Using IAM database authentication with workloads running on Amazon EKS

<https://aws.amazon.com/blogs/containers/using-iam-database-authentication-with-workloads-running-on-amazon-eks/>

by Praseeda Sathaye | on 27 MAY 2022 | in [Amazon Aurora](https://aws.amazon.com/blogs/containers/category/database/amazon-aurora/), [Amazon Elastic Kubernetes Service](https://aws.amazon.com/blogs/containers/category/compute/amazon-kubernetes-service/), [AWS Identity and Access Management (IAM)](https://aws.amazon.com/blogs/containers/category/security-identity-compliance/aws-identity-and-access-management-iam/), [Containers](https://aws.amazon.com/blogs/containers/category/containers/), [Technical How-to](https://aws.amazon.com/blogs/containers/category/post-types/technical-how-to/) | [Permalink](https://aws.amazon.com/blogs/containers/using-iam-database-authentication-with-workloads-running-on-amazon-eks/) | [Share](https://aws.amazon.com/blogs/containers/using-iam-database-authentication-with-workloads-running-on-amazon-eks/)

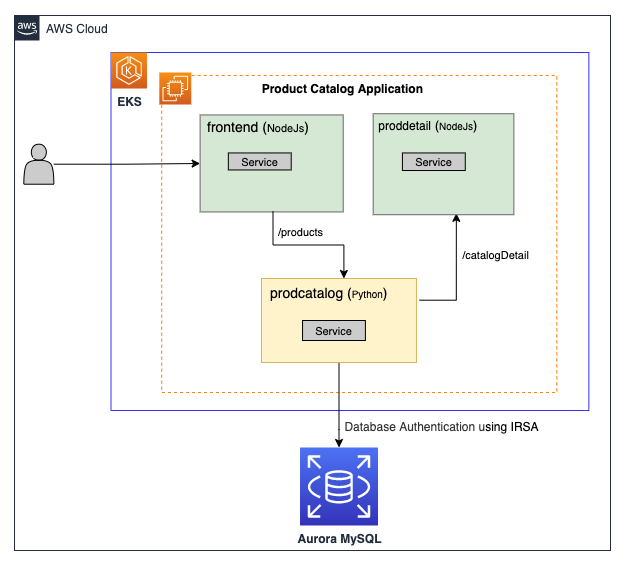
[Amazon Elastic Kubernetes Service (Amazon EKS)](https://aws.amazon.com/eks/) is a managed service that you can use to run Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane or nodes. When running containerized workloads on Amazon EKS, it is common to store the stateful parts of the application outside of the Kubernetes cluster in one or more SQL or NoSQL databases. However, a common challenge when using SQL databases with Kubernetes is storing the database credentials, as well as rotating the credentials on a regular basis and passing the sensitive parts securely into the Kubernetes cluster.

With Amazon Relational Database Service ([Amazon RDS](https://aws.amazon.com/rds/)) and [Amazon Aurora](https://aws.amazon.com/rds/aurora/), the whole process of managing user access credentials can be simplified by using [AWS Identity and Access Management (IAM) for authenticating to the database](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/UsingWithRDS.IAMDBAuth.html). IAM database authentication removes the burden and security risk of managing usernames and passwords for database authentication. Leveraging IAM database authentication with workloads running on Amazon EKS provides the following benefits:

* Network traffic to and from the database is encrypted using Secure Socket Layer (SSL) or Transport Layer Security (TLS).
* Authentication tokens have a lifespan of 15 minutes, so you don’t need to enforce password resets.
* Amazon EKS [IAM roles for service accounts](https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts.html) (IRSA) can be leveraged to provide IAM access keys and secret access keys to the application. Providing a secure way to retrieve tokens for IAM database authentication.

IAM database authentication works with [MariaDB, MySQL, and PostgreSQL running on Amazon RDS](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/UsingWithRDS.IAMDBAuth.html), and [Amazon Aurora MySQL and Amazon Aurora PostgreSQL](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.html).

In this blog post, we will demonstrate IAM database authentication with Amazon EKS by deploying a sample application and storing the state in an Amazon Aurora MySQL database. The demonstration application is the [Product Catalog Application](https://github.com/aws-containers/eks-app-mesh-polyglot-demo/tree/master/workshop), which has the following architecture:



The Product Catalog Application has three microservices:

* frontend service hosting a web UI for a product catalog.
* prodcatalog backend service that performs the following actions:
  + Talks to the Aurora MySQL database to:
    - Add a product into the database.
    - Get the product from the database.
  + Calls catalog detail backend service proddetail to get product catalog detail information.
* proddetail backend service that gets catalog detail, which includes version numbers and vendor names.

**Walkthrough**

In this blog post, we will first be creating a local database user, workshop\_user, in an Amazon Aurora MySQL database and enable this user to use IAM database authentication. We will then be creating an IAM policy called “Aurora\_IAM\_Policy” that will have permission to access the Aurora database as the user workshop\_user. Finally, we will leverage IRSA, a Kubernetes service account, and an IAM role to securely connect to the Amazon Aurora database.

**Prerequisites:**

This blog assumes that you have the following setup already done:

* An existing [EKS cluster](https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html) with an existing node group.
* Aurora MySQL cluster using [Quickstart](https://aws-quickstart.github.io/quickstart-amazon-aurora-mysql/" \l "_deployment_steps).
  + IAM DB authentication availability is enabled in this Quickstart.
  + The database endpoint is not available publicly in this Quickstart, as it is not recommended to have database endpoints accessible on the internet. When following this walkthrough, a connection within the VPC to connect to the database is required. The simplest way to achieve this is through an AWS Cloud9 instance or a bastion host when running the mysql client commands.
* For simplicity, in this blog, we have assumed that the Amazon EKS cluster, the AWS Cloud9 instance/bastion host, and Aurora MySQL are all deployed in the same VPC in the same AWS account.
* Tools required on a machine with access to the AWS and Kubernetes API server. This could be your workstation or an AWS Cloud9 instance/bastion host.
  + The [AWS CLI](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html).
  + The [eksctl](https://eksctl.io/introduction/) utility used for creating and managing Kubernetes clusters on Amazon EKS.
  + The [kubectl](https://docs.aws.amazon.com/eks/latest/userguide/install-kubectl.html) utility used for communicating with the Kubernetes cluster API server.
  + The [Helm CLI](https://helm.sh/docs/intro/install/) used for installing [Helm Charts](https://helm.sh/docs/intro/install/).
* Tools required on a machine with access to the Amazon Aurora database, such as an AWS Cloud9 instance or bastion host.
  + The [mysql client](https://dev.mysql.com/doc/refman/8.0/en/mysql.html).

