

Table of Contents

[Introduction 3](#_Toc70931959)

[Background & Design 3](#_Toc70931960)

[AWS Terminology 3](#_Toc70931961)

[Pipeline Design 4](#_Toc70931962)

[Scope of this Document 5](#_Toc70931963)

[Quick Start 7](#_Toc70931964)

[S3 Buckets 9](#_Toc70931965)

[Relational Database 12](#_Toc70931966)

[Creating the Database Instance in RDS 12](#_Toc70931967)

[Connecting to your Database 15](#_Toc70931968)

[MySQL Workbench 15](#_Toc70931969)

[Python 19](#_Toc70931970)

[Creating the Tables for the Database 21](#_Toc70931971)

[Viewing the Data 23](#_Toc70931972)

[Lambda Functions 24](#_Toc70931973)

[Amazon Chalice 24](#_Toc70931974)

[Data Cleaning 29](#_Toc70931975)

[Database Update 30](#_Toc70931976)

[API Interface 32](#_Toc70931977)

[Query Generator 33](#_Toc70931978)

[API Gateway 34](#_Toc70931979)

[Web-Facing Query & Tracker Page 34](#_Toc70931980)

[Participant Tracker 36](#_Toc70931981)

[Updating the Table 36](#_Toc70931982)

[Transcript Data 38](#_Toc70931983)

[Appendix A 40](#_Toc70931984)

[Appendix B 41](#_Toc70931985)

[Appendix C 43](#_Toc70931986)

# Introduction

This document was created by a School of Data Science Class of 2021 Capstone Team at the University of Virginia that partnered with the Collaboratory Replication Lab. The team members from the School of Data Science are Jordan Machita, Taylor Rohrich, Yiran Zheng, and Yusheng Jiang. More information about the Collaboratory Replication Lab can be found [here](https://www.edreplication.org/). The purpose of this document is to provide detailed instructions for implementing an end-to-end data pipeline in Amazon Web Services (AWS). These instructions include how to perform each of the necessary steps in AWS as well as how to tailor the code (written in Python) to the specifications of your data. All the code discussed in this document is available at [GitHub Link](https://github.com/taylorrohrich/MSDS_SERA_capstone) for you to utilize for your project.

# Background & Design

Currently, the field of education is undergoing a replication crisis, meaning that few of the studies being conducted and published are replications of existing research despite the importance of such replication. This problem in education research has prompted the development of the Collaboratory Replication Lab out of the University of Virginia’s School of Education. One project being undertaken by the lab is the Special Education Research Accelerator (SERA), a platform that will leverage crowdsourcing to conduct large-scale replication studies in special education. Our capstone team through the School of Data Science has designed and implemented a data pipeline for an existing research project with the intent that it will serve as a model for the data pipeline component of SERA.

The research study that we used to pilot the implementation of this pipeline consisted of two types of data. The first is survey data in the form of CSVs that is the result of participants in the study filling out surveys in Qualtrics. The second is transcription data that is the transcribed from the video/audio of simulation sessions that the participants complete throughout the study. In our pipeline, we focus primarily on the survey data, but we will briefly cover some aspects of the transcription data. For greater detail on the inspiration for the project and the study used to pilot this pipeline (TeachSIM), as well as how AWS was chosen as the platform, please see the paper titled “Designing a Replicable Data Infrastructure for Education Research” located in the GitHub repository.

## AWS Terminology

There are some key terms specific to AWS that need to be addressed before being able to fully digest our architecture design. *Amazon S3 (Simple Storage Service)* is the storage infrastructure service provided through AWS. Essentially, S3 buckets are where users can upload and store data, such as csv files. *AWS Lambda functions* are essentially serverless snippets of Python code that run in the Cloud in response to events, such as data being uploaded to an S3 bucket. Lambda functions, and serverless computing in general, have the added benefit of abstracting away the management of servers and simply focusing on writing code. The pricing model for serverless computing is beneficial and scalable. Users only pay for the time that the code is running. *Amazon RDS* is a distributed relational database service offered by AWS to host a database in the cloud. *Amazon API Gateway* is a service that enables the development and management of application programming interfaces (APIs) through AWS. An *API* allows a user or program to communicate with another program or database.

## Pipeline Design

A simple design of our data pipeline is shown in Figure 1. A researcher interacts with the pipeline at two places. First, they input the raw data. Second, they query the database using the API. We will cover how to implement and connect each piece of the pipeline but first we will provide a broad overview for context.

Going sequentially through the pipeline, from left to right in the diagram, we start with the researcher uploading raw survey data in the form of a CSV to the S3 bucket designated for raw data. The uploadof data triggers a Lambda function that hosts the data cleaning techniques specific to the project and produces a CSV file with the cleaned data (from the raw output of Qualtrics). This file is uploaded by the Lambda function to the S3 bucket designated for cleaned data. The upload of the cleaned data then triggers a second Lambda function that takes the cleaned data and uploads it to the MySQL database hosted on RDS. From here, the cleaned data can be accessed by researchers through an API. We have built a web-facing API that serves as a bridge between the researcher and the database so that the researcher can query the database simply by checking boxes on a form and receive the data as a downloadable CSV, requiring no knowledge of SQL. Additionally, we have built a Participant Tracker that tracks which surveys each participant has completed. We present the Participant Tracker in a web-facing format as well.

A picture containing text, electronics

Description automatically generated

Figure : Data Pipeline Design

## Scope of this Document

Now that we have covered the basics, the remainder of this document will go into greater detail and provide instructions for implementing a similar pipeline within your own AWS account for your own project. Appendix A contains a checklist of tasks to make sure you follow each step covered in the documentation. Additionally, the remaining appendices will cover a more in-depth code review for those with more advanced knowledge of Python, who want to dig deeper into understanding how each component works. Note that while it was easiest to present the pipeline design sequentially, mimicking how data flows through the pipeline, implementation is not able to be performed strictly in that order due to dependencies. Thus, the rest of the documentation will cover each component as is easiest for implementation. For convenience, we will continually return to the diagram in Figure 1 to illustrate which aspect of the pipeline we are covering.

While we order the sections by how we recommend implementing the entire pipeline, if you are interested in only a specific component of the pipeline, you can skip to that section. Each section has detailed enough instructions to implement that portion of the pipeline independent of the larger picture. For example, if you only wish to learn how to write and deploy a Lambda function, reading and following the section *Lambda Functions* will be sufficient for your needs. However, if you are doing so as part of a pipeline then there are dependencies, such as the S3 buckets accessed in your Lambda functions must already have been created. This document provides sufficient instructions for users whether they are concerned with only a particular component or the entire data pipeline.

# Quick Start

This section outlines a streamlined path to automatically deploy an example of our entire pipeline in several quick steps. If you are interested in a more in-depth explanation of each of the components in our pipeline, please proceed to the sections below. These steps assume you already have an AWS account and knowledge of terminal commands. Note that the database in this pipeline will be based on the TeachSIM study we used to design the pipeline. If you wish to implement this pipeline for a study with different specifications, you will need to follow the more detailed instructions outside of this section.

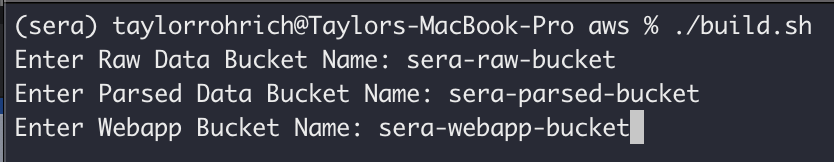
1. Clone our [repository](https://github.com/taylorrohrich/MSDS_SERA_capstone). For steps on how to clone a repository, look [here](https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository).

2. Navigate to the [aws subdirectory](https://github.com/taylorrohrich/MSDS_SERA_capstone/tree/main/aws) within this cloned folder in your terminal.

