**Big Data Overview:**

Big Data refers to extremely large datasets that are complex and grow at a rapid rate, making traditional data processing applications inadequate to handle them. It can be categorized by the following dimensions:

1. **Volume** – The sheer amount of data generated. This could be petabytes or even exabytes of data, coming from different sources like IoT devices, social media, sensors, and enterprise applications.
2. **Velocity** – The speed at which data is generated and processed. Big data solutions need to handle real-time or near real-time data flows from sources like IoT devices, social media streams, or financial markets.
3. **Variety** – Big data comes in various formats such as:
   * **Structured**: Data that follows a specific schema (e.g., relational databases like MySQL).
   * **Unstructured**: Data without a predefined model (e.g., text, images, videos).
   * **Semi-structured**: Data like JSON, XML files, which are not strictly tabular but still have some structure.

**Collection of Big Data:**

Big data collection involves gathering data from a multitude of sources. Here are a few common sources:

1. **IoT Devices**: Sensors, smart appliances, wearables, and connected vehicles continuously generate data. For example, a smart thermostat sends temperature data every few seconds.
2. **Machines**: Industrial equipment, servers, and other infrastructure create logs, performance data, and error reports, all of which contribute to large datasets.
3. **Social Media Platforms**: User-generated content from platforms like Twitter, Facebook, and Instagram create vast amounts of unstructured data.
4. **Enterprise Systems**: CRM, ERP systems, and databases produce structured and semi-structured data in large volumes.

**Storage for Big Data:**

Given the large size and complexity of data, traditional databases are inadequate for storing Big Data. Instead, specialized storage solutions are required. Common Big Data storage options include:

1. **Data Lakes (e.g., Amazon S3)**: A data lake is a centralized repository that allows you to store both structured and unstructured data at scale. In AWS, Amazon S3 is commonly used as the backend storage for data lakes. It is designed to handle massive amounts of data at relatively low costs and supports any format (CSV, JSON, video, etc.).
2. **Amazon S3**: S3 (Simple Storage Service) allows you to store vast amounts of data and retrieve it when needed. It’s ideal for handling raw data that may later be processed by other tools.
   * **Benefits**:
     + Scalable: Can store virtually unlimited data.
     + Durable: Ensures 99.999999999% durability.
     + Secure: Provides encryption and fine-grained access control.

**Processing Big Data:**

Once data is collected and stored, the next step is processing it. Depending on the use case, you may need to batch process large datasets or handle real-time streams of data.

1. **AWS Glue** (ETL Processing):
   * AWS Glue is a fully managed extract, transform, and load (ETL) service that helps you prepare and clean your data for analytics. Glue can crawl your data stored in S3, generate metadata (like schema), and transform it using Apache Spark jobs.
   * **Use Case**: When you have raw, semi-structured, or structured data and need to convert it into a structured format that can be queried efficiently.
2. **Amazon Redshift** (Data Warehousing):
   * Redshift is a fast, fully managed data warehouse that allows you to run SQL queries on massive datasets. It is optimized for structured data and is ideal for running analytics queries on large datasets.
   * **Use Case**: When you need to analyze large volumes of structured data in real-time and run complex SQL queries.
3. **Amazon EMR** (Hadoop/Spark Cluster):
   * EMR (Elastic MapReduce) provides a managed cluster platform for processing large data sets using open-source big data frameworks such as Apache Hadoop and Apache Spark.
   * **Use Case**: For processing unstructured data at scale (e.g., large-scale log analysis, machine learning workloads) using distributed computing models.
4. **AWS Lambda** (Serverless Processing):
   * Lambda is a serverless compute service that lets you run code without provisioning servers. It can be used to handle real-time data streams or process event-driven data in response to triggers (e.g., when new data lands in an S3 bucket).
   * **Use Case**: Real-time data processing like image recognition, log filtering, or lightweight ETL tasks triggered by specific events.

