



SMU

SINGAPORE MANAGEMENT
UNIVERSITY

CS301 IT Solution Architecture

Group Proposal

AY: 2021/22 T1

Prepared for:

Prof. Ouh Eng Lieh

Prepared by G1T5 (GitHub Team: g1-team5):


Emmanuel Oh Eu-gene 01327148

Liew Xi Wei 01419103

Tan Qi En 01365009

Tay Rui Xian 01370868

Yap Bing Yu 01410869

- 
- AWS Diagram needs to follow conventions.
 - Maintainability - good justifications to the decisions
 - Good answer to the Lambda vs EC2.
 - Service splitting for processing & ingestion looks good
 - For encrypted PANs, do all parts of your app require access to the raw PAN? (e.g. the metadata for the card type); Think of approaches for minimising exposure of the raw PAN across the diff apps
 - File storage: You can skip Exavault and use AWS SFTP
 - DDoS w Shield & WAF
 - Prepared stmts & ORM for SQL injection
 - MITM - TLS everywhere across your app for strengthened internal security too?
 - Since you're on Aurora, have a crack at multi-master if you have time"

Background and Business Needs

In today's increasingly crowded debit/credit card industry, there are many organizations providing competitive offerings with a multitude of benefits, and it is vital to provide fast and reliable transaction processing services to keep users up to date with their latest rewards. We also want to offer agile campaign management to capitalize on trends and increase user spending while also providing them with attractive rewards. These rewards should be presented to the users in a clear and intuitive interface to allow them to have a good understanding of the progress made. These features will help us to increase the perceived value of the cards from the users' perspective, allowing our affiliated banks to maintain brand loyalty and grow market share.

Broadly, our application aims to fulfill 3 primary business needs:

1. Process spend transactions from CSV file, and from API requests
2. Store and display points accumulated by customers for various reward programs across different cards
3. Launch and manage campaigns by banks, ensuring that the customers will receive benefits according to the relevant rules

Stakeholders

Stakeholder	Stakeholder Description	Permissions
Customers	Customers open accounts and sign up for credit cards issued by banks. Each customer can sign up for multiple cards, with each offering different reward programs.	<ul style="list-style-type: none">- View accumulated rewards by program- View accumulated rewards by card- View eligible campaign promotions
Banks	Banks manage user funds and issue credit cards to customers. They are responsible for recording transactions made by the customers, accumulating points for reward programs, and launching campaigns with merchants.	<ul style="list-style-type: none">- Add/Update customer transactions, card spend campaigns, spend exclusions, processed points/ miles/ cashback conditions- View overall transactions by card program, campaign, and merchant
Merchants	Merchants collaborate with the banks to provide rewards or incentives to customers whenever they spend on their cards as part of short-term campaigns.	<ul style="list-style-type: none">- View total transactions related to them

Key Use Cases

Transaction processing (CSV/API)	
Use Case ID	1
Description	<p>a) Process a large batch of transactions in CSV file format. This operation will be done on a daily basis. This is a crucial feature as many banks rely on file transfers to process large batches of transactions.</p> <p>b) Process a transaction that is sent through an exposed API endpoint. This is a feature that enables more dynamic processing of transaction records with a faster turnaround time, so that customers can see their spending and rewards updated sooner.</p>
Actors	Bank
Main Flow of Events	<p>1(i). Bank uploads a CSV file to a cloud FTP server on ExaVault</p> <p>1(ii). Bank sends transactions in a JSON object using HTTP POST to the application's API endpoint (API Gateway).</p> <p>2(i). For CSV files, system ingests individual transactions from the CSV file with serverless functions (AWS Lambda) and passes it to a message queue (Amazon SQS) for further processing</p> <p>3. Based on details of each transaction, auto-scaling servers (AWS EC2, Batch) apply rules to update customer details, card details, and points accumulation for rewards</p> <p>4a. Upon passing validation processes, changes are written into a relational database (AWS RDS).</p> <p>5a. If campaign rewards are applied, a notification email is sent to the user.</p>
Alternative Flow of Events	<p>4b. Upon failing validation processes, changes are discarded, and the transaction is pushed back into the message queue (Amazon SQS).</p> <p>5b. Upon 5 repeated failures of steps 3-4, message is placed into a dead letter queue, for transactions with errors in processing.</p> <p>6b. An email is sent to the developers for follow up and debugging.</p>
Pre-conditions	<p>i) - Bank must have a valid collection of transactions in CSV file format, uploaded successfully to ExaVault</p> <p>- Application must have a valid API key to access ExaVault's API</p> <p>ii) - Bank must have a valid collection of transactions in JSON file format</p> <p>- Bank must be able to access API endpoint over HTTPS with a valid API key</p>
Post-conditions	<p>- Cashback, miles and reward points accumulated are stored within the database.</p> <p>- New card details are stored within the database.</p> <p>- New customer details are stored within the database.</p> <p>- Transactions that cannot be processed are stored in a dead letter queue.</p> <p>- Notification email/SMS sent out to user(s) regarding applied campaign benefits</p>

