Deploying Denodo in Amazon Elastic Kubernetes Service (EKS)

Revision 20200911

Denodo North America & APAC: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA
Denodo Iberia & Latino América: Montalbán 5, 28014 Madrid, Spain
Denodo EMEA: 21st Floor, Portland House, Bressenden Place, London SW1E 5RS. UK
Denodo DACH: Karlstraße 10, 80333 München. Germany
www.denodo.com

The Elastic Kubernetes Service (EKS) is a service provided by Amazon AWS that supports deploying Kubernetes clusters in AWS. EKS has become the default technology when working with AWS + Kubernetes due to its ease of use and its integration features with other AWS services.

Also, since EKS is a certified Kubernetes conformant, any standard Kubernetes application can be easily migrated to EKS without the need of any code refactoring.

In this article we will show how to create a Kubernetes cluster in EKS using the AWS Command Line Interface, and how to deploy a Denodo Kubernetes application in it.

1 AWS COMMAND LINE INTERFACE

The AWS Command Line Interface (AWS CLI) is an open-source tool that can be used for administering the AWS resources and we will use it extensively in the article, we will assume that it is installed and up to date as a prerequisite for executing the commands presented below.

Although it is also possible to manage the EKS resources from the AWS portal, in this article we have chosen to use the AWS CLI because usually, the commands are self-explanatory and also because it is possible to use these commands and test them easily in any custom AWS deployment.

2 AMAZON ELASTIC KUBERNETES SERVICE

The Elastic Kubernetes Service (EKS) eases the management, configuration and daily work of containerized environments in AWS. In other Knowledge Base articles, we explained how to deploy a Denodo container in a local Kubernetes environment and now, we will see how to perform this deployment into a cluster that is hosted in Amazon AWS, making use of the Elastic Kubernetes Service.

Note that all the below statements are AWS CLI commands, so they can be copied into any environment to create a Denodo cluster in AWS.

The Kubernetes deployment is organized in three main parts:

- Creation of the container registry: the Kubernetes cluster will use an Elastic Container Registry to obtain the container images.
- Creation of the Kubernetes cluster: the cluster will accept the deployment of the Denodo image.
- Deployment of a Virtual DataPort application in the cluster: the last step shows the deployment of the Denodo container image, explaining the implications of doing this in AWS.

2.1 LIST OF PREREQUISITES

This article assumes that there is an environment already configured with the tools required to deploy a Docker image of Denodo in AWS, in other cases, it might be needed to install or configure something else before continuing with the next section. In summary:

- The AWS CLI and a valid AWS account with enough privileges to build all the AWS elements involved in the deployment of a Denodo image.
- A Docker image of Denodo in the local registry.
- Additionally, it will be easier to configure the deployment file if a local Kubernetes environment is created first, with a tested `denodo-service.yaml`.

2.2 AWS PERMISSIONS

An IAM entity (user, group or role) must be set up with the necessary privileges required by the AWS CLI for running the statements in this guide.

In this example, we have configured an IAM Role (`roleARN`) to use ECR and EKS services.

This IAM role must be able to list, push and pull images in the ECR repository, which is granted with the following AWS managed policy:

- `AmazonEC2ContainerRegistryPowerUser`.

The `AmazonEC2ContainerRegistryPowerUser` allows read and write access to repositories, but does not allow other actions such as actually creating the ECR registry. For that, we need to assign the following IAM action to the role:

- `ecr:CreateRepository`

Also, if the Service Linked Role "AWSServiceRoleForAmazonEKSNodegroup" does not exist, the following IAM actions will be required by that entity in order to create it:

- iam:CreateServiceLinkedRole
- iam:PutRolePolicy

Additionally, the role must be granted with the following permissions over the roles that will be used later for the cluster creation (eksRoleARN) and for nodegroup creation (ec2RoleARN) statements:

- iam:PassRole
- iam:GetRole
- iam:ListAttachedRolePolicies
- ec2:DescribeSubnets

The Amazon IAM service role for the EKS service (eksRoleARN) that will provide permissions for making calls to other AWS API operation must have the following policies attached:

- AmazonEKSClusterPolicy
- AmazonEKSServicePolicy

The IAM service role for the EC2 service (ec2RoleARN) that will be used by the Node Group workers must have the following policies attached:

- AmazonEKSWorkerNodePolicy
- AmazonEC2ContainerRegistryReadOnly
- AmazonEKS_CNI_Policy

2.3 CONFIGURE THE AWS CLI

To start with the AWS configuration, open a new console and execute the below AWS commands that will perform the following actions:

Configure the AWS access keys for the AWS account by using the following command:

```
$ aws configure
```

When entering this command, the AWS CLI prompts for some additional information. To read more about this check the AWS CLI user guide.

