AWS Document Retrieval

Daniel Kim, Fadi Durah, Xavier Pleimling, Brandon Le,

and Matthew Hwang

CS4624, Multimedia, Hypertext, and Information Access

Virginia Polytechnic Institute and State University

Blacksburg, VA, 24061

Instructor: Dr. Edward A. Fox

Client: Rahul Agarwal

May 6, 2020

Table of Contents

Table of Figures

Table of Tables

1. Executive Summary

In the course CS5604 Information Retrieval, the class built a functioning search engine/information retrieval system on the Computer Science Container Cluster. The objective of the original project was to create a system that allows users to request Electronic Theses and Dissertations (ETDs) and Tobacco Settlement Documents using various fields, through their queries. The objective of our project is to migrate this system onto Amazon Web Services (AWS) so that the system can be stood up independently from Virginia Tech's infrastructure. AWS was chosen due to its robust nature.

The system itself needs to be able to store the documents in an accessible way. This was accomplished by setting up a pipeline that will stream data directly to the search engine using AWS S3 buckets. Each of the two document types were placed into their own S3 bucket. We set up an RDS instance for login verification. This database is used to store user information as they sign-up with the front-end application and will be referenced when the application is validating a user's login attempt. This instance is publicly accessible and can connect to developer environments outside of the AWS group with the right endpoint and admin credentials. We worked with our client to set up an ElasticSearch instance to ingest the documents along with communicating and manage the health of the instance. This instance is accessible to all of us with permissions and we are able to manually ingest data using cURL commands in the command line. Once the login verification database and ElasticSearch search engine were properly implemented, we had to connect both components to the front-end application where users could create accounts and search for desired documents. After both were connected and all features were working properly, we used Docker to create a container for the front-end application. To migrate the front-end to AWS, we used the Elastic Container Registry (ECR) to push our front-end container image to AWS and store it in a registry. Then we used an ECS cluster running AWS Fargate, a serverless-compute engine for containers, to deploy the front-end to the network for all users to access. Additionally, we implemented data streaming using AWS Lambda so that new entries can be automatically ingested into our ElasticSearch instance. We note that the system is not in a fully demonstrable state due to conflicts with the expected data fields. However, the infrastructure around the various components is established and would just need proper data to read.

Overall, our team was able to learn many aspects of standing up and building the infrastructure of the project on AWS, along with learning to utilize many different Amazon services. The new system serves as a functioning proof of concept that would allow a feasible alternative other than relying on Virginia Tech's system.

2. Requirements

Regarding our main deliverable of hosting the system on AWS, the requirements are:

- Every component must be hosted within AWS, independent of the Virginia Tech CS Container Cloud.
- The service must be operational within AWS and be able to communicate across programs within the infrastructure.
- Users must be easily able to import and ingest data into the system.
- The system must be robust with minimal downtime.
- The system must be well documented for future use and development.

3. Design

As stated in the executive summary, our role is to take the existing document retrieval system developed by the CS5604 class[1][3][6][7] and stand it up in AWS. This involves developing a project design where the functionality of the original system is recreated with Amazon services. Figure 3.1 shows the architecture of the existing system on the CS Container Cloud[1]. The existing system features a pipeline to ingest data, common storage to contain the various types of dataset, and a system to process and display information to the user across Kibana and the Front-end Implementation. The data, including both ETDs and Tobacco Settlement Documents, is provided by University Libraries[2] and the UCSF Production Container Cluster[8], respectively.

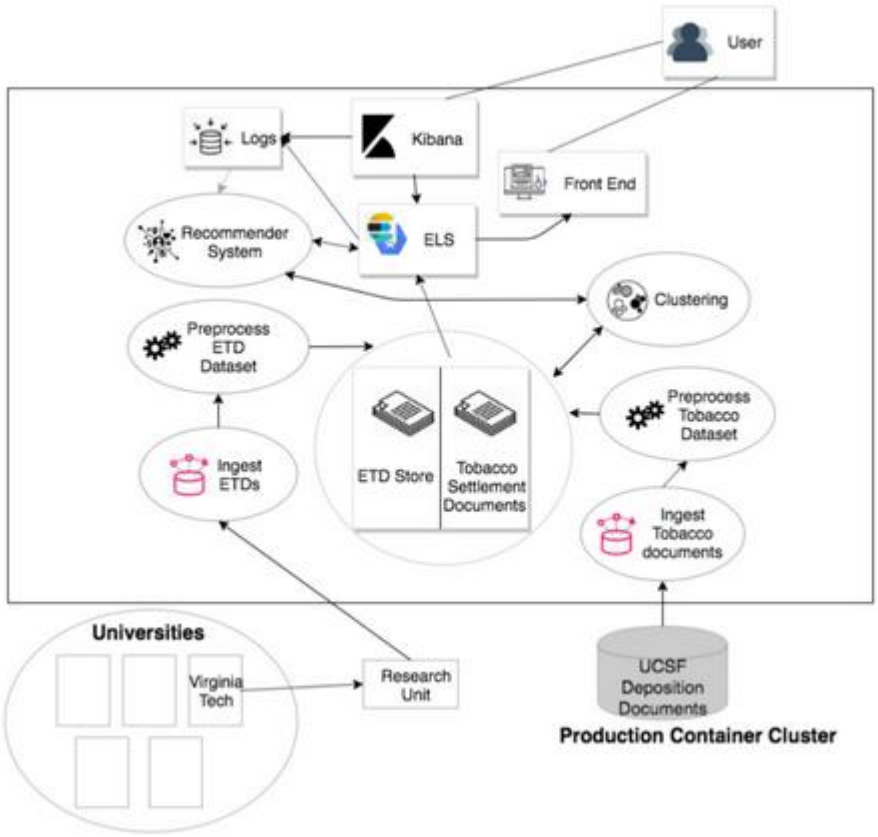

Fig. 3.1 - Outline of document retrieval system as hosted on CS Container Cloud.

Our design process involved a brainstorming phase where we outlined which AWS services we could use to replicate this functionality, which we then decided to visualize through a similar project outline diagram. The team will be able to import Tobacco and ETD documents into our system's storage made up of S3 buckets. From there, the documents will reside in a common storage unit where ElasticSearch will be able to ingest the data. This information will be relayed to the rest of the information retrieval system. Kibana, the UI interface for

researchers, will allow them to manipulate the searches as needed. The front-end interface for regular users will display the ingested information while interacting with the search engine. We also included a combination of AWS SNS and AWS Lambda to implement automatic data streaming as a stretch goal, where newly added documents would automatically be made accessible to the ElasticSearch instance. For the most part, the original project design from the CS 5604 Information Retrieval class was not changed: the goal was to migrate the system to AWS. Figure 3.2 represents our initial understanding of the project and our attempts to conceptualize each component within AWS.

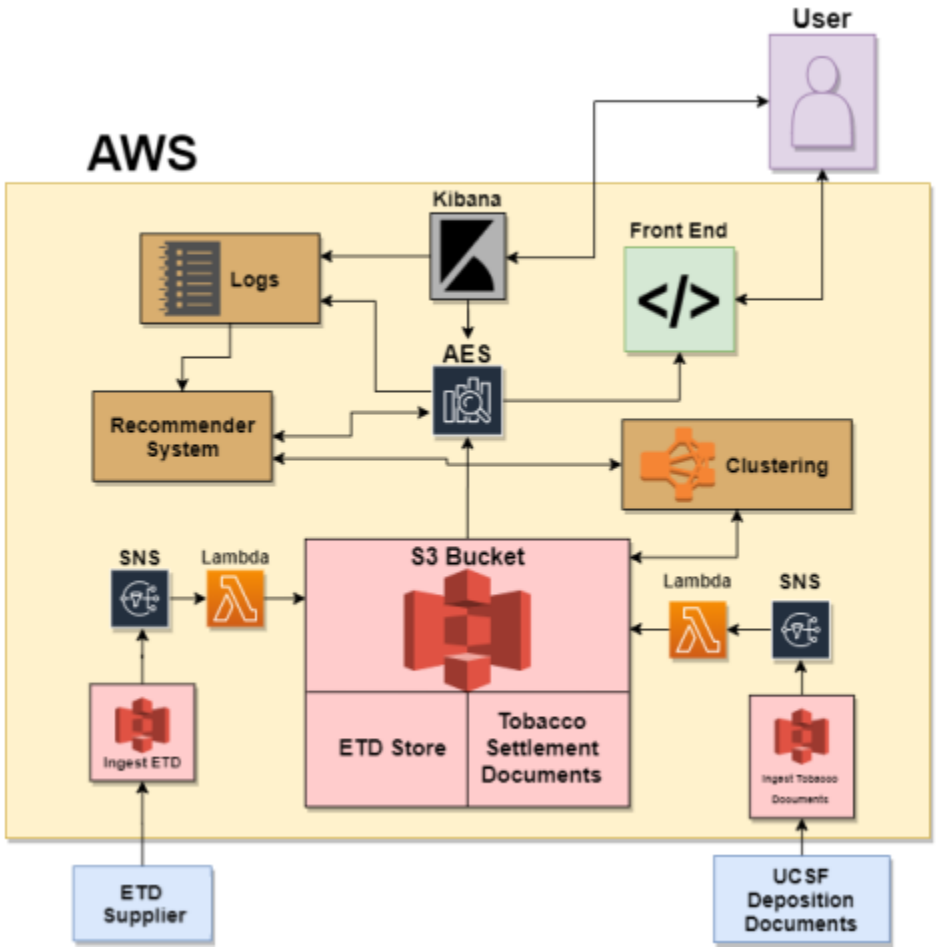

Fig. 3.2 - Outline of retrieval system as AWS-hosted service.

As stated, this diagram is very similar to the original project's outline. The functionality was preserved while the system was seemingly migrated to AWS. However, as we came to a better understanding of the initial system, we realized that some services were missing from our design while some aspects of our diagram were unnecessary. The original project included a containerized database for login verification purposes since the front-end application required users to create an account and login before they can request a document. Also, our decision to create 3 separate S3 buckets (2 intermediate buckets to store the documents as they came in and 1 large, central bucket to hold all the processed documents) was excessive. The use of SNS also

did not make much practical sense considering how data streaming would work, since no relevant messaging communication occurred between the services.

We expanded on the original design as per client feedback and requirements (Figure 3.3). The storage units for the documents were separated into their two main categories, Electronic Theses and Dissertations and Tobacco Settlement Documents. These buckets would eventually be streamed into the ElasticSearch service automatically upon importing new data (processed through AWS Lambda, SNS removed). Also note the inclusion of the RDS login verification database as it interacts with the front-end application. The design is simpler overall and fully encompasses the requirements for this project.

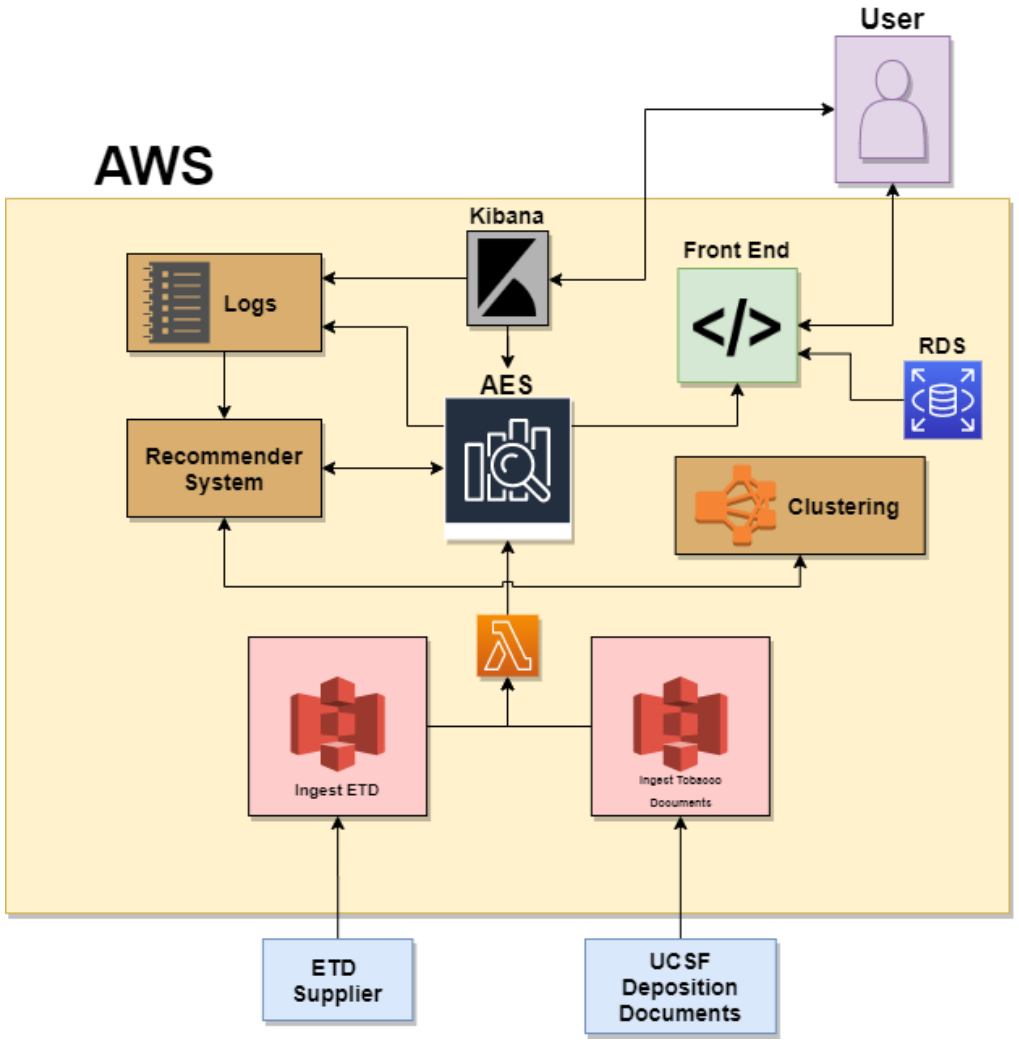

Fig. 3.3 - Updated outline of AWS-hosted document retrieval system.

