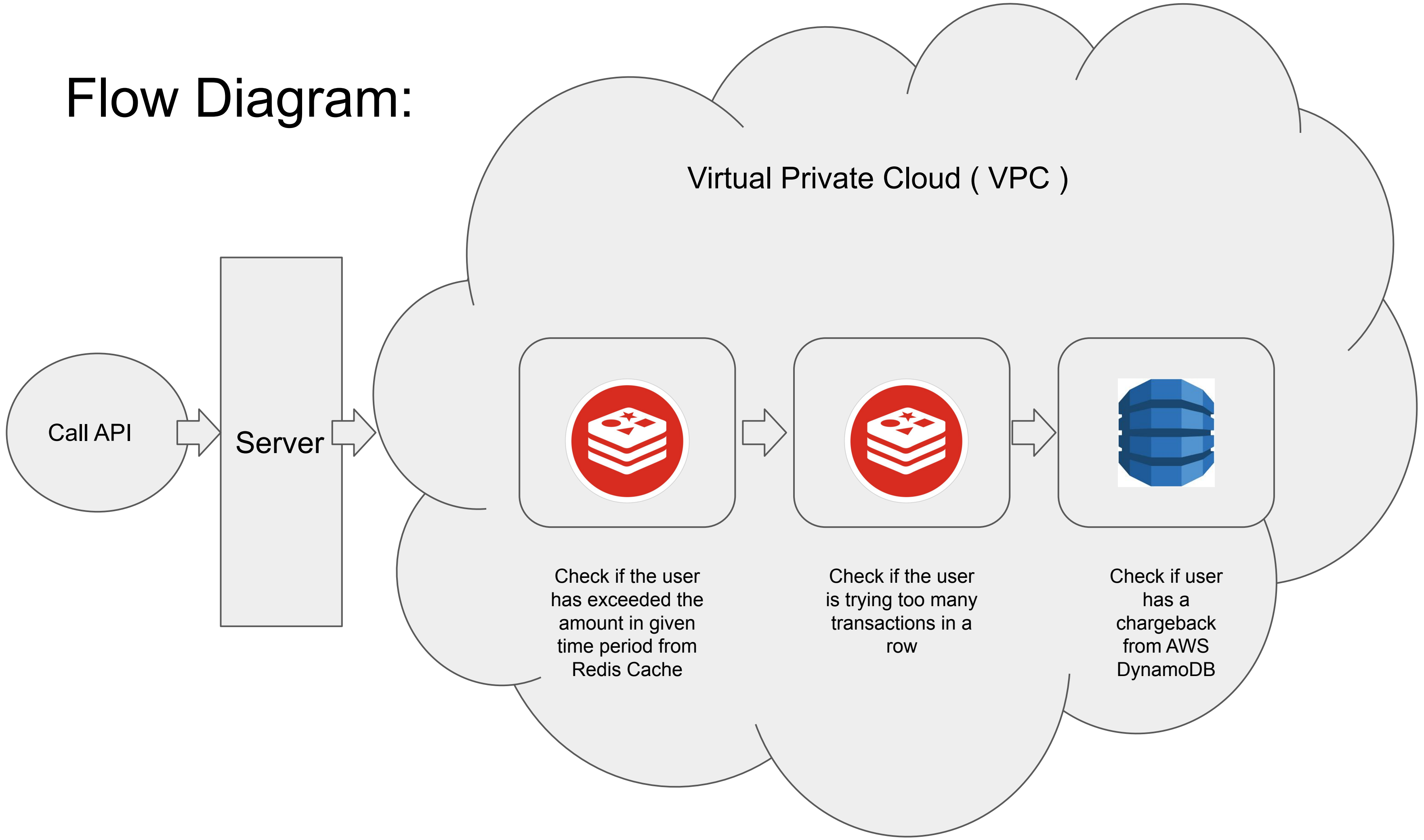
Anti Fraud System.

Technology stack used:

(1) NodeJs - 16.13.0 LTS
(2) AWS DynamoDB (An encrypted key-value pair database.)
(3) Redis Cache (An in memory cached database.)
(4) Json Web Tokens for encryption of content.
(5) Docker.
(6) Github and Git.
(7) Python libraries like Pandas, Numpy, Matplotlib for statistics.

