SCHOOL OF ENGINEERING, COMPUTING AND BUILT ENVIRONMENT

Department of Computing

BSc/BSc (Hons) Computing

Cloud Platform Development

Module code: MHI324190

Benson Mugure - S2038770

ALC Module Lead: Yovin Poorun

GCU Module Lead: Sajid Nazir

I declare that all work submitted for this coursework is the work of Benson Mugure alone unless stated otherwise.

Table of Contents

List Of Figures

List Of Tables

Problem

In today's digital era, the integration of cloud computing platforms like Amazon Web Services (AWS) has revolutionized the way applications are developed, deployed, and managed. However, as the reliance on cloud-based solutions grows, so does the need for robust and secure implementations that adhere to best practices in both development and operational aspects. This project explores the challenges and considerations involved in implementing an application on AWS, with a focus on optimizing for cost-effectiveness and ensuring robust security measures.

The primary objective of this project is to develop an application on the AWS platform while addressing key concerns related to cost optimization and security. The application, simulated within the AWS Academy Learner Lab environment, involves the creation of a camera system utilizing EC2 instances for image processing, S3 buckets for storage, SQS queues for message handling, DynamoDB tables for data storage, and Lambda functions for serverless computing. The application workflow includes the periodic upload of image files to S3, triggering of Lambda functions via SQS messages for image analysis using AWS Rekognition, and updating a DynamoDB database with relevant information extracted from the image analysis results. Depending on the nature of the vehicle in the image, an email notification may be sent to an administrator if the vehicle is blacklisted or if there is no information about it. My solution will seek to optimize this system for security and cost aspects.

A likely scenario where such a system may be utilized is in a smart home system, where a homeowner may want to monitor who is going in and out of their house remotely. This may involve multiple users, being a home to a family, with different and varied permissions.

Solution

To ensure a cost-effective and secure implementation of the application on AWS, the chosen approach focuses on leveraging a variety of AWS services tailored to specific requirements.

Firstly, an Amazon EC2 instance is utilized to simulate the camera system responsible for image processing. The instance is selected and configured based on workload demands to optimize cost while ensuring efficient resource utilization. I plan to use it to store the images before uploading them to the s3.

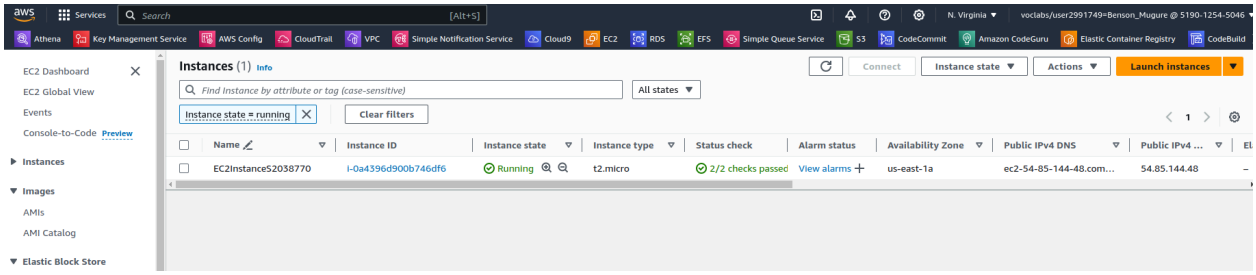

Figure 1. My EC2 Instance

Next, I employed Amazon S3 buckets for storing image files uploaded by the EC2 instance. The implementation included:

- Lifecycle policies are implemented to manage the storage lifecycle of objects stored in the S3 buckets. Infrequently accessed data is transitioned to cost-effective storage classes like S3 Glacier using lifecycle policies. By automatically moving data to lower-cost storage tiers based on defined rules and criteria, lifecycle policies help optimize storage costs over time without compromising data availability or durability. This approach aligns with cost optimization strategies by ensuring that storage resources are utilized efficiently based on the access patterns and lifecycle of the data.
- I configured my s3 bucket with Server Side Encryption (SSE) using AES256 encryption. By implementing SSE, data stored in S3 buckets remains encrypted at rest, providing an additional layer of security against unauthorized access or data breaches. This encryption mechanism ensures the confidentiality and integrity of data stored within the buckets, aligning with industry best practices and compliance requirements.
- Versioning is enabled on my S3 bucket to maintain multiple versions of objects stored within them. This feature provides protection against accidental deletion or modification of objects by preserving previous versions, thereby enhancing data durability and integrity. Versioning also facilitates data recovery and rollback processes, contributing to improved data management practices and mitigating the risk of data loss.

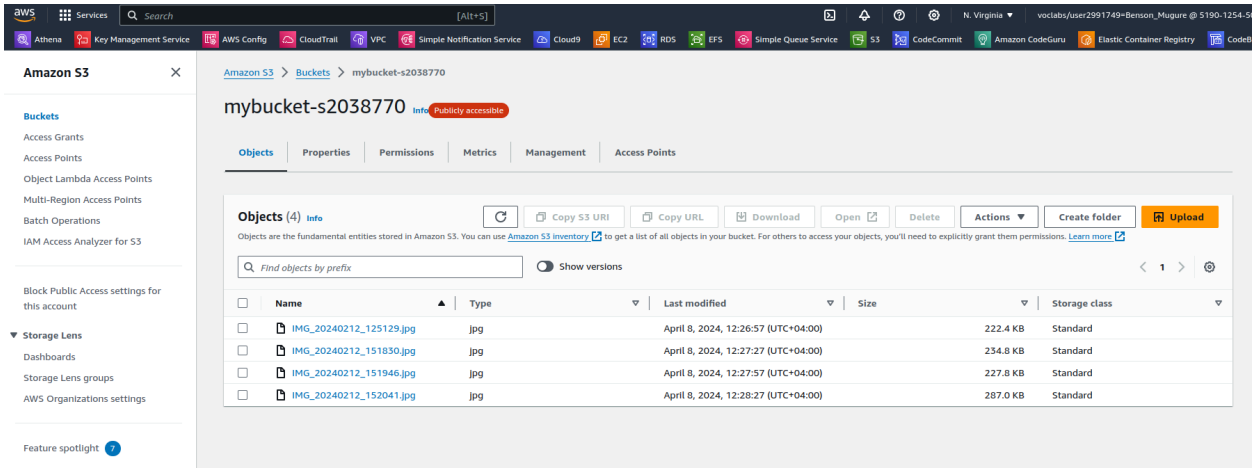

Figure 2. My s3 bucket with items

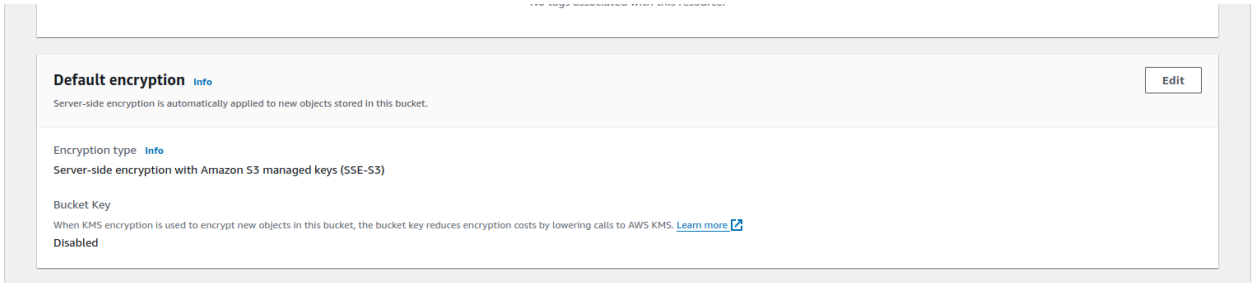

Figure 3. Data encryption in my s3 bucket

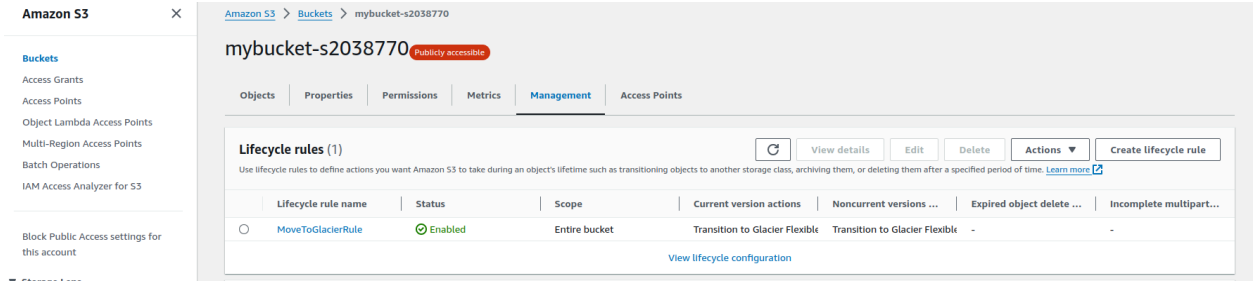

Figure 4. Lifecycle rules in my s3 bucket

I created a Cloudfront CDN (Content Delivery Network) in front of my S3 bucket in order to cache my data and help with availability. Cloudfront increases availability of resources especially if they are required far from where they are stored. It optimizes latency and improves access speeds for downloading resources such as the images stored in the bucket.

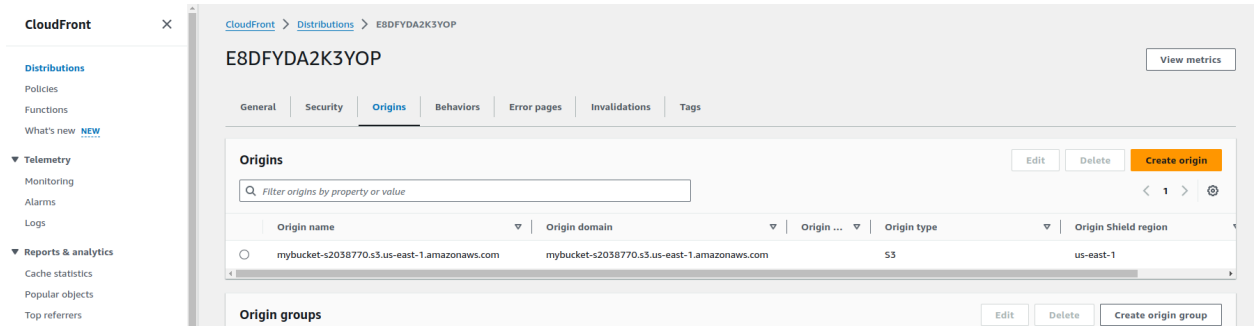

Figure 5. My Cloudfront CDN for my s3 bucket

I used boto3 python code to create the EC2 instance and therefore did not save the private key. When it came to uploading the images so that I can then upload them from EC2 to S3, I used wget. I first downloaded the zipped image file from GCU Learn then uploaded it to Google Drive and made it available to anyone with the link. I then used the following command on my ec2 in order to download it:
wget "https://drive.google.com/uc?id=1ZvkW3a02woJJ_3DL7C5pfmJBFulVaPbF" -O images.zip.
I then unzipped the file to get the folder with images. I also had to download pip and then boto3 on my EC2 instance in order to run the code that uploads the images to the s3 bucket. Even after that, I needed to either have credentials to access my s3 bucket, or update my EC2 Instance role to one that can access the buckets, such as LabRole. I chose the latter as it was simpler.

