Case Study

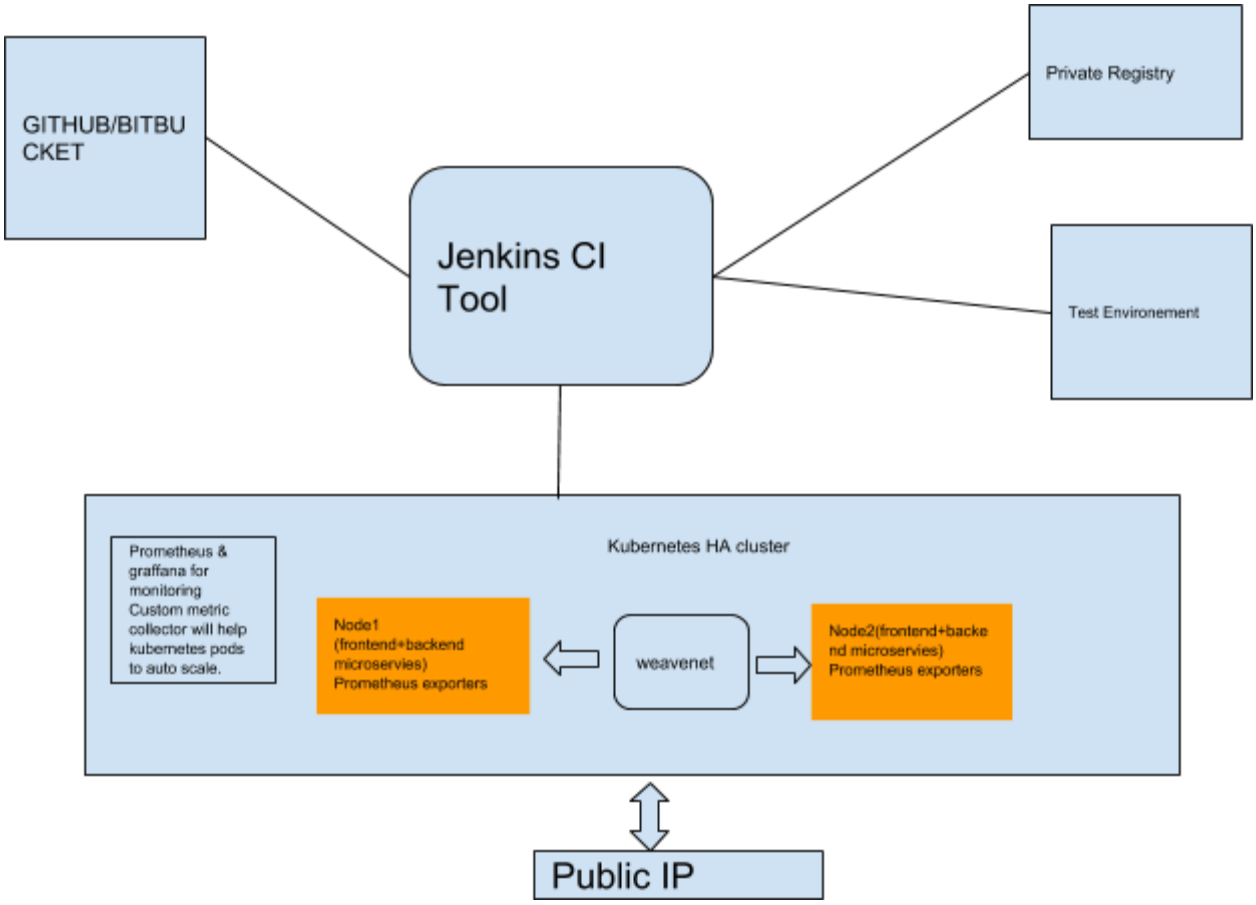
Here we have to move a typical 3- tier application; frontend, backend and persistence (A DataBase)
from a datacenter to AWS/kubernetes. Please, provide the architectural solution, where you show all
the services you would use to solve this problem and how you use them and how you would manage
the migration to minimize downtime. Also, the solution should satisfy the following requirements:
● Ability to withstand high peak loads. Typical peak load can be 50x of normal load and last typically 4h.
● Provide multiple testing environments for developers and realistic performance testing setup while keeping cost in control at the same time.
● Ability to test and deploy changes multiple times per day, even during the peak hours.
● The new solution should have monitoring and centralized logging across the board.
Bonus Points:
● Implement deployment strategies like blue-green and canary
● Describe a solid rollback strategy in case something goes wrong.