Managing campaigns and exclusions	
Use Case ID	2
Description	Posts / updates of campaigns and exclusions are done by the bank through our frontend, allowing them a convenient interface with input validation.
Actors	Bank, Merchant
Main Flow of Events	<ol style="list-style-type: none"> 1. Bank sees the form on the frontend view with details like from date, to date and etc to be filled in. 2. During the filling up of the form, there are necessary error messages to prevent invalid input. 3. After filling the form with necessary, valid details, the bank presses the submit button. 2. Based on these details provided, the frontend will send a HTTP POST request to an API endpoint (API Gateway). 3a. When the request passes the validation process in the backend (AWS Lambda), details on the new campaign will be written to the relational database (AWS RDS). 4a. Upon the creation / update of said campaign, a HTTP 200 code will be returned for a successful update or deletion, and 201 for a successful creation (API Gateway).
Alternative Flow of Events	<ol style="list-style-type: none"> 3b. Upon failing the validation process, all changes are discarded and the backend (AWS Lambda) generates an error message which is returned as a HTTP response (API Gateway). 4b. Frontend displays an error message to guide the bank on how to proceed with mitigating the errors.
Pre-conditions	<ul style="list-style-type: none"> - Bank must fill in the form with valid details. - Bank must have a valid user id to access the frontend website in order to fill the form.
Post-conditions	<ul style="list-style-type: none"> - Campaign or exclusion details are stored within the database.

View rewards, miles and cashback	
Use Case ID	3
Description	Retrieve transaction and rewards data from
Actors	Customer
Main Flow of Events	<ol style="list-style-type: none"> 1. User navigates to the rewards page to view their rewards. 2. Frontend sends a HTTP GET request (with default filtering and sorting options) to the API the backend provides (API Gateway). 3. Backend service (AWS Lambda) queries the relational database (AWS RDS) for accumulated points and related transaction details to serialize as Reward objects. 4a. A HTTP response is returned (API Gateway) to the frontend with Reward objects enumerated in JSON. 5a. Frontend renders and displays these Reward objects to the user.

Alternative Flow of Events	4b. Upon a failed query to the database, the backend (AWS Lambda) generates an error message which is returned to the user as a HTTP response (API Gateway). 5b. Frontend displays the error message and helpful resources for the user.
Pre-conditions	- User viewing the transactions must be a bank card owner, with details stored within the application database - Frontend must specify valid filtering and sorting options in the request.
Post-conditions	- HTTP GET request is idempotent so no data is changed.

Quality Attributes

Maintainability

Code

We will develop our application according to object-oriented design principles with C# on the .NET framework in order to promote modular and reusable code. Each of our classes and interfaces will be documented with well-defined, consistent roles and responsibilities. This will enable developers to easily identify areas where application logic is implemented and make it simple to refactor or build new features. We will also adopt a test-driven approach in development enforced through continuous integration and deployment pipelines in order to ensure that any changes made do not alter the core functionalities previously asserted.

Architecture

One key strategy for maintainability in architecture is by building the application on top of the AWS service stack. In this layered architecture, physical infrastructure and foundational services are maintained by AWS, while our scope of responsibility as developers lies in managing services and deploying applications. In addition to providing a consistent interface for deploying various services, AWS also allows us to configure cloud provisioning with infrastructure as code through CloudFormation templates. This allows us to achieve automated, consistent and repeatable deployments with minimal changes to underlying configurations.

Security

No	Asset/Asset Group	Potential Threat/ Vulnerability	Possible Mitigation Control
1	Data in transit AWS Cognito, SSL Certificate	MITM attack a. Hijack user credentials (Confidentiality) b. Post invalid transaction details (Integrity)	Enable SSL and HTTPS and ensure that it is used for the entire application a. Add 2 factor authentication b. Only allow whitelisted hosts to post requests to our API endpoint
2	Data at rest AWS RDS, AWS S3	SQL Injection a. Retrieve unhashed password (Confidentiality), Tamper user password in the database (Confidentiality, Availability) b. Alter the rewards table in the database (Integrity)	User input sanitisation Use prepared statements Implement AWS WAF Database should not be publicly accessible a. Hash passwords
3	Server AWS EC2, Lambda	DDoS Attack Attackers overloading the server with requests to disrupt service	Implement AWS WAF and utilise AWS Shield <i>Good F2B</i>

Performance

Framework

Our application will be written in C# on .NET. According to TechEmpower Performance Rating, .NET generally outperforms other popular frameworks in tasks like JSON Serialization and database access. With the use of appropriate .NET functionalities alongside well-designed algorithms and data structures, we will be able to develop a highly performant backend service. We also aim to utilize parallelization and concurrency to handle large volumes of write jobs simultaneously.

Architecture

Our application will be using AWS' services and functionality to achieve high performance. EC2 instances will be deployed in regions where the services are needed to reduce network latency between the client and servers, and will scale dynamically to meet workloads. Lambda functions will be running with provisioned concurrency to minimize cold start delays. We will use Aurora databases to achieve 5 times the throughput of a standard MySQL database and take advantage of integrated in-memory caches. We decided against using a dedicated in memory cache as we do not expect that the same query repeated multiple times will accept stale data, for example a user refreshing the page repeatedly to view their latest transactions, and the integrated cache will suffice. We will also be scaling the read replicas in order to offload the write database and increase throughput. We will use AWS' CloudFront CDN to ensure low latencies for our website regardless of user location.

