**Assignment 2**

**==================**

Basic design is ready in fista ssignment, add to the design, critical non-functional aspects and mention the tools that you would choose.

1. It should be highly scalable.

2. It should be highly available.

3. It should be resilient.

Hint: Service registry/discovery, load balancing, circuit breaker.

**Solution:**

I understand you need an updated version of the architecture design with a focus on critical non-functional aspects such as scalability, high availability, and resilience. Below, I’ll explain how we can address each requirement and suggest the appropriate tools for the job.

**Updated Microservices Architecture Design with Non-Functional Aspects**

**1. Scalability**

Scalability refers to the ability of the system to handle increased load by adding resources. We will ensure that the system is highly scalable both horizontally (adding more instances) and vertically (increasing the power of existing instances).

**Strategies to achieve scalability:**

1. **Horizontal Scaling**:
   * Each microservice will be stateless to enable horizontal scaling. This means new instances of the service can be added dynamically as load increases.
   * **Tools**:
     + **Kubernetes** or **AWS ECS** for automatic scaling of containers based on traffic.
     + **Docker** for containerization of microservices.
2. **Asynchronous Processing**:
   * Services that process heavy tasks, such as fee calculations or payment processing, will be decoupled using message queues to offload heavy tasks asynchronously.
   * **Tools**:
     + **Apache Kafka**, **RabbitMQ**, or **AWS SQS** for managing asynchronous queues.
3. **Database Sharding**:
   * For services with large databases, we will implement database sharding to distribute data across multiple instances, improving query performance.
   * **Tools**:
     + **Amazon RDS**, **Cassandra**, or **MongoDB** for distributed databases.

**2. High Availability**

High availability ensures that the system remains functional even if a service or a server fails. To ensure high availability, we will use the following strategies:

1. **Service Replication**:
   * Each microservice will be deployed in multiple instances across different availability zones to prevent a single point of failure.
   * **Tools**:
     + **Kubernetes** or **Docker Swarm** to manage containerized services with replication and fault tolerance.
     + **AWS ELB** (Elastic Load Balancer) to distribute incoming traffic.
2. **Database Replication**:
   * Implement replication for databases to ensure that data is available in case of server failures.
   * **Tools**:
     + **Amazon RDS** (Master-Slave Replication) for relational databases.
     + **Cassandra** or **MongoDB** for highly available NoSQL databases.
3. **Service Discovery**:
   * Microservices should register themselves dynamically and discover other services for communication. This ensures that if a service instance fails or is scaled up, the other services can discover and communicate with the new instance.
   * **Tools**:
     + **Consul**, **Eureka** (Netflix OSS), or **Zookeeper** for service discovery.
4. **Load Balancing**:
   * A load balancer will distribute incoming traffic evenly across multiple instances of the services, ensuring no single instance is overwhelmed.
   * **Tools**:
     + **NGINX**, **HAProxy**, or **AWS ALB** (Application Load Balancer).

**3. Resilience**

Resilience is the ability of the system to recover quickly from failures and continue functioning properly. To ensure resilience, we will implement the following strategies:

1. **Circuit Breaker**:
   * Implementing a circuit breaker pattern will prevent a service from repeatedly calling a failing service and allow the system to fail gracefully.
   * **Tools**:
     + **Hystrix** (part of Netflix OSS) or **Resilience4j** for implementing circuit breakers.
2. **Retry Logic**:
   * Services should have retry logic in case of temporary failures. This will ensure that the system can recover and continue processing requests after a transient issue.
   * **Tools**:
     + **Resilience4j** or **Spring Retry** for retry mechanisms.
3. **Fallback Mechanism**:
   * When a service is down, fallback mechanisms should be in place to ensure the system can still provide basic functionality or return a meaningful error message to the user.
   * **Tools**:
     + **Resilience4j** or **Hystrix** for handling fallback.
4. **Health Checks**:
   * Regular health checks should be implemented for each service to ensure it is up and running.
   * **Tools**:
     + **Prometheus** for monitoring, **Grafana** for visualization, and **Kubernetes Liveness/Readiness Probes** for health checks.
5. **Fault Injection**:
   * Injecting faults into the system allows us to test the system's behavior during failure conditions. This helps identify bottlenecks and ensures that failure scenarios are handled properly.
   * **Tools**:
     + **Chaos Monkey** (part of Netflix Simian Army) for fault injection in production.

**Tools Summary**

* **Kubernetes** / **Docker Swarm** for container orchestration and service replication.
* **Consul**, **Eureka**, or **Zookeeper** for service discovery.
* **NGINX**, **HAProxy**, or **AWS ALB** for load balancing.
* **Resilience4j** / **Hystrix** for implementing circuit breakers, retries, and fallback mechanisms.
* **Apache Kafka**, **RabbitMQ**, or **AWS SQS** for asynchronous communication.
* **Prometheus** and **Grafana** for monitoring and alerting.
* **Amazon RDS**, **Cassandra**, **MongoDB** for high-availability databases.
* **Chaos Monkey** for fault injection testing.