Nandkumar Kadivar (B00929627) CSCI 5409 Cloud Computing

Term Assignment

Contents

Nandkumar Kadivar (B00929627) CSCI 5409 Cloud Computing 1

Gitlab 2

Project Introduction 2

Features 2

Services Selected 2

Compute 2

AWS EC2 2

AWS Lambda 2

Storage 3

AWS DynamoDB 3

Network 3

AWS API Gateway 3

General 3

AWS EventBridge 3

AWS SNS 3

Deployment Model 3

Delivery Model 3

Architecture 4

cloud mechanisms 4

Where is data stored? 5

What programming languages did you use (and why) and what parts of your application required code? 5

How is your system deployed to the cloud? 5

If your system does not match an architecture taught in the course, you will describe why that is, and whether your choices are wise or potentially flawed. 6

Data Security 6

Which security mechanisms are used to achieve the data security described in the previous question? List them, and explain any choices you made for each mechanism (technology you used, algorithm, cloud provider service, etc.) 6

Costs 6

What would your organization have to purchase to reproduce your architecture in a private cloud while providing relatively the same level of availability as your cloud implementation? Try to give a rough estimate of what it would cost, don't worry if you are far off. These systems are complicated and you don't know all the exact equipment and software you would need to purchase. Just explore and try your best to figure out the combination of software and hardware you would need to buy to reproduce your app on-premise. 6

AWS Resources 8

Application 13

References 18

Gitlab

https://git.cs.dal.ca/courses/2023-summer/csci4145-5409/kadivar/-/tree/main/termassignment

Project Introduction

Expense Manager is a web application that runs on a browser and provides a handy solution for individuals to manage their daily transactions. The web application can record the daily transactions of the users, every individual transaction is mainly classified into two types (income or expense). The use of the web application can create category custom category for every transaction to better manage income or expense. The user can delete or update the category that they have created. While recording their transactions user can choose a specific pre-created category and provide a head to that transaction. The application allows users to create alerts. An alert can have multiple categories and user can specify their monthly budget for that item specifically. Users are allowed to create, update and delete multiple monthly alerts and can prevent overspending money. Moreover, the dashboard of Expense managers illustrates their spending habits in different forms of visualization. In addition, the analytical dashboard provides expense & income insights, graph filters, recent transactions, and financial statistics.

The overall objective of the Expense manager is to provide a minimalist platform to keep track of daily finance and budget planning. The web application can target any potential person who wants to organize their finance, especially students.

Features

- Analytical Dashboard
- Transaction management
- Categories
- Alerts monitor
- Reminders

The application status is completed and ready to be used by my target audience as it is deployed using AWS cloud and it is a Software as a Service delivery model.

Services Selected

Compute

AWS EC2

The web application ReactJS fronted is deployed on an AWS EC2 Linux instance. As EC2 provides full configuration to and OS level access which was required for the application to run frontend and user the API gateway endpoint as a variable stored inside the machine. Moreover, the service is cost-effective as the application moves to production, ec2 allows a wide range of options such as reserved instances spot instances, and on-demand instances so that based on the user's proper selection of instance type can save a lot of money.

AWS Lambda

The NodeJS backend is deployed using AWS Serverless Lambda. It uses the FAAS delivery model

and every function performs a specific task such as creating the transaction, updating the transaction, getting the transaction, etc. The reason for choosing lambda functions for backend logic is as it provides Function as a service the billing is based on API triggers/function execution. Which can be cost-effective and can handle unpredicted fluctuations of traffic.

Storage

AWS DynamoDB

NoSQL database AWS DynamoDB is used to keep users, transactions, categories, and alert data as an object. The application stores data from alerts which allows users to add as many categories as they want. Therefore, instead of going with an SQL database, it seems preferable to go with a schema-less database. This decision eliminates a majority of databases provided by AWS such as RDS, Athena, and AppSync. Moreover, DynamoDB is also a fully managed database so no need to do patches and updates.

Network

AWS API Gateway

The API Gateway acts as an entry point to the backend module, every API route has been created and integrated with a specific lambda function to route the request from the front.

General

AWS EventBridge

To create an event-driven service the application uses AWS EventBridge service. The service is used to trigger the SNS and publish the reminder message to the topic every day at 9 PM. The purpose of this scheduled event is to remind users to record their today's transactions so they don't forget.

AWS SNS

The SNS topic is used to send emails for reminders, every user is subscribed to this SNS topic and every time an event bridge publishes a message into the topic it sends an email to users.

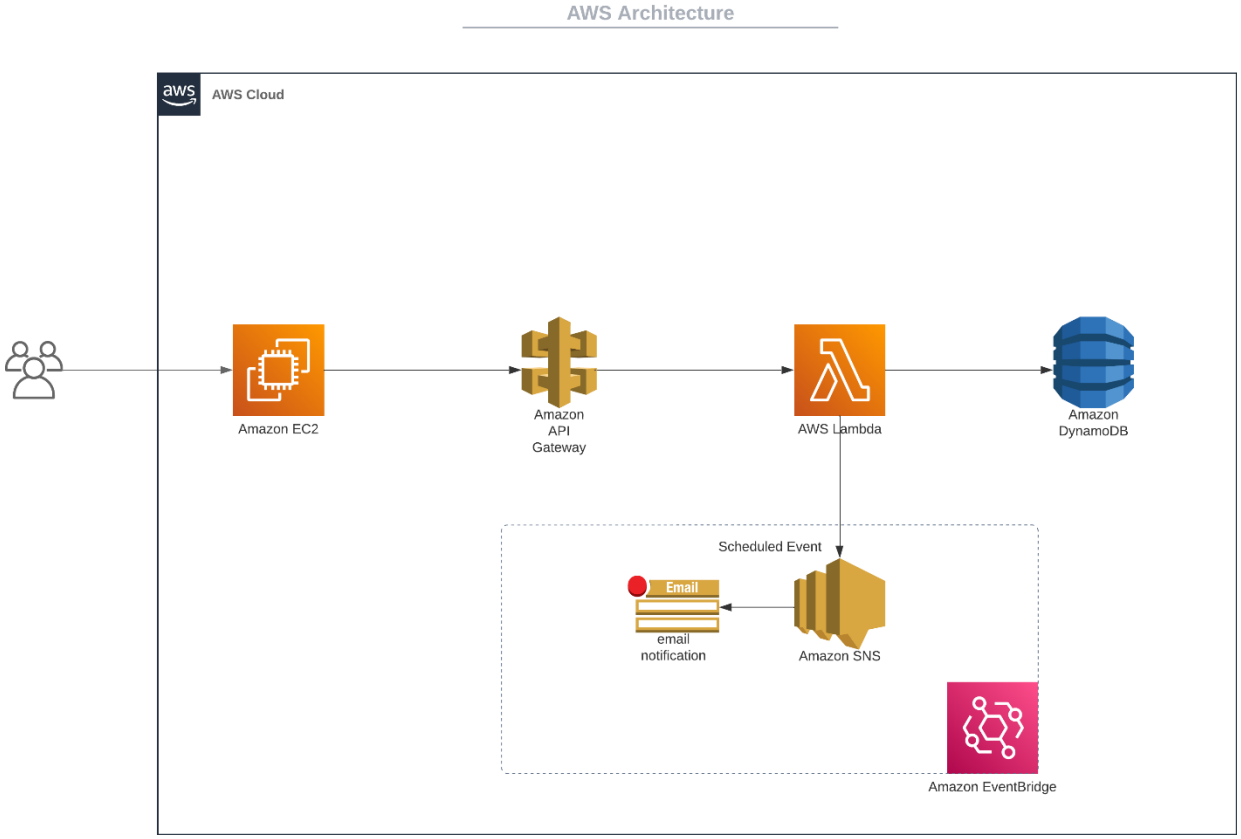
Deployment Model

The application is designed and deployed into the public cloud. As the entire application, frontend and backend are running on the AWS cloud. To use the "Pay as you go" model which brings down the cost of paying a huge chunk of money upfront for servers I have opted to deploy it on completely cloud. Moreover, it also reduces overhead for performing security measures, software updates, and database patches as it is managed by the public cloud provider.

Delivery Model

In terms of the delivery model, the application uses Software as a service. As it is considered software that provides solutions to manage daily financial activities and provide service over the Internet. Thus, a ready-to-use application hosted by the public could fall under the SAAS delivery model.

Architecture



cloud mechanisms

The image depicts the overall cloud architecture used to create the application. In this architecture, every user can access the frontend application running on an EC2 instance with its public IP. The front is responsible for the service user interface and interacts with the backend.

The EC2 instance can use the API routes provided by the HTTP API gateway. The gate has integrations for lambda and provided specific routes with its method so that the frontend can access the backend services rung on lambdas. The lambda function then receives the request and performs interaction with the NoSQL database AWS DynamoDB.

The architecture uses an event bridge scheduler. The event is a recurring and defined corn-based schedule that runs every day at 9 PM. The trigger creates a message and public into the SNS topic. The simple notification service then sends the email to subscribed recipient email ids to notify them.

Where is data stored?

The web application is using AWS-managed DynamoDB database to store application data. The application data includes all the categories, transactions, and alerts created by a user in tables. To manage the user authentication, data is also stored in the DynamoDB such as users' details and JWT tokens for stateless session management. In terms of managing user session data on the client side to provide protected frontend routes, the application stores data in local browser storage. Moreover, to deploy the application and create the resource stack using cloud formation, lambda function code, and frontend application is stored inside an S3 storage bucket in the form of a zip file. So that the resources can take code from s3 in deploy them while running the JSON script of cloud formation.

What programming languages did you use (and why) and what parts of your application required code?

As it is a web application it has two main components, frontend and backend. Each part requires code one is responsible for user interaction with the application and the other is business logic and database interactions.