Use these commands to configure the settings in the default profile:

```
$ aws configure set role_arn <roleARN>
$ aws configure set source_profile default
```

The ARN role will have this format: arn:aws:iam::<accountNumber>:role/<roleName>

In addition to configuring the settings in the default profile, it is also possible to configure a different named profile. In this case, most of the commands below will require providing the profile name to use as a parameter, it is convenient to configure the named profile as part of the local environment appropriately.

2.4 CREATE THE CONTAINER REGISTRY

Create a new Elastic Container Registry by providing a custom name for the repository:

```
$ aws ecr create-repository --repository-name <denodo-ecr>
```

Check that the `<denodo-ecr>` repository has been created successfully and obtain its `repositoryUri` from the output of the `create-repository` statement. The field `repositoryUri` will be used later to authenticate the local Docker client.

To obtain the `repositoryUri` value the following command can be used:

```
$ aws ecr describe-repositories --repository-name <denodo-ecr> --query
"repositories[].repositoryUri"
```

Authenticate the Docker client to the new registry to push the Denodo image to the repository using the `repositoryUri`:

```
$ aws ecr get-login-password | docker login --username AWS --password-stdin
<repositoryUri>
```

Finally, tag the Denodo image with the ECR registry and push it to the Elastic Container Registry.

```
$ docker tag denodo-platform <repositoryUri>
$ docker push <repositoryUri>
$ aws ecr list-images --repository-name <denodo-ecr>
```

2.5 CREATE THE KUBERNETES CLUSTER

The command that we will use to create the EKS cluster from the AWS CLI requires creating first the following elements that will be used in the command invocation:

- An Amazon IAM service role for the EKS service (`eksRoleARN`) that provides permissions to Amazon EKS for making calls to other AWS API operations on your behalf.
- The Amazon VPC subnets for the cluster to use (a list of subnet ids separated by commas - `subnetIdListC`) and dedicated security group (`securityGroupId`) for the Amazon EKS cluster.

Once we get previous requirements fulfilled, we can create the cluster with the following command by providing the previous parameters and a name for the new cluster:

```
$ aws eks create-cluster \
      --name <denodo-eks-cluster> \
```

```
      --role-arn <eksRoleARN> \
      --resources-vpc-config
subnetIds=<subnetIdListC>,securityGroupIds=<securityGroupId>
```

It can take some time for the cluster to be available, it is possible to check the cluster status with the following command:

```
$ aws eks describe-cluster --name <denodo-eks-cluster> --query
"cluster.status"
```

Once the cluster is `ACTIVE`, it is necessary to add worker nodes to it. These nodes are deployed in node groups, which are one or more Amazon EC2 instances deployed in an Amazon EC2 Auto Scaling group.

2.6 ADDING NODES TO THE CLUSTER

Amazon EKS clusters can schedule pods on different types of nodegroups or a combination of them:

- Self-managed nodes: with this option the Amazon EC2 nodes are created manually
- Managed node groups: in this case the nodes are provisioned automatically by EKS
- AWS Fargate: a Fargate profile defines which pods start on Fargate

For a comparison of the capabilities of each node management system you can check the EKS compute documentation.

In addition, the nodes of the nodegroups can use different Operating Systems such as Amazon Linux, Ubuntu Linux, Bottlerocket or a custom one although customizing the Operating System of the nodes may require using Self-managed nodes. Amazon does provide EKS optimized AMIs for some Operative Systems and instructions to build your own.

For the sake of simplicity, we have decided to use EKS managed node groups with Amazon Linux, but notice that the other options are also valid. The rest of the steps provided on this guide after the nodes are added to the cluster do not depend on how the nodes are managed nor the Operating Systems selected for them.

The command `aws eks create-nodegroup` creates the managed worker node group, but in order to create it the following information is required:

- The IAM service role for the EC2 service to be used by the Node Group workers (`ec2RoleARN`).
- The Amazon VPC subnets for the node group workers to use (the same subnets created in the previous step, but this time separated by spaces - `subnetIdListS`)

Once we get previous parameters we can create the cluster with the following command by providing a name for the new cluster and for the node group:

```
$ aws eks create-nodegroup \
      --cluster-name <denodo-eks-cluster> \
      --nodegroup-name <denodo-nodegroup> \
      --node-role <ec2RoleARN> \
      --subnets <subnetIdListS>
```

