RabbitMQ Partitioning with Consumers

i want to try partition type things like kafka in rabbitmq with multiple consumer so tell me code

To achieve Kafka-like partitioning behavior in RabbitMQ, you can simulate partitions by using multiple queues where each queue represents a partition. Then, you can distribute the messages across these partitions based on a key (e.g., a hash key) and use multiple consumers to process the messages. Here's a basic setup using RabbitMQ with Node.js to demonstrate partitioning with multiple consumers.

Steps:

1. **Install Required Dependencies**

```bash
npm install amqplib
```

2. **Publisher**: This script will send messages to different "partitions" (queues) based on a key (partitioning logic).

```javascript
const amqp = require('amqplib');

async function publishMessage() {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();

    const numPartitions = 3;  // Simulate 3 partitions (queues)
    const partitionPrefix = 'partition_';

    // Create partition queues
    for (let i = 0; i < numPartitions; i++) {
      const queue = partitionPrefix + i;
      await channel.assertQueue(queue, { durable: true });
    }

    // Send messages to a specific partition based on a key (round-robin, hash, etc.)
    for (let i = 0; i < 10; i++) {
      const key = i;  // Can be any key, such as user ID, message type, etc.
      const partition = key % numPartitions;
      const queue = partitionPrefix + partition;
      const msg = `Message ${i} to partition ${partition}`;

      channel.sendToQueue(queue, Buffer.from(msg), { persistent: true });
      console.log(`[x] Sent: ${msg}`);
    }

    setTimeout(() => {
      connection.close();
```

```
    }, 500);
  } catch (err) {
    console.error(err);
  }
}

publishMessage();
```

3. **Consumer**: This script will simulate multiple consumers for different partitions. Each consumer listens to a specific partition (queue).

```javascript
const amqp = require('amqplib');

async function consumeMessages(partitionIndex) {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();
    const partitionQueue = `partition_${partitionIndex}`;

    await channel.assertQueue(partitionQueue, { durable: true });
    console.log(`[*] Waiting for messages in ${partitionQueue}. To exit press
CTRL+C`);

    channel.consume(partitionQueue, (msg) => {
      if (msg !== null) {
        console.log(`[x] Received: ${msg.content.toString()} on partition
${partitionIndex}`);
        channel.ack(msg);
      }
    });
  } catch (err) {
    console.error(err);
  }
}

// Simulate multiple consumers, each consuming a specific partition
const partitions = 3;  // Number of partitions
for (let i = 0; i < partitions; i++) {
  consumeMessages(i);  // Each consumer handles a specific partition
}
```

How It Works:

- **Publisher**: The publisher distributes messages across multiple queues (which act as partitions) based on some key. In this example, the partitioning is done using `` `key % numPartitions` ``, where `` `key` `` could be any identifier.
- **Consumer**: Each consumer listens to a specific queue (partition). Multiple consumers can be instantiated, each consuming from a different partition.

Key Notes:

- You can scale consumers by adding more instances, but each instance will be tied to specific partitions.
- This is a basic simulation of Kafka's partitioning model, where each partition has its own queue, and the consumers are assigned to specific partitions.

Let me know if you need help with any further customization or scaling of this solution!

there is sharding or partitioning queues are there for cluster in tht thing i am asking

I see! You're asking about **partitioning or sharding queues in RabbitMQ**, similar to Kafka's partitioning in a **RabbitMQ Cluster** setup, to distribute the load across multiple nodes in a RabbitMQ cluster with multiple consumers.
RabbitMQ doesn't have native partitioning like Kafka, but you can achieve a similar effect using **sharding** or **consistent hashing** by routing messages to different queues across nodes in the cluster. You can configure multiple consumers to subscribe to these queues for distributed processing.

To simulate Kafka-like partitioning in a RabbitMQ cluster, you can use:

1. **Sharded queues** in RabbitMQ, which help with partitioning across a cluster.
2. **Consistent hashing** to distribute messages to queues (shards) based on a message key.
3. Multiple consumers to process messages in parallel from these queues.

