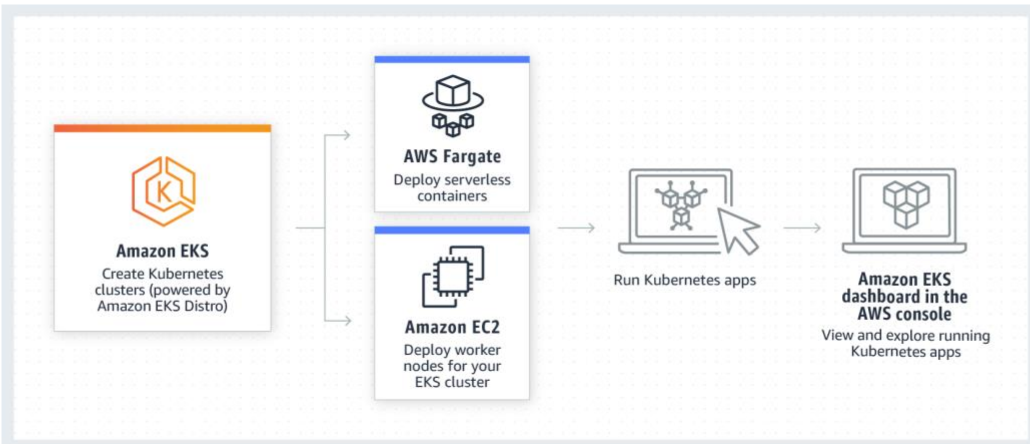
CS 548—Fall 2023
Enterprise Software Architecture and Design
Assignment Nine—Kubernetes

In this assignment, you will deploy the microservices that you developed in previous assignments into a Kubernetes cluster. We will use the Elastic Kubernetes Services (EKS) provided by Amazon Web Services[1]. We will base the guidelines for using EKS on the quickstart guide, specifically that using `kubectl` and `eksctl`[2], and we will use EC2 to deploy the worker nodes in the cluster.



Step 1: Install tools

Download and install AWS `kubectl` on your laptop[3]. If you have installed Docker Desktop, this will have installed its own version of `kubectl`, so you will need to set up your file path to make sure you use the Amazon version. Install version 1.25 from Amazon S3.

Download and install `eksctl` on your laptop[4]. This tool will allow you to manage your Kubernetes cluster in EKS from your laptop. If you use homebrew on MacOS to install `eksctl`, it will install `kubectl` as well, so you can skip the step above. If you have installed Docker Desktop on MacOS, it will have installed `kubectl` at the same location where homebrew wishes to install it, so you will have to rename the version that is already there.

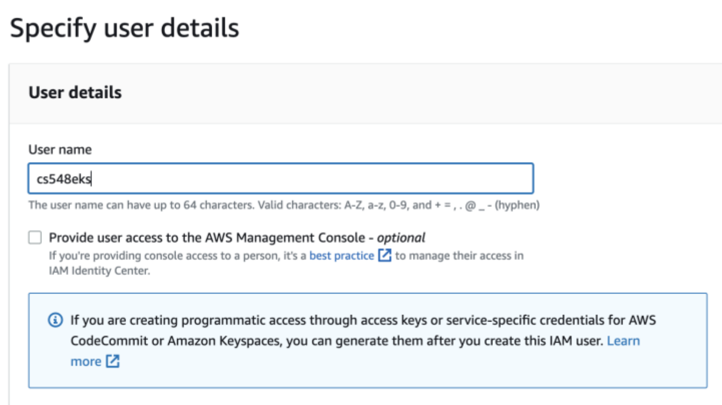
Download and install the AWS command line interface (CLI) on your laptop[5]. You will need this to configure the IAM identity you will use to authenticate to AWS with `eksctl` and `kubectl`.

Step 2: Create IAM User for EKS

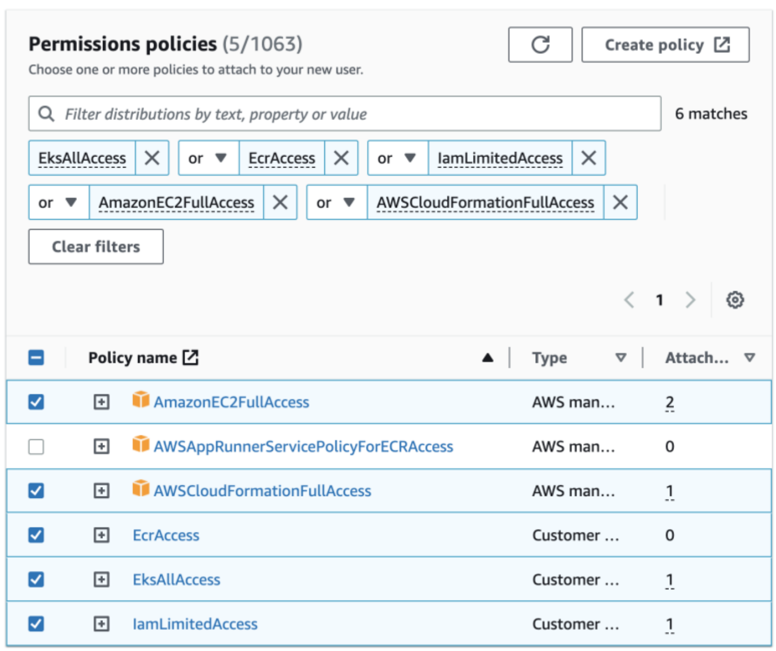
---

[1] https://aws.amazon.com/eks/.
[2] https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html.
[3] https://docs.aws.amazon.com/eks/latest/userguide/install-kubectl.html.
[4] https://docs.aws.amazon.com/eks/latest/userguide/eksctl.html.
[5] https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html.

