Sravanthi Gelley

Migrate Cloud Foundry Application to Kubernetes

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master’s Thesis

4 September 2022

Author Title Number of Pages Date	Sravanthi Gelley Migrating a cloud foundry application to Kubernetes 41 pages 4 September 2022
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Ville Jääskeläinen, Principal Lecturer

The case company has an insurance-related application. The application was running in a Pivotal cloud foundry environment. Due to high licensing costs, the management decided to migrate to the Kubernetes environment. The thesis is related to migrating the insurance application from Pivotal Cloud Foundry to Kubernetes. The thesis describes the various steps taken as part of the migration from the application developer's perspective.

The thesis objective is to document all the steps the developer did for the migration. The thesis is divided into multiple sections that describe application's state before migration, a literature overview of the different software needed for migration, the implementation of the migration steps, and the results after the migration.

The main challenge during the migration was that no one in the development team had any previous technical expertise related to the Kubernetes environment. So, a considerable learning path was needed along the migration.

The steps needed in the migration are described. The application migration was completed successfully in the development environment first and then in quality, customer test, and finally in the production environment.

Keywords	Docker, Kubernetes, Jenkins, Vault, Continuous Integration, Helm, Pivotal Cloud Foundry

Contents

List of Figures

List of Abbreviations

VMM	Virtual Machine Monitor or a hypervisor
OS	Operating System
PCF	Pivotal Cloud Foundry
K8s	Kubernetes
CLI	Command-Line Interpreter
PaaS	Platform as a Service
TAS	Tanzu Application Service

1 Introduction

The case company has an insurance-related application deployed in the Pivotal cloud foundry environment. Pivotal Cloud Foundry (PCF) is a platform as a service (PaaS) environment used to build, deploy, run, and scale the applications and services on the cloud [1]. PCF is an enterprise version of the open-source cloud foundry developed and supported by the cloud foundry foundation members Pivotal, Dell EMC, IBM, VMWare, and many others. Pivotal cloud foundry is now called the Tanzu Application Service [13] after it is acquired by VMWare.

To increase the deployment flexibility and to decrease licensing costs, the case company decided to migrate the application from the cloud foundry to the Tanzu Kubernetes application platform from VMWare. Kubernetes is an open-source cloud platform that is originated from Google's Project Borg. It is sponsored by the Cloud Native Computing Foundation members AWS, Azure, Intel, IBM, RedHat, Pivotal, and many others. Kubernetes is a container runtime used to manage the lifecycle of applications across environments [2].

Cloud Foundry offers a higher-level abstraction for deploying applications so that developers can mainly concentrate on application development and deployment. In contrast, Kubernetes, on the other hand, offers developers a resilient distributed framework that automatically scales clusters and applications and takes care of failovers. Kubernetes makes a good option for developers who require the flexibility of deploying applications on cloud infrastructures [3].

With the increased flexibility in Kubernetes comes more responsibility on developers since they need to write and maintain the configuration required for the deployment and scalability. The deployment descriptors used for Kubernetes have multiple parameters to configure the environment. Understanding manifests and the architecture for a beginner needs a great learning curve. This thesis briefs about the migration journey of the application from PCF to Kubernetes. The scope of the content is mainly on the things a developer needs to know and do in a Kubernetes environment.

This thesis is divided into six sections. The first section introduces the problem statement and the objective of the thesis. The second section gives a brief description of the application specification. The third section explains the tools and software used for migration, like docker, Kubernetes, vault, and Jenkins. The fourth section explains the implementation details of the migration. The fifth section concerns how to improve the deployment manifests in the future development. The last section is about the conclusion on how the migration went in different environments.

2 Project Specifications

The case company has an insurance related application. The application has two projects. The backend is a spring boot project. The backend uses a MySQL database. The frontend is a standalone react application that makes API calls to the backend. The backend is integrated into some other applications also. Both these projects were running in a cloud foundry environment. In cloud foundry, the cf tool is used as a CLI to interact with the server. Cloud foundry application will have a manifest file that contains metadata about the application. Below is a sample manifest of how it looks like:

```
- domain: app.net
  path: ./app/
  name: app
  space: app-prod
  organization: org1
  buildpack: java-buildpack-openjdk-v4-49-1
  memory: '1024M'
```

To deploy an application to cloud foundry, cf push command is used. In PCF, developers do not need to provide any descriptor about the dependencies required for the application to run in the cloud environment. For example, a java application needs java run time to run the application. Cloud Foundry automatically identifies all the necessary runtime tools needed for the application and packages the application uses to build packs. Build packs are a set of tools needed for the application to run. When running cf push command, cloud foundry identifies the necessary build packs for the application and downloads them automatically. Then the application is packaged and deployed in the cloud. When migrating to Kubernetes, it is the developer's responsibility to provide the descriptor needed to package the application in the docker file. The details about the docker file will be explained in the coming chapters.

In cloud foundry, it is a single command cf push that takes care of packaging and deploying. Whereas in Kubernetes, multiple descriptors are needed to be provided by the developer. One descriptor is needed to specify the details on how to package the application, one descriptor to provide the package that needs to be deployed, and one descriptor about how to expose the application to the outer world.

One more issue with the application is the configuration required for the application, such as the database to be used and the external application URLs used are all part of the application. This way, the exact copy of the package is not used when deploying to different environments. When migrating to Kubernetes, the application configuration must be moved out of the code using the configs and vaults described in later sections.

To provide continuous integration and deployment, Jenkins is used. All the jobs created in Jenkins are freestyle jobs. The entire configuration is a setup in Jenkins itself. Even the Jenkins used for the application must be migrated during application migration to run on the Kubernetes environment. The admin must manually create all the configurations for the jobs in the new environment. To avoid the manual creation of configurations in the freestyle Jenkins jobs, they are converted into pipeline jobs where the configuration for the jobs is mentioned in the descriptor files. This way, it is effortless when migrating the jobs.

3 Containerization and Kubernetes Concepts

This section introduces the various concepts used in the migration process. First, it explains how the journey of virtualization happened from virtual machines to the latest Kubernetes and then briefly introduces different Kubernetes concepts and Jenkins.

3.1 Virtualization

Virtualization is the concept of abstraction of a physical machine into multiple virtual machines (VMs). Before virtualization, it was the era of mostly running single applications on one server due to incompatibility problems faced by running different applications with the operating system. With the decrease in hardware costs and advancements in memory and storage capacity over time, resources were not utilized entirely by running a single application. Virtual machines help reduce the budget on hardware, increase CPU and memory utilization, and reduce the physical maintenance of the machines.

To achieve virtualization, software called a virtual machine monitor (VMM), or a hypervisor is used. Gerald J. Popek and Robert P. Goldberg wrote the framework that implements virtualization. According to Popek and Goldberg, a hypervisor is supposed to have three properties, such as fidelity which means the environment created for VM by VMM must be identical to the physical machine. The second property is isolation or safety. That is, the VMM should be able to completely control the machine's resources to share as needed with the VMs. The third property is that the performance of the VMM should be almost the same as the actual machine. The other uses of VMs are to provide high availability in case one server is down, another VM can be created quickly, and high security since one application cannot easily access another. In the below picture, each virtual machine has its copy of the operating system, which makes the VM heavyweight.

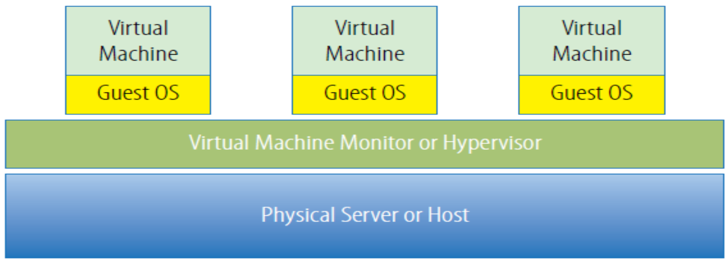


Figure 1: A basic virtual machine monitor (VMM) [8]

3.2 Containerization

As mentioned earlier, virtual machines solved the problem of running multiple applications on a machine without incompatibilities. However, there are still some problems with VMs. Since each VM is packed with its copy of the operating system, they are heavy and consume lots of resources. Each OS copy has its licensing costs. Also, migrating VMs across hypervisors is slow. Containers solve these problems.

Containers are lightweight compared to VMs as they do not need their copy of the operating system. Containers running on a machine share host OS. Sharing reduced the OS licensing costs. Portability also makes it easy to move containers across machines. Since they are lightweight, it would take just a few seconds to start a container, whereas VMs might take a few minutes to start.