To develop a user-friendly interface and provide a seamless single-page website application I have used the java script library ReactJS. In addition, it has component-based architecture so it provides code reusability and a maintainable website. The JavaScript library has rich community support too. Before choosing React JS for frontend I had considered other Java Script-based web frameworks such as Vue and Angular. However, React JS turned out to be more flexible and fast performing while rendering elements and doing DOM operations [4]. With above mentioned points I concluded and choose ReactJS.

For the backend, I have opted for Node JS as it is lightweight, asynchronous, and has huge active community support. This feature makes Node JS an obvious choice for the backend. In addition, as the application uses AWS serverless service Lambda and it is billed as per the execution time and memory used [5], node JS can save money as it is built on top of google chrome V8 engine and is memory efficient. Thus, I have finalized Node JS for backend support.

How is your system deployed to the cloud?

As application uses several AWS cloud services such as EC2, API Gateway, Lambda, SNS, EventBridge, and DynamoDB. To provision and deploy the code inside these resources I have used AWS Cloud formation. Once the cloud formation JSON script gets executed, it creates all resources which includes assigning roles, giving permissions to trigger events, and assigning instance profiles. It uses a zip file stored in an S3 bucket to deploy the code for Lambda and EC2.
Both frontend and backend get deployed with just one cloud formation script. The challenging part was to perform these operations in a single file was to replace the API call in frontend code with the endpoint of the backend. So once the backend is deployed and the API gateway completes staging all routes, the script uses the endpoint of the API Gateway and stores it inside the ec2 environment using user data in the instance initialization phase.

If your system does not match an architecture taught in the course, you will describe why that is, and whether your choices are wise or potentially flawed.

To deploy the application, the VPC and subnet can be considered to make the architecture More secure and isolated. Moreover, to make architecture effective AWS Cloud Front can add up to deliver content to the user in a much faster way and make architecture DDOS protected. During the initial development phase [7], I opted out of their architecture components, which can be considered to make the overall architecture flawless and secure.

Data Security

The security of the data in any application is one of the essential parts that need to be prioritized while developing and deploying stage of the application. In this application, data security has been implemented, and takes several measures at different levels to protect it. The application uses JWT tokens for user management and stateless session handling. All other routes except login and registration are protected and only authenticated users can able to handle it with valid tokens. This authentication system is on both sides frontend and backend. To protect routes on the client side, React JS uses local store and state data to make the application secure. The application also does email verification, it sends an email verification link to newly registered users' mail and user can only access the application if they are verified.

The code that is stored in S3 is inside a private bracket to ensure it can only be accessed by a cloud formation script running on an AWS account. To integrate lambda with API, the cloud formation creates permission for each lambda to ensure the least privilege principle. This is also considered during the creation of instance profiles for ec2 and defining security group inbound rules to open specific ports only.

Which security mechanisms are used to achieve the data security described in the previous question? List them, and explain any choices you made for each mechanism (technology you used, algorithm, cloud provider service, etc.)

The security mechanism described in the previous question includes a jwt token that generates toke by encoding user data with the help of secrete key and it can be verified with expiration bound expiration. It uses RSA secrete key to encrypt and decrypt the data. To make passwords secure in the database, the application uses Bcrypt hashing with salt and uses the Bcrypt package verify function to validate login attempts. AWS security mechanisms include the IAM role, security group for EC2, instance profile, and lambda permissions.

Costs

What would your organization have to purchase to reproduce your architecture in a private cloud while providing relatively the same level of availability as your cloud implementation? Try to give a rough estimate of what it would cost, don't worry if you are far off. These systems are complicated and you don't know all the exact equipment and software you would need to purchase. Just explore and try your best to figure out the combination of software and hardware you would need to buy to reproduce your app on-premise.

To set up the same architecture on-premises, the organization have to purchase not just a resource

but also hire server tech support and a system administrator that perform system checks, software updates, security patches, and response to maintain service available for use in term of any hardware or software failure. The organization will also be responsible for paying fees for reliable ISP and electricity generators to keep the server live during power outrage.

The main resource includes the physical server on which the frontend application is running. The routes and switches to manage networking. As in the cloud backend is deployed on serverless technology but on premises that too needs to be replaced by a physical server. To match the scalability of the lambda functions on changing request traffic, the first organization has to predict the number of requests and install physical servers for that. To make the architecture scalable and reduce downtime the organization needs to purchase cluster management software such as red hat cluster suites or Oracle application clusters [6].
For database, as DynamoDB is AWS fully managed database to find the alternative the organization can go with either MongoDB or Apache Casandra as distributed NoSQL database. Which requires a company to purchase additional storage hardware such as HDD/SSD. In addition, the cooling infrastructure also needs to be set up.

Table1: Cost Table [1][2]

Resource Type	Resource	Costing
Compute & Storage	Physical Servers	$3800
	Storage HDD	$10000
Network	Network Interface Card (NIC)	$1200
	Routers	$672
	Switch	$5593
Fault tolerance	Power Generator	$17000
	Cooling system	$26000
Software	OS	$1070
	Apache Cassandra / MongoDB	$5000
	Monitoring Software	$7000
	Cluster Management Software	$8000
Technical Staff	IT experts	$250000
Physical Security	Biometric System	$65000
	Surveillance Camera	$15000
	Security personals	$80000

Which cloud mechanism would be most important for you to add monitoring to make sure costs do not escalate out of budget unexpectedly? In other words, which of your cloud mechanisms has the most potential to cost the most money?

The front end of the application is deployed in AWS EC2 and I am using t2.meduim Linux instance. The instance is on demand and running 24x7 to handle user requests. Unlike other services such as lambda which gets charged per execution, the ec2 is charged on hourly bases. Thus, the instance cost holds the largest proportion in terms of cost. Therefore, the resource requires monitoring to keep an eye on its usage. With the help of monitoring incoming traffic, we analyze the frequency and number of users of the application. With the data generated by monitoring tools such as Cloud Watch, we can decide to go with reserve instances as we have an idea of several users. Reserve instances can save a lot of money compared to on-demand which can be beneficial for the organization. Although, the T2 instance series is the base and cheapest option

available, cost $0.0464 (medium 2vCPUs) per hour and that makes $406.44 a year [1].

How would your application evolve if you were to continue development? What features might you add next and which cloud mechanisms would you use to implement those features?

As a part of continuous development, to reduce the latency for users I can configure cloud front and CDN so that it can be accessed with high performance [3]. To make the architecture fault-tolerant, the application resource can be configured multi-AZ. As per the growth of the user base, more instances can be added with auto-scaling and load-balancing mechanisms to horizontally scale the application. In terms of new features, the application can provide backups and monthly financial reports (in pdf format) so that users can go through all the statistics with ease. To provide this feature, the AWS S3 can be used to store the generated reports for the user and data backups in specific time intervals.