To use `kubectl` from your laptop, you will need to authenticate to AWS. You should never use a root user for this purpose. Instead, following the principle of least privilege, create an IAM user with a policy that restricts their access:



You will need to set permissions for this user[6]. The following summarizes the policies that are required:



In addition to the two AWS-managed policies, that you cannot edit, you need to add three policies that you should customize with your AWS account id. First, there is the policy `EksAllAccess`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

[6] https://eksctl.io/usage/minimum-iam-policies/.

```
            "Effect": "Allow",
            "Action": "eks:*",
            "Resource": "*"
        },
        {
            "Action": [
                "ssm:GetParameter",
                "ssm:GetParameters"
            ],
            "Resource": [
                "arn:aws:ssm:*:<account_id>:parameter/aws/*",
                "arn:aws:ssm:*::parameter/aws/*"
            ],
            "Effect": "Allow"
        },
        {
             "Action": [
               "kms:CreateGrant",
               "kms:DescribeKey"
             ],
             "Resource": "*",
             "Effect": "Allow"
        },
        {
             "Action": [
               "logs:PutRetentionPolicy"
             ],
             "Resource": "*",
             "Effect": "Allow"
        }
    ]
}
```

Second, there is the policy `IamLimitedAccess`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreateInstanceProfile",
                "iam:DeleteInstanceProfile",
                "iam:GetInstanceProfile",
                "iam:RemoveRoleFromInstanceProfile",
                "iam:GetRole",
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:AttachRolePolicy",
                "iam:PutRolePolicy",
                "iam:ListInstanceProfiles",
                "iam:AddRoleToInstanceProfile",
                "iam:ListInstanceProfilesForRole",
                "iam:PassRole",
                "iam:DetachRolePolicy",
                "iam:DeleteRolePolicy",
                "iam:GetRolePolicy",
                "iam:GetOpenIDConnectProvider",
                "iam:CreateOpenIDConnectProvider",
                "iam:DeleteOpenIDConnectProvider",
```

```
                "iam:TagOpenIDConnectProvider",
                "iam:ListAttachedRolePolicies",
                "iam:TagRole",
                "iam:GetPolicy",
                "iam:CreatePolicy",
                "iam:DeletePolicy",
                "iam:ListPolicyVersions"
            ],
            "Resource": [
                "arn:aws:iam::<account_id>:instance-profile/eksctl-*",
                "arn:aws:iam::<account_id>:role/eksctl-*",
                "arn:aws:iam::<account_id>:policy/eksctl-*",
                "arn:aws:iam::<account_id>:oidc-provider/*",
                "arn:aws:iam::<account_id>:role/aws-service-role/eks-
nodegroup.amazonaws.com/AWSServiceRoleForAmazonEKSNodegroup",
                "arn:aws:iam::<account_id>:role/eksctl-managed-*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:GetRole"
            ],
            "Resource": [
                "arn:aws:iam::<account_id>:role/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": [
                        "eks.amazonaws.com",
                        "eks-nodegroup.amazonaws.com",
                        "eks-fargate.amazonaws.com"
                    ]
                }
            }
        }
    ]
}
```

You also need to provide access for Elastic Container Registry, call this EcrAccess:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "ecr:PutImageTagMutability",
                "ecr:StartImageScan",
                "ecr:DescribeImageReplicationStatus",
                "ecr:ListTagsForResource",
                "ecr:UploadLayerPart",
```

```
                "ecr:BatchDeleteImage",
                "ecr:ListImages",
                "ecr:BatchGetRepositoryScanningConfiguration",
                "ecr:DeleteRepository",
                "ecr:CompleteLayerUpload",
                "ecr:TagResource",
                "ecr:DescribeRepositories",
                "ecr:BatchCheckLayerAvailability",
                "ecr:ReplicateImage",
                "ecr:GetLifecyclePolicy",
                "ecr:PutLifecyclePolicy",
                "ecr:DescribeImageScanFindings",
                "ecr:GetLifecyclePolicyPreview",
                "ecr:PutImageScanningConfiguration",
                "ecr:GetDownloadUrlForLayer",
                "ecr:DeleteLifecyclePolicy",
                "ecr:PutImage",
                "ecr:UntagResource",
                "ecr:BatchGetImage",
                "ecr:DescribeImages",
                "ecr:StartLifecyclePolicyPreview",
                "ecr:InitiateLayerUpload",
                "ecr:GetRepositoryPolicy"
            ],
            "Resource": "arn:aws:ecr:*:<account_id>:repository/*"
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": [
                "ecr:GetRegistryPolicy",
                "ecr:BatchImportUpstreamImage",
                "ecr:CreateRepository",
                "ecr:DescribeRegistry",
                "ecr:DescribePullThroughCacheRules",
                "ecr:GetAuthorizationToken",
                "ecr:PutRegistryScanningConfiguration",
                "ecr:CreatePullThroughCacheRule",
                "ecr:DeletePullThroughCacheRule",
                "ecr:GetRegistryScanningConfiguration",
                "ecr:PutReplicationConfiguration"
            ],
            "Resource": "*"
        }
    ]
}
```

Finish creating the IAM user with these policies. Now create a secret access key[7] that will allow you to programmatically access the EKS cluster. Download the csv file that includes the secret access key, and **carefully** save it where no-one else can get access to it.

