ijar
ISSN NO. 2320-5407
Journal Homepage: -www.journalijar.com
INTERNATIONAL JOURNAL OF ADVANCED RESEARCH (IJAR)
Article DOI:10.21474/IJAR01/7449
DOI URL: http://dx.doi.org/10.21474/IJAR01/7449

RESEARCH ARTICLE

DOCKER AND GOOGLE KUBERNETICS.

Subash Thota.
Architect, Data Diverse.

Manuscript Info

Manuscript History

Received: 20 May 2018
Final Accepted: 22 June 2018
Published: July 2018

Keywords:-
Big Data Analytics, Social Analytics, Storage Analytics, Containers, Cloud computing, Virtual Machines, Google, Runtime, Data Mitigation, Metadata, Docker.

Abstract

In today's world, every customer is thinking of agility, parallel development and reducing cost, especially infrastructure cost and operational cost. We have seen SOA world where we have written code and multiple services talk to each other's for a business use case, but sometimes we end up with one big code base which is monolithic in nature and maintenance is becoming difficult. We have seen customers are using cloud and paying for on-demand services without effectively utilizing resources. These problems invite micro-services. In this paper, I am going to discuss how one should use micro-service in a production environment and local machine, how to scale, monitor and support Blue-Green deployment.

Introduction:-

About the Domain:-

Microservice is an architectural style that structures an application as a collection of loosely coupled components services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack and conduct parallel development, which enables faster time to market.

Why Microservice?

We have seen problems in monolithic applications which is hard to scale, hard to maintain, new releases take time which results in lack of agility, lack of innovation and frustrated customer. In today's agile world agility is one of the key components and therefore parallel development is one of the key aspects. At the same time, the smaller code base is easy to maintain as long as you can manage multiple code bases and monitor this. Also, customers are paying high prices especially when the code is running in the cloud without utilizing resources effectively. Microservice is the way to rescue.

1. Microservice is a variant of SOA architectural style that structures an application as a collection of loosely coupled services.
2. Service should be fine-grained
3. The protocol should be light weight

Advantages of micro-services:-

1. Allow the architecture of individual service to evolve through continuous refactoring and enable continuous delivery and better agility
2. Improves modularity, easier to understand, parallelize development scale independently

Corresponding Author:-Subash Thota.
Address:-Architect, Data Diverse.

3. Faster time to market and reduce the frustration of end user
4. Decreased Cost – reduce the cost to add more products, customers or business solutions

Comparison of the micro-service framework

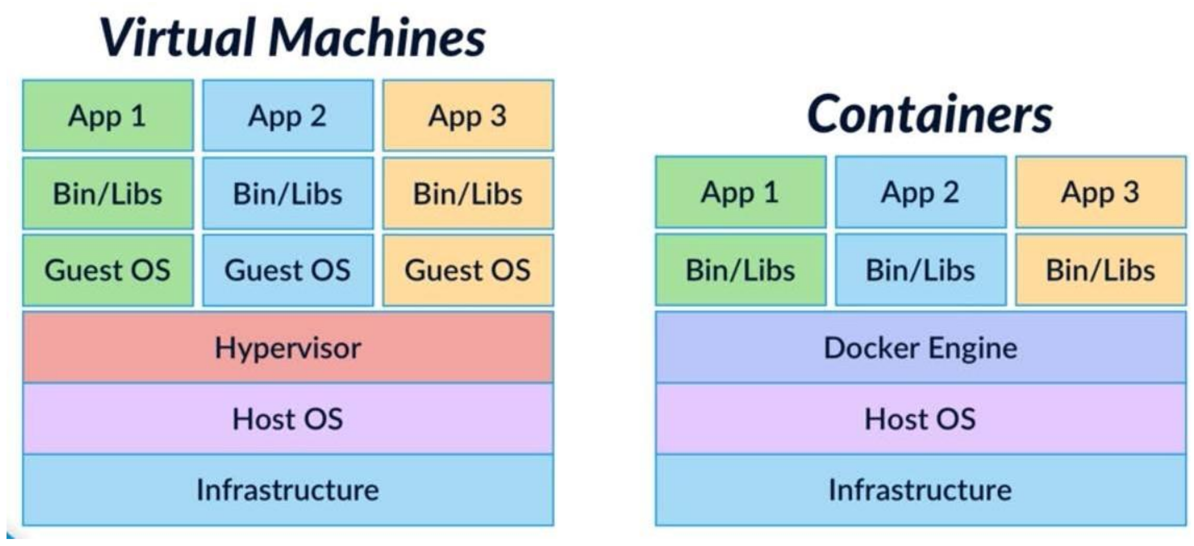
	Docker Swarm	**Kubernetes**	**Mesos**
Cluster Installation	Very easy to install and setup	Slightly complex to set up	Easy to set up for small cluster but considerably complex for a large cluster
Container Deployment	Completely docker based and very easy to set up	YAML based on all components	JSON based
Cluster Configuration	At least one server is running everything. For production, at least 2 nodes in each layer for discovery and managers	At least One master and one minion node. In production, at least 3 servers in each layer	At least One master and one slave. In production, at least 3 master and several workers
Scalability	This is an evolving point in Swarm	Medium to a large cluster. Very well suited for complex application	Large to Large-scale clusters.
Maturity	Mature but still evolving	Very mature. Direct descendent of Google internal platform.	Very mature (especially for the large cluster)

With the above comparison, it's very clear that Kubernetes and Mesos are matured for an enterprise solution. Considering the complexity, backed by experienced engineers and suitability in medium scale applications and increasing popularity, I am going to discuss Kubernetes framework in this document.

Networking:-
Internal network often uses 1 GBPS connections, or faster. Optical fiber connections allow much higher bandwidth between servers. So the question is how much such responses can be transmitted over 1 Gbps connection in one second? Let's actually do some math. 1 Gbps is 1048576 Kbps. If average JSON response is 5 KB (which is quite a lot!), you can send 209715 responses per second through the wire with just one pair of machines. That's why network connection is usually not a bottleneck. Another aspect of microservices is that it's scale easily. Imagine two servers, one hosting the service, another one consuming it. If ever the connection becomes a bottleneck, just add two other servers, and you can double the performance.

Docker and Kubernetes:-
Docker is containerization technology. "Docker containers wrap a piece of software in a complete file system that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment." – Source - https://www.docker.com/what-docker

How is Docker different from the Virtual machine?
The virtual machine runs on top of hypervisor and hypervisor emulates the computer hardware.



The New Way is to deploy containers based on operating-system-level virtualization rather than hardware virtualization. These containers are isolated from each other and from the host: they have their own file systems, they can't see each other's processes, and their computational resource usage can be bounded. They are easier to build than VMs, and because they are decoupled from the underlying infrastructure and from the host file system, they are portable across clouds and OS distributions.

Summary of container benefits:-

1. Agile application creation and deployment: Increased ease and efficiency of container image creation compared to VM image use.
2. Continuous development, integration, and deployment: Provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
3. Dev and Ops separation of concerns: Create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
4. Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
5. Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-prem, Google Container Engine, and anywhere else.
6. Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to run an application on an OS using logical resources.
7. Loosely coupled, distributed, elastic, liberated micro-services: Applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a fat monolithic stack running on one big single-purpose machine.
8. Resource isolation: Predictable application performance.
9. Resource utilization: High efficiency and density. In today's world when the customer is moving to cloud, effective resource utilization is one key aspect to reduce operational cost. Micro-service will help the customer to achieve that goal by deploying multiple micro-services into a virtual machine so that effective resource utilization is much better and it will reduce infra-structure cost also.

