Assignment - 1

- Create SNS
- Create Lamda function

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    for record in event['Records']:
        sns_message = record['Sns']['Message']
        logger.info(f"Received SNS message: {sns_message}")

        try:
            # Assuming the message is in JSON format
            data = json.loads(sns_message)
            name = data.get('name', 'Unknown')
            email = data.get('email', 'No email')
            logger.info(f"New signup: {name} ({email})")
        except Exception as e:
            logger.error("Failed to parse message", exc_info=True)

    return {
        'statusCode': 200,
        'body': 'SNS message processed'
    }
```

- Again, go to SNS and create the subscription
- Again go to SNS>Topic and publish message
- Test this message in the Services>CloudWatch>Logs>Log groups

Assignment - 2

- Go to services>SQS - Create Queue —> “order-processing-queue”
- Again, go to SQS>Create Queue(dlq)
- Go back to “order-processing-queue” — Edit and Enable the DLQ(Dead Letter Queue —> order-dlq)
- Go to Lambda>create function

```
import json
import time
import random
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
def lambda_handler(event, context):
    for record in event['Records']:
        body = record['body']
        logger.info(f"Processing order: {body}")
        # Simulate delay between 1 to 3 seconds
        delay = random.randint(1, 3)
        logger.info(f"Simulating processing time: {delay}s")
        time.sleep(delay)
        logger.info(f"Order processed: {body}")
    return {
        'statusCode': 200,
        'body': 'Batch processed'
    }
```

- Deploy and add a trigger —> “order-processing-queue”
- Go back to SQS>order-processing-queue, then send and receive messages>under message body type message in JSON format.

- Go to SNS>create topic(select fifo) — “order-topic”
- Come back to SQS>create queue> — “order-processing-queue.fifo”
- Again, go to SNS>order-topic>create subscription>select “order-processing-queue.fifo” as end point and create.
- Go to SNS > Topics > order-topic.fifo — Publsih message
- Go to SQS → order-processing-queue.fifo → Send and receive messages → Poll for messages

Assignment 3

- Create a Lambda function named FifoOrderProcessor

```
import json
import time
import random
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    for record in event['Records']:
        body = json.loads(record['body'])  # Decode message JSON
        logger.info(f"Processing order: {json.dumps(body)}")

        # Simulate processing delay (1–3 seconds)
        delay = random.randint(1, 3)
        logger.info(f"Simulating processing time: {delay}s")
        time.sleep(delay)

        logger.info(f"Order processed: {json.dumps(body)}")

    return {
        'statusCode': 200,
        'body': 'FIFO batch processed successfully'
    }
```

- Deploy it-> Add trigger, if permission denied, then go back to the SNS tab and select the .fifo and add permissions.
- After deploying and adding a trigger, go to SNS>order-topic. fifo and publish a message, and confirm in CloudWatch

Assignment 4

We're simulating CPU usage > 80%, creating an alarm, and auto-remediating with a Lambda that:

- Logs the alert to CloudWatch Logs
- Sends a notification to SNS

Create a custom CloudWatch metric

Go to CloudShell and run:

aws cloudwatch put-metric-data \

--metric-name HighCPUUsage \
--namespace Custom/EC2 \
--value 85 \
--dimensions InstanceId=i-1234567890abcdef0

Create an alarm using the above metric

- Go to **CloudWatch > Alarms > Create alarm**
- Select metric → Browse → Custom/EC2 → Select `HighCPUUsage`
- Condition:
  Threshold type: Static
  Whenever HighCPUUsage is **Greater than 80**

Configure actions

- SNS Topic: Select existing or create new → e.g., cpu-alert-topic.fifo
- Lambda action: Select existing function (e.g., CpuRemediator)

Lambda Function

Create a Lambda function (`CpuRemediator`)

Give it AWSLambdaBasicExecutionRole + permission to log and publish to SNS

Add this logic:

```
import json

import logging
import boto3

logger = logging.getLogger()
logger.setLevel(logging.INFO)

sns = boto3.client('sns')

def lambda_handler(event, context):
    logger.info("High CPU Alarm triggered!")
    logger.info(json.dumps(event))

