School Of Computing

ST505

ESDE (Cloud Computing)

WEB APPLICATION CLOUD COMPUTING REPORT

AY2021S2 2A/23

Team Members:

Sara Golkaran (P1935657)

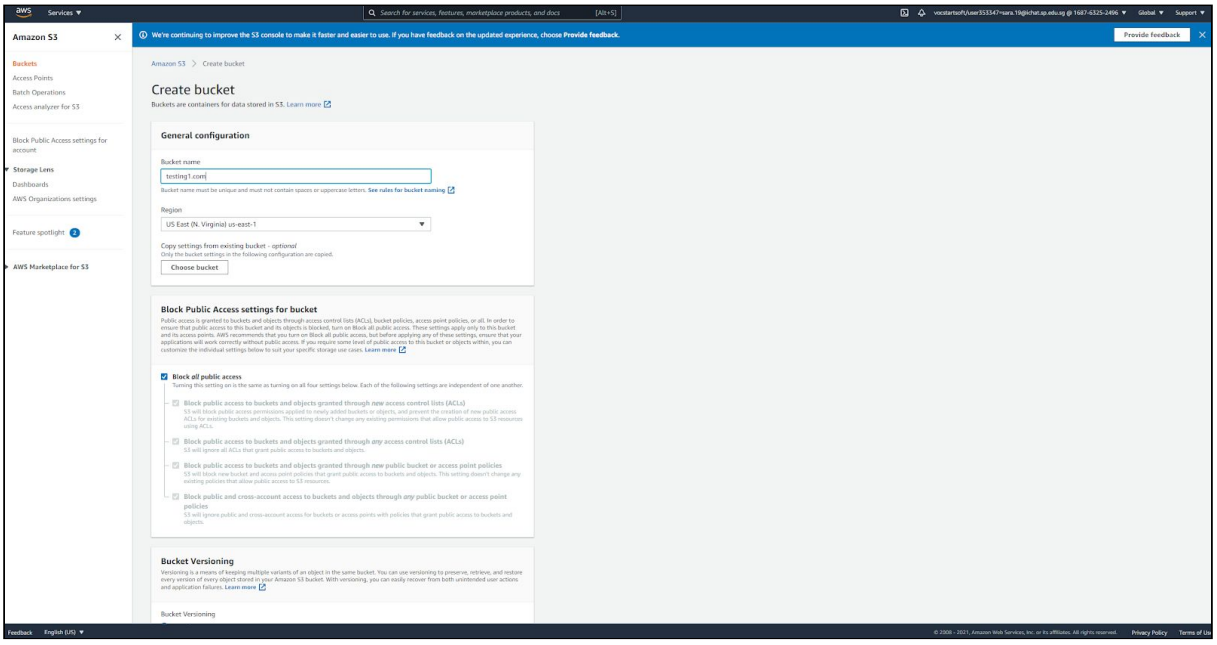
Yap Min Ru Vicki (P1904556)

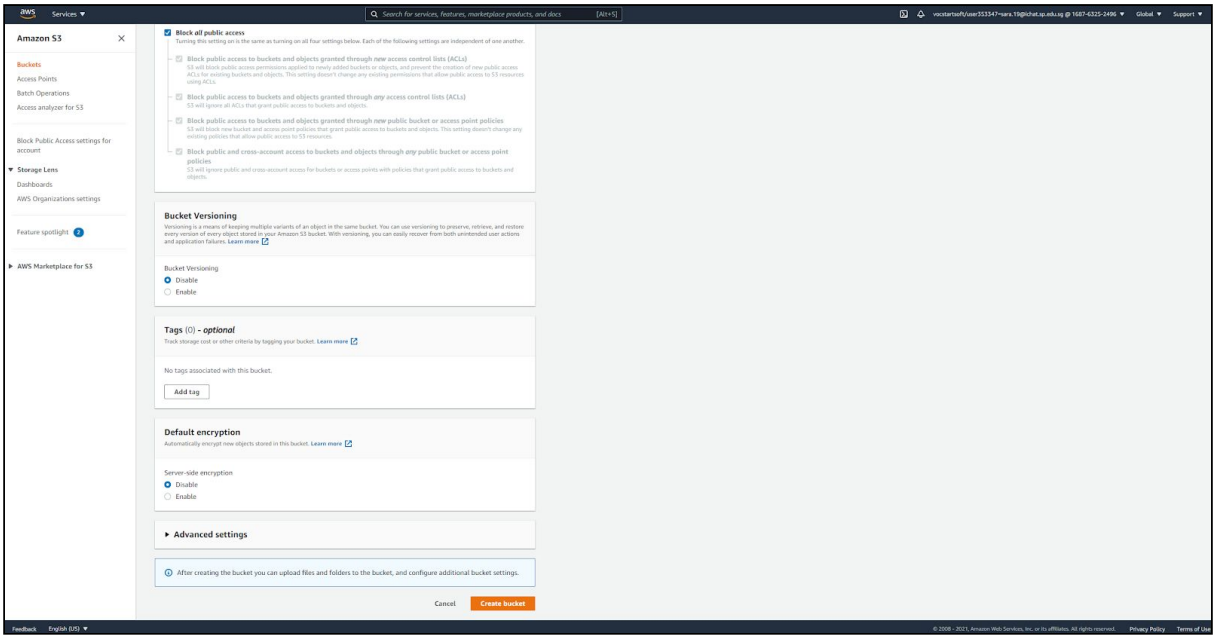
GitHub repository link: https://github.com/saragolkaran/ESDE-Cloud-Computing-CA2-

1. Deploy Existing Web Application on AWS (Static)

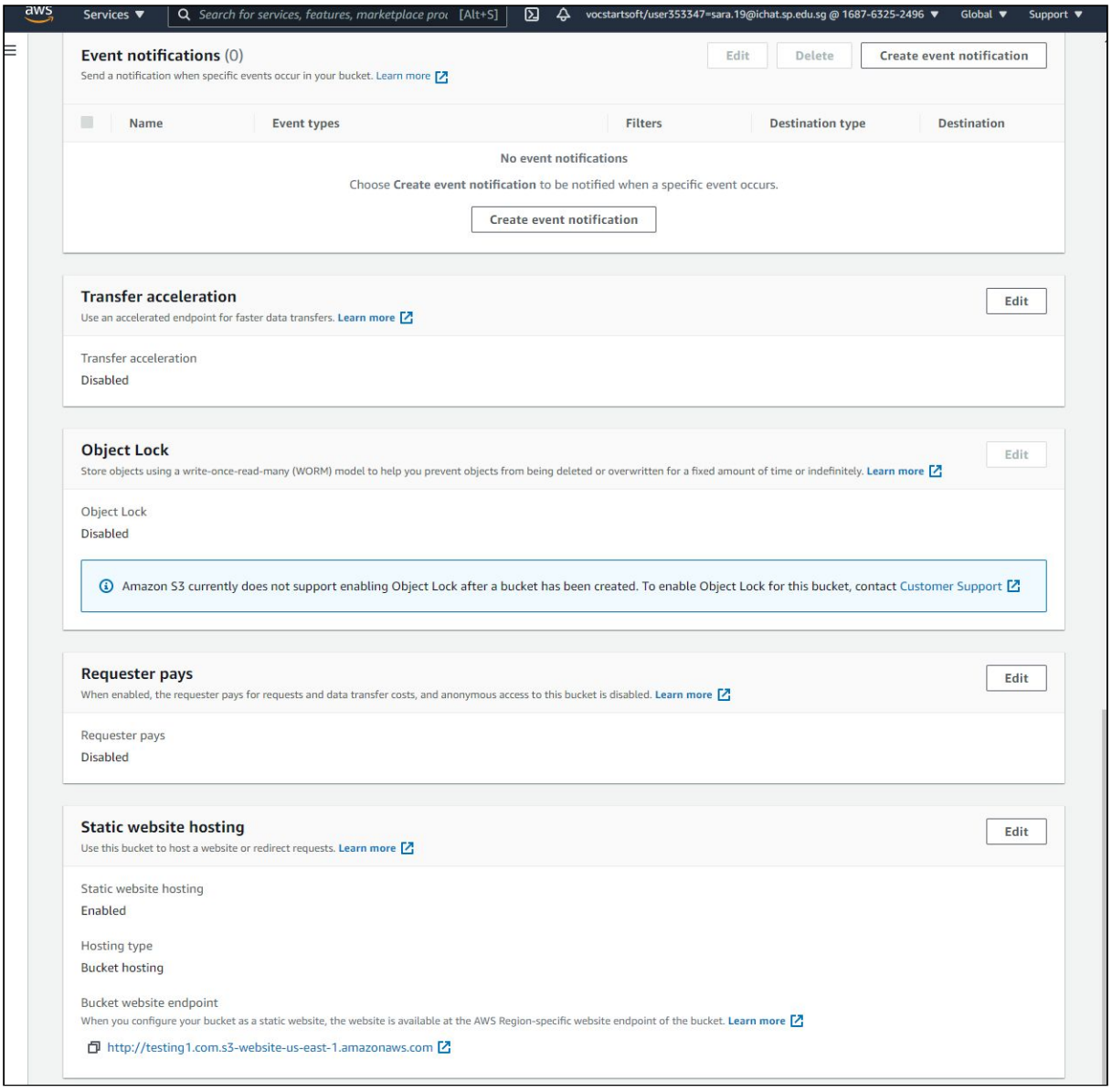
In this section, we will be showing a step-by-step approach on how to create a S3 bucket, adding a bucket policy to make its content publicly available and the deployment process of our existing web application using static API.

Step 1: After logging into our AWS account, we navigated to the S3 dashboard and clicked on the “Create bucket” button. We initially applied the default settings for our bucket configuration which includes blocking of all public access and disabled bucket versioning before naming it “testing1.com”.



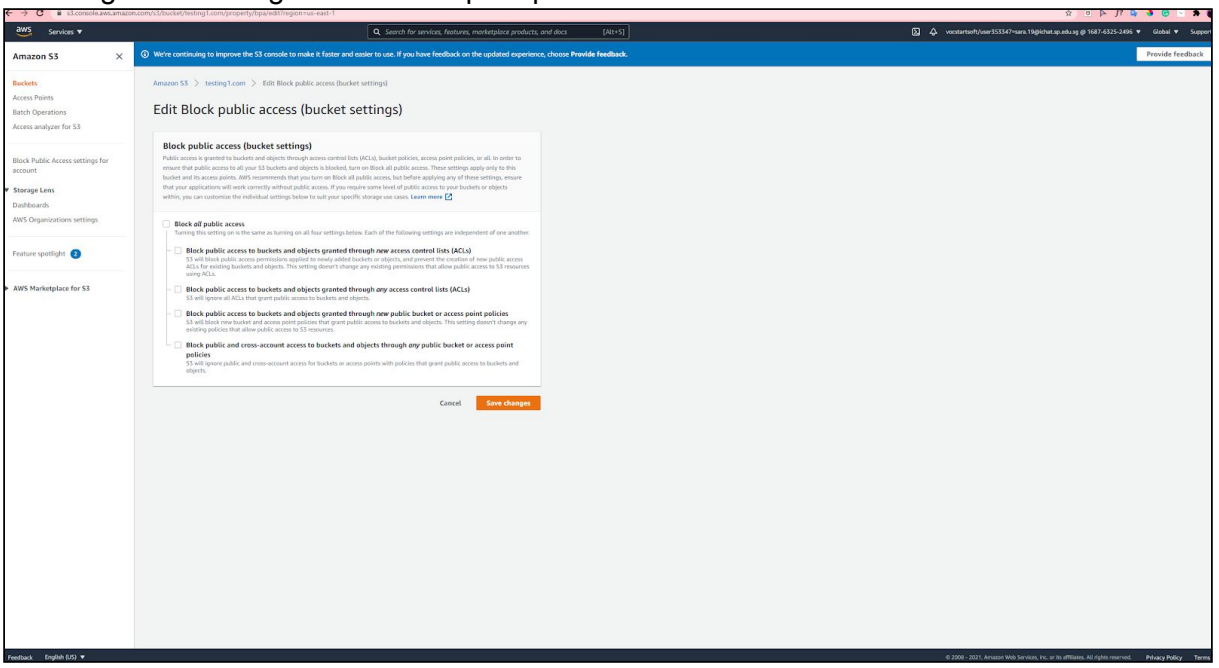


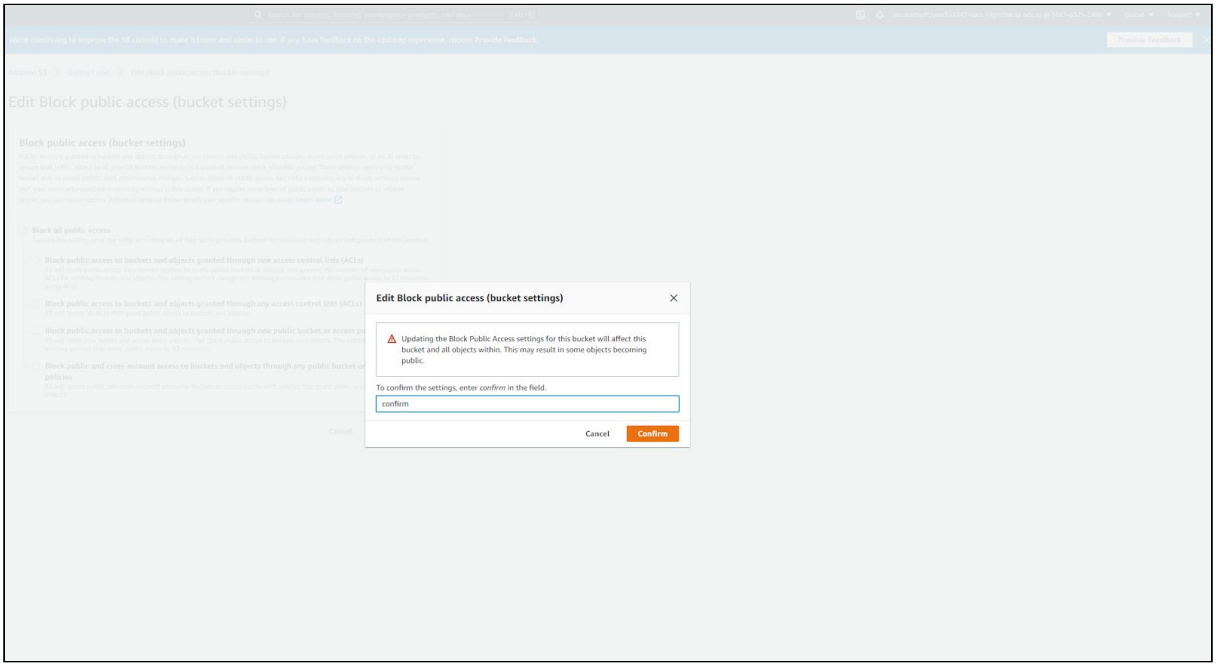
Step 2: After we have created a bucket, we can enable static website hosting for our bucket. We navigated to “Static website hosting” and clicked on the “edit” button to change to “enabled”.



Step 3: We clicked on the link at the bottom of the previous page and encountered a 403 forbidden error that prevents us from accessing the bucket website endpoint. This means that our permission settings need to be changed to allow universally everyone to access.

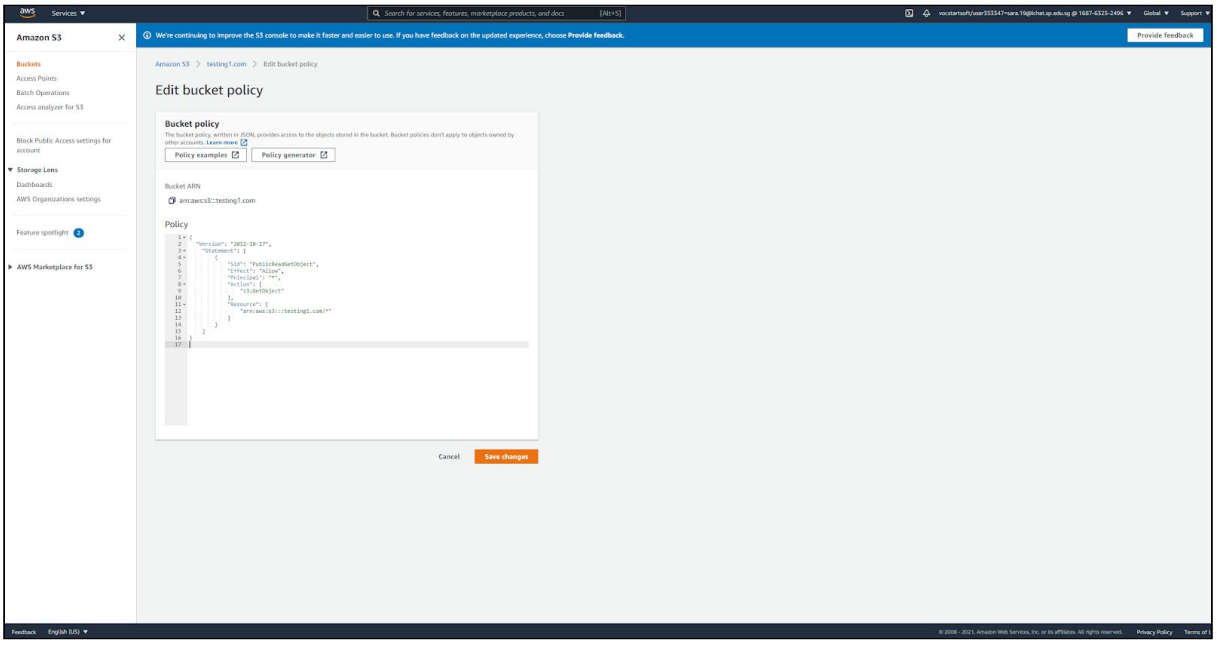
Step 4: We cleared all the ticked checkboxes under “Block all public access” before choosing “save changes”. We were prompted to enter “confirm” to authorize the edit.



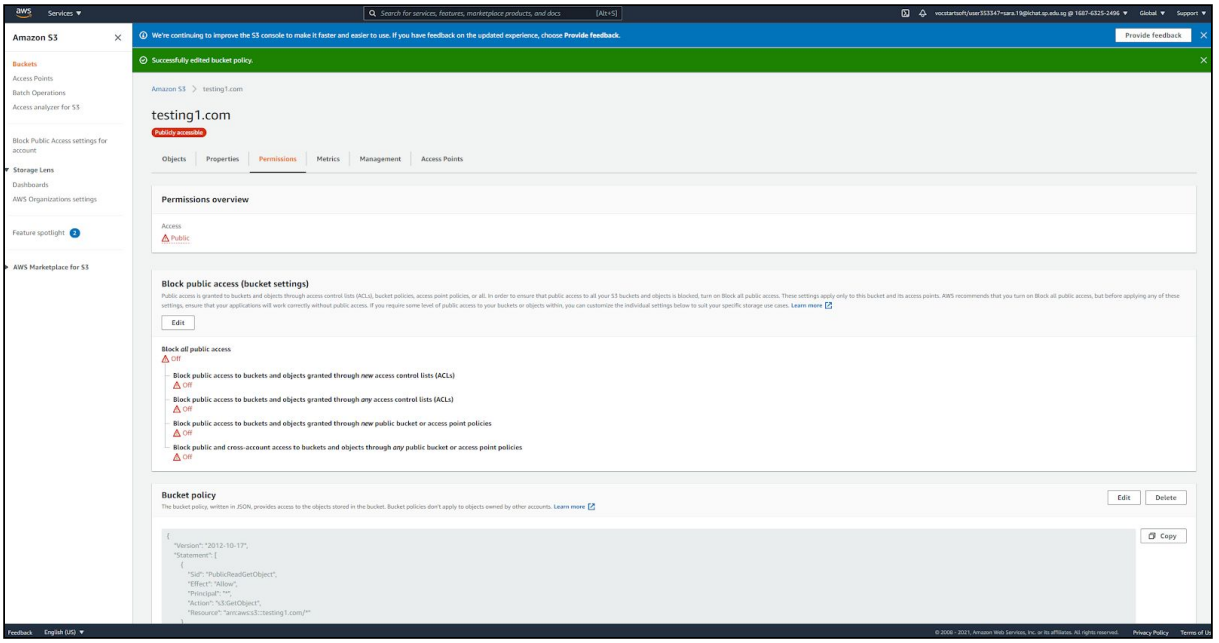


Step 5: In the interests of availability which is one of the top 5 business impacts in Cloud Computing, we also decided to add a bucket policy to grant public read access to your bucket. With reference to the following bucket policy, we modified the *example.com* to our *testing1.com* and pasted the policy within the Bucket policy editor.