**Step 1. Confirm access to an Amazon EKS cluster**

Ensure you have access to an EKS cluster via the kubectl client. For details on setting up the kubectl client for an Amazon EKS cluster, see the [Amazon EKS User Guide](https://docs.aws.amazon.com/eks/latest/userguide/create-kubeconfig.html).

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

ip-xxx-yyy-109-70.us-east-2.compute.internal Ready <none> 61m v1.XX.11-eks-f17b81

ip-xxx-yyy-153-253.us-east-2.compute.internal Ready <none> 61m v1.XX.11-eks-f17b81

ip-xxx-yyy-181-40.us-east-2.compute.internal Ready <none> 61m v1.XX.11-eks-f17b81

**Step 2. Confirm Aurora DB setup**

Ensure the Amazon Aurora MySQL Quickstart has been deployed successfully by navigating to the database within the Amazon RDS console. If you kept the Quickstart defaults, you should see an Amazon Aurora cluster with two instances (one is Writer and one is Reader).

A screenshot of a computer

Description automatically generated

Next, confirm that IAM DB authentication is enabled by selecting the Configuration tab. If IAM DB authentication is not enabled, then enable it by referring to the [Amazon RDS User Guide for Aurora](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.Enabling.html).A screenshot of a computer

Description automatically generated

**Step 3. Update the database security group**

To allow workloads on the Amazon EKS cluster to access the Aurora MySQL database, we need to add a few rules to the security group assigned to the Aurora MySQL database.

First, we need to retrieve the security group ID that workloads running on Kubernetes will use. By default in an Amazon EKS cluster, all Kubernetes pods share the same security group with their underlying EC2 instance. In a managed node group, this is the EKS cluster security group. This can be retrieved by browsing to the Amazon EKS console within the AWS Management Console. Select Configuration and then Networking. Copy this security group ID to your clipboard.

A screenshot of a computer

Description automatically generated

Next, we need to navigate to the security group associated with the Writer instance on the database cluster. Within the Amazon RDS console, navigate to the Aurora database cluster, select the Writer instance, and then select the VPC security group. This should take you to the security group console, with the relevant security group already selected.

A screenshot of a security

Description automatically generated

Now we can create inbound rules on this security group for the Kubernetes pods. Add an inbound rule with Protocol TCP, Port Range 3306, and in the Source box, paste in the security group ID retrieved from the Amazon EKS cluster. If you are planning to use an AWS Cloud9 instance or bastion host to run the mysql commands later on in this walkthrough, create a second Inbound rule with the relevant source security group ID so that traffic sourced from that instance can reach the database. Finally, select Save rules.

A screenshot of a computer

Description automatically generated

Step 4. Test the connectivity

Within the Amazon RDS console, select the Writer instance, and within the Connectivity & security tab, copy the database endpoint into your clipboard.

A screenshot of a computer

Description automatically generated

Within the AWS Cloud9 instance or bastion host, confirm that you can connect to the database with the user name msadmin and the password you set when deploying the Aurora Quickstart template. If you are unable to connect to the database, go back to Step 3 and ensure the security groups have been configured correctly for the AWS Cloud9 instance/bastion host.

# This is the database endpoint that should be in the clipboard.

$ export DB\_ENDPOINT=<rds\_database\_endpoint>

$ mysql -h $DB\_ENDPOINT -P 3306 -umsadmin -p

Enter password:

Welcome to the MariaDB monitor. Commands end with ; or \g.

Your MySQL connection id is 109

Server version: 5.7.12-log MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]>

**Step 5. Create the database,  table,  data, and user named “workshop\_user”**

Before deploying the Product Catalog application into Kubernetes, we will first seed the database by entering a product into a product table within a dev database.

MySQL [(none)]> CREATE DATABASE dev;

Query OK, 1 row affected (0.01 sec)

MySQL [(none)]> CREATE TABLE dev.product (prodId VARCHAR(120), prodName VARCHAR(120));

Query OK, 0 rows affected (0.02 sec)

MySQL [(none)]> INSERT INTO dev.product (prodId,prodName) VALUES ('999','Mountain New Bike');

Query OK, 1 row affected (0.00 sec)

MySQL [(none)]> SELECT \* FROM dev.product;

+--------+-------------------+

| prodId | prodName |

+--------+-------------------+

| 999 | Mountain New Bike |

+--------+-------------------+

1 row in set (0.00 sec)

Next, we will create a workshop\_user locally on the database instance. We will use this database user and its database privileges when IAM identities (such as Kubernetes service accounts) authenticate with IAM database authentication.

# Create user in the db (this is the user we will use for IAM authentication)

MySQL [(none)]> CREATE USER workshop\_user IDENTIFIED WITH AWSAuthenticationPlugin AS 'RDS';

Query OK, 0 rows affected (0.03 sec)

# Grant usage

MySQL [(none)]> GRANT USAGE ON \*.\* TO 'workshop\_user'@'%'REQUIRE SSL;

Query OK, 0 rows affected, 1 warning (0.01 sec)

MySQL [(none)]> GRANT ALL PRIVILEGES ON dev.\* TO 'workshop\_user'@'%'REQUIRE SSL;

Query OK, 0 rows affected, 1 warning (0.00 sec)

To verify that the workshop\_user has been configured to correctly use the AWS authentication plugin, run the following SQL commands.

# Get the user

MySQL [(none)]> select user,plugin,host from MySQL.user where user like '%workshop\_user%';

+---------------+-------------------------+------+

| user | plugin | host |

+---------------+-------------------------+------+

| workshop\_user | AWSAuthenticationPlugin | % |

+---------------+-------------------------+------+

1 row in set (0.00 sec)

#Show grants

MySQL [(none)]>show grants for workshop\_user;

+--------------------------------------------------------+

| Grants for workshop\_user@% |

+--------------------------------------------------------+

| GRANT USAGE ON \*.\* TO 'workshop\_user'@'%' |

| GRANT ALL PRIVILEGES ON `dev`.\* TO 'workshop\_user'@'%' |

+--------------------------------------------------------+

2 rows in set (0.00 sec)

**Step 6. Deploy the application onto Kubernetes**

On the machine that was used to verify Kubernetes cluster access in Step 1, clone the sample application from its GitHub repository and deploy it to the Amazon EKS cluster using Helm.