Availability

Our application will be architected to make full use of AWS' functionalities to maximize availability. Aurora databases will be in a multi-AZ read replica configuration, and will utilize AWS' automatic failover functionality to promote a replica to the primary instance in the event of downtime on the previous primary instance. EC2 instances will utilize cross zone auto scaling, ensuring that there are instances available in the event of individual instance or regional downtime. Components are designed to be stateless, relying on JWTs to store user authentication details, allowing minimal disruption if a failover occurs mid session. Lambda is run across multiple AZs by default, and is highly available. Static resources for our frontend are hosted in S3 buckets across multiple regions, which when combined with CloudFront's automatic origin failover ensure high availability for our website.

Proposed Budgets

Development Budget

Activity/ Hardware/Software/ Service	Description	Cost
Developing code, tests and documentation	Total cost to develop and test services for the product	5 * 30 man-hours/person = 150 man-hours
Continuous integration and deployment - AWS CodePipeline	Costs associated with building, integrating and deploying code	\$200 (AWS credits)
Cloud services for development and staging environment	Costs associated with experimental cloud services used for development	

Production Budget

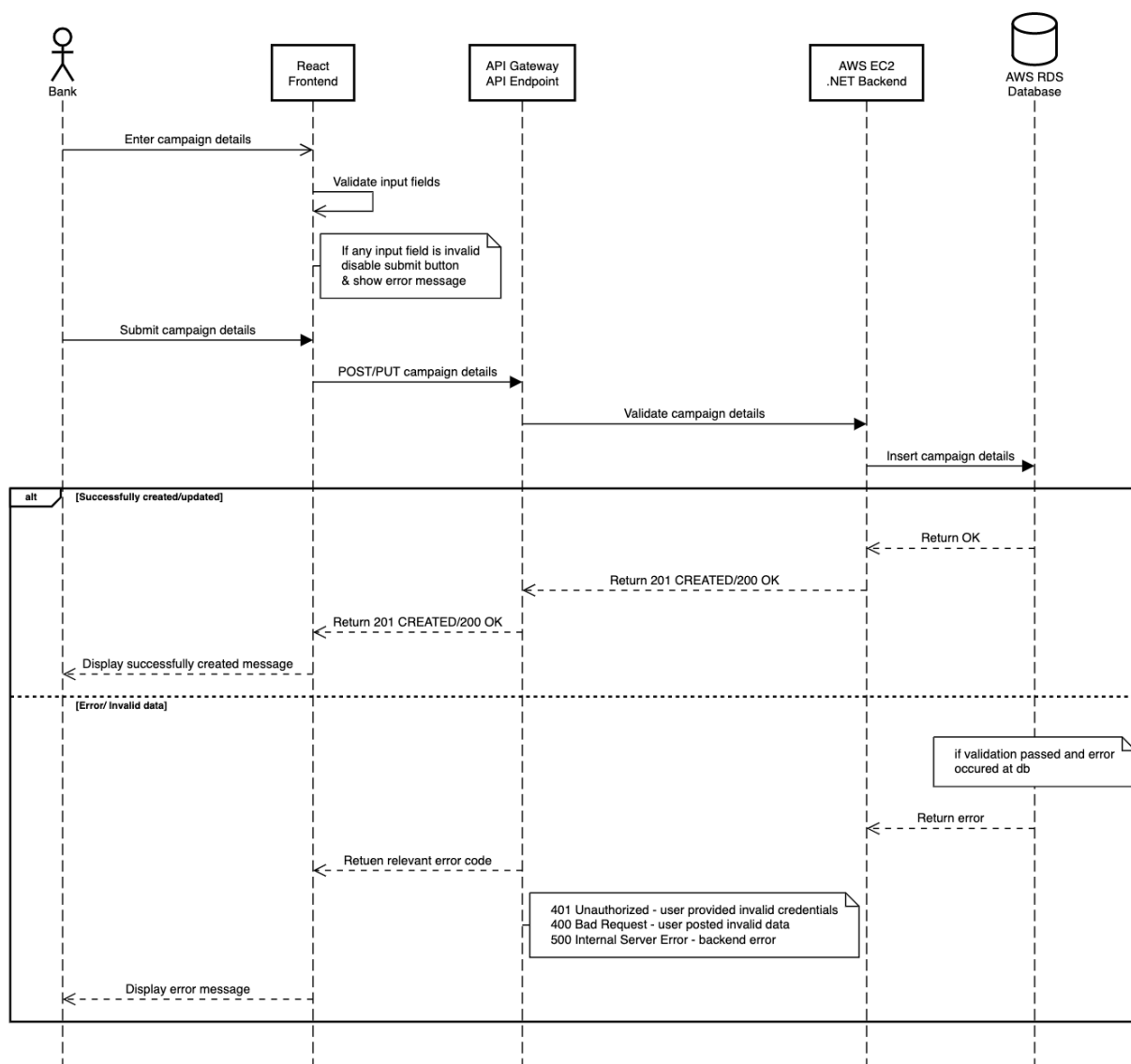
Activity/ Hardware/Software/ Service	Description	Cost (est. monthly)
Compute - Elastic Compute Cloud	We run workloads likely to exceed the 15 minute lambda timeout (CSV ingest and preprocessing) and with sustained demand (transaction processing) on EC2	\$142.80 c5.xlarge on demand, approx 28 compute-hours per day
Compute - Batch	To run batch jobs longer than 15 minutes	Free
Compute - Lambda	Serverless compute to preprocess transactions from the API, as well as for CRUD operations Consideration: EC2 vs Lambda The preprocessing of CSV is event-driven. If EC2 were to be used, there could be high idle time for the instance that serves this purpose.	\$28.37 10 reserved concurrency, 2000000 requests lasting 200 ms, 256 MB allocated to function
Storage - Aurora	We use the Aurora PostgreSQL database as it provides 3x the performance of a standard PostgreSQL database - required since we will have a high write throughput every day. Consideration: RDS vs Aurora Aurora scales faster, further and serves queries more quickly - read replicas are necessary to ensure high write throughput. Aurora also has faster recovery to minimize the delay in our write jobs.	\$249.64 2x db.t3.large, 100 GB storage with 3 million IOs per month
Storage - S3	S3 is used to store the transaction CSVs securely, with low cost. Consideration: S3 vs EBS vs EFS Main advantage of S3 is the ability to store objects cost effectively and durably where IOPS is not a primary concern	\$0.01 0.69 GB, standard IA, write once CSV size estimated to be 230KB / 1,000 * 1 million = 23MB 23 * 30 = 690 MB per month
API Gateway	API Gateway serves as a reverse proxy that intercepts incoming requests before routing to the services. Together with Cognito, it also allows us to handle authentication outside of the backend services.	\$0.01 2 million requests of 512KB
Cloudwatch	A monitoring / observability service to provide developers with more insights on application metrics.	Free
Cognito	Cognito is used together with API Gateway as a user management service to authenticate and authorize access to the backend services.	Free 50000 monthly active users

