# **Design Doc**

* Metadata  
  Authors: *Mohit Sharma*Date: *08th August, 2022*

## **Background**

### **Context**

💡 This is a payment feed system which is handled by finance microservice acting as the frontend system exposed to the user and backed by a SQL DB as backend. The microservice is talking to a 3rd party payment processor which handles all the payment transactions for the user and processes them.

### **Motivation**

💡 The finance service is being widely used by the customers in production and as a result the platform needs to be scaled up to accommodate the increased traffic keeping the basic functionality intact, robust and reliable at the same time

### **Requirements**

💡 In-scope changes

Finance service microservice infrastructure changes to handle more traffic

SQL DB resource configuration changes to support more read/write operations

Mocking of 3rd party payment processor to test the scalability of the same

Out-scope changes

No changes in REST API interfaces

No changes in user journey

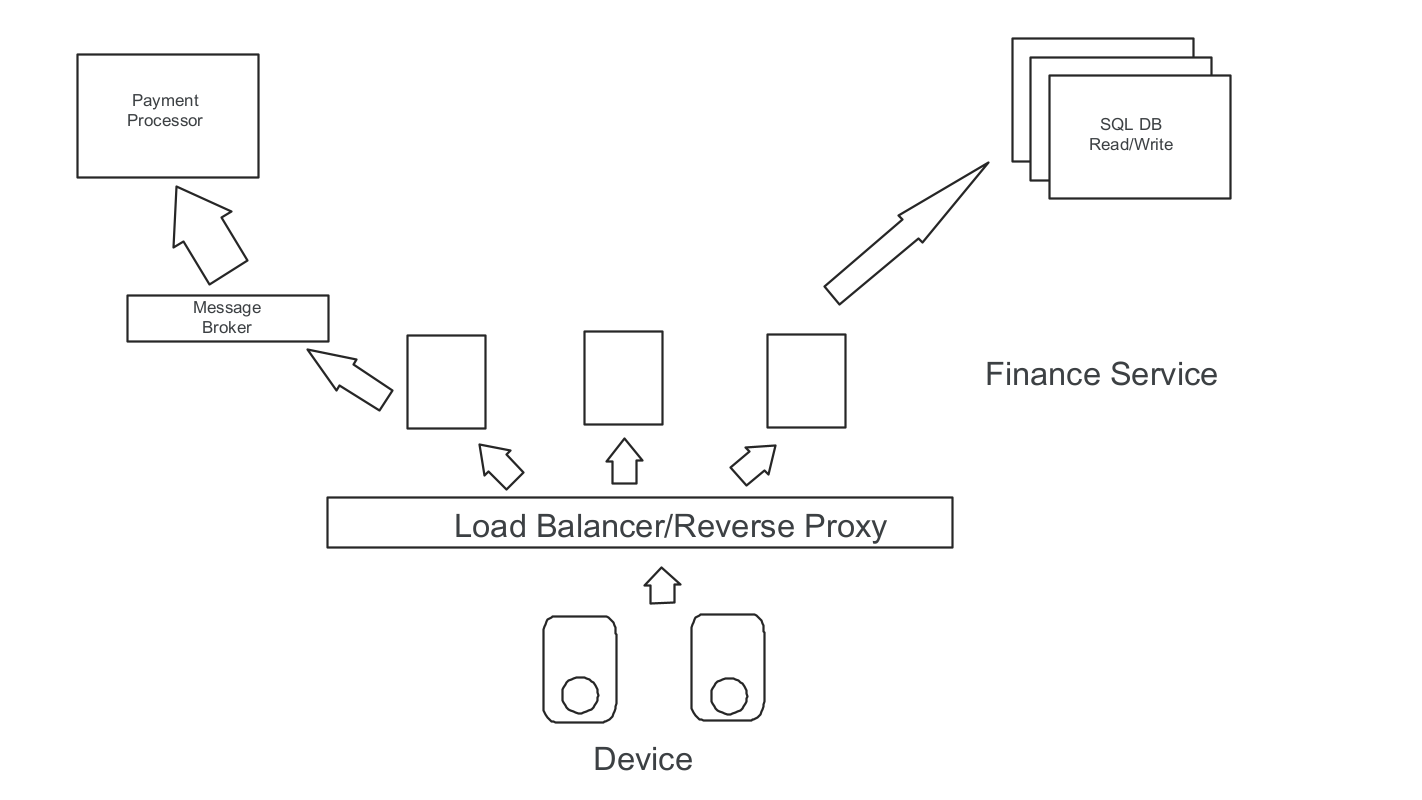
## **Design**

*The main focus is to make sure the finance service is able to support thousands of concurrent users and requests at any given point in time, backed by the SQL DB without any downtime. Also, as the payment processor is not within our vicinity, the system should be designed to work efficiently with the current configuration of the payment processor service.*