Here's how you can simulate it:

Steps:

1. **Set up RabbitMQ Cluster**:
   - First, ensure you have a RabbitMQ cluster with multiple nodes. RabbitMQ clusters provide scalability and redundancy.
   - You can configure a RabbitMQ cluster by setting up multiple RabbitMQ nodes and ensuring that queues are distributed across these nodes.
2. **Sharded Queues**: You can use RabbitMQ's **consistent-hashing exchange** to shard messages across multiple queues (shards).
   - This exchange allows you to distribute messages across queues based on a hash of the routing key (similar to Kafka partitions).

Example:

1. Publisher with Sharding Using a Consistent-Hashing Exchange

```javascript
const amqp = require('amqplib');

async function publishMessage() {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();

    // Declare a consistent-hashing exchange
    const exchange = 'sharded_exchange';
    await channel.assertExchange(exchange, 'x-consistent-hash', { durable: true });

    // Declare shard queues
    const numShards = 3;
    const shardPrefix = 'shard_';

    for (let i = 0; i < numShards; i++) {
      const queue = shardPrefix + i;
      await channel.assertQueue(queue, { durable: true });
```

```
      // Bind each shard queue to the consistent-hashing exchange
      await channel.bindQueue(queue, exchange, '');
    }

    // Publish messages with keys that will be hashed to determine the shard
    for (let i = 0; i < 10; i++) {
      const key = `key_${i}`;  // You can use any key
      const msg = `Message ${i} with key ${key}`;

      // Publish the message with the key, which will be hashed and routed to the correct
shard
      channel.publish(exchange, key, Buffer.from(msg), { persistent: true });
      console.log(`[x] Sent: ${msg}`);
    }

    setTimeout(() => {
      connection.close();
    }, 500);
  } catch (err) {
    console.error(err);
  }
}

publishMessage();
```

2. Consumers for Each Shard

```javascript
const amqp = require('amqplib');

async function consumeMessages(shardIndex) {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();

    const shardQueue = `shard_${shardIndex}`;

    await channel.assertQueue(shardQueue, { durable: true });
    console.log(`[*] Waiting for messages in ${shardQueue}. To exit press CTRL+C`);

    channel.consume(shardQueue, (msg) => {
      if (msg !== null) {
        console.log(`[x] Received: ${msg.content.toString()} on shard ${shardIndex}`);
        channel.ack(msg);
      }
    });
  } catch (err) {
    console.error(err);
  }
}

// Simulate multiple consumers, each consuming a specific shard
const shards = 3;  // Number of shards
for (let i = 0; i < shards; i++) {
  consumeMessages(i);  // Each consumer handles a specific shard
}
```

How This Works:

- **Consistent Hashing**: The `x-consistent-hash` exchange will distribute messages to different queues (shards) based on the hash of the routing key (`key_X` in the example).
  - Messages with similar keys will be hashed to the same shard (queue), ensuring that consumers process messages in a Kafka-like partitioned way.

- **Consumers**: Each consumer listens to one specific shard, just like in a Kafka partition model.

Key Concepts:

- **Sharded Queues**: Each "partition" (shard) is represented by a separate queue. You can scale consumers across shards just like you would in a Kafka system.
- **Consistent Hashing Exchange**: This exchange allows you to use a consistent hashing algorithm to route messages to specific queues based on a key.

Scaling and Fault Tolerance:

- You can have multiple consumers per shard, which gives you parallelism for processing messages from the same shard.
- You can distribute these consumers across different nodes in a RabbitMQ cluster for horizontal scaling.

Setting Up RabbitMQ Cluster with Sharding:

- A RabbitMQ cluster allows you to scale across multiple nodes. When a node in a cluster goes down, queues on that node will fail over to another node in the cluster.
- The **sharded queues** can be distributed across different nodes in the cluster for better load balancing.

Let me know if you'd like more details on setting up the RabbitMQ cluster itself or more advanced configurations!

3 nodes with 10 json data multiple consumer

To implement a RabbitMQ cluster with 3 nodes, where 10 JSON data messages are distributed to multiple consumers, you can use a **sharded queue** setup with consistent hashing. This will simulate partitioning where messages are spread across different queues (shards) based on a key, and multiple consumers can process these messages concurrently.

