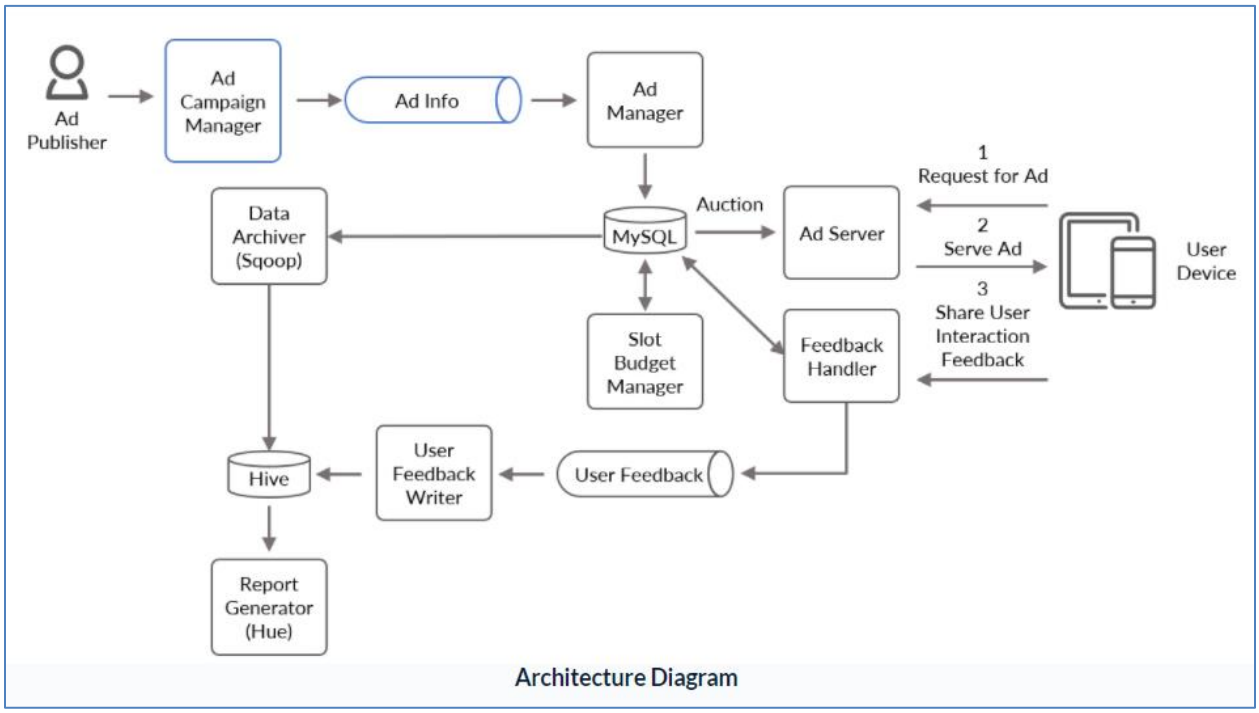
CODE LOGIC: ONLINE ADVERTISING

For this project, we are handling totally 7 number of elements in order to make it more effective and efficient



Architecture Diagram

The Following components will be made:

1. Ad Manager
2. Ad Server
3. Feedback Handler
4. Slot Budget Manager
5. User Feedback Writer
6. Data Archiver
7. Report Generator

As a part of this Mid-week submission, we would be made the first three viz., Ad Manager, Ad Server, Slot Budget Manager based on the instruction provided.

As a part of this project, we have to make three MySQL tables. But before we may need to import and have our MySQL environment ready by installing it in our EC2 instance. Also, the MySQL connectors, PyKafka, Pyspark, Flask Packages are needed.