**Example Use Case of Big Data Architecture in AWS:**

1. **Data Collection**: IoT sensors in a smart city generate data every second (temperature, air quality, vehicle movement) and send it to the cloud.
2. **Data Storage**: The raw data is stored in an Amazon S3 bucket. The data is unstructured and includes text, logs, images, and more.
3. **Data Processing**:
   * **AWS Glue** is used to run ETL jobs to clean the raw data and convert it into a structured format (e.g., converting sensor logs into a format that Redshift can query).
   * **Amazon EMR** is used for batch processing when you need to analyze large-scale datasets like log files from the past year.
   * **AWS Lambda** is triggered to process real-time data from IoT devices, providing real-time alerts when specific conditions (like high pollution) are met.
4. **Data Analytics**:
   * **Amazon Redshift** is used to analyze structured data stored in a data warehouse, helping city planners to make decisions based on historical trends.
   * **Data Lakes** on S3 allow for scalable, long-term storage of all unstructured data, and this data can be queried later using AWS Athena or Glue.

In summary, big data environments involve various components for data collection, storage, and processing. The flexibility and scalability of AWS services like S3, Glue, EMR, and Redshift enable seamless handling of massive datasets with diverse requirements.

**Lab Guide: Auto Scaling with Custom AMI and CPU Stress Test**

**Prerequisites:**

* AWS Account
* AWS CLI installed (optional)
* Basic knowledge of EC2 and Auto Scaling

**Step 1: Create a Custom AMI with Apache Installed**

1. **Launch a new EC2 instance:**
   * Go to the EC2 dashboard.
   * Choose **Amazon Linux 2 AMI** (ami-00f251754ac5da7f0).
   * Select instance type (e.g., t2.micro for testing).
   * Configure key pair for SSH access.
2. **Connect to the EC2 instance:**
   * Connect using SSH:

bash

Copy code

ssh -i "your-key.pem" ec2-user@your-instance-public-ip

1. **Install Apache (HTTPD) and other necessary packages:**

bash

Copy code

sudo yum update -y

sudo yum install httpd -y

sudo systemctl start httpd

sudo systemctl enable httpd

* + Verify Apache installation by visiting http://<instance-public-ip>.

1. **Create a custom AMI from this instance:**
   * Stop the EC2 instance.
   * Go to **Actions** > **Create Image**.
   * Name the image (e.g., apache-server-ami).
   * Wait for the AMI creation to complete.

**Step 2: Create an Auto Scaling Group (ASG)**

1. **Create a Launch Template:**
   * Go to the **EC2** dashboard and select **Launch Templates**.
   * Click **Create Launch Template**.
   * Name the template (e.g., apache-launch-template).
   * Choose the custom AMI you created in Step 1.
   * Configure instance type (e.g., t2.micro or larger if needed).
   * Attach a security group that allows HTTP (port 80) traffic.
   * Leave the rest as default and click **Create Launch Template**.
2. **Create the Auto Scaling Group:**
   * Go to **Auto Scaling Groups** in the EC2 dashboard.
   * Click **Create Auto Scaling Group**.
   * Select the launch template created earlier.
   * Set the desired instance count (e.g., 1).
   * Configure the network (select an existing VPC and subnets).
   * Configure scaling policies later; for now, set manual scaling.
   * Review and create the Auto Scaling Group.

**Step 3: Scale Up and Down Manually**

1. **Manual Scaling of ASG:**
   * Go to the **Auto Scaling Groups** dashboard.
   * Select your ASG.
   * Click **Instance Management** > **Desired Capacity**.
   * Increase the **Desired Capacity** to 2 (or more) to scale up.
   * Decrease the **Desired Capacity** to 1 (or fewer) to scale down.

**Step 4: Install stress Tool for Load Testing**

1. **Connect to one of the instances created by the ASG:**
   * Find the instances under your Auto Scaling Group.
   * SSH into one of the instances:

bash

Copy code

ssh -i "your-key.pem" ec2-user@<instance-public-ip>

1. **Install the stress tool:**
   * Run the following commands to install the EPEL repository and stress package:

bash

Copy code

sudo amazon-linux-extras install epel -y

sudo yum install stress -y

1. **Run the stress command to simulate CPU load:**
   * Use the stress tool to simulate high CPU load on the instance:

bash

Copy code

stress --cpu 32

* + This command will stress all available CPU cores (or a specific number, e.g., 32 threads) to simulate a high-load environment.