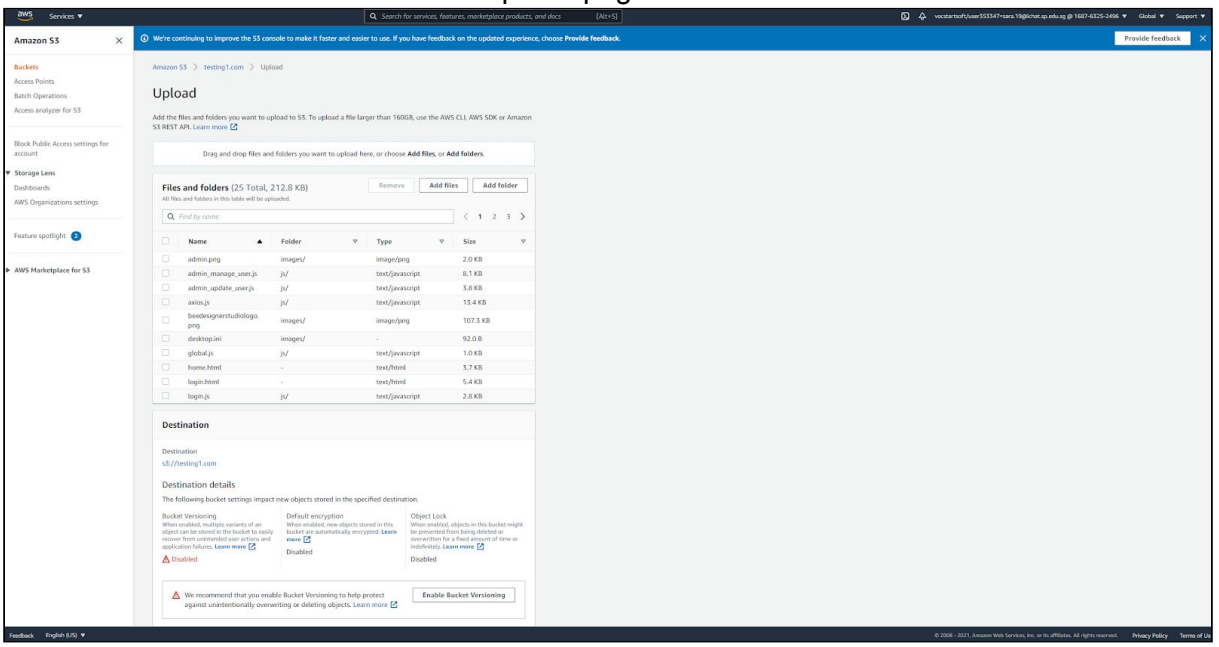
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::example.com/*"
            ]
        }
    ]
}
```

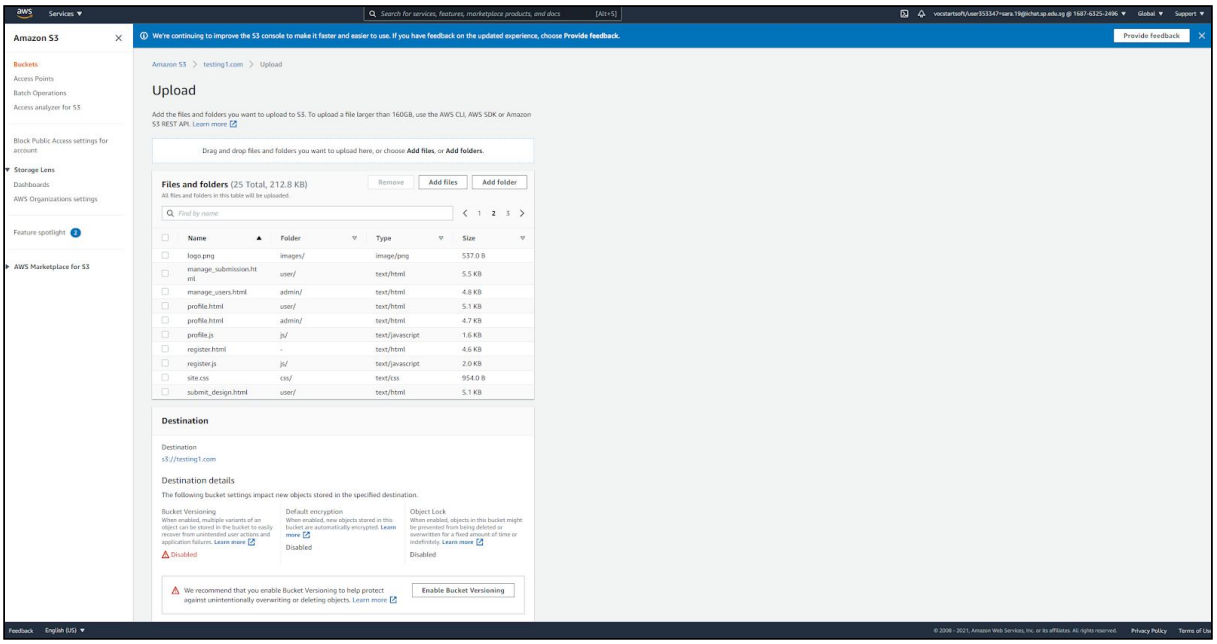


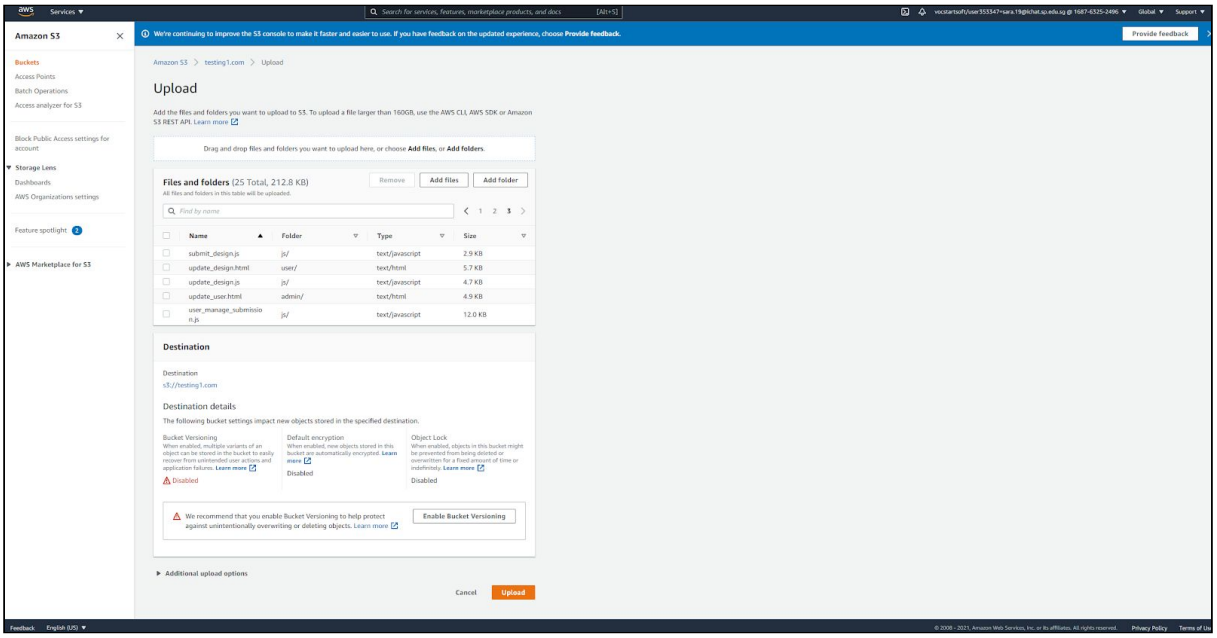
Step 6: We were informed that the editing of the bucket policy is successful and that the contents of *testing1.com* is now “publicly available”.

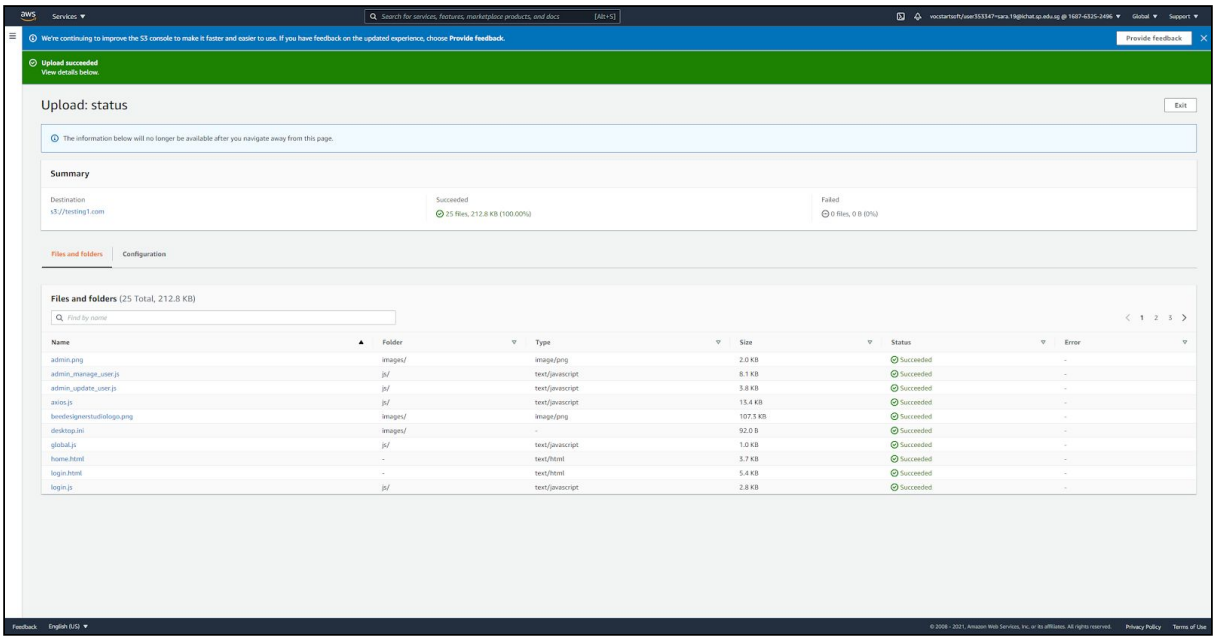


Step 7: We managed to upload the individual HTML and Javascript files within the frontend folder into the S3 bucket which totals up to 3 pages as seen below.

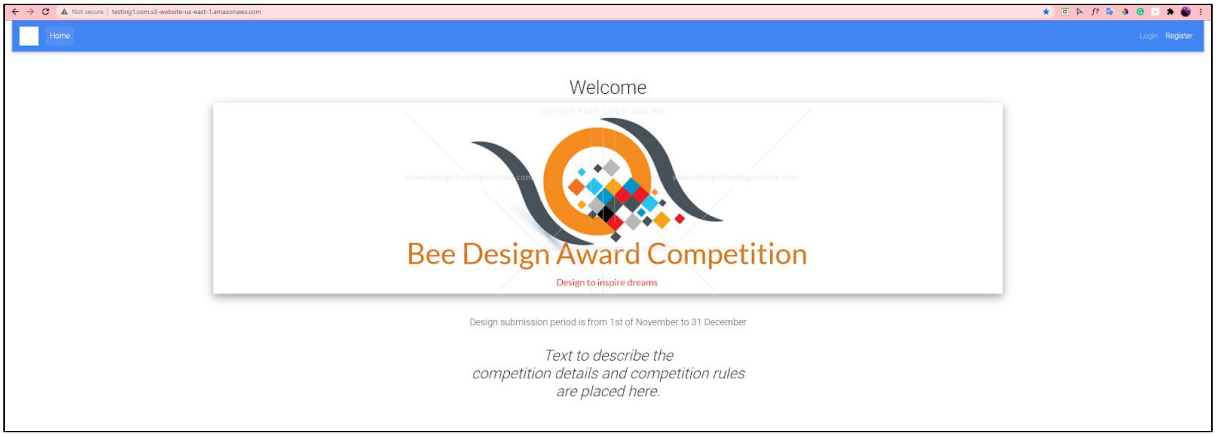



Amazon S3
Buckets
Access Points
Batch Operations
Access analyzer for S3
Block Public Access settings for account
Storage Lens
Dashboards
AWS Organizations settings
Feature spotlight
AWS Marketplace for S3
We're continuing to improve the S3 console to make it faster and easier to use. If you have feedback on the updated experience, choose Provide feedback.
Provide feedback
Amazon S3 > testing1.com > Upload
Upload
Add the files and folders you want to upload to S3. To upload a file larger than 160GB, use the AWS CLI, AWS SDK or Amazon S3 REST API. Learn more
Drag and drop files and folders you want to upload here, or choose Add files, or Add folders.
Files and folders (25 Total, 212.8 KB)
All files and folders in this table will be uploaded.
Remove
Add files
Add folder
Find by name
Name
Folder
Type
Size
logo.png images/ image/png 537.0 B
manage_submission.html user/ text/html 5.5 KB
manage_users.html admin/ text/html 4.8 KB
profile.html user/ text/html 5.1 KB
profile.html admin/ text/html 4.7 KB
profile.js js/ text/javascript 1.6 KB
register.html - text/html 4.6 KB
register.js js/ text/javascript 2.0 KB
site.css css/ text/css 954.0 B
submit_design.html user/ text/html 5.1 KB
Destination
Destination
s3://testing1.com
Destination details
The following bucket settings impact new objects stored in the specified destination.
Bucket Versioning
Disabled
Default encryption
Disabled
Object Lock
Disabled
We recommend that you enable Bucket Versioning to help protect against unintentionally overwriting or deleting objects. Learn more
Enable Bucket Versioning
Feedback
English (US)
Privacy Policy
Terms of Use


Amazon S3
Buckets
Access Points
Batch Operations
Access analyzer for S3
Block Public Access settings for account
Storage Lens
Dashboards
AWS Organizations settings
Feature spotlight
AWS Marketplace for S3
We're continuing to improve the S3 console to make it faster and easier to use. If you have feedback on the updated experience, choose Provide feedback.
Provide feedback
Amazon S3 > testing1.com > Upload
Upload
Add the files and folders you want to upload to S3. To upload a file larger than 160GB, use the AWS CLI, AWS SDK or Amazon S3 REST API. Learn more
Drag and drop files and folders you want to upload here, or choose Add files, or Add folders.
Files and folders (25 Total, 212.8 KB)
All files and folders in this table will be uploaded.
Remove
Add files
Add folder
Find by name
Name
Folder
Type
Size
submit_design.js js/ text/javascript 2.9 KB
update_design.html user/ text/html 5.7 KB
update_design.js js/ text/javascript 4.7 KB
update_user.html admin/ text/html 4.9 KB
user_manage_submission.js js/ text/javascript 12.0 KB
Destination
Destination
s3://testing1.com
Destination details
The following bucket settings impact new objects stored in the specified destination.
Bucket Versioning
Disabled
Default encryption
Disabled
Object Lock
Disabled
We recommend that you enable Bucket Versioning to help protect against unintentionally overwriting or deleting objects. Learn more
Enable Bucket Versioning
Additional upload options
Cancel
Upload
Feedback
English (US)
Privacy Policy
Terms of Use



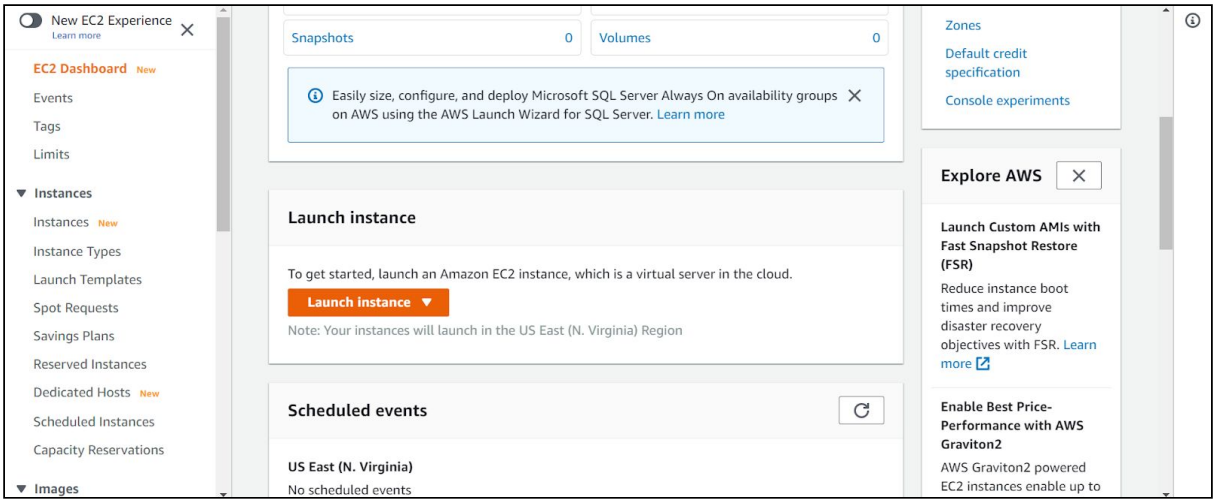
Step 8: We uploaded our ESDE CA1 Front-End files into AWS S3 and it was shown that the upload was successful. As shown previously, since only our static website was hosted successfully, we proceeded to make the dynamic portion in order to run it properly as a whole.



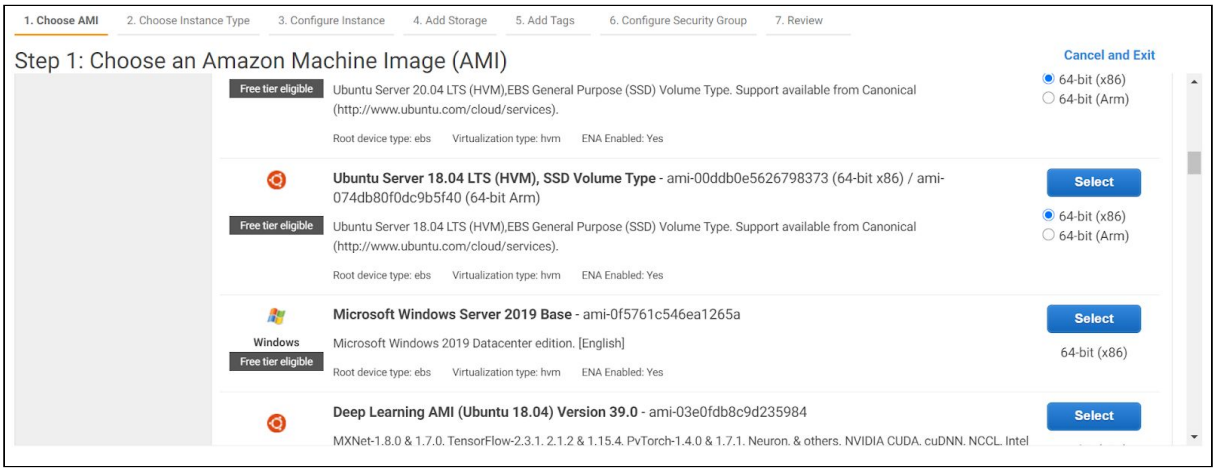
2. Deploy Existing Web Application on AWS (Dynamic)

In this section, we will be showing a step-by-step approach on how to configure the Ubuntu 18.04 Operating System on Amazon Web Services (AWS) Elastic Cloud Compute (EC2) instance and the deployment process of our existing web application using dynamic API.

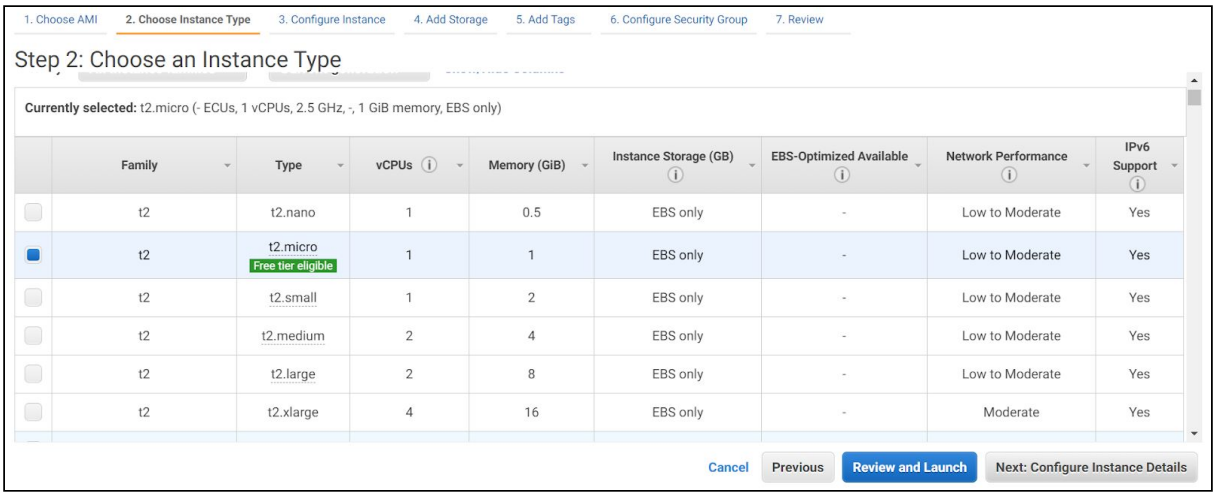
Step 1: After logging into our AWS account, we navigated to the EC2 dashboard and clicked on the “Launch Instance” button.



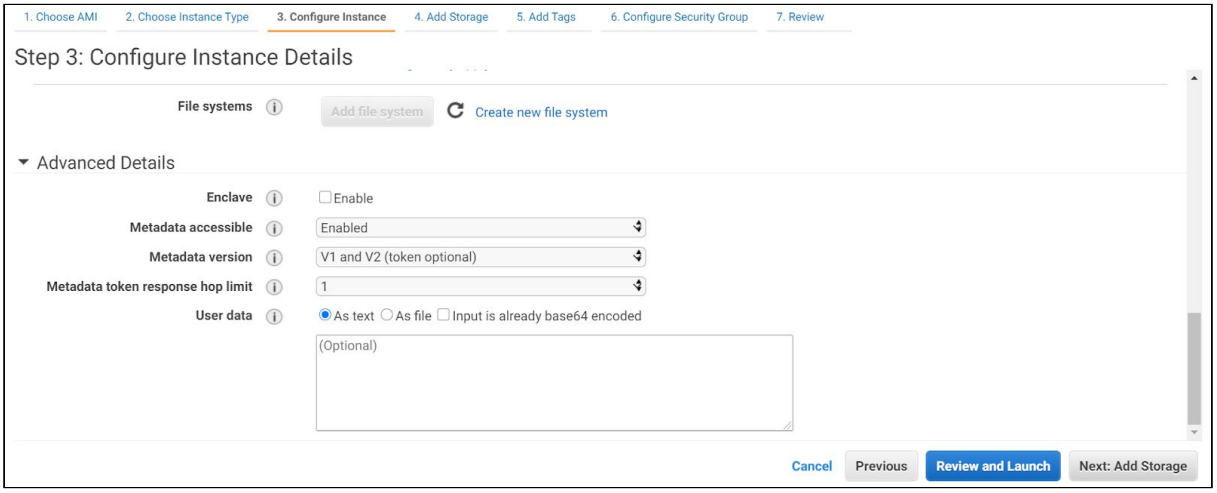
Step 2: We choose to use Ubuntu Server 18.04 LTS (64-bit) for our Amazon Web Image (AMI).



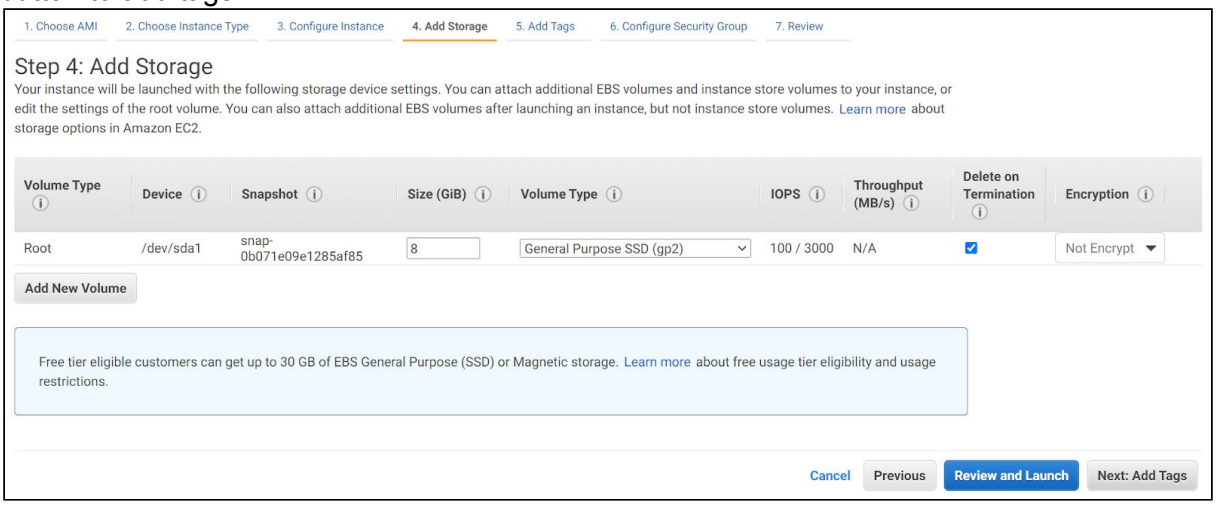
Step 3: We chose to use t2.micro instance type, which is a 1vCPU and 1GB memory server offered by AWS as an eligible free tier. We proceeded to click the “next” button to configure instance details.



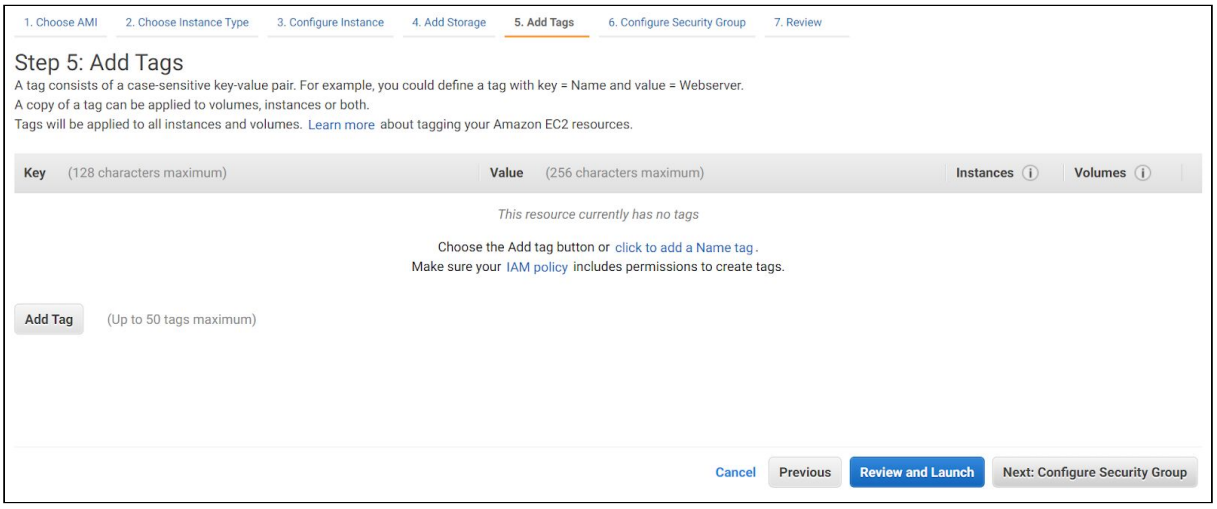
Step 4: We applied the default details for the instance and proceeded to click the “next” button to add storage.



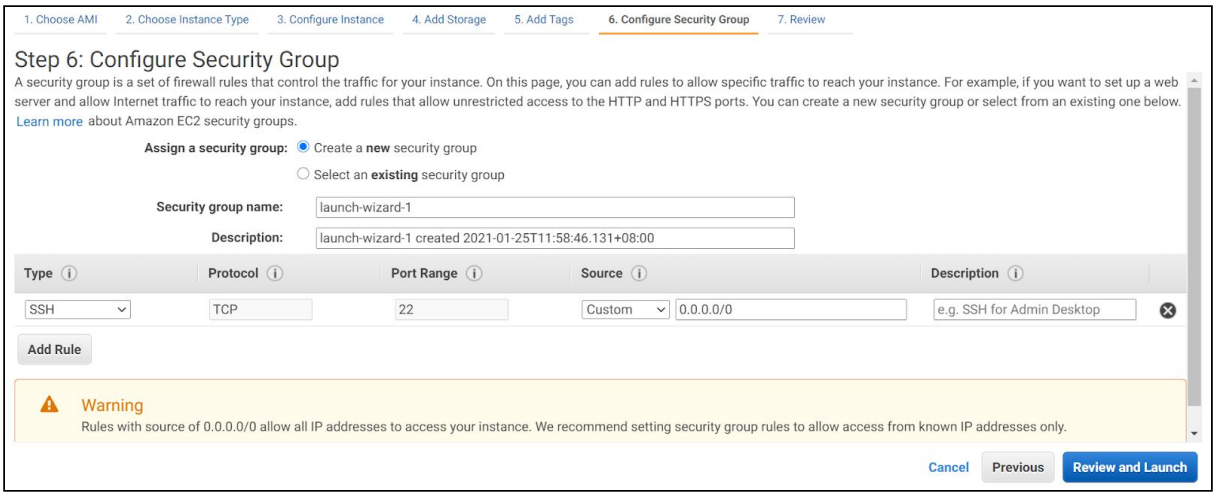
Step 5: We applied the default details for the storage and proceeded to click the “next” button to add tags.



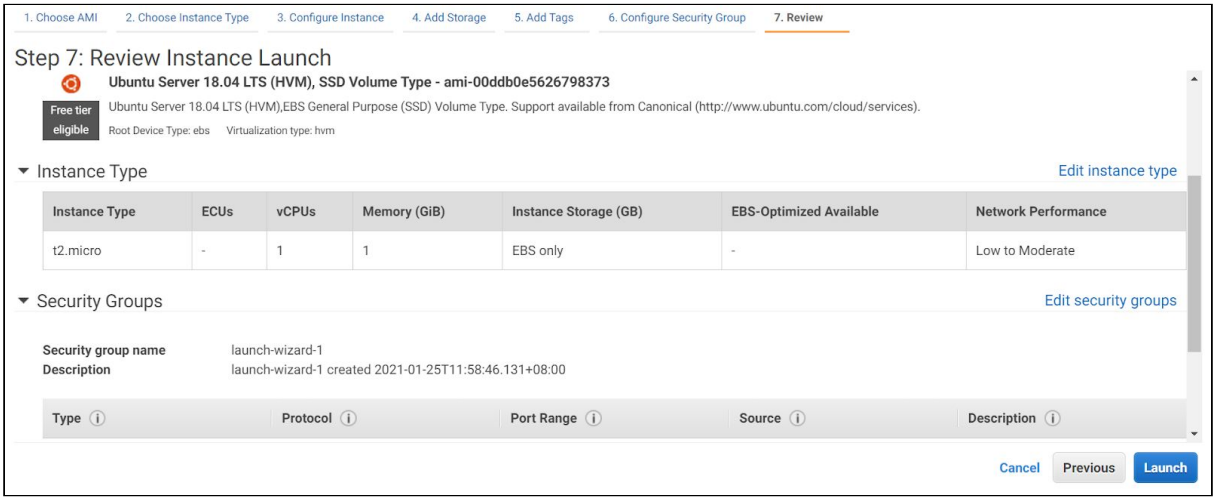
Step 6: We applied the default for the tags and proceeded to click the “next” button to configure the security group.



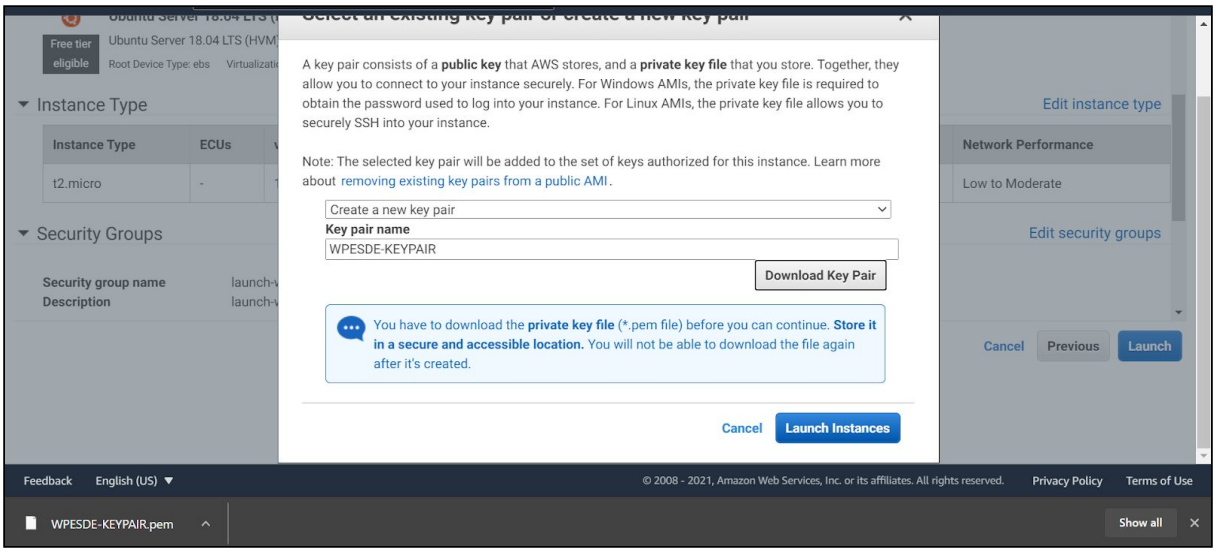
Step 7: We applied the default for the security group, which is 0.0.0.0/0 as a source out of convenience as our IP addresses are prone to changes in different environments. We proceeded to click the “review and launch” button to launch the EC2 instance.



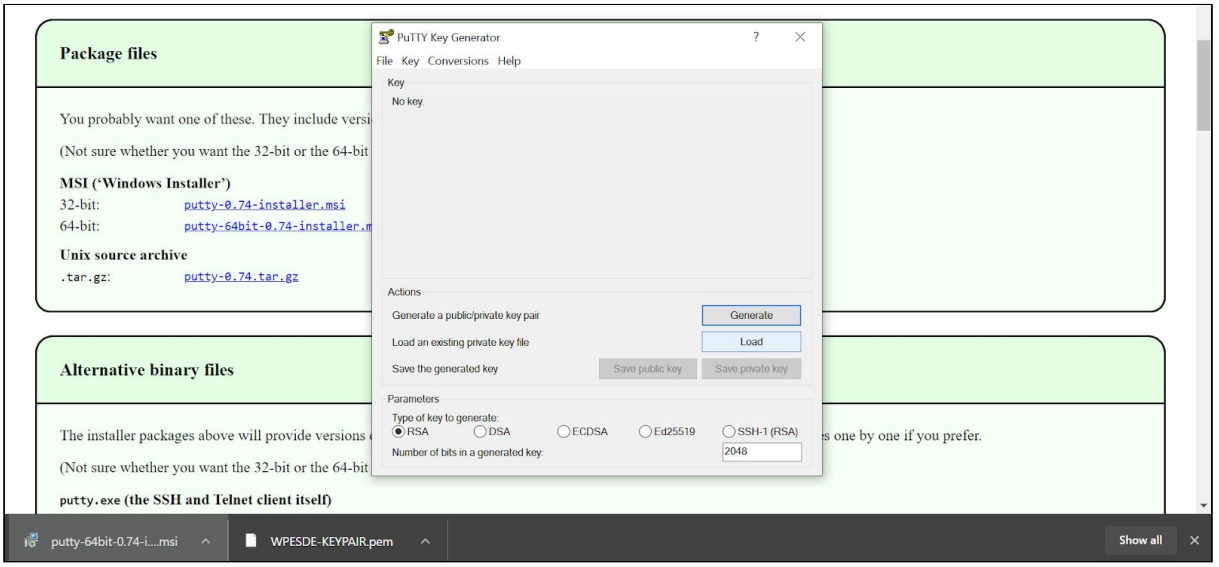
