1) Handling Post Requests

To handle incoming post requests, we will set up a REST API which will host the POST request at the endpoint /transaction/broadcast. The REST API can be hosted on the organisation's internal server or the cloud such as AWS API Gateway. The POST request will accept a JSON payload with the fields "message_type" and "data". Should any of the fields be missing or wrongly named, the API endpoint will return **HTTP 400** to the client for a bad request.

2) Transaction Processing and Broadcasting:

Once the transaction has been received, the input can be validated, signed, and ready to be broadcast to the blockchain network. If the inputs are invalid, an **HTTP 422** can be sent back to the client for unprocessable content. Once the transaction has been signed, an **HTTP 200** will be sent back to the client under the assumption that there will be eventual successful broadcasting.

Expecting a large number of transactions needing to be broadcasted, a **transaction queue** will be implemented. The queue would manage the transactions after they have been signed, acting as a buffer to manage the load and order of broadcasting to the blockchain networks. Since the maximum transaction time can exceed 30 seconds and are expecting a large volume of transactions, we would implement **asynchronous processing**. The service would employ multiple worker threads and processes to handle the various broadcasting requests. These workers would pick transactions from the queue and attempt to broadcast them.

Should a failure code be returned from the RPC request, the transaction will be pushed to the back of the queue to be broadcasted again after a delay. The delay can be calculated using an exponential backoff along with a jitter. The exponential backoff increases the delay exponentially after each failure, to prevent the server from becoming overwhelmed. The jitter introduces randomness to the delay which would prevent simultaneous retries, which could lead to potential bottlenecks in the broadcasting. However, a maximum retry cap would be implemented to prevent the delay from becoming too spaced out. Once the cap has been reached, the transaction will be flagged for manual review by the admins.

3) Persistent Storage:

After every transaction has been added to the transaction queue, it will be added to a persistent database. After a transaction has been broadcasted, the database will be updated to indicate that the transaction has been successfully broadcasted and if the transaction has failed, the database will also be updated to show the failure. If the maximum cap has been reached, the status would reflect the need for an admin review.

The persistent storage allows for the state of the software to be saved in the case of a restart or unexpected error. For the database, we can choose a **NoSQL structure** as they are horizontally scalable, which could allow for the addition of nodes to a distributed system, which can handle large and growing data. In addition, they also allow for high performance in read and write operations on large datasets due to their distributed nature, allowing for low-latency responses. This allows for the state to be recovered quickly during restart and for the status of transactions to be updated at a faster rate.

4) The page that shows a list of transactions that passed or failed:

Instead of constantly polling the database to find updates, a **web socket connection** can be established to the server hosting the API instead, allowing for two-way communication between the page and the server. Once a transaction has been processed (either successfully or failed), the worker would update the database and create a notification about the status change of the transaction. The server would then recognise the change in transaction status through the notification and send a message to the page over the web socket. The message would contain information about the transaction and its status.

Once the page receives this data, it will update its UI to reflect the new status of the transaction, allowing for real-time changes to the transaction data without the need to constantly poll the database. Nevertheless, the page would implement a fallback polling mechanism with the database in the case that the WebSocket loses connections or encounters an error.

5) Admin Interface:

Similar to the page in point 4, the admin interface would contain all the transactions that are constantly updated through a similar web socket connection. However, the admin interface would also display the queue position or delay time of a failed broadcast, or broadcasts that have requested a manual review. The admin should be able to retry the transaction and by doing so, would send a message through the web socket connection which will force the server to push the requested transaction to the front of the transaction queue, allowing it to be processed next.

6) Scalability:

This service can be designed to be scaled horizontally by utilising **load balancers** to distribute incoming POST requests to multiple instances of these services. This would also decouple the receiving of POST requests from the processing of transactions, allowing for each component to scale individually, introducing modularity.

7) Security:

The service would ensure the signing process remains secure by storing any required private keys securely. In addition, the service would implement rate limiting, to restrict the number of requests a client can make to our service. This would prevent any denial-of-service attacks and ensure a fair usage of resources. In addition, logging can be implemented to record information about activities and errors in the system, allowing for monitoring and investigations into potential security incidents.