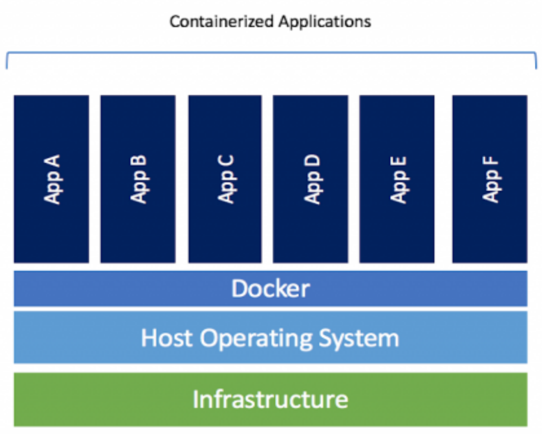


Figure 2: Containerized applications [9]

Docker software is used for building containers. Docker software is developed by Docker Inc. company [14]. With the help of docker, the container can be built and run on a machine. Docker Inc provides software called Docker Desktop for both Windows and Mac. It can be installed to set up a docker environment locally on a laptop for building or running containerized applications.

When docker software is installed on a laptop, it mainly contains two components, a docker client and a docker engine. When docker commands are executed, the docker client connects to the docker engine. The docker engine would take care of building and running the container.

Docker images can have multiple layers. Multiple images can share these layers. This way, the storage, and speed improve while pulling images. When docker tries to modify a layer, it creates a new copy of the layer and writes on the new copy. This way, the other containers sharing the layers are not affected.

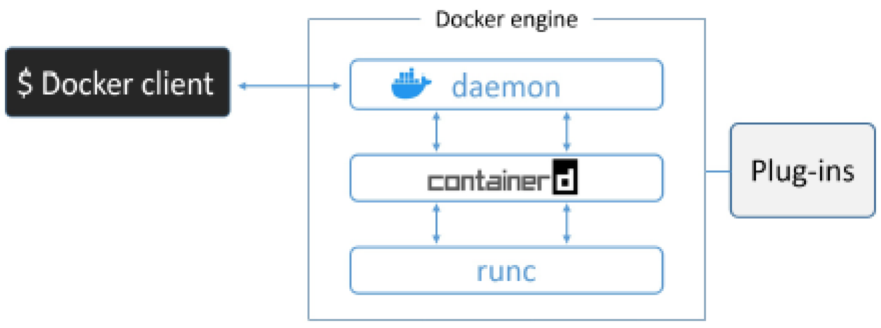


Figure 3: Docker architecture [10]

The runc is responsible for creating the containers. The containerd is responsible for managing the containers like starting, stopping, pausing, etc. To create a container, docker needs a template called docker image. A Docker image is a package of an application and all its dependencies, such as libraries and configurations. The instructions to create an image are provided to docker in the form of a docker file. Other images can be added as needed dependencies in the docker file when creating an image. The containerd component's responsibility is to pull or download the dependent images from docker image repositories called docker registries and create a new image for the application. Once the image is ready, it can be pushed to a docker registry so that the image can be downloaded onto any computer and can be run. This makes it easy to run the exact copy of code across all the environments from development to production.

The Docker image name will have a repository name and its tag. A tag is used to specify the version of the image. If there is no tag mentioned, it defaults to the latest image, which is an image tagged with the name 'latest'.

3.3 Kubernetes

So far, docker has been used to create docker images and containers. If the number of containers increases, it is difficult to manage them individually. Kubernetes comes as a rescue that provides a platform to run these containers. Kubernetes [15], also known as k8s, is a platform developed by Google to orchestrate or manage containerized applications. Kubernetes can be deployed on a VM or a physical machine. Some advantages of Kubernetes are that a developer or a system admin does not need to choose where to run the application. Kubernetes chooses the required resources and utilizes them at their best. Rollback to the previous state is easy if there are any issues during deployment. Kubernetes provides services such as auto scaling the pods based on the workload, replacing them if they go down, and monitoring them.

All the necessary descriptions to tell Kubernetes that an application has to be deployed will be provided in the configuration files in JSON or YAML structure format. These descriptions can be stored in the version control. This file includes different properties such as which image is to be deployed, how many instances to run, on which port the application is to be exposed etc.

3.3.1 Architecture

Coming to the Kubernetes architecture, the Kubernetes cluster is a set of hosts called nodes. These nodes are divided into master and worker nodes. Each node contains multiple pods. A pod is like a basic block in Kubernetes. Each pod has multiple or single containers running on them. Each pod has its IP address. Multiple Containers in a pod do not span on multiple nodes.

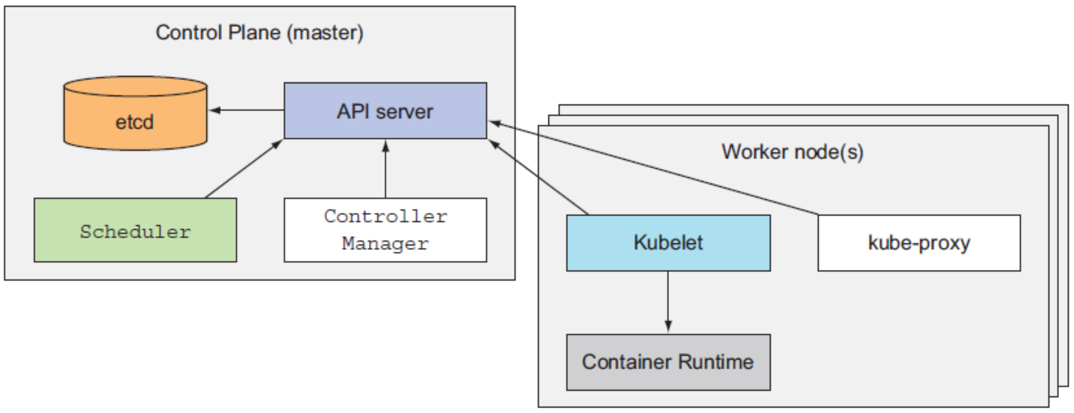


Figure 4: Components of Kubernetes cluster [11]

The master node has different components such as API Server, scheduler, etcd, and controller manager. All interactions with the Kubernetes cluster take place through an API server that exposes the rest APIs. Kubernetes provides a CLI called kubectl to interact with the API server. A scheduler is used to schedule pods in the worker nodes. The etcd is like a distributed storage space that stores all the configurations, secrets, etc. API server stores data in etcd. The controller manager monitors the health of worker nodes. If any worker nodes go down, the controller manager restarts them based on metadata configuration stored in etcd. Each node has components called kubelet and Kubernetes proxy. Kubernetes proxy acts as an abstract layer to communicate with the containers. Kubelet acts as a representative of the node and manages the pods running in the nodes.

3.3.2 Pod Creation

A pod can be created by using the kubectl run command or by using the yaml descriptor as shown below.

```
kubectl run kubia –image=luska/kubia –port=8080
```

The basic yaml descriptor used for creating a pod is shown below.

```
apiVersion: v1
kind: Pod
```

```
metadata:
   name: myapp
   namespace: default
   labels:
     app.kubernetes.io/name: myapp
     app.kubernetes.io/instance: myapp-1
     app.kubernetes.io/version: "1.0.0"
 spec:
    containers:
    -  image: app:latest
       name: myapp
    ports:
    -   containerPort: 8080
        protocol: TCP
```

The apiVersion parameter is used to provide the Kubernetes API version. Metadata includes the pod details such as name, namespace, and additional tag names called labels. Spec contains the details about the container running in the pod. The container is created with the image specified in the image parameter. The port parameter specifies which port container is running. Kubectl apply -f descriptor.yaml is the command used to create the pod. Once the pod is created, pods can be listed with the kubectl get pods
command.