CAPSTONE PROJECT – ONLINE ADVERTISING

MySQL Queries:

Users table:

```
Create table capstone.users(
ID varchar(100),Age int,Gender char(1),Internet_usage varchar(20),Income_bucket
char,User_agent_string varchar(2000),Device_type varchar(50),Websites varchar(1000),Movies
varchar(1000),Music varchar(1000),Programs varchar(1000),Books varchar(1000),Negatives
varchar(1000),Positives varchar(1000));
```

We have used the above query to create Users table

Ads table:

```
Create table capstone.ads(`text` text,Category varchar(300),Keywords text,Campaign_id
varchar(300),Status varchar(20),Target_gender varchar(20),Target_age_start
int,Target_age_end int,Target_city varchar(100),Target_state varchar(100),
Target_country varchar(100),Target_income_bucket varchar(100),Target_device varchar(100),
Cpc double,Cpa double,Cpm double,  Budget double,Current_slot_budget
double,Date_range_start varchar(100),Date_range_end varchar(100),
Time_range_start varchar(100),Time_range_end varchar(100));
```

We have used the above query to create Ads table

Served Ads table:

```
Create table capstone.served_ads(Request_id varchar(100),Campaign_id varchar(100),
User_id varchar(100),Auction_cpm double,Auction_cpc double,Auction_cpa double,
Target_age_range varchar(200),Target_location varchar(200),Target_gender varchar(10),
Target_income_bucket varchar(50),Target_device_type varchar(50),
Campaign_start_time datetime,Campaign_end_time datetime,Time_stamp timestamp);
```

We have used the above query to create Served Ads table

Load Query for Users table:

```
LOAD data local infile '~/capstone/users_500k.csv' INTO TABLE users  fields terminated BY '|'
ignore 1 rows;
```

We have used the above query to load Users table from a csv file downloaded

1. AD MANAGER:

A python code which handles the data that is being transmitted/produced from the kafka queue, transform it and point it to another system that is MySQL here to store the data.

First things first, we have to import all the necessary packages in to the environment.

CODE:

```
import sys
import mysql.connector
```

```
import json
from pykafka import KafkaClient
from pykafka.common import OffsetType
from pykafka.exceptions import SocketDisconnectedError, LeaderNotAvailable
from datetime import datetime
from decimal import *
import datetime
```

We have imported the necessary packages in to the deck and proceeding further. Sys package for system specific, mysql.connector for mysql connector, json to process json, pykafka to integrate python kafka with other connectors, datetime for processing datetime and decimal for processing decimal changes in it.

CODE:

```
if __name__ == "__main__":
    # Validating the Command line arguments
    if len(sys.argv) != 7:
        print("Usage: ad_manager.py <kafka_bootstrap_server> <kafka_topic> <database_host>
<database_username> <database_password> <database_name>")
        exit(-1)
```

Initiating the main method in order to capture the user provided system input and handling if it is lesser than the required values.

CODE:

```
kafka_bootstrap_server = sys.argv[1]
    kafka_topic = sys.argv[2]
    database_host = sys.argv[3]
    database_username = sys.argv[4]
    database_password = sys.argv[5]
    database_name = sys.argv[6]
```

Captured the values from CLI in to the variables to pass it to initiate method to use multiple times when required.

CODE:

```
    try:
        kafka_mysql_sink_connect = KafkaMySQLSinkConnect(kafka_bootstrap_server,
kafka_topic,database_host, database_username,database_password, database_name)
        kafka_mysql_sink_connect.process_kafka_events()
```

```
    except KeyboardInterrupt:
        print('Keyboard Interrupt detected!!! (ctrl+c) hitted, exiting the program...')

    finally:
        if kafka_mysql_sink_connect is not None:
            del kafka_mysql_sink_connect

def __del__(self):
        self.consumer.stop()
        self.db.close()
```

We are processing using exception handling in order to throw an error if it occurs. This will exit the program if we hit ctrl+c with the message printed in the console. As a best practice, we are terminating the instance and pass the same to stop consumer and DB connection using finally block which would handle to delete the instance which has been created earlier.

CODE:

```
def process_kafka_events(self):

        try:
            for queue_message in self.consumer:
                if queue_message is not None:
                    message = queue_message.value
                    self.process_row_data(message)

        except (SocketDisconnectedError, LeaderNotAvailable) as e:
            self.consumer.stop()
            self.consumer.start()
            self.process_kafka_events()
```

Then we are processing using exception handling in order to throw an error if it occurs from the main method to process_kafka_events. For every message received, we need to pass the same to process_row_data method in order to manipulate and use it wisely.
Restarting consumer and start processing, in any kafka connect error is the error handler do here.