Step 8: We did a final check of the settings for the instance type before proceeding to click the “launch” button to launch the EC2 instance.



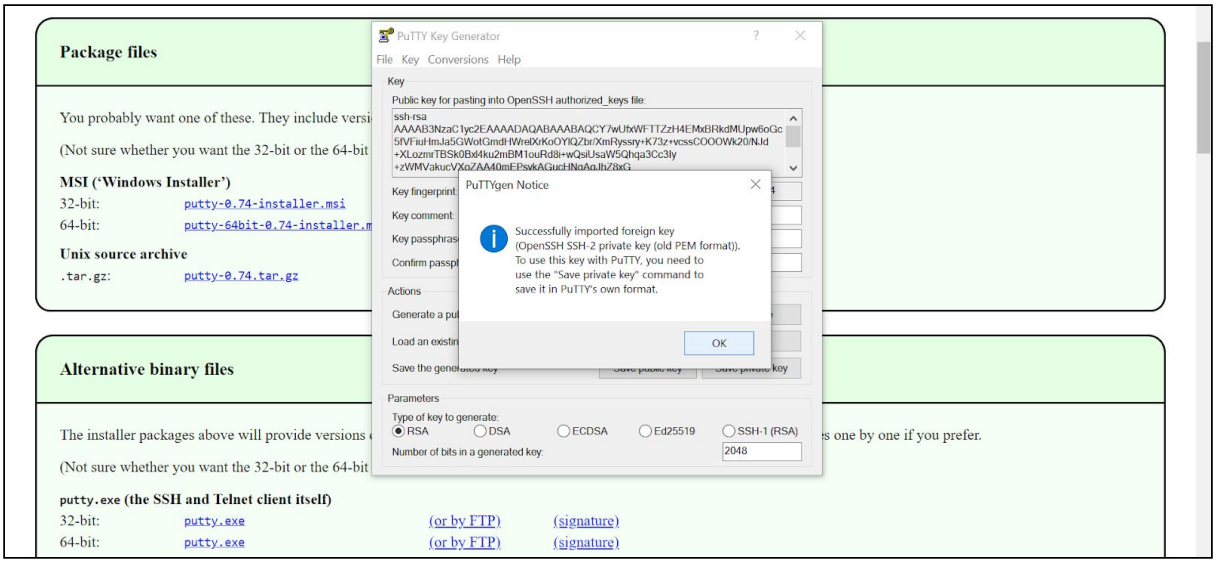
Step 9: We were prompted to create and download a private key, which allows us to connect with the EC2 instances using SSH. From the dropdown menu, we selected “create a new key pair” and named it “WPESDE-KEYPAIR” before clicking the “download key pair” button.



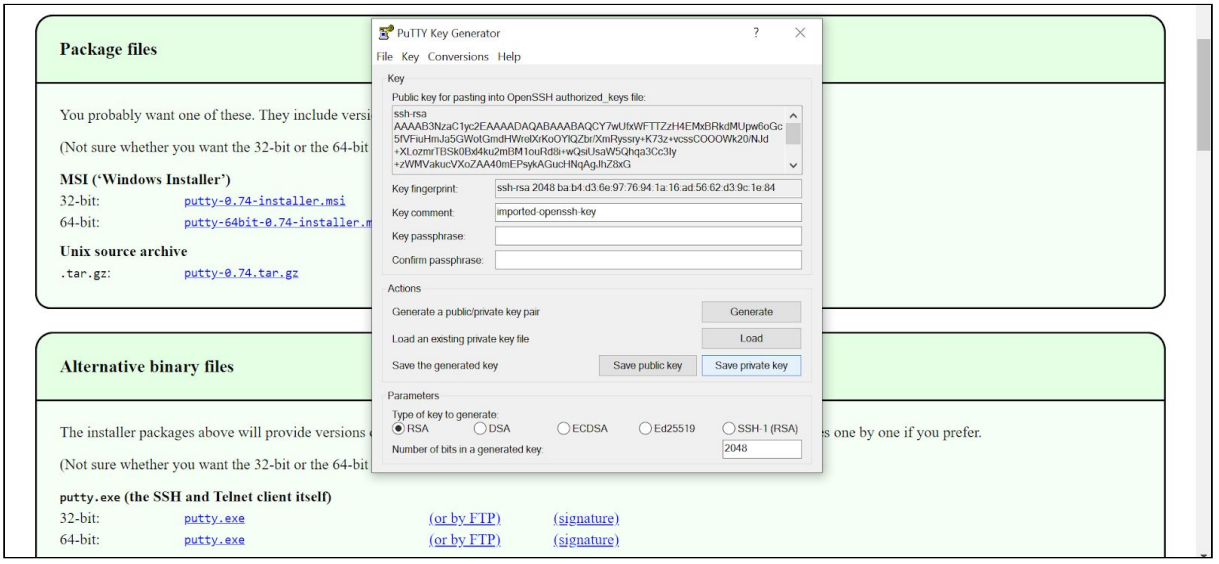
Step 10: We chose the 64-bit MSI to install putty which also installs puttygen. As the key pair downloaded previously is of .pem, we need to generate a private key to login to the remote instance as putty does not accept that kind of extension. After opening puttygen, we clicked on the “file” button then “load private key” button to convert it.



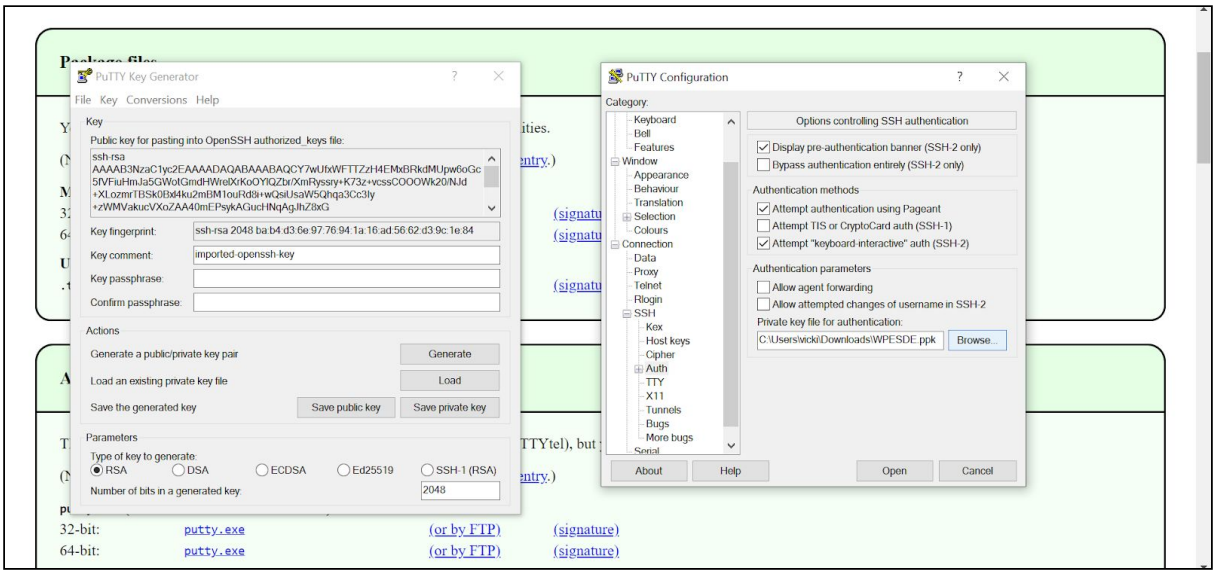
Step 11: We were informed that our private key of RSA type has been generated and proceeded to click “ok” to close off the notice.



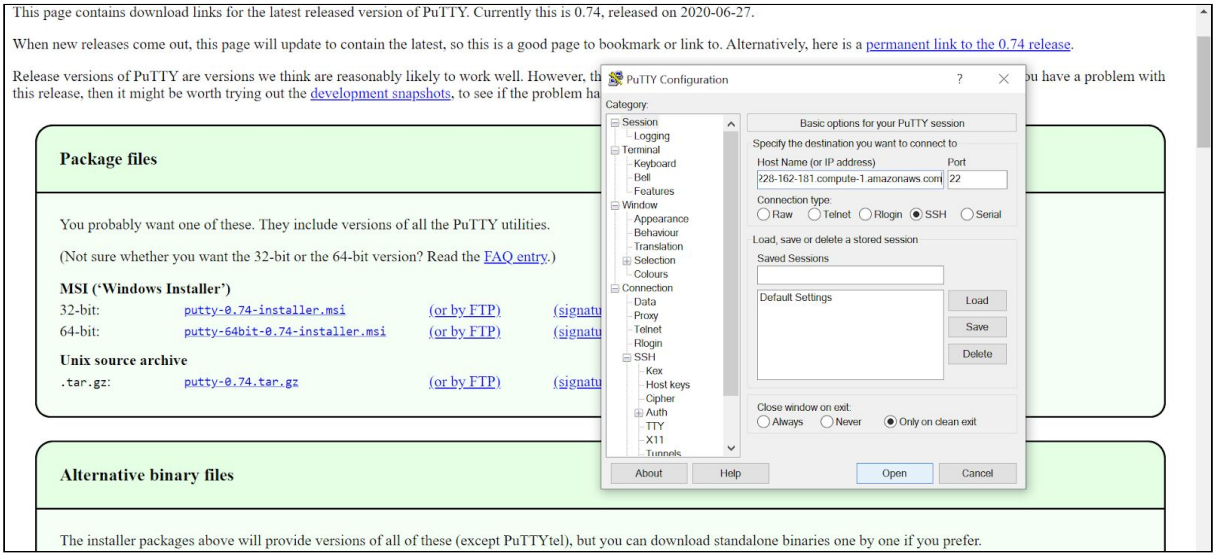
Step 12: We clicked the “save private key” button to save it to a file of .ppk extension and named it “WPESDE” . We also opted for no passphrase so that it will be less troublesome as manual human intervention is not needed to log on or copy files to an instance.



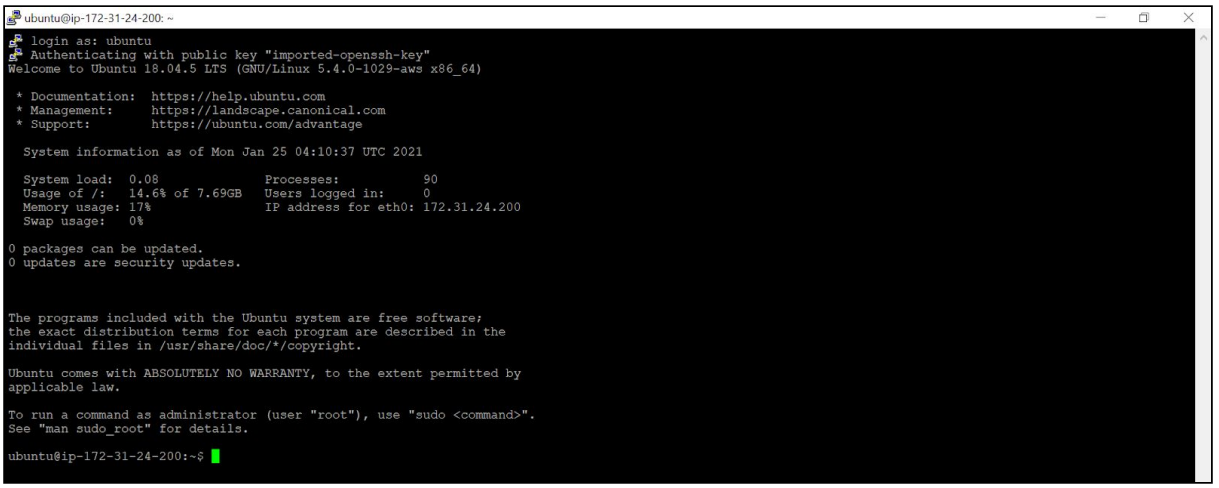
Step 13: After opening putty, we clicked on the “connection” button then the “SSH” button then the “Auth” button and lastly the “browse” button to search for the previously downloaded .ppk file and uploaded it to provide a key file for authentication.



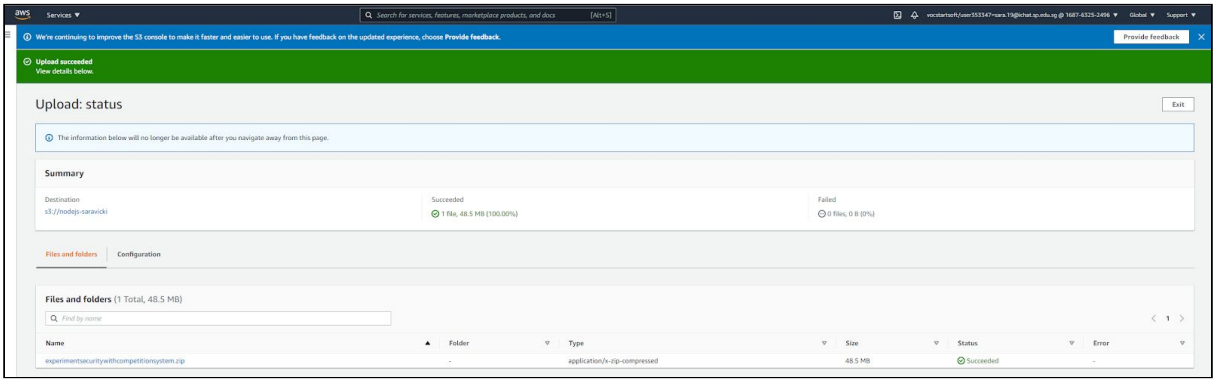
Step 14: We clicked on the “session” button to fill in the details for the putty session. We proceeded to fill in the host name (or IP address) field with our public DNS stated in AWS for the EC2 instance.

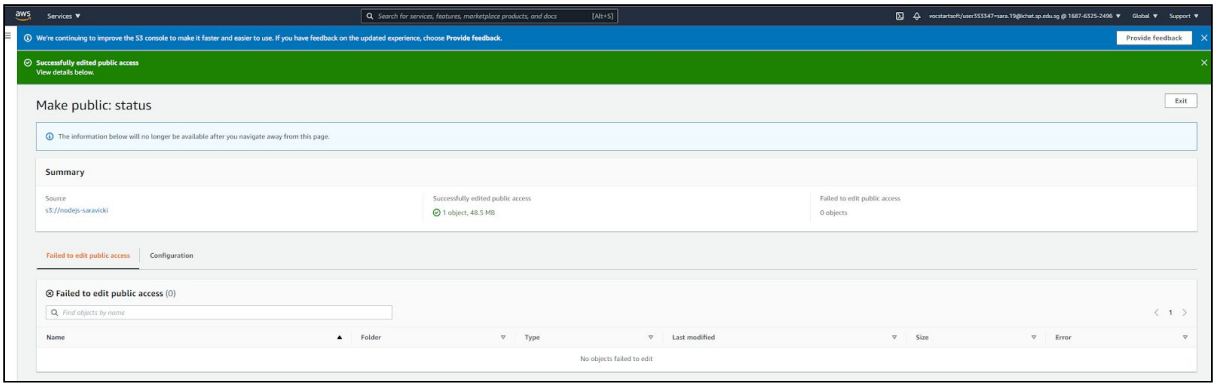


Step 15: We clicked on the “open” button at the bottom of the previous putty session configuration page and successfully logged into putty with the username of “ubuntu”.



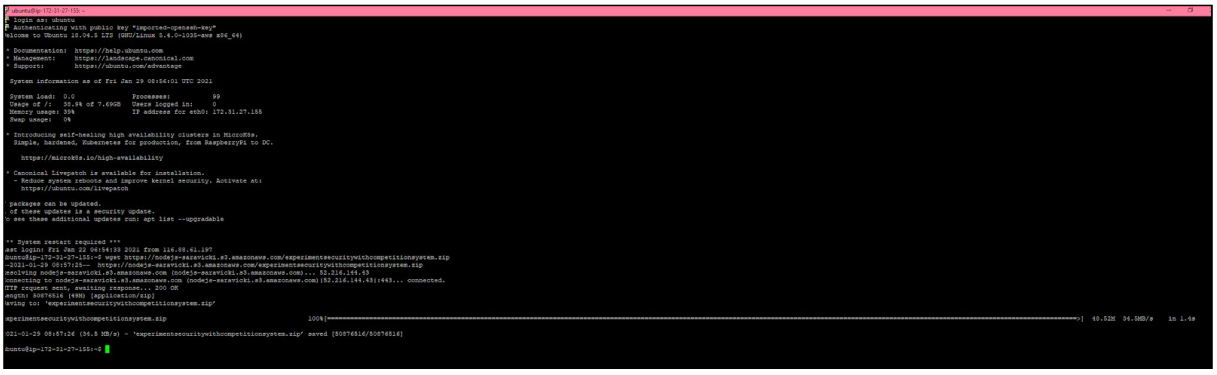
Step 16: We changed the status of our previously created S3 bucket that we used to upload our CA1 zipped folder into and changed the status of it to enable public access. We were notified that the status edit is successful.



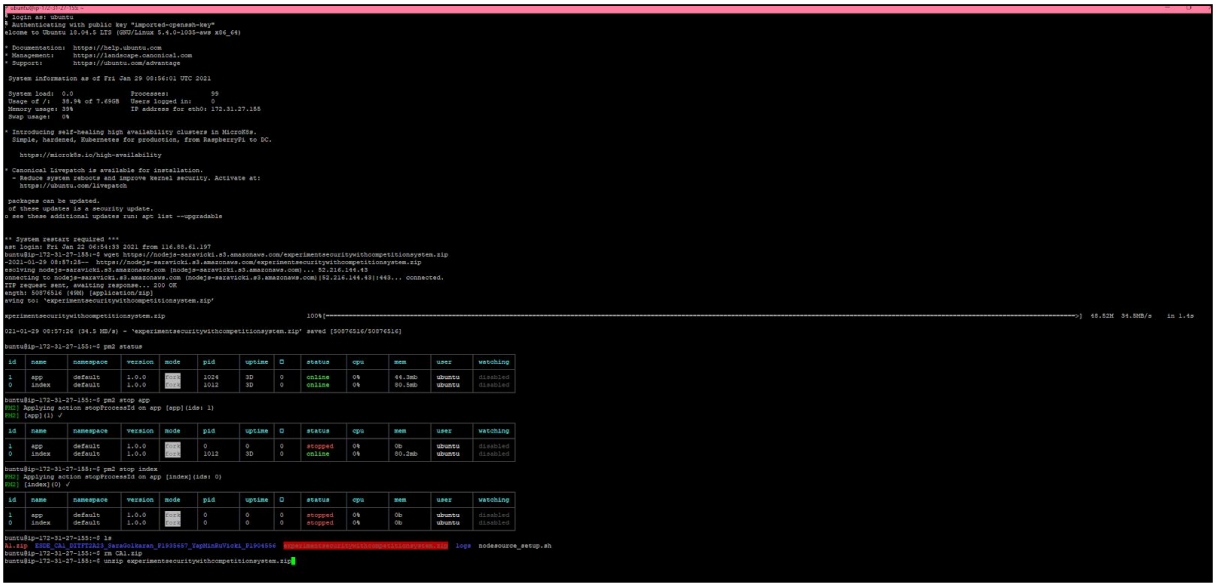


Step 17: We verified that the node application is copied to AWS by entering “wget https://nodejs-saravicki.s3.amazonaws.com/experimentsecuritywithcompetitionsystem.zip”.

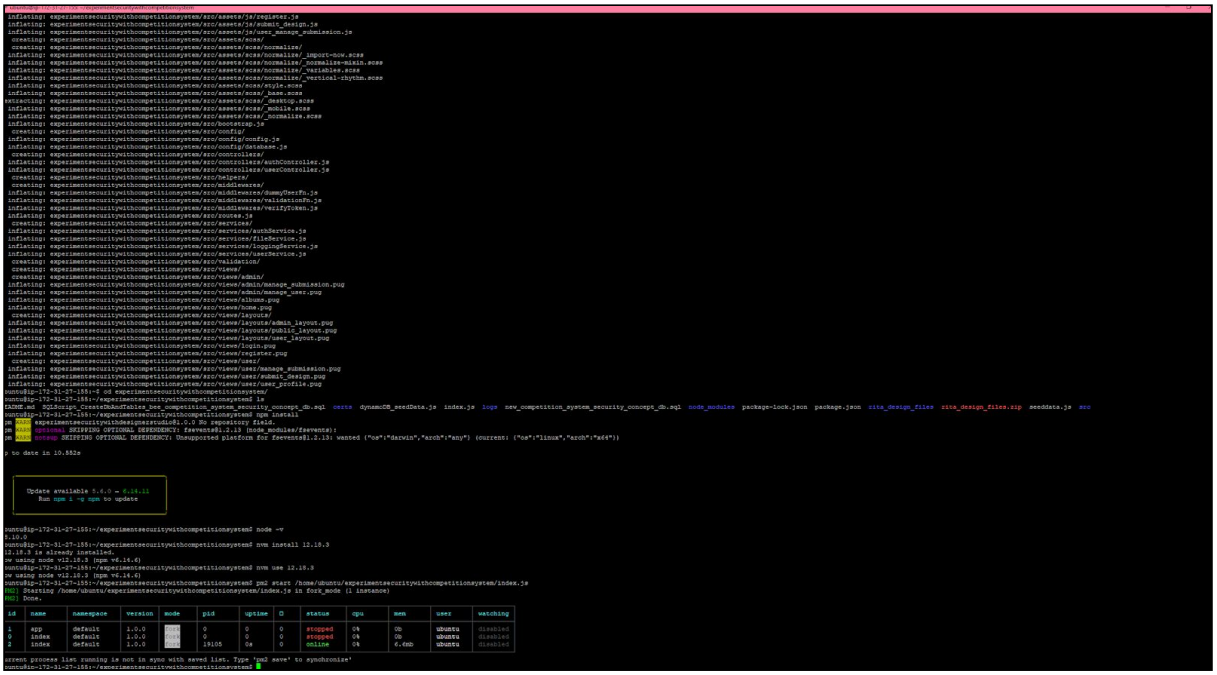
Step 18: We saw that the contents are fetched accordingly from our home directory. This shows that we managed to upload our zipped folder to the S3 bucket successfully.



Step 19: We removed a zipped folder consisting of both frontend and backend files which we uploaded incorrectly previously by entering “rm CA1.zip”. We proceeded to unzip the recently uploaded correct backend folder by entering “unzip experimentsecuritywithcompetitionsystem.zip”.



Step 20: We saw that the contents are extracted accordingly from the backend folder. We also installed the node version manager (nvm) by entering “nvm install 12.18.3 ” and checked the node version previously installed by entering “node-v”. We proceeded to run the command “pm2 start /home/ubuntu/experimentsecuritywithcompetitionsystem/index.js” to start the server automatically after each restart.

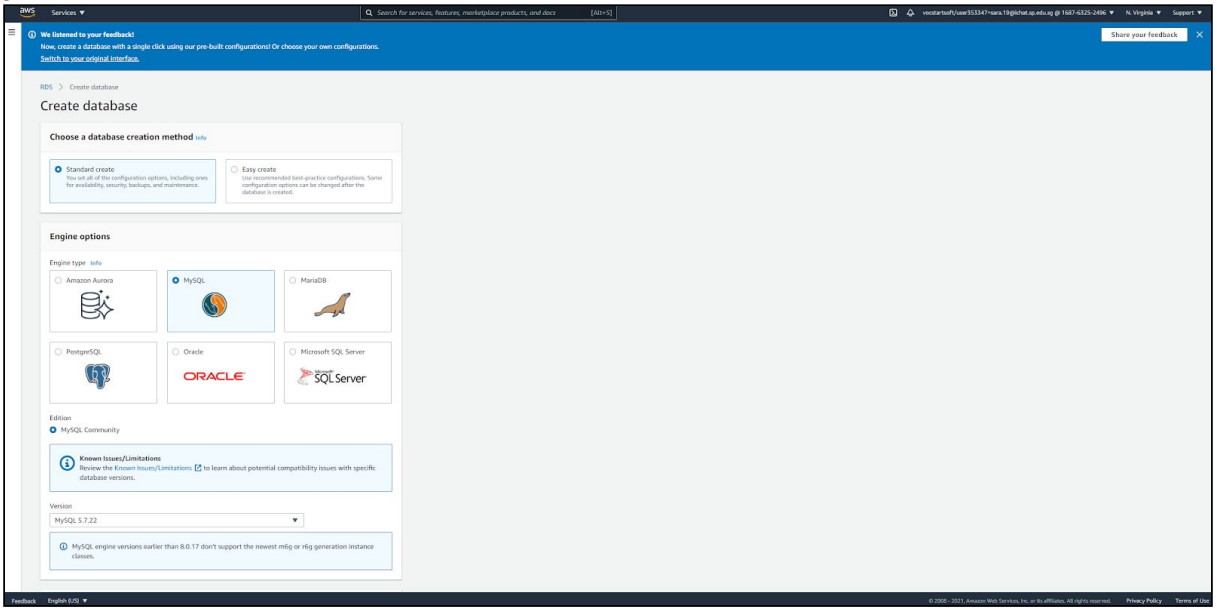


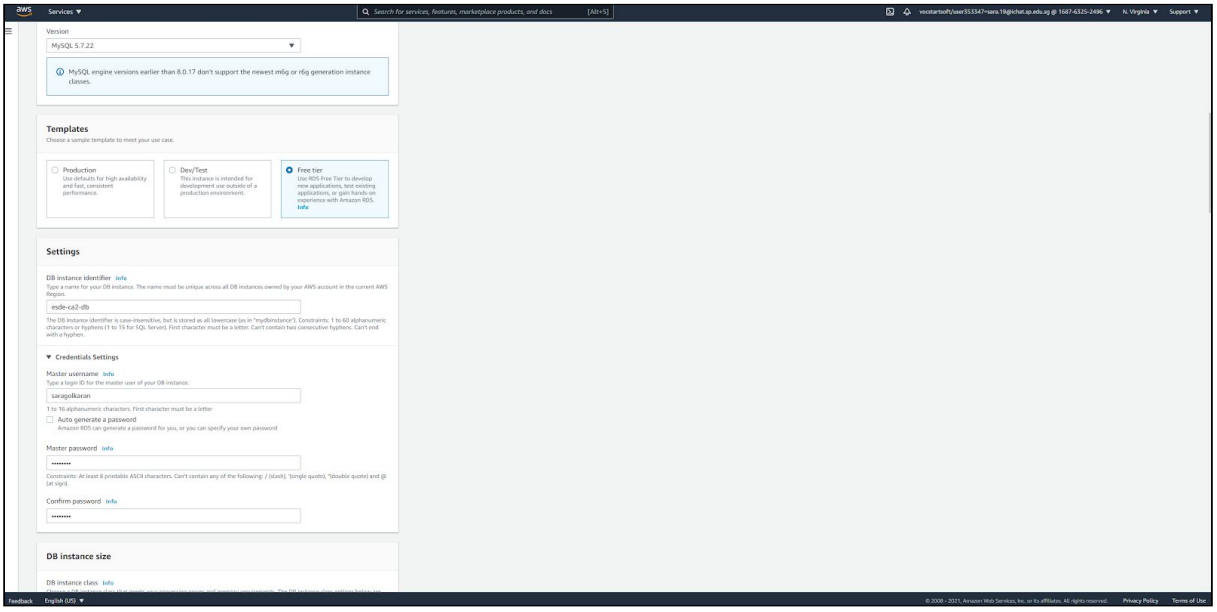
Step 21: We saw the following message that “this project provides only backend API support” after opening our browser at port 5000 and entering our public IP address and domain name stated in AWS for the EC2 instance. This shows that our backend web API is successfully hosted on EC2.

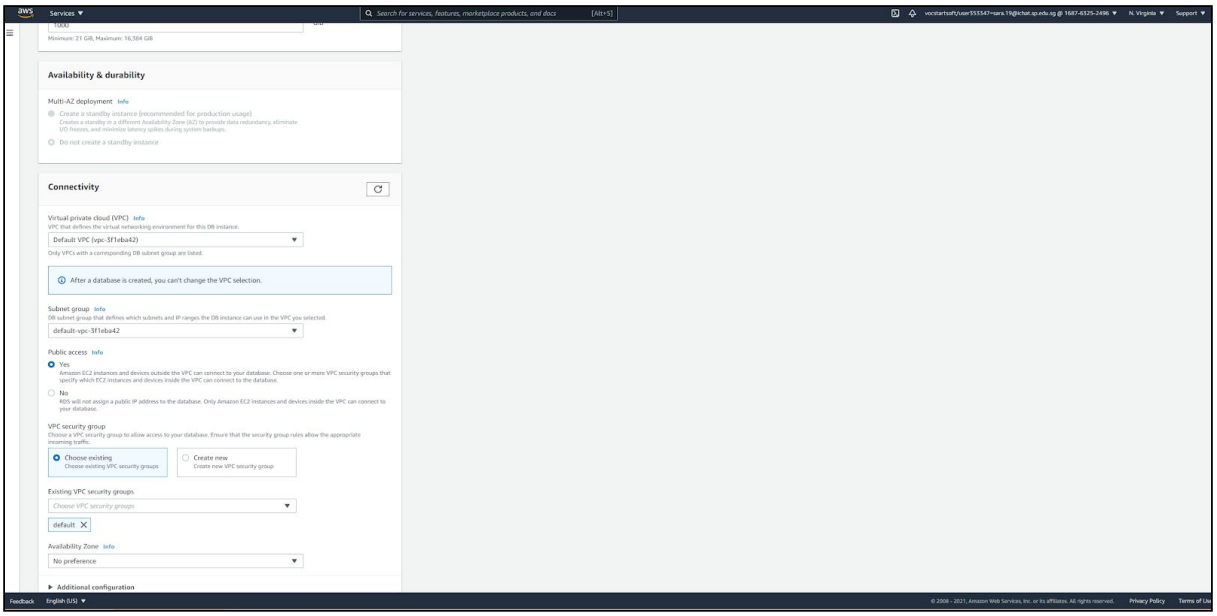
In this section, we will be showing a step-by-step approach on how to use Amazon RDS to create a MySQL DB Instance.

Although DynamoDB makes use of partitions to store data, we use Amazon RDS as it has bigger read-write capacity and is more flexible in terms of storage and faster in retrieving the datasets queried by us.

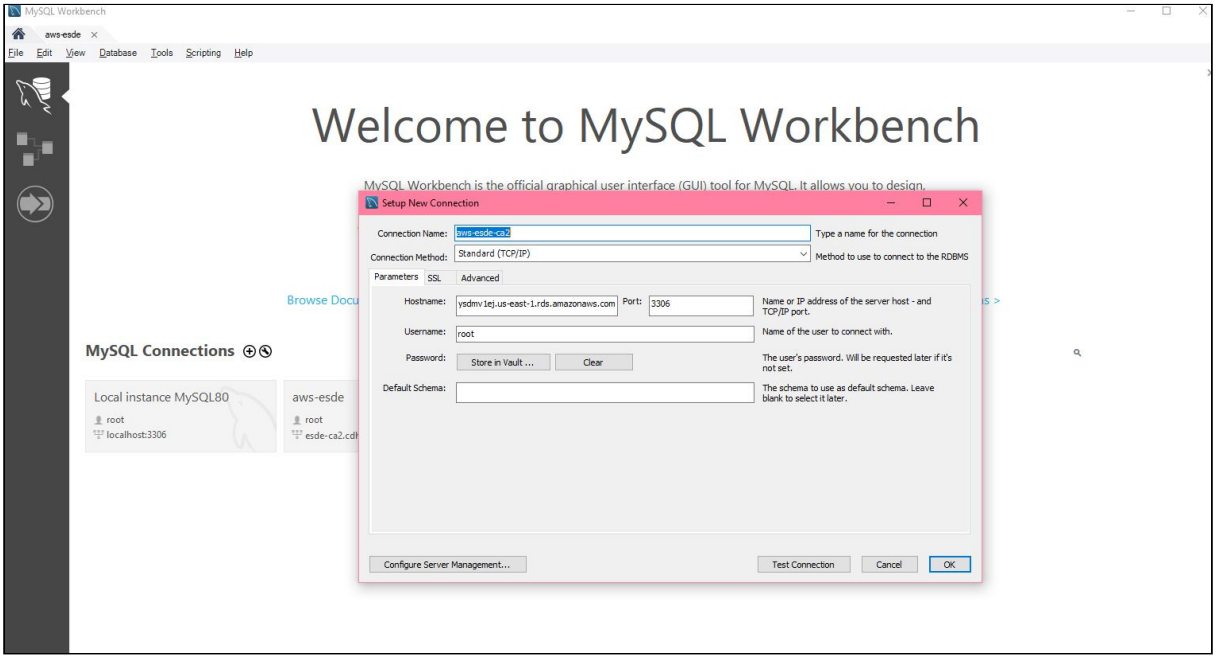
Step 22: After logging into our AWS account, we navigated to the RDS dashboard and clicked on the “Create database” button. For our engine options, we clicked the MySQL icon and selected the Free Tier template which is the most cost effective option to help us save up our limited $50 credits. We opted for the default settings which include public access and existing VPC security group, which allowed us to directly connect to the database from our own devices by allocating an IP address just for our database instance. We proceeded to give our database a unique name “esde-ca2-db”



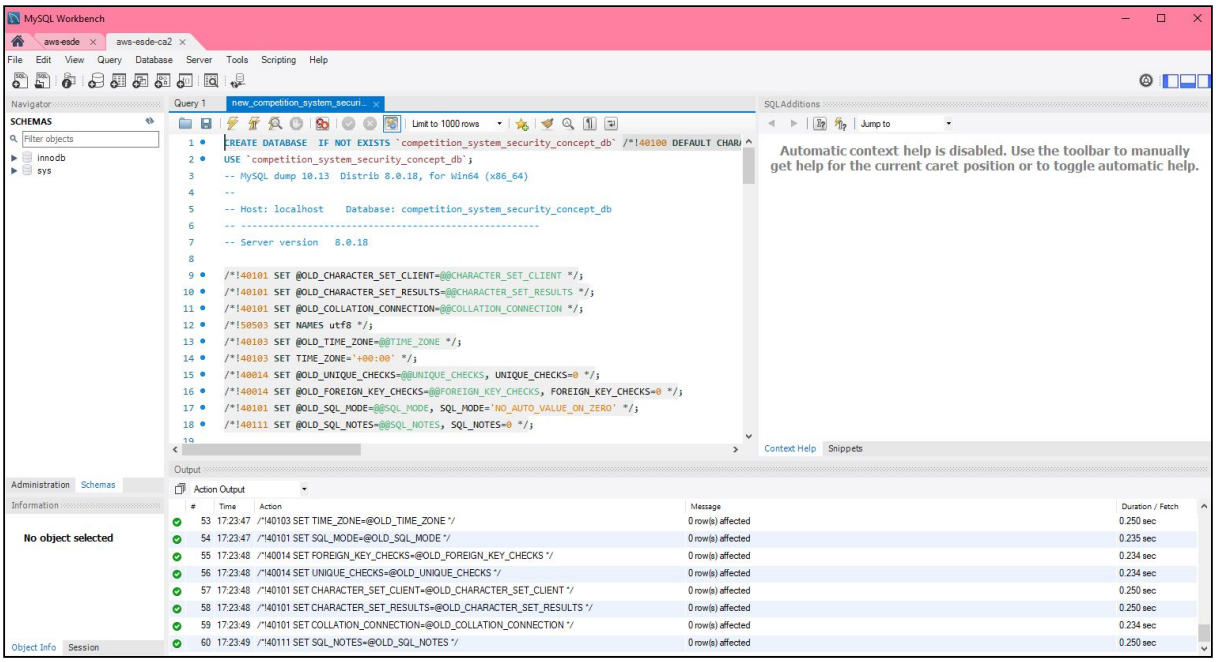




Step 23: As it will be more useful to access a RDS instance to remotely send SQL commands, we need to connect AWS to MySQL. After opening MySQL Workbench, we clicked the “+” icon next to “MySQL Connections” to set up a new connection. We proceeded to fill in the hostname, port and username fields with the endpoint, port number and principal username stated at AWS RDS instance.

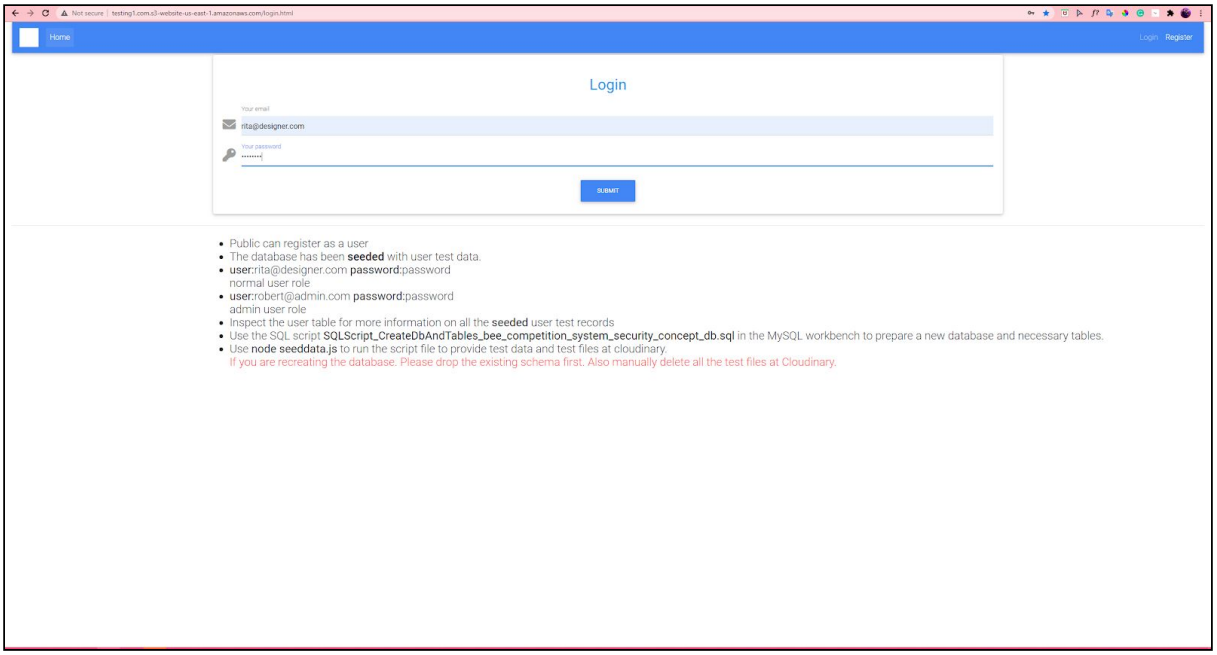


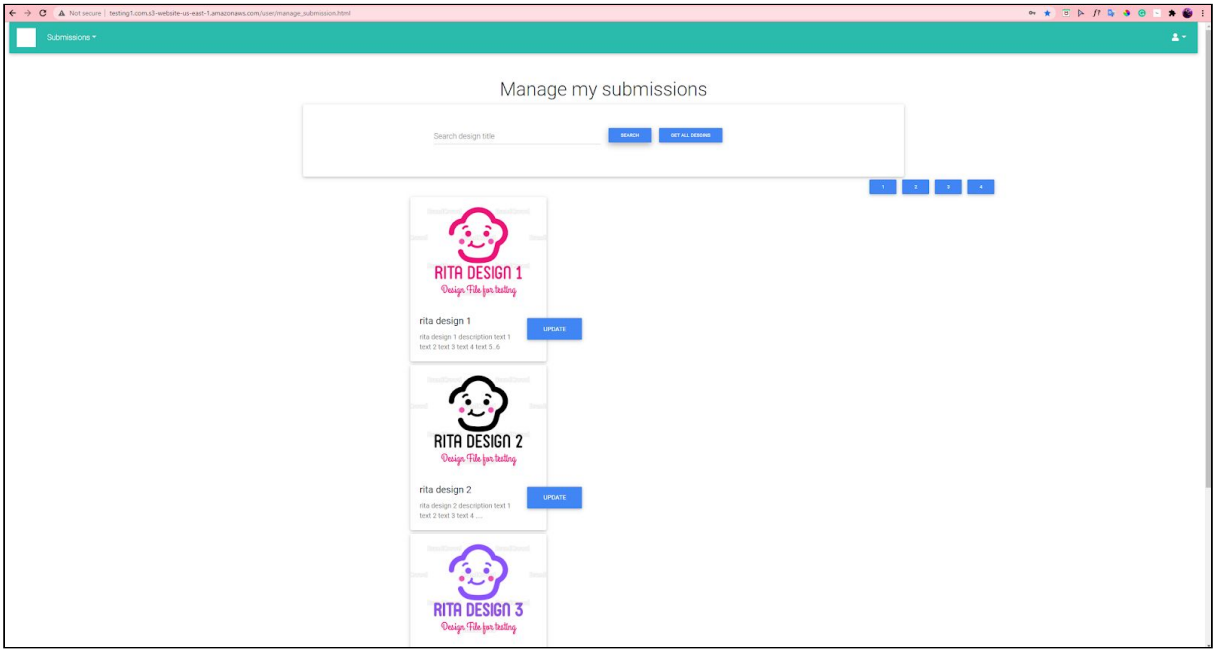
Step 24: We opened our SQL codes dump for CA1 and executed it by clicking the “lightning bolt” icon at the top of the page. We see that the files are successfully inserted into the created database.



Step 25: We used Ubuntu to modify the connection info string for host in databaseConfig.js with that of our RDS instance.

Step 26: We can now perform a login action as Rita on the frontend login page. This shows that our website is hosted on AWS as we are able to access our own submissions using the account of user id 100.