The node group takes some time to get ready, but it is possible to check the status of the node group with the command:

```
$ aws eks describe-nodegroup --cluster-name <denodo-eks-cluster>
--nodegroup-name <denodo-nodegroup> --query "nodegroup.status"
```

After this, the cluster should be running in AWS EKS. In order to connect to it with the Kubernetes CLI `kubectl` we can create the proper configuration in Kubernetes by executing the following command:

```
$ aws eks update-kubeconfig --name <denodo-eks-cluster>
```

To check if the configuration is successful, connect to the cluster and get the nodes information with `kubectl`.

```
$ kubectl get nodes
$ kubectl describe service
```

2.7 DEPLOYING DENODO VIRTUAL DATAPORT

The Denodo Platform requires a valid license in order to start, which can be obtained from a Denodo License Manager server or in some scenarios (like evaluation or Denodo Express) as a standalone license file. In order to avoid static references, we can use a Kubernetes config map that will embed a Solution Manager configuration file pointing to the License server or a valid license file.

2.7.1 Deploying Denodo Virtual DataPort in a Solution Manager Environment

If we have a Denodo License Manager server running and accessible by the Kubernetes cluster then it can be configured as part of the Solution Manager Configuration.

First, we need to create the config map containing the Solution Manager configuration that will be referenced later from the `denodo-service.yaml` file:

```
$ kubectl create configmap solution-manager --from-
file=SolutionManager.properties=<pathToConfigurationFile>
```

After this, we can deploy the Virtual DataPort server in the EKS cluster. Taking as starting point the *denodo-service.yaml* from the Knowledge Base article Deploying Denodo in Kubernetes, we will have to update two sections of the base file in order to accomplish a successful deployment.

The first modification will consist in setting up correctly the image name, replace the following line in the `denodo-service.yaml` file:

```
image: denodo-platform:8.0-latest
```

with:

```
image: <repositoryUri>:latest
```

Notice that the value of the placeholder `<repositoryUri>` was already obtained in a previous step and it refers to the login server address of your registry.

In addition, it is required to modify the YAML file in order to copy the solution manager configuration file from the config map to the right location. The reason is that in the original YAML file, the configuration file was mounted directly from the local file system, but in this case we will load the file from the config map `solution-manager` that we have just created with the previous kubectl statement.

Hence, after applying the mentioned updates, the Denodo 8.0 version of the file would look like this:

```
apiVersion: v1
kind: Service
metadata:
  name: denodo-service
spec:
  selector:
    app: denodo-app
  ports:
  - name: svc-jdbc
    protocol: "TCP"
    port: 8999
    targetPort: jdbc
  - name: svc-rmi-r
    protocol: "TCP"
    port: 8997
    targetPort: jmx-rmi-rgstry
  - name: svc-rmi-f
    protocol: "TCP"
    port: 8995
    targetPort: jmx-rmi-fctory
  - name: svc-odbc
    protocol: "TCP"
    port: 8996
    targetPort: odbc
  - name: svc-web
    protocol: "TCP"
    port: 8090
    targetPort: web-container
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: denodo-deployment
spec:
```

```
  selector:
    matchLabels:
      app: denodo-app
  replicas: 1
  template:
    metadata:
      labels:
        app: denodo-app
    spec:
      hostname: denodo-hostname
      containers:
      - name: denodo-container
        image: <repositoryUri>:latest
        command: ["./denodo-container-start.sh"]
        args: ["--vqlserver"]
        env:
        - name: FACTORY_PORT
          value: "8995"
        ports:
        - name: jdbc
          containerPort: 9999
        - name: jmx-rmi-rgstry
          containerPort: 9997
        - name: jmx-rmi-fctory
          containerPort: 8995
        - name: odbc
          containerPort: 9996
        - name: web-container
          containerPort: 9090
        lifecycle:
          postStart:
            exec:
                                              command:   ["/bin/sh",   "-c",   "cp
/opt/denodo/sm/SolutionManager.properties
/opt/denodo/conf/SolutionManager.properties"]
        volumeMounts:
        - name: config-volume
          mountPath: /opt/denodo/sm
          readOnly: true
      volumes:
      - name: config-volume
        configMap:
          name: solution-manager
          items:
          - key: SolutionManager.properties
            path: SolutionManager.properties
```

denodo-service.yaml for Denodo Platform 8.0

On the other hand, this would be the *denodo-service.yaml* version for Denodo Platform 7.0:

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: denodo-service
spec:
  selector:
    app: denodo-app
  ports:
  - name: svc-rmi-r
    protocol: "TCP"
    port: 8999
    targetPort: jdbc-rmi-rgstry
  - name: svc-rmi-f
    protocol: "TCP"
    port: 8997
    targetPort: jdbc-rmi-fctory
  - name: svc-odbc
    protocol: "TCP"
    port: 8996
    targetPort: odbc
  - name: svc-web
    protocol: "TCP"
    port: 8090
    targetPort: web-container
  type: LoadBalancer
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: denodo-deployment
spec:
  selector:
    matchLabels:
      app: denodo-app
  replicas: 1
  template:
    metadata:
      labels:
        app: denodo-app
    spec:
      hostname: denodo-hostname
      containers:
      - name: denodo-container
        image: <repositoryUri>:latest
        command: ["./denodo-container-start.sh"]
        args: ["--vqlserver"]
        env:
        - name: FACTORY_PORT
          value: "8997"
        ports:
        - name: jdbc-rmi-rgstry
          containerPort: 9999
        - name: jdbc-rmi-fctory
          containerPort: 8997
        - name: odbc
```

```
          containerPort: 9996
        - name: web-container
          containerPort: 9090
        lifecycle:
          postStart:
            exec:
                                              command:   ["/bin/sh",   "-c",   "cp
/opt/denodo/sm/SolutionManager.properties
/opt/denodo/conf/SolutionManager.properties"]
        volumeMounts:
        - name: config-volume
          mountPath: /opt/denodo/sm
          readOnly: true
      volumes:
      - name: config-volume
        configMap:
          name: solution-manager
          items:
          - key: SolutionManager.properties
            path: SolutionManager.properties
```

denodo-service.yaml for Denodo Platform 7.0

Then, take the appropriate version of the *denodo-service.yaml* and deploy the app and service in Kubernetes with the following command:

```
$ kubectl apply -f denodo-service.yaml
```

After all these steps a Virtual DataPort server will be running in the EKS cluster that we created. AWS will provide a public IP address for the Kubernetes service that can be used for connecting to Virtual DataPort. In order to know the address, we can use the following command to obtain the public IP address for the Kubernetes service:

```
$ kubectl get service denodo-service
```

```
C:\>kubectl get service denodo-service
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP                                PORT(S)
denodo-service    LoadBalancer   172.20.197.13   [redacted].elb.amazonaws.com   8999:31674/TCP,8997:32542/TCP,8995:30593/TCP,8996:31495/TCP,8090:31220/TCP
```

Output from the execution of the get service command

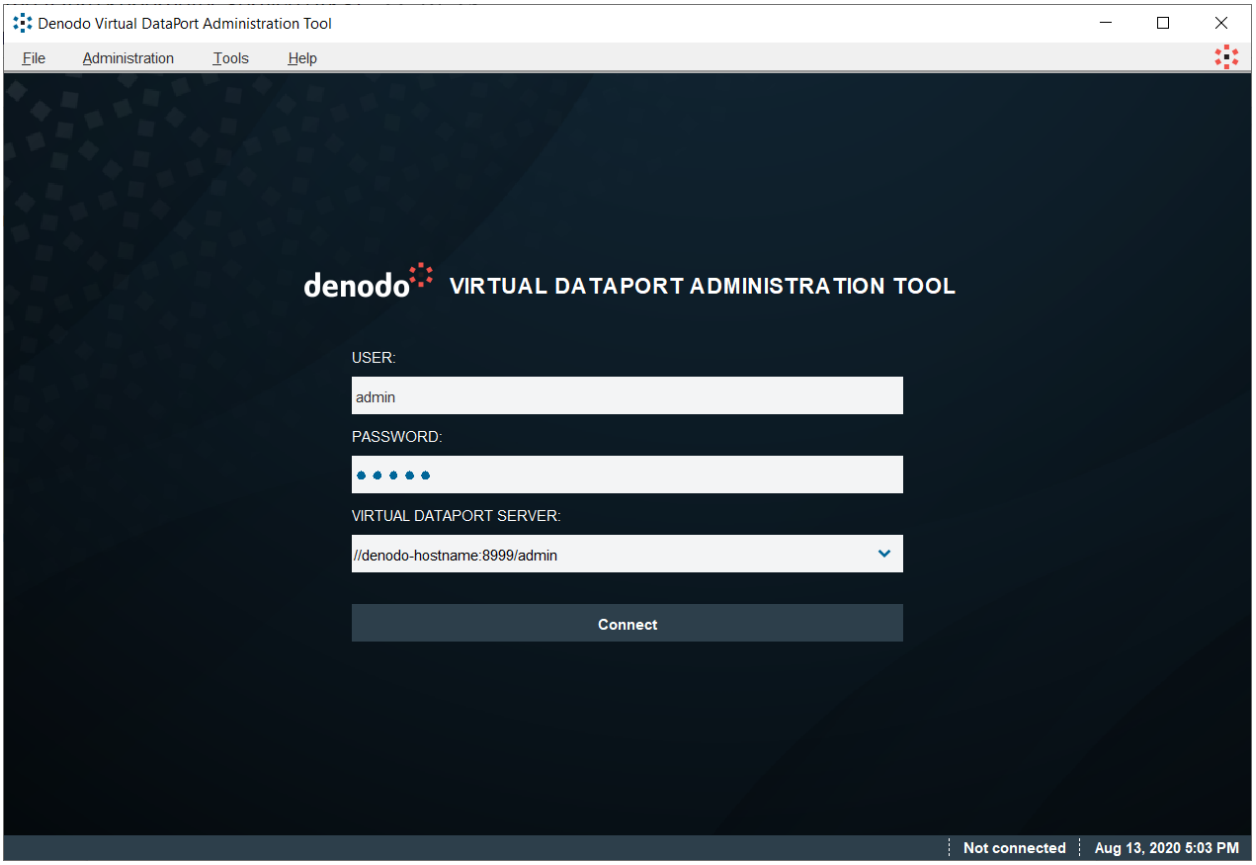
Also, notice that in the YAML configuration file, we are specifying the hostname for the Virtual DataPort server as `denodo-hostname`, so in order to connect to the server from a VDP client, we will need to ensure that the VDP client is able to resolve this hostname to the public IP address of the Kubernetes service, and we can do that by adding an entry in our local hosts configuration file to map that hostname with the IP address:

```
# Denodo Kubernetes service
20.30.40.50 denodo-hostname
```

The new entry in the Hosts file

Now we can open a new Virtual DataPort Administration Tool and connect to the server using the following Denodo Server URI:

```
//denodo-hostname:8999/admin
```



2.7.2 Deploying Denodo Virtual DataPort with a Standalone License

To use a standalone license file, we can use the following statement to create a map with the contents of the license file that will be referenced later from the `denodo-service.yaml` file:

```
$ kubectl create configmap denodo-license --from-file=denodo.lic=<pathToLicenseFile>
```

Then, we need to modify the YAML file to load the license file in the Denodo installation folder `<DENODO_HOME>/conf` as it is done with the solution manager configuration. For this, replace the following part of the previous file `denodo-service.yaml`:

```
...
lifecycle:
  postStart:
```

```
            exec:
                                              command:   ["/bin/sh",   "-c",   "cp
/opt/denodo/sm/SolutionManager.properties
/opt/denodo/conf/SolutionManager.properties"]
        volumeMounts:
        - name: config-volume
          mountPath: /opt/denodo/sm
          readOnly: true
      volumes:
      - name: config-volume
        configMap:
          name: solution-manager
          items:
          - key: SolutionManager.properties
            path: SolutionManager.properties
```

with:

```
        ...
        lifecycle:
          postStart:
            exec:
                                              command:   ["/bin/sh",   "-c",   "cp
/opt/denodo/conf/license/denodo.lic /opt/denodo/conf/denodo.lic"]
        volumeMounts:
        - name: config-volume
          mountPath: /opt/denodo/conf/license
          readOnly: true
      volumes:
      - name: config-volume
        configMap:
          name: denodo-license
          items:
          - key: denodo.lic
            path: denodo.lic
```

2.8 AWS LOAD BALANCER IDLE TIMEOUT

When a query is sent to Virtual DataPort, the connection open to the load balancer will remain idle while the query runs and there is no data returned back. By default, the idle timeout for the Classic Load Balancer connections is 60 seconds, so this means that by default, the queries can be running successfully up to one minute, after that, they will be closed by the ELB.

In general, we can expect Virtual DataPort to receive queries that will execute for longer than one minute, so in those cases, it is very important to increase the value for the idle timeout in order to avoid getting closed the connections from those queries.

The idle timeout issue can be solved by adding the following annotation to the YAML definition of the Denodo service. Notice that Kubernetes and AWS provide many annotations that can be used for tuning the load balancer created in the service. One of

those annotations allows to change the default value of the idle timeout, for instance to 900 seconds:

```
apiVersion: v1
kind: Service
metadata:
  name: denodo-service
  annotations:
       service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout:
"900"
...
```

3 REFERENCES

AWS Command Line Interface
Amazon Elastic Kubernetes Service
Deploying Denodo in Kubernetes
How to create your own Docker images for Denodo Platform Containers