CODE:

```
class KafkaMySQLSinkConnect:
    def __init__(self, kafka_bootstrap_server, kafka_topic_name,
database_host,database_username, database_password,database_name):
        kafka_client = KafkaClient(kafka_bootstrap_server)
```

```
    self.consumer = kafka_client \
     .topics[kafka_topic_name] \
     .get_simple_consumer(consumer_group="groupid",
     auto_offset_reset=OffsetType.LATEST)

    self.db = mysql.connector.connect(
     host=database_host,
     user=database_username,
     password=database_password,
     database=database_name
    )
```

Initializing Kafka Consumer with input from CLI with the bootstrap server. Parallelly, we have also initializing MySQL database connection with input from CLI using connector connection parameters.

CODE:

```
def process_row_data(self, text):
    db_cursor = self.db.cursor()
    db_cursor_select = self.db.cursor()
    db_cursor_replace = self.db.cursor()

    text_value = json.loads(text)

    text_derived = text_value['text']

    status = ''
    target_age_start= 0
    target_age_end= 0
    cpm = 0.0000
    current_slot_budget = 0.0000
    date_range_start = ''
    date_range_end = ''
    time_range_start = ''
    time_range_end = ''
```

This is a method to process single entire row, it would Getting the db cursor to read the entire db. The following step is parsing the json'ed value text into loads function. Assigning a value from Kafka to another variable and Initializing various variables to use further with default values.

CODE:

```
    if (text_value['action'].lower() == 'new campaign') | (text_value['action'].lower()  == 'update campaign'):
```

```
    status = 'ACTIVE'
  else:
    status = 'INACTIVE'
```

Deriving the status depends on the action received and it will update the MySQL ads table

CODE:

```
target_age_start = int(text_value['target_age_range']['start'])
target_age_end= int(text_value['target_age_range']["end"])
```

Deriving the target_age_start and end from the target_age_range received in the form of json

CODE:

```
cpc_weight = 0.0075
cpa_weight = 0.0005
cpm = (cpc_weight*float(text_value['cpc']))+(cpa_weight*float(text_value['cpa']))
```

Deriving the cpm depends on the cpa and cpc received and values are taken from upgrad portal as asked to use for cpc_weight and cpa_weight. The Formula given to sum the product values between weight and cpc_weight values respectively.

CODE:

```
slot1 = datetime.datetime.strptime(text_value['date_range']["end"]+"
"+text_value['time_range']["end"],"%Y-%m-%d %H:%M:%S")
    slot2 = datetime.datetime.strptime(text_value['date_range']["start"]+"
"+text_value['time_range']["start"],"%Y-%m-%d %H:%M:%S")
```

Deriving the slots and number of slots using the date time values provided

CODE:

```
slots = ((slot1-slot2).total_seconds() / 3600)*6
```

As the number of slots given, slot duration will be 10 minutes in upgrad portal. The same is followed here to take slot of 10 minutes each.

CODE:

```
current_slot_budget = float(text_value['budget']/int(slots))
```

Deriving the current_slot_budget using the budget and the number of slots derived from the formula provided.

CODE:

```
    date_range_start = text_value['date_range']["start"]
    date_range_end = text_value['date_range']["end"]
    time_range_start = text_value['time_range']["start"]
    time_range_end = text_value['time_range']["end"]
```

Deriving the date_range_start , time_range_start and end from the date_range,time_range received in the form of json.

CODE:

```
    db_cursor_select.execute("select * from capstone.ads where campaign_id = %s", text_value['campaign_id'])
    ads_retrieve = db_cursor_select.fetchall()
```

DB query for supporting SELECT operation, and it retrieves the values from ads table if available before.

CODE:

```
    replace_sql_query = """REPLACE INTO capstone.ads(text, category,keywords,campaign_id,status,target_gender,target_age_start,target_age_end,target_city,target_state,target_country,target_income_bucket,target_device,cpc,cpa,cpm,budget,current_slot_budget,date_range_start,date_range_end, time_range_start,time_range_end
    VALUES (%s, %s,%s,%s,%s,%s,%s,%s,%s, %s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s, %s,%s,%s)"""
replace_sql_values = (text_derived, text_value['category'],text_value['keywords'],text_value['campaign_id'],status,text_value['target_gender'],target_age_start,target_age_end,text_value['target_city'],text_value['target_state'],text_value['target_country'],text_value['target_income_bucket'],text_value['target_device'],text_value['cpc'],text_value['cpa'],cpm,text_value['budget'],current_slot_budget,date_range_start,date_range_end,time_range_start,time_range_end)
```

DB query for supporting UPSERT operation and the values which we have to pass in the replace query if the ad is available already to update the new values especially status.