Steps Overview:

1. **Set up RabbitMQ cluster** with 3 nodes.
2. **Use consistent-hashing exchange** to shard messages across different queues.
3. **Publish JSON data** into RabbitMQ.
4. **Set up multiple consumers** to listen to these shards.

Pre-requisites:

- A RabbitMQ cluster with 3 nodes running.
- Ensure that RabbitMQ's **consistent hashing plugin** is enabled on your cluster:

```bash
rabbitmq-plugins enable rabbitmq_consistent_hash_exchange
```

Setup:

1. Publisher Code: Send 10 JSON data messages across 3 shards.

Here, we'll use a consistent-hash exchange to distribute the JSON messages to different shards based on a key.

```javascript
const amqp = require('amqplib');

const jsonData = [
  { id: 1, name: "John", age: 25 },
  { id: 2, name: "Alice", age: 30 },
  { id: 3, name: "Bob", age: 28 },
  { id: 4, name: "David", age: 22 },
  { id: 5, name: "Eve", age: 35 },
  { id: 6, name: "Frank", age: 40 },
  { id: 7, name: "Grace", age: 23 },
  { id: 8, name: "Hank", age: 27 },
  { id: 9, name: "Ivy", age: 31 },
  { id: 10, name: "Jack", age: 29 }
];

async function publishMessages() {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();

    // Declare a consistent-hashing exchange
    const exchange = 'sharded_exchange';
    await channel.assertExchange(exchange, 'x-consistent-hash', { durable: true });

    // Declare shard queues
    const numShards = 3;
    const shardPrefix = 'shard_';

    for (let i = 0; i < numShards; i++) {
      const queue = shardPrefix + i;
      await channel.assertQueue(queue, { durable: true });
      // Bind each shard queue to the consistent-hashing exchange
      await channel.bindQueue(queue, exchange, '');
    }

    // Publish the 10 JSON messages with a unique key for sharding
    jsonData.forEach((data) => {
      const key = data.id;  // Use the ID as the key for consistent hashing
      const msg = JSON.stringify(data);

      // Publish the message to the consistent hashing exchange
      channel.publish(exchange, key.toString(), Buffer.from(msg), { persistent: true });
      console.log(`[x] Sent: ${msg} with key ${key}`);
    });

    setTimeout(() => {
      connection.close();
    }, 500);
  } catch (err) {
    console.error(err);
  }
}
```

```
publishMessages();
```

Explanation:

- This publisher sends 10 JSON messages, each with a unique key (using `id`), to a **consistent-hash exchange** that distributes the messages across 3 queues (shards) based on the hash of the key.

2. Consumer Code: Multiple consumers to listen to the shard queues and process messages concurrently.

```javascript
const amqp = require('amqplib');

async function consumeMessages(shardIndex) {
  try {
    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();

    const shardQueue = `shard_${shardIndex}`;

    await channel.assertQueue(shardQueue, { durable: true });
    console.log(`[*] Waiting for messages in ${shardQueue}. To exit press CTRL+C`);

    channel.consume(shardQueue, (msg) => {
      if (msg !== null) {
        const jsonData = JSON.parse(msg.content.toString());
        console.log(`[x] Received: ${JSON.stringify(jsonData)} on shard ${shardIndex}`);
        channel.ack(msg);
      }
    });
  } catch (err) {
    console.error(err);
  }
}

// Simulate multiple consumers, each consuming a specific shard
const shards = 3;  // Number of shards
for (let i = 0; i < shards; i++) {
  consumeMessages(i);  // Each consumer handles a specific shard
}
```

Explanation:

- Each consumer listens to a specific shard (queue), where messages are distributed by the consistent-hashing exchange.
- This setup allows you to have multiple consumers for different shards, simulating Kafka's partitioned consumption.