AWS Resources

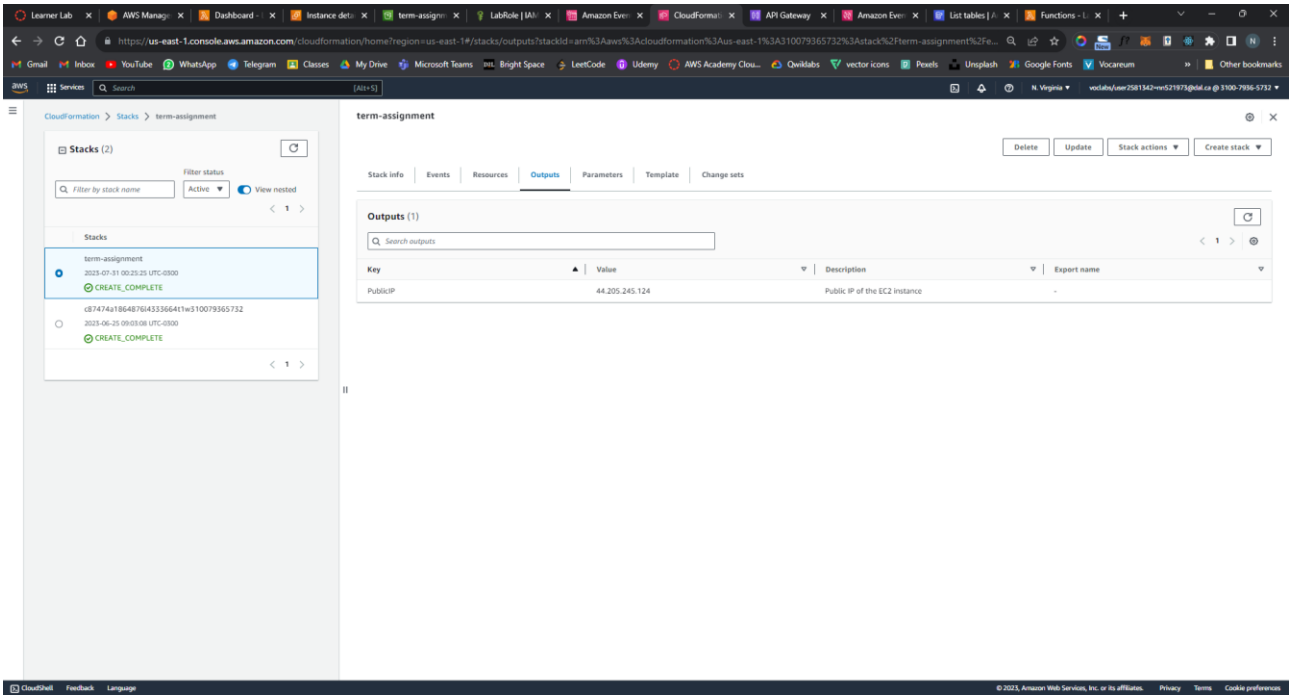


Figure 1: Cloud formation output

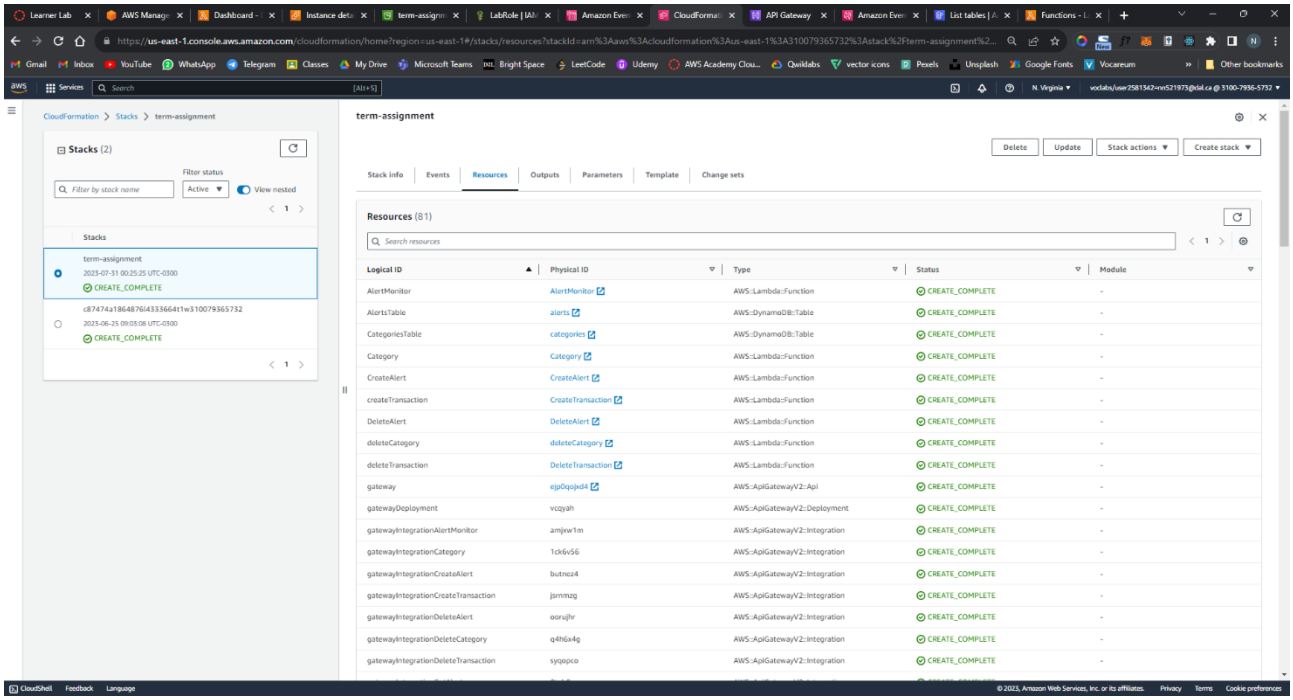


Figure 2: Cloud formation resources

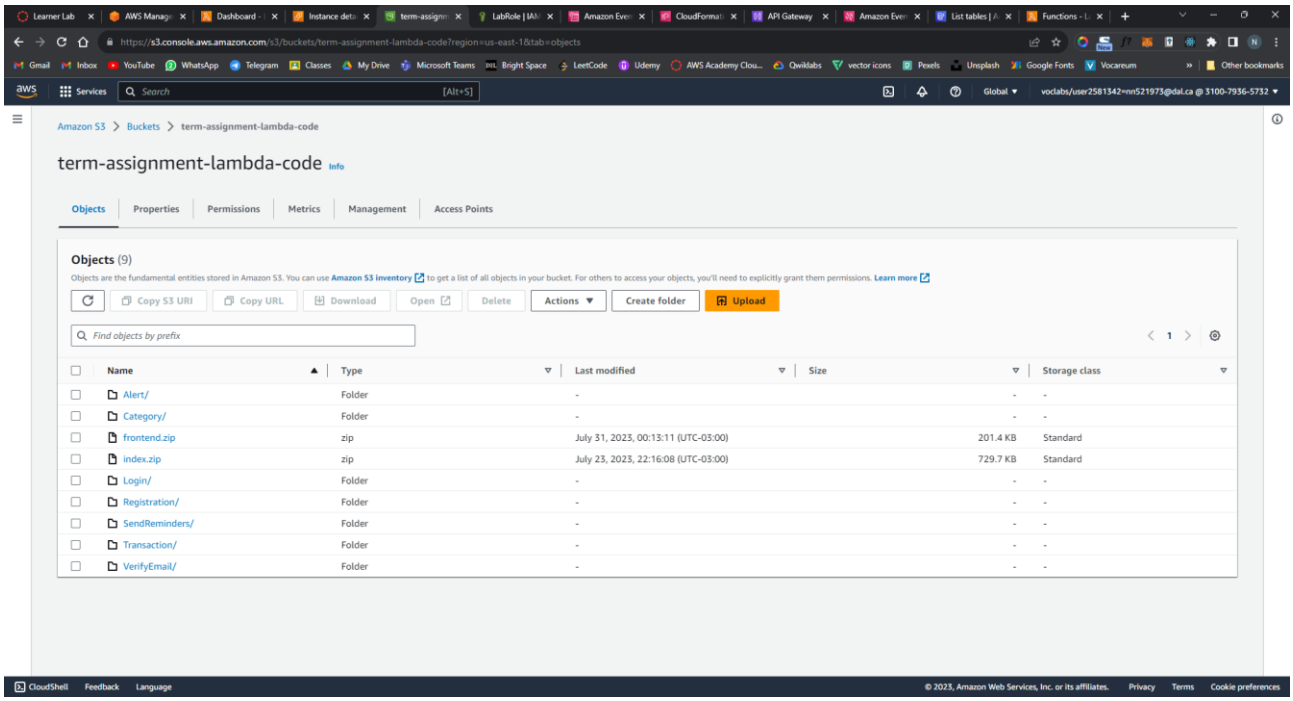


Figure 3: code zip file in s3 bucket

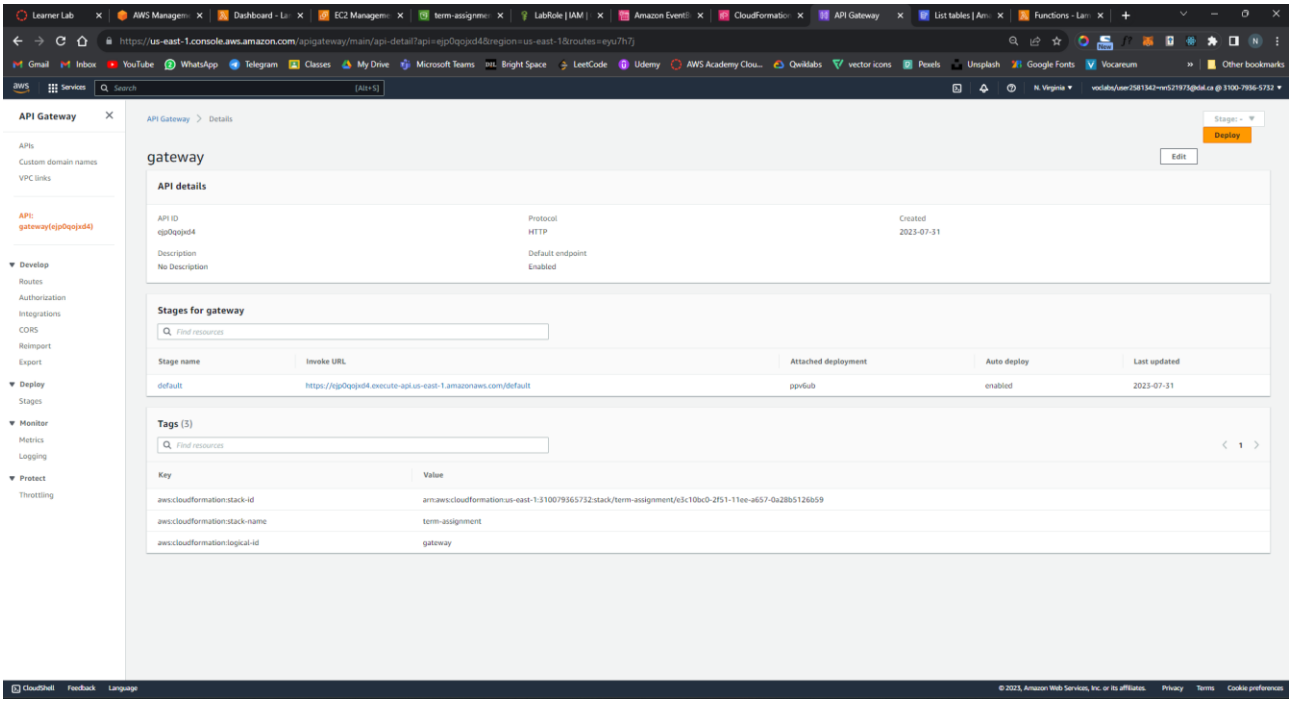


Figure 4: API gateway

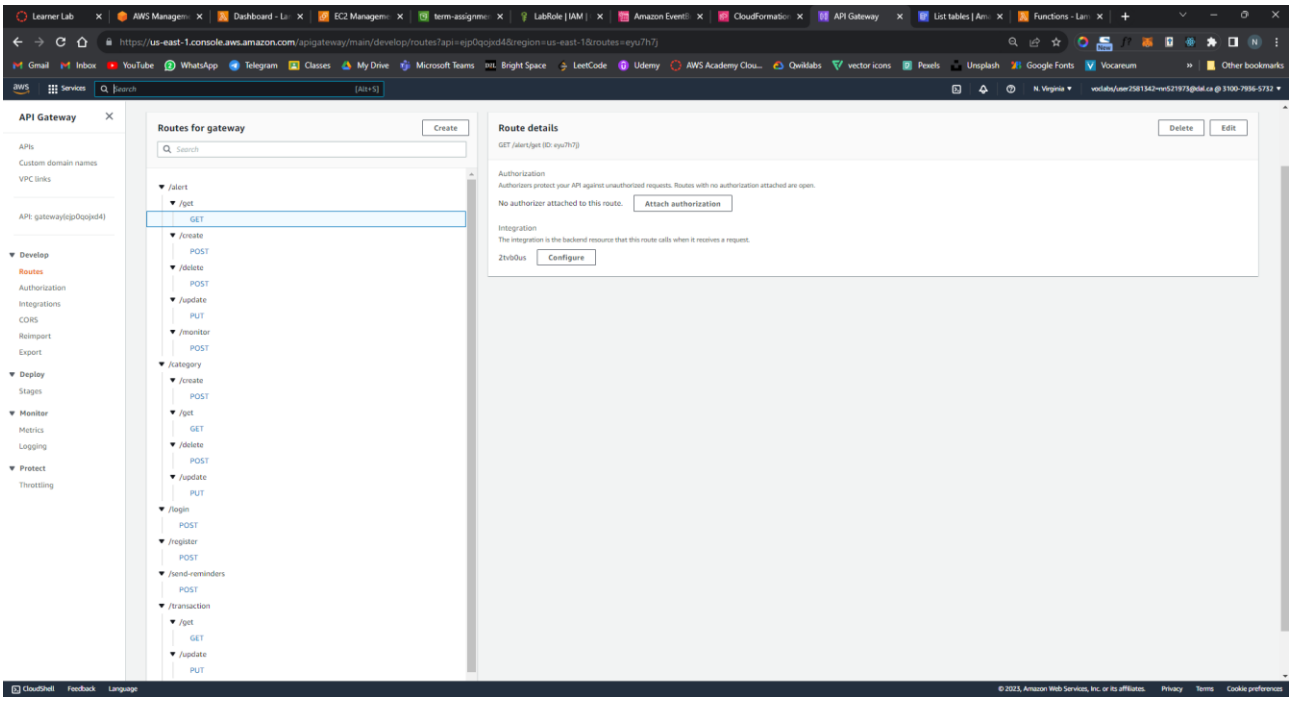


Figure 5: Gateway routes

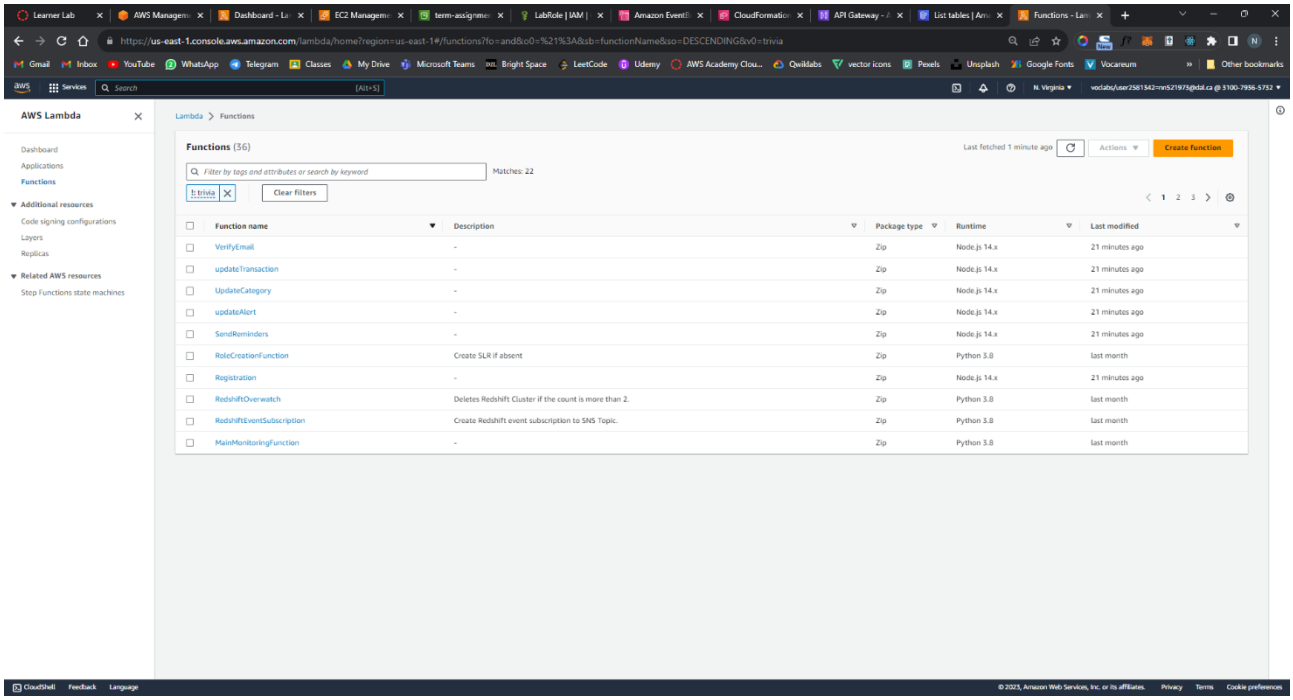


Figure 6: Lambda functions