[7] Secret access keys are **not** a best practice, having been the source of major data breaches, and should now be considered obsolete. However, we will use them for this assignment, for convenience. A better approach for a production system, with more setup cost, would be the use of IAM Roles Anywhere, setting up a public key infrastructure (PKI) to authenticate external programmatic access to AWS services. Another approach, in an enterprise setting, would be to set up single sign-on (SSO) with IAM Identity Center, enabling federated authentication with an enterprise identity provider (IdP), with access based on a temporary access token provided by the IdP when you authenticate to it.

Now that you have created an IAM user with a secret access key and restricted permissions, you should configure the AWS CLI on your own computer with the credentials you have created for the IAM user `cs548eks`:

```
$ aws configure
```

You should save your access ID and secret access key for the IAM user you created when you are prompted, as well as the default region[8] (e.g., `us-east-2`) and output format (`json`). Verify that you have configured your credentials to authenticate as `cs548eks`:

```
$ aws sts get-caller-identity
```

Step 3: Create an EKS Cluster

Use the `eksctl` CLI to create your Kubernetes cluster in EKS[9]:

```
$ eksctl create cluster --name=cluster-name
```

This will create a cluster of managed nodes, with a public and private subnet, in your default region. It will be replicated across several availability zones in your region. For debugging purposes, you can restrain the managed nodes to a single availability zone:

```
$ eksctl create cluster --name=cluster-name --zones=us-east-2a,us-east-2b --
node-zones=us-east-2a
```

By default, `eksctl` will use `m5.large` EC2 instances, which can be expensive (Remember that the idea is to scale up the number of pods on running worker nodes). You can specify criteria for choosing an instance and limit the number of worker nodes that are created, and you can use the `--dry-run` option to see what the actual configuration of the cluster will look like:

```
$ eksctl create cluster --name=cluster-name --zones=us-east-2a,us-east-2b --
node-zones=us-east-2a --instance-selector-vcpus=2 --instance-selector-memory=4
--nodes=1 --dry-run
```

When you are finished with the cluster, delete it as follows:

```
$ eksctl delete cluster --name=cluster-name
```

You can view the resources created for a cluster in the CloudFormation console[10]. Creating a cluster may take some time, sometimes up to twenty minutes, so you should review creation and deletion in the CloudFormation console. Make sure that you select the appropriate region in the AWS console (e.g., Ohio for `us-east-2`). You can view the cluster nodes and the workloads deployed in the cluster using these commands:

```
$ kubectl get nodes -o wide
$ kubectl get pods -A -o wide
```

---

[8] There are capacity issues with `us-east-1`, so pick another region for your EKS cluster.
[9] https://eksctl.io/usage/creating-and-managing-clusters/.
[10] https://console.aws.amazon.com/cloudformation/.

By default, the cluster is open to public access. As with your EC2 instance, you will want to restrict access to your IP address or the Stevens network IP address range. If you do restrict public access, then you should allow private access within the Amazon network so that pods in the cluster can communicate among themselves. Use the AWS CLI to add this restriction[11], replacing the IP address below with your actual IP address[12]:

```
$ aws eks update-cluster-config
    --name cluster-name
    --resources-vpc-config endpointPublicAccess=true,publicAccessCidrs="your-ip-address/32",endpointPrivateAccess=true
```

You should execute the `kubectl` commands again to confirm that you still have access to the cluster. Optionally execute the following command to enable control plane logging in your cluster[13]:

```
$ aws eks update-cluster-config
    --name cluster-name
    --logging
'{"clusterLogging":[{"types":["api","audit","authenticator","controllerManager"],"enabled":true}]}'
```

You can view the logs at the AWS CloudWatch console[14].