For message handling and triggering of image analysis tasks, an Amazon SQS queue is integrated into the application architecture. SQS provides a scalable and cost-effective solution for decoupling components, ensuring seamless communication between services while optimizing resource usage. A Queue needs a destination that will poll it for messages and in this case I used an AWS Lambda. By creating a queue, I am creating an asynchronous workflow in the event that there are more messages (or files uploaded) in our queue than how many lambdas we have limited to run this process in parallel. Queues are good for one-to-one asynchronous communication between services, as a temporary message holding pool and ordered message processing that needs to be set up at queue construction. The FIFO system has limitations on the number of messages that can be in the Queue and thus other systems of message processing are the default.

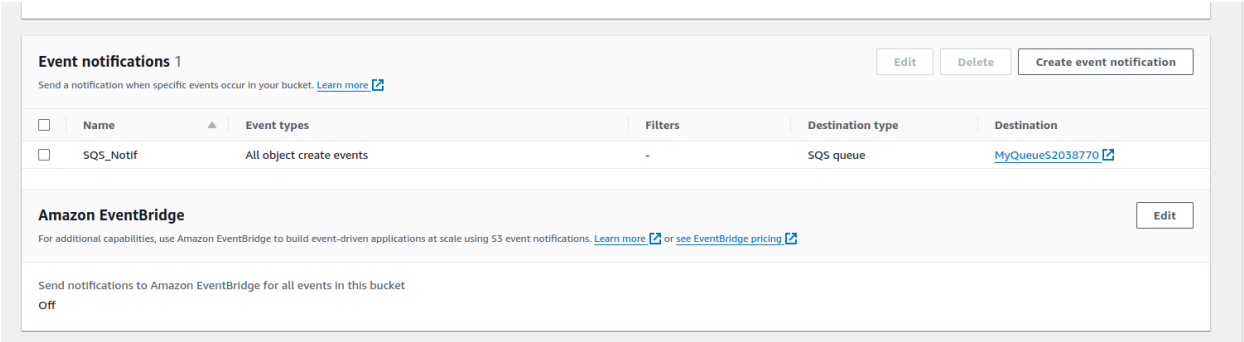

Figure 6. My s3 notification to my SQS Queue

AWS Lambda functions are employed for serverless computing, enabling on-demand execution of image analysis tasks triggered by messages from SQS. With Lambda's pay-as-you-go pricing model, the application only incurs costs proportional to the compute resources consumed during execution, leading to cost-efficient operations. I particularly employed Lambda functions in two instances. The first was in getting the data of the image uploaded to the s3 bucket from the data in the SQS Queue. As a destination and polling agent of the queue, the lambda retrieves important information such as the image name, the time of upload and the IP of the source of the image.

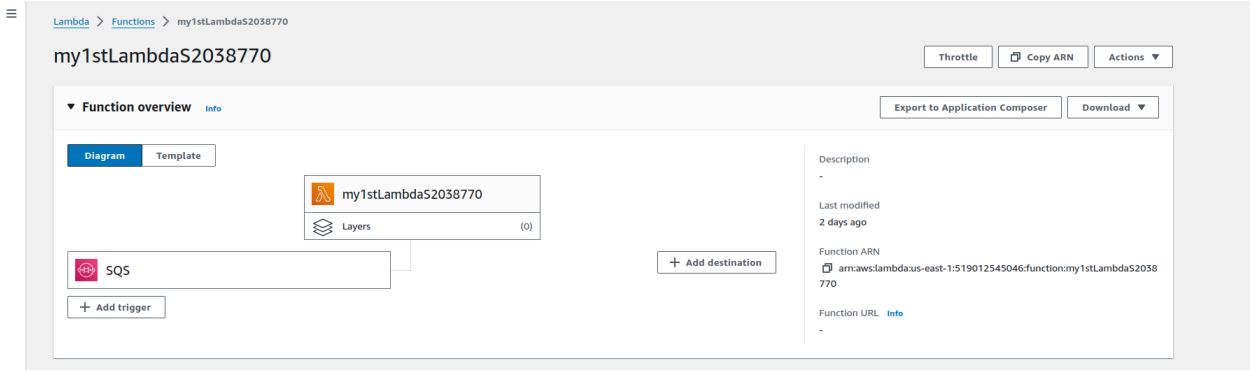

Figure 7. My First Lambda with an SQS trigger

AWS Rekognition is leveraged in the first lambda function for label and text detection in images uploaded to S3. With its pay-as-you-go pricing model, Rekognition ensures cost efficiency by charging only for the specific features utilized, aligning with the overall cost optimization strategy of the application. The lambda function then stores the image data, together with the detected texts and labels, in a dynamoDB entryTable.

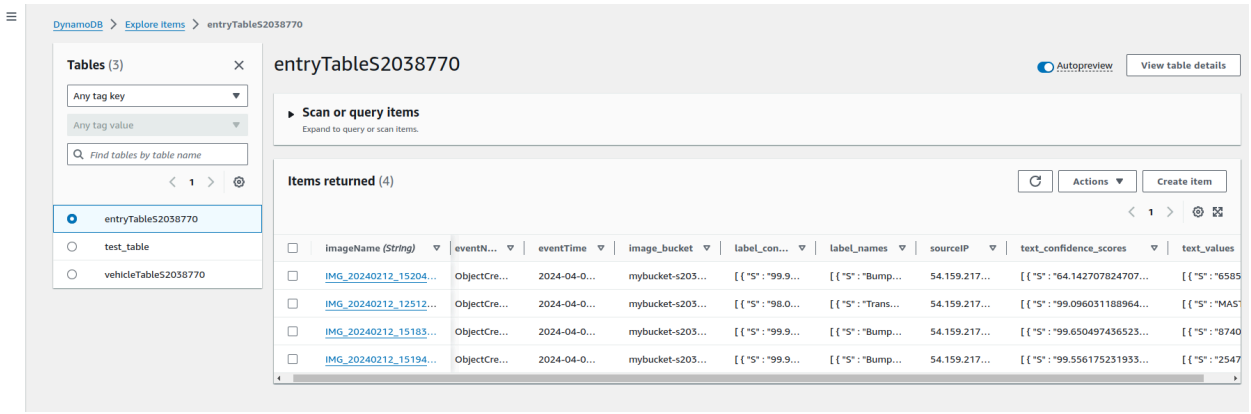

Figure 8. My populated entry table

To store image information and vehicle details, Amazon DynamoDB tables are utilized. Provisioned capacity mode with auto-scaling is implemented to optimize costs based on actual usage patterns, ensuring that resources are provisioned optimally to meet demand while minimizing unnecessary expenses. DynamoDB is a standard NoSQL database that scales horizontally. It has very fast performance, is very easy to use and very popular so it has a large community for support, in addition to being used by many applications and businesses. I created two tables. The “master” table is called vehicleTableS2038770 and it stores which vehicles are blacklisted and which ones are not. I only stored three of the four vehicles here, as I wanted to simulate the scenario of one unknown vehicle. In the three, only one was blacklisted.

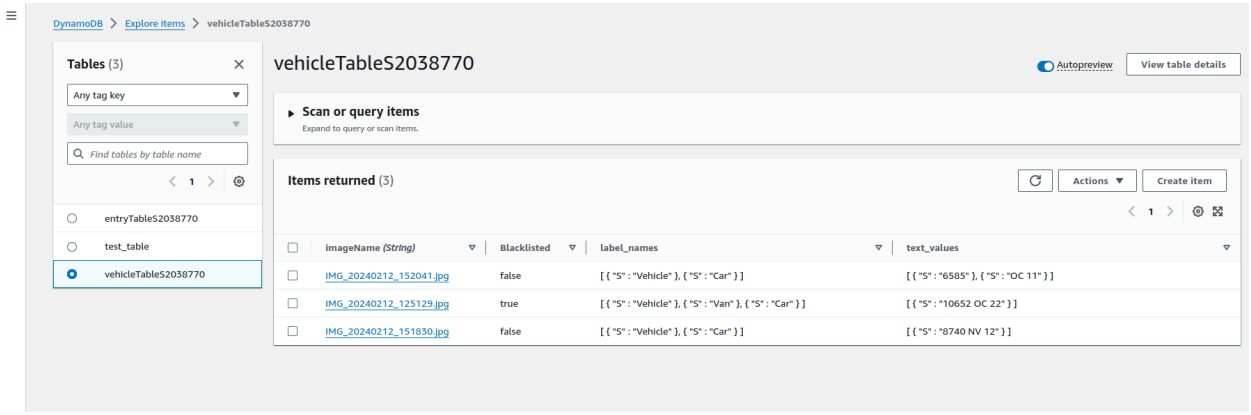

Figure 9. My populated vehicle table

Entry into the entry table by the first lambda triggers the second lambda which is used to check for blacklisted and unknown vehicles. It does this by scanning the vehicles table and comparing the text stored there with the text detected in the entry table row that was just entered. If blacklisted text, or an unknown vehicle is detected then an SNS topic is triggered and emails are sent to the subscribers.

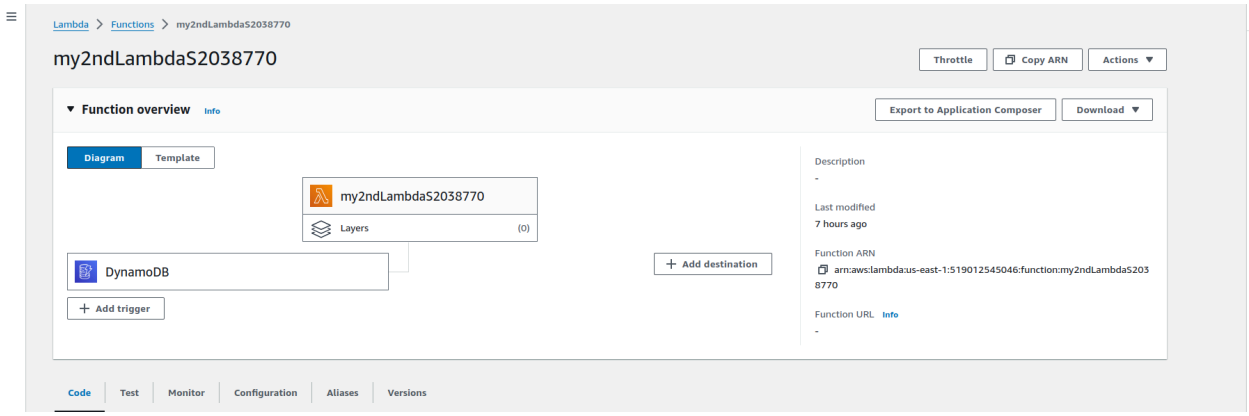

Figure 10. My second lambda with a dynamoDB trigger

The lambdas required setting up a role for access to SQS and dynamoDB and I chose LabRole for this.

I used Amazon SNS for the email notification. SNS uses a Publisher-subscriber system where you own and publish to a topic and subscribers get notified of events that are delivered to that topic. A topic can have many subscribers (fan out approach) of different types (SQS, Lambda, Email). You should use an SNS if other people/entities care that an event has happened and so you need to notify them. It is like a newsletter. SNS allows asynchronous data processing by the different subscribers that offer a layer of protection, compared to sequential processing where if any one of the processes fail or data is corrupted in one of the processes, then the later processes will all be erroneous or end up failing all together. It is advisable that an SNS topic have SQS queues as subscribers instead of the actual processing endpoint. This is because if the endpoint fails, we will not need to depend on the retry mechanism available in SNS as a queue's messages are available until the polling agent requests them. In my case, however, I could not use an SQS queue as a subscriber since the processing endpoint was an external agent, an email address, and thus could not poll an AWS SQS queue. I also included an s3 link to the concerned image in the email sent.

