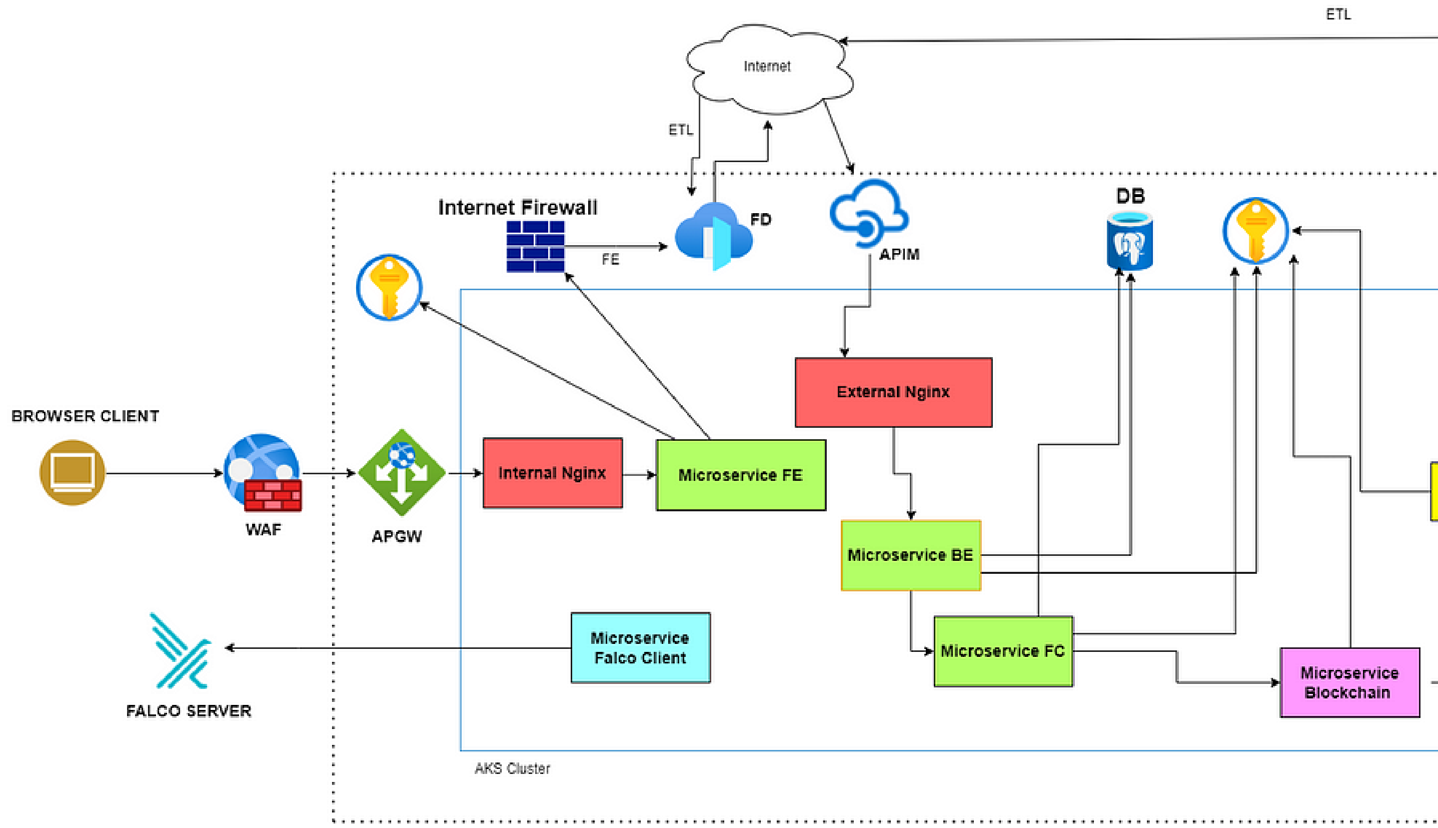
# Deploy a microservice application on AKS cluster and access it using public internet

## STEP 1 : Understanding the microservice application and its network routing flows

Focus on this step and make sure you understand your application very well . If you know and understand routing flows then only it is possible to figure out the critical entry points where Network policy restrictions are required.