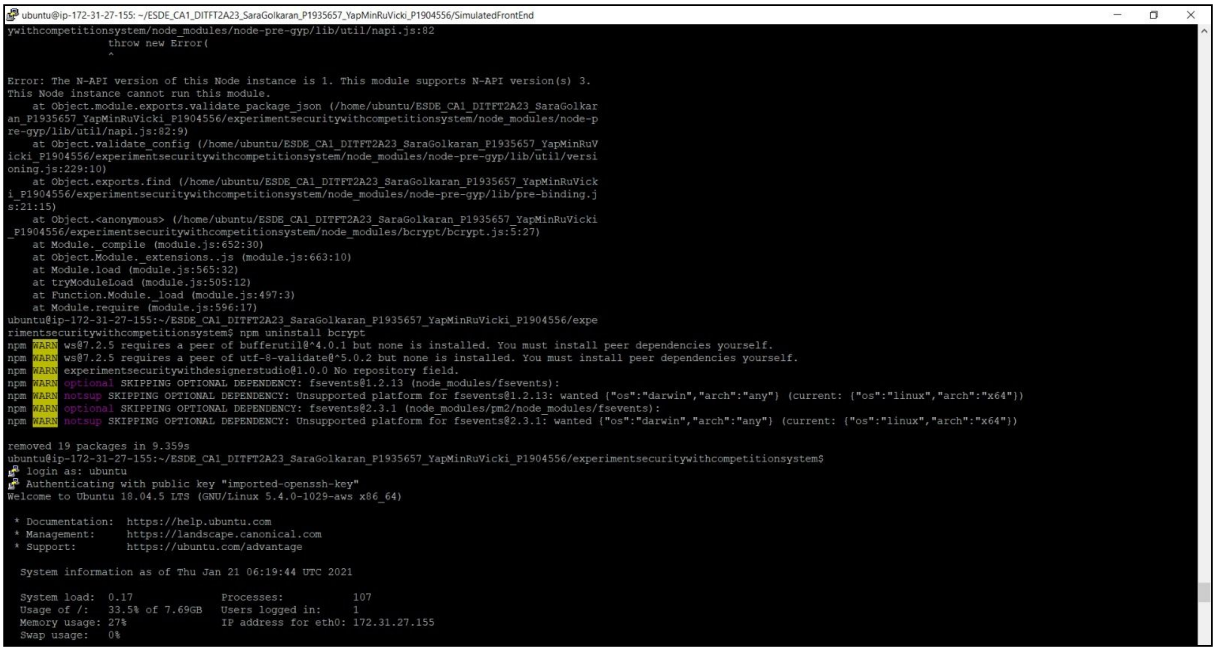


3. Troubleshoot and Test during Deployment Stage

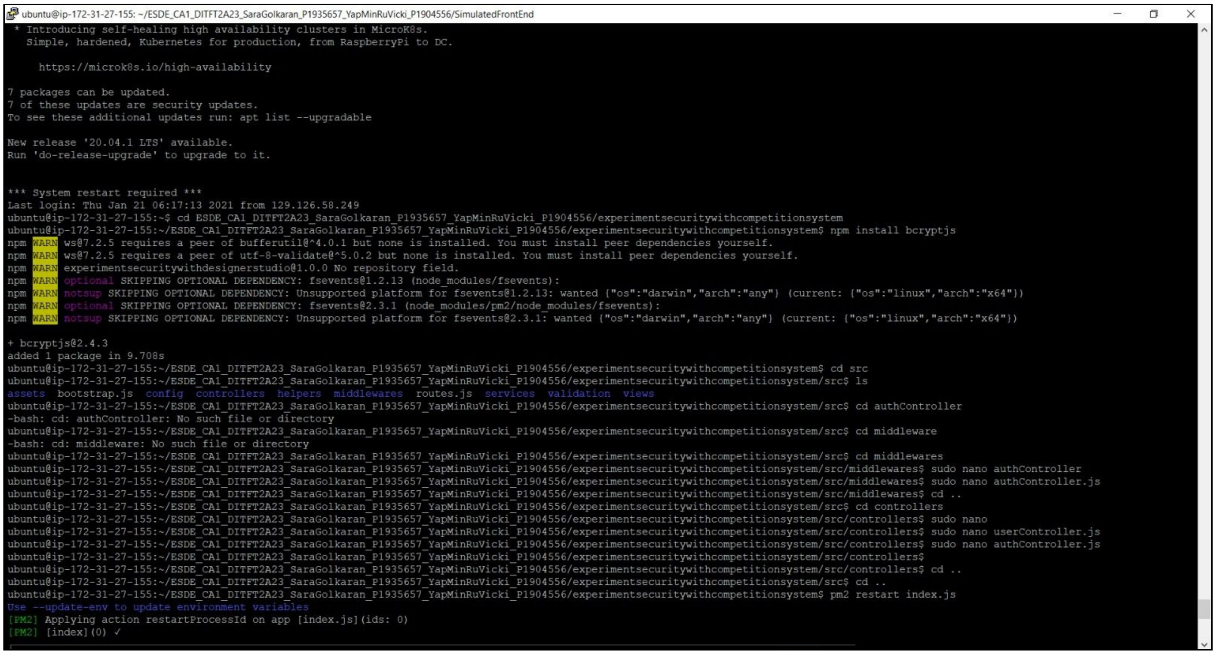
3.1 Changing Modules from Bcrypt to BcryptJS

At first, we only managed to deploy the dynamic website into AWS. However, we face some issues in the backend side thus we are unable to do any login in the webpage. Basically, the front end is working but the backend is not working. After troubleshooting many times, we

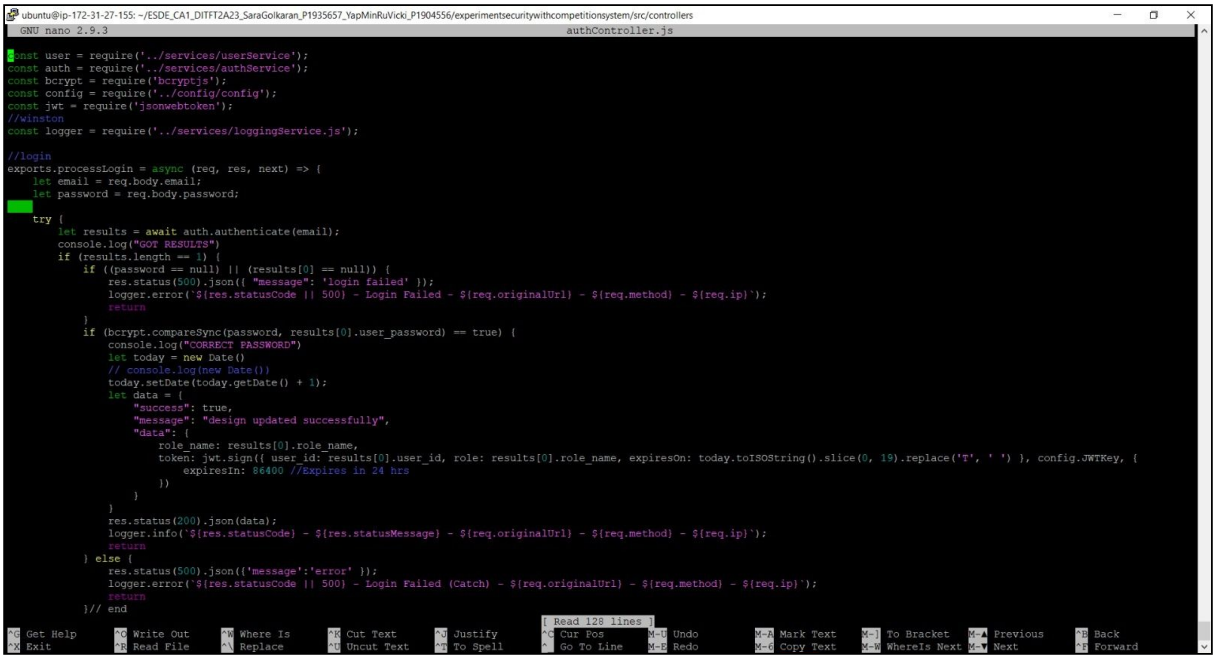
realised there were some issues with our bcrypt modules, thus we replaced our bcrypt with bcryptjs instead. The following image is the process of uninstalling bcrypt modules.



The following image is the process of installing bcryptjs for the upper part of the image. The second part of the image is the “sudo nano” command to edit the authcontroller.js file directly in the ubuntu terminal without opening Visual Studio Code. As the authcontroller.js file used bcrypt modules, thus we also need to replace it with bcryptjs.



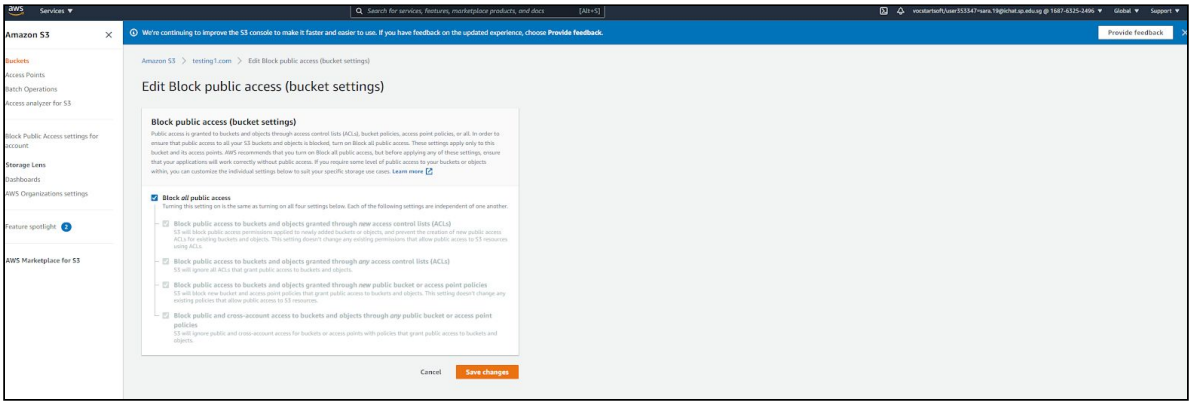
The following image is editing the authcontroller.js file. We replace bycrypt with bycrpt.js.



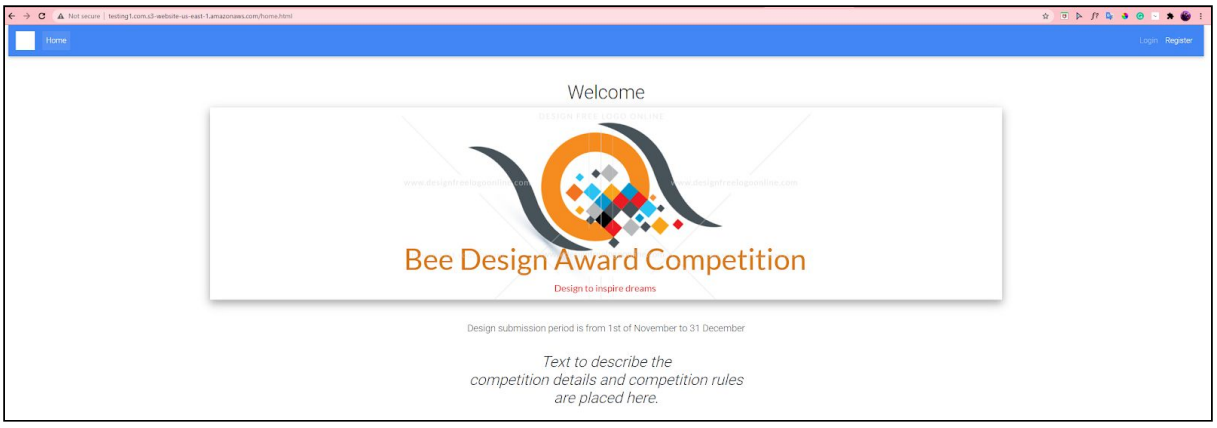
3.2 Faced 403 Forbidden Error Issue

At first, we encountered this 403 Forbidden error when we clicked on the link as a public user.

We troubleshooted and realised that this is due to the public access settings for our S3 bucket used to host the backend. Initially, we applied the default settings where we blocked public access. As such, we unchecked the checkbox to allow public access before clicking on the “save changes” button.



After unchecking the checkbox to allow public access, we can now access our hosted website.



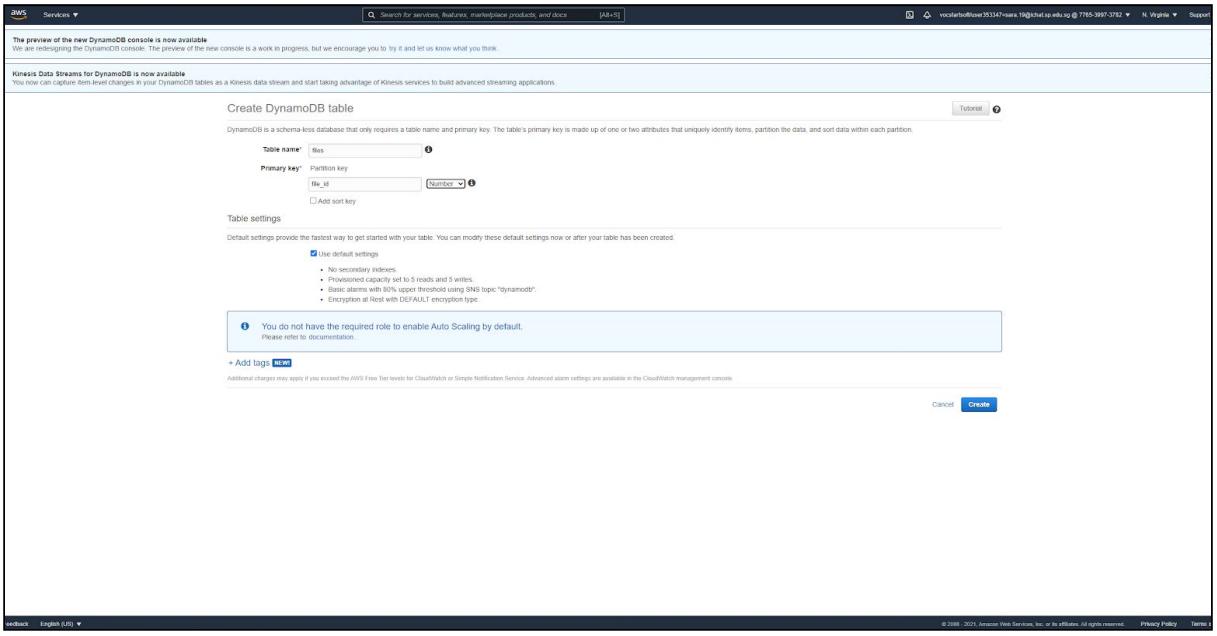
4. Refactoring Application using Lambda Function

In this section, we will be showing a step-by-step approach on how to develop using AWS Lambda Function and enable Cross-Origin-Resource-Sharing (CORS) on the endpoint from the Amazon API Gateway console.

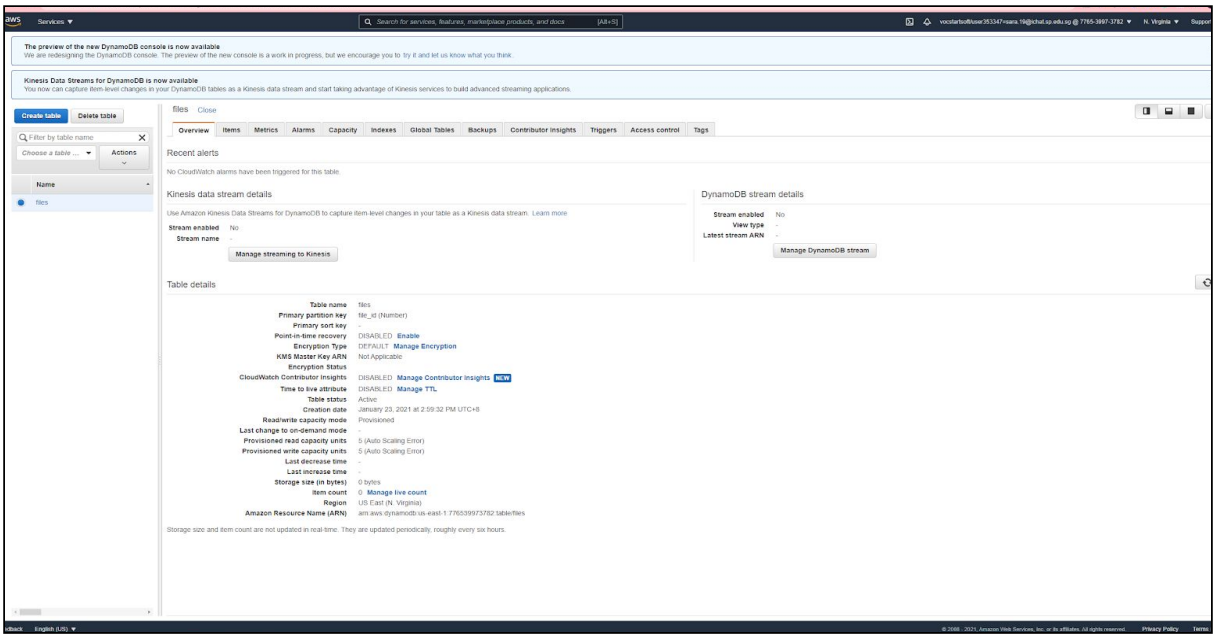
We came across the error of the browser blocking our query request with the message “Failed due to CORS”. After troubleshooting many times, we realised that this is due to our browser enforcing a cross-domain policy rule.

4.1 Creation of Amazon DynamoDB

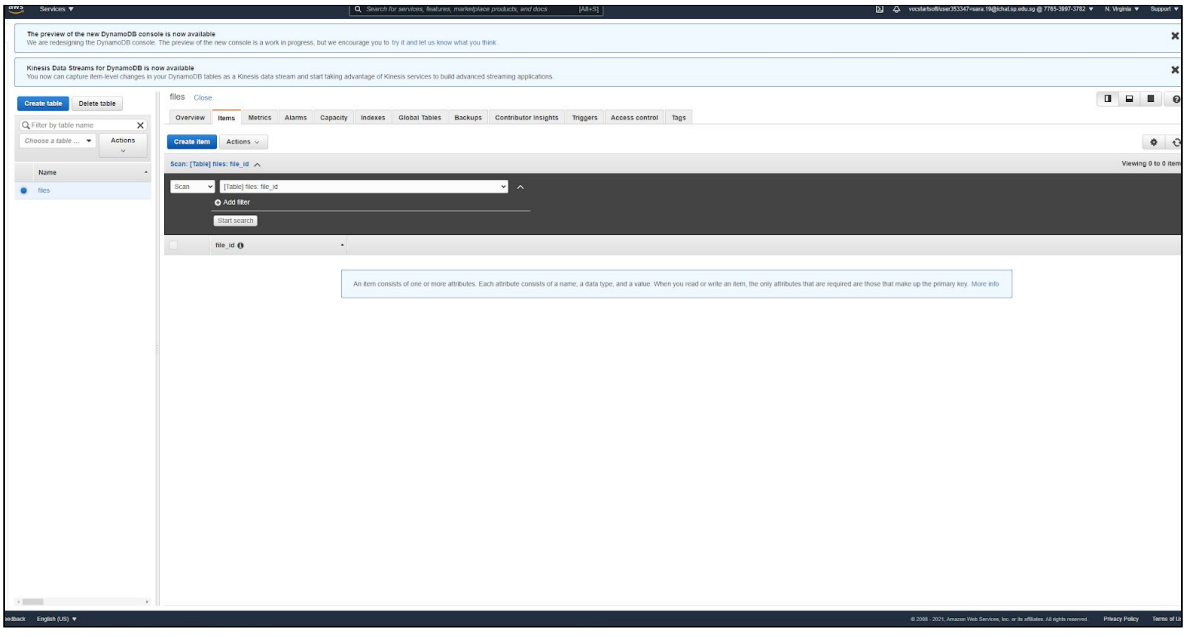
Step 1: After logging into our AWS account, we navigated to the Amazon API Gateway console to select the API type. We chose to “build” a REST API that works with Lambda, HTTP and other AWS services.

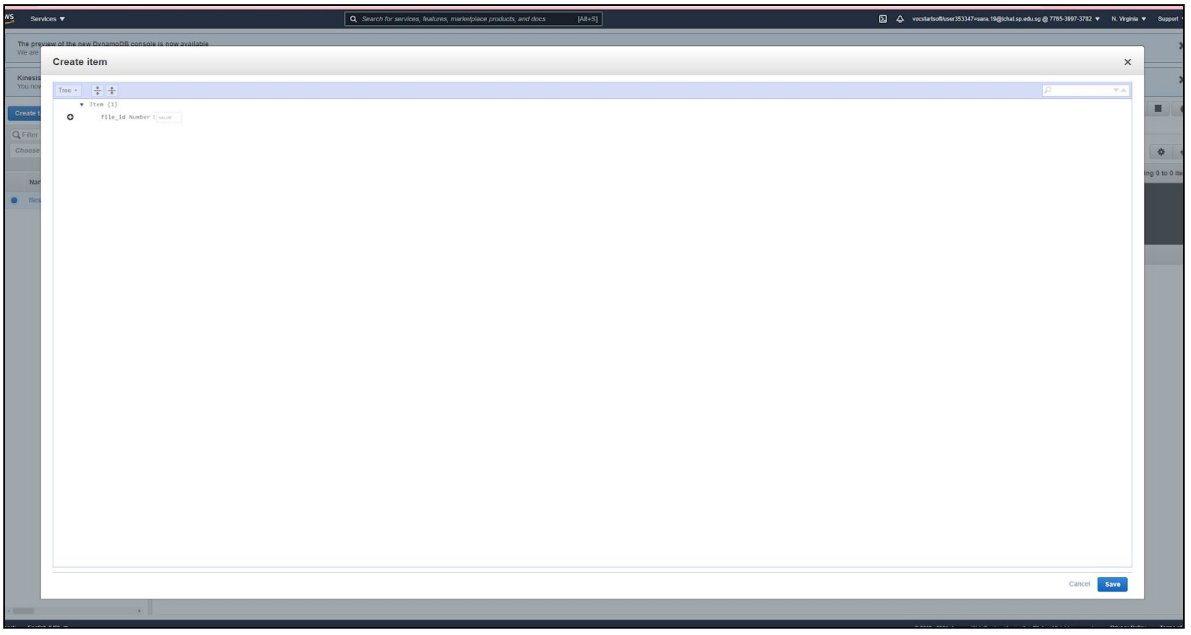


Step 2: After logging into our AWS account, we navigated to the Amazon API Gateway console to select the API type. We chose to “build” a REST API that works with Lambda, HTTP and other AWS services.

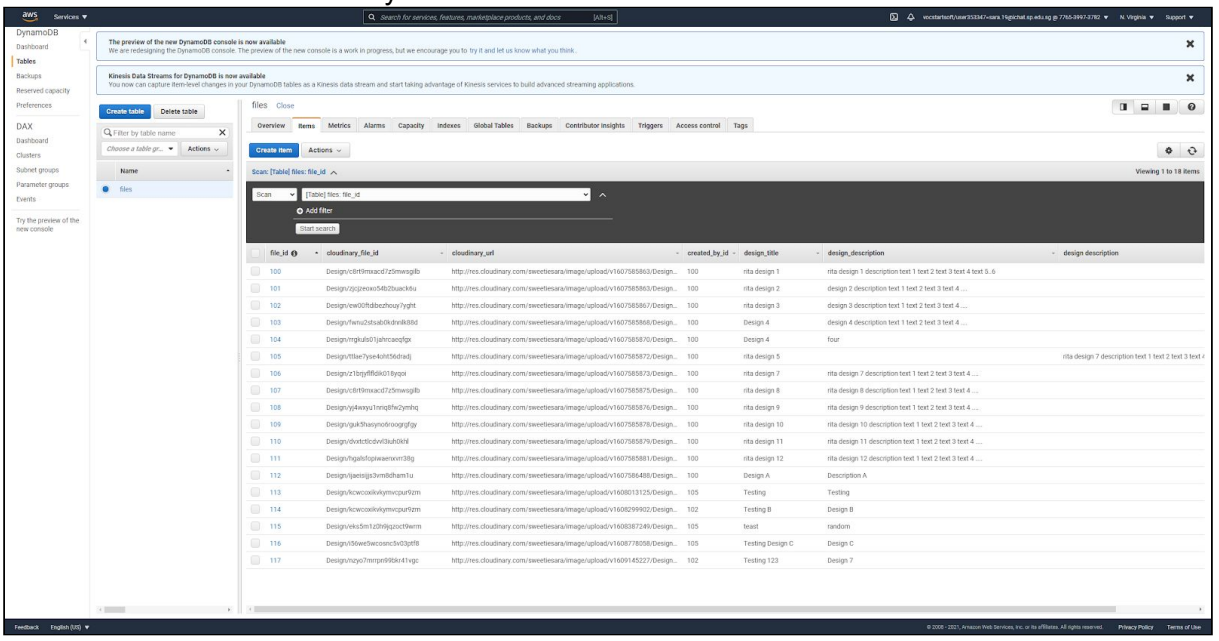


Step 3: After creating the REST API for “files”, we navigated back to the items tab and clicked on the “create item” button to add rows to the table where we saw an example of a column attribute required to be added in: file_id.



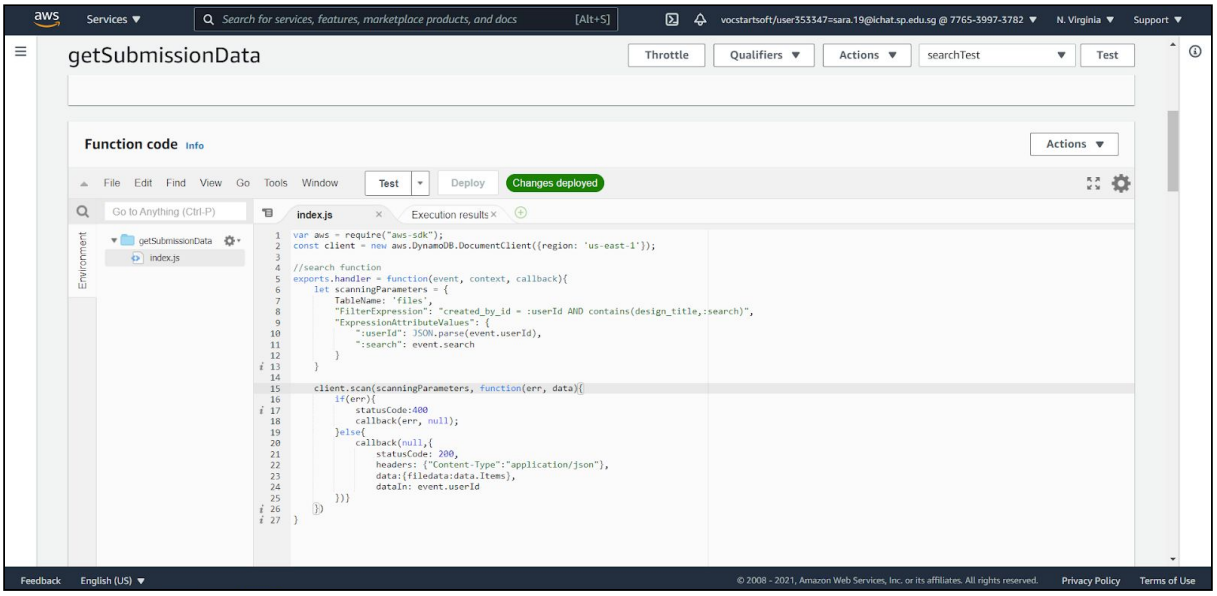


Step 4: We successfully added in the previous Cloudinary submissions of file id 100 to 117 to the current “files” table in DynamoDB as seen below.

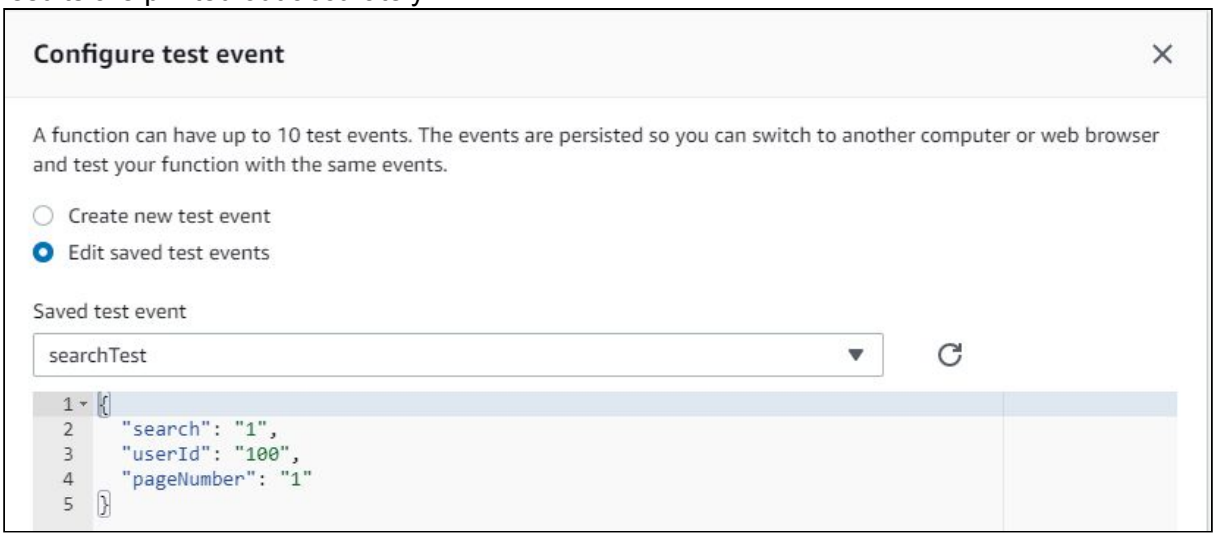


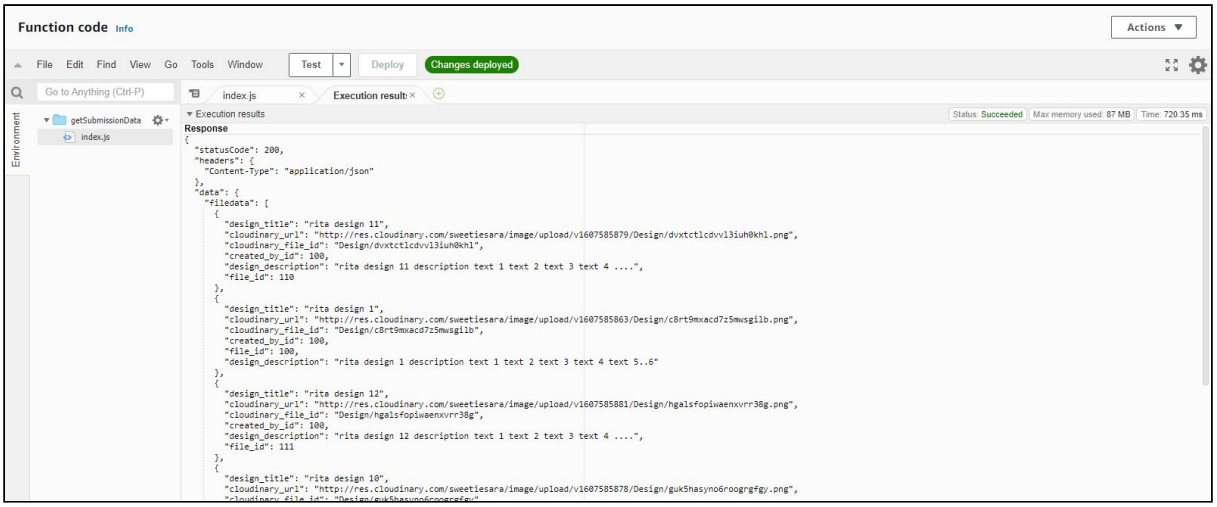
4.2 Creation of Lambda Function to Get All Users Submission

Step 1: By initiating a client that operates DynamoDB in our user account's region, we modified the index.js file to write a Lambda Function called "getSubmissionData" that serves as a search to return all submitted files from the database. We saw that our changes had been deployed.



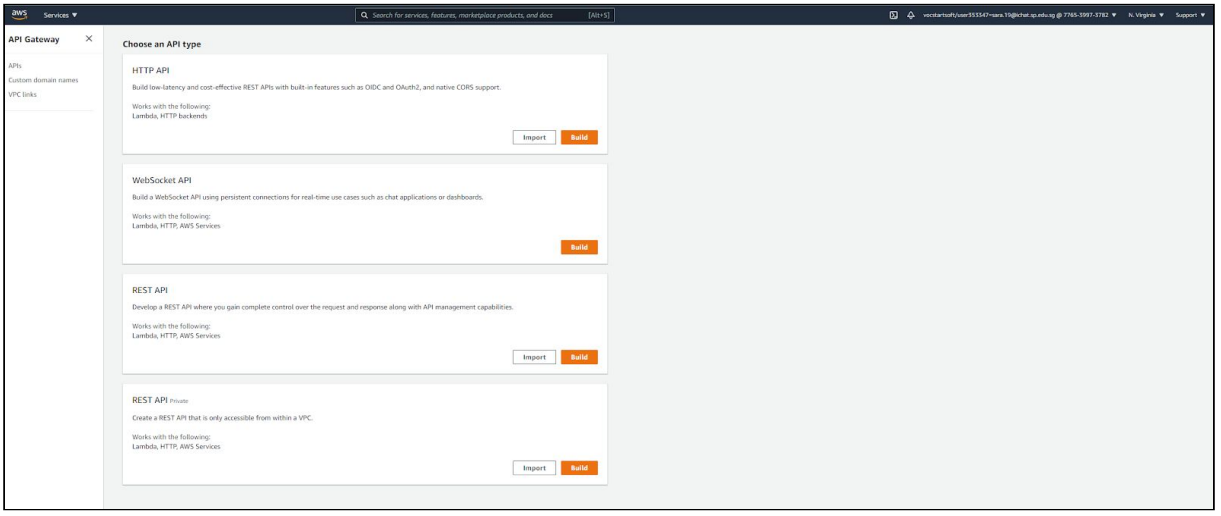
Step 2: We then proceed to configure a test event in order to test on the lambda function. After that, we executed the result. Based on the below screenshot, we can see that the results are printed out accurately.



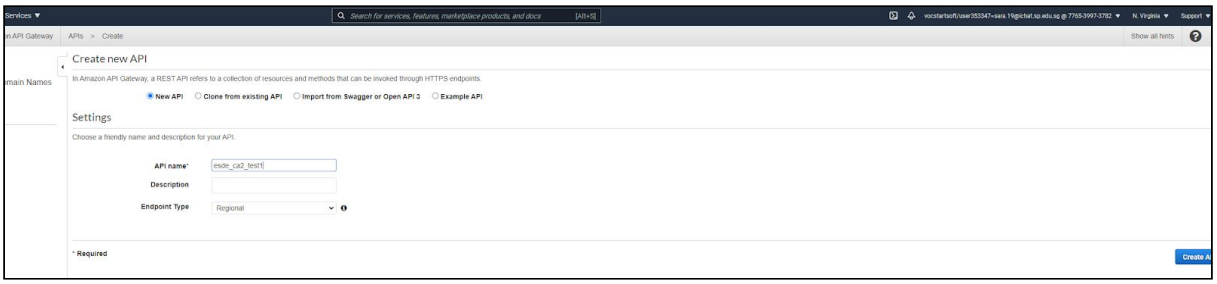


4.3 Creation of AWS Web API Gateway

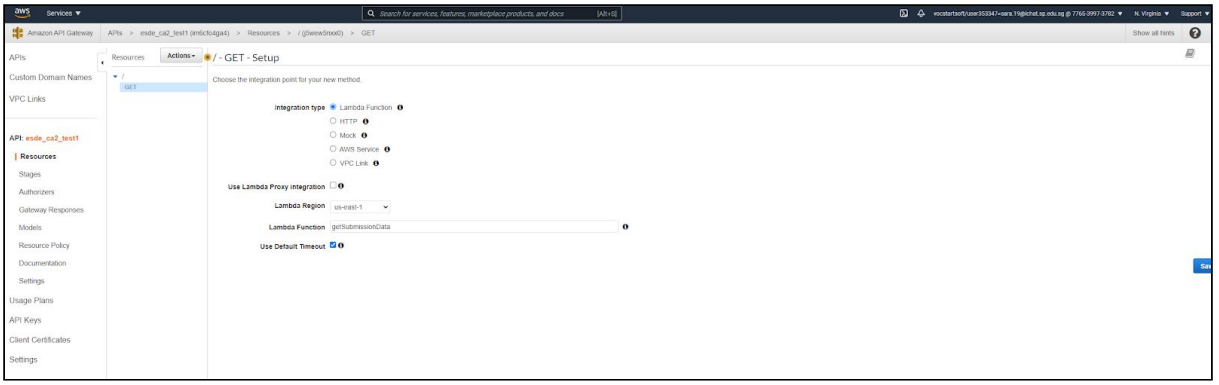
Step 1: After logging into our AWS account, we navigated to the Amazon API Gateway console to select the API type. We chose to “build” a REST API that works with Lambda, HTTP and other AWS services.



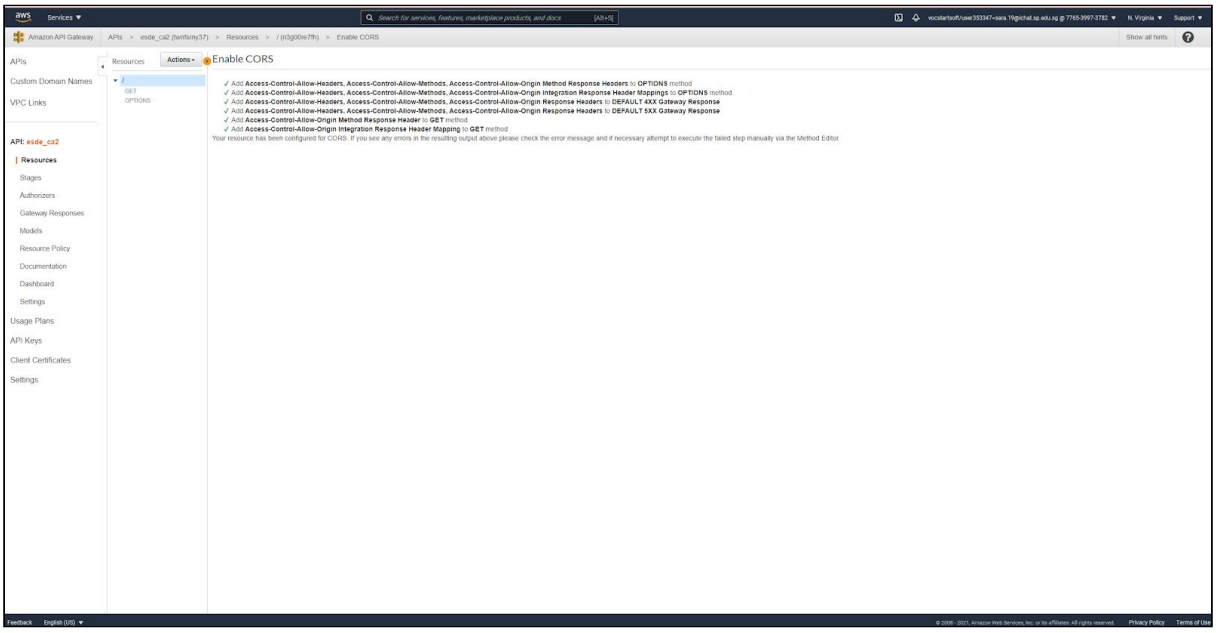
Step 2: We selected the option to create a “new API” and named it “esde_ca2_test1”.



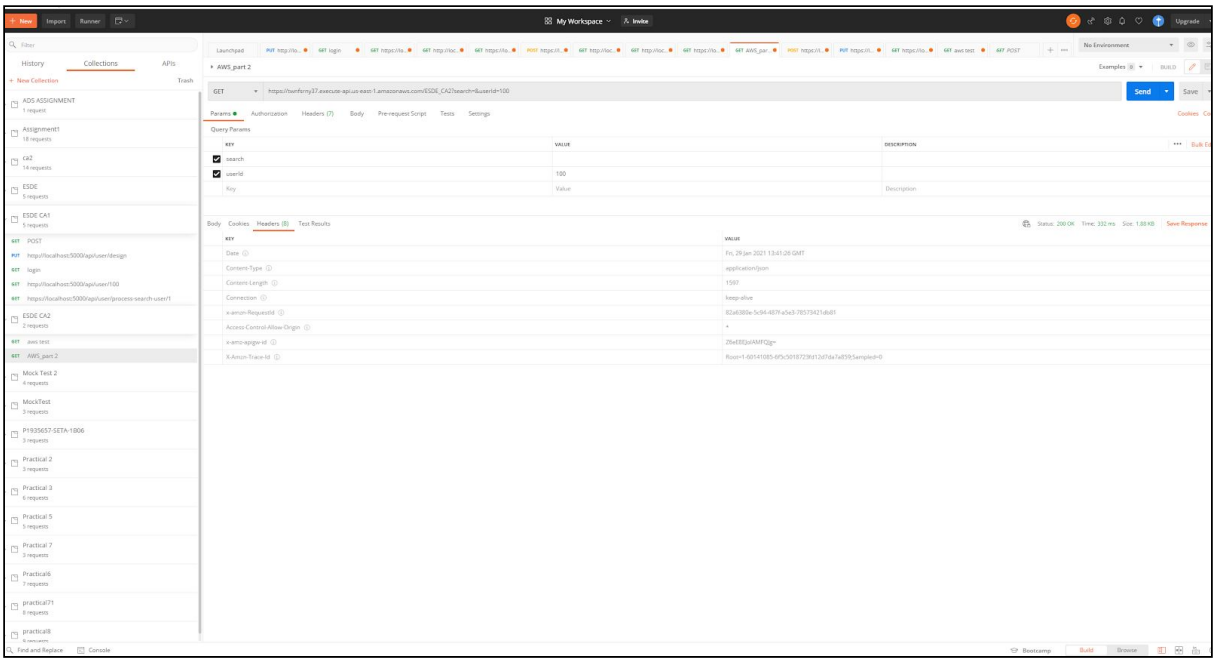
Step 3: We clicked on the name of the created API and selected “Lambda Function” as the integration type.



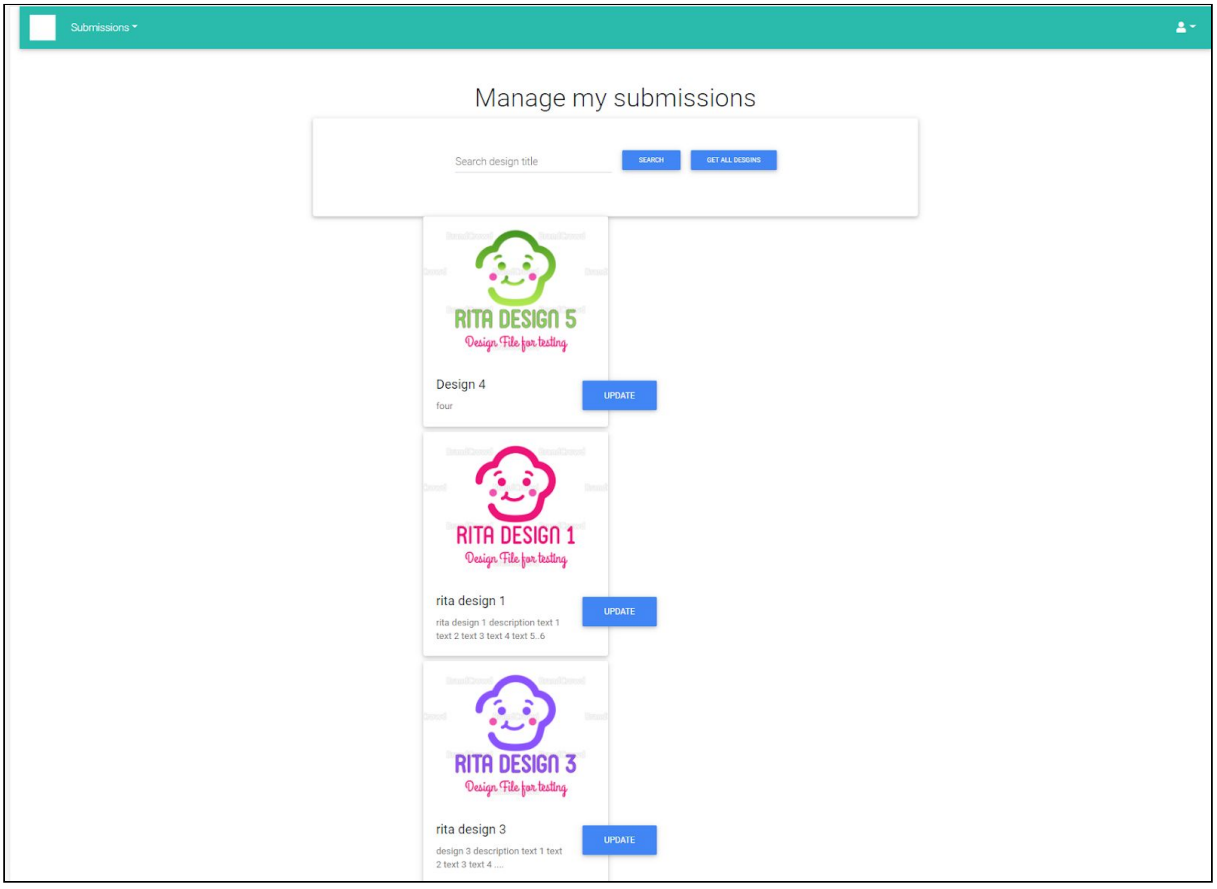
Step 4: Above GET, we choose the resource / then select “actions”, “enable CORS” and “yes” to replace existing CORS headers. We applied the default settings and only ticked the checkboxes for Default 4XX and Default 5XX. We then ignored the warnings and proceeded to click the “deploy API” button to deploy it into the production stage.



Step 5: Using Postman, we tested the serverless application by entering the Amazon Gateway API endpoint to get all submission files for Rita whose user id is 100. As seen below, the headers successfully returned an application/json type of response.



Step 6: We navigated back to our Amazon S3 website in the browser. As seen below, we were able to view the design submission files of the user Rita upon entering the query string.



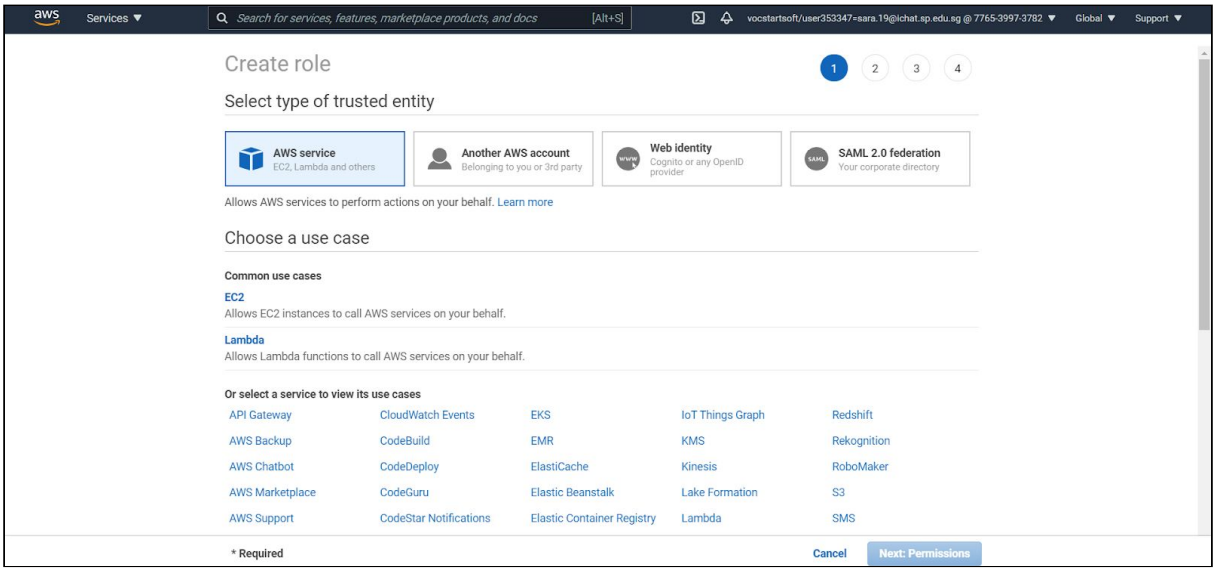
5. Cloud Security Features

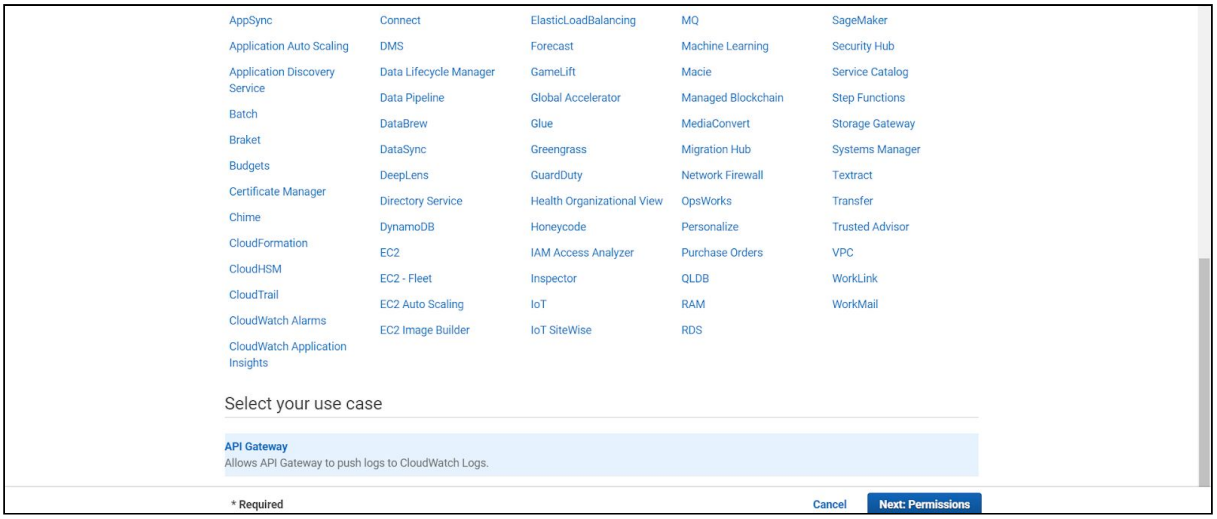
In this section, we will be addressing the design and implementation of 4 such cloud security features in our web application.

5.1 Centralised Logging for Web API Access

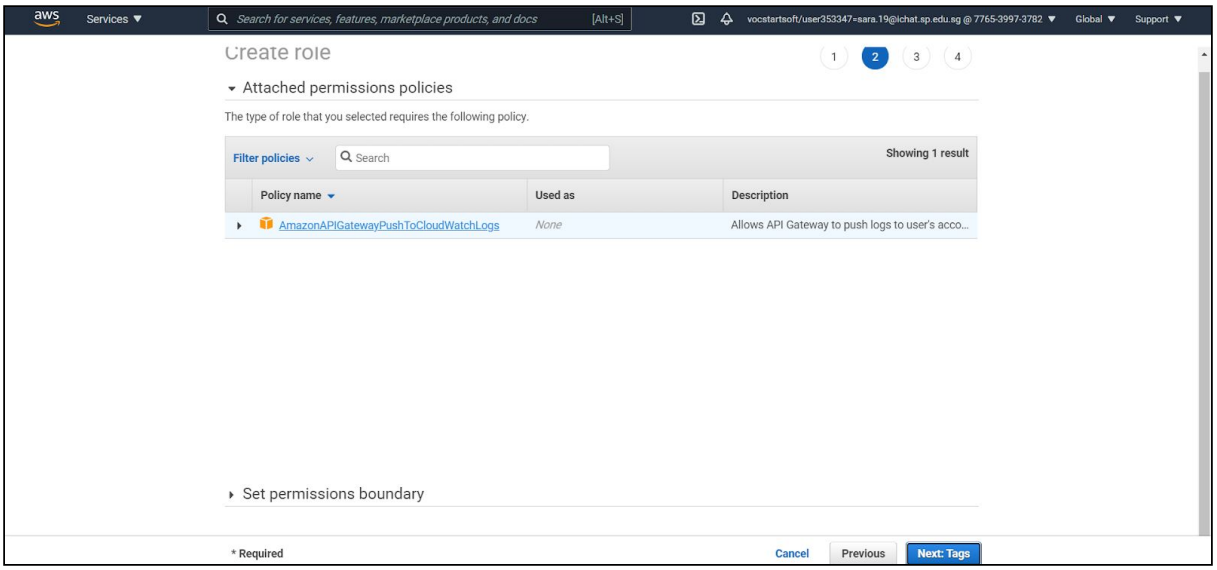
In this section, we will be showing a step-by-step approach on how to setup CloudWatch API logging using the API Gateway console.