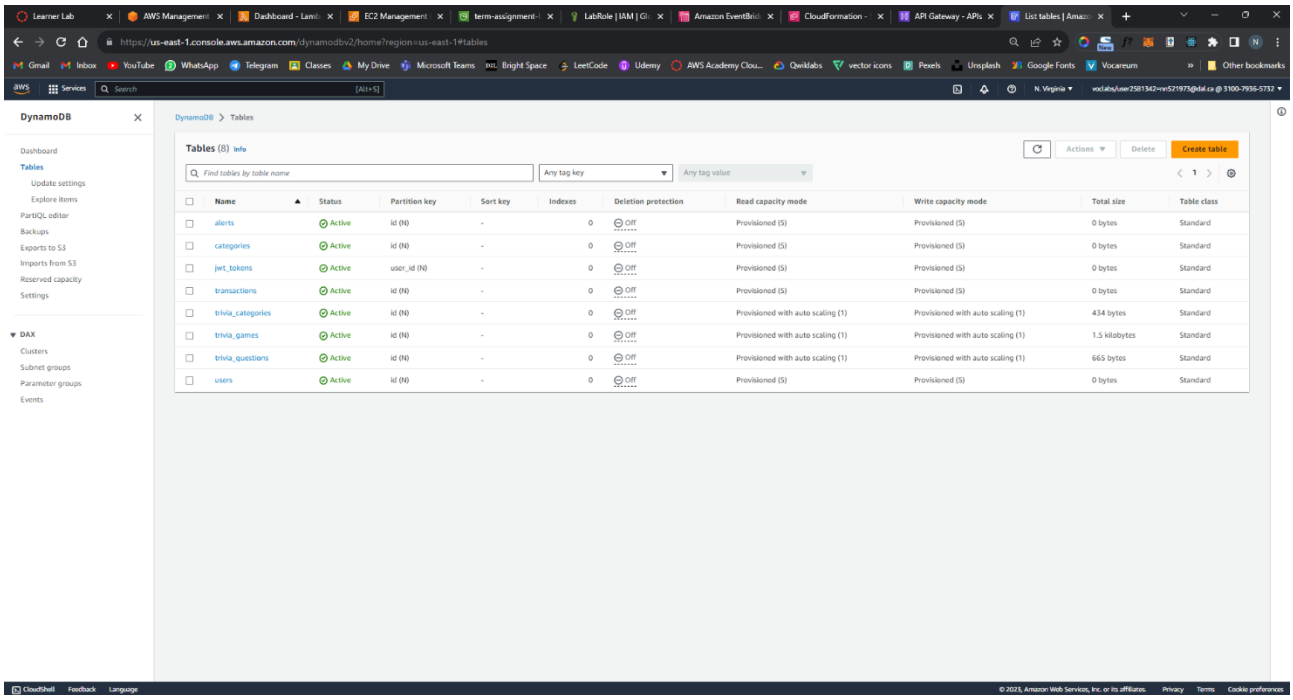


Figure 7: DynamoDB tables

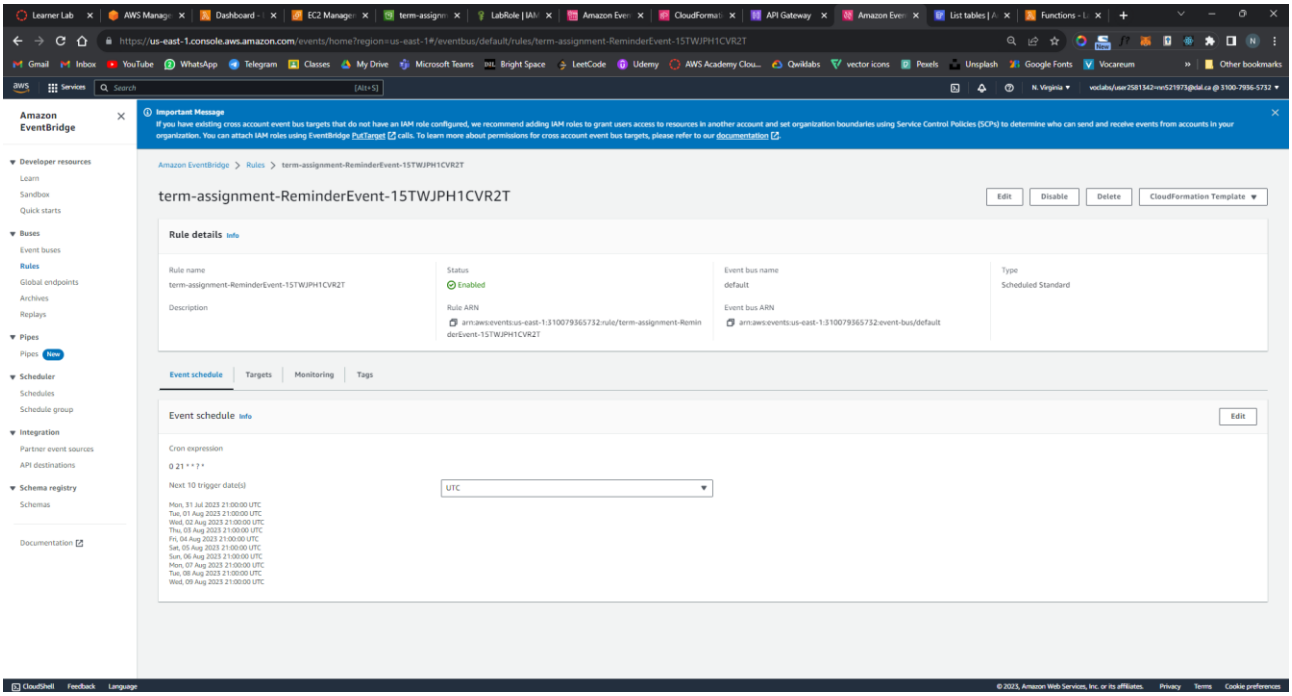


Figure 8: Event bridge rule

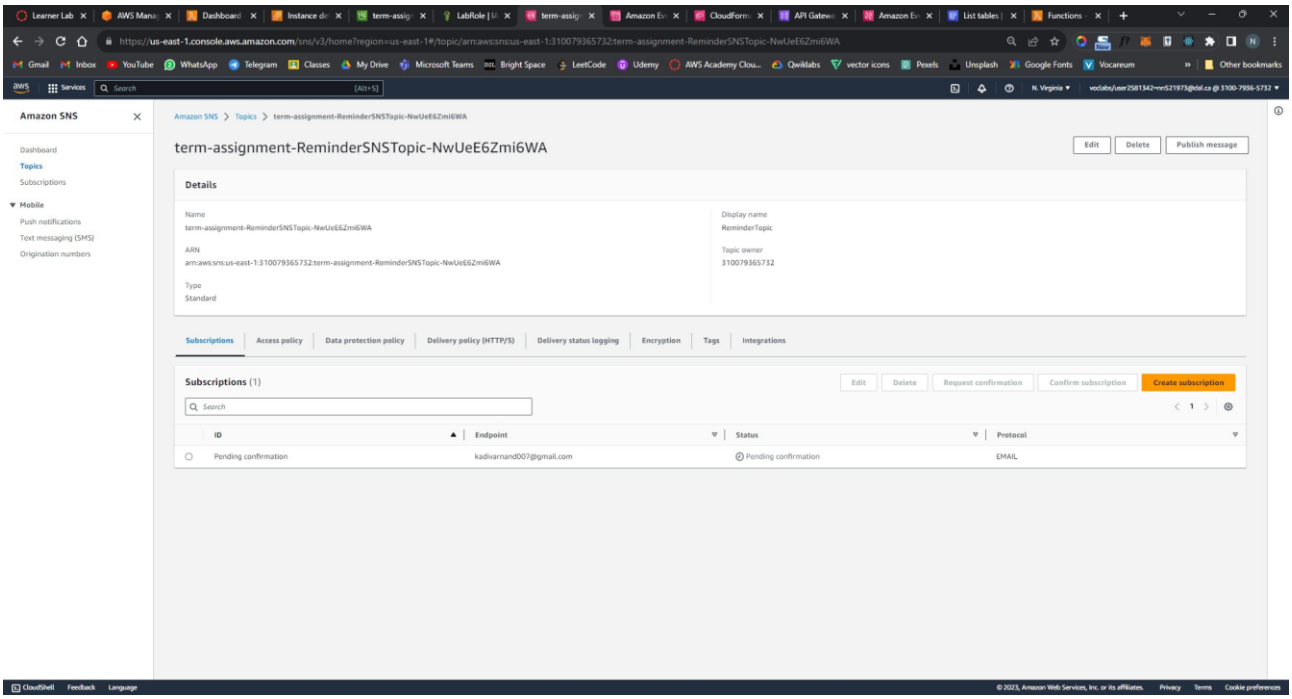


Figure 9: SNS

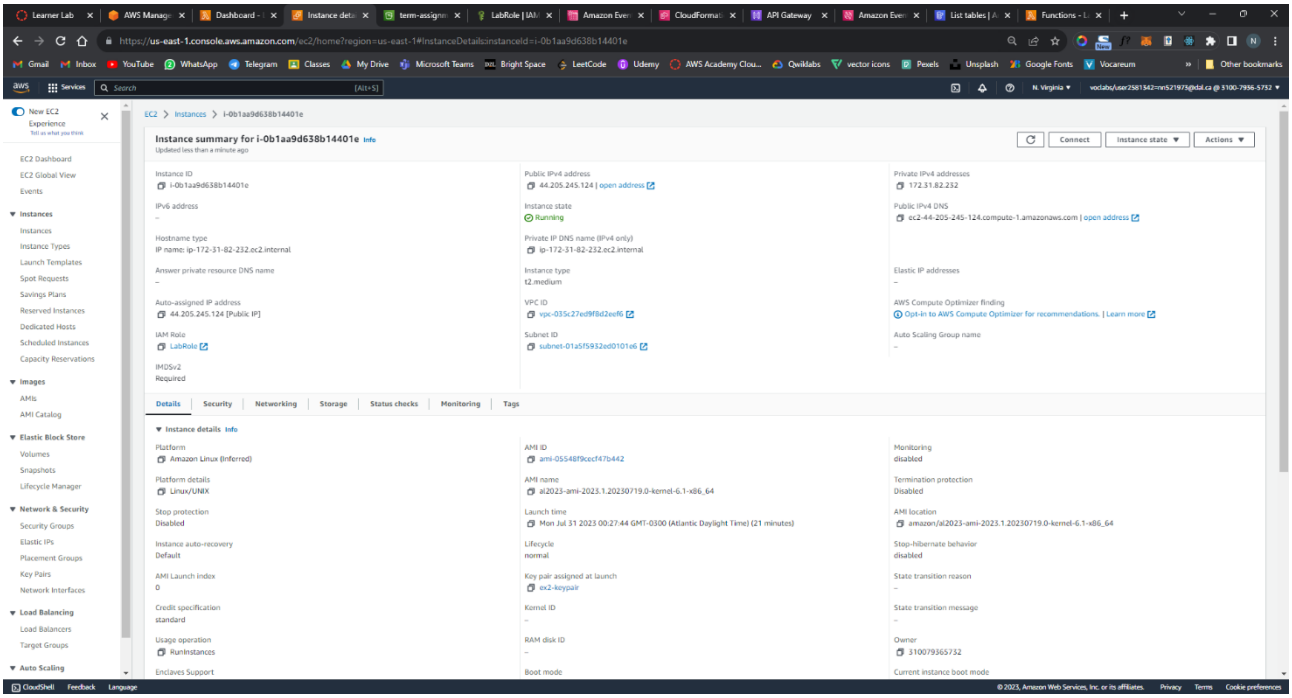


Figure 10: frontend ec2 instance

Application

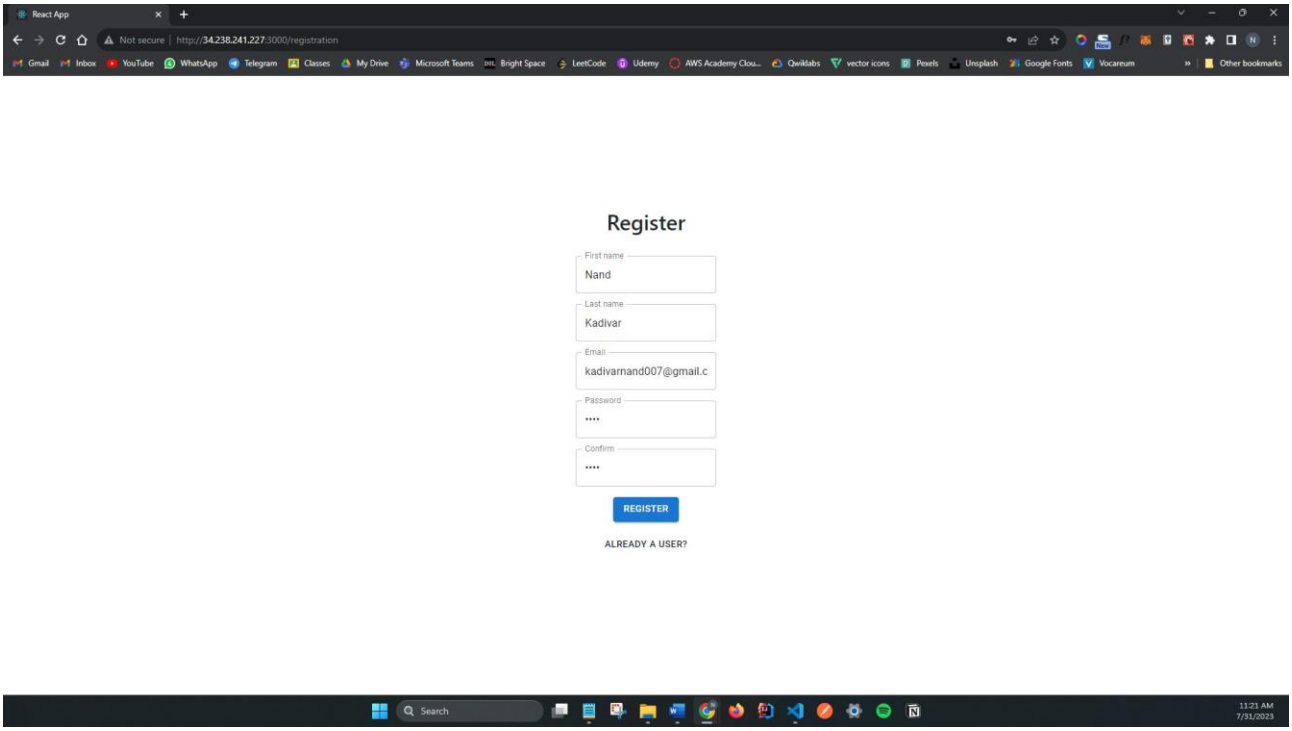


Figure 11: Registration

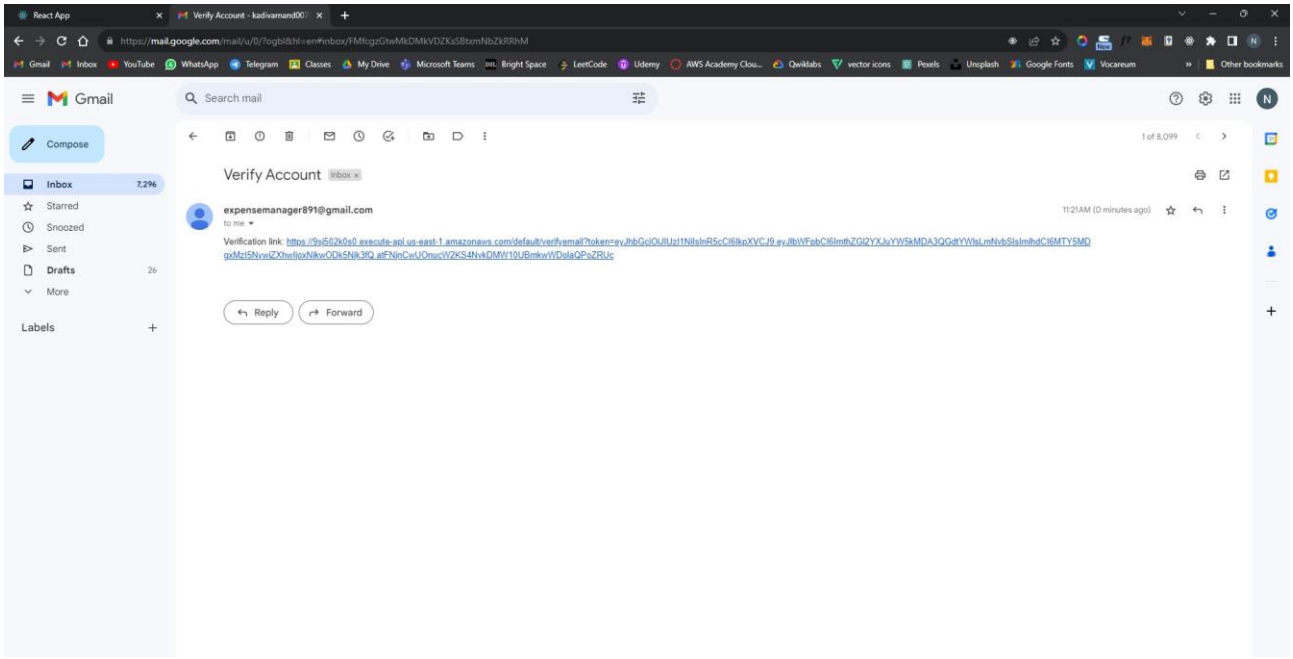


Figure 12: Email verification

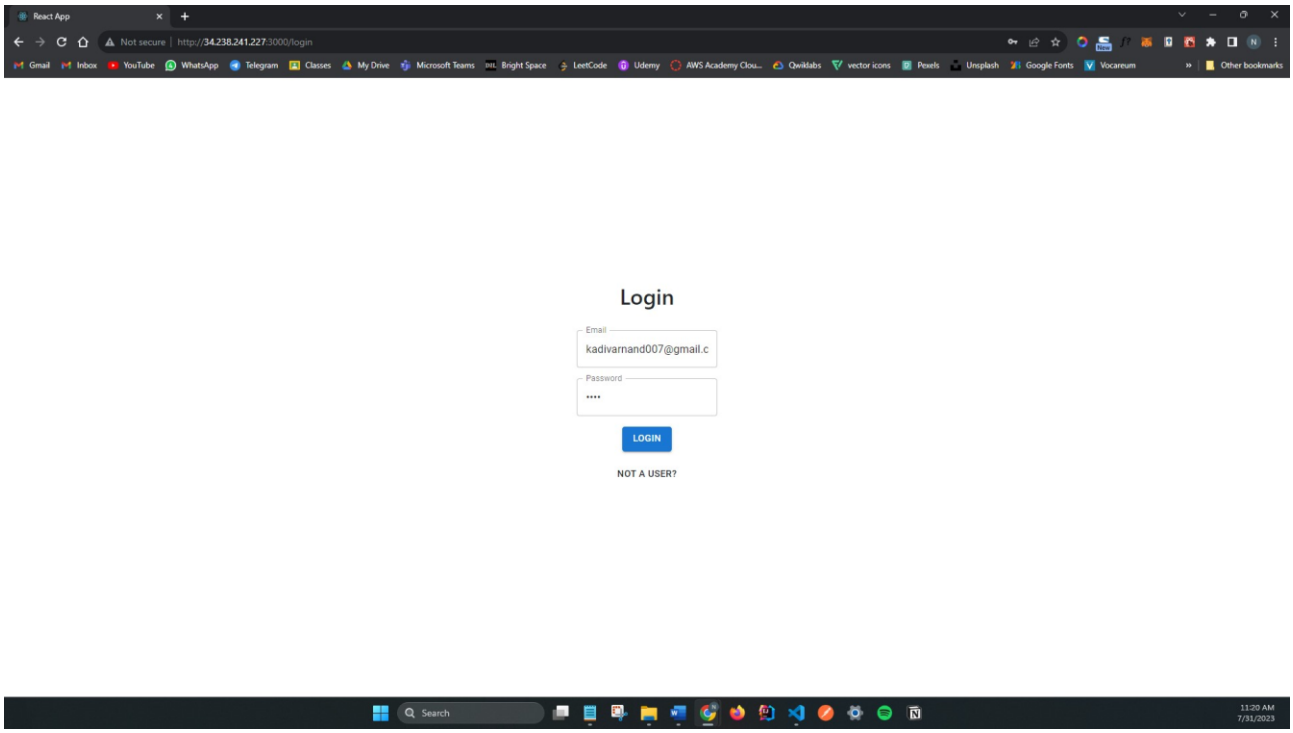