$ git clone https://github.com/aws-containers/eks-app-mesh-polyglot-demo.git

$ cd eks-app-mesh-polyglot-demo

$ helm install workshop workshop/helm-chart/

After a few minutes, the Kubernetes pods should be deployed successfully to the Amazon EKS cluster, and an Elastic Load Balancer should have been created to access the front-end UI. To verify that the pods have been deployed and to retrieve the URL for the Loadbalancer, run the following commands.

# Verify that all 3 microservices are running.

kubectl get pods -n workshop

NAME READY STATUS RESTARTS AGE

frontend-54884b8c67-m6hfg 1/1 Running 0 2m43s

prodcatalog-6b45bcfd4f-b959q 1/1 Running 0 2m43s

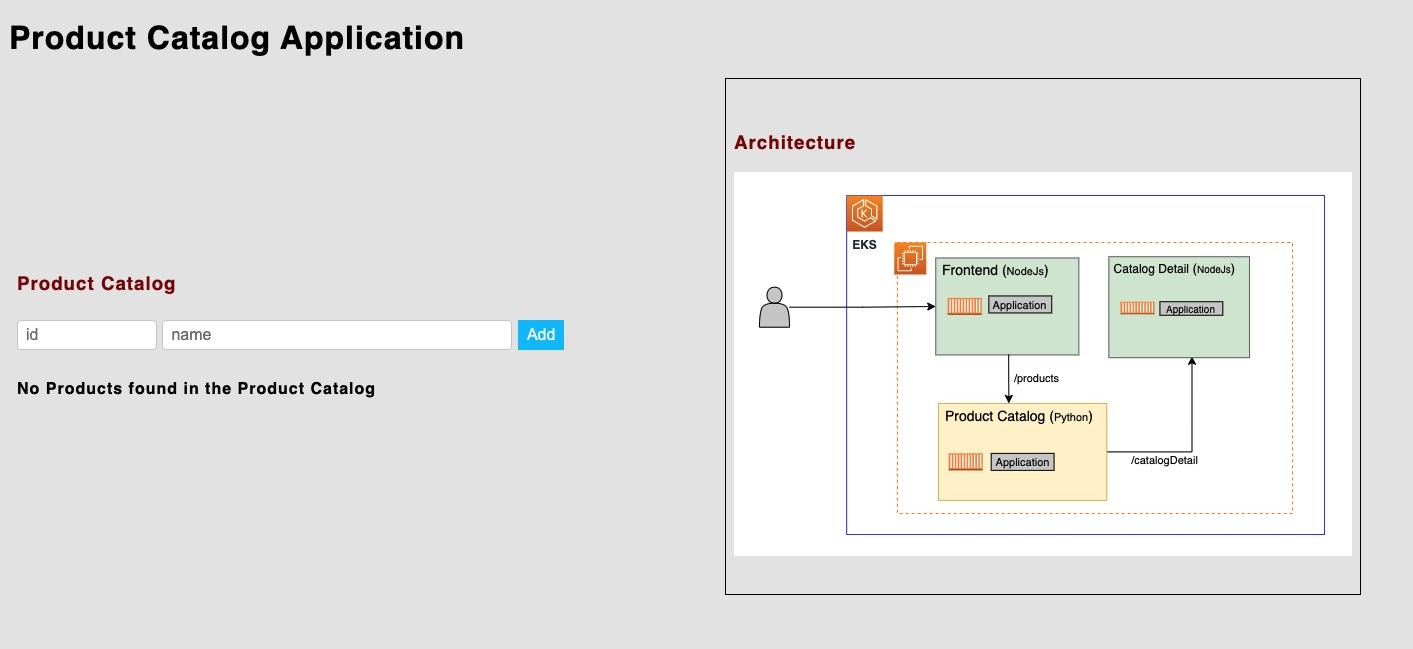
proddetail-7cdffcb79-96ptj 1/1 Running 0 2m43s

# Retrieve the Loadbalancer URL.

export LB\_NAME=$(kubectl get svc frontend -n workshop -o jsonpath="{.status.loadBalancer.ingress[\*].hostname}")

echo $LB\_NAME

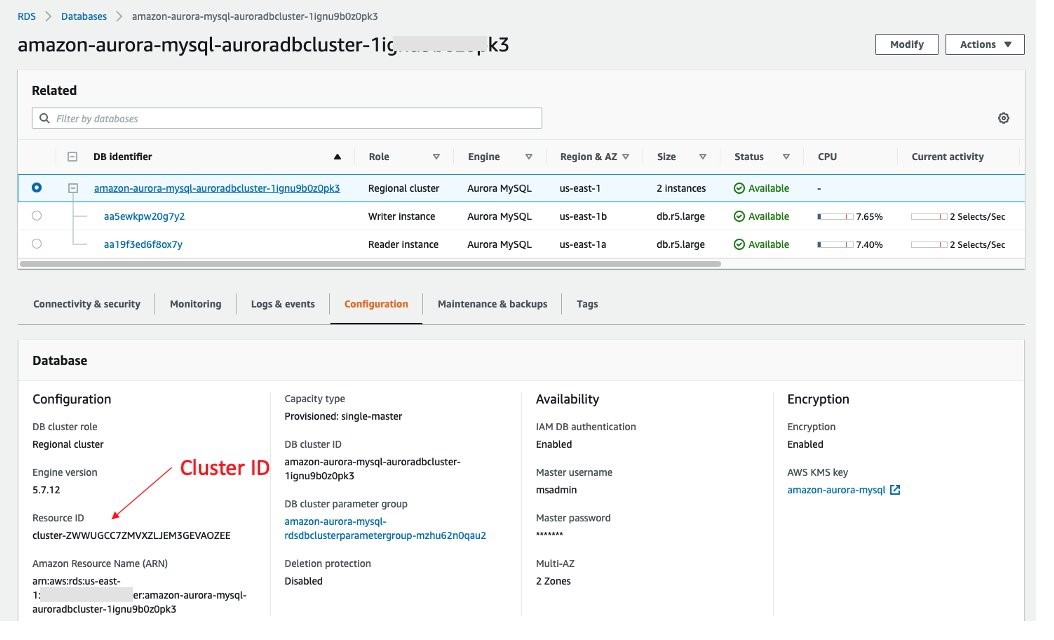
In a web browser, browse to the Load Balancer URL. You should see the following UI, and you should see that there is no data found for the product catalog. At this point, the authentication for the database has not been configured, so we are unable to see the seed data in the database, hence “No Products found in the Product Catalog.”