Flow Diagram:


Virtual Private Cloud (VPC)
Call API
Server
Check if the user has exceeded the amount in given time period from Redis Cache
Check if the user is trying too many transactions in a row
Check if user has a chargeback from AWS DynamoDB

Flow:

(1) The API endpoint is called.
(2) The server connects to Redis Cache to check if the user has exceeded the amount for transaction, if yes, send deny transaction response. else
(3) Check further in cache if the user is transacting too much in a row, if yes, send deny transaction response, else
(4) Connect to AWS Dynamo DB in the ‘transactions_with_chargeback’ table to check if the user has any chargeback, if yes, send deny transaction response, else.
(5) Send an approve transaction response.

Reasons for using the stack.

(1) Node Js is a javascript runtime that uses Chrome v8 engine.
(2) Redis Cache for reducing the latency, a traditional database call can take upto 600 ms where as a redis call only take upto 30-50 ms.
(3) Encrypted AWS DynamoDB for security of the data.
(4) The whole of production environment can be in a VPC over a VPN connected by a gateway and will be hosted over a VPS (Virtual Private Server).
(5) Json Web Tokens for authorizing and authenticating every API call.
(6) Docker for dockerizing the application and making it easy to deploy on any machine.

Thought process and architecture.

(1) A Traditional database (Dynamo) or (RDS) or (GSQL).

- Two tables: transactions and transactions_with_chargebacks.
- ‘transactions’ table stores all the transactional data where has_cbk = false.
- Since we know that chargebacks are filed days after a transaction is over, we can flag the has_cbk in transactions table to true and move it to transactions_with_chargebacks table.
- Since chargebacks are rare, the ‘transactions_with_chargebacks’ table would be small and easy to scan.
- To introduce efficient page reads we can create a global index on AWS based on user_id that helps in faster querying.
- This will reduce latency.

(2) An In-memory cache (Redis or Memcached)

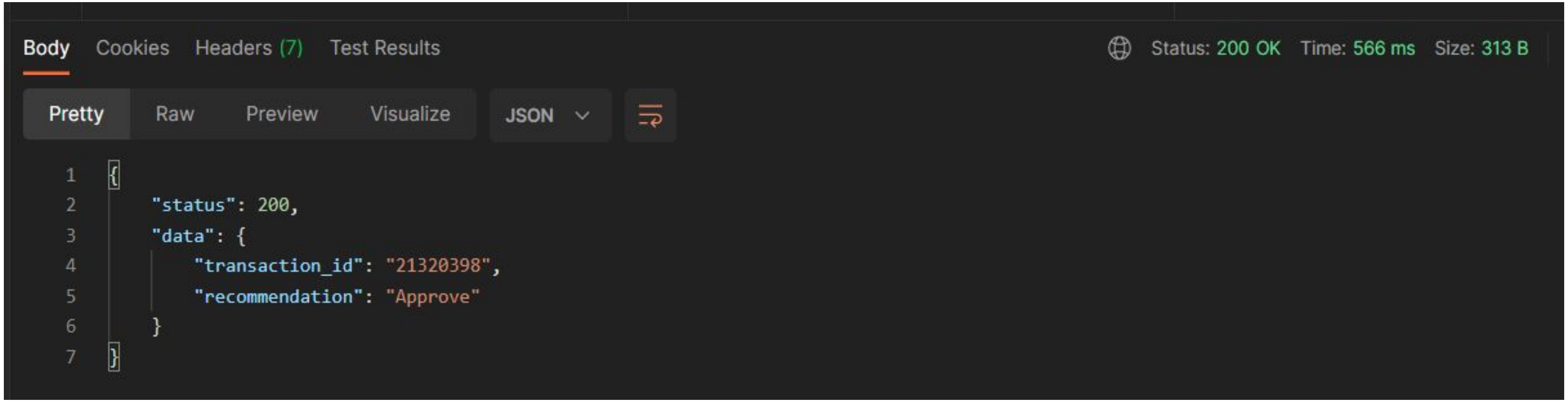
- We can manage a cache where all the recent transactions pertaining to a fixed period will be stored, for example 10 days or 20 days.
- This is reduce latency by a factor of 10 as caches are extremely fast.
- This will also help us with the rule-based prediction when we want to find amount exceeded for a user in a given time period or too many transactions in a row.
- We can write a lambda function or google cloud function that triggers transferring of data from the cache to the actual database when the date expires.

(3) A Rule based system.