CODE:

```
    insert_sql_query = """INSERT IGNORE INTO capstone.ads(text, category,keywords,campaign_id,status,target_gender,target_age_start,target_age_end,target_city,target_state,target_country,target_income_bucket,target_device,cpc,cpa,cpm,budget,current_slot_budget,date_range_start,date_range_end, time_range_start,time_range_end) \
    VALUES (%s, %s,%s,%s,%s,%s,%s,%s,%s, %s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s, %s,%s,%s)"""
insert_sql_values = (text_derived, text_value['category'],text_value['keywords'],text_value['campaign_id'],status,text_value['target_gender'],target_age_start,target_age_end,text_value['target_city'],text_value['target_state'],text_value['target_country'],text_value['target_income_bucket'],text_value['target_device'],text_value['cpc'],text_value['cpa'],cpm,text_value['budget'],current_slot_budget,date_range_start,date_range_end,time_range_start,time_range_end)
```

DB query for supporting INSERT operation and the values which we have to pass in the insert query if the ad is not available already. INSERT IGNORE has been used here to escape from error if occurs then.

CODE:

```
if len(ads_retrieve)!= 0:
        db_cursor_replace.execute(replace_sql_query,replace_sql_values)
    else:
        db_cursor.execute(insert_sql_query, insert_sql_values)
```

To handle the above-mentioned way for insert/replace, the above query uses the fetched result from the cursor to see if at least one row ad with same campaign_id. If we get the campaign, then we will replace them with new status we have got.

After which, *self.db.commit()* command will commit all the changes done if any in the MySQL query would be synced globally.

2. AD SERVER:

A python code which handles the data that is being transmitted/produced from the user simulator program which has been gathered from users table queue, transform it and point it to another system that is MySQL here to store the data using the Flask API.

First things first, we have to import all the necessary packages in to the environment.

CODE:

```
import sys
import flask
from flask import request, jsonify,make_response
import uuid
from flask import abort
import mysql.connector
```

We have imported the necessary packages in to the deck and proceeding further. Sys package for system specific, mysql.connector for mysql connector, flask to process API Request, jsonify to make the data jsonified and uuid for generating new unique id.

CODE:

```
app = flask.Flask(__name__)
app.config["DEBUG"] = True
app.config['RESTFUL_JSON'] = {"ensure_ascii": False}
```

we are setting the environment values for Flask for app configuration

CODE:

```
if __name__ == "__main__":
    if len(sys.argv) != 6:
        print("Usage: ad_server.py  <database_host> <database_username>
<database_password> <database_name>  <flask_app_port>")
        exit(-1)
```

Initiating the main method in order to capture the user provided system input and handling if it is lesser than the required values.

CODE:

```
    database_host = sys.argv[1]
    database_username = sys.argv[2]
    database_password = sys.argv[3]
    database_name = sys.argv[4]
```

Captured the values from CLI in to the variables to pass it to initiate method to use multiple times when required.

CODE:

```
    try:
        app.run(host=sys.argv[1], port=sys.argv[5])
    # Will exit the program if we hit ctrl+c
    except KeyboardInterrupt:
        print('Keyboard Interrupt detected!!! (ctrl+c) hitted, exiting the program...')
```

We are processing using exception handling in order to throw an error if it occurs. This will exit the program if we hit ctrl+c with the message printed in the console.

CODE:

```
@app.route('/ad/user/<request_id>/serve', methods=['GET'])
def serve(request_id):
    db = mysql.connector.connect(
            host=app.config.get('database_host'),
            user=app.config.get('database_username'),
            password=app.config.get('database_password'),
            database=app.config.get('database_name')
        )

    db_cursor_ad = db.cursor(buffered=True)
    db_cursor_users = db.cursor(buffered=True)
```

```
db_cursor_served_ads = db.cursor()
text1=''
```

Initializing MySQL database connection with input from CLI using connector connection parameters. Before that, we have created the route for the provided url in order to fetch the required parameters and query parameters respectively. And, we are initializing three cursors in order to do three distinct operations simultaneously.

