SMU
SINGAPORE MANAGEMENT UNIVERSITY

CS301 IT Solution Architecture

Group Project Report

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

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.	- View accumulated rewards by program - View accumulated rewards by card
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.	- 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.	- View total transactions related to them

Key Use Cases

Transaction processing (CSV/API)	
Use Case ID	1
Description	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. 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.
Actors	Bank

Main Flow of Events	1(i). Bank uploads a CSV file to a cloud FTP server on **ExaVault** 1(ii). Bank sends transactions in a JSON object using HTTP POST to the application's API endpoint (**API Gateway**). 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 3. Based on details of each transaction, auto-scaling servers (**AWS Fargate, Batch**) apply rules to update customer details, card details, and points accumulation for rewards 4a. Upon passing validation processes, changes are written into a relational database (**AWS RDS**).
Alternative Flow of Events	4b. Upon **failing** validation processes, changes are discarded, and the transaction is pushed back into the message queue (**Amazon SQS**). 5b. Upon 3 repeated failures of steps 3-4, the transaction is stored within a failed_transaction table in the database, for transactions with errors in processing. 6b. An email is sent to the developers for follow up and debugging.
Pre-conditions	i) - Bank must have a valid collection of transactions in CSV file format, uploaded successfully to ExaVault - Application must have a valid API key to access ExaVault's API ii) - Bank must have a valid collection of transactions in JSON file format - Bank must be able to access API endpoint over HTTPS with a valid API key
Post-conditions	- Cashback, miles and reward points accumulated are stored within the database. - New card details are stored within the database. - New customer details are stored within the database. - Failed transactions are stored in a table and developers are notified

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	1. Bank sees the form on the frontend view with details like start date, end 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	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	- 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	- 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 existing card programs and campaigns
Actors	Customer
Main Flow of Events	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 Requirements

Speed	
Ingestion	< 5ms (average from 14,000 requests)
Scalability	
Auto Scaling Group	Fargate's Auto Scaling Group increases task count when CPU utilization increases above 70% and reduces tasks count when CPU utilization decreases below 20%
Ease of Maintenance	
Architectural Style	Tiered
Framework Design Pattern	.NET Core (Onion Architecture)
Data Security	

Amazon GuardDuty	Protect APIs from Anomalous Behaviour
AWS WAF	Specify Web ACL for APIs using a blacklisting approach
Personal Information	AWS Cognito, JWT Token to access APIs in API Gateway
File Retention Policy	
AWS S3	30 days retention policy before transferring files to infrequent-access
Resilience and Disaster Policy	
AWS Aurora RDS	Read Replica in multiple AZs
AWS S3 Standard IA	S3 Standard-IA objects are resilient to the loss of an Availability Zone.

Key Architectural Decisions

Architectural Decision - Parsing all transactions into SQS (Simple Queue Service)	
ID	1
Issue	Transactions can be sent to the application as JSON objects or in csv format, which can be tricky for backend services to manage
Architectural Decision	Create a single point of access for processing transaction records, acting as a facade and abstracting the complexity away from the backend service
Assumptions	Assume that JSON and CSV transactions have the same fields and formats
Alternatives	Separate services for processing JSON transactions and CSV transactions
Justification	It requires more effort to manage 2 different code bases in terms of development/deployment. Furthermore, it is harder to scale those services separately as compared to handling scaling for a single service.

Architectural Decision - Deploying backend service to ECS as a Fargate Cluster	
ID	2
Issue	Challenging to configure load balancing, auto-scaling groups and networking rules for ECS services when deployed on EC2 instances
Architectural Decision	Use AWS Fargate, which removes the operational overhead of provisioning, scaling and container orchestration
Assumptions	Backend services will only be deployed as docker containers
Alternatives	Deploy containers as Clusters on EC2 instances, configure networking rules, associate auto-scaling groups and handle load balancers for traffic
Justification	Faced challenges with configuring infrastructure given the time constraints in development, aiming for the best solution that meets current needs

Architectural Decision - .NET framework performance	
ID	3
Issue	Minimally, 700 transactions have to be ingested together with the points allocated according to Programs, Campaigns and Exclusions
Architectural Decision	.NET Core Framework deployed on ECS Fargate
Assumptions	.NET Core is almost as developer friendly as Spring Boot
Alternatives	Use Rust or SpringBoot as the ingesting service framework
Justification	Rust has a high learning curve and time is limited for the project implementation due to its lifetimes and the ownership/borrowing system. .NET Core has a higher performance score according to benchmarks, scoring 89th, 51st on JSON Serialization and Single Query while Spring ranked more than 190th for both

Architectural Decision - Reading from CSVs as an AWS batch job	
ID	4
Issue	A large CSV file needs to be ingested from an FTP server at the end of each day, and the records sent for further processing.
Architectural Decision	Use a batch job triggered by an EventBridge CRON event to poll the FTP server for the CSV file, then enqueue each row into an SQS queue.
Assumptions	The job will take a lot of time and memory.
Alternatives	Cron job on a persistent server, AWS Lambda, AWS SFTP
Justification	A persistent server is an ineffective utilization of resources when there is no processing being done and the server is idling. The large job size means it is likely to exceed the 15 minute execution limit or 3GB memory limit of lambda functions. AWS SFTP is expensive as it is billed hourly, not by transactions performed. We use a single large instance for this job as splitting the CSV file into segments for parallel processing using horizontal scaling would take almost as much time as just doing it on one instance, since the job is bottlenecked by disk and network IO capacity. Using batch we can select instance types optimized for this workload.We also benefit from spot pricing during periods of low demand.

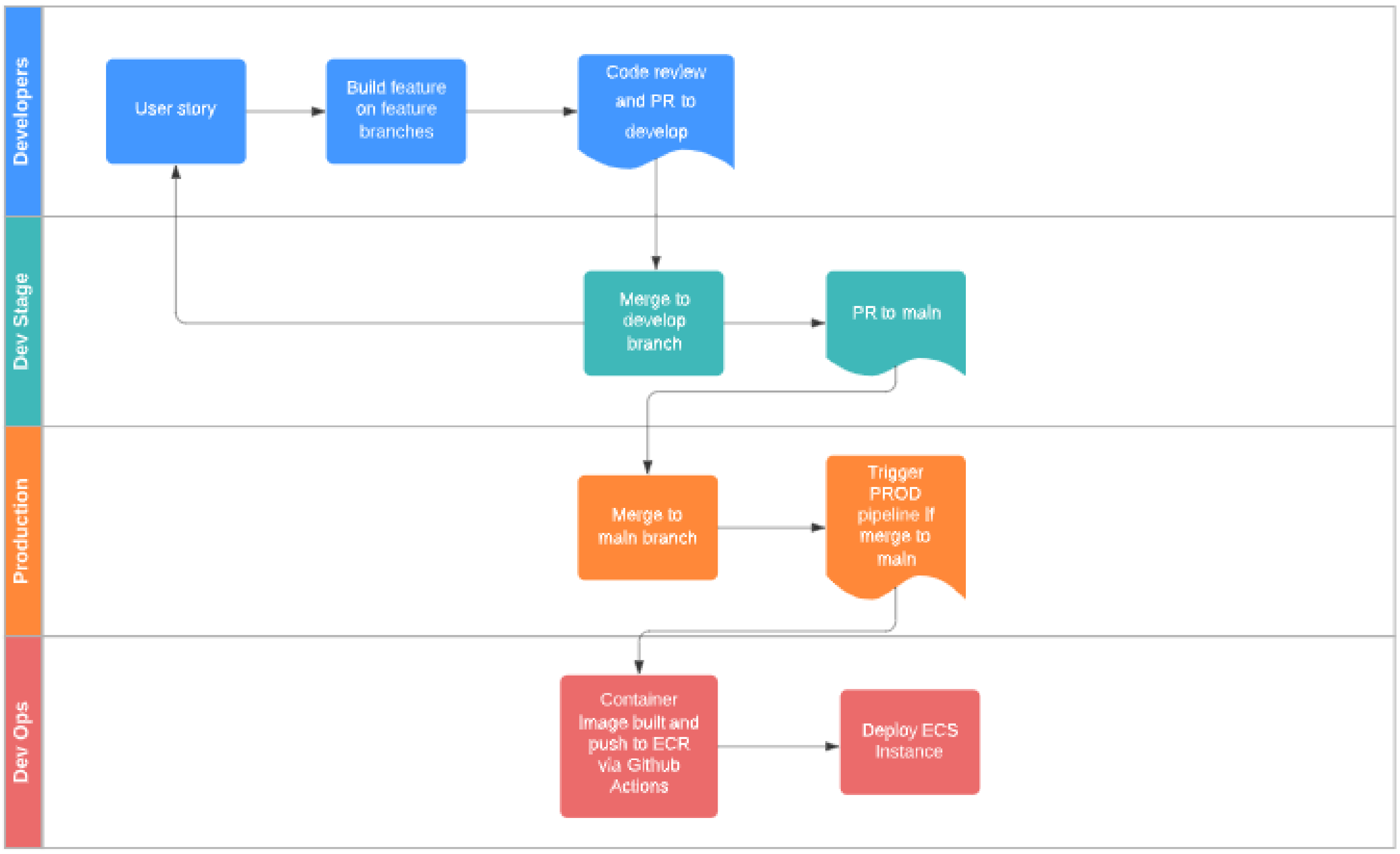
Architectural Decision - VPC with subnets across all 3 AZs	
ID	5
Issue	Deploying applications and services in 1 availability zone is not reliable
Architectural Decision	Include a subnet for each availability zone (ap-southeast-1a, ap-southeast-1b, ap-southeast-1c) within our VPC.
Assumptions	Networking and resources deployed across 3 availability zones in

	ap-southeast-1 are highly reliable
Alternatives	Deploying with other cloud providers
Justification	Too much complexity to handle given the scope of the project

Architectural Decision - Using AWS Aurora RDS database cluster	
ID	6
Issue	As primary data stores, databases should be configured to with performance and redundancy in mind, allowing for easy scaling to meet demands and replication to guard against failure
Architectural Decision	Using AWS Aurora to manage clusters of MySQL database instances for performance and availability
Assumptions	All the data our application processes and uses will be stored in relational databases
Alternatives	Running separate DB instances and configuring read replicas
Justification	As a fully managed database service, AWS RDS is simpler and easier to scale compared to other options, and makes it easier to configure reader-writer pairs and cross-region replicas if necessary.