1) First we need to ask below questions to development team in order to understand the architecture and requirement better.

    a) How many environment is required? Like PROD /UAT/RC /QA
    b) What would be expected load in terms of http requests, during normal and peak hours ?
    c) What would be the security policies, if any or vulnerabilities?
    d) How frequently they would release the application?
    e) How many requests per second or daily traffic expected?
    f) How many RPS can application handle?
    g) What will be the application versioning strategy?

2) Second we can set up infrastructure on AWS cloud or bare metal servers for test machines.

3) The architecture diagram will be look like below using one test environment and a production environment with kubernetes high available arch.



In the above diagram, Developer will either do the code commit or click on build job to start the new jobs which will create a docker image and push it to repository. The same image will be used by jenkins to create pods. This docker image will have mircoservies . The final deployment on kubernetes cluster will be done using yaml file

After the jenkins has deployed on kubernetes cluster, the service is exposed using weavenet networking and a public IP available for end user to access the webcontent over browser.

- JENKINS : I am using this tool to integrate the CI & CD pipeline. Jenkins is using plugins to integrate the GITHUB and kubernetes to seamless deliver the code with a click of a button.

- Kubernetes : I am using kubernetes to orchestrate the docker containers.
- Weavenet : I am using weavenet as overlay network and to provide networking so my pods can talk to each other
- Using the docker private registry.
- Prometheus & Grafana: I am using them as monitoring tools and metric collectors. Based on metrics, The pods will be auto scaled.
- Prometheus exporter: This exporter will run on both nodes which will collect the metrics and send it to prometheus.

4) We can minimize the cost by using minikube and test the yaml files or we can use the one or two machines as test set up where we can perform our UAT or regression tests.

5) We can start migrating the services by few initial microservices to minimize downtime and later on we can introduce or scale up all the other microservices to include as full scale application whereas we can use a global loadbalancer to take care of the traffic between the services to new microservies and the legacy application.

6) Testing the services under stress load will determine the actual production capacity for pods running on kubernetes. To withstand load based on the outcome of expected load , we can define auto-scaling by running the service to use the metric collector based on http requests and using the average value threshold to decide on spinning off another pod. This metric collector will be collected by prometheus operator and running as a horizontal auto scaler pod. The service is running with min replica as 2 and max replica as 10 lets consider.

7) We can perform canary deployment of microservices by creating another service with track label as canary.

```
kind: Service
apiVersion: v1
metadata:
 name: frontend-app-canary
 spec:
 type: NodePort
 selector:
 app: frontend-app
 track: canary
```

```
 ports:
 - protocol: TCP
   port: 8080
   nodePort: 8081

---

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: frontend-app-canary
 labels:
    app: frontend-app
 spec:
 replicas: $CANARY_REPLICAS
 selector:
    matchLabels:
    app: frontend-app
    track: canary
    template:
    metadata:
     labels:
      app: frontend-app
       track: canary
       spec:
       containers:
       - name: frontend-app
          image: $DOCKER_IMAGE_NAME:$BUILD_NUMBER
          ports:
          - containerPort: 8080
          livenessProbe:
          httpGet:
          path: /
          port: 8080
          initialDelaySeconds: 15
          timeoutSeconds: 1
          periodSeconds: 10
          resources:
          requests:
          cpu: 200m
```