3. Run the following command in terminal:

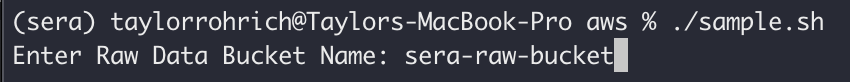


This will prompt you to name the S3 buckets that will be required to spin up our pipeline. Please note: S3 buckets need to be globally unique. An example of this process is shown below.



After naming the buckets, the process will spin up the entire pipeline. This process may take some time as the database takes some time to create.

4. After the pipeline has been built, the pipeline does not yet have data in it. To add the sample files to your AWS raw data S3 bucket, run the following command, which will prompt you for the name of your raw data S3 bucket:



5. After the steps above, the pipeline will be created; navigate to your AWS console to see the resources created.

6. To delete the entire pipeline, run the following command in your terminal:



# S3 Buckets

This section covers how to set up an S3 bucket in AWS. As seen in the diagram below, there are two S3 buckets in the data pipeline and they are highlighted in blue.

A model of a space ship

Description automatically generated with low confidence

First, login to your AWS Management Console. From the services dropdown in the top-left corner, select S3 under the Storage category. If you cannot locate the services dropdown, you can simply use the search bar to search for and select S3. See below for an image of the Services dropdown.

A screenshot of a computer screen

Description automatically generated with medium confidence

Now that we are on the Amazon S3 page, we can see a list of all of our existing S3 buckets as seen below. To create a new bucket, click the orange Create Bucket button in the top right corner.

Graphical user interface, application, email, website

Description automatically generated

Here, you will specify the bucket name and AWS region. **Make sure that whichever region you choose now, you use in every aspect of the pipeline because that is required for integration of components.** We name our raw data bucket “sera-raw-data” and our clean data bucket “sera-parsed-data”, but you are free to choose whatever names you would like.

Graphical user interface, text, application, email

Description automatically generated

We can leave the rest of the settings to their default values. This includes leaving the box checked to “Block *all* public access”, having Bucket Versioning set to “Disable”, and Server-side encryption to “Disable”. You can then hit the orange “Create Bucket” button at the bottom of the screen.

Now that you have successfully created a bucket, it will appear on your S3 Buckets page that we showed above. Make sure to create both the bucket for raw data and the bucket for cleaned data. If you are following along for the entire pipeline, the S3 buckets portion of the pipeline is now complete! This means you can check off the following items on your checklist from Appendix A.

* Create an S3 bucket for raw data
* Create an S3 bucket for cleaned data

# Relational Database

## Creating the Database Instance in RDS

This section provides instructions for how to create a database in Amazon RDS (Relational Database Service). See the pipeline diagram below to picture where we are in the pipeline.

A model of a space ship

Description automatically generated with medium confidence

We will be using the AWS Management Console to perform this task. From the Services dropdown or search bar, select RDS, which is under the Database service category. Below is an image of the page you should see immediately after clicking RDS.

Graphical user interface, text, application

Description automatically generated

From here, click on DB Instances under Resources in the center of the screen, which will take you to the page seen below.

Graphical user interface, text, application, email

Description automatically generated

Next, click the orange Create database button in the top right corner. This brings us to the page seen below.

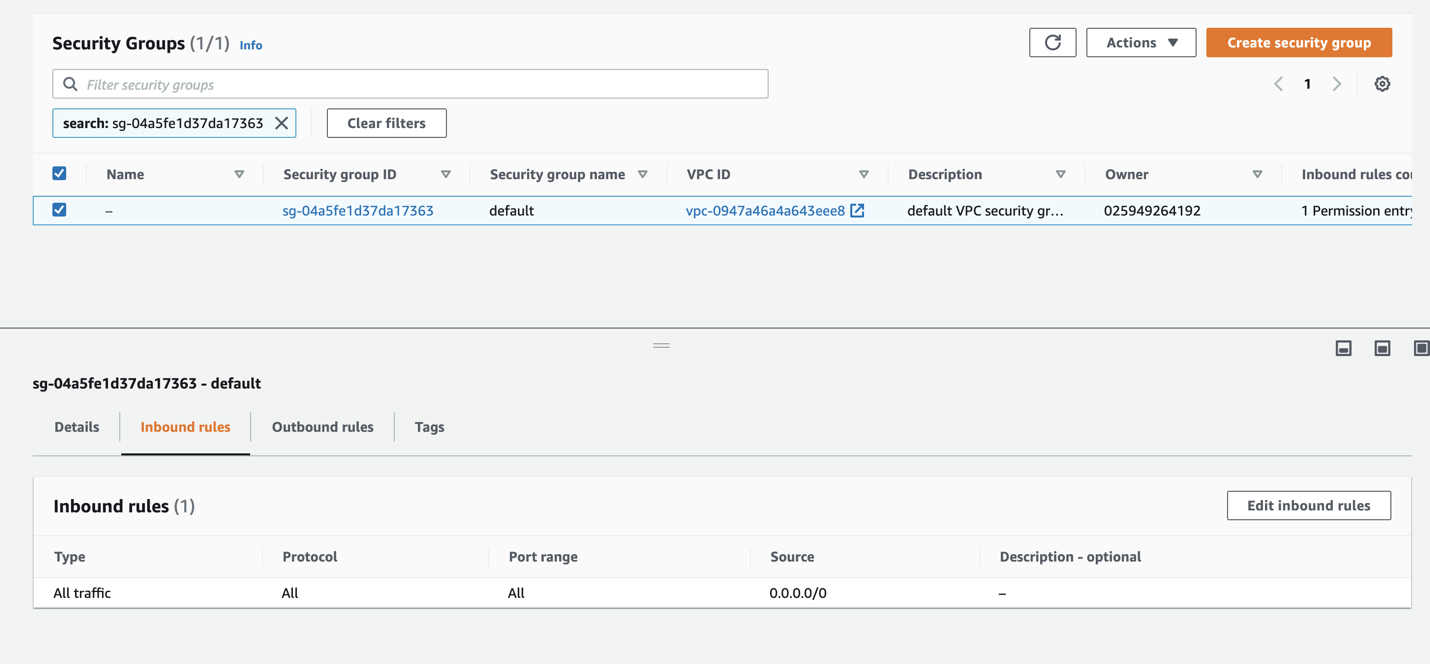
Graphical user interface, application

Description automatically generated

We will now take you through the configuration choices for creating your database. One of the most important selections in the Engine. We use MySQL and the remaining components of the pipeline are built based on this choice. Thus, we recommend selecting the same option if you plan to implement our pipeline as this will make our code the most reusable. With that being said, it is likely possible to adapt much of our code to a different relational database engine, but this will require more work to implement and adapt.

When setting up your database, make sure that you make note of the username and password that you choose here because they will be required to establish a connection to the database. Additionally, make sure that you choose to have the database be publicly accessible by selecting “Yes” under “Public access”. Outside of these choices, you can use the default options set using “Standard create” as seen above.

After creating the database, ensure that the proper security group settings are configured. Under the ‘Connectivity & Security’ section of the database console, click on the VPC security group. Ensure that the security group’s ‘Inbound Rules’ Source is set for 0.0.0.0/0 as in the image below:



Now that you have created your database instance, you will see it on the DB Instances page as shown previously. If you are following along with the entire pipeline, you can check the following box from the Appendix A guide.

* Create the database instance in Amazon RDS

## Connecting to your Database

Throughout the pipeline, we will use Python to connect to our MySQL database. We can also connect to it using the MySQL Workbench. We will cover both ways in the following sections.

### MySQL Workbench

You can follow the instructions to download and install the MySQL Workbench here: <https://dev.mysql.com/downloads/workbench/>

When you open the MySQL Workbench app, you will connect to your database by clicking the plus sign inscribed in a circle to the right of MySQL Connections as seen below.

