

I want

1 - to create MSK provisioned cluster (name - shan-msk-cluster) with new VPC with 2 brokers and 2 subnets in us-east-1a and 1b, with broker t2.small, zones = 2, Storage = 10GB, default Cluster configuration

2 - a new Security groups to allow related ports with broker with SSL/IAM/TLS SASL/SCRAM access as mentioned in

<https://docs.aws.amazon.com/msk/latest/developerguide/port-info.html>

use the following port numbers so that Amazon MSK can communicate

To communicate with brokers in plaintext, use port 9092.

To communicate with brokers with TLS encryption, use port 9094.

To communicate with brokers with SASL/SCRAM, use port 9096 for:

To communicate with brokers in a cluster that is set up to use:

To communicate with Apache ZooKeeper by using TLS encryption, use:

3 - for access control use "IAM role-based authentication" and "SASL/SCRAM authentication" 

4 - Encrypt data at rest, the data need to be encrypted with new customer KMS

5 - for Broker log delivery, create and deliver to Amazon CloudWatch Logs and the CloudWatch logs to be encrypted with new KMS

15 web pages

Below is a step-by-step guide to achieving your requirements for creating an Amazon MSK (Managed Streaming for Apache Kafka) provisioned cluster with the specified configurations. Since I can't execute the setup for you, I'll provide detailed instructions that you can follow using the AWS Management

Console, AWS CLI, or Infrastructure as Code (e.g., Terraform). For simplicity, I'll focus on the AWS Management Console approach, but I'll also Grok **beta** support for AWS CLI or Terraform where applicable.

Step 1: Create an MSK Provisioned Cluster

Cluster Name: `shan-msk-cluster`

VPC: New VPC with 2 subnets in `us-east-1a` and `us-east-1b`

Brokers: 2 brokers, `t2.small`, 2 zones, 10GB storage

Configuration: Default cluster configuration

Instructions (AWS Management Console):

1. Sign in to AWS Management Console:

- Navigate to the Amazon MSK console: <https://console.aws.amazon.com/msk/> .

2. Create a New VPC:

- Go to the VPC service (<https://console.aws.amazon.com/vpc/>).

- Click **Create VPC**:

- Name: `shan-msk-vpc`
- IPv4 CIDR block: `10.0.0.0/16`
- Click **Create VPC**.

- Create 2 subnets:

- Subnet 1:
 - Name: `shan-msk-subnet-1`
 - Availability Zone: `us-east-1a`
 - IPv4 CIDR block: `10.0.0.0/24`
- Subnet 2:
 - Name: `shan-msk-subnet-2`
 - Availability Zone: `us-east-1b`
 - IPv4 CIDR block: `10.0.1.0/24`
- Click **Create subnet** for each.

3. Create the MSK Cluster:

- In the MSK console, click **Create cluster**.
- **Creation method:** Select **Custom create**.
- **Cluster name:** Enter `shan-msk-cluster` .

- **Cluster type:** Select **Provisioned**.
- **Apache Kafka version:** Choose the latest **Apache Kafka** (e.g., **3.7.x** as of March 2025, if supported).
- **Broker settings:**
 - Broker type: **kafka.t2.small**
 - Number of brokers: **2**
 - Number of zones: **2**
 - Storage per broker: **10 GB**
- **Networking:**
 - VPC: Select **shan-msk-vpc**.
 - Subnets: Select **shan-msk-subnet-1** (us-east-1a) and **shan-msk-subnet-2** (us-east-1b).
- **Configuration:** Choose **Default configuration** (you can customize later if needed).
- Proceed to the next steps (security and monitoring will be configured below).

CLI Alternative:

```
aws msk create-cluster \
--cluster-name "shan-msk-cluster" \
--kafka-version "3.7.x" \
--number-of-broker-nodes 2 \
--broker-node-group-info "{
  \"InstanceType\": \"kafka.t2.small\",
  \"StorageInfo\": {\"EbsStorageInfo\": {\"VolumeSize\": 10}},
  \"ClientSubnets\": [\"subnet-id-1\", \"subnet-id-2\"],
  \"ConnectivityInfo\": {\"PublicAccess\": {\"Type\": \"DISABLED\"}}
}" \
--region us-east-1
```

Replace **subnet-id-1** and **subnet-id-2** with the actual subnet IDs from your VPC creation.

Step 2: Create a New Security Group

Purpose: Allow communication with brokers using specified ports for SSL/IAM/TLS and SASL/SCRAM access.

Reference: [Amazon MSK Port Info](#)

Instructions (AWS Management Console):

1. Create a Security Group:

Grok 3 **beta** ▾



- Go to the VPC console, under **Security**, click **Security Groups**.
- Click **Create security group**:
 - Name: `shan-msk-sg`
 - Description: "Security group for shan-msk-cluster"
 - VPC: Select `shan-msk-vpc`.

2. Add Inbound Rules:

- Add the following rules to allow traffic from within AWS (adjust source as needed, e.g., self-referencing the security group or a specific client SG):
 - **Port 9092 (Plaintext)**: Type: Custom TCP, Port: 9092, Source: `shan-msk-sg` (self-referencing).
 - **Port 9094 (TLS)**: Type: Custom TCP, Port: 9094, Source: `shan-msk-sg`.
 - **Port 9096 (SASL/SCRAM)**: Type: Custom TCP, Port: 9096, Source: `shan-msk-sg`.
 - **Port 9098 (IAM)**: Type: Custom TCP, Port: 9098, Source: `shan-msk-sg`.
 - **Port 2181 (ZooKeeper Plaintext)**: Type: Custom TCP, Port: 2181, Source: `shan-msk-sg`.
 - **Port 2182 (ZooKeeper TLS)**: Type: Custom TCP, Port: 2182, Source: `shan-msk-sg`.
- For public access (if needed later), you can add rules for 9194, 9196, 9198 with a specific source (e.g., your IP).

3. Associate with MSK Cluster:

- During cluster creation (Step 1), under **Networking**, select `shan-msk-sg` as the security group.

CLI Alternative:

```
aws ec2 create-security-group \
--group-name "shan-msk-sg" \
--description "Security group for shan-msk-cluster" \
--vpc-id "vpc-id" \
--region us-east-1
```

```
aws ec2 authorize-security-group-ingress \
--group-id "sg-id" \
--protocol tcp \
--port 9092 \
--source-group "sg-id" \
--region us-east-1
```

Repeat for ports 9094, 9096, 9098, 2181, and 2182, replacing `vpc-id` and `sg-id` with actual IDs.

Dash



Grok 3 beta ▾



Step 3: Configure Access Control

Access Control: Enable both "IAM role-based authentication" and "SASL/SCRAM authentication".

Instructions (AWS Management Console):

1. During Cluster Creation:

- In the MSK console, under **Security settings** during cluster creation:
 - **Authentication:**
 - Check **IAM role-based authentication** (port 9098).
 - Check **SASL/SCRAM authentication** (port 9096).
 - For SASL/SCRAM, you'll need to create credentials in AWS Secrets Manager (see below).

2. Set Up SASL/SCRAM Credentials:

- Go to AWS Secrets Manager (<https://console.aws.amazon.com/secretsmanager/>).
- Click **Store a new secret**:
 - Secret type: Other type of secret.
 - Key/value pairs: Enter `username` and `password` (e.g., `username: shan-user` , `password: your-strong-password`).
 - Secret name: `shan-msk-sasl-secret` .
 - Click **Next** and store the secret.
- Associate with the cluster:
 - After the cluster is created, go to the cluster details, click **Actions > Associate credentials**, and select `shan-msk-sasl-secret` .

3. IAM Role for Clients:

- Create an IAM role with the `AWSMSKExecutionRole` policy for clients to use IAM authentication.

CLI Alternative:

```
aws msk update-security \
--cluster-arn "cluster-arn" \
--client-authentication "{
  \"Sasl\": {\"Scram\": {\"Enabled\": true}, \"Iam\": {\"Enabled\": true}}}
```

```
}" \
--region us-east-1
```

Grok 3 beta ▾



bash

Step 4: Encrypt Data at Rest with a New Customer KMS Key

Requirement: Data encrypted with a new customer-managed KMS key.

Instructions (AWS Management Console):

1. Create a KMS Key:

- Go to AWS KMS (<https://console.aws.amazon.com/kms/>).
- Click **Create key**:
 - Key type: Symmetric.
 - Key usage: Encrypt and decrypt.
 - Key name: `shan-msk-kms-key`.
 - Add your AWS account as the key administrator and key user.
 - Click **Create key**.

2. Configure MSK Cluster Encryption:

- During cluster creation, under **Encryption**:
 - Select **Use customer-managed key**.
 - Choose `shan-msk-kms-key` from the dropdown.
- This ensures data at rest is encrypted with your KMS key.

CLI Alternative:

bash

```
aws msk create-cluster \
--encryption-info "{
  \"EncryptionAtRest\": {\"DataVolumeKMSKeyId\": \"arn:aws:kms:us-east-1:account-id:key/key-id\"}
}" \
--region us-east-1
```

Add this to the `create-cluster` command in Step 1.

Step 5: Broker Log Delivery to CloudWatch Logs

Grok 3 beta ▾



Requirement: Deliver broker logs to Amazon CloudWatch Logs, encrypted with a new KMS key.

Instructions (AWS Management Console):

1. Create a New KMS Key for CloudWatch Logs:

- In KMS, create another key:
 - Name: `shan-cloudwatch-kms-key` .
 - Key type: Symmetric.
 - Key usage: Encrypt and decrypt.
 - Add your account as administrator and user.

2. Create a CloudWatch Log Group:



- Name: `shan-msk-logs` .
- Retention: Set as needed (e.g., 30 days).
- Encryption: Select `shan-cloudwatch-kms-key` .

3. Configure MSK Log Delivery:

- During cluster creation, under **Monitoring**:
 - Check **Deliver to Amazon CloudWatch Logs**.
 - Select `shan-msk-logs` as the log group.
- After creation, logs will be encrypted with the specified KMS key.

CLI Alternative:

bash

```
aws msk create-cluster \
--logging-info "{
  \"BrokerLogs\": {
    \"CloudWatchLogs\": {\"Enabled\": true, \"LogGroup\": \"shan-msk-logs\"}
  }
}" \
--region us-east-1
```

Ensure the log group is created beforehand with encryption:

bash

```
aws logs create-log-group \
  --log-group-name "shan-msk-logs" \
  --kms-key-id "arn:aws:kms:us-east-1:account-id:key/key-id" \
  --region us-east-1
```



Final Steps

1. Review and Create:

- In the MSK console, review all settings (networking, security, encryption, monitoring) and click **Create cluster**.
- Cluster creation takes ~20-30 minutes. Check the status in the MSK console.

2. Verify Setup:

- Once active, verify:
 - Security group rules in VPC console.
 - IAM and SASL/SCRAM authentication in cluster details.
 - Encryption key in KMS.
 - Logs in CloudWatch under `shan-msk-logs` .

Let me know if you need help with Terraform code or further clarification!

i want cdk code for the steps mentioned

Below is an example of AWS Cloud Development Kit (CDK) code in TypeScript to create the Amazon MSK (Managed Streaming for Apache Kafka) cluster with all the specified requirements. The CDK allows you to define your infrastructure as code and deploy it programmatically. This code assumes you have the AWS CDK installed and configured with appropriate credentials.

Prerequisites

- Install Node.js and AWS CDK: `npm install -g aws-cdk`
- Initialize a CDK project: `cdk init app --language typescript`
- Install required CDK modules: `npm install @aws-cdk/aws-msk @aws-cdk/aws-ec2 @aws-cdk/aws-kms @aws-cdk/aws-logs @aws-cdk/aws-secretsmanager`

CDK Code

Create or update `lib/cdk-stack.ts` with the following code:



typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';
import * as secretsmanager from 'aws-cdk-lib/aws-secretsmanager';

export class MskCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Step 1: Create a new VPC with 2 subnets in us-east-1a and us-east-1b
    const vpc = new ec2.Vpc(this, 'ShanMskVpc', {
      vpcName: 'shan-msk-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2, // Ensures 2 AZs (us-east-1a and us-east-1b)
      subnetConfiguration: [
        {
          cidrMask: 24,
          name: 'shan-msk-subnet',
          subnetType: ec2.SubnetType.PRIVATE_ISOLATED, // MSK typically uses private subnets
        },
      ],
    });
  }

  // Step 2: Create a Security Group for MSK
  const mskSecurityGroup = new ec2.SecurityGroup(this, 'ShanMskSecurityGroup', {
    vpc,
    securityGroupName: 'shan-msk-sg',
    description: 'Security group for shan-msk-cluster',
    allowAllOutbound: true,
  });

  // Add inbound rules for required ports (self-referencing SG for internal access)
  const ports = [
    9092, // Plaintext
    9094, // TLS
    9096, // SASL/SCRAM
    9098, // IAM
    2181, // ZooKeeper Plaintext
    2182, // ZooKeeper TLS
  ];
}
```



```
ports.forEach((port) => {
  mskSecurityGroup.addIngressRule( Grok 3 beta ▾
    mskSecurityGroup, // Self-referencing SG
    ec2.Port.tcp(port),
    `Allow port ${port} for MSK communication`
  );
});

// Step 3: Create a KMS Key for Data Encryption
const mskKmsKey = new kms.Key(this, 'ShanMskKmsKey', {
  alias: 'shan-msk-kms-key',
  description: 'KMS key for MSK data encryption',
  enableKeyRotation: true,
});

// Step 4: Create SASL/SCRAM Credentials in Secrets Manager
const saslSecret = new secretsmanager.Secret(this, 'ShanMskSaslSecret', {
  secretName: 'shan-msk-sasl-secret',
  description: 'SASL/SCRAM credentials for shan-msk-cluster',
  generateSecretString: {
    secretStringTemplate: JSON.stringify({ username: 'shan-user' }),
    generateStringKey: 'password',
    passwordLength: 16,
    excludePunctuation: true,
  },
});

// Step 5: Create a KMS Key for CloudWatch Logs Encryption
const cloudwatchKmsKey = new kms.Key(this, 'ShanCloudwatchKmsKey', {
  alias: 'shan-cloudwatch-kms-key',
  description: 'KMS key for CloudWatch Logs encryption',
  enableKeyRotation: true,
});

// Step 6: Create a CloudWatch Log Group
const logGroup = new logs.LogGroup(this, 'ShanMskLogGroup', {
  logGroupName: 'shan-msk-logs',
  retention: logs.RetentionDays.ONE_MONTH,
  encryptionKey: cloudwatchKmsKey,
});

// Step 7: Create the MSK Cluster
const mskCluster = new msk.CfnCluster(this, 'ShanMskCluster', {
  clusterName: 'shan-msk-cluster',
  kafkaVersion: '3.7.x', // Use the latest version supported by CDK/MSK
  numberOfWorkers: 2,
  workerNodeGroupInfo: {
    instanceType: 'kafka.t2.small',
    storageInfo: {
      ebsStorageInfo: {
        volumeSize: 100
      }
    }
  }
});
```

```
volumeSize: 10, // 10 GB
  },
  },
  clientSubnets: vpc.privateSubnets.map((subnet) => subnet.subnetId),
  securityGroups: [mskSecurityGroup.securityGroupId],
},
encryptionInfo: {
  encryptionAtRest: {
    dataVolumeKmsKeyId: mskKmsKey.keyArn,
  },
},
clientAuthentication: {
  sasl: {
    scram: { enabled: true },
    iam: { enabled: true },
  },
},
loggingInfo: {
  brokerLogs: {
    cloudWatchLogs: {
      enabled: true,
      logGroup: logGroup.logGroupName,
    },
  },
},
});
}

// Associate SASL/SCRAM Secret with the MSK Cluster
const secretTargetAttachment = new secretsmanager.CfnSecretTargetAttachment(
  this,
  'ShanMskSecretAttachment',
{
  secretId: saslSecret.secretArn,
  targetId: mskCluster.attrArn,
  targetType: 'AWS::MSK::Cluster',
}
);
}

// Ensure the secret attachment happens after the cluster is created
secretTargetAttachment.node.addDependency(mskCluster);

// Output useful information
new cdk.CfnOutput(this, 'ClusterArn', {
  value: mskCluster.attrArn,
  description: 'ARN of the MSK Cluster',
});
new cdk.CfnOutput(this, 'SaslSecretArn', {
  value: saslSecret.secretArn,
  description: 'ARN of the SASL/SCRAM Secret',
});
```

```

new cdk.CfnOutput(this, 'LogGroupName', {
  value: logGroup.logGroupName,      Grok 3 beta ▾
  description: 'CloudWatch Log Group Name',
});
}

const app = new cdk.App();
new MskCdkStack(app, 'MskCdkStack', {
  env: { region: 'us-east-1' },
});

```



Explanation of the Code

1. VPC and Subnets:

- A new VPC (`shan-msk-vpc`) is created with 2 private subnets in `us-east-1a` and `us-east-1b` .

2. Security Group:

- A security group (`shan-msk-sg`) is created with inbound rules for ports 9092, 9094, 9096, 9098, 2181, and 2182, allowing self-referential traffic (within the SG).

3. KMS Keys:

- Two customer-managed KMS keys are created: one for MSK data encryption (`shan-msk-kms-key`) and one for CloudWatch Logs encryption (`shan-cloudwatch-kms-key`).

4. SASL/SCRAM Credentials:

- A secret (`shan-msk-sasl-secret`) is created in Secrets Manager with a username and auto-generated password, then associated with the MSK cluster.

5. CloudWatch Logs:

- A log group (`shan-msk-logs`) is created with encryption using the `shan-cloudwatch-kms-key` .

6. MSK Cluster:

- The `shan-msk-cluster` is provisioned with 2 brokers (`kafka.t2.small`), 10GB storage, IAM and SASL/SCRAM authentication, and encrypted data at rest.
- Logs are delivered to the CloudWatch log group.

Deploying the Stack

1. Compile the Code:

bash



```
npm run build
```

Grok 3 **beta** ▾



2. Synthesize the CloudFormation Template:

bash

```
cdk synth
```



3. Deploy the Stack:

bash

```
cdk deploy
```



- Follow the prompts to confirm the deployment. This will create all resources in your AWS account.

4. Destroy the Stack (Optional):

If you need to tear down the resources:

bash

```
cdk destroy
```



Notes

- Kafka Version:** Replace '`3.7.x`' with the latest version supported by MSK at the time of deployment (check AWS documentation or CDK API).
- Permissions:** Ensure your AWS credentials have sufficient permissions (e.g., `kms:CreateKey` , `ec2:CreateVpc` , `msk:CreateCluster` , etc.).
- Public Access:** The code disables public access by default. To enable ports 9194, 9196, or 9198 for public access, modify the `brokerNodeGroupInfo` with `connectivityInfo` :

typescript

```
connectivityInfo: {  
  publicAccess: { type: 'SERVICE_PROVIDED_EIPS' },  
},
```



Then add corresponding ingress rules to the security group.



Let me know if you need adjustments or further assistance!

there are 2 ecs service which will put data to msk - say `ecs-ingestor-service` and `report-service` and `gremlin` or `beta` or, how to add ingress rule for that

To allow two ECS services (`ecs-ingestor-service` and `report-service`) to communicate with the Amazon MSK cluster (`shan-msk-cluster`), you need to update the security group associated with the MSK cluster (`shan-msk-sg`) to include ingress rules that permit traffic from the security groups of these ECS services. This ensures that the ECS services can send data to the MSK brokers over the specified ports (e.g., 9092 for plaintext, 9094 for TLS, 9096 for SASL/SCRAM, 9098 for IAM).

Since your setup already includes a security group for MSK, I'll modify the CDK code to:

1. Define security groups for the ECS services.
2. Add ingress rules to the MSK security group to allow traffic from the ECS services' security groups.

Below is the updated CDK code, assuming the ECS services are running in the same VPC (`shan-msk-vpc`). If the ECS services don't exist yet, I'll create placeholder security groups for them, which you can later associate with the ECS services.

Updated CDK Code

Update `lib/cdk-stack.ts` with the following:

typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';
import * as secretsmanager from 'aws-cdk-lib/aws-secretsmanager';

export class MskCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Step 1: Create a new VPC with 2 subnets in us-east-1a and us-east-1b
    const vpc = new ec2.Vpc(this, 'ShanMskVpc', {
      vpcName: 'shan-msk-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2,
      subnetConfiguration: [

```

```
{  
  cidrMask: 24,           Grok 3  beta  ▾  
  name: 'shan-msk-subnet',  
  subnetType: ec2.SubnetType.PRIVATE_ISOLATED,  
},  
],  
});  
  
// Step 2: Create a Security Group for MSK  
const mskSecurityGroup = new ec2.SecurityGroup(this, 'ShanMskSecurityGroup', {  
  vpc,  
  securityGroupName: 'shan-msk-sg',  
  description: 'Security group for shan-msk-cluster',  
  allowAllOutbound: true,  
});  
  
// Define ports for MSK communication  
const mskPorts = [  
  9092, // Plaintext  
  9094, // TLS  
  9096, // SASL/SCRAM  
  9098, // IAM  
  2181, // ZooKeeper Plaintext  
  2182, // ZooKeeper TLS  
];  
  
// Self-referencing rules for MSK internal communication  
mskPorts.forEach((port) => {  
  mskSecurityGroup.addIngressRule(  
    mskSecurityGroup,  
    ec2.Port.tcp(port),  
    `Allow port ${port} for MSK internal communication`  
  );  
});  
  
// Step 3: Create Security Groups for ECS Services  
const ingestorSecurityGroup = new ec2.SecurityGroup(this, 'IngestorSecurityGroup', {  
  vpc,  
  securityGroupName: 'ecs-ingestor-sg',  
  description: 'Security group for ecs-ingestor-service',  
  allowAllOutbound: true,  
});  
  
const reportSecurityGroup = new ec2.SecurityGroup(this, 'ReportSecurityGroup', {  
  vpc,  
  securityGroupName: 'ecs-report-sg',  
  description: 'Security group for report-service',  
  allowAllOutbound: true,  
});
```



```
// Step 4: Add Ingress Rules to MSK Security Group for ECS Services
mskPorts.forEach((port) => {
  // Allow traffic from ecs-ingestor-service
  mskSecurityGroup.addIngressRule(
    ingestorSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from ecs-ingestor-service`
  );

  // Allow traffic from report-service
  mskSecurityGroup.addIngressRule(
    reportSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from report-service`
  );
});

// Step 5: Create a KMS Key for Data Encryption
const mskKmsKey = new kms.Key(this, 'ShanMskKmsKey', {
  alias: 'shan-msk-kms-key',
  description: 'KMS key for MSK data encryption',
  enableKeyRotation: true,
});

// Step 6: Create SASL/SCRAM Credentials in Secrets Manager
const saslSecret = new secretsmanager.Secret(this, 'ShanMskSaslSecret', {
  secretName: 'shan-msk-sasl-secret',
  description: 'SASL/SCRAM credentials for shan-msk-cluster',
  generateSecretString: {
    secretStringTemplate: JSON.stringify({ username: 'shan-user' }),
    generateStringKey: 'password',
    passwordLength: 16,
    excludePunctuation: true,
  },
});

// Step 7: Create a KMS Key for CloudWatch Logs Encryption
const cloudwatchKmsKey = new kms.Key(this, 'ShanCloudwatchKmsKey', {
  alias: 'shan-cloudwatch-kms-key',
  description: 'KMS key for CloudWatch Logs encryption',
  enableKeyRotation: true,
});

// Step 8: Create a CloudWatch Log Group
const logGroup = new logs.LogGroup(this, 'ShanMskLogGroup', {
  logGroupName: 'shan-msk-logs',
  retention: logs.RetentionDays.ONE_MONTH,
  encryptionKey: cloudwatchKmsKey,
});
```



```
// Step 9: Create the MSK Cluster
const mskCluster = new msk.CfnCluster(this, Grok 3. beta 'mskCluster', {
  clusterName: 'shan-msk-cluster',
  kafkaVersion: '3.7.x', // Replace with the latest supported version
  numberOfWorkers: 2,
  workerNodeGroupInfo: {
    instanceType: 'kafka.t2.small',
    storageInfo: {
      ebsStorageInfo: {
        volumeSize: 10,
      },
    },
    clientSubnets: vpc.privateSubnets.map((subnet) => subnet.subnetId),
    securityGroups: [mskSecurityGroup.securityGroupId],
  },
  encryptionInfo: {
    encryptionAtRest: {
      dataVolumeKmsKeyId: mskKmsKey.keyArn,
    },
  },
  clientAuthentication: {
    sasl: {
      scram: { enabled: true },
      iam: { enabled: true },
    },
  },
  loggingInfo: {
    brokerLogs: {
      cloudWatchLogs: {
        enabled: true,
        logGroup: logGroup.logGroupName,
      },
    },
  },
});
});
```

```
// Associate SASL/SCRAM Secret with the MSK Cluster
const secretTargetAttachment = new secretsmanager.CfnSecretTargetAttachment(
  this,
  'ShanMsSecretAttachment',
  {
    secretId: saslSecret.secretArn,
    targetId: mskCluster.attrArn,
    targetType: 'AWS::MSK::Cluster',
  }
);
secretTargetAttachment.node.addDependency(mskCluster);
```

```
// Outputs
new cdk.CfnOutput(this, 'ClusterArn', {
```

```

    value: mskCluster.attrArn,
    description: 'ARN of the MSK Cluster'
  });
new cdk.CfnOutput(this, 'IngestorSecurityGroupId', {
  value: ingestorSecurityGroup.securityGroupId,
  description: 'Security Group ID for ecs-ingestor-service',
});
new cdk.CfnOutput(this, 'ReportSecurityGroupId', {
  value: reportSecurityGroup.securityGroupId,
  description: 'Security Group ID for report-service',
});
}

const app = new cdk.App();
new MskCdkStack(app, 'MskCdkStack', {
  env: { region: 'us-east-1' },
});

```



Key Changes

1. ECS Security Groups:

- Added `ingestorSecurityGroup` for `ecs-ingestor-service` and `reportSecurityGroup` for `report-service`, both in the same VPC (`shan-msk-vpc`).

2. Ingress Rules:

- Updated the MSK security group (`mskSecurityGroup`) to allow inbound traffic from `ingestorSecurityGroup` and `reportSecurityGroup` on ports 9092, 9094, 9096, 9098, 2181, and 2182.
- These rules ensure that both ECS services can communicate with the MSK brokers using the specified authentication methods (plaintext, TLS, SASL/SCRAM, IAM) and ZooKeeper.

3. Outputs:

- Added outputs for the security group IDs of the ECS services, which you can use when configuring your ECS tasks.

How to Use This with ECS Services

1. If ECS Services Already Exist:

- Replace the `ingestorSecurityGroup` and `reportSecurityGroup` definitions with references to the existing security groups:

typescript

```
const ingestorSecurityGroup = ec2.SecurityGroup.fromSecurityGroupId(
  this,
  'IngestorSecurityGroup',
  'existing-ingestor-sg-id'
);
const reportSecurityGroup = ec2.SecurityGroup.fromSecurityGroupId(
  this,
  'ReportSecurityGroup',
  'existing-report-sg-id'
);
```

Replace `'existing-ingestor-sg-id'` and `'existing-report-sg-id'` with the actual security group IDs.