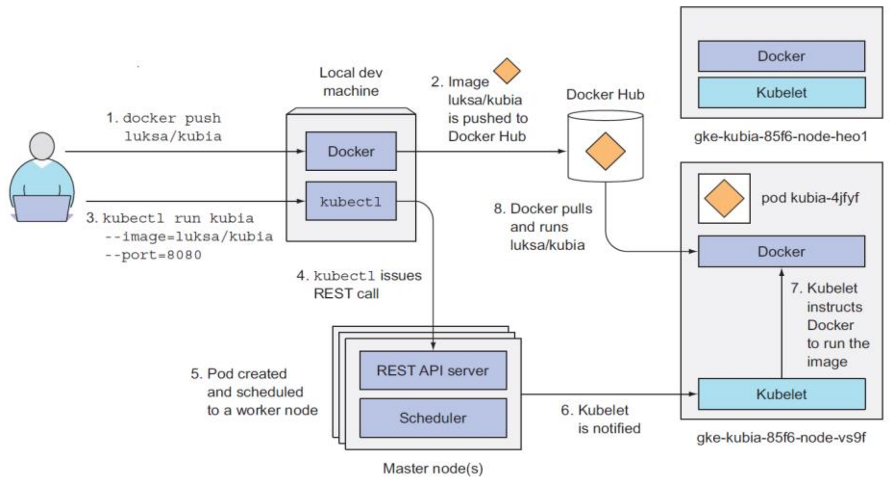

Local dev machine
1. docker push luksa/kubia
Docker
kubectl
3. kubectl run kubia --image=luksa/kubia --port=8080
2. Image luksa/kubia is pushed to Docker Hub
Docker Hub
4. kubectl issues REST call
8. Docker pulls and runs luksa/kubia
Docker
Kubelet
gke-kubia-85f6-node-heo1
pod kubia-4jfyf
Docker
7. Kubelet instructs Docker to run the image
Kubelet
gke-kubia-85f6-node-vs9f
5. Pod created and scheduled to a worker node
REST API server
Scheduler
Master node(s)
6. Kubelet is notified

Figure 5: Running the container inside Kubernetes [11]

By running the kubectl command or kubectl apply command, the request is sent to the API server. There are different resources listening to the requests received by the API server, such as the deployment controller, scheduler, and replication controller. Since the yaml corresponds to the pod creation request, the scheduler reacts to this. The scheduler creates a new pod and assigns the created pod to one of the worker nodes. The Kubelet identifies that a pod is scheduled and asks the docker to pull the image from the registry if the image is not available already. After the image is downloaded, docker starts the container.

3.3.3 Deployment

One of the main advantages of Kubernetes is to make sure the deployment of the application is up and restarted automatically in case of any failures. This is achieved by creating a deployment resource instead of creating the pods directly. This is done by submitting a deployment request to the API server. It is similar to the above pod descriptor, but the kind changes to ‘Deployment’ is shown below.

```
apiVersion: v1
kind: Deployment
metadata:
    name: myapp
    namespace: default
    labels:
       app.kubernetes.io/name: myapp
       app.kubernetes.io/instance: myapp-1
       app.kubernetes.io/version: "1.0.0"
 spec:
      replicas: 2
      strategy:
         type: RollingUpdate
      selector:
          matchLabels:
          app: app1
      containers:
      -  image: app:latest
         name: myapp
      ports:
      -   containerPort: 8080
          protocol: TCP
     readinessProbe:
     periodSeconds: 2
        exec:
            command:
              - ls
              - /var/ready
```

Here in the deployment, replicas indicate how many pods are to be created. When a deployment request is sent, the deployment controller creates a replica set. The replication controller will check the number of replicas needed and creates those needed pod replicas.

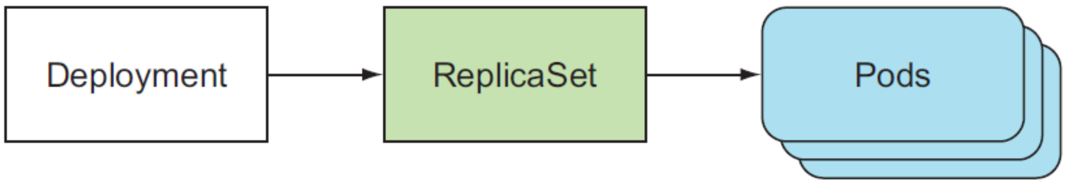


Figure 6: Replica set created by deployment to manage pods [11]

The replication controller always makes sure that the same number of pods are running as mentioned in the descriptor using the replica set. If any nodes fail, a new node automatically takes over the failed node. Kubernetes can monitor and automatically scale the number of nodes based on usage. It also helps the app to discover other required services so that a developer does not need to implement methods to choose the master that delegates the requests to other services.

After the application is running in Kubernetes, and in case a new version is to be deployed again, specific strategies for deploying the new version. It is specified in the strategy parameter in the descriptor. RollingUpdate is the default strategy. After receiving a new deployment request, one of the old pods is removed, and a new pod is created with a new version of the app. This way, deployment is done step by step so that there will not be any downtime. Another strategy is Recreate, where first all the old pods are removed, and then new pods are created. Here there will be some downtime for the application.

Sometimes, as soon as the pod is created, it may not be ready to accept the requests from the client. There might be some configuration to be loaded. The readiness probe parameter specified under the container is used to tell whether the pod is ready or not. A readiness probe can be a get URL or a process that notifies that the pod is ready or not to take the requests. The PeriodSeconds parameter above specifies the delay between calling the readiness URL again. Until the pod is ready, it will not receive any requests. This helps to make sure that client is always connected to a healthy pod [7].

3.3.4 Labels and Annotations

When there are hundreds of pods, it would be difficult to quickly identify which pod belongs to which application. Labels can fix this. In the above yaml descriptor, labels are used to group pods in an abstract way so that it is easy to identify what the pod is related to. A label is like a key-value pair. A pod can have multiple labels. Pods can be filtered by using the labels in different ways, such as listing pods that match or doesn't match a certain label key. The label can be added or updated after the creation of a pod also. Labels can also be used to tell Kubernetes that certain pods are to be created on certain nodes. For example, the below descriptor tells that the pod must be created on nodes that have the label gpu with value true.

```
spec:
  nodeSelector:
      gpu: "true"
```

Like labels, annotations can also provide additional information about the pods, like created by whom, about the version, etc. However, they cannot be used to filter the pods. Usually, the label has a small value, whereas annotation can contain more information.

Another type of grouping is by using a namespace. A namespace is used to prevent the overlapping of different groups of pods. Resource names inside a namespace should be unique, and the same name can be used in a different namespace. Kubernetes has a default namespace. All pods will be created under the default namespace if no namespace is provided. These namespaces organize pods of different applications within the same cluster.

```
apiVersion: v1
kind: Namespace
metadata:
name: app1-namespace
```

The above descriptor can be used to create a namespace with the command kubectl create -f namespace.yaml. A namespace can be used to filter the pods. Pods can be deleted using labels or namespaces. When a pod is deleted, the container inside it will be stopped. To delete a pod, its replica set object should also be deleted. Otherwise, as soon as the pod is deleted, the replica set thinks that the number of pods does not match the descriptor and immediately creates a new pod.

Pod labels are used by the replica set to make sure the same number of pods exist, as mentioned in the description. So, when a label is renamed, the pod goes out of scope from the replica set. Since the replica count is mismatched, the replica set creates a new pod.

3.3.5 Services

Now the application is running in a container inside a pod in a worker node. The next step is accessing the application. As said before, each pod has an IP address. If the pod is accessed with this address, in case the pod is recreated or moved, the new pod gets a new address. So, another abstraction layer, Service, is created to make the access. Service will have a fixed IP address as long as it exits. All the pods related to a service can be accessed with the same address. Service determines the pod to send a request
based on the pod selector.

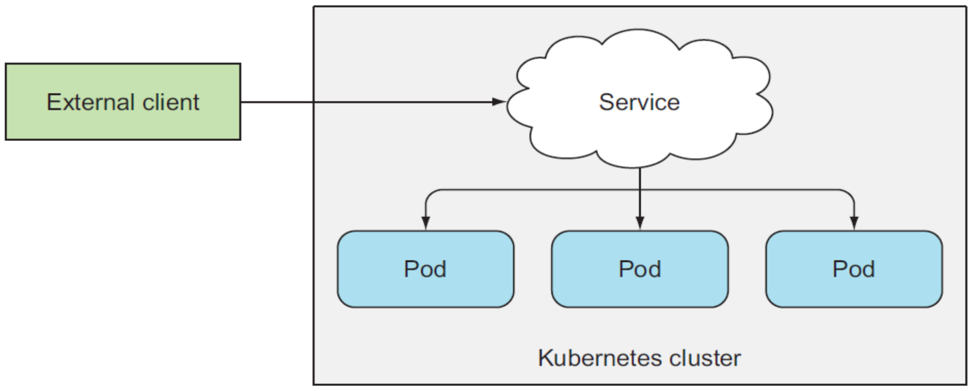


Figure 7: Exposing a service to external clients [11]

The below descriptor shows how to create a service that exposes the application outside the cluster using NodePort type.

```
 apiVersion: v1
kind: Service
metadata:
    name: app1
    spec:
       type: NodePort
       sessionAffinity: ClientIP
    ports:
    -  name: http
       port: 80
       targetPort: 8080
       nodePort: 30123
    -  name: https
       port: 443
       targetPort: 8443
       nodePort: 30124
    selector:
       app: app1
```

Service randomly selects one of its pods and sends the client request. To always forward the request to the same pod from the same client, the sessionAffinity parameter as ClientIP is used. By default, its value is none. Each service can be exposed on multiple ports.