So let’s understand the application in our case ..

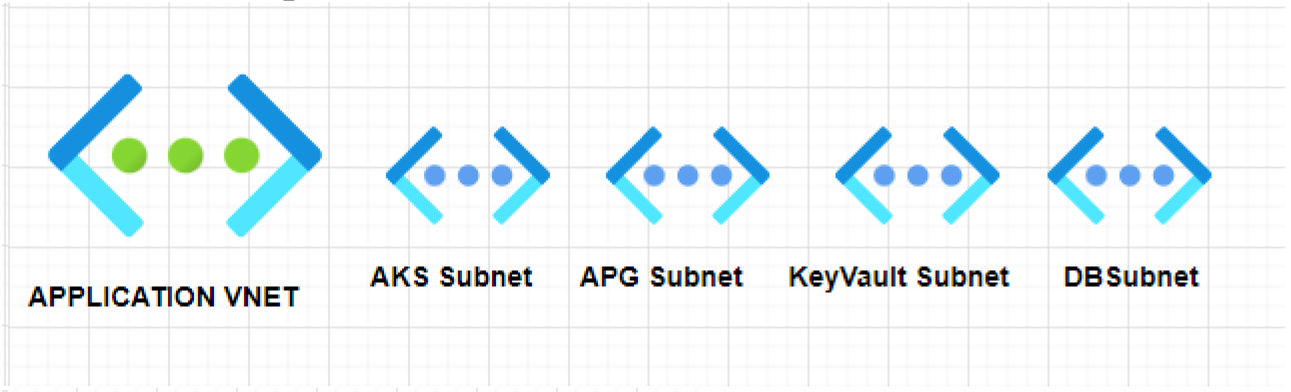


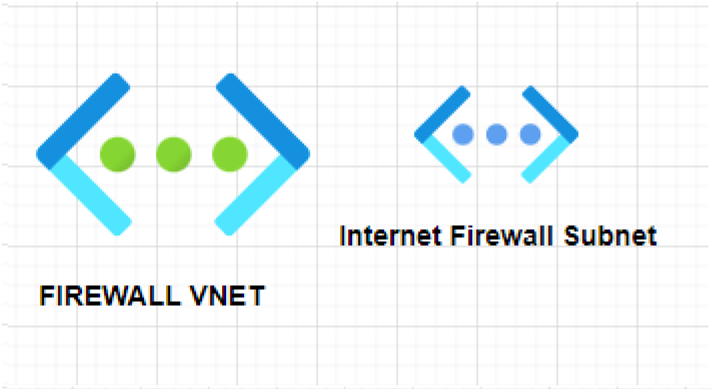
Routing flow for the microservice application