**Step 7. Create an IAM policy for DB authentication**

To allow a user or a Kubernetes service account to access the Aurora MySQL database, we need to create an IAM policy. Following least privileges, the IAM policy specifies the Amazon Aurora database resource ID and the local database user workshop\_user in the IAM policy document. For more details on this topic, you can also refer to the [Creating and using an IAM policy for IAM database access](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.IAMPolicy.html) document.

To create the policy document, we need to retrieve the Amazon Aurora Resource ID from the RDS console. Open the RDS console, select the Aurora MySQL cluster, and navigate to the Configuration tab. Copy the resource ID to your clipboard.



On a machine that has the AWS CLI installed, create an IAM policy that can access the database.

# Export the required Environment variables

$ export RESOURCE\_ID=<REPLACE\_WITH\_THE\_RESOURCE\_ID>

$ export AWS\_ACCOUNT=<REPLACE\_WITH\_AWS\_ACCOUNT\_ID>

$ export AWS\_REGION=<REPLACE\_WITH\_AWS\_REGION>

# Create the IAM Policy File

$ cat << EOF > iam\_policy.json

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"rds-db:connect"

],

"Resource": [

"arn:aws:rds-db:$AWS\_REGION:$AWS\_ACCOUNT:dbuser:$RESOURCE\_ID/workshop\_user"

]

}

]

}

EOF

# Create the IAM Policy

$ aws iam create-policy \

--region ${AWS\_REGION} \

--policy-name "Aurora\_IAM\_Policy" \

--policy-document file://iam\_policy.json

# Export the Policy ARN

$ export AURORA\_IAM\_POLICY\_ARN=$(aws --region ${AWS\_REGION} iam list-policies --query 'Policies[?PolicyName==`'Aurora\_IAM\_Policy'`].Arn' --output text)

**Step 8. Create an IAM role and map this to a Kubernetes service account**

In this step, we will consume the IAM policy created in Step 7 and attach it to an IAM role. We will then create a trust relationship on that IAM role to map the role to a Kubernetes service account. These steps could be done by using the AWS CLI; however, the [eksctl](https://docs.aws.amazon.com/eks/latest/userguide/eksctl.html)tool simplifies all of the steps into a single command.

# Export the EKS Cluster Name

$ export EKS\_CLUSTER=<cluster\_name>

# Create an IAM OIDC provider for your cluster

$ eksctl utils associate-iam-oidc-provider \

--region=$AWS\_REGION \

--cluster=$EKS\_CLUSTER \

--approve

# Create a service account

$ eksctl create iamserviceaccount \

--cluster $EKS\_CLUSTER \

--name aurora-irsa \

--namespace workshop \

--attach-policy-arn $AURORA\_IAM\_POLICY\_ARN \

--override-existing-serviceaccounts \

--approve

Under the hood, the previous command carries out two things:

* It creates an IAM role and attaches the specified policy to it; in our case, arn:aws:iam::61801138XXXX:policy/Aurora\_IAM\_Policy
* It creates a Kubernetes service account aurora-irsa and annotates the service account with the IAM role.

View the created service account.

$ kubectl describe sa aurora-irsa -n workshop

Name: aurora-irsa

Namespace: workshop

Labels: app.kubernetes.io/managed-by=eksctl

Annotations: eks.amazonaws.com/role-arn: arn:aws:iam::XXXX:role/eksctl-eksworkshop-eksctl-addon-iamserviceac-Role1-1W01ZHLJPJ3TQ