There are three kinds of services. The first one is ClusterIP which is the default service type. This type is not accessible from outside the cluster. ClusterIP service is more suitable when an external client does not access the service and is just used internally by other pods. The second one is NodePort. This service type is used to expose nodes to the external world through one of the ports in the range 30000-32768. With the above descriptor, Kubernetes would reserve ports 30123 and 30124 on all the nodes. It makes the service be accessed from outside the cluster. NodePort type requires the port to be opened in the firewall rules.

The third one is the Load Balancer type. A Load balancer is created externally, which connects to the pods and redirects the traffic. The load balancer’s IP address is used to access the application. If there are multiple load balancer services, each service will have a separate IP address.

A pod can find the IP address of the service by using the environment variables which are prefixed by service name with names SERVICE_NAME_SERVICE_HOST and SERVICE_NAME_SERVICE_PORT. Another way is by using the internal DNS lookup. The DNS name will be in the format of service-name.namespace.svc.cluster.local, where svc.cluster.local is a cluster suffix.

3.3.6 Ingress

Ingress is a way to access the service outside the cluster. As seen above with Nodeport, the firewall is to be updated. With the load balancer, each service requires a separate IP address. Whereas with ingress, a single IP address is enough to access any service. The service to which the request is to be forwarded is determined by the service name present in the request URL.

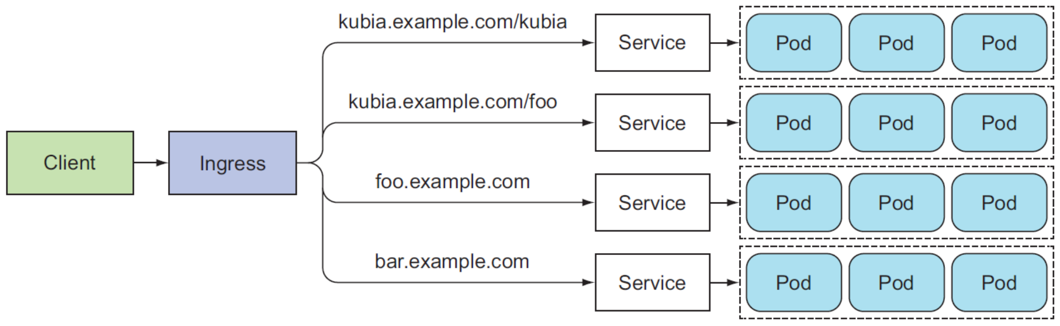


Figure 8: Expose multiple services through single ingress [11]

To use ingress, Kubernetes should have an ingress controller. For example, Nginx server can be deployed to provide this functionality.

3.3.7 Config Maps and Secrets

Usually, all the configuration related data will be stored in a file in the form of key value properties, and it will be packaged into the application code. So, when an image is created, all this data goes into the image, and whoever downloads the image can see the configuration data. Also, if some of the configurations need to be modified, then the whole image must be recreated. This is not a good idea.

So, this configuration should be stored outside the image and needs to be supplied to the container as arguments. One option to pass the configuration is through environment variables in the pod descriptor to the container.

```
kind: Pod
spec:
containers:
 - image: app1:latest
   env:
   - name: database_name
     value: "mydb"
```

However, the problem with this is that separate pod descriptors must be created for each environment in case different values are to be passed in the environment variables. To solve this problem, ConfigMap is used. A config map is a list of key-value pairs, and the config map name can be set in the pod descriptor. Different environments can have different config maps with same name. To create config map, the below command is used.

```
kubectl create -f configmap.yaml
```

To set the created config map in the pod,

```
apiVersion: v1
kind: Pod
metadata:
     name: app1
spec:
  containers:
  - image: app1:latest
env:
-   name: db_name
    valueFrom:
        configMapKeyRef:
            name: db-config
            key: db_name
```

The config map db-config has a property with the name db_name. This property is passed to the container. The data in the config map is stored as plain text. So, it should not include any sensitive information.

There is another Kubernetes object called Secret to store sensitive information. It is a key value pair with value encrypted. This secret file can be passed as a parameter in the container section of the pod descriptor, like a config map.

```
env:
-   name: db_password
    valueFrom:
        secretKeyRef:
            name: appsecret
            key: db_password
```

appsecret is the secret created using the command kubectl create -f appsecret.yaml. This secret is passed to the container.

3.4 Continuous Integration

Continuous Integration helps to ensure the code builds correctly without any test failures. When multiple developers work on the same application by creating feature branches, it would be good to know if there are any build failures or test failures whenever the developer pushes the code to the code repository like GIT or BitBucket. Jenkins is one such tool for the build management system. Jenkins is an open-source java application. It can be customized with 1000’s of plugin tools that can be integrated easily with many of the development related software like GIT, docker, Kubernetes, etc.

In a development life cycle, a developer creates a feature branch first from the main branch when making changes to the application code so that the code is not broken. When Jenkins is integrated with the code repository, it builds the code whenever there is a new feature branch or new merge request, runs the unit and integration tests and runs other validations such as code style, PMD and SonarQube. Build fails if any of these validations fail. This continuous process that enables multiple developers to work in parallel that ensures there are no build failures is called continuous integration. This helps to maintain software quality.

Once the feature branch is tested, the branch is merged with the main branch. When new changes are added to the main branch, a Jenkins job is triggered, which creates a docker image and pushes it to the docker registry. This is called continuous delivery. With this, the code is already ready to be deployed to a production environment.

If Jenkins is integrated with Kubernetes, whenever a docker image is ready with new changes, a Jenkins job can deploy the new image to Kubernetes development environment. This is called continuous deployment. With this, any deployment issues that might arise when application goes to production can be caught early and fixed.

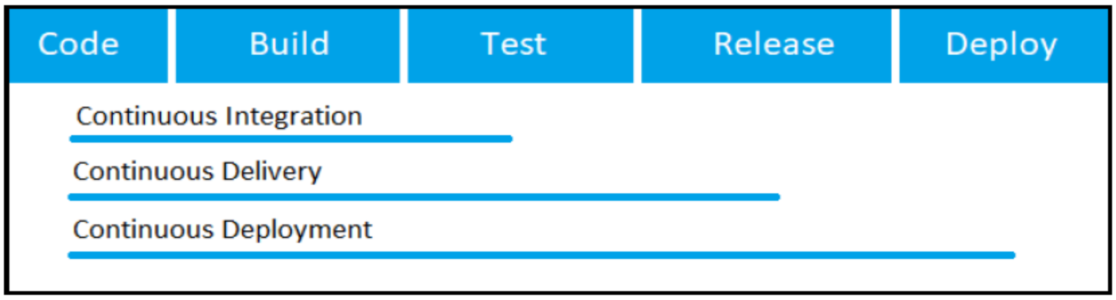


Figure 9: Continuity in Software Development Lifecycle [12]

Coming to creating jobs in Jenkins, there are mainly free-style and pipeline jobs. In free style jobs, all the configuration for the job will be created using the Jenkins GUI. If the job is to be migrated to new Jenkins, then this configuration needs to be recreated again manually. Another style is pipeline job. Here all the configuration is written in a file called Jenkins file and this file can be stored in the code repository. So, this way, migration to another Jenkins is quite easy. Also, its better suitable in case the configurations are complex and better to track the changes.

4 Migration Implementation

This section gives an overview of the frontend and backend pods of the application. Then describes the different manifests created for containerization and deployment like docker file, service yaml, deployment yaml, and ingress yaml. Then explains the Jenkins pipelines created to push the application image to the image repository and another pipeline for the application deployment.

4.1 Pod Structure

Backend and frontend containers are not deployed into the same pod because both can have their scaling requirements. It is easy to scale horizontally when both components are in different pods. Also, since pod creation takes very a short time, having separate pods is not an overhead.

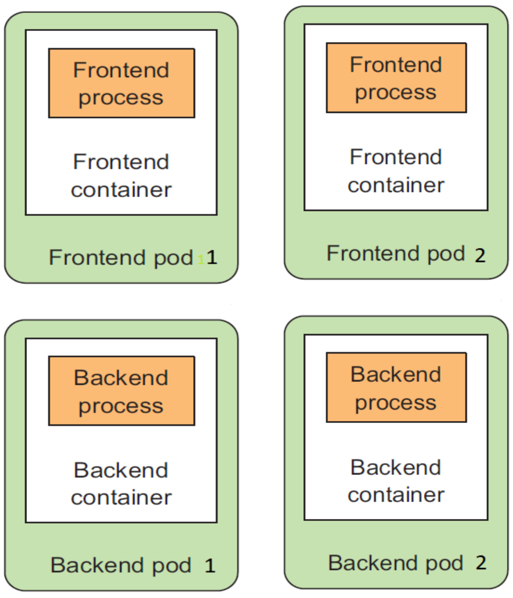


Figure 10: Frontend and backend pods [11]

4.2 Dockerfile

The first step is creating a docker image. An image is a layered file system. An image is like a template for creating a container. An image contains multiple layers. An image is immutable. It cannot be updated once created. It can only be deleted. To create an image using docker, a Dockerfile is created. A Dockerfile is a text file with instructions on how to create an image.