2. If ECS Services Are Defined in CDK:

- When defining your ECS services (e.g., using `aws-ecs` and `aws-ecs-patterns`), assign `ingestorSecurityGroup` and `reportSecurityGroup` to the ECS tasks:

typescript

```
const ingestorService = new ecs_patterns.ApplicationLoadBalancedFargateService(this, 'Ing
  vpc,
  securityGroups: [ingestorSecurityGroup],
  // Other ECS configurations...
);

const reportService = new ecs_patterns.ApplicationLoadBalancedFargateService(this, 'Repor
  vpc,
  securityGroups: [reportSecurityGroup],
  // Other ECS configurations...
);
```

3. Deploy the Stack:

- Run the following commands:

bash

```
npm run build
cdk synth
```

cdk deploy

Grok 3 beta ▾



- After deployment, note the output security group IDs if you need to reference them elsewhere.

Verification

- After deployment, check the AWS Management Console:
 - **VPC > Security Groups:** Confirm that `shan-msk-sg` has inbound rules allowing traffic from `ecs-ingestor-sg` and `ecs-report-sg` on the specified ports.
 - **MSK Cluster:** Ensure the cluster is associated with `shan-msk-sg`.
- Test connectivity from your ECS services to the MSK cluster using the appropriate Kafka client libraries and ports.



Let me know if you need help integrating this with your ECS service definitions or further adjustments!

- 1 - create a new typescript project and provide the above msk provisioned code .
- 2 - Assume there are 2 ecs instances instead of ecs services. ec2 instance 'producer-ec2' will push data to cluster for topic 'order' and another ec2 instance 'consumer-ec2' will consume the topic.
- 3 - create necessary cdk to create t2.micro ec2 instances 'producer-ec2' and 'consumer-ec2' in the same vpc but in different regions and allow proper permission to send/receive message from/to msk cluster . write test code in

Below is a step-by-step guide to create a new TypeScript CDK project with the MSK provisioned cluster code, updated to include two EC2 instances (`producer-ec2` and `consumer-ec2`) instead of ECS services. The EC2 instances will be placed in the same VPC but in different availability zones (not regions, as MSK clusters are region-specific), and I'll provide the necessary CDK code along with test code to produce and consume messages from the `order` topic.

Step 1: Create a New TypeScript CDK Project

1. Install AWS CDK Globally:

bash

```
npm install -g aws-cdk
```

Grok 3 beta ▾



2. Create a New CDK Project:

bash

```
mkdir msk-ec2-cdk
cd msk-ec2-cdk
cdk init app --language typescript
```

3. Install Required Dependencies:

bash

```
npm install @aws-cdk/aws-msk @aws-cdk/aws-ec2 @aws-cdk/aws-kms @aws-cdk/aws-logs @aws-cdk/aws-
```



4. Update tsconfig.json :

Ensure the TypeScript configuration supports the CDK modules by adding "esModuleInterop": true if not already present:

json

```
{
  "compilerOptions": {
    "target": "ES2018",
    "module": "commonjs",
    "lib": ["es2018"],
    "esModuleInterop": true,
    // Other options...
  }
}
```

Step 2: CDK Code for MSK and EC2 Instances

Replace the content of `lib/msk-ec2-cdk-stack.ts` with the following code:

typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
```

```
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';
import * as secretsmanager from 'aws-cdk-lib/aws-secretsmanager';
import * as iam from 'aws-cdk-lib/aws-iam';

export class MskEc2CdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Step 1: Create a new VPC with 2 subnets in us-east-1a and us-east-1b
    const vpc = new ec2.Vpc(this, 'ShanMskVpc', {
      vpcName: 'shan-msk-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2,
      subnetConfiguration: [
        {
          cidrMask: 24,
          name: 'shan-msk-subnet',
          subnetType: ec2.SubnetType.PRIVATE_ISOLATED,
        },
        {
          cidrMask: 24,
          name: 'shan-ec2-subnet',
          subnetType: ec2.SubnetType.PUBLIC, // Public subnets for EC2 with NAT for outbound
        },
      ],
    });
  }

  // Step 2: Create a Security Group for MSK
  const mskSecurityGroup = new ec2.SecurityGroup(this, 'ShanMskSecurityGroup', {
    vpc,
    securityGroupName: 'shan-msk-sg',
    description: 'Security group for shan-msk-cluster',
    allowAllOutbound: true,
  });

  const mskPorts = [9092, 9094, 9096, 9098, 2181, 2182];
  mskPorts.forEach((port) => {
    mskSecurityGroup.addIngressRule(
      mskSecurityGroup,
      ec2.Port.tcp(port),
      `Allow port ${port} for MSK internal communication`
    );
  });

  // Step 3: Create a KMS Key for Data Encryption
  const mskKmsKey = new kms.Key(this, 'ShanMskKmsKey', {
    alias: 'shan-msk-kms-key',
  });
}
```

```
description: 'KMS key for MSK data encryption',
enableKeyRotation: true,
});
```

// Step 4: Create SASL/SCRAM Credentials in Secrets Manager

```
const saslSecret = new secretsmanager.Secret(this, 'ShanMskSaslSecret', {
  secretName: 'shan-msk-sasl-secret',
  description: 'SASL/SCRAM credentials for shan-msk-cluster',
  generateSecretString: {
    secretStringTemplate: JSON.stringify({ username: 'shan-user' }),
    generateStringKey: 'password',
    passwordLength: 16,
    excludePunctuation: true,
  },
});
```

// Step 5: Create a KMS Key for CloudWatch Logs Encryption

```
const cloudwatchKmsKey = new kms.Key(this, 'ShanCloudwatchKmsKey', {
  alias: 'shan-cloudwatch-kms-key',
  description: 'KMS key for CloudWatch Logs encryption',
  enableKeyRotation: true,
});
```

// Step 6: Create a CloudWatch Log Group

```
const logGroup = new logs.LogGroup(this, 'ShanMskLogGroup', {
  logGroupName: 'shan-msk-logs',
  retention: logs.RetentionDays.ONE_MONTH,
  encryptionKey: cloudwatchKmsKey,
});
```

// Step 7: Create the MSK Cluster

```
const mskCluster = new msk.CfnCluster(this, 'ShanMskCluster', {
  clusterName: 'shan-msk-cluster',
  kafkaVersion: '3.7.x',
  numberOfWorkers: 2,
  workerNodeGroupInfo: {
    instanceType: 'kafka.t2.small',
    storageInfo: { ebsStorageInfo: { volumeSize: 10 } },
    clientSubnets: vpc.privateSubnets.map((subnet) => subnet.subnetId),
    securityGroups: [mskSecurityGroup.securityGroupId],
  },
  encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyArn } },
  clientAuthentication: {
    sasl: { scram: { enabled: true }, iam: { enabled: true } },
  },
  loggingInfo: {
    brokerLogs: { cloudWatchLogs: { enabled: true, logGroup: logGroup.logGroupName } },
  },
});
```

```
const secretTargetAttachment = new secretsmanager.CfnSecretTargetAttachment(
  this,
  'ShanMskSecretAttachment',
  {
    secretId: saslSecret.secretArn,
    targetId: mskCluster.attrArn,
    targetType: 'AWS::MSK::Cluster',
  }
);
secretTargetAttachment.node.addDependency(mskCluster);

// Step 8: Create Security Groups for EC2 Instances
const producerSecurityGroup = new ec2.SecurityGroup(this, 'ProducerSecurityGroup', {
  vpc,
  securityGroupName: 'producer-ec2-sg',
  description: 'Security group for producer-ec2',
  allowAllOutbound: true,
});

const consumerSecurityGroup = new ec2.SecurityGroup(this, 'ConsumerSecurityGroup', {
  vpc,
  securityGroupName: 'consumer-ec2-sg',
  description: 'Security group for consumer-ec2',
  allowAllOutbound: true,
});

// Add ingress rules to MSK SG for EC2 instances
mskPorts.forEach((port) => {
  mskSecurityGroup.addIngressRule(
    producerSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from producer-ec2`
  );
  mskSecurityGroup.addIngressRule(
    consumerSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from consumer-ec2`
  );
});

// Step 9: Create IAM Role for EC2 Instances (for MSK access)
const ec2Role = new iam.Role(this, 'Ec2MskRole', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),
  description: 'Role for EC2 instances to access MSK',
});
ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonMSKFullAccess'));
ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonSSMManagedInstanceCore'));

// Step 10: Create EC2 Instances
const producerEc2 = new ec2.Instance(this, 'ProducerEc2', {
```

```

instanceName: 'producer-ec2',
instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),
machineImage: ec2.MachineImage.latestAmazonLinux2(),
vpc,
vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
availabilityZone: vpc.availabilityZones[0], // us-east-1a
securityGroup: producerSecurityGroup,
role: ec2Role,
keyName: 'your-key-pair', // Replace with your EC2 key pair name
});

const consumerEc2 = new ec2.Instance(this, 'ConsumerEc2', {
instanceName: 'consumer-ec2',
instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),
machineImage: ec2.MachineImage.latestAmazonLinux2(),
vpc,
vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
availabilityZone: vpc.availabilityZones[1], // us-east-1b
securityGroup: consumerSecurityGroup,
role: ec2Role,
keyName: 'your-key-pair', // Replace with your EC2 key pair name
});

// Outputs
new cdk.CfnOutput(this, 'ClusterArn', { value: mskCluster.attrArn });
new cdk.CfnOutput(this, 'ProducerEc2Id', { value: producerEc2.instanceId });
new cdk.CfnOutput(this, 'ConsumerEc2Id', { value: consumerEc2.instanceId });
new cdk.CfnOutput(this, 'ProducerPublicIp', { value: producerEc2.instancePublicIp });
new cdk.CfnOutput(this, 'ConsumerPublicIp', { value: consumerEc2.instancePublicIp });

}

}

const app = new cdk.App();
new MskEc2CdkStack(app, 'MskEc2CdkStack', {
env: { region: 'us-east-1' },
});

```

Step 3: Notes on the Code

1. VPC Configuration:

- Added a public subnet group (`shan-ec2-subnet`) for EC2 instances alongside the private subnets for MSK.

2. EC2 Instances:

- ✗ Grok 3 beta ▾
Upload Download Refresh Edit
- producer-ec2 is placed in `us-east-1a` and consumer-ec2 in `us-east-1b` (different AZs, not regions, as MSK is region-specific).
 - Both use `t2.micro` instances with Amazon Linux 2 and are assigned public IPs for SSH access.

3. Security Groups:

- producer-ec2-sg and consumer-ec2-sg are created, and ingress rules are added to `shan-msk-sg` to allow traffic on MSK ports.

4. IAM Role:

- An IAM role (`Ec2MskRole`) is created with `AmazonMSKFullAccess` for MSK access and `AmazonSSMManagedInstanceCore` for SSM (optional for debugging).

5. Key Pair:

- Replace '`your-key-pair`' with an existing EC2 key pair name in your AWS account (create one in the EC2 console if needed).
-

Step 4: Deploy the Stack

1. Build and Deploy:

```
bash  
  
npm run build  
cdk synth  
cdk deploy
```

- Note the outputs (e.g., `ClusterArn`, `ProducerPublicIp`, `ConsumerPublicIp`).

2. SSH into EC2 Instances:

- Use the public IPs and your key pair to SSH:

```
bash  
  
ssh -i your-key.pem ec2-user@<producer-public-ip>  
ssh -i your-key.pem ec2-user@<consumer-public-ip>
```

Step 5: Test Code for Producer and Consumer

Grok 3 beta



Prerequisites on EC2 Instances

1. Install Java and Kafka:

SSH into each EC2 instance and run:

bash

```
sudo yum update -y
sudo yum install java-11-amazon-corretto -y
wget https://archive.apache.org/dist/kafka/3.7.0/kafka_2.13-3.7.0.tgz
tar -xzf kafka_2.13-3.7.0.tgz
cd kafka_2.13-3.7.0
```

2. Get MSK Bootstrap Brokers:

- In the AWS MSK console, go to `shan-msk-cluster` > "View client information" and note the **Bootstrap servers** string (e.g., `b-1.shan-msk-cluster.xxx.kafka.us-east-1.amazonaws.com:9092,b-2.shan-msk-cluster.xxx.kafka.us-east-1.amazonaws.com:9092`).

Producer Test Code (on `producer-ec2`)

Create a file `produce.sh`:

bash

```
#!/bin/bash
cd ~/kafka_2.13-3.7.0
bin/kafka-console-producer.sh --broker-list <bootstrap-servers> --topic order
```

- Replace `<bootstrap-servers>` with the MSK bootstrap brokers.
- Run:

bash

```
chmod +x produce.sh
./produce.sh
```

- Type messages (e.g., "Order 1", "Order 2") and press Enter to send them.

Consumer Test Code (on `consumer-ec2`)

Create a file `consume.sh`:

bash

```
#!/bin/bash
cd ~/kafka_2.13-3.7.0
bin/kafka-console-consumer.sh --bootstrap-server <bootstrap-servers> --topic order --from-beginning
```



- Replace <bootstrap-servers> with the MSK bootstrap brokers.

- Run:

```
bash
```

```
chmod +x consume.sh
./consume.sh
```

- You should see the messages sent by the producer.

Using IAM Authentication (Optional)

If you prefer IAM authentication (port 9098):

1. Configure Kafka Client:

- Create `client.properties` on both EC2 instances:

```
properties
```

```
security.protocol=SASL_SSL
sasl.mechanism=AWS_MSK_IAM
sasl.jaas.config=software.amazon.msk.auth.iam.IAMLoginModule required;
sasl.client.callback.handler.class=software.amazon.msk.auth.iam.IAMClientCallbackHandler
```



- Download the IAM auth JAR:

```
bash
```

```
wget https://github.com/aws/aws-msk-iam-auth/releases/download/v2.0.3/aws-msk-iam-auth-2.0.3-all.jar
mv aws-msk-iam-auth-2.0.3-all.jar ~/kafka_2.13-3.7.0/libs/
```



- Update the scripts to use port 9098 and the properties file:

```
bash
```

```
# produce.sh
bin/kafka-console-producer.sh --broker-list <bootstrap-servers>:9098 --topic order --prod
```

```
# consume.sh
bin/kafka-console-consumer.sh --bootstrap-server <bootstrap-servers>:9098 --topic order --from-beginning
```

Notes

- **Regions vs. AZs:** MSK clusters are region-specific (e.g., `us-east-1`), so I placed the EC2 instances in different AZs (`us-east-1a`, `us-east-1b`). For different regions, you'd need a multi-region MSK setup, which is more complex.
- **Topic Creation:** The `order` topic isn't created automatically. Run this on one EC2 instance to create it:

bash

```
bin/kafka-topics.sh --create --bootstrap-server <bootstrap-servers> --replication-factor 2 --topic order
```

- **Cleanup:** Destroy the stack with `cdk destroy` when done.

Let me know if you need further refinements!

more detailed test code

Below is a more detailed test code setup for the `producer-ec2` and `consumer-ec2` instances to interact with the Amazon MSK cluster (`shan-msk-cluster`) using Node.js with the `kafkajs` library. This provides a programmatic approach instead of shell scripts, including error handling, configuration for IAM authentication (port 9098), and detailed logging. I'll also include instructions to set up the environment on the EC2 instances and deploy the test code.

Step 1: Environment Setup on EC2 Instances

On Both `producer-ec2` and `consumer-ec2` :

1. SSH into the Instances:

- Use the public IPs from the CDK outputs and your key pair:

bash Grok 3 beta ▾

```
ssh -i your-key.pem ec2-user@<producer-public-ip>
ssh -i your-key.pem ec2-user@<consumer-public-ip>
```

⤒ ⌂ ⌃ ⌄

2. Install Node.js:

bash ⤒

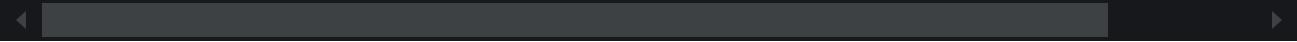
```
sudo yum update -y
curl -o https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
source ~/.bashrc
nvm install 20
node -v # Should output v20.x.x
npm -v # Should output a version
```

3. Install Kafka IAM Authentication Library:

- Since we'll use IAM authentication (port 9098), download the AWS MSK IAM auth library:

bash ⤒

```
mkdir -p ~/msk-test/libs
cd ~/msk-test/libs
wget https://github.com/aws/aws-msk-iam-auth/releases/download/v2.0.3/aws-msk-iam-auth-2.0.3.jar
```



4. Get MSK Bootstrap Brokers:

- In the AWS MSK console, go to `shan-msk-cluster` > "View client information" and copy the **IAM authenticated bootstrap servers** (e.g., `b-1.shan-msk-cluster.xxx.kafka.us-east-1.amazonaws.com:9098,b-2.shan-msk-cluster.xxx.kafka.us-east-1.amazonaws.com:9098`).

Step 2: Detailed Test Code

Producer Code (on `producer-ec2`)

Create a directory and initialize a Node.js project:

bash ⤒

```
mkdir ~/msk-test/producer
cd ~/msk-test/producer
```

```
npm init -y
npm install kafkajs aws-msk-iam-auth
```

Grok 3 beta ▾



Create producer.ts :

typescript

```
import { Kafka, logLevel } from 'kafkajs';
import { IAM } from 'aws-msk-iam-auth';

// MSK configuration
const bootstrapServers = '<bootstrap-servers>'; // Replace with your IAM bootstrap servers (e.g.,
const topic = 'order';

// Configure IAM authentication
const sasl = new IAM({
  accessKeyId: process.env.AWS_ACCESS_KEY_ID, // Optional: EC2 role provides credentials
  secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY, // Optional
  sessionToken: process.env.AWS_SESSION_TOKEN, // Optional
});

// Initialize Kafka client
const kafka = new Kafka({
  clientId: 'producer-ec2',
  brokers: bootstrapServers.split(','),
  ssl: true,
  sasl,
  logLevel: logLevel.INFO,
});

const producer = kafka.producer();

async function runProducer() {
  try {
    console.log('Connecting producer to MSK...');
    await producer.connect();
    console.log('Producer connected successfully');

    // Send messages every 5 seconds
    let orderId = 1;
    setInterval(async () => {
      const message = {
        key: `order-${orderId}`,
        value: JSON.stringify({
          orderId,
          product: `Product ${orderId}`,
          quantity: Math.floor(Math.random() * 10) + 1,
          timestamp: new Date().toISOString(),
        })
      }
      await producer.send({
        topic,
        messages: [message]
      });
      orderId++;
    }, 5000);
  } catch (err) {
    console.error('Error connecting to MSK:', err);
  }
}
```

```

    }),

  });

  await producer.send({
    topic,
    messages: [message],
  });
  console.log(`Sent message: ${message.key} - ${message.value}`);
  orderId++;
}, 5000);
} catch (error) {
  console.error('Producer error:', error);
  process.exit(1);
}
}

// Handle shutdown gracefully
process.on('SIGINT', async () => {
  console.log('Disconnecting producer...');
  await producer.disconnect();
  process.exit(0);
});

runProducer();

```

Consumer Code (on consumer-ec2)

Create a directory and initialize a Node.js project:

bash

```

mkdir ~/msk-test/consumer
cd ~/msk-test/consumer
npm init -y
npm install kafkajs aws-msk-iam-auth

```

Create `consumer.ts`:

typescript

```

import { Kafka, logLevel } from 'kafkajs';
import { IAM } from 'aws-msk-iam-auth';

// MSK configuration
const bootstrapServers = '<bootstrap-servers>'; // Replace with your IAM bootstrap servers (e.g.,
const topic = 'order';

```



```
// Configure IAM authentication
const sasl = new IAM({
  accessKeyId: process.env.AWS_ACCESS_KEY_ID, // Optional: EC2 role provides credentials
  secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY, // Optional
  sessionToken: process.env.AWS_SESSION_TOKEN, // Optional
});

// Initialize Kafka client
const kafka = new Kafka({
  clientId: 'consumer-ec2',
  brokers: bootstrapServers.split(','),
  ssl: true,
  sasl,
  logLevel: logLevel.INFO,
});

const consumer = kafka.consumer({ groupId: 'order-consumer-group' });

async function runConsumer() {
  try {
    console.log('Connecting consumer to MSK...');
    await consumer.connect();
    console.log('Consumer connected successfully');

    await consumer.subscribe({ topic, fromBeginning: true });

    await consumer.run({
      eachMessage: async ({ topic, partition, message }) => {
        const key = message.key?.toString() || 'null';
        const value = message.value?.toString() || 'null';
        const timestamp = message.timestamp || 'unknown';
        console.log({
          topic,
          partition,
          key,
          value: JSON.parse(value), // Parse JSON message
          timestamp: new Date(parseInt(timestamp)).toISOString(),
        });
      },
    });
  } catch (error) {
    console.error('Consumer error:', error);
    process.exit(1);
  }
}

// Handle shutdown gracefully
process.on('SIGINT', async () => {
  console.log('Disconnecting consumer...');
  await consumer.disconnect();
})
```

```
process.exit(0);
});
runConsumer();
```

Grok 3 beta ▾



Compile TypeScript

On both instances, install TypeScript and compile the code:

bash

```
npm install -g typescript
cd ~/msk-test/producer
tsc --init
tsc producer.ts
cd ~/msk-test/consumer
tsc --init
tsc consumer.ts
```

Step 3: Create the order Topic

On either EC2 instance, use the Kafka CLI to create the topic (assuming Kafka is installed as per the previous instructions):

bash

```
cd ~/kafka_2.13-3.7.0
bin/kafka-topics.sh --create --bootstrap-server <bootstrap-servers>:9098 --replication-factor 2 --
```

- 
- Use the `client.properties` file from the previous IAM setup:

properties

```
security.protocol=SASL_SSL
sasl.mechanism=AWS_MSK_IAM
sasl.jaas.config=software.amazon.msk.auth.iam.IAMLoginModule required;
sasl.client.callback.handler.class=software.amazon.msk.auth.iam.IAMClientCallbackHandler
```

Step 4: Run the Test Code

On producer-ec2 :

bash

```
cd ~/msk-test/producer
export CLASSPATH=$CLASSPATH:~/msk-test/libs/aws-msk-iam-auth-2.0.3-all.jar
node producer.js
```

- Output will look like:

```
Connecting producer to MSK...
Producer connected successfully
Sent message: order-1 - {"orderId":1,"product":"Product 1","quantity":7,"timestamp":"2025-03-03T12:00:00.000Z"}
Sent message: order-2 - {"orderId":2,"product":"Product 2","quantity":3,"timestamp":"2025-03-03T12:00:05.000Z"}
```

On consumer-ec2 :

bash

```
cd ~/msk-test/consumer
export CLASSPATH=$CLASSPATH:~/msk-test/libs/aws-msk-iam-auth-2.0.3-all.jar
node consumer.js
```

- Output will look like:

```
Connecting consumer to MSK...
Consumer connected successfully
{
  topic: 'order',
  partition: 0,
  key: 'order-1',
  value: { orderId: 1, product: 'Product 1', quantity: 7, timestamp: '2025-03-03T12:00:00.000Z' }
}
{
  topic: 'order',
  partition: 0,
  key: 'order-2',
  value: { orderId: 2, product: 'Product 2', quantity: 3, timestamp: '2025-03-03T12:00:05.000Z' }
}
```

timestamp: '2025-03-03T12:00:05.000Z'

X }

Grok 3 beta ▾



Step 5: Enhancements and Troubleshooting

Enhancements:

1. Error Retry Logic:

Add retry logic for connection failures:

typescript

```
const connectWithRetry = async (client: any, type: string, maxRetries = 5) => {
  for (let i = 0; i < maxRetries; i++) {
    try {
      await client.connect();
      console.log(`#${type} connected successfully`);
      return;
    } catch (error) {
      console.error(`Attempt ${i + 1} failed:`, error);
      if (i === maxRetries - 1) throw error;
      await new Promise((resolve) => setTimeout(resolve, 2000 * (i + 1)));
    }
  }
};

// Replace producer.connect() with connectWithRetry(producer, 'Producer')
```



2. Message Acknowledgment:

Ensure messages are acknowledged by setting `acks: 'all'` in the producer:

typescript

```
await producer.send({
  topic,
  messages: [message],
  acks: 'all',
});
```



Troubleshooting:

- **Connection Issues:** Verify the bootstrap servers, security group rules, and IAM role permissions.
- **IAM Auth Errors:** Ensure the `CLASSPATH` includes the IAM auth JAR and the EC2 role has `AmazonMSKFullAccess`.

- **Topic Not Found:** Confirm



explain below code as why it is needed
Grok 3 beta ▾



This setup provides a robust, pi
MSK cluster. Let me know if you

```
const secretTargetAttachment = new secretsmanager.CfnSecretTargetAttachment(
  this,
  'ShanMskSecretAttachment',
  {
    secretId: saslSecret.secretArn,
    targetId: mskCluster.attrArn,
    targetType: 'AWS::MSK::Cluster',
  }
);
secretTargetAttachment.node.addDependency(mskCluster);
```



Let me break down the provided code snippet and explain why it is needed in the context of your AWS CDK project for provisioning an Amazon MSK (Managed Streaming for Apache Kafka) cluster with SASL/SCRAM authentication.

Code Snippet

typescript



```
const secretTargetAttachment = new secretsmanager.CfnSecretTargetAttachment(
  this,
  'ShanMskSecretAttachment',
  {
    secretId: saslSecret.secretArn,
    targetId: mskCluster.attrArn,
    targetType: 'AWS::MSK::Cluster',
  }
);
secretTargetAttachment.node.addDependency(mskCluster);
```

Explanation

1. What is `CfnSecretTargetAttachment` ?

- `secretsmanager.CfnSecretTargetAttachment` is a low-level CloudFormation resource construct in the AWS CDK. It corresponds to the AWS CloudFormation resource `AWS::SecretsManager::SecretTargetAttachment` .

- This resource is used to associate an AWS Secrets Manager secret with a specific target, such as an Amazon MSK cluster, Amazon RDS ~~database~~ **beta** other supported AWS services.   
- In your case, it links the SASL/SCRAM credentials (stored in the `saslSecret` secret) to the MSK cluster (`mskCluster`) so that the cluster can use these credentials for authentication.

2. Why is it Needed?

- **Purpose of SASL/SCRAM Authentication:**

- Your MSK cluster is configured to use SASL/SCRAM (Simple Authentication and Security Layer / Salted Challenge Response Authentication Mechanism) as one of its authentication methods (`clientAuthentication.sasl.scram.enabled: true`).
- SASL/SCRAM requires a username and password to authenticate clients connecting to the Kafka brokers. These credentials are securely stored in AWS Secrets Manager as a secret (`shan-msk-sasl-secret` in your code).

- **Association Requirement:**

- For MSK to use these credentials, the cluster must know which Secrets Manager secret contains them. AWS doesn't automatically associate a secret with an MSK cluster—you must explicitly link them.
- The `CfnSecretTargetAttachment` resource performs this association by updating the secret with metadata that points to the MSK cluster, enabling the cluster to retrieve and use the credentials.

- **What Happens Without It?**

- Without this attachment, the MSK cluster wouldn't know where to find the SASL/SCRAM credentials, and clients attempting to connect using SASL/SCRAM (e.g., via port 9096) would fail authentication because the cluster wouldn't have access to the username/password pair.

3. Breaking Down the Parameters

typescript 

