AdAptify: AI-Driven Personalized Marketing PLatform with Flask on AWS EC2 and RDS

Project Description

The **"AWS AI-Driven Personalized Marketing Platform"** project involves developing a scalable and intelligent marketing solution that utilizes artificial intelligence to deliver personalized customer experiences. This platform is built using Flask for backend development, hosted on AWS Elastic Beanstalk for seamless deployment, and integrates Amazon RDS for robust database management. By leveraging AWS services, this cloud-native solution provides high availability, scalability, and efficient management of user interactions, marketing campaigns, and data analytics. The project illustrates how advanced machine learning techniques can create a powerful infrastructure for engaging customers through targeted marketing strategies.

Scenario 1: Personalized Marketing Campaigns for E-Commerce

In the e-commerce landscape, the AWS AI-Driven Personalized Marketing Platform allows businesses to create tailored marketing campaigns that resonate with individual customer preferences. For instance, an online clothing retailer can use the platform to analyze customer browsing behavior and purchase history to deliver personalized email campaigns featuring recommended products. By leveraging the scalability of AWS services, the platform can handle increased customer traffic during promotional events, ensuring that marketing content is delivered in real time without lag, thereby enhancing customer engagement and driving sales.

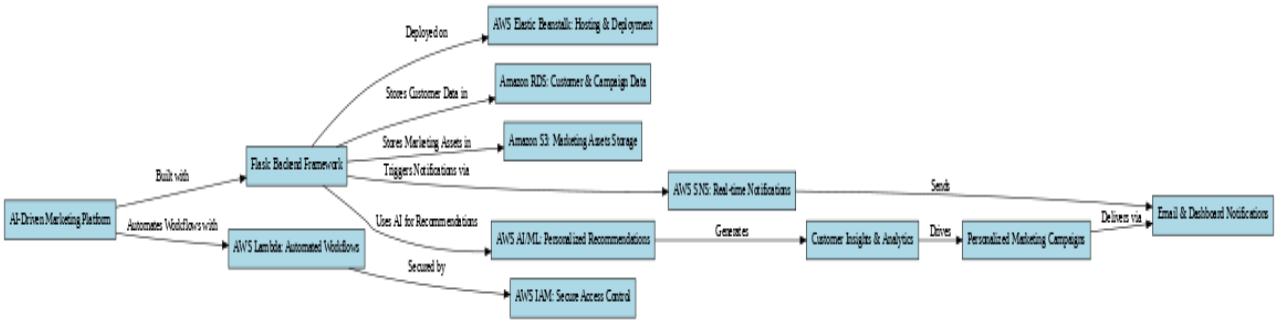
Scenario 2: Real-Time Customer Insights for B2B Solutions

For B2B applications, the platform offers valuable insights into customer behavior through machine learning-driven analytics. Consider a SaaS company providing marketing tools for businesses. The platform can analyze user data to identify trends, segment audiences, and recommend personalized strategies for client campaigns. Utilizing Amazon RDS ensures that user data is securely stored and easily accessible, enabling quick analysis and reporting. This managed database solution simplifies operations, allowing the SaaS platform to scale alongside its user base while maintaining data integrity and high performance.

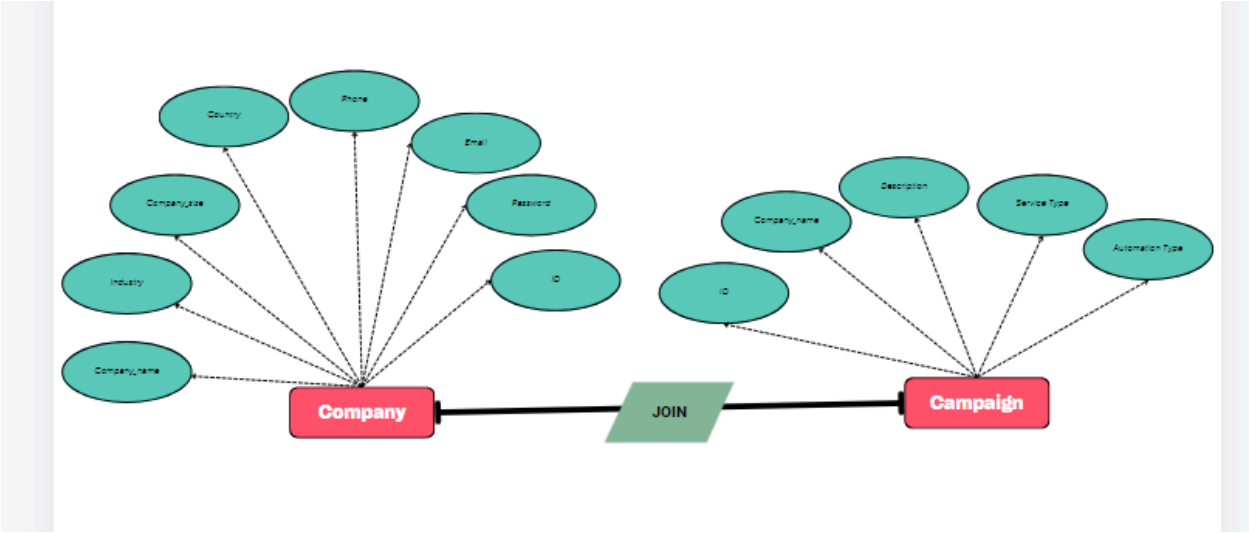
Scenario 3: Automated Marketing Workflows for Health and Wellness

In the health and wellness sector, the AWS AI-Driven Personalized Marketing Platform can automate marketing workflows to engage patients effectively. For example, a telehealth provider can use the platform to send personalized reminders for appointments, follow-up care, or wellness tips based on patient interactions and preferences. By integrating AWS Lambda for automating these processes and ensuring secure access through IAM (Identity and Access Management), the platform maintains compliance with healthcare regulations while providing timely communication. The scalability of AWS services ensures that the application can adapt to varying levels of patient engagement, enhancing the overall user experience.

TECHNICAL ARCHITECTURE:



ER Diagram:



Pre-requisites:

1. AWS Account Setup: https://youtu.be/CjKhQoYeR4Q?si=ui8Bvk_M4FfVM-Dh
2. Understanding of IAM: https://youtu.be/gsgdAyGhV0o?si=3qg-bULgkD4LXNvR
3. Knowledge of Amazon EC2 : https://youtu.be/8TlukLu11Yo?si=MUj0nEAOESRhHUIz
4. Mobaxtream : https://youtu.be/dvoU2SKG6oA?si=Hs8Pu4Crry5-BRrD
5. RDS : https://www.youtube.com/live/MPau9c7PT74?si=A8OK-zFGbSKkAFWN
6. MySQL WorkBench: https://youtu.be/wALCw0F8e9M?si=ovMF9qMx5rLxaznB

Project WorkFlow:

1. Project Initialization

- Define objectives, scope, and KPIs for deploying the AWS AI-Driven Personalized Marketing Platform.
- Set up the AWS environment, including the configuration of Elastic Beanstalk, Amazon RDS, and S3 for content storage.
- Outline the use of Flask for backend development and detail the integration of AWS services such as SageMaker and Lambda for automation and machine learning capabilities.

2. Elastic Beanstalk Environment Creation:

- Launch an Elastic Beanstalk environment to host the marketing platform application.
- Choose the appropriate configuration based on expected user load, resource requirements, and application architecture.
- Ensure the environment is set up to support automatic scaling and load balancing.

3. RDS Configuration:

- Set up Amazon RDS for database management with MySQL to store user profiles, marketing preferences, and campaign data.
- Configure the RDS instances, including security settings, parameter groups, and access controls to ensure data integrity and protection.

4. Flask Application Development and Deployment:

- Develop the AWS AI-Driven Personalized Marketing Platform application using Flask.
- Transfer application files to the Elastic Beanstalk environment and configure the deployment settings to ensure compatibility with AWS services.
- Implement necessary APIs for user authentication, campaign management, and data analytics.

5. Content Storage Configuration:

- Set up Amazon S3 for storing marketing assets, such as images, templates, and videos.
- Configure bucket policies and permissions to ensure secure access and efficient retrieval of content by the application.

6. Testing and Optimization:

- Conduct comprehensive testing of the application for functionality, performance, and security, including unit tests, integration tests, and user acceptance testing.

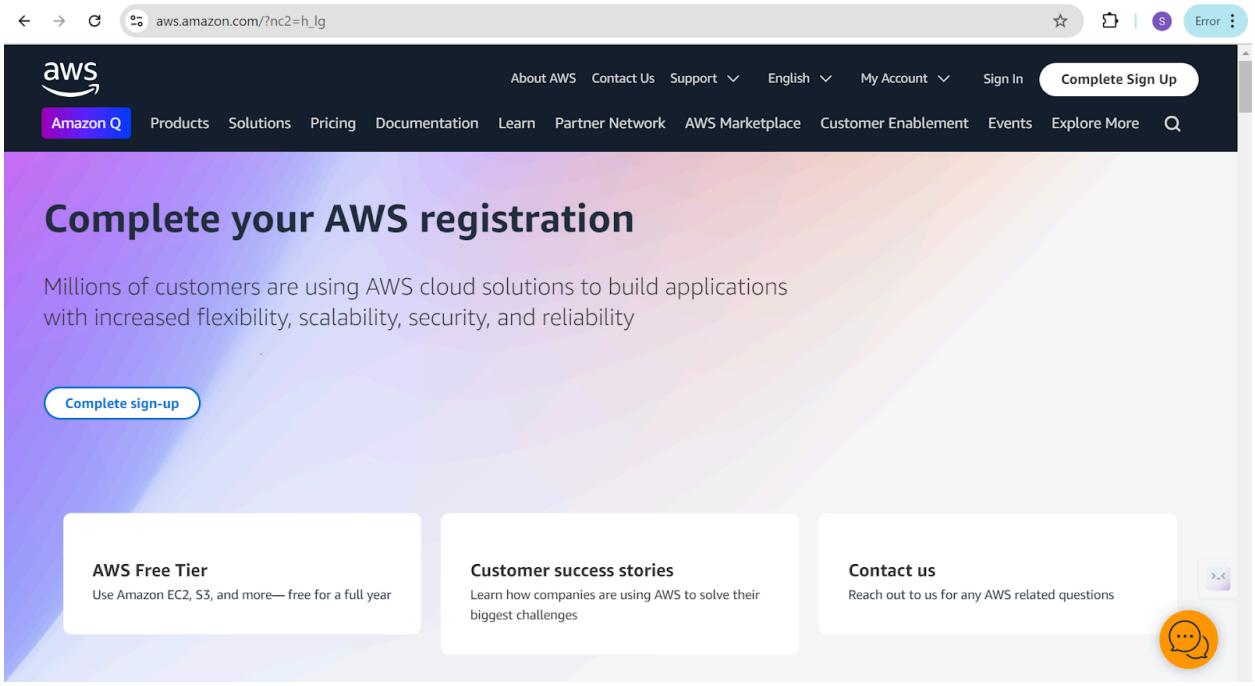
- Optimize server settings, database configurations, and application performance to ensure smooth operation and quick response times under various loads.

7. Monitoring and Maintenance:

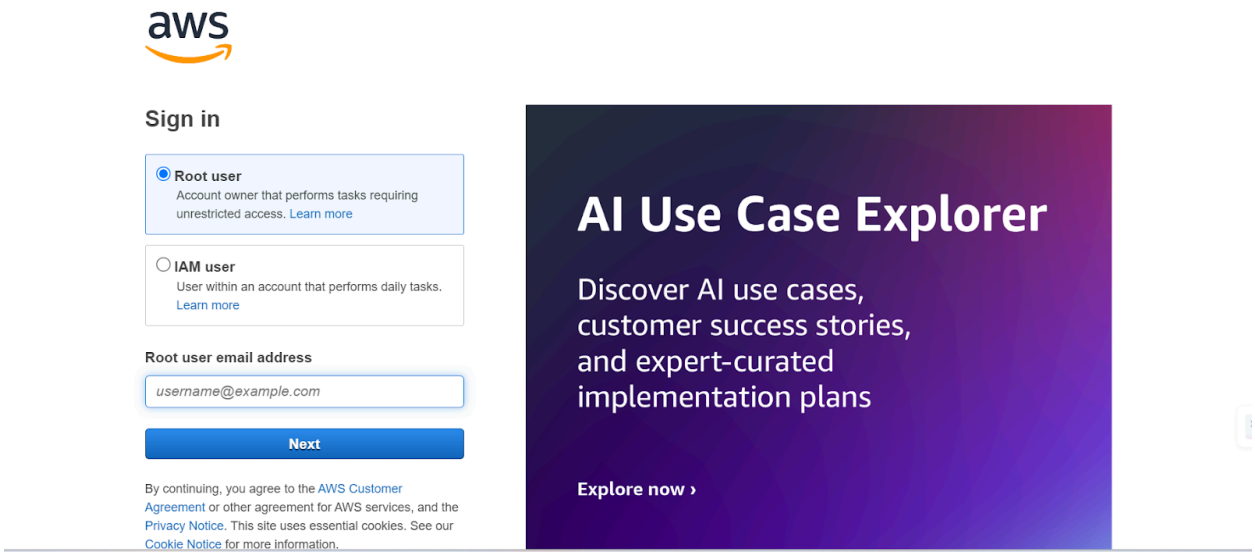
- Implement monitoring tools (e.g., AWS CloudWatch) to track application performance, user engagement, and overall uptime.
- Establish regular maintenance routines, including backups, updates, and security patches to ensure ongoing reliability, scalability, and compliance with industry standards.

Milestone 1: AWS Account Setup and Login

- **Activity 1.1: Create AWS Account**
  - Sign up for an AWS account and configure billing settings.

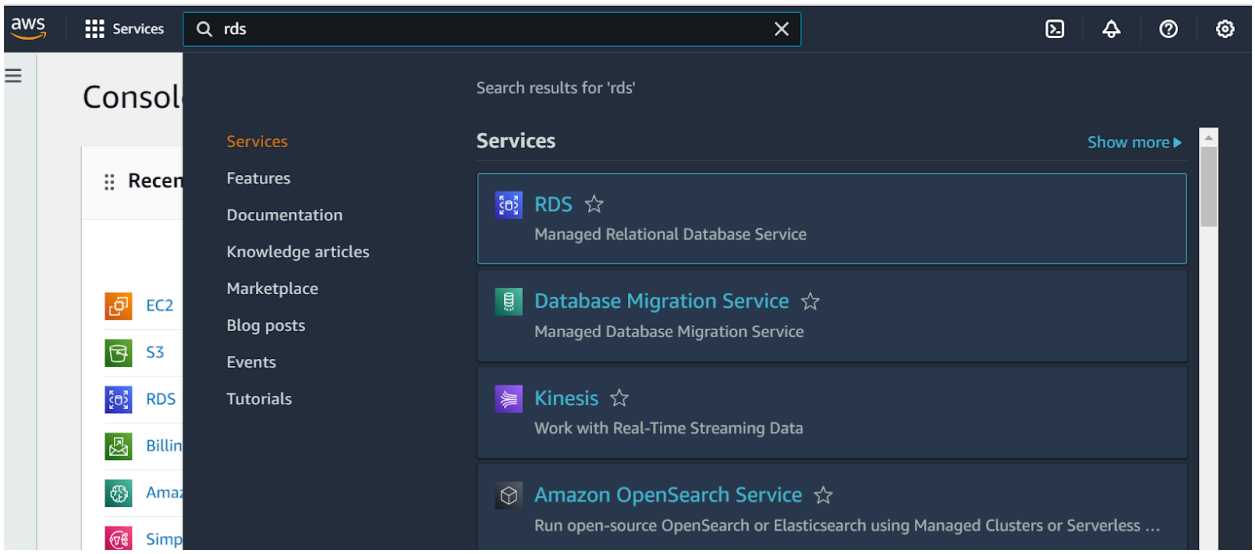


- **Activity 1.2: Log in to AWS Management Console**
  - Access the AWS Management Console using your login credentials.