Below is a docker file for a java spring application. This is a multi-stage docker build.

```
FROM maven:3.8.3-openjdk-8 AS build
COPY /src /src
COPY pom.xml .
RUN mvn -f pom.xml clean package
#
# build package
#
FROM openjdk:8
COPY --from=build  target/application-service.jar /application-service.jar
EXPOSE 8080
CMD ["java", "-jar", "application-service.jar"]
```

In a multi-stage docker build, there will be multiple FROM statements. In the first stage, the maven image is used as a base docker image. Maven build is executed, and the application jar file is generated. In the next stage, java image is the base docker image. The jar generated in the previous image is copied to the docker image. The last statement tells what command to run inside the container. The advantage of multiple stage builds is that the files not necessarily needed from the previous stage can be discarded in the next stage. So here, maven image will not be added to the docker image as it is not needed to run the application and only the jar file is copied to the image.

Once Dockerfile is ready, image can be built by using the command,

```
docker build -t application-service -f Dockerfile
```

-t will be the image name and tag and -f will be the location of the dockerfile. This way, image can be built. It can be pushed to a docker registry. A docker registry is a repository for the images. It can be private or public repository. There are multiple package management systems available like GitHub, DockerHub, Proget [16] etc. Once the image is pushed into the repository, it can be pulled again from any of the environments for deployment.

4.3 Deployment Files

Once the docker image is ready, it can be deployed to a Kubernetes environment. For development purposes locally, docker desktop is used. It has an option to set up Kubernetes cluster on the local environment.

Kubectl is a command line tool that comes when installing Kubernetes. This internally calls the Kubernetes API server to execute the commands. To deploy an application, we need to describe the configuration details to create the pods in the form of yaml descriptor files. On a high level, yaml file contains three sections pod metadata, pod spec which contains details about the pods, containers and volumes details and status of the pods and containers. Three descriptor files service, deployment, and Ingress are needed.

4.3.1 Service

Frontend will have one service so that the client can access all the frontend related pods with single IP address. Similarly, backend will have a service so that frontend can connect to the backend and whenever backend pods are relocated in a cluster, frontend need not be reconfigured.
The service yaml descriptor looks like this.

```
apiVersion: v1
kind: Service
metadata:
   name: app-service
   namespace: "demo"
   labels:
      app: app-service
spec:
     type: ClusterIP
  ports:
    - port: 8080
      targetPort: 8080
  selector:
    app: app-service
```

The apiVersion specifies the version of the Kubernetes api used. v1 is the first stable version released. Kind specifies what kind of resource are created. Label is used to group the pods with a tag. Pods can be filtered with a label selector. Namespace ‘demo’ is used to group resources without overlapping. Without namespace, resource will be added to default namespace. Port specifies the port internally used and target port is the port used to send requests to the container. On receiving request on port 8080, service redirects the request to one of the pods. The pods are identified by using the label selector mentioned in the service descriptor. Type clusterIP means the pod is accessible only within the cluster.

4.3.2 Deployment

Deployments make sure that applications remain available by keeping the desired number of pods running and replacing unhealthy pods with new ones.

```
apiVersion: apps/v1

kind: Deployment

metadata:
  name: app-service
  namespace: "demo"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app-service
  template:
    metadata:
      annotations:
        vault.security.banzaicloud.io/vault-addr:
        vault.security.banzaicloud.io/vault-path:
        vault.security.banzaicloud.io/vault-role: vault-webhook
      labels:
        app: app-service
    spec:
      containers:
        - image: app-service:latest
          imagePullPolicy: IfNotPresent
          env:
            - name: DB_USERNAME
              value: vault:app-service/#datasource.username
          resources:
```

```
          limits:
            cpu: 2
            memory: 1Gi
          requests:
            cpu: 100m
            memory: 64Mi
      livenessProbe:
          failureThreshold: 10
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 60
          successThreshold: 1
          timeoutSeconds: 240
        ports:
          - name: http
            containerPort: 8080
            protocol: TCP
        volumeMounts:
    serviceAccountName: default
```

Replicas specify the number of replicas of pods to be created by the replica set. The replica set gets the count of pods based on the label selectors and tries to match it with the number of replicas. If there are less or more pods, replica set creates or deletes the pods. Annotations are name value pairs like labels that can be used to group resources, but they are mostly used to describe larger information. They are mostly used by the tools or developers to provide more information about the application running. Here there are some annotations related to vault that will be discussed in the later section in detail.

Under spec parameter, details about the container that needs to be run inside the pod will be specified. The image pull policy is ‘IfNotPresent’. This means kubelet does not pull the image every time a new pod is created. if there is an image with the same name and tag already available, it will be used. Another pull policy available is ‘Always’ which means the image should be downloaded every time. All the other environment variables that should be passed to the container will be set in the spec container parameter. The data source name is obtained from the vault and is injected into the process running in the container.

It is possible to specify the required CPU and memory for the container using resource limits and requests parameters under spec container parameters. These requests are used by the scheduler and creates the pods under the nodes which could satisfy the requirement. Pods can request additional resources if needed than the amount specified in the requests parameter. In case the application uses more memory than the limit, additional memory is not granted and out of memory error is thrown. Monitoring tools can be used to see the live and peak resource usages of the application and can adjust the limits accordingly. It is a good practice to specify the limits so that the pods with unhealthy behaviour do not disrupt the healthy pods by consuming all the resources [5].

Liveness probe settings are used to monitor if the container is up or not. Kubelet will invoke the url formed by using the IP address of the container and the path and the port specified in the liveness probe parameter at regular intervals to check if the process is running or not. If the http status code returned by the url is >=200 and <400, it is considered as success. If the url does not return the success code, the container is restarted automatically. It is possible to set an initial delay when the probing must start after the container is started using initialDelaySeconds. PeriodSeconds specify how frequently the probing must happen. FailureThreshold parameter is used to specify how many failures can happen before the container gets restarted. Here in the above file, after 10 continuous failures, the container is restarted [7].

Then there is the volume mount parameter. It is used to specify and storage location that can be shared with other containers. Since this app does not use any external storage, there is no specific information. There are still many more parameters that can be set to customize the deployments as per the requirements.

4.3.3 Ingress

A Kubernetes Ingress is a robust way to expose services outside the cluster. It lets one consolidate routing rules to a single resource and gives powerful options for configuring these rules.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-service
  namespace: "demo"
  labels:
    app: app-service
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: hostname
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app-service
                port:
                  name: http
```

In the above ingress yaml, single rule is defined under the spec rules. When there is an incoming request, ingress matches the host and path and identifies the service and the port to which the request is to be redirected to.

4.4 Migrate Secrets to Vault

Kubernetes provides secrets object to store sensitive information such as passwords needed for the application, and these secrets can be passed in the pod descriptor so that containers can use them. They are made available to container by injecting them through environment variables or by storing in a file and sharing it with the container. Problems with these secrets are that they are just base 64 encoded and not encrypted. So, they can be easily decoded. Another issue is that these secrets are available inside the pod. If anyone can get into the pod for example, with ssh, they could get the passwords with environmental variables.

Secrets are stored in Kubernetes database etcd without encryption. So, anyone who has access to this etcd can know the secrets.

To solve above problems, vault webhook is used. It is like an event notification mechanism that listens to events such as creating or updating pods. When a pod is created or updated, Kubernetes api makes a call to the webhook. Webhook details are specified in the deployment descriptor through annotations. Webhook authenticates to the secret management system such as HashiCorp Vault and fetches the passwords and inject them as secrets directly into the container running in the pod. Using this mechanism, passwords are not stored in etcd or on any file and are not available through environment variables inside the pods [4]. The webhook details are provided with annotations as shown below in pod descriptor.