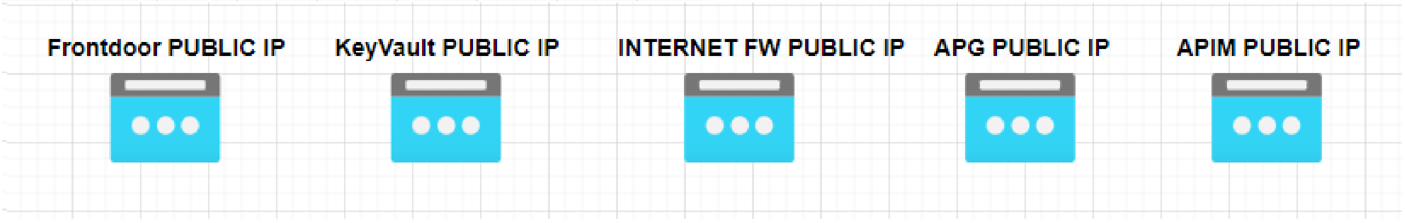
1.Namespace Segregation

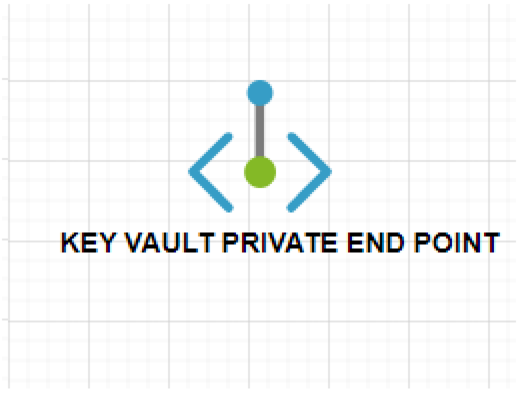
- Green namespace is for application
- Yellow namespace is for Cron Jobs
- Purple namespace is for Blockchain
- Blue namespace is for Falco Client Pods
- Red namespace is for Nginx

2.Vnet , Public Ip & Private link









3.Traffic Flow

- Browser client to Frontend microservice

***BROWSER CLIENT → WAF → APGW → MICROSERVICE FRONTEND***

- Browser client connects to the frontend microservice hosted behind a Web application Firewall(WAF).
- WAF connects to Application gateway( *used for custom security rules, centralized management of frontend services , SSL offloading , exposing frontend services to Internet in a secure way*) on its **public IP** .
- Application gateway internally routes traffic to internal nginx ingress on private subnet. **[Subnet-Subnet]**
- Internal ingress then directs the traffic to frontend Pod. **[Internal Svc- Internal Svc]**

- Frontend microservice to Backend Microservice

***MICROSERVICE FRONTEND → INTERNET FIREWALL → FRONTDOOR → APIM → MICROSERVICE BACKEND***

- Frontend Pod communicates with the Backend pod via its public hosted URL.
- The Backend public URL is hosted behind a Frontdoor which has WAF filtering and other security controls.
- Every internet traffic is routed to the Internet firewall via Vnet-to-Vnet communication , thus traffic to Frontdoor from frontend pod also goes via Internet firewall. **[Vnet-Vnet]**
- From Frontdoor traffic is routed to API Management service (APIM) for authentication and authorization. **[Public IP — Public IP]**
- APIM then routes the traffic to external Nginx ingress which is used to expose the Backend microservice. **[Public IP — External Svc]**
- Backend Microservice to FC Microservice

***MICROSERVICE BACKEND → MICROSERVICE FC***

- Backend Pod communicates with FC pod via internal service-to-service routing. **[Internal Svc- Internal Svc]**
- FC Microservice to Blockchain Microservice

***MICROSERVICE FC → MICROSERVICE BLOCKCHAIN***

- FC Pod communicates with blockchain pod via internal service-to-service routing. **[Internal Svc- Internal Svc]**
- ETL to MICROSERVICE BACKEND