ECR	Hosting for Docker images	\$1.00 10 GB
SQS	SQS used as a message queue service that allows us to decouple the preprocessing and writing to database service.	\$12.00 30 Million requests
Cloudfront	Content delivery network	\$0.08 Less than 1GB data transfer
Maintenance	Code revisions and changes	5 man-hours
Total financial cost	All costs excluding man hours	\$433.91

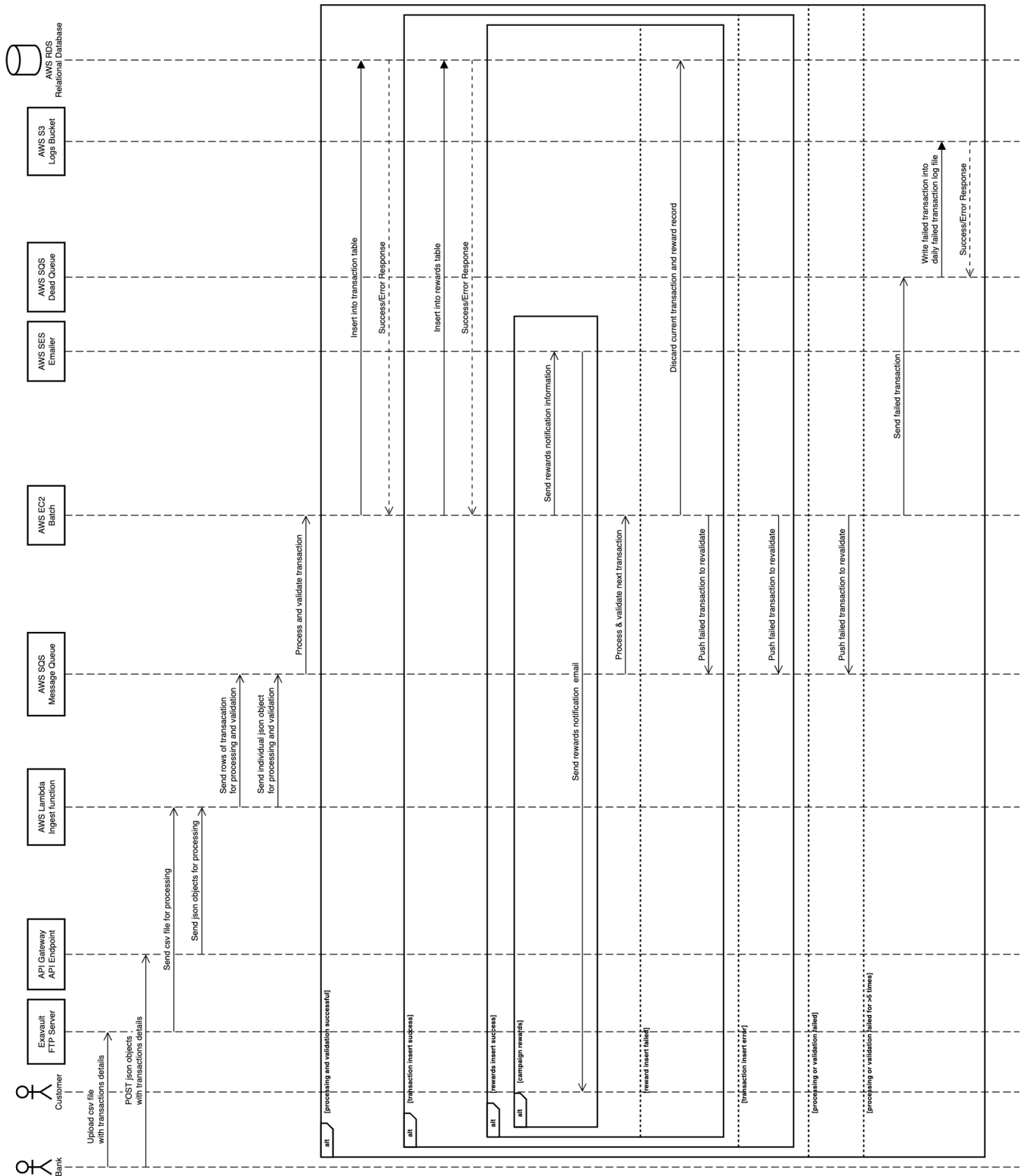
Views

Sequence diagram