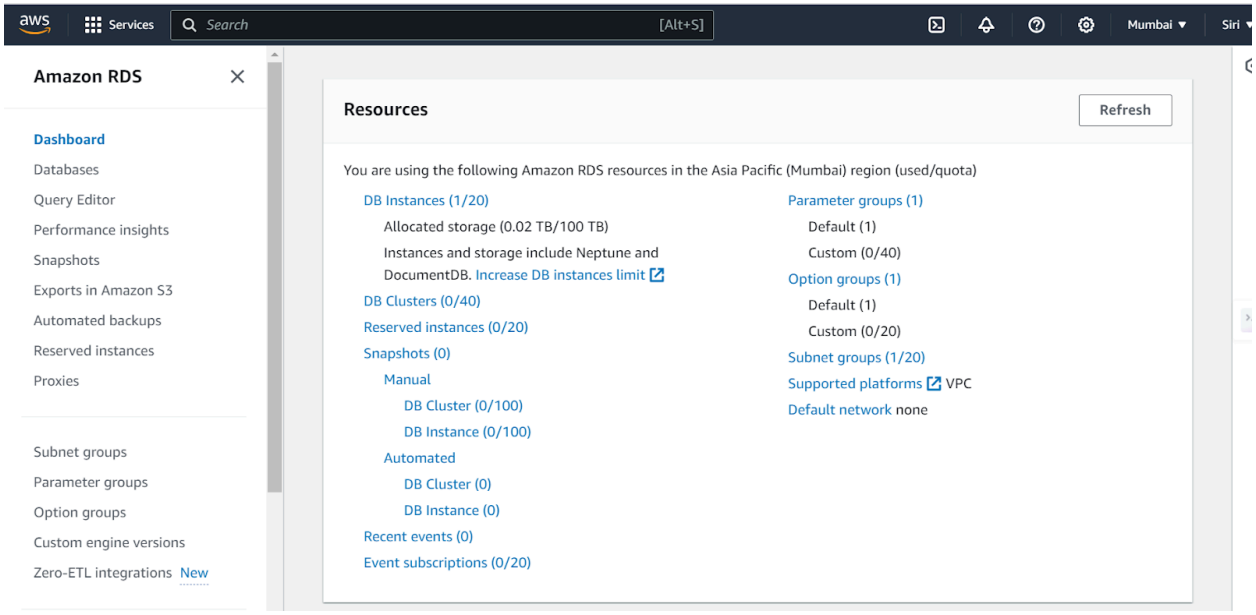


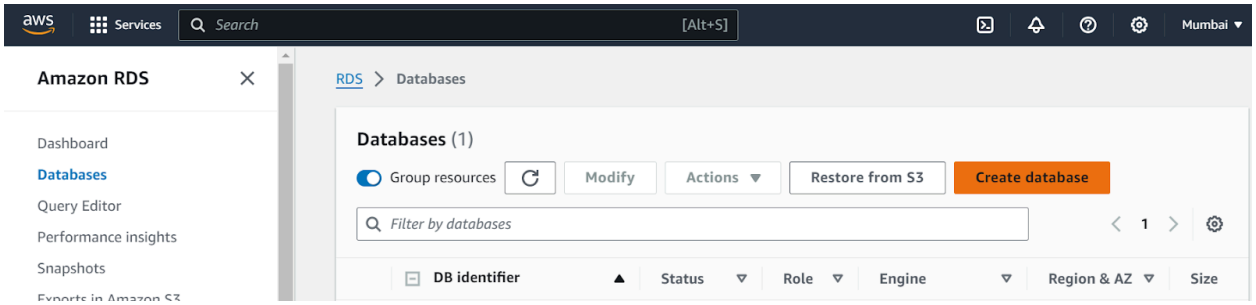
Milestone 2: RDS Database Creation and Setup

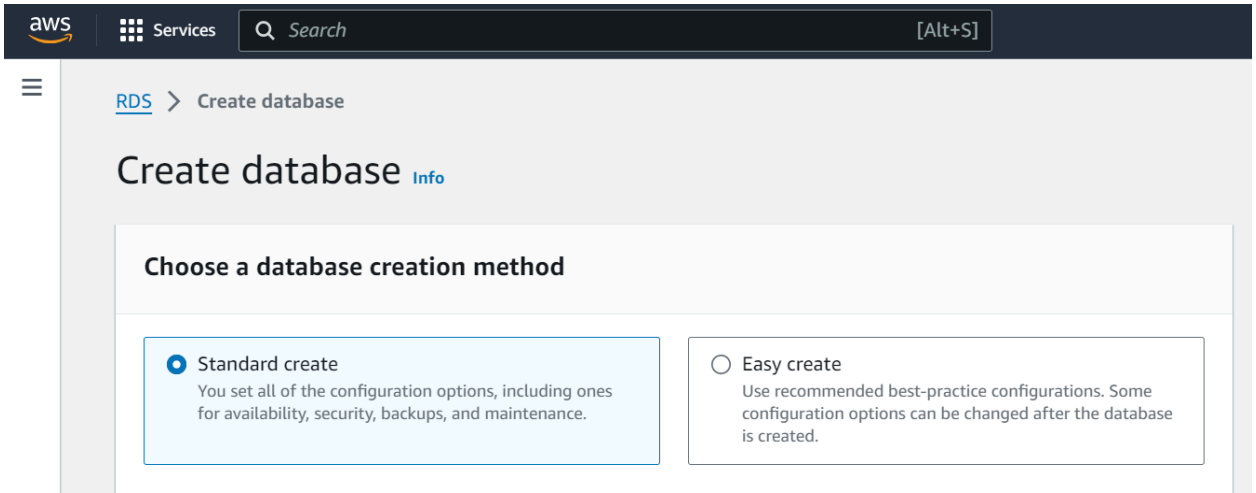
- **Activity 2.1: Create an RDS Instance**
  - Choose the RDS service from the AWS Management Console.

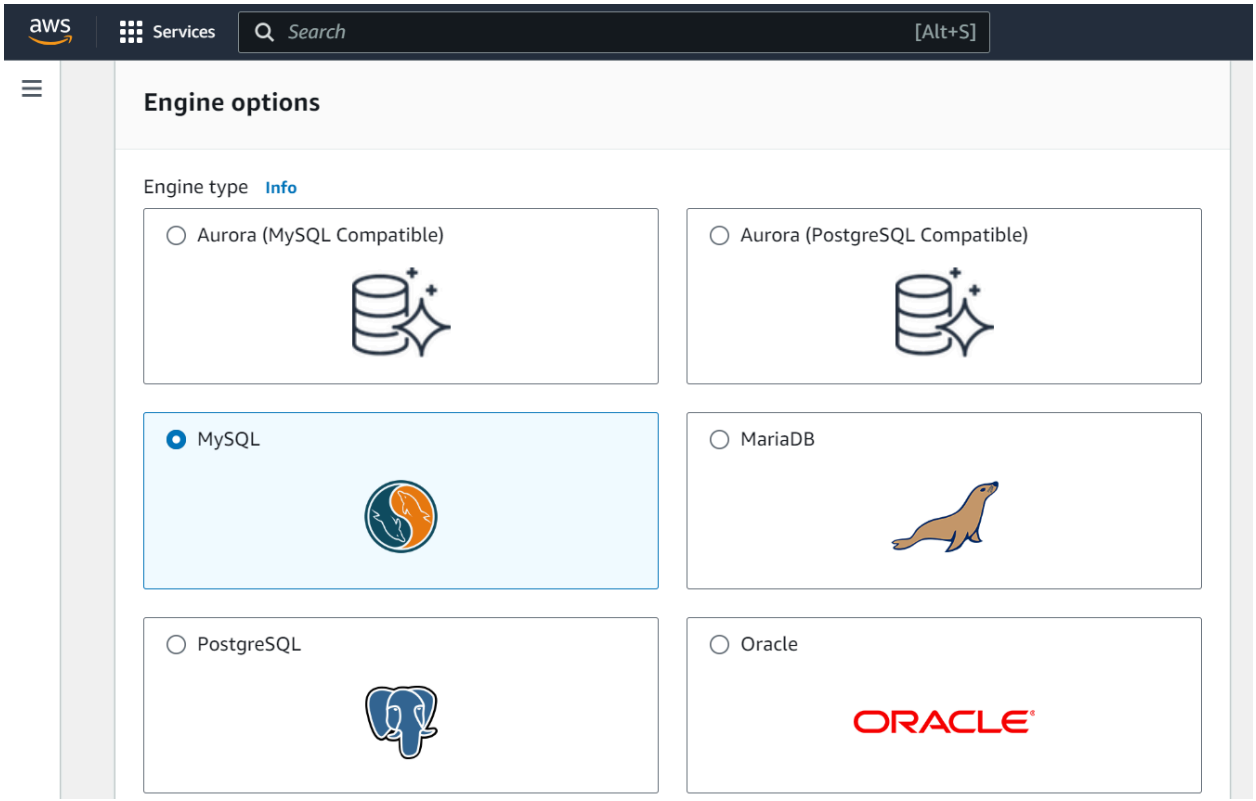


  - Select MySQL as the database engine, configure the instance settings (e.g., storage, instance class), and launch the RDS instance.


Services
Search
[Alt+S]
Mumbai
Siri
Amazon RDS
Dashboard
Databases
Query Editor
Performance insights
Snapshots
Exports in Amazon S3
Automated backups
Reserved instances
Proxies
Subnet groups
Parameter groups
Option groups
Custom engine versions
Zero-ETL integrations New
Resources
Refresh
You are using the following Amazon RDS resources in the Asia Pacific (Mumbai) region (used/quota)
DB Instances (1/20)
Allocated storage (0.02 TB/100 TB)
Instances and storage include Neptune and DocumentDB. Increase DB instances limit
DB Clusters (0/40)
Reserved instances (0/20)
Snapshots (0)
Manual
DB Cluster (0/100)
DB Instance (0/100)
Automated
DB Cluster (0)
DB Instance (0)
Recent events (0)
Event subscriptions (0/20)
Parameter groups (1)
Default (1)
Custom (0/40)
Option groups (1)
Default (1)
Custom (0/20)
Subnet groups (1/20)
Supported platforms VPC
Default network none


Services
Search
[Alt+S]
Mumbai
Amazon RDS
Dashboard
Databases
Query Editor
Performance insights
Snapshots
RDS > Databases
Databases (1)
Group resources
Modify
Actions
Restore from S3
Create database
Filter by databases
1
DB identifier
Status
Role
Engine
Region & AZ
Size


Services
Search
[Alt+S]
RDS > Create database
Create database Info
Choose a database creation method
Standard create
You set all of the configuration options, including ones for availability, security, backups, and maintenance.
Easy create
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.



Edition

MySQL Community

Engine version Info

View the engine versions that support the following database features.

Hide filters

Show versions that support the Multi-AZ DB cluster Info

Create a A Multi-AZ DB cluster with one primary DB instance and two readable standby DB instances. Multi-AZ DB clusters provide up to 2x faster transaction commit latency and automatic failover in typically under 35 seconds.

Show versions that support the Amazon RDS Optimized Writes Info

Amazon RDS Optimized Writes improves write throughput by up to 2x at no additional cost.

Engine Version

MySQL 8.0.35

Enable RDS Extended Support Info

Amazon RDS Extended Support is a paid offering. By selecting this option, you consent to being charged for this offering if you are running your database major version past the RDS end of standard support date for that version. Check the end of standard support date for your major version in the RDS for MySQL documentation.

Templates

Choose a sample template to meet your use case.

- ○ **Production**
  Use defaults for high availability and fast, consistent performance.
- ○ **Dev/Test**
  This instance is intended for development use outside of a production environment.
- ◉ **Free tier**
  Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS.
  Info

Availability and durability

Deployment options Info

The deployment options below are limited to those supported by the engine you selected above.

- ○ Multi-AZ DB Cluster
  Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.
- ○ Multi-AZ DB instance (not supported for Multi-AZ DB cluster snapshot)
  Creates a primary DB instance and a standby DB instance in a different AZ. Provides high availability and data redundancy, but the standby DB instance doesn't support connections for read workloads.
- ◉ Single DB instance (not supported for Multi-AZ DB cluster snapshot)
  Creates a single DB instance with no standby DB instances.

DB instance identifier Info

Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

marketings

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB instance.

root

1 to 16 alphanumeric characters. The first character must be a letter.

Credentials management

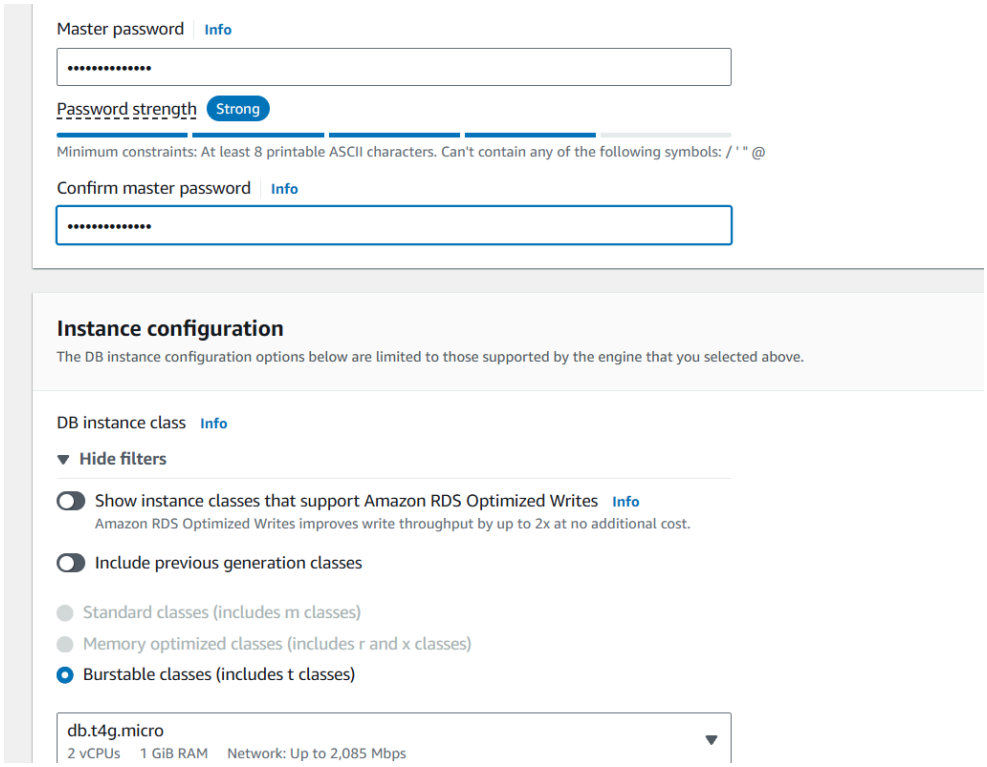
You can use AWS Secrets Manager or manage your master user credentials.