***MICROSERVICE ETL → INTERNET FIREWALL → FRONTDOOR → APIM → MICROSERVICE BACKEND***

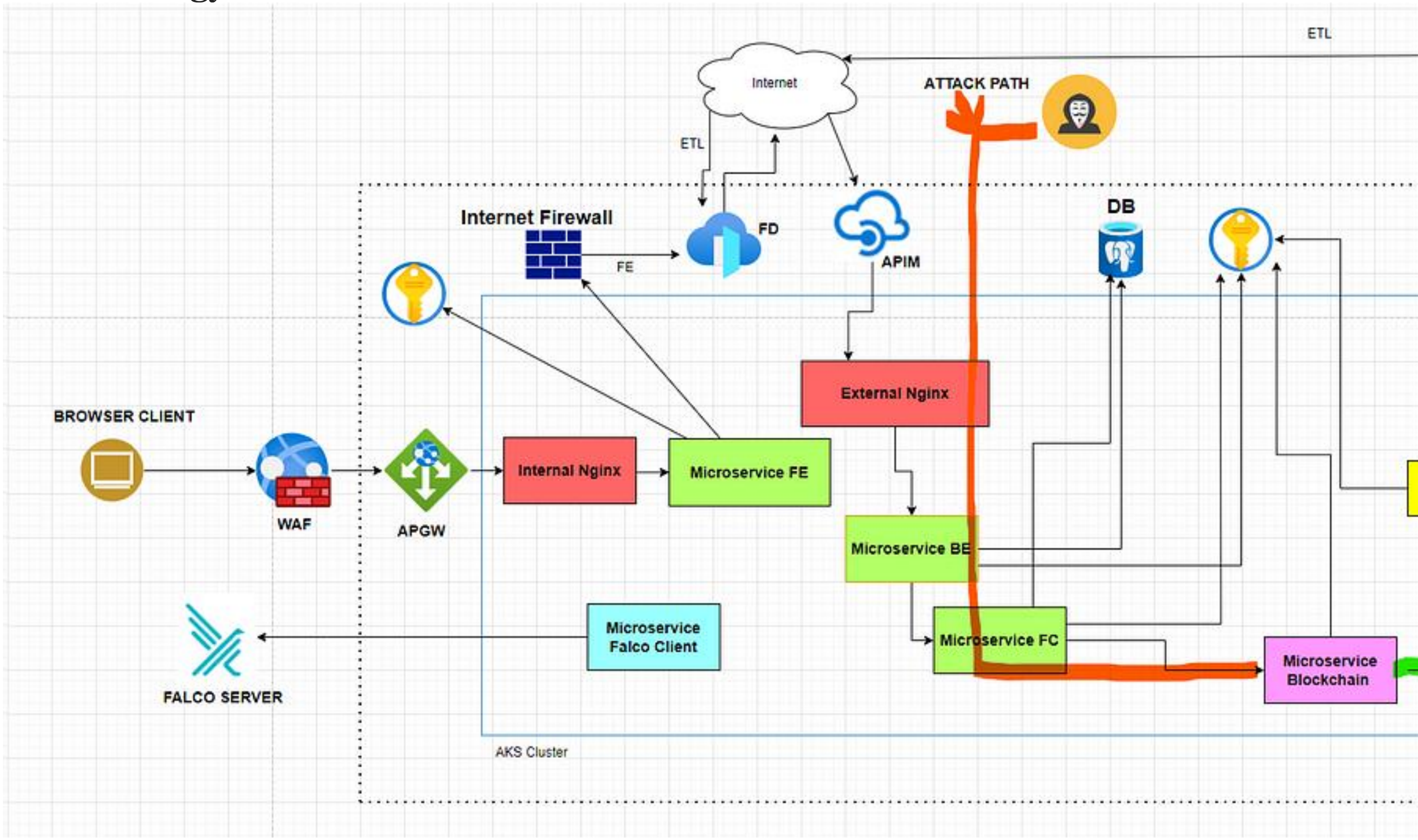
- ETL pod communicates with backend microservice via its public hosted URL.
- This routing is similar to the one between frontend and backend microservice.
- Microservice to cloud resources .
- Frontend, backend, ETL, FC and blockchain pods communicates with Key Vault on private link i.e. subnet-to-subnet Communication. **[Subnet-Subnet]**
- Backend, ETL and FC pods communicates with DB via subnet-to-subnet Communication. **[Subnet-Subnet]**

## STEP 2 : Determining where traffic needs to be controlled

This is the most trickiest part. If you get this wrong then Network policies won’t be effective in reducing the attack surface .

