VPC NAT Gateway Setup

Goal: Allow private subnet instances to access the internet using a NAT Gateway.

Steps:

1. **Create a VPC**
   - Go to VPC > Create VPC > Name: MyVPC, CIDR: 10.0.0.0/16
2. **Create Subnets**
   - Public Subnet: 10.0.1.0/24, Availability Zone: a
   - Private Subnet: 10.0.2.0/24, Availability Zone: a
3. **Create Internet Gateway (IGW)**
   - Attach it to MyVPC
4. **Route Table for Public Subnet**
   - Add route: 0.0.0.0/0 → IGW
   - Associate this route table with the public subnet
5. **Create NAT Gateway**
   - Allocate Elastic IP
   - Select the public subnet
   - Wait for NAT Gateway to be "Available"
6. **Route Table for Private Subnet**
   - Add route: 0.0.0.0/0 → NAT Gateway
   - Associate this route table with the private subnet

2.S3 Object Triggering Lambda → Update DynamoDB

Goal: Uploading an object to S3 triggers a Lambda that updates DynamoDB.

Steps:

1.Create an S3 Bucket

- Enable event notification: Put → Send to Lambda

2.Create a DynamoDB Table

- Name: S3Uploads
- Partition key: fileName (String)

3.Create Lambda Function

- Runtime: Python/Node.js
- IAM role with access to DynamoDB and S3
- Code reads object metadata and puts into S3Uploads table

- Function:

```
import boto3
from uuid import uuid4
def lambda_handler(event, context):
    s3 = boto3.client("s3")
    dynamodb = boto3.resource('dynamodb')
    for record in event['Records']:
        bucket_name = record['s3']['bucket']['name']
        object_key = record['s3']['object']['key']
        size = record['s3']['object'].get('size', -1)
        event_name = record ['eventName']
        event_time = record['eventTime']
        dynamoTable = dynamodb.Table('newtable')
        dynamoTable.put_item(
            Item={'unique': str(uuid4()), 'Bucket': bucket_name, 'Object': object_key,'Size': size, 'Event': event_name, 'EventTime': event_time})
```

4.Add S3 Trigger

- Bucket: Your bucket
- Event type: PUT
- Destination: Lambda

5.Upload file to S3

- Check Lambda logs and DynamoDB updates

3.SNS + S3 + Lambda + Email

- **Goal: Upload triggers Lambda → SNS → Email**
- **Steps:**
- **Create SNS Topic**
- **Name: S3UploadAlert**
- **Add Email subscription**
- **Confirm email**
- **Create Lambda Function**
- **Publish to SNS Topic**
- **Add environment variable for SNS topic ARN**
- **S3 Setup**
- **Enable PUT event to trigger Lambda**
- **Lambda Logic**
- **Extract file info from event**
- **Send SNS message with file name and time**

4.SQS + Purge + Lambda Trigger

1.Go to SQS → Create queue:

- **Type: Standard, name: MyQueue → Create.**

2.Create Lambda function:

- **Runtime example: Python 3.9**
- **Attach IAM policy: AmazonSQSFullAccess**

python

CopyEdit

```
import json
def handler(event, context):
  for rec in event['Records']:
    print("Received:", rec['body'])
```

⍰ Save → Under Function overview, click Add trigger:

- **Trigger type: SQS, select MyQueue, batch size if needed → Add.**

⍰ **Test by sending test message:**

- **Go to SQS queue → Send and receive messages → Send message, type any text → check Lambda logs.**

⍰ **Purge messages:**

- **In SQS console, queue selected → click Purge queue → confirm.**

5.DynamoDB Scan, Query, PartiQL, JSON Change

Goal: Perform various operations on DynamoDB

Steps:

1. **Create DynamoDB Table**
   - Name: Users, Partition Key: userId
2. **Insert Items**
   - Use AWS Console or CLI to add JSON items
3. **Scan**
   - Console → Explore Table → Scan
   - CLI: aws dynamodb scan --table-name Users
4. **Query**
   - Add sort key (e.g., timestamp) if needed
   - Query by userId
5. **PartiQL**
   - Query: SELECT * FROM "Users" WHERE "userId"='abc123'
   - Insert: INSERT INTO "Users" VALUE {'userId':'x','name':'Y'}
6. **JSON Format Change**
   - Use Lambda or AWS Glue to transform JSON from one format to another