Architectural Decision - Using AWS Lambda to handle API processing	
ID	7
Issue	Handling 1 million API requests per day
Architectural Decision	AWS Lambda functions are easily deployable and scalable, and can easily integrate with SQS
Assumptions	JSON transaction formats will always be the same
Alternatives	Running a server instance to handle the API requests
Justification	The handling of API processing only requires ingesting the transaction into the message queue hosted by SQS. It is not necessary to deploy a server instance to support this function. Furthermore, the ease of scaling will allow us to handle the varying API request loads throughout the day in a very fine grained manner.

Development View

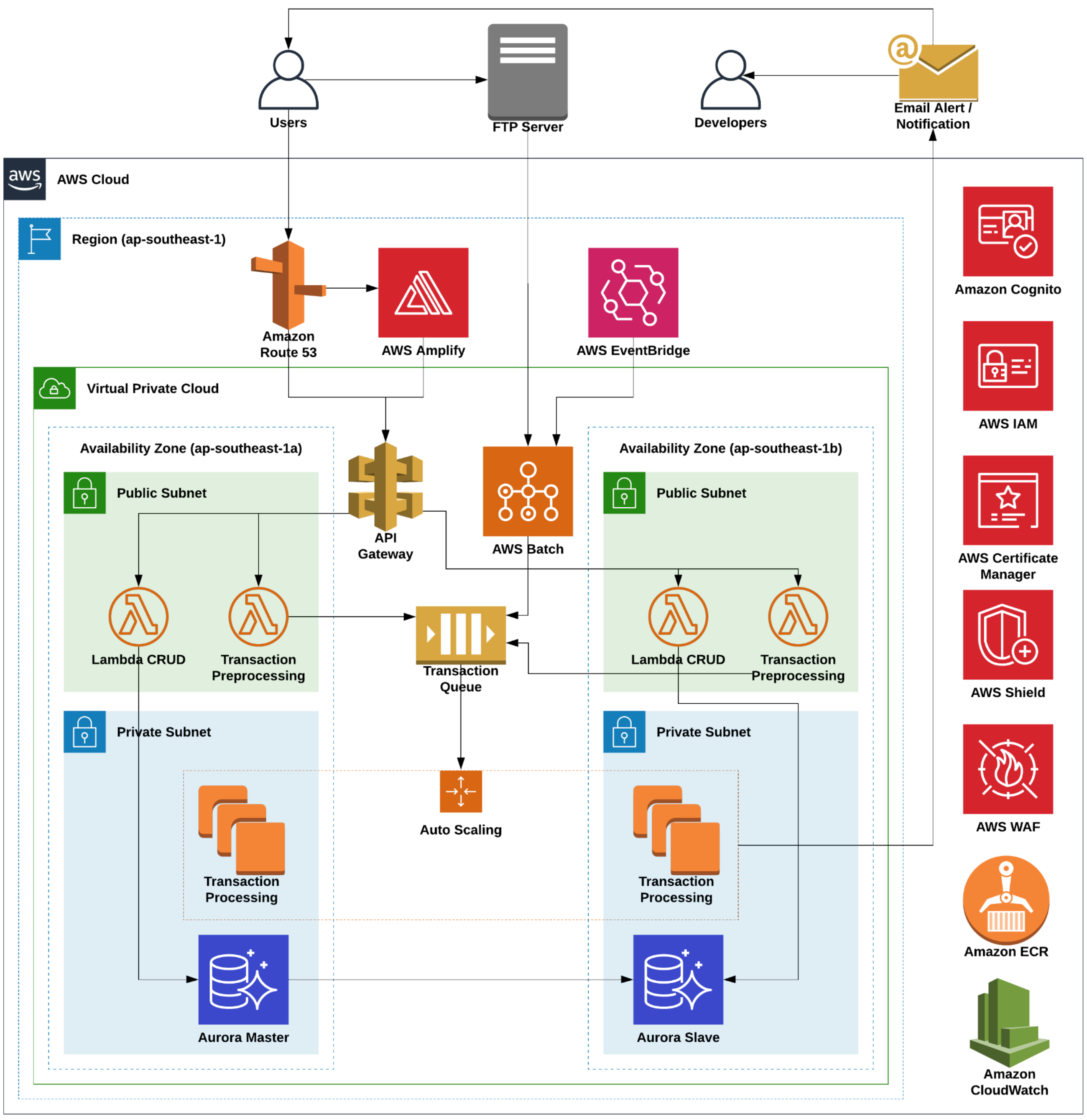

Developers
Dev Stage
Production
Dev Ops
User story
Build feature on feature branches
Code review and PR to develop
Merge to develop branch
PR to main
Merge to main branch
Trigger PROD pipeline if merge to main
Container Image built and push to ECR via Github Actions
Deploy ECS Instance

GitHub Action with WorkFlow

1. Build image using Dockerfile in root directory
2. Tag as latest with ECR Registry, Repository name
3. Push to ECR Repository

Refer to Appendix A for aws.yml and Dockerfile

Solution View (Maintainability)



Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Bank	Exavault FTP Server	SFTP	CSV	Asynchronous
Exavault FTP Server	AWS Batch	SFTP	CSV	Asynchronous
AWS Batch	AWS SQS	HTTPS	JSON	Asynchronous
AWS SQS	AWS ECS	HTTPS	JSON	Asynchronous
AWS ECS	AWS Aurora	HTTPS	Text	Synchronous
Bank/Customer	AWS API Gateway	HTTPS	JSON	Synchronous

Bank/Customer	AWS Cloudfront	HTTPS	HTML	Synchronous

Flexibility to new changes and campaigns

The way that exclusions, programs and campaigns have been implemented in the backend service makes it easy for banks to make modifications and layer new features on top of existing card schemes.

Exclusions apply to specific card types and MCC codes, and if any transaction of that card type involves the relevant MCC, no further rewards will be processed.

Programs are the default reward scheme for card types. Often, a card will have a base earning scheme that applies for all non-excluded transactions, and a special scheme for certain types of spending related to specific MCCs. These rules are stored as separate entries within the database. For each transaction, we first check to see if the MCC implies a special scheme for that card type. Otherwise, we look up the table to find the base scheme (MCC = -1) and apply that instead. This allows for banks to add different clauses for rewards earned upon spending with different merchants if necessary. Programs are mutually exclusive, so each transaction can accumulate rewards for only one program

Campaigns are short-term reward schemes that are implemented in conjunction with specific merchants, and therefore not linked to MCCs, but instead unique merchant names. They have a start date and an end date during which they are valid, and will not apply beyond that time frame. Campaigns are completely independent of programs and can also be stacked on top of one another. For example, if a certain merchant like Grab has a cashback and shopping points campaign for a certain card running at the same time, both of those campaigns will be processed.

*For code maintainability and code flexibility refer to Appendix B

Proposed Budgets

Development Budget

Activity/ Hardware/Software/ Service	Description	Cost (USD)
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 Cost

Activity/ Hardware/Software/ Service	Description	Cost
Compute - Batch	We run workloads likely to exceed the 15 minute lambda timeout (CSV ingest and preprocessing) on EC2	**$8.40** i3en.large 30 hours/month
Compute - ECS	To run transaction processing	**$69.20**

		Base: 1 vCPU, 8GB memory in ap-southeast-1, subject to auto-scaling rules
Compute - Lambda	Serverless compute to preprocess transactions from the API, as well as for GET operations for the frontend Consideration**:** More fine-grained scalability for varied, unpredictable workloads typical of an API backing a webapp, as opposed to instance level scaling using EC2	**$28.37** 10 reserved concurrency, 2000000 requests lasting 200 ms, 256 MB allocated to function
Storage - Aurora	We use the Aurora MySQL 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	**$0.009** 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.	**$0.001** 2 million requests of 512KB
Cloudwatch	A monitoring / observability service to provide developers with more insights on application metrics.	**$0.13**
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	**$0.002** Less than 1GB data transfer

Route 53	DNS Service	**$0.50**
GuardDuty	Threat detection for API	**$0.67**
Amplify	Manage and build frontend repository	**$0.50**
SES	Emailer service	**$0.0008**
Maintenance	Code revisions and changes	**15 man-hours**
Total financial cost	All costs excluding man hours	**$503.11**

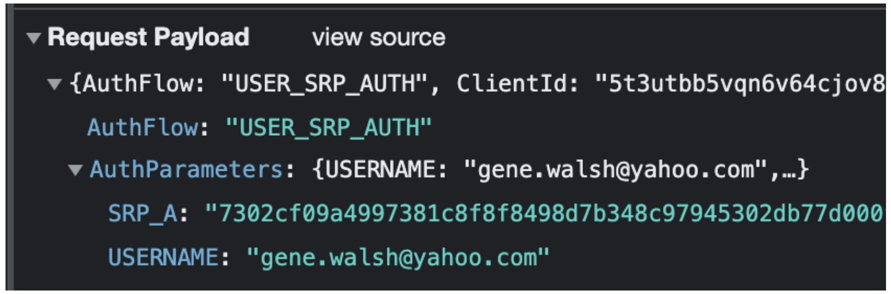
Availability View

Node	**Redundancy**	**Clustering**			**Replication (if applicable)**			
		Node Config.	**Failure Detection**	**Failover**	**Repl. Type**	**Session State Storage**	**DB Repl. Config.**	**Repl. Mode**
Batch	Vertical	Single node	Health check	Job restarted	NA			
Aurora	Horizontal	Active-Active	Ping	Update cluster endpoint	DB	NIL	Master-Slave	Asynchronous
ECS	Horizontal	Active-Active	Health check	ASG replaced/Task restarted	NA			

For Batch, it is difficult to avoid a single point of failure effectively. Splitting the CSV file for processing is not efficient at our scale as outlined under architectural decision 4. While the content based deduplication of SQS protects against repeated transactions, it is still inefficient to have to process the same rows again. To mitigate this in the future, we might keep track of the row of the CSV file reached as a form of session state storage, allowing us to resume processing from the last successful group of rows before the job failed.