```
const secretTargetAttachment = new secretsmanager.CfnSecretTargetAttachment(
  this, // Scope (the stack)
  'ShanMsksSecretAttachment', // Logical ID for the resource
  {
    secretId: saslSecret.secretArn, // ARN of the Secrets Manager secret
    targetId: mskCluster.attrArn, // ARN of the MSK cluster
    targetType: 'AWS::MSK::Cluster', // Type of target (MSK cluster)
  }
);
```

- `secretId: saslSecret.secretArn` :

- This is the Amazon Resource Name (ARN) of the secret (`shan-msk-sasl-secret`) created earlier in the code using `secretsmanager.Secret`. **Grok 3. beta** contains the SASL/SCRAM username and password (e.g., `{"username": "shan-user", "password": "<generated-password>"}`).
- The ARN uniquely identifies the secret in your AWS account.
- `targetId: mskCluster.attrArn` :
 - This is the ARN of the MSK cluster (`shan-msk-cluster`) created using `msk.CfnCluster`. The `attrArn` property retrieves the ARN dynamically after the cluster is provisioned.
 - It specifies the target resource that will use the secret.
- `targetType: 'AWS::MSK::Cluster'` :
 - This indicates the type of resource being associated with the secret. For MSK, it must be `'AWS::MSK::Cluster'` to match the service's expectations.
 - AWS uses this to validate that the target is compatible with the secret's intended use (e.g., SASL/SCRAM for Kafka).

4. Why `secretTargetAttachment.node.addDependency(mskCluster)` ?

- **Dependency Management:**
 - In AWS CDK, resources are created in parallel by CloudFormation unless dependencies are explicitly defined. The `cfnSecretTargetAttachment` resource needs the MSK cluster to exist before it can associate the secret with it.
 - If the secret attachment is attempted before the MSK cluster is fully created, the operation will fail because the `targetId` (the cluster ARN) won't yet be valid.
- `node.addDependency(mskCluster)` :
 - This line ensures that the `mskCluster` resource is fully provisioned before the `secretTargetAttachment` resource is created.
 - It adds a dependency in the CloudFormation template, forcing the creation order: MSK cluster first, then the secret attachment.
 - Without this, you might encounter a race condition where CloudFormation tries to attach the secret before the cluster's ARN is available, leading to a deployment error.

5. What Does It Do Under the Hood?

- When this code is synthesized and deployed:
 - The `CfnSecretTargetAttachment` resource updates the Secrets Manager secret (`shan-msk-sasl-secret`) by adding a `SecretTargetAttachment` section to its metadata.
 - This metadata includes a reference to the MSK cluster's ARN, allowing the MSK service to fetch the secret's contents (username and password) when clients authenticate via SASL/SCRAM.
 - The resulting secret in Secrets Manager might look like this (in JSON format):



```

json
Grok 3 beta ▾
{
  "SecretString": "{\"username\":\"shan-user\", \"password\":\"<generated-password>\"}",
  "SecretTargetAttachment": {
    "TargetType": "AWS::MSK::Cluster",
    "TargetId": "arn:aws:kafka:us-east-1:123456789012:cluster/shan-msk-cluster/xxx"
  }
}

```

6. Why Use `Cfn` (Low-Level) Instead of a Higher-Level Construct?

- The AWS CDK provides higher-level constructs (e.g., `secretsmanager.Secret`) for most use cases, but as of the latest CDK versions, there isn't a direct high-level construct to associate a secret with an MSK cluster for SASL/SCRAM authentication.
- The `CfnSecretTargetAttachment` is a lower-level (L1) construct that maps directly to CloudFormation, giving you the flexibility to perform this specific association, which isn't yet abstracted into a higher-level (L2) construct for MSK.

7. Real-World Context

- In your setup, `producer-ec2` and `consumer-ec2` need to authenticate with the MSK cluster to send and receive messages on the `order` topic. If you're using SASL/SCRAM (port 9096), the client must provide the username and password stored in `shan-msk-sasl-secret`. The MSK cluster, in turn, needs to know where to find these credentials—hence the need for this attachment.

Summary

- **Why `CfnSecretTargetAttachment`?**
 - It's required to link the SASL/SCRAM credentials in Secrets Manager to the MSK cluster, enabling the cluster to authenticate clients using those credentials.
- **Why the Dependency?**
 - It ensures the MSK cluster exists before the attachment is created, preventing deployment failures due to timing issues.
- **Overall Need:**
 - Without this code, SASL/SCRAM authentication wouldn't work, as the MSK cluster wouldn't have access to the necessary credentials, breaking the authentication flow for your Kafka



Let me know if you need further clarification or details!

can we perate msk stack and ec2 stack in separate file in same project     

Yes, you can separate the MSK stack and the EC2 stack into different files within the same AWS CDK project. This modular approach improves code organization, readability, and maintainability, especially as your infrastructure grows. Below, I'll guide you through restructuring your TypeScript CDK project to split the MSK and EC2 resources into separate stack files while keeping them in the same project.

Project Structure

Here's how the updated project structure will look:

```
msk-ec2-cdk/
├── bin/
│   └── msk-ec2-cdk.ts      # Entry point to deploy both stacks
├── lib/
│   ├── msk-stack.ts        # MSK stack definition
│   └── ec2-stack.ts        # EC2 stack definition
├── package.json
├── tsconfig.json
├── cdk.json
└── ... (other CDK files)
```

Step 1: Update the Project

Assuming you've already initialized the CDK project (as described earlier), let's split the code.

1. lib/msk-stack.ts - MSK Stack

This file will contain the MSK cluster, VPC, security group, KMS keys, Secrets Manager secret, and related resources.

typescript 

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';
```



```
import * as secretsmanager from 'aws-cdk-lib/aws-secretsmanager';
export class MskStack extends cdk.Stack {
  public readonly vpc: ec2.Vpc; // Export VPC for use in EC2 stack
  public readonly mskSecurityGroup: ec2.SecurityGroup; // Export SG for EC2 ingress rules
  public readonly mskClusterArn: string; // Export cluster ARN

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create VPC with public and private subnets
    this.vpc = new ec2.Vpc(this, 'ShanMskVpc', {
      vpcName: 'shan-msk-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2,
      subnetConfiguration: [
        {
          cidrMask: 24,
          name: 'shan-msk-subnet',
          subnetType: ec2.SubnetType.PRIVATE_ISOLATED,
        },
        {
          cidrMask: 24,
          name: 'shan-ec2-subnet',
          subnetType: ec2.SubnetType.PUBLIC,
        },
      ],
    });
  }

  // Create Security Group for MSK
  this.mskSecurityGroup = new ec2.SecurityGroup(this, 'ShanMskSecurityGroup', {
    vpc: this.vpc,
    securityGroupName: 'shan-msk-sg',
    description: 'Security group for shan-msk-cluster',
    allowAllOutbound: true,
  });

  const mskPorts = [9092, 9094, 9096, 9098, 2181, 2182];
  mskPorts.forEach((port) => {
    this.mskSecurityGroup.addIngressRule(
      this.mskSecurityGroup,
      ec2.Port.tcp(port),
      `Allow port ${port} for MSK internal communication`
    );
  });

  // KMS Key for Data Encryption
  const mskKmsKey = new kms.Key(this, 'ShanMskKmsKey', {
    alias: 'shan-msk-kms-key',
    description: 'KMS key for MSK data encryption',
  });
}
```

```
enableKeyRotation: true,  
});  
  
// SASL/SCRAM Credentials in Secrets Manager  
const saslSecret = new secretsmanager.Secret(this, 'ShanMskSaslSecret', {  
  secretName: 'shan-msk-sasl-secret',  
  description: 'SASL/SCRAM credentials for shan-msk-cluster',  
  generateSecretString: {  
    secretStringTemplate: JSON.stringify({ username: 'shan-user' }),  
    generateStringKey: 'password',  
    passwordLength: 16,  
    excludePunctuation: true,  
  },  
});  
  
// KMS Key for CloudWatch Logs  
const cloudwatchKmsKey = new kms.Key(this, 'ShanCloudwatchKmsKey', {  
  alias: 'shan-cloudwatch-kms-key',  
  description: 'KMS key for CloudWatch Logs encryption',  
  enableKeyRotation: true,  
});  
  
// CloudWatch Log Group  
const logGroup = new logs.LogGroup(this, 'ShanMskLogGroup', {  
  logGroupName: 'shan-msk-logs',  
  retention: logs.RetentionDays.ONE_MONTH,  
  encryptionKey: cloudwatchKmsKey,  
});  
  
// MSK Cluster  
const mskCluster = new msk.CfnCluster(this, 'ShanMskCluster', {  
  clusterName: 'shan-msk-cluster',  
  kafkaVersion: '3.7.x',  
  numberOfWorkers: 2,  
  workerNodeGroupInfo: {  
    instanceType: 'kafka.t2.small',  
    storageInfo: { ebsStorageInfo: { volumeSize: 10 } },  
    clientSubnets: this.vpc.privateSubnets.map((subnet) => subnet.subnetId),  
    securityGroups: [this.mskSecurityGroup.securityGroupId],  
  },  
  encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyArn } },  
  clientAuthentication: {  
    sasl: { scram: { enabled: true }, iam: { enabled: true } },  
  },  
  loggingInfo: {  
    brokerLogs: { cloudWatchLogs: { enabled: true, logGroup: logGroup.logGroupName } },  
  },  
});  
  
// Associate SASL/SCRAM Secret with MSK Cluster
```

```

const secretTargetAttachment = new secretsmanager.CfnSecretTargetAttachment(
  this,
  'ShanMskSecretAttachment',
  {
    secretId: saslSecret.secretArn,
    targetId: mskCluster.attrArn,
    targetType: 'AWS::MSK::Cluster',
  }
);
secretTargetAttachment.node.addDependency(mskCluster);

// Export Cluster ARN
this.mskClusterArn = mskCluster.attrArn;

// Outputs
new cdk.CfnOutput(this, 'ClusterArn', { value: this.mskClusterArn });
}
}

```

2. lib/ec2-stack.ts - EC2 Stack

This file will contain the EC2 instances (`producer-ec2` and `consumer-ec2`), their security groups, and IAM role, referencing the MSK stack's VPC and security group.

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as iam from 'aws-cdk-lib/aws-iam';
import { MskStack } from './msk-stack';

export class Ec2Stack extends cdk.Stack {
  constructor(scope: Construct, id: string, mskStack: MskStack, props?: cdk.StackProps) {
    super(scope, id, props);

    // Reference the VPC and MSK Security Group from MskStack
    const vpc = mskStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    // IAM Role for EC2 Instances
    const ec2Role = new iam.Role(this, 'Ec2MskRole', {
      assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),
      description: 'Role for EC2 instances to access MSK',
    });
    ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonMSKFullAccess'));
    ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonSSMManagedInstanceC

    // Security Groups for EC2 Instances

```

const producerSecurityGroup = new ec2.SecurityGroup(this, 'ProducerSecurityGroup', {
 vpc,
 securityGroupName: 'producer-ec2-sg',
 description: 'Security group for producer-ec2',
 allowAllOutbound: true,
});

const consumerSecurityGroup = new ec2.SecurityGroup(this, 'ConsumerSecurityGroup', {
 vpc,
 securityGroupName: 'consumer-ec2-sg',
 description: 'Security group for consumer-ec2',
 allowAllOutbound: true,
});

// Add ingress rules to MSK SG for EC2 instances
const mskPorts = [9092, 9094, 9096, 9098, 2181, 2182];
mskPorts.forEach((port) => {
 mskSecurityGroup.addIngressRule(
 producerSecurityGroup,
 ec2.Port.tcp(port),
 `Allow port \${port} from producer-ec2`
);
 mskSecurityGroup.addIngressRule(
 consumerSecurityGroup,
 ec2.Port.tcp(port),
 `Allow port \${port} from consumer-ec2`
);
});

// EC2 Instances
const producerEc2 = new ec2.Instance(this, 'ProducerEc2', {
 instanceName: 'producer-ec2',
 instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),
 machineImage: ec2.MachineImage.latestAmazonLinux2(),
 vpc,
 vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
 availabilityZone: vpc.availabilityZones[0], // us-east-1a
 securityGroup: producerSecurityGroup,
 role: ec2Role,
 keyName: 'your-key-pair', // Replace with your EC2 key pair name
});

const consumerEc2 = new ec2.Instance(this, 'ConsumerEc2', {
 instanceName: 'consumer-ec2',
 instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),
 machineImage: ec2.MachineImage.latestAmazonLinux2(),
 vpc,
 vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
 availabilityZone: vpc.availabilityZones[1], // us-east-1b
 securityGroup: consumerSecurityGroup,
});

```

role: ec2Role,
  keyName: 'your-key-pair', // Replace with Grok beta EC2 key pair name
});

// Outputs
new cdk.CfnOutput(this, 'ProducerEc2Id', { value: producerEc2.instanceId });
new cdk.CfnOutput(this, 'ConsumerEc2Id', { value: consumerEc2.instanceId });
new cdk.CfnOutput(this, 'ProducerPublicIp', { value: producerEc2.instancePublicIp });
new cdk.CfnOutput(this, 'ConsumerPublicIp', { value: consumerEc2.instancePublicIp });
}
}

```

3. bin/msk-ec2-cdk.ts - Entry Point

Update the entry point to instantiate both stacks and establish their relationship.

typescript

```

#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MskStack } from '../lib/msk-stack';
import { Ec2Stack } from '../lib/ec2-stack';

const app = new cdk.App();

// Create MSK stack
const mskStack = new MskStack(app, 'MskStack', {
  env: { region: 'us-east-1' },
});

// Create EC2 stack, passing the MSK stack as a dependency
const ec2Stack = new Ec2Stack(app, 'Ec2Stack', mskStack, {
  env: { region: 'us-east-1' },
});

// Ensure EC2 stack depends on MSK stack
ec2Stack.addDependency(mskStack);

```

Step 2: Key Changes and Why They're Needed

1. Separation of Concerns:

- **MskStack** : Contains all MSK-related resources (VPC, MSK cluster, security group, KMS keys, etc.). This isolates the Kafka infrastructure.     
- **Ec2Stack** : Contains the EC2 instances and their dependencies (security groups, IAM role). This isolates the compute resources.

2. Exported Properties:

- **vpc** : Exported from **MskStack** so **Ec2Stack** can place EC2 instances in the same VPC.
- **mskSecurityGroup** : Exported so **Ec2Stack** can add ingress rules to allow EC2 traffic to MSK.
- **mskClusterArn** : Exported for potential future use (e.g., referencing the cluster in other stacks).

3. Dependency Management:

- **ec2Stack.addDependency(mskStack)** : Ensures the MSK stack is fully deployed (VPC, security group, cluster) before the EC2 stack is created. This is critical because the EC2 stack references the VPC and MSK security group.
- Without this dependency, CloudFormation might attempt to create EC2 resources before the VPC or security group exists, causing a deployment failure.

4. Passing **mskStack** to **Ec2Stack**:

- The **Ec2Stack** constructor takes **mskStack** as a parameter to access its exported properties (**vpc** and **mskSecurityGroup**). This establishes a clear relationship between the stacks without duplicating VPC or security group definitions.

Step 3: Deploy the Stacks

1. Build the Project:

```
bash
```

```
npm run build
```

2. Synthesize the CloudFormation Templates:

```
bash
```

```
cdk synth
```

- You'll see two stacks: **MskStack** and **Ec2Stack**.

3. Deploy the Stacks:

```
bash
```

cdk deploy --all

Grok 3 beta ▾

×

- This deploys both stacks in the correct order (`MsKStack` first, then `Ec2Stack`).
- Note the outputs (e.g., `ClusterArn` , `ProducerPublicIp` , `ConsumerPublicIp`).

4. **Destroy (Optional):**

bash

cdk destroy --all

□

Step 4: Testing Integration

The test code (producer and consumer) from the previous response remains unchanged. After deployment:

1. SSH into `producer-ec2` and `consumer-ec2` using their public IPs.
2. Set up Node.js and the `kafkajs` test code as described earlier.
3. Use the MSK cluster's IAM bootstrap servers (port 9098) from the MSK console to configure the producer and consumer.

The EC2 instances will communicate with the MSK cluster via the security group rules defined in `Ec2Stack` , which reference `mskSecurityGroup` from `MsKStack` .

Benefits of This Approach

- **Modularity:** Each stack focuses on a specific part of the infrastructure, making it easier to modify or extend (e.g., add more EC2 instances or MSK configurations).
- **Reusability:** The `MsKStack` could be reused in another project without the EC2 instances if needed.
- **Dependency Clarity:** Explicit dependencies ensure proper deployment order and resource referencing.



Let me know if you need further adjustments or additional details!



what to do with below line

Grok 3 beta ▾



keyName: 'your-key-pair', // Replace with your EC2 key pair name

The line `keyName: 'your-key-pair'` in the CDK code for the EC2 instances (`producer-ec2` and `consumer-ec2`) specifies the name of an **EC2 key pair** that AWS will use to enable SSH access to these instances. This is an important configuration because, without a key pair, you won't be able to log in to the EC2 instances to set up your test code or perform other administrative tasks.

Here's what you need to do with this line and why it's necessary:

Why It's Needed

- **EC2 Key Pair Purpose:**
 - A key pair consists of a public key (stored by AWS) and a private key (downloaded by you as a `.pem` file). When an EC2 instance is launched, the public key is placed on the instance, allowing you to use the private key to SSH into it securely.
 - The `keyName` property tells AWS which key pair to associate with the instance.
 - **Default Behavior:**
 - If you don't specify a `keyName`, the EC2 instance will launch without an associated key pair, and you won't be able to SSH into it using a private key unless you use alternative methods like AWS Systems Manager (SSM) Session Manager (which your code supports via the `AmazonSSMManagedInstanceCore` policy).
 - **Your Use Case:**
 - Since you're deploying test code (e.g., Kafka producer and consumer scripts) on `producer-ec2` and `consumer-ec2`, you need SSH access to upload files, install dependencies, and run the code. Thus, specifying a key pair is essential.
-

What to Do with This Line

You need to replace `'your-key-pair'` with the name of an existing EC2 key pair in your AWS account, or create a new one if you don't have one. Below are the steps to handle this:

Option 1: Use an Existing Key Pair

1. Check for Existing Key Pairs:

- Go to the AWS Management Console > EC2 > "Key Pairs" (under "Network & Security").
- Look for an existing key pair name (e.g., `my-ec2-key`) and ensure you have the corresponding `.pem` file downloaded and accessible on your local machine.

2. Update the Code:

- Replace `'your-key-pair'` with the name of your existing key pair. For example:

typescript

```
keyName: 'my-ec2-key', // Replace with your actual key pair name
```

3. Verify Access:

- After deploying the stack, use the `.pem` file to SSH into the instances:

bash

```
ssh -i my-ec2-key.pem ec2-user@<producer-public-ip>
ssh -i my-ec2-key.pem ec2-user@<consumer-public-ip>
```

Option 2: Create a New Key Pair

If you don't have a key pair or want a new one:

1. Create a Key Pair in AWS Console:

- Go to EC2 > "Key Pairs" > "Create key pair".
- **Name:** Enter a name (e.g., `shan-ec2-key`).
- **Key pair type:** Select "RSA" (default).
- **Private key file format:** Select `.pem` (for SSH).
- Click "Create key pair".
- Download the `.pem` file (e.g., `shan-ec2-key.pem`) and store it securely (e.g., in `~/.ssh/`).

2. Update the CDK Code:

- Replace `'your-key-pair'` with the name you chose:

typescript

```
keyName: 'shan-ec2-key', // Use the name of the new key pair
```

- Update this in both `producerEc2` and `consumerEc2` definitions in `lib/ec2-stack.ts`.

3. Secure the Private Key:

- Set appropriate permissions on the `.pem` file:

bash

Grok 3 **beta** ▾

```
chmod 400 ~/.ssh/shan-ec2-key.pem
```

4. Deploy and Test:

- Deploy the stack:

bash

```
cd msk-ec2-cdk
npm run build
cdk deploy --all
```

- SSH into the instances using the new key:

bash

```
ssh -i ~/.ssh/shan-ec2-key.pem ec2-user@<producer-public-ip>
ssh -i ~/.ssh/shan-ec2-key.pem ec2-user@<consumer-public-ip>
```

Option 3: Remove `keyName` (Not Recommended for Your Case)

- If you don't want to use a key pair and prefer SSM for access:

- Remove the `keyName` line entirely from both EC2 instance definitions:

typescript

```
const producerEc2 = new ec2.Instance(this, 'ProducerEc2', {
  instanceName: 'producer-ec2',
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),
  machineImage: ec2.MachineImage.latestAmazonLinux2(),
  vpc,
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
  availabilityZone: vpc.availabilityZones[0],
  securityGroup: producerSecurityGroup,
  role: ec2Role,
  // No keyName here
});
```

- Use AWS Systems Manager Session Manager to access the instances:

bash

```
aws ssm start-session --target <producer-ec2-instance-id>
aws ssm start-session --target <consumer-ec2-instance-id>
```

- **Why Not Recommended:** Your test code setup requires SSH to install Node.js, Kafka libraries, and run scripts, which is simpler with **Grok** **beta** than configuring SSM for file transfers and execution.
-

Updated Code Example

Here's how the `producerEc2` definition in `lib/ec2-stack.ts` would look after replacing the key pair name:

typescript

```
const producerEc2 = new ec2.Instance(this, 'ProducerEc2', {  
  instanceName: 'producer-ec2',  
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),  
  machineImage: ec2.MachineImage.latestAmazonLinux2(),  
  vpc,  
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },  
  availabilityZone: vpc.availabilityZones[0],  
  securityGroup: producerSecurityGroup,  
  role: ec2Role,  
  keyName: 'shan-ec2-key', // Updated with actual key pair name  
});
```

Repeat the same change for `consumerEc2`.

Next Steps After Updating

1. Deploy the Stack:

- After updating `keyName`, deploy the stacks:

bash

```
cdk deploy --all
```

2. Verify SSH Access:

- Use the public IPs from the CDK outputs (`ProducerPublicIp`, `ConsumerPublicIp`) to SSH in:

bash

```
ssh -i ~/.ssh/shan-ec2-key.pem ec2-user@producer-public-ip>
```

Grok 3 beta ▾



3. Proceed with Test Code:

- Once you've confirmed SSH access, proceed with installing Node.js and running the producer/consumer test code as outlined earlier.

Additional Notes

- Key Pair Region:** Ensure the key pair exists in the same region as your stack (`us-east-1` in this case). Key pairs are region-specific.
- Lost Private Key:** If you lose the `.pem` file for an existing key pair, you can't recover it—you'll need to create a new key pair and update the code.
- Security:** Store the `.pem` file securely and avoid committing it to version control.

Let me know if you need help creating the key pair or troubleshooting SSH access!

can we create key-pair thru cdk

Yes, you can create an EC2 key pair programmatically using the AWS CDK by leveraging the `aws-ec2` module's `CfnKeyPair` resource, which maps to the AWS CloudFormation `AWS::EC2::KeyPair` resource. This allows you to generate a key pair during stack deployment and retrieve the private key material for SSH access to your EC2 instances (`producer-ec2` and `consumer-ec2`).

However, there are a few important considerations:

- The private key material is returned as a CloudFormation output, but it's sensitive data that must be handled carefully.
- Unlike the AWS Management Console, CDK/CloudFormation doesn't automatically download the `.pem` file—you'll need to capture and save it manually after deployment.
- The key pair is created once and persists in your AWS account until explicitly deleted (e.g., via stack deletion or manual removal).

Below, I'll modify the existing CDK code to create a key pair and integrate it with the EC2 stack.

Updated CDK Code with Key Pair Creation

Grok 3 **beta**

1. lib/msk-stack.ts

No changes are needed here—it remains as previously defined.

2. lib/ec2-stack.ts

Add the `CfnKeyPair` resource and use its name in the EC2 instance definitions. Here's the updated file:

typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as iam from 'aws-cdk-lib/aws-iam';
import { MskStack } from './msk-stack';

export class Ec2Stack extends cdk.Stack {
  constructor(scope: Construct, id: string, mskStack: MskStack, props?: cdk.StackProps) {
    super(scope, id, props);

    // Reference the VPC and MSK Security Group from MskStack
    const vpc = mskStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    // Create EC2 Key Pair
    const keyPair = new ec2.CfnKeyPair(this, 'ShanEc2KeyPair', {
      keyName: 'shan-ec2-key', // Unique name for the key pair
    });

    // IAM Role for EC2 Instances
    const ec2Role = new iam.Role(this, 'Ec2MskRole', {
      assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),
      description: 'Role for EC2 instances to access MSK',
    });
    ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonMSKFullAccess'));
    ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonSSMManagedInstanceCore'));

    // Security Groups for EC2 Instances
    const producerSecurityGroup = new ec2.SecurityGroup(this, 'ProducerSecurityGroup', {
      vpc,
      securityGroupName: 'producer-ec2-sg',
      description: 'Security group for producer-ec2',
      allowAllOutbound: true,
    });

    const consumerSecurityGroup = new ec2.SecurityGroup(this, 'ConsumerSecurityGroup', {
      vpc,
      securityGroupName: 'consumer-ec2-sg',
      description: 'Security group for consumer-ec2',
    });

  }
}
```

```
allowAllOutbound: true,  
});  
  
// Add ingress rules to MSK SG for EC2 instances  
const mskPorts = [9092, 9094, 9096, 9098, 2181, 2182];  
mskPorts.forEach((port) => {  
  mskSecurityGroup.addIngressRule(  
    producerSecurityGroup,  
    ec2.Port.tcp(port),  
    `Allow port ${port} from producer-ec2`  
  );  
  mskSecurityGroup.addIngressRule(  
    consumerSecurityGroup,  
    ec2.Port.tcp(port),  
    `Allow port ${port} from consumer-ec2`  
  );  
});  
  
// EC2 Instances with the created key pair  
const producerEc2 = new ec2.Instance(this, 'ProducerEc2', {  
  instanceName: 'producer-ec2',  
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),  
  machineImage: ec2.MachineImage.latestAmazonLinux2(),  
  vpc,  
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },  
  availabilityZone: vpc.availabilityZones[0], // us-east-1a  
  securityGroup: producerSecurityGroup,  
  role: ec2Role,  
  keyName: keyPair.keyName, // Reference the created key pair  
});  
  
const consumerEc2 = new ec2.Instance(this, 'ConsumerEc2', {  
  instanceName: 'consumer-ec2',  
  instanceType: ec2.InstanceType.of(ec2.InstanceClass.T2, ec2.InstanceSize.MICRO),  
  machineImage: ec2.MachineImage.latestAmazonLinux2(),  
  vpc,  
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },  
  availabilityZone: vpc.availabilityZones[1], // us-east-1b  
  securityGroup: consumerSecurityGroup,  
  role: ec2Role,  
  keyName: keyPair.keyName, // Reference the created key pair  
});  
  
// Outputs  
new cdk.CfnOutput(this, 'ProducerEc2Id', { value: producerEc2.instanceId });  
new cdk.CfnOutput(this, 'ConsumerEc2Id', { value: consumerEc2.instanceId });  
new cdk.CfnOutput(this, 'ProducerPublicIp', { value: producerEc2.instancePublicIp });  
new cdk.CfnOutput(this, 'ConsumerPublicIp', { value: consumerEc2.instancePublicIp });  
new cdk.CfnOutput(this, 'KeyPairName', { value: keyPair.keyName });  
new cdk.CfnOutput(this, 'PrivateKey', {
```

```
value: keyPair.attrPrivateKey,  
      description: 'Private key material (save Grok 3 beta securely!)',  
    });  
  }  
}
```



3. bin/msk-ec2-cdk.ts

No changes are needed here—it remains as previously defined, orchestrating the deployment of both stacks.

