🧱 Step 1 — Create the S3 Bucket

✅ **Purpose:**

Store raw order data and cleaned output.

🪜 **Steps:**

1. Go to **S3 console → Create bucket**
    - Bucket name: orders-pipeline-vaishnavi (for example)
    - Region: us-east-1
2. Inside the bucket, create two folders:
    - raw/ → for incoming raw files
    - processed/ → for cleaned output files

📤 **Upload test CSV**

Upload orders_detailed.csv to the raw/ folder.

🧠 Step 2 — Create and Configure RDS (MySQL)

✅ **Purpose:**

Store cleaned order data from Lambda.

🪜 **Steps:**

1. Go to **RDS → Create Database**
    - Engine: **MySQL**
    - Template: Free tier
    - DB instance id: ordersdb
    - Master username: admin
    - Master password: (your chosen password)
    - Connectivity → VPC: same as your Lambda function
    - Enable public access: **Yes** (for EC2 connection)
2. Once created, note the **endpoint** from:
    - RDS → Databases → ordersdb → *Connectivity & security*

- Example: ordersdb.cg6hfudzyklp.us-east-1.rds.amazonaws.com

🧰 **Step 3 — Connect EC2 to RDS (for setup)**

✅ **Purpose:**

Use EC2 to create the database & table in RDS.

🪜 **Steps:**

1. Launch an **EC2 (Amazon Linux)** instance
2. Connect using MobaXterm or terminal:
3. ssh -i MyKey.pem ec2-user@<EC2-Public-IP>
4. Install MySQL client:
5. sudo yum install mariadb105 -y
6. Connect to RDS:
7. mysql -h <RDS-ENDPOINT> -u admin -p
8. Inside MySQL:
9. CREATE DATABASE ordersdb;
10. USE ordersdb;
11.
12. CREATE TABLE cleaned_orders (
13. order_id INT,
14. customer_name VARCHAR(100),
15. city VARCHAR(50),
16. product VARCHAR(100),
17. quantity INT,
18. price FLOAT,
19. channel VARCHAR(50),
20. payment_mode VARCHAR(50),
21. discount_code VARCHAR(20),
22. order_date DATE

23.);

✅ Table created successfully.

🧱 Step 4 — Create the Lambda Function

✅ **Purpose:**

Triggered when a new file arrives in S3.

🪜 **Steps:**

1. Go to **AWS Lambda → Create function**
    - Runtime: Python 3.12
    - Permissions: Create new role with basic Lambda permissions
2. Upload your lambda_function.py
   (It contains logic to read → clean → upload → insert → notify)

Example lambda_function.py (simplified):

```
import boto3, os, pandas as pd, pymysql
from io import StringIO

s3 = boto3.client('s3')
sns = boto3.client('sns')

def lambda_handler(event, context):
  try:
    bucket_name = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']
    print(f"Triggered by file: {key}")

    # 1️⃣ Read file from S3
    response = s3.get_object(Bucket=bucket_name, Key=key)
    df = pd.read_csv(response['Body'])
```

```
    print(f"Original DataFrame shape: {df.shape}")

    # 2️⃣ Clean data
    cleaned_df = df.dropna().drop_duplicates()
    print(f"Cleaned DataFrame shape: {cleaned_df.shape}")

    # 3️⃣ Save back to S3
    csv_buffer = StringIO()
    cleaned_df.to_csv(csv_buffer, index=False)
    s3.put_object(Bucket=bucket_name, Key=f"processed/cleaned_{key}", Body=csv_buffer.getvalue())
    print(f"✅ Cleaned file uploaded to: processed/cleaned_{key}")

    # 4️⃣ Connect to RDS
    conn = pymysql.connect(
        host=os.environ['DB_HOST'],
        user=os.environ['DB_USER'],
        password=os.environ['DB_PASSWORD'],
        database=os.environ['DB_NAME']
    )
    cur = conn.cursor()
    print("Connected to RDS")

    # 5️⃣ Insert rows
    for _, row in cleaned_df.iterrows():
        try:
            cur.execute("""
```

```
                INSERT INTO cleaned_orders (order_id, customer_name, city, product, quantity, price, channel, payment_mode, discount_code, order_date)
                VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)
            """, tuple(row))
        except Exception as e:
            print(f"Skipping row due to error: {e}")
    conn.commit()
    rows_inserted = len(cleaned_df)
    conn.close()
    print(f"✅ {rows_inserted} rows inserted into RDS")

    # 6️⃣ SNS Notification
    sns.publish(
        TopicArn=os.environ['SNS_TOPIC_ARN'],
        Message=f"✅ Orders data pipeline successful. {rows_inserted} rows inserted into RDS.",
        Subject="Orders Pipeline Success"
    )
    print("📩 Notification sent successfully.")

  except Exception as e:
    print(f"❌ Error: {e}")
    sns.publish(
        TopicArn=os.environ['SNS_TOPIC_ARN'],
        Message=f"❌ Orders data pipeline failed with error: {e}",
        Subject="Orders Pipeline Failed"
    )
    raise e
```

🌎 Step 5 — Add Lambda Layers

If Pandas or PyMySQL are too big, upload them as Lambda **layers**.