Graphical user interface, application

Description automatically generated

This brings up the following window to create a new connection.

Graphical user interface

Description automatically generated

We will fill this out according to the information provided in AWS when you click on your database instance. An example of this screen in AWS is seen below. In MySQL Workbench, set the Connection Name at the top to whatever you would like. Leave the Connection Method as Standard (TCP/IP). Under Parameters, set the Hostname to the Endpoint seen on the AWS page (simply copy and paste). For example, our Hostname would be teachsim-db.cfmjo2xhkjgh.us-east-1.rds.amazonaws.com. Next, set the Port to the Port under the Endpoint in AWS. The Username and Password will be what you set when creating your database (common usernames are root and admin). You can leave the default schema blank. Click Test Connection at the bottom of the page to see if you can successfully connect to the database. If this is successful, click OK, if not, then make sure your information was filled out correctly and the database has the proper permissions, such as allowing public accessibility.

Graphical user interface, application, website

Description automatically generated

Before we connect to the database in Python, let us create a new schema in the database as this schema essentially be what we refer to as the database from here on out. Conceptually, this can be a bit confusing because when we refer to a database, we often mean what MySQL calls a schema (a collection of related tables). While the actual database instance in RDS/MySQL can contain multiple of these schemas (databases). In our database, we create only one schema, and we essentially refer to it as the database. First open your database connection by double clicking on it from the MySQL Workbench homepage that we showed above (with the big “Welcome to MySQL Workbench”). This will take you to a page that looks like below. In the row of options under the house at the top, click the fourth from the left (“Create a new schema in the connected server”) that looks like a cylinder divided into three parts horizontally.

Graphical user interface, text, application

Description automatically generated

This brings up the following page where you name the schema. We called ours teachsim and it was already visible on the left-hand side above.

Graphical user interface, text, application

Description automatically generated

Once you create the schema, you are done with MySQL Workbench for now. Note that while we almost exclusively use Python to query and test the database, you can also run queries in MySQL Workbench, which makes it another tool for you to use at your convenience.

* Create the schema instance in MySQL Workbench

### Python

Once a schema instance is in the database, we can connect to the database in Python. There are several ways to do this. It is also possible to connect the overall database in Python and create the schema instance there instead of MySQL Workbench, so if you are more familiar with that approach, go for it.

The way we connect to the database in Python is through the mysql.connector package. In Python, you can simply run “import mysql.connector”, which you may need to install into your environment if you do not already have the package installed. We can then connect to our database with just a few lines of code. This is going to be very similar to how we connected to the database from the MySQL Workbench. The code is as follows:

|  |
| --- |
| import mysql.connector  endpoint = "seratestdatabase.c4cjk1vto1om.us-east-2.rds.amazonaws.com"  port = "3306"  usr = "username”  pswd = "password"  region = "us-east-2b"  dbname = "teachsim"  cnx = mysql.connector.connect(user=usr, password=pswd, host=endpoint, database=dbname)  cursor = cnx.cursor(buffered=True) |

You would replace the endpoint, port, username, password, and region with those of your database instance from the AWS RDS page that we showed earlier exactly how we did for MySQL Workbench. **One extremely important note:** the dbname variable that we define is the name of your *schema instance* not the name of your database in RDS. Again, this is the schema that you just created and names in MySQL Workbench, not the name of the database you created in RDS through the AWS Management Console. This is a very common mistake that can lead to much frustration because the connection will fail.

Another common way of connecting to the database that we sometimes use is by using the pymysql package instead of mysql.connector. This functions almost the exact same and is set up with the following lines of code:

import pymysql

endpoint = "seratestdatabase.c4cjk1vto1om.us-east-2.rds.amazonaws.com"

port = "3306"

usr = "username”

pswd = "password"

region = "us-east-2b"

dbname = "teachsim"

cnx = pymysql.connections.Connect(user=usr, passwd=pswd, host=endpoint, db=dbname)

cursor = cnx.cursor()

It should be easy to see the parallels. The main differences are the spelling of some of the parameters when calling the connection. Which you choose to use is largely up to you and should not have any effect on the outcome. We often use mysql.connector in Jupyter notebooks and pymysql in Lambda functions, but you can pick one and stick with it if you would like, just make sure that whichever you use in your Lambda functions is installed in the corresponding environment and named in your requirements text file.

* Connect to your database in Python

## Creating the Tables for the Database

Once we have our database created and we are able to connect to it in Python, we can add tables. This first requires your team determining what the schema (tables and connections among tables) of your database will be. A major consideration here is not having too many columns per table, especially if many will contain text, because MySQL may throw an error telling you it is too much. For the purposes of our project, we broke all of our data into five tables. Each table uses the same primary key (a unique identifier for each row), which in our circumstance is the participant id (the column in our tables in “id\_participant”). Since all of our tables use the same primary key, they in theory could be just one table but the size limitations of MySQL prevented us from doing this, so we found a way to break them up that made since in the scope of the project. Specifically, we have an Identifiers table that has columns related to the site, year, group, etc. of participant. The Participant\_Measures, Survey\_Measures, and Performance\_Measures tables contain all of the data from the surveys filled out by participants in the study and the surveys are naturally broken into these three categories, which are why we use them as our tables since we needed to split the data up somehow. Our last table is Participant\_Tracker, which simply has a binary indicator for each survey that identifies whether or not each participant has completed that survey. You will want to work with your team and those who understood the data and project well to determine what tables you want and how to break up the data if it is too large. For example, our tables range from 12 columns to a little over 500 columns, with a total of about 1300 columns between all five tables.

Once you have decided on the tables you want, the columns in each table, and the variable type of each column (numeric or text), you are ready to write SQL queries to create the tables. We do this in a Jupyter notebook using Python; however, you could also write and execute these queries in MySQL Workbench. The reason we use Jupyter notebook/Python is because we have so many variables (columns) so we can use loops to write the queries instead of manually typing them in or copy/pasting them. The code for how we do this is available on the GitHub and is a Jupyter notebook titled *Create\_Database.ipynb* under the RDS folder.

The first thing we do in this notebook is connect to the database as just described above. Next, we read in an excel file to get the column names. We have all of our column names broken out in the data dictionary that is used for our specific project. If you have some file that has all of your column names, you may want to consider finding a similar way to read them into Python so that you do not have to manually type them out or copy/paste them. While the actual specifications in the code for how we get our column names is going to be extremely specific to how we have them stored in our excel sheet, the approach can provide guidance.

The way that we are going to generate our queries to create the tables requires a list of all the column names for that table, which we just described how to get above. However, it also requires a list of all the text (also called string or non-numeric) columns, which is a subset of the list of all columns. How you get this list for your tables is also entirely dependent on how you have your data for the columns. For example, the data dictionary for our project has a column with all of the variables (columns for that table) as well as other columns that provide metadata for these variables, such as one called Values that describes the type of data that will be stored in that variable/column. This is what we use to get the list of text columns for some of our tables by filtering on the specific value types. However, for some it is just easier to manually create the list in Python of what columns will be string/text. Under the Create and Execute Queries portion of the notebook, we perform these steps where the tableName\_column\_list is the list of all columns for that table and the tableName\_char is a subset of that list containing only those columns that will hold text or non-numeric data.

The functions that we have in this notebook are used to generate the queries of the form:

‘CREATE TABLE tableName (column1 varchar(100), column2 float, column3 varchar(100), PRIMARY KEY (column1) );’

The reason for the two lists above is that all of the columns in your text list will get varchar(100) in the database schema, denoting that the column can have strings. The remaining column will get float meaning that the data type will be numeric. You also need to specify your primary key, which we already mentioned is ‘id\_participant’ for all of our tables. Additionally, we have a second function to create a table that includes extremely large text columns, the reason we don’t make all text columns large by default is because then MySQL will reduce how many columns you can have in the table to even it out size wise, so you should understand your data well enough to know how to determine what will be stored in the columns. We also have a function used to show the table that is great for testing after creating the table by executing the query generated by our function.