Google Kubernetes:-

Nearly all applications nowadays need to have answers for things like

1. Replication of components
2. Auto-scaling
3. Load balancing
4. Rolling updates
5. Logging across components
6. Monitoring and health checking

7. Service discovery
8. Authentication

Google has given a combined solution for that which is Kubernetes, or how it's called in short – K8s.
At a minimum, Kubernetes can schedule and run application containers on clusters of physical or virtual machines. However, Kubernetes also allows developers to 'cut the cord' to physical and virtual machines, moving from a host-centric infrastructure to a container-centric infrastructure, which provides the full advantages and benefits inherent to containers. Kubernetes provides the infrastructure to build a truly container-centric development environment.

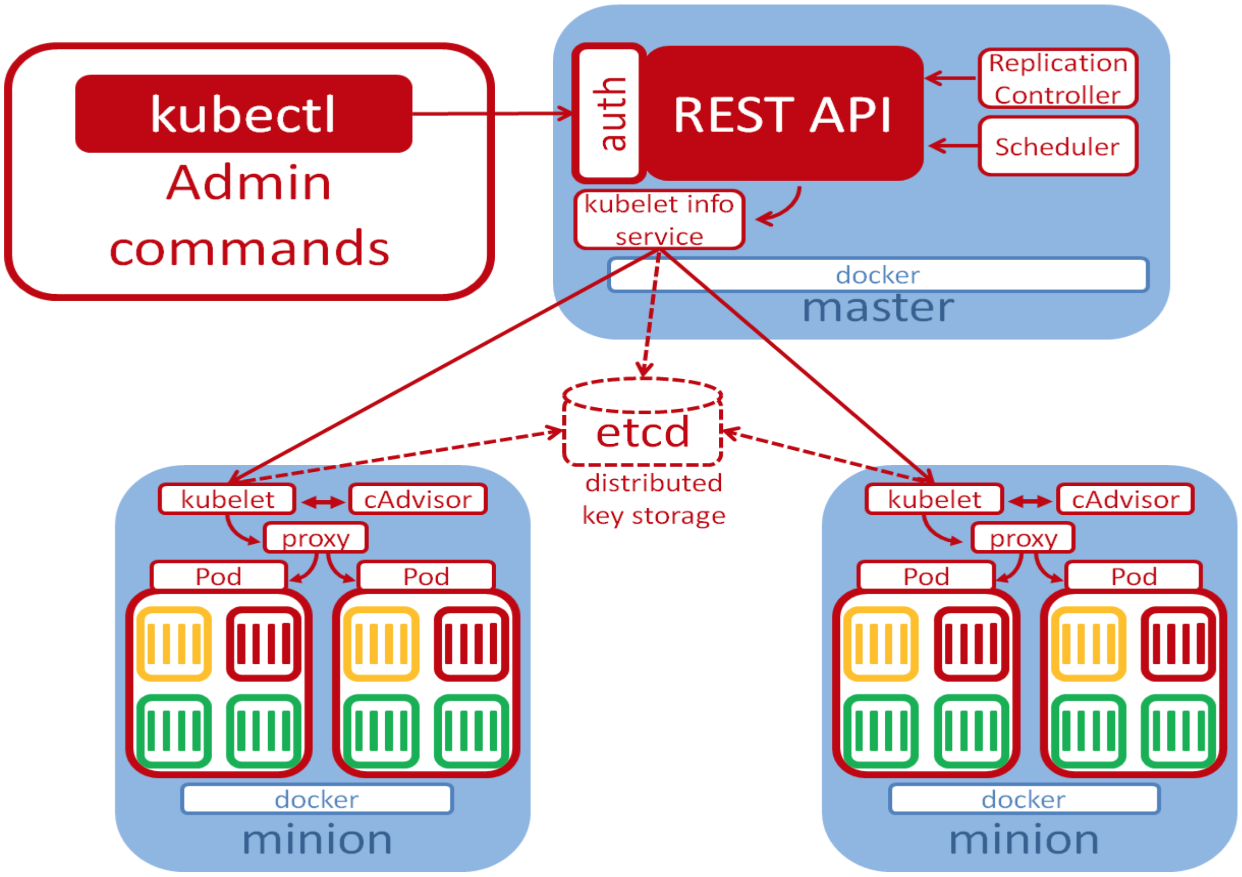
Moving parts of Kubernetes:-
PODS:-
1. May contain multiple containers
2. The life cycle of these containers bound together
3. Containers in the pod can see each other on localhost

Pods are not intended to live long. They are created, destroyed and re-created on demand, based on the state of the server and the service itself.

Service:-
As pods have a short lifetime, there is no guarantee about the IP address they are served on. This could make the communication of micro-services hard. Imagine a typical Front-End communication with Backend services. Hence K8s has introduced the concept of service, which is an abstraction on top of a number of pods, typically requiring running a proxy on top, for other services to communicate with it via a Virtual IP address. This is where you can configure load balancing for your numerous pods and expose them via a service.

Kubernetes components:-
Kubernetes consists of several parts, some of them optional, some mandatory for the whole system.
1. Master Node
2. API Server
3. Replication Controller
4. Scheduler
5. ETDC storage
6. Controller Manager
7. Worker Node
8. Docker
9. Kubelet
10. Kube proxy
11. Kubectl

Kubernetes:-High-level architecture



Master Node:-
The master node is responsible for management of Kubernetes cluster. The following components belong to the master node

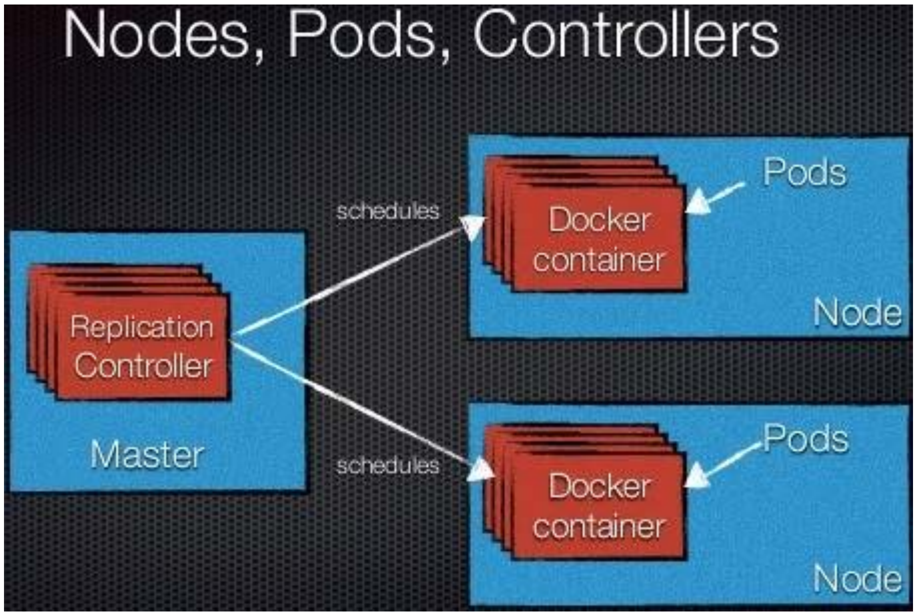
API Server:-
API server is the entry points for all REST commands to control the cluster. It processes the rest requests, validates them, and executes the bound business logic. The resulting state has to be persisted somewhere, and that brings us to the next component of the master node.