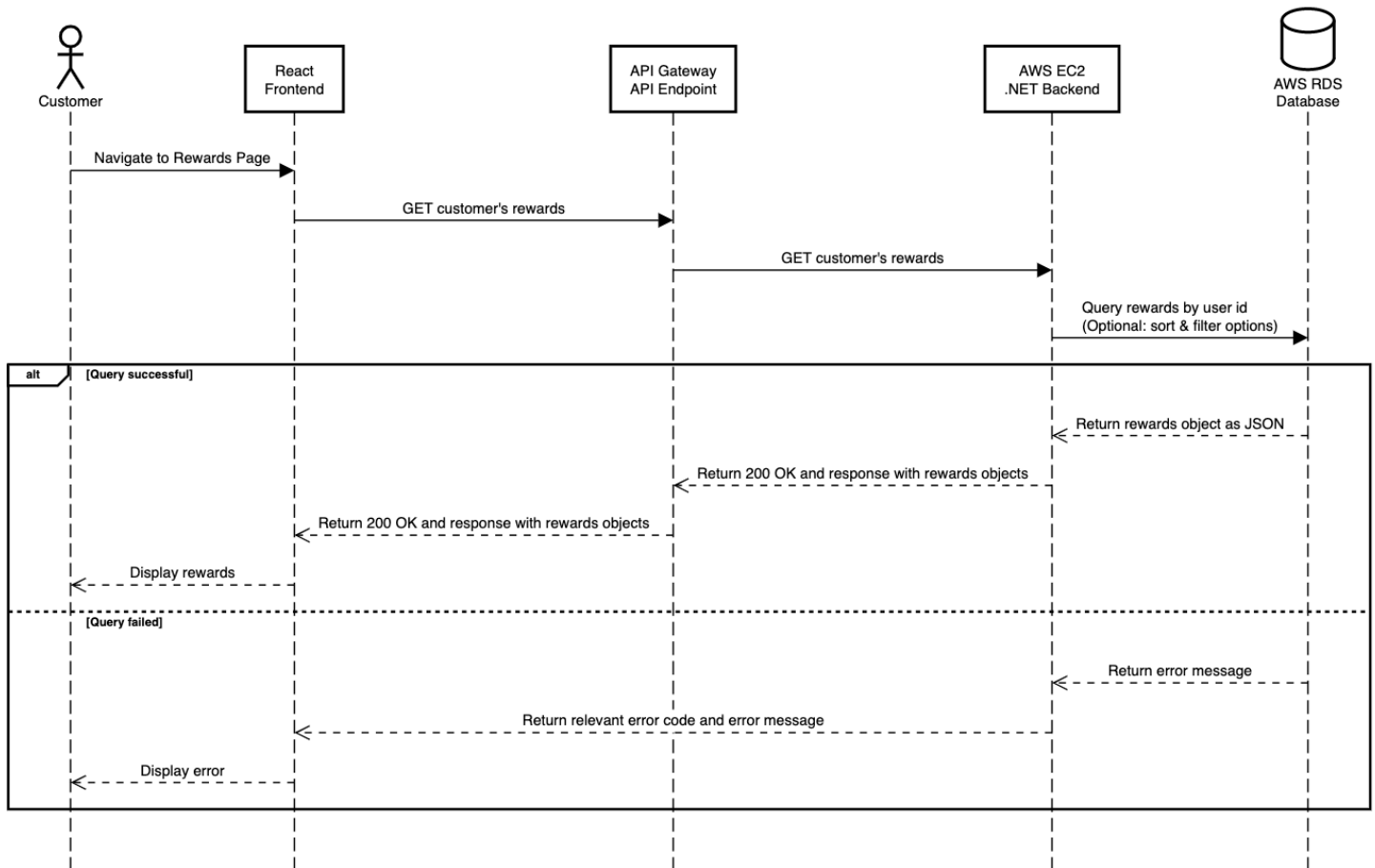
1. Posting campaign



2. Transaction Processing



3. View Rewards



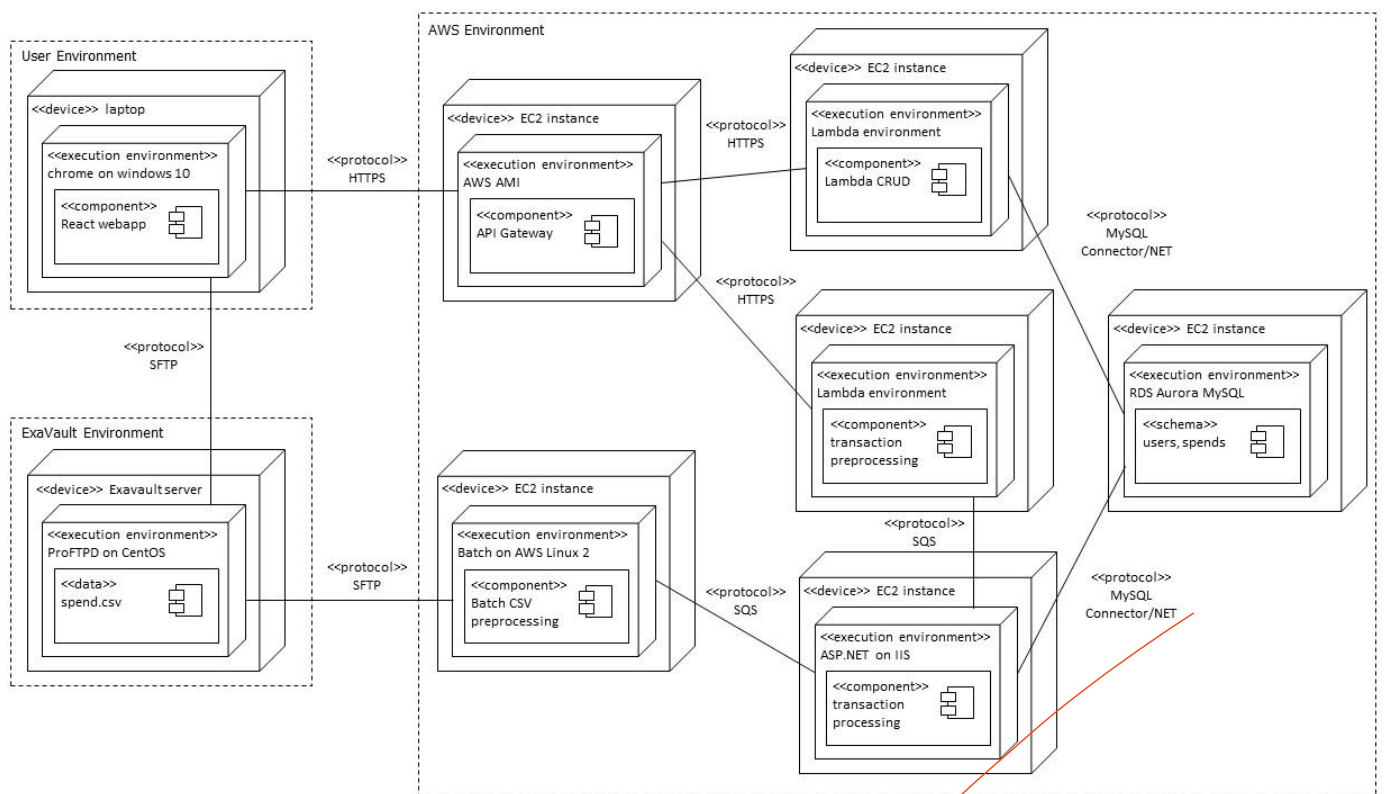
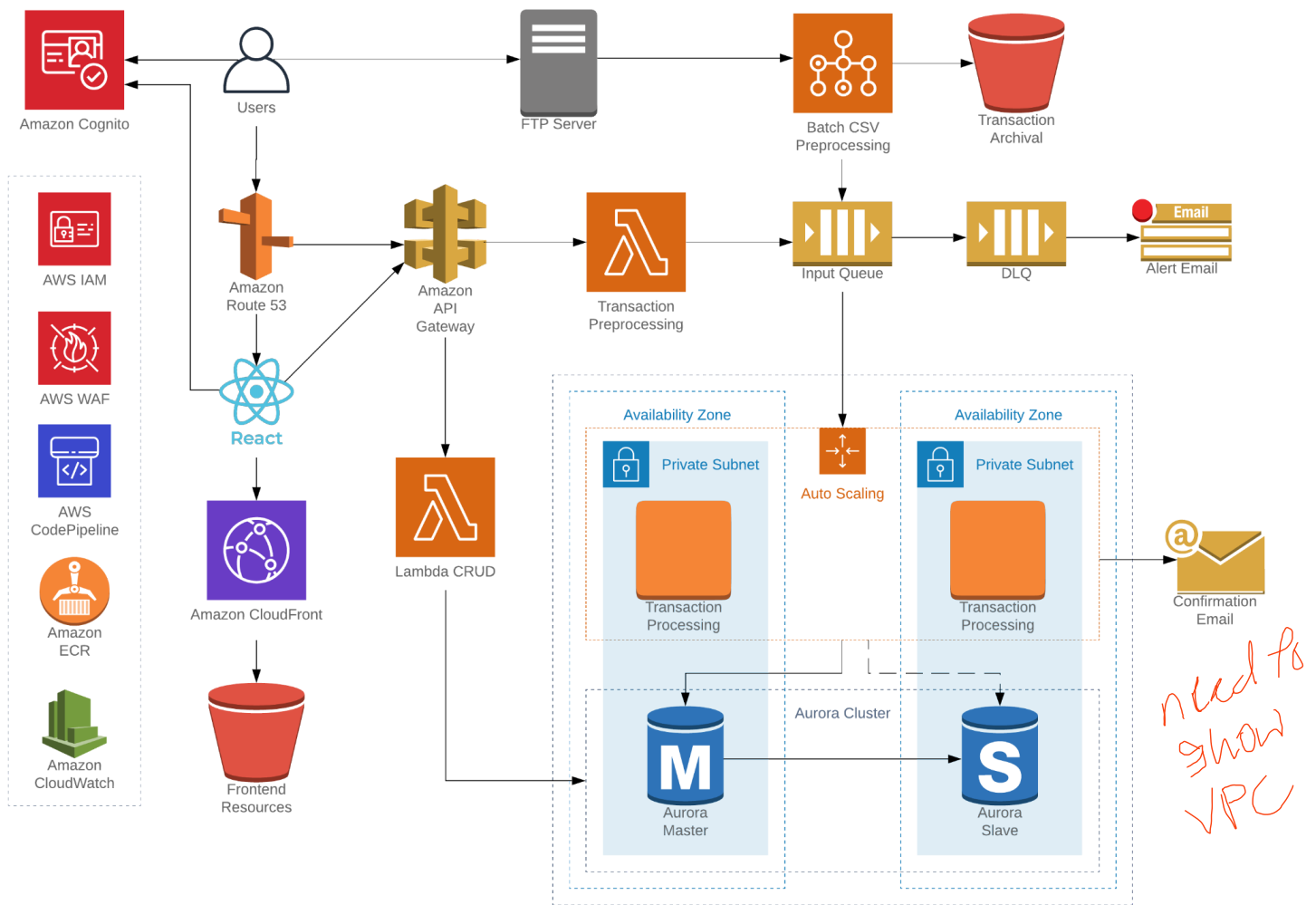
Deployment diagram