Key Changes Explained

1. Adding `CfnKeyPair`:

- The `ec2.CfnKeyPair` resource creates a new key pair named `shan-ec2-key` in your AWS account.
- `keyName: 'shan-ec2-key'` specifies the name of the key pair, which must be unique within the region (`us-east-1` in your case).

2. Using the Key Pair in EC2 Instances:

- The `keyName` property of `producerEc2` and `consumerEc2` is set to `keyPair.keyName`, dynamically referencing the name of the key pair created by `CfnKeyPair`.

3. Outputting the Private Key:

- `keyPair.attrPrivateKey` retrieves the private key material (the contents of what would be a `.pem` file).
 - A `CfnOutput` named `PrivateKey` is added to display this sensitive data after deployment. You'll need to capture it manually.
-

Step-by-Step Deployment and Key Retrieval

1. Build and Deploy:

```
bash
```

```
cd msk-ec2-cdk  
npm run build
```



```
cdk deploy --all
```

Grok 3 **beta** ▾



- During deployment, CloudFormation will create the key pair and associate it with the EC2 instances.

2. Capture the Private Key:

- After deployment, the CDK will output the stack details, including the `PrivateKey`. Look for something like:

Outputs:

```
Ec2Stack.PrivateKey = -----BEGIN RSA PRIVATE KEY-----  
MIIEpAIBAAKCAQEA...  
-----END RSA PRIVATE KEY-----
```

- Copy the entire private key text (including `-----BEGIN RSA PRIVATE KEY-----` and `-----END RSA PRIVATE KEY-----`).

3. Save the Private Key Locally:

- On your local machine, create a file (e.g., `shan-ec2-key.pem`) and paste the private key:

bash

```
nano shan-ec2-key.pem  
# Paste the key, save, and exit (Ctrl+O, Enter, Ctrl+X in nano)  
chmod 400 shan-ec2-key.pem
```

4. SSH into the Instances:

- Use the public IPs from the outputs (`ProducerPublicIp`, `ConsumerPublicIp`):

bash

```
ssh -i shan-ec2-key.pem ec2-user@<producer-public-ip>  
ssh -i shan-ec2-key.pem ec2-user@<consumer-public-ip>
```

Important Considerations

1. Security of the Private Key:

- The private key is displayed in the CDK output, which is visible in your terminal and potentially in CloudFormation stack outputs. ~~Grok 3 beta~~ ~~data~~—do NOT share it or ~~commit~~ it to ~~version control~~.
- After capturing it, consider deleting the stack output from CloudFormation (manually via the AWS Console) or modifying the code to avoid printing it in future deployments (see below).

2. Persistent Key Pair:

- The `CfnKeyPair` resource creates a key pair that persists in your AWS account even if the stack is deleted, unless you explicitly delete it via the EC2 console or CLI:

bash

```
aws ec2 delete-key-pair --key-name shan-ec2-key --region us-east-1
```

- To automate deletion with stack destruction, you'd need a custom resource (not natively supported by `CfnKeyPair`).

3. Avoiding Private Key Output (Optional):

- If you don't want the private key exposed in outputs:

- Remove the `PrivateKey` output line (`new cdk.CfnOutput(this, 'PrivateKey', ...)`).
- Deploy once to create the key pair, retrieve the private key via the AWS CLI after deployment, and reuse it for future deployments:

bash

```
aws ec2 describe-key-pairs --key-names shan-ec2-key --region us-east-1 --include-publ
```

- Note: The CLI doesn't return the private key after creation—you'd need to capture it during the initial deployment.

4. Key Pair Uniqueness:

- If `shan-ec2-key` already exists in your AWS account (e.g., from a previous deployment or manual creation), deployment will fail with an error like "Key pair 'shan-ec2-key' already exists." You'll need to:

- Delete the existing key pair manually:

bash

```
aws ec2 delete-key-pair --key-name shan-ec2-key --region us-east-1
```

- Or use a unique name (e.g., append a timestamp or stack ID).



Alternative: Reuse Existing Key Pair

If you prefer not to create a key pair via CDK and want to reuse an existing one (e.g., `my-ec2-key`), keep the original approach:

typescript

```
keyName: 'my-ec2-key', // Hardcoded existing key pair name
```

This avoids the complexity of handling the private key output but requires you to manage the key pair separately.

Final Steps

- After deploying with the updated code and saving `shan-ec2-key.pem`, proceed with the test code setup (Node.js, `kafkajs`, etc.) as described earlier.
- Verify SSH access and ensure the producer and consumer can interact with the MSK cluster.

Let me know if you need help with deployment, key management, or further customization!

how to create dist folder , also the intermediate js files and .d.ts also created in same dir

To create a `dist` folder containing the compiled JavaScript (`.js`) files along with TypeScript declaration files (`.d.ts`) in the same directory using the AWS CDK project you've set up, you'll need to configure the TypeScript compiler (`tsc`) via the `tsconfig.json` file. By default, CDK projects generate intermediate `.js` files in the `lib` directory alongside the `.ts` files, but you can redirect all outputs (`.js` and `.d.ts`) to a `dist` folder.

Here's how to achieve this step-by-step for your `msk-ec2-cdk` project:

Step 1: Update `tsconfig.json`

Modify the `tsconfig.json` file in the root of your project to output compiled files to a `dist` folder and enable declaration file generation.

Grok 3 **beta** ▾



1. Open `tsconfig.json`:

- It's likely already present from the `cdk init` command. If not, create it with `tsc --init`.

2. Update the File:

Replace its contents with the following configuration:

json

```
{
  "compilerOptions": {
    "target": "ES2018", // Target JavaScript version
    "module": "commonjs", // Module system for Node.js
    "lib": ["es2018"], // Libraries to include
    "declaration": true, // Generate .d.ts files
    "outDir": "./dist", // Output directory for .js and .d.ts files
    "rootDir": "./", // Root directory of input files
    "strict": true, // Enable strict type checking
    "esModuleInterop": true, // Enable interop between CommonJS and ES Modules
    "skipLibCheck": true, // Skip type checking of declaration files
    "forceConsistentCasingInFileNames": true // Ensure consistent file casing
  },
  "include": ["bin/**/*.ts", "lib/**/*.ts"], // Files to compile
  "exclude": ["node_modules", "dist"] // Files to exclude
}
```

Key Changes:

- `"outDir": "./dist"` : Specifies that all compiled `.js` and `.d.ts` files go into the `dist` folder.
- `"declaration": true` : Ensures TypeScript generates `.d.ts` declaration files alongside `.js` files.
- `"rootDir": "./"` : Sets the root directory to preserve the folder structure (e.g., `bin/` and `lib/`) in `dist`.
- `"include"` : Ensures both `bin` and `lib` directories are compiled.

Step 2: Update `package.json` Scripts

Modify the `build` script in `package.json` to use `tsc` for compilation and optionally clean the `dist` folder beforehand.

1. Open `package.json`:

- It's in the root of your `msk-ec2-cdk` project.

Grok 3 **beta** ▾



2. Update the `scripts` Section:

Replace the existing `build` script (and add a `clean` script if desired):

```
json

{
  "scripts": {
    "build": "tsc",
    "clean": "rm -rf dist",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  }
}
```

- `"build": "tsc"` : Compiles TypeScript files according to `tsconfig.json`.
- `"clean": "rm -rf dist"` : (Optional) Deletes the `dist` folder before building to ensure a fresh compilation.

Step 3: Compile the Project

Run the build command to generate the `dist` folder with `.js` and `.d.ts` files.

1. Clean (Optional):

```
bash
```

```
npm run clean
```

2. Build:

```
bash
```

```
npm run build
```

3. Verify Output:

After running the build, your project structure should look like this:

```
msk-ec2-cdk/
  └── bin/
```



```

    |   └── msk-ec2-cdk.ts
    └── lib/
        ├── msk-stack.ts
        └── ec2-stack.ts
    └── dist/
        ├── bin/
        │   ├── msk-ec2-cdk.js
        │   └── msk-ec2-cdk.d.ts
        └── lib/
            ├── msk-stack.js
            ├── msk-stack.d.ts
            ├── ec2-stack.js
            └── ec2-stack.d.ts
    └── package.json
    └── tsconfig.json
    └── ... (other files)

```

- The `dist` folder mirrors the `bin` and `lib` structure, containing both `.js` (compiled JavaScript) and `.d.ts` (TypeScript declaration) files.
-

Step 4: Update CDK Entry Point

Since the compiled files are now in `dist`, update the CDK entry point to reference the compiled JavaScript files.

1. Modify `bin/msk-ec2-cdk.ts`:

- Ensure it imports from the TypeScript files (not `.js`), as CDK will handle the runtime execution post-compilation:

```

typescript

#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MskStack } from '../lib/msk-stack';
import { Ec2Stack } from '../lib/ec2-stack';

const app = new cdk.App();

const mskStack = new MskStack(app, 'MskStack', {
  env: { region: 'us-east-1' },
});

const ec2Stack = new Ec2Stack(app, 'Ec2Stack', mskStack, {

```

```
    env: { region: 'us-east-1' },
  });
  ec2Stack.addDependency(mskStack);
```

Grok 3 beta ▾



- No change is needed here because the imports (`../lib/msk-stack`) remain TypeScript-based, and `tsc` handles the compilation to `dist` .

2. Run CDK Commands:

- Use the `cdk` CLI as usual—it will execute the compiled `.js` files in `dist` :

bash

```
npm run build # Compile TS to JS
cdk synth      # Synthesize CloudFormation templates
cdk deploy --all # Deploy both stacks
```

Step 5: Verify and Troubleshoot

1. Check the `dist` Folder:

- After `npm run build`, confirm that `dist/bin/msk-ec2-cdk.js` , `dist/lib/msk-stack.js` , and `dist/lib/ec2-stack.js` exist alongside their `.d.ts` counterparts.

2. Run CDK Synth:

- Ensure the synthesis works:

bash

```
cdk synth
```

- This generates CloudFormation templates in the `cdk.out` directory based on the compiled `.js` files.

3. Troubleshooting:

- **No `dist` Folder:** Check if `tsc` ran successfully (`npm run build` should show no errors). Verify `outDir` in `tsconfig.json` .
- **Missing `.d.ts` Files:** Ensure `"declaration": true` is set in `tsconfig.json` .
- **Path Errors:** If CDK can't find the compiled files, confirm the `include` array in `tsconfig.json` covers all `.ts` files.



Optional: Clean Build in Workflow

To ensure a fresh `dist` folder each time, combine `clean` and `build` in a single script:

1. Update `package.json`:

```
json
```

```
{  
  "scripts": {  
    "build": "npm run clean && tsc",  
    "clean": "rm -rf dist",  
    "watch": "tsc -w",  
    "test": "jest",  
    "cdk": "cdk"  
  }  
}
```

2. Run:

```
bash
```

```
npm run build
```

Final Project Structure

After building, your project will look like this:

```
msk-ec2-cdk/  
  └── bin/  
  |   └── msk-ec2-cdk.ts  
  └── lib/  
  |   └── msk-stack.ts  
  |   └── ec2-stack.ts  
  └── dist/  
      └── bin/  
      |   └── msk-ec2-cdk.js  
      |   └── msk-ec2-cdk.d.ts  
      └── lib/  
          └── msk-stack.js
```

```

|   |   └── msk-stack.d.ts
|   |   └── ec2-stack.js
|   |   └── ec2-stack.d.ts
└── package.json
└── tsconfig.json
└ ... (other files)

```

Grok 3 beta ▾



- The `.js` files are the compiled JavaScript executed by CDK.
 - The `.d.ts` files provide TypeScript type definitions, useful if you were importing these stacks into another TypeScript project (though not strictly necessary for CDK runtime).
-

Deploy and Test

1. Deploy:

bash

```

npm run build
cdk deploy --all

```



2. Use the Output:

- The deployed stacks will still work as before, and you can use the `PrivateKey` output from `Ec2Stack` to SSH into the instances.

This setup ensures all intermediate `.js` and `.d.ts` files are neatly organized in the `dist` folder. Let me know if you need further assistance!



There are two ecs fargate services with java springboot framework one will send message to msk cluster and other ecs fargate will consumer the message in cluer. Provide the cdk code for creating ecs fargate cluster. Also make necessary changes in msk cluster cdk code to be accessed by ecs fargate services to send and receive message. provide sample java springboot application to be run in ecs to produce and consume msgs.

1 - provide updated cdk code for msk provisioned clustered with policy modified. Do we really need SASL/SCRAM in msk provisioned? Only Iam role beasec acces is enough?

- 2 - Provide ecs fargate cdk code with required kms keys
- 3 - provide ~~Grok 3~~ **beta** code for produce and consumer in  
- corresponding ecs service
- 4 - what are the steps to be made to run the java code in ecs cluster
- 5 - any ecr and image need to be maintained?

15 web pages

Below, I'll address your requirements step-by-step, providing updated CDK code for the MSK cluster, new CDK code for the ECS Fargate cluster with two services, sample Java Spring Boot code for producing and consuming messages, and detailed instructions for running the application, including ECR and Docker image management.

Step 1: Updated CDK Code for MSK Provisioned Cluster with IAM Role-Based Access

Do We Need SASL/SCRAM if IAM Role-Based Access is Enough?

- **Answer:** No, SASL/SCRAM is not strictly necessary if you're using IAM role-based authentication exclusively. IAM authentication (port 9098) is sufficient for securing access to the MSK cluster, especially since it integrates seamlessly with AWS services like ECS Fargate via IAM roles. SASL/SCRAM (port 9096) is an alternative authentication mechanism that requires managing credentials in Secrets Manager, adding complexity. For your use case (ECS Fargate services interacting with MSK), IAM-based access is simpler and more secure due to its native AWS integration.
- **Decision:** I'll remove SASL/SCRAM and use only IAM authentication, updating the MSK cluster configuration accordingly.

Updated `lib/msk-stack.ts`

typescript 

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';

export class MskStack extends cdk.Stack {
  public readonly vpc: ec2.Vpc;
  public readonly mskSecurityGroup: ec2.SecurityGroup;
```



```
public readonly mskClusterArn: string;
constructor(scope: Construct, id: string, props?: cdk.StackProps) {
  super(scope, id, props);

  // Create VPC with public and private subnets
  this.vpc = new ec2.Vpc(this, 'ShanMskVpc', {
    vpcName: 'shan-msk-vpc',
    cidr: '10.0.0.0/16',
    maxAzs: 2,
    subnetConfiguration: [
      { cidrMask: 24, name: 'shan-msk-subnet', subnetType: ec2.SubnetType.PRIVATE_ISOLATED },
      { cidrMask: 24, name: 'shan-ecs-subnet', subnetType: ec2.SubnetType.PUBLIC },
    ],
  });

  // Security Group for MSK
  this.mskSecurityGroup = new ec2.SecurityGroup(this, 'ShanMskSecurityGroup', {
    vpc: this.vpc,
    securityGroupName: 'shan-msk-sg',
    description: 'Security group for shan-msk-cluster',
    allowAllOutbound: true,
  });

  const mskPorts = [9098, 2181]; // IAM (9098) and ZooKeeper plaintext (2181)
  mskPorts.forEach((port) => {
    this.mskSecurityGroup.addIngressRule(
      this.mskSecurityGroup,
      ec2.Port.tcp(port),
      `Allow port ${port} for MSK internal communication`
    );
  });

  // KMS Key for Data Encryption
  const mskKmsKey = new kms.Key(this, 'ShanMskKmsKey', {
    alias: 'shan-msk-kms-key',
    description: 'KMS key for MSK data encryption',
    enableKeyRotation: true,
  });

  // KMS Key for CloudWatch Logs
  const cloudwatchKmsKey = new kms.Key(this, 'ShanCloudwatchKmsKey', {
    alias: 'shan-cloudwatch-kms-key',
    description: 'KMS key for CloudWatch Logs encryption',
    enableKeyRotation: true,
  });

  // CloudWatch Log Group
  const logGroup = new logs.LogGroup(this, 'ShanMskLogGroup', {
    logGroupName: 'shan-msk-logs',
```

```

    retention: logs.RetentionDays.ONE_MONTH,
    encryptionKey: cloudwatchKmsKey, Grok 3 beta ▾
  });

// MSK Cluster with IAM authentication only
const mskCluster = new msk.CfnCluster(this, 'ShanMskCluster', {
  clusterName: 'shan-msk-cluster',
  kafkaVersion: '3.7.x',
  numberOfWorkers: 2,
  workerNodeGroupInfo: {
    instanceType: 'kafka.t2.small',
    storageInfo: { ebsStorageInfo: { volumeSize: 10 } },
    clientSubnets: this.vpc.privateSubnets.map((subnet) => subnet.subnetId),
    securityGroups: [this.mskSecurityGroup.securityGroupId],
  },
  encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyArn } },
  clientAuthentication: {
    sasl: { iam: { enabled: true } }, // Only IAM authentication
  },
  loggingInfo: {
    brokerLogs: { cloudWatchLogs: { enabled: true, logGroup: logGroup.logGroupName } },
  },
});

this.mskClusterArn = mskCluster.attrArn;

// Outputs
new cdk.CfnOutput(this, 'ClusterArn', { value: this.mskClusterArn });
}
}

```

Changes:

- Removed SASL/SCRAM and Secrets Manager-related code (`saslSecret` and `secretTargetAttachment`).
- Updated `mskPorts` to include only IAM (9098) and ZooKeeper plaintext (2181), as SASL/SCRAM (9096) and TLS (9094, 2182) are no longer needed.
- Simplified `clientAuthentication` to use only IAM.

Step 2: CDK Code for ECS Fargate Cluster with Required KMS Keys

lib/ecs-stack.ts

This creates an ECS Fargate cluster with two services: one for producing messages and one for consuming messages, using IAM roles for MSK access.



typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as ecr from 'aws-cdk-lib/aws-ecr';
import * as iam from 'aws-cdk-lib/aws-iam';
import * as kms from 'aws-cdk-lib/aws-kms';
import { MskStack } from './msk-stack';

export class EcsStack extends cdk.Stack {
  constructor(scope: Construct, id: string, mskStack: MskStack, props?: cdk.StackProps) {
    super(scope, id, props);

    // Reference MSK resources
    const vpc = mskStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    // KMS Key for ECS Task Logs
    const ecsLogsKmsKey = new kms.Key(this, 'EcsLogsKmsKey', {
      alias: 'shan-ecs-logs-kms-key',
      description: 'KMS key for ECS Fargate logs',
      enableKeyRotation: true,
    });

    // ECS Cluster
    const cluster = new ecs.Cluster(this, 'ShanEcsCluster', {
      vpc,
      clusterName: 'shan-ecs-cluster',
    });

    // IAM Role for ECS Tasks (MSK access)
    const taskRole = new iam.Role(this, 'EcsTaskRole', {
      assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
      description: 'Role for ECS tasks to access MSK',
    });
    taskRole.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonMSKFullAccess'));

    // Task Execution Role (for pulling images and logging)
    const executionRole = new iam.Role(this, 'EcsExecutionRole', {
      assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
      description: 'Role for ECS task execution',
    });
    executionRole.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('service-role/AmazonMSKFullAccess'));
  }
}
```

```
// ECR Repositories
const producerRepo = new ecr.Repository(this, 'producerRepo', { repositoryName: 'shan-producer' })
const consumerRepo = new ecr.Repository(this, 'ConsumerRepo', { repositoryName: 'shan-consumer' })

// Producer Task Definition
const producerTaskDef = new ecs.FargateTaskDefinition(this, 'ProducerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
  taskRole,
  executionRole,
});
producerTaskDef.addContainer('ProducerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(producerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'producer',
    logRetention: logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

// Consumer Task Definition
const consumerTaskDef = new ecs.FargateTaskDefinition(this, 'ConsumerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
  taskRole,
  executionRole,
});
consumerTaskDef.addContainer('ConsumerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(consumerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'consumer',
    logRetention: logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

// Security Group for ECS Services
const ecsSecurityGroup = new ec2.SecurityGroup(this, 'EcsSecurityGroup', {
  vpc,
  securityGroupName: 'shan-ecs-sg',
  description: 'Security group for ECS Fargate services',
  allowAllOutbound: true,
});

// Allow ECS to access MSK
mskPorts.forEach((port) => {
  mskSecurityGroup.addIngressRule(
    ecsSecurityGroup,
```

```

    ec2.Port.tcp(port),
    `Allow port ${port} from ECS services` Grok 3 beta ▾
  );
});

// Producer Service
new ecs.FargateService(this, 'ProducerService', {
  cluster,
  taskDefinition: producerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});

// Consumer Service
new ecs.FargateService(this, 'ConsumerService', {
  cluster,
  taskDefinition: consumerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});

// Outputs
new cdk.CfnOutput(this, 'ProducerRepoUri', { value: producerRepo.repositoryUri });
new cdk.CfnOutput(this, 'ConsumerRepoUri', { value: consumerRepo.repositoryUri });
}
}

```

Notes:

- Added a KMS key for ECS logs (`ecsLogsKmsKey`).
- Created ECR repositories for producer and consumer images.
- Defined IAM roles for task execution and MSK access.
- Configured security group rules to allow ECS services to access MSK on port 9098.

Step 3: Java Spring Boot Code for Producer and Consumer

Producer Application (`shan-producer`)

1. Project Structure:

Grok 3 **beta** ▾

```
shan-producer/
├── src/main/java/com/example/producer/
│   ├── ProducerApplication.java
│   ├── KafkaProducerService.java
├── src/main/resources/
│   └── application.yml
├── Dockerfile
└── build.gradle
```

2. build.gradle:

```
gradle

plugins {
    id 'org.springframework.boot' version '3.2.0'
    id 'io.spring.dependency-management' version '1.1.0'
    id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    implementation 'org.springframework.kafka:spring-kafka'
    implementation 'software.amazon.msk:aws-msk-iam-auth:2.0.3'
}
```

3. src/main/resources/application.yml:

```
yaml

spring:
  kafka:
    bootstrap-servers: ${BOOTSTRAP_SERVERS} # Set via environment variable
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
    properties:
      security.protocol: SASL_SSL
      sasl.mechanism: AWS_MSK_IAM
```

```
sasl.jaas.config: software.amazon.msk.auth.iam.IAMLoginModule required;
sasl.client.callback.handler.class: grok 3s beta com.amazonaws.msk.auth.iam.IAMClientCallbackHandler
```

4. src/main/java/com/example/producer/ProducerApplication.java :

java

```
package com.example.producer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;

@SpringBootApplication
@EnableScheduling
public class ProducerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class, args);
    }

    @Bean
    public KafkaProducerService kafkaProducerService() {
        return new KafkaProducerService();
    }

    @Scheduled(fixedRate = 5000)
    public void produceMessages() {
        kafkaProducerService().sendMessage("order", "Order-" + System.currentTimeMillis(), "PI")
    }
}
```

5. src/main/java/com/example/producer/KafkaProducerService.java :

java

```
package com.example.producer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class KafkaProducerService {
```



```

@Autowired
private KafkaTemplate<String, String> kafkaTemplate;
Grok 3 beta

public void sendMessage(String topic, String key, String value) {
    kafkaTemplate.send(topic, key, value);
    System.out.println("Produced message: " + key + " -> " + value);
}
}

```

6. Dockerfile :

dockerfile

```

FROM openjdk:17-jdk-slim
COPY build/libs/shan-producer-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]

```

Consumer Application (shan-consumer)

1. Project Structure:

```

shan-consumer/
├── src/main/java/com/example/consumer/
│   ├── ConsumerApplication.java
│   ├── KafkaConsumerService.java
├── src/main/resources/
│   └── application.yml
├── Dockerfile
└── build.gradle

```

2. build.gradle :

Same as the producer's `build.gradle` .

3. src/main/resources/application.yml :

Same as the producer's `application.yml` .

4. src/main/java/com/example/consumer/ConsumerApplication.java :

java

```

package com.example.consumer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class ConsumerApplication {

```



```

public static void main(String[] args) {
    SpringApplication.run(ConsumerApplication.class, args);
}

@Bean
public KafkaConsumerService kafkaConsumerService() {
    return new KafkaConsumerService();
}
}

```

5. src/main/java/com/example/consumer/KafkaConsumerService.java :

java

```

package com.example.consumer;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumerService {

    @KafkaListener(topics = "order", groupId = "order-consumer-group")
    public void consumeMessage(String message) {
        System.out.println("Consumed message: " + message);
    }
}

```

6. Dockerfile :

dockerfile

```

FROM openjdk:17-jdk-slim
COPY build/libs/shan-consumer-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]

```

Step 4: Steps to Run the Java Code in the ECS Cluster

1. Build the Spring Boot Applications Locally:

- For shan-producer :

bash

cd shan-producer
./gradlew build

Grok 3 beta ▾

- For shan-consumer :

bash

```
cd shan-consumer  
./gradlew build
```

2. Build Docker Images:

- For shan-producer :

bash

```
docker build -t shan-producer:latest .
```

- For shan-consumer :

bash

```
docker build -t shan-consumer:latest .
```

3. Tag and Push to ECR:

- Get the ECR URIs from the CDK outputs (ProducerRepoUri , ConsumerRepoUri).
- Tag and push:

bash

```
docker tag shan-producer:latest <producer-repo-uri>:latest  
docker push <producer-repo-uri>:latest  
docker tag shan-consumer:latest <consumer-repo-uri>:latest  
docker push <consumer-repo-uri>:latest
```

- Authenticate with ECR first:

bash

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-st
```

4. Deploy the CDK Stacks:

- Update bin/msk-ec2-cdk.ts to include EcsStack :
- typescript

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MskStack } from '../lib/msk-stack';
import { EcsStack } from '../lib/ecs-stack';

const app = new cdk.App();

const mskStack = new MskStack(app, 'MskStack', { env: { region: 'us-east-1' } });
const ecsStack = new EcsStack(app, 'EcsStack', mskStack, { env: { region: 'us-east-1' } });

ecsStack.addDependency(mskStack);
```

- Build and deploy:

bash

```
npm run build
cdk deploy --all
```

5. Set Environment Variables:

- After deployment, get the MSK bootstrap servers (IAM authenticated) from the MSK console.
- Update the ECS task definitions manually in the AWS Console to add `BOOTSTRAP_SERVERS` as an environment variable (e.g., `b-1.shan-msk-cluster.xxx.kafka.us-east-1.amazonaws.com:9098, b-2.shan-msk-cluster.xxx.kafka.us-east-1.amazonaws.com:9098`).

6. Create the order Topic:

- Use the Kafka CLI or a script on an EC2 instance with MSK access:

bash

```
kafka-topics.sh --create --bootstrap-server <bootstrap-servers>:9098 --replication-factor
```

Where `client.properties` contains:

```
security.protocol=SASL_SSL
sasl.mechanism=AWS_MSK_IAM
sasl.jaas.config=software.amazon.msk.auth.iam.IAMLoginModule required;
sasl.client.callback.handler.class=software.amazon.msk.auth.iam.IAMClientCallbackHandler
```

7. Verify Execution:

- Check CloudWatch Logs for the `producer` and `consumer` log groups to see messages being produced and consumed.

Grok 3 beta



Step 5: ECR and Image Management

- Do We Need ECR and Images?