Security View

No	Asset/Asset Group	Potential Threat/ Vulnerability	Mitigation Control

1	Data in transit
	AWS Cognito, SSL Certificate
		MITM attack
		a. Hijack user credentials (Confidentiality)
		b. Post invalid transaction details (Integrity)
			[Implemented]
			Enable SSL and HTTPS and ensure that it is used for the entire application
			a. All requests to Amazon Cognito must be made over TLS (reference)
			b. AWS Cognito utilises Secure Remote Password Protocol (SRP_A).
			i. Sends SRP details to the cognito endpoint. Passwords are never revealed during transit or at rest.
			
			Request Payload view source
			`{AuthFlow: "USER_SRP_AUTH", ClientId: "5t3utbb5vqn6v64cjov8`
			`AuthFlow: "USER_SRP_AUTH"`
			`AuthParameters: {USERNAME: "gene.walsh@yahoo.com",…}`
			`SRP_A: "7302cf09a4997381c8f8f8498d7b348c97945302db77d000`
			`USERNAME: "gene.walsh@yahoo.com"`
			c. 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)	[Implemented] a. User input sanitisation i. Validate user email and password before sending authentication request b. AWS Cognito obfuscates all user passwords. Admins are only allowed to set the user's initial login password, which will be changed on their first login. Subsequently, admins can only trigger a reset password request, and users will have to reset their password on their next login c. Use of Object Relational Mapping tools i. ORMs use parameterised SQL statements to query the database d. Implement AWS WAF on API Gateway i. Provides an additional layer of security against common web exploits and bots ii. Configured logging & CloudWatch Database should not be publicly accessible
3	Server	DDoS Attack	[Implemented]

	AWS EC2, Lambda	a. Attackers overloading the server with requests to disrupt service	a. AWS WAF i. Monitors incoming HTTP and HTTPS requests to check for any malicious attempts b. AWS Shield i. Automatically baselines traffic, identifies anomalies, and creates mitigations when necessary.
4	Web portal, user information	User Impersonation a. Attackers may pose as users of the platform and retrieve user sensitive information.	[Implemented] a. AWS Cognito i. Provides user pool management capability, handling authentication, authorisation and user management. b. JWT Token i. A JWT token that complies to the OAuth standard is issued by AWS cognito to ensure that requests are coming from an authorised body.
5	User's credentials in the database	Server Side Request Forgery	[Future Implementation] a. Multi Factor Authentication i. Users would need to login using 2 or more authentication methods.
6	Sensitive information	Leaking of sensitive information by committing to github	[Implemented] a. AWS SSM Parameter Store i. Sensitive information is stored encrypted in Parameter Store, and retrieved using the active AWS credentials or role. ii. This also allows for a single centralized location to update should any of the information like database credentials change.

Performance View

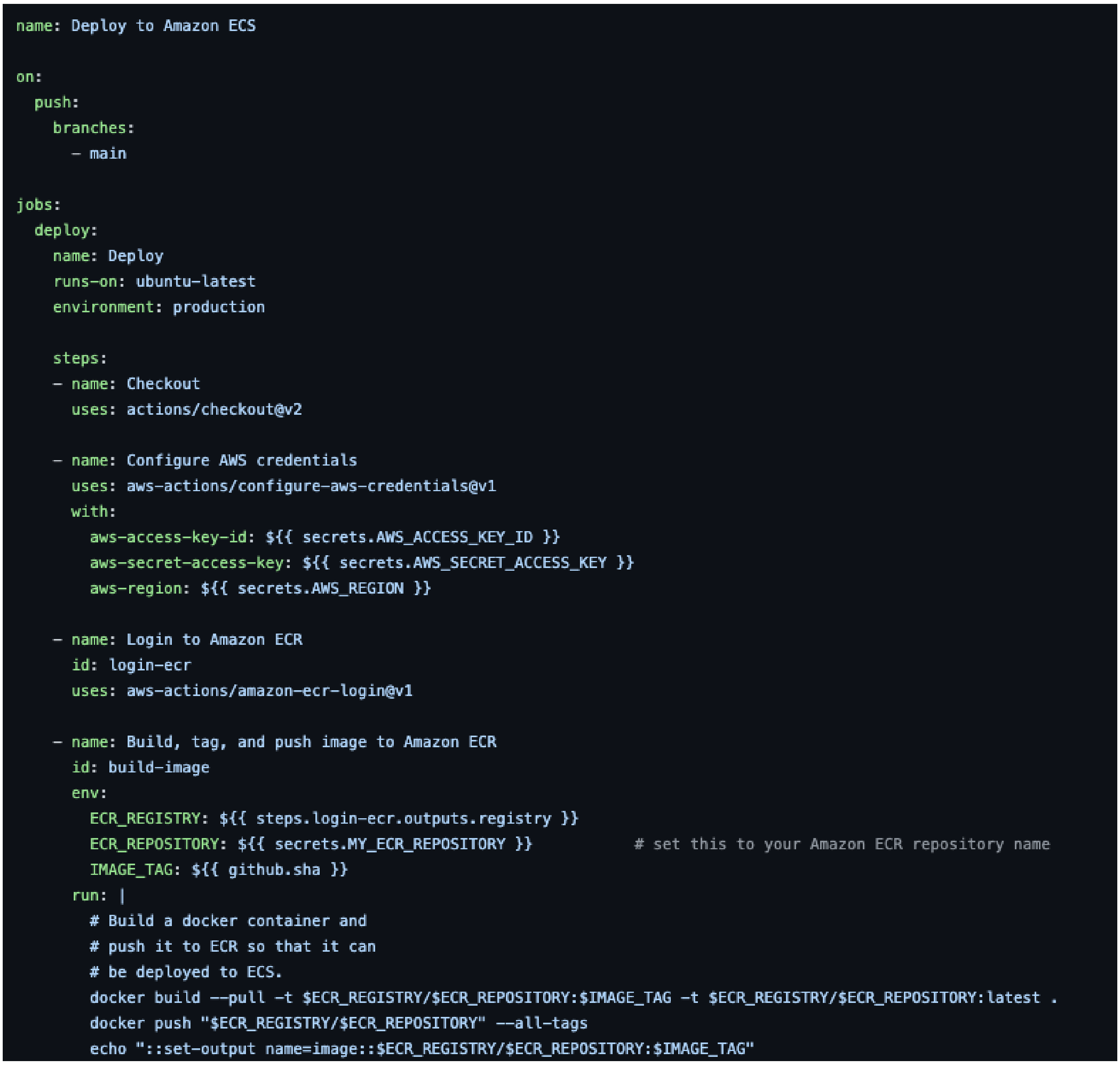
No	Description of the Strategy	Justification	Performance Testing (Optional)
1	[Implemented] Used Batch to pull the transactions CSV files from the bank's FTP server and then process the data row by row before publishing to the SQS service.	This allows the decoupling of the uploading of transactions by CSV and the ingestion of the transactions. Giving us a few advantages: 1. Different components of our service can fail independently, making the service more fault tolerant as a whole. 2. Asynchronous upload and ingestion of transactions improves the performance as the queue allows multiple transactions to be processed concurrently.	**43,000 transactions/ minute** **Refer to [Appendix C]()**
2	[Implemented] Used Parallel and Asynchronous Programming in .NET to process queried transaction JSONs from the SQS	Parallel Programming in .NET is simplified by the Task Parallel Library (TPL), which enabled us to add parallelism and concurrency to our ingestion service. Handling ingestion in parallel and asynchronously (where possible) allows for more efficient use of resource utilization in performing queries and transaction ingestion.	**14,000 transactions/ minute** **Refer to [Appendix C]()**
3	[Implemented] Denormalize tables that are READ intensive. EG rules [Future Implementation] Vertically Partition transactions and points table by date	Decided to denormalize tables campaigns and programs into a single "rules" table to improve read performance. These tables are not write intensive hence write performance will not be affected. Since approximated transactions to be processed per day is around 1 million, the logical partitioning is to do interval partitioning by date for transactions and points.	**-**
4	[Implemented] Configure target-based auto-scaling rules for ECS Fargate Cluster running backend service	The backend service processes transactions from a Simple Queue Service (SQS). However, transactions are not transmitted at an even pace - when daily batch jobs to process CSV files are run, the volume of transactions is much higher. As such, the backend service needs to be scaled up to accommodate periods of high traffic, and scale down in other periods to save cost.	**Refer to [Appendix D]()**
5	[Implemented] Enabled Multi-AZ Amazon RDS Read Replica to offload read workload from the write instance	Amazon RDS Read Replicas allows for offload read requests traffic from the write instance, allowing the write instance to maintain its writes per second. This allows for a high availability as we utilize AWS' automatic failover functionality to promote a replica to the primary instance in the event of downtime on the previous primary instance.	
6	[Implemented] Cache transaction	Retrieving transaction data from the endpoint is the most time-consuming task on the frontend. Frequently	**Refer to**

	data on the frontend, using the total points from card to verify if the transaction data in cache is stale	accessing the transaction data will increase our backend's workload as well. The security risk of caching this data is low as there was no sensitive information in the data. and there were no endpoints available for users to possibly tamper with the data.	Appendix E
7	[Partially Implemented] Reduce bundle size for production build	All assets were optimized and tree shaking was used to eliminate unused code imports. The bundle size was reduced by 100kb	**Refer to Appendix F**
8	[Implemented] Search & Pagination Optimization	Implement debounce and throttle to prevent the user from invoking the search and pagination functions too frequently.	**-**