As previously mentioned, you can write these queries manually in the form described above and execute them in MySQL Workbench or Python without our code to generate them. However, if you have a large number of columns, it will be much easier to automate the process at least partially by utilizing this code.

**Important Note:** When you are finished altering a database in Python/Jupyter notebook, make sure to close the database connection by executing the following:

cnx.commit()

cnx.close()

These lines can be seen at the bottom of each of our notebooks because otherwise any changes you made to the data/database will not take effect.

We have now created our tables and fully set up our database. Thus, the next step is to begin implementing and integrating the pipeline.

* Generate the tables in your database (schema)

## Viewing the Data

During development and testing, you will likely want to view the contents of the tables in your database. Often, you may also find it helpful to delete all of the data, so that you can reupload files to S3 and test to make sure the data repopulates. For these purposes, we create a Jupyter notebook called *View\_Tables.ipynb* under the RDS folder. This includes the functions showTable() and clearData() for these exact purposes. We hope these help you test to make sure your pipeline is functioning properly as you go through the implementation.

It is important to remember that if multiple team members are working on implementing and testing the pipeline, they may delete data from the database in the process. Thus, don’t panic if you see that the tables are suddenly empty as this could have been a team member’s doing and you may just need to remember to repopulate the data either manually or by uploading the CSV files to S3 if that portion of the pipeline is functional.

# Lambda Functions

Now that we have our S3 buckets to hold our CSVs and our database set up to host our data in a relational format, it is time to get the data from the CSVs to the database. This is the job for the Lambda function. As we mentioned in the background on AWS, a Lambda function is a serverless snippet of code that executes in response to an event, such as a CSV file being uploaded to an S3 bucket. They allow us to cheaply trigger code execution so that we can keep our database up to date in real time. For a refresher on where we are in the data pipeline, see the diagram below.

A model of a space ship

Description automatically generated with low confidence

## Amazon Chalice

We utilized AWS Chalice to write and deploy our Lambda functions; however, we could have also built them solely using the AWS Management Console. The benefit of Chalice is that the back-end integration with S3 and API Gateway is wrapped in a decorator that simplifies the construction of Lambda functions. It can be thought of as Infrastructure as Code (IaC), where a developer specifies the resources that need to be deployed in a programmatic fashion. Chalice also allows the python code to be written locally and deployed from the command line. Whether the Lambda functions are built through Chalice or the Management Console, they function the same.

Chalice can be intimidating at first, so we will walk you through exactly how to get it set up and going because the benefits far outweigh any resistance you may initially have. The first step is to create an environment and install the necessary packages. You can do this by opening up a new terminal window and issuing the following commands where “your\_name” is whatever you choose as your environment name:

conda create -name your\_name

conda activate your\_name

conda install pandas

conda install pymysql

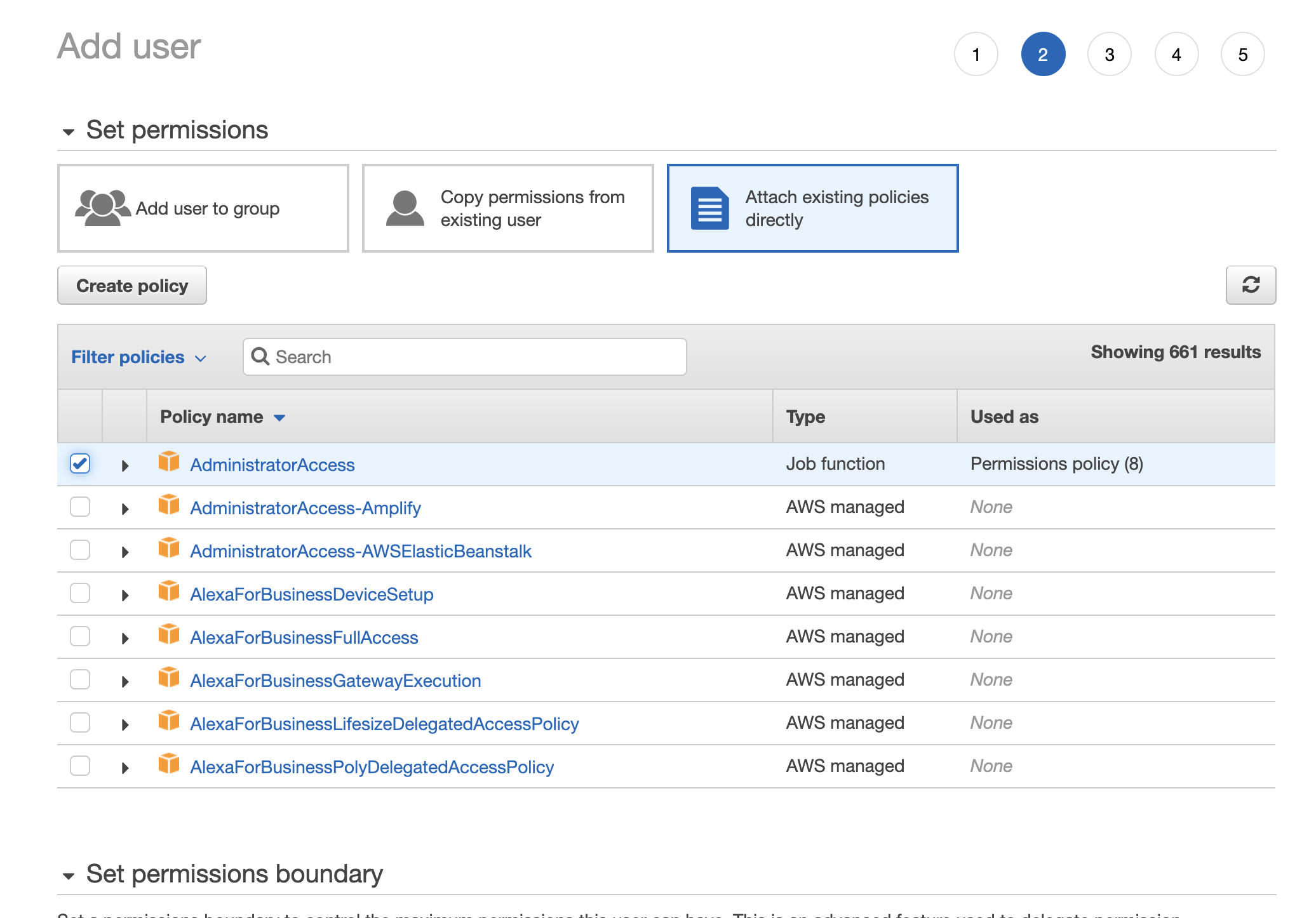
conda install boto3

pip install chalice

After you create your environment once, you will only need to use the following command when using Chalice in the future:

conda activate your\_name

In order for Chalice to work, you will need to create a configuration file with your AWS credentials. AWS allows you to generate credentials for users with specific permissions. It is best practice to create users with specific privileges, such as “Admin” etc. This [tutorial](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html) covers how to create a new AWS user. Follow the steps in the tutorial for the console creation of a user, and give the user the ‘Admin’ privileges as in the screenshot below:



After running through this configuration, you will be able to get the access key and secret key for this user – write these down as this is the only time you will be able to view the secret key.

You can enter your AWS credentials (only need to do this once) with the following commands:

mkdir ~/.aws/config

To view contents of the file: cat ~/.aws/config

To overwrite contents of the file: cat > ~/.aws/config (Use control+d on mac to exit)

To append content to the file: cat >> ~/.aws/config (Use control+d on mac to exit)

Alternatively, you can download the [AWS CLI](https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html) and run ‘aws configure’ in your terminal to have the software automatically configure your computer with the generated user credentials.