Etcd storage:-
Etcd is a simple, distributed, consistent key-value store. It's mainly used for shared configuration and service discovery. It provides a REST API for CRUD operations as well as an interface to register watchers on specific nodes, which enables a reliable way to notify the rest of the cluster about configuration changes.
Example of data stored by Kubernetes in etcd is jobs being scheduled, created and deployed pod/service details and state, namespaces, and replication information, etc.

Scheduler:-
The deployment of configured pods and services onto the nodes happens thanks to the scheduler component. The scheduler has the information regarding resources available on the members of the cluster, as well as the ones required for the configured service to run and hence is able to decide where to deploy a specific service.

Controller-manager:-
Optionally you can run different kinds of controllers inside the master node. controller-manager is a daemon embedding those. A controller uses API server to watch the shared state of the cluster and makes corrective changes to the current state to bring it to the desired one. An example of such a controller is the Replication controller, which takes care of the

number of pods in the system. The replication factor is configured by the user, and that's the controller's responsibility to recreate a failed pod or remove an extra-scheduled one. Other examples of controllers are endpoints controller, namespace controller, and service accounts controller, but we will not dive into details here.



Worker Node:-
The pods are run here, so the worker node contains all the necessary services to manage the networking between the containers, communicate with the master node, and assign resources to the containers scheduled.

Docker:-
Docker runs on each of the worker nodes and runs the configured pods. It takes care of downloading the images and starting the containers.

Kubelet:-
Kubelet gets the configuration of a pod from the API server and ensures that the described containers are up and running. This is the worker service that's responsible for communicating with the master node. It also communicates with etcd, to get information about services and write the details about newly created ones.

Kube-proxy:-
Kube-proxy acts as a network proxy and a load balancer for a service on a single worker node. It takes care of the network routing for TCP and UDP packets.

Kubectl:-
And final bit – a command line tool to communicate with API service and send commands to the master node.

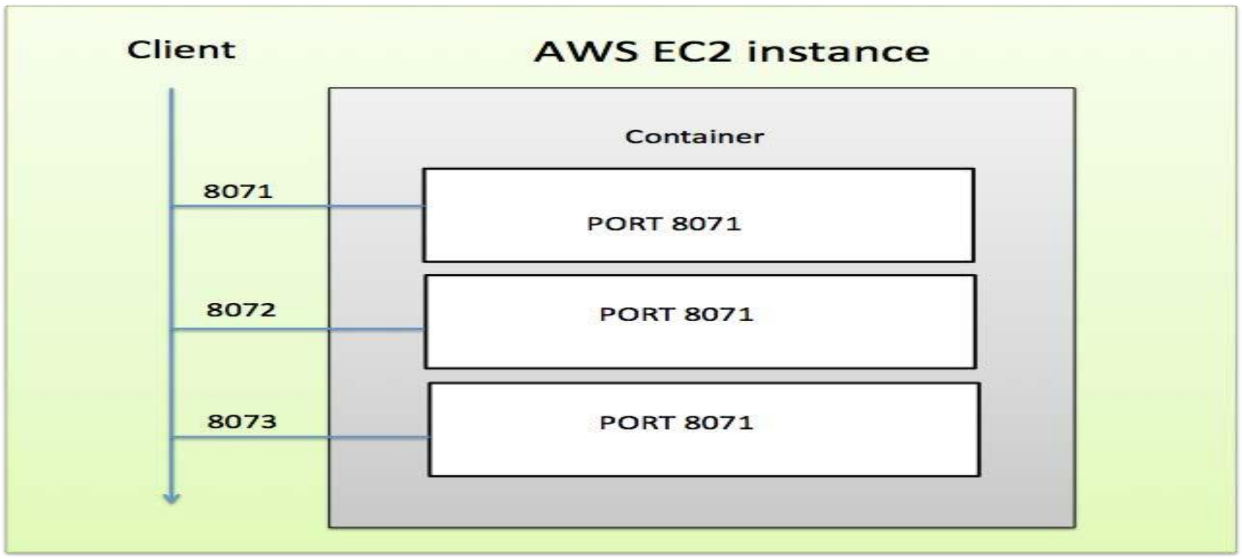
POC Results and lessons learned:-
POC using simple docker:-
Here is the example of docker file which we used in our POC. This file installs JDK1.8 mount the volume and copy the jar file as app.jar. Start the jar as the application build on using Spring boot and expose one PORT so that one can access the application using the PORT

```
FROM frolvlad/alpine-oraclejdk8:slim            // installing JDK8 from an image


VOLUME /tmp              //The VOLUME command is used to enable access from your container to a directory
on the host machine (i.e. mounting it)


ADD gs-spring-boot-docker-0.1.0.jar app.jar //copies the files from the source on the host into the
container's own filesystem at the set destination


RUN sh -c 'touch /app.jar'


ENV JAVA_OPTS=""


ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]

//ENTRYPOINT argument sets the concrete default application that is used every time a container is created


EXPOSE 8071 // Exposing port8071
```

```
$./gradlew build buildDocker -x test  //build the docker file. PORT mapping
$docker run -p 8071:8071 -t springio/gs-spring-boot-docker
```

The above comment deploys the application from the image springio/gs-spring-boot-docker. The application is running on 8071, and we can map the same port or different PORT to map it from the source host.

Here is the pictorial view of the mapping of ports. So it creates 3 containers within the same virtual machine, and each one is running with PORT 8071, and we can access 3 containers of the same application using 8071, 8072 and 8073. We need this port mapping as one host machine can’t have same port.