There are two types of users, banks and customers. Authentication and authorization for both will be handled by Cognito, across both the web app and API gateway. Banks can upload CSV files containing transaction details to their FTP server, which we retrieve via a scheduled Batch job. This job will parse the CSV file into individual transactions, perform input validation, and enqueue them into the input SQS queue. It will also store a copy of the CSV file in S3 for archival purposes. Alternatively, individual transactions can be posted via API gateway, which will invoke a Lambda function to handle preprocessing of the transaction details, input validation, and enqueueing. It also handles the initial API response.

Consuming from this queue, we have EC2 instances in a multi AZ auto scaling group that read transactions and update the corresponding users' transaction history and rewards accordingly based on prevailing campaigns and exclusions, as well as sending a confirmation email when campaigns are applied successfully. This data is stored in an Aurora cluster in a master-slave multi AZ read replica configuration. Both the EC2 and Aurora instances are in a private subnet. Failed transaction processing jobs are returned to the queue, and in the event that a transaction fails to be processed after a certain number of retries, it will be sent to a dead letter queue for follow up and debugging, and an alert email will be sent to the developers to notify them of the issue.

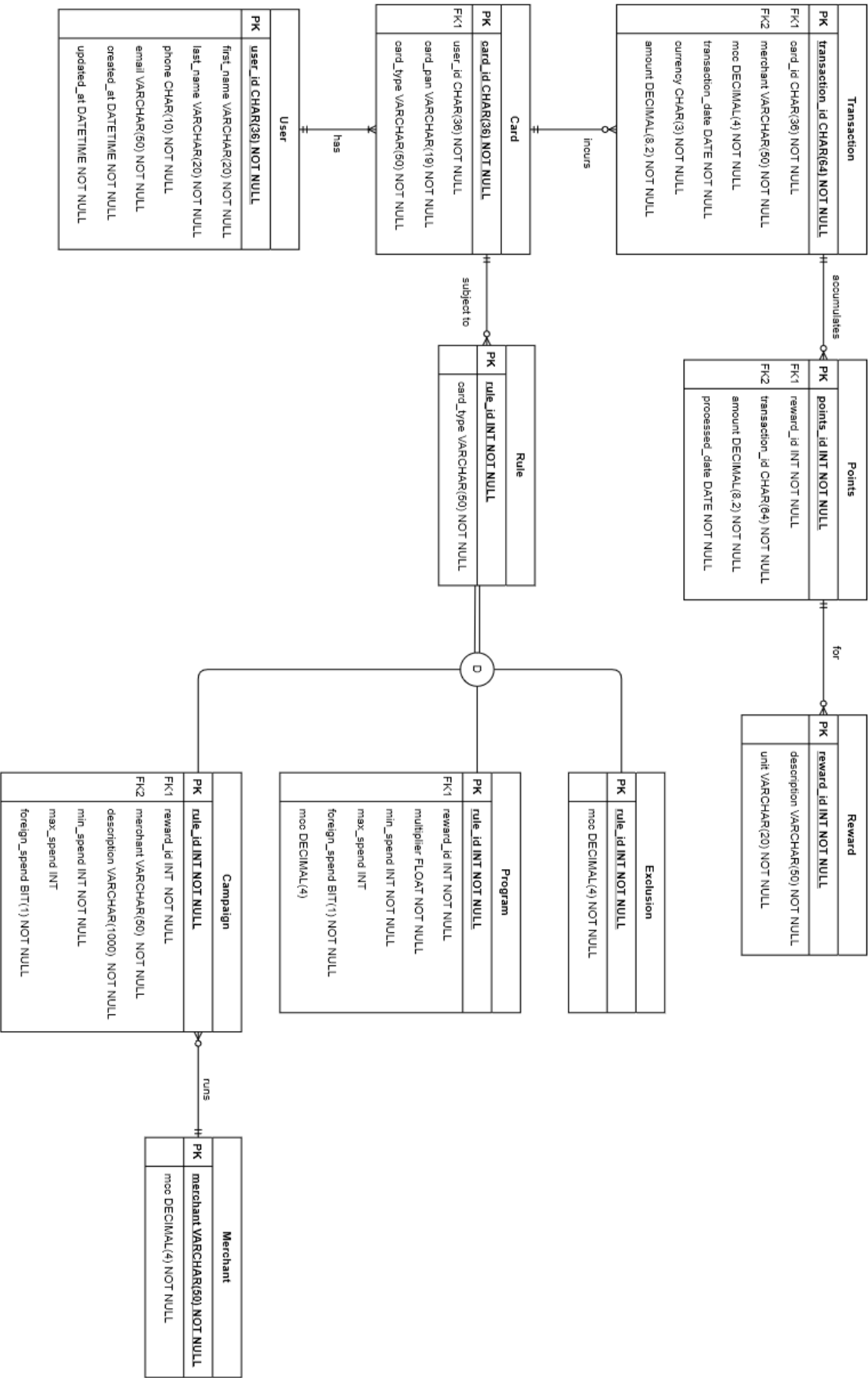
Customers will access our product via the react webapp, which will be hosted in S3 buckets in multiple regions, and distributed via the CloudFront CDN. The frontend is backed by API gateway, which invokes lambda functions that perform CRUD operations on the database. CI/CD will be managed by CodePipeline. IAM roles will be used to grant services access to other AWS resources by the principle of least privilege. ECR will be used to store Docker images that will be run on batch and auto scaling groups. CloudWatch will be used for logging and monitoring.