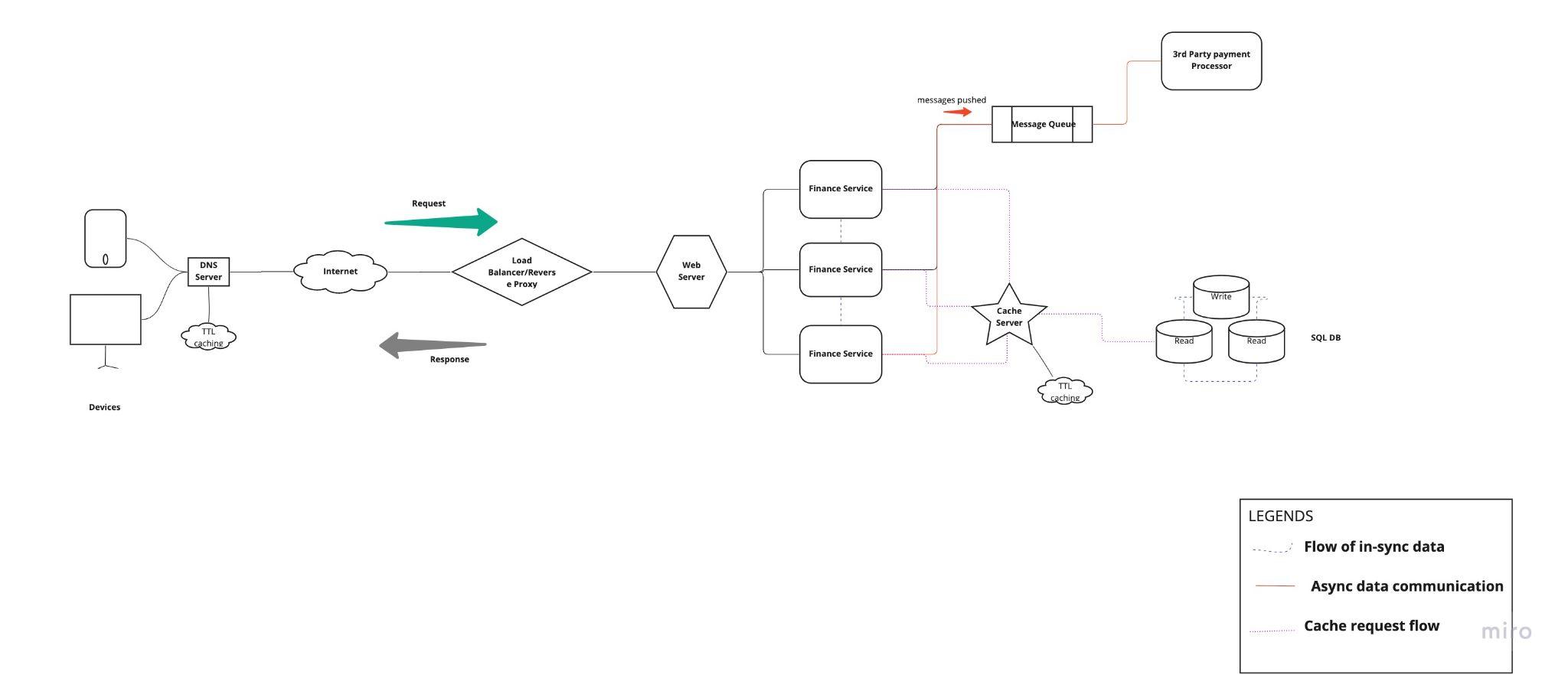
*The key areas to be taken into consideration:*