Now we are all setup to create a Lambda function using Chalice. First, make sure you are in the environment you created by using the activate command mentioned above. Second, make sure to change your directory to where you want your Lambda function/Chalice code to live (ex: cd Desktop). We can create the folder that will host all of the code for the function with the following command where \*enter-project-name\* is whatever you want the name of your Lambda function to be:

chalice new-project \*enter-project-name\*

This will create a folder titled with your project name specified above. The contents of which include a folder ‘\_pycache\_’ that you do not need to touch, an app.py file, and a requirements.txt file. You can edit these from your favorite code editor, such as VSCode or Spyder. The app.py file is where you will write the actual Lambda function. The requirements.txt file is where you specify what packages your function uses (which you will import in app.py). For example, your requirements.txt may simply be:

pandas

numpy

pymysql

It is important to note that you do not need to include chalice or boto3 in the requirements despite importing them in app.py. Additionally, we recommend that any functions you write to utilize in your Lambda function be stored outside of app.py. We often create another folder under the project called “chalicelib” with a python file such as “rdsFunctions.py” where you can store all the functions you wrote. Then, in your app.py file you can simply import them such as by saying “import chalicelib.rdsFunctions as rd” at the beginning of your app.py file. Also note that you do not need to specify this file in requirements because it is not a package.

For a Lambda function triggered by an upload to an S3 bucket, your Chalice app.py file should begin with a version of the following:

from chalice import Chalice

import numpy as np

import pandas as pd

import pymysql

import chalicelib.yourfunctions

import boto3

app = Chalice(app\_name=’the-name-of-this-project/folder-here’)

app.debug = True

@app.on\_s3\_event(bucket=’your-s3-bucket-name’, events=[‘s3:ObjectCreated:Put’])

def handler(event):

s3 = boto3.client(‘s3’)

obj = s3.get\_object(Bucket=event.bucket, Key=event.key)

data = pd.read\_csv(obj[‘Body’])

# your code here

return {‘hello’:’world’}

This is the code needed to have the Lambda be triggered by an S3 upload, as well as to get the data from the file that was uploaded and store it in a pandas dataframe. The rest of your code will continue in the handler function and will likely return nothing. However, you may end up uploading a new CSV to a different S3 bucket or connecting to and altering the database (don’t forget to commit and close the connection in the function).

When you are finished writing your Lambda function, make sure all files are saved. Then, from the terminal, make sure you are in your environment (using the conda activate line from above) and that your current directory is the project folder (ex: cd Desktop, then cd \*enter-project-name\*). All you have to do now is issue the following command:

chalice deploy

After a few moments, your Lambda function will be deployed on AWS, where you can see it by clicking Lambda from the Services dropdown under the Compute category. This should pull up a list of functions as seen below and yours should be included. Note that the function may be called \*enter-project-name\*-dev-handler, which is totally fine and expected.

Graphical user interface, application, Teams

Description automatically generated

Now you should be all setup with Amazon Chalice so that you can write and deploy your Lambda functions from your local machine. We hope that this instruction helped ease some of the anxiety of using Chalice and you can benefit from all the advantages it has over building Lambda functions in the Management Portal, such as having to manually upload packages in a zip file and add them to functions as layers. We will now discuss the two Lambda functions in our pipeline and how to implement them.

* Set up Amazon Chalice on your local machine

## Data Cleaning

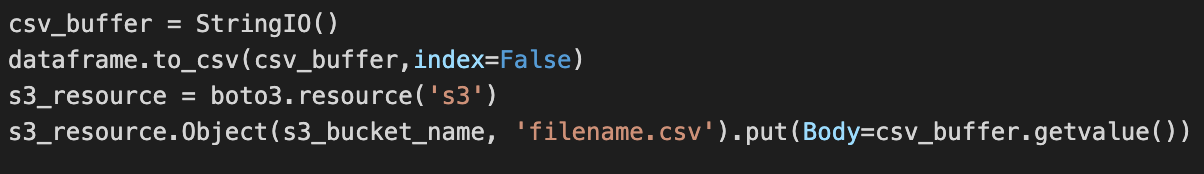
With Chalice set up, you are ready to write and deploy your Lambda functions! As a reminder, the first Lambda function takes raw data and turns it into clean data, while the second takes the clean data and uploads it to the database. We will start sequentially through the pipeline by beginning with the Lambda function for data cleaning. However, you may skip to the next section if you will only be uploading cleaned data.

The code for this function is going to be largely project specific. For the research project for which we implemented this pipeline (TeachSIM), we took the code that the research team used to clean and merge survey files in Stata and translated it into Python. We first replicated their cleaning efforts in Python at which point we created various functions to consolidate and automate much of the repetitive cleaning that they were doing across similar files. We then turned this into a Lambda function to automate the entire process. While some of the functions for cleaning the data may be applicable to and/or reusable for other projects, the specific file merger and other cleaning techniques are substantially specific to this project. Nevertheless, the structure of the Lambda function is certainly a useful tool for you to use as a baseline and to simply add in your own data cleaning code.

We provide some of the generic data cleaning functions that we have created in the GitHub repository under Data Cleaning/Data\_Cleaning.py. This Python script with the functions also includes descriptions of what each function does. For a more exhaustive list of the functions we created and used for the TeachSIM project, see Data Cleaning/Stata Replication/utils.py. The Data\_Cleaning.py contains a subset of the functions in utils.py that we believe are the most likely to be useful and transferrable to other projects and thus the most interesting to those reading this documentation.

It is also important that we discuss the input and output of the data cleaning code. For our data cleaning, we take in many input files and output a single cleaned file because we merge the data among other things. However, this limits the generalizability of the Lambda function. It may in fact be more useful for you to have your Lambda function take a single file, clean it, and output the cleaned version of that file. This would require either a cleaning process that works on all files or specific cleaning processes in the Lambda that are conditional on the name of the file being uploaded. One of the biggest drawbacks of the way we have our data cleaning with requiring merging is that we need to have all the files used in the mergers otherwise our code will fail. Additionally, we cannot just pass it a new file that has not been seen before because it will not clean it properly. We could include conditionals or break this out into more than one Lambda, but for our purposes it worked fine the way we did it, especially because this was a pilot pipeline based on an already completed project and we were trying to mimic the exact data cleaning methods the research team used.

The main features of your data cleaning function are that it takes the input CSV file from the S3 bucket and that it outputs a cleaned version of this file in a second S3 bucket. We have already covered how to get the data from a CSV that is uploaded to an S3 bucket when we introduced Chalice. However, we also want to cover here how to write data to an S3 bucket in the form of a CSV file from a Lambda function. Our Lambda functions utilize the boto3 library, the AWS SDK for python which allows us to interact with AWS resources in python code. In our lambda functions, we often reuse the following template of code for writing data to an s3 bucket:



The code above converts a pandas dataframe to a csv, writing it to a csv buffer. We utilize the boto3 library to create a connection to the s3 service. We then use the Object method to specify the name of our bucket and the name of the csv file we want to write. We finally use the put method to specify the contents of the body of the csv.

## Database Update

With the cleaned data files arriving in the proper S3 bucket, we want to make sure that data gets transferred to the database. Note that if you have cleaned data, you can upload it directly to this bucket to skip the data cleaning. This is often useful in development and testing when you want to make sure this part of the pipeline works without needing to test your data cleaning. This may also be useful for existing projects where the data has already been cleaned locally.

