Team '**Sushma'** and '**Tushar'**

Uber Application - Monolith
Microservices - Request/Ride/Hop out
                                Schedule now/later
Advantages: Reduce in costs, Efficiency

Reducing technical debt is an important benefit offered by microservices; however, measurable savings do not end with just technical debt. Microservices offers other benefits to the business which can reduce costs and impact the bottom line. Those benefits include:

- Agility: By breaking down functionality to the most basic level and then abstracting the related services, DevOps can focus on only updating the relevant pieces of an application. This removes the painful process of integration normally associated with monolithic applications. Microservices speed development, turning it into a process that can be accomplished in weeks and not months.
- Efficiency: Leveraging a microservices based architecture can result in a far more efficient use of code and underlying infrastructure. It is not uncommon to experience significant cost savings by as much as 50% by reducing the amount of infrastructure required to run a given application.
- Resiliency: By dispersing functionality across multiple services eliminates an applications susceptibility to a single point of failure. Resulting in applications which can perform better, experience less downtime and can scale on demand.
- Revenue: Faster iterations and decreased downtime can increase revenue (either using efficiencies created via a chargeback ideology or by improving user engagement). User retention and engagement increases with continuous improvements offered by microservices.

Uber went from monolith to SOA in 2015. This SOA followed a microservice-based architecture. And we've been sharing what we learned along the way: the steps it usually takes to build a microservice, addressing testing problems with a multi-tenancy approach or how and why we use distributed tracing. We also open sourced some of our tools like Jaeger, which is part of the Cloud Native Foundation's graduated projects, alongside Kubernetes and Prometheus...All of these can serve as inspiration: but the end of the day, you need to make decisions in your environment that you think will work best. Anyone *copying the likes of Google, Uber, Shopify, Stack Overflow or others when they're not even similar in setup will be disappointed.*

At Uber, we adopted a microservice architecture because we had (circa 2012-2013) primarily two monolithic services and ran into many of the operational issues that microservices solve.

- **Availability Risks.** A single regression within a monolithic code base can bring the whole system (in this case, all of Uber) down.
- **Risky, expensive deployments.** These were painful and time consuming to perform with the frequent need for rollbacks.
- **Poor separation of concerns.** It was difficult to maintain good separations of concerns with a huge code base. In an exponential growth environment, expediency sometimes led to poor boundaries between logic and components.
- **Inefficient execution.** These issues combined made it difficult for teams to execute autonomously or independently.

As a result, we adopted a microservice architecture. Ultimately our systems became more *flexible*, which allowed teams to be more *autonomous.*

- **System reliability.** Overall system reliability goes up in a microservice architecture. A single service can go down (and be rolled back) without taking down the whole system.
- **Separation of concerns.** Architecturally, microservice architectures force you to ask the question "why does this service exist?" more clearly defining the roles of different components.
- **Clear Ownership.** It becomes much clearer who owned what code. Services are typically owned at the individual, team, or org level enabling faster growth.
- Autonomous execution. Independent deployments + clearer lines of ownership unlock autonomous execution by various product and platform teams.
- **Developer Velocity.** Teams can deploy their code independently, which enables them to execute at their own pace.

UBER CASE STUDY

UBER's Previous Architecture

Like many startups, UBER began its journey with a monolithic architecture built for a single offering in a single city. Having one codebase seemed cleaned at that time, and solved UBER's core business problems. However, as UBER started expanding worldwide they rigorously faced various problems with respect to the scalability and continuous integration.

Benefits:

Reduced Complexity

Products & Platforms

The above diagram depicts UBER’s previous architecture.

- A REST API is present with which the passenger and driver connect.

- Three different adapters are used with API within them, to perform actions such as billing, payments, sending emails/messages that we see when we book a cab.
- A MySQL database to store all their data.

So, if you notice here all the features such as passenger management, billing, notification features, payments, trip management, and driver management were composed within a single framework.