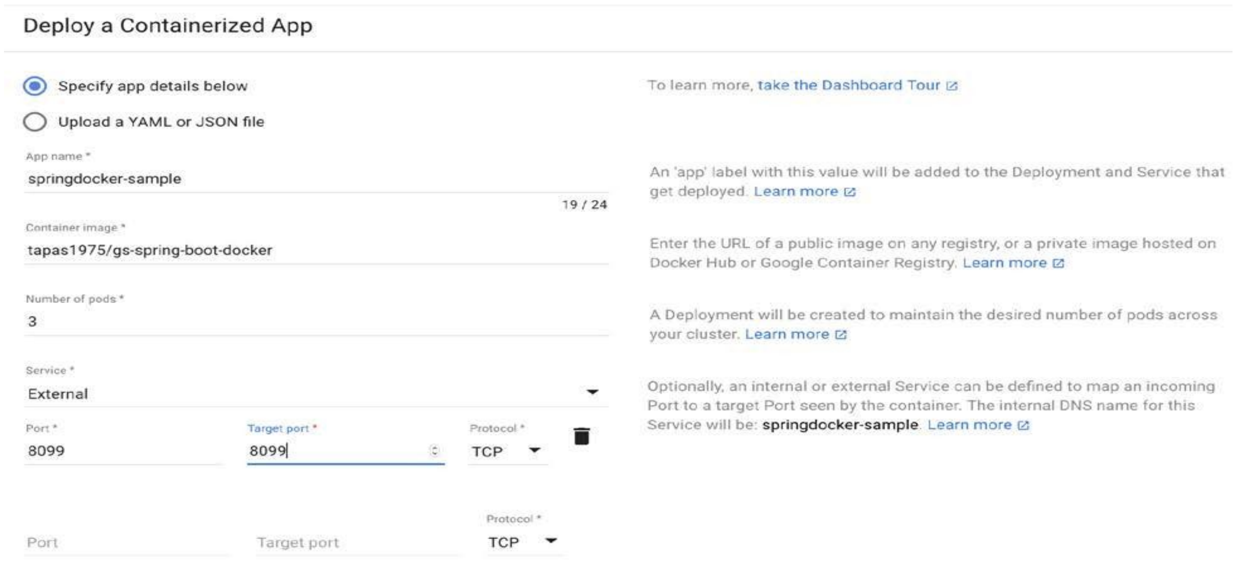


POC using Docker and Google Kubernetes:-
We put one sample code which we developed using Spring boot (Hello world) in public docker hub. We started Kubernetes cluster using minikube in mac machine.

$minikube start (To start Kubernetes) - Pre-requisite is to install docker in the machine.
$minikube service springdocker --URL (get the end point URI of the application)

Get the pods and services running

$kubectl get pods // Get the pods
$kubectl get services // Get the services
$kubectl get replicasets // Get replica sets, this is like auto-scaling group
$docker ps // Get all the processes that are running within docker container
Here is the sample screenshot of deployment UI where you can pull the code from docker hub.


Deploy a Containerized App

Specify app details below
Upload a YAML or JSON file

To learn more, take the Dashboard Tour

App name *
springdocker-sample
19 / 24

An 'app' label with this value will be added to the Deployment and Service that get deployed. Learn more

Container image *
tapas1975/gs-spring-boot-docker

Enter the URL of a public image on any registry, or a private image hosted on Docker Hub or Google Container Registry. Learn more

Number of pods *
3

A Deployment will be created to maintain the desired number of pods across your cluster. Learn more

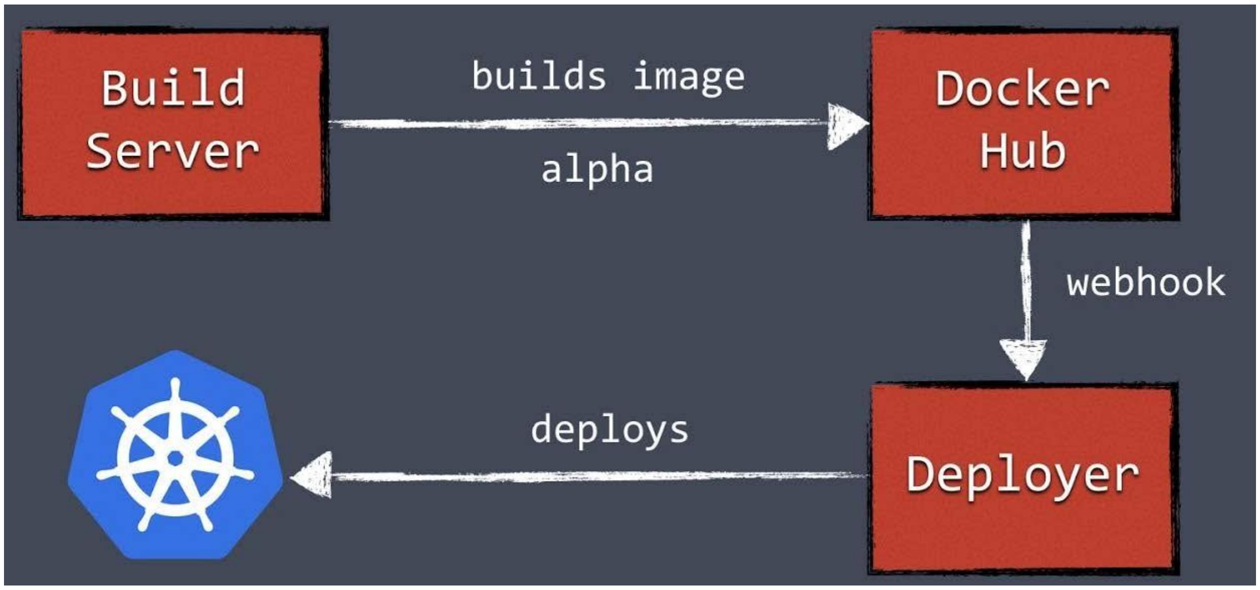
Service *
External

Optionally, an internal or external Service can be defined to map an incoming Port to a target Port seen by the container. The internal DNS name for this Service will be: springdocker-sample. Learn more

Port *	Target port *	Protocol *
8099	8099	TCP
Port	Target port	TCP

Automatic Deployment:-

With the Deployer in place, we were able to hook up deployments to a build pipeline. Our build server can, after a successful build, push a new Docker image to a registry such as Docker Hub. Then the build server can invoke the Deployer to automatically deploy the new version to a test environment. The same image can be promoted to production by triggering the Deployer on the production environment.



Rolling Update:-

In today's production deployment, almost every customer needs deployment without any downtime. Kubernetes support Blue Green deployment, and in the below section, I have described how we can do rolling deployment in Kubernetes.

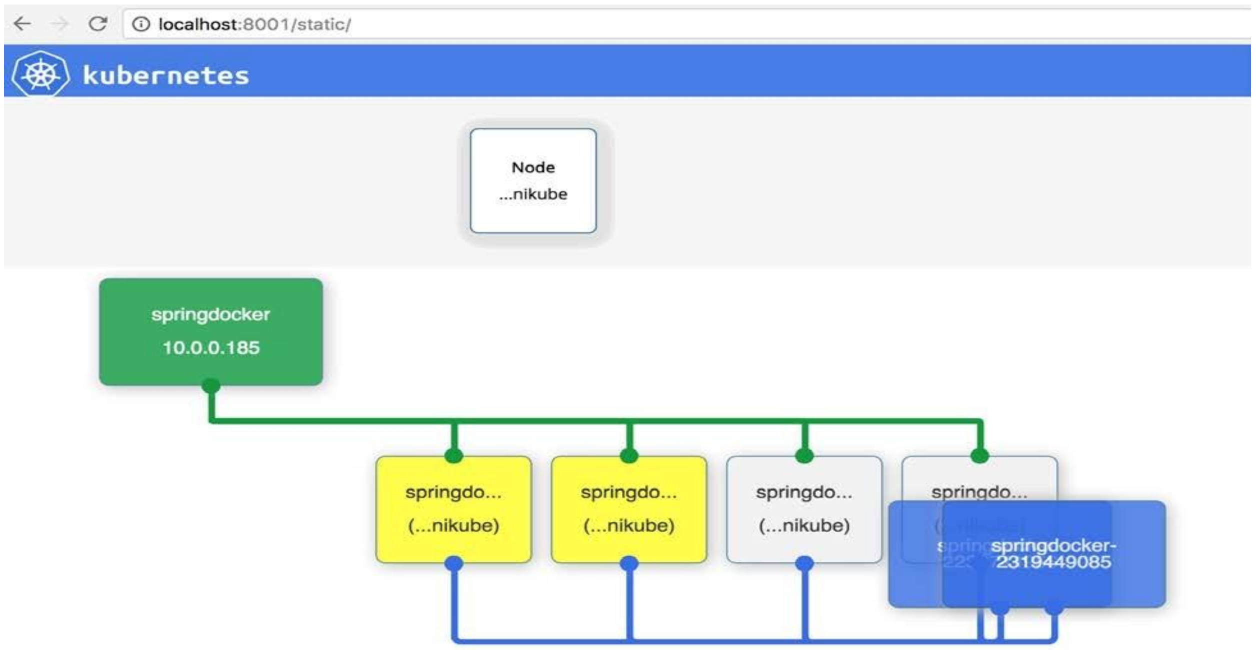
Change the code and put version 1 and build the code. Here are the steps where we modified the code, build the code to create an image, tag the image and push it to docker hub. Set the deployment to a new image.