- ○ Managed in AWS Secrets Manager - *most secure*
  RDS generates a password for you and manages it throughout its lifecycle using AWS Secrets Manager.
- ◉ Self managed
  Create your own password or have RDS create a password that you manage.

☐ Auto generate password

Amazon RDS can generate a password for you, or you can specify your own password.

Master password Info


Master password Info
Password strength Strong
Minimum constraints: At least 8 printable ASCII characters. Can't contain any of the following symbols: / ' " @
Confirm master password Info
Instance configuration
The DB instance configuration options below are limited to those supported by the engine that you selected above.
DB instance class Info
Hide filters
Show instance classes that support Amazon RDS Optimized Writes Info
Amazon RDS Optimized Writes improves write throughput by up to 2x at no additional cost.
Include previous generation classes
Standard classes (includes m classes)
Memory optimized classes (includes r and x classes)
Burstable classes (includes t classes)
db.t4g.micro
2 vCPUs 1 GiB RAM Network: Up to 2,085 Mbps

Storage

Storage type **Info**

Provisioned IOPS SSD (io2) storage volumes are now available.

General Purpose SSD (gp3)
Performance scales independently from storage

Allocated storage **Info**

20 GiB

Minimum: 20 GiB. Maximum: 6,144 GiB

After you modify the storage for a DB instance, the status of the DB instance will be in storage-optimization. Your instance will remain available as the storage-optimization operation completes. Learn more

▶ **Advanced settings**

Baseline IOPS of 3,000 IOPS and storage throughput of 125 MiBps are included for allocated storage less than 400 GiB.

▶ **Storage autoscaling**

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

- (●) Don't connect to an EC2 compute resource
  Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.
- ( ) Connect to an EC2 compute resource
  Set up a connection to an EC2 compute resource for this database.

Network type **Info**

To use dual-stack mode, make sure that you associate an IPv6 CIDR block with a subnet in the VPC you specify.

- (●) IPv4
  Your resources can communicate only over the IPv4 addressing protocol.
- ( ) Dual-stack mode
  Your resources can communicate over IPv4, IPv6, or both.

Virtual private cloud (VPC) **Info**

Choose the VPC. The VPC defines the virtual networking environment for this DB instance.

vpc-0c9a09cc4c0ef9dd5
3 Subnets, 3 Availability Zones

Only VPCs with a corresponding DB subnet group are listed.

After a database is created, you can't change its VPC.

Public access Info

- (•) Yes
  RDS assigns a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.
- ( ) No
  RDS doesn't assign a public IP address to the database. Only Amazon EC2 instances and other resources inside the VPC can connect to your database. Choose one or more VPC security groups that specify which resources can connect to the database.

VPC security group (firewall) Info

Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

- ( ) Choose existing
  Choose existing VPC security groups
- (•) Create new
  Create new VPC security group

New VPC security group name

new security group

Availability Zone Info

No preference

RDS Proxy

RDS Proxy is a fully managed, highly available database proxy that improves application scalability, resiliency, and security.

- [ ] Create an RDS Proxy Info
  RDS automatically creates an IAM role and a Secrets Manager secret for the proxy. RDS Proxy has additional costs. For more information, see Amazon RDS Proxy pricing.

Database authentication

Database authentication options Info

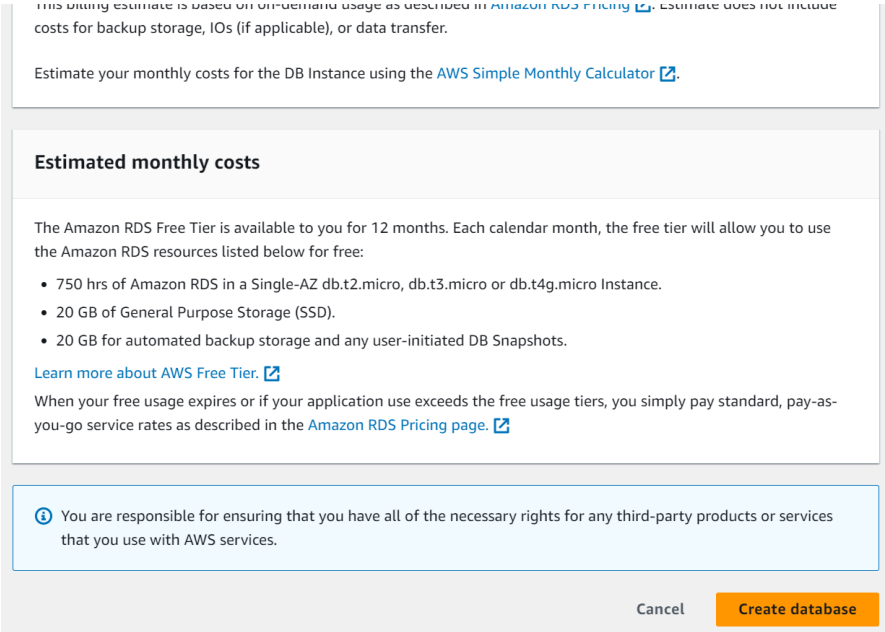
- (•) Password authentication
  Authenticates using database passwords.
- ( ) Password and IAM database authentication
  Authenticates using the database password and user credentials through AWS IAM users and roles.
- ( ) Password and Kerberos authentication
  Choose a directory in which you want to allow authorized users to authenticate with this DB instance using Kerberos Authentication.

Monitoring

- [ ] Enable Enhanced Monitoring
  Enabling Enhanced Monitoring metrics are useful when you want to see how different processes or threads use the CPU.

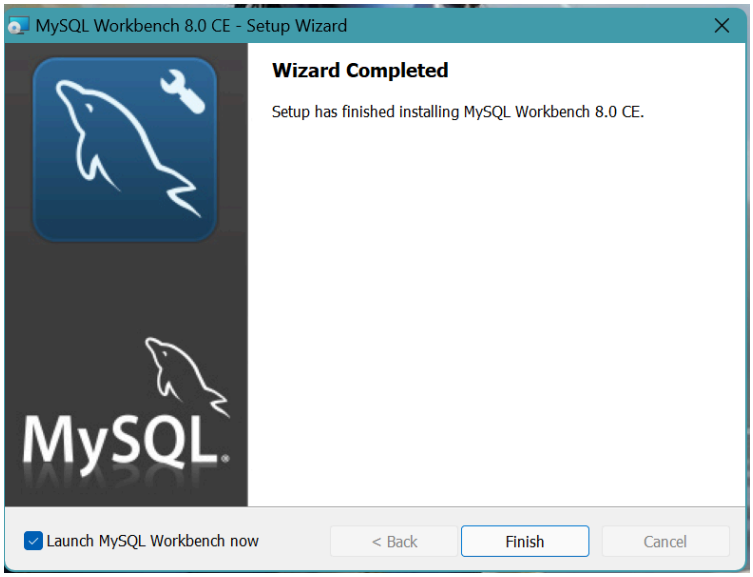
▸ Additional configuration

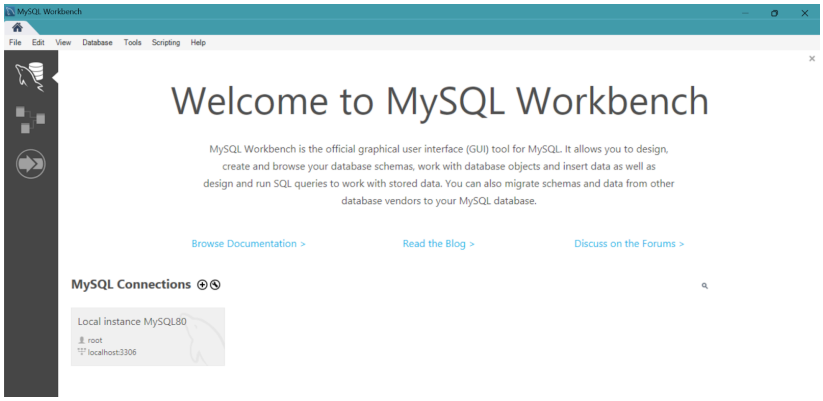
Database options, encryption turned on, backup turned on, backtrack turned off, maintenance, CloudWatch Logs, delete protection turned off.



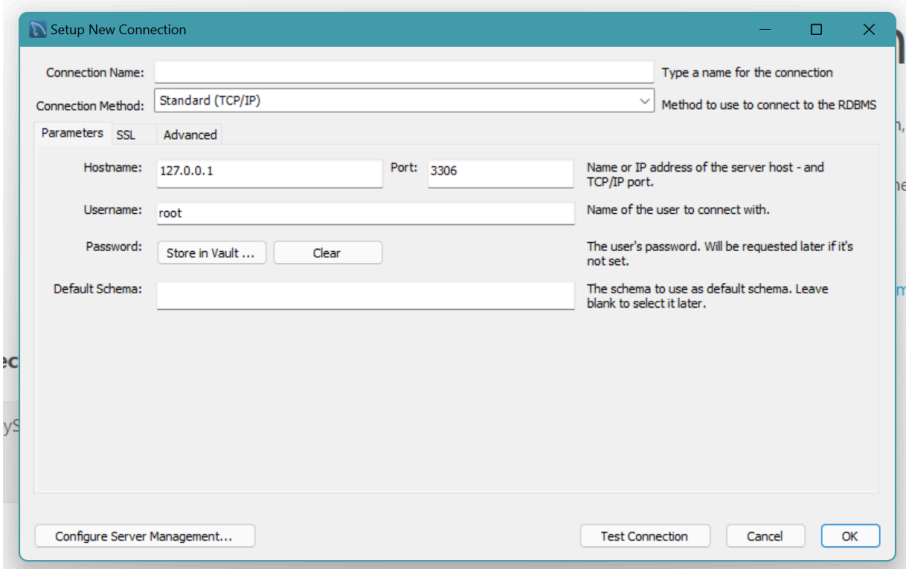
- **Activity 2.2: Configure Database Access**
  - Set up security groups, create database credentials, and configure access policies to ensure secure connectivity to the database.

- **Activity 2.3: Install MySQL Workbench**
  - Download and install MySQL Workbench on your local machine for database management.

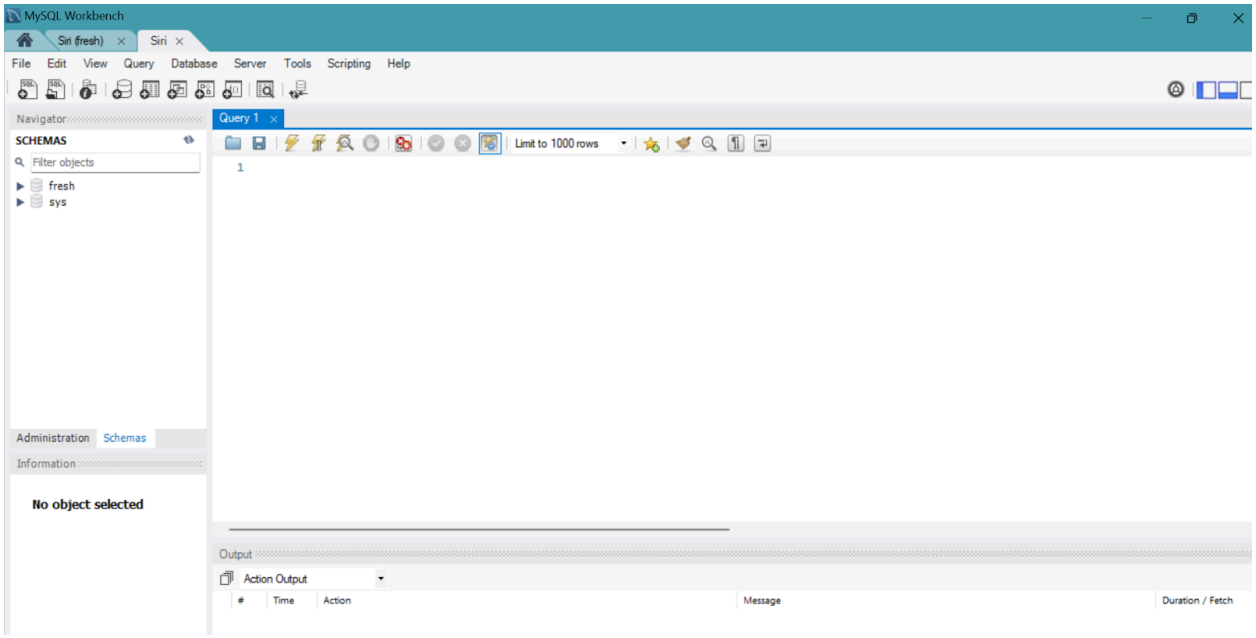




- Connect to the RDS instance via MySQL Workbench using the endpoint and credentials from AWS



- Give a connection name.
- Copy the endpoint from the RDS database that is created in AWS and paste it in **Hostname.**
- Write the username and enter the password , then click on **Test Connection**.
- Once the connection is successful, you'll be welcomed with this interface



Activity 2.4: Create the Database and the tables which are required.

- **Create a basic database schema for an AI Marketing platform**

```
Limit to 1000 rows
1 • create database marketing;
2 • use marketing;
```

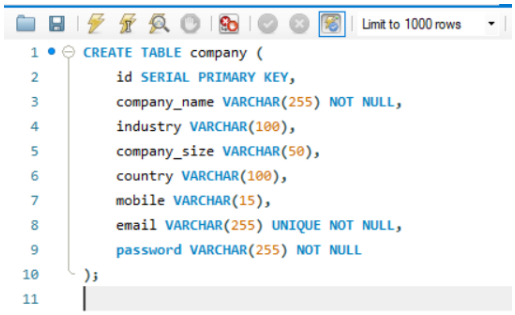
Tables Created :

1. **`Company`**:

- Stores company's information such as `company_name`, `industry`, `company_size`, `country`, `mobile`, `email`, `password`.
- Each user has a unique ID (`id`), which is the primary key.

Columns:

- `id` (Primary Key)
- `company_name`, `industry`, `company_size`, `country`, `mobile`, `email`, `password`.