For connections between devices, we use SFTP and HTTPS for increased security in transit, and MySQL Connector/NET to provide a standardized, abstracted interface with the MySQL database.



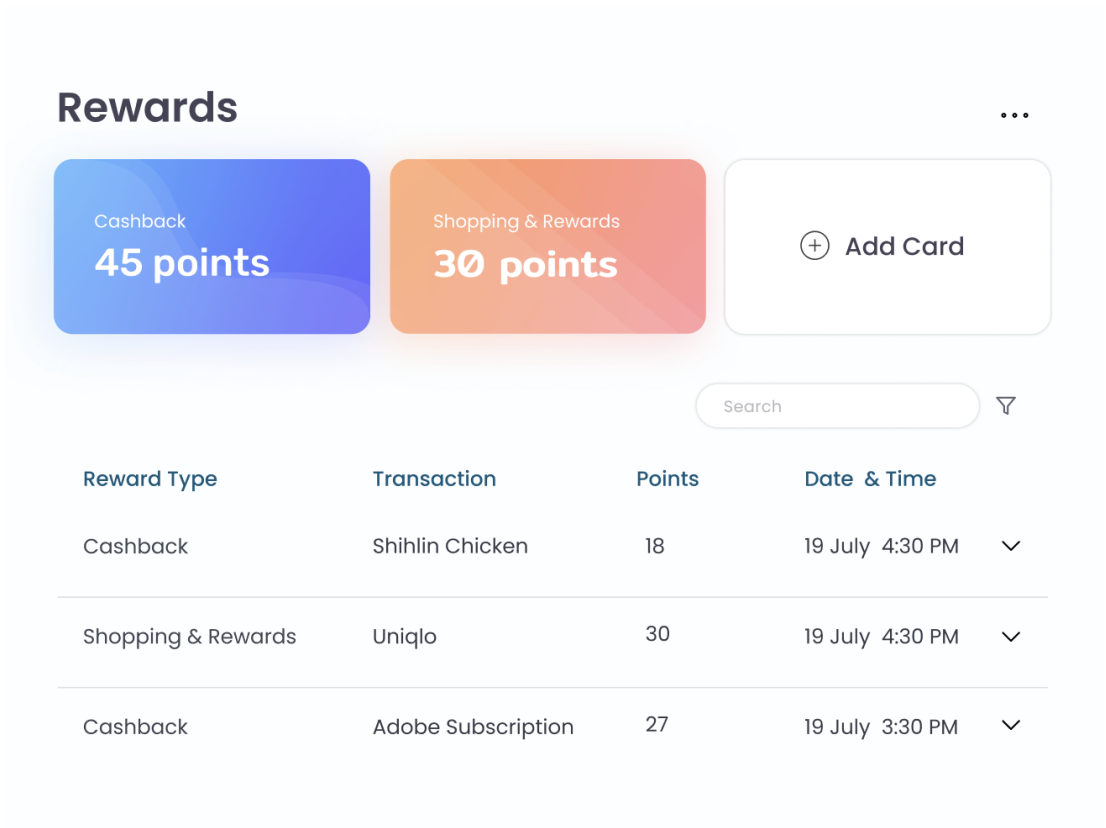
Appendix

Database schema



Frontend (Rewards Screen) mockup

Customers can view their reward points by logging into our web application. This frontend interacts with our API Gateway to retrieve relevant data.



Campaign rewards EDM mockup

After a campaign benefit is applied, an alert email will be generated and sent to the customer.