4. Implementation

This section summarizes our current implementation progress and what has been completed so far. The project tasks are assigned here to each team member and are further explained in the Implementation section, where we discuss how our team proceeded with the project over the course of the term. Table 1 is a breakdown of team roles broken up across the team members.

Group Members	Project Roles	Technical Roles
Daniel Kim	• Team Leader	• AWS S3 Buckets • Data Streaming
Xavier Pleimling	• Presentation Lead	• ElasticSearch Ingestion
Fadi Durah	• Reporting • Team Leader (Back-Up)	• AWS RDS Database • Containers (Back-Up)
Brandon Le	• Notetaker • Reporting (Back-Up)	• ElasticSearch Ingestion
Matthew Hwang	• Research	• Front-End Application • Containers

Table 1 - Breakdown of roles for each team member.

4.1. Data Migration

In order to stand up an independent system, the data was migrated from the Virginia Tech Computer Science Cloud servers to Amazon S3 Buckets. These act as storage units that are hosted by AWS, able to be accessed by the other programs and services within AWS. The ETDs and tobacco documents were all migrated to their own S3 Buckets.

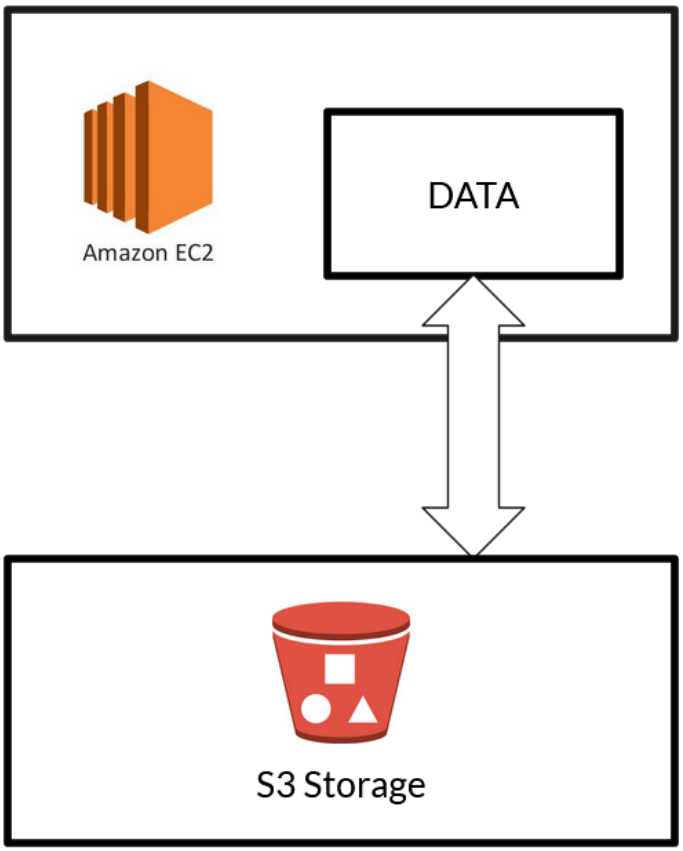

Fig. 4.1.1 - Data pipeline using EC2 and S3.

To act as an external file system, servers were created using AWS EC2 instances. These servers have the S3 Buckets mounted on them, giving them access to the files within (Figure 4.1.1). This would act as a pipeline between the external data source and the common storage of AWS S3. Users would be able to access these servers with a private key, and able to use a terminal to access the data without the need for direct access into the AWS account.

4.2. ElasticSearch

We want to ingest all the data and documents that are currently on the Virginia Tech Computer Science Container Cluster to Amazon Web Services. AWS provides ElasticSearch services to hold such information. We originally had our data on the CS cluster in JSON files, so we set up an ElasticSearch domain and attempted to connect to it to transfer the files (Figure 4.2.1). This ElasticSearch instance is up and running, but we have encountered several problems attempting to connect to it and, as a result, we had trouble progressing beyond the point of creating the domain[4].

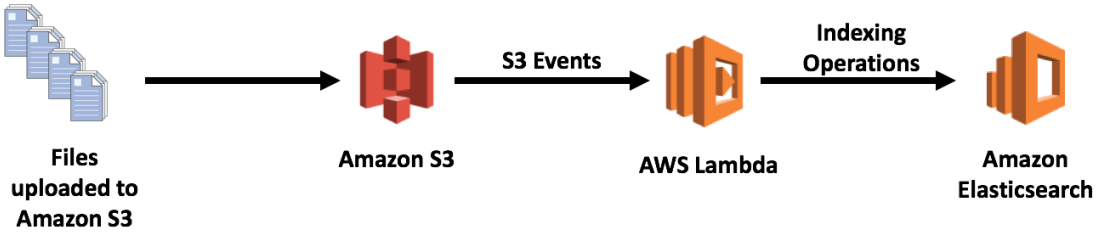

Fig. 4.2.1 - Data chain feeding ingested entries into AES.

After experimenting more with our original domain, we decided to switch to a different AWS ElasticSearch domain that our client had set up and we were finally able to manually ingest sample data into that instance so that our front-end can use that data for testing. While we figured out how to manually ingest the data into ElasticSearch, this task would be tedious to do for every single file that is currently stored in the CS Container Cluster, so we looked into ways to automate this process such as data streaming.

4.3. Data Streaming into ElasticSearch

With the use of AWS Lambda serverless scripts, an event will trigger whenever a new file is placed into an S3 bucket. The data will be automatically processed and streamed into an appropriate index within the ElasticSearch domain (Figure 4.3.1). This process was developed to streamline the data into ElasticSearch rather than manually ingesting each new piece of data as it is presented.

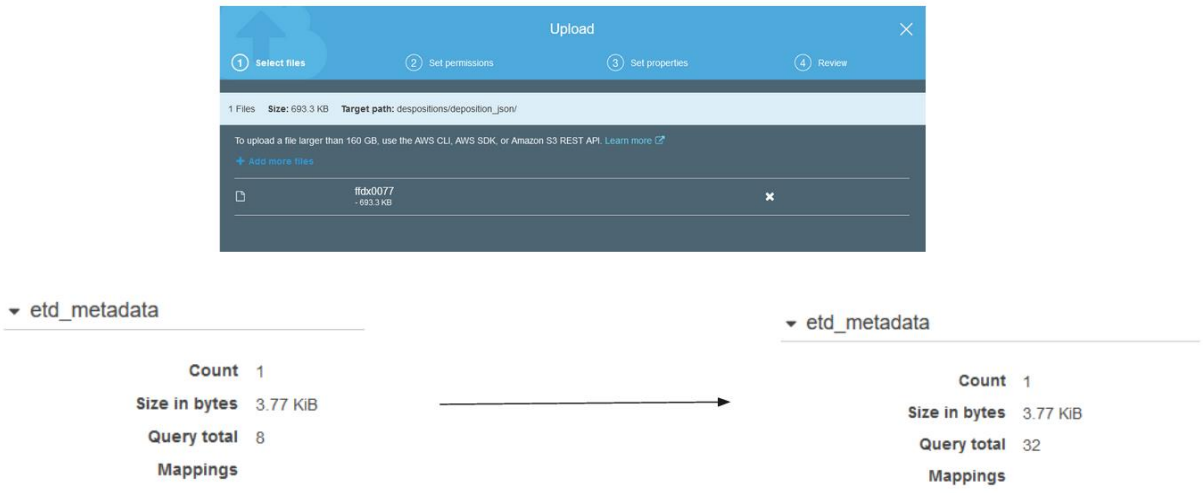

Fig. 4.3.1 - Example flow of streaming data using AWS Lambda.

The data streaming services are functional for both types of buckets and indices, able to ingest new data as it arrives. Note that the traditionally large Tobacco file sizes, ranging from roughly 300,000 to 500,000 kilobytes each, tend to lead to errors when being ingested through AWS Lambda. AWS Lambda typically fails with large files that generate long runtimes, automatically shutting down the process after a set amount of time. Furthermore, adding large files directly to AWS ElasticSearch through cURL commands will also take a significant amount of time, potentially taking hours at the very least provided the command even succeeds. For those reasons, we would recommend splitting up long tobacco files into much smaller files of roughly 100 lines each, ultimately ranging from mere tens or hundreds of kilobytes to no more than roughly 12,000 kilobytes, before attempting to stream the tobacco data.

4.4. RDS Login Database

This section of the implementation explains our current progress in setting up the login verification database. The end goal is to have an accessible database hosted on RDS so that the front-end application can communicate with it to verify a user login attempt.

4.4.1. Login Verification

For a user to use the front-end Flask application, they must be registered within the service's login verification database and access the system securely by providing their credentials. The client's initial implementation utilizes a containerized database hosted on the VT CS Container Cloud to store user login information. Entries in this database consist of a unique username and password, along with an email and registration time. These fields are stored in a "users" table and are shown in the FEK group's[7] table schema (Figure 4.4.1). This same schema will be used to create the final RDS instance so that login information is stored in the same format to comply with the front-end application's code.

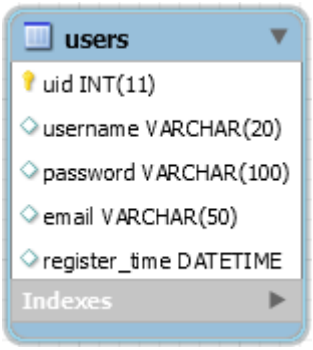

Fig. 4.4.1 - Schema for "users" table, FEK Group.

When a user completes the sign-up form, the information they enter is used to populate a new entry in the database, which the front-end application later references to validate a login attempt. To migrate this verification process, we needed to create an RDS instance to hold the

login information in a similar manner while being accessible to the front-end container. While much login data has been amassed during the previous system's lifetime, we decided to not worry about migrating it to the new database. Our client advised against migrating the old data since the previous implementation's database was not operational at the time, thus the original login information was inaccessible. Instead, we would rely on the front-end's sign-up form (mentioned before) to repopulate the database with user login information.

Ultimately, it is imperative that this verification process be migrated to AWS to maintain system security since all other aspects of the project are similarly being stood up on AWS. Below, we outline the steps we took to fully migrate the process as well as make it developer-friendly so that future project members can contribute to the design.

4.4.2. Local Database

The first step in migrating the login verification database was familiarizing ourselves with setting up database connections in order to request information. To accomplish this, we researched and hosted a MySQL database on one of the team member's personal desktop. The reasoning behind this was that establishing a connection to fetch a username and password from a database entry is functionally the same as what will be done by the front-end application. With this in mind, our local database is shown in Figure 4.4.2, populated by fake, placeholder data.

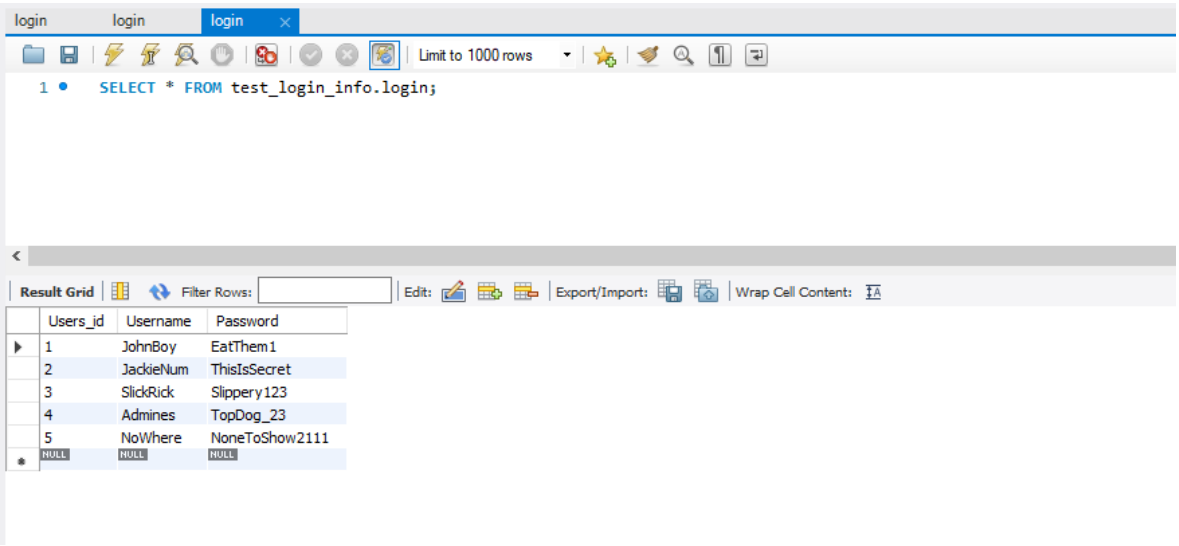

Fig. 4.4.2 - Locally hosted database for testing connections and requests.

After the database was set up, we created a short Python script to establish a connection to the database using the MySQL Python library. Once we were able to query the database, we ran a simple input prompt to ask the user for a username and password. Once entered, the script would loop through the database entries and return whether there was an entry with a matching username and password. This works as an effective representation of the front-end application's login verification. Ideally, we would have liked to containerize the script and reproduce the same

results to better simulate the front-end process, but we ran into some issues with the local database connection that will be discussed in the Lessons Learned section.