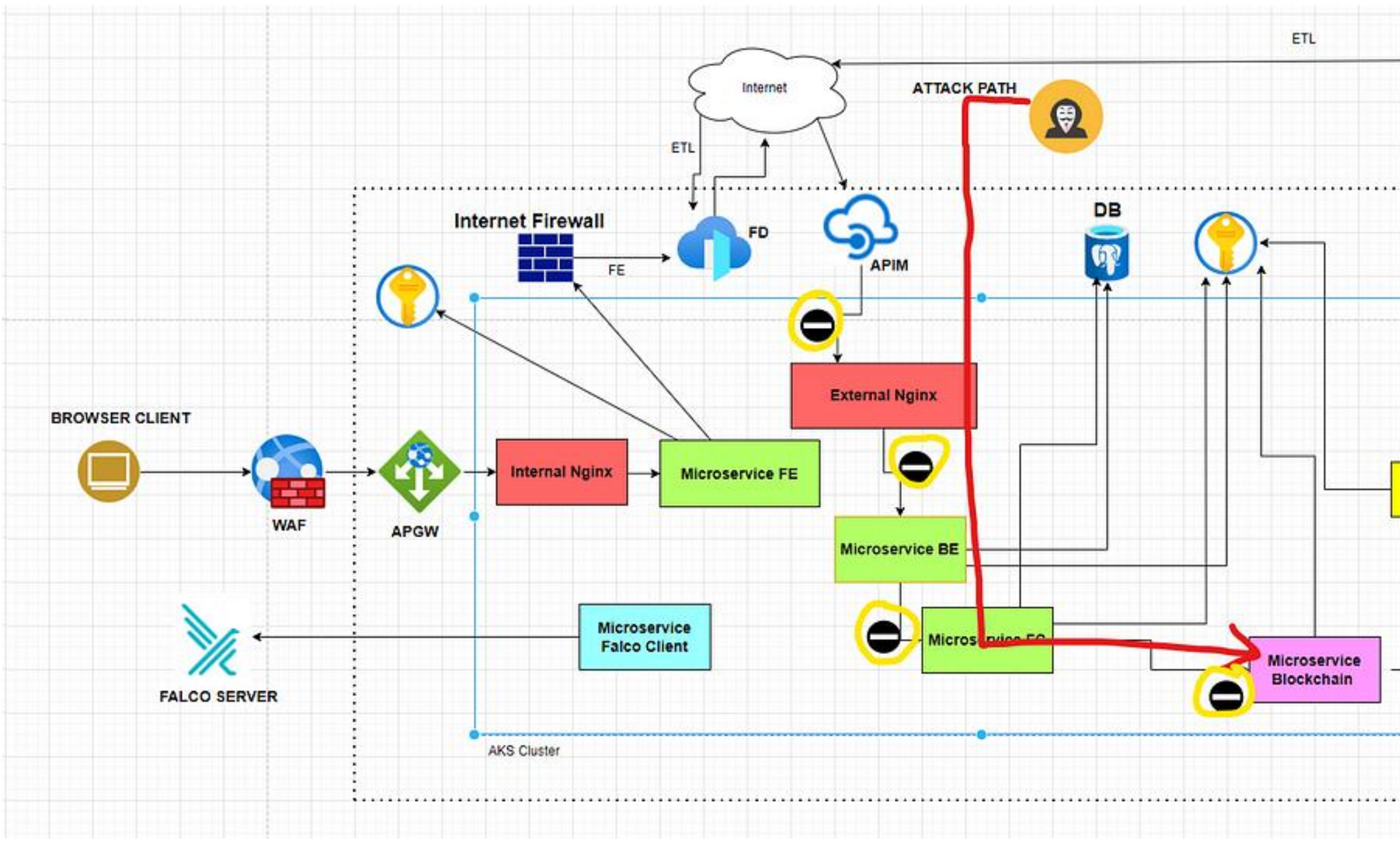
The ‘One shot Kill approach’ for getting this right is Security architecture design by Defence in Depth methodology .

Assuming that you have read the above article , we will apply the same methodology in our case .