Examples:

- arn:aws:lambda:us-east-1:770693421928:layer:Klayers-p312-pandas:1
- arn:aws:lambda:us-east-1:770693421928:layer:Klayers-p312-PyMySQL:1

Add these under:

Lambda → Configuration → Layers → Add layer → Specify ARN

🔑 Step 6 — Set Environment Variables in Lambda

Under **Configuration → Environment Variables**, add:

Key	Value
DB_HOST	your RDS endpoint
DB_USER	admin
DB_PASSWORD	your password
DB_NAME	ordersdb
SNS_TOPIC_ARN	ARN of your SNS topic

📩 Step 7 — Setup SNS for Email Notification

🪜 Steps:

1. Go to **SNS → Topics → Create topic**
    - Type: Standard
    - Name: orders-pipeline-topic
2. Copy the **Topic ARN** (use it in Lambda environment variables)
3. Under **Subscriptions → Create subscription**
    - Protocol: Email
    - Endpoint: your email

- Confirm via the email you receive

🧪 Step 8 — Test the Entire Flow

Trigger the Lambda:

Upload orders_detailed.csv again to S3 → raw/.

Then check:

- CloudWatch logs (Lambda ran successfully)
- RDS table (rows inserted)
- Your email (SNS notification)

Expected log output:

Triggered by file: raw/orders_detailed.csv

Original DataFrame shape: (90, 10)

Cleaned DataFrame shape: (56, 10)

✅ Cleaned file uploaded to processed/cleaned_orders_detailed.csv

Connected to RDS

✅ 56 rows inserted into RDS

📩 Notification sent successfully.

🧾 Step 9 — Verify Results

🧩 In EC2 / MySQL:

USE ordersdb;

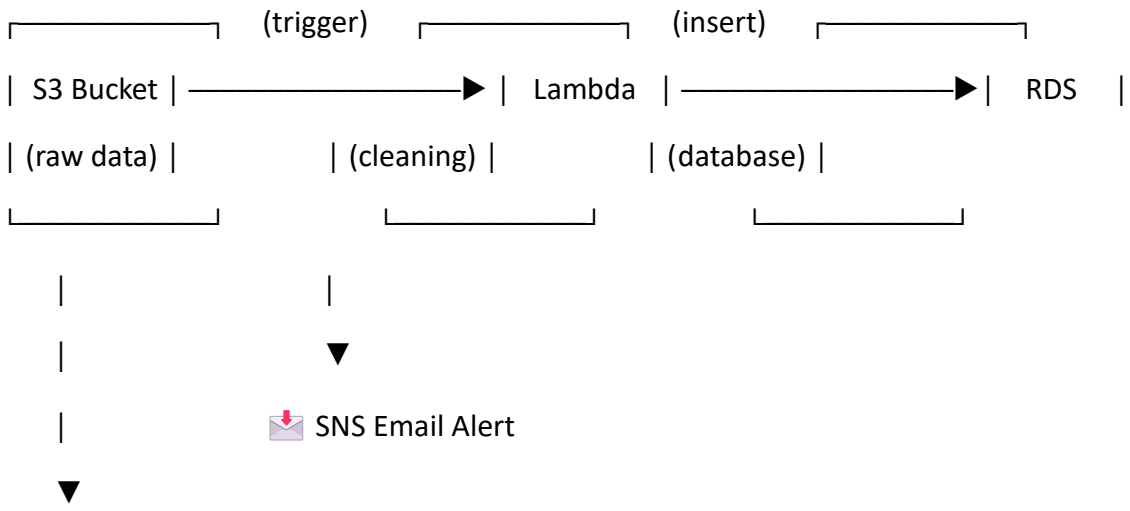
SELECT COUNT(*) FROM cleaned_orders;

SELECT * FROM cleaned_orders LIMIT 5;

📨 In Email:

Check SNS message confirming pipeline success.

🎉 Final Architecture



Code Breakdown

1️⃣ Import libraries

import boto3, os, pandas as pd, pymysql

from io import StringIO

Explanation:

- boto3: AWS SDK for Python (used to connect with S3, SNS)
- os: to read environment variables (like DB credentials)
- pandas as pd: to clean CSV files (drop missing/duplicate data)
- pymysql: to connect to MySQL (RDS)
- StringIO: to handle CSV in memory (no need to save locally)

✅ So this section sets up the tools your code will use.

2️⃣ Create AWS service clients

s3 = boto3.client('s3')

sns = boto3.client('sns')

Explanation:
These two lines connect your Lambda function to:

- Amazon **S3** (to read/write files)
- Amazon **SNS** (to send notifications)

Every time you want to use AWS services in Python, you make a client like this.