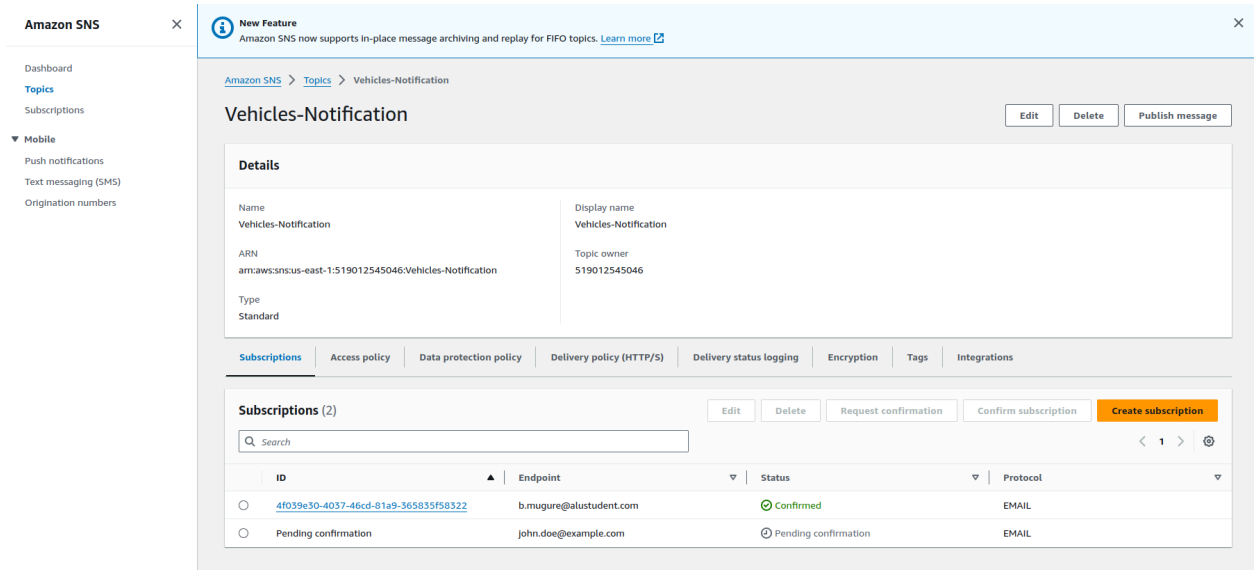

Figure 11. My vehicle notification SNS topic

In the email, I included a link to an API generated using API gateway that sends an SMS notification, using an SNS topic subscription, requesting for assistance then displays emergency contacts when clicked. This is in case the email recipient wants to call for help. The technical implementation is that the API link triggers a lambda that notifies the SMS SNS topic and then returns a response with emergency contacts.

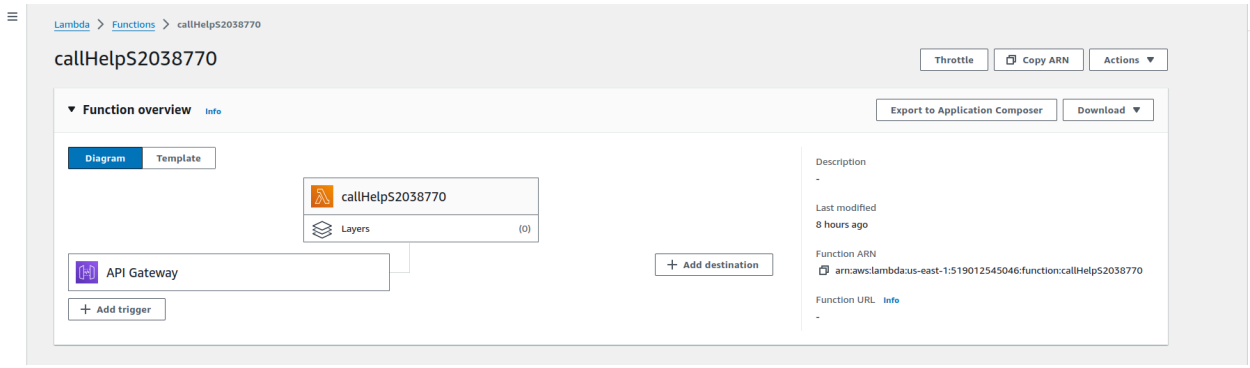

Figure 12. My third lambda with an API gateway trigger

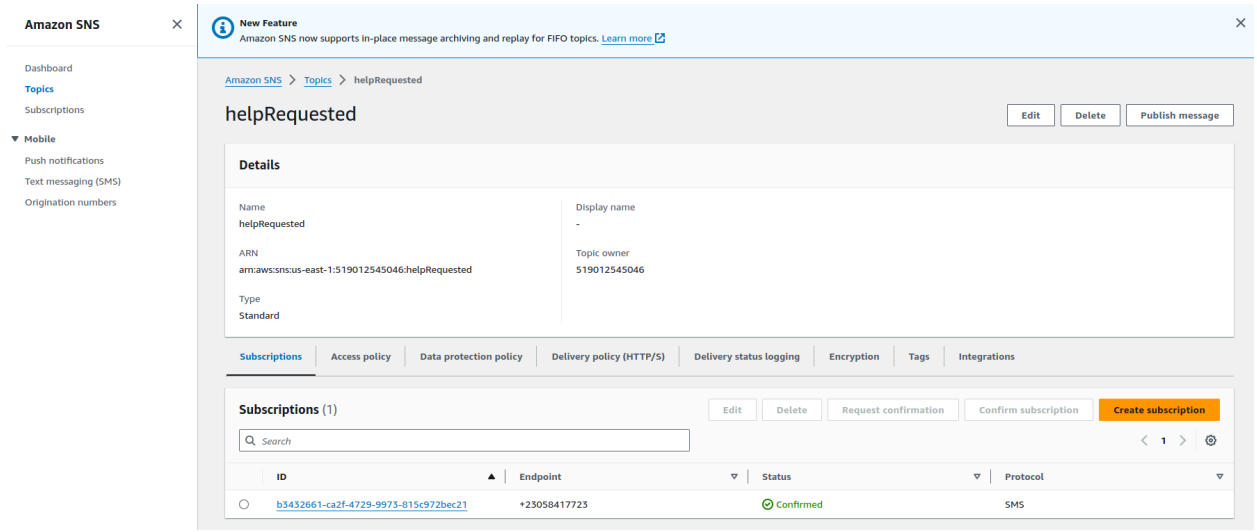

Figure 13. My helpRequested SNS topic

Now my application looks as below:

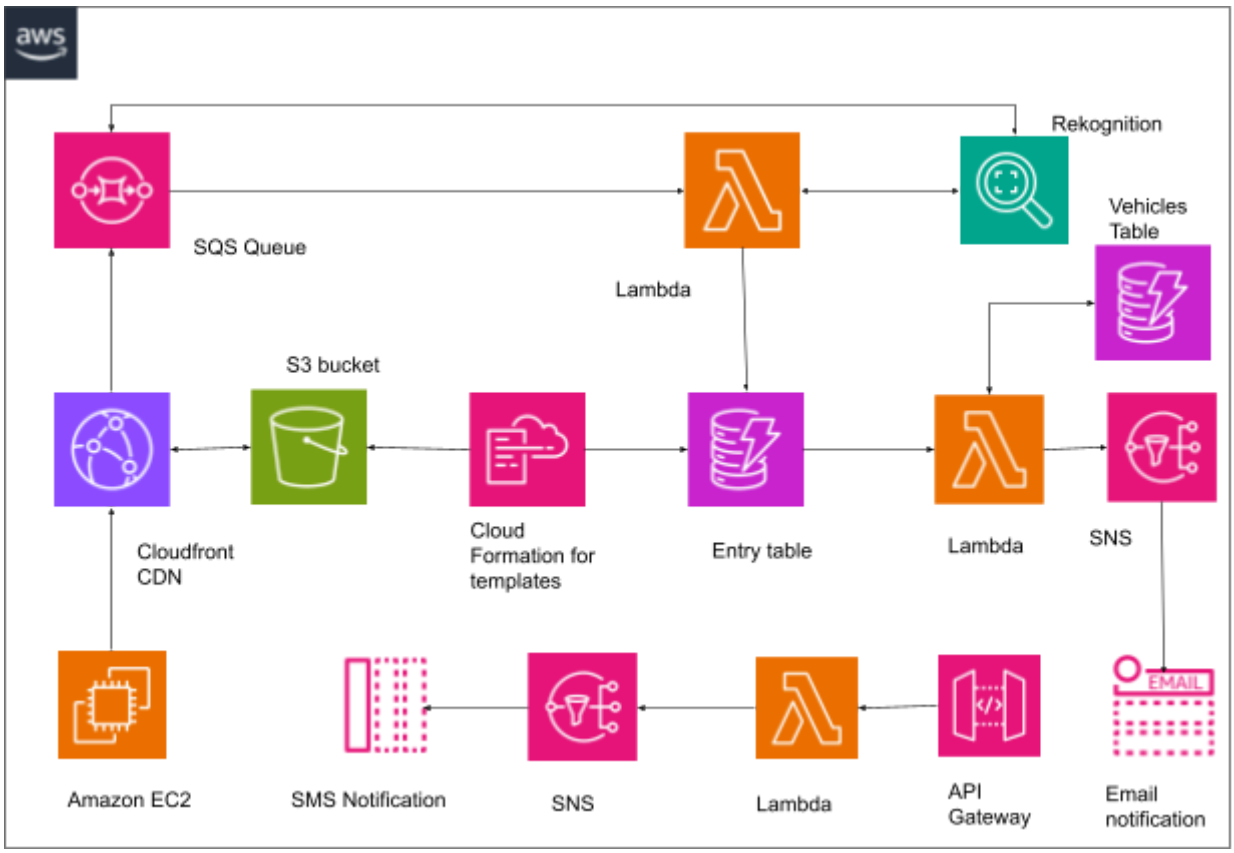

Figure 14. My new modified application architecture

I used the IAM access analyzer to generate my access policies and roles. I then limited them according to the least privilege permissions. This, for instance, allows only my EC2 and SQS to have access to the data stored in my S3 regardless of the roles assigned to either of them.

