ASSIGNMENT 1
SDA

COMSATS University Islamabad

Abbottabad, Pakistan

<u>Assignment 1</u>

OF

<u>SDA</u>

By

<u>Muhammad Haris Afzal FA21-BSE-116</u>

Supervisor Name:

Sir Mukhtiar

Zamin

Date: 9/05/2024

Microservices Architecture

Microservices architecture is like building a big project with lots of small, independent parts. Instead of making one giant project, we break it into many little pieces that each do their own job. These pieces, called services, work together like a team, but they can also do their own thing without depending too much on each other.
Now, let's talk about why this is cool:

Breaking It Down: Think of it like making a big puzzle. Instead of trying to fit all the pieces together at once, we break it into smaller puzzles. Each piece (or service) is easier to manage and work on.

Working Independently: Each service can be updated or changed without messing up the others. It's like if one person on a team finishes their part of the project early, they can move on without waiting for everyone else.

Choosing the Right Tools: Just like you might use different tools for different tasks (like a hammer for nails and a screwdriver for screws), each service can use the best tools for its job. One might use a database, while another might use something else—it depends on what they need to do.

Keeping Things Going: If one service has a problem, it doesn't bring down the whole project. It's like if one light bulb burns out, the rest of the lights still work.

Talking to Each Other: Services communicate with each other using simple messages. It's like passing notes in class. This way, they can work together without being too dependent on each other.

But, there are some things to watch out for:

Lots of Moving Parts: With so many little pieces, it can get confusing to keep track of everything. It's like trying to juggle too many balls at once.

Keeping an Eye on Things: Each service needs to be watched to make sure it's doing its job properly. It's like babysitting a bunch of kids—you need to keep an eye on all of them.

Making Sure They Play Nice: Sometimes, services need to talk to each other a lot. It's like making sure everyone in a group project is on the same page and working together.

Finding Your Way Around: With so many services, it can be hard to find what you need. It's like trying to find your way in a big, new school—you need a map to get around.

<u>Getting Everyone on Board:</u> Not everyone might like the idea of working this way. It's like convincing your friends to play a new game—they might need some convincing at first.

In the end, microservices can help make big projects easier to manage and work on. But, you need to be careful and make sure everyone is on the same page.

Pipeline architecture:

A pipeline in software is like a conveyor belt for making and testing changes to a program. Here's how it works:
Continuous Integration (CI):

<u>Sharing Changes:</u> When a bunch of people are working on the same project, they use a tool to save and share their work, like saving an essay on Google Docs.

<u>Automatic Checks:</u> Whenever someone adds or changes something in the project, a robot checks to see if it still works okay. It's like having a spell-checker for your writing.

<u>Testing Time:</u> After the robot checks the changes, another robot runs some tests to make sure everything still fits together. It's like a teacher checking your homework.

<u>Cleaning Up:</u> If everything looks good, the changes are saved so everyone else can see them. If not, the person who made the changes gets a chance to fix them.

Continuous Delivery (CD):

<u>Testing Playground:</u> Before putting the changes in the final project, they're tested in a special area that's just like the real thing. It's like rehearsing a play before performing it in front of an audience.

<u>Deploying Changes:</u> If the tests go well, the changes are added to the main project. It's like adding a new chapter to a book, but making sure it's perfect first.

<u>Keeping an Eye Out:</u> Robots keep watch to make sure everything is running smoothly. If something goes wrong, they'll let the team know so they can fix it quickly.

Continuous Deployment (CD):

<u>Automatic Updates:</u> Once the changes pass all the tests, they're automatically added to the final project. It's like having a robot add toppings to a pizza as soon as they're ready.

Staying Safe: To make sure nothing goes wrong, the changes are added slowly, like pouring syrup on pancakes a little bit at a time.

In short, the pipeline makes sure that changes to a program are carefully checked and added to the project in a safe and controlled way. It's like making sure each piece of a puzzle fits perfectly before completing the picture.

Successful Example:

Netflix

Architecture Overview:
Netflix is a prime example of a company that successfully implemented the microservices architecture. Their platform is divided into hundreds of microservices, each responsible for a specific function, such as user authentication, recommendation algorithms, video streaming, etc.

Reasons for Success:

Scalability: Microservices allowed Netflix to scale each component independently, enabling them to handle millions of concurrent users without significant downtime.

Flexibility and Agility: With microservices, Netflix can quickly deploy updates to individual services without affecting the entire system, enabling faster innovation and experimentation.

Fault Isolation: If one microservice fails, it doesn't necessarily bring down the entire system, ensuring high availability and reliability.

Financial Impact: Netflix's adoption of microservices has contributed significantly to its financial success. By providing a highly scalable and reliable streaming platform, Netflix has attracted millions of subscribers worldwide, leading to substantial revenue growth.

Failed Example:

Twitter's Attempt at Microservices

Architecture Overview:
Twitter, a popular social media platform, attempted to transition from a monolithic architecture to microservices to address scalability and development agility concerns.

Reasons for Failure:

Technical Debt: Twitter had accumulated significant technical debt within its monolithic codebase, making it challenging to decouple services and migrate to a microservices architecture seamlessly.

Performance Challenges: Transitioning to microservices introduced new performance bottlenecks, such as increased network latency and service-to-service communication overhead, impacting the overall responsiveness of the platform.

Operational Complexity: Managing and monitoring a large number of microservices proved to be more complex and resource-intensive than anticipated, leading to operational challenges and increased system downtime.

Financial Impact: Twitter's attempt to adopt microservices faced various challenges, including disruptions in service availability and slower feature delivery. These issues could potentially have a negative impact on user experience and advertiser confidence, impacting Twitter's revenue generation capabilities.