3️⃣ Define the main Lambda handler

def lambda_handler(event, context):

Explanation:

- AWS always looks for a function called lambda_handler
- It is automatically called whenever the Lambda is triggered (in your case, by S3 file upload)
- event contains info about what triggered the function (like the bucket name & file path)

4️⃣ Extract the S3 file info

bucket_name = event['Records'][0]['s3']['bucket']['name']

key = event['Records'][0]['s3']['object']['key']

print(f"Triggered by file: {key}")

Explanation:

- Every time someone uploads a file, this event has details about which bucket and which file triggered the function.
- bucket_name → name of your bucket
- key → path to the file (e.g., raw/orders_detailed.csv)
- This lets your code know **what file to read**.

5️⃣ Read the file from S3 into Pandas

response = s3.get_object(Bucket=bucket_name, Key=key)

df = pd.read_csv(response['Body'])

print(f"Original DataFrame shape: {df.shape}")

Explanation:

- Downloads the CSV directly into memory (not to disk)
- Uses pd.read_csv() to load data into a Pandas DataFrame
- df.shape shows number of rows & columns in the original file

So now you have your CSV data in memory, ready to clean.

6️⃣ Clean the data

cleaned_df = df.dropna().drop_duplicates()

print(f"Cleaned DataFrame shape: {cleaned_df.shape}")

Explanation:

- dropna() removes rows that have missing (NaN) values
- drop_duplicates() removes any duplicate rows
- After cleaning, it prints how many rows remain

This ensures that only clean, valid data will go to RDS.

7️⃣ Upload cleaned CSV back to S3

csv_buffer = StringIO()

cleaned_df.to_csv(csv_buffer, index=False)

s3.put_object(Bucket=bucket_name, Key=f"processed/cleaned_{key}", Body=csv_buffer.getvalue())

print(f"✅ Cleaned file uploaded to: processed/cleaned_{key}")

Explanation:

- Converts the cleaned DataFrame back into CSV text (in memory)
- Uploads it to the **processed/** folder in your S3 bucket
- No need for local files — everything is in-memory

Now, your clean data is saved in S3.

8️⃣ Connect to RDS MySQL

conn = pymysql.connect(

```
    host=os.environ['DB_HOST'],
    user=os.environ['DB_USER'],
    password=os.environ['DB_PASSWORD'],
    database=os.environ['DB_NAME']
)
cur = conn.cursor()
print("Connected to RDS")
```

Explanation:

- Connects your Lambda to the RDS database
- Credentials are taken securely from **environment variables**
  (set in Lambda → Configuration → Environment variables)
- cursor() lets you execute SQL queries later

🐬 Insert data row by row into MySQL

```
for _, row in cleaned_df.iterrows():
    try:
        cur.execute("""
            INSERT INTO cleaned_orders
            (order_id, customer_name, city, product, quantity, price, channel, payment_mode, discount_code, order_date)
            VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)
        """, tuple(row))
    except Exception as e:
        print(f"Skipping row due to error: {e}")

conn.commit()
rows_inserted = len(cleaned_df)
conn.close()
```

print(f"✅ {rows_inserted} rows inserted into RDS")

Explanation:

- Loops through each row in the cleaned DataFrame
- Executes an SQL INSERT command to store it in your MySQL table
- If any row fails (e.g., bad format), it prints and skips it
- Commits changes, closes connection, and prints how many rows were inserted

This is the step that actually loads your cleaned data into RDS.

🔔 10 Send SNS Notification (Success)

```
sns.publish(
    TopicArn=os.environ['SNS_TOPIC_ARN'],
    Message=f"✅ Orders data pipeline successful. {rows_inserted} rows inserted into RDS.",
    Subject="Orders Pipeline Success"
)
print("📩 Notification sent successfully.")
```

Explanation:

- Uses SNS to send an email (or SMS) once the pipeline succeeds
- Message content includes how many rows were inserted
- You'll get this as an email in your subscribed inbox

⚠️ 11 Error Handling (Failure)

```
except Exception as e:
    print(f"❌ Error: {e}")
    sns.publish(
        TopicArn=os.environ['SNS_TOPIC_ARN'],
        Message=f"❌ Orders data pipeline failed with error: {e}",
        Subject="Orders Pipeline Failed"
```

)

raise e

Explanation:

- If *anything* fails (e.g., wrong DB password, missing column),
  it catches the error and sends you an SNS alert that something broke.
- Then re-raises the error so CloudWatch logs the details.

This ensures you're notified about both success ✅ and failure ❌.

💡 **Summary: What Happens When You Upload a File**

Step	What Happens	AWS Service
1	CSV uploaded	**S3** triggers Lambda
2	Lambda runs code	**Lambda**
3	Reads file from S3	**boto3**
4	Cleans data	**Pandas**
5	Uploads cleaned file	**S3 (processed/)**
6	Inserts into MySQL	**RDS (PyMySQL)**
7	Sends notification	**SNS**
8	Logs everything	**CloudWatch**