4.4.3. RDS Instance

Once we had created a suitable database prototype on a local machine, we wanted to move that functionality to AWS. There were two ways to do this: create a containerized database and run on Amazon ECS or deploy a database instance. The previous team opted for the containerized database, but this design choice was influenced by lack of an integrated database hosting service available to them within their network. In our case, we are already using many AWS services, such as S3 and AES, so it would make more sense to implement the login database as an RDS instance. This not only makes it easier to integrate with other AWS services, but also reduces the additional container layer that would otherwise house our database. This means that establishing a connection should be even simpler than with the containerized approach. Members of the team also had prior experience working with RDS, so the switch made a lot of practical sense.

As mentioned in the first login verification section, the RDS instance borrows its table schema from the FEK group's implementation. This simplifies things quite a bit, since it allows the front-end to push new data to the database as users fill out the sign-up page and verify login credentials without any issues related to column names or table structure. As the RDS instance was created, we are able to configure various settings related to the database, such as storage size and bandwidth, giving us more control over the instance based on workload and how the team monitors demand. We discuss connection configuration and accessibility more in the next section.

4.4.4. RDS Connection

RDS provides many ways to configure an instance's connection capability. In our case, we wanted the database to be able to connect to our front-end container for the login verification task, so that the application can reference and update login entries. Aside from that, we decided that the database also needed to be accessible through a MySQL platform for schema manipulation and monitoring. The front-end container was relatively simple to connect, since all that was required was passing the RDS endpoint and admin user credentials to our front-end developer. As far as setting up a MySQL platform for development, it's possible to set up an EC2 instance within our AWS environment and have MySQL tools downloaded on it for monitoring and updating the RDS instance. However, this would require the developer to establish a connection to an EC2 instance along with connecting to the database itself, while also limiting them to a command prompt. This would suffice for our project since we only needed to set up the schema, but future developers may want to use visualization tools to make their database interactions more effective. Thus, we decided to make our RDS instance publicly accessible, meaning anyone could attempt to access it, but the proper endpoint and login

information would be required to establish a connection. At the time of this report, the RDS instance only has an admin user set-up. This design choice allows team members to connect from local machines and use their preferred development environment to manipulate the database rather than rely on EC2.

In our case, we used the public accessibility to establish a connection to a MySQL Workbench running on a local machine. This allowed us to create a schema model to build our database's users table, along with establishing necessary constraints for the table's fields (usernames are unique, for example). Using Workbench, we monitored the ingestion of login information as test sign-up attempts were conducted, allowing us to confirm that new data was being added properly once the front-end was connected to the database (discussed in the next section). Table 2 shows a sample of login information from our test runs. Workbench also allowed us to debug more effectively thanks to its visualization capabilities, which helped when we would run into issues when implementing the login verification.

uid	username	password	email	register_time
1	1234	5rounds=535000$zscTNVyJl2T7rfPZ$lVJ44Bf...	123456	2020-04-15 21:06:41
2	username	5rounds=535000$9J9pnAgUuTKxDTFs$7EGx...	email@email	2020-04-15 21:15:31
3	username1	5rounds=535000$KygQ9lvLGjH3q7BJ$kiWwB...	email1@email	2020-04-16 00:33:59
4	username2	5rounds=535000$XyBiNDGfl88DsaZP$WZSp...	asdfasdf	2020-04-16 00:46:57
5	asdfasdf	5rounds=535000$q9aZGwQtIrhJy/mO$NFp/...	asdfasdfasdf	2020-04-16 00:50:54
6	asdfasdfasdf	5rounds=535000$2Cid5JuGsXnezKHg$40iBF...	asdfasdfasdfasdf	2020-04-16 00:55:44
7	asdfasdfasdfasdf	5rounds=535000$7IEEBi5NhGtRO6nK$7GwT...	asdfasdfasdfasdfasdf	2020-04-16 01:05:45
8	asdfasdfasdfasfasdf	5rounds=535000$A5UPrXpPnPIz4fsH$4jmdF...	asdfasdfasdfasdfasdfasdf	2020-04-16 01:10:25
NULL	NULL	NULL	NULL	NULL

Table 2 - Dummy login information in database.

4.5. Front-End Container

In order for users to interact with and explore the ETD and Tobacco Settlement results generated by the ElasticSearch search engine, the original CS5604 project team developed a front-end application (Figure 4.5.1). This application was developed using React.js to create the interface and Flask to create a bridge to the ElasticSearch engine. In the original implementation, the application was packaged into a Docker container which was deployed using Kubernetes. Our goal is to migrate the deployment of the front-end container to AWS using the Elastic Container Service (ECS).

4.5.1. Local configuration

We have been able to gain access to and pull the front-end code from the GitLab repository from the original project. The dockerfile needed to be modified in order for the container to be built locally. Once the proper modifications were made, the container was able to be built and ran on our local machines to ensure that the container was functioning properly. Before we could begin to migrate the system onto AWS, we had to connect the front-end application to the new login database and our new ElasticSearch instance. These connections

were established by creating a settings.yaml that contains all the endpoint information to send and receive data to the database and ElasticSearch engine. We followed the instructions in the Readme file written by the original team for how to properly configure the settings.yaml file. An in-depth explanation of this connection process can be found in the developer manual. Additionally, the front-end component for the search UI had to be built separately in order for the search UI to run concurrently with the main app.

4.5.2. AWS ECR and ECS

Once we configured the front-end to work locally with our login database and ElasticSearch instances, we rebuilt the Docker container for the front-end. Then we figured out how to push the container image to the AWS elastic container registry (ECR) and deploy the container on ECS. We used the AWS Command Line Interface (CLI) on our local machines to perform the push commands on the container we built locally to upload it to ECR. We completed the deployment of the container by using an ECS cluster using AWS Fargate which allows us to run the application on a public IP address. An in-depth explanation of the AWS container deployment process can be found in the developer manual.

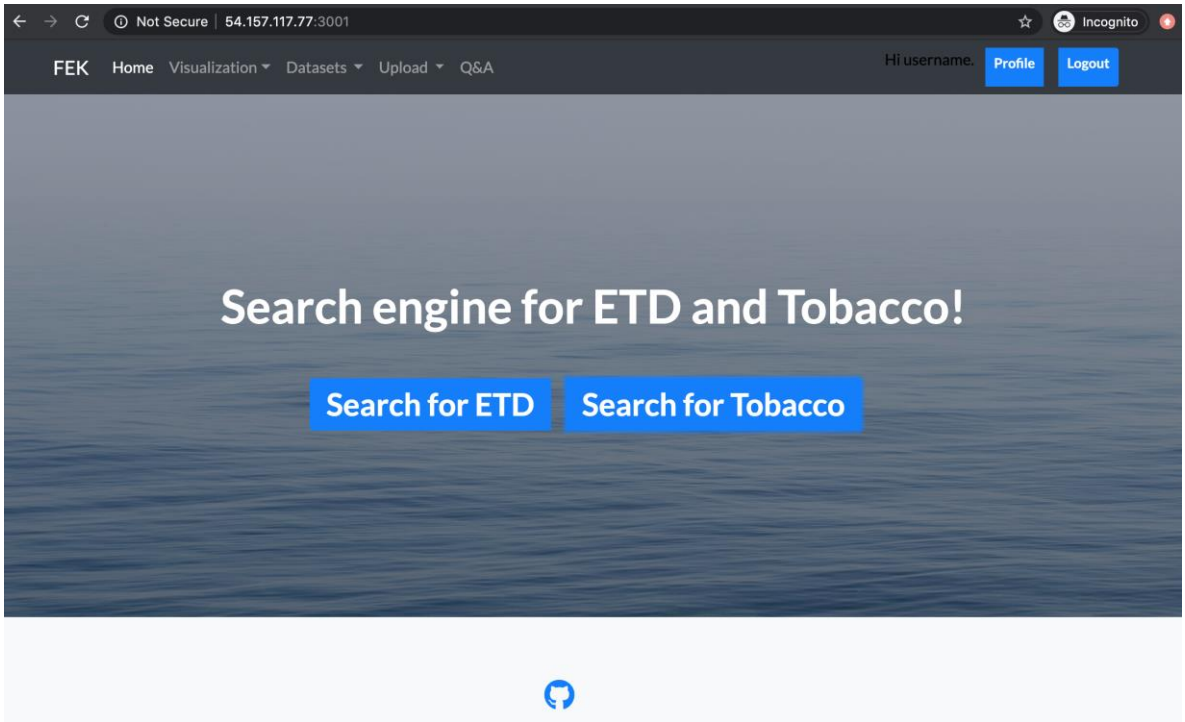

Fig. 4.5.1 - Front-end container running through AWS ECS.

4.5.3. ElasticSearch data conflict

We were able to get the front-end fully functioning on AWS, however, when the front-end is running, and the user decides to search for documents, the document information is incorrectly displayed as “Undefined” for fields such as title, author, etc. (Figure 4.5.2). This is not caused by a flaw in the front-end, rather it is due to incorrectly formatted data coming from ElasticSearch.

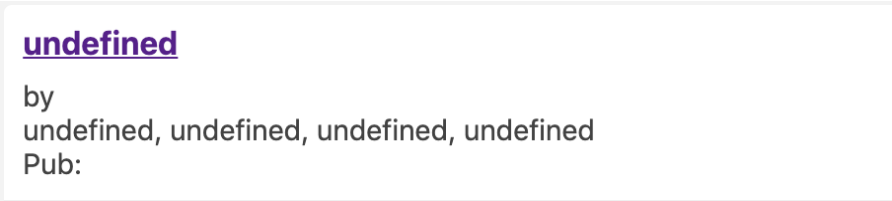

Fig. 4.5.2 - Undefined document information.

The front-end section used to display search results is built from Reactivesearch components in the React framework. In order to display information such as the document’s author or date published, the data needs to contain field names where the Reactivesearch components can pull the data from. A segment of properly formatted ElasticSearch data is shown in Figure 4.5.3.

Fig. 4.5.3 - Segment of properly formatted ElasticSearch data.

The front-end code can call this data through Reactivesearch components and display it in the front-end. This is shown in Figure 4.5.4 where it is calling the “title-none” field from the ElasticSearch data represented by “res”.

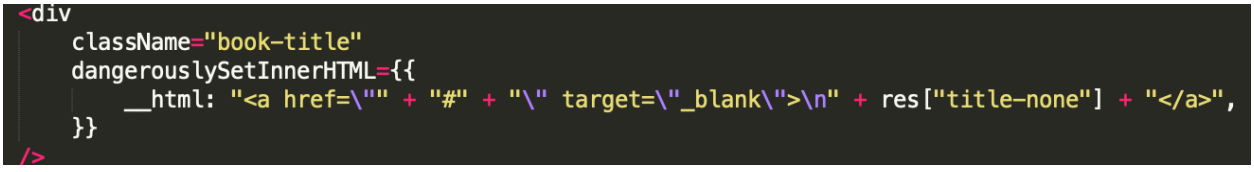

Fig. 4.5.4 - Reactivesearch calling field from ElasticSearch data.

The data that our instance of ElasticSearch returns is formatted incorrectly because it is missing the field that Reactivesearch uses to display information in the front-end. Figure 4.5.5 shows the data our ElasticSearch instance returns. The data only contains fields for ‘page’ and ‘content’ under text_content. Since it does not contain fields for title, author, and date published, the front-end displays “Undefined” in the UI.


```
"_index": "etd_metadata_docs",
"_type": "_doc",
"_id": "4uDY4XEBLwrlmWOym1YB",
"_score": 1.0,
"_source": {
    "text_content": [
        {
            "page": 1,
            "content": "GASTON OSTIGUY Int. Ex.      "
        },
        {
            "page": 2,
            "content": "L'an mil neuf cent quatre-vingt-di
                fevrier, a comparu comme temoin dans cette
```


Fig. 4.5.5 - ElasticSearch data from our implementation.

In order to resolve this issue, the data needs to be reformatted to include all the fields that the front-end needs to display. This is something that needs to be worked on by the ElasticSearch team in the future.

5. Future Work

This section is an outline of future plans for the project and work that could be further developed after our team has finished working on it for the Spring 2020 semester. This is not an exhaustive list of what could be improved but were areas that our team considered worthwhile improvements to the overall system.

5.1. Relocating RDS Instance

Currently, our RDS instance is running in the Ohio (us-east-2) region, while almost all other services are running in the North Virginia (us-east-1) region. This is a small error and is pretty much inconsequential for our implementation since our connection to the front-end application is not hindered by it. If we were to fix it, we would need to tear down our current RDS instance and recreate it in the correct region, which would cause significant downtime for our application. In the interest of time, our client said that recreating the database is not necessary for our project scope so that we can focus on implementing other functionality. Future developers likely will recreate the instance, however, and will need to recreate the connection to our front-end application.

5.2. Continuous Integration and Continuous Delivery (CI/CD)

CI/CD was a stretch goal presented to us by our client to set up a pipeline that would allow for continuous development of the codebase. AWS CodePipeline along with other services like AWS CodeDeploy can be used to establish a CI/CD pipeline within AWS. Amazon Elastic Container Registry (Amazon ECR) could be used as a source within AWS CodePipeline to implement a continuous deployment pipeline with the containers in the project.

If found worth pursuing, this would help to further automate the processes behind the architecture and would streamline deployment and future development.

5.3. Aligning Data Fields

Compared to the original project code, the metadata that the front end application expects is different compared to what has been provided. Depending on what is simpler or desired, either the code that parses the data or the data format itself should be changed to align with each other. That way the data that is fetched by the application will be readable rather than undefined.

6. User's Manual

This section outlines how a user should interact with our front-end application. Screenshots of the various pages (including, sign-up, login, and request pages) are included.

6.1. Signing up

Figure 6.1.1 shows the form for registering an account on the system depicted by the CS5604 Information Retrieval Final Report Front-end and Kibana (FEK). Users will enter their login details in the form and register an account on the system.