Step 1: After logging into our AWS account, we navigated to the IAM dashboard and clicked on “Roles” at the side navigation panel. We selected “AWS Service” as the type of trusted entity and “API Gateway” as the service that will use the newly created role. We proceeded to click the “next” button to edit the permissions.

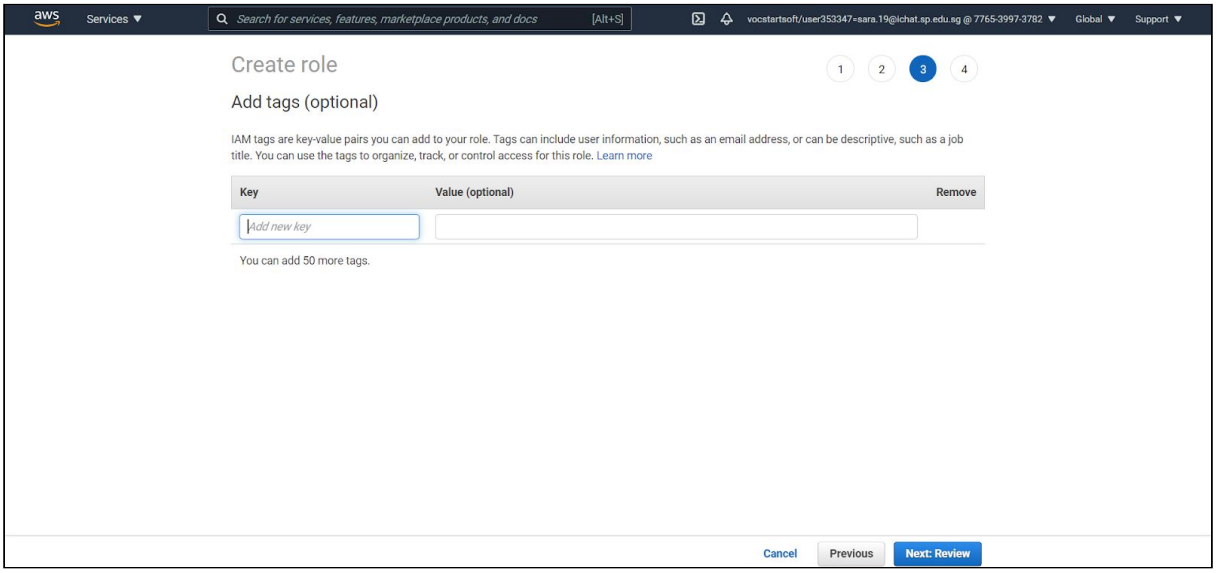




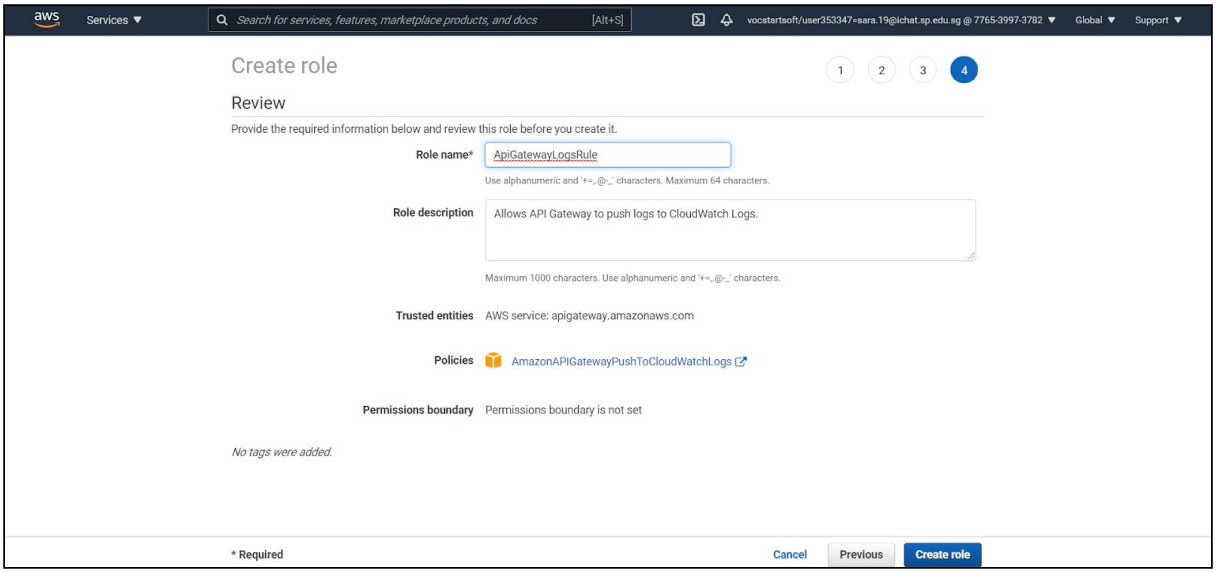
Step 2: We clicked on the policy name that appeared. By establishing API Gateway as the use case, this service will be entitled to push logs to CloudWatch. We proceeded to click the “next” button to view the settings for tags.



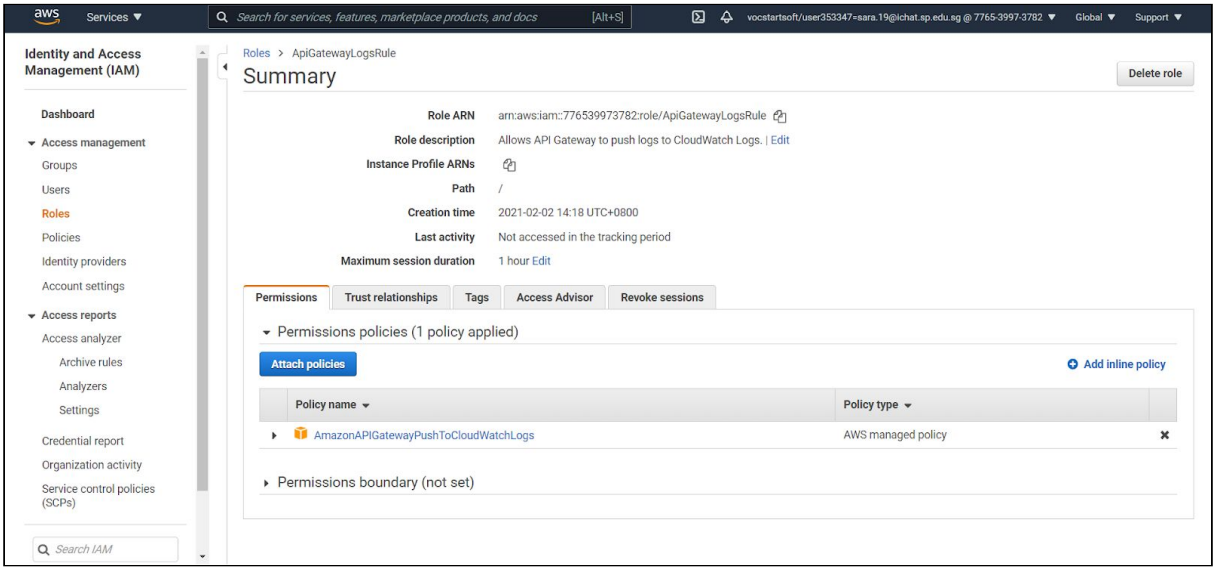
Step 3: Since adding tags is an optional step, we applied the default settings for it and proceeded to click on the “next” button to review the summary of the role created.



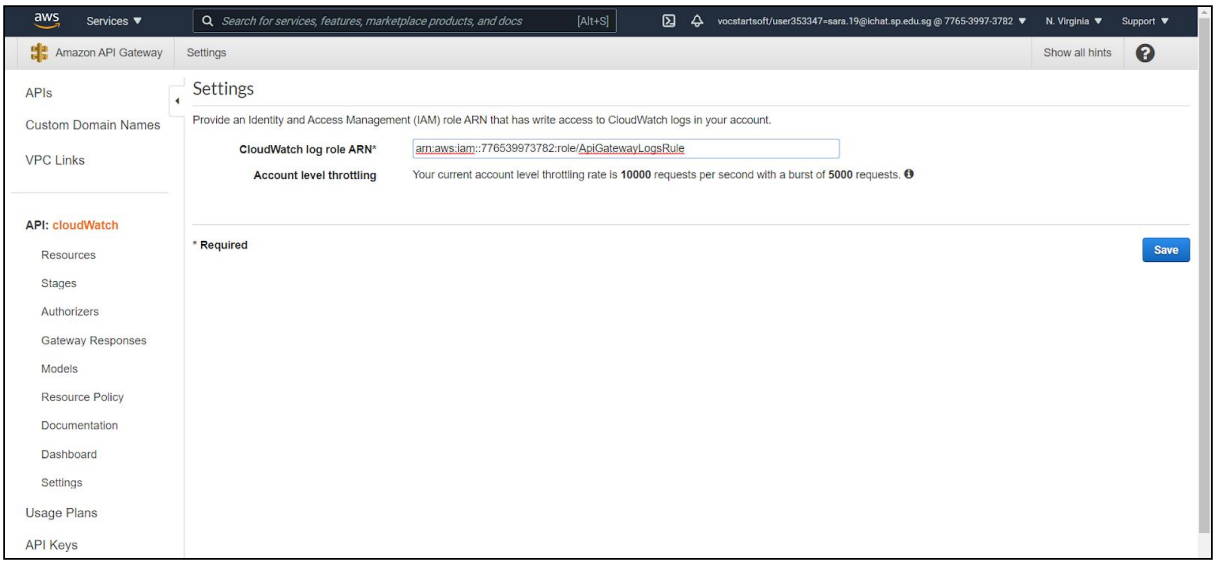
Step 4: We named our role as “ApiGatewayLogsRule” and applied the default role description. We proceeded to click on the “next” button to confirm the role creation.



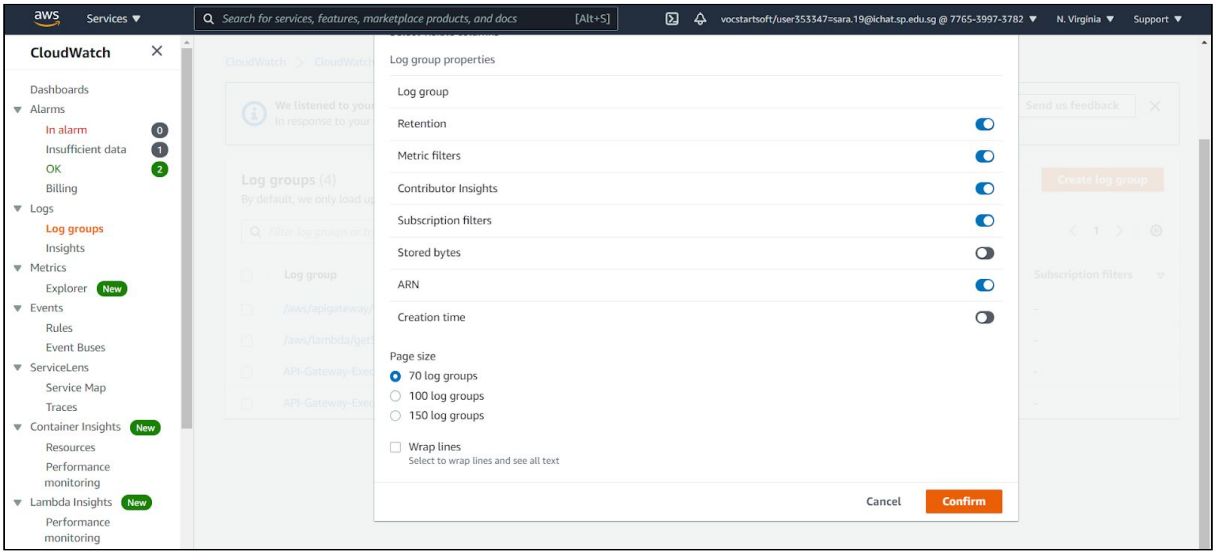
Step 5: We viewed the summary of the role and took note of the IAM’s Role ARN which needs to be transferred over to the API Gateway console to enable access logging later on.



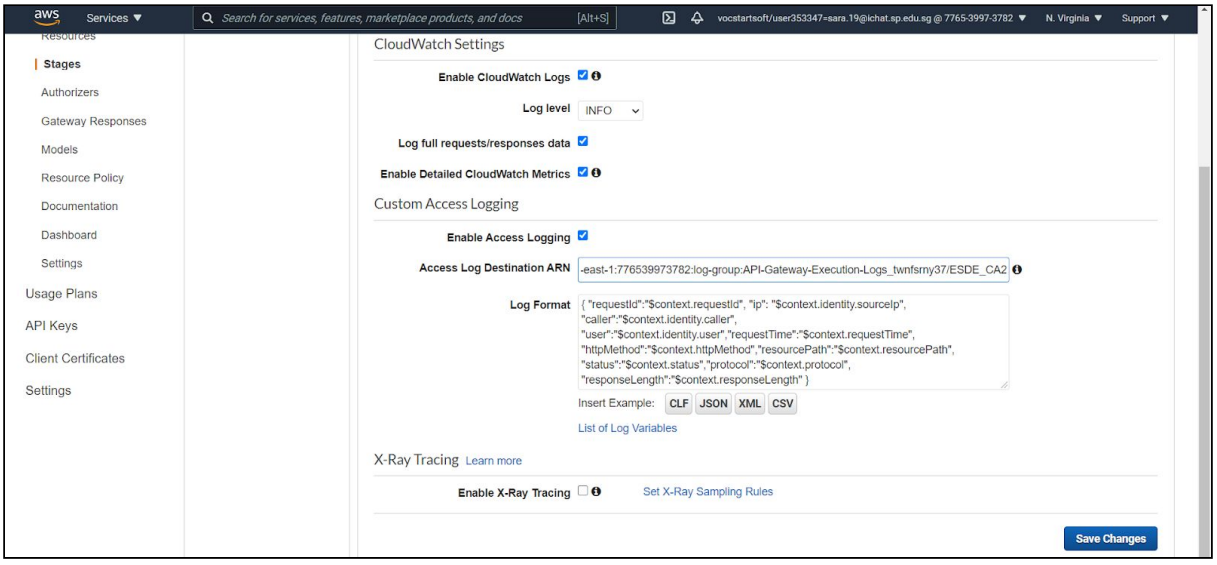
Step 6: We navigated to the API Gateway dashboard and clicked on the settings in the side navigation panel and transferred over the Role ARN to fill in the CloudWatch log role ARN before clicking the “save” button to confirm the changes.



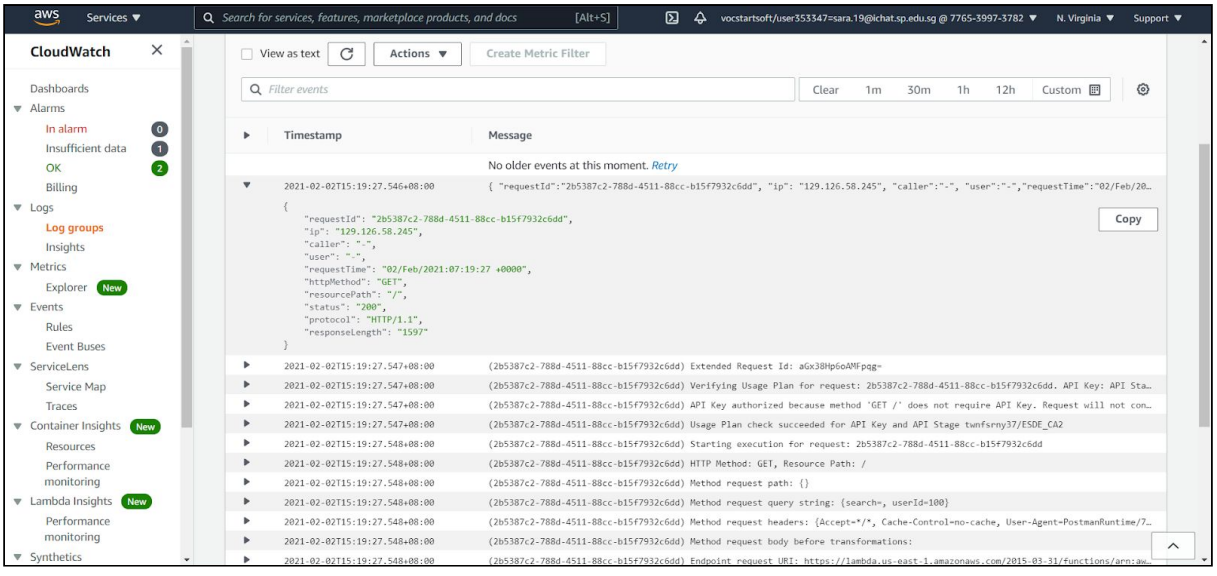
Step 7: We navigated back to the CloudWatch dashboard and clicked on the settings in the side navigation panel to change the ARN column from “hidden” to “visible” so that we can see the log destination of the Access Log Destination ARN and strictly follow its format of arn:aws:logs:{region}:{account-id}:log-group:log-group-name.



Step 8: We clicked on the previously created API called “esde_ca2_test” to access its stage editor and “logs/tracing” tab section. Under the CloudWatch settings, we selected “Info” instead of “Error” from the dropdown menu and ticked the checkboxes to enable CloudWatch logs and log full requests and responses data. Under the Custom Access Logging, we transferred over ARN from the previous step and opted for the log to be printed in JSON format. We proceeded to click on the “save changes” button to finish the setup.



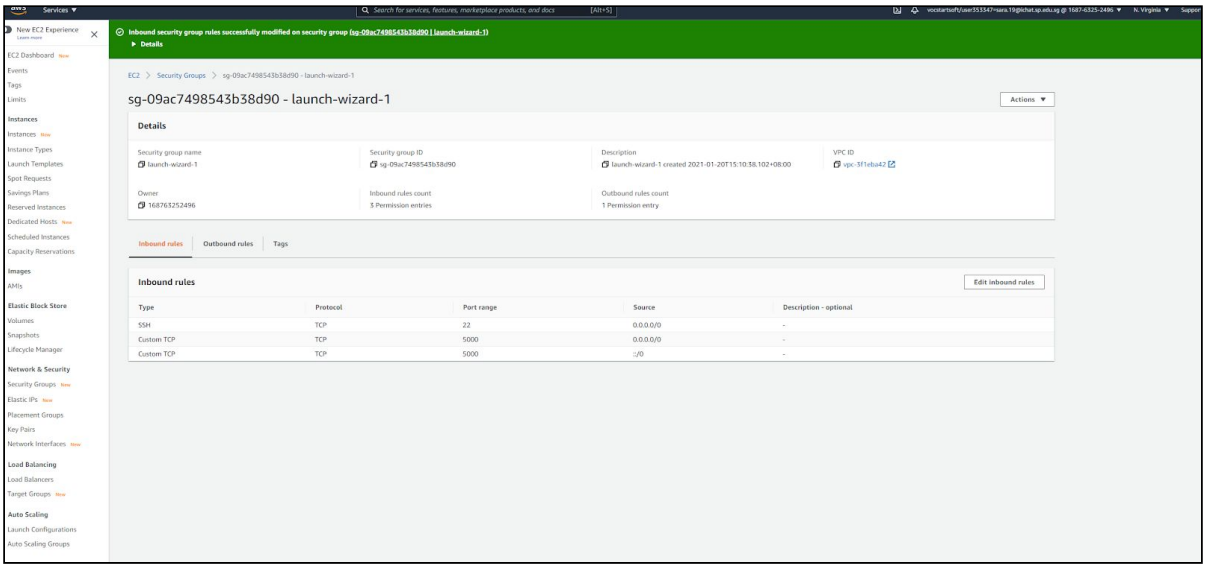
Step 9: We navigated back to the CloudWatch dashboard and clicked on the log group for the above mentioned “esde_ca2_test” API. We are able to view the following metrics for “where”, “when”, “why” and “how” in the logging. For example, in the most recent log for getting all submissions, the details included 129.126.58.245 as the source IP address, the request time made at 3.19pm, the HTTP method and resource path called “GET” and “/”respectively and the response length is 1597.



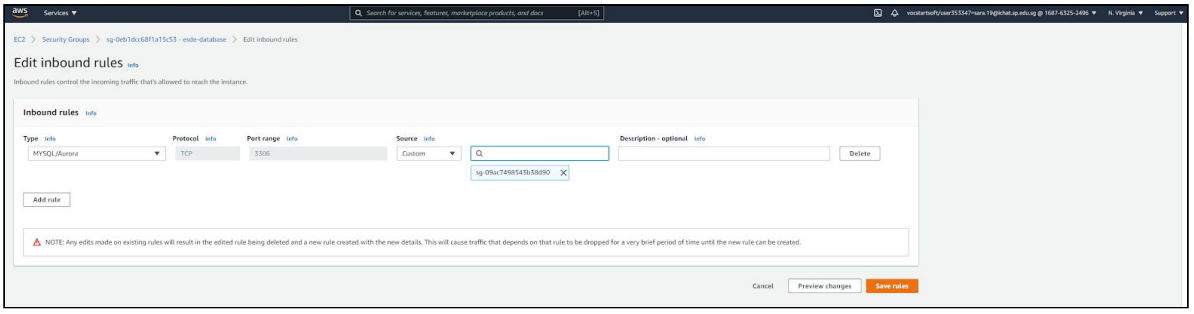
5.2 MySQL RDS Database Security

In this section, we will be showing a step-by-step approach on the features we implemented to make our MySQL RDS Database more secure.

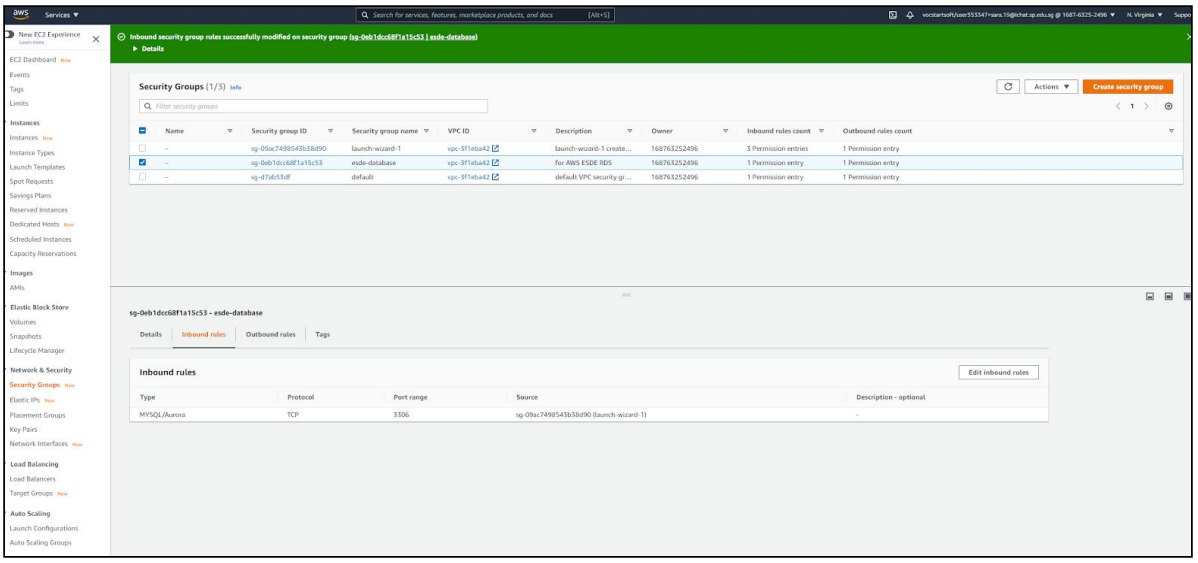
Step 1: After logging into our AWS account, we navigated to the EC2 dashboard and clicked on the security groups for the previously launched instance. We see that the inbound rules for it involve only the TCP protocol of SSH and the EC2 instance running on port 22 and 5000 respectively. This means that to access the RDS Database, the instance must also run NodeJS codes.



Step 2: After seeing that the source is set to 0.0.0.0/0 for the inbound rule of RDS Database, we realised that this is not a good measure as the contents are publicly available and anyone can have read-write access to it anywhere. Thus, to ensure the least amount of privileges for the database connection credentials, we proceeded to click on the “edit inbound rule” button to customize the source from the previous 0.0.0.0/0 to only allow the user account’s number.



Step 3: We see that the details for the security groups of the RDS Database has been successfully modified to allow only 1 permission entry for all inbound and outbound rules. This verifies that the data stored within it is secured.

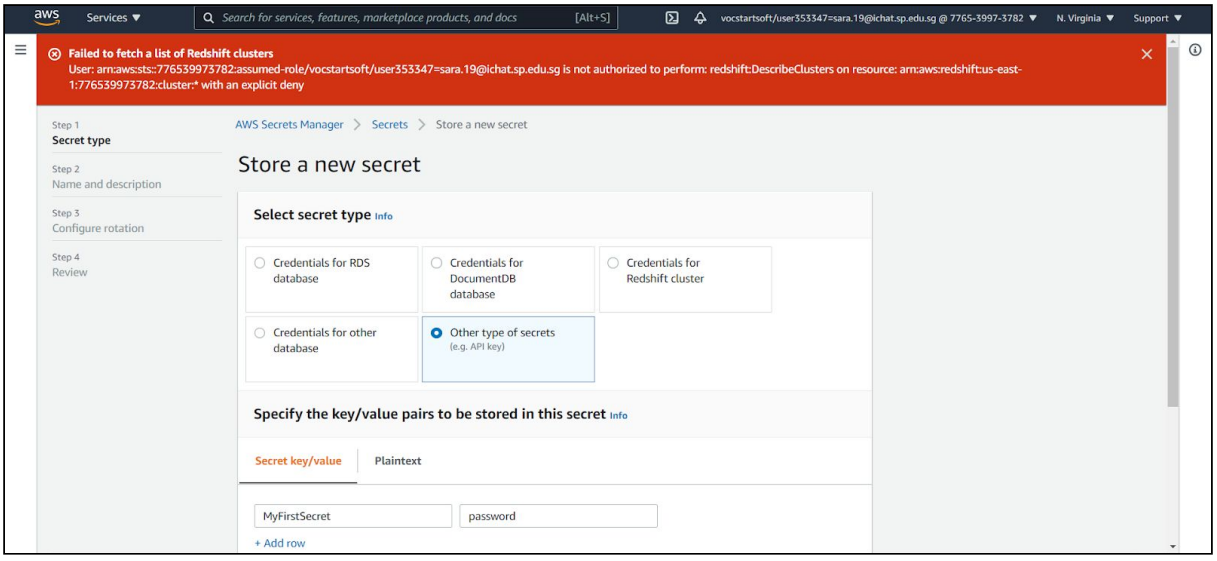


Step 4: We navigated to our terminal in Visual Studio Code and saw that our access to the database has been logged where the results are successfully retrieved after a TCP protocol handshake is established to call the connection. As shown from the screenshot, we could not run it on localhost due to permission blocked issues that we created.

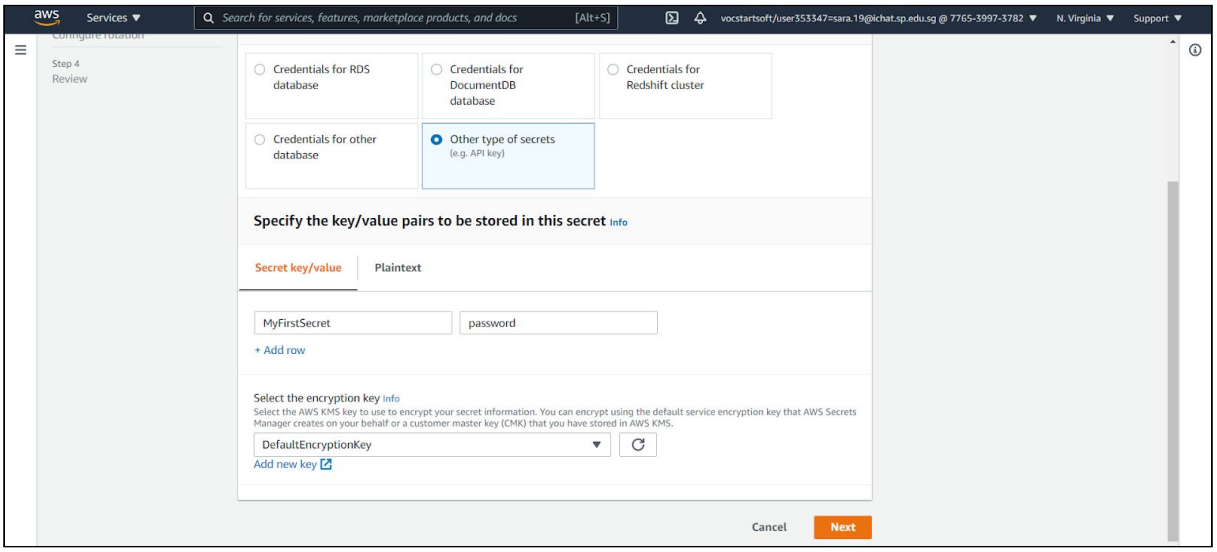
5.3 Storing Private Key using AWS Secrets Manager

In this section, we will be showing a step-by-step approach on how to create and store secret keys in AWS Secrets Manager, and retrieve it using its console. By doing so, the security aspect of our web application is improved as we can manage access to our private keys through the implementation of IAM's fine-grained created policies. This includes secret information, such as credentials, connection details, key name and value string pairs.

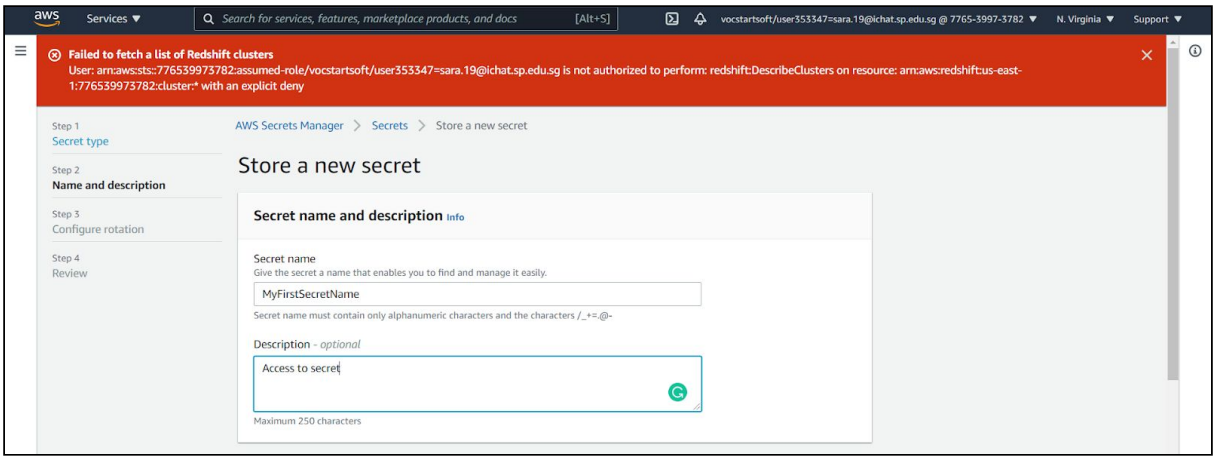
Step 1: After logging into our AWS **API Gateway** account, we navigated to the Secrets Manager dashboard and clicked on the "Store a new secret" button. We selected the option of "other type of secrets" as it has no relation to our database credentials.

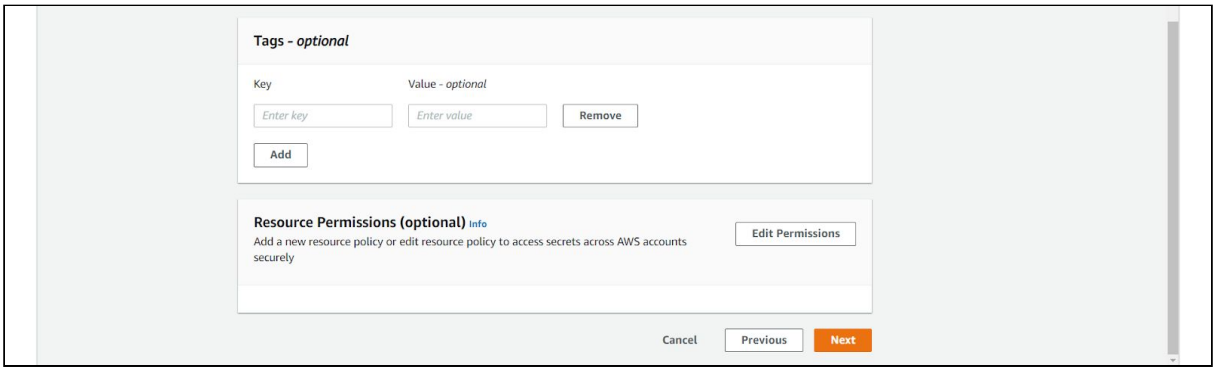