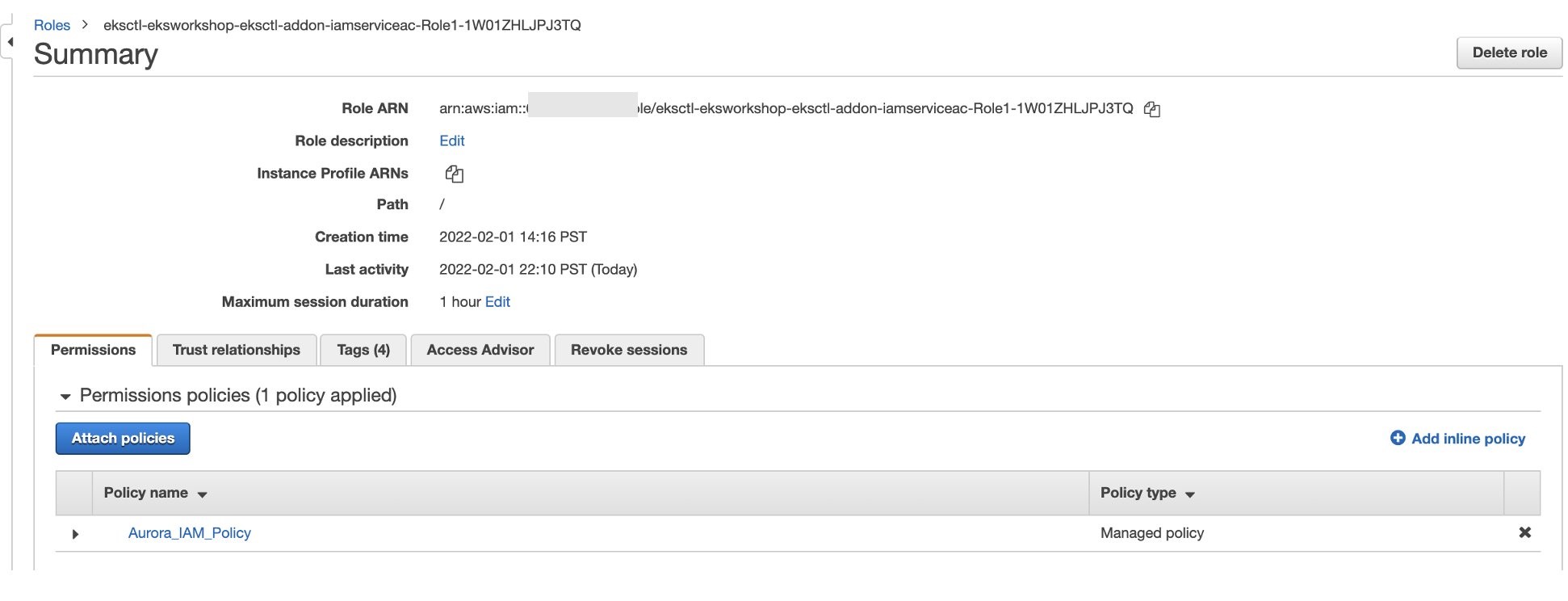
Image pull secrets: <none>

Mountable secrets: aurora-irsa-token-698bg

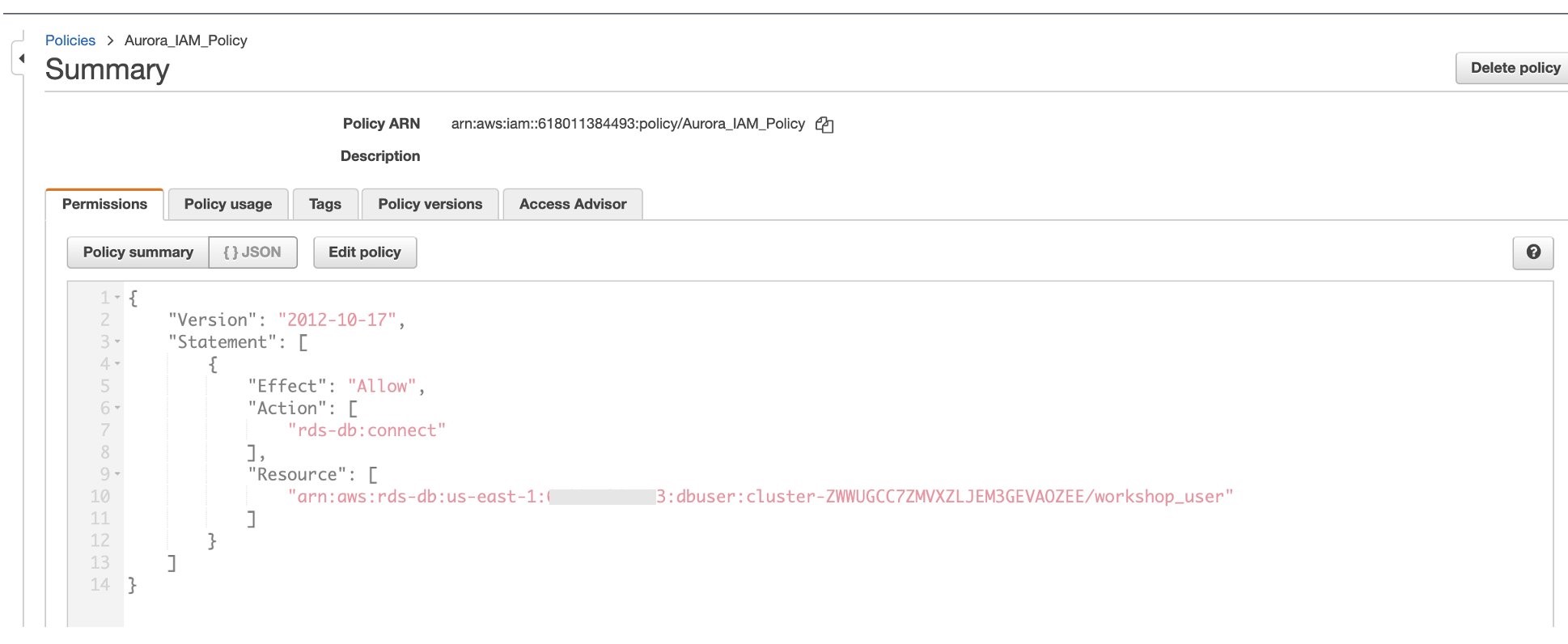
Tokens: aurora-irsa-token-698bg

Events: <none>

You can also go to the IAM console within the AWS Management Console and search for the previous role-arn. You should see that the IAM role has been created successfully.



Select the Aurora\_IAM\_Policy, and you should be able to confirm the previously created IAM DB authentication policy has been attached to this role.



**Step 9. Redeploy the Helm Chart and specify the database credentials**

Now that an IAM role has been mapped to a Kubernetes service account, the application can use the service account credentials to communicate to the Aurora MySQL database. We will pass in the nonsensitive RDS connection information to the application through environment variables in the Helm Chart.

A Helm Chart variables file is located at workshop/helm-chart/values-aurora.yaml in the cloned repository. Once that file has been opened in a text editor, we need to update the DATABASE\_SERVICE\_URL and DB\_REGION variables. These are found at lines 160 and 168.

$ vi workshop/helm-chart/values-aurora.yaml

<snip>

- name: DATABASE\_SERVICE\_URL

value: <database\_endpoint\_string>

- name: DATABASE\_USER\_NAME

value: "workshop\_user"

- name: DB\_NAME

value: "dev"