**Step 5: Monitor and Test Auto Scaling**

1. **Monitor Instance CPU Load:**
   * Open **CloudWatch** > **Metrics** > **EC2** to observe CPU utilization on the running instance.
   * If the CPU load is high, you can manually scale up by increasing the desired capacity in the Auto Scaling Group to spin up more instances.
2. **Terminate Instances:**
   * After testing, you can stop the stress test by simply terminating the stress process on the instance:

bash

Copy code

killall stress

1. **Verify Auto Scaling:**
   * Adjust the desired capacity manually in the ASG and observe how instances are launched or terminated accordingly.

**Step 6: Clean Up**

1. **Delete Auto Scaling Group:**
   * Navigate to the **Auto Scaling Groups** section.
   * Select your ASG and click **Delete**.
2. **Delete Launch Template:**
   * Go to **Launch Templates** and delete the template if no longer needed.
3. **Deregister AMI (Optional):**
   * Go to **AMI** section and deregister the custom AMI if you no longer need it.
   * Optionally, delete associated snapshots for cost management.

**Conclusion:**

You’ve successfully set up an Auto Scaling Group using a custom AMI with Apache installed. You manually scaled the instances up and down and used the stress tool to simulate high CPU usage. Monitoring can be further automated using CloudWatch Alarms to trigger scaling events based on metrics like CPU usage.

This guide can be extended by adding scaling policies that automatically adjust instance count based on CPU load or other metrics.

**Kinesis Data Streams Overview:**

Amazon Kinesis Data Streams is a scalable, real-time stream processing service that allows developers to collect, process, and analyze data as it is generated from various sources. It's ideal for high-throughput applications that require real-time analytics, like weather data collection or tracking mobile device metrics.

In this scenario, you are collecting data from thousands of sources (e.g., weather stations and iPhones), and need to process them efficiently using **Kinesis SDK**, **Kinesis Producer Library (KPL)**, and **Kinesis Agent**.

**Components:**

1. **Kinesis Data Streams:**
   * Data in Kinesis streams is divided into **shards**, and each shard can handle a certain capacity of records.
   * In this case, you have **5 shards** for distributing the data across multiple partition keys.
2. **Source:**
   * You are collecting weather data from **5 cities** (['London', 'New York', 'Paris', 'Tokyo', 'Sydney']).
   * Additionally, you are collecting mobile device data, specifically from **iPhones**, capturing attributes like device ID, type, color, size, and battery status.

**Data Collection Tools:**

You have multiple ways to put data into Kinesis streams:

1. **Kinesis SDK**:
   * Allows real-time streaming by sending records directly to the Kinesis stream.
   * Suitable for use cases where low latency is required, but each **PutRecord** API call counts against API limits and can increase the cost.
2. **Kinesis Producer Library (KPL)**:
   * This library is more efficient than using the SDK alone, as it batches and aggregates records before sending them to Kinesis.
   * Reduces the number of API calls by combining multiple records into a single put request, optimizing throughput.
3. **Kinesis Agent**:
   * A lightweight agent that can monitor and capture log files or other streams of data and send them to Kinesis.
4. **CLI**: Useful for manually sending data to Kinesis for testing.

**Data Collection Scenarios:**

You are collecting two types of data:

1. **Weather Data from 5 Cities:**
   * Each record consists of the following fields:
     + city: The name of the city (used as the **partition key**).
     + timestamp: The current timestamp.
     + temperature: The temperature of the city.
     + humidity: The humidity level.
     + weather: A short description of the weather (e.g., "clear sky").

This dataset could generate **100,000 weather records** per second, distributed across the 5 cities. Each record would look something like this:

json

Copy code

{

"city": "London",

"timestamp": "2024-10-09T10:11:12.345Z",

"temperature": 15.6,

"humidity": 60,

"weather": "clear sky"

}

1. **iPhone Data:**
   * Data related to iPhone devices, such as:
     + device\_id: A unique identifier for the iPhone.
     + type: The model of the iPhone.
     + color: The color of the device.
     + size: The storage size of the iPhone (e.g., 128GB).
     + battery: The current battery level of the iPhone.

