Devops Exercises

Please pick just 2 of the exercises below and complete them. Once you have completed your submission, place it in a github repo that you share.

We will schedule a time to review your solution with you after we have had a chance to review it internally.

AWS VPC Exercise

You have been assigned a project to create a new terraform module for creating a staging VPC in AWS. The following requirements have been given to you.

- The network supernet of 172.16.0.0/16
- The VPC should provide an endpoint for AWS Systems Manager (SSM), internally.
- The VPC should provide an internal endpoint for S3.
- The VPC should have two different availability zones, with two private subnets and two public subnets for each availability zone.
- The VPC should contain appropriate routing tables.
- The VPC should include appropriate NAT gateways and an internet gateway.

Create module code to fulfill this request utilizing terragrunt/terraform. Show how you would use this module code to bring up the given architecture.

What I understood from the exercise related to AWS VPC Exercise

- A **staging VPC** on **172.16.0.0/16**
- **Two AZs**, and **two private + two public subnets per AZ** (→ 4 private + 4 public)
- **Routing tables**, **IGW**, **NAT gateways** (one per AZ)
- **Private endpoints**: **SSM (interface)** and **S3 (gateway)**
- Show **module code** and **how to use it**; optionally via **Terragrunt**; and show **plan only for VPC** resources.

Structure of the files and folder in the mono repository

1. **`modules/vpc`**
    - Creates the VPC (`cidr = 172.16.0.0/16`) with two AZs.
    - 4 private + 4 public subnets (you pass explicit CIDRs).
    - **IGW** for public subnets.
    - **NAT gateways per AZ** and private route tables with default route via NAT.
    - **VPC endpoints submodule**:

        - `s3` **gateway** endpoint associated to the route tables for internal S3.
        - `ssm` **interface** endpoint (in subnets) with private DNS for internal SSM.

2. **`modules/s3_bucket_state`**
    - Bootstraps (first run) the **S3 state bucket** and **DynamoDB lock table**.
    - Has **creation toggles**: `create_bucket` / `create_lock_table`.
    - On later runs we flip them to `false` and feed back the existing names.

3. **`modules/iam_tf_policies`**
    - Creates/uses IAM policies the pipeline needs:
        - RW to the state bucket + lock table.
        - Minimal VPC "apply" permissions (Describe + CreateVpc/CreateTags etc.).
    - Skips creation if an **existing policy ARN** is detected/passed.

4. **`modules/github_oidc`**
    - Uses an **existing** GitHub OIDC provider ARN (or can create one),
    - Creates an **assumable role** for the workflow and **attaches** the policies from (3).

Putting all the code centralized in the mono repository

`The root file is main.tf` calling the modules and passes inputs:

```
module "vpc" {                        # ← the VPC with subnets, NAT, routes, 
endpoints
  source = "./modules/vpc"
  name   = var.vpc_name
  cidr   = "172.16.0.0/16"
  azs    = ["eu-west-1a", "eu-west-1b"]

  private_subnets = 
["172.16.0.0/20","172.16.16.0/20","172.16.32.0/20","172.16.48.0/20"]
  public_subnets  = 
["172.16.64.0/20","172.16.80.0/20","172.16.96.0/20","172.16.112.0/20"]

  enable_nat_gateway     = true
  one_nat_gateway_per_az = true
  single_nat_gateway     = false
}

module "s3_bucket_state_oidc" {       # ← state bucket + lock table (first run 
only)
  source              = "./modules/s3_bucket_state"
  bucket_prefix_name  = var.bucket_prefix_name
  lock_table          = var.lock_table
  create_bucket       = var.create_bucket
  create_lock_table   = var.create_lock_table
  state_key           = "envs/${var.environment}/terraform.tfstate"
  existing_bucket_name = var.existing_bucket_name
```

```
  existing_lock_table  = var.existing_lock_table
}

module "iam_tf_policies" {              # ← policies used by the OIDC role
  source          = "./modules/iam_tf_policies"
  bucket_name     = module.s3_bucket_state_oidc.s3_bucket_id
  lock_table_name = module.s3_bucket_state_oidc.lock_table_name
  region          = var.region
  depends_on      = [module.s3_bucket_state_oidc]
}

module "github_oidc" {                   # ← OIDC role that attaches those policies
  source             = "./modules/github_oidc"
  create_oidc_provider = false
  oidc_provider_arn    = local.existing_provider_arn
  create_oidc_role     = true
  repositories = var.repository_list
  oidc_role_attach_policies = [
    module.iam_tf_policies.tf_backend_rw_policy_arn,
    module.iam_tf_policies.tf_vpc_apply_policy_arn
  ]
  depends_on = [module.iam_tf_policies]
}
```