Problem Statement

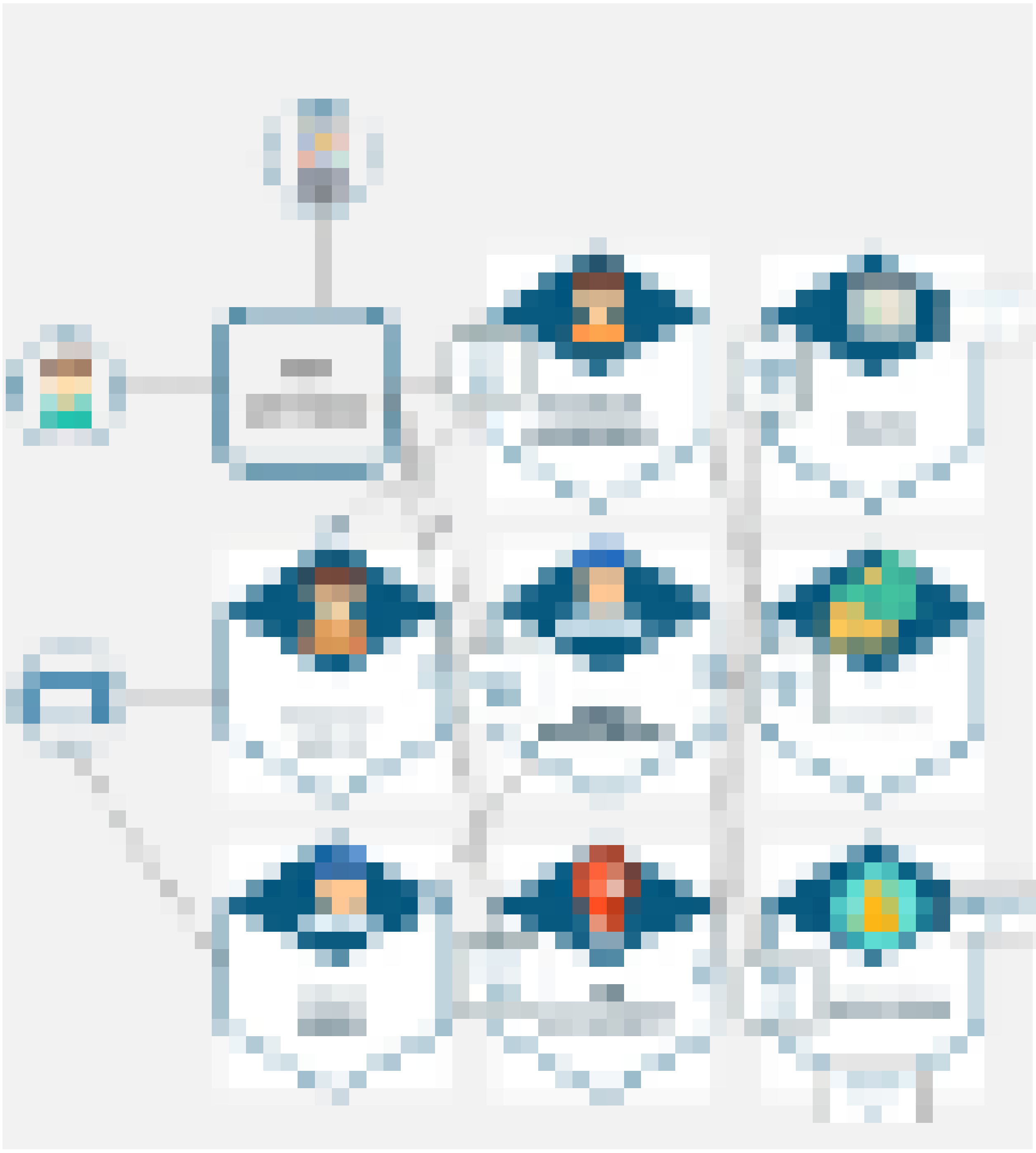
While UBER started expanding worldwide this kind of framework introduced various challenges. The following are some of the prominent challenges