- Yes, ECR (Elastic Container Registry) is required to store the Docker images for the Spring Boot applications. ECS Fargate pulls these images from ECR to run the containers.
- The CDK code creates two ECR repositories (`shan-producer` and `shan-consumer`), and you must build, tag, and push the Docker images to these repositories manually before deployment.

- Maintenance:

- Tag images with versions (e.g., `latest`, `v1.0`) to manage updates.
- Use lifecycle policies in ECR to clean up old images and reduce costs.

Summary

- MSK Cluster:** Updated to use only IAM authentication, removing SASL/SCRAM complexity.
- ECS Fargate Cluster:** Created with producer and consumer services, using KMS-encrypted logs and IAM roles for MSK access.
- Java Code:** Provided Spring Boot applications for producing and consuming messages with MSK IAM authentication.
- Deployment Steps:** Detailed process to build, push, and run the applications in ECS Fargate.
- ECR:** Essential for storing and managing Docker images, integrated into the CDK setup.

Let me know if you need furthe

use maven instead of gradle. and if ec2 serviceis not there , how to create topics from outside? is ther any bastion host to be used

Below, I'll update the Java Spring Boot applications to use Maven instead of Gradle, address how to create topics from outside the ECS services (since there's no EC2 service in this setup), and discuss whether a bastion host is necessary. I'll also provide guidance on creating topics externally without modifying the existing CDK code significantly.



Step 1: Updated Java Spring Boot Code with Maven

Producer Application (shan-producer)

1. Project Structure:

```
shan-producer/
├── src/main/java/com/example/producer/
│   ├── ProducerApplication.java
│   ├── KafkaProducerService.java
├── src/main/resources/
│   └── application.yml
├── Dockerfile
└── pom.xml
```

2. pom.xml :

```
pom.xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.0</version>
    <relativePath/>
  </parent>
  <groupId>com.example</groupId>
  <artifactId>shan-producer</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>shan-producer</name>
  <description>Producer for MSK</description>

  <properties>
    <java.version>17</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.kafka</groupId>
```

```

<artifactId>spring-kafka</artifactId>
</dependency>           Grok 3  beta  ▾
<dependency>
  <groupId>software.amazon.msk</groupId>
  <artifactId>aws-msk-iam-auth</artifactId>
  <version>2.0.3</version>
</dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>

```

3. `src/main/resources/application.yml` , `ProducerApplication.java` , `KafkaProducerService.java` , and `Dockerfile` :

- These remain unchanged from the previous response (same as the Gradle version).

Consumer Application (shan-consumer)

1. Project Structure:

```

shan-consumer/
├── src/main/java/com/example/consumer/
│   ├── ConsumerApplication.java
│   └── KafkaConsumerService.java
├── src/main/resources/
│   └── application.yml
├── Dockerfile
└── pom.xml

```

2. `pom.xml` :

```

xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>

```

```

<artifactId>spring-boot-starter-parent</artifactId>
<version>3.2.0</version>  Grok 3  beta  ▾
<relativePath/>

</parent>
<groupId>com.example</groupId>
<artifactId>shan-consumer</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>shan-consumer</name>
<description>Consumer for MSK</description>

<properties>
  <java.version>17</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
  </dependency>
  <dependency>
    <groupId>software.amazon.msk</groupId>
    <artifactId>aws-msk-iam-auth</artifactId>
    <version>2.0.3</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>

```

3. `src/main/resources/application.yml` , `ConsumerApplication.java` , `KafkaConsumerService.java` , and `Dockerfile` :

- These remain unchanged from the previous response.

Build Commands (Replacing Gradle)

- For `shan-producer` :

```
bash
cd shan-producer
mvn clean package
```

- For shan-consumer :

```
bash
cd shan-consumer
mvn clean package
```

- This generates `target/shan-producer-0.0.1-SNAPSHOT.jar` and `target/shan-consumer-0.0.1-SNAPSHOT.jar` , which the `Dockerfile` uses.

Step 2: Creating Topics from Outside Without EC2 Services

Since your setup now uses ECS Fargate services instead of EC2 instances, you need a way to create the `order` topic externally. Here are your options, including whether a bastion host is necessary:

Options to Create Topics

1. Local Machine or CI/CD Pipeline with AWS CLI and Kafka Tools:

- Steps:

1. Install Kafka tools locally:

```
bash
wget https://archive.apache.org/dist/kafka/3.7.0/kafka_2.13-3.7.0.tgz
tar -xzf kafka_2.13-3.7.0.tgz
cd kafka_2.13-3.7.0
```

2. Download the MSK IAM auth JAR:

```
bash
wget https://github.com/aws/aws-msk-iam-auth/releases/download/v2.0.3/aws-msk-iam-aut
```

3. Configure AWS CLI with credentials that have MSK access (`AmazonMSKFullAccess`).
4. Get the MSK bootstrap servers (IAM) from the MSK console or CDK output.
5. Create a `client.properties` file:

security.protocol=SASL_SSL
 sasl.mechanism=AWS_MSK_IAM
 sasl.jaas.config=software.amazon.msk.auth.iam.IAMLoginModule required;
 sasl.client.callback.handler.class=software.amazon.msk.auth.iam.IAMClientCallbackHand

6. Run the topic creation command:

bash

```
bin/kafka-topics.sh --create --bootstrap-server <bootstrap-servers>:9098 --replicatio
```

- **Pros:** No additional infrastructure needed.
- **Cons:** Requires local setup and network access to MSK (see below).

2. Bastion Host (EC2 Instance):

- **Why Use a Bastion Host?**
 - Your MSK cluster is in private subnets (**PRIVATE_ISOLATED**), meaning it's not directly accessible from the public internet unless public access is enabled (not recommended for security).
 - A bastion host in a public subnet of the same VPC can access the MSK cluster internally, avoiding the need to expose MSK publicly.
- **CDK Modification:** Add a bastion host to `lib/ecs-stack.ts` (example below).
- **Steps:**
 1. SSH into the bastion host.
 2. Install Kafka tools and MSK IAM auth as above.
 3. Create the topic using the same command.

3. AWS Lambda Function:

- Use a Lambda function in the VPC to run a Kafka client and create the topic programmatically.
- **Pros:** Serverless, no manual SSH.
- **Cons:** Requires coding a Lambda function (e.g., in Java or Python with Kafka libraries).

Do We Need a Bastion Host?

- **Not Strictly Necessary:** If you enable public access to MSK (via `connectivityInfo.publicAccess.type: 'SERVICE_PROVIDED_EIPS'` in the MSK CDK code) and configure your local machine's IAM credentials, you can create topics directly from outside without a bastion host. However, this exposes MSK to the internet, which may not align with security best practices.

- **Recommended:** Use a bastion host for a secure, internal approach without modifying MSK's private nature.

Grok 3 beta ▾



Adding a Bastion Host to `lib/ecs-stack.ts`

Here's how to add a bastion host for topic creation:

typescript

```
// Add to imports
import * as ec2 from 'aws-cdk-lib/aws-ec2';

// Inside EcsStack class, after ECS services
const bastionHost = new ec2.BastionHostLinux(this, 'BastionHost', {
  vpc,
  subnetSelection: { subnetType: ec2.SubnetType.PUBLIC },
  instanceName: 'shan-bastion',
  securityGroup: ecsSecurityGroup, // Reuse ECS SG for simplicity
});
bastionHost.instance.role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonMSKFullAccess'));

// Output
new cdk.CfnOutput(this, 'BastionHostPublicIp', { value: bastionHost.instancePublicIp });
```



- **Deploy:** After adding this, redeploy with `cdk deploy --all`.
- **Use:** SSH into the bastion host (`ssh -i <key.pem> ec2-user@<bastion-public-ip>`), install Kafka tools, and create the topic as shown above.

Step 3: Updated Deployment Steps (No EC2)

1. Build and Package Java Apps:

- For `shan-producer`:

bash

```
cd shan-producer
mvn clean package
docker build -t shan-producer:latest .
```

- For `shan-consumer`:

bash

```
cd shan-consumer
mvn clean package
docker build -t shan-consumer:latest .
```



2. Push to ECR:

- Tag and push as before, using the ECR URIs from CDK outputs.

3. Create the order Topic:

- **Option 1 (Local):** Configure your local machine and run the `kafka-topics.sh` command (ensure MSK is publicly accessible or you're in the VPC).
- **Option 2 (Bastion Host):** Deploy the updated CDK with the bastion host, SSH in, and create the topic.

4. Deploy CDK Stacks:

- Ensure `bin/msk-ec2-cdk.ts` includes both stacks and deploy:

```
bash
```

```
npm run build
cdk deploy --all
```



5. Set Environment Variables:

- Update ECS task definitions in the AWS Console to add `BOOTSTRAP_SERVERS` with the MSK IAM bootstrap servers.

6. Verify:

- Check CloudWatch Logs for producer and consumer activity.

Step 4: ECR and Image Management (Unchanged)

- **ECR:** Still required to store Docker images (`shan-producer` and `shan-consumer`).
 - **Images:** Build and push images as described, no changes needed from the previous setup.
-

Summary

- **Maven:** Replaced Gradle with Maven in the Spring Boot projects.
- **Topic Creation Without EC2:** Use a local machine (with public MSK access) or a bastion host (recommended for private MSK). Added bastion host CDK code as an option.
- **CDK:** MSK updated for IAM-only access; ECS Fargate code remains functional with the bastion host addition optional.

Let me know if you need further details or assistance with the bastion host setup!

thumb up, thumb down, reply, comment, share

ok, provide the outcome as below

- 1 - Step by step of creation of msk cluster , ecs fargate cluster , ecr repo with clear separation of cdk code along with bastion code
- 2 - explain the necessary policies attached for accessing mask cluster, cloudwatch, ecs, bastion
- 3 - have separate cdk code for different stacks
- 4 - provide a summary description of each steps separately so that it can be given to team members as what and how to do/implement msk cluster/ecs service , to make them understand

Below is a comprehensive response addressing your requirements, including step-by-step creation of the MSK cluster, ECS Fargate cluster, ECR repositories, and a bastion host with separate CDK stacks, necessary IAM policies, and detailed summaries for team members.

1. Step-by-Step Creation with CDK Code Separation

Project Structure



```
msk-ecs-cdk/
├── bin/
│   └── msk-ecs-cdk.ts
├── lib/
│   ├── msk-stack.ts
│   ├── ecs-stack.ts
│   ├── ecr-stack.ts
│   └── bastion-stack.ts
└── package.json
```

```
└── tsconfig.json
└── ... (other files)
```

Grok 3 beta ▾



Step 1: MSK Cluster Creation (lib/msk-stack.ts)

typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';

export class MskStack extends cdk.Stack {
  public readonly vpc: ec2.Vpc;
  public readonly mskSecurityGroup: ec2.SecurityGroup;
  public readonly mskClusterArn: string;

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    this.vpc = new ec2.Vpc(this, 'ShanMskVpc', {
      vpcName: 'shan-msk-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2,
      subnetConfiguration: [
        { cidrMask: 24, name: 'shan-msk-subnet', subnetType: ec2.SubnetType.PRIVATE_ISOLATED },
        { cidrMask: 24, name: 'shan-public-subnet', subnetType: ec2.SubnetType.PUBLIC },
      ],
    });
  }

  this.mskSecurityGroup = new ec2.SecurityGroup(this, 'ShanMskSecurityGroup', {
    vpc: this.vpc,
    securityGroupName: 'shan-msk-sg',
    description: 'Security group for shan-msk-cluster',
    allowAllOutbound: true,
  });

  const mskPorts = [9098, 2181]; // IAM and ZooKeeper plaintext
  mskPorts.forEach((port) => {
    this.mskSecurityGroup.addIngressRule(
      this.mskSecurityGroup,
      ec2.Port.tcp(port),
      `Allow port ${port} for MSK internal communication`
    );
  });

  const mskKmsKey = new kms.Key(this, 'ShanMskKmsKey', {
```

```

alias: 'shan-msk-kms-key',
description: 'KMS key for MSK data encryption' Grok beta ▾
enableKeyRotation: true,
});

const cloudwatchKmsKey = new kms.Key(this, 'ShanCloudwatchKmsKey', {
alias: 'shan-cloudwatch-kms-key',
description: 'KMS key for CloudWatch Logs encryption',
enableKeyRotation: true,
});

const logGroup = new logs.LogGroup(this, 'ShanMskLogGroup', {
logGroupName: 'shan-msk-logs',
retention: logs.RetentionDays.ONE_MONTH,
encryptionKey: cloudwatchKmsKey,
});

const mskCluster = new msk.CfnCluster(this, 'ShanMskCluster', {
clusterName: 'shan-msk-cluster',
kafkaVersion: '3.7.x',
numberOfBrokerNodes: 2,
brokerNodeGroupInfo: {
instanceType: 'kafka.t2.small',
storageInfo: { ebsStorageInfo: { volumeSize: 10 } },
clientSubnets: this.vpc.privateSubnets.map((subnet) => subnet.subnetId),
securityGroups: [this.mskSecurityGroup.securityGroupId],
},
encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyArn } },
clientAuthentication: { sasl: { iam: { enabled: true } } },
loggingInfo: { brokerLogs: { cloudWatchLogs: { enabled: true, logGroup: logGroup.logGroupName } } },
});

this.mskClusterArn = mskCluster.attrArn;

new cdk.CfnOutput(this, 'ClusterArn', { value: this.mskClusterArn });
}
}

```

Step 2: ECR Repositories (lib/ecr-stack.ts)

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ecr from 'aws-cdk-lib/aws-ecr';

export class EcrStack extends cdk.Stack {
  public readonly producerRepo: ecr.Repository;

```



```

public readonly consumerRepo: ecr.Repository;
constructor(scope: Construct, id: string, props?: cdk.StackProps) {
  super(scope, id, props);

  this.producerRepo = new ecr.Repository(this, 'ProducerRepo', { repositoryName: 'shan-producer' });
  this.consumerRepo = new ecr.Repository(this, 'ConsumerRepo', { repositoryName: 'shan-consumer' });

  new cdk.CfnOutput(this, 'ProducerRepoUri', { value: this.producerRepo.repositoryUri });
  new cdk.CfnOutput(this, 'ConsumerRepoUri', { value: this.consumerRepo.repositoryUri });
}
}

```

Step 3: ECS Fargate Cluster (lib/ecs-stack.ts)

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as iam from 'aws-cdk-lib/aws-iam';
import * as kms from 'aws-cdk-lib/aws-kms';
import { MskStack } from './msk-stack';
import { EcrStack } from './ecr-stack';

export class EcsStack extends cdk.Stack {
  constructor(scope: Construct, id: string, mskStack: MskStack, ecrStack: EcrStack, props?: cdk.StackProps) {
    super(scope, id, props);

    const vpc = mskStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    const ecsLogsKmsKey = new kms.Key(this, 'EcsLogsKmsKey', {
      alias: 'shan-ecs-logs-kms-key',
      description: 'KMS key for ECS Fargate logs',
      enableKeyRotation: true,
    });

    const cluster = new ecs.Cluster(this, 'ShanEcsCluster', {
      vpc,
      clusterName: 'shan-ecs-cluster',
    });

    const taskRole = new iam.Role(this, 'EcsTaskRole', {
      assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
      description: 'Role for ECS tasks to access MSK',
    });
  }
}

```

```
taskRole.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonMSKFullAccess'));
```

Grok 3 **beta** ▾

```
const executionRole = new iam.Role(this, 'EcsExecutionRole', {
  assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
  description: 'Role for ECS task execution',
});
executionRole.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('service-role/AmazonMSKFullAccess'));
```

```
const producerTaskDef = new ecs.FargateTaskDefinition(this, 'ProducerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
  taskRole,
  executionRole,
});
producerTaskDef.addContainer('ProducerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.producerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'producer',
    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});
```

```
const consumerTaskDef = new ecs.FargateTaskDefinition(this, 'ConsumerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
  taskRole,
  executionRole,
});
consumerTaskDef.addContainer('ConsumerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.consumerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'consumer',
    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});
```

```
const ecsSecurityGroup = new ec2.SecurityGroup(this, 'EcsSecurityGroup', {
  vpc,
  securityGroupName: 'shan-ecs-sg',
  description: 'Security group for ECS Fargate services',
  allowAllOutbound: true,
});
```

```
const mskPorts = [9098, 2181];
mskPorts.forEach((port) => {
  mskSecurityGroup.addIngressRule(
```

X **Grok 3 beta** ▾

```

  ecsSecurityGroup,
  ec2.Port.tcp(port),
  `Allow port ${port} from ECS services`
);

});

new ecs.FargateService(this, 'ProducerService', {
  cluster,
  taskDefinition: producerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});

new ecs.FargateService(this, 'ConsumerService', {
  cluster,
  taskDefinition: consumerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});
}
}
}

```

Step 4: Bastion Host (lib/bastion-stack.ts)

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as iam from 'aws-cdk-lib/aws-iam';
import { MskStack } from './msk-stack';

export class BastionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, mskStack: MskStack, props?: cdk.StackProps) {
    super(scope, id, props);

    const vpc = mskStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    const bastionSecurityGroup = new ec2.SecurityGroup(this, 'BastionSecurityGroup', {
      vpc,
      securityGroupName: 'shan-bastion-sg',
      description: 'Security group for bastion host',
      allowAllOutbound: true,
    });
  }
}

```

```

const mskPorts = [9098, 2181];
mskPorts.forEach((port) => {
  mskSecurityGroup.addIngressRule(
    bastionSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from bastion host`
  );
});

const bastionHost = new ec2.BastionHostLinux(this, 'BastionHost', {
  vpc,
  subnetSelection: { subnetType: ec2.SubnetType.PUBLIC },
  instanceName: 'shan-bastion',
  securityGroup: bastionSecurityGroup,
});
bastionHost.instance.role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonVPCBastionHostPowerUserPolicy'));

new cdk.CfnOutput(this, 'BastionHostPublicIp', { value: bastionHost.instancePublicIp });
}
}

```

Step 5: Entry Point (bin/msk-ecs-cdk.ts)

typescript

```

#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MskStack } from '../lib/msk-stack';
import { EcrStack } from '../lib/ecr-stack';
import { EcsStack } from '../lib/ecs-stack';
import { BastionStack } from '../lib/bastion-stack';

const app = new cdk.App();

const mskStack = new MskStack(app, 'MskStack', { env: { region: 'us-east-1' } });
const ecrStack = new EcrStack(app, 'EcrStack', { env: { region: 'us-east-1' } });
const ecsStack = new EcsStack(app, 'EcsStack', mskStack, ecrStack, { env: { region: 'us-east-1' } });
const bastionStack = new BastionStack(app, 'BastionStack', mskStack, { env: { region: 'us-east-1' } });

ecsStack.addDependency(mskStack);
ecsStack.addDependency(ecrStack);
bastionStack.addDependency(mskStack);

```

2. Necessary Policies for Accessing MSK, CloudWatch, ECS, and Bastion

Grok 3 beta ▾



MSK Cluster Access

- **Policy:** [AmazonMSKFullAccess](#)
 - **Attached To:**
 - [EcsTaskRole](#) (ECS tasks for producer/consumer)
 - [BastionHost](#) role
 - **Purpose:** Allows ECS Fargate tasks and the bastion host to interact with the MSK cluster (e.g., produce/consume messages, create topics).
 - **Permissions:** Grants full access to MSK resources, including [kafka:*](#) actions.

CloudWatch Logs Access

- **Policy:** [service-role/AmazonECSTaskExecutionRolePolicy](#)
 - **Attached To:** [EcsExecutionRole](#)
 - **Purpose:** Allows the ECS Fargate execution role to write logs to CloudWatch.
 - **Permissions:** Includes [logs:CreateLogStream](#) , [logs:PutLogEvents](#) , etc.
- **Additional Note:** The KMS keys ([shan-cloudwatch-kms-key](#) , [shan-ecs-logs-kms-key](#)) automatically grant decryption permissions to roles that use them for logging.

ECS Fargate Access

- **Policy:** [service-role/AmazonECSTaskExecutionRolePolicy](#)
 - **Attached To:** [EcsExecutionRole](#)
 - **Purpose:** Enables ECS to pull container images from ECR and manage task execution.
 - **Permissions:** Includes [ecr:GetAuthorizationToken](#) , [ecr:BatchGetImage](#) , etc.

Bastion Host Access

- **Policy:** [AmazonMSKFullAccess](#)
 - **Attached To:** Bastion host's instance role
 - **Purpose:** Allows the bastion host to manage MSK topics.
- **Additional Policy (Optional):** [AmazonSSMManagedInstanceCore](#)
 - **Purpose:** Enables SSM Session Manager access to the bastion host for SSH-free management.

3. Separate CDK Code for Different Stacks

- Already provided above in `msk-stack.ts`, `ecr-stack.ts`, `ecs-stack.ts`, and `bastion-stack.ts`.

X Grok 3 beta ▾    

4. Summary Descriptions for Team Members

Step 1: MSK Cluster Creation

- **What:** Deploy an Amazon MSK cluster (`shan-msk-cluster`) to manage Kafka messaging with IAM authentication.
- **How:**
 1. Use `lib/msk-stack.ts` to define the MSK cluster with 2 brokers (`kafka.t2.small`), 10GB storage, and IAM authentication (port 9098).
 2. Configure a VPC (`shan-msk-vpc`) with private subnets for MSK and public subnets for ECS/bastion.
 3. Set up a security group (`shan-msk-sg`) allowing IAM traffic (port 9098) and ZooKeeper (port 2181).
 4. Encrypt data at rest with a KMS key (`shan-msk-kms-key`) and logs with another KMS key (`shan-cloudwatch-kms-key`).
 5. Deploy via `cdk deploy MskStack`.

Step 2: ECR Repositories Creation

- **What:** Create two ECR repositories (`shan-producer` and `shan-consumer`) to store Docker images for the ECS services.
- **How:**
 1. Use `lib/ecr-stack.ts` to define the repositories.
 2. Deploy via `cdk deploy EcrStack`.
 3. Build and push Spring Boot app images to these repositories after deployment (details in Step 5).

Step 3: ECS Fargate Cluster Creation

- **What:** Set up an ECS Fargate cluster (`shan-ecs-cluster`) with two services: one to produce messages to MSK and one to consume them.
- **How:**
 1. Use `lib/ecs-stack.ts` to define the cluster, task definitions, and services.
 2. Reference the MSK VPC and security group to allow traffic to MSK.
 3. Create IAM roles (`EcsTaskRole` for MSK access, `EcsExecutionRole` for execution) and a KMS key (`shan-ecs-logs-kms-key`) for logs.

4. Deploy via `cdk deploy EcsStack` after MSK and ECR stacks.
5. Update task definitions post-deployment to include MSK bootstrap servers.



Step 4: Bastion Host Creation

- **What:** Deploy a bastion host (`shan-bastion`) to create MSK topics from within the VPC.

- **How:**

1. Use `lib/bastion-stack.ts` to define an EC2 instance in the public subnet with MSK access.
2. Attach `AmazonMSKFullAccess` to the instance role.
3. Deploy via `cdk deploy BastionStack` after MSK stack.
4. SSH into the bastion, install Kafka tools, and create the `order` topic.

Step 5: Running the Application

- **What:** Build, deploy, and run the producer and consumer Spring Boot apps in ECS Fargate.

- **How:**

1. Build the apps (`shan-producer` and `shan-consumer`) with Maven: `mvn clean package`.
2. Create Docker images: `docker build -t <name>:latest ..`
3. Push images to ECR using URIs from `EcrStack` outputs.
4. Deploy all stacks: `cdk deploy --all`.
5. Use the bastion host to create the `order` topic.
6. Update ECS task definitions with `BOOTSTRAP_SERVERS` via the AWS Console.
7. Verify logs in CloudWatch.

Deployment Steps (Team Instructions)

1. Initialize Project:

```
bash
```

```
mkdir msk-ecs-cdk
cd msk-ecs-cdk
cdk init app --language typescript
npm install @aws-cdk/aws-msk @aws-cdk/aws-ec2 @aws-cdk/aws-ecs @aws-cdk/aws-ecr @aws-cdk/aws-ecr
```



2. Add CDK Files:

Copy the above `lib/` and `bin/` files.

3. Update `tsconfig.json`:

Set `"outDir": "./dist"`, `"declaration": true`, etc., as previously shown.

4. Build and Deploy:

bash

Grok 3 **beta** ▾



```
npm run build
cdk deploy --all
```

5. Post-Deployment:

- Push Docker images to ECR.
- Create the `order` topic via the bastion host.
- Set `BOOTSTRAP_SERVERS` in ECS task definitions.

This setup provides a clear separation of concerns, detailed IAM policies, and actionable instructions for your team. Let me know if you need further refinements!

provide me MS excel sheet with separate sheets for each stack and, each sheet with every steps we create on that stack , so thta we will track and ad the corresponsing arn used

Since I can't directly provide an Excel file here, I'll outline the structure and content for an MS Excel workbook with separate sheets for each stack (`MskStack`, `EcrStack`, `EcsStack`, `BastionStack`). Each sheet will list the steps to create the resources in that stack, including columns to track the corresponding ARNs (Amazon Resource Names) or other identifiers generated during deployment. You can copy this content into Excel manually or use a script to generate the `.xlsx` file.

Below, I'll define the sheets with tables in a markdown-friendly format that you can easily transcribe into Excel. Each table includes:

- **Step:** The sequential step in creating the resource.
- **Description:** What is being done.
- **Resource Name:** The logical or physical name of the resource in CDK.
- **ARN/Identifier:** A placeholder column for you to fill in post-deployment with the ARN or other identifier output by CDK or AWS Console.

After transcribing these into Excel, you can deploy the stacks (`cdk deploy --all`) and populate the ARN/Identifier column with the actual values from the CDK outputs or AWS Console.