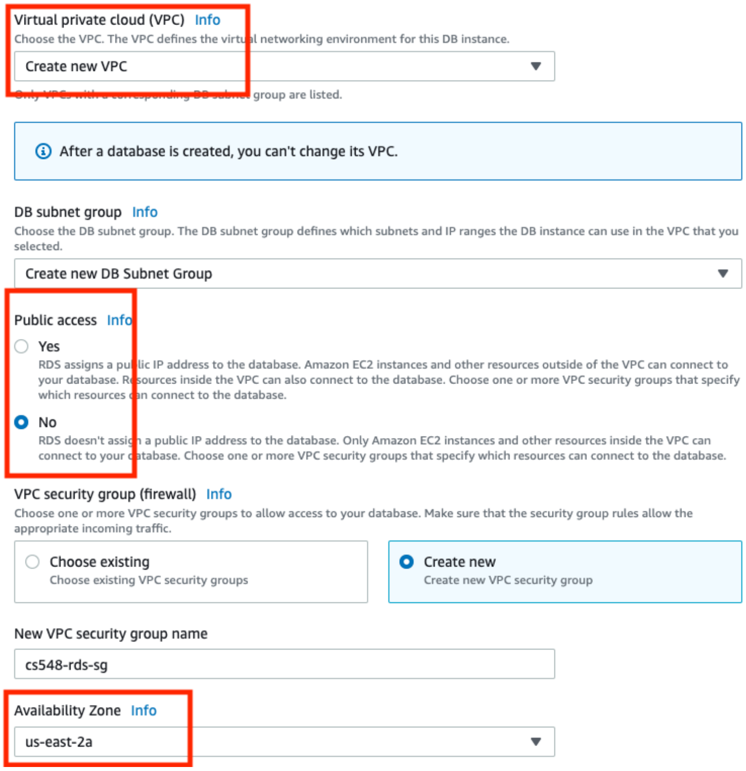
Step 4: Deploy the Database Server

Deploying a database server in Kubernetes is not for the faint of heart. We will use AWS Relational Database Service (RDS) to deploy the backend database as a service instead[15]. *Make sure you are doing this in the same region as the cluster you have already created.* Navigate to the RDS console[16] and select the button for creating a new database.

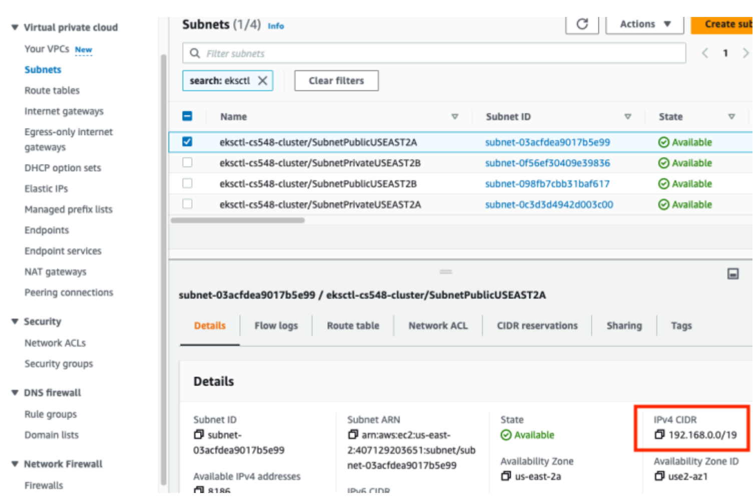
1. Choose "standard create" as the creation method.
2. Choose Postgresql as the engine type and "free tier" as the instance size.
3. Specify "`cs548db`" as the instance identifier, leave "`postgres`" as the master username and choose a good master password.
4. Set the allocated storage to the minimum for the free tier, 20G, and disable storage autoscaling.
5. Choose password authentication.
6. Disable performance insights.
7. For additional configuration, disable backup and encryption. Do not create the application database yet.
8. For other settings, see here:

---

[11] https://docs.aws.amazon.com/eks/latest/userguide/cluster-endpoint.html.
[12] https://whatismyipaddress.com/.
[13] https://docs.aws.amazon.com/eks/latest/userguide/control-plane-logs.html.
[14] https://console.aws.amazon.com/cloudwatch/home#logs:prefix=/aws/eks.
[15] https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_CreateDBInstance.html.
[16] https://console.aws.amazon.com/rds/.

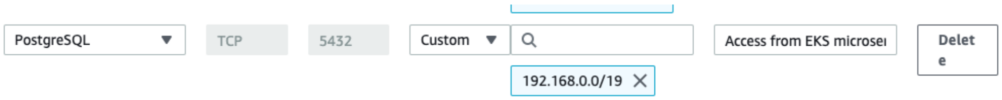


A new virtual private cloud (VPC) will have been created for the EKS cluster, and you are creating another VPC for the database. You will need to manually allow access from your Kubernetes cluster to the database. You will see that four subnets were created when the EKS virtual cloud was created: a public and a private subnet, in each of the two availability zones. We have configured the cluster to always deploy worker nodes in the public subnet of one of these (`us-east-2a`, above). You can find the CIDR information about this subnet by navigating to the VPC console[17] and selecting this subnet:

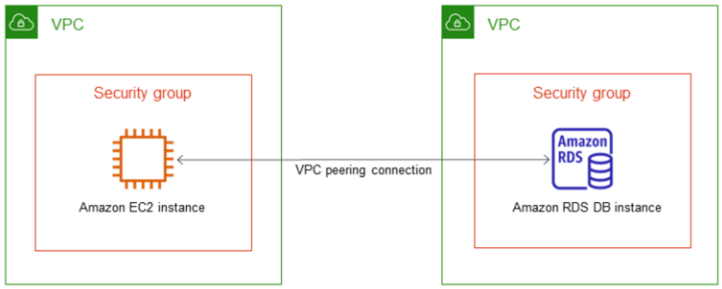


Now add a rule to the security group for the database that allows access from this subnet:

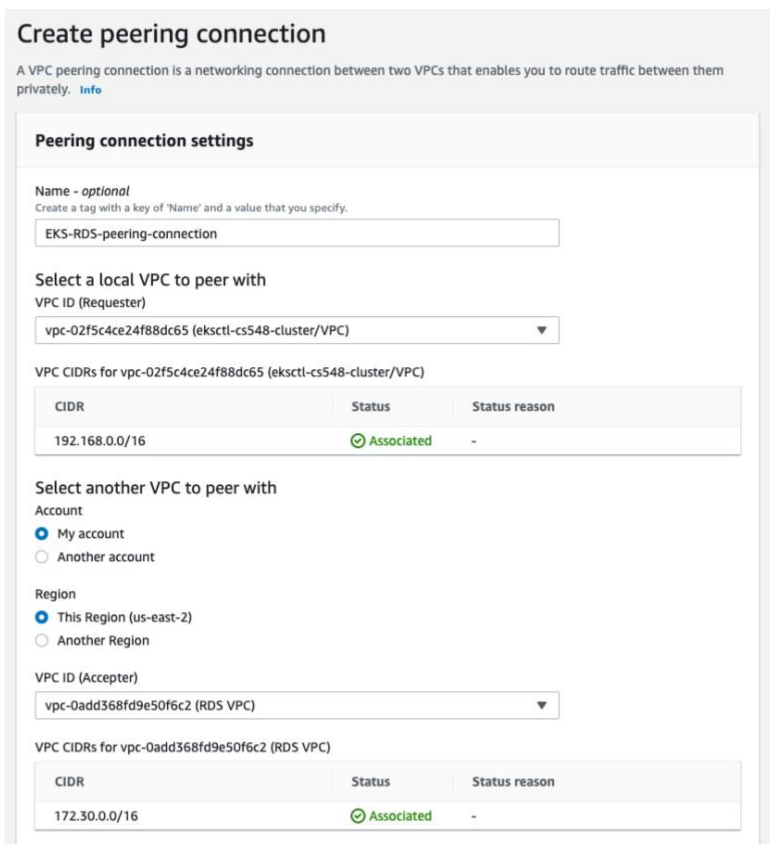
[17] https://console.aws.amazon.com/vpc.



You still need to enable routing from the VPC for the cluster to the VPC for the database, to allow them to communicate[18]. We will do this by creating a *peering connection* between the networks:



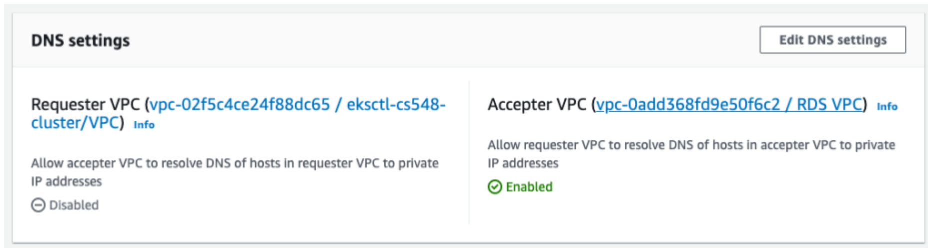
In the VPC console, select Peering Connections and click on Create Peering Connection. Then select the EKS VPC as the requester and the RDS VPC as the accepter:



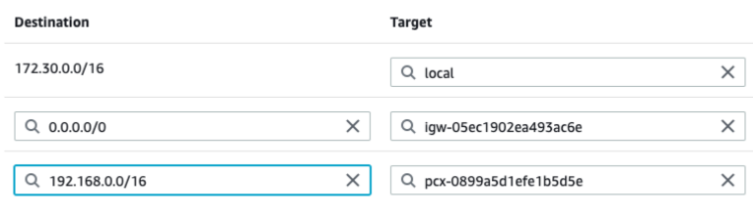
Be sure to accept your request! You will also need to allow DNS resolution of database host names from the EKS RDS:

---

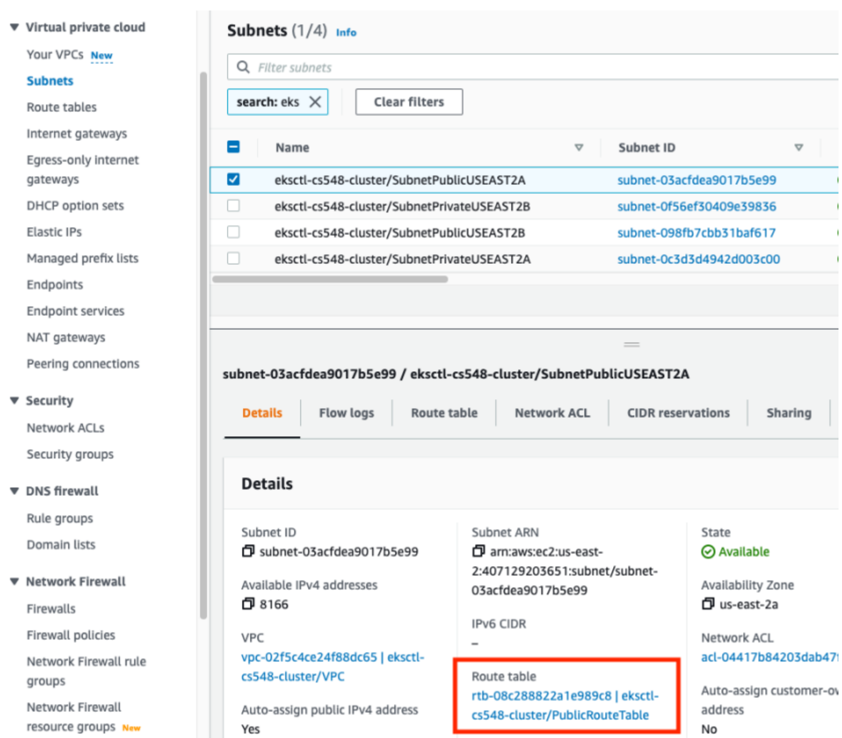
[18] https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_VPC.Scenarios.html.