$./gradlew build buildDocker -x test //Create docker image
$docker images //list of docker images // Push the image into docker registry$docker tag <image tag> <image>:<version> //tag it to v1

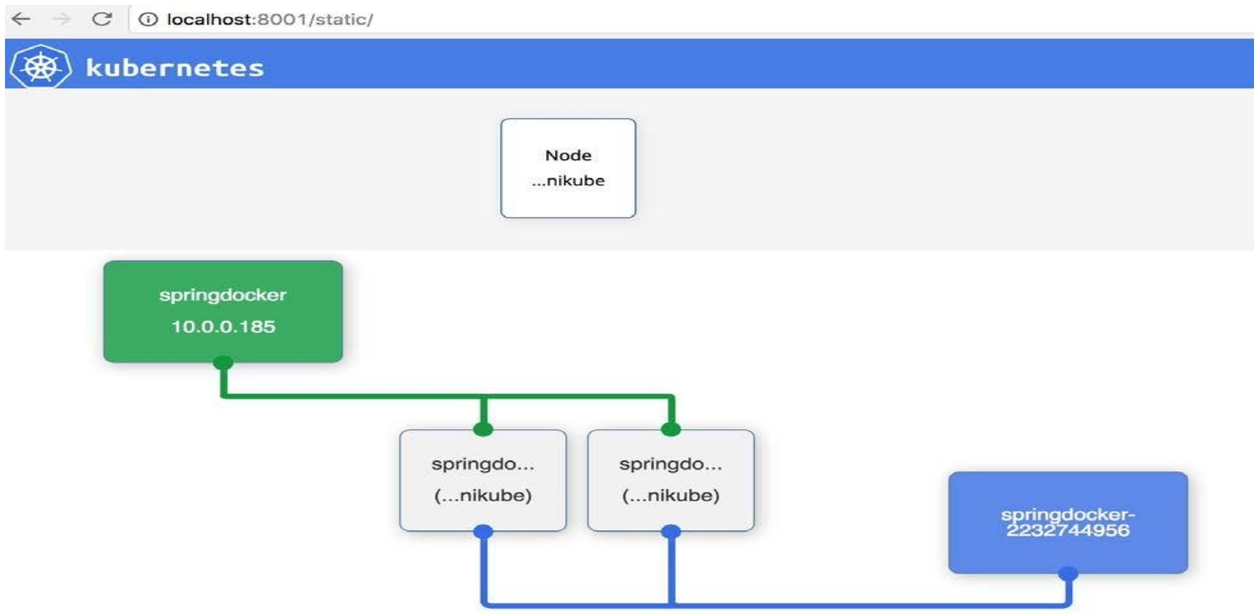
$docker login

$docker push <image>:<tag_version>

// Push the tagged image into docker registry
$kubectl set image deployment/springdocker springdocker=tapas1975/gs-spring-boot-docker:v2
//Set the deployment target to a new image (let's say v2)



Yellow color boxes have v2 is running with one new replication controller. Once it’s running, then replication controller will make it 2 pods as the scale of the application was 2. Here is the final state after deployment .
Once the deployment is done V2 pods will be active and slowly V1 pod will go away.



The above UI which we used for live monitoring. I downloaded the following UI code and modified as per our need.
https://github.com/omerio/k8s-visualizer

Download the code execute the following command

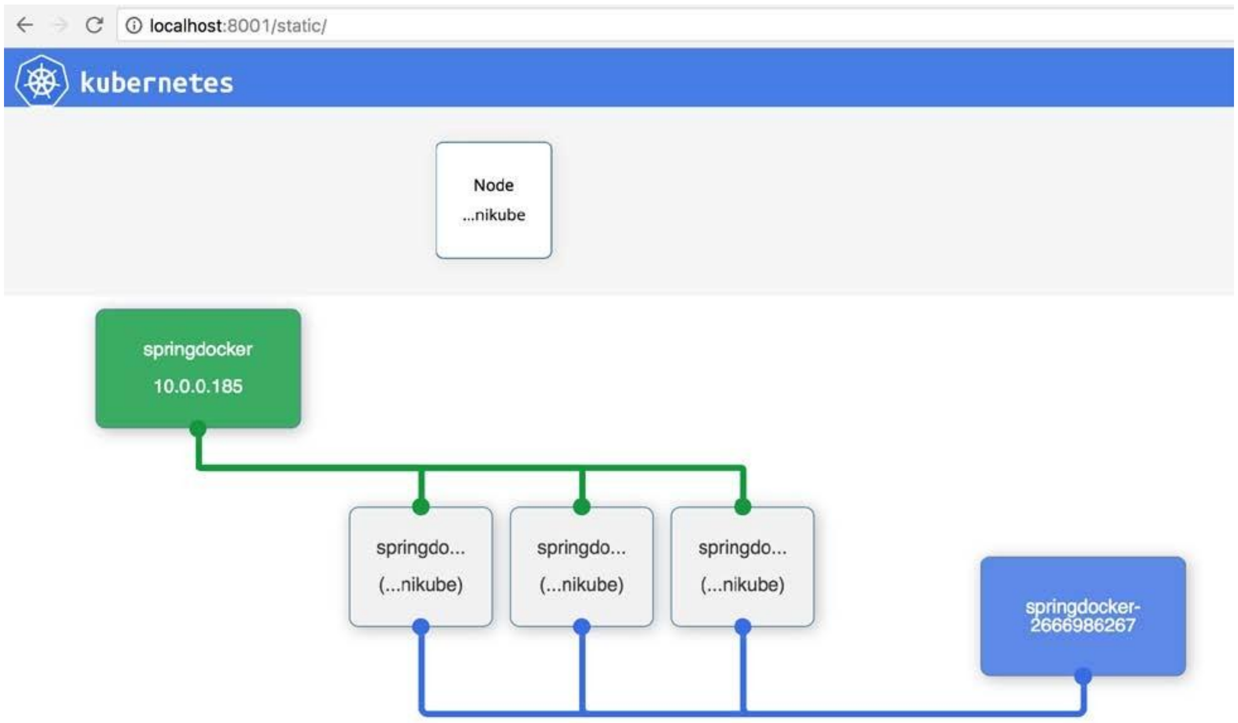
```
$cd k8s-visualizer
$kubectl proxy -w=.
```