Attack path towards the critical assets from the open attack surface

- Microservice backend is the only one exposed to internet without any WAF filtering and thus the only attack surface .
- This can pave way ( highlighted path) to the critical asset
- Thus , we need to add a number of Line of defense along this attack path as discussed here [Security architecture design by Defence in Depth methodology]()
- However , we will only consider the network policies for this article which is relevant to our discussion



Attack path highlighted with Network Policies as Line of Defense

In priority order ..

- For blocking the attack path and reducing the attack surface — Add network policies for the Microservices along the attack path to critical asset to accept and initiate only legit traffic.

Though the remaining microservices does not paves way for an attack surface , in case of an attacker having initial footsteps in the cluster by some other means a strict Network policy can block other stages of attack .

- To block lateral movement , command & control , exfiltration — Add network policies for all the remaining Microservices to accept and initiate only legit traffic.

**STEP 3 : Choosing the Plugin**

There are a number of options for choosing the plugin. Depending upon your environment , compatibility and ease of use you may pick one.

Refer for an overview of the types and their Pros & Cons .

In this case we are choosing *Azure CNI*

IMP NOTES BEFORE WE START WRITING POLICIES

*Allow policy is always checked first before the deny policy .*

*Network policy are stateful . If we add an ingress policy for pod A to accept traffic from pod B then there is no need to add an egress policy for pod A to communicate back with pod B .*

## STEP 4: Create default deny policy

The most safest approach is to deny all traffic ( ingress & egress ) for all pods . Then explicitly allow only legit traffic . This will help you achieve granular level control on the traffic flow.

The following Yaml can be executed in all namespaces

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-traffic-all-ns
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
kubectl apply -f deny-all-traffic-all-ns.yaml
{{add_all_namespaces_to_be_controlled}}
```

## STEP 5: Whitelist control plane traffic

kube-system namespace is by default present in AKS. It has a number of pods which perform the management task for the cluster . We can trust this traffic and should be allowed for all pods bidirectionally*(ingress & egress)*

The following Yaml can be executed in all namespaces

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-kube-system-traffic-all-ns
spec:
  podSelector: {}
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: "kube-system"
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: "kube-system"
  policyTypes:
  - Egress
  - Ingress
kubectl apply -f allow-kube-system-traffic-all-ns.yaml
{add_all_namespaces_to_be_controlled}
```

## STEP 6: Determining the Pods which needs all allow policy

There will be scenarios where some microservices needs to connect (ingress/egress) to Internet URLs which have a dynamic Public IP.

We cannot determine these dynamic Public IPs and thus cannot create a Network policy for a particular IP Block

So, add a default allow policy which will permit all traffic. *{ A question may arise : Is this not a Security Loophole ? Let's discuss this in Step 8}*

In our case we have 2 microservice which need all allow policy

- Frontend & ETL- egress all allow for communicating with backend on public IP

The following policy will allow front end pod to initiate traffic to any host

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-webapp-legit-network-policy
  namespace: green
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: webapp
  policyTypes:
    - Egress
  egress:
    - {}
kubectl apply -f allow-webapp-legit-network-policy.yaml -n green
```

The following policy will allow etl pods to initiate traffic to any host

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-cronjob-legit-network-policy
  namespace: yellow
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: etl
  policyTypes:
    - Egress
  egress:
    - {}
kubectl apply -f allow-cronjob-legit-network-policy.yaml -n yellow
```

## STEP 7: Create custom rules to allow only legit traffic

Now we will start adding only legit traffic for each pod. Let’s start with the priority order determined in Step 2.

1] Nginx pod

If you observe closely in Step 1 , nginx ingress are configured for Frontend and Backend Microservices.