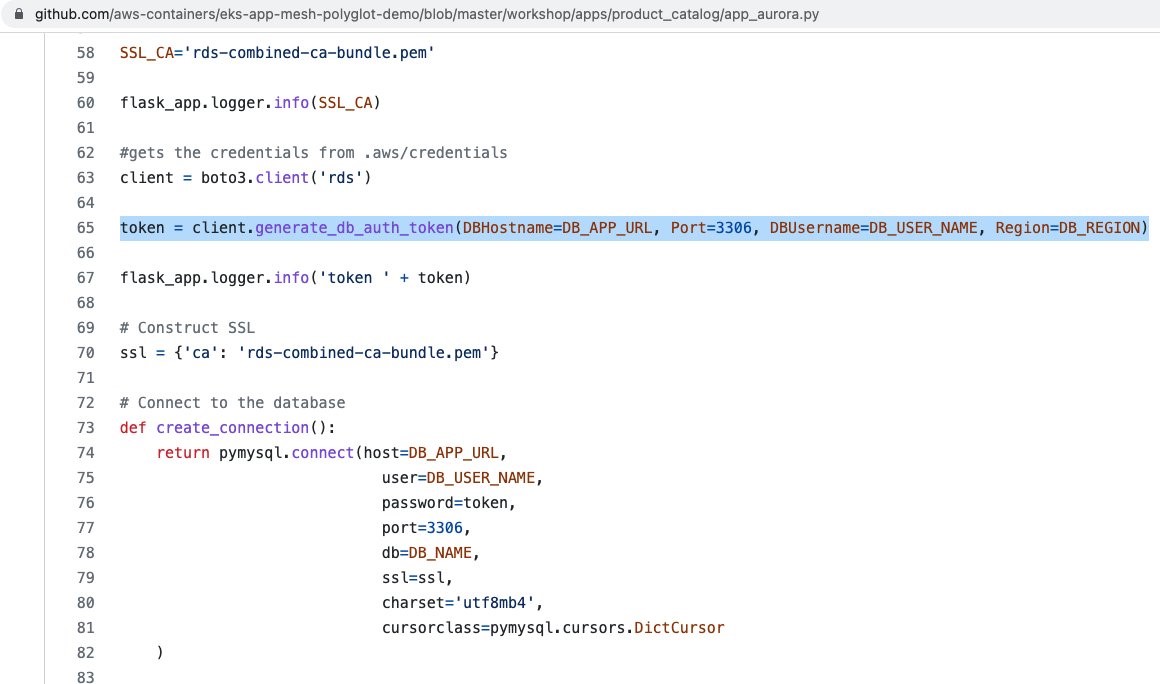
- name: DB\_PORT

value: "3306"

- name: DB\_REGION

value: <aws\_region>

Within the *[prodcatalog](https://github.com/aws-containers/eks-app-mesh-polyglot-demo/blob/master/workshop/apps/product_catalog/app_aurora.py" \l "L69)*[application code](https://github.com/aws-containers/eks-app-mesh-polyglot-demo/blob/master/workshop/apps/product_catalog/app_aurora.py" \l "L69), we use the[boto3 SDK](https://boto3.amazonaws.com/v1/documentation/api/latest/index.html) to consume these environment variables when authenticating to IAM to retrieve a token. We then pass this token into our pymysql connect command alongside the remaining database information to authenticate to the database.



So that the application can take advantage of the IAM roles for service account credentials and to pick up the new environment variables, we can redeploy the application with Helm.

# Redeploy the Application

$ helm upgrade \

-f workshop/helm-chart/values-aurora.yaml \

workshop \

workshop/helm-chart/

# Use kubectl to verify new pods are being deployed

$ kubectl get pods -n workshop

NAME READY STATUS RESTARTS AGE

frontend-54884b8c67-m6hfg 1/1 Running 0 4h35m

prodcatalog-6b45bcfd4f-b959q 1/1 Terminating 0 4h35m

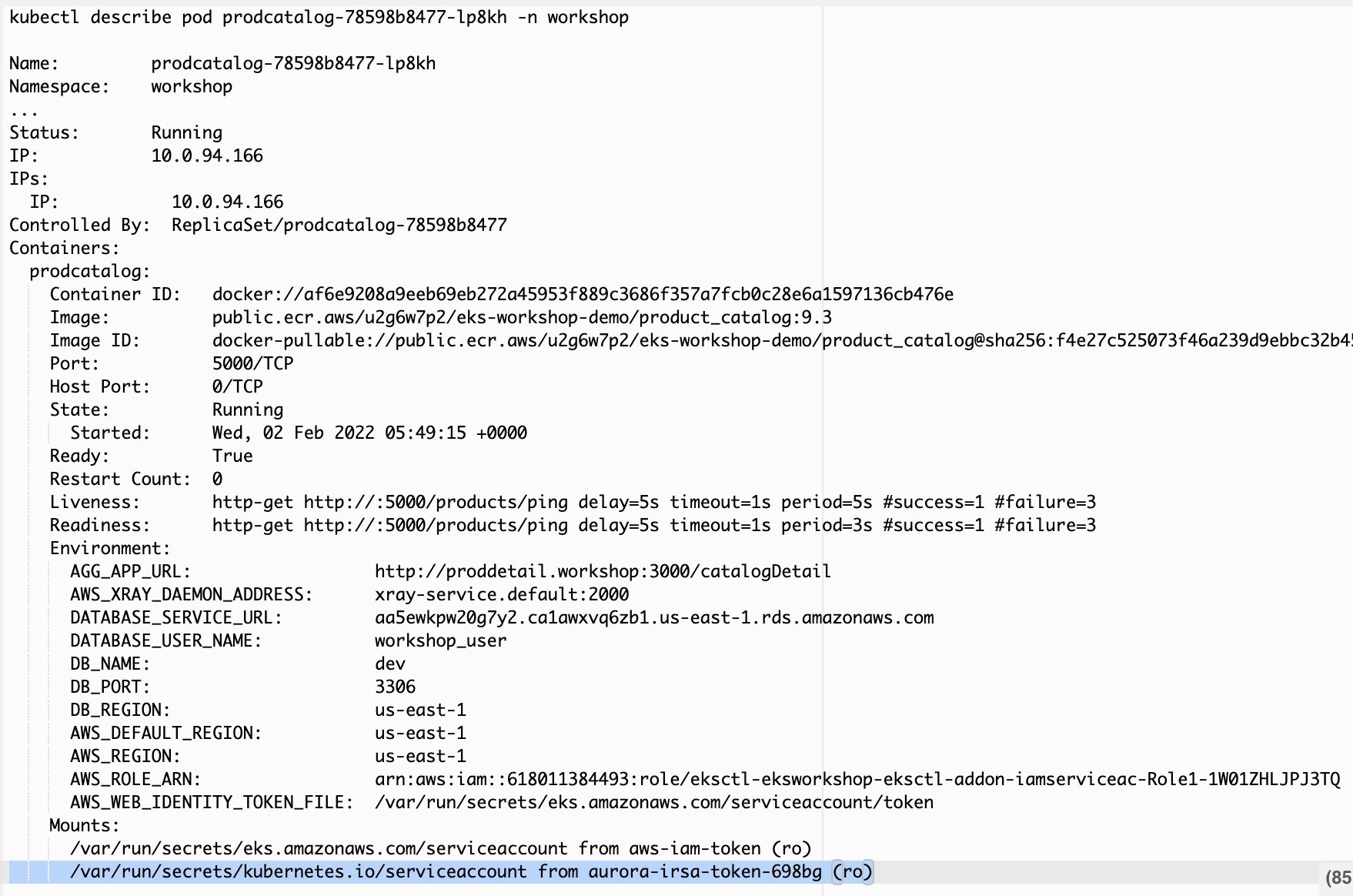
prodcatalog-6c98fd985d-t4v9q 1/1 Running 0 27s

proddetail-7cdffcb79-96ptj 1/1 Running 0 4h35m

To verify that IAM roles for service accounts are working correctly, use kubectl to describe the podcatalog pod.