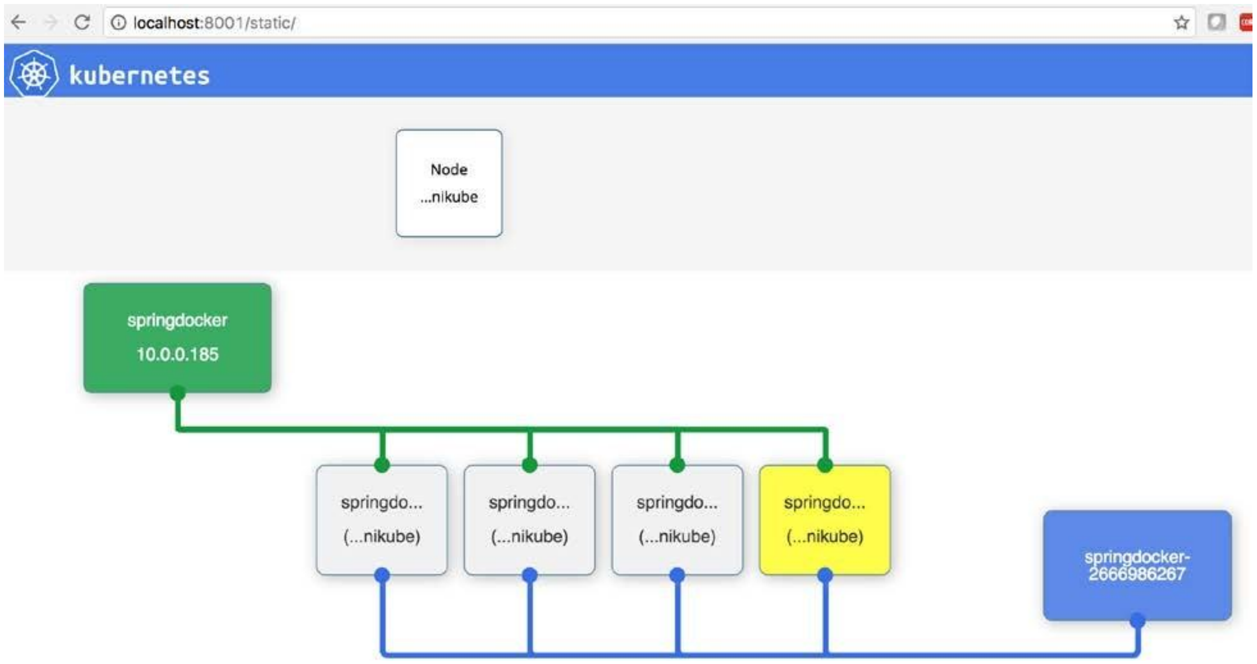
Here is the sample screen shot when we scaled the application from 3 pods to 4 pods

Here are following commands to scale anumber of pods to 4. Auto-scaling also can be done based on resource utilization of PODS.

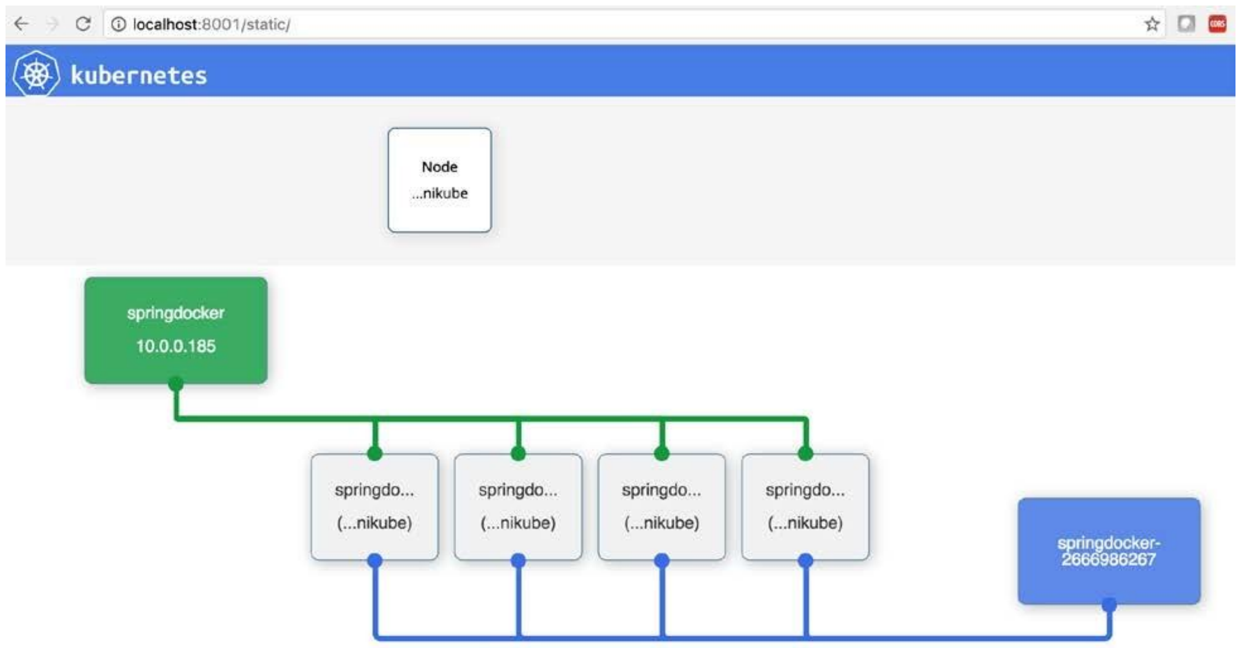
```
$kubectl config use-context minikube //Switch the context to minikube if you are locally testing this.
$kubectl scale --replicas=4 deployment springdocker // Simple scaling
```

kubectl autoscale deployment springdocker --min=2 --max=4 --CPU-percent=20 // Scaling based on CPU utilization. The minimum number of pods are 2, and a maximum number of pods can go up to 4 if CPU utilization increases more than 20 percent.



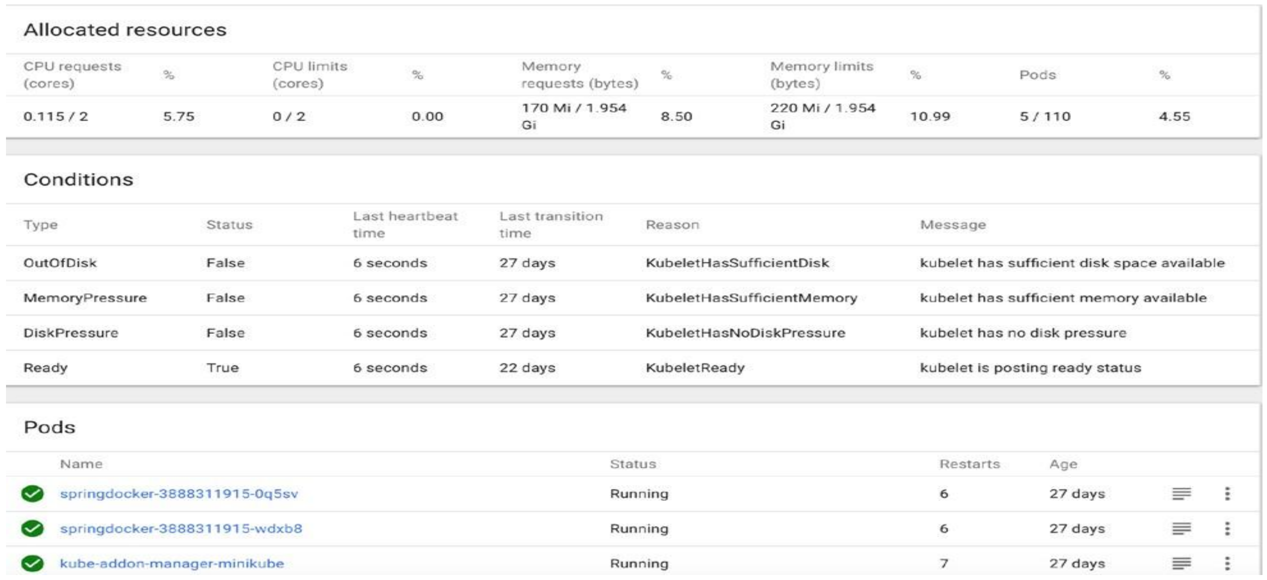


The yellow color is indicating pending pod.