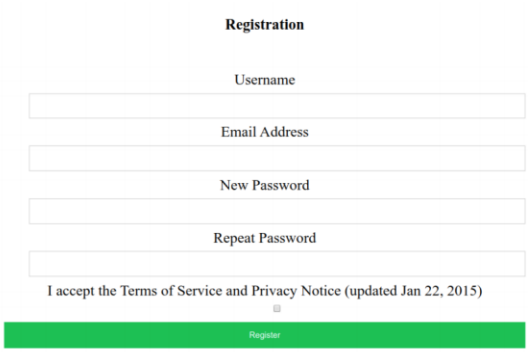

Fig. 6.1.1 - User registration form depicted in the CS5604 FEK report.

6.2. Logging in to the Service

Login is handled by a simple login form (Figure 6.2.1) where users enter the Username and Password they created for registration to gain access to the system.

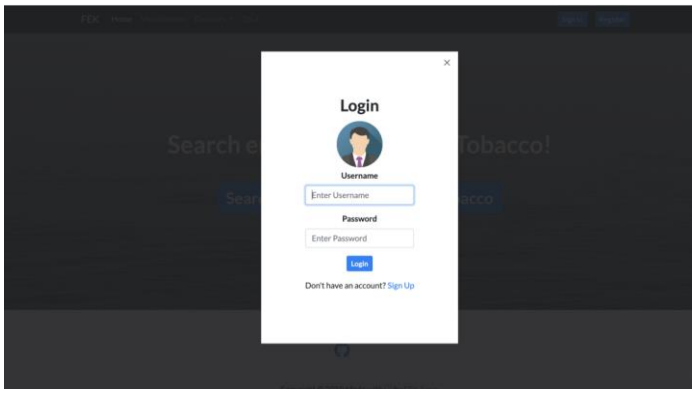

Fig. 6.2.1 - User Login form depicted in the CS5604 FEK report.

6.3. Requesting Document

Users have the ability to search between Electronic Theses and Dissertations (ETDs)[2] and Tobacco Settlement Documents[8]. Selecting one of the two options on the home screen will bring the user to one of the pages to type in what information they are looking for and refine their search by selecting various values. Examples are shown in Figure 6.3.1 and Figure 6.3.2.

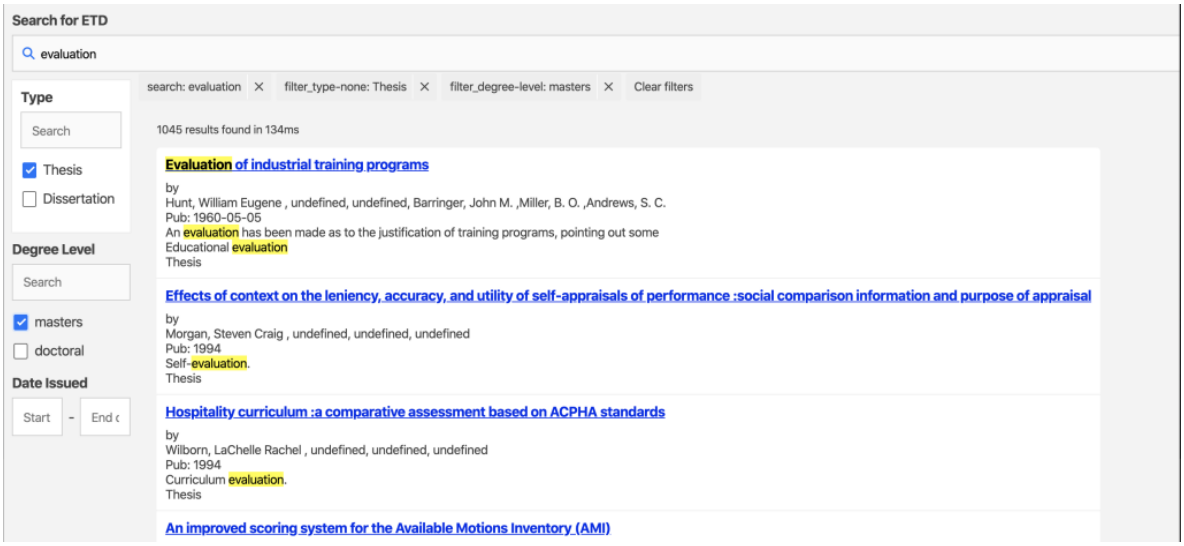

Fig. 6.3.1 - ETD search form depicted in the CS5604 FEK report.

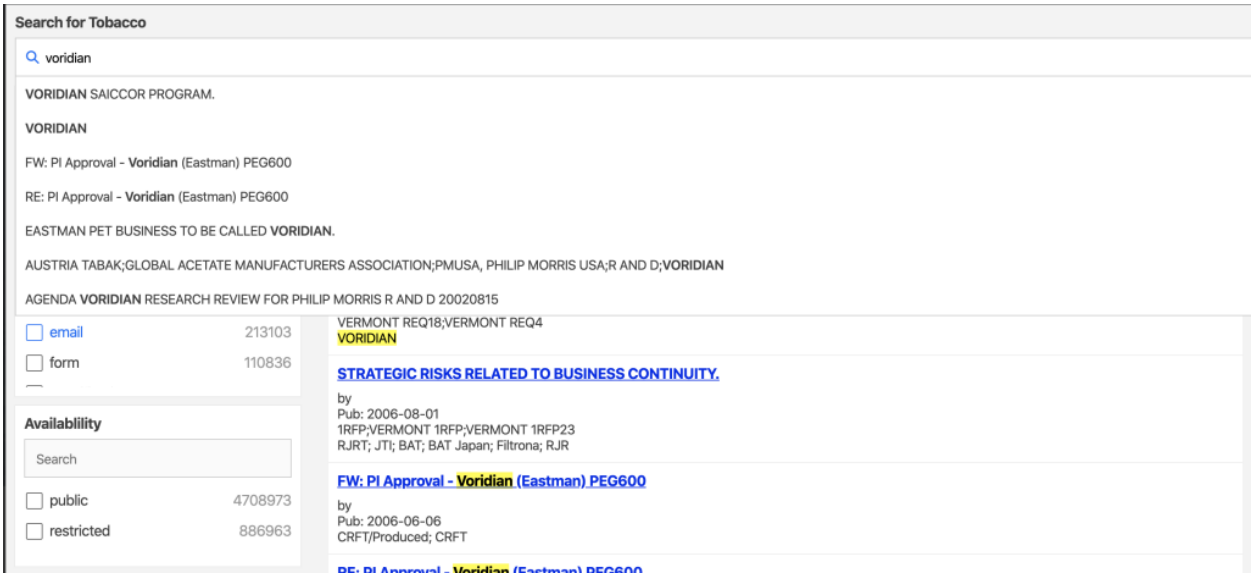

Fig. 6.3.2 - Tobacco Settlement search form depicted in the CS5604 FEK report.

7. Developer's Manual

This manual is a guide to interaction with the project from a development point-of-view. As more features are migrated and implemented on AWS, we will include instruction on maintaining the service.

To start, a proper understanding of our system's users and their goals is important so developers can implement towards satisfying those requirements. We identified 3 main user personas that our system needs to accommodate. The first user persona is the standard front-end user who will communicate with our Flask application to request documents from the system. The second is that of system developers (CS5604 students) that would need to study and understand our system along with being able to continue development and maintenance of the service. Finally, the third persona is that of the researchers, who need to be able to import desired data into the system while also being able to view and manage ingested documents. The diagram shown in Figure 7.1 outlines the user goals and maps them to our system solutions (features of our design that achieve each user goal). From there, the diagram breaks the solution down to its respective Amazon services. The rest of the manual will go into greater detail into each feature of our system.

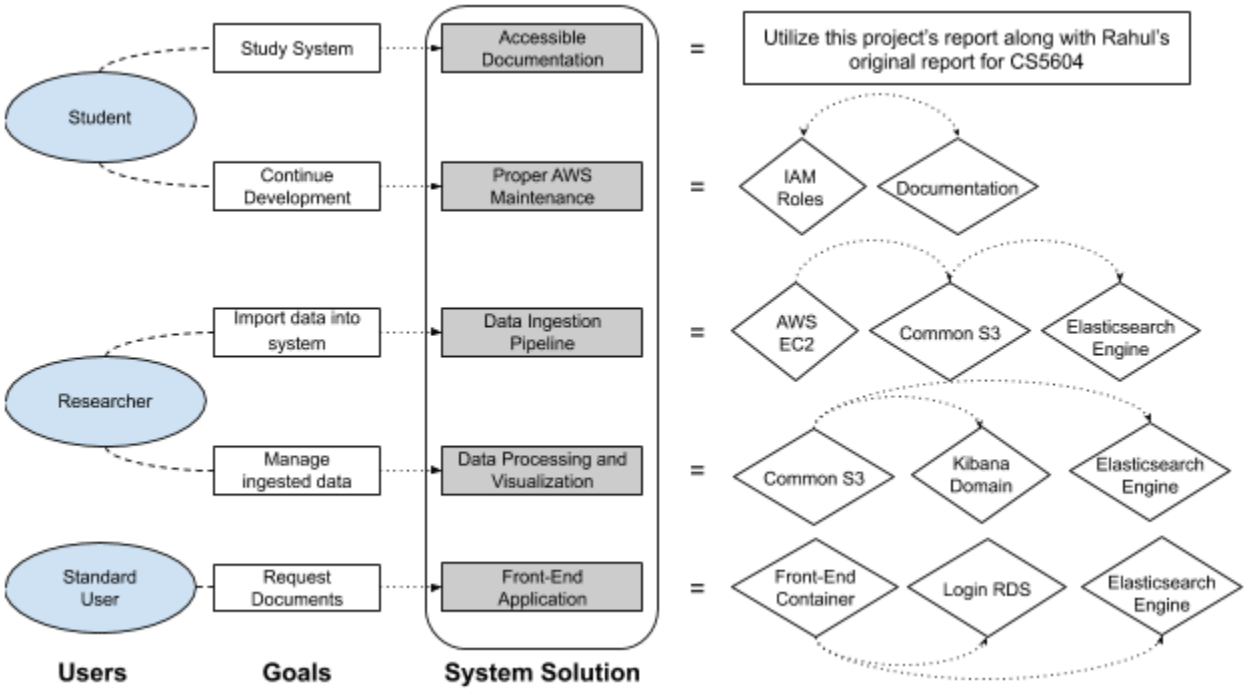

Fig. 7.1 - System Workflow diagram.

7.1. Importing and Ingesting Data

This part of the developer's manual pertains to the management of the services concerning data. This will outline how to work with AWS S3 Buckets, the external file systems, and components concerning data streaming.

7.1.1. Data Storage

We have migrated the existing document entries that are currently accessible in the CS Container Cloud implementation into AWS common storage (Figure 7.1.1). These documents are currently stored in S3 buckets (*depositions* and *tobacco* which respectively hold the ETDs and Tobacco Documents) accessible to team members with the proper IAM roles.

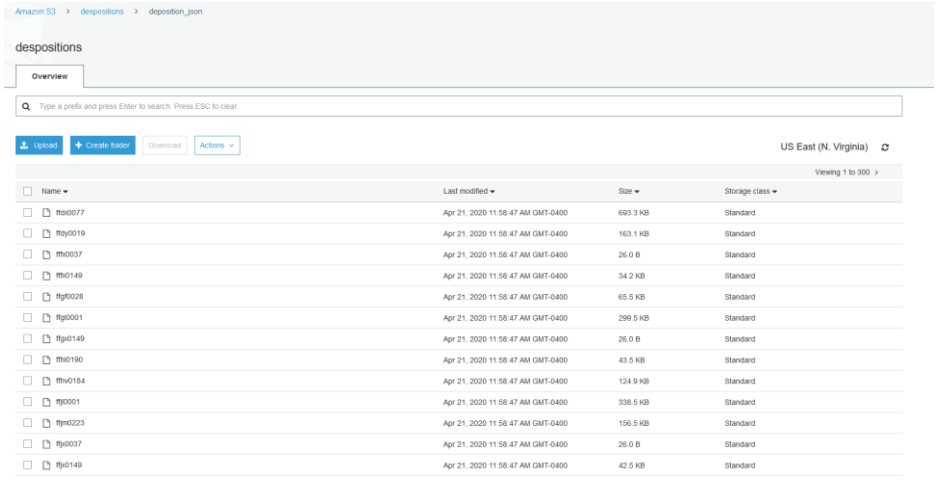

Fig. 7.1.1 - Screenshot of AWS S3 Console.

7.1.2. External File Servers

As for now there are two servers, each having one of the data buckets mounted on them. If you navigate to the "EC2" page in the AWS account, it will list the IP addresses of the servers after selecting the "Connect" option on the instance. AWS EC2 Instances will change their IPs whenever they start or stop so the IPs may change since the writing of this document. A private .pem key will also be needed to connect to the servers.

Once inside one of the file servers, the respective bucket can be found in the "/buckets" directory (Figure 7.1.2). The bucket can be listed, read to, and/or written in from this server.

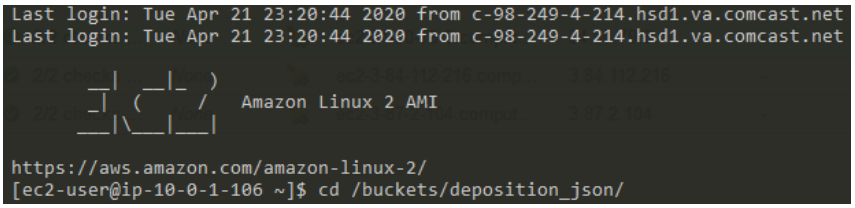

Fig. 7.1.2 - Example image of directory within AWS EC2.

7.1.3. Data Streaming

The AWS Lambda scripts can be found on the AWS account. The script will trigger whenever a file is placed into the ETD or tobacco buckets. The data will be sent toward the named ElasticSearch domain in the manner specified in the script. The two Lambda functions responsible for each bucket are named “stream-s3-tobacco” and “stream-s3-etd”.