Short summary

- The module vpc creates the VPC with subnets, NAT, routes, endpoints.
- The module iam_tf_policies creates the iam tf polcicy and identify required for github oidc
- The module github_oidc is creating the OIDC role that attahes the polcies in the module iam_tf_polcies

The dependencies are as follows

1) "iam_tf_policies" module depending on "s3_bucket_state_oidc"
2) "github_oidc" depending on the "iam_tf_policies" module.

More information about the plan can be seen in this link.

How I split in CI/CD flow

- **Bootstrap job** (long-lived AWS keys just once):
  1. init locally (backend disabled),
  2. create **state S3 + DDB**, **policies**, **OIDC role** (module targets),
  3. export bucket/table/role outputs.
- **Deploy job** (assume OIDC role):
  1. flip `create_bucket=false` and `create_lock_table=false`,

2. migrate TF backend to S3 + DDB,
3. plan/apply the whole stack.

How I plan the exercise

Three practical ways (pick one):

- **Simple (demo)**: target the module

  ```
  terraform plan -target=module.vpc -input=false -no-color
  ```

  (Good for showcasing; do not use `-target` routinely for full lifecycles.It was used for cost reasons)

- **Clean separation**: run the VPC as its **own stack** (Terragrunt)

  ```
  live/staging/vpc/terragrunt.hcl → source = ../../..//modules/vpc
  ```

  Then:

  ```
  cd live/staging/vpc
  terragrunt plan
  ```

- I set inputs so non-VPC modules don't create (e.g. `create_bucket=false`, `create_lock_table=false`) and just `terraform plan to create` VPC resources.

How the requirements for this exercise are met

- CIDR and subnet layout match **172.16.0.0/16** having each **2 AZs × (2 private + 2 public)**.
- **IGW** for public subnets;
- **NAT per AZ** for private egress.
- **Route tables** were created and associated appropriately for public and the private subnets.
- **Endpoints**:
  - **S3**: **Gateway** endpoint for internal S3 access.
  - **SSM**: **Interface** endpoint with private DNS for internal SSM.

Deploying An Application
Helm
- Create a simple templated helm chart that loads up an nginx server and an appropriate ingress. (assume either an nginx or alb controller is just fine)
- Create a values.yml file that fulfills the values for this helm chart.
- Show a command that dumps the template generated.

Environment used was vagrant to deploy a local kubernetes cluster by using vagrant up for the files and folder in [this repository]
I followed these steps to complete the exercise

1. `helm repo update`

  * This refreshes my locally cached index of all Helm repositories that I have added from my side.
  * It returns the Output shows the repos that were refreshed (`kor`, `ingress-nginx`). You now see the newest chart versions.

2. `helm pull ingress-nginx/ingress-nginx --untar`

   This command downloads the ingress-nginx chart and it extracted it into folder `~/ingress-nginx/` so I can edit `values.yaml` and templates locally.

3. I create `metallb-pool.yaml`
4. I launched on the first attempt `kubectl apply -f metallb-pool.yaml`

  * This return the error error: `no matches for kind "IPAddressPool" ... ensure CRDs are installed first`
  * It means that tje MetalLB’s **CRDs** (custom resource definitions) and the webhook weren’t in the cluster yet,
  * At that stage, Kubernetes service did not detect the `IPAddressPool` and `L2Advertisement` were.