Step 2: Under the first field for key/value pairs to be stored, we specified our secret as “MyFirstSecret” and configured a password by entering “password” in the next field. Under the section to select the encryption key, we chose the option of “DefaultEncryptionKey”. Using this cost-effective option where AWS Secrets Manager provides the standard encryption service at no charge, compared to using a custom key, we can spend our $50 credits wisely. We proceeded to click the “next” button to set the name, description and tags.

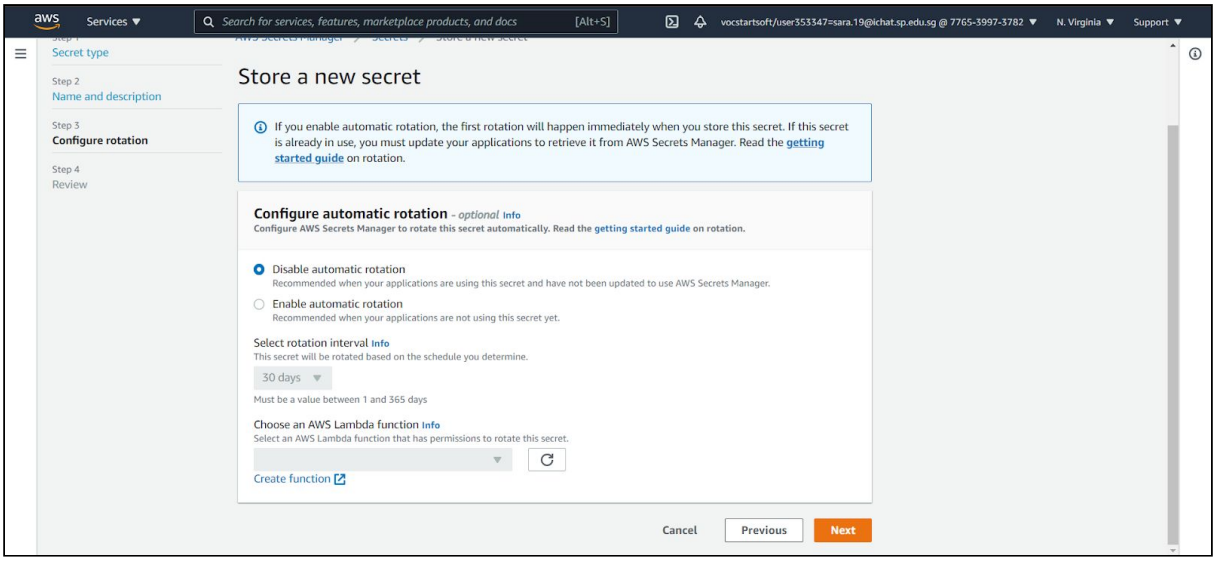


Step 3: We named our secret “MyFirstSecretName” and described its purpose to allow us “access to secret”. Since tags is an optional thing, we applied the default settings to have no tags and proceeded to click on the “next” button to configure rotation.

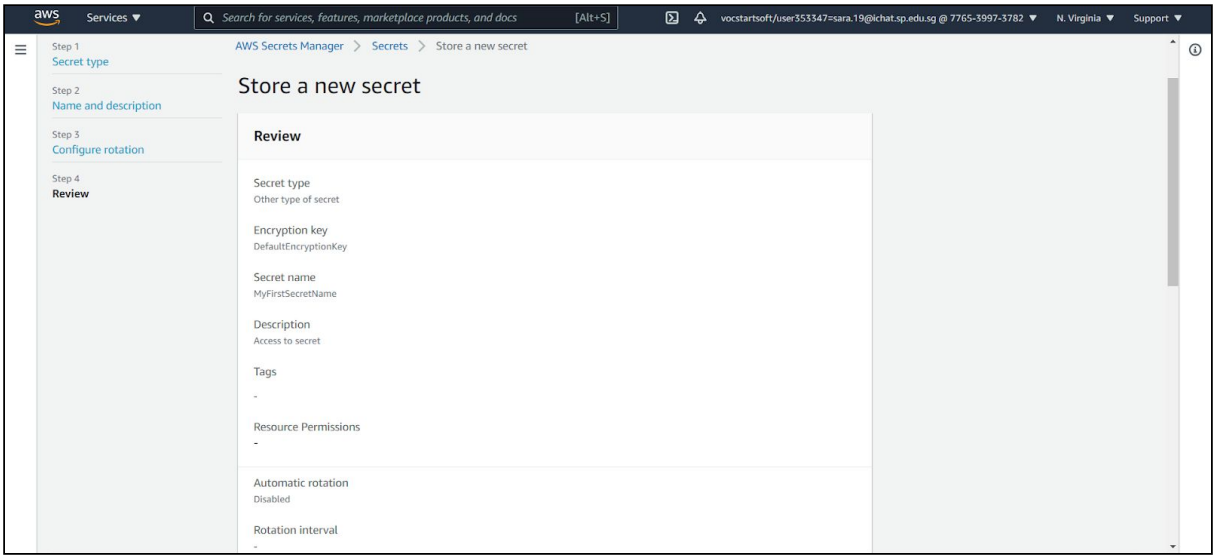


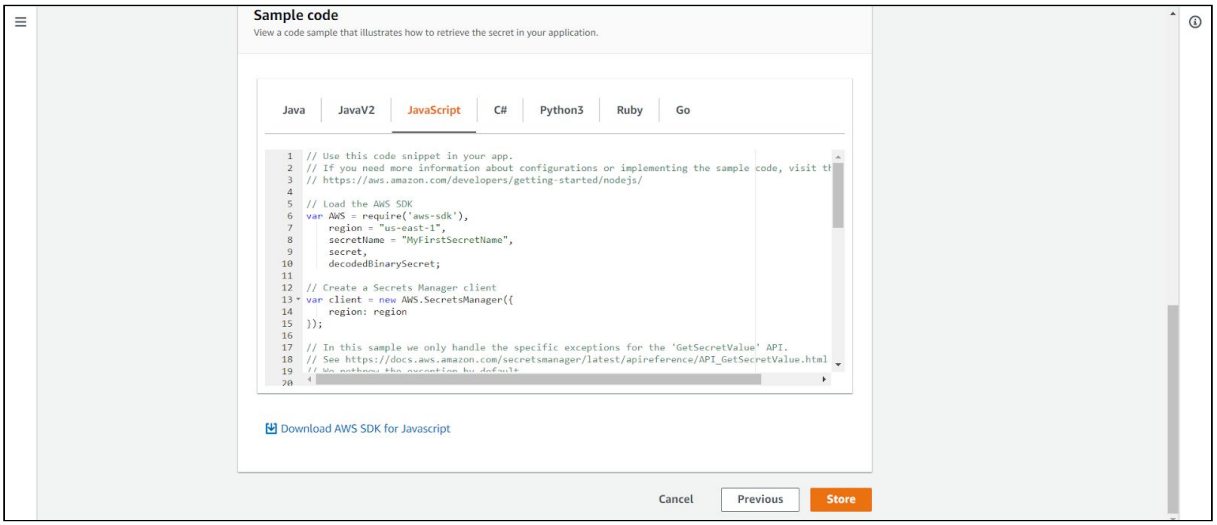


Step 4: We selected the “disable” option when configuring automatic rotation and did not choose any Lambda Function as we did not provide database credentials as our secret and so there is no need to invoke any functions along with it.

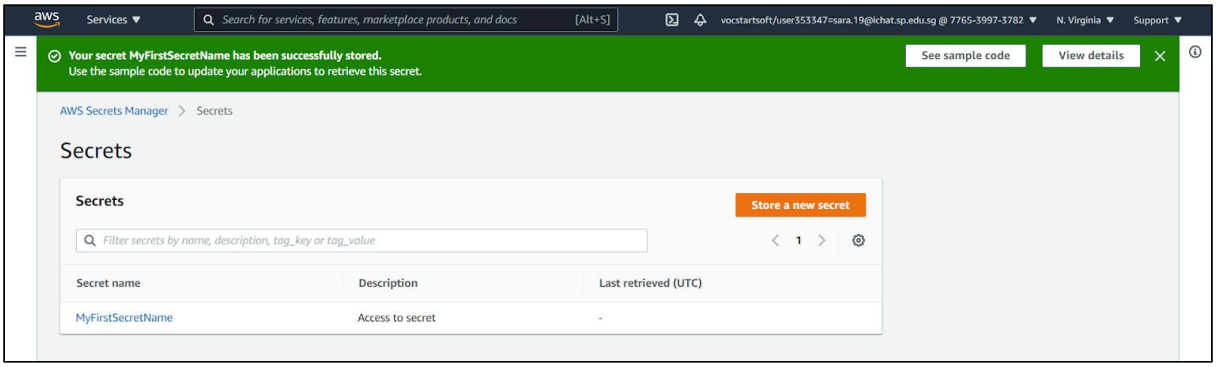


Step 5: We proceeded to click on the “next” button at the bottom of the previous page to review the summary of the created secret. We also viewed the code snippet to retrieve the secret using an API written in Javascript language but decided not to implement it within our codes as we have already uploaded it to putty previously. We proceeded to click the “store” button to save the secret.





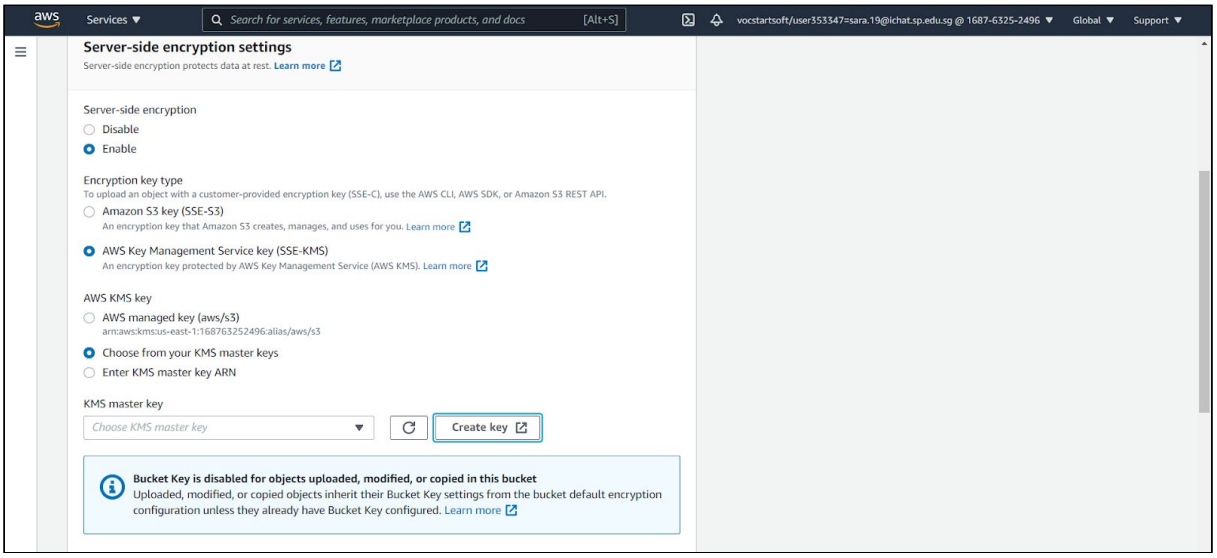
Step 6: We managed to successfully store the secret in AWS Secrets Manager as seen below.



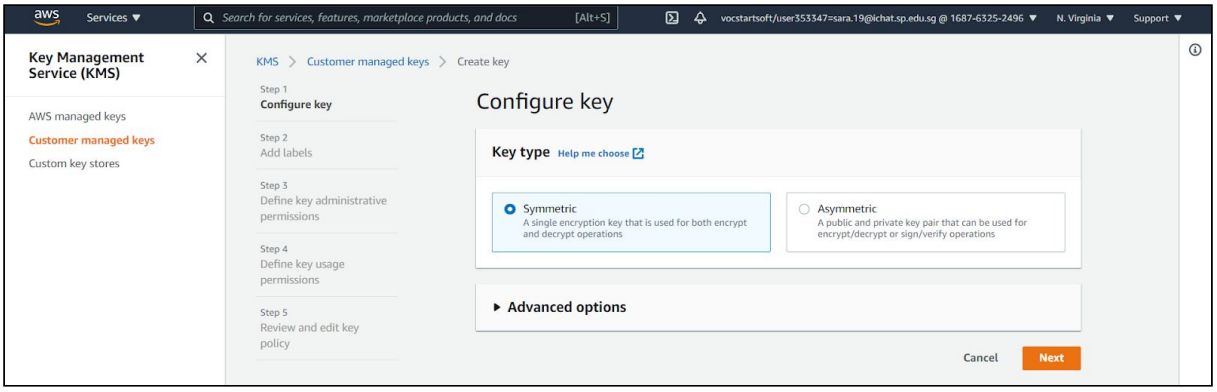
5.4 Server Side Encryption (SSE) on S3 Bucket

In this section, we will be showing a step-by-step approach on how to enhance the security of an S3 Bucket by using AWS Key Management Service (KMS) to encrypt the data within it. By doing so, the data is transformed into a way that can only be read by the administrator who has the correct decryption key, which further prevents unauthorized access to sensitive information such as private keys.

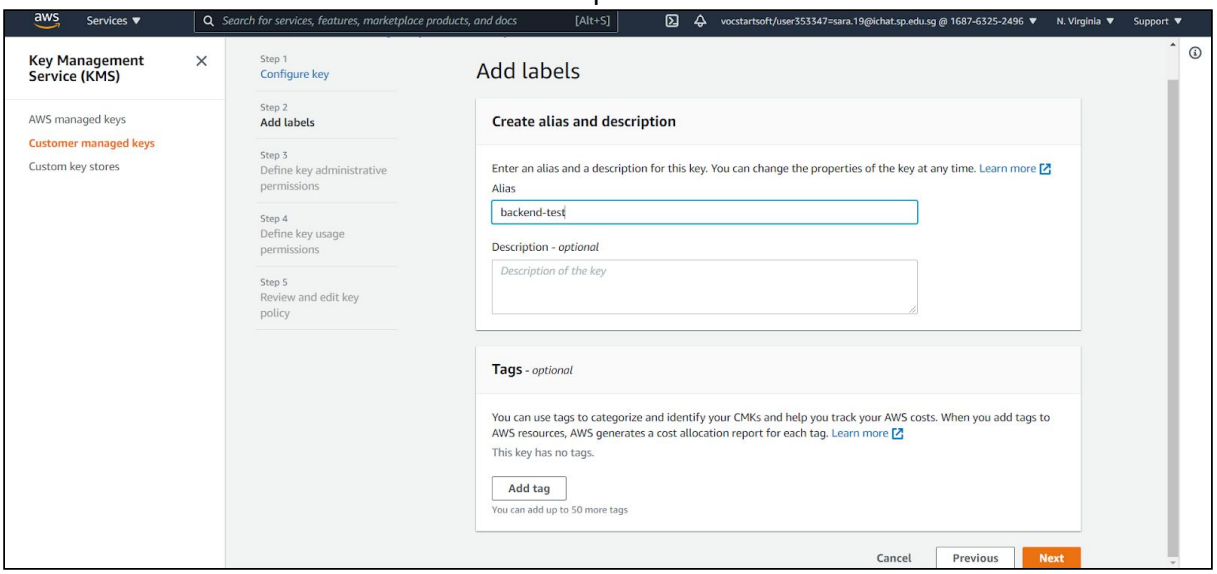
Step 1: After logging into our AWS **main** account, we navigated to the S3 dashboard and clicked on the name of the previously created bucket for the backend zipped folder. We navigated to its server-side encryption settings, clicked on “enable” and selected “SSE-KMS” for the type. As we do not have an existing KMS master key, we proceeded to click on the “create key” button to create the key policy.



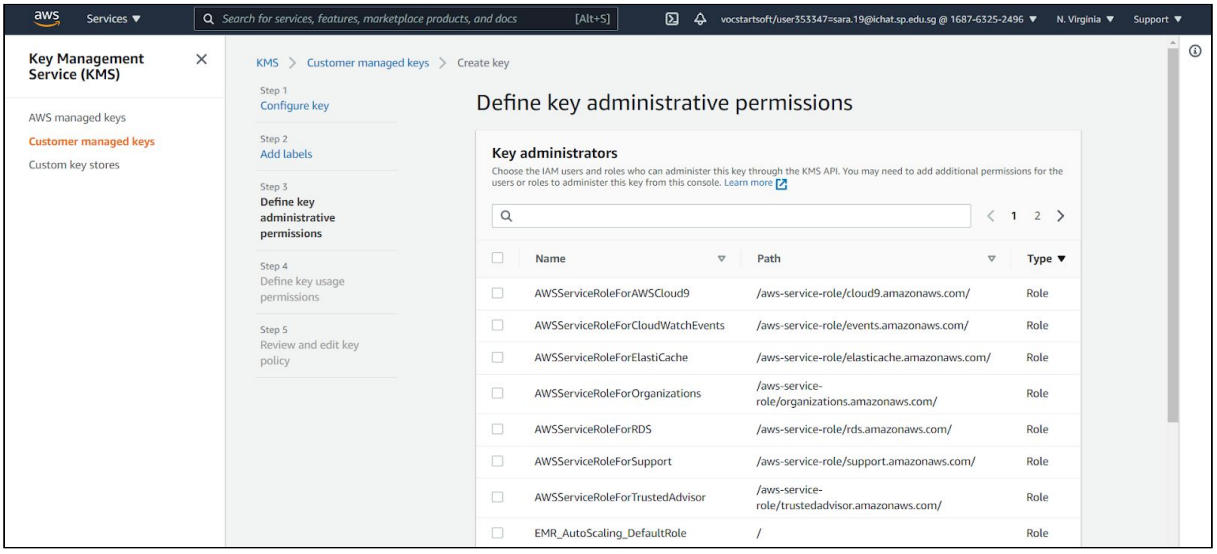
Step 2: We were redirected to the KMS dashboard where we had to decide between a symmetric or asymmetric key type. The main difference is that symmetric encryption uses a single key that needs to be shared among the users who receive the message while asymmetric encryption uses a pair of public and private keys to encrypt and decrypt messages when communicating. Thus, we chose to configure a symmetric key type as it takes a faster time to set up and proceeded to ignore the advanced options and click on the “next” button to add labels.

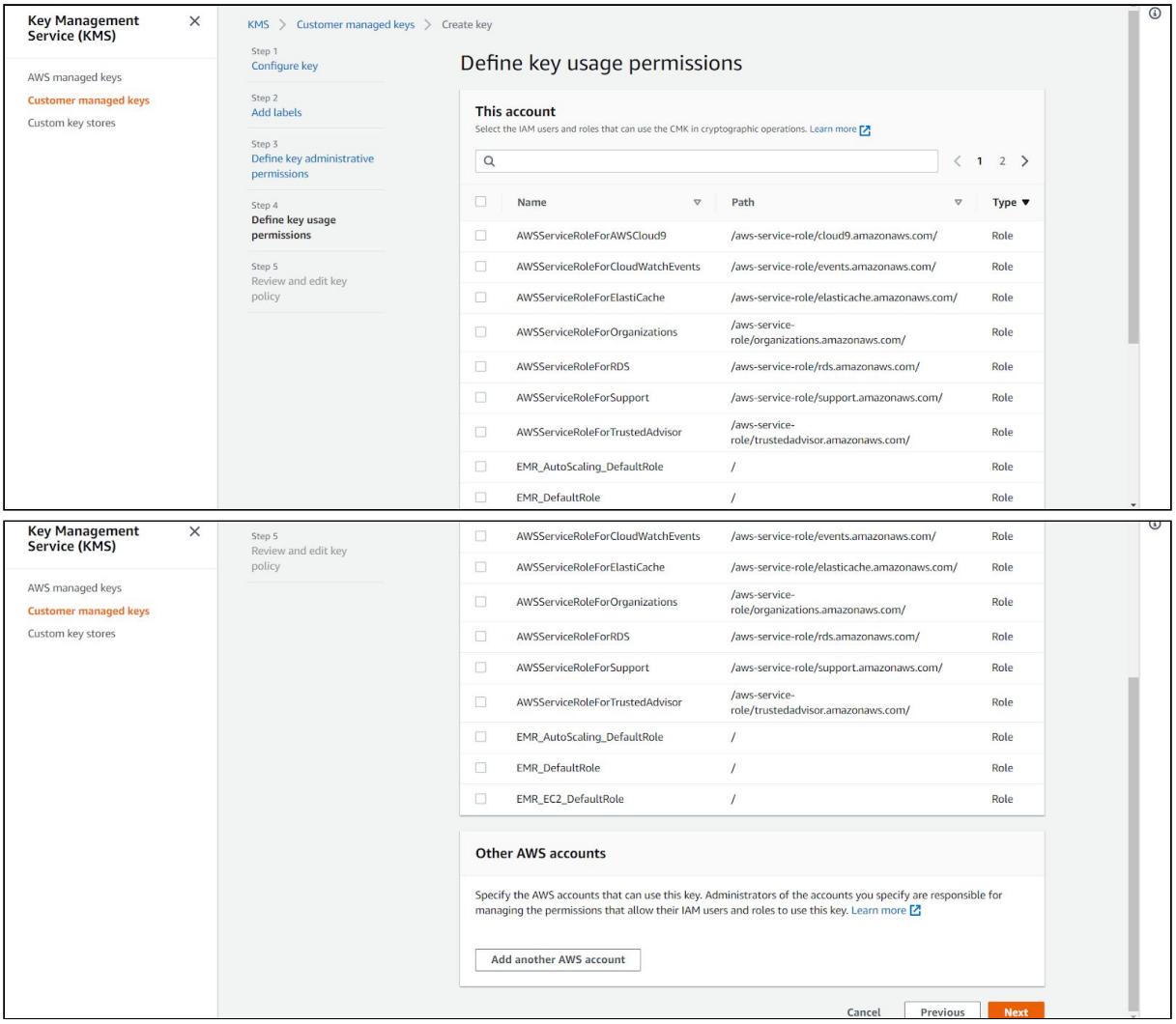


Step 3: In the alias field, we named our key “backend-test” so that it will be easier to identify which part of the application the key is encrypting. Since the description and tags were optional, we just skipped the settings for it and applied the default. We proceeded to click the “next” button to define administrator and user permissions.

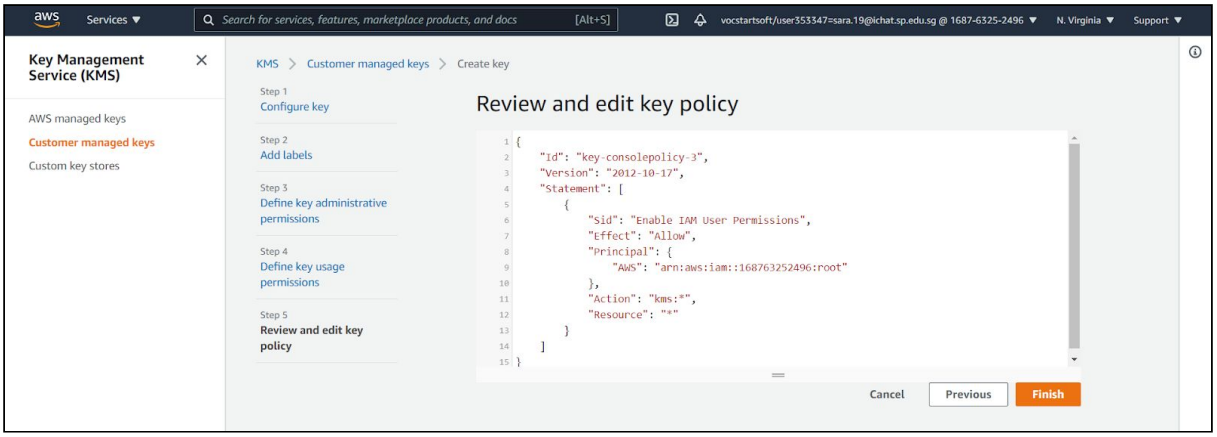


Step 4: For the administrative and usage permissions, we simply skipped through them as we already have other existing IAM policies that can manage our roles. We proceeded to click the “next” button to review the key policy created.

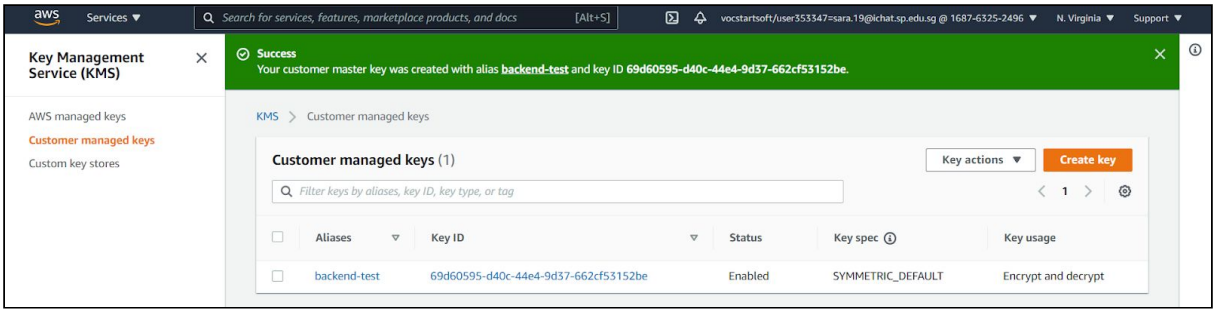




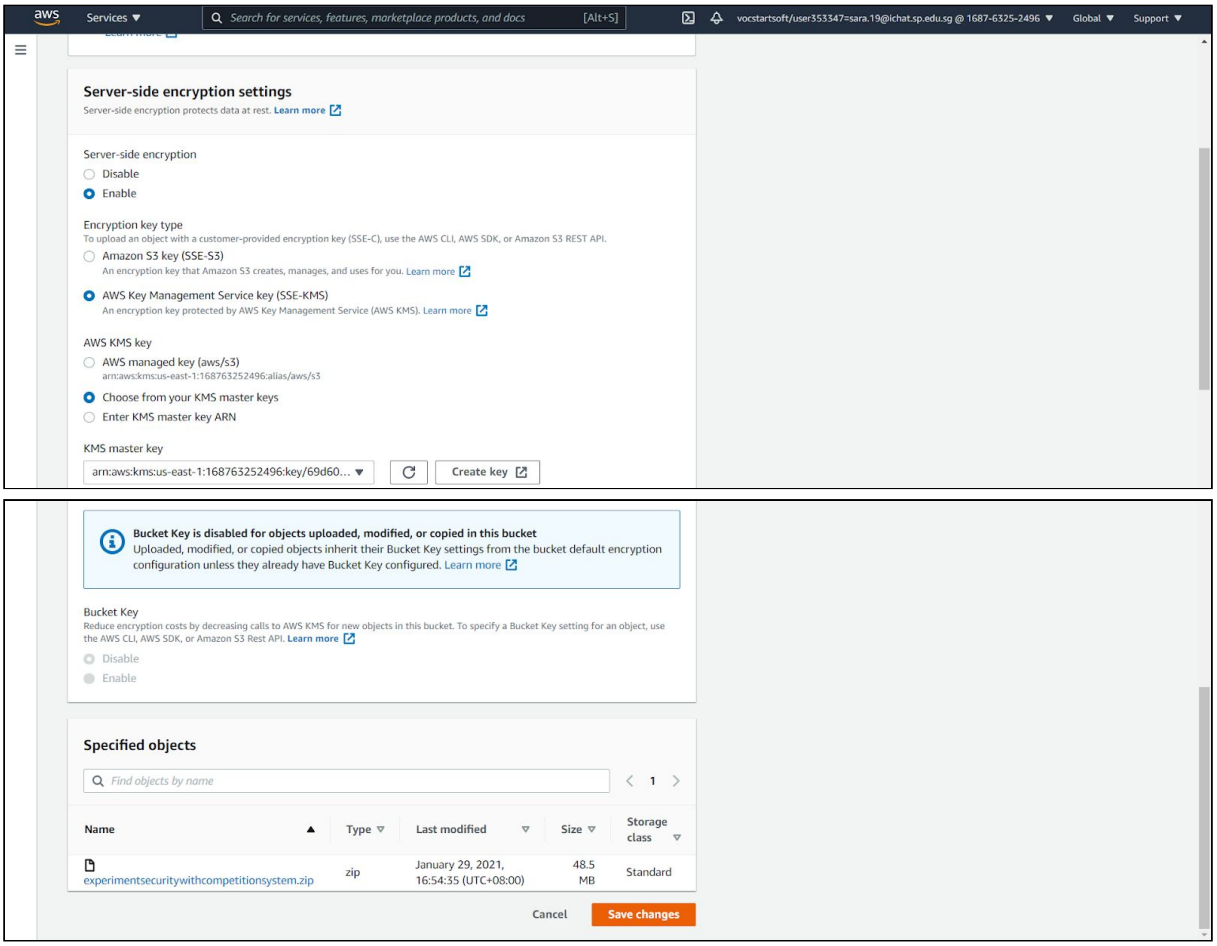
Step 5: After viewing the key policy, we proceeded to click the “finish” button to complete the policy configuration process.



Step 6: We managed to successfully create a customer master key of our own for the S3 bucket used to store the backend of our web application as seen below.



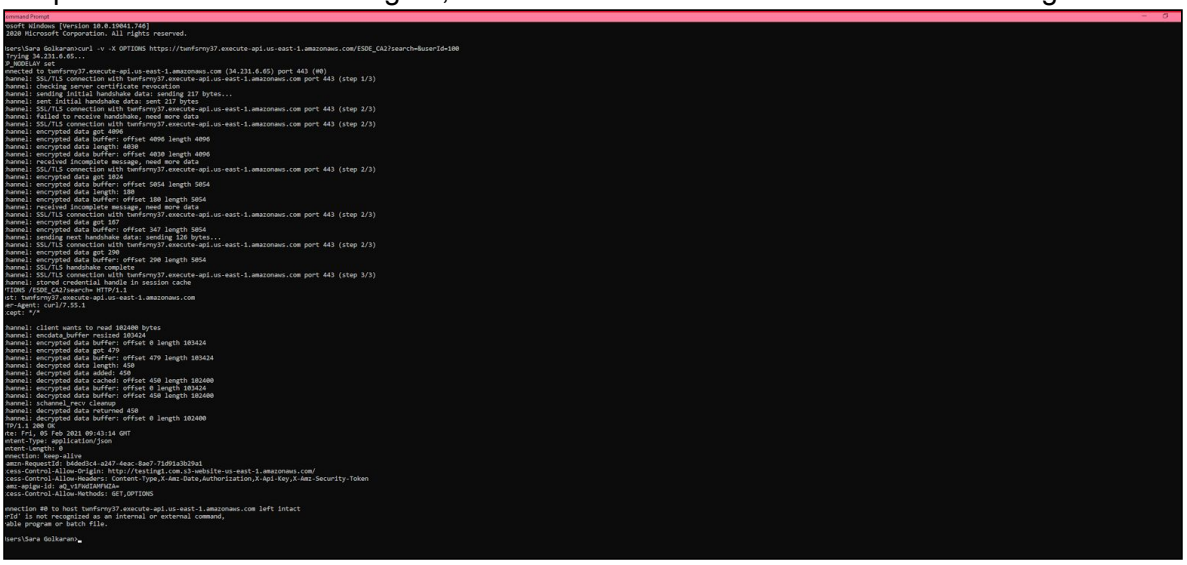
Step 7: We navigated back to the server-side encryption settings at the S3 dashboard and selected the option to “choose from our existing master key” so that we can upload our recently created “backend-test” master key. We proceeded to click on the “save changes” button to finalise the master key setup.



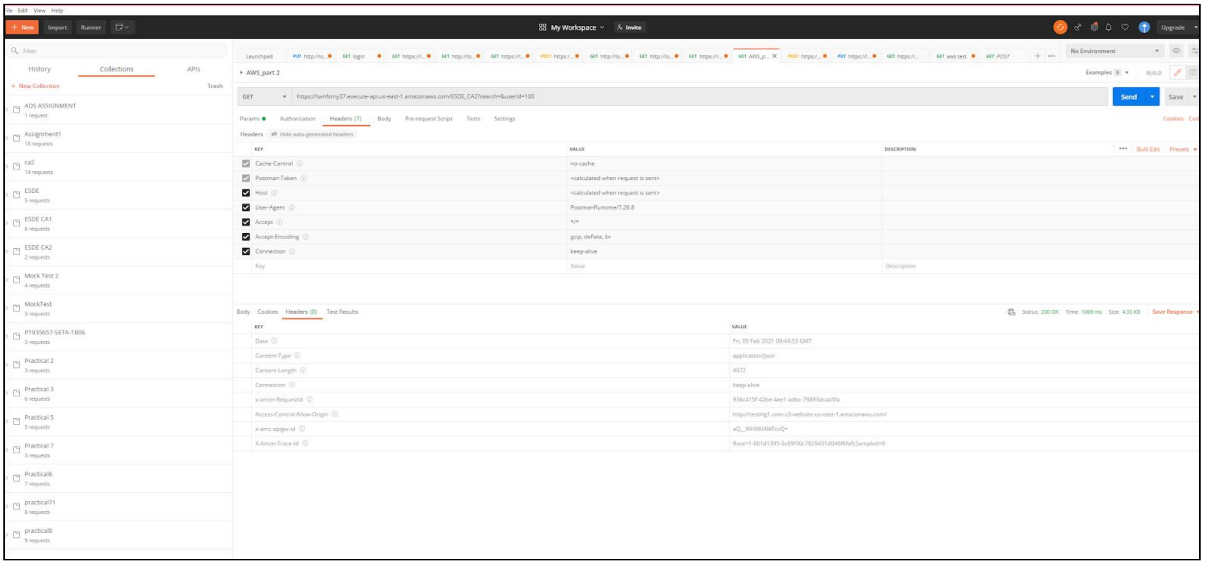
5.5 Enforcing CORS

In this section, we will be showing two pieces of evidence of CORS being employed to increase the security aspect of our web application in the cloud. Previously, CORS has caused us some problems for our query requests as our browser policy indicates that it only allows preflight requests. This means that it does not tolerate having our Amazon S3 website and API Gateway endpoints being hosted in different domains.

Step 1: In command prompt, we used the curl parameter to test the CORS OPTIONS request. We saw that our AWS host has completed the SSL/TLS handshake through the acceptance of curl as a user-agent, which activated the Access-Control-Allow Origin.



Step 2: In Postman, we ran the AWS API created previously to get all submissions data and saw that Access-Control-Allow-Origin has been activated for that endpoint too.



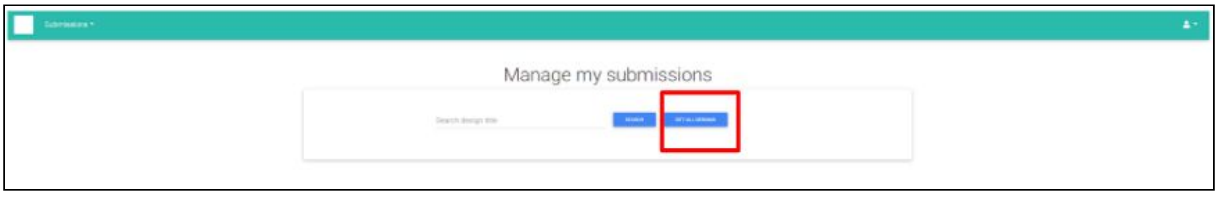
6. Additional Feature

6.1 Created Button in Frontend to execute Lambda Function

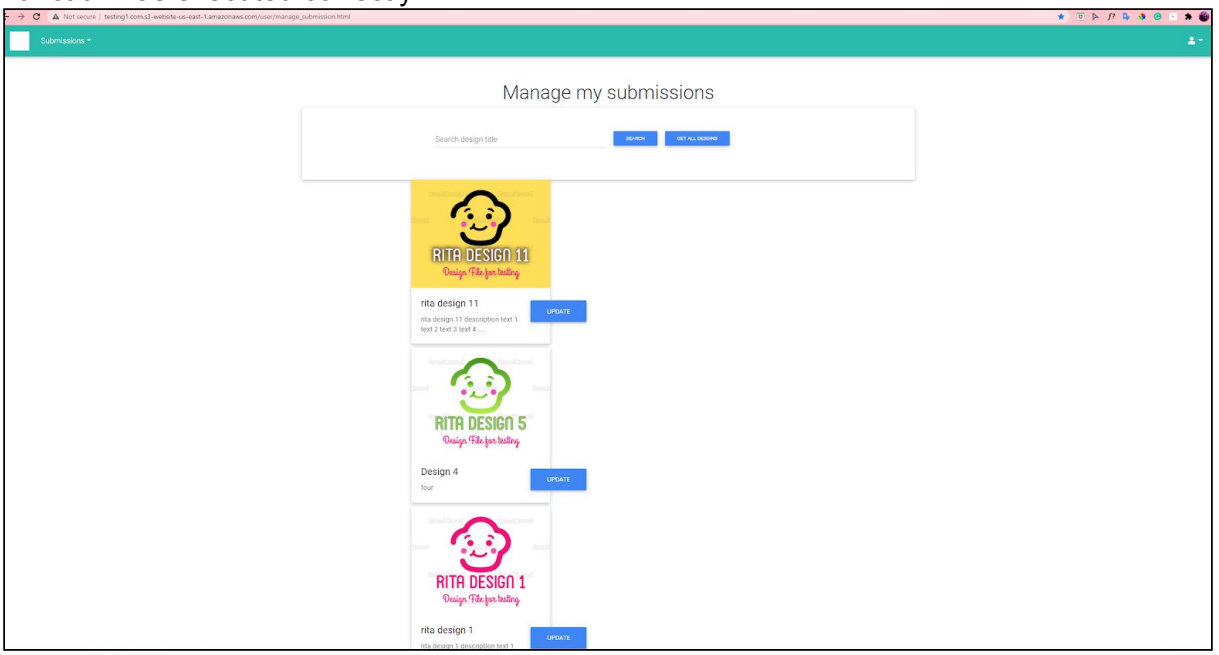
Step 1: We created a button called “get all designs” to execute the lambda function. The following image is the code snippet to print out all the submissions of Rita who is userId 100.

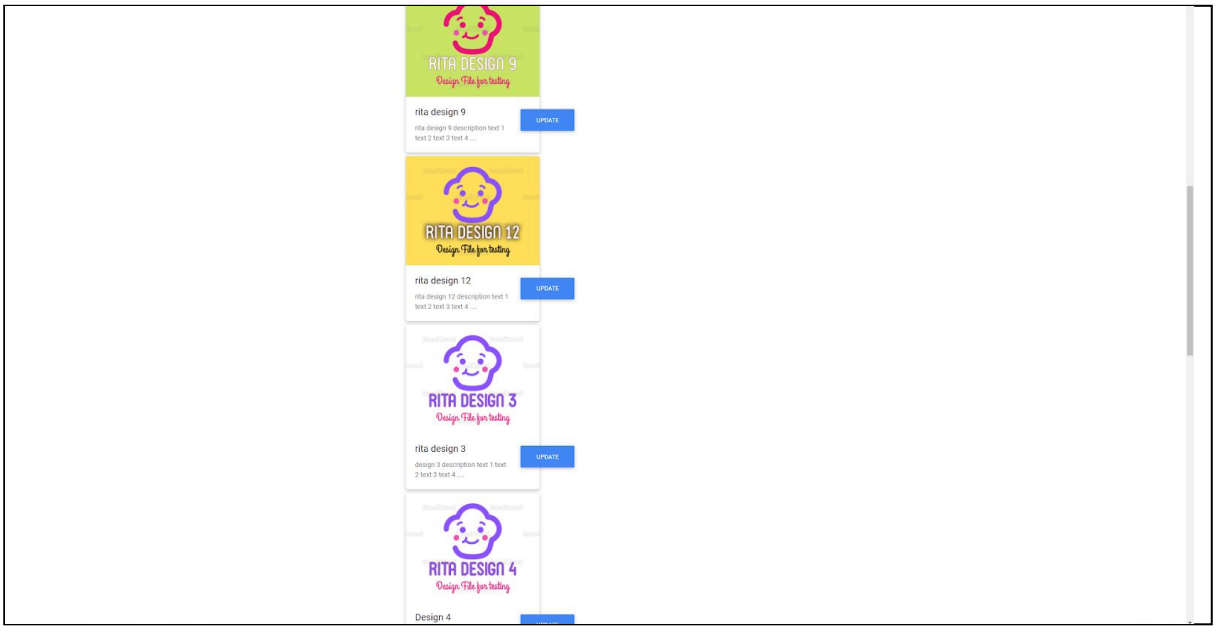
```
$('#getAllDesignButton').on('click', function(event) {
    event.preventDefault();
    axios({
        headers: {

