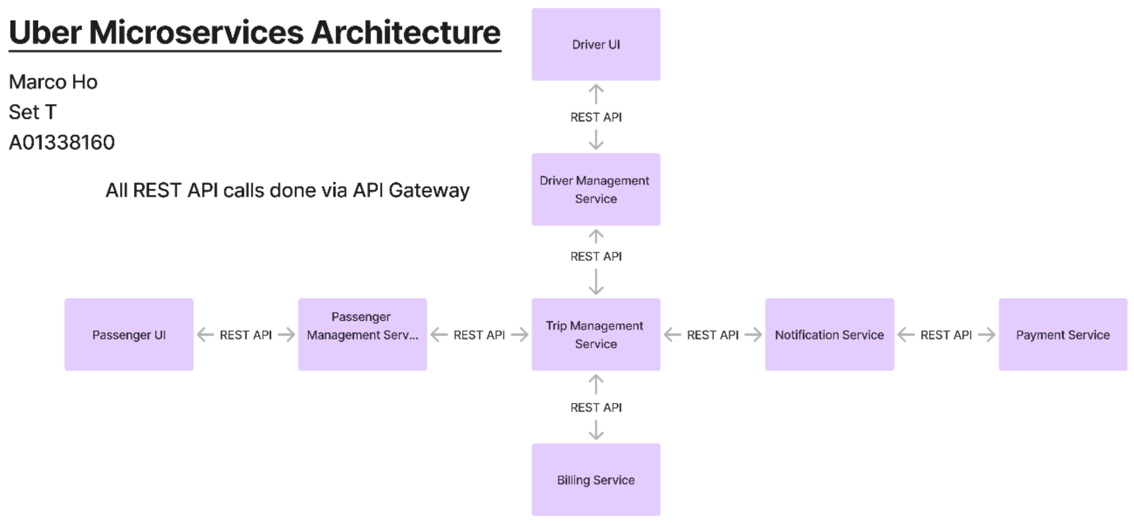
Marco Ho
Set T
A01338160

Lab 2: Microservices at Uber



Uber's product initially started off as a Monolithic Architecture. Due to the growth of their product and user base, they started running into several issues such as having to redeploy their entire code base every time they wanted to make an update to a single feature, code became tightly coupled and difficult to modify without side effects, scaling all their features simultaneously due to an increase in demand in certain components was difficult to handle. Therefore, they decided to switch their codebase to a Microservice Architecture. Three key microservices in Uber's Architecture include:

1. Passenger Management Service – responsible for passenger related operations
2. Driver Management Service – responsible for driver related operations
3. Trip Management Service – responsible for coordinating between the two above services and overall ride management

By breaking down their application into modular components with separate functionalities as listed above, several advantages were observed:

1. Scalability: Each service can now be scaled independently based on the number of requests/demands there is for that specific service.

2. Development Efficiency: Each service and now be assigned to different teams and they will no longer interfere with each other with the code being more modular and loosely coupled.
3. Robustness: Even if one of the services fails or degrades in performance, other services will not be affected.
4. Improved Deployment: Individual services can be updated without redeploying all the other services.

Despite the valuable advantages, there are some drawbacks to switching over to a Microservice Architecture:

1. Volume of API calls: Since each microservice communicates with another via API Gateway, the product needs additional resources and capabilities to manage a large volume of API calls.
2. Increased Latency: As opposed to a Monolithic Architecture, each microservice component will have to communicate with another via API calls which could introduce increased overall latency.
3. Data Consistency: The application will need to perform additional work to ensure consistency of data being communicated between services.

References:

https://blog.dreamfactory.com/microservices-examples

https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a