- All incoming payloads will be first subjected to the cache, so if recurrent transactions are found in the cache we can deny the request without even querying the database.
- Firstly, amount exceeded will be checked, if it passes,
- Secondly, row transactions will be checked, if it passes,
- Thirdly, chargeback would be checked.
- If all the checks passes, we can implement a score-based ML model to predict a score in the chain, the ML model can also be a Lambda function that is triggered on successful dynamo call.
- If everything is fine, transaction is approved or rejected.

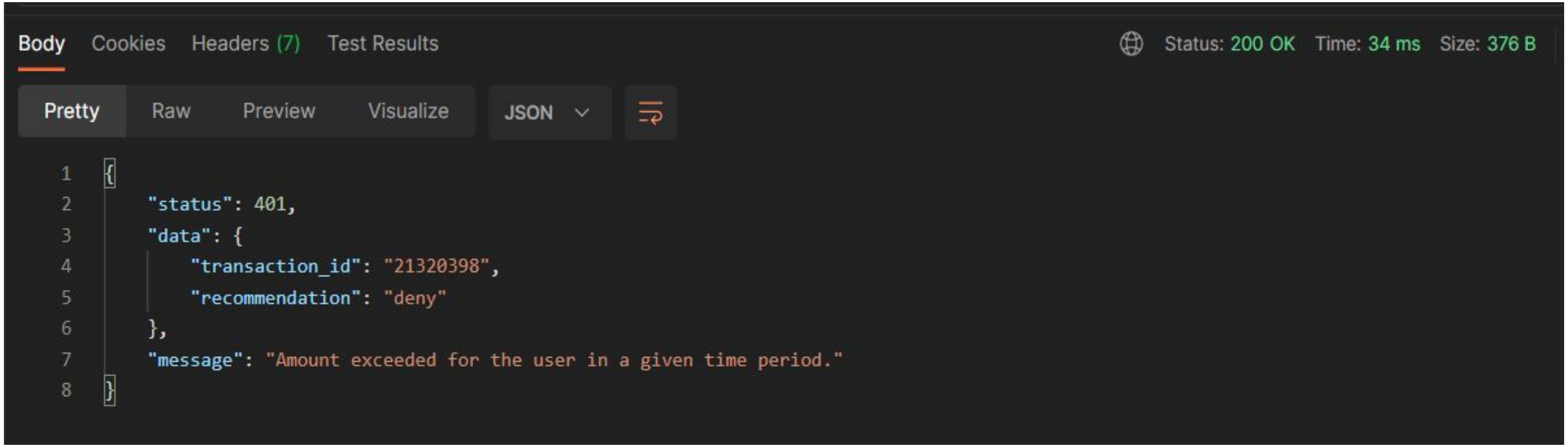
Latency.

- Latency will be reduced drastically by using an in-memory cache which can further be improved by sharding and clustering.
- For instance, let’s see this two api calls, one request dynamoDB and other comes from redis cache and check the response time.



Status: 200 OK Time: 566 ms Size: 313 B

```
{
    "status": 200,
    "data": {
        "transaction_id": "21320398",
        "recommendation": "Approve"
    }
}
```


Body Cookies Headers (7) Test Results — Status: 200 OK Time: 34 ms Size: 376 B

Pretty Raw Preview Visualize JSON

```
{
    "status": 401,
    "data": {
        "transaction_id": "21320398",
        "recommendation": "deny"
    },
    "message": "Amount exceeded for the user in a given time period."
}
```

566 ms for database call and 34 ms for cache call.

Security.

Security in such systems can be achieved by implementing DevSecOps.

Some ideas.

- We can use a singe cloud provider network such as AWS or GCP, like every service used will be in the same network, for example Elasticache for caching, Dynamo or RDS for database, EC2 or Lightsail for hosting. This will give us TLD verification and support.
- All services can be in our virtual private cloud with a unique subnet.
- Using AWS or GCP encrypted databases.
- Using top tier encryption for API calls.
- Using https over TSL/SSL

Architecture.

- Used a polylith (microservice) architecture instead of a monolith architecture.
- The architecture used contains two broad elements: backend and cloud.
- Backend is diversified into microservices such as the redisClient, dynamoClient, jsonClient and follows the Model-View-Controller architecture.

Coding Style.

(1) Conservative coding style with emphasis on documentation.
(2) Common problems like pyramid of doom, garbage values are eliminated.
(3) Block coding implemented, with each function resolving a single value.
(4) Promises and callbacks used as per industry standards.
(5) Modules are exported from each file to be used in another file.