    # Mock remediation logic
    logger.info("Auto-remediation triggered: Would
scale up EC2 or notify admin")

    # Optional SNS notification (if wired)
    sns.publish(
        TopicArn='arn:aws:sns:us-east-1:1234567890:cpu-
alert-topic.fifo',
        Message='High CPU detected. Auto-remediation
initiated.',
        Subject='HighCPU Alert'
    )

    return {
        'statusCode': 200,
        'body': 'Alarm handled.'
    }
```

Trigger the alarm

Go to CloudShell and rerun:

```
aws cloudwatch put-metric-data \
--metric-name HighCPUUsage \
--namespace Custom/EC2 \
--value 95 \
--dimensions InstanceId=i-1234567890abcdef0
```

Verify

- Go to CloudWatch > Alarms → You'll see the alarm is In alarm
- Go to Lambda > Monitor > View logs in CloudWatch
- You should see logs like:

"High CPU Alarm triggered!"

"Auto-remediation triggered..."

Again Go to Lambda > Create function > "CpuRemediator"

```
import json
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
def lambda_handler(event, context):
    logger.info("High CPU Alarm Triggered!")
    for record in event['Records']:
        sns_msg = record['Sns']['Message']
        logger.info(f"Alarm details: {sns_msg}")
        # Simulate remediation action
        logger.info("Taking mock remediation action: Scaling up EC2 instance...")

    return {
        'statusCode': 200,
        'body': 'Remediation Lambda executed.'
    }
```

Assignment 5:

- Go to RDS console and create a MySQL database (Free Tier)
  → Example DB name: userdb
  → Note the endpoint and port

- Go to Secrets Manager and create a new secret
  → Store DB username and password
  → Name it something like prod/AppBeta/Mysql

- Go to IAM and attach permissions to your Lambda's execution role
  → Add SecretsManagerReadWrite or a policy that includes secretsmanager: GetSecretValue

- Create Lambda function (or use existing CpuRemediator)
  → Runtime: Python 3.12
  → Add environment variable DB_HOST with your RDS endpoint
  → Upload a zip containing pymysql + lambda_function.py

- Code includes:
  → Getting DB credentials from Secrets Manager
  → Connecting to RDS
  → Deleting existing test data
  → Bulk inserting using ThreadPoolExecutor

- Go to Lambda Test tab
  → Create a test event
  → Click "Test" and check logs

- Go to CloudShell and confirm inserts
  → Run SELECT * FROM users;

Lambda Code:

```
import boto3
import os
import json
import pymysql
from botocore.exceptions import ClientError
import concurrent.futures

# Get credentials from AWS Secrets Manager
def get_db_credentials():
    secret_name = "prod/AppBeta/Mysql"
    region_name = "us-east-1"
    client = boto3.client("secretsmanager",
region_name=region_name)

    try:
        response = client.get_secret_value(SecretId=secret_name)
        return json.loads(response['SecretString'])
    except ClientError as e:
        raise Exception("Error retrieving secret: " + str(e))

# Thread-safe DB insert (creates new connection & cursor)
def insert_user(host, user, password, name, email):
    try:
        conn = pymysql.connect(
            host=host,
            user=user,
            password=password,
            database='userdb',
            connect_timeout=5
        )
        with conn.cursor() as cursor:
            cursor.execute(
                "INSERT INTO users (name, email) VALUES (%s,
%s)",
                (name, email)
            )
            conn.commit()
        conn.close()
    except Exception as e:
        print(f"Insert failed for {email}: {e}")

# Lambda entrypoint
```

```
def lambda_handler(event, context):
    creds = get_db_credentials()
    host = os.environ['DB_HOST']
    user = creds['username']
    password = creds['password']

    # Sample users to insert
    names = [
        ("Nithin", “nithin@gmail.com”),
        ("Nani", “nani@gmail.com”),
        ("Lalitha", “lalitha@gmail.com”),
        ("Lucky", “lucky@gmail.com”),
        ("Niveditha", “nivi@gmail.com”)
    ]

    try:
        # Step 1: Delete existing matching emails
        conn = pymysql.connect(
            host=host,
            user=user,
            password=password,
            database='userdb',
            connect_timeout=5
        )
        with conn.cursor() as cursor:
            cursor.execute("DELETE FROM users WHERE email LIKE
'%@lunexa.ai'")
            conn.commit()
        conn.close()