$ kubectl describe pod prodcatalog-<pod\_id> -n workshop

The following image shows that the mutating admission controller we ran in EKS (via a webhook) automatically injected the environment variables AWS\_ROLE\_ARN and AWS\_WEB\_IDENTITY\_TOKEN\_FILE as well as the aws-iam-token volume.



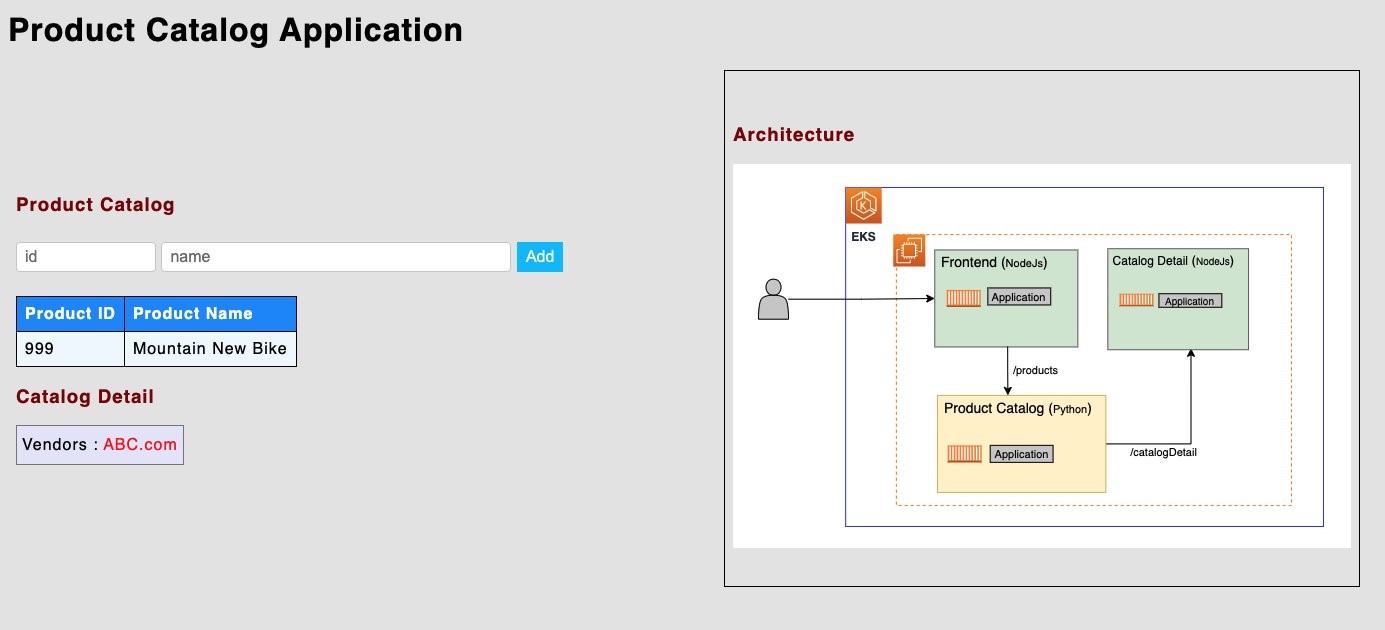
**Step 10. Confirm the application is connected to database**

Finally, we will browse to the application front end again and verify that the application can now see our seed data. If you need to retrieve the front-end URL again, use kubectl to get details about the Kubernetes service.

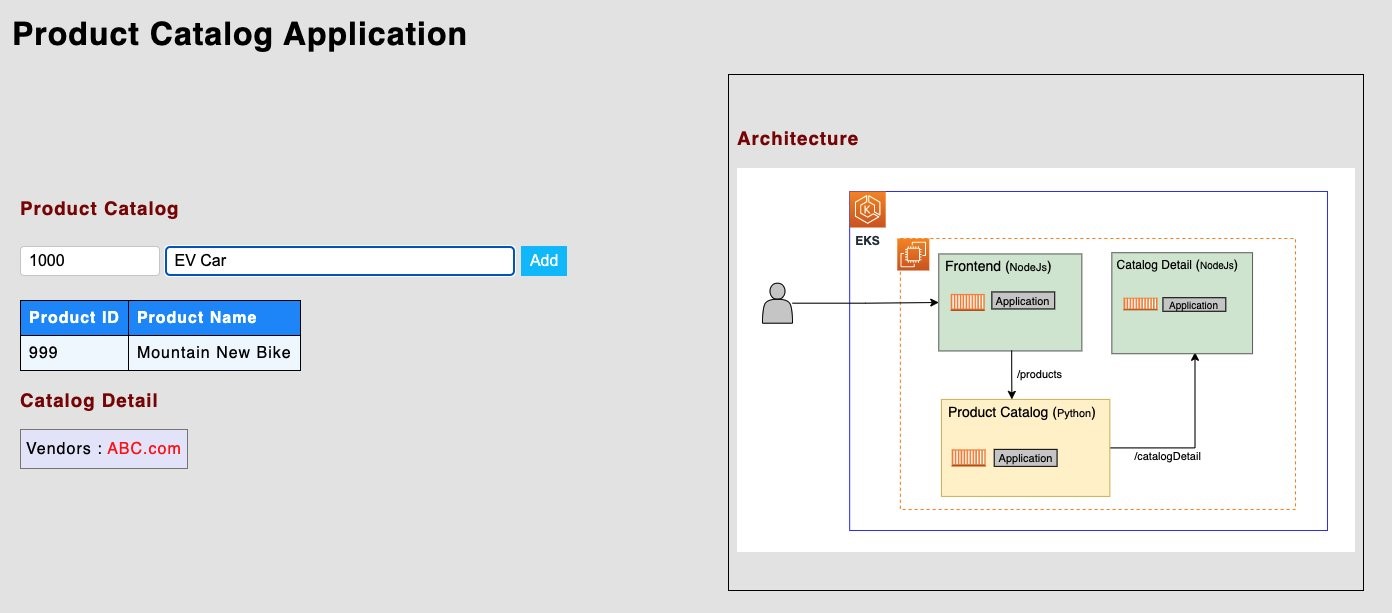
export LB\_NAME=$(kubectl get svc frontend -n workshop -o jsonpath="{.status.loadBalancer.ingress[\*].hostname}")