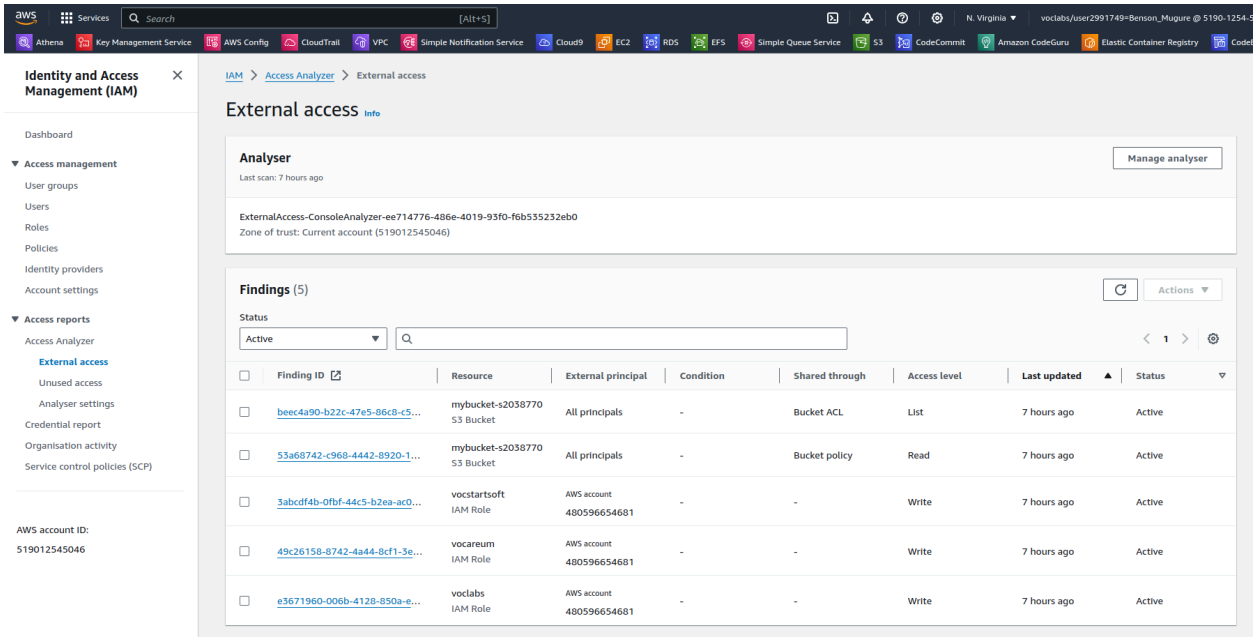

Figure 15. IAM access analyzer results

Overall, by strategically selecting and configuring AWS services tailored to the application's requirements, the approach ensures a cost-effective and secure implementation on the AWS platform. This combination of services enables the application to operate efficiently while adhering to best practices in both cost optimization and security.

Architectural Improvements

The first fundamental change in architecture that I would suggest is adding more EC2 instances for the storage and backup of the images. These can be managed by the use of a load balancer that will allocate the requests based on availability.

I recommend the use of cloudtrail and cloudwatch to monitor the individual services as well as the entire application. I used cloudwatch logs to debug my application but other features of cloudwatch such as dashboards or graphs, events, alerts and grouping of metrics so that you can look at them in a much more cohesive way. We could also set up cron jobs using Cloudwatch events that trigger some other events to ensure that the services are running in a much more predictable way. Cloudtrail, on the other hand, is an audit service that logs who is doing what and when in your application. If we have multiple accounts with different permissions and access levels, cloudtrail would help in reinforcing security as well as debugging.

AWS Eventbridge can also be utilized to trigger services based on rules ranging from simple schedules to complex multi-step ones. A possible scenario where eventbridge would come in handy is, say, in the case of a smart home system. Vehicles come and go but there are vehicles that are only allowed into the house on the weekends, such as if you have a cleaner come over the weekend, or maybe even a plumber. If the concerned worker is, however, detected during the weekday, you want to be notified immediately. Such a case would be a perfect example of when to make a rule for an Eventbridge event. It has the limitation of 5 subscribers to an event, which is significantly lower than an SNS topic which can have a multitude of subscribers in a fan out approach. Even the publishing filtration feature when subscribers only get the specific subtopic that they are interested in is not unique to eventbridge. The advantage of using eventbridge is its numerous integrations with third party services such as PagerDuty, Datadog, Salesforce and even MongoDB.

In terms of Amazon Rekognition, I have two recommendations to make. The first is custom labeling. Some number plates have text in two rows and the service saved them as two distincts texts.

Figure 16. A sample car's number plate

The above text was saved as “6585” and “OC 11”. This can be avoided by custom labeling of number plates. Rekognition can be trained to identify number plates first then identify the text in them. However, this would require a large and diverse dataset which was not available for this coursework. I also suggest that Amazon Rekognition be utilized to identify the faces of the driver. Say, for example, you blacklisted your brother's car from your smart home system because of his temper and his drinking. However, on this particular day, it is your nephew whom you love who is driving the vehicle. There should be a way to check faces as well so that blacklisted individuals don’t simply change vehicles and obtain entry.

I also suggest the use of step functions to orchestrate the workflow of the application. It allows you to coordinate multiple AWS services into serverless workflows, providing a visual representation of your application's logic. AWS Step Functions has strong integration with aws compute services, database services, messaging services, data analytics services and apis. With step functions, you define your workflows as state machines which transform complex code into easy to understand statements and diagrams. It maintains the state of your application during execution including tracking what step execution it's in and storing the data moving between the step functions of your workflow. You can, thus, see where exactly an error occurred, and what parameters were used so that debugging becomes easier and less cumbersome.

Finally, I recommend the integration of the application with AWS IoT Core for the remote management of physical devices. For instance, using IoT Core, the house owner might be able to let a car into their house even when they are not in. This would add great value to the application I made, allowing virtual access to homes even for viewing/tour purposes in case of a real estate agent looking to sell houses and apartments.

Cost Optimization

The below recommendations cater for the scenario in which many images will need to be uploaded and, thus, the application will need to be running for an extended period of time.

Optimizations for EC2

Firstly, it's essential to regularly assess the types and sizes of EC2 instances to ensure they align with the workload requirements. This process, known as right-sizing, involves identifying instances that are either underutilized or overprovisioned and adjusting them accordingly. AWS provides tools like AWS Cost Explorer and AWS Trusted Advisor to assist in this evaluation, helping organizations allocate resources optimally and reduce unnecessary expenses.

Another cost-saving strategy is the utilization of Reserved Instances (RIs) for workloads with predictable usage patterns. RIs offer discounted hourly rates compared to On-Demand instances in exchange for a commitment to usage over a specific term, typically one or three years. By leveraging RIs, organizations can achieve significant cost savings over the long term.

For fault-tolerant and flexible workloads that can tolerate interruptions, Spot Instances provide a cost-effective option. Spot Instances offer spare EC2 capacity at substantially lower prices compared to On-Demand instances. While they may be reclaimed with short notice, they can deliver cost savings of up to 90% for certain workloads.

Implementing Auto Scaling is another key strategy to optimize EC2 costs. Auto Scaling enables automatic adjustments to the number of EC2 instances based on fluctuating demand. By dynamically scaling resources in response to workload changes, organizations can ensure optimal resource utilization while maintaining performance levels.

Finally, continuous monitoring and optimization are essential. AWS CloudWatch can be used to monitor EC2 usage and performance metrics continually. By analyzing usage patterns and performance data, organizations can identify optimization opportunities and fine-tune resource allocations to mitigate unnecessary costs and ensure efficient resource utilization.

Optimizations for S3

One such strategy involves implementing Lifecycle Policies. These policies automate the transition of objects between different storage classes, such as S3 Infrequent Access or S3 Glacier, based on predefined rules. Additionally, lifecycle policies can be configured to automatically delete objects after a specified retention period. This approach helps organizations optimize storage costs while ensuring data accessibility and compliance .

Another important measure is to Enable Server-Side Encryption for data stored in S3 buckets. This involves encrypting data at rest to safeguard its confidentiality and integrity. AWS provides

several encryption options, including SSE-S3, SSE-KMS, and SSE-C, catering to various security requirements and compliance standards. By encrypting data at rest, organizations can mitigate the risk of unauthorized access and protect sensitive information from potential security threats.

Cross-region replication is a critical strategy for enhancing data resilience and disaster recovery capabilities. Through Cross-Region Replication, organizations can replicate objects stored in an S3 bucket to another bucket located in a different AWS region. This replication process ensures redundancy across multiple geographical locations, thereby reducing the risk of data loss. In the event of a regional outage or disaster, organizations can rely on replicated data to maintain business continuity and ensure data availability.

Lastly, organizations can leverage Object Tagging to categorize and organize objects within S3 buckets based on specific attributes or metadata. By assigning tags to objects, organizations can implement more granular access policies, automate lifecycle management, and allocate costs effectively. Object tagging enhances visibility and control over data stored in S3, facilitating better governance and management practices.

Lambda Optimization

Firstly, it's essential to Minimize Execution Time by optimizing the function's code for efficiency. This involves writing streamlined code, minimizing dependencies, and avoiding unnecessary processing steps. By reducing the execution time, organizations can decrease the duration billed by AWS, leading to cost savings.

Another optimization technique is to utilize Reserved Concurrency for Lambda functions. Reserved concurrency allows organizations to allocate dedicated capacity for their functions, ensuring that sufficient resources are always available to handle incoming requests. By reserving concurrency, organizations can maintain consistent performance levels and better control costs, especially for workloads with predictable concurrency requirements.

Proper resource utilization is also crucial for Lambda optimization. Organizations should ensure that Lambda functions are appropriately configured with the optimal memory allocation. Over-provisioning resources can lead to unnecessary costs, while under-provisioning can impact performance. Adjusting the memory allocation based on workload requirements is essential for balancing performance and cost efficiency.

SQS Cost Optimization

Firstly, it's crucial to consider Right-Sizing the queue type based on application requirements. Standard queues are more cost-effective but do not guarantee message ordering, whereas

FIFO (First-In-First-Out) queues ensure message ordering at a higher cost. Choosing the appropriate queue type ensures optimal performance and cost efficiency for your workload.

Utilizing Batch Operations is another effective optimization technique. Batch operations allow sending or receiving multiple messages in a single API call, reducing the number of API requests and lowering costs, particularly for high-volume workloads.

Setting an Optimal Visibility Timeout for messages is essential to ensure they are processed within a reasonable timeframe. Avoiding unnecessarily long timeouts helps minimize queue holding time and associated costs.

Configuring the Message Retention Period according to application requirements is crucial. Avoiding unnecessarily long retention periods helps reduce storage costs associated with storing old messages.

Regularly reviewing and Purging Unused Queues is essential to avoid incurring unnecessary costs. Using AWS Config or other monitoring tools can help identify and manage unused resources effectively.

Optimizing the Polling Frequency of consumers based on application requirements is vital. Adjusting the polling frequency ensures efficient message processing without incurring excessive API requests and costs.

Lastly, utilizing Auto Scaling to dynamically adjust the number of consumers based on queue depth or message throughput helps ensure optimal resource utilization and cost efficiency for varying workloads.

DynamoDB Tables

Firstly, it's crucial to Choose the Right Provisioned Capacity based on your application's read and write throughput requirements. Avoid over-provisioning to minimize costs. DynamoDB offers provisioned capacity mode and on-demand capacity mode. On-demand capacity mode charges you based on the resources your application uses, providing cost savings for unpredictable workloads.

Using Sparse Indexes is another effective optimization technique. Creating sparse global secondary indexes (GSIs) reduces the storage and cost associated with secondary indexes. Only include necessary attributes in indexes to minimize index size and storage costs.

Utilizing DynamoDB Accelerator (DAX) can significantly improve the performance of read-heavy workloads. By reducing the number of read requests to DynamoDB, DAX helps lower DynamoDB costs for read operations.

Implementing Time-To-Live (TTL) allows automatic deletion of expired items from DynamoDB tables, reducing storage costs by removing old data that is no longer needed.

Employing Batch Write Operations like BatchWriteItem efficiently writes multiple items to DynamoDB in a single request, reducing the number of write capacity units consumed and resulting in cost savings.

Optimizing the Data Model is essential to minimize the number of read and write operations required to fulfill application requests. Denormalizing data where necessary reduces the need for expensive join operations.

Regularly Monitoring and Adjusting Capacity based on workload patterns ensures optimal resource allocation. DynamoDB's auto-scaling feature automatically adjusts capacity based on demand, optimizing costs without sacrificing performance.

Using On-Demand Backup and Restore sparingly instead of continuous backups helps save costs. On-demand backups allow creating full backups of tables without the need for continuous backups.