        # Step 2: Insert new users in parallel using thread pool
        with concurrent.futures.ThreadPoolExecutor() as
executor:
            futures = [
                executor.submit(insert_user, host, user,
password, name, email)
                for name, email in names
            ]
            for f in futures:
                f.result()

        return {"message": "Bulk insert successful"}

    except Exception as e:
        return {"error": str(e)}
```

Bonus Assignment

Parallel Processing Framework using S3, SQS, and Lambda Fan-out Model

Step 1: Create Sample Data

- Write a Python script using Faker and csv to generate fake user data.
- Save the file as users.csv to path:
  /Users/nithinrajulapati/Desktop/users.csv

Step 2: Upload CSV File to S3

- Go to S3 console and create a new bucket:
  parallel-processing-bucket-nithin (Region: us-east-1)
- Upload the users.csv file to this bucket.

Step 3: Create SQS Queue

- Go to the **SQS Console** and create a Standard Queue:
  UserRecordQueue
- Copy the SQS **ARN** for later use.

Step 4: Create First Lambda (S3 to SQS Dispatcher)

- Name: S3ToSQSDispatcher
- Runtime: Python 3.12
- IAM Role:
  - Add **AmazonS3ReadOnlyAccess**
  - Add **AmazonSQSFullAccess**
  - Add **AWSLambdaBasicExecutionRole**
- Attach this Lambda to the S3 bucket as a trigger (event type: **PUT**).

- Lambda Code:
    - Reads the uploaded CSV file line by line.
    - Sends each record as a message to the SQS queue.
    - File key and bucket name are dynamically extracted from the event.
    - Uses boto3 to send messages to SQS.
- Note: This Lambda is triggered automatically when a new CSV file is uploaded.

Step 5: Create Second Lambda (SQS Processor)

- Name: ProcessCsvRecord
- Runtime: Python 3.12
- IAM Role:
    - Add **AWSLambdaBasicExecutionRole**
    - Add custom inline policy AllowSQSTrigger (for sqs:ReceiveMessage)
- Trigger: Add SQS queue (UserRecordQueue) as event source.
- Lambda Code:
    - Triggered per message received from SQS.
    - Parses and logs each user record.
    - Simulates parallel processing for each message.
- Verified from **CloudWatch Logs** that each record was processed correctly.

Step 6: Benchmarking

- Generated a larger dataset (benchmark-users.csv) using Faker with 80+ records.
- Uploaded it to S3 to trigger fan-out pipeline.
- Verified in:

    - CloudWatch logs of S3ToSQSDispatcher: All records sent to SQS.
    - CloudWatch logs of ProcessCsvRecord: All records received and processed.
- Final stats:
    - Messages processed: 80+
    - SQS delivery: Success
    - Lambda parallelism: Verified
    - Timeout: Avoided for small files; will need tuning for bigger workloads.

Final Outcome:

- Successfully implemented fan-out architecture using:
    - **S3** (file trigger)
    - **Lambda 1** (splits and sends to SQS)
    - **SQS** (message queue)
    - **Lambda 2** (parallel record processor)
- Logs confirm complete pipeline execution with zero loss.
- Architecture is scalable and production-ready.

Lambda Function 1: S3ToSQSDispatcher

```
import boto3
import csv
import os
sqs = boto3.client('sqs')
def lambda_handler(event, context):
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']
```

```
    s3 = boto3.client('s3')
    response = s3.get_object(Bucket=bucket, Key=key)
    lines = response['Body'].read().decode('utf-8').splitlines()
    reader = csv.DictReader(lines)
    queue_url = "https://sqs.us-east-1.amazonaws.com/204769685377/UserRecordQueue"
    for row in reader:
        sqs.send_message(
            QueueUrl=queue_url,
            MessageBody=str(row)
        )
        print(f"Sent to SQS: {row}")
```

Lambda Function 2: ProcessCsvRecord

```
import json
def lambda_handler(event, context):
    for record in event['Records']:
        body = record['body']
        message = json.loads(body.replace("'", '"'))  # fix for single quotes in stringified dict

        name = message.get('name')
        email = message.get('email')

        print(f"Received message: {message}")
        print(f"Processing user: {name} with email: {email}")
```