Appendix

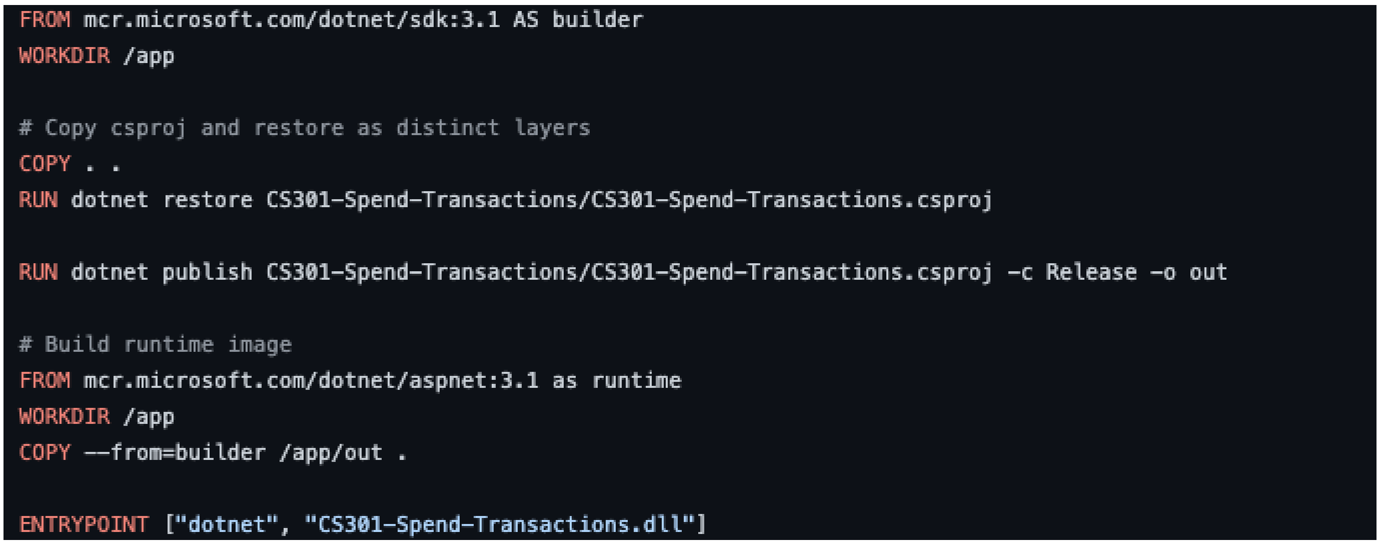
Appendix A

aws.yml



```yaml
name: Deploy to Amazon ECS

on:
  push:
    branches:
      - main

jobs:
  deploy:
    name: Deploy
    runs-on: ubuntu-latest
    environment: production

    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: ${{ secrets.AWS_REGION }}

    - name: Login to Amazon ECR
      id: login-ecr
      uses: aws-actions/amazon-ecr-login@v1

    - name: Build, tag, and push image to Amazon ECR
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        ECR_REPOSITORY: ${{ secrets.MY_ECR_REPOSITORY }}           # set this to your Amazon ECR repository name
        IMAGE_TAG: ${{ github.sha }}
      run: |
        # Build a docker container and
        # push it to ECR so that it can
        # be deployed to ECS.
        docker build --pull -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG -t $ECR_REGISTRY/$ECR_REPOSITORY:latest .
        docker push "$ECR_REGISTRY/$ECR_REPOSITORY" --all-tags
        echo "::set-output name=image::$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG"
```

Dockerfile



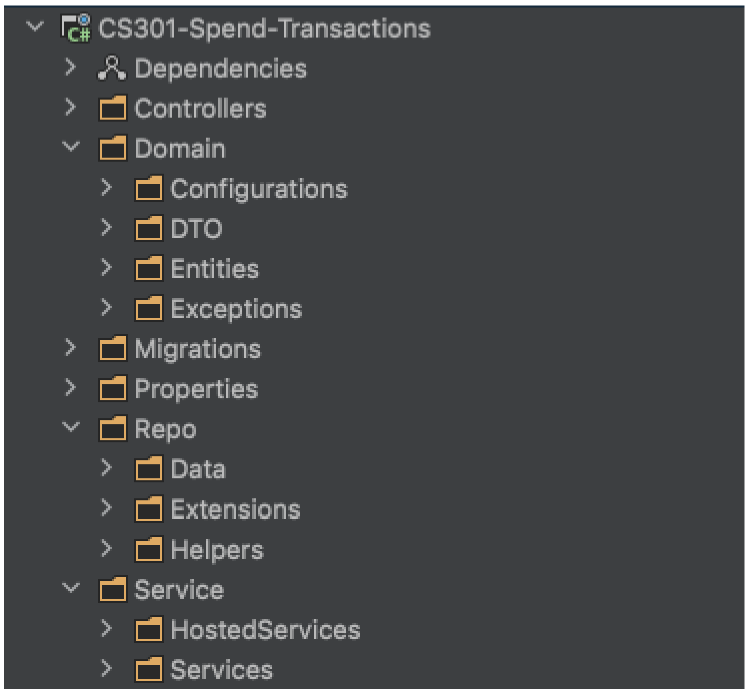
```
FROM mcr.microsoft.com/dotnet/sdk:3.1 AS builder
WORKDIR /app

# Copy csproj and restore as distinct layers
COPY . .
RUN dotnet restore CS301-Spend-Transactions/CS301-Spend-Transactions.csproj

RUN dotnet publish CS301-Spend-Transactions/CS301-Spend-Transactions.csproj -c Release -o out

# Build runtime image
FROM mcr.microsoft.com/dotnet/aspnet:3.1 as runtime
WORKDIR /app
COPY --from=builder /app/out .

ENTRYPOINT ["dotnet", "CS301-Spend-Transactions.dll"]
```

Appendix B

Code

We developed our application according to object-oriented design principles with C# on the .NET framework in order to promote modular and reusable code. Onion architecture is used to reduce dependability and coupling between components of the code.

3 main components are: Domain, Repository and Service, listed from innermost to the outermost layer of the architecture, where outer layer is built upon inner layer. Only the outer layers are dependent on the inner layers as the coupling is towards the centre.



```
CS301-Spend-Transactions
    Dependencies
    Controllers
    Domain
        Configurations
        DTO
        Entities
        Exceptions
    Migrations
    Properties
    Repo
        Data
        Extensions
        Helpers
    Service
        HostedServices
        Services
```

Adding new features on Onion Architecture is relatively straightforward as each component has very specific functions. General Rules are as below:

To add new features on CS301-backend:

1. Domain
    a. Add Entities for objects that will be persisted into Database
    b. Add DTO for objects that will be converted from services like SQS / Controllers (eg JSON objects)
    c. Add Configurations to store option variables required by Helper classes
2. Repo
    a. Data/AppDbContext
        i. Add DbSet for Entities to enable query and insert
        ii. Add FluentAPI to define database constraints (foreign key, primary key and column names etc)
    b. Helper
        i. Add interfaces and classes to handle logic related to external services (eg SES, SQS)
    c. Extensions/OptionVariablesStartup
        i. Add method to read options from environment variables
    d. Extensions/DataAccessStartup
        i. Define lifetime scope for Services/Helper (Singleton/Scoped/Transient)

3. Service
    a. Add interfaces and classes to handle business logic related to feature
4. HostedServices are Services that is meant to run in the background

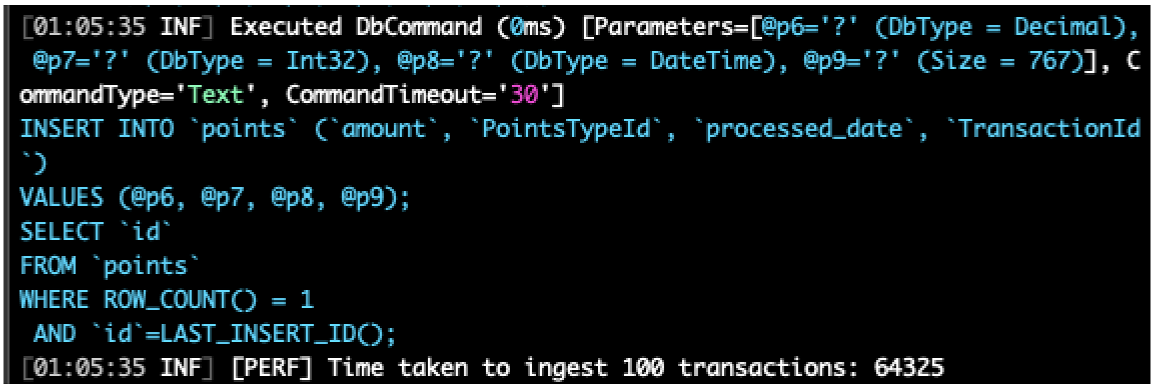
Appendix C

Local benchmarks on M1 Macbook Air, 16GB

Average time taken to ingest 100 transactions from SQS to database:	1. Without parallelization: ~**64000ms** 2. With parallelization : ~**4600ms**
Average number of transactions ingested in 60 seconds:	1. Without parallelization: ~**93.75 transactions** 2. With parallelization : ~**1300 transactions**

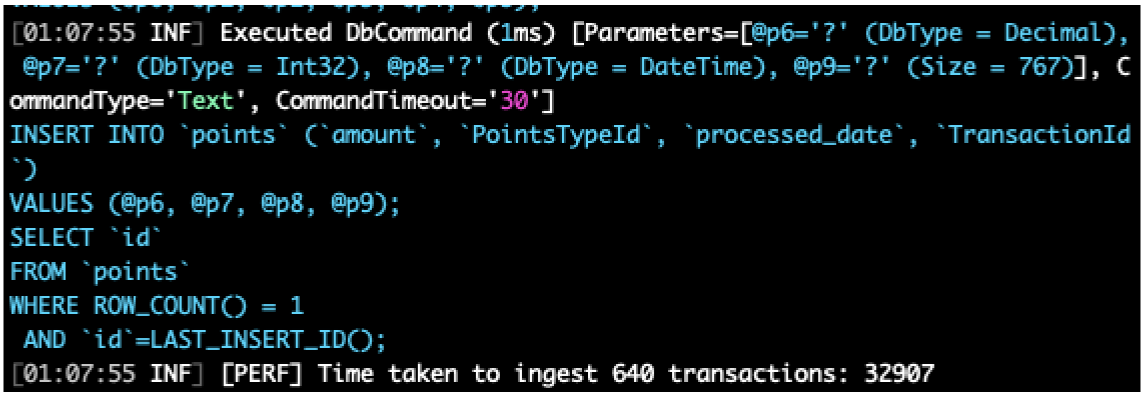
Terminal output examples

- Without parallelization



```
[01:05:35 INF] Executed DbCommand (0ms) [Parameters=[@p6='?' (DbType = Decimal),
 @p7='?' (DbType = Int32), @p8='?' (DbType = DateTime), @p9='?' (Size = 767)], C
ommandType='Text', CommandTimeout='30']
INSERT INTO `points` (`amount`, `PointsTypeId`, `processed_date`, `TransactionId
`)
VALUES (@p6, @p7, @p8, @p9);
SELECT `id`
FROM `points`
WHERE ROW_COUNT() = 1
 AND `id`=LAST_INSERT_ID();
[01:05:35 INF] [PERF] Time taken to ingest 100 transactions: 64325
```