You still need to update the route tables for the subnets for the EKS and RDS, to allow routing of traffic between the subnets[19]. In the RDS console, select the instance for the database. All subnets in the RDS VPC use the same route table, so pick any subnet associated with the database instance, and edit the route table to add a route for any EKS VPC CIDR destination that selects the peering connection as a target (*Remember to use the CIDR for your EKS VPC, from above, do not simply copy what is here*):

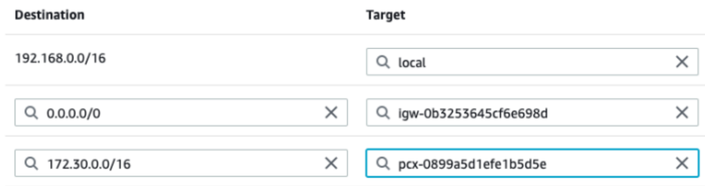


You also need to set up routing from the EKS cluster. From the VPC console, select for the EKS subnets and select the subnet for the availability zone where nodes are deployed:



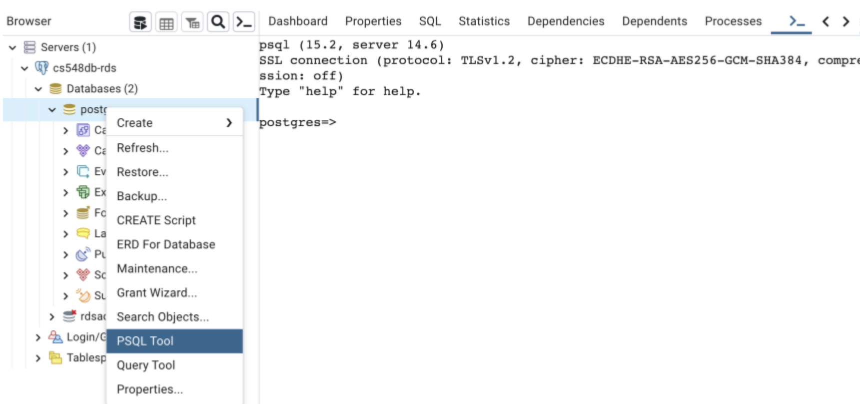
[19] https://docs.aws.amazon.com/vpc/latest/peering/vpc-peering-routing.html.

This time you will add a route from this subnet to the RDS VPC via the peering connection:



In order to set up the database, you will need to temporarily allow public access for the database server. Select the database server in the RDS console, click Modify, then under Connectivity | Additional Configuration, select Publicly Accessible. You may also need to add a rule to the security group for the database instance, allowing access from your IP address if it is not already present.

To connect to the database server, you can use `psql` if you have installed Postgresql on your laptop. Another possibility is to use `pgAdmin`[20], which provides a desktop application:



Once you have opened a `psql` window to the server, you can execute the SQL commands to create the database user used by the application and giving it access to the database:

```
CREATE USER cs548user PASSWORD '...';
GRANT cs548user TO postgres;
CREATE DATABASE cs548 WITH OWNER cs548user;
GRANT ALL PRIVILEGES ON DATABASE cs548 TO cs548user;
REVOKE cs548user FROM postgres;
# With Postgresql 15, need to grant permission to create tables in schema
# "\c" command connects to database as superuser "postgres"
\c cs548 postgres
GRANT ALL ON SCHEMA public TO cs548user;
```

The `GRANT` and `REVOKE` are required in RDS PostgreSQL because a user creating a database must own it, so the database superuser `postgres` (on whose behalf you are executing the commands from `psql`) must have the role of `cs548user` when creating the database. You

[20] https://www.pgadmin.org/.

can find more information about connecting to a database server in the RDS documentation[21].

Step 5: Copy Images to Private ECR Repositories

You need to copy the Docker images from the previous assignment to the AWS Elastic Container Registry[22]. You will need to create a private repository for each Docker image you are going to deploy, tag the image with metadata that ECR requires, and then copy the image to ECR[23]. We will illustrate this for the microservice image. We assume you have already created the image `cs548/clinic-domain` in your local registry.

