | A secure vault that keeps all the sensitive information, like passwords and keys, safe from unauthorized access. It ensures that secrets are well-protected and can be shared safely among different services. |

The following are the Event Grid Subscribers:

| **Subscriber** | **Technology** | **Description** |
| --- | --- | --- |
| Notification | Logic App | This trip processor sends notifications to admins, like emails or SMS messages, when a trip progresses through different stages. It keeps admins informed about the trip's status and any important updates related to it. |
| SignalR | C# Azure Function | This trip processor is responsible for updating passengers in real-time about their trip status. It communicates with browsers or mobile apps used by passengers, ensuring they get instant updates on their trip progress as it happens. |
| Power BI | C# Azure Function | This trip processor inserts trip information into an SQL Database and potentially into a Power BI dataset using APIs. This allows for data analysis and visualization in Power BI, helping to derive insights and create informative reports. |
| Archiver | Node.js Azure Function | This trip processor archives trip information into Azure Cosmos DB. It ensures that trip data is securely stored for future reference or analysis, even after the trip has been completed. |

**Relecloud has decided to consider a certain piece of functionality as a Microservice if it meets the following criteria:**

1. \*\*Independent Scalability and Deployment:\*\* The functionality should be able to scale and be deployed separately from other parts of the system. This means that it can grow and handle more workload on its own without affecting other parts, and it can be updated or changed without causing disruptions to the rest of the system.

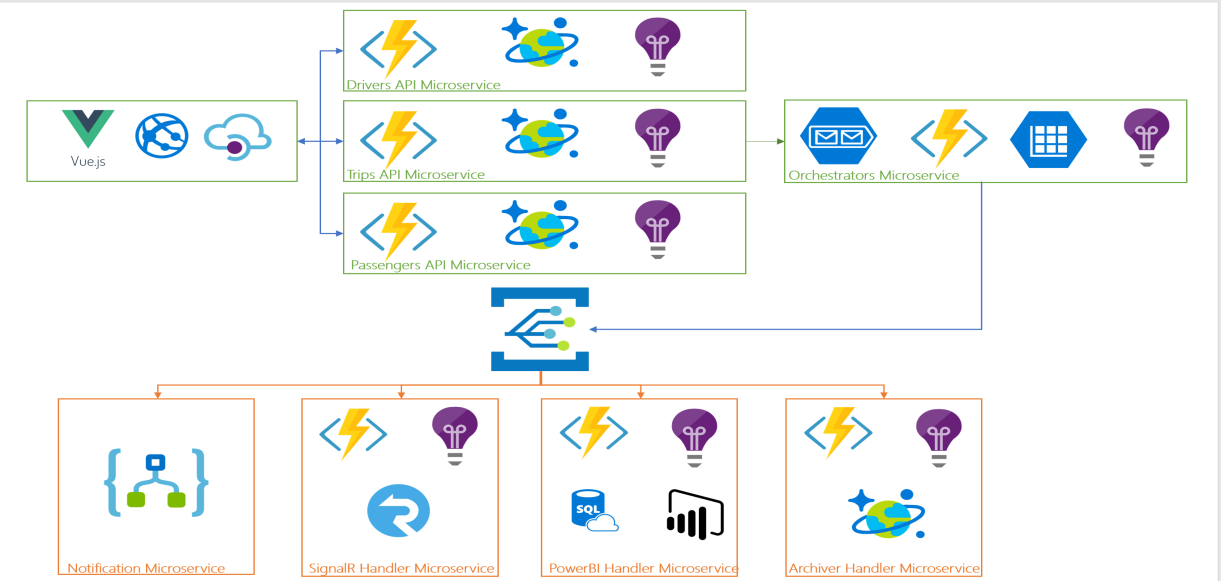
2. \*\*Different Language/Technology:\*\* If there is a specific expertise or knowledge available only in a particular language or technology, the functionality should be written using that language. This allows the team to leverage the strengths of different technologies and make the best use of available skills.

3. \*\*Clean Boundary and Isolation:\*\* The functionality should have a clear and well-defined boundary, separating it from other parts of the system. This ensures that it operates independently and doesn't interfere with other components. It should be isolated so that changes to one microservice do not affect others, promoting better maintainability and flexibility in the system.