3. RabbitMQ Cluster Setup:

Ensure you have a 3-node RabbitMQ cluster running, with queues distributed across the nodes. You can scale this by configuring consumers on different nodes.
For example, if you have 3 RabbitMQ nodes (`node1`, `node2`, and `node3`), you can distribute the queues across these nodes to balance the load:

1. Connect consumers to different nodes in the cluster.
2. RabbitMQ will handle queue failover and distribution automatically.

Key Concepts:

- **Sharded Queues**: These act as partitions, distributing messages across multiple queues (shards) for parallel consumption.
- **Consistent Hashing**: Used to route messages based on a key to specific queues, ensuring that related messages are sent to the same queue.
- **Multiple Consumers**: Each consumer listens to a specific shard, allowing parallel processing of messages.

This setup mimics Kafka's partitioning behavior in RabbitMQ using consistent hashing, with the messages (JSON data) distributed across the cluster and processed by multiple consumers.
Let me know if you need more help with scaling, or configuring the RabbitMQ cluster itself!

why we dont need three files for different consumers

Good question! While we could create three different files for each consumer in the RabbitMQ setup, it's not necessary because of the flexibility of a **single consumer file** handling multiple queues (shards). Let me explain why:

1. Dynamic Queue Assignment:

Each consumer can dynamically be assigned a different queue (or shard) to consume from. This can be done by simply passing a different shard index (or queue name) when starting the consumer, which is what we did in the `for` loop:

```javascript
for (let i = 0; i < shards; i++) {
  consumeMessages(i);  // Each consumer handles a specific shard
}
```

This creates a separate consumer for each shard without needing separate files. Each instance of the consumer function is assigned a unique shard (queue) to listen to.

2. Concurrency:

By using the same file and function to spawn multiple consumers dynamically, you avoid code duplication. Each instance of the consumer will run independently and handle messages from its respective queue. RabbitMQ ensures each queue gets its own consumer instance.

3. Scaling and Flexibility:

Using a single file to create multiple consumers allows you to:

- Easily scale the number of consumers.
- Adjust the number of queues or shards without modifying multiple files.
- Dynamically adjust consumers based on system load or cluster capacity.

For example, if you wanted to consume from 5 shards instead of 3, you could simply change `const shards = 3;` to `const shards = 5;` without needing extra files.

4. DRY Principle:

Following the **DRY (Don't Repeat Yourself)** principle helps to reduce redundant code. Having one file that dynamically consumes different queues based on parameters keeps your code clean and efficient. If you had three separate files for three consumers, they would have almost identical logic, leading to duplication.

Example with Three Different Consumers:

If you really wanted three separate files, you could have something like this:

- `consumer1.js`: consumes from `shard_0`.
- `consumer2.js`: consumes from `shard_1`.
- `consumer3.js`: consumes from `shard_2`.

However, the logic would be the same for all three files, with only the shard index differing. Instead, it's more efficient to use **one file with a parameterized shard index** like in the previous example.

Conclusion:

Using a single consumer file with dynamic queue assignment is:

- More flexible.
- Easier to scale.
- Keeps the code DRY (avoiding repetition).
- Reduces maintenance overhead since changes need to be made in just one place.

By controlling the shard assignment inside a loop or based on dynamic parameters, we can achieve the same behavior as using multiple files without the redundancy. Let me know if you'd like further clarification!

what things i have to do for metrics of one instance in aws ec2 for each second uisng rabbit mq

To gather and process metrics every second on an AWS EC2 instance using RabbitMQ, you'll need to design a system where metrics (like CPU, memory, disk usage, or custom application-specific metrics) are collected, published to RabbitMQ, and consumed for real-time monitoring or processing.

High-Level Steps:

1. **Set Up Metrics Collection**:

    - Use Node.js (or your preferred language) to gather system-level metrics every second from the EC2 instance.
    - You can use libraries like **os-utils** or **node-os-monitor** to capture system metrics like CPU usage, memory, and disk statistics.
2. **Publish Metrics to RabbitMQ**:
    - Use RabbitMQ as the message broker to send these metrics to a queue every second.
    - Each EC2 instance can publish its own metrics with a unique identifier (e.g., instance ID) for easier tracking.
3. **Consumer Setup**:
    - Set up RabbitMQ consumers that will read the metrics from the queue and process them in real-time (e.g., store in a database, trigger alerts, or visualize using a dashboard).
4. **Optionally Store/Process Metrics**:
    - Store metrics in a database for long-term monitoring (e.g., InfluxDB, Elasticsearch).
    - Visualize or aggregate the data in real-time using a dashboard (e.g., Grafana, Kibana).