- With parallelization

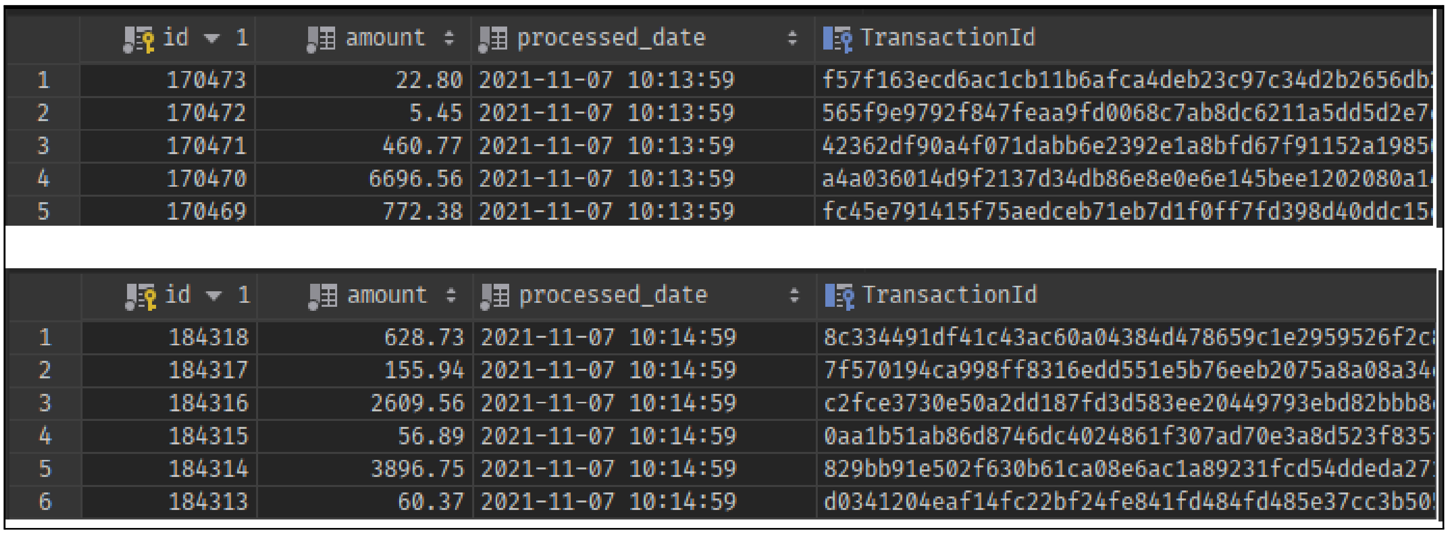


```
[01:07:55 INF] Executed DbCommand (1ms) [Parameters=[@p6='?' (DbType = Decimal),
 @p7='?' (DbType = Int32), @p8='?' (DbType = DateTime), @p9='?' (Size = 767)], C
ommandType='Text', CommandTimeout='30']
INSERT INTO `points` (`amount`, `PointsTypeId`, `processed_date`, `TransactionId
`)
VALUES (@p6, @p7, @p8, @p9);
SELECT `id`
FROM `points`
WHERE ROW_COUNT() = 1
 AND `id`=LAST_INSERT_ID();
[01:07:55 INF] [PERF] Time taken to ingest 640 transactions: 32907
```

Benchmarks on ECS Fargate Cluster

When processing transactions on an ECS Fargate cluster, we were able to achieve approximately 14,000 write operations per minute, based on the ID numbers of points for each transaction processed by the application.

The example screenshot shows that Transactions with integer Id from 170, 473 to 184, 318 are ingested within 60 seconds (2021-11-07 10:13:59 to 2021-11-07 10:14:59), totalling up to 13, 845 transactions ingested in 60 seconds.

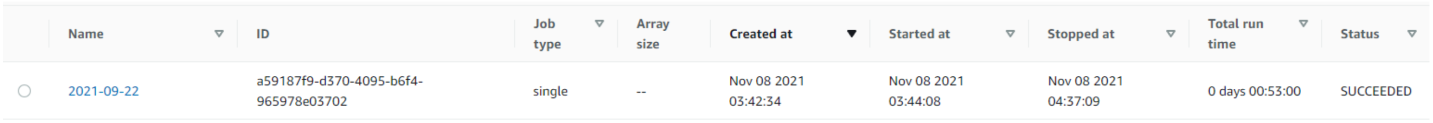


	id 1	amount	processed_date	TransactionId
1	170473	22.80	2021-11-07 10:13:59	f57f163ecd6ac1cb11b6afca4deb23c97c34d2b2656db
2	170472	5.45	2021-11-07 10:13:59	565f9e9792f847feaa9fd0068c7ab8dc6211a5dd5d2e7
3	170471	460.77	2021-11-07 10:13:59	42362df90a4f071dabb6e2392e1a8bfd67f91152a1985
4	170470	6696.56	2021-11-07 10:13:59	a4a036014d9f2137d34db86e8e0e6e145bee1202080a1
5	170469	772.38	2021-11-07 10:13:59	fc45e791415f75aedceb71eb7d1f0ff7fd398d40ddc15

	id 1	amount	processed_date	TransactionId
1	184318	628.73	2021-11-07 10:14:59	8c334491df41c43ac60a04384d478659c1e2959526f2c
2	184317	155.94	2021-11-07 10:14:59	7f570194ca998ff8316edd551e5b76eeb2075a8a08a34
3	184316	2609.56	2021-11-07 10:14:59	c2fce3730e50a2dd187fd3d583ee20449793ebd82bbb8
4	184315	56.89	2021-11-07 10:14:59	0aa1b51ab86d8746dc4024861f307ad70e3a8d523f835
5	184314	3896.75	2021-11-07 10:14:59	829bb91e502f630b61ca08e6ac1a89231fcd54ddeda27
6	184313	60.37	2021-11-07 10:14:59	d0341204eaf14fc22bf24fe841fd484fd485e37cc3b50

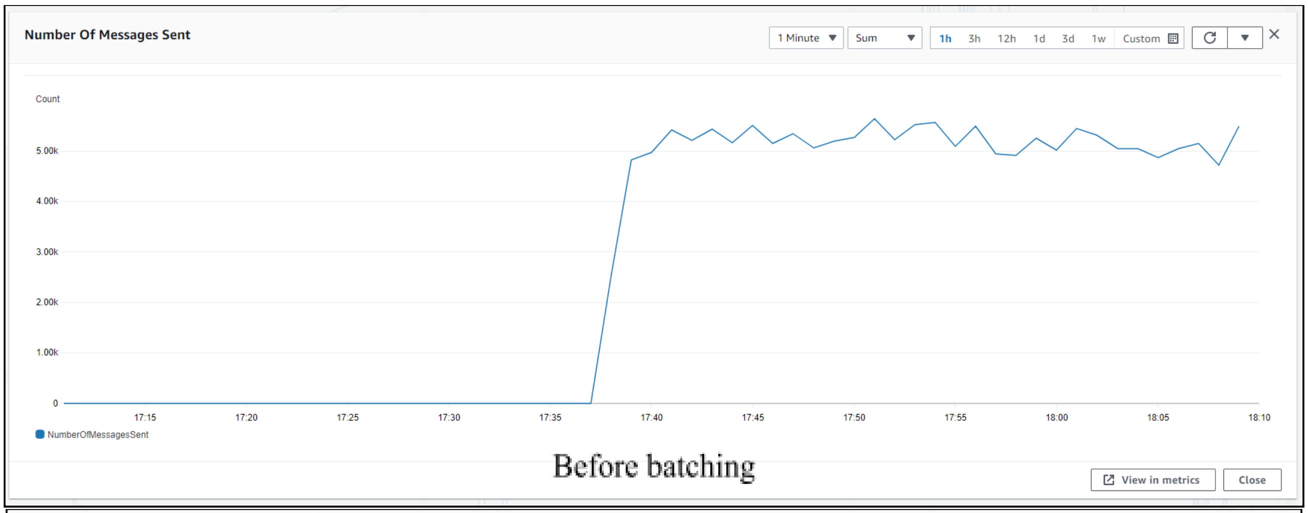
<u>Benchmarks on Batch</u>

When processing the transactions CSV file, we were able to process and enqueue over 40,000 rows per minute, with an average of around 43,000 rows per minute. This resulted in us being able to process a CSV file with over 2 million records (2021-09-22) in less than an hour, far outstripping the functional requirements, at a cost effective less than $0.20 per run. This was achieved by identifying the bottlenecks of the process to be disk and network performance, and choosing an appropriate instance type (i3en.large) to optimize for this. We also batched uploads to SQS to minimize the number of API calls required to reduce overhead.

