**devops**

**Forward**

We need to deploy an SPA to our account in the cloud.

The app is:

1. JS based
2. Using rds postgres/mysql
3. Using redis for session management
4. Configuration by env variables
5. Code on gitlab
6. Internet Facing

**Our job:**

1. Build & deploy to cloud K8s, CI/CD.
2. Set up internal Redis
3. Make HA as possible

**Deliverables:**

* Block diagram of the solution
* Detailed design of the k8s deployment
* Defend your decisions in front of the team.

Response:

**Note**

Everything in the response below – is taken and based on my work experience from the past, and on current solutions within my current organization. As the saying goes – “There are many ways to skin a cat “– and therefore this is probably not the only solution.

**Assumptions:**

* Coding of App done on prem.
* Use of GitLab (given) – Note: Never used – assume it is similar in operation to GIT
* Jenkins (Master and Worker) on prem for building CICD pipeline.
* Push / Merge to Master will result in the CICD pipeline being kicked off (webhook / trigger).
* App is going to be deployed as a docker container on K8s
* Have access to AWS account + permissions
* Authentication done by the APP through some sort of mechanism – not detailed here.
* 3 AZs

**Step 1: Build of VPC**

Create a VPC on AWS Account:

**Details:**

Cidr blocks:

Main:

* 10.1.0.0/16

Main Subnets:

DMZ (with auto-allocated Public Ips)

* Az1- 10.1.8.0/24
* Az2- 10.1.24.0/24
* Az3 - 10.1.40.0/24

INTERNAL:

* Az1- 10.1.0.0 /24
* Az2- 10.1.16.0/24
* Ac3- 10.1.32.0/24

[

*If we want to put EKS nodes in another network – previous work has done in the past due to lack of address space – but not 100% necessary*

*Additional VPC CIDRs for EKS: 10.10.0.0/16*

*Additional EKS Subnets:*

* *Az1- 10.10.0.0/18*
* *Az2- 10.10.64.0/18*
* *Az3- 10.10.128.0/18*

]

\*\* Route Tables need to be set up to – (after set up of NAT GW/IGW )

**Step 2 – Additional AWS Resources**

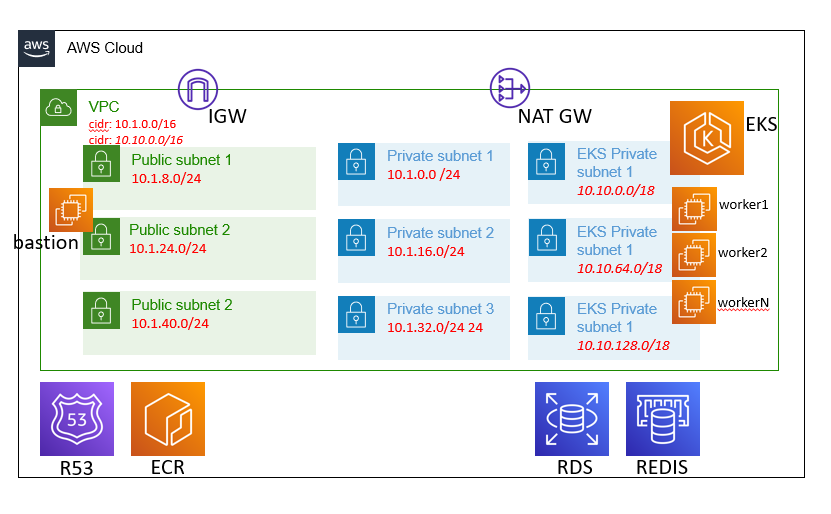
* NAT GW or NAT Instance (cheaper for lower volume traffic)
* Internet GW
* Keypair
* Route 53 – domain + records
* ECR – For holding images
* ACM – For generating Certificates (unless you have your own cert to import)
* RDS – MySQL
* Redis
* Load Balancers (created by EKS ingress pods, and some more perhaps manual.)
* EC2 – bastion host
* IAM – various roles and keys ( e.g. for ECR Access )

**Bastion Host:**

* Access to VPC Resources is via the Bastion Host (on DMZ network)
* Bastion given IAM role – with required abilities.
* Secured by Security Groups.
* Access to Bastion based on SSH keys – either updated with personal keys or the private key from the keypair given to person.