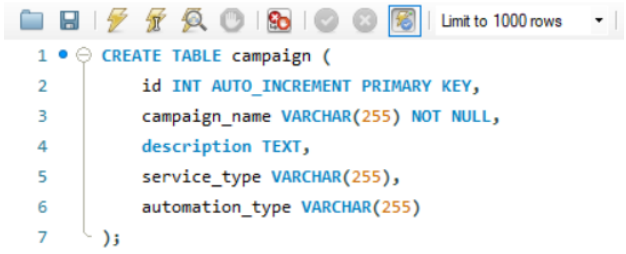


2. **`campaign`**:

- Contains available products with fields like `campaign_name`, `Description`, `Service_type` and `Automation_type`.
- Each item has a unique identifier `id` and the auto-increment starts from 01

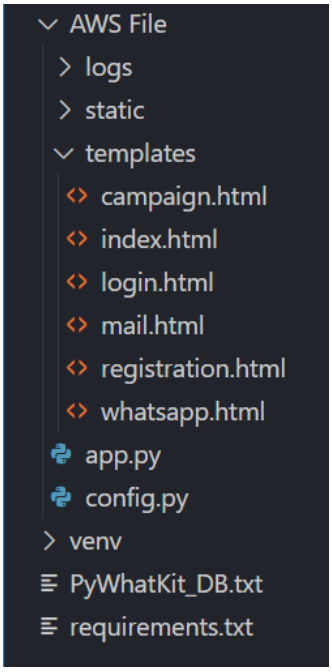
Columns:

- `id` (Primary Key)
- `Campaign_name, description, service_type, automation_type`



Milestone 3: Frontend Development and Application Setup

- **Activity 3.1: Build the Frontend**
  - Develop HTML, CSS, and Python-based Flask application files for Personalised Marketing's frontend interface.



- **Activity 3.2: Integrate Application with RDS**
  - Connect `app.py` (Flask application) to the MySQL RDS database by configuring database connection settings and verifying connectivity.

Description of the code :

1. **Flask App Initialization**: Initializes a Flask application with secret key for sessions.

```python
from flask import Flask, render_template, request, redirect, url_for, flash, session, jsonify
import mysql.connector
from mysql.connector import Error,pooling
from werkzeug.security import generate_password_hash, check_password_hash
import pywhatkit as kit
import pyautogui
import threading
import time
import os

app = Flask(__name__)
app.secret_key = "8019356963"  # Use a secure, random key for production
```

2. **Database Configuration**: Configures MySQL RDS with connection pooling for efficient database access.

```python
DB_CONFIG = {
            'host':'marketing.cbcikkyosovq.us-east-1.rds.amazonaws.com',  # Replace with your MySQL Workbench host
            'user':'root',  # Replace with your MySQL username
            'password':'abdulrmohammed',  # Replace with your MySQL password
            'database':'ai_marketing'
}
```

3. **Connection Pool**: Uses MySQL connection pooling to handle multiple database connections.

```
# MySQL connection configuration
connection_pool = pooling.MySQLConnectionPool(
    pool_name="mypool",
    pool_size=5,  # Define the number of connections in the pool
    **DB_CONFIG
)
def create_connection():
    try:
        connection = connection_pool.get_connection()  # Get a connection from the pool
        return connection
    except mysql.connector.Error as e:
        print(f"Error: {e}")
        return None
```

4. **Home Route**: Renders the home page template when the root URL is accessed.
5. **Register Route (GET/POST)**: Handles user registration, inserts user data into the database.

```
@app.route('/')
def home():
    return render_template('index.html')

# Registration route
@app.route('/registration', methods=['GET', 'POST'])
def registration():
    if request.method == 'POST':
        # Get form data
        company_name = request.form['company_name']
        industry = request.form['industry']
        company_size = request.form['company_size']
        country = request.form['country']
        mobile = request.form['phone']
        email = request.form['email']
        password = request.form['password']
        confirm_password = request.form['confirm_password']

        # Validate password confirmation
        if password != confirm_password:
            flash('Passwords do not match. Please try again.', 'danger')
            return redirect(url_for('registration'))

        # Hash the password
        hashed_password = generate_password_hash(password, method='sha256')

        # Insert company data into the database
        connection = create_connection()
        if connection is None:
```

```
            flash("Database connection failed. Please try again.", "danger")
            return redirect(url_for('registration'))
        cursor = connection.cursor()

        try:
            cursor.execute(
                '''INSERT INTO company
                   (company_name, industry, company_size, country, mobile, email, password)
                   VALUES (%s, %s, %s, %s, %s, %s, %s)''',
                (company_name, industry, company_size, country, mobile, email, hashed_password)
            )
            connection.commit()
            flash('Registration successful! You can now log in.', 'success')
            return redirect(url_for('login'))
        except Error as e:
            flash(f"An error occurred: {e}", 'danger')
        finally:
            cursor.close()
            connection.close()

    return render_template('registration.html')
```

6. **Login Route (GET/POST)**: Authenticates user with email and password, creates session on success.

```
# Login route
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Get form data
        email = request.form['email']
        password = request.form['password']

        # Database connection
        connection = create_connection()
        if connection is None:
            flash("Database connection failed. Please try again.", "danger")
            return redirect(url_for('login'))

        cursor = connection.cursor(dictionary=True)

        try:
            # Check if the user exists
            cursor.execute('SELECT * FROM company WHERE email = %s', (email,))
            user = cursor.fetchone()

            if user and check_password_hash(user['password'], password):
                # Store session data
                session['logged_in'] = True
                session['user_id'] = user['id']
                session['user_email'] = user['email']

                flash('Login successful!', 'success')
                # Redirect to the campaign page after successful login
```

```
            return redirect(url_for('campaign'))
        else:
            flash('Login failed. Incorrect email or password.', 'danger')
    except mysql.connector.Error as err:
        flash(f"An error occurred: {err}", 'danger')
    finally:
        cursor.close()
        connection.close()

return render_template('login.html')
```

7. **Campaign Route**: This route renders the Campaign page. The page includes details such as the campaign name, description, service type, and automation type.
8. After that Next button redirect to the whatsapp.html page.

```
@app.route('/campaign', methods=['GET', 'POST'])
def campaign():
    if request.method == 'POST':
        # Get form data
        campaign_name = request.form['campaign-name']
        description = request.form['description']
        service_type = request.form['service-type']  # Free or Paid
        automation_type = request.form['automation-type']  # WhatsApp only

        # Database connection
        connection = create_connection()
        if connection is None:
            flash("Database connection failed. Please try again.", "danger")
            return redirect(url_for('campaign'))  # Redirect to campaign page if connection fails

        cursor = connection.cursor()

        try:
            # Insert campaign data into the database
            cursor.execute(
                '''INSERT INTO campaign
                    (campaign_name, description, service_type, automation_type)
                    VALUES (%s, %s, %s, %s)''',
                (campaign_name, description, service_type, automation_type)
            )
            connection.commit()
            flash('Campaign created successfully!', 'success')
        except Error as e:
```

```
            flash(f"An error occurred: {e}", 'danger')
            return redirect(url_for('campaign'))  # Stay on the campaign page if there's an error
        finally:
            # Ensure cursor and connection are closed properly
            if cursor:
                cursor.close()
            if connection:
                connection.close()

        # Redirect based on WhatsApp service type
        if automation_type == 'whatsapp':
            if service_type == 'free':
                return redirect(url_for('whatsapp_automation', service='free'))
            elif service_type == 'paid':
                return redirect(url_for('whatsapp_automation', service='paid'))
        else:
            return "Invalid Automation Type"

    return render_template('campaign.html')
```

9. **Whatsapp message scheduling**:The WhatsApp Free page at `http://127.0.0.1:5000/whatsapp/free` allows users to send free WhatsApp messages easily by inputting a recipient's phone number and a custom message, with immediate feedback on message status..

```
# Global variables to keep track of tasks and threading
task_counter = 0
scheduled_tasks = {}
task_threads = {}
stop_events = {}

@app.route('/whatsapp/<service>', methods=['GET', 'POST'])
def whatsapp_automation(service):
    if service == 'free':
        limit = 5  # Free users can only send messages to 5 members
        flash(f"As a free user, you can only send messages to {limit} members.", 'info')

    if service == 'paid':
        flash("As a paid user, you can send messages to an unlimited number of members.", 'info')

    if request.method == 'POST':
        # Get the form data from the WhatsApp messaging form
        message = request.form['message']
        recipients = [recipient.strip() for recipient in request.form['recipients'].split(',')]  # Split and strip whitespace

        # If free, restrict the number of recipients
        if service == 'free' and len(recipients) > limit:
            flash("You can only send messages to 5 members as a free user.", "danger")
            return redirect(url_for('whatsapp_automation', service='free'))
```

```
    # Increment task counter for unique ID
    global task_counter
    task_counter += 1
    task_id = task_counter
    scheduled_tasks[task_id] = {"numbers": recipients, "message": message, "status": "pending"}

    # Create a stop event for this task
    stop_event = threading.Event()
    stop_events[task_id] = stop_event

    # Schedule message sending
    thread = threading.Thread(target=lambda: send_whatsapp_messages(recipients, message, task_id, stop_event))
    thread.start()
    task_threads[task_id] = thread  # Store the thread for potential cancellation

    flash('Your message is being sent!', 'success')
    return redirect(url_for('campaign'))  # Redirect back to the campaign page after submission

# Render the appropriate template based on the service
if service == 'free':
    return render_template('whatsapp.html', service=service)
elif service == 'paid':
    return render_template('whatsapp.html', service=service)
```

```
def send_whatsapp_messages(numbers, message, task_id, stop_event):
    global scheduled_tasks
    try:
        # Open WhatsApp and send the first message
        kit.sendwhatmsg_instantly(numbers[0], message, 15)  # Opens WhatsApp and sends the first message
        time.sleep(5)  # Wait for the first message to be sent

        # Use pyautogui to send messages to the other numbers
        for number in numbers[1:]:
            if stop_event.is_set():  # Stop sending if the event is set
                scheduled_tasks[task_id]['status'] = 'cancelled'
                return

            # Click on the search bar using pyautogui
            pyautogui.click(x=553, y=171)  # Adjust the x and y coordinates to the search bar position
            time.sleep(1)  # Wait for a moment to ensure the click is registered

            # Type the number in the search bar
            pyautogui.typewrite(number)  # Type the contact number
            time.sleep(1)  # Wait for the contact to be found

            # Press 'Enter' to select the contact
            pyautogui.press('enter')
            time.sleep(5)  # Wait for the chat to open

            # Type and send the message
            pyautogui.typewrite(message)  # Type the message
            pyautogui.press('enter')  # Send the message
```

```
            time.sleep(1)  # A small delay between sending each message

        # Notify the front-end that the message was sent
        scheduled_tasks[task_id]['status'] = 'sent'
    except Exception as e:
        print(f"Error sending messages: {e}")
        scheduled_tasks[task_id]['status'] = 'failed'  # Update task status in case of failure
```

10. **Scheduling the message:** The Message Scheduling feature allows users to automate WhatsApp messages for future delivery by selecting a specific date and time, ensuring timely communication without the need for manual sending.

```
@app.route('/schedule', methods=['POST'])
def schedule_message():
    global task_counter
    data = request.get_json()
    numbers = data['numbers']
    message = data['message']

    # Increment the task counter for unique ID
    task_counter += 1
    task_id = task_counter

    # Get the scheduled time
    hour, minute = map(int, data['time'].split(':'))
    now = time.localtime()
    send_time = time.mktime((now.tm_year, now.tm_mon, now.tm_mday, hour, minute, 0, 0, 0, -1))

    delay = send_time - time.time()

    if delay < 0:
        return jsonify({"status": "Please set a time in the future."}), 400

    # Schedule the message sending
    scheduled_tasks[task_id] = {"numbers": numbers, "message": message, "status": "pending"}

    # Create a stop event for this task
    stop_event = threading.Event()
    stop_events[task_id] = stop_event
```

```
    # Start a thread to wait for the delay and then send the messages
    thread = threading.Thread(target=lambda: wait_and_send(numbers, message, task_id, delay, stop_event))
    thread.start()
    task_threads[task_id] = thread  # Store thread to manage cancellation

    return jsonify({"status": "Message scheduled.", "task_id": task_id})

def wait_and_send(numbers, message, task_id, delay, stop_event):
    start_time = time.time()
    while time.time() - start_time < delay:
        if stop_event.is_set():  # Immediately stop if the event is set
            scheduled_tasks[task_id]['status'] = 'cancelled'
            return
        time.sleep(1)  # Check every second

    send_whatsapp_messages(numbers, message, task_id, stop_event)
```

11. **Cancel Message scheduling:** The Cancel Message feature allows users to withdraw or delete scheduled WhatsApp messages before they are sent, providing flexibility and control over their communication plans.

```
@app.route('/stop', methods=['POST'])
def stop_all_messages():
    global scheduled_tasks

    # Signal all running threads to stop
    for task_id, stop_event in stop_events.items():
        stop_event.set()  # Signal all threads to stop

    # Wait for all threads to complete
    for task_id, thread in task_threads.items():
        thread.join()

    # Clear all threads and stop events
    task_threads.clear()
    stop_events.clear()
    scheduled_tasks.clear()  # Clear all scheduled tasks

    return jsonify({"status": "All scheduled messages stopped."})
```