*Required Ingress Traffic* : From Application gateway Subnet & APIM Public IP

*Required Egress Traffic* : To Frontend and Backend pods

The following policy will do the trick

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-nginx-legit-network-policy
  namespace: red
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: nginx    #for all nginx pods
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 10.0.1.0/24     #from APGW Subnet
    - ipBlock:
        cidr: 115.13.11.18/24     #from APIM public IP
  egress:
  - to:
    - namespaceSelector:
          matchLabels:
              kubernetes.io/metadata.name: "green"   #namespace to be checked
for below podselector policy
      podSelector:
        matchLabels:
          app.kubernetes.io/name: webapp    #to frontend pod
          app.kubernetes.io/name: backend   #to backend pod
kubectl apply -f allow-nginx-legit-network-policy.yaml -n red
```

2] Backend Pod

*Required Ingress Traffic* : From nginx pods

*Required Egress Traffic* : to FC pod , DB , Key vault

Add the following policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-backend-legit-network-policy
  namespace: green
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: backend    #for all backend pods
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: "red"   #namespace to be checked for
below podselector policy
      podSelector:
        matchLabels:
          app.kubernetes.io/name: nginx    #from nginx pod
  egress:
  - to:
    - podSelector:
        matchLabels:
          app.kubernetes.io/name: FC    #to FC pod
    - ipBlock:
        cidr: 10.0.2.0/24         #to DB Subnet
    - ipBlock:
        cidr: 10.0.3.0/24     #to KeyVault Subnet
kubectl apply -f allow-backend-legit-network-policy.yaml -n green
```

3] FC Pod

*Required Ingress Traffic* : From backend pods

*Required Egress Traffic* : to Blockchain pod , DB , Key vault

Add the following policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-fc-legit-network-policy
  namespace: green
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: FC    #for all FC pods
  policyTypes:
  - Ingress    `
  - Egress
  ingress:
  - from:
      podSelector:
        matchLabels:
          app.kubernetes.io/name: backend     #from backend pods
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: "purple"   #namespace to be checked
for below podselector policy
      podSelector:
        matchLabels:
          app.kubernetes.io/name: blockchain    #to blockchain pod
    - ipBlock:
        cidr: 10.0.2.0/24         #to DB Subnet
    - ipBlock:
        cidr: 10.0.3.0/24     #to KeyVault Subnet
kubectl apply -f allow-fc-legit-network-policy.yaml -n green
```

## 4] Blockchain Pod

*Required Ingress Traffic* : From FC pods

*Required Egress Traffic* : Key vault

Add the following policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-1-legit-network-policy
  namespace: green
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: blockchain    #for all blockchain pods
  policyTypes:
  - Ingress    `
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: "green"   #namespace to be checked for
below podselector policy
      podSelector:
        matchLabels:
          app.kubernetes.io/name: FC     #from FC pods
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.3.0/24     #to KeyVault Subnet
kubectl apply -f allow-blockchain-legit-network-policy.yaml -n purple
```

5] Frontend Pod

*Required Ingress Traffic* : From nginx pods

*Required Egress Traffic* : allow all as discussed in Step 6

We can enhance the same policy created for frontend in STEP 6

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-webapp-legit-network-policy
  namespace: green
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: webapp
  policyTypes:
    - Egress
    - Ingress
  egress:
    - {}
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: "red"   #namespace to be checked for
below podselector policy
      podSelector:
        matchLabels:
          app.kubernetes.io/name: nginx    #from nginx pod
```

```
kubectl apply -f allow-webapp-legit-network-policy.yaml -n green
```

6] ETL

*Required Ingress Traffic* : none

*Required Egress Traffic* : allow all as discussed in Step 6

We can use the same policy created for etl in STEP 6

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-cronjob-legit-network-policy
  namespace: yellow
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: etl
  policyTypes:
    - Egress
  egress:
    - {}
kubectl apply -f allow-cronjob-legit-network-policy.yaml -n yellow
```

**STEP 8: Dealing with all allow Ingress/Egress policies**

Adding such policies actually does not makes difference. It presence is as good as its absence . So how do we fill this loophole ?