Each record might look like this:

json

Copy code

{

"device\_id": "abcd1234",

"type": "iPhone 13",

"color": "black",

"size": "128GB",

"battery": "85%"

}

**Partitioning Strategy:**

* **Partition Key**:
  + For weather data, you use city as the partition key, ensuring that all weather data from a particular city is assigned to the same shard.
  + For iPhone data, you can use device\_id as the partition key, ensuring even distribution of data across shards.
* **Hot Partition Issue**:
  + If one partition key (e.g., a very popular city or device) sends disproportionately more records than others, it could lead to a **hot partition** where one shard is overutilized.
  + To avoid this, you might introduce randomization or further partitioning (e.g., hashing partition keys) to distribute the load evenly.

**Data Ingestion:**

**Kinesis SDK (Real-time):**

* The **Kinesis SDK** allows sending one record at a time using the PutRecord API.
  + **Example**: You can send 1000 records per second using this method, but each call incurs an API request. Each **shard** can handle **1000 PutRecord requests per second**, and a record size up to **1 MB**.
  + This method is **fully real-time** but can be inefficient if you have a large number of small records, as each API call introduces overhead.

**Kinesis Producer Library (KPL - Batching and Aggregation):**

* The **Kinesis Producer Library (KPL)** optimizes ingestion by batching and aggregating records before sending them to the Kinesis stream. This approach reduces the number of API calls significantly.
  + **Batching**: Multiple small records are combined into a single API call to reduce overhead. For example, you could send **1000 records** batched into a few kilobytes in a single request, rather than 1000 separate requests.
  + **Aggregation**: KPL aggregates multiple records into a single record (up to **1 MB** per aggregate record). This allows you to send more than 1000 records per second without exceeding the API limits.
  + **Advantages**:
    - Fewer API calls, improving throughput and reducing cost.
    - Allows efficient data transmission without breaching the 1 MB limit per record.
  + **Example**:
    - If you're sending **1000 weather records per second** using KPL, it will batch these records into fewer API calls while adhering to the 1 MB record limit. You can transmit more than 1000 records per second by leveraging aggregation.
  + KPL is particularly useful when high throughput is required without sacrificing real-time processing.

**Key Metrics and Limits:**

1. **Shards**:
   * Each shard supports:
     + **1 MB per second** write capacity.
     + **1000 records per second**.
   * By having **5 shards**, your total capacity is:
     + **5 MB per second** for writes.
     + **5000 records per second** across all shards.
2. **Batching and Aggregation**:
   * By using the KPL for **batching** and **aggregation**, you can minimize the number of API calls and ensure that you don't breach the **1 MB per record limit**.
   * You could send more than **1000 records per second per shard** using KPL's aggregation feature.
3. **Partition Key Distribution**:
   * Good distribution of partition keys (e.g., hashing the city or device\_id) ensures that no single shard becomes a bottleneck due to a hot partition.

**Summary:**

* **Kinesis SDK**:
  + Provides real-time data ingestion but can lead to a high number of API calls, especially when dealing with small records.
* **Kinesis Producer Library (KPL)**:
  + Reduces the number of API calls by batching and aggregating records, making it suitable for high-throughput applications (more than 1000 records per second) while staying under the 1 MB record limit.
* **Sharding**:
  + The stream has 5 shards, and each shard can handle **1000 records/sec** and **1 MB/sec** of data, providing a total of **5000 records/sec** throughput.
* **Hot Partitioning**:
  + Be aware of partition key skew (e.g., one city generating more traffic), which can overload a shard. Consider a better partitioning strategy to balance the load.

**Lab Guide: Using AWS Kinesis Data Streams with CLI**

**Prerequisites:**

* AWS account with necessary IAM permissions to create Kinesis streams and manage data.
* AWS CLI installed and configured with appropriate access keys.
* Basic knowledge of shell scripting.

**Step 1: Create a Kinesis Data Stream**

1. **Open your terminal.**
2. **Create a new Kinesis stream:**

bash

Copy code

aws kinesis create-stream --stream-name test --shard-count 1

* + This command creates a stream named test with one shard.

1. **Wait for the stream to become active:**
   * Use the following command to check the stream status:

bash

Copy code

aws kinesis describe-stream --stream-name test

* + The status should be ACTIVE before you proceed to the next steps.