Excel Workbook Structure

Grok 3 **beta** ▾



- Workbook Name: **MSK-ECS-CDK-Tracking.xlsx**

- Sheets:

1. **MskStack**
2. **EcrStack**
3. **EcsStack**
4. **BastionStack**

Sheet 1: MskStack

Step	Description	Resource Name	ARN/Identifier
1	Create VPC with public and private subnets	ShanMskVpc	arn:aws:ec2:us-east-1:ACCOUNT_ID:vpc/vpc-xxx
2	Create security group for MSK	ShanMskSecurityGroup	sg-xxx (Security Group ID)
3	Add ingress rules to MSK SG (ports 9098, 2181)	N/A	N/A
4	Create KMS key for MSK data encryption	ShanMskKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
5	Create KMS key for CloudWatch logs encryption	ShanCloudwatchKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
6	Create CloudWatch log group for MSK logs	ShanMskLogGroup	arn:aws:logs:us-east-1:ACCOUNT_ID:log-group:shan-msk-logs:*
7	Create MSK cluster with IAM authentication	ShanMskCluster	arn:aws:kafka:us-east-1:ACCOUNT_ID:cluster/shan-msk-cluster/xxx

	Export MSK cluster ARN as output	ClusterArn <small>and DSI beta</small> ▾	(Same as above)    
---	----------------------------------	--	---

Notes:

- Replace `ACCOUNT_ID` with your AWS account ID.
 - After deployment, use `cdk deploy MskStack` and check outputs (`ClusterArn`) or AWS Console (VPC, Security Groups, KMS, Logs, MSK) to fill in ARNs/IDs.
-

Sheet 2: EcrStack

Step	Description	Resource Name	ARN/Identifier
1	Create ECR repository for producer app	ProducerRepo (shan-producer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/shan-producer
2	Create ECR repository for consumer app	ConsumerRepo (shan-consumer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/shan-consumer
3	Export producer repository URI as output	ProducerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/shan-producer
4	Export consumer repository URI as output	ConsumerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/shan-consumer

Notes:

- ARNs and URIs are available post-deployment via `cdk deploy EcrStack` outputs or ECR console.
-

Sheet 3: EcsStack

Step	Description	Resource Name	ARN/Identifier
------	-------------	---------------	----------------

			Grok 3 beta ▾	    
1	Create KMS key for ECS logs encryption	EcsLogsKmsKey		arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
2	Create ECS cluster	ShanEcsCluster		arn:aws:ecs:us-east-1:ACCOUNT_ID:cluster/shan-ecs-cluster
3	Create IAM role for ECS task execution	EcsExecutionRole		arn:aws:iam::ACCOUNT_ID:role/EcsExecutionRole
4	Create IAM role for ECS tasks (MSK access)	EcsTaskRole		arn:aws:iam::ACCOUNT_ID:role/EcsTaskRole
5	Create producer task definition	ProducerTaskDef		arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/ProducerTaskDef:1
6	Create consumer task definition	ConsumerTaskDef		arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/ConsumerTaskDef:1
7	Create security group for ECS services	ShanEcsSecurityGroup		sg-xxx (Security Group ID)
8	Add ingress rules to MSK SG from ECS SG	N/A		N/A

9	Create producer Fargate service	ProducerService	Grok 3 beta aws:ecs:us-east-1:ACCOUNT_ID:service/shan-ecs-cluster/ProducerService
10	Create consumer Fargate service	ConsumerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/shan-ecs-cluster/ConsumerService

Notes:

- Task definition ARNs include a revision number (e.g., `:1`), which increments with updates.
- Use `cdk deploy EcsStack` and AWS Console (ECS, IAM, KMS) to retrieve ARNs.

Sheet 4: BastionStack

Step	Description	Resource Name	ARN/Identifier
1	Create security group for bastion host	ShanBastionSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG from bastion SG	N/A	N/A
3	Create bastion host EC2 instance	ShanBastion	i-xxx (Instance ID)
4	Attach MSK access policy to bastion role	(Auto-generated role)	arn:aws:iam::ACCOUNT_ID:role/xxx
5	Export bastion public IP as output	BastionHostPublicIp (Output)	<public-ip>

Notes:

- The bastion host's IAM role is auto-generated by `BastionHostLinux`.
- Use `cdk deploy BastionStack` and outputs (`BastionHostPublicIp`) or EC2 console to fill in details.

Grok 3 beta ▾



How to Create the Excel Sheet

1. Open Excel:

- Create a new workbook named `MSK-ECS-CDK-Tracking.xlsx`.

2. Add Sheets:

- Create four sheets named `MskStack`, `EcrStack`, `EcsStack`, and `BastionStack`.

3. Copy Tables:

- For each sheet, create a table with columns: `Step`, `Description`, `Resource Name`, `ARN/Identifier`.
- Paste the content from the markdown tables above into each sheet.

4. Format:

- Adjust column widths for readability.
- Add borders or colors to distinguish steps.

5. Post-Deployment:

- Deploy the stacks:

bash



```
cd msk-ecs-cdk
npm run build
cdk deploy --all
```

- Capture ARNs/IDs from:

- CDK outputs (e.g., `ClusterArn`, `ProducerRepoUri`).
- AWS Console (MSK, ECR, ECS, EC2, IAM, KMS).

- Fill in the `ARN/Identifier` column for each resource.

Example Excel Transcription (MskStack Sheet)

Step	Description	Resource Name Grok 3 beta	ARN/Identifier
1	Create VPC with public and private subnets	ShanMskVpc	arn:aws:ec2:us-east-1:123456789012:vpc/vpc-Oabcdef1234567890
2	Create security group for MSK	ShanMskSecurityGroup	sg-0123456789abcdef0
3	Add ingress rules to MSK SG (ports 9098, 2181)	N/A	N/A
4	Create KMS key for MSK data encryption	ShanMskKmsKey	arn:aws:kms:us-east-1:123456789012:key/1234abcd-5678-efgh-ijkl-9012mnopqrst
...

- Replace **123456789012** with your AWS account ID and fill in actual resource IDs post-deployment.

Tracking Benefits

- Visibility:** Each team member can see the resources created and their ARNs in one place.
- Accountability:** Assign team members to update specific ARNs after deployment.
- Reference:** Use ARNs for troubleshooting, configuring applications, or integrating with other systems.

Let me know if you need a script to generate this Excel file programmatically (e.g., using **xlsx** in Node.js) or further assistance!



instead of using Shan as prefix use Rdp

Below, I've updated the Excel workbook structure by replacing the **Shan** prefix with **Rdp** for all resource names across the MSK, ECR, ECS, and Bastion stacks. The content remains functionally identical, but

the naming convention now reflects `Rdp` as requested. Each sheet outlines the steps for creating resources in the respective stack, with columns `Grok` `beta` `ARNs/Identifiers` post-deployment.  

You can transcribe this into an Excel workbook named `MSK-ECS-CDK-Tracking.xlsx` with four sheets: `MskStack`, `EcrStack`, `EcsStack`, and `BastionStack`.

Excel Workbook Structure

- **Workbook Name:** `MSK-ECS-CDK-Tracking.xlsx`
 - **Sheets:**
 1. `MskStack`
 2. `EcrStack`
 3. `EcsStack`
 4. `BastionStack`
-

Sheet 1: `MskStack`

Step	Description	Resource Name	ARN/Identifier
1	Create VPC with public and private subnets	<code>RdpMskVpc</code>	<code>arn:aws:ec2:us-east-1:ACCOUNT_ID:vpc/vpc-xxx</code>
2	Create security group for MSK	<code>RdpMskSecurityGroup</code>	<code>sg-xxx</code> (Security Group ID)
3	Add ingress rules to MSK SG (ports 9098, 2181)	N/A	N/A
4	Create KMS key for MSK data encryption	<code>RdpMskKmsKey</code>	<code>arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx</code>
5	Create KMS key for CloudWatch logs encryption	<code>RdpCloudwatchKmsKey</code>	<code>arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx</code>

6	Create CloudWatch log group for MSK logs	Grok 3 beta ▾ RdpMskLogGroup	arn:aws:logs:us-east-1:ACCOUNT_ID:log-group:rdp-msk-logs:*
7	Create MSK cluster with IAM authentication	RdpMskCluster	arn:aws:kafka:us-east-1:ACCOUNT_ID:cluster/rdp-msk-cluster/xxx
8	Export MSK cluster ARN as output	ClusterArn (Output)	(Same as above)

Notes:

- Replace `ACCOUNT_ID` with your AWS account ID.
- Post-deployment (`cdk deploy MskStack`), populate ARNs/IDs from CDK outputs (`ClusterArn`) or AWS Console (VPC, Security Groups, KMS, Logs, MSK).

Sheet 2: EcrStack

Step	Description	Resource Name	ARN/Identifier
1	Create ECR repository for producer app	RdpProducerRepo (rdp-producer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-producer
2	Create ECR repository for consumer app	RdpConsumerRepo (rdp-consumer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-consumer
3	Export producer repository URI as output	ProducerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-producer
4	Export consumer repository URI as output	ConsumerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-consumer

Notes:

- ARNs and URIs are available after `cdk deploy EcrStack` via outputs or ECR console.

Sheet 3: EcsStack

Step	Description	Resource Name	ARN/Identifier
1	Create KMS key for ECS logs encryption	RdpEcsLogsKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
2	Create ECS cluster	RdpEcsCluster	arn:aws:ecs:us-east-1:ACCOUNT_ID:cluster/rdp-ec cluster
3	Create IAM role for ECS task execution	RdpEcsExecutionRole	arn:aws:iam::ACCOUNT_ID:role/RdpEcsExecutionR
4	Create IAM role for ECS tasks (MSK access)	RdpEcsTaskRole	arn:aws:iam::ACCOUNT_ID:role/RdpEcsTaskRole
5	Create producer task definition	RdpProducerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpProducerTaskDef:1
6	Create consumer task definition	RdpConsumerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpConsumerTaskDef:1
7	Create security group for ECS services	RdpEcsSecurityGroup	sg-xxx (Security Group ID)
8	Add ingress	N/A	N/A

	rules to MSK SG from ECS SG	Grok 3 beta ▾	   
9	Create producer Fargate service	RdpProducerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ec cluster/RdpProducerService
10	Create consumer Fargate service	RdpConsumerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ec cluster/RdpConsumerService

Notes:

- Deploy with `cdk deploy EcsStack` after `MskStack` and `EcrStack`. Fill ARNs from ECS console or CloudFormation outputs.

Sheet 4: BastionStack

Step	Description	Resource Name	ARN/Identifier
1	Create security group for bastion host	RdpBastionSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG from bastion SG	N/A	N/A
3	Create bastion	RdpBastionHost	i-xxx (Instance ID) or arn:aws:ec2:us-east-1:ACCOUNT_ID:instance/i-xxx

	host EC2 instance	Grok 3 beta	Grok
4	Attach MSK access policy to bastion role	(Implicit Role)	arn:aws:iam::ACCOUNT_ID:role/RdpBastionHost (auto-generated)
5	Export bastion host public IP as output	BastionHostPublicIp (Output)	<public-ip> (e.g., 54.123.45.67)

Notes:

- Deploy with `cdk deploy BastionStack` after `MskStack`. Use the `BastionHostPublicIp` output to SSH and retrieve the instance role ARN from EC2 console if needed.

Instructions to Create the Excel File

1. Open Excel:

- Create a new workbook named `MSK-ECS-CDK-Tracking.xlsx`.

2. Add Sheets:

- Create four sheets: `MskStack`, `EcrStack`, `EcsStack`, `BastionStack`.

3. Set Up Columns:

In each sheet, add the following headers in Row 1:

- A1: Step
- B1: Description
- C1: Resource Name
- D1: ARN/Identifier

4. Populate Rows:

- Copy the content from each table above into the respective sheet, starting from Row 2.

- For example, in `MsKStack` :

- A2: 1 , B2: Create VPC with public and private subnets , C2: `RdpMsKVpc` , D2: (leave blank initially).

Grok 3 beta ▾



5. Deploy and Fill ARNs:

- Deploy the stacks:

bash

```
cd msk-ecs-cdk
npm run build
cdk deploy --all
```

- After deployment, check the CDK outputs (e.g., `ClusterArn` , `ProducerRepoUri`) or AWS Console (VPC, MSK, ECR, ECS, EC2) to populate the `ARN/Identifier` column.

6. Save and Share:

- Save the workbook and distribute it to your team for tracking.

Updated CDK Code with `Rdp` Prefix

To align with the Excel sheets, here's the updated CDK code with `Rdp` prefixes (only changes highlighted):

lib/msk-stack.ts

typescript

```
this.vpc = new ec2.Vpc(this, 'RdpMsKVpc', { vpcName: 'rdp-msk-vpc', ... });
this.mskSecurityGroup = new ec2.SecurityGroup(this, 'RdpMsKSecurityGroup', { securityGroupName: 'r
const mskKmsKey = new kms.Key(this, 'RdpMsKKmsKey', { alias: 'rdp-msk-kms-key', ... });
const cloudwatchKmsKey = new kms.Key(this, 'RdpCloudwatchKmsKey', { alias: 'rdp-cloudwatch-kms-key
const logGroup = new logs.LogGroup(this, 'RdpMsKLogGroup', { logGroupName: 'rdp-msk-logs', ... });
const mskCluster = new msk.CfnCluster(this, 'RdpMsKCluster', { clusterName: 'rdp-msk-cluster', ...
```



lib/ecr-stack.ts

typescript

```

X Grok 3 beta ▾
this.producerRepo = new ecr.Repository(this, 'RdpProducerRepo', { repositoryName: 'rdp-producer' })
this.consumerRepo = new ecr.Repository(this, 'RdpConsumerRepo', { repositoryName: 'rdp-consumer' })

```

lib/ecs-stack.ts

typescript

```

const ecsLogsKmsKey = new kms.Key(this, 'RdpEcsLogsKmsKey', { alias: 'rdp-ecs-logs-kms-key', ... })
const cluster = new ecs.Cluster(this, 'RdpEcsCluster', { clusterName: 'rdp-ecs-cluster', ... });
const taskRole = new iam.Role(this, 'RdpEcsTaskRole', { ... });
const executionRole = new iam.Role(this, 'RdpEcsExecutionRole', { ... });
const producerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpProducerTaskDef', { ... });
const consumerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpConsumerTaskDef', { ... });
const ecsSecurityGroup = new ec2.SecurityGroup(this, 'RdpEcsSecurityGroup', { securityGroupName: 'new ecs.FargateService(this, 'RdpProducerService', { ... });
new ecs.FargateService(this, 'RdpConsumerService', { ... });

```

lib/bastion-stack.ts

typescript

```

const bastionSecurityGroup = new ec2.SecurityGroup(this, 'RdpBastionSecurityGroup', { securityGroupDescription: 'Bastion host security group' })
const bastionHost = new ec2.BastionHostLinux(this, 'RdpBastionHost', { instanceName: 'rdp-bastion' })

```

bin/msk-ecs-cdk.ts (No prefix change needed, just stack names remain)

typescript

```

const mskStack = new MskStack(app, 'MskStack', { env: { region: 'us-east-1' } });
const ecrStack = new EcrStack(app, 'EcrStack', { env: { region: 'us-east-1' } });
const ecsStack = new EcsStack(app, 'EcsStack', mskStack, ecrStack, { env: { region: 'us-east-1' } });
const bastionStack = new BastionStack(app, 'BastionStack', mskStack, { env: { region: 'us-east-1' } });

```

Next Steps

1. **Transcribe to Excel:** Copy each table into a separate sheet in Excel as described.

2. **Deploy Stacks:** Use the updated CDK code and deploy with `cdk deploy --all`.

Grok 3 beta ▾



3. **Track ARNs:** Populate the `ARN/Identifier` column post-deployment using CDK outputs or AWS Console.

4. **Share with Team:** Provide the updated CDK code and embed the necessary policy statements directly within the stacks instead of creating separate IAM roles. This approach attaches inline policies to the resources (e.g., ECS task definitions, bastion host instance) that need them, reducing the number of standalone IAM roles while maintaining the required permissions.

Let me know if you need help with transcription or additional details!

Below, I've updated the CDK code for each stack (`MskStack`, `EcrStack`, `EcsStack`, `BastionStack`) to embed the necessary IAM policy statements directly within the stacks instead of creating separate IAM roles. This approach attaches inline policies to the resources (e.g., ECS task definitions, bastion host instance) that need them, reducing the number of standalone IAM roles while maintaining the required permissions.

I'll also provide the updated Excel sheets reflecting these changes, ensuring the policy statements are tracked instead of role ARNs.

Updated CDK Code with Embedded Policy Statements

`lib/msk-stack.ts`

No IAM roles are needed here, so it remains largely unchanged except for the `Rdp` prefix update.

typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';

export class MskStack extends cdk.Stack {
  public readonly vpc: ec2.Vpc;
  public readonly mskSecurityGroup: ec2.SecurityGroup;
  public readonly mskClusterArn: string;

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    this.vpc = new ec2.Vpc(this, 'RdpMskVpc', {
      vpcName: 'rdp-msk-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2,
      subnetConfiguration: [

```

```
  { cidrMask: 24, name: 'rdp-msk-subnet', subnetType: ec2.SubnetType.PRIVATE_ISOLATED },
  { cidrMask: 24, name: 'rdp-public-subnet', subnetType: ec2.SubnetType.PUBLIC }
],
});

this.mskSecurityGroup = new ec2.SecurityGroup(this, 'RdpMskSecurityGroup', {
  vpc: this.vpc,
  securityGroupName: 'rdp-msk-sg',
  description: 'Security group for rdp-msk-cluster',
  allowAllOutbound: true,
});

const mskPorts = [9098, 2181];
mskPorts.forEach((port) => {
  this.mskSecurityGroup.addIngressRule(
    this.mskSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} for MSK internal communication`
  );
});

const mskKmsKey = new kms.Key(this, 'RdpMskKmsKey', {
  alias: 'rdp-msk-kms-key',
  description: 'KMS key for MSK data encryption',
  enableKeyRotation: true,
});

const cloudwatchKmsKey = new kms.Key(this, 'RdpCloudwatchKmsKey', {
  alias: 'rdp-cloudwatch-kms-key',
  description: 'KMS key for CloudWatch Logs encryption',
  enableKeyRotation: true,
});

const logGroup = new logs.LogGroup(this, 'RdpMskLogGroup', {
  logGroupName: 'rdp-msk-logs',
  retention: logs.RetentionDays.ONE_MONTH,
  encryptionKey: cloudwatchKmsKey,
});

const mskCluster = new msk.CfnCluster(this, 'RdpMskCluster', {
  clusterName: 'rdp-msk-cluster',
  kafkaVersion: '3.7.x',
  numberOfWorkers: 2,
  workerNodeGroupInfo: {
    instanceType: 'kafka.t2.small',
    storageInfo: { ebsStorageInfo: { volumeSize: 10 } },
    clientSubnets: this.vpc.privateSubnets.map((subnet) => subnet.subnetId),
    securityGroups: [this.mskSecurityGroup.securityGroupId],
  },
  encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyArn } },
});
```

```

clientAuthentication: { sasl: { iam: { enabled: true } } },
loggingInfo: { brokerLogs: { cloudwatchLogGroup: Grok 3 beta enabled: true, logGroup: logGroup.logGroupName } });
}

this.mskClusterArn = mskCluster.attrArn;

new cdk.CfnOutput(this, 'ClusterArn', { value: this.mskClusterArn });
}
}

```

lib/ecr-stack.ts

No IAM policies are needed here, just ECR repository creation.

typescript 

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ecr from 'aws-cdk-lib/aws-ecr';

export class EcrStack extends cdk.Stack {
  public readonly producerRepo: ecr.Repository;
  public readonly consumerRepo: ecr.Repository;

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    this.producerRepo = new ecr.Repository(this, 'RdpProducerRepo', { repositoryName: 'rdp-producer' });
    this.consumerRepo = new ecr.Repository(this, 'RdpConsumerRepo', { repositoryName: 'rdp-consumer' });

    new cdk.CfnOutput(this, 'ProducerRepoUri', { value: this.producerRepo.repositoryUri });
    new cdk.CfnOutput(this, 'ConsumerRepoUri', { value: this.consumerRepo.repositoryUri });
  }
}

```

lib/ecs-stack.ts

Embed MSK access and ECS execution policies directly into the task definitions.

typescript 

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as iam from 'aws-cdk-lib/aws-iam';
import * as kms from 'aws-cdk-lib/aws-kms';

```



```
import { MskStack } from './msk-stack';
import { EcrStack } from './ecr-stack'; Grok 3 beta ▾

export class EcsStack extends cdk.Stack {
  constructor(scope: Construct, id: string, mskStack: MskStack, ecrStack: EcrStack, props?: cdk.StackProps) {
    super(scope, id, props);

    const vpc = mskStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    const ecsLogsKmsKey = new kms.Key(this, 'RdpEcsLogsKmsKey', {
      alias: 'rdp-ecs-logs-kms-key',
      description: 'KMS key for ECS Fargate logs',
      enableKeyRotation: true,
    });

    const cluster = new ecs.Cluster(this, 'RdpEcsCluster', {
      vpc,
      clusterName: 'rdp-ecs-cluster',
    });

    // MSK Access Policy Statement
    const mskPolicyStatement = new iam.PolicyStatement({
      actions: ['kafka:*'],
      resources: [mskStack.mskClusterArn],
    });

    // ECS Execution Policy Statement (for ECR and CloudWatch Logs)
    const executionPolicyStatement = new iam.PolicyStatement({
      actions: [
        'ecr:GetAuthorizationToken',
        'ecr:BatchCheckLayerAvailability',
        'ecr:GetDownloadUrlForLayer',
        'ecr:BatchGetImage',
        'logs>CreateLogStream',
        'logs:PutLogEvents',
      ],
      resources: ['*'], // Broad permissions for simplicity; refine as needed
    });

    const producerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpProducerTaskDef', {
      memoryLimitMiB: 512,
      cpu: 256,
    });
    producerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement); // Embed MSK policy
    producerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement); // Embed execution role
    producerTaskDef.addContainer('ProducerContainer', {
      image: ecs.ContainerImage.fromEcrRepository(ecrStack.producerRepo, 'latest'),
      logging: ecs.LogDrivers.awsLogs({
        streamPrefix: 'producer',
      })
    });
  }
}
```



```
logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
  encryptionKey: ecsLogsKmsKey,  Grok 3 beta ▾
}),
environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

const consumerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpConsumerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
});
consumerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement); // Embed MSK policy
consumerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement); // Embed execution role
consumerTaskDef.addContainer('ConsumerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.consumerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'consumer',
    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

const ecsSecurityGroup = new ec2.SecurityGroup(this, 'RdpEcsSecurityGroup', {
  vpc,
  securityGroupName: 'rdp-ecs-sg',
  description: 'Security group for ECS Fargate services',
  allowAllOutbound: true,
});

const mskPorts = [9098, 2181];
mskPorts.forEach((port) => {
  mskSecurityGroup.addIngressRule(
    ecsSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from ECS services`
  );
});

new ecs.FargateService(this, 'RdpProducerService', {
  cluster,
  taskDefinition: producerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});

new ecs.FargateService(this, 'RdpConsumerService', {
  cluster,
  taskDefinition: consumerTaskDef,
  desiredCount: 1,
```

```
  securityGroups: [ecsSecurityGroup],  
  vpcSubnets: { subnetType: ec2.SubnetType.beta }  
});  
}  
}
```

lib/bastion-stack.ts

Embed the MSK access policy into the bastion host's instance role.

typescript

```
import * as cdk from 'aws-cdk-lib';  
import { Construct } from 'constructs';  
import * as ec2 from 'aws-cdk-lib/aws-ec2';  
import * as iam from 'aws-cdk-lib/aws-iam';  
import { MskStack } from './msk-stack';  
  
export class BastionStack extends cdk.Stack {  
  constructor(scope: Construct, id: string, mskStack: MskStack, props?: cdk.StackProps) {  
    super(scope, id, props);  
  
    const vpc = mskStack.vpc;  
    const mskSecurityGroup = mskStack.mskSecurityGroup;  
  
    const bastionSecurityGroup = new ec2.SecurityGroup(this, 'RdpBastionSecurityGroup', {  
      vpc,  
      securityGroupName: 'rdp-bastion-sg',  
      description: 'Security group for bastion host',  
      allowAllOutbound: true,  
    });  
  
    const mskPorts = [9098, 2181];  
    mskPorts.forEach((port) => {  
      mskSecurityGroup.addIngressRule(  
        bastionSecurityGroup,  
        ec2.Port.tcp(port),  
        `Allow port ${port} from bastion host`  
      );  
    });  
  
    const bastionHost = new ec2.BastionHostLinux(this, 'RdpBastionHost', {  
      vpc,  
      subnetSelection: { subnetType: ec2.SubnetType.PUBLIC },  
      instanceName: 'rdp-bastion',  
      securityGroup: bastionSecurityGroup,  
    });
```

```
// Embed MSK access policy
const mskPolicyStatement = new iam.PolicyStatement(beta true)
  .actions(['kafka:*'])
  .resources([mskStack.mskClusterArn])
);
bastionHost.instance.role.addToPrincipalPolicy(mskPolicyStatement);

new cdk.CfnOutput(this, 'BastionHostPublicIp', { value: bastionHost.instancePublicIp });
}
}
```

bin/msk-ecs-cdk.ts (Unchanged)

typescript

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MskStack } from '../lib/msk-stack';
import { EcrStack } from '../lib/ecr-stack';
import { EcsStack } from '../lib/ecs-stack';
import { BastionStack } from '../lib/bastion-stack';

const app = new cdk.App();

const mskStack = new MskStack(app, 'MskStack', { env: { region: 'us-east-1' } });
const ecrStack = new EcrStack(app, 'EcrStack', { env: { region: 'us-east-1' } });
const ecsStack = new EcsStack(app, 'EcsStack', mskStack, ecrStack, { env: { region: 'us-east-1' } });
const bastionStack = new BastionStack(app, 'BastionStack', mskStack, { env: { region: 'us-east-1' } });

ecsStack.addDependency(mskStack);
ecsStack.addDependency(ecrStack);
bastionStack.addDependency(mskStack);
```

Updated Excel Sheets with Policy Statements

Sheet 1: MskStack

Step	Description	Resource Name	ARN/Identifier
1	Create VPC with public and private subnets	RdpMskVpc	arn:aws:ec2:us-east-1:ACCOUNT_ID:vpc/vpc-xxx

2	Create security group for MSK	RdpMskSecurityGroup	sg-xxx (Security Group ID) 
3	Add ingress rules to MSK SG (ports 9098, 2181)	N/A	N/A
4	Create KMS key for MSK data encryption	RdpMskKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
5	Create KMS key for CloudWatch logs encryption	RdpCloudwatchKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
6	Create CloudWatch log group for MSK logs	RdpMskLogGroup	arn:aws:logs:us-east-1:ACCOUNT_ID:log-group:rdp-msk-logs:*
7	Create MSK cluster with IAM authentication	RdpMskCluster	arn:aws:kafka:us-east-1:ACCOUNT_ID:cluster/rdp-msk-cluster/xxx
8	Export MSK cluster ARN as output	ClusterArn (Output)	(Same as above)

Notes: No IAM policies are embedded here as MSK itself doesn't require them; policies are attached to consumers (ECS, bastion).

Sheet 2: EcrStack

Step	Description	Resource Name	ARN/Identifier
1	Create ECR repository for producer app	RdpProducerRepo (rdp-producer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-producer
2	Create ECR repository for consumer app	RdpConsumerRepo (rdp-consumer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-consumer

3	Export producer repository URI as output	ProducerRepoUri <small>Grok UI beta</small> ✓ (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-producer
4	Export consumer repository URI as output	ConsumerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-consumer

Notes: No policies embedded; ECR access is managed by ECS execution policy.

Sheet 3: EcsStack

Step	Description	Resource Name	ARN/Identifier
1	Create KMS key for ECS logs encryption	RdpEcsLogsKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
2	Create ECS cluster	RdpEcsCluster	arn:aws:ecs:us-east-1:ACCOUNT_ID:cluster/rdp-ecs-cluster
3	Create producer task definition	RdpProducerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpProducerTaskDef:1
4	Embed MSK access policy in producer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Embed execution policy in producer execution role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:logs"], "Resource": "*" }
6	Create consumer task definition	RdpConsumerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpConsumerTaskDef:1
7	Embed MSK access policy in consumer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }

8	Embed execution policy in consumer execution role	Grok 3 beta N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:","/logs:"], "Resource": "*" }
9	Create security group for ECS services	RdpEcsSecurityGroup	sg-xxx (Security Group ID)
10	Add ingress rules to MSK SG from ECS SG	N/A	N/A
11	Create producer Fargate service	RdpProducerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpProducerService
12	Create consumer Fargate service	RdpConsumerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpConsumerService

Notes:

- Inline policies are embedded in `taskRole` and `executionRole` of each task definition.
- Replace `<mskClusterArn>` with the actual ARN from `MskStack.ClusterArn`.