First create the repository for the image in your private ECR registry:

```
$ aws ecr create-repository --repository-name clinic-domain
```

Now tag the image with the registry and repository from where it will be pulled by EKS:

```
$ docker tag cs548/clinic-domain
       account-id.dkr.ecr.us-east-2.amazonaws.com/clinic-domain
```

Authenticate to your registry in ECR:

```
$ aws ecr get-login-password | docker login --username AWS --password-stdin
       account-id.dkr.ecr.us-east-2.amazonaws.com
```

Push the image to the repository that you created earlier:

```
$ docker push account-id.dkr.ecr.us-east-2.amazonaws.com/clinic-domain
```

Repeat this step for each of the frontend applications from the previous assignment: web application (`cs548/clinic-webapp`) and web service (`cs548/clinic-rest`). You can view these private repositories at the ECR console[24].

Step 6: Deploy the Microservice

We will deploy each of these three applications in separate pods in EKS. Create a deployment configuration `clinic-domain-deployment.yaml` for the backend microservice as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: clinic-domain
  labels:
```

---

[21] https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ConnectToPostgreSQLInstance.html.

[22] https://aws.amazon.com/ecr/.

[23] https://docs.aws.amazon.com/AmazonECR/latest/userguide/getting-started-cli.html.

[24] https://console.aws.amazon.com/ecr/.

```
    app: clinic-domain
spec:
  replicas: 1
  selector:
    matchLabels:
      app: clinic-domain
  template:
    metadata:
      labels:
        app: clinic-domain
    spec:
      restartPolicy: Always
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/arch
                operator: In
                values:
                - amd64
      containers:
      - name: clinic-domain
        image: account-id.dkr.ecr.us-east-2.amazonaws.com/clinic-domain
        env:
        - name: DATABASE_USERNAME
          value: cs548user
        - name: DATABASE_PASSWORD
          value: YYYYYY
        - name: DATABASE
          value: cs548
        - name: DATABASE_HOST
          value: RDS_endpoint
        - name: memory.threshold
          value: 10485760
        imagePullPolicy: Always
```

The `matchLabels` field specifies how this deployment matches the pods that it may manage. The `imagePullPolicy` ensures that Kubernetes should pull the docker image to the local repository if it is not already present[25]. The affinity ensures that the pod is deployed on a AMD64 machine. The image identifier specifies that the image should be pulled to the pod from the private repository that you set up in the previous step. The environment variable bindings for the container match those from the previous assignment, except that the database host must refer to the endpoint in RDS for the database server (Remember that you should have enabled host name resolution in the peering connection). Start this microservice up as a pod in EKS:

```
$ kubectl apply -f clinic-domain-deployment.yaml
$ kubectl get pods
$ kubectl describe pod pod-name
$ kubectl logs pod-name
```

[25] It is obviously unsatisfactory to have to specify an Amazon account in the YAML file. *Helm Charts* would allow these specifications to be parameterized and then instantiated as part of deployment.

You can see from the latter the container being assigned to the worker nodes in the EKS cluster and the image being pulled from ECR (if it is not already in the cache). Next, create a service configuration `clinic-domain-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: clinic-domain
  labels:
    app: clinic-domain
spec:
  type: NodePort
  ports:
  - name: http
    port: 8080
  selector:
    app: clinic-domain
```

The *node port* is the port on the node (the EC2 instance) on which this service is exposed outside the cluster. Since it is not specified here, it will be generated automatically. The *port* is internal to the cluster and is that on which other pods in the cluster can access this service (with "virtual host name" `clinic-domain`), and accesses on the node port are forwarded to this service port. Accesses on the service port are forwarded to the *target port* 8080 in the pod (if not specified, this target port defaults to be the same as port). This target port is the port that the server in the pod should be listening on. We don't specify a HTTPS port because Payara Micro Community Edition does not support it. Start this service up in Kubernetes[26]:

```
$ kubectl apply -f clinic-domain-service.yaml
$ kubectl get service clinic-domain
$ kubectl describe service clinic-domain
```

Here is an example of the output from the last step:

```
Namespace:                default
Labels:                   app=clinic-domain
Annotations:              <none>
Selector:                 app=clinic-domain
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.100.212.39
IPs:                      10.100.212.39
Port:                     http  8080/TCP
TargetPort:               8080/TCP
NodePort:                 http  32435/TCP
Endpoints:                192.168.9.23:8080
Session Affinity:         None
External Traffic Policy:  Cluster
```

[26] You can also use the YAML file to undeploy an application, e.g.,

```
$ kubectl delete -f clinic-domain-service.yaml
$ kubectl delete -f clinic-domain-deployment.yaml
```

From the node port, you get the port on the EC2 instance that you can use to connect to the microservice. Communications to this port are forwarded to port 8080 on the container in the pod (the targetport). You can get the external IP addresses of the worker nodes using `kubectl`:

```
$ kubectl get nodes -o wide
```

If you have a bash shell, you can get a nicer output this way:

```
$ kubectl get nodes -o wide |  awk {'print $1" " $2 " " $7'} | column -t
```

You can test the deployment at the following URL, using the external IP address for the worker node and the node port (see above):

http://*external-ip-address*:32435/api/application.wadl

You may have to manually edit the security group for the worker node in the EC2 console, to enable access from your IP address. *This is not something that you would do in a production environment.*