7.1.4. System Monitoring

Using AWS Cloudwatch, one can easily view system metrics and function logs to check if the data is being streamed properly. Navigate to Cloudwatch -> Log Groups -> “desired function” (Figure 7.1.3) in order to view the events with time stamps. Error messages would also be displayed here.

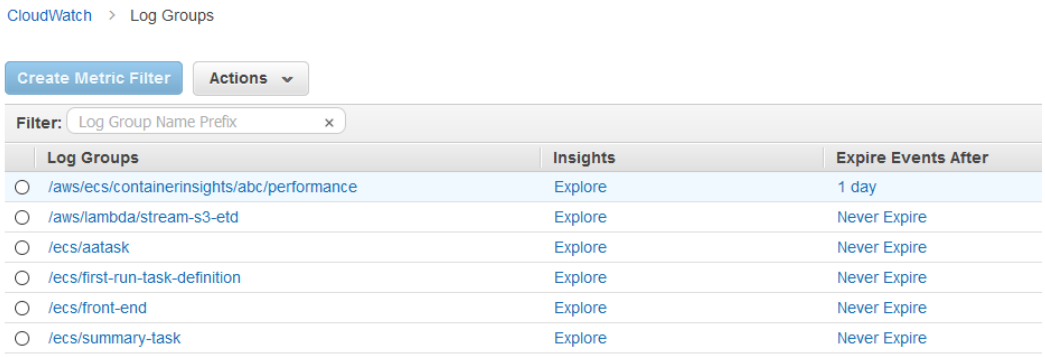

Fig. 7.1.3 - Example of AWS Cloudwatch Console with Log Groups.

7.2. Login Database Management

This part of the developer’s manual pertains to the configuration and maintenance of the RDS database instance for login verification. This will outline how to access the RDS console for our database instance, along with setting up a connection to a local MySQL Workbench.

7.2.1. RDS Console

Our RDS instance is named *retrieval-login-db* and is currently running on MySQL engine version 5.7.22. To access the instance console, go to the RDS service page and click on *Databases* on the left-hand side of the screen, then click on our instance. Upon selecting the instance, the console screen will appear with many options available to the developer (Figure 7.2.1). As a quick note, developers can use the monitoring tab to see connection attempts and recent activity regarding how services are accessing the database.

On this screen, you can see the database's network settings, including endpoint, port, security groups, and access rules. These can be used to connect various services to the RDS instance, but developers will also require the admin password to establish a connection. Contact our client for the admin password.

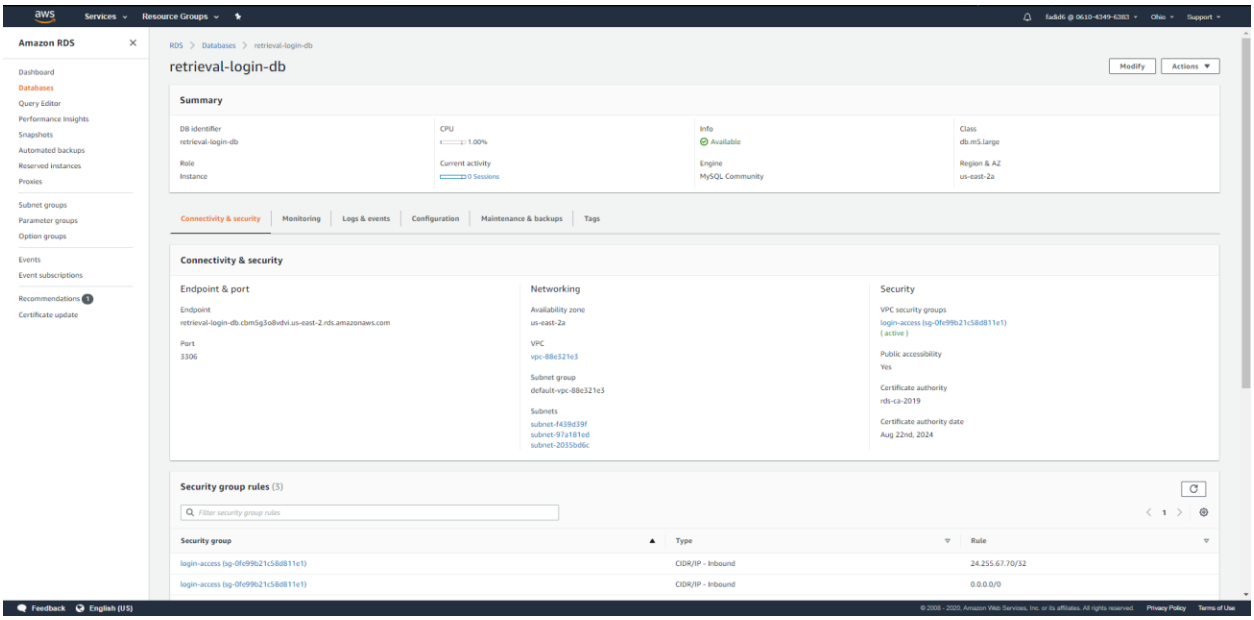

Fig. 7.2.1 - Console screen for RDS instance retrieval-login-db.

Many developer settings are configured during the creation of the instance and may cause server downtime if a developer attempts to reconfigure them. For example, public accessibility was set to "yes" as the database was being instantiated. To reconfigure the database, use the modify button in the top right corner (Figure 7.2.2). An account must have the proper IAM roles given to them by a supervisor to edit this instance's settings. In our case, we specifically needed RDS roles in order to create and edit an instance and configure it to fit our project's needs. Contact your supervisor to confirm that you have the needed permissions to configure this RDS instance.

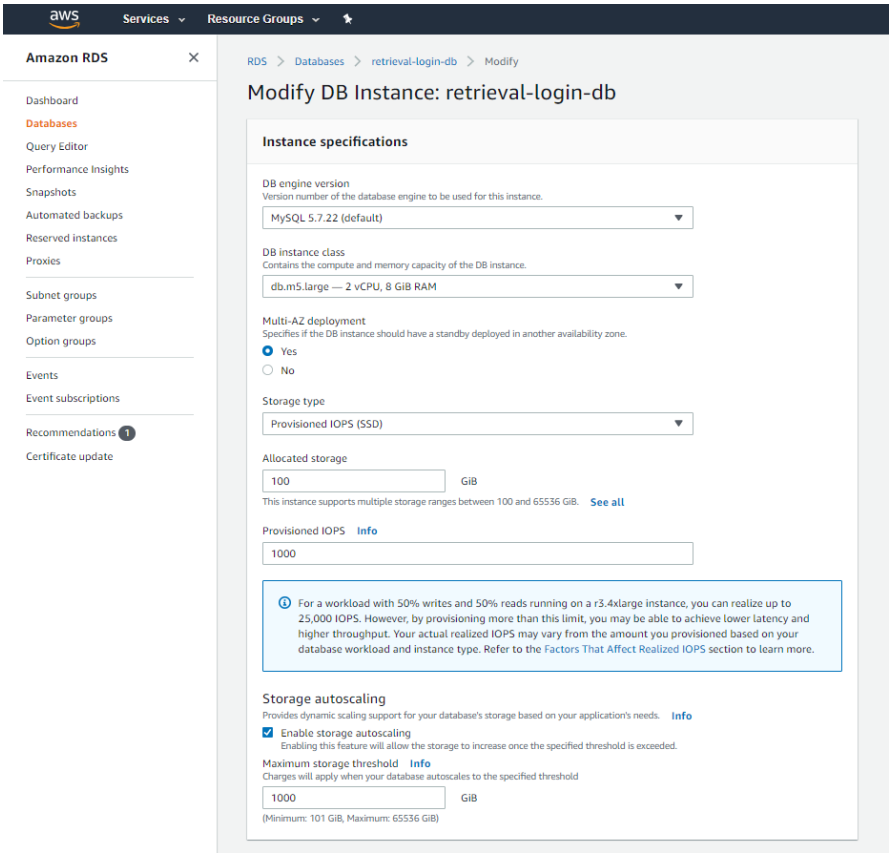

Fig. 7.2.2 - RDS console modify screen.

Also note that we created a custom VPC security group for this RDS instance, named *login-access*. This security group includes an inbound rule with source “0.0.0.0/0”, which allows access to the instance from any IP address (Figure 7.2.3). This was necessary to allow us to establish remote access through MySQL Workbench. To replicate, use the VPC security group link in Figure 7.2.1 and create a new security group, or edit a pre-existing group.

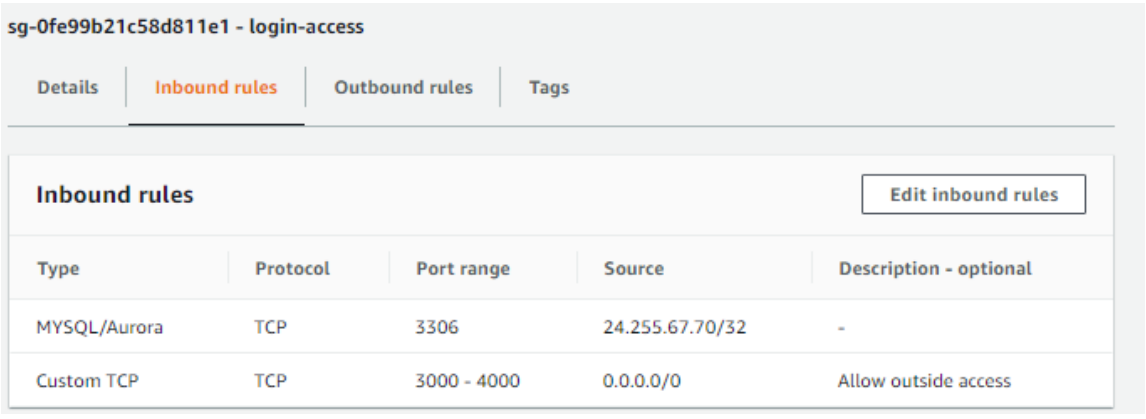

Fig. 7.2.3 - VPC inbound rules.

7.2.2. MySQL Workbench

To establish a connection between the RDS instance and MySQL Workbench, the developer must install MySQL and MySQL Workbench. This can be accomplished in a variety of ways such as through terminal commands. Our team used the MySQL community installer (found online at MySQL's download page), which allowed us to install all required software. Once downloaded, open MySQL Workbench. Find the add MySQL connection button (shaped like a plus sign next to "MySQL Connection"), which opens the screen shown in Figure 7.2.4.

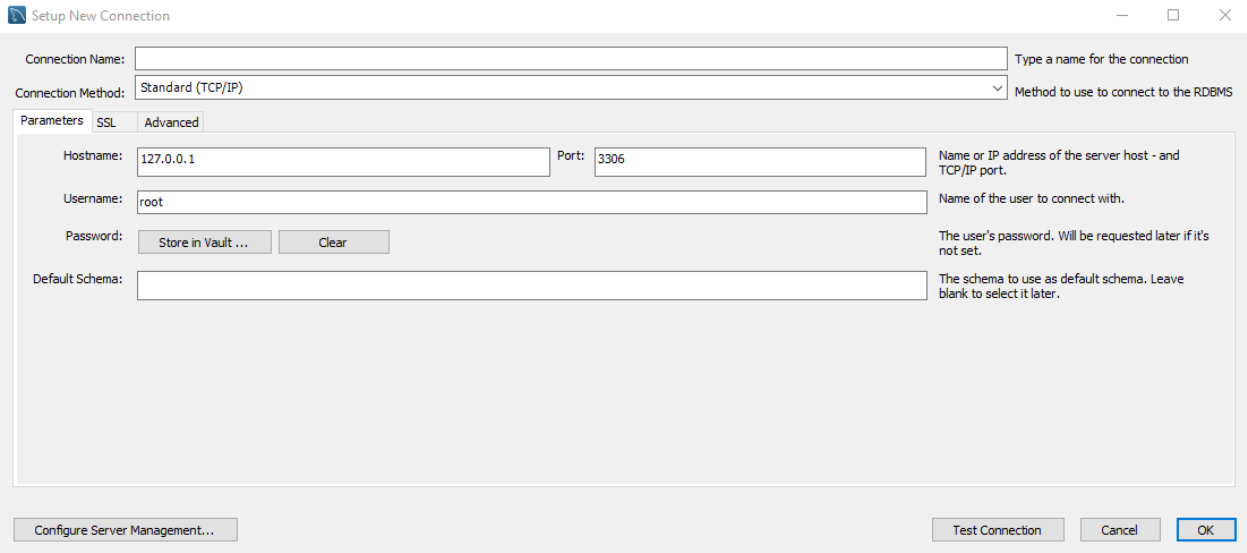

Fig. 7.2.4 - MySQL Workbench connection page.

Begin by giving this new connection a name in **Connection Name** (this could be anything, e.g., "Retrieval DB"). For our RDS instance, the connection can be established by providing the server endpoint in the **Hostname** box, "admin" in the **Username** box (may change depending on future development), and clicking on **Store in Vault** and then inputting the admin password. Then, click **Test Connection** to confirm that the connection was established successfully.

Once the connection is set up, click **OK** and double click the new connection to open the database navigation page (Figure 7.2.5, sections relevant to this manual are boxed in red). At the bottom of the **Navigator** tab, click **Schemas** to show the server's current deployed schemas. Expand the **verification** schema and find the **users** table (this is where our login data is stored). Right click the table and choose **Select Rows - Limit 1000** to select the first 1000 rows in the table and show them in Workbench. This will bring up the **Query Input Tab**, where the developer can run SQL queries to interact with the database, along with the **Query Output Console**, which displays the output produced by running queries above. There are also **View Options** to the right, which provide different functionality and information based on what the developer is trying to accomplish. The **Form Editor** is particularly useful in editing table entries.