By following these criteria, Relecloud aims to create a flexible, scalable, and maintainable system architecture using microservices, where each piece of functionality can work independently and efficiently.

| **Microservices** | **Technology** | **Reason** |
| --- | --- | --- |
| Drivers APIs | C# | The Drivers API is implemented using C# and deployed as a separate Function App. It can scale and deploy independently, which allows it to handle driver-related functionalities on its own without affecting other parts of the system. |
| Trips APIs | C# | The Trips API is also built with C# and deployed as a separate Function App. Like the Drivers API, it is independent in code and deployment, providing functionalities related to trips without being tightly coupled with other components. |
| Passengers APIs | C# | Similarly, the Passengers API is coded in C# and deployed as a separate Function App. It is independent, enabling it to handle passenger-related functionalities without any direct dependencies on other services. |
| Durable Orchestrators | C# | The Durable Orchestrators, including Trip Manager, Monitor, and Demo, are crucial components that form the core of the solution. They are written in C# and operate independently, providing management and monitoring functions throughout the trip. |
| Event Grid Notification Handler | Logic App | The Notification Handler, implemented as a Logic App, adds value to the solution by notifying admins about trip updates. It operates independently, providing a separate service for this specific task. |
| Event Grid SignalR Handler | C# | The SignalR Handler, developed in C#, enhances the solution by updating passengers in real-time about their trip status. It is an independent service dedicated to this functionality. |
| Event Grid Power BI Handler | C# | The Power BI Handler, written in C#, brings value to the solution by inserting trip data into an SQL Database and possibly into Power BI. It works independently to serve this purpose. |
| Event Grid Archiver | Node.js | The Archiver Handler, built with Node.js, adds value to the solution by archiving trip data into Azure Cosmos DB. It is an independent component responsible for this specific task. |

**The following is a detailed diagram showing how the different architecture components communicate and the Azure services they use: (Data Flow)**



the sample application uses a user-friendly website (SPA Web App) that allows passengers to do the following:

1. Login: Passengers can log in to the website with their credentials.

2. Manage Trips: Passengers can request new trips and also view their previous trips.

3. API Manager: The website uses an API manager, acting as a gatekeeper, to access the necessary back end APIs.

When a passenger wants to request a trip, they fill in their information along with the starting and destination locations. This information is then sent as a request to the "Trips Microservice" through the front-end API.

Essentially, the application provides an easy-to-use website where passengers can log in, request trips, and view their trip history. Behind the scenes, there is a microservice called the "Trips Microservice," which handles the trip requests from passengers through the exposed front-end API.

{

"passenger": {

"code": "joe@gmail.com",

"firstName": "Joe",

"lastName": "James",

"mobileNumber": "+13105551212",

"email": "joe@gmail.com"

},

"source": {

"latitude": -31.7654,

"longitude": 54.9011

},

"destination": {

"latitude": -32.5625,

"longitude": 60.6276

},

"type": 1

}

**here's what happens when a passenger requests a trip:**

1. \*\*Passenger Request\*\*: The passenger fills in the required details like their information, starting location, and destination on the website.

2. \*\*Trips Microservice\*\*: The "Trips Microservice," which is a small independent application, takes this trip request and stores it in a database called "Azure Cosmos DB." This way, all trip information is safely stored for future reference.

3. \*\*Orchestrators Microservice\*\*: After storing the trip in the database, the "Trips Microservice" sends a message (like a notification) to the "Orchestrators Microservice" to let it know that a new trip request has come in. The "Orchestrators Microservice" is like a manager that handles trips and ensures everything runs smoothly.

4. \*\*Trip Information\*\*: The "Trips Microservice" then sends back the newly created trip's information, including a unique code and other relevant details, to the passenger. This way, the passenger knows that their trip request has been successfully received and processed.

5. \*\*Optional Trigger\*\*: There's an option for the "Orchestrators Microservice" to be triggered through its own internal API. It means that when the "Trips Microservice" enqueues the trip information, it can also directly notify the "Orchestrators Microservice" without waiting for it to check for new trips.

In summary, when a passenger requests a trip, the "Trips Microservice" saves the trip details in the database, informs the "Orchestrators Microservice" about the new trip, and returns the trip information back to the passenger. The "Orchestrators Microservice" then takes care of managing the trip and ensuring everything proceeds as intended.

**here's what happens when the Orchestrators Microservice's Durable Trip Manager manages a trip:**

1. \*\*Notify Available Drivers\*\*: The Trip Manager first identifies available drivers who are close (within a certain distance) to the trip's starting location and are not currently serving other passengers. It then notifies these drivers about the new trip request. This notification is sent as a "Drivers notified" event to the Event Grid, like a signal to all interested parties.

2. \*\*Wait for Driver Response\*\*: The Trip Manager waits for a response from the drivers. There are two possibilities:

- If none of the drivers accept the trip within a specified time (timeout), the Trip Manager considers that no driver is interested, and it aborts the trip. This decision is communicated as a "Trip aborted" event to the Event Grid.