Finally, Leveraging Cost Allocation Tags helps track and allocate costs to specific resources or applications within the organization, identifying cost drivers and optimizing spending accordingly.

SNS optimization

Firstly, employing message filtering policies within SNS topics enables subscribers to receive only relevant messages, thereby reducing unnecessary message dissemination and associated costs. By tailoring message delivery at the topic level, the volume of messages transmitted to subscribers can be minimized, resulting in cost savings.

Secondly, consolidating multiple messages into a single publish operation through SNS batch publish APIs decreases the number of API calls made. This batching mechanism is particularly advantageous for applications with high message volumes, as it reduces the overall number of requests and subsequently mitigates costs.

Choosing the most economical delivery formats according to application requirements is another key consideration. For instance, opting for delivery methods such as HTTP endpoints or AWS Lambda functions over more costly alternatives like email or SMS can lead to significant savings.

Prudent management of message attributes is essential to avoid unnecessary overhead. Minimizing the inclusion of extraneous attributes in SNS messages helps optimize message size and mitigate associated costs. Prioritizing essential attributes required for effective message processing ensures efficient resource utilization.

Furthermore, evaluating the suitability of different endpoint types supported by SNS is crucial. By selecting endpoint types that align with specific messaging needs and volume considerations, organizations can optimize delivery methods while controlling costs.

Regular review of SNS subscriptions to remove redundant or inactive subscriptions helps streamline resource allocation and eliminates unnecessary expenses associated with delivering messages to disengaged subscribers.

Structuring SNS topics hierarchically to group related messages and subscribers can simplify management and reduce the number of topics required. This organizational strategy promotes efficiency and fosters potential cost savings.

Utilizing message attributes for filtering purposes enables subscribers to categorize and filter messages based on predefined criteria, reducing the need for multiple topics. This streamlined approach optimizes resource allocation and minimizes costs associated with managing numerous topics.

Lastly, monitoring SNS usage and associated costs using tools like AWS Cost Explorer or billing reports facilitates ongoing optimization efforts. Analyzing usage patterns and adjusting SNS configurations accordingly ensures that cost-effective practices are consistently maintained while meeting application requirements.

Security Features

Optimizations for EC2

To enhance the security of EC2 instances, it is imperative to prioritize patch management, ensuring that all software components and operating systems are regularly updated to address known vulnerabilities (Smith et al., 2017). Additionally, implementing robust security groups and network ACLs is crucial for enforcing the principle of least privilege and restricting unnecessary traffic both inbound and outbound (Sharma & Garg, 2020). By following this approach, organizations can effectively mitigate the risk of unauthorized access and potential security breaches.

Moreover, encryption plays a pivotal role in safeguarding data confidentiality and integrity. Leveraging AWS Key Management Service (KMS) for encryption at rest and utilizing SSL/TLS protocols for encryption in transit help fortify data protection measures (Ahmed et al., 2019). This ensures that sensitive information remains secure, even if it is stored or transmitted across networks.

Furthermore, adopting robust IAM policies is essential for enforcing access control and adhering to the principle of least privilege. By carefully assigning IAM roles and permissions to EC2 instances, organizations can restrict access to AWS resources and prevent unauthorized actions (Lee et al., 2018).

In addition to proactive measures, effective monitoring and logging mechanisms are critical for detecting and responding to security incidents promptly. Enabling services like CloudTrail for API activity logging and CloudWatch Logs for instance-level monitoring enhances visibility into system activities, enabling organizations to identify and mitigate security threats in real-time (Wang et al., 2021).

Furthermore, implementing instance isolation by segregating instances into different security groups based on their roles and sensitivity levels helps contain potential security breaches and limit the impact of compromised instances (Raj et al., 2020). Additionally, establishing backup and disaster recovery mechanisms using AWS services such as EBS snapshots, RDS backups, and S3 versioning ensures data resilience and facilitates quick recovery in the event of data loss or system failures (Kanwal et al., 2021).

Lastly, embracing security automation through AWS Config Rules, AWS Security Hub, and AWS GuardDuty enables organizations to automate security checks, continuously monitor for security threats, and swiftly respond to security incidents (Son et al., 2019). By integrating these automated security tools into their workflows, organizations can enhance their overall security posture and proactively address emerging threats.

Optimizations for the SQS

To bolster the security of SQS queues, several key optimizations can be implemented to safeguard message integrity and confidentiality. Firstly, enabling encryption at rest through server-side encryption (SSE) is essential to protect messages stored within SQS queues. By leveraging AWS Key Management Service (KMS) managed keys or customer-managed keys for encryption, organizations can ensure that sensitive data remains encrypted and secure (Alizadeh et al., 2018).

Additionally, access control measures should be implemented using IAM policies to regulate access to SQS queues. Adhering to the principle of least privilege, organizations should carefully define permissions, restricting access to only authorized users, roles, and applications (Khan et al., 2020). IAM conditions can further enhance access control by specifying conditions based on factors such as IP ranges or VPC endpoints, thereby reducing the risk of unauthorized access.

Moreover, encryption in transit should be enforced by configuring applications to utilize HTTPS when sending and receiving messages from SQS queues. This ensures that data is encrypted during transmission, safeguarding it against interception and eavesdropping attacks (Narayana et al., 2019).

Dead Letter Queues (DLQs) should be configured to capture messages that cannot be processed successfully, allowing organizations to identify and troubleshoot processing errors without compromising data integrity (Li et al., 2021). This enables timely resolution of issues while preventing message loss or corruption.

Furthermore, implementing message authentication mechanisms such as digital signatures or message integrity checks adds an extra layer of security to SQS messages. By verifying message attributes or contents, organizations can detect and prevent unauthorized tampering or modification of messages during transit (Lee et al., 2021).

Lastly, enabling comprehensive logging and monitoring is essential for maintaining visibility into SQS queue activities and detecting security incidents. By enabling CloudTrail logging for SQS API calls and utilizing CloudWatch Logs and metrics for monitoring queue performance, organizations can proactively identify and respond to potential security threats (Sung et al., 2020). This ensures the continuous security of SQS queues and the data they contain.

Lambda Optimizations

To optimize the security of Lambda functions, several key strategies can be implemented to mitigate risks and safeguard sensitive data. Firstly, adhering to the principle of least privilege is paramount when configuring IAM roles for Lambda functions (Wang et al., 2019). By granting only the permissions necessary for the function to execute its designated tasks, organizations

can minimize the potential attack surface and mitigate the risk of unauthorized access or misuse.

Additionally, secure parameter handling practices should be adopted to prevent the exposure of sensitive information within Lambda function code (Wang et al., 2019). Rather than hard coding` API keys, passwords, or other secrets directly into the code, organizations should utilize secure storage solutions such as environment variables or AWS Secrets Manager. This ensures that sensitive data remains protected and inaccessible to unauthorized parties, reducing the likelihood of data breaches or unauthorized access.

Furthermore, enabling encryption at rest for Lambda function code adds an extra layer of security by encrypting the function's code stored in S3 buckets (Zhang et al., 2021). By leveraging encryption mechanisms, such as AWS Key Management Service (KMS), organizations can safeguard their code from unauthorized access or tampering attempts. This helps maintain the integrity and confidentiality of Lambda functions, even in the event of unauthorized access to underlying storage resources.

DynamoDB Optimizations

Firstly, enabling encryption at rest for DynamoDB tables is essential to safeguard data stored within the tables against unauthorized access or data breaches (Pirzada et al., 2021). By leveraging Server-Side Encryption (SSE) with AWS Key Management Service (KMS), organizations can encrypt data stored in DynamoDB, ensuring that it remains protected even if physical storage media are compromised.

Fine-grained access control is crucial for enforcing least privilege access control and limiting access to DynamoDB tables and operations based on specific user roles (Pirzada et al., 2021). Organizations should implement IAM policies and DynamoDB access control policies to define roles and permissions, ensuring that only authorized users or applications can access and manipulate table data.

Monitoring and logging DynamoDB activities are essential for detecting and responding to security incidents promptly (Pirzada et al., 2021). By enabling DynamoDB Streams to capture changes to table data in real-time and leveraging AWS CloudTrail and CloudWatch Logs, organizations can monitor API calls and table activities for security analysis, auditing, and compliance purposes.

Data validation mechanisms should be implemented to prevent injection attacks and ensure data integrity in DynamoDB (Pirzada et al., 2021). Organizations should validate input data before writing it to DynamoDB tables, mitigating risks associated with malicious inputs that could compromise data integrity or lead to security vulnerabilities.

Regularly backing up DynamoDB tables is essential to protect against data loss due to accidental deletions, corruption, or other unforeseen events (Pirzada et al., 2021).

Organizations should implement a robust backup and restore strategy, leveraging automated backups or on-demand backups to ensure data availability and recoverability.

Lastly, designing DynamoDB data models with security in mind is crucial for securing sensitive information effectively (Pirzada et al., 2021). Organizations should implement encryption, access controls, and data masking techniques to protect sensitive data from unauthorized access or disclosure, ensuring compliance with regulatory requirements and industry standards.

SNS Optimizations

Firstly, encryption in transit is crucial to ensure the confidentiality and integrity of data transmitted via SNS (Chaudhary et al., 2017). By utilizing HTTPS endpoints for publishing and subscribing to SNS topics, organizations can encrypt data transmission over the network, preventing eavesdropping and unauthorized access to sensitive information.

Access control mechanisms should be implemented to restrict access to SNS topics and subscriptions based on the principle of least privilege (Chaudhary et al., 2017). Organizations should apply IAM policies and SNS access control policies to limit who can publish or subscribe to topics, ensuring that only authorized users or applications can access and interact with SNS resources.

Message filtering policies can be used to deliver messages selectively to subscribers based on specific message attributes (Chaudhary et al., 2017). By implementing message filtering, organizations can ensure that only subscribers who meet specified criteria receive relevant messages, preventing unauthorized access to sensitive information by subscribers who do not meet the specified criteria.

Monitoring and auditing SNS activities are essential for detecting and responding to security incidents promptly (Chaudhary et al., 2017). By enabling AWS CloudTrail to log API calls for SNS activities and monitoring CloudWatch Logs for security-related events, organizations can monitor for unauthorized access attempts or suspicious activities, enabling timely intervention and remediation.

Message encryption should be considered, especially when handling sensitive information (Chaudhary et al., 2017). Organizations can implement client-side encryption before publishing messages to SNS topics, adding an extra layer of security by encrypting messages before they are sent to the SNS service, thereby protecting sensitive data from unauthorized access or disclosure.

Finally, subscriber validation mechanisms should be implemented to ensure that subscriber endpoints are authorized and legitimate (Chaudhary et al., 2017). Organizations can implement mechanisms such as token-based authentication or endpoint verification to validate subscriber endpoints, preventing unauthorized access to SNS messages and enhancing overall security.

Optimizations against CSA's top ranked security threats and attacks

Addressing security concerns in cloud computing is paramount to safeguarding applications and data from potential threats and vulnerabilities. According to the Cloud Security Alliance (CSA), several top threats were identified in 2019, including data breaches, misconfiguration, inadequate change control, lack of security architecture, and insufficient identity and access management (CSA, 2019). To optimize application security in light of these concerns, various strategies can be employed.

Firstly, to mitigate the risk of data breaches, encryption mechanisms should be implemented to protect sensitive data both in transit and at rest (Cloud Security Alliance, 2019). By leveraging encryption technologies such as AWS Key Management Service (KMS), organizations can ensure that data remains secure, even if unauthorized access occurs.