echo $LB\_NAME

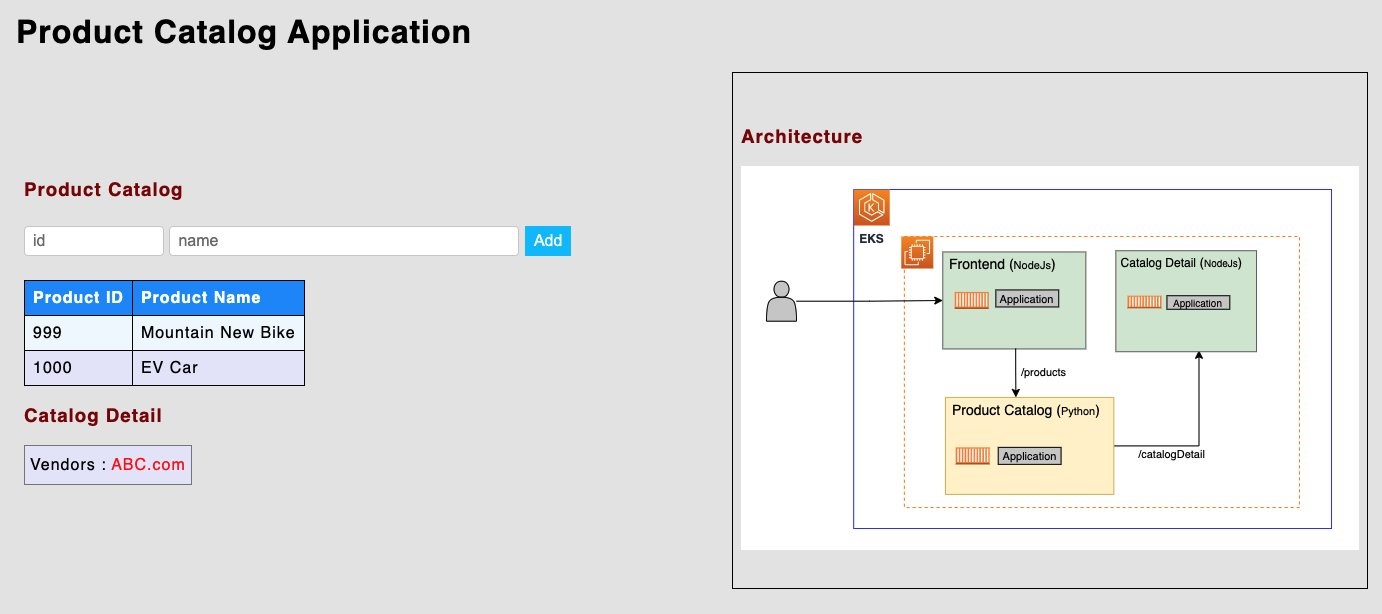
Once you go to the Load Balancer URL in a web browser, you should be able to confirm that the Product Catalog data (Product ID: 999 and Product Name: Mountain New Bike) is coming from the Aurora MySQL DB. This means our *prodcatalog* service deployed in EKS is able to talk to Aurora MySQL DB using IRSA authentication with the service account role we created for the user workshop\_user.



We can also add a new product to the catalog by entering a new Product ID and Product Name into the corresponding fields.



You can confirm that the new product has been added to the Aurora MySQL database successfully.



Finally, we can verify that the new item has been added to the database by using the MySQL client on a machine with access to the database endpoint.

$ mysql -h $DB\_ENDPOINT -P 3306 -umsadmin -p

Enter password:

Welcome to the MariaDB monitor. Commands end with ; or \g.

Your MySQL connection id is 145068

Server version: 5.7.12-log MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> SELECT \* FROM dev.product;

+--------+-------------------+

| prodId | prodName |

+--------+-------------------+

| 999 | Mountain New Bike |

| 1000 | EV Car |

+--------+-------------------+

2 rows in set (0.01 sec)

MySQL [(none)]>

**Clean up**

# Remove the Demonstration Application

helm uninstall workshop

# Remove the IAM Resources.

eksctl delete iamserviceaccount --name aurora-irsa --namespace workshop --cluster eksworkshop-eksctl --wait

aws iam delete-policy --policy-arn arn:aws:iam::${ACCOUNT\_ID}:policy/Aurora\_IAM\_Policy

**Deleting Aurora DB**

You can delete the Aurora database using the instructions in the [User Guide for Aurora](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/USER_DeleteCluster.html#USER_DeleteCluster.DeleteCluster).

**Deleting EKS cluster**

You can delete the EKS cluster using the instructions [Deleting an Amazon EKS cluster](https://docs.aws.amazon.com/eks/latest/userguide/delete-cluster.html) in the Amazon EKS User Guide.

**Conclusion**

In this blog, we learned that you can authenticate to your Aurora MySQL DB clusters using IAM database authentication from an application running on Amazon EKS. With this authentication method, you don’t need to use a password when you connect to your database cluster. Instead, we have used a short-lived authentication token so that you don’t have to worry about the storage and lifecycle of username and password credentials. Finally, we have also seen that network traffic to and from the database is encrypted using SSL and used as part of our connection to the database.

In this blog post, we have shown an application deployed in Amazon EKS, but the IAM database authentication can be used with many other types of AWS infrastructure, such as AWS Lambda, Amazon EC2, AWS Fargate, or an Amazon ECS task. For more information, refer to:

* <https://aws.amazon.com/blogs/database/iam-role-based-authentication-to-amazon-aurora-from-serverless-applications/>
* <https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.html>