The main benefit of the code that we provide here is that it is almost entirely independent of your database schema or project specifications. This means that you should be able to reuse it with minimal modifications. The basic concept of the code is simple, you pass to the function three things: your data in a pandas dataframe, the table in the database you want to update, and a cursor that connects to the database. The output is a list of queries that can then be executed. This code was developed and tested in the Jupyter notebook titled *Input\_Data.ipynb* under the RDS folder of the GitHub. The actual Lambda function code is contained in the update-database folder.

In the Lambda function app.py file, we perform just a few simple tasks. We first get the cleaned data from the S3 bucket and connect to the database as have previously been described. We then create a list of all the tables in the database that we wish to update and use a for loop to pass these to the generateQueries function along with the data and a cursor. Subsequently, we use a nested for loop to execute each of the queries in the list returned by the function before the loop turns to generate the queries for the next table. Note that it is possible to alter the code fairly easily such that the function accepts a list of tables rather than a single table and/or that the function executes the queries rather than returning them. These would simply move the for loops from the app.py file to the function itself.

As mentioned, little modification is needed, but there is an important yet easy to execute change that must be made. In each of the functions, ‘id\_participant’ must be replaced with your specific primary key. Also, in the way we have the code set up, all of the functions must have the same primary key (namely id\_participant), but the code could be fairly easily modified to accept the primary key as a parameter to the function, which can then pass it back to all places where it is used to make the code more dynamic. While we did not do this, it may benefit you if you are dealing with tables that have different primary keys.

Appendix B provides a much more detailed description of the code used in this Lambda function and how it works. However, the information here should be enough for you to deploy this Lambda function for your own project. With this deployed, keeping your database updated is as simple as uploading a CSV file to an S3 bucket (in other words, extremely simple), because the Lambda function will handle the rest for you. Thus, when you query the database from the MySQL Workbench or from a connection in Python, you will see the updated data. This is a great way to test functionality in real time to make sure this piece of the pipeline is working.

If you are following along for the entire pipeline, you can check off the following item on your to-do list.

* Write and deploy database update Lambda function

Now that we have finished writing and deploying our Lambda functions, we are a significant portion of the way through our data pipeline. We have set up our S3 buckets, created our database and set the schema, and wrote Lambda functions to move the data through the pipeline. This is a huge achievement and for some may be the end of the pipeline if your end goal was to have a database in MySQL. However, the remainder of the pipeline focuses on making the database easy to query and extract data from, especially for those with little to no knowledge of SQL. See the diagram below that illustrates what we have covered so far and what should be fully functional if you have been implementing this pipeline for your project as you follow along.

Graphical user interface

Description automatically generated

# API Interface

The documentation so far has focused on implementing a data pipeline that takes raw data and writes it into the database; however, to follow the instructions in this section, all you need to have functioning is a MySQL database. The database is able to be queried using SQL with the methods we have previously mentioned. However, not everyone who needs access to the data is likely to have sufficient SQL knowledge. Even for those who do, it would certainly be easier to simply be able to click a few buttons and receive a CSV of the data we want. Thus, this portion of the pipeline focuses on creating a web-facing query form that allows users to simply check boxes pertaining to filtering the data and then hit submit to download the queried data in a CSV directly to their local machine. The diagram showing where we are in the pipeline can be seen below. This piece of the pipeline has two main components, the web-facing interface built using React to obtain the users input and the code that takes this input and queries the database to return the data to web-facing component to send out as a download. We will cover both of these sections here, starting with the behind-the-scenes code.

A model of a space ship

Description automatically generated with low confidence

## Query Generator

The backbone of the API takes input parameters to query the database accordingly and return the data as output. The web-facing component simply acts as a way for users to provide these parameters and receive the output. Thus, we will start from the inside out by covering what parameters to take and how to turn this into data. We have quite a few functions used to do this. They are implemented and tested in the *Automate\_Queries.ipynb* file in the GitHub.

Let us start with the overall wrapper function request(). The request() function takes as input the following:

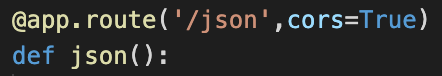
* cursor: to provide connection to the database
* measure\_list: a list of the tables from which to query
* time\_list: a list of study years that we want to filter on
* specific\_measure\_list: a list of survey names that map to groups of variables within the tables
* field\_type: a list with either ‘Numeric’, ‘Text’, or both to filter the columns by data type

This means that on the query page, users will be able to select the years for which they want to filter the data, what columns/variables they want to receive as designated by measure\_list and/or specific\_measure\_list, and whether they want numeric data, text data, or both. We use all of this information to produce and execute one large SQL query that returns the relevant data. Please see Appendix C for a detailed description of the code to generate this query.

* Set up the code to generate queries

## API Gateway

The API Gateway serves as a ‘middle-man’ between our database and researchers, allowing users to query the database via API endpoints. Our API Gateway was likewise written in Chalice and the code can be found [here](https://github.com/taylorrohrich/MSDS_SERA_capstone/tree/main/aws/sera-api). However, we will note the basics of creating an API in Chalice here. When creating the API you write functions for each ‘pathway’ you want users to be able to query your database through in the generated App.py file. The only addition is this ‘decorator’ on top of your function:



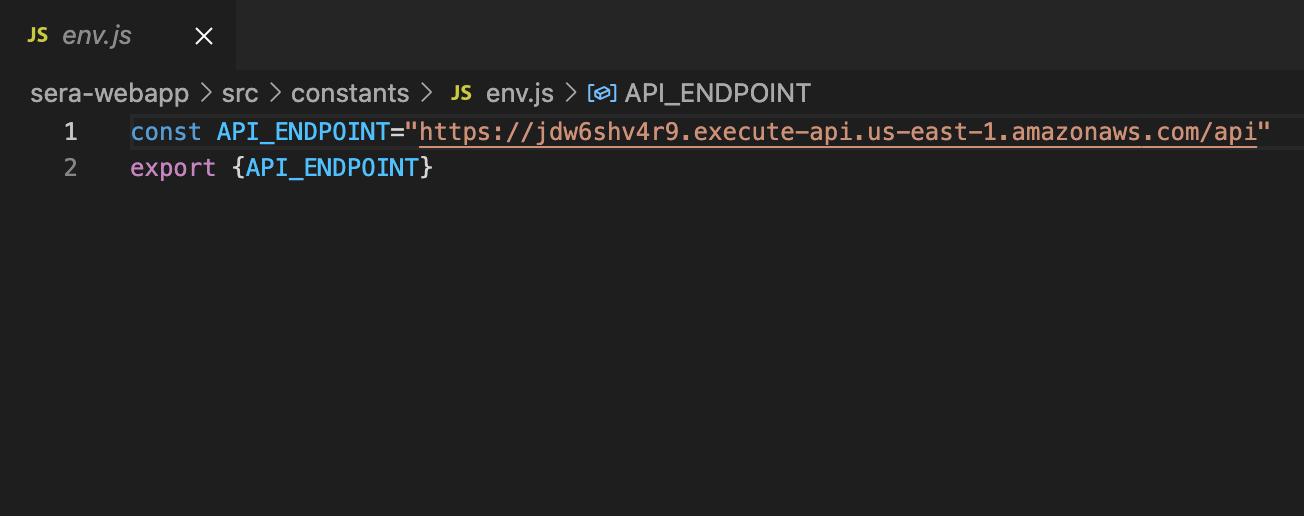
This @app.route method specified the api endpoint for this function to run. The second parameter, ‘cors=True’ simply allows the API to be query-able from another webpage. For example, once this application is deployed using the chalice deploy command, I can enter the url of “<api-gateway-url>/json” into my browser and the code defined in the json function will run.