Sheet 4: BastionStack

Step	Description	Resource Name	ARN/Identifier
1	Create security group for bastion host	RdpBastionSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG from bastion SG	N/A	N/A
3	Create bastion host EC2 instance	RdpBastionHost	i-xxx (Instance ID)

4	Embed MSK access policy in bastion instance role	Grok 3. beta  N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Export bastion host public IP as output	BastionHostPublicIp (Output)	<public-ip>

Notes:

- The MSK access policy is embedded in the bastion host's instance role.
 - Replace `<mskClusterArn>` with the actual ARN from `MskStack.ClusterArn`.
-

Instructions to Create the Excel File

1. Open Excel:

- Create a new workbook named `MSK-ECS-CDK-Tracking.xlsx`.

2. Add Sheets:

- Create four sheets: `MskStack`, `EcrStack`, `EcsStack`, `BastionStack`.

3. Set Up Columns:

- Headers in Row 1: `Step`, `Description`, `Resource Name`, `ARN/Identifier`.

4. Populate Rows:

- Copy each table above into its respective sheet, starting from Row 2.

5. Deploy and Track:

- Deploy with:

bash

```
cd msk-ecs-cdk
npm run build
cdk deploy --all
```

- Fill `ARN/Identifier` with outputs (e.g., `ClusterArn`, `ProducerRepoUri`) or AWS Console values (MSK, ECR, ECS, EC2).
- For inline policies, note the policy JSON as shown; ARNs are not generated for inline policies but are tied to the resource's role.



Key Changes

- **No Separate Roles:** Removed `RdpEcsTaskRole`, `RdpEcsExecutionRole`, and standalone role creation. Policies are now inline within `taskRole` and `executionRole` of ECS task definitions and the bastion host's instance role.
- **Policy Statements:** Embedded directly in the stacks where permissions are needed, aligning with AWS best practices for least privilege when possible (though `ecr:*` and `logs:*` are broad for simplicity).
- **Excel Updates:** Reflects inline policies instead of role ARNs, with policy JSON in the `ARN/Identifier` column where applicable.

This setup keeps the architecture before creating kms key or log groups check it already exists, and know if you need further adjustment use them. if kms got created and any failure on msk cluster creation , again we should not create key . use the already existing key or group created

To address your requirements—checking for existing KMS keys and CloudWatch log groups before creating them, and reusing them if they exist, while ensuring that KMS keys aren't recreated unnecessarily if MSK cluster creation fails—I'll modify the CDK code to use lookup methods (`kms.Key.fromLookup` and `logs.LogGroup.fromLogGroupName`) instead of always creating new resources. This approach leverages AWS CDK's ability to reference existing resources by their attributes (e.g., alias for KMS, name for log groups) and avoids redundant creation.

Here's how it works:

- **KMS Keys:** Use `kms.Key.fromLookup` to check for existing keys by alias (e.g., `alias/rdp-msk-kms-key`). If the alias doesn't exist, CDK will fail unless we explicitly create it, so I'll include a fallback creation with a condition to avoid re-creation on failure.
- **Log Groups:** Use `logs.LogGroup.fromLogGroupName` to reference an existing log group by name (e.g., `rdp-msk-logs`). If it doesn't exist, we'll create it conditionally.
- **Failure Handling:** If MSK cluster creation fails, the stack won't roll back KMS keys or log groups unless explicitly configured, and subsequent deployments will reuse the existing resources.

Below is the updated CDK code for `MskStack` (other stacks remain unchanged unless they also use KMS/log groups), followed by updated Excel sheets reflecting these changes.

Updated CDK Code

Grok 3 **beta** ▾

lib/msk-stack.ts

typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';

export class MskStack extends cdk.Stack {
  public readonly vpc: ec2.Vpc;
  public readonly mskSecurityGroup: ec2.SecurityGroup;
  public readonly mskClusterArn: string;

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Step 1: Create VPC
    this.vpc = new ec2.Vpc(this, 'RdpMskVpc', {
      vpcName: 'rdp-msk-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2,
      subnetConfiguration: [
        { cidrMask: 24, name: 'rdp-msk-subnet', subnetType: ec2.SubnetType.PRIVATE_ISOLATED },
        { cidrMask: 24, name: 'rdp-public-subnet', subnetType: ec2.SubnetType.PUBLIC },
      ],
    });
  }

  // Step 2: Create Security Group
  this.mskSecurityGroup = new ec2.SecurityGroup(this, 'RdpMskSecurityGroup', {
    vpc: this.vpc,
    securityGroupName: 'rdp-msk-sg',
    description: 'Security group for rdp-msk-cluster',
    allowAllOutbound: true,
  });

  const mskPorts = [9098, 2181];
  mskPorts.forEach((port) => {
    this.mskSecurityGroup.addIngressRule(
      this.mskSecurityGroup,
      ec2.Port.tcp(port),
      `Allow port ${port} for MSK internal communication`
    );
  });

  // Step 3: Lookup or Create KMS Key for MSK Data Encryption
```



```
let mskKmsKey: kms.IKey;
try {
  mskKmsKey = kms.Key.fromLookup(this, 'RdpMskKmsKeyLookup', {
    aliasName: 'alias/rdp-msk-kms-key',
  });
} catch (e) {
  // If lookup fails (key doesn't exist), create it
  mskKmsKey = new kms.Key(this, 'RdpMskKmsKey', {
    alias: 'rdp-msk-kms-key',
    description: 'KMS key for MSK data encryption',
    enableKeyRotation: true,
    removalPolicy: cdk.RemovalPolicy.RETAIN, // Retain key if stack fails
  });
}

// Step 4: Lookup or Create KMS Key for CloudWatch Logs Encryption
let cloudwatchKmsKey: kms.IKey;
try {
  cloudwatchKmsKey = kms.Key.fromLookup(this, 'RdpCloudwatchKmsKeyLookup', {
    aliasName: 'alias/rdp-cloudwatch-kms-key',
  });
} catch (e) {
  cloudwatchKmsKey = new kms.Key(this, 'RdpCloudwatchKmsKey', {
    alias: 'rdp-cloudwatch-kms-key',
    description: 'KMS key for CloudWatch Logs encryption',
    enableKeyRotation: true,
    removalPolicy: cdk.RemovalPolicy.RETAIN, // Retain key if stack fails
  });
}

// Step 5: Lookup or Create CloudWatch Log Group
let logGroup: logs.ILogGroup;
const logGroupName = 'rdp-msk-logs';
logGroup = logs.LogGroup.fromLogGroupName(this, 'RdpMskLogGroupLookup', logGroupName);
if (!logGroup.logGroupArn) { // If not found, create it
  logGroup = new logs.LogGroup(this, 'RdpMskLogGroup', {
    logGroupName,
    retention: logs.RetentionDays.ONE_MONTH,
    encryptionKey: cloudwatchKmsKey,
    removalPolicy: cdk.RemovalPolicy.RETAIN, // Retain log group if stack fails
  });
}

// Step 6: Create MSK Cluster
const mskCluster = new msk.CfnCluster(this, 'RdpMskCluster', {
  clusterName: 'rdp-msk-cluster',
  kafkaVersion: '3.7.x',
  numberOfWorkers: 2,
  workerNodeGroupInfo: {
    instanceType: 'kafka.t2.small',
  }
})
```

```

storageInfo: { ebsStorageInfo: { volumeSize: 10 } },
clientSubnets: this.vpc.privateSubnets, Grok 3. beta (subnet) => subnet.subnetId),
securityGroups: [this.mskSecurityGroup.securityGroupId],
},
encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyId } },
clientAuthentication: { sasl: { iam: { enabled: true } } },
loggingInfo: { brokerLogs: { cloudWatchLogs: { enabled: true, logGroup: logGroup.logGroupName } } };

this.mskClusterArn = mskCluster.attrArn;

new cdk.CfnOutput(this, 'ClusterArn', { value: this.mskClusterArn });
}
}

```

lib/ecs-stack.ts (Updated for KMS Key)

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as iam from 'aws-cdk-lib/aws-iam';
import * as kms from 'aws-cdk-lib/aws-kms';
import { MskStack } from './msk-stack';
import { EcrStack } from './ecr-stack';

export class EcsStack extends cdk.Stack {
  constructor(scope: Construct, id: string, mskStack: MskStack, ecrStack: EcrStack, props?: cdk.StackProps) {
    super(scope, id, props);

    const vpc = mskStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    // Lookup or Create KMS Key for ECS Logs
    let ecsLogsKmsKey: kms.IKey;
    try {
      ecsLogsKmsKey = kms.Key.fromLookup(this, 'RdpEcsLogsKmsKeyLookup', {
        aliasName: 'alias/rdp-ecs-logs-kms-key',
      });
    } catch (e) {
      ecsLogsKmsKey = new kms.Key(this, 'RdpEcsLogsKmsKey', {
        alias: 'rdp-ecs-logs-kms-key',
        description: 'KMS key for ECS Fargate logs',
        enableKeyRotation: true,
        removalPolicy: cdk.RemovalPolicy.RETAIN, // Retain key if stack fails
      });
    }
  }
}

```

```
}

Grok 3 beta ▾
const cluster = new ecs.Cluster(this, 'RdpEcsCluster', {
  vpc,
  clusterName: 'rdp-ecs-cluster',
});

const mskPolicyStatement = new iam.PolicyStatement({
  actions: ['kafka:*'],
  resources: [mskStack.mskClusterArn],
});

const executionPolicyStatement = new iam.PolicyStatement({
  actions: [
    'ecr:GetAuthorizationToken',
    'ecr:BatchCheckLayerAvailability',
    'ecr:GetDownloadUrlForLayer',
    'ecr:BatchGetImage',
    'logs>CreateLogStream',
    'logs:PutLogEvents',
  ],
  resources: ['*'],
});

const producerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpProducerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
});
producerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement);
producerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement);
producerTaskDef.addContainer('ProducerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.producerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'producer',
    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

const consumerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpConsumerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
});
consumerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement);
consumerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement);
consumerTaskDef.addContainer('ConsumerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.consumerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'consumer',
  })
});
```



```

    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,  Grok 3 beta ▾
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

const ecsSecurityGroup = new ec2.SecurityGroup(this, 'RdpEcsSecurityGroup', {
  vpc,
  securityGroupName: 'rdp-ecs-sg',
  description: 'Security group for ECS Fargate services',
  allowAllOutbound: true,
});

const mskPorts = [9098, 2181];
mskPorts.forEach((port) => {
  mskSecurityGroup.addIngressRule(
    ecsSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from ECS services`
  );
});

new ecs.FargateService(this, 'RdpProducerService', {
  cluster,
  taskDefinition: producerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});

new ecs.FargateService(this, 'RdpConsumerService', {
  cluster,
  taskDefinition: consumerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});
}
}
}

```

lib/ecr-stack.ts and lib/bastion-stack.ts

- No KMS keys or log groups are used here, so these remain unchanged from the previous version.

bin/msk-ecs-cdk.ts

- Unchanged, as it only orchestrates stack dependencies.

Key Changes Explained

Grok 3 **beta** ▾

1. KMS Key Lookup:

- `kms.Key.fromLookup` attempts to find an existing key by alias. If it fails (e.g., alias not found), the code falls back to creating a new key with `removalPolicy: RETAIN`, ensuring the key persists even if the MSK cluster creation fails.
- On subsequent deployments, the existing key is reused if the alias matches.

2. Log Group Lookup:

- `logs.LogGroup.fromLogGroupName` checks for an existing log group. If not found (`logGroupArn` is undefined), a new one is created with `removalPolicy: RETAIN` to persist across failures.
- The `fromLogGroupName` method doesn't throw an error if the group doesn't exist, so we check `logGroupArn` to determine existence.

3. Failure Handling:

- By setting `removalPolicy: RETAIN` on KMS keys and log groups, these resources won't be deleted if the stack fails (e.g., MSK cluster creation fails due to a misconfiguration). This ensures they can be reused in the next deployment attempt without re-creation.

4. Limitations:

- `fromLookup` requires the CDK CLI to have AWS credentials with `kms:DescribeKey` permissions during synthesis to check for existing keys. Ensure your local AWS profile or deployment role has these permissions.

Updated Excel Sheets

Sheet 1: MskStack

Step	Description	Resource Name	ARN/Identifier
1	Create VPC with public and private subnets	RdpMskVpc	arn:aws:ec2:us-east-1:ACCOUNT_ID:vpc/vpc-xxx
2	Create security group for MSK	RdpMskSecurityGroup	sg-xxx (Security Group ID)
3	Add ingress rules to MSK SG (ports 9098, 2181)	N/A	N/A

4	Lookup or create KMS key for MSK data encryption	Grok 3 beta ▾ RdpMskKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
5	Lookup or create KMS key for CloudWatch logs	RdpCloudwatchKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
6	Lookup or create CloudWatch log group for MSK	RdpMskLogGroup	arn:aws:logs:us-east-1:ACCOUNT_ID:log-group:rdp-msk-logs:*
7	Create MSK cluster with IAM authentication	RdpMskCluster	arn:aws:kafka:us-east-1:ACCOUNT_ID:cluster/rdp-msk-cluster/xxx
8	Export MSK cluster ARN as output	ClusterArn (Output)	(Same as above)

Notes:

- Steps 4-6 now reflect lookup-or-create logic. ARNs remain the same whether looked up or newly created.

Sheet 2: EcrStack

- Unchanged, as no KMS keys or log groups are involved.

Step	Description	Resource Name	ARN/Identifier
1	Create ECR repository for producer app	RdpProducerRepo (rdp-producer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-producer
2	Create ECR repository for consumer app	RdpConsumerRepo (rdp-consumer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-consumer
3	Export producer repository URI as output	ProducerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-producer

4	Export consumer repository URI as output	Consumer <small>beta</small> <small>beta</small> (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-consumer
---	--	---	---

Sheet 3: EcsStack

Step	Description	Resource Name	ARN/Identifier
1	Lookup or create KMS key for ECS logs encryption	RdpEcsLogsKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
2	Create ECS cluster	RdpEcsCluster	arn:aws:ecs:us-east-1:ACCOUNT_ID:cluster/rdp-ecs-cluster
3	Create producer task definition	RdpProducerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpProducerTaskDef:1
4	Embed MSK access policy in producer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Embed execution policy in producer execution role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:logs"], "Resource": "*" }
6	Create consumer task definition	RdpConsumerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpConsumerTaskDef:1
7	Embed MSK access policy in consumer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
8	Embed execution policy in consumer execution role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:logs"], "Resource": "*" }

9	Create security group for ECS services	RdpEcsSecurityGroup	Grok 3 beta sg-xxx (Security Group ID)    
10	Add ingress rules to MSK SG from ECS SG	N/A	N/A
11	Create producer Fargate service	RdpProducerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpProducerService
12	Create consumer Fargate service	RdpConsumerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpConsumerService

Notes:

- Step 1 updated to lookup or create the KMS key for ECS logs.

Sheet 4: BastionStack

- Unchanged, as no KMS keys or log groups are involved.

Step	Description	Resource Name	ARN/Identifier
1	Create security group for bastion host	RdpBastionSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG from bastion SG	N/A	N/A
3	Create bastion host EC2 instance	RdpBastionHost	i-xxx (Instance ID)
4	Embed MSK access policy in bastion instance role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Export bastion host	BastionHostPublicIp	<public-ip>



Deployment Instructions

- 1. Update CDK Code:** Replace the relevant files with the code above.
- 2. Ensure Permissions:** Your CDK deployment role/user needs `kms:DescribeKey` and `logs:DescribeLogGroups` permissions to perform lookups.
- 3. Deploy:**

bash

```
cd msk-ecs-cdk
npm run build
cdk deploy --all
```



- 4. Track ARNs:**

- If KMS keys or log groups exist, their ARNs will be reused.
- If created, note the new ARNs from outputs or AWS Console.
- Populate the Excel sheet accordingly.

Notes

- Fallback Creation:** The `try-catch` blocks are a workaround since `fromLookup` doesn't natively support conditional creation in CDK. If you prefer stricter lookup-only behavior, remove the `catch` block and handle missing resources manually.
- Retention:** `removalPolicy: RETAIN` ensures KMS keys and log groups persist, avoiding recreation after failures.
- Excel Updates:** Steps now explicitly mention "lookup or create" to reflect the conditional logic.



This approach meets your requirements while maintaining idempotency and reusability. Let me know if you need further refinements!

moveou tjhe vpc stack separtely , so that same vpc stack vcan be
Grok 3 beta used of other resources to be created in the sam e vpc

To modularize the VPC into a separate stack (`VpcStack`) that can be reused across other stacks (`MsKStack` , `EcsStack` , `BastionStack`), I'll extract the VPC creation from `MsKStack` and update the other stacks to reference it. This ensures that all resources (MSK cluster, ECS Fargate services, and bastion host) are deployed within the same VPC, promoting reusability and consistency. I'll also update the Excel sheets to reflect this new stack structure.

Below is the revised CDK code with a separate `VpcStack` , followed by the updated Excel workbook structure.

Updated CDK Code with Separate `VpcStack`

Project Structure

```
msk-ecs-cdk/
├── bin/
│   └── msk-ecs-cdk.ts
├── lib/
│   ├── vpc-stack.ts
│   ├── msk-stack.ts
│   ├── ecr-stack.ts
│   ├── ecs-stack.ts
│   └── bastion-stack.ts
├── package.json
├── tsconfig.json
└── ... (other files)
```

lib/vpc-stack.ts (New Stack)

typescript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';

export class VpcStack extends cdk.Stack {
    public readonly vpc: ec2.Vpc;

    constructor(scope: Construct, id: string, props?: cdk.StackProps) {
        super(scope, id, props);
```

```

X this.vpc = new ec2.Vpc(this, 'RdpVpc', Grok 3 beta ▾
  vpcName: 'rdp-vpc',
  cidr: '10.0.0.0/16',
  maxAzs: 2,
  subnetConfiguration: [
    { cidrMask: 24, name: 'rdp-msk-subnet', subnetType: ec2.SubnetType.PRIVATE_ISOLATED },
    { cidrMask: 24, name: 'rdp-public-subnet', subnetType: ec2.SubnetType.PUBLIC },
  ],
);
new cdk.CfnOutput(this, 'VpcId', { value: this.vpc.vpcId });
}
}

```

lib/msk-stack.ts (Updated)

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';
import { VpcStack } from './vpc-stack';

export class MskStack extends cdk.Stack {
  public readonly mskSecurityGroup: ec2.SecurityGroup;
  public readonly mskClusterArn: string;

  constructor(scope: Construct, id: string, vpcStack: VpcStack, props?: cdk.StackProps) {
    super(scope, id, props);

    const vpc = vpcStack.vpc;

    this.mskSecurityGroup = new ec2.SecurityGroup(this, 'RdpMskSecurityGroup', {
      vpc,
      securityGroupName: 'rdp-msk-sg',
      description: 'Security group for rdp-msk-cluster',
      allowAllOutbound: true,
    });

    const mskPorts = [9098, 2181];
    mskPorts.forEach((port) => {
      this.mskSecurityGroup.addIngressRule(
        this.mskSecurityGroup,
        ec2.Port.tcp(port),
        `Allow port ${port} for MSK internal communication`
      );
    });
  }
}

```

```
  );
  });

  let mskKmsKey: kms.IKey;
  try {
    mskKmsKey = kms.Key.fromLookup(this, 'RdpMskKmsKeyLookup', {
      aliasName: 'alias/rdp-msk-kms-key',
    });
  } catch (e) {
    mskKmsKey = new kms.Key(this, 'RdpMskKmsKey', {
      alias: 'rdp-msk-kms-key',
      description: 'KMS key for MSK data encryption',
      enableKeyRotation: true,
      removalPolicy: cdk.RemovalPolicy.RETAIN,
    });
  }

  let cloudwatchKmsKey: kms.IKey;
  try {
    cloudwatchKmsKey = kms.Key.fromLookup(this, 'RdpCloudwatchKmsKeyLookup', {
      aliasName: 'alias/rdp-cloudwatch-kms-key',
    });
  } catch (e) {
    cloudwatchKmsKey = new kms.Key(this, 'RdpCloudwatchKmsKey', {
      alias: 'rdp-cloudwatch-kms-key',
      description: 'KMS key for CloudWatch Logs encryption',
      enableKeyRotation: true,
      removalPolicy: cdk.RemovalPolicy.RETAIN,
    });
  }

  let logGroup: logs.ILogGroup;
  const logGroupName = 'rdp-msk-logs';
  logGroup = logs.LogGroup.fromLogGroupName(this, 'RdpMskLogGroupLookup', logGroupName);
  if (!logGroup.logGroupArn) {
    logGroup = new logs.LogGroup(this, 'RdpMskLogGroup', {
      logGroupName,
      retention: logs.RetentionDays.ONE_MONTH,
      encryptionKey: cloudwatchKmsKey,
      removalPolicy: cdk.RemovalPolicy.RETAIN,
    });
  }

  const mskCluster = new msk.CfnCluster(this, 'RdpMskCluster', {
    clusterName: 'rdp-msk-cluster',
    kafkaVersion: '3.7.x',
    numberOfWorkers: 2,
    brokerNodeGroupInfo: {
      instanceType: 'kafka.t2.small',
      storageInfo: { ebsStorageInfo: { volumeSize: 10 } },
    }
  });

```

```

clientSubnets: vpc.privateSubnets.map((subnet) => subnet.subnetId),
  securityGroups: [this.mskSecurityGroup], Grok 3. beta typeGroupId],
},
encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyId } },
clientAuthentication: { sasl: { iam: { enabled: true } } },
loggingInfo: { brokerLogs: { cloudWatchLogs: { enabled: true, logGroup: logGroup.logGroupName } } };

this.mskClusterArn = mskCluster.attrArn;

new cdk.CfnOutput(this, 'ClusterArn', { value: this.mskClusterArn });
}
}

```

lib/ecs-stack.ts (Updated)

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as iam from 'aws-cdk-lib/aws-iam';
import * as kms from 'aws-cdk-lib/aws-kms';
import { VpcStack } from './vpc-stack';
import { MskStack } from './msk-stack';
import { EcrStack } from './ecr-stack';

export class EcsStack extends cdk.Stack {
  constructor(scope: Construct, id: string, vpcStack: VpcStack, mskStack: MskStack, ecrStack: EcrStack, props: cdk.StackProps) {
    super(scope, id, props);

    const vpc = vpcStack.vpc;
    const mskSecurityGroup = mskStack.mskSecurityGroup;

    let ecsLogsKmsKey: kms.IKey;
    try {
      ecsLogsKmsKey = kms.Key.fromLookup(this, 'RdpEcsLogsKmsKeyLookup', {
        aliasName: 'alias/rdp-ecs-logs-kms-key',
      });
    } catch (e) {
      ecsLogsKmsKey = new kms.Key(this, 'RdpEcsLogsKmsKey', {
        alias: 'rdp-ecs-logs-kms-key',
        description: 'KMS key for ECS Fargate logs',
        enableKeyRotation: true,
        removalPolicy: cdk.RemovalPolicy.RETAIN,
      });
    }
  }
}

```



```
const cluster = new ecs.Cluster(this, 'RdpE beta ter', {
  vpc,
  clusterName: 'rdp-ecs-cluster',
});

const mskPolicyStatement = new iam.PolicyStatement({
  actions: ['kafka:*'],
  resources: [mskStack.mskClusterArn],
});

const executionPolicyStatement = new iam.PolicyStatement({
  actions: [
    'ecr:GetAuthorizationToken',
    'ecr:BatchCheckLayerAvailability',
    'ecr:GetDownloadUrlForLayer',
    'ecr:BatchGetImage',
    'logs>CreateLogStream',
    'logs:PutLogEvents',
  ],
  resources: ['*'],
});

const producerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpProducerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
});
producerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement);
producerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement);
producerTaskDef.addContainer('ProducerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.producerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'producer',
    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
    encryptionKey: ecsLogsKmsKey,
  }),
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

const consumerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpConsumerTaskDef', {
  memoryLimitMiB: 512,
  cpu: 256,
});
consumerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement);
consumerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement);
consumerTaskDef.addContainer('ConsumerContainer', {
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.consumerRepo, 'latest'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'consumer',
    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
  })
});
```



```

    encryptionKey: ecsLogsKmsKey,
  },
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

const ecsSecurityGroup = new ec2.SecurityGroup(this, 'RdpEcsSecurityGroup', {
  vpc,
  securityGroupName: 'rdp-ecs-sg',
  description: 'Security group for ECS Fargate services',
  allowAllOutbound: true,
});

const mskPorts = [9098, 2181];
mskPorts.forEach((port) => {
  mskSecurityGroup.addIngressRule(
    ecsSecurityGroup,
    ec2.Port.tcp(port),
    `Allow port ${port} from ECS services`
  );
});

new ecs.FargateService(this, 'RdpProducerService', {
  cluster,
  taskDefinition: producerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});

new ecs.FargateService(this, 'RdpConsumerService', {
  cluster,
  taskDefinition: consumerTaskDef,
  desiredCount: 1,
  securityGroups: [ecsSecurityGroup],
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },
});
}
}

```

lib/bastion-stack.ts (Updated)

typescript



```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as iam from 'aws-cdk-lib/aws-iam';
import { VpcStack } from './vpc-stack';

```

```

import { MskStack } from './msk-stack';
× Grok 3 beta ▾
export class BastionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, vpcStack: VpcStack, mskStack: MskStack, props?: cdk.St
    super(scope, id, props);

  const vpc = vpcStack.vpc;
  const mskSecurityGroup = mskStack.mskSecurityGroup;

  const bastionSecurityGroup = new ec2.SecurityGroup(this, 'RdpBastionSecurityGroup', {
    vpc,
    securityGroupName: 'rdp-bastion-sg',
    description: 'Security group for bastion host',
    allowAllOutbound: true,
  });

  const mskPorts = [9098, 2181];
  mskPorts.forEach((port) => {
    mskSecurityGroup.addIngressRule(
      bastionSecurityGroup,
      ec2.Port.tcp(port),
      `Allow port ${port} from bastion host`
    );
  });
}

const bastionHost = new ec2.BastionHostLinux(this, 'RdpBastionHost', {
  vpc,
  subnetSelection: { subnetType: ec2.SubnetType.PUBLIC },
  instanceName: 'rdp-bastion',
  securityGroup: bastionSecurityGroup,
});
}

const mskPolicyStatement = new iam.PolicyStatement({
  actions: ['kafka:*'],
  resources: [mskStack.mskClusterArn],
});
bastionHost.instance.role.addToPrincipalPolicy(mskPolicyStatement);

new cdk.CfnOutput(this, 'BastionHostPublicIp', { value: bastionHost.instancePublicIp });
}
}

```

lib/ecr-stack.ts (Unchanged)

- No VPC dependency, so it remains the same.

bin/msk-ecs-cdk.ts (Updated)

typescript

