Bill Payment Application Design

## Roles & responsibilities for each mS

1. User Registration Service: This service will register the user for paying bills. It will store information of user’s registered billers. It will call wallet service for any transactions to be done in wallet.

Tech Stack: .Net6, Rest API, K8s, Docker, AWS, NoSQL

Data model: UserId, EmailId (Masked), WalletId, RegisteredBillerId

1. Wallet Service: This service will manage user’s wallet. Debit/Credit transaction will be performed in this service. It will call the payment service.

Tech Stack: .Net6, Rest API, K8s, Docker, AWS, NoSQL, Kafka

Data Model: WalletId, UserId, Balance. Separate table for transaction history.

1. Biller Service: This service will fetch billers and bill from 3rd party external applications. And initiates the transaction to pay bill. It will call the wallet service.

Tech Stack: .Net6, Rest API, K8s, Docker, AWS, NoSQL

Data Model: AvailableBillerId, BillerName, BillerAPI. Separate table for bill payment history.

1. Payment Service: This service will talk to external payment gateways to perform the transaction and upon successful response it will update the calling service.

Tech Stack: .Net6, Rest API, K8s, Docker, AWS, NoSQL, Kafka

Data Model: TransactionId, Amount, Status, TransactionDate.

1. Bulk Bill Payment Service: It will be a background service. This service will implement strategy design pattern to allow processing of multiple bill pay file types and will be extendable in future.

Tech Stack: .Net6, Rest API, Background Service, K8s, Docker, AWS, NoSQL, Kafka

Data Model: FileId, PaymentRecord, Status.

## Testing strategy

Apart from the manual testing we will implement automated tests. We will write following types of automated tests:

End to End: Here we will mostly test positive scenarios without mocking. Some important validation errors can also be tested here. It will be executed in both CI and CD pipeline.

Behavioral: We will mock the dependencies and write tests for both positive and negative scenarios. It will be executed in CI pipeline.

Contract: These tests will perform sanity test on the dependencies like database tables, other mS. The aim is to capture any changes in the schema that would impact our application. It will be executed in CI pipeline.

# Non Functional requirements

## Security: For network security we will implement WAF (Web Application Firewall) in front of our applications. For application security, we will have API gateway for rate limiting, CORS check, authentication and authorization.

## Performance: NoSQL databases will help us achieve greater performance. There are many performance improvements done in .Net 6 hence I have chosen it in tech stack. We will use Redis cache to store data of frequently run queries.

## Logging & Monitoring: For enhanced logging and monitoring we can use tools like New Relic. Alerts can be configured for high error percentage, high response time etc.

## Auto scaling: Application auto scaling will be configured in K8s. Horizontal scaling made by configuring the min and max replicas. Scaling will happen based on target average CPU utilization. Vertical scaling configured through “Request/Limit”. Each defines the CPU and memory needed. We have used NoSQL databases so we can configure horizontal scaling also for them.

## CI CD processes and tech stack: Azure Devops can be used for CI/CD. The pipeline configuration need to be done in yaml files. Each microservice will have its own pipeline for independent deployment and custom configuration. End to End tests will be run on each environment before deploying the application. If the tests fail, the deployment will roll back. For K8s config files management and deployment we can use Helm chart.