**Step 2: Produce Data to the Stream**

1. **Create a new shell script file:**

bash

Copy code

touch kinesis\_producer.sh

chmod +x kinesis\_producer.sh

1. **Open the file in a text editor and add the following content:**

bash

Copy code

#!/bin/bash

# Get the AWS CLI version

aws --version

# PRODUCER

# CLI v2 - Sending user signup event

aws kinesis put-record --stream-name test --partition-key user1 --data "user signup" --cli-binary-format raw-in-base64-out

# CLI v1 - Sending user login event

aws kinesis put-record --stream-name test --partition-key user1 --data "user signup"

aws kinesis put-record --stream-name test --partition-key user1 --data "user login"

1. **Run the script to send records to the Kinesis stream:**

bash

Copy code

./kinesis\_producer.sh

1. **Verify that the data has been sent:**
   * You can check the Kinesis dashboard in the AWS Management Console to see the records in the test stream.

**Step 3: Consume Data from the Stream**

1. **Create a new shell script file for the consumer:**

bash

Copy code

touch kinesis\_consumer.sh

chmod +x kinesis\_consumer.sh

1. **Open the file in a text editor and add the following content:**

bash

Copy code

#!/bin/bash

# CONSUMER

# Describe the stream to find the shard ID

aws kinesis describe-stream --stream-name test

# Get the shard iterator

SHARD\_ITERATOR=$(aws kinesis get-shard-iterator --stream-name test --shard-id shardId-000000000000 --shard-iterator-type TRIM\_HORIZON --query 'ShardIterator' --output text)

# Consume some data using the shard iterator

aws kinesis get-records --shard-iterator $SHARD\_ITERATOR

1. **Run the consumer script to read data from the Kinesis stream:**

bash

Copy code

./kinesis\_consumer.sh

1. **Check the output:**
   * You should see the records that were produced in the earlier step. The output will include the Data, PartitionKey, SequenceNumber, and other metadata.

**Step 4: Clean Up**

1. **Delete the Kinesis stream (optional):**

bash

Copy code

aws kinesis delete-stream --stream-name test

1. **Verify deletion:**
   * Run the describe command to ensure the stream no longer exists:

bash

Copy code

aws kinesis describe-stream --stream-name test

**Key Concepts Covered:**

* **Creating a Kinesis Data Stream**: You learned how to set up a new Kinesis stream and check its status.
* **Producing Data**: You explored how to send records to Kinesis using both CLI v1 and v2 commands.
* **Consuming Data**: You consumed data from the stream using the shard iterator.
* **Data Handling**: You saw how to check and manage Kinesis streams effectively.

**Conclusion:**

In this lab, you successfully created a Kinesis Data Stream, produced data to it, and consumed data from it using the AWS CLI. This fundamental understanding of Kinesis can be expanded into more complex data processing workflows and integrations with other AWS services, such as Lambda, S3, or real-time analytics applications.

Feel free to modify the data you send and explore the capabilities of Kinesis further!

**Lab Guide: Setting Up Kinesis Firehose to Deliver Logs to S3**

**Prerequisites:**

* AWS account with necessary IAM permissions to create Kinesis Firehose, S3, and EC2 resources.
* AWS CLI installed and configured.
* Basic knowledge of shell scripting and AWS services.

**Step 1: Create an S3 Bucket for Log Storage**

1. **Log in to the AWS Management Console.**
2. **Navigate to the S3 service.**
3. **Create a new S3 bucket:**
   * Click on "Create bucket".
   * Enter orderlogss as the bucket name.
   * Choose the region (e.g., us-east-1).
   * Configure other settings as needed and click "Create bucket".
4. **Set up the bucket prefix:**
   * Note the prefix format for storing logs:

sql

Copy code

year=!{timestamp:yyyy}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/

1. **Set up the bucket error output prefix:**
   * Note the error output prefix format:

lua

Copy code

fherroroutputbase/!{firehose:random-string}/!{firehose:error-output-type}/!{timestamp:yyyy/MM/dd}/

**Step 2: Create a Kinesis Firehose Delivery Stream**

1. **Navigate to the Kinesis service in the AWS Management Console.**
2. **Create a new Firehose delivery stream:**
   * Click on "Create delivery stream".
   * Name your stream PurchaseLogs.
   * Choose the source as **Direct Put**.
   * Click "Next".
3. **Configure the destination:**
   * Choose **Amazon S3** as the destination.
   * Select the orderlogss bucket you created earlier.
   * Enter the S3 bucket prefix and error output prefix in their respective fields.
   * Click "Next".
4. **Configure buffering and other settings as needed:**
   * Use default settings for buffering (e.g., buffer size of 5 MB and buffer interval of 300 seconds).
   * Click "Next".
5. **Review the settings and click "Create delivery stream".**
6. **Enable the delivery stream.**