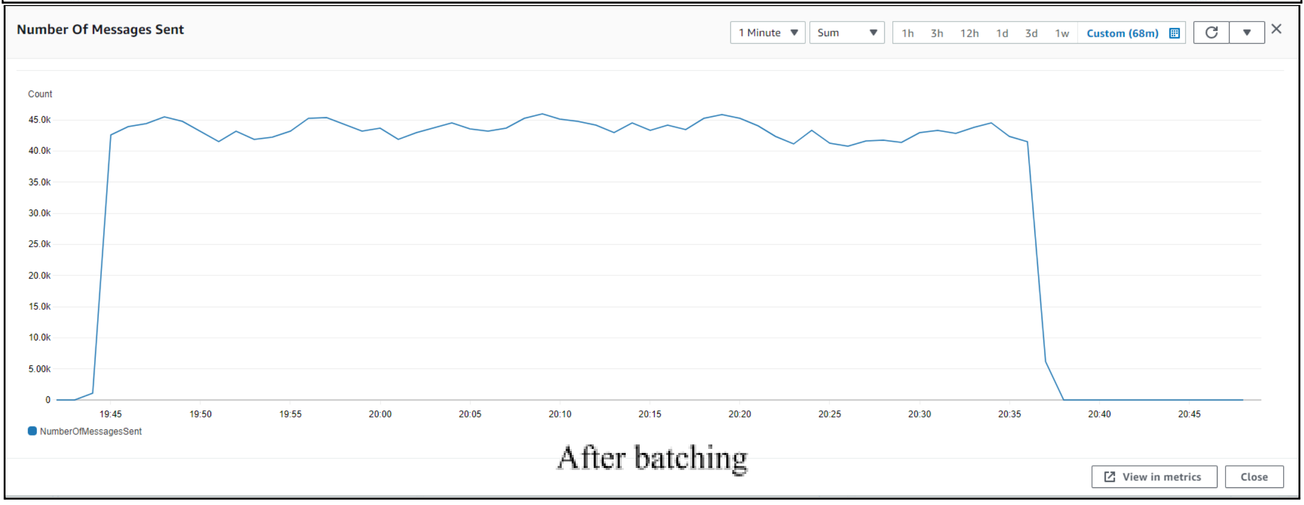


Name	ID	Job type	Array size	Created at	Started at	Stopped at	Total run time	Status
2021-09-22	a59187f9-d370-4095-b6f4-965978e03702	single	--	Nov 08 2021 03:42:34	Nov 08 2021 03:44:08	Nov 08 2021 04:37:09	0 days 00:53:00	SUCCEEDED

Successful ingest batch job in 53 minutes



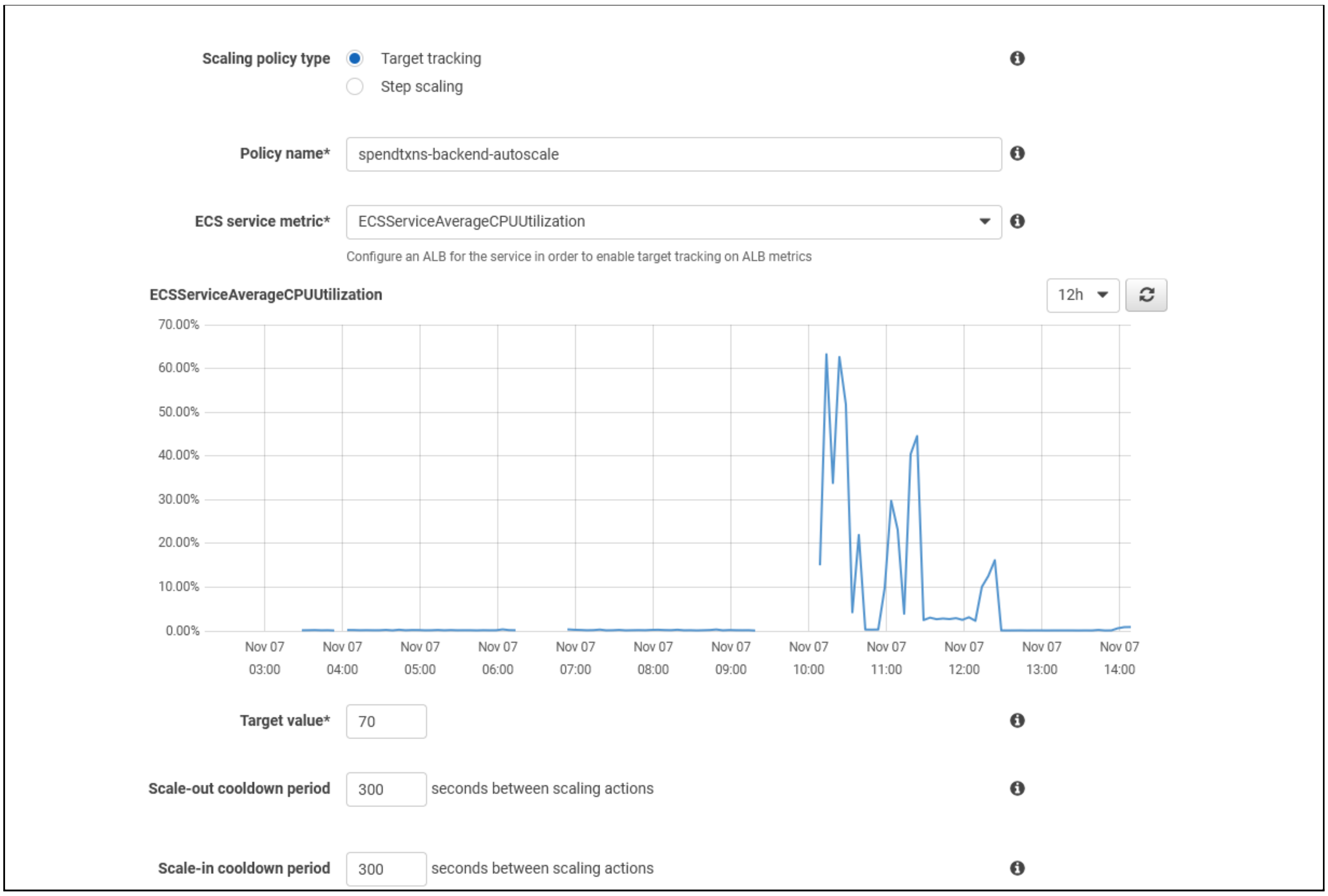
Before batching



After batching

Appendix D

Auto-scaling rules



Scaling policy type: [✓ Target tracking] Step scaling

Policy name*: spendtxns-backend-autoscale

ECS service metric*: ECSServiceAverageCPUUtilization

Configure an ALB for the service in order to enable target tracking on ALB metrics

Target value*: 70

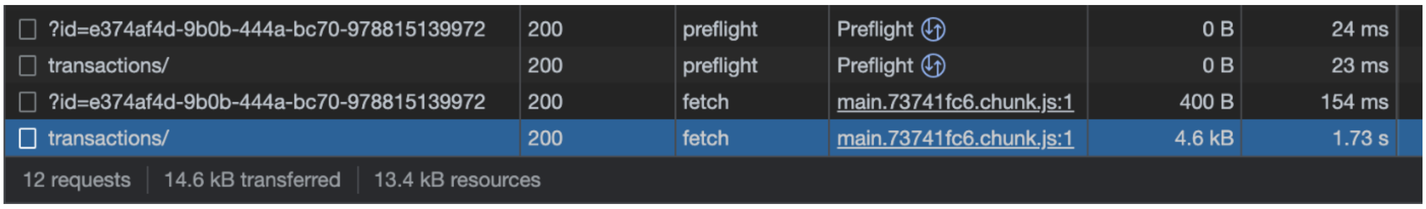
Scale-out cooldown period: 300 seconds between scaling actions

Scale-in cooldown period: 300 seconds between scaling actions

Auto-scaling configured for ECS Fargate Cluster, using a target tracking policy based on CPU utilization (>70%)

Appendix E

Cache Transaction Data



?id=e374af4d-9b0b-444a-bc70-978815139972	200	preflight	Preflight	0 B	24 ms
transactions/	200	preflight	Preflight	0 B	23 ms
?id=e374af4d-9b0b-444a-bc70-978815139972	200	fetch	main.73741fc6.chunk.js:1	400 B	154 ms
transactions/	200	fetch	main.73741fc6.chunk.js:1	4.6 kB	1.73 s

12 requests | 14.6 kB transferred | 13.4 kB resources

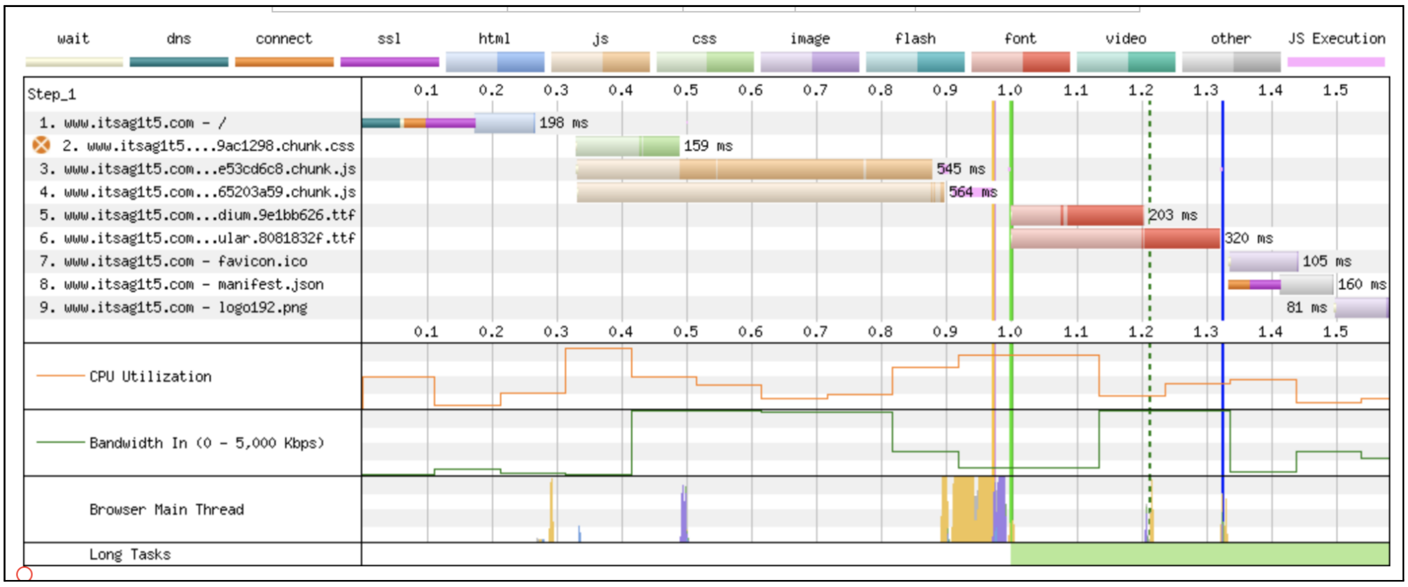
Fetch transaction data from the transaction endpoint takes about 2 seconds while retrieving 100kB - 1MB of data from cache takes <1ms (source). Since our data is 4.6kB, the retrieval time from cache is negligible.