```
 template:
   metadata:
     annotations:
       vault.security.banzaicloud.io/vault-addr: https://vault-address
       vault.security.banzaicloud.io/vault-path: app1
       vault.security.banzaicloud.io/vault-role:
```

vault-addr is used to specify the vault url. Vault-path specifies the path where the secrets are created in the vault. Vault-role specifies what role is used to access the secrets. Now to inject the secrets into the container, the secrets are accessed this way:

```
env:
    - name: DB_PASSWORD
      value: vault:app-service#datasource.password
```

Here the password stored in the vault is injected as environment variable DB_PASSWORD into the container process. Another advantage is, the application configuration is moved out of the code and so when the configuration is modified, there is no need to create the docker image again.

4.5 Jenkins Pipelines

So far, all the deployment related configurations in Kubernetes are described. Even the Jenkins is also migrated to Kubernetes. So, the Jenkins configuration from old Jenkins is to be moved to new Jenkins. All the free style jobs in old Jenkins are converted to pipeline jobs in the new Jenkins. The configuration is moved into the application code so that it is useful to migrate in the future and easy to maintain.

Jenkins has a master and slave architecture. Jobs are run on the slave nodes or worker nodes. Jenkins can be configured so that the slave nodes run on the worker nodes in Kubernetes. The worker pod is removed once the job completes. This way even when multiple jobs are run on Jenkins, it does not go down due to heavy load. To run the job on worker node in Kubernetes, a template is to be defined how to create the pod. This will also be part of the Jenkins pipeline.

4.5.1 Jenkins Build Application Image Pipeline

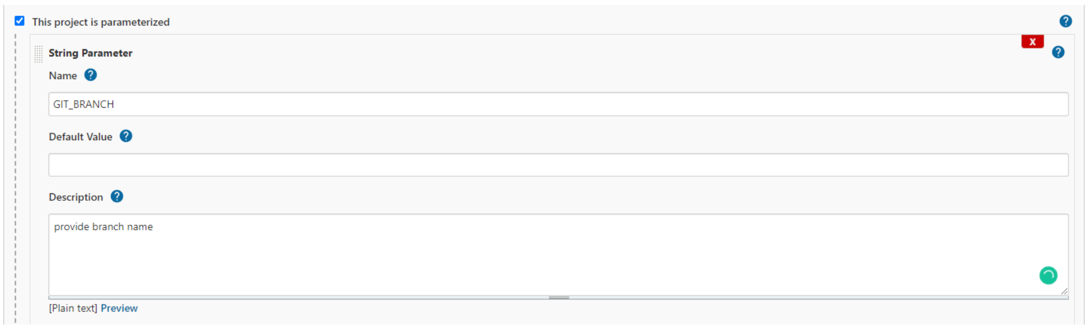
Below is a pipeline script that contains the pod template and multiple stages to build the application and push to the docker registry.

```
pipeline {
    agent {
        kubernetes {
            yaml """
---
apiVersion: v1
kind: Pod
spec:
  containers:
  -
      command:
        - cat
      image: "maven:3.8.3-openjdk-8"
      name: maven
      tty: true
  -
      name: kaniko
      imagePullPolicy: IfNotPresent
      image: "gcr.io/kaniko-project/executor:v1.7.0-debug"
      command: ['/busybox/cat']
      tty: true"""
        }
}
```

The pod template is defined inside the Kubernetes block. The pod will have 2 containers. One container is to build the application. Since the application is on java, maven tool is used to build the application. The next container is to run kaniko. Kaniko is a tool from Google which is used to build images from a docker file similar to docker. Kaniko is used inside a container or in a Kubernetes cluster. Here the image is created inside a Jenkins worker pod. If docker is used, then docker is to be run inside another docker. This is called DinD approach. There is no need to build image because docker needs root level access to create an image and therefore DinD approach is not secure. Kaniko does not need any root access and it runs within the user space [6].

After defining the pod template, different stages in the Jenkins pipeline are defined. First stage is defining the parameters for the job. Below job has the git branch name parameter that can be used to check out and create image for that branch. Using parameter, job can be customized to run for any working branch and image can be created.

```
stages {
  stage('Setup parameters') {
    steps {
      script {
        properties([
          parameters([
            string(defaultValue: 'master',
            description: 'GIT Branch Name',
            name: 'BRANCH_NAME')
          ])
        ])
      }
     }
    }
```



This project is parameterized

String Parameter

Name: GIT_BRANCH

Default Value:

Description: provide branch name

[Plain text] Preview

Figure 11: Parameters in a Jenkins pipeline

Second stage is checking out the application from git.

```
   stage ('checkout') {
        steps {
          checkout([
         $class: 'GitSCM',
         branches: [[name: '*/${BRANCH_NAME}']],
         doGenerateSubmoduleConfigurations: false,
         extensions: [[$class: 'RelativeTargetDirectory', relativeTargetDir:
'demo']],
         submoduleCfg: [],
         userRemoteConfigs:[[credentialsId:'',url: 'git@gitlab.local:url ']]])
        }
  }
```

Third stage is building the application using maven tool. The application depends on dependencies that reside in a private maven repository. So, to pull those dependencies when creating a maven package, first there should be authentication process to the remote maven repository. These credentials are stored in the Jenkins credentials repository, and they are pulled by using the credential name 'maven-key'. After authentication, application jar is built by running the maven package command.

```
stage ('maven') {
  steps {
    container('maven') {
      withCredentials([usernamePassword(credentialsId: 'maven-key',
username
      Variable: 'USER_NAME', passwordVariable: 'PASS_WORD')])
      {
        sh 'mvn -f app-code/pom.xml clean package -s settings.xml
        -Dusername=${'USER_NAME } -Dpassword=${PASS_WORD}'
```

```
       }
     }
   }
 }
```

After building the application jar, the image is built using Kaniko tool and pushed to the private docker registry repository. In order to use the private docker registry, first the credentials ‘registry_cred’ are fetched from the Jenkins credentials and they are used for authentication.

```
stage('Build & push image') {
   steps {
     withCredentials([usernameColonPassword(credentialsId: 'registry_cred', vari
     able: 'registry_credentials')]) {
     container(name: 'kaniko', shell: '/busybox/sh') {
       sh """
          #!/busybox/sh
          mkdir -p /kaniko/.docker
          cat << EOF >  /kaniko/.docker/config.json
          {
           "auths": {
                 "${DOCKER_REGISTRY}": {
                       "auth": "`echo -n "${docker_credentials}" | base64`"
                 }
            }
       }
      EOF
  """
```

After the image is built using the docker file present in the application code, image is pushed to the registry. The path to push is specified in the destination parameter.

```
   sh """
    #!/busybox/sh
    /kaniko/executor --dockerfile `pwd`/Dockerfile --context `pwd` --cache=true --
    destination=dockerhub.net/docker/app-service:latest
   """
   }
 }
}
}
```

4.5.2 Jenkins Deploy Application Pipeline

Now the Jenkins job has finished that pushes the image to the image registry. Now it can be used inside the Kubernetes for deployment. To deploy the application to Kubernetes, there is a separate Jenkins pipeline job.

```
pipeline {
    agent {
        kubernetes {
            yaml """
---
apiVersion: v1
kind: Pod
spec:
  containers:
  -
      command:
        - cat
      image: "alpine/k8s:1.20.7"
      name: kubectl
      tty: true
  -
      command:
        - cat
      image: "centos:8"
      name: curl
      tty: true
"""
        }
    }
```

Next in the environment block, the service token is obtained from the Jenkins credentials. This will be used to deploy to Kubernetes.

```
environment {
    API_TOKEN = credentials("${API_TOKEN}")
}
```

Next the different stages in the pipeline are specified. First one is deploying the app to Kubernetes. The deployment descriptors present in the application code is passed in the kubectl apply command. Kubeconfig parameter will point to the descriptor which contains the Kubernetes server location where the application is to be deployed. It is stored in the Jenkins credentials.

```
   stage ('deploy to env') {
       steps {
        container('kubectl') {
            withCredentials([file(credentialsId:   "${CREDENTIAL_ID}",   variable:
'config')]) {
   sh  'kubectl  apply  -f  deployment.yaml  --token=${API_TOKEN}  --kubecon-
fig=${config}'
       }
  }
```

The next stage is to check if the deployment is successful. This is verified by checking if the application health URL is returning http status code 200.