**Step 3: Launch an EC2 Instance**

1. **Navigate to the EC2 service in the AWS Management Console.**
2. **Launch a new EC2 instance:**
   * Choose **Amazon Linux 2** AMI.
   * Select an instance type (e.g., t2.micro).
   * Configure instance details as needed, ensuring it is in the same region as your S3 bucket and Firehose stream.
3. **Assign an IAM Role to the Instance:**
   * Create a new role with the following permissions:
     + AmazonKinesisFirehoseFullAccess
     + AmazonS3FullAccess
     + CloudWatchAgentServerPolicy
   * Assign this role to your EC2 instance during the launch.
4. **Configure security group rules:**
   * Ensure that the security group allows SSH (port 22) access.
5. **Launch the instance.**

**Step 4: Install and Configure the AWS Kinesis Agent**

1. **SSH into your EC2 instance:**

bash

Copy code

ssh -i your-key.pem ec2-user@your-ec2-public-dns

1. **Install the AWS Kinesis Agent:**

bash

Copy code

sudo yum install aws-kinesis-agent -y

1. **Download the log generator script:**

bash

Copy code

wget http://media.sundog-soft.com/AWSBigData/LogGenerator.zip

1. **Unzip the downloaded file:**

bash

Copy code

unzip LogGenerator.zip

1. **Make the log generator script executable:**

bash

Copy code

chmod a+x LogGenerator.py

1. **Create the necessary directory for logs:**

bash

Copy code

mkdir /var/log/cadabra

**Step 5: Configure the Kinesis Agent**

1. **Modify the Kinesis agent configuration file:**

bash

Copy code

sudo vi /etc/aws-kinesis/agent.json

* + Update the configuration as follows:

json

Copy code

{

"cloudwatch.emitMetrics": true,

"kinesis.endpoint": "",

"firehose.endpoint": "firehose.us-east-1.amazonaws.com",

"flows": [

{

"filePattern": "/var/log/cadabra/\*.log",

"deliveryStream": "PurchaseLogs"

}

]

}

* + Save and exit the file.

**Step 6: Start the AWS Kinesis Agent**

1. **Start the Kinesis agent:**

bash

Copy code

sudo systemctl start aws-kinesis-agent

* + Alternatively, you can use:

bash

Copy code

sudo service aws-kinesis-agent start

1. **Enable the Kinesis agent to start on boot:**

bash

Copy code

sudo chkconfig aws-kinesis-agent on

**Step 7: Generate Log Data**

1. **Run the log generator script:**

bash

Copy code

python3 LogGenerator.py 500000

* + This will generate approximately 500,000 log entries.

1. **Monitor the logs being sent to Firehose:**

bash

Copy code

tail -f /var/log/aws-kinesis-agent/aws-kinesis-agent.log

* + You can also monitor the logs generated:

bash

Copy code

tail -f /var/logs/cadabra/

**Step 8: Verify Logs in S3**

1. **Go back to the S3 service in the AWS Management Console.**
2. **Navigate to the orderlogss bucket.**
3. **Check for logs stored in the specified prefix.**
   * You should see logs organized by year, month, day, and hour as specified in your Firehose configuration.

**Step 9: Clean Up**

1. **Stop the EC2 instance if no longer needed.**
2. **Delete the Kinesis Firehose delivery stream:**

bash

Copy code

aws firehose delete-delivery-stream --delivery-stream-name PurchaseLogs

1. **Delete the S3 bucket:**
   * Ensure that the bucket is empty before deletion, then:

bash

Copy code

aws s3 rb s3://orderlogss --force

**Conclusion**

In this lab, you have successfully set up a Kinesis Firehose delivery stream to collect log data and deliver it to an S3 bucket. You configured an EC2 instance to run the AWS Kinesis Agent, which sends logs to Firehose, demonstrating near-real-time log collection and storage.

You can further explore this setup by integrating other data sources or processing the logs stored in S3 using AWS Lambda or AWS Glue for analytics.