Step 7: Deploy the applications

Each frontend application will be a client of the microservice. For the Web application, define the deployment descriptor:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: clinic-webapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: clinic-webapp
  template:
    metadata:
      labels:
        app: clinic-webapp
    spec:
      restartPolicy: Always
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/arch
                operator: In
                values:
                - amd64
      containers:
      - name: clinic-webapp
```

```
        image: account-id.dkr.ecr.us-east-2.amazonaws.com/clinic-webapp
        imagePullPolicy: Always
```

And also, the service descriptor:

```
apiVersion: v1
kind: Service
metadata:
  name: clinic-webapp
  labels:
    app: clinic-webapp
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 8080
  selector:
    app: clinic-webapp
```

This will expose the application to clients as a cloud application, with client requests routed through a load balancer that distributes the load among a replica set of pods. The endpoint for the service will now be exposed by the public port number 8080 *outside the cluster*, and you can view the state of the domain via a Web browser at this URL:

```
http://external-ip-address:8080/clinic
```

Accesses to the web application, outside the cluster at this URL, are forwarded by this service to the pod for the container at its internal cluster IP address and targetport 8080. Use a similar strategy to deploy the frontend Web service, `clinic-rest`. Define YAML files describing its deployment and service and use these to launch it in your Kubernetes cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: clinic-rest
spec:
  replicas: 1
  selector:
    matchLabels:
      app: clinic-rest
  template:
    metadata:
      labels:
        app: clinic-rest
    spec:
      restartPolicy: Always
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/arch
                operator: In
```

```
                values:
                - amd64
      containers:
      - name: clinic-rest
        image: account-id.dkr.ecr.us-east-2.amazonaws.com/clinic-rest
        imagePullPolicy: Always
```

And also, the service descriptor:

```
apiVersion: v1
kind: Service
metadata:
  name: clinic-rest
  labels:
    app: clinic-rest
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 9090
    targetPort: 8080
  selector:
    app: clinic-rest
```

You can access the web service e.g., using `clinic-rest-client`, at this URL:

`http://`*`external-ip-address`*`:9090/api`

Note that you need to specify different ports for the web application and web service, because both must be accessible outside the cluster at the same external IP address. This was not necessary for the microservice because it was never exposed outside the cluster (except via the randomly assigned nodeport). Within the cluster, it is accessed on its own pod via a DNS lookup that resolves to the pod IP address. The combination of pod IP address and port are unique within the cluster.

You can use a Web browser to test some of the query operations for the web application, and use the REST client from a previous assignment to upload data to the web service. Use `kubectl logs` to show via logs that both the web application and the web service are using the backend microservice to access the domain. Direct the log output to files and submit those files with your submission.

Once your apps are deployed, you can get a list of the running pods with:

```
$ kubectl get pods
```

You can get the details of a particular pod with:

```
$ kubectl describe pod pod-name
```

And you can see the logs at a pod with:

```
$ kubectl logs pod-name
```

Submission

For your submission, you should provide your dockerfiles (and their inputs, including WAR files) and YAML configuration files. You should also provide a video that demonstrates:

1. Showing the permissions granted to the IAM user that you have set up, in the IAM console.
2. Showing via e.g. pgAdmin or Intellij IDEA that the tables have been created on the database server in Amazon RDS, and showing the RDS web page for the running database instance. The endpoint in the RDS console should match that shown for the connection to show the tables.
3. Creating docker images for the services that you deploy, and tagging and pushing these services to repositories in Elastic Container Registry. Show these private repositories in ECR when you are done.
4. Launching your pods and services, and using `kubectl` (e.g. `kubectl describe service`) to show the endpoint information for these services. Use `kubectl` to also show the details of the pods. You should have deleted any previous instances of these pods and services before starting.
5. Use a web browser via the node port to show the WADL description for the microservice, deployed in the cluster.
6. Use the REST client CLI to invoke your frontend web service and show with the logs that it is invoking the domain microservice in the background.
7. Access the web application with a web browser to query the domain model and show with the logs that it is invoking the domain microservice in the background. Use the Web service to add patients, providers and treatments, and use the Web application to show these additions.

For your submission, make sure that your Stevens username appears in the name of the cluster that you create, as well as the name of the database. You should save the logs for each of the three pods that you create in files and submit those files with your submission. Your name must appear in all logs files. Add a log statement to microservice and Web service code, to make your name appear in the logs if it does not already.

Make sure that your name appears in each video that you submit. For example, display the contents of a file that provides your name. *Do not provide private information such as your email or cwid in the video*. Be careful of any "free" apps that you download to do the recording, there are known cases of such apps containing Trojan horses, including key loggers. **Your video must be MP4 format!**

Your submission should be uploaded via the Canvas classroom, as a zip file. Your solution should consist of a zip archive with one folder, identified by your name. This folder should contain the files and subfolders with your submission.

It is important that you provide your source code (Intellij proect with all modules), dockerfiles (and their inputs), YAML files and log files, as well as WAR and JAR files. You should also provide a video demonstrating setting up and running your services in EKS, as described above. You should also provide a completed rubric.