White indicates a healthy node
1. Red indicates a non-healthy node
2. Grey indicates a running pod
3. Yellow indicates a pending pod
4. Green indicates a service
5. Blue indicates a replication controller/Replication Sets

Kubernetes does an excellent job of recovering when there's an error. When pods crash for any reason, Kubernetes will restart them. When Kubernetes is running replicated, end users probably won't even notice a problem. **Kubernetes recovery works very well in case of out of memory error. This recovery happens so smoothly, that DevOps team can also miss it unless there is monitoring on it.**



Allocated resources

CPU requests (cores)	%	CPU limits (cores)	%	Memory requests (bytes)	%	Memory limits (bytes)	%	Pods	%
0.115 / 2	5.75	0 / 2	0.00	170 Mi / 1.954 Gi	8.50	220 Mi / 1.954 Gi	10.99	5 / 110	4.55

Conditions

Type	Status	Last heartbeat time	Last transition time	Reason	Message
OutOfDisk	False	6 seconds	27 days	KubeletHasSufficientDisk	kubelet has sufficient disk space available
MemoryPressure	False	6 seconds	27 days	KubeletHasSufficientMemory	kubelet has sufficient memory available
DiskPressure	False	6 seconds	27 days	KubeletHasNoDiskPressure	kubelet has no disk pressure
Ready	True	6 seconds	22 days	KubeletReady	kubelet is posting ready status

Pods

	Name	Status	Restarts	Age
✓	springdocker-3888311915-0q5sv	Running	6	27 days
✓	springdocker-3888311915-wdxb8	Running	6	27 days
✓	kube-addon-manager-minikube	Running	7	27 days

Lesson learned (Anti Pattern and Valid Pattern)
This brings me to share some of the lesson learnt as part of this journey.

Layered Service Architecture:-

One common mistake people made with SOA were misunderstanding how to achieve the re-usability of services. For example, several services functioned as a data access layer (ORM) to expose table as service with the assumption that it will highly scalable. This created a physical layer which caused delivery dependency. Any service created should be highly autonomous – meaning independent of each other.

Try to look at service as one atomic business entity which must implement to achieve the desired business functionality and scalability.

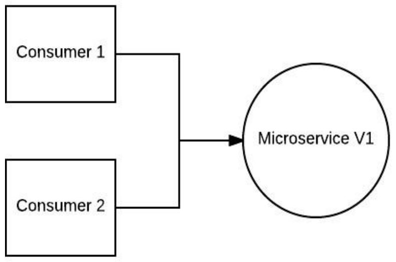
Avoid manual Configuration:-

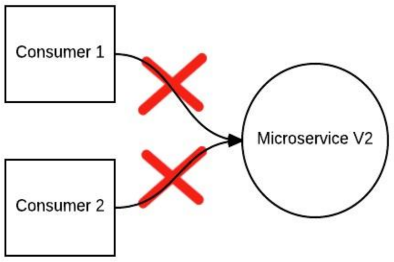
Set up the micro-service cluster and monitoring is painful activities unless one is using some framework like Kubernetes and we should have monitoring like New Relic, Datadog, heapster, etc.

Service discovery and API Gateway:-

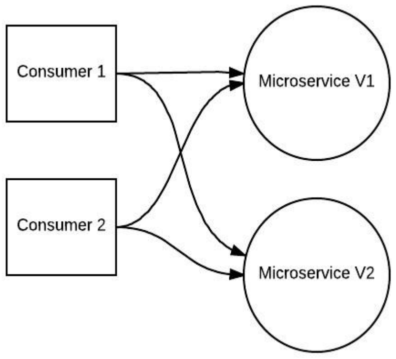
Service discovery is very important where one service can talk to each other without any IP address. Kubernetes framework provided out of the box facility like Cluster PORT, Node PORT and external load balancer for cloud provider for exposing service. AWS ECS is also providing micro-services but does not provide any out of the box service discovery feature. One can use Hasicorp Consul to achieve service discovery in AWS.

Invest in API Management solutions to centralize, manage and monitor some of the non-functional concerns and which would also eliminate the burden of consumer's managing several microservices configurations