        },
        method: 'get',
        url: 'https://twnfsrny37.execute-api.us-east-1.amazonaws.com/ESDE_CA2?search=&userId=100',
    })
    .then(function(response) {
        const records = response.data.data.filedata;
        const totalNumOfRecords = response.data.data.total_number_of_records;
        //Find the main container which displays page number buttons
        let $pageButtonContainer = $('#pagingButtonBlock');
        //Find the main container which has the id, dataBlock
        let $dataBlockContainer = $('#dataBlock');
        $dataBlockContainer.empty();
        $pageButtonContainer.empty();
        if (records.length == 0) {
            new Noty({
                type: 'information',
                layout: 'topCenter',
                timeout: '5000',
                theme: 'sunset',
                text: 'No submission records found.',
            }).show();
        }
        for (let index = 0; index < records.length; index++) {
            let record = records[index];
            console.log(record.cloudinary_url);
            let $card = $('<div></div>').addClass('card').attr('style', 'width: 18rem;');
            $card.append($('<img></img>').addClass('card-img-top').addClass('app_thumbnail').attr('src', record.cloudinary_url));
            let $cardBody = $('<div></div>').addClass('card-body');
            let $editDesignButtonBlock = $('<div></div>').addClass('col-md-2 float-right');
            $editDesignButtonBlock.append($('<a>Update</a>').addClass('btn btn-primary').attr('href', 'update_design.html?id=' + record.file_id));
            $cardBody.append($editDesignButtonBlock);
            $cardBody.append($('<h5></h5>').addClass('card-title').text(record.design_title));
            $cardBody.append($('<p></p>').addClass('card-text').text(record.design_description));
            $card.append($cardBody);
            //After preparing all the necessary HTML elements to describe the file data,
            //I used the code below to insert the main parent element into the div element, dataBlock.
            $dataBlockContainer.append($card);
            $dataBlockContainer.append($('<h5></h5>'));
        } //End of for loop
        let totalPages = Math.ceil(totalNumOfRecords / 4);