---

6. CloudFront With & Without S3 Website Hosting

A. With S3 Static Website Hosting

1. **S3 → bucket → Properties → Static website hosting → Enable.**
   - **Index document: index.html**
   - **Error document: error.html**
2. **Go to Permissions → enable Block Public Access off → policies must allow public read.**

3. **Upload index.html and error.html, click Upload → Make public.**
4. **Go to CloudFront → Create distribution → select Web:**
   - **Origin domain: pick bucket website endpoint**
   - **Default root object: index.html**
   - **Leave other settings default → Create distribution.**
5. **Wait for status "Deployed" → use Domain Name given by CloudFront.**

B. Without S3 Website Hosting

1. **In S3 bucket, Static website hosting: Disable.**
2. **In CloudFront distribution, under Origins and origin groups:**
   - **Add OAC/OAI:**
     - **Click Origins → Create origin → Origin domain: select bucket (not website endpoint).**
     - **Expand Origin access → Create OAC (or OAI) → apply.**

7.IAM Creation & CLI Access

1. **Console navigation: Services → IAM.**
2. **Create User:**
   - **Click Users → Add users.**
     - **Username: cli-user**
     - **Access: check Programmatic access only → Next.**
3. **Assign Permissions:**
   - **Choose existing group or add policies directly: e.g., AmazonS3FullAccess.**
   - **Click Next, review, Create user → download .csv with Access Key ID & Secret.**
4. **Create Group (optional):**
   - **IAM → User groups → Create group, assign policies → Add cli-user to group.**

5. **Create Role (e.g. for EC2 or Lambda):**
    - **IAM → Roles → Create role.**
        - **Trusted entity: AWS service (e.g., Lambda)**
        - **Attach permissions (e.g., AWSLambdaBasicExecutionRole)**
        - **Name role → Create role.**
6. **CLI Configuration:**

bash

CopyEdit

```
aws configure
# Enter access key, secret, region (e.g. ap-south-1), output format (json)
```

7. **Test:**

bash

CopyEdit

```
aws s3 ls
aws dynamodb list-tables
```

8. Lex Bot + Twilio Integration Step A: Build Lex Bot

1. Lex Console → **Create bot (Lex V2)**.
    - Bot name: MyLexBot
    - Language: English (US)
    - Intent: click **Add intent** → **Create intent** → Name: HelloIntent.
        - Sample utterances: “Hello”, “Hi”.
        - Response: “Welcome to my bot!”
    - Save intent → **Build** bot.
2. Add alias:
    - Go to **Bot details** → **Add alias** → Name: TestAlias → alias points to latest version.
3. Test in console -> chat with bot.

Step B: Lambda + Twilio Setup

1. Create Lambda function:

  - Runtime: Node.js 18.x
  - Permissions: create or use role with Lex runtime access.

2. Deploy code:

js

CopyEdit

```
const AWS = require('aws-sdk');
const lex = new AWS.LexRuntimeV2();
exports.handler = async event => {
const userMsg = event.Body;
  const params = {
    botId: "YOUR_BOT_ID",
    botAliasId: "TestAliasID",
    localeId: "en_US",
    sessionId: event.From,
    text: userMsg
  };
  let res = await lex.recognizeText(params).promise();
  return {
    statusCode: 200,
    body: `
    <Response>
      <Message>${res.messages.map(m => m.content).join("")}</Message>
    </Response>`
  };
};
```

3. Expose via API Gateway:
   - Create **HTTP API** → route POST /sms → integrate with Lambda → deploy + note endpoint URL.

4. Twilio Console:
   - Buy a phone number.
   - Go to **Phone Numbers** → your number → **Messaging** → set **A CALL COMES IN / A MESSAGE COMES IN** webhook to your API Gateway endpoint.

    - Save.

🟩 Now messages to your Twilio number go to Lex and reply automatically.