**Things to Consider:**

* Autoscalers – AWS Internal
* Spot IO – Good for managing Spots. ( from experience ), other autoscalers.
* Logging – how will the logging be maintained.
* Metrics – how will we measure the system – and alert on system outages.



**CICD Flow:**

CICD – to be managed by an on prem Jenkins (could also be in AWS)

Developer will be coding locally and storing the information on GIT (lab).

Upon merge to master – the pull request (PR) – will trigger an event to Jenkins.

Jenkins set up to receive the event – and a pipeline created linked to the GIT, will receive that PR.

This pipeline – will build the app into a docker image.

The pipeline will login into the AWS ECR using the key and secret available to it.

The pipeline will continue to tag and push the image to the ECR in AWS.

At this point the pipeline will trigger the addition of the APP as a POD to the K8s platform.

Theoretically, the POD should be deployed as a canary service – i.e., not live.

Automated testing should then occur on the APP.

Based on the automated testing, if the tests pass, the APP will either be redeployed to the K8s platform as the active pod – or the canary service will be made active.

If the automated testing fails – we require a rollback – at this point – the current ACTIVE version of the POD (assumed to be good) – will be made as the canary version also (so that we don’t have a bad POD on the system).

(Another method for deploying on the system is ARGO Rollouts – personally I don’t have experience with this – but from what I hear, it can manage the canary and load balancing between versions).

**Example Deployment YAML:**

(Based on code from our current system)

[ missing – set up of the ingress controller – was provided by infra team in current system,   
what I understand how it works explained in next section. ]

**Block Diagram of Deployment**

Pod

Pod

Pod

Service: type Cluster IP

main

Pod

Pod

Pod

Service: type Cluster IP

canary

NLB

ingress

**Yaml deployment:**

apiVersion: apps/v1

kind: Deployment

metadata:

name: myapp-v1.0

labels:

functionGroup: myapp

service: myapp-service

deployedby: user-x

version: "1.0"

spec:

selector:

matchLabels:

functionGroup: myapp

service: myapp-service

version: "1.0"

deploymentTag: "1.0"

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 0

maxSurge: 1

template:

metadata:

annotations:

tags: myapp-service

labels:

functionGroup: myapp

service: myapp-service

version: "1.0"

deploymentTag: "1.0"

spec:

containers:

- name: myapp

image: 12345678901234.dkr.ecr.eu-west-1.amazonaws.com/myapp/myapp:1.0

imagePullPolicy: IfNotPresent

env:

- name: POD\_NAME

valueFrom:

fieldRef:

fieldPath: metadata.name

- name: NODE\_ENV

value: production

ports:

- name: server-port

containerPort: 80

protocol: TCP

livenessProbe:

httpGet:

path: /manage/alive

port: server-port

initialDelaySeconds: 3

timeoutSeconds: 5

periodSeconds: 10

readinessProbe:

httpGet:

path: /manage/health

port: server-port

initialDelaySeconds: 5

timeoutSeconds: 5

periodSeconds: 20

resources:

requests:

memory: "512Mi"

cpu: "100m"

limits:

memory: "512Mi"

cpu: "200m"

volumes:

configMap:

name: my-app-1-0

---

apiVersion: v1

kind: ConfigMap

metadata:

name: my-app-1-0

data:

code.js: |

module.exports = {

"server": {

"port": 80

},

"redis\_url": {

"url": "ep of redis"

},

"mysql\_url": {

"url": "rd ep"

}

}

---

apiVersion: autoscaling/v1

kind: HorizontalPodAutoscaler

metadata:

name: my-app-1.0

labels:

functionGroup: myapp

service: myapp-service

deployedby: user-x

version: "1.0"

wait: "false"