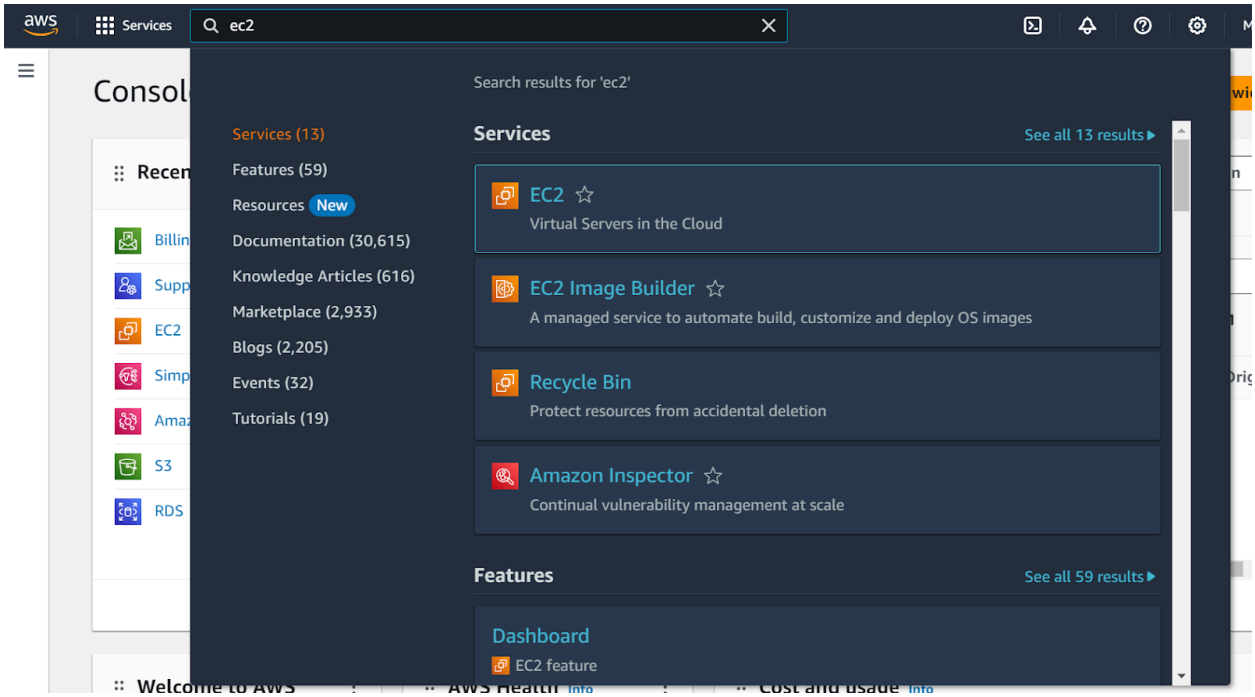
```python
@app.route('/events')
def events():
    # Use a simple event-streaming method to notify when messages are sent
    def generate():
        while True:
            time.sleep(1)  # Check every second
            for task_id, task in list(scheduled_tasks.items()):
                if task['status'] == 'sent':
                    yield f"data: {{\"status\": \"message_sent\"}}\n\n"
                    del scheduled_tasks[task_id]  # Remove task once notified

    return app.response_class(generate(), mimetype='text/event-stream')# @app.route('/email', methods=['GET', 'POST'])
#
@app.route('/logout')
def logout():
    session.clear()
    flash('You have been logged out!', 'success')
    return redirect(url_for('home'))

if __name__ == '__main__':
    print('flask is running')
    app.run(host='0.0.0.0'  , port=5000, debug=True)
```

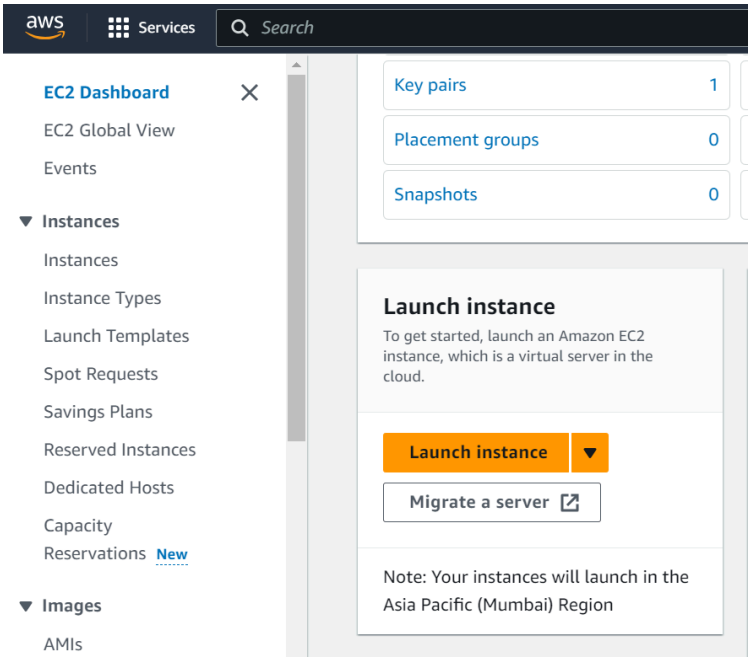
12. **Stop messages :** The Stop Messages feature enables users to halt the delivery of ongoing WhatsApp messages, ensuring they can interrupt or cancel any messages in the process of being sent as needed.

13. **Event Management**: The **/events** page allows users to view, manage, and monitor all scheduled WhatsApp messages in one centralized location, offering options to edit or delete events as needed.

Milestone 4: EC2 Instance Setup

- **Activity 4.1: Launch EC2 Instance**
    - Choose a Linux-based EC2 instance from the AWS Console to host the AI-Driven personalized Marketing application.


aws
Services
ec2
Search results for 'ec2'
Services (13)
Features (59)
Resources New
Documentation (30,615)
Knowledge Articles (616)
Marketplace (2,933)
Blogs (2,205)
Events (32)
Tutorials (19)
Services
See all 13 results
EC2
Virtual Servers in the Cloud
EC2 Image Builder
A managed service to automate build, customize and deploy OS images
Recycle Bin
Protect resources from accidental deletion
Amazon Inspector
Continual vulnerability management at scale
Features
See all 59 results
Dashboard
EC2 feature


aws
Services
Search
EC2 Dashboard
EC2 Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations New
Images
AMIs
Key pairs 1
Placement groups 0
Snapshots 0
Launch instance
To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.
Launch instance
Migrate a server
Note: Your instances will launch in the Asia Pacific (Mumbai) Region

SMARTBRIDGE
Let's Bridge the Gap

Smart Internz

Launch an instance Info

Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.

Name and tags Info

Name

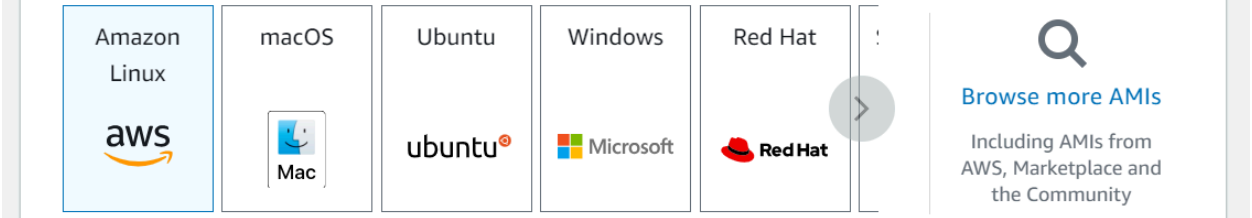
AI-Marketing

Add additional tags

▼ Application and OS Images (Amazon Machine Image) Info

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Search our full catalog including 1000s of application and OS images



Amazon Machine Image (AMI)

Amazon Linux 2023 AMI — Free tier eligible
ami-02b49a24cfb95941c (64-bit (x86), uefi-preferred) / ami-04ad8c7fcc828fad4 (64-bit (Arm), uefi)
Virtualization: hvm ENA enabled: true Root device type: ebs

Description

Amazon Linux 2023 is a modern, general purpose Linux-based OS that comes with 5 years of long term support. It is optimized for AWS and designed to provide a secure, stable and high-performance execution environment to develop and run your cloud applications.

Architecture	Boot mode	AMI ID	
64-bit (x86)	uefi-preferred	ami-02b49a24cfb95941c	Verified provider

Instance type Info | Get advice

Instance type

t2.micro — Free tier eligible
Family: t2 1 vCPU 1 GiB Memory Current generation: true
On-Demand Linux base pricing: 0.0124 USD per Hour
On-Demand Windows base pricing: 0.017 USD per Hour
On-Demand RHEL base pricing: 0.0268 USD per Hour
On-Demand SUSE base pricing: 0.0124 USD per Hour

All generations

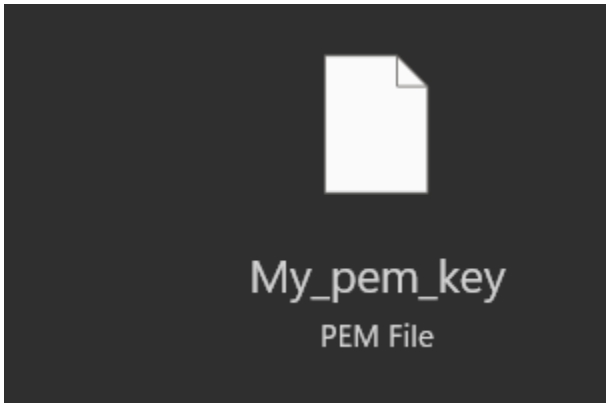
Compare instance types

Additional costs apply for AMIs with pre-installed software

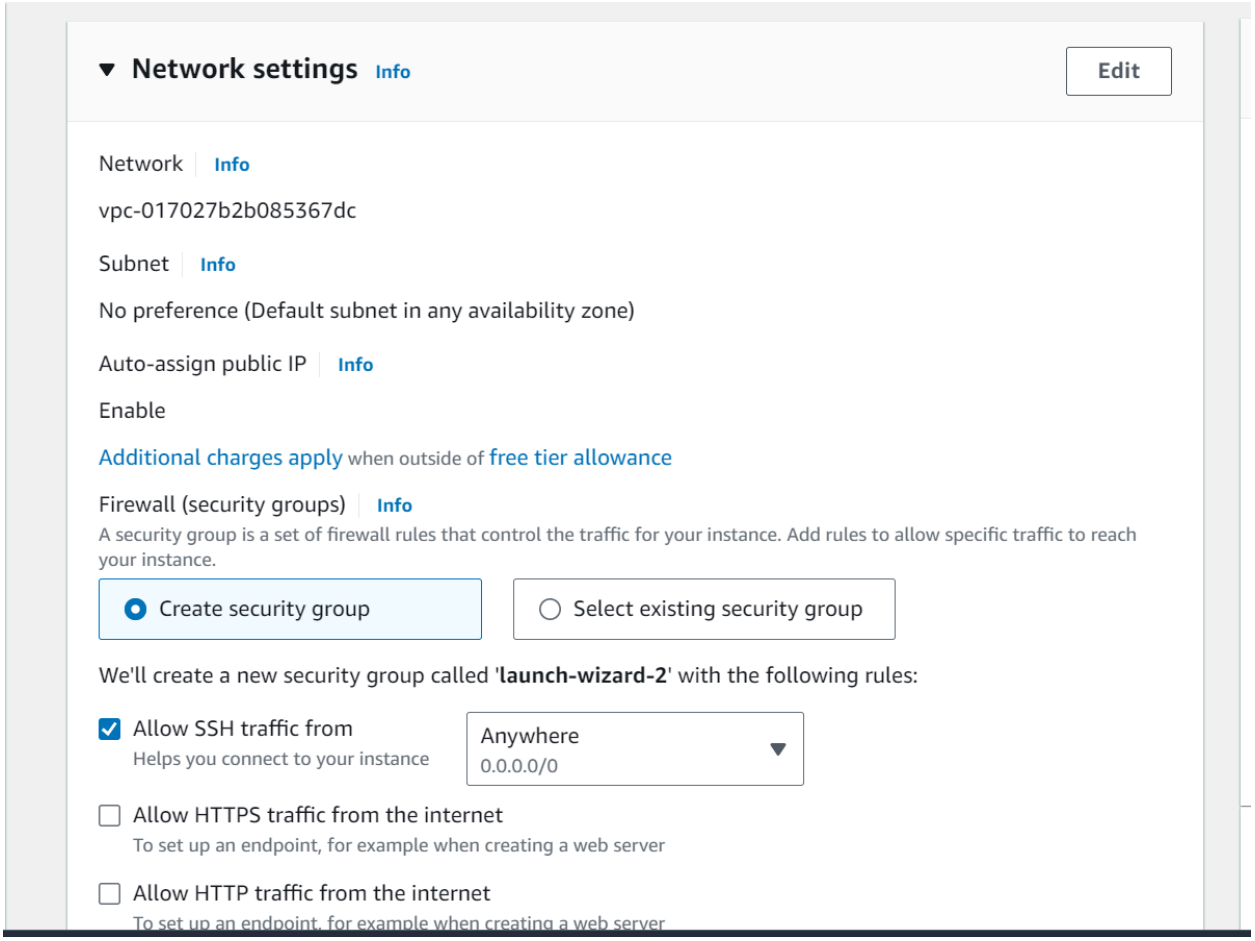
Key pair (login) Info

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

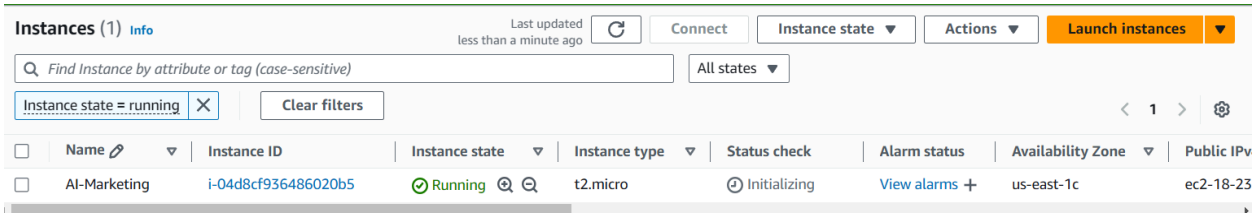
Key pair name - *required*



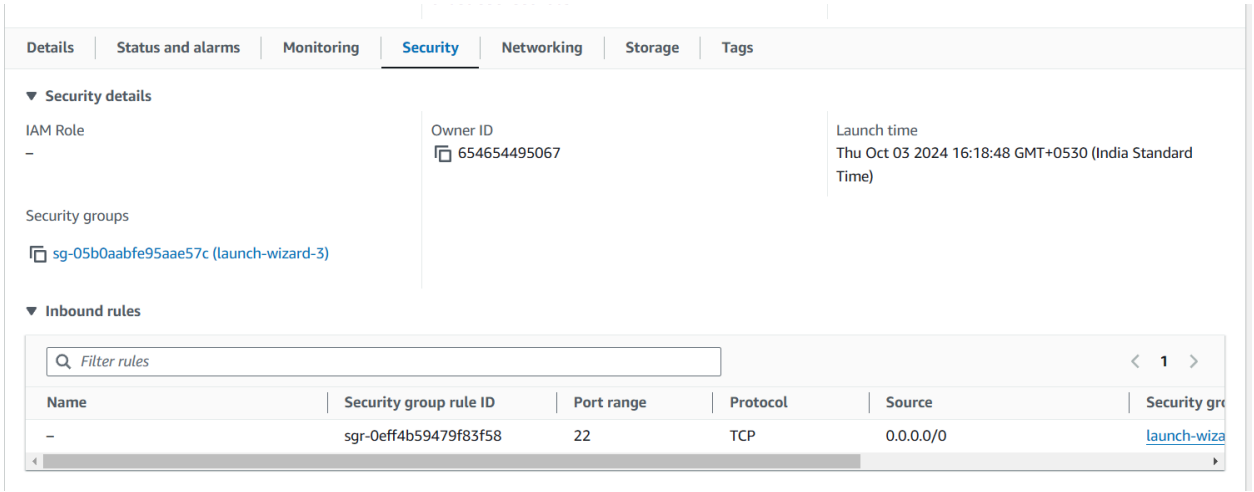
- **Activity 4.2: Configure Network Settings**
  - Set up the security group to allow HTTP, HTTPS, and SSH traffic.