CODE:

```
if request_id != '1111-1111-1111-1111':

    device_t = request.args['device_type']
    city_t = request.args['city']
    state_t = request.args['state']
    user_id = request_id

    db_cursor_users.execute("Select * from users where id = %s",(user_id,))

    users = db_cursor_users.fetchall()
```

Here, we are handling the request to not to go for masked user, and then obtained values will be captured thru the request.args respectively. We are fetching the userid from request_id and retrieve other values from the users table to pass next.

CODE:

```
    for user in users:
        target_gen = user[2]
        target_age = user[1]
        target_income = user[4]

    select_query_ads = """Select distinct text,campaign_id,
target_device,cpa,cpc,cpm,target_age_start,target_age_end,target_city,target_state,target_c
ountry,target_gender, target_income_bucket,Date_range_start,Date_range_end
,Time_range_start ,Time_range_end  from ads where  (target_device='All' or target_device =
%s)  and ( (target_city='All' or target_city = %s) and (target_state='All' or target_state = %s))
and  (target_country='All' or target_country = %s) and (Target_gender = %s or Target_gender=
'All') and ((target_age_start >= %s and target_age_end <= %s) or (target_age_start=0 and
target_age_end=0)) and (target_income_bucket = %s or target_income_bucket='All') and status
= 'ACTIVE'  order by cpm desc limit 2 """

db_cursor_ad.execute(select_query_ads,(device_t,city_t,state_t,target_gen,target_age,target_
age,target_income,))
```

```
ads = db_cursor_ad.fetchall()
```

Then, looping over user to get the details of gender, age, income_bucket to pass the values into select query. DB select query to capture the available ads for the specified condition and with 'All' wildcards

The DB execute query to capture the available ads for the specified condition to execute and fetching all the values for the cursors.

***CODE**:*

```
if len(ads)!=2:
        for ad in ads:
            text1=ad[0]
            campaign_id = ad[1]
            auc_cpm = ad[5] # Paying the same auction as it is the single winner
            auc_cpc = ad[4]
            auc_cpa = ad[3]
            targ_age_range = {"start": ad[6],"end": ad[7]} # Clubbing the values in the json format
            targ_loc = {"city": ad[8],"state": ad[9],"country": ad[10]} # Clubbing the values in the
json format
            targ_gen = ad[11]
            targ_income = ad[12]
            targ_dev = ad[2]
            camp_start = ad[13]+" "+ad[15] # Clubbing the values in the string format
            camp_end = ad[14]+" "+ad[16]
```

Here, handling the condition to see if there is only one ad available for the condition specified and the required parameter values would be fetched.

The same kind of block will also go for else i.e., more than one ad available.

***CODE**:*

```
else:
        text1=ads[0][0]
        campaign_id = ads[0][1]
        auc_cpm = ads[1][5] # Paying the auction to the second ad winner
        auc_cpc = ads[0][4]
        auc_cpa = ads[0][3]
        targ_age_range = {"start": ads[0][6],"end": ads[0][7]} # Clubbing the values in the json
format
        targ_loc = {"city": ads[0][8],"state": ads[0][9],"country": ads[0][10]} # Clubbing the values
in the json format
        targ_gen = ads[0][11]
        targ_income = ads[0][12]
```

```
        targ_dev = ads[0][2]
        camp_start = ads[0][13]+" "+ads[0][15] # Clubbing the values in the string format
        camp_end = ads[0][14]+" "+ads[0][16]
```

As fore-mentioned, more than one ad available would be taken (only top 2 ads are taken as per sql query descending order by cpm value). So, the winner will be paid with the second winner bid amount as given