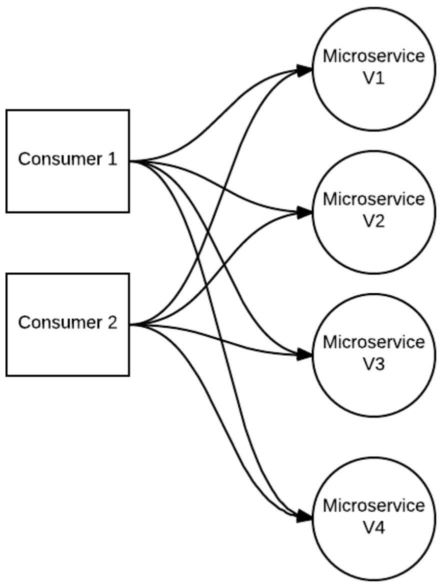




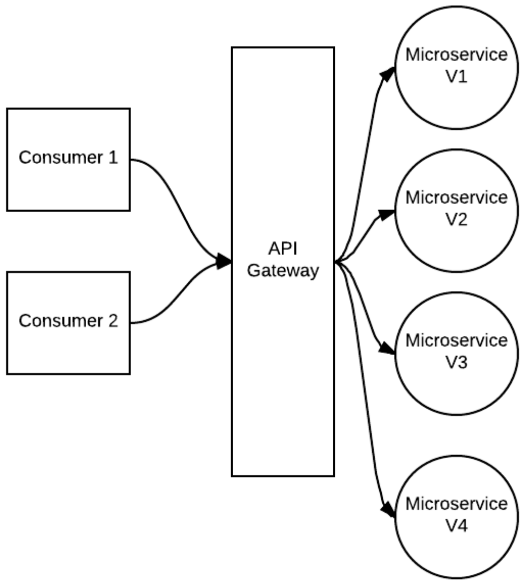
Breaking Change



Graceful Migration to V2



Anti-pattern



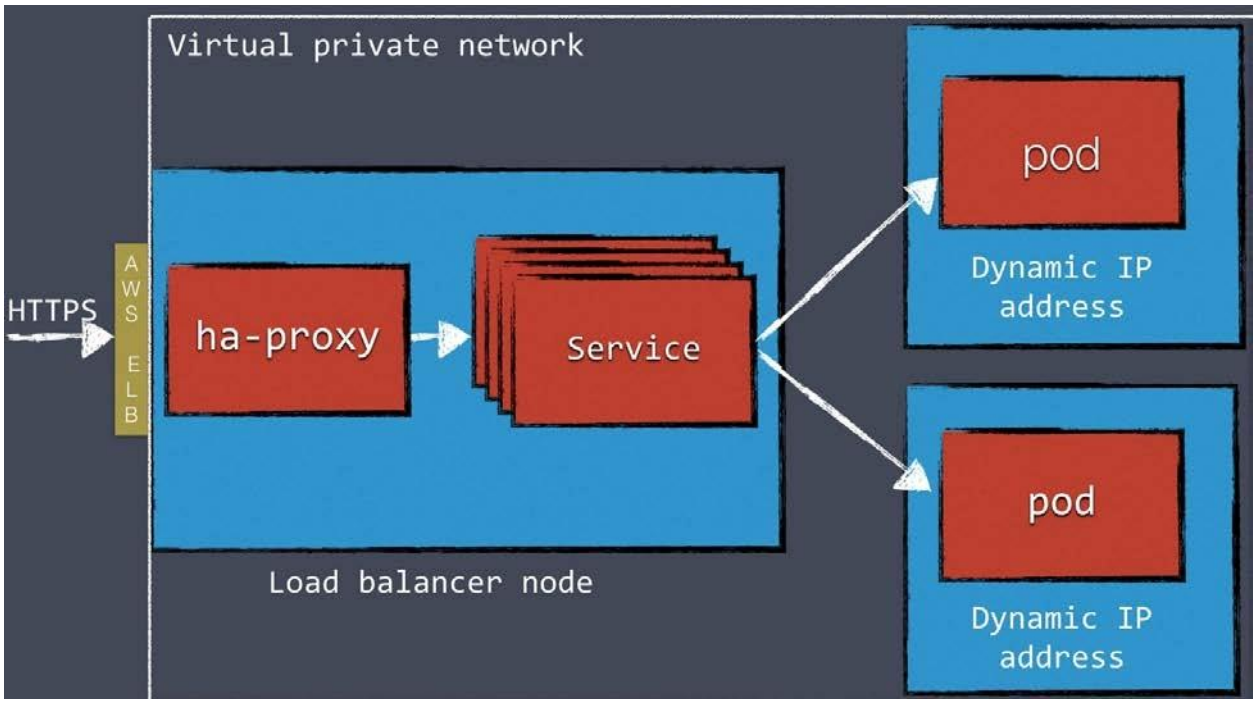
Valid Pattern

Versioning and rolling deployment:-
Have a versioning strategy that can allow the consumers a graceful migration to a higher version.

AWS load balancing:-
Here is one of the recommended approach for load balancing in **AWS**.

It's much better toapproach is to configure a load balancer such as HAProxy or NGINX in front of the Kubernetes cluster. Kubernetes clusters inside a VPC/VPN on AWS and using an AWS Elastic Load Balancer to route external web traffic to an internal HAProxy cluster. HAProxy is configured with a "back-end" for each Kubernetes service, which proxies traffic to individual pods.

This two-step load-balancer setup is mostly in response AWS ELB's fairly limited configuration options. One of the limitations is that it can't handle multiple vhosts. This is the reason for using HAProxy as well. Just using HAProxy (without an ELB) could also work, but you would have to work around dynamic AWS IP addresses on the DNS level.



Conclusion:-
There are multiple ways one can deploy container service. Some of the popular ways are to use open source frame work like Docker Swarm, Kubernetes, Mesos or cloud provided service like AWS ECS. One can choose any framework but considering above discussion points, we are recommending to use Kubernetes framework because of its key features like out of the box service discovery, portability, namespace configuration, features like config map, a secret map, ingress load balancer, local deployment support with minikube and persistence volume support. Among multiple cloud providers, AWS is now one of the popular choices for many customers.

Reference:-
1. Gonzalez-Garay, M.: The road from next-generation sequencing to personalized medicine. Pers. Med. 11(5), 523–544 (2014)
2. DePristo, M., Banks, E., et al.: A framework for variation discovery and genotyping using next-generation DNA sequencing data. Nature Genet. 43(5), 491–498 (2011)
3. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows and wheeler transform. Bioinformatics 25(14), 1754–1760 (2009)
4. Wang, K., Li, M., Hakonarson, H.: Annovar: functional annotation of genetic variants from high-throughput sequencing data. Nucleic Acids Res. 38(16), e164 (2010)

5. Guerrero, G., Wallace, R., Vázquez-Poletti, J., et al.: A performance/cost model for a cuda drug discovery application on physical and public cloud infrastructures. Concurrency Comput.: Pract. Experience 26(10), 1787–1798 (2014)
6. Folarin, A., Dobson, R., Newhouse, S.: NGSeasy: a next-generation sequencing pipeline in Docker containers. F1000Research 4, 997 (2015)
7. B. Kepes, "VoltDB Puts the Boot into Amazon Web Services Claims IBM Is Five Times Faster", Forbes, Aug 2014, [online] Available: www.forbes.com/sites/benkepes/2014/08/06/voltdb-puts-the-boot-into-amazon-web-services-claims-ibm-5-faster
8. J. Petazzoni, "Linux Containers (LXC) Docker and Security", Jan. 2014, [online] Available: www.slideshare.net/jpetazzo/linux-containers-lxc-docker-and-security.
9. Thota, S., 2017. Big Data Quality. Encyclopedia of Big Data, pp.1-5. https://link.springer.com/referenceworkentry/10.1007/978-3-319-32001-4_240-1
10. [10] B. Butler, "Containers: Buzzword du Jour or Game-Changing Technology?", NetworkWorld, Sept. 2014, [online] Available: www.networkworld.com/article/2601925/cloud-computing/container-party-vmware-microsoft-cisco-and-red-hat-all-get-in-on-app-hoopla.html.
11. Garzon, J., Lopéz-Blanco, J., Pons, C., et al.: Frodock: a new approach for fast rotational protein-protein docking. Bioinformatics 25(19), 2544–2551 (2009)
12. Metz, C. 2015. Google is 2 billion lines of code—and it's all in one place. Wired (September); http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/
13. Burrows, M. 2006. The Chubby lock service for loosely coupled distributed systems. Symposium on Operating System Design and Implementation (OSDI), Seattle, WA.
14. Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., Wilkes, J. 2015. Large-scale cluster management at Google with Borg. European Conference on Computer Systems (EuroSys), Bordeaux, France.
15. Hadoop For Dummies Dirk deRoos, Paul C. Zikopoulos, Bruce Brown, Rafael Coss, and Roman B. Melnyk, John Wiley & Sons, Inc., 1st edition 2014.
16. https://x-team.com/blog/introduction-kubernetes-architecture/
17. https://github.com/omerio/k8s-visualizer
18. https://amdatu.org/generaltop/gettingstarted/
19. https://www.linux.com/learn/rolling-updates-and-rollbacks-using-kubernetes-deployments
20. Simanta Shekhar Sarmah, Data Migration, Science and Technology, Vol. 8 No. 1, 2018, pp. 1-10. doi: 10.5923/j.scit.20180801.01.