Steps in Detail:

1. Setting up the EC2 instance:

- Make sure you have a working **AWS EC2 instance** running **Ubuntu** (or any other OS).
- Install RabbitMQ on your EC2 instance:

```bash
sudo apt update
sudo apt install rabbitmq-server
sudo systemctl enable rabbitmq-server
sudo systemctl start rabbitmq-server
```

2. Collect Metrics using Node.js:

You can collect metrics every second using the `os-utils` package in Node.js:

```bash
npm install os-utils amqplib
```

```javascript
const os = require('os-utils');
const amqp = require('amqplib');

async function sendMetrics() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();
  const queue = 'metrics_queue';

  // Ensure the queue exists
  await channel.assertQueue(queue, { durable: true });

  // Function to collect metrics and publish to RabbitMQ every second
  setInterval(() => {
    os.cpuUsage((cpuUsage) => {
```

```
      const metrics = {
        instanceId: 'i-12345678', // Replace with actual EC2 instance ID
        cpuUsage: cpuUsage * 100, // CPU usage in percentage
        freeMemory: os.freemem(), // Free memory in MB
        totalMemory: os.totalmem(), // Total memory in MB
        platform: os.platform(), // OS platform (e.g., Linux)
        uptime: os.sysUptime(), // System uptime in seconds
        timestamp: new Date().toISOString(),
      };

      // Convert metrics to string for sending
      const message = JSON.stringify(metrics);
      channel.sendToQueue(queue, Buffer.from(message));
      console.log(`[x] Sent metrics: ${message}`);
    });
  }, 1000); // Collect every second
}

sendMetrics().catch(console.error);
```

3. RabbitMQ Consumer to Process Metrics:

Now, set up consumers to process or store the metrics from the RabbitMQ queue.

```javascript
const amqp = require('amqplib');

async function consumeMetrics() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();
  const queue = 'metrics_queue';

  await channel.assertQueue(queue, { durable: true });

  console.log('[*] Waiting for metrics. To exit press CTRL+C');

  channel.consume(queue, (msg) => {
    if (msg !== null) {
      const metrics = JSON.parse(msg.content.toString());
      console.log(`[x] Received metrics: ${JSON.stringify(metrics)}`);

      // Example: Process metrics here (e.g., store in DB, trigger alerts, etc.)
      // You can also store these metrics in a time-series database like InfluxDB or
display them on a real-time dashboard.

      channel.ack(msg);
    }
  });
}

consumeMetrics().catch(console.error);
```

4. Optionally Storing Metrics:

- You can extend the consumer to store the collected metrics in a database like **InfluxDB**, **Elasticsearch**, or **MongoDB** for real-time analysis.
- Here's a simple example of how you can push the metrics to InfluxDB:

```javascript
const Influx = require('influx');
const influx = new Influx.InfluxDB({
  host: 'localhost',
  database: 'metrics_db',
  schema: [
    {
      measurement: 'system_metrics',
      fields: {
        cpuUsage: Influx.FieldType.FLOAT,
        freeMemory: Influx.FieldType.FLOAT,
        totalMemory: Influx.FieldType.FLOAT,
        uptime: Influx.FieldType.INTEGER,
      },
      tags: ['instanceId', 'platform'],
    },
  ],
});

async function consumeMetrics() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();
  const queue = 'metrics_queue';

  await channel.assertQueue(queue, { durable: true });
  console.log('[*] Waiting for metrics. To exit press CTRL+C');

  channel.consume(queue, async (msg) => {
    if (msg !== null) {
      const metrics = JSON.parse(msg.content.toString());
      console.log(`[x] Received metrics: ${JSON.stringify(metrics)}`);

      // Insert metrics into InfluxDB
      await influx.writePoints([
        {
          measurement: 'system_metrics',
          tags: { instanceId: metrics.instanceId, platform: metrics.platform },
          fields: {
            cpuUsage: metrics.cpuUsage,
            freeMemory: metrics.freeMemory,
            totalMemory: metrics.totalMemory,
            uptime: metrics.uptime,
          },
          timestamp: new Date(metrics.timestamp),
        },
      ]);

      channel.ack(msg);
    }
  });
}

consumeMetrics().catch(console.error);
```