***CODE**:*

```
else:
    # Gathering the arguments from the request
    device_t = request.args['device_type']
    city_t = request.args['city']
    state_t = request.args['state']
    user_id = request_id

    select_query_ads = """Select distinct text,campaign_id,
target_device,cpa,cpc,cpm,target_age_start,target_age_end,target_city,target_state,target_c
ountry,
    target_gender,target_income_bucket,Date_range_start,Date_range_end
,Time_range_start ,Time_range_end
    from ads where (target_device='All' or target_device = %s) and (target_city='All' or
target_city = %s)
    and (target_state='All' or target_state = %s)and (target_country='All' or target_country =
%s) and status = 'ACTIVE'  order by cpm desc limit 2
    """

    # DB select query to capture the available ads for the specified condition
    db_cursor_ad.execute(select_query_ads,(device_t,city_t,state_t,))

    # Fetching all the values for the cursors
    ads = db_cursor_ad.fetchall()

    if len(ads)!=2:
      for ad in ads:
        text1=ad[0]
        campaign_id = ad[1] # Paying the same auction as it is the single winner
        auc_cpm = ad[5]
        auc_cpc = ad[4]
        auc_cpa = ad[3]
        targ_age_range = {"start": ad[6],"end": ad[7]}
        targ_loc = {"city": ad[8],"state": ad[9],"country": ad[10]}
        targ_gen = ad[11]
        targ_income = ad[12]
```

```
        targ_dev = ad[2]
        camp_start = ad[13]+" "+ad[15]
        camp_end = ad[14]+" "+ad[16]
    else:
      text1=ads[0][0]
      campaign_id = ads[0][1]
      auc_cpm = ads[1][5] # Paying the auction to the second ad winner
      auc_cpc = ads[0][4]
      auc_cpa = ads[0][3]
      targ_age_range = {"start": ads[0][6],"end": ads[0][7]}
      targ_loc = {"city": ads[0][8],"state": ads[0][9],"country": ads[0][10]}
      targ_gen = ads[0][11]
      targ_income = ads[0][12]
      targ_dev = ads[0][2]
      camp_start = ads[0][13]+" "+ads[0][15]
      camp_end = ads[0][14]+" "+ads[0][16]
```

The above full portion is coded for non-masked user, this else condition will handle the same for masked user i.e**., user_id = '1111-1111-1111-1111'.**

***CODE*:**

```
if len(ads)!=0:

    request_id_derived = str(uuid.uuid1())

    insert_sql = """INSERT IGNORE INTO capstone.served_ads(Request_id,
Campaign_id,User_id,Auction_cpm,Auction_cpc,Auction_cpa,Target_age_range,Target_locatio
n,Target_gender,Target_income_bucket,Target_device_type,Campaign_start_time,Campaign_
end_time,Time_stamp) \
    VALUES (%s, %s,%s,%s,%s,%s,%s,%s,%s, %s,%s,%s,%s,%s)"""


    insert_val = (request_id_derived,
campaign_id,user_id,auc_cpm,auc_cpc,auc_cpa,str(targ_age_range),str(targ_loc),targ_gen,tar
g_income,targ_dev,camp_start,camp_end,"")

    db_cursor_served_ads.execute(insert_sql, insert_val)

    db.commit()

    db.close()
```

Here, we are handling the condition to verify if at least one row is getting updated. Deriving the unique id from **uuid** package would be the id for this table. DB insert query for the captured ads for the user will be processed. Insert Query value to pass in to insert query for **served_ads** table and executing the query. Committing the DB in order to values get reflected instantly in the table globally.

It will return the jsonify object with the values provided as **request_id** and **ad campaign text**

3. SLOT BUDGET MANAGER:

A python code which handles the current_slot_budget that is being stored in the ads table, would update the values based on when the slot ends and store the same in system that is MySQL here to store the updated data using the Cron Job.

First things first, we have to import all the necessary packages in to the environment.

CODE:

```
import sys
import mysql.connector
import datetime
from decimal import Decimal
```

A python code which handles the data that is being transmitted/produced from the kafka queue, transform it and point it to another system that is MySQL here to store the data.

First things first, we have to import all the necessary packages in to the environment.

CODE:

```
if __name__ == "__main__":
    if len(sys.argv) != 5:
        print("Usage: slot_budget_updater.py <database_host> <database_username>
<database_password> <database_name>")
        exit(-1)
```

Initiating the main method in order to capture the user provided system input and handling if it is lesser than the required values.

CODE:

```
    app.config['database_host'] = sys.argv[1]
    app.config['database_username'] = sys.argv[2]
    app.config['database_password'] = sys.argv[3]
    app.config['database_name'] = sys.argv[4]
```

Captured the values from CLI in to the variables to pass it to initiate method to use multiple times when required.