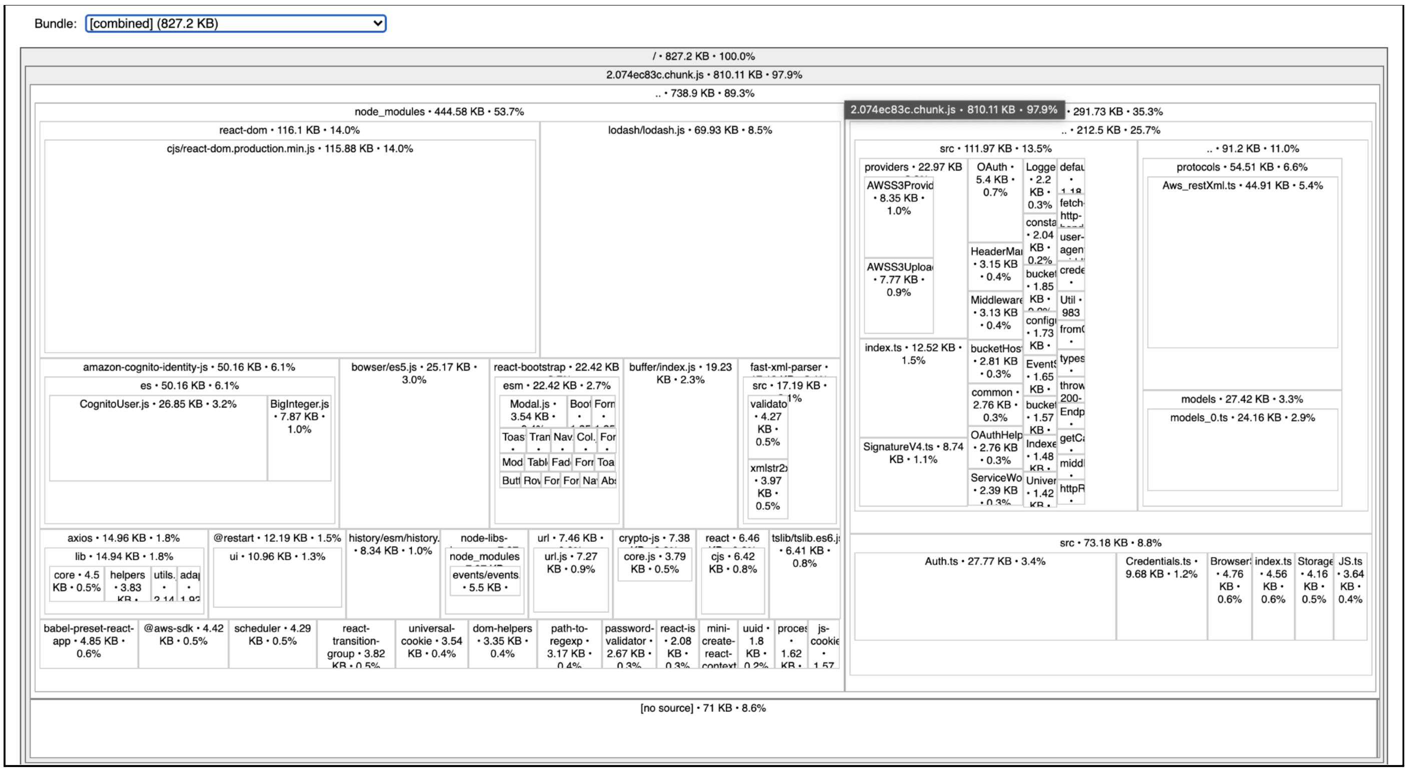
Appendix F

Reduce bundle size for production build

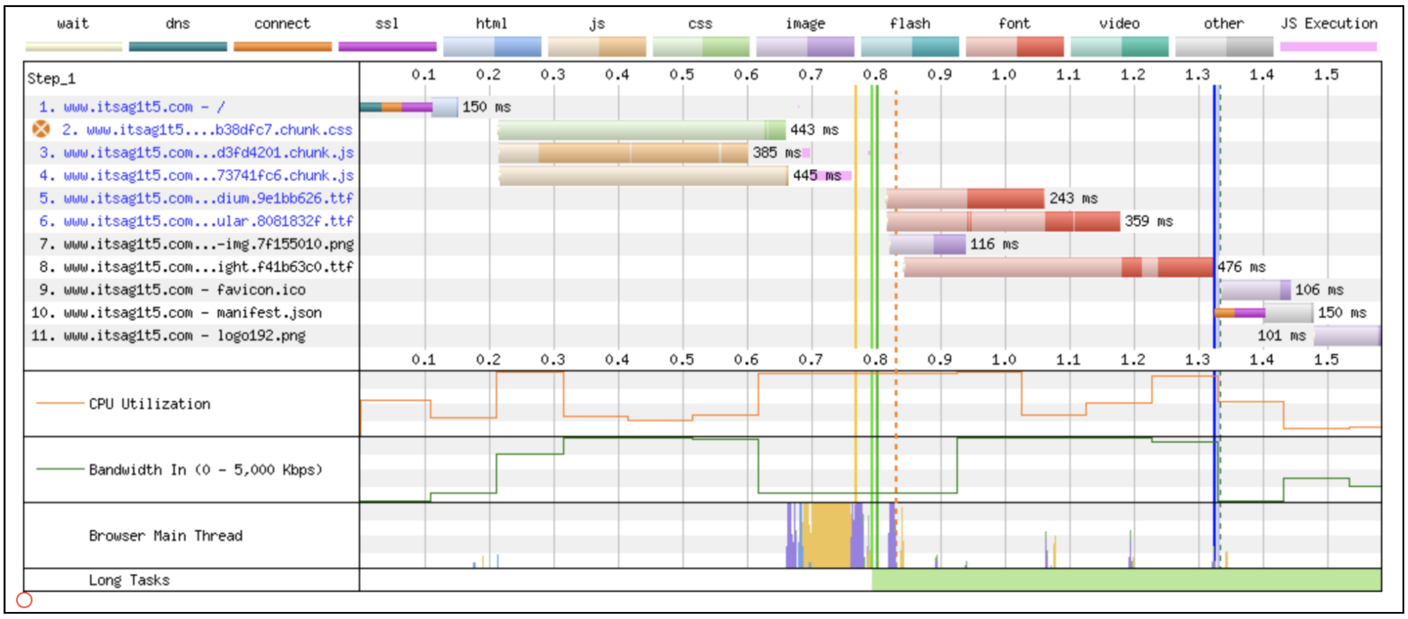
Note: some css code was added after running the first test. There are also a lot more optimization techniques to be implemented (e.g. code splitting), which can be investigated beyond the timeline of this project.

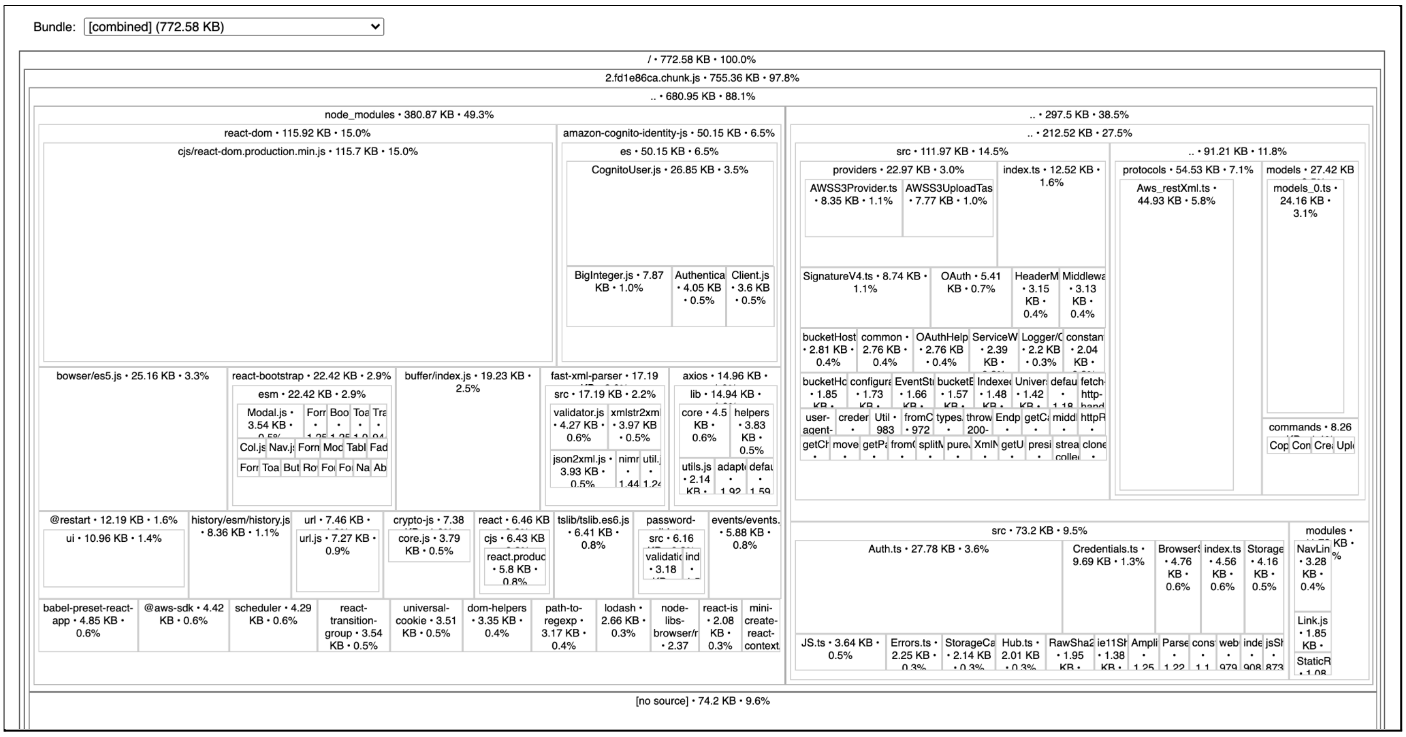
Before Optimization (Bundle size 827.2KB)