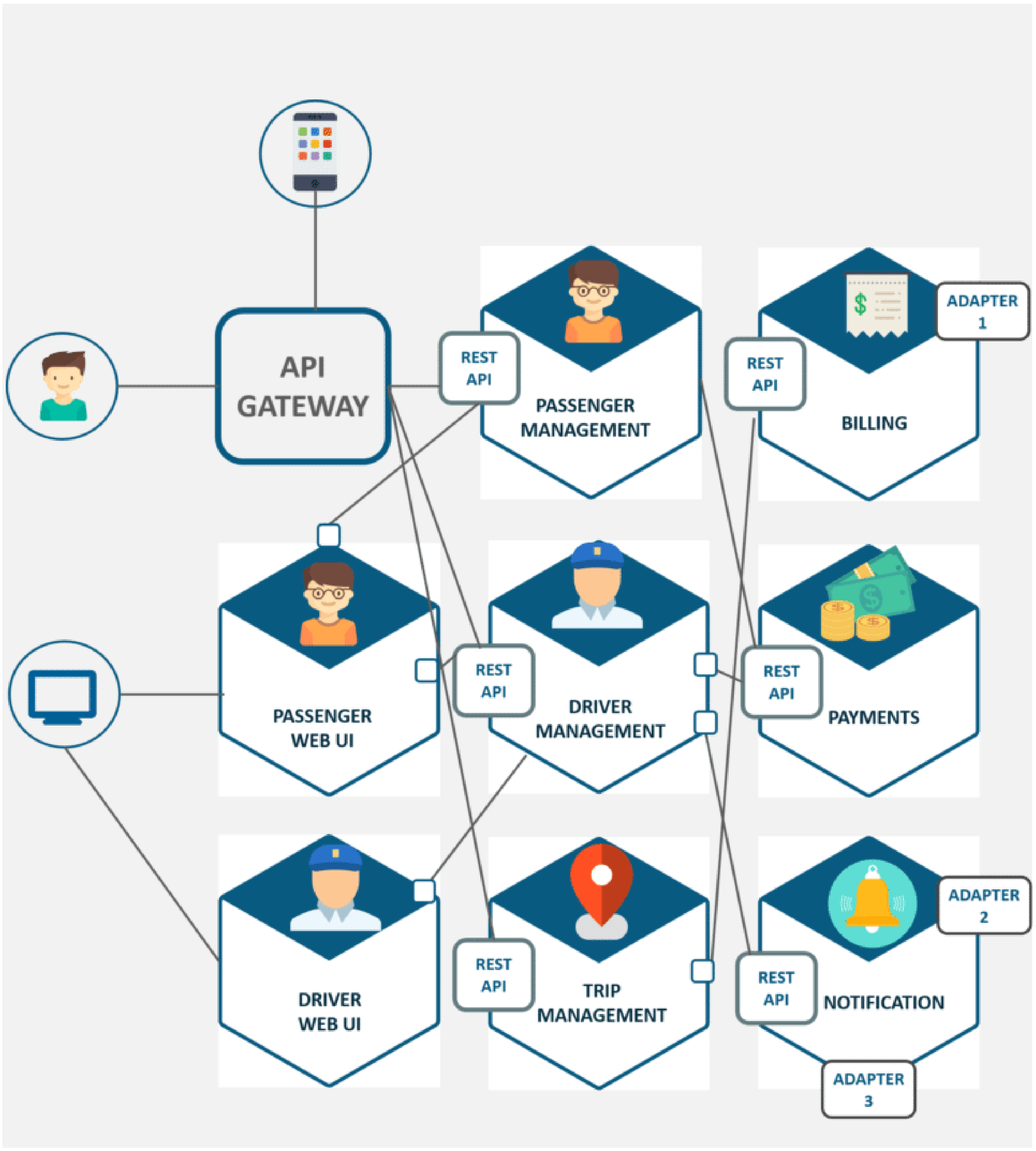
- All the features had to be re-built, deployed and tested again and again to update a single feature.
- Fixing bugs became extremely difficult in a single repository as developers had to change the code again and again.
- Scaling the features simultaneously with the introduction of new features worldwide was quite tough to be handled together.

Solution

To avoid such problems UBER decided to change its architecture and follow the other hyper-growth companies like Amazon, Netflix, Twitter and many others. Thus, UBER decided to break its monolithic architecture into multiple codebases to form a microservice architecture.

Refer to the diagram below to look at UBER's microservice architecture.





Microservice Architecture Of UBER — Microservice Architecture

- The major change that we observe here is the introduction of API Gateway through which all the drivers and passengers are connected. From the API Gateway, all the internal points are connected such as passenger management, driver management, trip management, and others.

- The units are individual separate deployable units performing separate functionalities.