5. Install MetalLB via Helm
```bash
kubectl create ns metallb-system
helm repo add metallb https://metallb.github.io/metallb
helm repo update
helm install metallb metallb/metallb -n metallb-system
```
  * The command used above was used to install the following services:
    ** metallb-controller + metallb-speaker (the brains + per-node announcers)
    ** CRDs and webhook for validating MetalLB resources

6. Immediately applying the pool again → `connect: connection refused` to webhook
  * This is a condtion used to validate if the Webhook was created, but it was not ready yet.
  * Then, I ran:
```bash
helm upgrade --install metallb metallb/metallb \
  -n metallb-system --create-namespace --wait --timeout 3m
```

* `--wait` made Helm hold until the controller/webhook were up and serving.

7. Verifying MetalLB readiness

```bash
kubectl -n metallb-system get pods
kubectl -n metallb-system get svc metallb-webhook-service
kubectl -n metallb-system get endpoints metallb-webhook-service
kubectl -n metallb-system logs deploy/metallb-controller
```
 This was the output received
 * I saw the controller with teh status Running amd the webhook service were having an endpoint (port 9443)
 * The logs showed webhooks enabled which it means that it worked

8. I applied again `metallb-pool.yaml` on my second attempt
```bash
# metallb-pool.yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: vagrant-pool
  namespace: metallb-system
spec:
  addresses:
    - 192.168.56.240-192.168.56.250
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: vagrant-l2
  namespace: metallb-system
spec:
  ipAddressPools:
    - vagrant-pool
```

9. Now, it worked the `ipaddresspool.metallb.io/vagrant-pool created` and `l2advertisement because they were created.

10. I edited `values.yaml` for ingress-nginx to installfrom the local chart

```bash
cd ~/ingress-nginx
helm upgrade --install my-ingress . \
  -n ingress-nginx --create-namespace -f values.yaml --wait
```

```
vagrant@controlplane:~/ingress-nginx$ helm upgrade --install my-ingress . -n ingress-nginx --create-namespace -f values.yaml --wait
Release "my-ingress" does not exist. Installing it now.
NAME: my-ingress
LAST DEPLOYED: Sun Oct 19 23:25:11 2025
NAMESPACE: ingress-nginx
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The ingress-nginx controller has been installed.
It may take a few minutes for the load balancer IP to be available.
You can watch the status by running 'kubectl get service --namespace ingress-nginx my-ingress-ingress-nginx-controller --output wide --watch'

An example Ingress that makes use of the controller:
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    name: example
    namespace: foo
  spec:
    ingressClassName: nginx
    rules:
      - host: www.example.com
        http:
          paths:
            - pathType: Prefix
              backend:
                service:
                  name: exampleService
                  port:
                    number: 80
              path: /
    # This section is only required if TLS is to be enabled for the Ingress
    tls:
      - hosts:
        - www.example.com
        secretName: example-tls

If TLS is enabled for the Ingress, a Secret containing the certificate and key must also be provided:

  apiVersion: v1
  kind: Secret
  metadata:
    name: example-tls
```

11. I was watching the Service to check if it was running

```bash
kubectl -n ingress-nginx get svc my-ingress-ingress-nginx-controller --watch
```

```
vagrant@controlplane:~/ingress-nginx$ kubectl get service --namespace ingress-nginx my-ingress-ingress-nginx-controller --output wide --watch
NAME                                  TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)                      AGE     SELECTOR
my-ingress-ingress-nginx-controller   LoadBalancer   172.17.33.105  192.168.56.240   80:31960/TCP,443:32311/TCP   4m18s   app.kubernetes.io/component=controller,app.kubernetes.io/insta
nce=my-ingress,app.kubernetes.io/name=ingress-nginx
```

12. I saw `TYPE=LoadBalancer` with **EXTERNAL-IP 192.168.56.240** — MetalLB assigned the first address from your pool which seems that it worked.

Why those specific errors happened:
* CRDs were not found.
  That is why it required to apply CRs (IPAddressPool/L2Advertisement) before MetalLB installed its CRDs → Kubernetes does not recognize those kinds yet.
* Webhook connect refused.
  The ValidatingWebhookConfiguration existed, but the backing webhook server was not serving (pod still initializing). Using `helm ... --wait` and giving it a moment fixed it.
* I edited values.yaml to ensure that i was in the right path containing the Helm files relative to folder with all helm templates and charts .
* I switched into the chart folder to apply successfully the manifest files.