## Web-Facing Query & Tracker Page

The Web Application that displays the query page as well as the participant tracker was written in React.js, and JavaScript framework. In order to run this code you need to have the following installed:

1. [Node.js](https://nodejs.org/en/download/)
2. [React.js](https://github.com/facebook/create-react-app)

With the above libraries installed, clone the [Git Repository](https://github.com/taylorrohrich/MSDS_SERA_capstone) and navigate to the following [folder](https://github.com/taylorrohrich/MSDS_SERA_capstone/tree/main/aws/sera-webapp) in your local directory. In the [constants folder](https://github.com/taylorrohrich/MSDS_SERA_capstone/tree/main/aws/sera-webapp/src/constants), create a new file called ‘env.js’ that includes the endpoint to your API Gateway. An example of this is shown below:



After this file is created as defined above, we are now ready to deploy the app, in the root folder of this web application (sera-webapp), you can run the following commands in your terminal:



The command above will start the web app locally; you will be redirected to a browser window with the web application running.



The command above will build a production bundle of the app. The result of running this command will be a ‘build’ folder within the ‘sera-webapp’ folder. This folder can then be deployed on [S3 static web server hosting](https://docs.aws.amazon.com/AmazonS3/latest/userguide/WebsiteHosting.html).

# Participant Tracker

## Updating the Table

A bonus portion of our data pipeline is the Participant Tracker. This is a table in the database called “Participant\_Tracker”, which consists of the names of the surveys as columns, the participants as rows, and the values as indicating whether or not the participant has completed that particular survey. Previously, the researchers were manually keeping track of which participants completed which surveys, so this serves as a way to automate that process.

There are two ways we considered implementing this. The first, and the way we chose, was to implement this in tandem with files being uploaded by using a mapping of CSV file names to columns in the tracker table. The second, which is possible but not how we chose to do it, was to use specific variables as a proxy for a survey and if that variable for a particular participant is not null, then the participant is assumed to have completed the survey. The potential issue with this is the reverse assumption that if the value is null then the participant is assumed to have not completed the survey, which may not be the case.

The code that we used to develop and test the updating of the participant tracker is housed in the Jupyter notebook called *Participant\_Tracker.ipynb* on the GitHub under the RDS folder. This code is also incorporated into our data cleaning Lambda function. The essential idea is that when a file is uploaded to the data cleaning S3 bucket, the CSV file name is checked against a predefined mapping of file names to columns in Participant\_Tracker. If the file has a corresponding column, then for every participant in the file being uploaded, the value corresponding to their row and the relevant column in Participant\_Tracker is updated to the value of “Completed” from “-“ if they had not previously been identified as having completed the survey. While we chose to perform this update in the data cleaning Lambda function (and thus for only those files uploaded to the data cleaning S3 bucket), you may choose to instead perform this task in the Lambda function that writes the cleaned data to the database. Thus, your mapping of CSV names would need to be consistent with your naming conventions when writing files to the cleaned data S3 bucket.

The code used to perform this is relatively simple. We define a function called getTrackerQueries() that requires the dictionary of CSV name mappings, the current CSV name, the current data, and a cursor. We first search the mapping to get the column name in Participant\_Tracker that corresponds to the CSV name. As we have done in previous steps, we also extract all of the participant IDs currently in the table as well as all of the columns in the table. If the participant already exists in Participant\_Tracker, we create a query of the form: “UPDATE Participant\_Tracker SET columnName = ‘Completed’ WHERE Participant\_ID = ‘example\_id’;”. On the other hand, if the participant is not yet in the tracker, we use the form: “REPLACE INTO Participant\_Tracker SET (Participant\_ID, desiredColumn, otherColumn1, otherColumn2, otherColumn3) VALUES (‘example\_id’, ‘Completed’, ‘-‘, ‘-‘, ‘-‘);”, where we use a loop to include all of the other columns as “-“ so that they do not appear as None in the tracker. The function returns a list of all these queries, which we then execute by looping through the list.

Now that you understand the nature of the Participant Tracker, you can create the table for your project, define the mapping of file names, and choose to include the update in your existing Lambda function.

* Create the Participant Tracker table
* Add the update of the tracker to an existing Lambda function

# Transcript Data

As we mentioned in the beginning of the documentation, the study for which we implemented this pipeline included transcript data in addition to the survey data that has been the focus thus far. Another team has been focusing on the development of code to analyze the raw transcript files and produce a tabular output summarizing the analysis. One of our goals was to incorporate this into the pipeline such that raw transcript files could be uploaded to an S3 bucket, which would trigger a Lambda that performs the NLP analysis and outputs the results to another S3 bucket, which then triggers a second Lambda to write these to the database. This should sound familiar as it is essentially the same design as what we have implemented for the survey data. We can see our design for the two parallel aspects of the pipeline below. However, we ran into issues related to running the NLP analysis in the cloud. Nevertheless, we were able to create the second S3 bucket and Lambda function for that portion of the pipeline. This means that a CSV with the output of the NLP analysis can be uploaded to an S3 bucket, which triggers a Lambda function that then stores this data in an additional table in the database. However, the primary key for this table is the file name of the transcript rather than the participant ID.

This table is called Fidelity\_Measures. We create it in the same Jupyter notebook (*Create\_Database.ipynb*) that we covered when creating the other tables. Additionally, this Lambda function (called nlp-database-update) is an exact replica of the update database Lambda used for the rest of the tables, but with the primary key of filename rather than id\_participant. Still, we demonstrate this code in a Jupyter notebook called *NLP.ipynb* under the NLP folder of the GitHub repository, specifically for this table. You may have a desire to implement something similar or have parallel aspects of your pipeline that you want to have separate S3 buckets and Lambda functions. We use this to demonstrate that it is possible to do so. However, when possible, you may want to consolidate to the same S3 buckets and Lambda functions just to keep things simple and manageable. Not to mention that if the Lambdas perform virtually the same task, then when you change the code in one, you will likely want to change it in another, which is redundant and leaves room for errors.

A close-up of a map

Description automatically generated with low confidence

# Appendix A

A checklist of tasks to implement the data pipeline are below. The checklist is organized in the order that the tasks are intended to be completed, although this is not the only order that will lead to correct functionality.

###### S3 Buckets

Create an S3 bucket for raw data

Create an S3 bucket for cleaned data

###### Relational Database

Create the database instance in Amazon RDS

Create the schema instance in MySQL Workbench

Connect to your database in Python

Generate the tables in your database (schema)

###### Lambda Functions

Set up Amazon Chalice on your local machine

Write and deploy data cleaning Lambda function

Write and deploy database update Lambda function

###### API Interface

Set up the code to generate queries

Instantiate the API Gateway

Create the query page interface

###### Participant Tracker

Create the Participant Tracker table

Add the update of the tracker to an existing Lambda function

# Appendix B

This appendix provides a more in-depth explanation of the code used in the Update Database Lambda function. The Jupyter notebook containing this code is called *Input\_Data.ipynb* under the RDS folder of the GitHub and the Lambda function with the same functions (but without the testing as this was the code used for deployment) is contained in the update-database folder.

As described in the main document, the Lambda function interacts with only one function, namely generateQueries(), which in turn calls several other backend functions that we have written. The only parameters needed are the data in a pandas dataframe, the table name, and a cursor connected to the database. The function returns a list of SQL queries.

The first step of this function is to determine what columns in the data being passed to it exist in the table being passed to it. To discover this, we use a function that we call overlapColumns() that takes two lists of columns and returns a list of those that are in both. Thus, we pass this function the columns from the dataframe and the columns from the table. In order to get the columns from the table, we use our getColumnNames() function that requires the table name and the cursor to generate and execute a SQL query that uses the database schema to identify and return all of the column names from that table.

Second, we need to figure out which participants already exist in the table as this will determine whether we update the data in their row or create a new row for them. To generate a list of all the participants in this table, we use our getExistingIds() function that uses a simple SQL query to return all of the values in the id\_participant column of the table. You would need to replace id\_participant with your relevant primary key.