5. Real-Time Visualization:

- You can visualize the collected metrics using **Grafana**, which supports RabbitMQ as a data source or can connect to time-series databases like **InfluxDB**.
- Set up alerts in Grafana for real-time monitoring and threshold notifications.

Summary of the Setup:

1. **Metrics Collection**: Every second, metrics like CPU, memory, and uptime are collected from the EC2 instance using Node.js and `os-utils`.
2. **Publish to RabbitMQ**: These metrics are published to a RabbitMQ queue (`metrics_queue`).

3. **Consumer**: RabbitMQ consumers read the metrics from the queue in real-time.
4. **Process/Store**: The consumer can process these metrics, store them in a database (e.g., InfluxDB), or trigger alerts.
5. **Visualization**: Optionally, visualize the metrics using a tool like **Grafana**.

This setup provides a real-time monitoring solution for gathering and processing metrics every second on an AWS EC2 instance using RabbitMQ. Let me know if you need help with specific integrations or further customization!

how to store this data in rds

To store the metrics data collected from the EC2 instance into an **Amazon RDS** (Relational Database Service), you need to perform the following steps:

Steps Overview:

1. **Set up an Amazon RDS instance** (MySQL/PostgreSQL/etc.).
2. **Connect Node.js to RDS** using a MySQL/PostgreSQL client.
3. **Create a table in the RDS database** to store the metrics.
4. **Modify the RabbitMQ consumer** to insert the collected metrics into the RDS database.

Detailed Steps:

1. Set up Amazon RDS:

- Log in to your AWS Management Console and navigate to the **RDS** service.
- Create a new RDS instance (e.g., MySQL or PostgreSQL). Ensure that the instance is publicly accessible if your EC2 instance and RDS are not in the same VPC.
- Configure the security group to allow traffic from your EC2 instance (or the VPC) to the RDS instance on the database port (e.g., 3306 for MySQL).

Once the RDS instance is up, note down the **RDS endpoint**, **database name**, **username**, and **password**.

2. Install MySQL/PostgreSQL Client:

To connect to RDS, install the respective client library in Node.js.
For **MySQL**:

```bash
npm install mysql2
```

For **PostgreSQL**:

```bash
npm install pg
```

3. Create a Table in the RDS Database:

Connect to your RDS instance and create a table to store the metrics. You can do this using a database client like **MySQL Workbench** (for MySQL) or **pgAdmin** (for PostgreSQL), or directly using SQL commands.
For MySQL:

```sql
CREATE DATABASE metrics_db;

USE metrics_db;

CREATE TABLE system_metrics (
  id INT AUTO_INCREMENT PRIMARY KEY,
  instance_id VARCHAR(50),
  cpu_usage FLOAT,
  free_memory FLOAT,
  total_memory FLOAT,
  uptime INT,
  platform VARCHAR(50),
  timestamp DATETIME
);
```

For PostgreSQL:

```sql
CREATE DATABASE metrics_db;

\c metrics_db

CREATE TABLE system_metrics (
  id SERIAL PRIMARY KEY,
  instance_id VARCHAR(50),
  cpu_usage FLOAT,
  free_memory FLOAT,
  total_memory FLOAT,
  uptime INT,
  platform VARCHAR(50),
  timestamp TIMESTAMP
);
```