```
stage ('Service up') {
   steps {
      container('curl') {
         timeout (time: 10, unit: 'MINUTES') {
            script {
              waitUntil {
               try {
                sh 'curl -s --head  --request GET  "http://app.com/actuator/health" |
                   grep "200"'
                   return true
               } catch (Exception e) {
               return false
            }
}
```

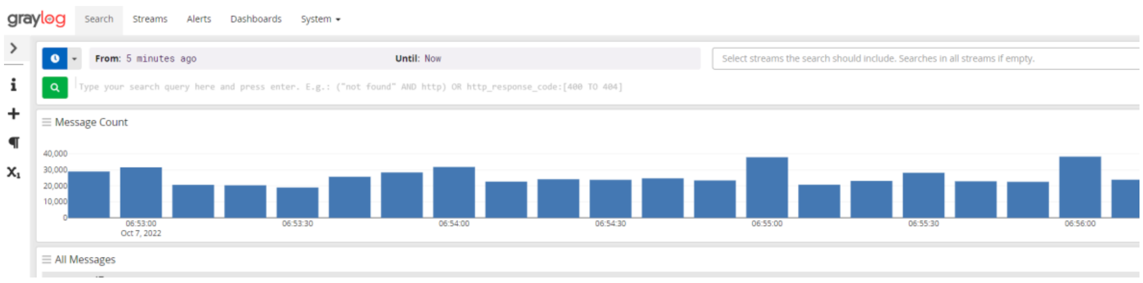
Above pipelines are for the backend application. Similar pipelines are created to deploy frontend project also. Jenkins needs Kubernetes plugin to run the above pipelines. In the Kubernetes plugin, the Kubernetes server where the worker nodes are to be run the above jobs needs to be configured. This is the task done by the operations team.

4.6 Results

The application needs to be run in four different environments development, quality assurance, customer test and production. So, there will be four different namespaces created in the Kubernetes cluster to deploy the application into different environments. During the migration process, the descriptors are created and tested in the local environment setup using the docker desktop. It provides both docker and Kubernetes, so it is easy to test the descriptors. After the development, the application is deployed to the development environment. Even the database is to be migrated to the Kubernetes environment. But it is taken care by the admin team responsible for the databases. The secrets used by the application are created in the HashiCorp vault. Once the migration is completed for both backend and frontend projects in the development environment, there were some network issues in accessing the app from other applications. So, network team did the firewall rule changes. After successful migration in development, similar migration happened in QA environment, customer test and production environments. There were two separate Jenkins to deploy, one for development and QA and one for test and production. There were several challenges in gaining access to different systems and in connecting to different networks. But finally, the migration went successfully.

4.7 Logging

Application logging is very crucial to identify the different issues when accessing the application. There are different tools available to log the data. Graylog is used for the current application. The logs of different applications deployed in all the namespaces will be logging to the same graylog. There will be different filtering options to filter logs based on the time and based on namespaces.



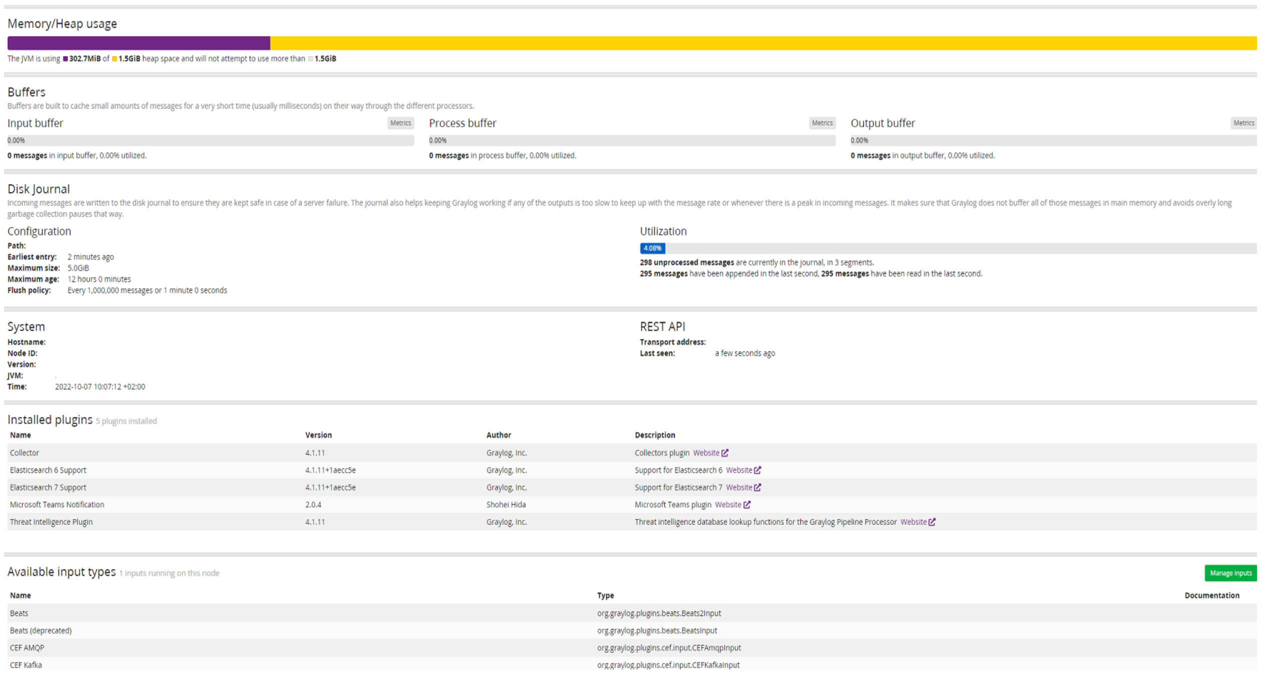
From: 5 minutes ago Until: Now

Select streams the search should include. Searches in all streams if empty.

Type your search query here and press enter. E.g.: ("not found" AND http) OR http_response_code:[400 TO 404]

Figure 12: Graylog showing the message count

Above graph shows number of messages received in a certain time frame. These metrics are useful to analyze the traffic of the application and helps to evaluate the peak loads. There is also network information logged like number of open connections and port information and helps to view the global configuration of the system. It helps to see the node and load balancer state if its running or not.



Memory/Heap usage

The JVM is using ■ **302.7MiB** of ■ **1.5GiB** heap space and will not attempt to use more than □ **1.5GiB**

Buffers

Buffers are built to cache small amounts of messages for a very short time (usually milliseconds) on their way through the different processors.

Input buffer — Metrics
0.00%
0 messages in input buffer, 0.00% utilized.

Process buffer — Metrics
0.00%
0 messages in process buffer, 0.00% utilized.

Output buffer — Metrics
0.00%
0 messages in output buffer, 0.00% utilized.

Disk Journal

Incoming messages are written to the disk journal to ensure they are kept safe in case of a server failure. The journal also helps keeping Graylog working if any of the outputs is too slow to keep up with the message rate or whenever there is a peak in incoming messages. It makes sure that Graylog does not buffer all of those messages in main memory and avoids overly long garbage collection pauses that way.

Configuration

Path:	
Earliest entry:	2 minutes ago
Maximum size:	5.0GiB
Maximum age:	12 hours 0 minutes
Flush policy:	Every 1,000,000 messages or 1 minute 0 seconds

Utilization

4.08%

298 unprocessed messages are currently in the journal, in 3 segments.
295 messages have been appended in the last second, **295 messages** have been read in the last second.

System

Hostname:	
Node ID:	
Version:	
JVM:	
Time:	2022-10-07 10:07:12 +02:00

REST API

Transport address:	
Last seen:	a few seconds ago

Installed plugins 5 plugins installed

Name	Version	Author	Description
Collector	4.1.11	Graylog, Inc.	Collectors plugin Website
Elasticsearch 6 Support	4.1.11+1aecc5e	Graylog, Inc.	Support for Elasticsearch 6 Website
Elasticsearch 7 Support	4.1.11+1aecc5e	Graylog, Inc.	Support for Elasticsearch 7 Website
Microsoft Teams Notification	2.0.4	Shohei Hida	Microsoft Teams plugin Website
Threat Intelligence Plugin	4.1.11	Graylog, Inc.	Threat intelligence database lookup functions for the Graylog Pipeline Processor Website

Available input types 1 inputs running on this node — Manage inputs

Name	Type	Documentation
Beats	org.graylog.plugins.beats.Beats2Input	
Beats (deprecated)	org.graylog.plugins.beats.BeatsInput	
CEF AMQP	org.graylog.plugins.cef.input.CEFAmqpInput	
CEF Kafka	org.graylog.plugins.cef.input.CEFKafkaInput	

Figure 13: Graylog showing the system overview

Above pictures gives an overview of the graylog system state. Journal is a place-holder for the incoming log messages before they are processed by graylog. Picture shows about the different plugins installed on graylog.

4.8 Monitoring(wavefront)

Tanzu Kubernetes provides a wavefront tool to monitor the applications. There are different dashboards to view the metrics such as CPU, memory, number of requests, metrics related to the health of different pods and clusters.

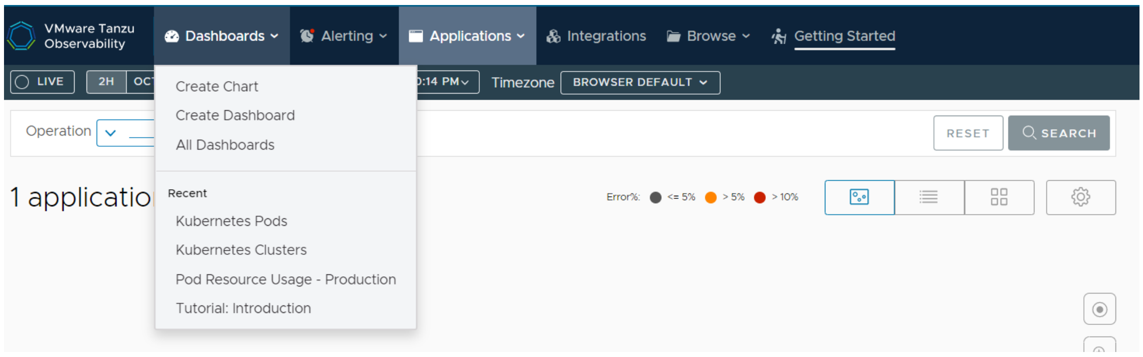


Figure 14: WaveFront dashboard menu

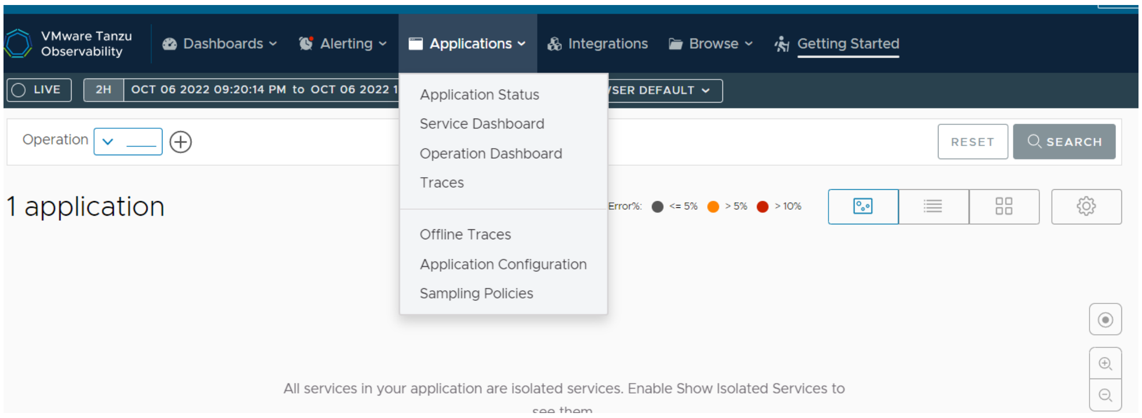


Figure 15: WaveFront applications menu

The dashboards can also be customized so that the necessary columns need can be configured for the statistic reports.

It is possible to different types of graphs related to the restart rates of different pods, bytes transferred rate, network errors, file system usage per pod, number of threads. all these metrics can be viewed at cluster level, namespace level, node level or at the pod level. This helps to find out if there is any unusual activity in the system. When there is any sudden downtime of the application, these metrics are helpful to get find out the root cause. In case if the application is behaving slow, the thread rate and latency rate related metrics could be useful. As per the metrics, if the developer wants to allocate more resources for the application, the deployment descriptor can be modified, and Kubernetes will move the pods to new machines that satisfy the new requirements. It is possible to export these metrics in different formats like csv, histogram, pie graph etc. The application errors are also shown based on different severity levels like critical, warnings, info etc.