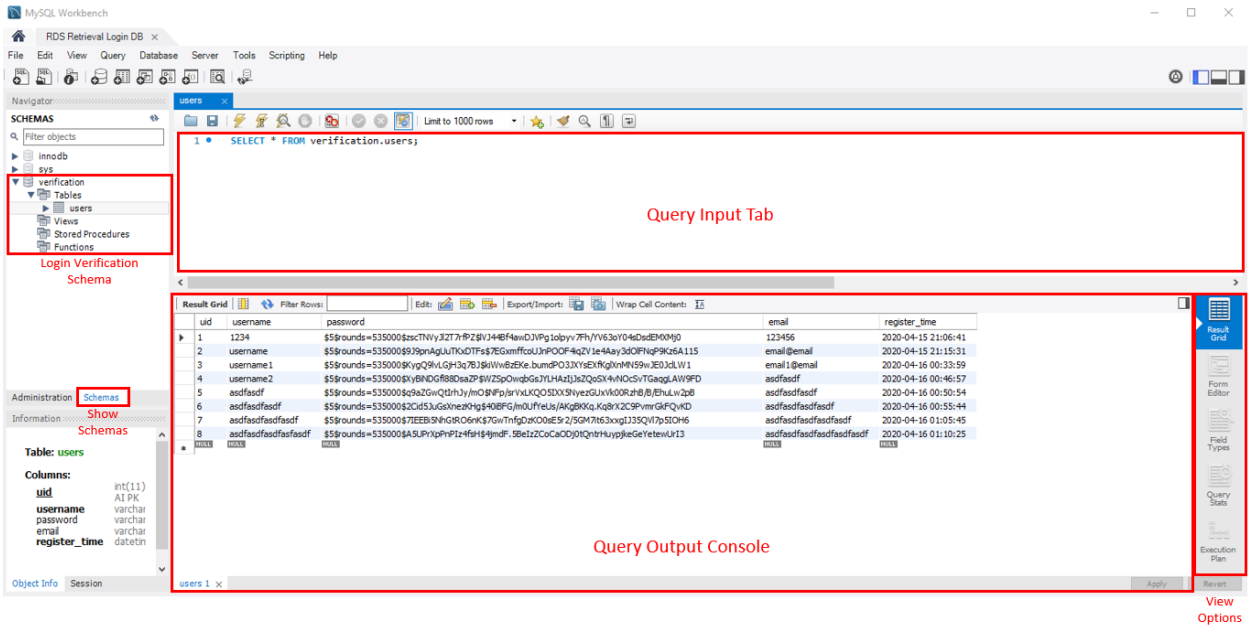

Fig. 7.2.5 - MySQL Workbench navigation page, annotated.

This was a brief introduction to Workbench. Developers well versed in SQL will find many of the software's other functionalities useful. These instructions are written with regards to MySQL Workbench 8.0. Newer or older versions may have a different interface for setting up a connection.

7.3. ElasticSearch

This part of the developer's manual pertains to the manual ingesting of data into an Amazon Web Service instance of ElasticSearch. This will outline how to set up an ElasticSearch domain and how to access it with commands to manually ingest files into it. Maintaining it is as simple as logging into the AWS console and going to ElasticSearch services to check on the domain health and amount of data ingested.

7.3.1. Creating a Domain

To establish an ElasticSearch domain, you simply go to the Amazon Web Services website and login. You then search for ElasticSearch Service and choose **Create a New Domain**. On the **Create ElasticSearch Domain** page, choose **Development and Testing** to access different options to customize your domain such as ElasticSearch version and policies.

7.3.2. Manually Ingesting Data

To upload a single file manually to the ElasticSearch domain, you simply run a cURL command with the specific document. You can also upload a JSON file that contains multiple documents by first creating the JSON file and uploading using the cURL command. For example:

“curl -XPUT "https://search-dlrl-elasticsearch-rmd75zwsfd7rmi7doj52bzbonu.us-east-1.es.amazonaws.com/tobacco_docs/_doc/0" -H "Content-Type: application/json" -d { /*contents of JSON*/ }”

could be used to upload a JSON string into the tobacco_docs index under the ID 0.

However, for this project, we were provided the documents and were simply tasked with uploading them, so cURL commands were used to upload the files individually for testing. The Kibana Dev Tools interface could be used in place of cURL commands as an alternative. The Kibana Dev Tools version of the above command is as follows:

```
PUT tobacco_docs/_doc/0
{
   /*contents of JSON*/
}
```

7.4. Front-End Application

This part of the developer’s manual is related to setting up the front-end application to run with the proper components. This includes connecting the login database and ElasticSearch service, building and running through Docker, and using ECR and ECS. Once you have access to the front-end code locally, proceed with the following steps.

7.4.1. Connecting database and Elaticsearch

To connect the login database and ElasticSearch to the front-end application, you must create a folder under the main folder called “config”. In that folder you must create a file called “settings.yaml” and insert the information for the database and ElasticSearch endpoints. The file should look similar to Figure 7.4.1.

Fig. 7.4.1 - Proper settings.yaml file.

Once this is done, the front-end application should be able to use this information to establish the connections to the login database and ElasticSearch. At this point, you should be able to run the application locally to test the connections. To run the application, use a proper terminal application and, in the main directory, run "**pip3 install -r requirements.txt**". Once that is finished, run "**python app.py**". You can see the application running on "**http://localhost:3001/**" through a web browser. Make sure your system has Python properly installed before running.

Logging in and registering users should be fully functional. If not, make sure in the login code that there are no lines that cause errors with your database. To get the ElasticSearch search UI functioning, you need to build the separate search application. Go into the directory called "reactivesearch" through the terminal and then run the command "**npm run build**". This should build the ElasticSearch search application so that it can be executed when the main front-end application is run.

7.4.2. Building the Docker container

Following are the instructions on how to properly build the front-end application into a Docker container. First, make sure you have Docker properly installed on your computer. Using a proper terminal application, "cd" into the main directory that contains the "**app.py**" file. Then run the following Docker command, "**docker build -t fek:1 .**" This builds the Docker container. Make sure to run the container locally so that everything is functioning properly in the app. This can be done by executing the command, "**docker run -it -p 3001:3001 fek:1**". Note that the ports can be different based on what you want to set them as. Make sure they match with everything in the code. Once the container is running, you can see the application running on "**http://localhost:3001/**" in any web browser.

7.4.3. AWS ECR and ECS

Now that we have a functioning Docker container for the front-end application, we can deploy it through AWS using the Elastic Container Registry (ECR) and Elastic Container Services (ECS). First, go to the AWS console and under the "**services**" tab, locate **Elastic Container Registry**. The page should look like Figure 7.4.2.

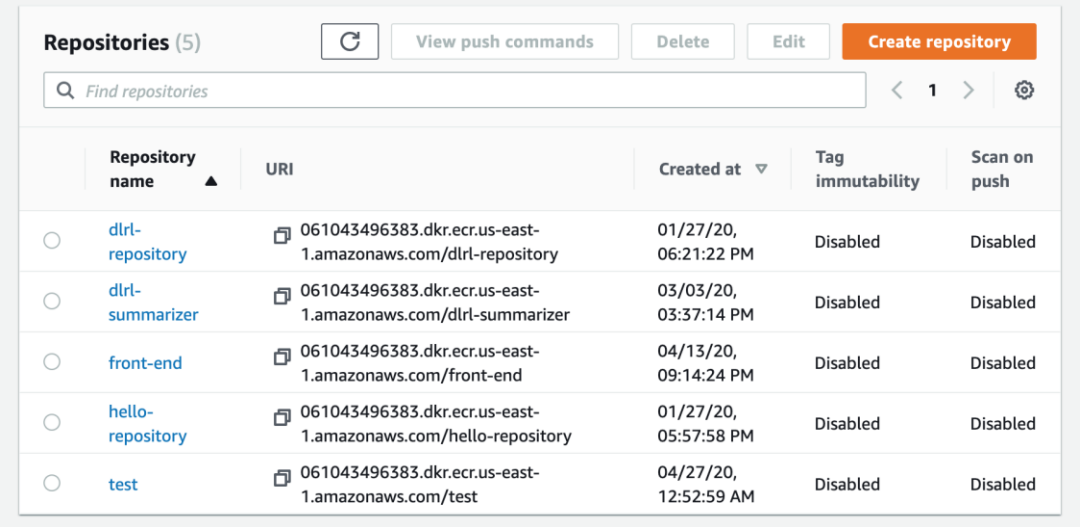

Fig. 7.4.2 - ECR page layout.

Once on the ECR page, click on the "**create repository**" then name the repository and click "**create repository**". Now we need to upload our container image to the repository. To upload the container, you will need the AWS CLI installed. Once that is installed, in the directory where you build the Docker container run the following command: "**aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 061043496383.dkr.ecr.us-east-1.amazonaws.com/test**" and then follow the prompts that appear. Note "**test**" is the name for the ECR repository and "**061043496383"** is your unique AWS account ID. Then build the Docker container if it needs to be rebuilt using the following command: "**docker build -t test .**" Note that "test" represents the name you want to call your container which can be changed to your liking. Then run the command: "**docker tag test:latest 061043496383.dkr.ecr.us-east-1.amazonaws.com/test:latest"**. Lastly run: "**docker push 061043496383.dkr.ecr.us-east-1.amazonaws.com/test:latest"** to push the container to ECR. In the end, your ECR repository should contain the container image depicted in Figure 7.4.3.

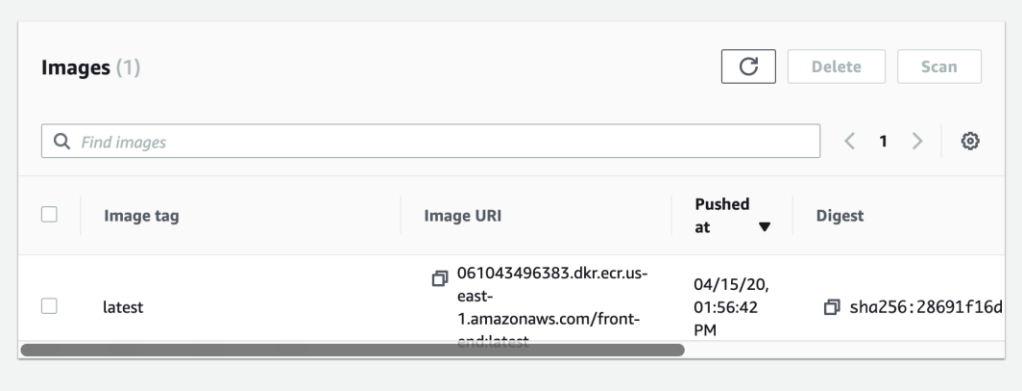

Fig. 7.4.3 - ECR container image.

Now to deploy using ECS, navigate to the Elastic Container Services page which should look like Figure 7.4.4.

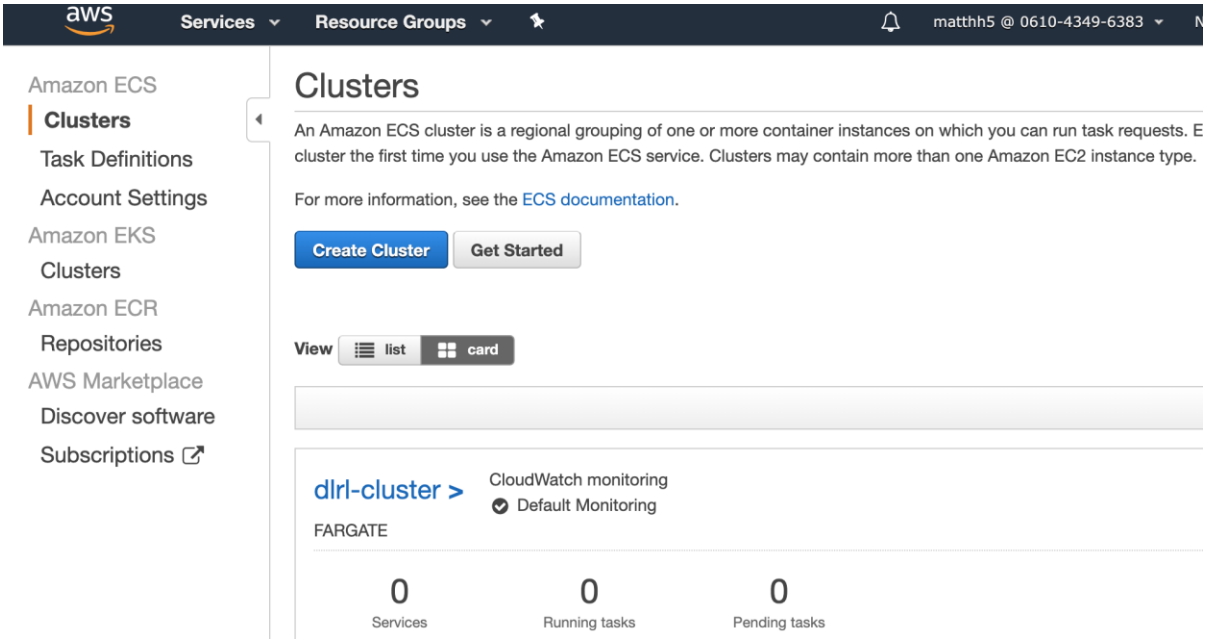

Fig. 7.4.4 - ECS front page.

If you have no clusters available, click "**Create Cluster**". Select **Network Only** for the cluster template. On the following page shown in Figure 7.4.5, input the desired name in **Cluster Name** and select **Create VPC**. Then click "**Create**" to create the cluster.