Figure 13: Login

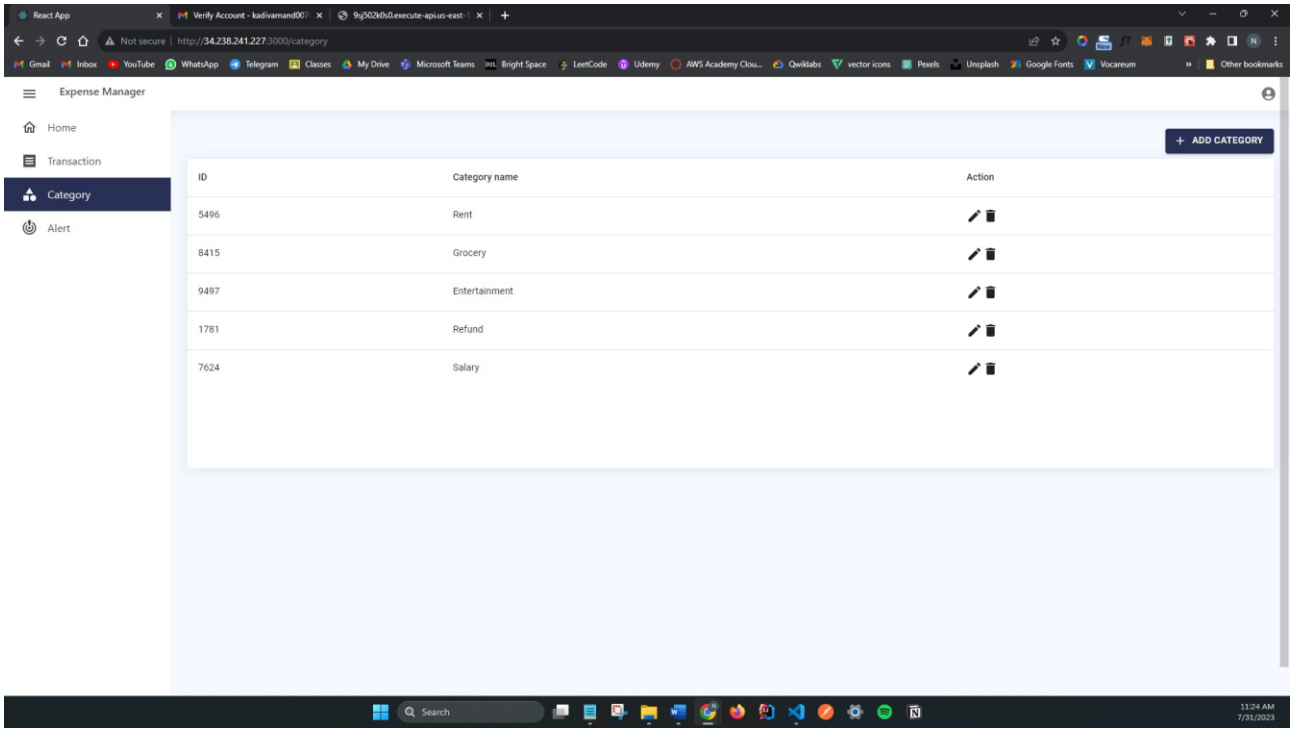


Figure 14: Categories

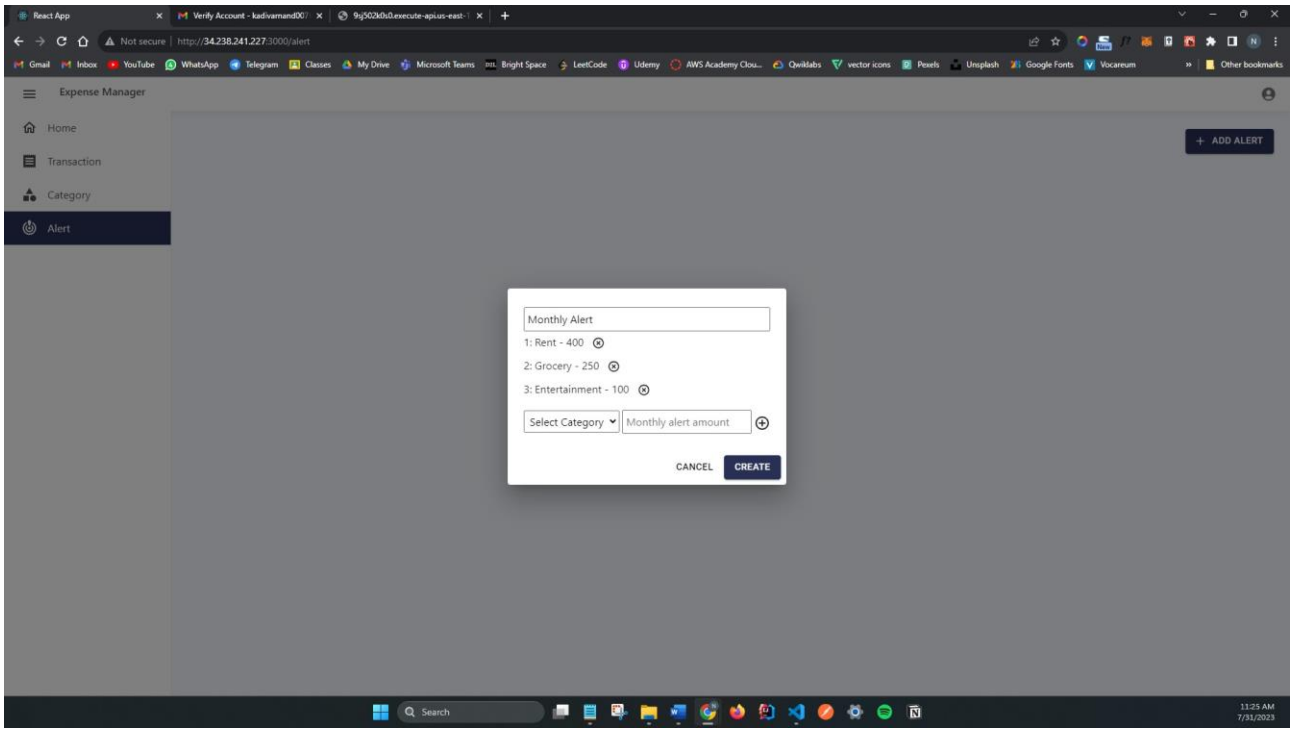


Figure 15: Create alert

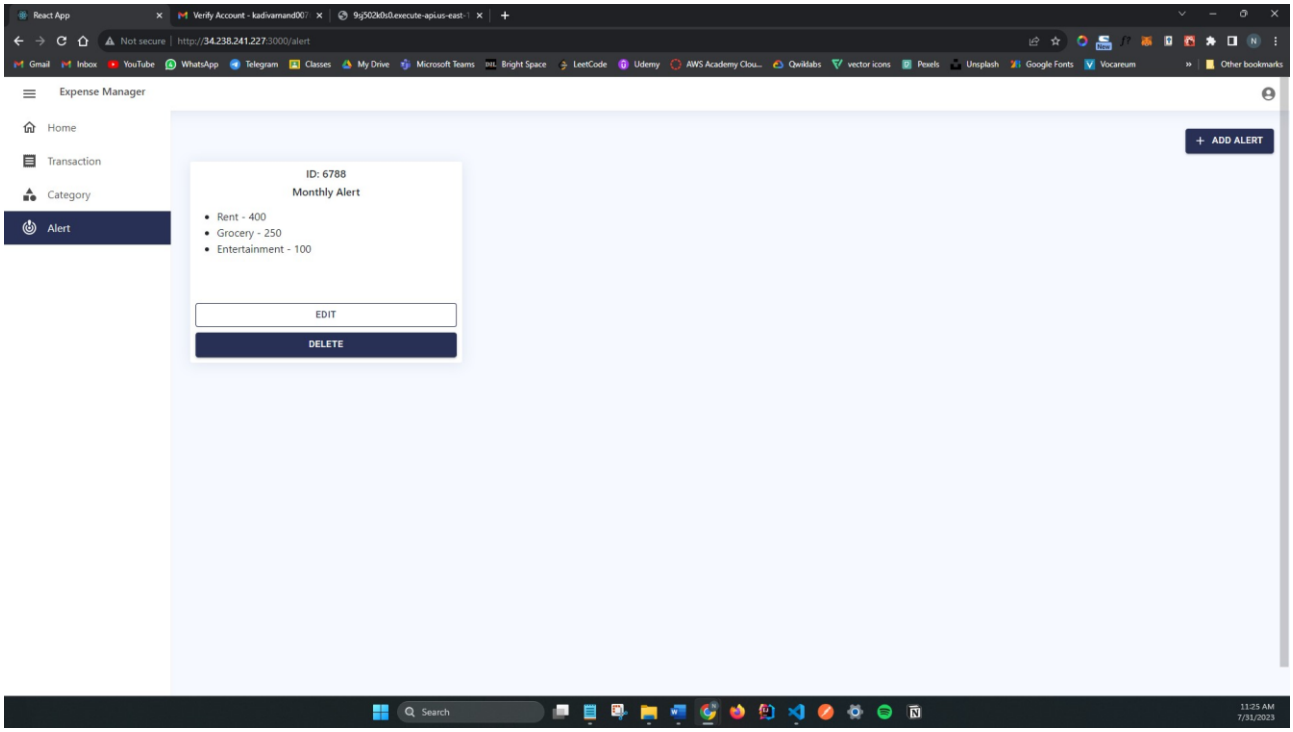


Figure 16: Alert

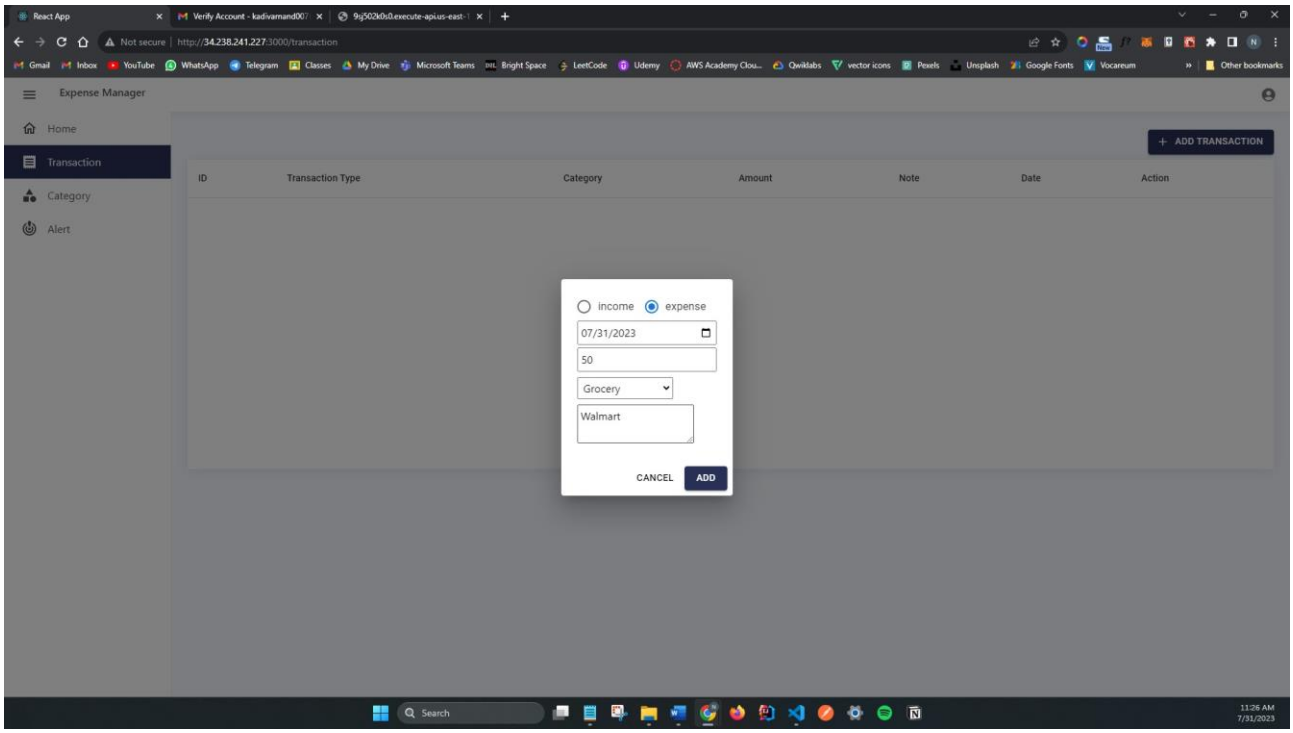


Figure 17: Add transaction

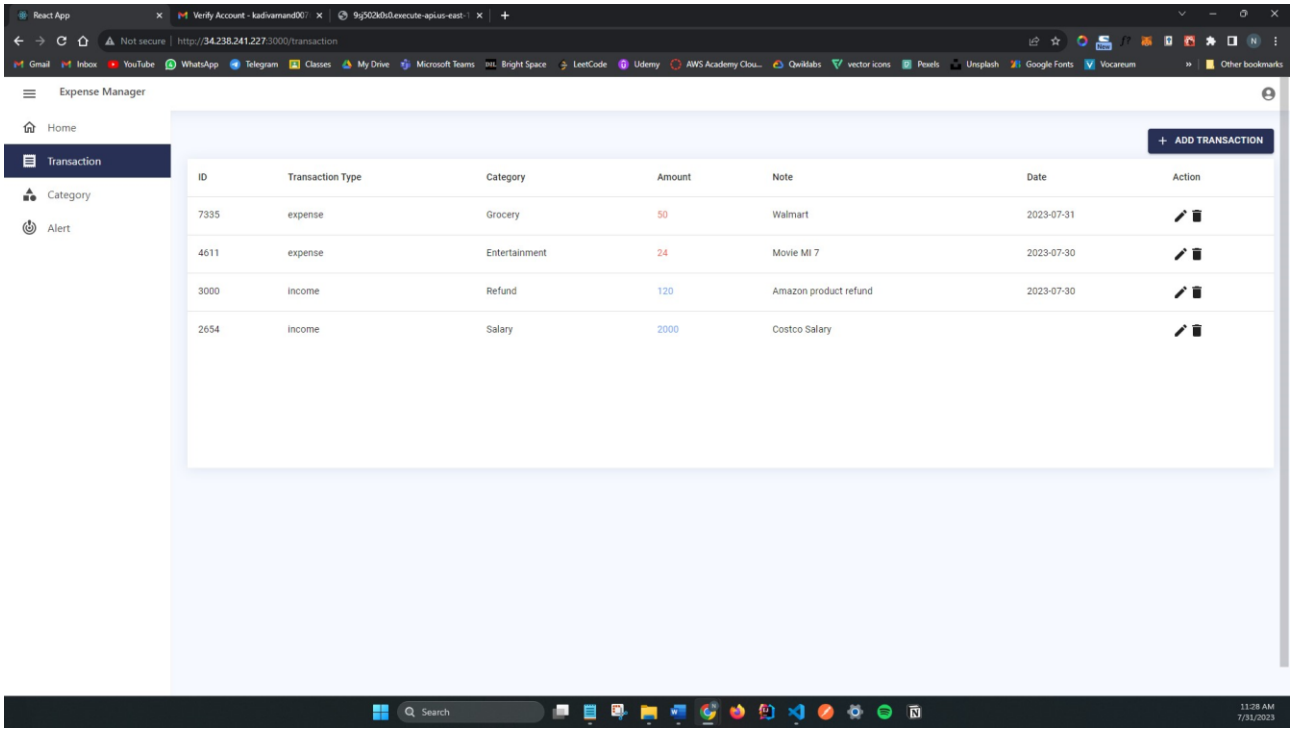


Figure 18: Transactions

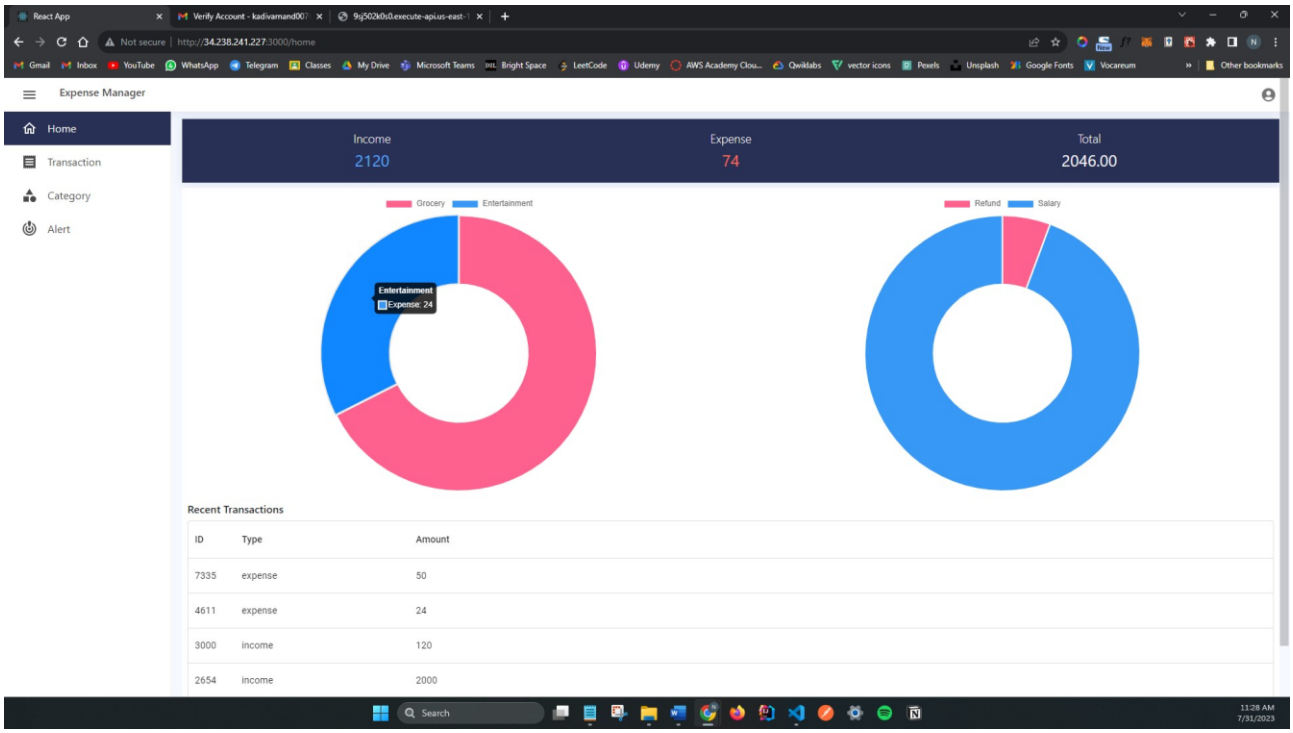


Figure 19: Analytical Dashboard

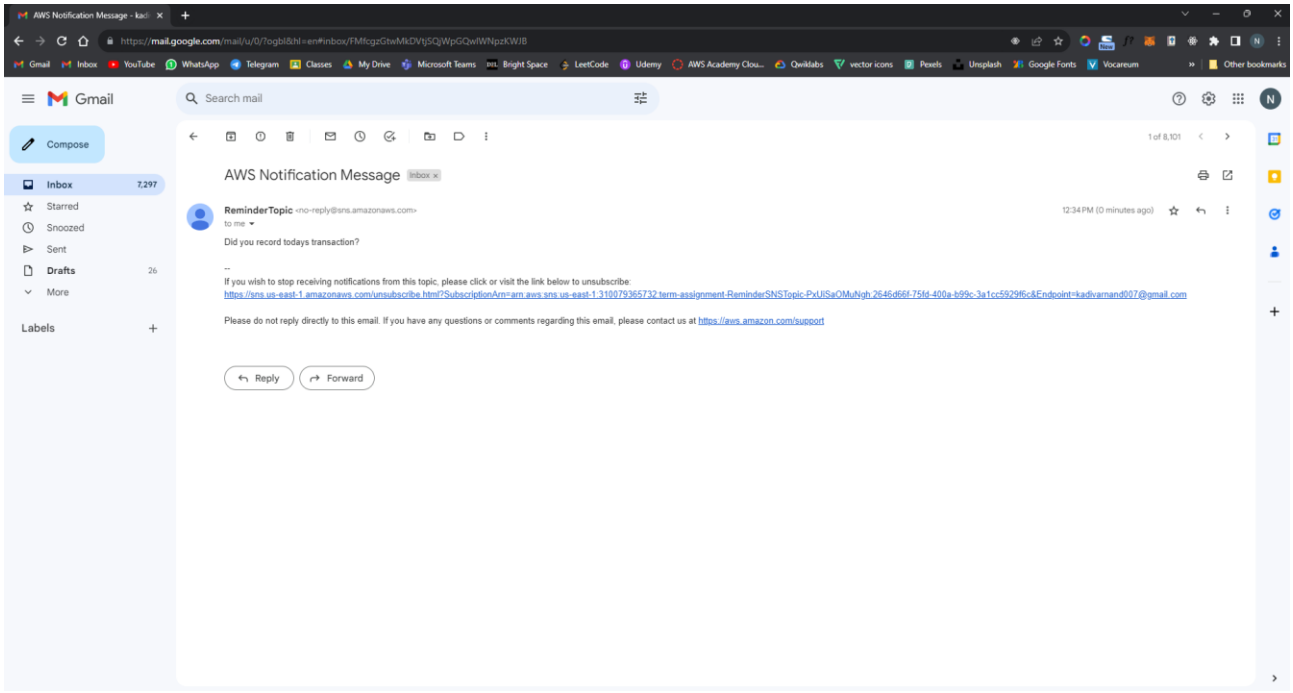