4.8 Future Development

When migration to Kubernetes, the major work was creating different descriptor files like service, deployment, and ingress. The configuration also looks complicated. These configuration files also need to be created separately for each application. This increases a lot of duplication. Sometimes if the production descriptor file is modified directly and the changes are not done in the git repository, there will be a mismatch between the copy saved and the configuration that is in production. To solve this problem, there is a configuration package manager called Helm. In helm, there will be a template descriptor that can be used by various applications and different applications can inject their own values to the template. This way duplication of the descriptors can be reduced. The company has created a helm template so that it can be used across different applications and each application do not need to create its own copy of descriptors. In the next development cycle, it is planned to understand more about helm and use it in the future.

5 Conclusion

The thesis is based on the problem faced by the case company with high licensing costs to run the insurance-related application in Cloud Foundry. It was decided to move the application from Cloud Foundry to the Kubernetes cluster. Since the application was already based on a microservices pattern, there was not much change needed in the application code. However, deployment manifests were needed to deploy to the Kubernetes environment.

This thesis explained how Kubernetes provides so much flexibility in configuring the application deployment with many parameters. Before, the application could be deployed only in a specific cloud foundry environment. However, it is now ready to be deployed in any of the cloud providers such as AWS, GCP, etc. This made it easy to auto-scale the application based on the request load. Also, it is easy to migrate the application without downtime using the Kubernetes rolling update strategy.

This thesis explains the different steps taken as part of the migration. First, the application is containerized. To containerize the application, a docker file manifest was created that contains information on how to create the docker image. Then deployment files were created to deploy the containerized application to the Kubernetes environment. A service file and ingress file were created to expose the application to the outer world. All the application-related passwords were migrated to the Hashicorp vault so that the exact copy of the application code is ready to be deployed to different environments. Then the free-style Jenkins jobs of the application were converted to pipeline jobs by moving all the configurations to the file and stored in the version control.

The main challenge faced during the migration was a huge learning curve involved in understanding Kubernetes concepts for creating the manifests. There were a few challenges in getting access to connect to the Kubernetes cluster and Jenkins in the development environment to validate the created manifests. For the application to be accessed from other external applications, it needed a firewall and network changes. All the network-related issues were solved by the network team, and the application database migration to the Kubernetes environment was taken care of by the database team.

Once all the access-related, network, and database-related issues were resolved, the application was migrated successfully in the development environment. Next, the application was migrated successfully into the other quality, customer test, and production environments. After the migration was done in all the environments, the old environments related to the PCF were decommissioned.

As part of future development, the helm package is to be explored to obtain the advantage of replacing the separate deployment files in different environments by creating a helm template and injecting different values related to each environment. This migration process has taught a lot about the Kubernetes architecture and deployment procedures and how the Jenkins pipelines can be utilized to provide continuous delivery and deployment.

6 References

1. Farmer R, Jain R, Wu D. Cloud Foundry for Developers. Packt Publishing Ltd; 2017.
2. Shopen O. Comparing Kubernetes to Pivotal Cloud Foundry — A Developer's Perspective [Internet]. Medium. 2019 [cited 2022 Sep 18]. Available from: https://medium.com/@odedia/comparing-kubernetes-to-pivotal-cloud-foundry-a-developers-perspective-6d40a911f257
3. Arun R. CloudTweaks | Cloud Foundry vs Kubernetes: Which One is Better? [Internet]. CloudTweaks. 2022 [cited 2022 Oct 18]. Available from: https://cloudtweaks.com/2022/03/cloud-foundry-vs-kubernetes-which-one-is-better/
4. Mutating Webhook · Banzai Cloud [Internet]. banzaicloud.com. [cited 2022 Oct 18]. Available from: https://banzaicloud.com/docs/bank-vaults/mutating-webhook/
5. Resource Management for Pods and Containers [Internet]. Kubernetes [cited 2022 Oct 18]. Available from: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
6. Baeldung. An Introduction to Kaniko | Baeldung [Internet]. www.baeldung.com. 2020 [cited 2022 Oct 18]. Available from: https://www.baeldung.com/ops/kaniko
7. Configure Liveness, Readiness and Startup Probes [Internet]. Kubernetes. [cited 2022 Oct 18]. Available from: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
8. Portnoy M. Virtualization: essentials. 2016.
9. Lessing M. Containers vs VMs: Key Differences [Internet]. SDxCentral. [cited 2022 Oct 18]. Available from: https://www.sdxcentral.com/cloud/containers/definitions/containers-vs-vms/
10. Poulton N. Docker deep dive: zero to Docker in a single book. Germany: Nigel Poulton; 2020.
11. Luksa M. Kubernetes in Action. Simon and Schuster; 2017.

12. Sander Rossel. Continuous Integration, Delivery And Deployment. S.L.: Packt Publishing Limited; 2017.
13. VMware Tanzu Application Service [Internet]. Vmware.com. VMware, Inc. or its affiliates; 2020 [Accessed 2022 Oct 18]. Available from: https://tanzu.vmware.com/application-service
14. Docker. Enterprise Application Container Platform | Docker [Internet]. Docker. 2018 [Accessed 2022 Oct 18]. Available from: https://www.docker.com/
15. Kubernetes. Production-Grade Container Orchestration [Internet]. Kubernetes.io. 2019 [Accessed 2022 Oct 18]. Available from: https://kubernetes.io/
16. ProGet | Package your Applications and Components – Inedo [Internet]. inedo.com. [cited 2022 Oct 18]. Available from: https://inedo.com/proget