        for (let count = 1; count <= totalPages; count++) {

            let $button = $(`<button class="btn btn-primary btn-sm" />`);
            $button.text(count);
            $button.click(clickHandlerForPageButton);

            $pageButtonContainer.append($button);
        } //End of for loop to add page buttons

    }).catch(function(response) {
        //Handle error
        console.dir(response);
        new Noty({
            type: 'error',
            layout: 'topCenter',
            timeout: '5000',
            theme: 'sunset',
            text: 'Unable to search',
        }).show();
    });
```

Step 2: The manage submissions page is updated with the new button as seen below.



Step 3: We clicked on the button bordered in red and were able to retrieve all the submissions data that spans across 2 pages as seen below. This means that our Lambda Function has executed correctly.

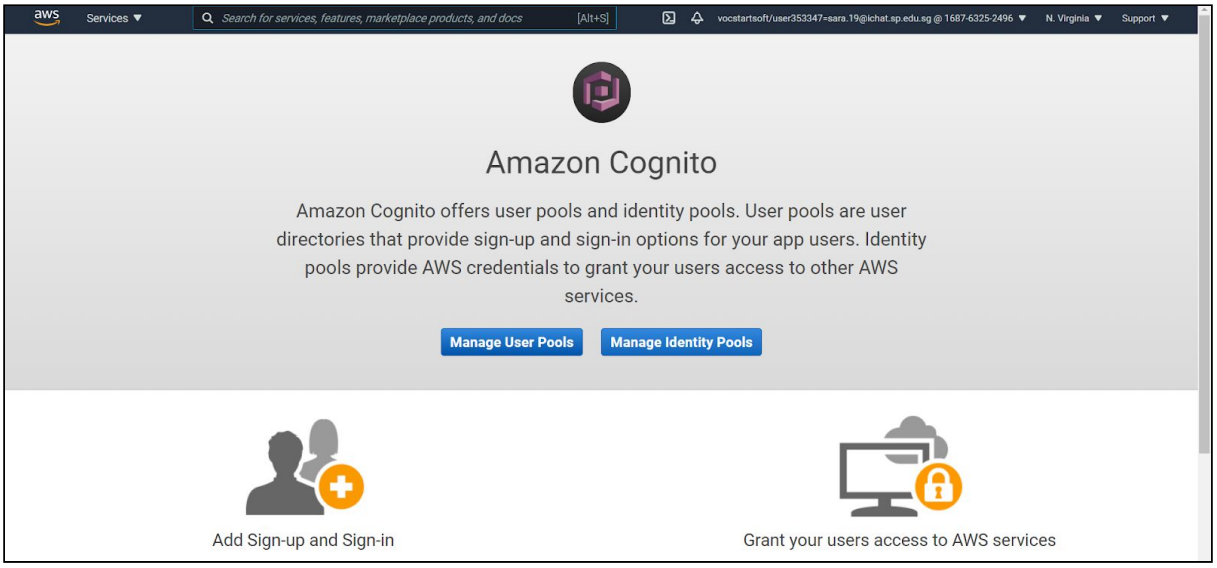




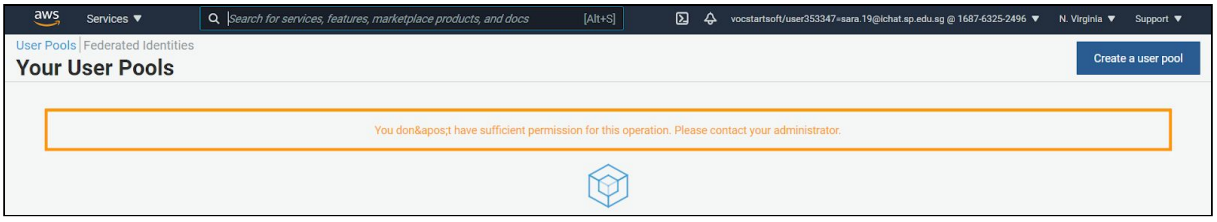
7. AWS Cognito service for Authentication (Advanced)

In this section, we will be showing a step-by-step approach on how to create an Amazon Cognito Pool to manage the creation of first-time user accounts at the registration page.

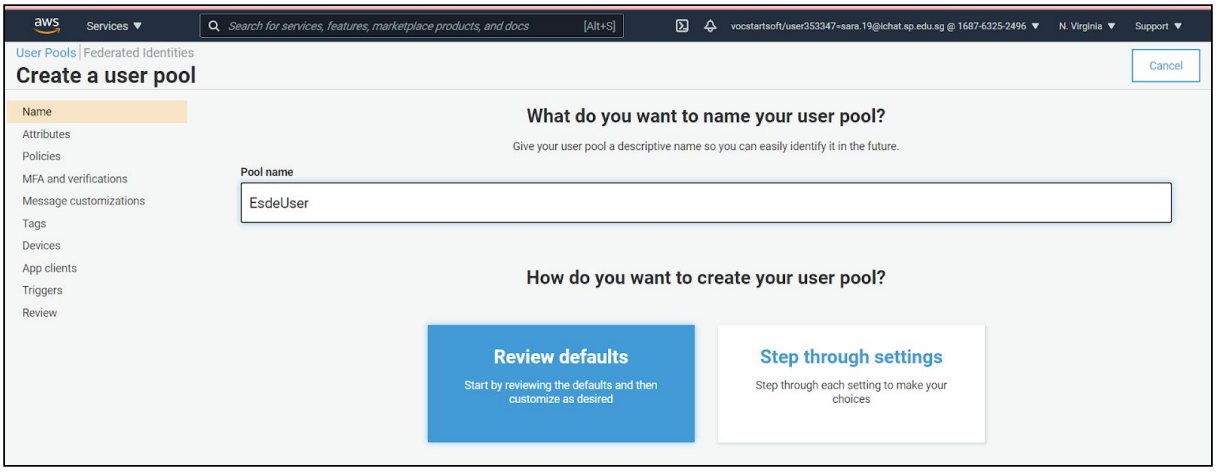
Step 1: After logging into our AWS **API Gateway** account, we navigated to the Amazon Cognito dashboard and clicked on the “Manage your User Pools” button.



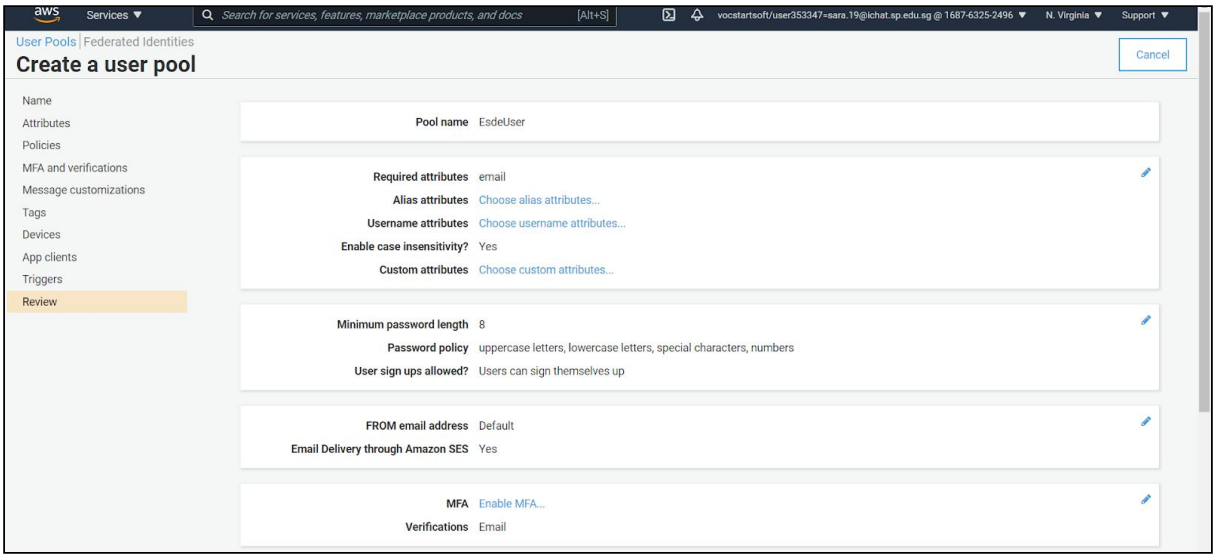
Step 2: We were redirected to the User Pools dashboard where we encountered this warning that we didn’t have sufficient administrative privileges. We ignored it and proceeded to click on the “Create a user pool” button.

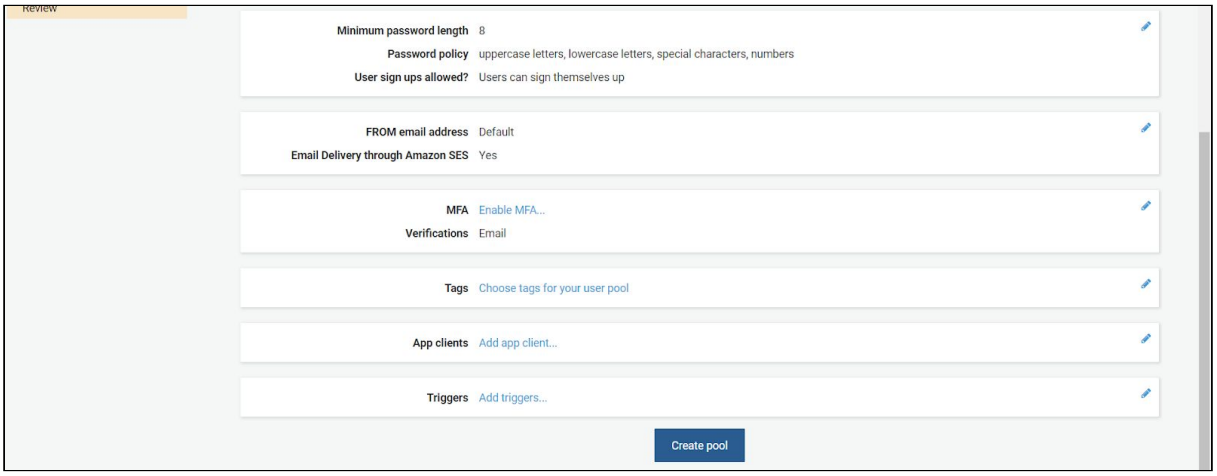


Step 3: For the creation of our user pool, we named it “EsdeUser” before choosing the option to “review defaults” as we wish to apply the default settings for the majority.

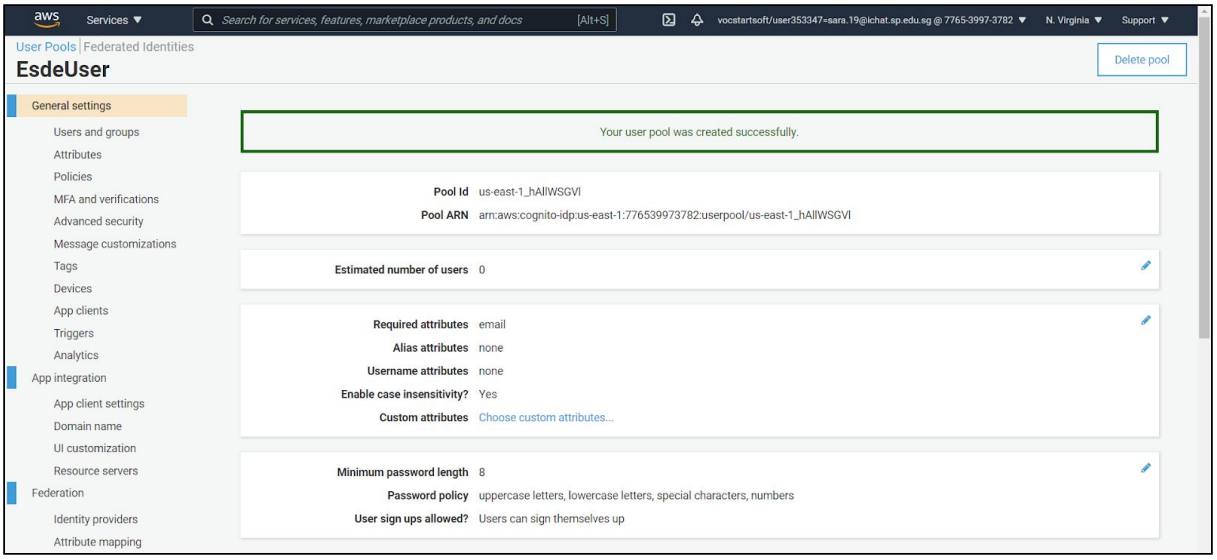


Step 4: We were redirected to the review page where we checked through each of the details before clicking on the “create pool” button.

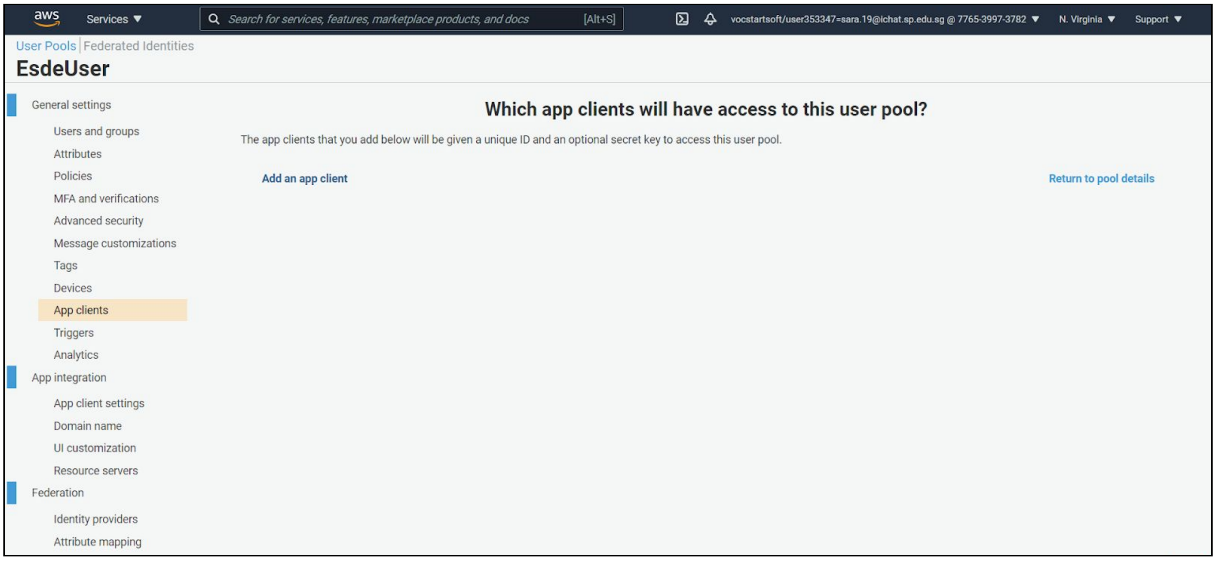




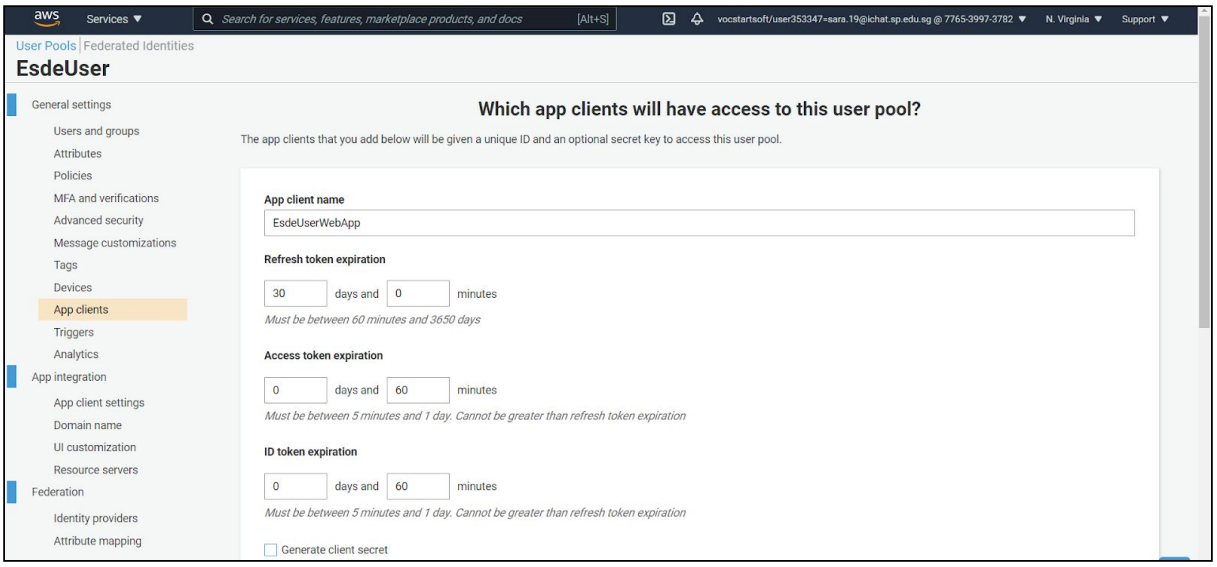
Step 5: We were informed that our user pool was successfully created as seen below. We also took note of the Pool Id that we need to use to update our config.js file later on.



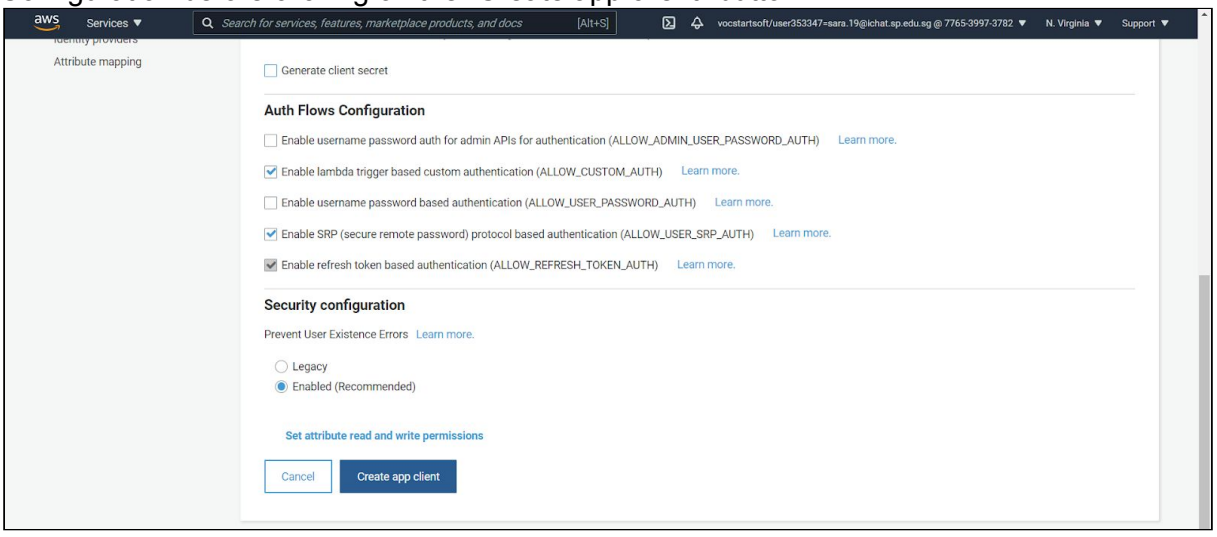
Step 6: From the Pool Details page of our user pool, we selected “App Clients” from the General Settings section in the left navigation panel. We proceeded to click on the “Add an app client” to connect our web application to the previously created user pool.



Step 7: We named our App client “EsdeUserWebApp” and applied the default settings for the Refresh, Access and ID token expiration. We also unchecked the checkbox for “Generate client secret” after learning that client secrets are not supported in terms of usage by browser-based applications.



Step 8: We applied the default settings for Auth Flows Configuration and Security Configuration before clicking on the “Create app client” button.



Step 9: We successfully created the app client as seen below. We also took note of the App client Id that we need to use to update our config.js file later on.