Secondly, misconfiguration and inadequate change control can be addressed through regular review and audit of configuration settings, as well as the implementation of automated configuration management tools (Cloud Security Alliance, 2019). By enforcing consistent and secure configuration changes, organizations can minimize the likelihood of security vulnerabilities resulting from misconfigurations.

Moreover, establishing a comprehensive cloud security architecture and strategy is essential to address concerns related to the lack of security architecture and strategy (Cloud Security Alliance, 2019). By incorporating risk assessment, threat modeling, and incident response plans, organizations can effectively mitigate security risks and ensure compliance with security best practices.

Furthermore, organizations should focus on strengthening identity, credentials, access, and key management to mitigate threats such as insufficient identity and access management and account hijacking (Cloud Security Alliance, 2019). Implementing principles of least privilege, multi-factor authentication, and regular credential rotation can enhance security posture and reduce the risk of unauthorized access.

Additionally, addressing insider threats requires monitoring user activities and behaviors using tools such as AWS CloudTrail and CloudWatch Events (Cloud Security Alliance, 2019). By detecting anomalous behavior indicative of insider threats, organizations can take proactive measures to prevent insider attacks and protect sensitive data.

Furthermore, securing interfaces and APIs is crucial to mitigate the risk of insecure interfaces and APIs (Cloud Security Alliance, 2019). Implementing authentication, authorization, encryption, and input validation mechanisms can help prevent unauthorized access and data breaches resulting from insecure APIs.

Moreover, organizations should strengthen the control plane by implementing robust authentication and authorization mechanisms (Cloud Security Alliance, 2019). By monitoring control plane activities and responding to unauthorized changes or access attempts, organizations can enhance security and maintain control over their cloud environment.

Additionally, implementing backup, disaster recovery, and high availability strategies is essential to mitigate the risk of infrastructure failures (Cloud Security Alliance, 2019). By ensuring data protection and resilience, organizations can minimize the impact of metastructure and applistructure failures on their applications and data.

Furthermore, gaining visibility into cloud usage and activities is critical to addressing concerns related to limited cloud usage visibility and abuse of cloud services (Cloud Security Alliance, 2019). By leveraging AWS services such as CloudTrail, AWS Config, and CloudWatch, organizations can gain insights into resource usage, configurations, and operational activities, enabling them to detect and respond to security incidents effectively.

Application Testing

The tests cover the functionality and extended features of the application. Please have a look at them below:

Table 1. Application tests and test results

Test	Expected Result	Actual Result
EC2 successful Creation	The instance is created successfully and can be seen from the ec2 console	The instance is created successfully as per the message output when the code is run and it is visible from the ec2 console. See Appendix 1
S3 bucket successful creation	The bucket is created successfully and can be seen from the ec2 console	The bucket is created successfully as per the message output when the code is run and it is visible from the ec2 console. See Appendix 2
Creation of the SQS Queue using a CloudFormation Template and Linked to a Lambda	Evidence of the linking of the Queue to the Lambda	The SQS Queue was created and it was linked to an already created Lambda. See Appendix 3
The boto3 code uploads the images in 30 second intervals	There is evidence of the files being uploaded and the s3 bucket has the images on inspection	The python3 code displays success messages on uploading each of the images and the s3 bucket was found to have the images, See Appendix 4
The DynamoDB database should have an entry table with only a single partition (primary) key as the image name to store the image information.	An entry table exists and its partition key is the image name	Entry table was seen with its partition key being the image name. See Appendix 5
From the Label and Text detection response, the Lambda function must extract relevant information and save the results for the relevant vehicle labels with the confidence scores in the entry DynamoDB table.	The entry table populated by data from the lambda	The entry table with the data from the lambda including the text and labels together with their confidence levels. See Appendix 6
The vehicle table will contain a column that will help determine if the vehicle is one which is whitelisted or blacklisted by the	The vehicles table should have a blacklisted column	The vehicles table has a blacklisted column which is of boolean type. See Appendix 7

security facility.		
For vehicles identified as blacklisted or for which no records are found, the Lambda function must immediately notify a security officer. Since the instance is a test instance, the notification can be in the form of an email that is sent with the relevant message. You may use any email address for implementation/testing but should use john.doe@example.com for the completed coursework code and report submission.	An email notification is sent where the vehicle is blacklisted or where there is no data on the vehicle stored in the vehicle table	An email was sent where the vehicle is blacklisted or where there is no data on the vehicle stored in the vehicle table. See Appendix 8 and Appendix 9
The email has a means to see the concerned vehicle	An s3 link to the concerned object (image) is sent in the email	An s3 link to the image is observed in the emails sent. See Appendix 8 and Appendix 9
The email recipient has an option of seeking assistance	A link to request emergency assistance is included in the email notification and the link works	A link to request for emergency services was included in the email. See Appendix 8 and Appendix 9
The link in the email to seeks additional assistance works	An SMS notification is sent to the number specified and emergency contacts are displayed	An SMS was sent to the concerned number and emergency contacts were displayed. See Appendix 10 and Appendix 11

Conclusion

In conclusion, our journey toward optimizing our application for both security and cost efficiency has been marked by a meticulous examination of the challenges and opportunities inherent in cloud computing. Through a comprehensive approach that addresses prominent security threats while also maximizing cost savings, we have not only fortified our application against potential risks but also enhanced its operational efficiency and value proposition.

The Cloud Security Alliance (CSA) has outlined a range of top cloud computing threats and security concerns, each presenting unique challenges to organizations leveraging cloud services. By meticulously analyzing and addressing these threats, we have taken proactive measures to safeguard our application's integrity, confidentiality, and availability. From mitigating data breaches and implementing robust access controls to fortifying our application against insider threats and API vulnerabilities, our security enhancements align with industry best practices and regulatory requirements. Furthermore, our adoption of encryption mechanisms, rigorous IAM policies, and continuous monitoring tools underscores our commitment to maintaining a secure and resilient cloud environment.

In parallel, our efforts to optimize costs have yielded significant benefits in terms of operational efficiency and resource utilization. By leveraging AWS cost optimization strategies such as right-sizing instances, utilizing reserved instances, and optimizing storage usage, we have achieved substantial cost savings without compromising on performance or security. Moreover, our proactive approach to cost management, coupled with ongoing monitoring and evaluation, ensures that we can adapt to changing usage patterns and requirements while maximizing the value derived from our cloud investments.

The synergy between security and cost optimization underscores our commitment to delivering a solution that not only meets the highest standards of security but also provides tangible value to our stakeholders. By striking a balance between security, cost, and performance considerations, we have created a robust and cost-effective cloud environment that enhances our application's resilience and competitiveness. Additionally, our dedication to continuous improvement and innovation ensures that we remain at the forefront of emerging security trends and cost optimization practices, enabling us to adapt and thrive in an ever-evolving digital landscape.

References

1. Amazon Web Services. (2023a, March 22). Amazon SQS Limits. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-quotas.html
2. Amazon Web Services. (2023b, March 22). AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
3. Amazon Web Services. (2023c, March 29). Amazon Rekognition Batch Operations. https://aws.amazon.com/blogs/machine-learning/batch-image-processing-with-amazon-rekognition-custom-labels/
4. Amazon Web Services. (2023d, March 28). DynamoDB Provisioned Throughput. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ProvisionedThroughput.html
5. Aws. (2023, March 9). Amazon SQS fan-out pattern. https://aws.amazon.com/getting-started/hands-on/send-fanout-event-notifications/
6. Chaudhary, S., Yadav, S., Singh, A., & Agarwal, P. (2017). A Study on Security Implementation in Amazon Web Services. International Journal of Computer Applications, 159(11), 6-10. DOI: 10.5120/ijca2017915255
7. Choudhury, D. (2019). "Enhancing Security in Amazon Web Services (AWS) Using Identity and Access Management (IAM)." International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), 4(5), 242-247.
8. Cloud Security Alliance. (2019). Top Threats to Cloud Computing: Egregious Eleven Deep Dive. Retrieved from https://downloads.cloudsecurityalliance.org/assets/research/top-threats
9. Garg, S., Srivastava, A., & Kumar, A. (2020). Security and optimization of AWS services: A review. In 2020 International Conference on Computer Science, Engineering and Applications (ICCSEA) (pp. 1-6). IEEE.
10. Haque, M. A., & Hasan, M. M. (2018). "Security Analysis of Amazon Simple Notification Service (SNS) in Cloud Computing." International Journal of Computer Applications (0975 – 8887), 179(43), 18-22.
11. Hossain, M. A., & Sohraby, K. (2020). "A Comprehensive Study on Security Considerations of Amazon Simple Notification Service (SNS) in Cloud Computing Environments." International Journal of Cloud Applications and Computing (IJCAC), 10(2), 1-14.
12. Kaur, P., & Kaur, J. (2020). "An Analytical Study of Security Features in Amazon Web Services (AWS)." International Journal of Advanced Trends in Computer Science and Engineering (IJATCSE), 9(1.2), 470-476.
13. Krishnan, R., & Chandrasekar, S. (2020). "Security Issues and Solutions in Cloud-Based Message Queue Systems: A Review." International Journal of Engineering Research & Technology (IJERT), 9(3), 47-53.
14. Kumar, A., Garg, S., & Bala, P. (2018). "A Study of Data Security in Cloud Computing." International Conference on Computing, Communication and Automation (ICCCA). IEEE.

15. Noura, M., Atiquzzaman, M., & Gaedke, M. (2018). "Security and Privacy in Fog and Edge Computing: Challenges and Solutions." IEEE Internet Computing, 22(3), 67-75.
16. Pirzada, A. A., Ahmed, N., Altaf, M., & Alghathbar, K. (2021). Security in Amazon Web Services: A Review. In 2021 IEEE International Conference on Electro Information Technology (EIT) (pp. 1-6). IEEE.
17. Raju, R., & Kumar, A. (2020). "Security Enhancements in Amazon DynamoDB." International Journal of Advanced Computer Science and Applications (IJACSA), 11(5), 335-342.
18. Ravi, D., & Rajaram, S. (2019). "Security Measures in DynamoDB - A Review." International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), 4(5), 182-186.
19. Sharma, S., & Tyagi, N. (2018). "A Review on Security Issues and Measures in Cloud Computing Using Amazon Web Services." International Journal of Engineering & Technology (UAE), 7(4.38), 169-173.