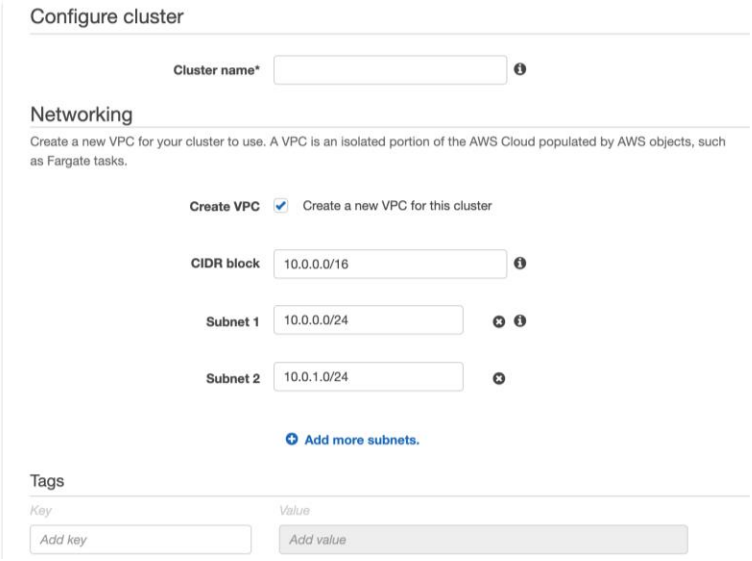

Fig. 7.4.5 - Configure cluster page.

Click on "**Task Definitions**" on the side of the ECS page, which should bring you to page shown in Figure 7.4.6.

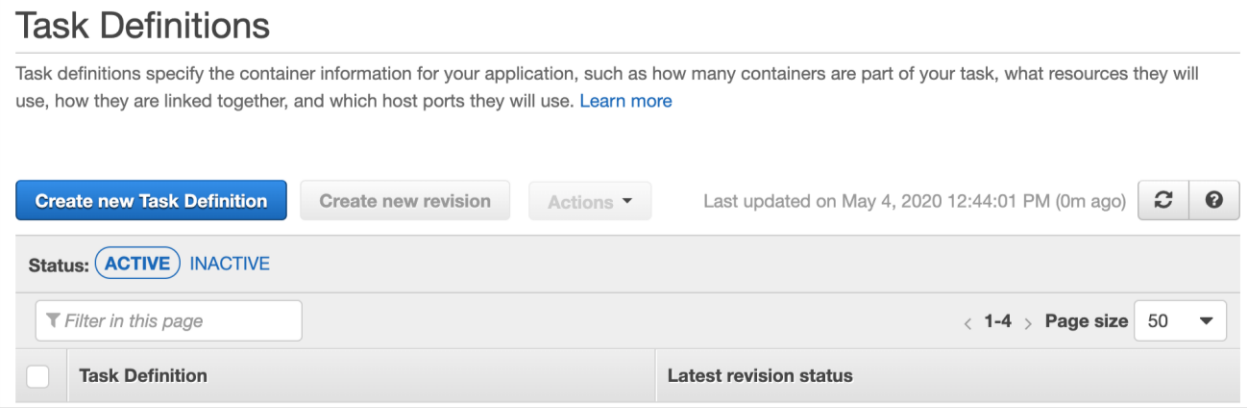

Fig. 7.4.6 - Task Definition page.

Select "**Create new Task Definition**" for the Container Definition, then on the next page, click on the **Fargate** option for "Select launch type compatibility" and click "**Next Step**". On the next page (Figure 7.4.7), type in a **Task Definition Name**.

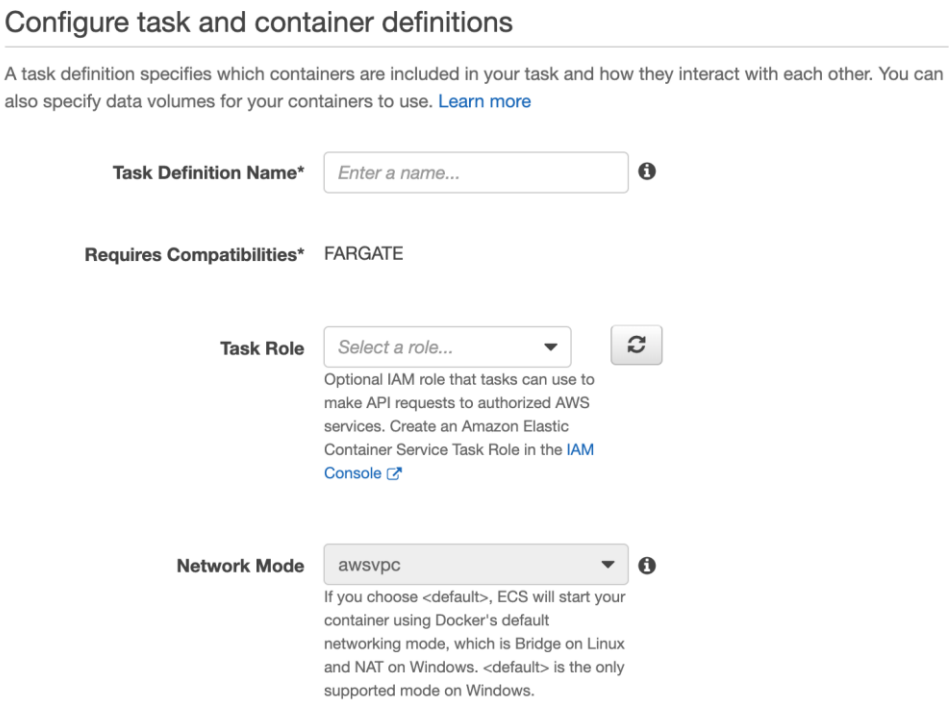

Fig. 7.4.7 - Task Definition configuration.

Then input the desired **Task Memory** and **Task CPU** on the section shown in Figure 7.4.8.

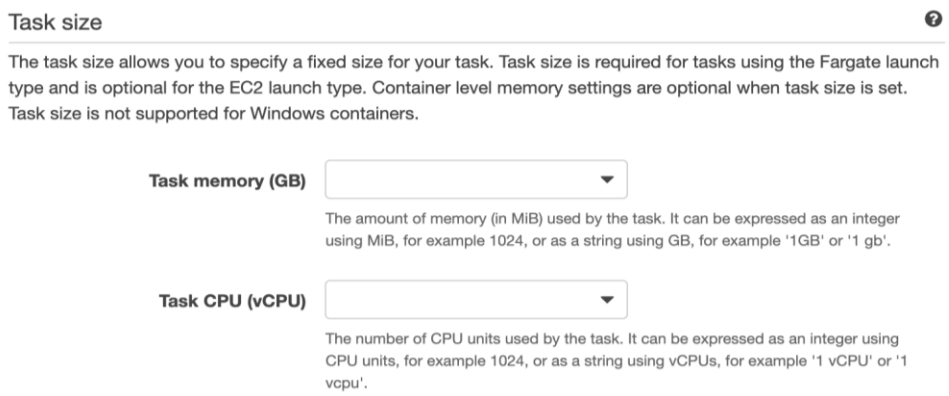

Fig. 7.4.8 - Task Size configuration.

Then click the "**Add container**" button which should bring up the following page shown in Figure 7.4.9.

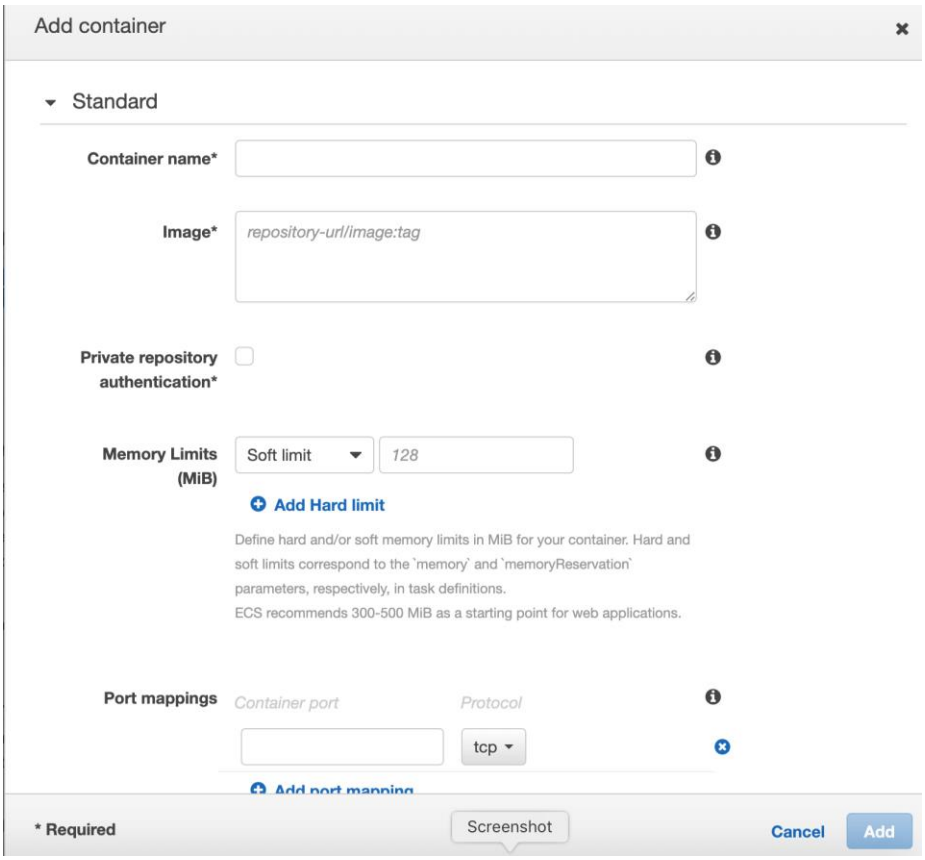

Fig. 7.4.9 - Add Container page.

In the Add Container page, fill out the **Container Name, Image**, which is the URI for your image in ECR, and **Port mappings**. Then click "**Add**". Then click "**Create**" for the task definition.

You should see the newly created Task Definition. Click on your new Task Definition and select the "**Actions**" toggle and select "**Run Task**". This should bring up the page shown in Figure 7.4.10.

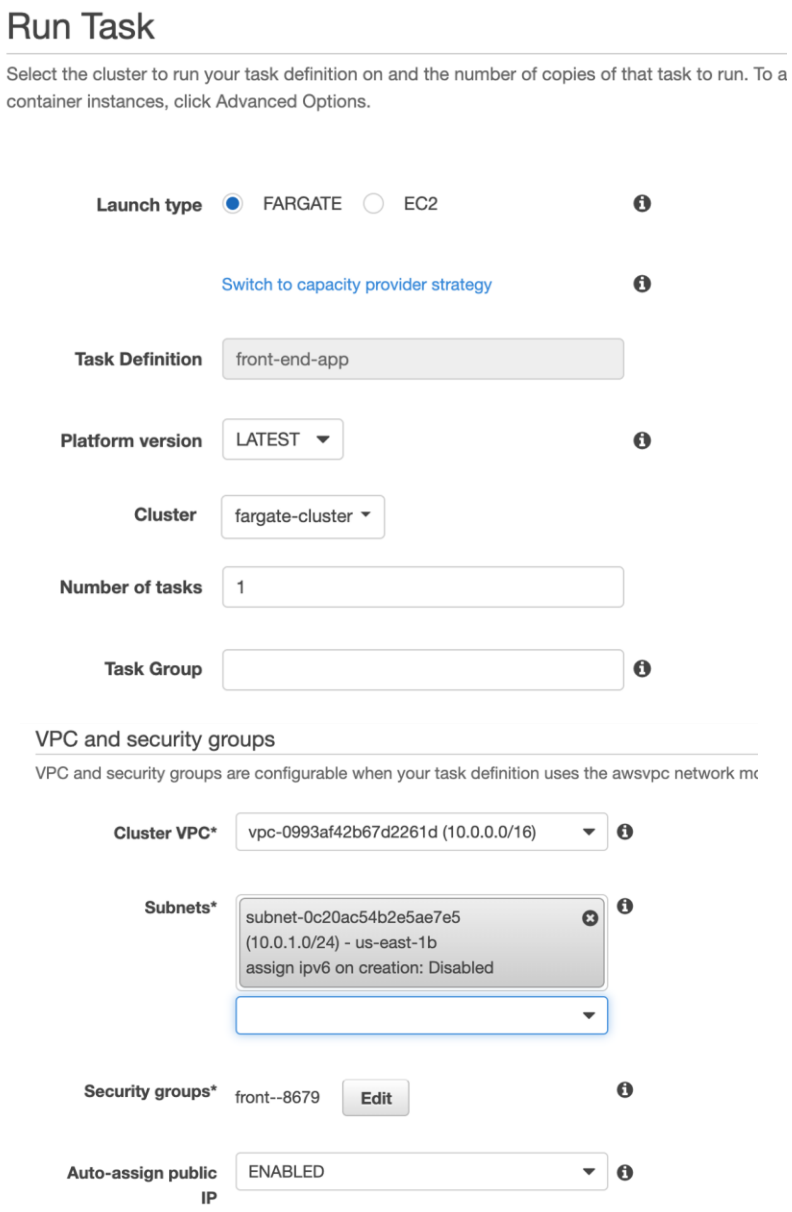

Fig. 7.4.10 - Run Task Configuration page.

On the Run Task page, select **Fargate** for the **Launch Type** then choose the desired cluster, then choose the desired **Subnets**. Then click **Run Task**. It should show your task running in your cluster's page as shown in Figure 7.4.11.