```

#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { VpcStack } from '../lib/vpc-stack';
import { MskStack } from '../lib/msk-stack';
import { EcrStack } from '../lib/ecr-stack';
import { EcsStack } from '../lib/ecs-stack';
import { BastionStack } from '../lib/bastion-stack';

const app = new cdk.App();

const vpcStack = new VpcStack(app, 'VpcStack', { env: { region: 'us-east-1' } });
const mskStack = new MskStack(app, 'MskStack', vpcStack, { env: { region: 'us-east-1' } });
const ecrStack = new EcrStack(app, 'EcrStack', { env: { region: 'us-east-1' } });
const ecsStack = new EcsStack(app, 'EcsStack', vpcStack, mskStack, ecrStack, { env: { region: 'us-east-1' } });
const bastionStack = new BastionStack(app, 'BastionStack', vpcStack, mskStack, { env: { region: 'us-east-1' } });

mskStack.addDependency(vpcStack);
ecsStack.addDependency(vpcStack);
ecsStack.addDependency(mskStack);
ecsStack.addDependency(ecrStack);
bastionStack.addDependency(vpcStack);
bastionStack.addDependency(mskStack);

```



Updated Excel Workbook Structure

Workbook Name: MSK-ECS-CDK-Tracking.xlsx

- Sheets:

1. VpcStack (New)
2. MskStack
3. EcrStack
4. EcsStack
5. BastionStack

Sheet 1: VpcStack

Step	Description	Resource Name	ARN/Identifier

 1	Create VPC with public and private subnets	GRok beta 	arn:aws:ec2:us-east-1:ACCOUNT_ID:vpc/vpc-xxx    
2	Export VPC ID as output	VpcId (Output)	vpc-xxx

Notes:

- Deploy first with `cdk deploy VpcStack` to get the VPC ID.

Sheet 2: MskStack

Step	Description	Resource Name	ARN/Identifier
1	Create security group for MSK	RdpMskSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG (ports 9098, 2181)	N/A	N/A
3	Lookup or create KMS key for MSK data encryption	RdpMskKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
4	Lookup or create KMS key for CloudWatch logs	RdpCloudwatchKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
5	Lookup or create CloudWatch log group for MSK	RdpMskLogGroup	arn:aws:logs:us-east-1:ACCOUNT_ID:log-group:rdp-msk-logs:*
6	Create MSK cluster with IAM authentication	RdpMskCluster	arn:aws:kafka:us-east-1:ACCOUNT_ID:cluster/rdp-msk-cluster/xxx
7	Export MSK cluster ARN as output	ClusterArn (Output)	(Same as above)

Notes:

- VPC creation moved to `VpcStack` ; steps start with security group.

Sheet 3: EcrStack

Step	Description	Resource Name Grok 3 beta	ARN/Identifier
1	Create ECR repository for producer app	RdpProducerRepo (rdp-producer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-producer
2	Create ECR repository for consumer app	RdpConsumerRepo (rdp-consumer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-consumer
3	Export producer repository URI as output	ProducerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-producer
4	Export consumer repository URI as output	ConsumerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-consumer

Notes:

- No VPC dependency, unchanged.

Sheet 4: EcsStack

Step	Description	Resource Name	ARN/Identifier
1	Lookup or create KMS key for ECS logs encryption	RdpEcsLogsKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
2	Create ECS cluster	RdpEcsCluster	arn:aws:ecs:us-east-1:ACCOUNT_ID:cluster/rdp-ecs-cluster
3	Create producer task definition	RdpProducerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpProducerTaskDef:1
4	Embed MSK access policy in producer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Embed execution policy in producer execution role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:logs"], "Resource": "*" }

6	Create consumer task definition	RdpConsumerTaskDef beta	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpConsumerTaskDef:1
7	Embed MSK access policy in consumer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
8	Embed execution policy in consumer execution role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:logs"], "Resource": "*" }
9	Create security group for ECS services	RdpEcsSecurityGroup	sg-xxx (Security Group ID)
10	Add ingress rules to MSK SG from ECS SG	N/A	N/A
11	Create producer Fargate service	RdpProducerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpProducerService
12	Create consumer Fargate service	RdpConsumerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpConsumerService

Notes:

- Uses `VpcStack.vpc` instead of creating its own.

Sheet 5: BastionStack

Step	Description	Resource Name	ARN/Identifier
1	Create security group for bastion host	RdpBastionSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG from bastion SG	N/A	N/A

3	Create bastion host EC2 instance	RdpBastionHost <small>beta</small> ▾	i-xxx (Instance ID)   
4	Embed MSK access policy in bastion instance role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Export bastion host public IP as output	BastionHostPublicIp (Output)	<public-ip>

Notes:

- Uses `VpcStack.vpc` instead of creating its own.
-

Deployment Instructions

1. Add New Stack:

- Create `lib/vpc-stack.ts` with the code above.
- Update other stacks to import and use `VpcStack` .

2. Deploy:

bash 

```
cd msk-ecs-cdk
npm run build
cdk deploy --all
```

- Deploy `VpcStack` first due to dependencies.

3. Track ARNs:

- Populate the Excel sheets with ARNs/IDs from CDK outputs (e.g., `VpcId` , `ClusterArn`) or AWS Console after deployment.
-

Key Changes

- **VpcStack:** A new standalone stack containing only the VPC (`RdpVpc`), exported for reuse.

Dependency Updates: `MskStack`, `EcsStack`, and `BastionStack` now take `vpcStack` as a constructor parameter and use `vpcStack`.



- Excel: Added `VpcStack` sheet; removed VPC creation from `MskStack` sheet and adjusted step numbering.
dont pass entire vpc object, but pass what is needed only

This modularizes the VPC, allowing other resources (beyond MSK, ECS, and bastion) to use `RdpVpc` by ~~referencing the entire VpcStack object and its dependencies~~. Instead, ~~the dependent components~~ (e.g., the `IVpc` interface or specific properties like `vpcId` or subnets), I'll modify the CDK code to pass just the `ec2.IVpc` interface from `VpcStack` to the dependent stacks (`MskStack`, `EcsStack`, `BastionStack`). This adheres to the principle of least privilege and reduces coupling by exposing only what's required—namely, the VPC object itself for subnet and security group configurations.

Below is the updated CDK code with this change, followed by the revised Excel sheets reflecting the adjusted dependency structure.

Updated CDK Code

`lib/vpc-stack.ts`

typescript

X **Grok 3 beta** ▾    

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';

export class VpcStack extends cdk.Stack {
  public readonly vpc: ec2.IVpc; // Changed to IVpc for minimal exposure

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    this.vpc = new ec2.Vpc(this, 'RdpVpc', {
      vpcName: 'rdp-vpc',
      cidr: '10.0.0.0/16',
      maxAzs: 2,
      subnetConfiguration: [
        { cidrMask: 24, name: 'rdp-msk-subnet', subnetType: ec2.SubnetType.PRIVATE_ISOLATED },
        { cidrMask: 24, name: 'rdp-public-subnet', subnetType: ec2.SubnetType.PUBLIC },
      ],
    });
  }

  new cdk.CfnOutput(this, 'VpcId', { value: this.vpc.vpcId });
}
}
```

lib/msk-stack.ts

typescript 

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as msk from 'aws-cdk-lib/aws-msk';
import * as kms from 'aws-cdk-lib/aws-kms';
import * as logs from 'aws-cdk-lib/aws-logs';

export class MskStack extends cdk.Stack {
  public readonly mskSecurityGroup: ec2.SecurityGroup;
  public readonly mskClusterArn: string;

  constructor(scope: Construct, id: string, vpc: ec2.IVpc, props?: cdk.StackProps) {
    super(scope, id, props);

    this.mskSecurityGroup = new ec2.SecurityGroup(this, 'RdpMskSecurityGroup', {
      vpc,
      securityGroupName: 'rdp-msk-sg',
      description: 'Security group for rdp-msk-cluster',
      allowAllOutbound: true,
    });
  }
}
```

```
});  
const mskPorts = [9098, 2181];  
mskPorts.forEach((port) => {  
  this.mskSecurityGroup.addIngressRule(  
    this.mskSecurityGroup,  
    ec2.Port.tcp(port),  
    `Allow port ${port} for MSK internal communication`  
  );  
});  
  
let mskKmsKey: kms.IKey;  
try {  
  mskKmsKey = kms.Key.fromLookup(this, 'RdpMskKmsKeyLookup', {  
    aliasName: 'alias/rdp-msk-kms-key',  
  });  
} catch (e) {  
  mskKmsKey = new kms.Key(this, 'RdpMskKmsKey', {  
    alias: 'rdp-msk-kms-key',  
    description: 'KMS key for MSK data encryption',  
    enableKeyRotation: true,  
    removalPolicy: cdk.RemovalPolicy.RETAIN,  
  });  
}  
  
let cloudwatchKmsKey: kms.IKey;  
try {  
  cloudwatchKmsKey = kms.Key.fromLookup(this, 'RdpCloudwatchKmsKeyLookup', {  
    aliasName: 'alias/rdp-cloudwatch-kms-key',  
  });  
} catch (e) {  
  cloudwatchKmsKey = new kms.Key(this, 'RdpCloudwatchKmsKey', {  
    alias: 'rdp-cloudwatch-kms-key',  
    description: 'KMS key for CloudWatch Logs encryption',  
    enableKeyRotation: true,  
    removalPolicy: cdk.RemovalPolicy.RETAIN,  
  });  
}  
  
let logGroup: logs.ILogGroup;  
const logGroupName = 'rdp-msk-logs';  
logGroup = logs.LogGroup.fromLogGroupName(this, 'RdpMskLogGroupLookup', logGroupName);  
if (!logGroup.logGroupArn) {  
  logGroup = new logs.LogGroup(this, 'RdpMskLogGroup', {  
    logGroupName,  
    retention: logs.RetentionDays.ONE_MONTH,  
    encryptionKey: cloudwatchKmsKey,  
    removalPolicy: cdk.RemovalPolicy.RETAIN,  
  });  
}
```

Grok 3 beta ▾





```

X const mskCluster = new msk.CfnCluster(this, Grok 3 beta 'mskCluster', {
  clusterName: 'rdp-msk-cluster',
  kafkaVersion: '3.7.x',
  numberOfWorkers: 2,
  workerNodeGroupInfo: {
    instanceType: 'kafka.t2.small',
    storageInfo: { ebsStorageInfo: { volumeSize: 10 } },
    clientSubnets: vpc.privateSubnets.map((subnet) => subnet.subnetId),
    securityGroups: [this.mskSecurityGroup.securityGroupId],
  },
  encryptionInfo: { encryptionAtRest: { dataVolumeKmsKeyId: mskKmsKey.keyId } },
  clientAuthentication: { sasl: { iam: { enabled: true } } },
  loggingInfo: { brokerLogs: { cloudWatchLogs: { enabled: true, logGroup: logGroup.logGroupName } } },
};

this.mskClusterArn = mskCluster.attrArn;

new cdk.CfnOutput(this, 'ClusterArn', { value: this.mskClusterArn });
}
}

```

lib/ecs-stack.ts

typescript

```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as iam from 'aws-cdk-lib/aws-iam';
import * as kms from 'aws-cdk-lib/aws-kms';
import { MskStack } from './msk-stack';
import { EcrStack } from './ecr-stack';

export class EcsStack extends cdk.Stack {
  constructor(scope: Construct, id: string, vpc: ec2.IVpc, mskStack: MskStack, ecrStack: EcrStack, props: cdk.StackProps) {
    super(scope, id, props);

    const mskSecurityGroup = mskStack.mskSecurityGroup;

    let ecsLogsKmsKey: kms.IKey;
    try {
      ecsLogsKmsKey = kms.Key.fromLookup(this, 'RdpEcsLogsKmsKeyLookup', {
        aliasName: 'alias/rdp-ecs-logs-kms-key',
      });
    } catch (e) {
      ecsLogsKmsKey = new kms.Key(this, 'RdpEcsLogsKmsKey', {
        aliasName: 'alias/rdp-ecs-logs-kms-key',
      });
    }
  }
}

```

alias: 'rdp-ecs-logs-kms-key',
description: 'KMS key for ECS Fargate',
enableKeyRotation: true,
removalPolicy: cdk.RemovalPolicy.RETAIN,
});
}

const cluster = new ecs.Cluster(this, 'RdpEcsCluster', {
vpc,
clusterName: 'rdp-ecs-cluster',
});

const mskPolicyStatement = new iam.PolicyStatement({
actions: ['kafka:*'],
resources: [mskStack.mskClusterArn],
});

const executionPolicyStatement = new iam.PolicyStatement({
actions: [
'ecr:GetAuthorizationToken',
'ecr:BatchCheckLayerAvailability',
'ecr:GetDownloadUrlForLayer',
'ecr:BatchGetImage',
'logs>CreateLogStream',
'logs:PutLogEvents',
],
resources: ['*'],
});

const producerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpProducerTaskDef', {
memoryLimitMiB: 512,
cpu: 256,
});
producerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement);
producerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement);
producerTaskDef.addContainer('ProducerContainer', {
image: ecs.ContainerImage.fromEcrRepository(ecrStack.producerRepo, 'latest'),
logging: ecs.LogDrivers.awsLogs({
streamPrefix: 'producer',
logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,
encryptionKey: ecsLogsKmsKey,
}),
environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },
});

const consumerTaskDef = new ecs.FargateTaskDefinition(this, 'RdpConsumerTaskDef', {
memoryLimitMiB: 512,
cpu: 256,
});
consumerTaskDef.taskRole.addToPrincipalPolicy(mskPolicyStatement);

```
consumerTaskDef.executionRole.addToPrincipalPolicy(executionPolicyStatement);  
consumerTaskDef.addContainer('ConsumerContainer', {  
  image: ecs.ContainerImage.fromEcrRepository(ecrStack.consumerRepo, 'latest'),  
  logging: ecs.LogDrivers.awsLogs({  
    streamPrefix: 'consumer',  
    logRetention: cdk.aws_logs.RetentionDays.ONE_MONTH,  
    encryptionKey: ecsLogsKmsKey,  
  }),  
  environment: { MSK_CLUSTER_ARN: mskStack.mskClusterArn },  
});  
  
const ecsSecurityGroup = new ec2.SecurityGroup(this, 'RdpEcsSecurityGroup', {  
  vpc,  
  securityGroupName: 'rdp-ecs-sg',  
  description: 'Security group for ECS Fargate services',  
  allowAllOutbound: true,  
});  
  
const mskPorts = [9098, 2181];  
mskPorts.forEach((port) => {  
  mskSecurityGroup.addIngressRule(  
    ecsSecurityGroup,  
    ec2.Port.tcp(port),  
    `Allow port ${port} from ECS services`  
  );  
});  
  
new ecs.FargateService(this, 'RdpProducerService', {  
  cluster,  
  taskDefinition: producerTaskDef,  
  desiredCount: 1,  
  securityGroups: [ecsSecurityGroup],  
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },  
});  
  
new ecs.FargateService(this, 'RdpConsumerService', {  
  cluster,  
  taskDefinition: consumerTaskDef,  
  desiredCount: 1,  
  securityGroups: [ecsSecurityGroup],  
  vpcSubnets: { subnetType: ec2.SubnetType.PUBLIC },  
});  
}  
}  
}
```

lib/bastion-stack.ts

typescript



```

import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';  Grok 3  beta  ▾
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as iam from 'aws-cdk-lib/aws-iam';
import { MskStack } from './msk-stack';

export class BastionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, vpc: ec2.IVpc, mskStack: MskStack, props?: cdk.StackProps) {
    super(scope, id, props);

    const mskSecurityGroup = mskStack.mskSecurityGroup;

    const bastionSecurityGroup = new ec2.SecurityGroup(this, 'RdpBastionSecurityGroup', {
      vpc,
      securityGroupName: 'rdp-bastion-sg',
      description: 'Security group for bastion host',
      allowAllOutbound: true,
    });

    const mskPorts = [9098, 2181];
    mskPorts.forEach((port) => {
      mskSecurityGroup.addIngressRule(
        bastionSecurityGroup,
        ec2.Port.tcp(port),
        `Allow port ${port} from bastion host`
      );
    });

    const bastionHost = new ec2.BastionHostLinux(this, 'RdpBastionHost', {
      vpc,
      subnetSelection: { subnetType: ec2.SubnetType.PUBLIC },
      instanceName: 'rdp-bastion',
      securityGroup: bastionSecurityGroup,
    });

    const mskPolicyStatement = new iam.PolicyStatement({
      actions: ['kafka:*'],
      resources: [mskStack.mskClusterArn],
    });
    bastionHost.instance.role.addToPrincipalPolicy(mskPolicyStatement);

    new cdk.CfnOutput(this, 'BastionHostPublicIp', { value: bastionHost.instancePublicIp });
  }
}

```

lib/ecr-stack.ts (Unchanged)

- No VPC dependency, remains the same.

bin/msk-ecs-cdk.ts

typescript

Grok 3 beta ▾



```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { VpcStack } from '../lib/vpc-stack';
import { MskStack } from '../lib/msk-stack';
import { EcrStack } from '../lib/ecr-stack';
import { EcsStack } from '../lib/ecs-stack';
import { BastionStack } from '../lib/bastion-stack';

const app = new cdk.App();

const vpcStack = new VpcStack(app, 'VpcStack', { env: { region: 'us-east-1' } });
const mskStack = new MskStack(app, 'MskStack', vpcStack.vpc, { env: { region: 'us-east-1' } });
const ecrStack = new EcrStack(app, 'EcrStack', { env: { region: 'us-east-1' } });
const ecsStack = new EcsStack(app, 'EcsStack', vpcStack.vpc, mskStack, ecrStack, { env: { region: 'us-east-1' } });
const bastionStack = new BastionStack(app, 'BastionStack', vpcStack.vpc, mskStack, { env: { region: 'us-east-1' } });

mskStack.addDependency(vpcStack);
ecsStack.addDependency(vpcStack);
ecsStack.addDependency(mskStack);
ecsStack.addDependency(ecrStack);
bastionStack.addDependency(vpcStack);
bastionStack.addDependency(mskStack);
```



Updated Excel Workbook Structure

Workbook Name: MSK-ECS-CDK-Tracking.xlsx

- Sheets:

1. VpcStack
2. MskStack
3. EcrStack
4. EcsStack
5. BastionStack

Sheet 1: VpcStack

Step	Description	Resource Name	ARN/Identifier

1	Create VPC with public and private subnets	Grok beta      	arn:aws:ec2:us-east-1:ACCOUNT_ID:vpc/vpc-xxx
2	Export VPC ID as output	VpcId (Output)	vpc-xxx

Sheet 2: MskStack

Step	Description	Resource Name	ARN/Identifier
1	Create security group for MSK	RdpMskSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG (ports 9098, 2181)	N/A	N/A
3	Lookup or create KMS key for MSK data encryption	RdpMskKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
4	Lookup or create KMS key for CloudWatch logs	RdpCloudwatchKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
5	Lookup or create CloudWatch log group for MSK	RdpMskLogGroup	arn:aws:logs:us-east-1:ACCOUNT_ID:log-group:rdp-msk-logs:*
6	Create MSK cluster with IAM authentication	RdpMskCluster	arn:aws:kafka:us-east-1:ACCOUNT_ID:cluster/rdp-msk-cluster/xxx
7	Export MSK cluster ARN as output	ClusterArn (Output)	(Same as above)

Sheet 3: EcrStack

Step	Description	Resource Name	ARN/Identifier
1	Create ECR repository for producer app	RdpProducerRepo (rdp-producer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-producer

2	Create ECR repository for consumer app	RdpConsumer Grok Beta (rdp-consumer)	arn:aws:ecr:us-east-1:ACCOUNT_ID:repository/rdp-consumer
3	Export producer repository URI as output	ProducerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-producer
4	Export consumer repository URI as output	ConsumerRepoUri (Output)	<ACCOUNT_ID>.dkr.ecr.us-east-1.amazonaws.com/rdp-consumer

Sheet 4: EcsStack

Step	Description	Resource Name	ARN/Identifier
1	Lookup or create KMS key for ECS logs encryption	RdpEcsLogsKmsKey	arn:aws:kms:us-east-1:ACCOUNT_ID:key/key-xxx
2	Create ECS cluster	RdpEcsCluster	arn:aws:ecs:us-east-1:ACCOUNT_ID:cluster/rdp-ecs-cluster
3	Create producer task definition	RdpProducerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpProducerTaskDef:1
4	Embed MSK access policy in producer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Embed execution policy in producer execution role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:", "logs:"], "Resource": "*" }
6	Create consumer task definition	RdpConsumerTaskDef	arn:aws:ecs:us-east-1:ACCOUNT_ID:task-definition/RdpConsumerTaskDef:1
7	Embed MSK access policy in consumer task role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
8	Embed execution policy in	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": ["ecr:",

			Grok
	consumer execution role	Grok 3 beta	"/logs:"], "Resource": "*" } ▼
9	Create security group for ECS services	RdpEcsSecurityGroup	sg-xxx (Security Group ID)
10	Add ingress rules to MSK SG from ECS SG	N/A	N/A
11	Create producer Fargate service	RdpProducerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpProducerService
12	Create consumer Fargate service	RdpConsumerService	arn:aws:ecs:us-east-1:ACCOUNT_ID:service/rdp-ecs-cluster/RdpConsumerService

Sheet 5: BastionStack

Step	Description	Resource Name	ARN/Identifier
1	Create security group for bastion host	RdpBastionSecurityGroup	sg-xxx (Security Group ID)
2	Add ingress rules to MSK SG from bastion SG	N/A	N/A
3	Create bastion host EC2 instance	RdpBastionHost	i-xxx (Instance ID)
4	Embed MSK access policy in bastion instance role	N/A (Inline Policy)	Policy: { "Effect": "Allow", "Action": "kafka:*", "Resource": "<mskClusterArn>" }
5	Export bastion host public IP as output	BastionHostPublicIp (Output)	<public-ip>

Key Changes

Grok 3 beta ▾



- **VpcStack:** Now exposes only `public readonly vpc: ec2.IVpc`, limiting what dependent stacks can access to the VPC interface (e.g., `vpcId`, `publicSubnets`, `privateSubnets`).
- **Constructor Updates:** `MskStack`, `EcsStack`, and `BastionStack` now take `vpc: ec2.IVpc` as a parameter instead of the entire `VpcStack`. `EcsStack` and `BastionStack` still need `MskStack` for `mskSecurityGroup` and `mskClusterArn`.
- **Dependency Management:** In `bin/msk-ecs-cdk.ts`, dependencies are updated to pass `vpcStack.vpc` instead of the full stack object.
- **Excel Sheets:** No structural change needed, as the steps and ARNs remain the same; only the dependency mechanism is refined.

Deployment Instructions

1. Update CDK Files:

- Add `lib/vpc-stack.ts` and update the other files as shown.



2. Deploy:

bash

```
cd msk-ecs-cdk
npm run build
cdk deploy --all
```

whats the meaning
new cdk.CfnOutput(this, 'VpcId', { value: this.vpc.vpcId });

Let's break down the `cdk deploy` first due to dependencies.

2. ~~cdk deploy~~

typescript

```
new cdk.CfnOutput(this, 'VpcId', { value: this.vpc.vpcId });
```

This line is part of an AWS CDK (Cloud Development Kit) stack definition, written in TypeScript, and it creates a CloudFormation output for the stack. Here's a detailed explanation of each component:

Benefits

- **Minimal Exposure:** Dependent stacks only receive the `IVpc` interface, reducing access to unnecessary `VpcStack` properties.

1. new `cdk.CfnOutput`

- `cdk` ~~sability~~ of the `aws-cdk` library, which `stack` is imported (typically via `import * as cdk from 'aws-cdk-lib';`). It provides constructs ~~Grok~~ **beta** to define AWS infrastructure as code.  
 - `CfnOutput` ~~class~~ is the `CDK` that ~~represents~~ `CloudFormation` for the ~~stack~~ `dependent` structure.
CloudFormation are values that are exported from a stack after it's deployed. They can be used:
Let me know if you need further clarification or additional refinements!
 - To display important information (e.g., resource IDs, ARNs) in the AWS Console or CLI after deployment.
 - To share values with other stacks via cross-stack references.
 - To make key resource details easily accessible to users or automation scripts.
 - `new` : This keyword creates a new instance of the `CfnOutput` class, adding it to the stack's CloudFormation template.
-

2. `this`

- `this` : Refers to the current instance of the stack class (e.g., `VpcStack` in your case). In CDK, resources like VPCs, security groups, or outputs are scoped to a specific stack, and `this` is the context in which they're defined.
 - The first argument to `CfnOutput` is the scope (the stack it belongs to), so `this` ties the output to the `VpcStack`.
-

3. `'VpcId'`

- This is the **logical ID** or name of the output within the CloudFormation template. It's a unique identifier within the stack for this specific output.
 - In this case, `'VpcId'` is a human-readable name indicating that this output will contain the ID of the VPC created in the stack. When you deploy the stack, you'll see `VpcId` listed in the outputs section of the CloudFormation console or CLI.
-

4. `{ value: this.vpc.vpcId }`

- This is the configuration object passed to `CfnOutput`, defining what the output represents.

- `value` : A required property of `CfnOutput` that specifies the actual data to export. It can be a static string, a reference to a resource property, or `new` `Grok` `beta` `value`.
 - `this.vpc` : Refers to the VPC resource defined earlier in the stack (e.g., `this.vpc = new ec2.Vpc(...)`). It's an instance of the `ec2.Vpc` class (or implements `ec2.IVpc`).
 - `vpcId` : A property of the `ec2.Vpc` object that returns the unique identifier of the VPC (e.g., `vpc-0abcdef1234567890`). This is the AWS-assigned ID for the VPC resource created in your AWS account.
 - So, `{ value: this.vpc.vpcId }` means the output named `VpcId` will contain the VPC's ID (e.g., `vpc-0abcdef1234567890`) after the stack is deployed.
-

Full Meaning

Putting it all together:

typescript

```
new cdk.CfnOutput(this, 'VpcId', { value: this.vpc.vpcId });
```

- **What it does:** Creates a CloudFormation output in the current stack (`this`), named `VpcId` , and sets its value to the ID of the VPC (`this.vpc.vpcId`) created within the same stack.
- **Purpose:**
 - Makes the VPC ID available post-deployment for reference or use in other stacks.
 - You can see this value in the AWS CloudFormation console under the "Outputs" tab or retrieve it via the AWS CLI with `aws cloudformation describe-stacks --stack-name VpcStack --query "Stacks[0].Outputs"` .
- **Example Output:** After deploying `VpcStack` , the output might look like:

Outputs:

`VpcId: vpc-0abcdef1234567890`

Why It's Useful

- **Visibility:** After running `cdk deploy VpcStack`, you'll see the `VpcId` in the terminal or CloudFormation console, making it easy to use elsewhere.
 - **Cross-Stack Reference:** If another stack needs this VPC (e.g., to place resources in it), you can import it using `ec2.Vpc.fromLookup` or `cdk.Fn.importValue('VpcId')` by referencing the exported `VpcId`.
 - **Documentation:** It explicitly documents the VPC ID as a key output of the stack, improving maintainability.
-

Context in Your Code

In your `VpcStack`:

typescript

```
export class VpcStack extends cdk.Stack {  
  public readonly vpc: ec2.IVpc;  
  
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {  
    super(scope, id, props);  
  
    this.vpc = new ec2.Vpc(this, 'RdpVpc', { ... });  
    new cdk.CfnOutput(this, 'VpcId', { value: this.vpc.vpcId });  
  }  
}
```

- `this.vpc` is the VPC resource you've created.
 - `this.vpc.vpcId` extracts its ID.
 - The `CfnOutput` ensures this ID is exported, so `MskStack`, `EcsStack`, and `BastionStack` (or any future stacks) can use it without needing the full `VpcStack` object—just the `IVpc` interface or the exported `VpcId` if needed.
-

Let me know if you'd like further clarification or examples!