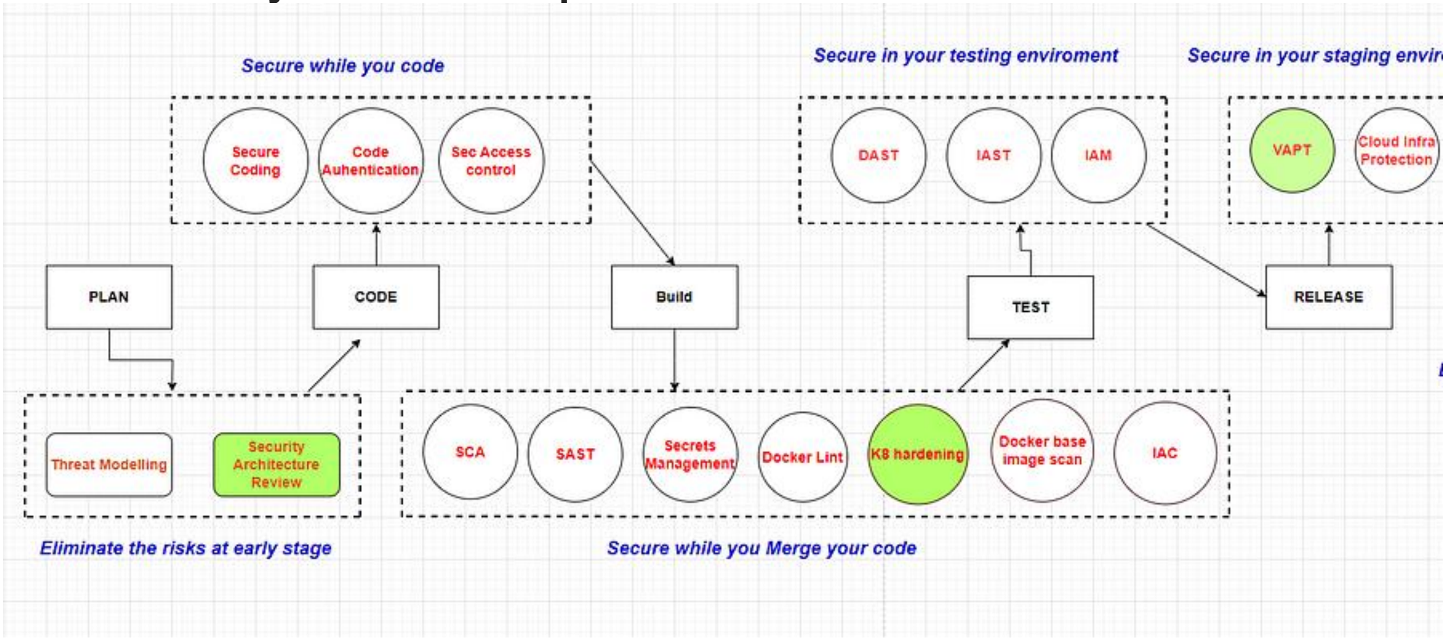
We discussed previously about [Defense in Depth and DevSecOps ](). Using the same approach we will be having 2 Line of defense :

1. Access control list at ***Internet firewall*** for egress traffic
2. Security Logging and monitoring tool for ***suspicious events and traffic***.

Now , focus on our *all allow* policies .

- Frontend and ETL - All allowed Egress traffic can be controlled at Internet firewall with a Layer 7 access control list to whitelist ( source: *AKS subnet*. destination: *Frontdoor URL*)
- Also we have a monitoring tool *Falco* installed for flagging suspicious events and traffic

## Network Policy and DevSecOps:



DevSecOps Model highlighted with security mechanisms where Network policies can be implemented/enforced

1. During security architecture design determine where Network policies are required and implement them.

2. If you miss doing that , while you push your code to your repo a Kubernetes security shift left tool ( like Kubescape ) can scan the deployments for security misconfigurations like missing network policy for a Pod

3. If the above two are missed a *cluster VAPT* would highlight the missing Network policies for the running pods.