Appendix

Appendix 1: Created EC2

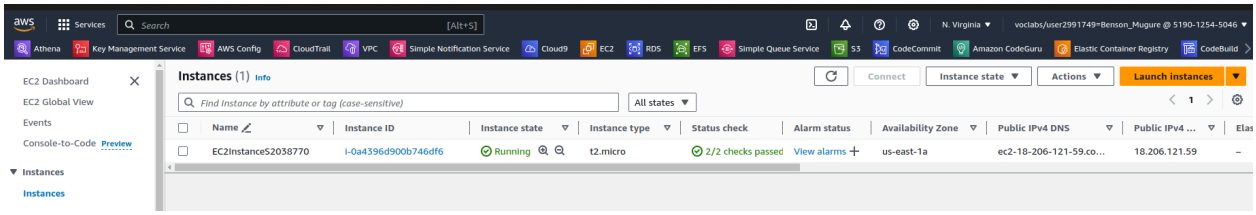

Figure 17. My EC2 from the AWS Console

Appendix 2: S3 Bucket Creation

Creating s3 bucket using boto3 python code:

```
eee_W_2753628@runweb119143:~$ python3 s3.py
/usr/lib/python3.7/site-packages/boto3/compat.py:82: PythonDeprecationWarning: Boto3 will no longer support Python 3.7 starting December 13,
2023. To continue receiving service updates, bug fixes, and security updates please upgrade to Python 3.8 or later. More information can be f
ound here: https://aws.amazon.com/blogs/developer/python-support-policy-updates-for-aws-sdks-and-tools/
  warnings.warn(warning, PythonDeprecationWarning)
Successfully created S3 bucket with name: mybucket-s2038770
Versioning enabled on S3 bucket: mybucket-s2038770
Server-side encryption enabled on S3 bucket: mybucket-s2038770
Lifecycle policy configured for S3 bucket
eee_W_2753628@runweb119143:~$
```

Figure 18. Bucket creation via code

Created s3 bucket:

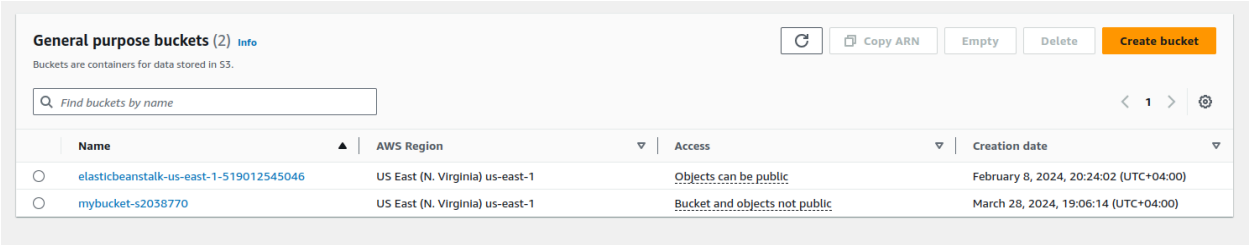

Figure 19. My s3 bucket on AWS Console

Appendix 3: SQS Queue linked to Lambda

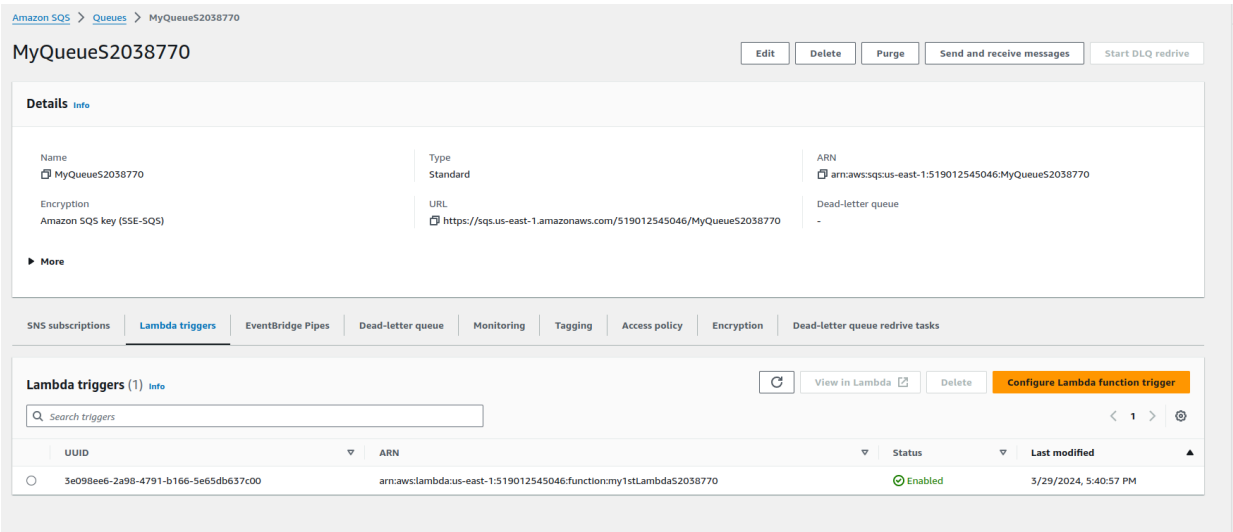

Figure 20. My SQS Queue from AWS Console

Appendix 4: Image upload from EC2 to s3

```
[ec2-user@ip-172-31-89-245 ~]$ python3 image_upload.py
Uploaded IMG_20240212_125129.jpg to S3
Uploaded IMG_20240212_151830.jpg to S3
Uploaded IMG_20240212_151946.jpg to S3
Uploaded IMG_20240212_152041.jpg to S3
[ec2-user@ip-172-31-89-245 ~]$
```