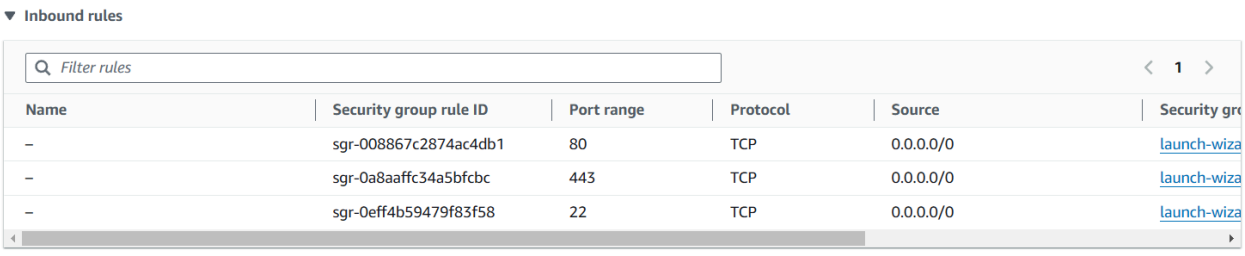


  - Create and download the key pair for SSH access.



Setting up Inbound and Outbound rules

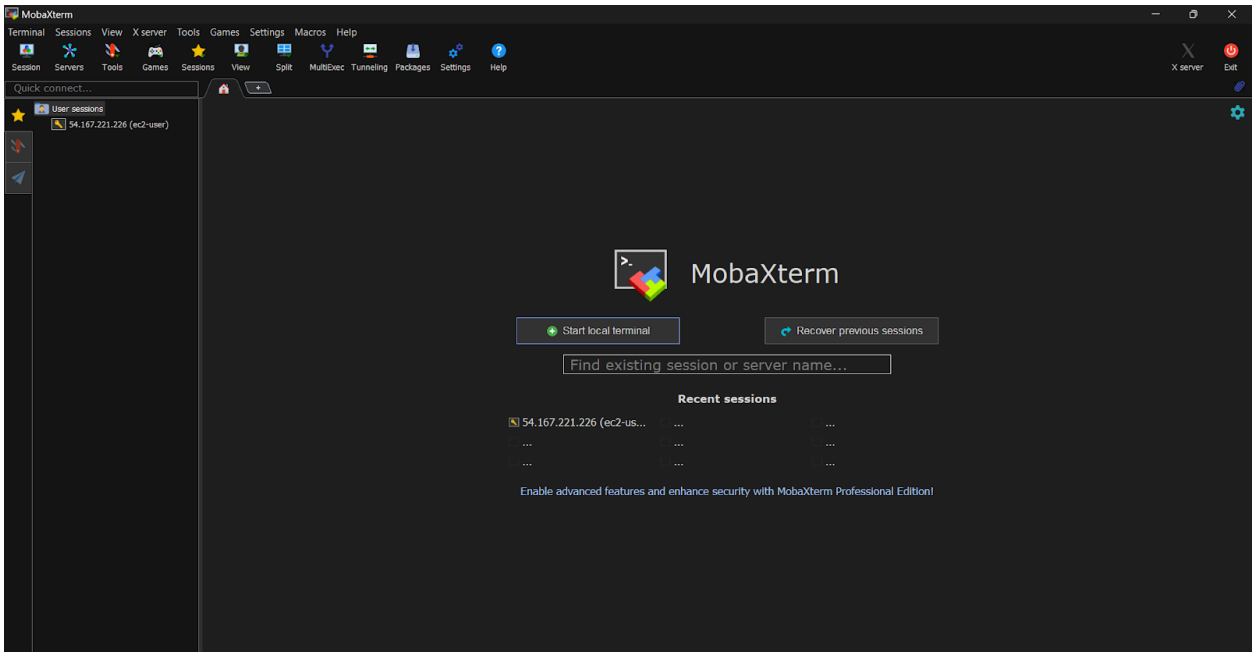


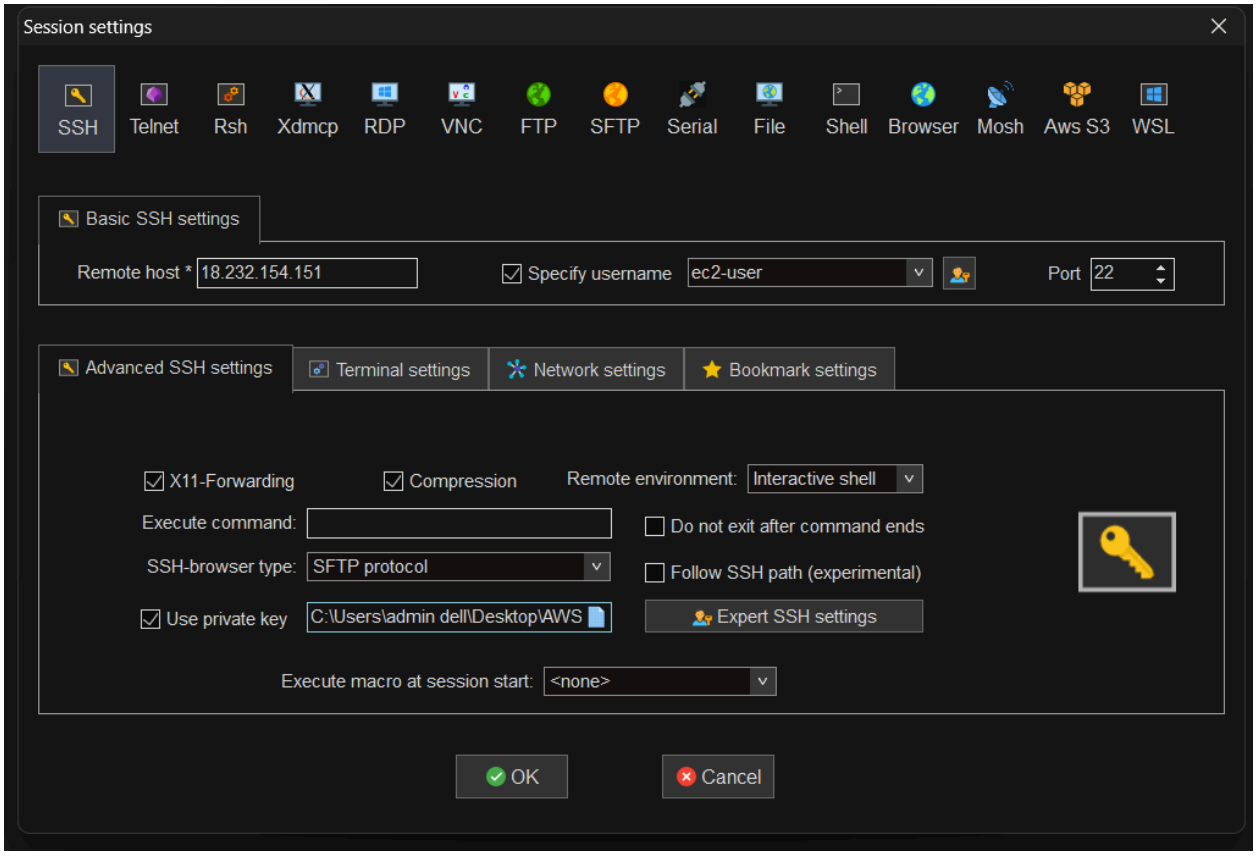


Milestone 5: MobaXterm Setup and SSH Access

- **Activity 5.1: Install and Configure MobaXterm**

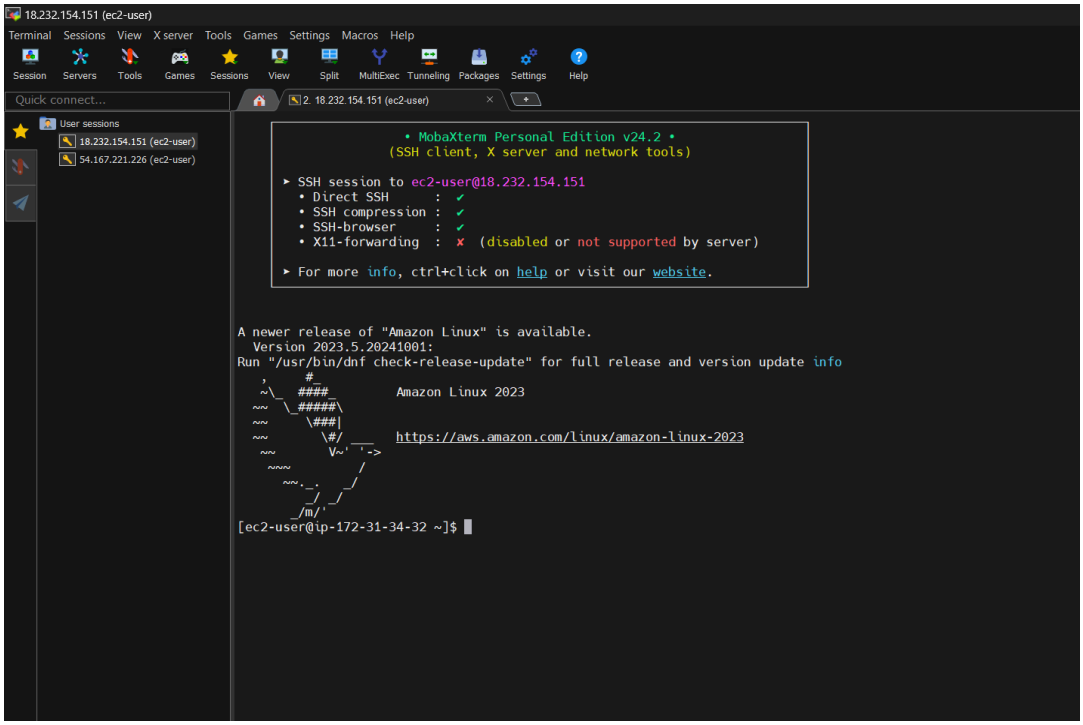
- Download and install MobaXterm on your local machine.
- Establish an SSH session with the EC2 instance by uploading the `.pem` key file.





Activity 5.2: Login into the SSH Session

- Login as : ec2-user



Update and Install Web Server Software

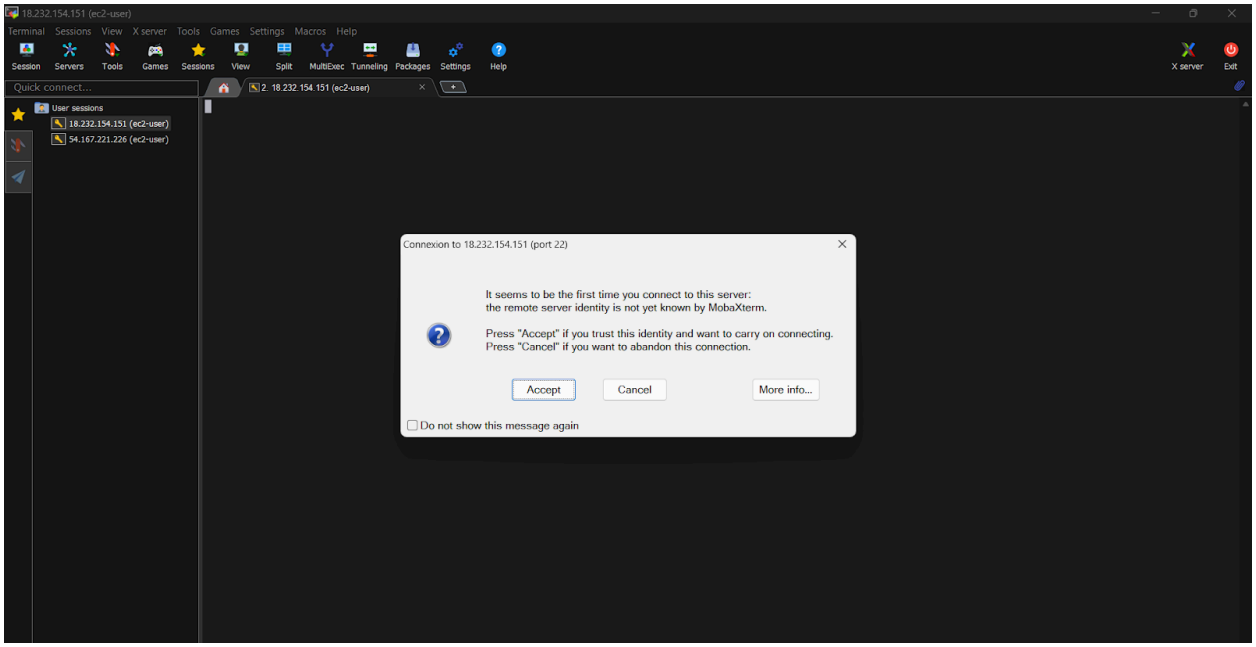
- Update package lists with `sudo apt-get update` (Ubuntu) or `sudo yum update` (Amazon Linux).

Install Apache or Nginx:

- For Apache: `sudo apt-get install apache2` (Ubuntu) or `sudo yum install httpd` (Amazon Linux).
- For Nginx: `sudo apt-get install nginx` (Ubuntu) or `sudo yum install nginx` (Amazon Linux).

```
    https://docs.aws.amazon.com/linux/al2023/release-notes/relnotes-2023.5.20241001.html

================================================================================================================================================
Dependencies resolved.
Nothing to do.
Complete!
[root@ip-172-31-34-32 ec2-user]# yum update httpd -y
Last metadata expiration check: 0:05:14 ago on Thu Oct  3 10:49:21 2024.
Package httpd available, but not installed.
No match for argument: httpd
Error: No packages marked for upgrade.
[root@ip-172-31-34-32 ec2-user]# yum install httpd -y
Last metadata expiration check: 0:05:30 ago on Thu Oct  3 10:49:21 2024.
Dependencies resolved.
================================================================================================================================================
 Package                                Architecture                Version                                  Repository                     Size
================================================================================================================================================
Installing:
 httpd                                  x86_64                      2.4.62-1.amzn2023                        amazonlinux                    48 k
Installing dependencies:
 apr                                    x86_64                      1.7.2-2.amzn2023.0.2                     amazonlinux                   129 k
 apr-util                               x86_64                      1.6.3-1.amzn2023.0.1                     amazonlinux                    98 k
 generic-logos-httpd                    noarch                      18.0.0-12.amzn2023.0.3                   amazonlinux                    19 k
 httpd-core                             x86_64                      2.4.62-1.amzn2023                        amazonlinux                   1.4 M
 httpd-filesystem                       noarch                      2.4.62-1.amzn2023                        amazonlinux                    14 k
 httpd-tools                            x86_64                      2.4.62-1.amzn2023                        amazonlinux                    81 k
 libbrotli                              x86_64                      1.0.9-4.amzn2023.0.2                     amazonlinux                   315 k
 mailcap                                noarch                      2.1.49-3.amzn2023.0.3                    amazonlinux                    33 k
Installing weak dependencies:
 apr-util-openssl                       x86_64                      1.6.3-1.amzn2023.0.1                     amazonlinux                    17 k
 mod_http2                              x86_64                      2.0.27-1.amzn2023.0.3                    amazonlinux                   166 k
 mod_lua                                x86_64                      2.4.62-1.amzn2023                        amazonlinux                    61 k

Transaction Summary
================================================================================================================================================
Install  12 Packages

Total download size: 2.3 M
Installed size: 6.9 M
Downloading Packages:
(1/12): apr-util-openssl-1.6.3-1.amzn2023.0.1.x86_64.rpm                                                 314 kB/s |  17 kB     00:00
```