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: my-app-1-0

minReplicas: 2

maxReplicas: 20

targetCPUUtilizationPercentage: 70

---

apiVersion: policy/v1beta1

kind: PodDisruptionBudget

metadata:

name: my-app-1-0

labels:

functionGroup: myapp

service: myapp-service

deployedby: user-x

version: "1.0"

wait: "false"

spec:

maxUnavailable: 5%

selector:

matchLabels:

functionGroup: myapp

service: myapp-service

version: "1.0"

deploymentTag: "1.0"

---

apiVersion: v1

kind: Service

metadata:

name: myapp-1-0

labels:

functionGroup: myapp

service: myapp-service

deployedby: user-x

version: "1.0"

wait: "false"

spec:

selector:

functionGroup: myapp

service: myapp-service

version: "1.0"

deploymentTag: "1.0"

ports:

- protocol: TCP

port: 80

targetPort: server-port

type: ClusterIP

---

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-app

labels:

wait: "false"

annotations:

nginx.ingress.kubernetes.io/server-alias: "myapp.example.com"

spec:

ingressClassName: public-ingress-nginx

rules:

- host: myapp.example.com

http:

paths:

- path: /

pathType: "Prefix"

backend:

service:

name: myapp-service

port:

number: 80

---

apiVersion: v1

kind: Service

metadata:

name: myapp-canary

labels:

functionGroup: myapp

service: myapp-service

version: "1.0"

wait: "false"

spec:

selector:

functionGroup: myapp

service: myapp-service

version: "1.0"

deploymentTag: "1.0"

ports:

- protocol: TCP

port: 80

targetPort: server-port

type: ClusterIP

---

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: myapp-canary

labels:

wait: "false"

annotations:

nginx.ingress.kubernetes.io/server-alias: "myapp.example.com"

nginx.ingress.kubernetes.io/canary: "true"

nginx.ingress.kubernetes.io/canary-by-header: "x-myapp-canary"

spec:

ingressClassName: public-ingress-nginx

rules:

- host: myapp.example.com

http:

paths:

- path: /

pathType: "Prefix"

backend:

service:

name: myapp-canary

port:

number: 80

**Access from Internet using ingress-nginx:**

**Either:**

**Use of ingress-nginx control in K8**

Ingress-nginx: will sit in k8s: registry.k8s.io/ingress-nginx/controller:v1.2.1 two types:

**Public:**

Will create NLB (internet facing)

Pod will manage target groups – e.g. port 443, certificate on the NLB

Target group will forward to high node port on K8s nodes ( > 30000 )

e.g. 443 🡪 30100

**Internal:**

Will create NLB (internal facing schema)

Pod will manage 2 target groups – e.g. ports 443 and 80

Will forward to 2 target groups for both ports ( both > 30000 )

e.g. 443 🡪 30200, 80 🡪 30300

**Register in R53 –**

App.example.com CNAME NLB internet address of public NLB

So internet access 🡪 NLB 🡪 Node port TG 🡪 Get to K8s worker node

**Alternative Solution – EC2 deployment:**

Instead of K8s and Pods –

* Use Ec2 servers in an autoscaling group.
* Autoscaling would need to be based on either CPU or network load.
* Each ec2 will contain an http server – servicing the application.
* The http webserver(s) will have access to the Redis and MySQL DBs
* Create nlb or alb which forward to target group, which will forward to the EC2 web servers

Target group is managed according to spec of AWS to the autoscaling group: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/attach-load-balancer-asg.html>

The http web server could also be based on a docker image – hence the CICD workflow – could be similar – in that the developer builds the docker images – and the launch template in the scaling group could pull the latest image to the web servers.

Versions of the web app – would need to be managed via a second autoscaling group with a 2nd set of EC2s –

The active web-app would be based on a DNS routing ( R53 possible )– for instance:

App.example.com 🡪 CNAME Version1.app.example.com

Version1.app.example.com 🡪 1st NLB 🡪 TG1 🡪 ASG1

Version2.app.example.com 🡪 2nd NLB 🡪 TG2 🡪 ASG2

Of course, management of this design is much more manual – and probably more expensive than the K8s one.