Figure 20: Reminder email

References

[1] "AWS Pricing Calculator," *Calculator.aws*, 2023. Available: https://calculator.aws/#/. [Accessed: Jul. 28, 2023]

[2] E. Brinkman, "What's the Cost of a Server for Small Business," *Servermania.com*, 2023. Available: https://www.servermania.com/kb/articles/how-much-does-a-server-cost-for-a-small-business. [Accessed: Jul. 29, 2023]

[3] "What is Amazon CloudFront? - Amazon CloudFront," *Amazon.com*, 2023. Available: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html. [Accessed: Jul. 30, 2023]

[4] M. Joshi, "Angular vs React vs Vue: Core Differences | BrowserStack," *BrowserStack*, May 11, 2023. Available: https://www.browserstack.com/guide/angular-vs-react-vs-vue. [Accessed: Jul. 30, 2023]

[5] "Serverless Computing – AWS Lambda Pricing – Amazon Web Services," *Amazon Web Services, Inc.*, 2023. Available: https://aws.amazon.com/lambda/pricing/. [Accessed: Jul. 30, 2023]

[6] "Run a Single Database across Multiple Servers," *Oracle.com*, 2014. Available: https://www.oracle.com/ca-en/database/real-application-clusters/. [Accessed: Jul. 30, 2023]

[7] AWS Official, “I want to protect my application from Distributed Denial of Service (DDoS) attacks with AWS Shield Standard.,” Amazon Web Services, Inc., Aug. 17, 2022. Available: https://repost.aws/knowledge-center/shield-standard-ddos-attack. [Accessed: Jul. 31, 2023]