Upload Website Files Using MobaXterm



- Use MobaXterm's SFTP functionality to transfer website files to the EC2 instance.
- Navigate to the `/var/www/html` directory (or the relevant directory for Nginx) and upload the

files.

```
Complete!
[root@ip-172-31-34-32 ec2-user]# cd /var/www/html/
[root@ip-172-31-34-32 html]# pwd
/var/www/html
[root@ip-172-31-34-32 html]#
```

```
Installed:
  apr-1.7.2-2.amzn2023.0.2.x86_64                    apr-util-1.6.3-1.amzn2023.0.1.x86_64          apr-util-openssl-1.6.3-1.amzn2023.0.1.x86_64
  generic-logos-httpd-18.0.0-12.amzn2023.0.3.noarch  httpd-2.4.62-1.amzn2023.x86_64                httpd-core-2.4.62-1.amzn2023.x86_64
  httpd-filesystem-2.4.62-1.amzn2023.noarch          httpd-tools-2.4.62-1.amzn2023.x86_64          libbrotli-1.0.9-4.amzn2023.0.2.x86_64
  mailcap-2.1.49-3.amzn2023.0.3.noarch               mod_http2-2.0.27-1.amzn2023.0.3.x86_64        mod_lua-2.4.62-1.amzn2023.x86_64

Complete!
[root@ip-172-31-34-32 ec2-user]# cd /var/www/html/
[root@ip-172-31-34-32 html]# pwd
/var/www/html
[root@ip-172-31-34-32 html]# yum install git
Last metadata expiration check: 0:06:35 ago on Thu Oct  3 10:49:21 2024.
Dependencies resolved.
================================================================================================================================================
 Package                          Architecture              Version                                    Repository                          S
================================================================================================================================================
Installing:
 git                              x86_64                    2.40.1-1.amzn2023.0.3                      amazonlinux                         5
Installing dependencies:
 git-core                         x86_64                    2.40.1-1.amzn2023.0.3                      amazonlinux                       4.
 git-core-doc                     noarch                    2.40.1-1.amzn2023.0.3                      amazonlinux                       2.
 perl-Error                       noarch                    1:0.17029-5.amzn2023.0.2                   amazonlinux                         4
 perl-File-Find                   noarch                    1.37-477.amzn2023.0.6                      amazonlinux                         2
 perl-Git                         noarch                    2.40.1-1.amzn2023.0.3                      amazonlinux                         4
 perl-TermReadKey                 x86_64                    2.38-9.amzn2023.0.2                        amazonlinux                         3
 perl-lib                         x86_64                    0.65-477.amzn2023.0.6                      amazonlinux                         1

Transaction Summary
================================================================================================================================================
Install  8 Packages
```

- **Activity 5.3 : Launch Flask Application**
    - Run the Flask app on the EC2 instance through the SSH session to serve the AI-Driven personalised Marketing frontend.

Milestone 6: Testing and Deployment

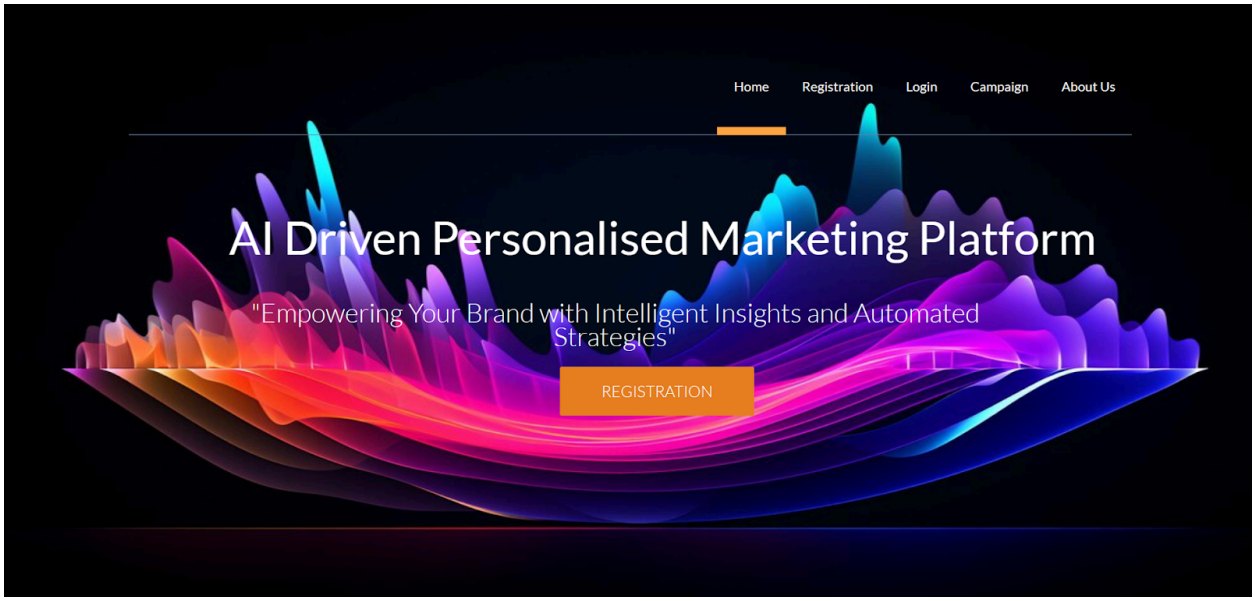
- **Activity 6.1:Testing and Deployment**
    - Test the AI-Driven Personalised Marketing Platform application for functionality, including database interactions and frontend features.
    - Run the Flask app **python3 app.py**
    - It will give you the link

```
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
 * Debugger is active!
 * Debugger PIN: 849-618-898
```

- **Activity 6.2: Deployment**
  - Deploy the application in a production environment, ensuring high availability and performance.

Click on the link above and it will take you to the webpage:

home:



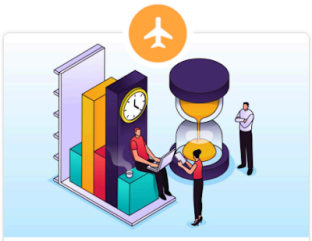
An AI-driven personalized marketing platform tailors campaigns using customer data, boosting engagement, conversions, and marketing efficiency in real-time.

Enhanced Customer Engagement

Tailored content and personalized offers resonate with customers, increasing engagement and satisfaction..

Improved Conversion Rates

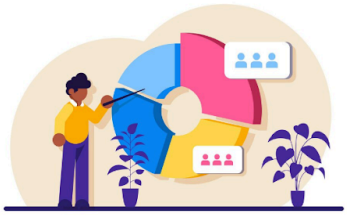
AI-driven insights help target high-potential prospects, optimizing campaigns for higher conversions and sales..



Real-Time Optimization

The platform adjusts strategies based on live data, ensuring campaigns remain effective and relevant as conditions change..

TOP FEATURES

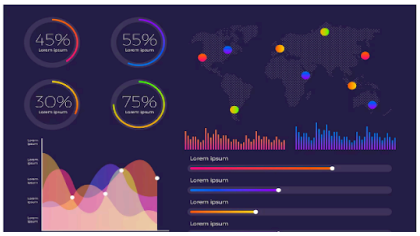


Automated Customer Segmentation

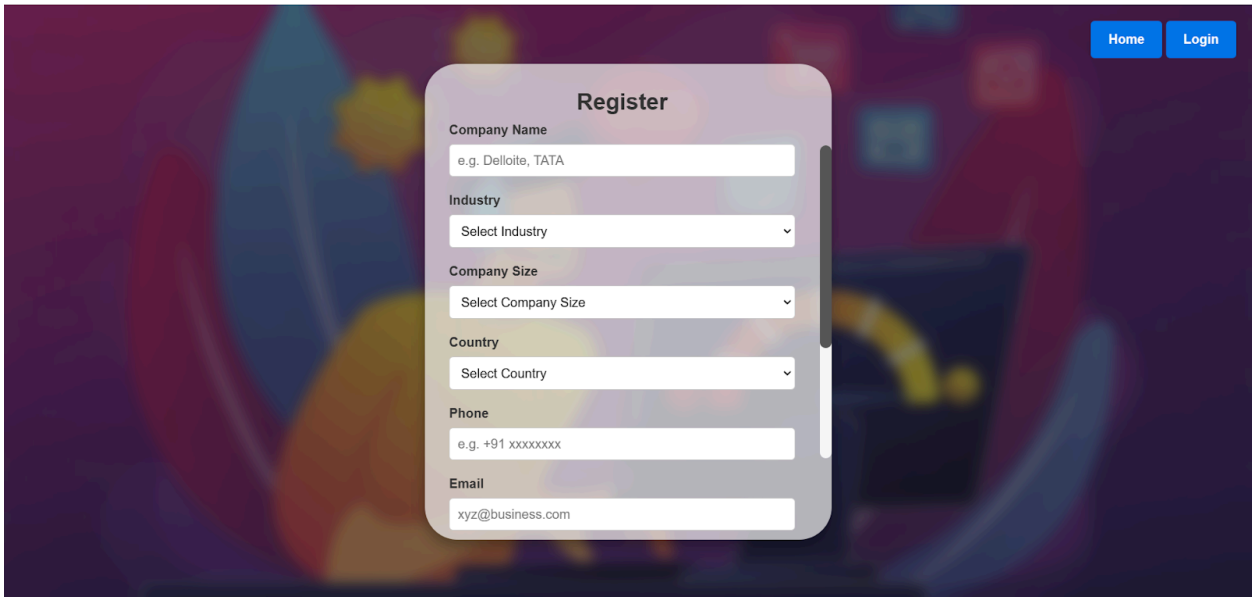
The platform uses AI to automatically segment customers based on various data points, such as demographics, behavior, purchasing patterns, and engagement levels. This intelligent segmentation allows businesses to create highly targeted marketing strategies for specific customer groups. AI-driven segmentation ensures that each group receives personalized offers and messages, improving relevance and engagement. By understanding each customer segment's unique needs and preferences, businesses can deliver more effective campaigns, resulting in higher customer satisfaction and increased conversion rates.

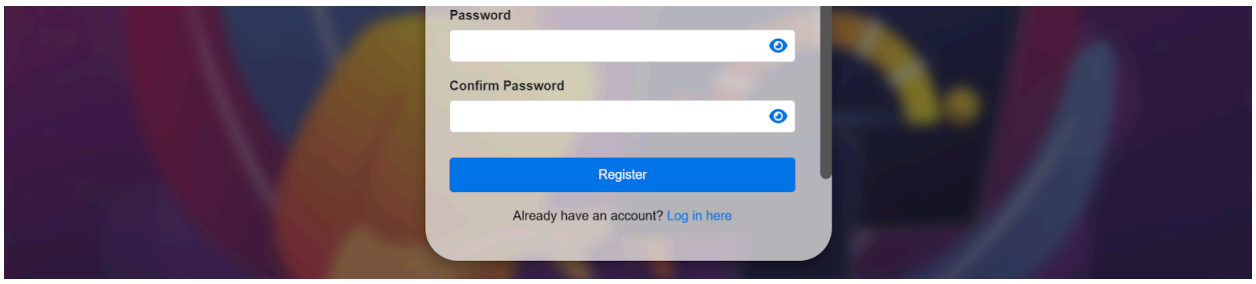
Predictive analytics

The AI-driven personalized marketing platform leverages predictive analytics to anticipate customer behavior, trends, and market shifts. By analyzing historical data and real-time user interactions, AI can forecast future actions such as purchase intent or churn risk. This enables businesses to make proactive, data-driven decisions, such as when to launch campaigns or target specific customer segments. Predictive insights help refine marketing strategies, optimize resource allocation, and increase the chances of delivering the right message at the right time, driving higher engagement and conversions.

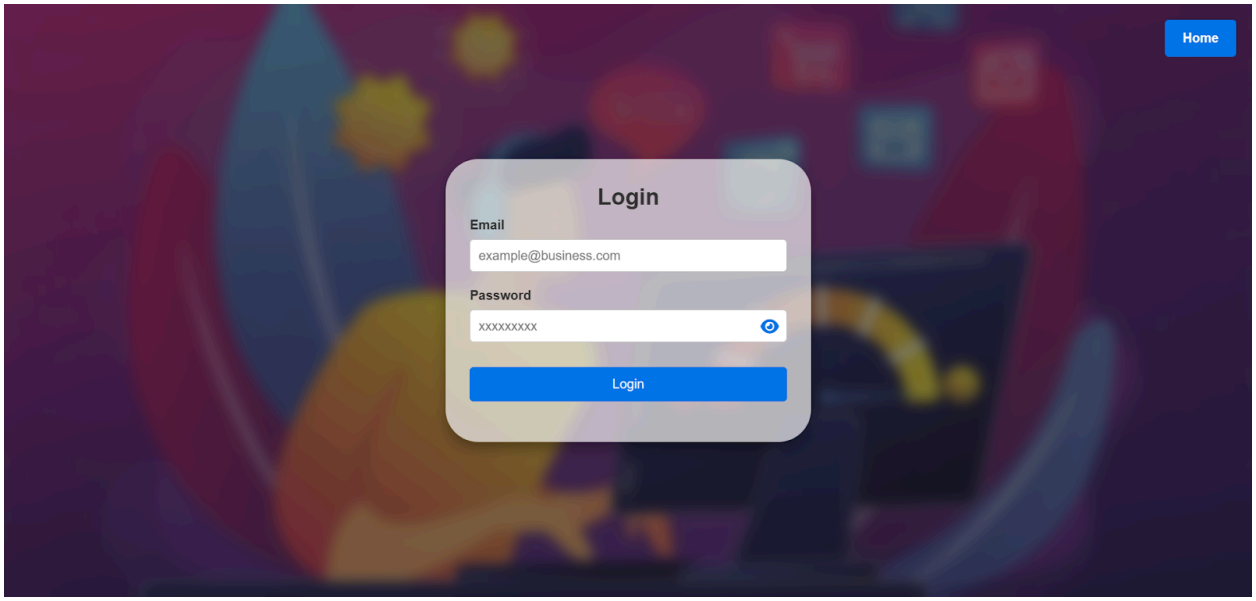


Register:

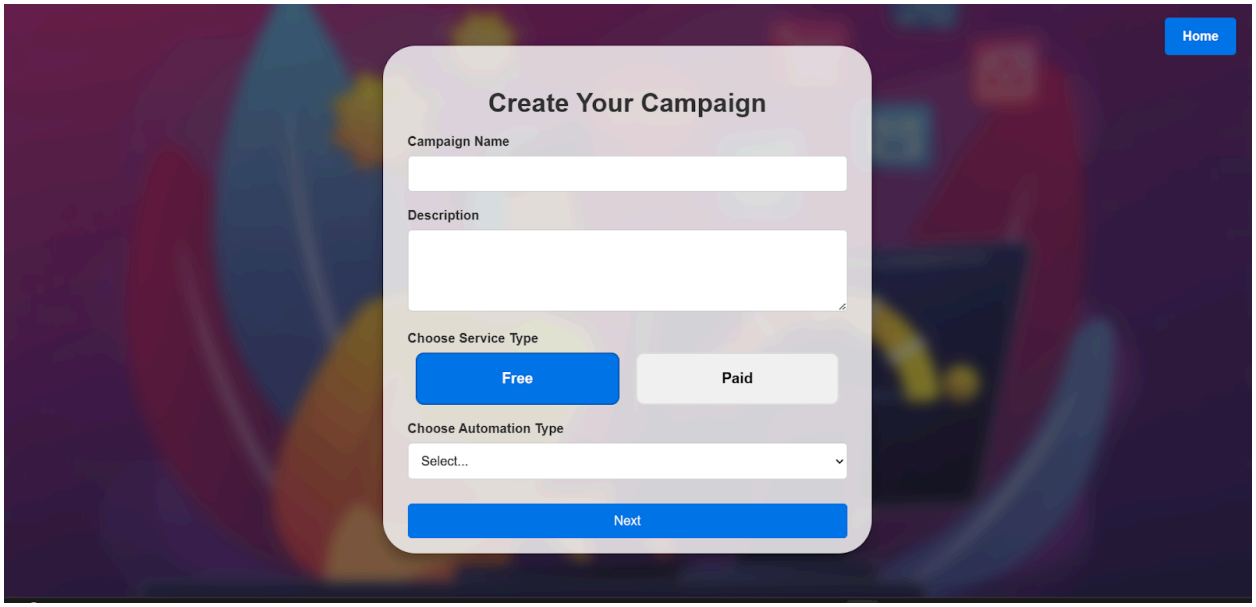

Home
Login
Register
Company Name
e.g. Delloite, TATA
Industry
Select Industry
Company Size
Select Company Size
Country
Select Country
Phone
e.g. +91 xxxxxxxx
Email
xyz@business.com

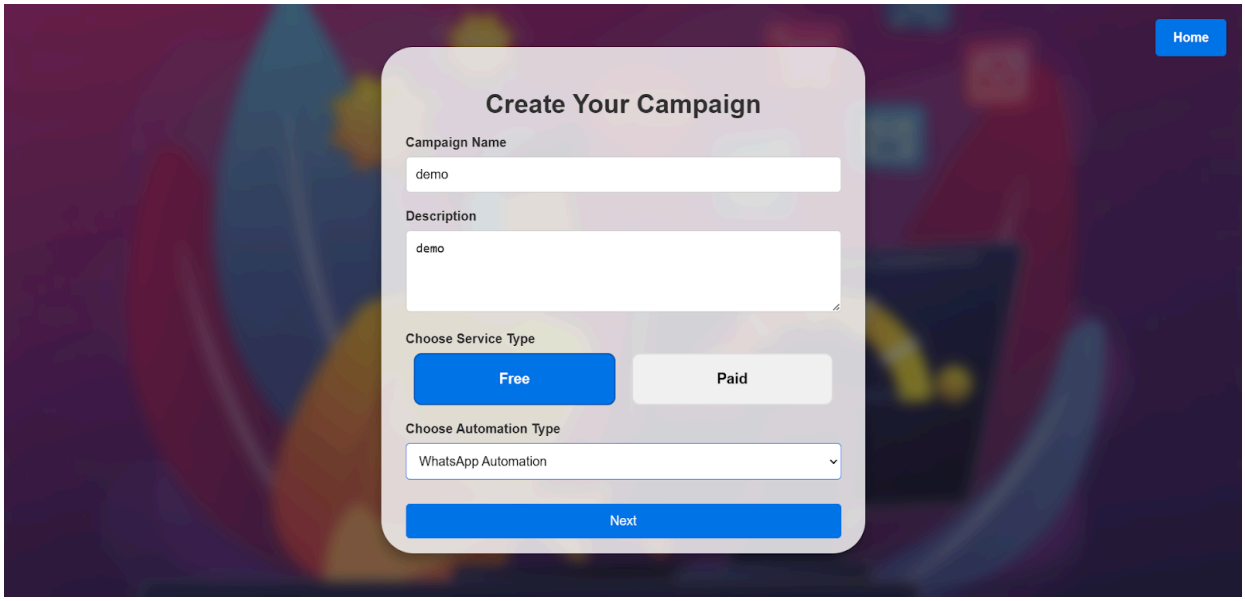

Password
Confirm Password
Register
Already have an account? Log in here

Login:

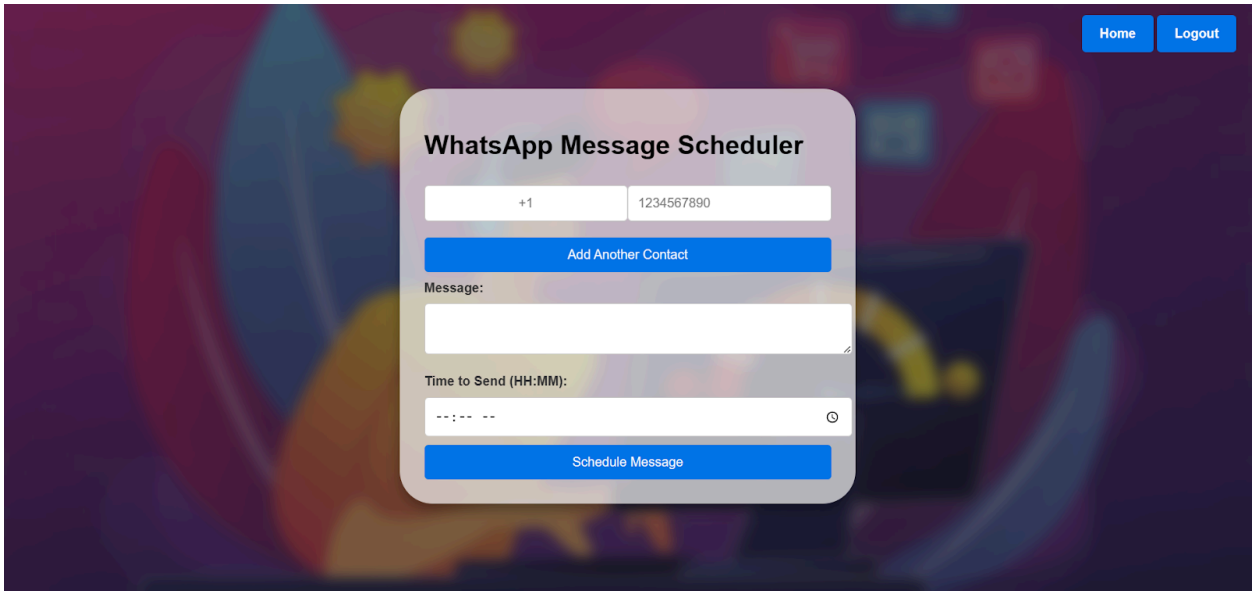


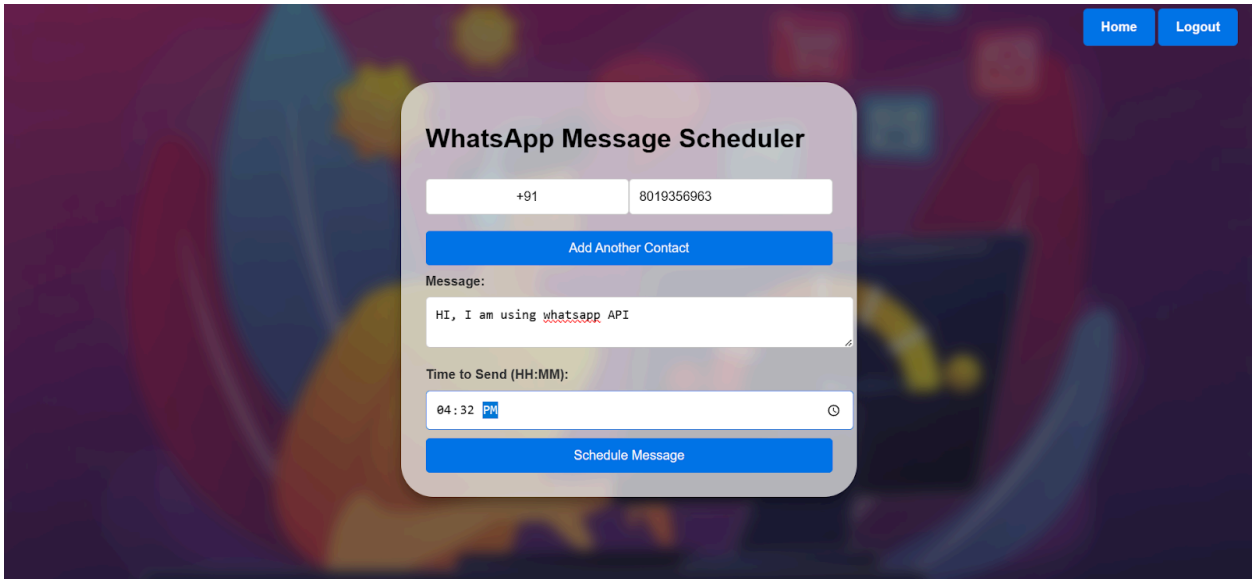
Campaign:



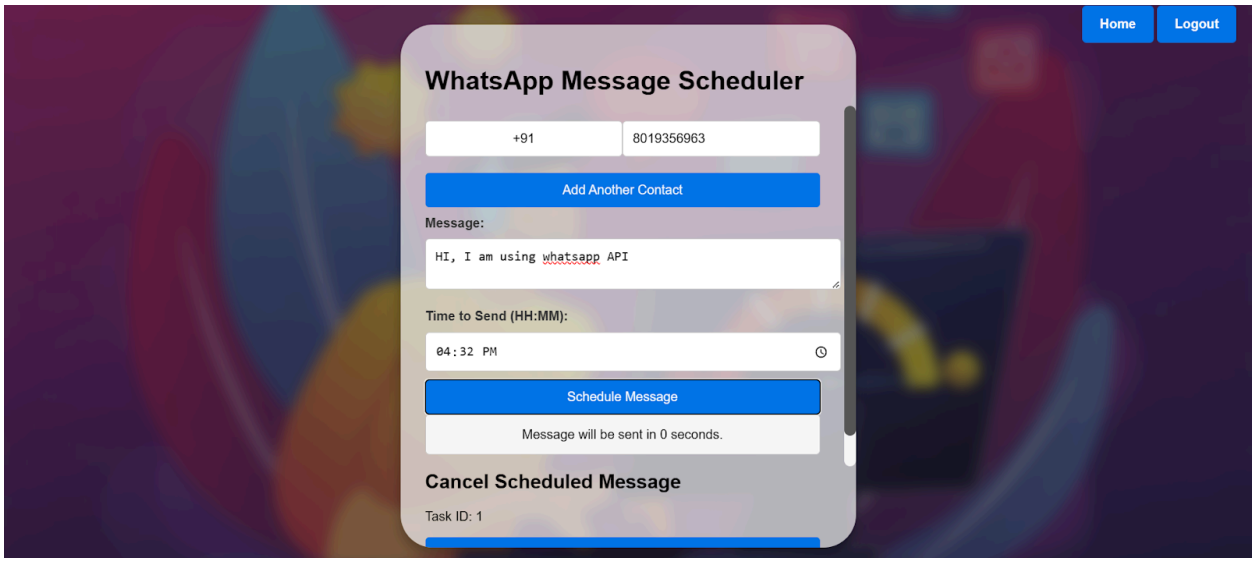


Whatsapp Message Scheduler(automation):





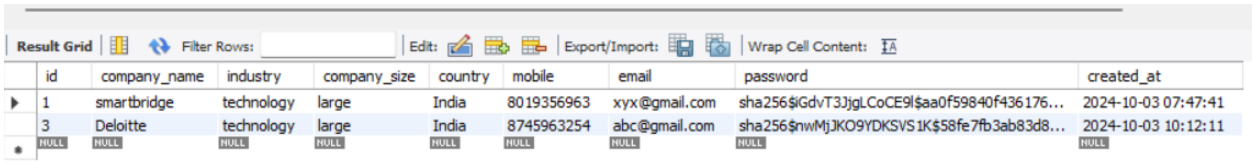
Message Sending Mechanism:



- **Activity 6.3: Check the updations in the mysql database.**

MySql Database updations :

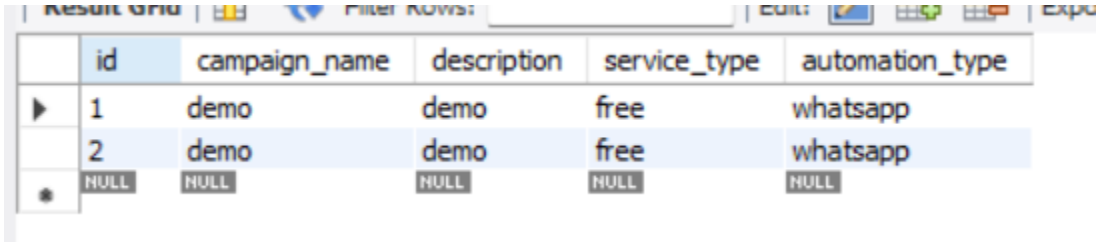
1. **Company table :**



Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

id	company_name	industry	company_size	country	mobile	email	password	created_at
1	smartbridge	technology	large	India	8019356963	xyx@gmail.com	sha256$iGdvT3JjgLCoCE9l$aa0f59840f436176...	2024-10-03 07:47:41
3	Deloitte	technology	large	India	8745963254	abc@gmail.com	sha256$nwMjJKO9YDKSVS1K$58fe7fb3ab83d8...	2024-10-03 10:12:11
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

2. **Campaign table :**



id	campaign_name	description	service_type	automation_type
1	demo	demo	free	whatsapp
2	demo	demo	free	whatsapp
NULL	NULL	NULL	NULL	NULL

Milestone 7: Monitoring and Optimization

- **Activity 7.1: Performance Monitoring**
  - Set up AWS CloudWatch for monitoring EC2 and RDS performance metrics.
  - Implement alerts and notifications for critical performance thresholds.
- **Activity 7.2: Optimization**
  - Optimize the server and database configurations based on monitoring results, including adjusting instance types and query optimization.

Conclusion:

The **AI-Driven Personalized Marketing Platform** project highlights the implementation of a robust and scalable marketing solution using AWS. By integrating Flask with MySQL RDS, it provides seamless data management and secure backend operations, ensuring efficient handling of user interactions and campaign data. The EC2 instance, paired with MobaXterm, facilitates smooth remote management and deployment of the application. From database configuration to frontend design and deployment, every step was meticulously executed to guarantee reliability and performance. AWS CloudWatch is utilized for performance monitoring, ensuring optimal functioning and responsiveness. Overall, the project exemplifies how AWS services can effectively support dynamic, scalable marketing applications in real-world business environments.