1. *Optimizing interaction between users (device) and finance service*
2. *Using algorithms/mechanism not to overload payment processor when thousands of requests are made to it*
3. *Creating a mock service similar to payment processor to test the overall system handling capacity*
4. *Applying correct configurations to SQL DB to handle read/write operations efficiently*

### **System Architecture**



### **Communication**

**

## **Data**

High level approach

* *User records creation and insertion in database*
* *Using an in-memory data store distributed system which can scale and store large volumes data during run-time*
* *Client load test tool which can facilitate in-memory data store and use data simulating thousands of virtual concurrent users at large scale*
* *Mock service to be created which can simulate basic functionality of the payment processor system*

Low level approach

Insert user\_id records into SQL DB using any of the programming languages (like Python/Java). In this manner, the records will exist and can be queried from the database at any point in time. This will reduce the write operations overhead during run-time of tests.

In order to distribute the data to simulate large scale user volumetric tests and instead of using flat data files, in-memory key value based data store systems (like redis) can be used where records such as user\_id, amount and merchant\_id mapped to each user can be created and stored.

Scripting python code with redis libraries to simulate thousands of data with respect to amount, random user\_id and merchant\_id.

Using any of the load script tools (like jmeter), the user journey API can be scripted with the correct flow. Redis plugins can be used and with the help of a post processor code, the required redis keys can be called and data can be pushed and popped (rpush, lpop) out of the redis keys in a sequential fashion.

### **Model**

Common focus areas within the system

Stateless microservice - Building the finance microservice stateless in nature will allow not to store any data on the server and will protect it from overhead. All data will be stored within the in-memory distributed data store caching systems. This would support large scale user requests at faster rates

Caching - Using in-memory data store persistent caching techniques placed near to the application server and in between databases can cause less overload to the database.  
With DNS caching placed closer to the client, the application servers will not be under heavy load when thousands of requests are fired.

Message Queue - This being placed in between application server and payment processor will not cause the later to get overloaded with thousands of requests being sent from finance service at any given point in time. As message queue enables asynchronous communication, means that the endpoints that are producing and consuming messages interact with the queue, not each other. Producers can add requests to the queue without waiting for them to be processed. Consumers process messages only when they are available. No component in the system is ever stalled waiting for another, optimizing data flow.   
Queues make data persistent, thus reducing the errors that happen when different parts of a system go offline thus making the overall system more fault tolerant.

Auto scaling - Auto Scaling instances of finance service will allow the system to be more vigilant in handling peak traffic as per the need and scale it down when not in use thus handling traffic efficiently and optimizing infrastructure costs

Database - Placing more read instances of SQL DB with right database parameter configuration (like buffer size, vCPU’s) helps in cost optimization  
Unnecessary creation of indexes to be avoided and dead rows to be deleted, thus making the database highly performant

Payment processor - Building Mock service (using Java/Python/test tool) simulating the functionality of 3rd party payment processor (like Addition/Subtraction) by feeding in correct responses

### **Storage**

Object storage technique/tools to be used rather than flat file based systems when storing data.

Few considerations to be taken during data storage -   
  
Data retention policy

Data security

Data accessibility period

Data recovery when lost

Data governance/compliance policies

Storage approach - Storing data actively within an SSD helps to retrieve data at a faster rate. Using cloud archival tools like AWS glacier helps in handling data and retrieving them whenever needed keeps costs at a lower rate.

Security - Data needs to be in encrypted format at all times. AWS cloud provider tools provide data security during transit and at rest using strong hashing algorithms.

## **Further Considerations**

### **Previous Art**

*Any previous attempts to solve this problem? Provide references, links or further context.* NA

### **Open Questions**

1. What kind of caching approach to be used?
2. How to mitigate if the 3rd payment processor breaks ?
3. What approach to be taken if any component suffers down-time?
4. Testing tools and environment availability ?
5. Which type of reverse proxy, web server and messaging queue to be used?
6. Observability system set-up for monitoring ?
7. What cloud providers (if any) to be used ?
8. CI/CD deployment approach ?
9. Containerization approach ?

### **Next Steps**

A design review meeting with stakeholders, developers, test team and infrastructure team is needed to discuss the above open questions and approach for the way ahead

Post discussion, a blueprint or prototype of the system will be built and teams will be asked to focus on developing, implementing and preparing the test bed for the system.