The third step is a precautionary data cleaning measure that can help prevent function failure. The way we generate our SQL queries requires that no text data in the dataframe contains either a single or double quotation mark as this will result in an incorrect query being returned that will not execute. Thus, we use the fixStrings() function to parse through all of the columns of data that are strings (text) and simply remove and quotes. Note that this may lead to some minor data quality issues depending on how important it is for you to have these quotes in your data. However, we were unable to see a way around this, but that does not mean was does not exist.

Lastly, we use the cleaned data, the overlapping columns, and the existing IDs to generate the SQL queries in our list\_queries() function. This function uses id\_participant several times so make sure that you replace each instance with your primary key. We start with an empty list of queries and iterate through each row of the data whereby we will generate a query and append it to the list. There are two cases of queries that can be created here. Those for existing IDs that will update a row in the table and those for new IDs that will add a row to the table.

Let us start with the case for an exitsing ID, which we check by seeing if the ID is in the list of existing IDs that we passed to the function. If it is, then we generate a SQL query of the form:

“UPDATE tableName SET column1 = value1, column 2 = value2, column 3 = value3 WHERE id\_participant = ‘example\_id’;”

We generate this by iterating through the columns and adding those that have non-null values so that we do not overwrite any existing data for this participant. This is extremely important because it allows us to minimize data loss. Additionally, we only add the query generated to the list of all queries if at least one column other than id\_participant is going to be updated as it would be trivial otherwise.

The second case is that the participant does not exist in this table, so we need to create a row for them using a SQL query of the form:

“REPLACE INTO tableName (column 1, column2, column3) VALUES (value1, value2, value3);”

We generate the columns and values in the same way, but we alter the construction of the query to adhere to the syntax required. Additionally, we only keep the query if at least one other column besides id\_participant has a non-null value. Lastly, an important addition in the case from the first is that since we are adding a row to the table for this participant, we add their ID to the existing\_ids list so that if there happen to be two rows in the data for the same new participant, we do not overwrite the first row but rather keep all of the data from both. However, if there is any contradictory data, the data of the last row for that participant will be used.

The list\_queries() function returns the list with all of the queries generate above, which is ultimately then the output that is used for the generateQueries function, but it is more digestible to have them broken out to allow for an easier front-facing function.

This concludes the coverage of the code used to update the database.

# Appendix C

This appendix covers the code used by the API to query the database according to the user input. A Jupyter notebook with this code and test cases exists on the GitHub under the API Gateway folder and is called *Automate\_Queries.ipynb*.

As previously mentioned, the input parameters to the wrapper request() function are as follows:

* cursor: to provide connection to the database
* measure\_list: a list of the tables from which to query
* time\_list: a list of study years that we want to filter on
* specific\_measure\_list: a list of survey names that map to groups of variables within the tables
* field\_type: a list with either ‘Numeric’, ‘Text’, or both to filter the columns by data type

We use these to generate the larger query that will return the relevant data. This is broken into two distinct parts. First is a subquery that joins all of the tables together based on their shared primary key, id\_participant, and filters based on the year as provided in time\_list. If you are adding filters for your own project (such that you would use a WHERE clause in SQL), this is where you would add them, and you can simply mimic how we do this for the time\_list as we will now describe. Second is a selection from the subquery based on the relevant columns desired.

Let us first examine the subquery. We generate it with the subquery() function that only requires as input the time\_list. If you are adding more filters, you would need to pass additional parameters to this function. The first part of the subquery is fairly straightforward. We start with the identifiers table and perform left joins with the other tables using id\_participant to combine rows. Note that the Participant\_Tracker table has a different name for the id\_participant column, which requires slightly different syntax, but the concept remains the same. It is also important to note that MySQL does not permit full joins, meaning that all rows from both tables cannot be kept in the case that the IDs are not in both. To avoid this causing an issue of data loss, we must start with the Identifiers table, making the assumption that this will have every participant ID that exists in the other tables, and we use left joins because it will keep every single data participant from Identifiers whether or not they exist in the joining table. By its nature, the left join will drop any participant IDs that exist in the joining table (the right table) but not the Identifiers table (the left table). This means that the ordering of this section of the subquery matters. A right join would do the opposite and an inner join would keep only those participants that exist in both tables.

Now that we have joined all the tables, we want to filter based on id\_year (or whichever column you want to do this with) by adding a WHERE clause. Since the web-facing query page will produce a list of time periods in the format such as “Fall 2017” or “Spring 2019”, we use a background function called convert\_semester() to translate these values into the actual values in the database. For example: convert\_semester([“Fall 2018”, “Spring 2019”]) would return [1718, 1819]. We can then use these values to filter the data in our WHERE column by adding the following to our query: “WHERE id\_year in (1718, 1819)”. This can be done in a similar way for other columns. The import thing to remember is that you will need to map the values given from the web-facing page to the corresponding values in the data, because it is easier and cleaner to look at “Fall 2018” than “1819.”

Now that our subquery is complete, we need to specify which columns to select from its output. To do this, we use the get\_variables() function that requires a cursor, the measure and specific measure lists, and the field type list. These are different than the filtering based on time because we are simply looking at a subset of the columns rather than filtering the rows based on values. The first piece of this function returns the columns according to the measure and specific measure lists and the second component filters these based on their data type.

The first portion requires a deeper understanding of how measure\_list and specific\_measure\_list relate. Measure\_list is a subset of the tables in the database, while specific\_measure\_list is a list of groups of variables that are further subsets of these tables. For example, “Survey\_Measures” is a table in the database with subsets such as “big5”, which pertains to all columns whose names begin with “big5\_” like “big5\_base\_survey”, or “exit\_post\_survey”, which references all columns in this table that start with “exit\_”. We have a custom function that we built to create the mapping of these subsets of our tables called generate\_mapping(), which will need to be altered for your specific project. However, if this type of sub setting is not applicable to your project, simply skip this. We will now describe how we get the desired columns. If the table in measure\_list does not have any subsets as defined in the mapping (our Participant\_Measures and Participant\_Tracker tables do not), then this is simple. If the table is in measure\_list, we include all the columns from this table by using the table\_variables() function that obtains the list of columns from the table schema. However, if the table does have subsets, this is a bit more complicated, and we will go through the cases now. First, if the table is included in measure\_list and none of its subsets are included in specific\_measure\_list, we include all the columns from the table. However, if one or more of the subsets are included in specific\_measure\_list then whether or not the table is included in measure\_list, we include only those columns specified by the subsets, which we get from the mapping. If neither are selected, obviously none of those columns are included. We made the choice here that if a subset is selected (in specific\_measure\_list), but the overall table is not selected (in measure\_list), we treat it the same as if the table is selected by only including the columns in the relevant subsets. This may wish to be changed such that the subset is not included if the main table is not selected, which is possible to adapt the code to handle. All of these columns are combined into one overall list and we ensure that there are no duplicates and that id\_participant is included.

Secondly, we filter this list of columns based on data type. If both “Text” and “Numeric” are included in field\_type (checked on the query form) or if neither are included, then no filtering is done, and we simply return the columns produced by the first portion. However, if only one is included, we filter for only columns with those data types (still including id\_participant if numeric data is desired). We use our background functions text() and numeric() to generate lists of all the columns across the relevant tables that have the desired data type. We then filter by returning the columns that exist in both the list of columns from the first section of this get\_variables() function and this data type specific list. We return a list of the relevant columns as output.

Now that we have the subquery that merges all of the data and filters based on time (or other filtering as needed) and a list of desired columns, we combine these two in our wrapper request() function to return the data pertaining to the relevant columns from the subquery. There are certainly other ways to piece these together, but we found this to be the easiest way for us to do this in a manner that allows us to break the problem down into smaller parts for interpretability. Much of this should be reusable by changing table and primary key names where needed.