After Optimization (Bundle Size 772.58kb)





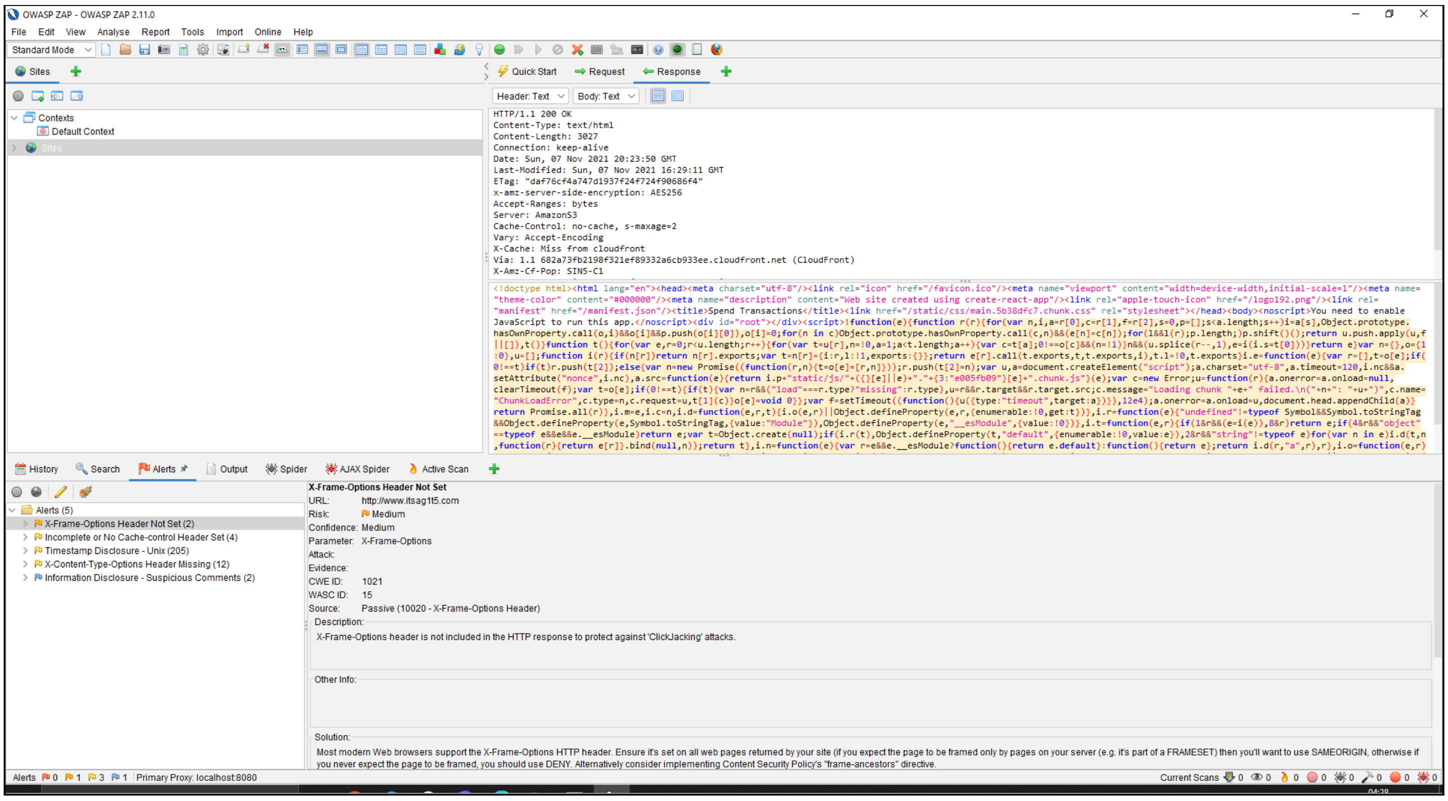
Appendix G

Penetration Testing with OWASP ZAP

We performed penetration testing on our deployed webapp hosted on Amplify to identify any remaining vulnerabilities. Here are the vulnerabilities identified, in descending order of severity:

Vulnerability	**Risk**	**Mitigation**
X-Frame-Options Header Not Set	Our content might be embedded into other websites and used in a clickjacking attack	Set X-Frame-Options to DENY, since we do not expect any of our content to be embedded.
X-Content-Type-Options Header missing	Browsers might perform MIME sniffing on content, leading to potential execution of malicious code despite a non-executable content type like text/plain	Set X-Content-Type-Options to nosniff
Incomplete or No Cache-control Header Set	Browsers are allowed to cache content without restriction	NA, we are performing custom cache invalidation on the frontend
Timestamp Disclosure	Attackers might be able to use timestamps to gain access to information like password hashes or random seeds	NA, most of the detected timestamps were random 10 digit integers. We are also not using time as a parameter for any sensitive operations.
Information Disclosure - Suspicious Comments	Information in comments might help an attacker	NA, detected comments were a link to the Axios github issues page, and a variable name

Sample OWASP ZAP output



```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 3027
Connection: keep-alive
Date: Sun, 07 Nov 2021 20:23:50 GMT
Last-Modified: Sun, 07 Nov 2021 16:29:11 GMT
ETag: "daf76cf4a747d1937f24f724f90686f4"
x-amz-server-side-encryption: AES256
Accept-Ranges: bytes
Server: AmazonS3
Cache-Control: no-cache, s-maxage=2
Vary: Accept-Encoding
X-Cache: Miss from cloudfront
Via: 1.1 682a73fb2198f321ef89332a6cb933ee.cloudfront.net (CloudFront)
X-Amz-Cf-Pop: SIN5-C1
```

```
<!doctype html><html lang="en"><head><meta charset="utf-8"/><link rel="icon" href="/favicon.ico"/><meta name="viewport" content="width=device-width,initial-scale=1"/><meta name=
"theme-color" content="#000000"/><meta name="description" content="Web site created using create-react-app"/><link rel="apple-touch-icon" href="/logo192.png"/><link rel=
"manifest" href="/manifest.json"/><title>Spend Transactions</title><link href="/static/css/main.5b38dfc7.chunk.css" rel="stylesheet"></head><body><noscript>You need to enable
JavaScript to run this app.</noscript><div id="root"></div><script>!function(e){function r(r){for(var n,i,a=r[0],c=r[1],f=r[2],s=0,p=[];s<a.length;s++)i=a[s],Object.prototype.
hasOwnProperty.call(o,i)&&o[i]&&p.push(o[i][0]),o[i]=0;for(n in c)Object.prototype.hasOwnProperty.call(c,n)&&(e[n]=c[n]);for(l&&l(r);p.length;)p.shift()();return u.push.apply(u,f
||[]),t()}function t(){for(var e,r=0;r<u.length;r++){for(var t=u[r],n=!0,a=1;a<t.length;a++){var c=t[a];0!==o[c]&&(n=!1)}n&&(u.splice(r--,1),e=i(i.s=t[0]))}return e}var n={},o={1
:0},u=[];function i(r){if(n[r])return n[r].exports;var t=n[r]={i:r,l:!1,exports:{}};return e[r].call(t.exports,t,t.exports,i),t.l=!0,t.exports}i.e=function(e){var r=[],t=o[e];if(
0!==t)if(t)r.push(t[2]);else{var n=new Promise((function(r,n){t=o[e]=[r,n]}));r.push(t[2]=n);var u,a=document.createElement("script");a.charset="utf-8",a.timeout=120,i.nc&&a.
setAttribute("nonce",i.nc),a.src=function(e){return i.p+"static/js/"+({}[e]||e)+"."+{3:"e005fb09"}[e]+".chunk.js"}(e);var c=new Error;u=function(r){a.onerror=a.onload=null,
clearTimeout(f);var t=o[e];if(0!==t){if(t){var n=r&&("load"===r.type?"missing":r.type),u=r&&r.target&&r.target.src;c.message="Loading chunk "+e+" failed.\n("+n+": "+u+")",c.name=
"ChunkLoadError",c.type=n,c.request=u,t[1](c)}o[e]=void 0}};var f=setTimeout((function(){u({type:"timeout",target:a})}),12e4);a.onerror=a.onload=u,document.head.appendChild(a)}
return Promise.all(r)},i.m=e,i.c=n,i.d=function(e,r,t){i.o(e,r)||Object.defineProperty(e,r,{enumerable:!0,get:t})},i.r=function(e){"undefined"!=typeof Symbol&&Symbol.toStringTag
&&Object.defineProperty(e,Symbol.toStringTag,{value:"Module"}),Object.defineProperty(e,"__esModule",{value:!0})},i.t=function(e,r){if(1&r&&(e=i(e)),8&r)return e;if(4&r&&"object"
==typeof e&&e&&e.__esModule)return e;var t=Object.create(null);if(i.r(t),Object.defineProperty(t,"default",{enumerable:!0,value:e}),2&r&&"string"!=typeof e)for(var n in e)i.d(t,n
,function(r){return e[r]}.bind(null,n));return t},i.n=function(e){var r=e&&e.__esModule?function(){return e.default}:function(){return e};return i.d(r,"a",r),r},i.o=function(e,r)
```

Alerts (5)
- X-Frame-Options Header Not Set (2)
- Incomplete or No Cache-control Header Set (4)
- Timestamp Disclosure - Unix (205)
- X-Content-Type-Options Header Missing (12)
- Information Disclosure - Suspicious Comments (2)

X-Frame-Options Header Not Set

URL:	http://www.itsag1t5.com
Risk:	Medium
Confidence:	Medium
Parameter:	X-Frame-Options
Attack:	
Evidence:	
CWE ID:	1021
WASC ID:	15
Source:	Passive (10020 - X-Frame-Options Header)

Description:

X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.

Other Info:

Solution:

Most modern Web browsers support the X-Frame-Options HTTP header. Ensure it's set on all web pages returned by your site (if you expect the page to be framed only by pages on your server (e.g. it's part of a FRAMESET) then you'll want to use SAMEORIGIN, otherwise if you never expect the page to be framed, you should use DENY. Alternatively consider implementing Content Security Policy's "frame-ancestors" directive.