- If a driver accepts the trip within the allowed time, an external event is triggered, signaling that a driver is willing to take the trip. This external event is communicated to the Trip Manager through the Orchestrators Microservice API.

3. \*\*Assign the Driver\*\*: Once the Trip Manager receives the signal that a driver has accepted the trip, it proceeds with the next steps. It assigns the accepting driver to the trip, marking that the trip is now being served by this particular driver. This assignment is communicated as a "Drivers picked" event to the Event Grid.

4. \*\*Enqueue Trip Monitor\*\*: The Trip Manager enqueues a message to the Trip Monitor queue, which means it puts a message in a waiting line to be processed later. The Trip Monitor is responsible for monitoring the progress of the trip and keeping track of its status.

In summary, the Orchestrators Microservice's Durable Trip Manager manages the trip from start to finish. It notifies available drivers about the new trip, waits for a response from them, and either aborts the trip if no driver is interested or proceeds with the trip if a driver accepts it. Once a driver is assigned, it enqueues a message to the Trip Monitor queue, ensuring that the trip's progress is monitored effectively. This orchestration process helps in efficiently handling trip requests and ensuring smooth passenger experiences.

**here's what happens when the Trip Monitor queue is triggered and the Durable Trip Monitor is instantiated:**

1. \*\*Trip Monitor Queue Trigger\*\*: The Trip Monitor queue is triggered by the Orchestrators Microservice when it's time to monitor a particular trip's progress. This happens after the trip has been assigned to a driver.

2. \*\*Durable Trip Monitor\*\*: The Orchestrators Microservice starts a Durable Trip Monitor, which is like a watchful monitor specifically assigned to keep an eye on this trip.

3. \*\*Trip Progress Timer\*\*: The Durable Trip Monitor sets up a timer that triggers every x seconds to check the trip's status. It's like setting a reminder to regularly check on how the trip is doing.

4. \*\*Checking Trip Completion\*\*: Each time the timer triggers, the Durable Trip Monitor checks whether the trip is completed or not. If the trip is completed, it means the passenger has reached their destination, and the Durable Trip Monitor indicates this by sending a "Trip completed" state change event to the Event Grid.

5. \*\*Ongoing Trip\*\*: If the trip is still ongoing and not yet completed, the Durable Trip Monitor sends a "Trip running" state change event to the Event Grid. This means the passenger is still on the way to their destination.

6. \*\*Abort Trip\*\*: The Durable Trip Monitor ensures trips don't run forever. If the trip does not complete within a configurable amount of time, it will abort the trip. Aborting the trip means it takes action to end the trip, as it's not progressing as expected.

In summary, the Durable Trip Monitor is responsible for monitoring the progress of a trip. It sets up a timer to regularly check if the trip is completed or ongoing. If the trip is completed, it reports it as "Trip completed," and if it's still running, it reports it as "Trip running." If the trip takes too long and doesn't complete within the expected time, the Durable Trip Monitor will abort the trip to ensure it doesn't keep running indefinitely. This monitoring process helps in keeping track of trips and ensuring that passengers reach their destinations smoothly and within a reasonable time frame.

**when events are sent to the Event Grid Topic, they trigger the different handler Microservices to perform further processing on the trip:**

1. \*\*Notification Microservice\*\*: This Microservice acts like a notification system for administrators. When it receives events from the Event Grid Topic, it processes them to notify administrators about important trip updates. For example, it may send emails or SMS messages to administrators, keeping them informed about the trip's progress and any significant changes.

2. \*\*SignalR Handler Microservice\*\*: This Microservice handles real-time updates for passengers. When it receives events from the Event Grid Topic, it processes them to update passengers' browsers or mobile apps in real-time about their trip status. Passengers can see live updates as their trip progresses, ensuring they are always aware of any changes or delays.

3. \*\*Power BI Handler Microservice\*\*: This Microservice is responsible for data analysis and visualization. When it receives events from the Event Grid Topic, it processes them to insert trip data into an SQL Database and possibly into Power BI datasets. This allows for generating reports and insights based on trip data, helping to monitor performance and make data-driven decisions.

4. \*\*Archiver Handler Microservice\*\*: This Microservice takes care of archiving trip data. When it receives events from the Event Grid Topic, it processes them to store trip information in Azure Cosmos DB. Archiving ensures that trip data is securely retained for future reference or analysis, even after the trip is completed.

In summary, the Event Grid Topic acts like a central hub, receiving events related to trip updates. These events then trigger different handler Microservices to perform specific tasks:

- The Notification Microservice notifies administrators about trip updates.

- The SignalR Handler Microservice provides real-time updates to passengers.

- The Power BI Handler Microservice stores trip data for analysis and reporting.

- The Archiver Handler Microservice archives trip information for future use.