13. I verified the final state

```bash
# MetalLB objects
kubectl -n metallb-system get ipaddresspools,l2advertisements

~~~
hello from ingress
vagrant@controlplane:~/ingress-nginx$ kubectl -n metallb-system get ipaddresspools,l2advertisements
NAME                                   AUTO ASSIGN   AVOID BUGGY IPS   ADDRESSES
ipaddresspool.metallb.io/vagrant-pool   true          false             ["192.168.56.240-192.168.56.250"]

NAME                                   IPADDRESSPOOLS     IPADDRESSPOOL SELECTORS   INTERFACES
l2advertisement.metallb.io/vagrant-l2   ["vagrant-pool"]
~~~

# Ingress-NGINX service has an external IP from your pool
kubectl -n ingress-nginx get svc my-ingress-ingress-nginx-controller -o wide

~~~
vagrant@controlplane:~/ingress-nginx$ kubectl -n ingress-nginx get svc my-ingress-ingress-nginx-controller -o wide
NAME                                  TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)                      AGE   SELECTOR
my-ingress-ingress-nginx-controller   LoadBalancer   172.17.33.105   192.168.56.240   80:31960/TCP,443:32311/TCP   35m   app.kubernetes.io/component=controller,app.kubernetes.io/insta
e=my-ingress,app.kubernetes.io/name=ingress-nginx
~~~

# IngressClass is default (if you set default: true)
```bash
kubectl get ingressclass nginx -o yaml
```

~~~
vagrant@controlplane:~/ingress-nginx$ kubectl get ingressclass nginx -o yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  annotations:
    meta.helm.sh/release-name: my-ingress
    meta.helm.sh/release-namespace: ingress-nginx
  creationTimestamp: "2025-10-19T23:25:20Z"
  generation: 1
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: my-ingress
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
    app.kubernetes.io/version: 1.13.3
    helm.sh/chart: ingress-nginx-4.13.3
  name: nginx
  resourceVersion: "4479"
  uid: f3286580-ae92-4ea0-a257-5fde43a0e4f7
spec:
  controller: k8s.io/ingress-nginx
~~~

14. Quick smoke test (optional but recommended)

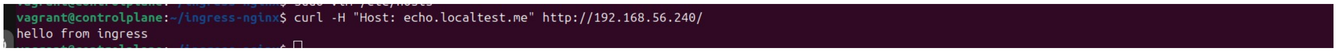
 # Deploy a tiny echo app + Service + Ingress:
```yaml
# echo-app.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echo
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels: { app: echo }
  template:
    metadata:
      labels: { app: echo }
    spec:

      containers:
        - name: echo
          image: hashicorp/http-echo
          args: ["-text=hello from ingress"]
          ports:
            - containerPort: 5678
---
apiVersion: v1
kind: Service
metadata:
  name: echo
  namespace: default
spec:
  selector: { app: echo }
  ports:
    - port: 80
      targetPort: 5678
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: echo
  namespace: default
spec:
  ingressClassName: nginx
  rules:
    - host: echo.localtest.me   # resolves to 127.0.0.1; replace if you prefer
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: echo
                port:
                  number: 80
```

```bash
kubectl apply -f echo-app.yaml
```

15. Test it. Since your LB IP is `192.168.56.240`, either:

* Add a hosts entry: `192.168.56.240  echo.localtest.me`, then:

  ```bash
  curl -H "Host: echo.localtest.me" http://192.168.56.240/




* Change the Ingress host to a name you control and map it to `192.168.56.240` in `/etc/hosts`.

I received `hello from ingress`.

# My `values.yaml` (MetalLB-friendly) recap

```yaml
controller:
  ingressClassResource:
    name: nginx
    default: true
  service:
    type: LoadBalancer
    annotations:
      metallb.universe.tf/address-pool: vagrant-pool
      # metallb.universe.tf/loadBalancerIPs: 192.168.56.241  # optional pin
    externalTrafficPolicy: Local
  config:
    proxy-body-size: "64m"
    enable-brotli: "true"
```

Summary
The whole flow was the following
* Update repos to pull and edit the chart locally
* install MetalLB (wait for webhook)
* Apply pool/L2Ad to install ingress-nginx with `LoadBalancer`
* Get an external IP
* Test with a simple app.