Figure 21. Image upload from EC2 to s3

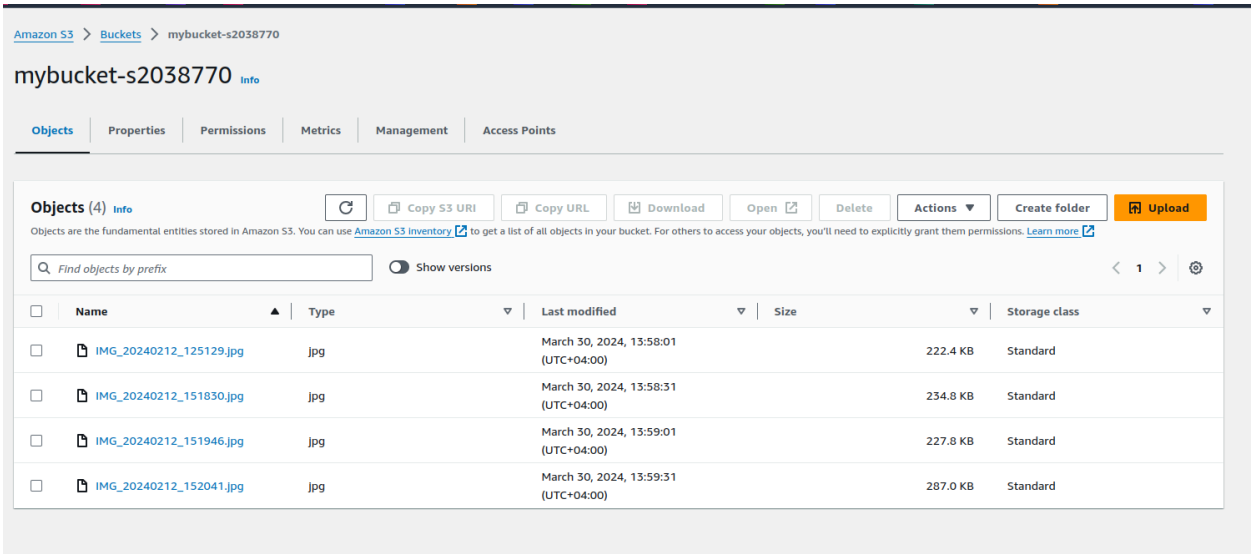

Figure 22. Populate s3 bucket as viewed from AWS Console

Appendix 5: Entry table and its partition key

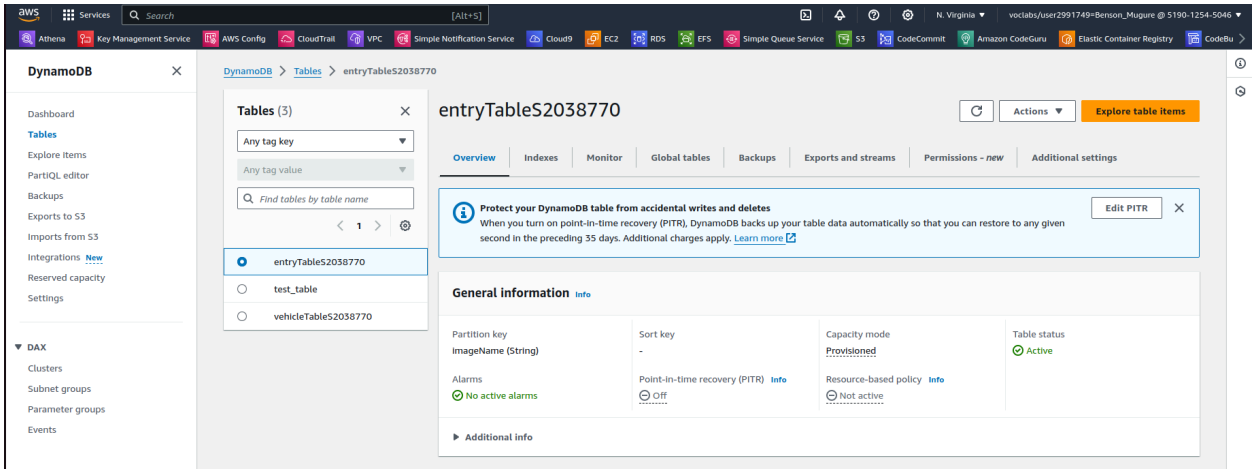

Figure 23. Entry table and its partition key

Appendix 6: Populated Entry table

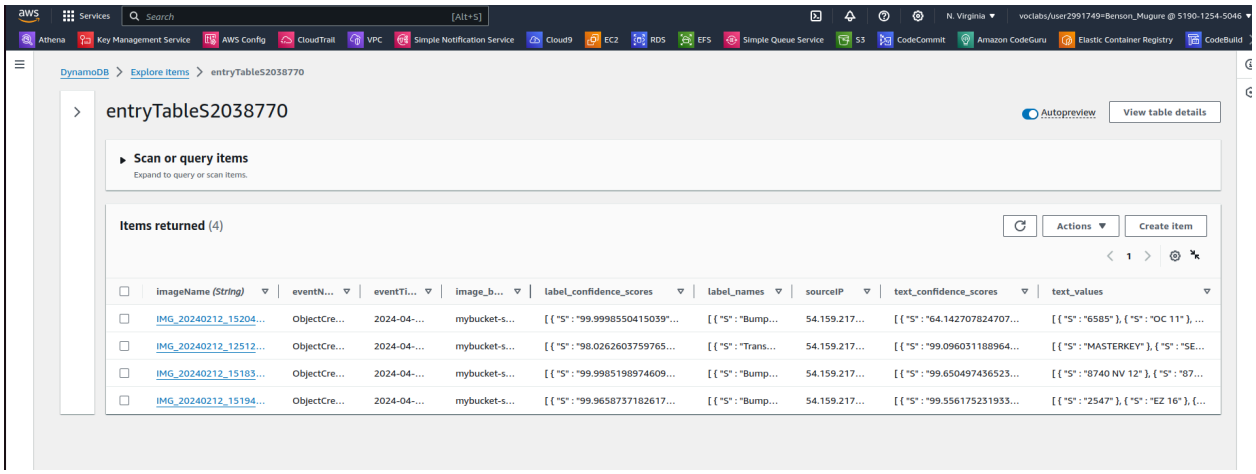

Figure 24. Populated entry table

Appendix 7: Vehicle's table with the blacklisted column

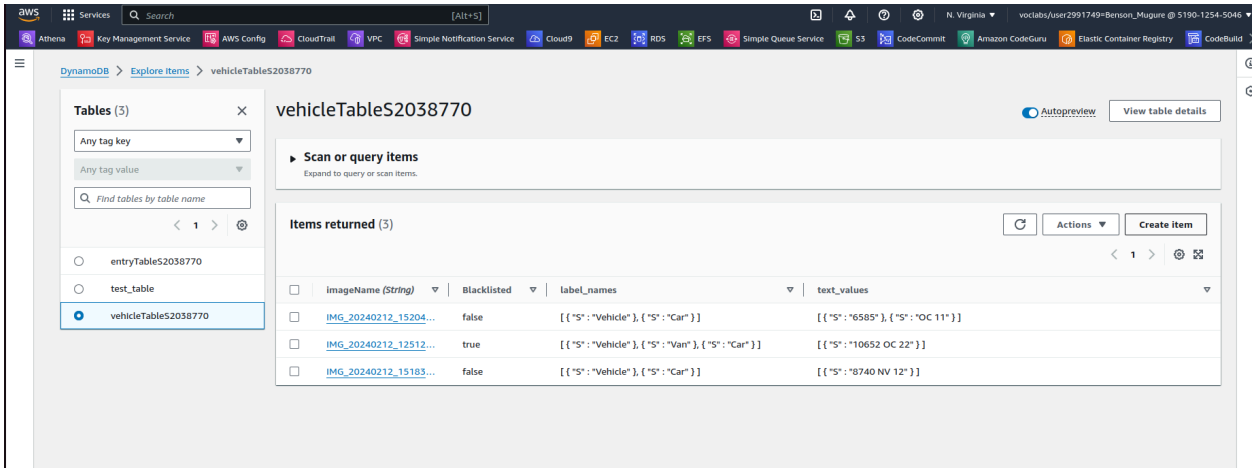

Figure 25. Vehicle's table with the blacklisted column

Appendix 8: Blacklisted vehicle email notification

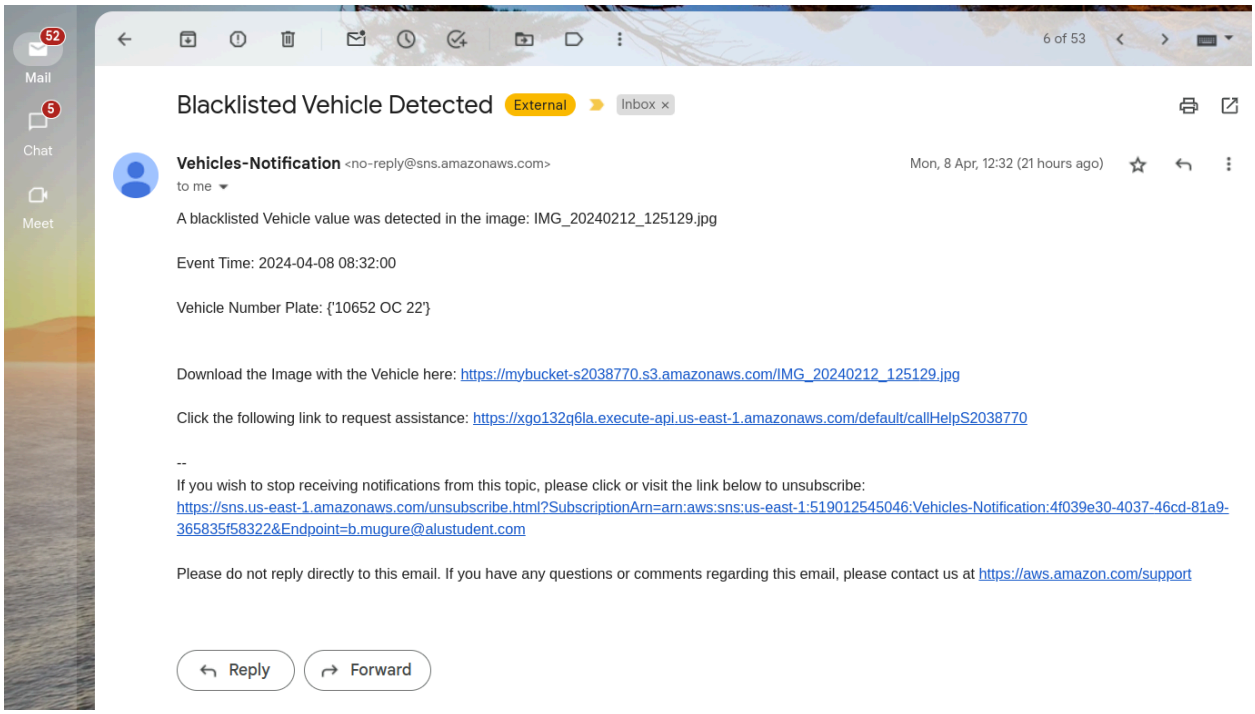

Figure 26. Blacklisted vehicle email notification

Appendix 9: Unidentified vehicle email notification

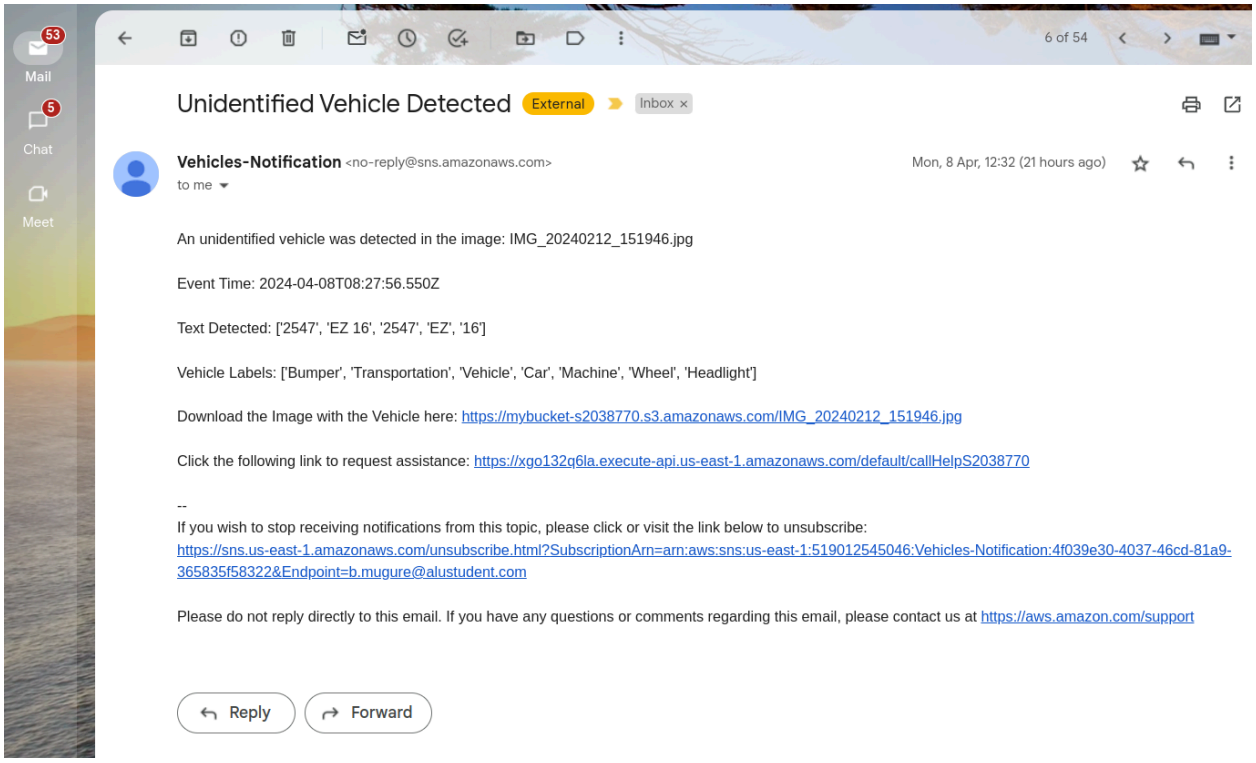

Figure 27. Unidentified vehicle email notification

Appendix 10: SMS notification

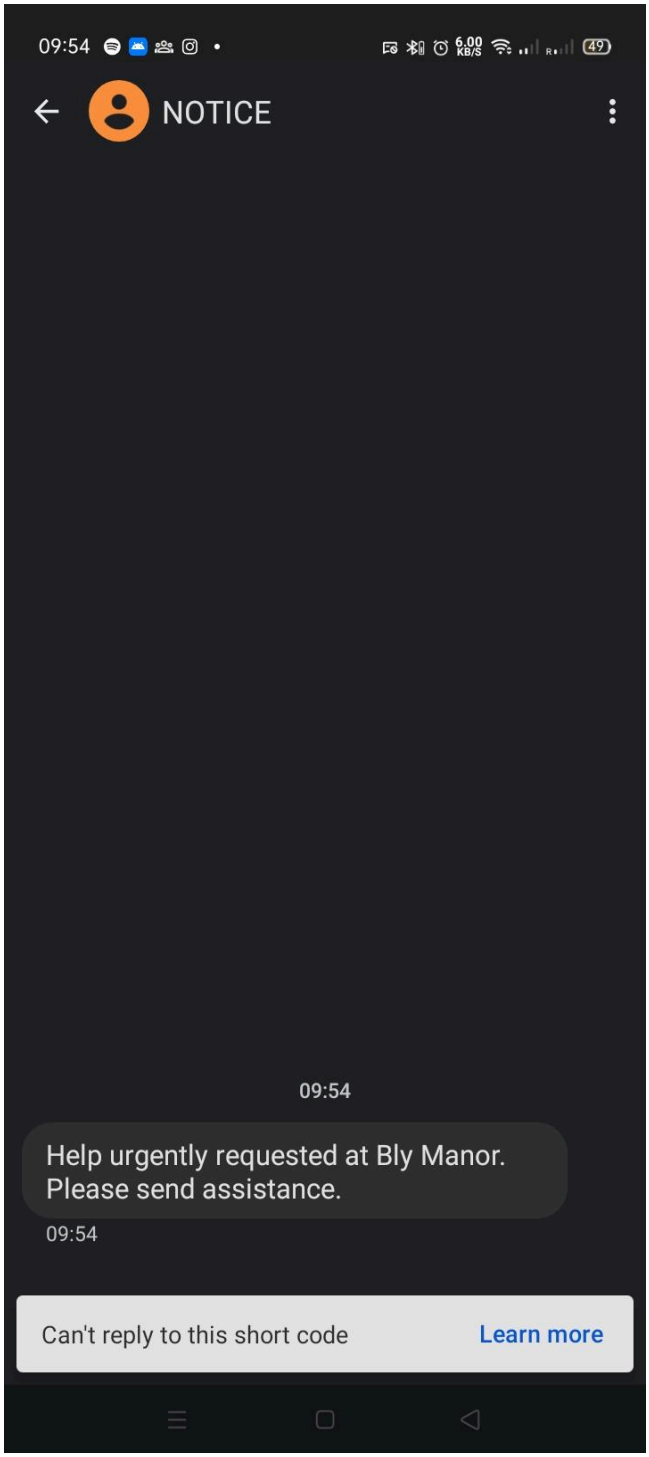

Figure 28. SMS notification requesting for help

Appendix 11: Emergency Contacts

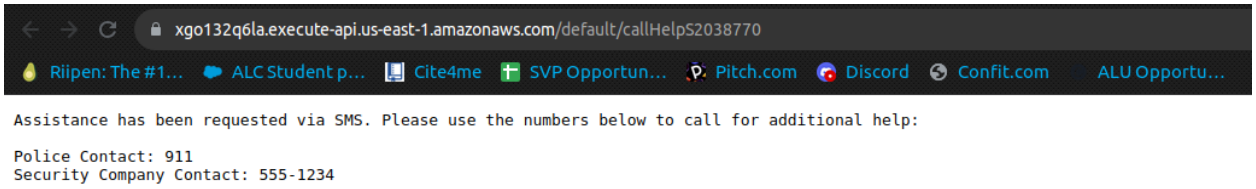

Figure 29. Emergency contacts displayed

Appendix 12: Link to my zipped file

Link: https://drive.google.com/file/d/1yDH6q2qgR7I4DLyDQ9pIkbufPfUG0fuP/view?usp=sharing