**Lab Guide: Working with Amazon EBS Snapshots**

**Prerequisites:**

* An AWS account with necessary IAM permissions to manage EBS volumes and snapshots.
* A running EC2 instance with attached EBS volumes.
* Basic understanding of AWS EC2 and EBS.

**Step 1: Identify Your EBS Volumes**

1. **Log in to the AWS Management Console.**
2. **Navigate to the EC2 service.**
3. **Identify your EBS volumes:**
   * Click on “Volumes” under the “Elastic Block Store” section.
   * Note the details of your EBS volumes, including their IDs (e.g., vol-01e4cf7953360807d, vol-017240d8b87b1e980, etc.) and availability zones (e.g., us-east-1d, us-east-1c).
4. **Check the root volume:**
   * Note the root volume associated with your EC2 instance (e.g., vol-0b5e1c81b43b8b469).

**Step 2: Create a Snapshot of an EBS Volume**

1. **Select the volume you want to snapshot:**
   * For example, choose vol-01e4cf7953360807d in us-east-1c.
2. **Create a snapshot:**
   * Right-click on the volume and select **Create Snapshot**.
   * In the dialog box, provide a description for the snapshot (e.g., "Snapshot of volume vol-01e4cf7953360807d").
   * Click **Create Snapshot**.
3. **Verify the snapshot creation:**
   * Navigate to the “Snapshots” section under the “Elastic Block Store”.
   * You should see the newly created snapshot (e.g., snap-020af8b3f762c08a1).

**Step 3: Create a New EBS Volume from a Snapshot**

1. **Select the snapshot you created:**
   * Locate the snapshot you just created in the “Snapshots” section.
2. **Create a volume from the snapshot:**
   * Right-click on the snapshot and select **Create Volume**.
   * Specify the following:
     + **Availability Zone**: Choose the same availability zone where you want to create the new volume (e.g., us-east-1c).
     + **Size**: Specify the size for the new volume (it can be the same size or larger than the snapshot).
   * Click **Create Volume**.
3. **Verify the new volume creation:**
   * Go back to the “Volumes” section and check for the new volume (e.g., vol-027bbde7cdc2db925).

**Step 4: Attach the New Volume to an EC2 Instance**

1. **Select the EC2 instance:**
   * Go to the “Instances” section and choose the EC2 instance where you want to attach the new volume.
2. **Attach the new volume:**
   * Right-click on the new volume (e.g., vol-027bbde7cdc2db925) and select **Attach Volume**.
   * In the dialog box, select the instance you want to attach the volume to and specify a device name (e.g., /dev/sdf).
   * Click **Attach**.
3. **Verify the volume attachment:**
   * Log in to your EC2 instance and run:

bash

Copy code

lsblk

* + You should see the newly attached volume listed.

**Step 5: Format and Mount the New Volume (if necessary)**

1. **If the volume is empty or unformatted:**
   * Format the new volume (e.g., /dev/sdf):

bash

Copy code

sudo mkfs -t ext4 /dev/sdf

1. **Create a mount point:**

bash

Copy code

sudo mkdir /mnt/new\_volume

1. **Mount the new volume:**

bash

Copy code

sudo mount /dev/sdf /mnt/new\_volume

1. **Verify the mount:**
   * Check if the volume is mounted:

bash

Copy code

df -h

**Step 6: Clean Up Resources**

1. **Unmount the volume (if mounted):**

bash

Copy code

sudo umount /mnt/new\_volume

1. **Detach the volume from the instance:**
   * Go to the “Volumes” section in the EC2 console.
   * Right-click on the volume and select **Detach Volume**.
2. **Delete the EBS volume (optional):**
   * Right-click on the volume and select **Delete Volume** if it is no longer needed.
3. **Delete the snapshot (optional):**
   * Go to the “Snapshots” section, select the snapshot you created, and choose **Delete Snapshot** if it is no longer needed.

**Conclusion**

In this lab, you successfully created an Amazon EBS snapshot from an existing volume, created a new volume from that snapshot, and attached the volume to an EC2 instance. You also learned how to format and mount the new volume, providing you with a solid understanding of how to manage EBS volumes and snapshots effectively.

This knowledge can be extended to backup strategies, disaster recovery plans, and scaling your storage resources as needed in AWS.