CODE:

```
try:
      budget_manager = BudgetManager(database_host,
database_username,database_password, database_name)
      budget_manager.process_row_data()
   # Will exit the program if we hit ctrl+c
   except KeyboardInterrupt:
      print('KeyboardInterrupt (ctrl+c) da SATHYA, exiting...')
   # As a best practice, we are terminating the instance and pass the same to stop consumer and
DB connection using finally
   finally:
      if budget_manager is not None:
         del budget_manager

def __del__(self):
      self.db.close()
```

We are processing using exception handling in order to throw an error if it occurs. This will exit the program if we hit ctrl+c with the message printed in the console. As a best practice, we are terminating the instance and pass the same to stop consumer and DB connection using finally block which would handle to delete the instance which has been created earlier.

CODE:

```
class BudgetManager:
   def __init__(self, database_host,database_username, database_password,database_name):

      # Initializing MySQL database connection with input from CLI using connector
      self.db = mysql.connector.connect(
       host=database_host,
       user=database_username,
       password=database_password,
       database=database_name
      )
```

Initializing Kafka Consumer with input from CLI with the bootstrap server. Parallelly, we have also initializing MySQL database connection with input from CLI using connector connection parameters.

CODE:

```
def process_row_data(self):
```

```
    # We are initializing two cursors in order to do two distinct operations simultaneously
    db_cursor_ads = self.db.cursor(buffered=True)
    db_cursor_update = self.db.cursor()

    # DB select query to capture the available all ads data to update current_slot_budget
    db_cursor_ads.execute("Select
campaign_id,budget,Target_city,Target_device,Date_range_start,Date_range_end
,Time_range_start ,Time_range_end from ads")

    # Fetching all the values for the cursors
    ads = db_cursor_ads.fetchall()
```

Initializing MySQL database connection with input from CLI using connector connection parameters. Before that, we have created the route for the provided url in order to fetch the required parameters and query parameters respectively. And, we are initializing two cursors in order to do two distinct operations simultaneously.

CODE:

```
for ad in ads:
        camp_id = ad[0]
        budget = ad[1]
        target_city = ad[2]
        target_dev = ad[3]
        dt_range_end = ad[5]
        tm_range_end = ad[7]
        dt_range_start = ad[4]
        tm_range_start = ad[6]
```

Assigning all the values from the ad variable received from sql.

CODE:

```
slot1 = datetime.datetime.strptime(text_value['date_range']["end"]+"
"+text_value['time_range']["end"],"%Y-%m-%d %H:%M:%S")
    slot2 = datetime.datetime.strptime(text_value['date_range']["start"]+"
"+text_value['time_range']["start"],"%Y-%m-%d %H:%M:%S")
```

Deriving the slots and number of slots using the date time values provided

CODE:

```
slots = ((slot1-slot2).total_seconds() / 3600)*6
```

As the number of slots given, slot duration will be 10 minutes in upgrad portal. The same is followed here to take slot of 10 minutes each.

CODE:

```
if slots_number!=0 or slots_number!=0.0 or budget!=0.0:
        current_slot_budget = float(budget)/float(slots_number)# The formula given in upgrad portal is followed here
    else:
        current_slot_budget = 0.0
        budget = 0.0
```

Updating the current_slot_budget from the retrieved value and update the value depends on the slots calculated.

CODE:

```
budget_derived = budget - current_slot_budget
```

Subtracting the budget to derive the values.

CODE:

```
if slots_number !=0 or slots_number!=0.0:
        dt_time_difference = slot2+datetime.timedelta(minutes = 10)
        dt_range_start = dt_time_difference.date() # Updating the new start date
        tm_range_start = dt_time_difference.time() # Updating the new time date
    else:
        dt_range_start = dt_range_start # Updating the new start date
        tm_range_start = tm_range_start # Updating the new time date
```

Moving the window time by ten minutes and updating the datetime elements accordingly.

CODE:

```
    sql_update = """UPDATE capstone.ads SET budget = %s,Current_slot_budget = %s ,Date_range_start = %s,Date_range_end = %s, Time_range_start = %s, Time_range_end = %s
    where  campaign_id = %s and target_city = %s and target_device = %s """

    update_val = (budget_derived,current_slot_budget,dt_range_start,dt_range_end,tm_range_start,tm_range_end,camp_id,target_city,target_dev )

    db_cursor_update.execute(sql_update, update_val)

    self.db.commit()
```

DB query for supporting UPDATE operation and the values which we have to pass in the update query are given and then Execute command to update the sql in to the ads table.
At last Commits the operation, so that it reflects globally in the ads table.