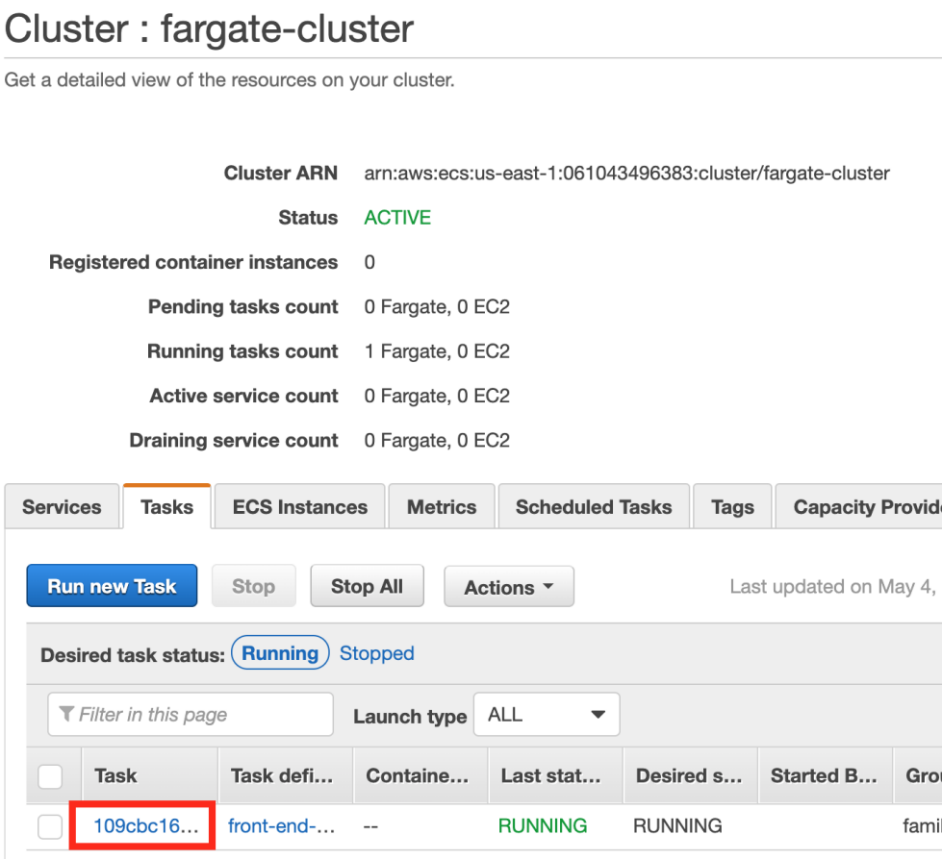

Fig. 7.4.11 - Running tasks on Cluster.

Click on the link under the Task column indicated by the red square. This should bring up a page with more information about the running task. On that page, find the **ENI Id** link under **Network** and click on the link, which should bring up the page shown in Figure 7.4.12.

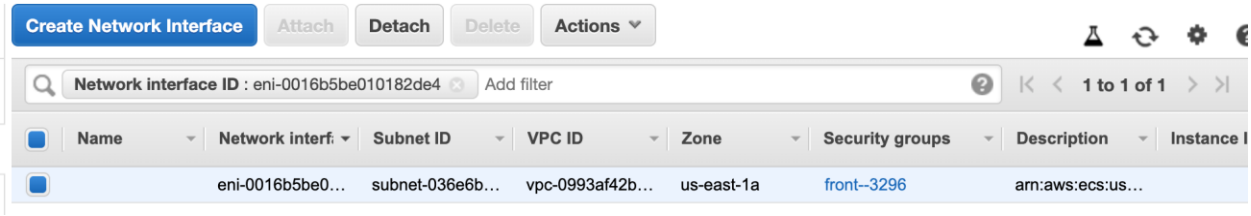

Fig. 7.4.12 - ENI ID page.

On this page, click on the link under **Security Groups**, which should bring up the Security Groups page (Figure 7.4.13).

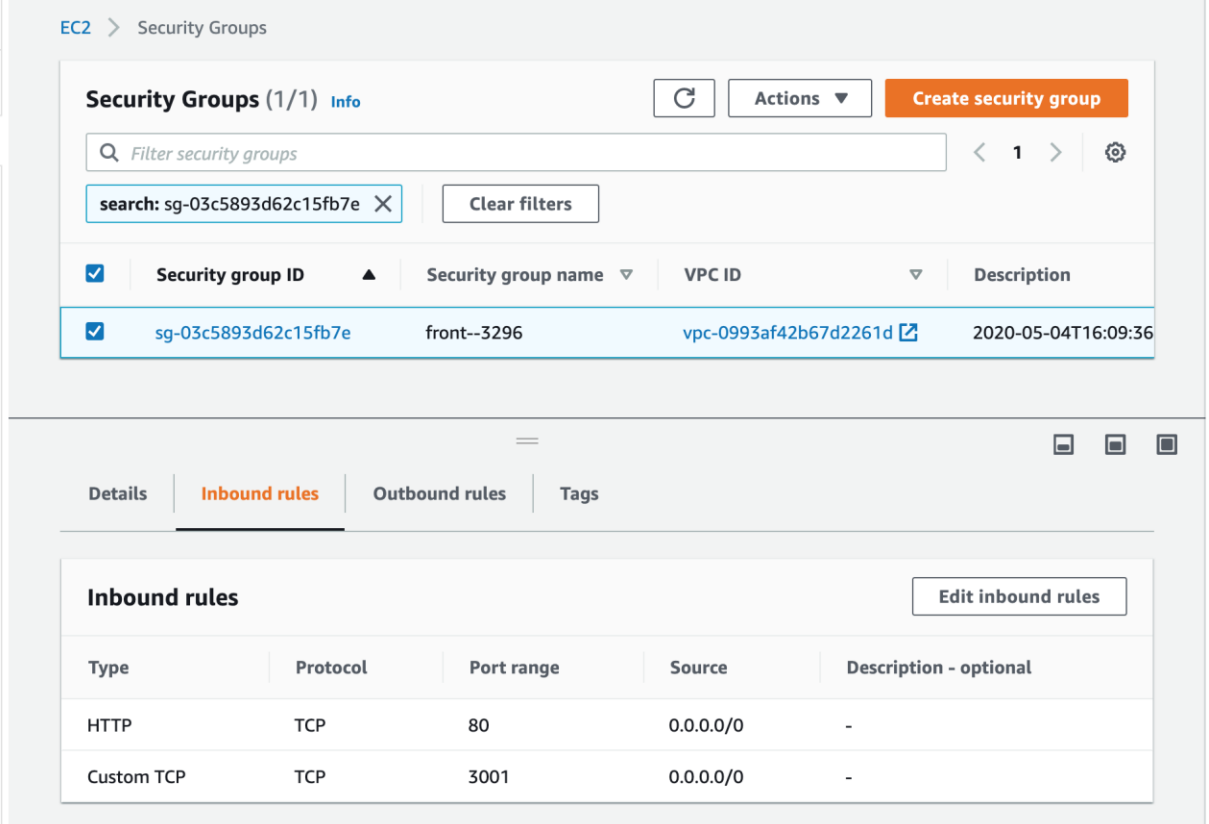

Fig. 7.4.13 - Security Groups page.

On the Security Groups page, find the I**nbound rules** and edit it so that there is an added rule for the port used by the app. The port we used is "**3001**". Then go back to the page with the network information about the running task and find the section called **Network**.

Fig. 7.4.14 -Network information for running task.

Locate the **Public IP**, this is where you will find the application container running on the network. To view the running application, copy the Public IP and paste it in the URL in any browser. Make sure to add the port to the end of the IP so that it looks similar to "**http://54.157.117.77:3001/**". You should see the application running.

8. Lessons Learned

In this section we discuss the roadblocks we faced in implementation along with the lessons we learned from tackling these issues.

8.1. Scheduling and COVID-19

As precautionary safety measures to combat the COVID-19 outbreak, Virginia Tech decided to close campus for the Spring 2020 semester and make all classes online, along with extending spring break by an extra 7 days. As a result of this, our team cannot have physical meetings to plan the project or to meet with our client in person. This has made project status reports less frequent and has forced us to rely more on online communication and collaboration. The combination of these factors has slightly lowered our productivity as well as made it significantly harder to troubleshoot project tasks as a team.

Despite the complications in our scheduling, our team managed to complete all requirements of the project despite the setback in scheduling. The timeline that we were following before was pushed back slightly to compensate for the missing week but the team managed to redefine the needed tasks to finish the deliverables.

8.2. Local Database Docker Integration

After creating a stable connection between a local Python script and a locally hosted login database, we looked into dockerizing the Python script and maintaining the database connection through the container. This proved more difficult than we expected, since multiple attempts were met with a failed connection test. According to our research of the issue, we believe that the locally hosted database was hard to access from within the Docker container due to the database taking up a host machine's port and Docker subsequently being unable to listen to that port. With this in mind, we figured it would make sense to try connecting a remote database, and bypass the port usage issue.

While it's possible to test the remote verification process by expanding this database's connection to allow team members on other machines to access information inside of it (Container ran on one machine, database ran on another), we figured that testing the RDS database connection would be more worth our time, so we decided to move on to working on getting the AWS RDS instance running. This particular issue gave our Docker members a better understanding of locally run databases and how they interact with the host machine's port communication capability.

8.3. RDS and MySQL Workbench

Through the process of setting up our RDS instance, we gained a much better understanding of configuring many of the settings available in the RDS console. This includes configuring the available storage for holding login information, as well as available bandwidth to accommodate the demand on the login process. We also learned about the various security settings provided in RDS, such as public accessibility along with how VPC security groups affect what ranges of IP addresses are granted access to the database instance.

Once the RDS instance was active, we learned a lot about the process of establishing a connection to that instance. This concerned the importance of maintaining security surrounding admin login credentials, along with the connectivity infrastructure of our Front-End application and how to incorporate database access. We also used this knowledge to establish a connection to a local MySQL Workbench, which allowed us a lot of freedom in how to interact with the database, along with providing a demo for future developers looking to use this software. The inclusion of MySQL Workbench was an essential piece in creating an effective MySQL development environment, one in which we had established a connection to our remote database on AWS. This connection is practically useful to developers, but is non essential to the project's core services, which are all hosted on AWS.

8.4. Ingesting Data into ElasticSearch

With ElasticSearch being a new concept to the entire team, a lot of research had to be made. Over the course of the semester project, we learned how to set up a domain and instance of ElasticSearch on our AWS accounts, how to communicate with an instance so that we can transfer the JSON files that have been provided for us on the CS Container Cloud, and how to set up an automatic form of ingestion using AWS Lambda.

8.5. Front-End

Since the front-end application was written completely by a different team, it took some time to fully understand how the application was integrated with all the other components such as the login database and ElasticSearch. We learned that Flask applications connect to databases and services through a global settings.yaml file. We spend a lot of time looking through each file of the code trying to find where the database was connected, but instead all we had to do was create a universal file. Another lesson we learned was that we had no idea the search UI was a completely separate application. We were confused for a long time on why nothing showed up for the search feature. All we had to do was build the search app separately.

9. Acknowledgements

We would like to acknowledge our client Rahul Agarwal who worked as part of the Integration and Implementation Team of the original CS 5604 project for his help and guidance up to this point in the project. For contact: email rahula@vt.edu.

We would like to acknowledge Dr. Edward Fox for his continued guidance and advice throughout this semester. For contact: email fox@vt.edu or visit http://fox.cs.vt.edu/.

We would like to acknowledge the INT[1] and FEK[7] groups of the CS 5604 class for creating the original project.

We would like to acknowledge Bill Ingram for contributing the ETD documents (email waingram@vt.edu), along with Dr. David Townsend for contributing the Tobacco documents (email dtown@vt.edu).

Finally, we would like to acknowledge the IMLS LG-37-19-0078-19 Project Grant for the funding of the project.

10. References

[1] R. Agarwal, H. Albahar, E. Roth, M. Sen, and L. Yu, "Integration and Implementation (INT) CS5604 Fall 2019," *VTechWorks*, 11-Dec-2019. [Online]. Available: http://hdl.handle.net/10919/96488. [Accessed: 04-Feb-2020].

[2] ETDs: Virginia Tech Electronic Theses and Dissertations. *VTechWorks*. [Online]. Available: https://hdl.handle.net/10919/5534. [Accessed: 14-Feb-2020].

[3] K. K. Kaushal, R. Kulkarni, A. Sumant, C. Wang, C. Yuan, and L. Yuan, "Collection Management of Electronic Theses and Dissertations (CME) CS5604 Fall 2019," *VTechWorks*, 23-Dec-2019. [Online]. Available: https://hdl.handle.net/10919/96484. [Accessed: 03-May-2020].

[4] Y. Li, S. Chekuri, T. Hu, S. A. Kumar, and N. Gill, "ElasticSearch (ELS) CS5604 Fall 2019," *VTechWorks*, 12-Dec-2019. [Online]. Available: http://hdl.handle.net/10919/96310. [Accessed: 10-Mar-2020].

[5] R. S. Mansur, P. Mandke, J. Gong, S. M. Bharadwaj, A. S. Juvekar, and S. Chougule, "Text Analytics and Machine Learning (TML) CS5604 Fall 2019," *VTechWorks*, 29-Dec-2019. [Online]. Available: http://hdl.handle.net/10919/96226. [Accessed: 11-Feb-2020].

[6] S. Muhundan, A. Bendelac, Y. Zhao, A. Svetovidov, D. Biswas, and A. Marin Thomas, "Collection Management Tobacco Settlement Documents (CMT) CS5604 Fall 2019," VTechWorks, 11-Dec-2019. [Online]. Available: https://hdl.handle.net/10919/96437. [Accessed: 03-May-2020].

[7] E. Powell, H. Liu, R. Huang, Y. Sun, and C. Xu, "Front-End Kibana (FEK) CS5604 Fall 2019," *VTechWorks*, 13-Jan-2020. [Online]. Available: http://hdl.handle.net/10919/96418. [Accessed: 04-Feb-2020].

[8] Truth Tobacco Industry Documents Library. [Online]. Available: https://www.industrydocuments.ucsf.edu/tobacco/. [Accessed: 14-Feb-2020].