4. Modify RabbitMQ Consumer to Insert into RDS:

Now, update the RabbitMQ consumer to store the metrics in the RDS database. Below is an example of how you can modify the consumer for **MySQL**.
MySQL Example:

```javascript
const amqp = require('amqplib');
const mysql = require('mysql2');

// Create a connection pool to the RDS instance
const connection = mysql.createPool({
  host: 'your-rds-endpoint.amazonaws.com',
  user: 'your-username',
  password: 'your-password',
  database: 'metrics_db',
});

async function consumeMetrics() {
  const amqpConnection = await amqp.connect('amqp://localhost');
  const channel = await amqpConnection.createChannel();
  const queue = 'metrics_queue';

  await channel.assertQueue(queue, { durable: true });
  console.log('[*] Waiting for metrics. To exit press CTRL+C');

  channel.consume(queue, (msg) => {
    if (msg !== null) {
      const metrics = JSON.parse(msg.content.toString());
      console.log(`[x] Received metrics: ${JSON.stringify(metrics)}`);

      // Insert metrics into RDS (MySQL)
      const query = `
        INSERT INTO system_metrics (instance_id, cpu_usage, free_memory, total_memory,
uptime, platform, timestamp)
        VALUES (?, ?, ?, ?, ?, ?, ?)
      `;

      connection.query(query, [
        metrics.instanceId,
        metrics.cpuUsage,
        metrics.freeMemory,
        metrics.totalMemory,
        metrics.uptime,
        metrics.platform,
        new Date(metrics.timestamp)
      ], (error, results) => {
        if (error) {
          console.error("Error inserting into RDS:", error);
        } else {
          console.log("Inserted metrics into RDS successfully:", results.insertId);
        }
      });

      channel.ack(msg);
    }
  });
}

consumeMetrics().catch(console.error);
```

Explanation:

- This consumer reads messages from RabbitMQ, parses them, and inserts the data into the RDS table.
- The MySQL connection pool is used to efficiently manage connections to the RDS database.

If you are using **PostgreSQL**, the code would look similar, but you'd replace the MySQL client with the `pg` client.

PostgreSQL Example:

```javascript
const amqp = require('amqplib');
const { Pool } = require('pg');

// Create a connection pool to the RDS instance
const pool = new Pool({
  host: 'your-rds-endpoint.amazonaws.com',
  user: 'your-username',
  password: 'your-password',
  database: 'metrics_db',
});

async function consumeMetrics() {
  const amqpConnection = await amqp.connect('amqp://localhost');
  const channel = await amqpConnection.createChannel();
  const queue = 'metrics_queue';

  await channel.assertQueue(queue, { durable: true });
  console.log('[*] Waiting for metrics. To exit press CTRL+C');

  channel.consume(queue, async (msg) => {
    if (msg !== null) {
      const metrics = JSON.parse(msg.content.toString());
      console.log(`[x] Received metrics: ${JSON.stringify(metrics)}`);

      // Insert metrics into RDS (PostgreSQL)
      const query = `
        INSERT INTO system_metrics (instance_id, cpu_usage, free_memory, total_memory,
uptime, platform, timestamp)
        VALUES ($1, $2, $3, $4, $5, $6, $7)
      `;

      try {
        const res = await pool.query(query, [
          metrics.instanceId,
          metrics.cpuUsage,
          metrics.freeMemory,
          metrics.totalMemory,
          metrics.uptime,
          metrics.platform,
          new Date(metrics.timestamp)
        ]);
        console.log("Inserted metrics into RDS successfully:", res.rowCount);
      } catch (error) {
        console.error("Error inserting into RDS:", error);
      }

      channel.ack(msg);
    }
  });
}

consumeMetrics().catch(console.error);
```

5. Security Considerations:

- Ensure your **EC2 instance** is allowed to communicate with the **RDS instance** by modifying the RDS security group.
- Use **IAM roles** for more secure authentication between your EC2 and RDS instance (instead of storing credentials in your code).

6. Scaling:

- You can easily scale this system by having more consumers on different EC2 instances consuming metrics from RabbitMQ and storing them in RDS.

Summary:

- **RabbitMQ** acts as the message broker to gather metrics every second from the EC2 instance.
- The **consumer** reads these metrics and stores them in an **Amazon RDS** database.
- **MySQL or PostgreSQL** can be used to persist these metrics for further analysis, visualization, or alerting.

Let me know if you need any further clarifications or help with any step!