# Distributed Broker Network System in Cloud

Team 2: Sharad, Pratikshya, Lakshmipriyadarshini

# Motivation

- In the last decade, cloud computing has been a major trend in the industry.
- Cloud Computing is a virtualized computer system that contains all software and applications needed for businesses and enables users to access the applications all over the world through a web browser.
- It provides a different model of operation in which the service providers are liable for hardware resources, upgrade, maintenance, failover, backups, security, etc.
- There are no extra investments on the server, Instead, users can only pay for the services on demand.

# Goals

- Our project's main aim is to containerize the Distributed Brokers Nodes in a highly scaled, PubSub based Emergency notification system and deploy them on the cloud.

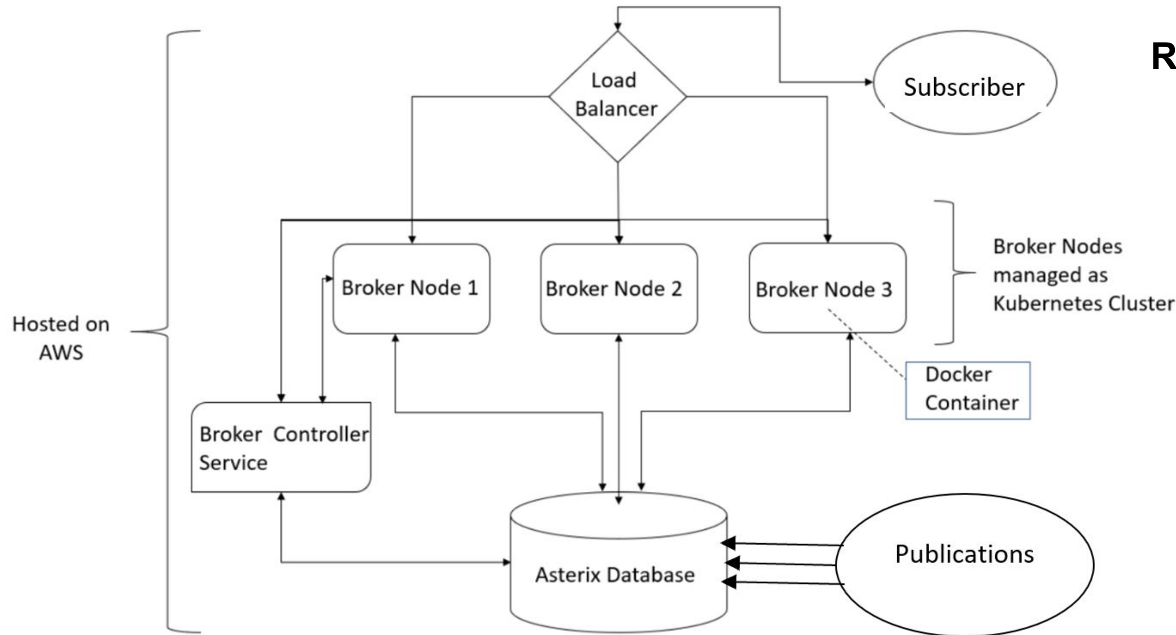
- This PubSub based Emergency notification system is based on Big Active Data that uses BAD AsterixDB as its database. The broker Nodes are connected with the users providing them notifications about various emergencies based on their subscription.

- These Broker Nodes are managed as Kubernetes cluster to ensure availability and load balancing.

# Related Work

1. Research on Distributed Brokers
    a. Stratos: Cloud Broker Service
        i. Automatic provisioning based on decision making algorithms
        ii. Has an application manager [Run time management] and cloud manager [BCS]
2. Research on Kubernetes
    a. Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework
        i. Develop a clustered system of five Raspberry Pi (RPi) embedded boards by utilizing Docker containers and the Kubernetes cluster framework
3. Research on Asterisk Database
    a. Data Replication and Fault Tolerance in AsterixDB
    b. A BAD Thesis: The Vision, Creation, and Evaluation of a Big Active Data Platform

# Architecture:



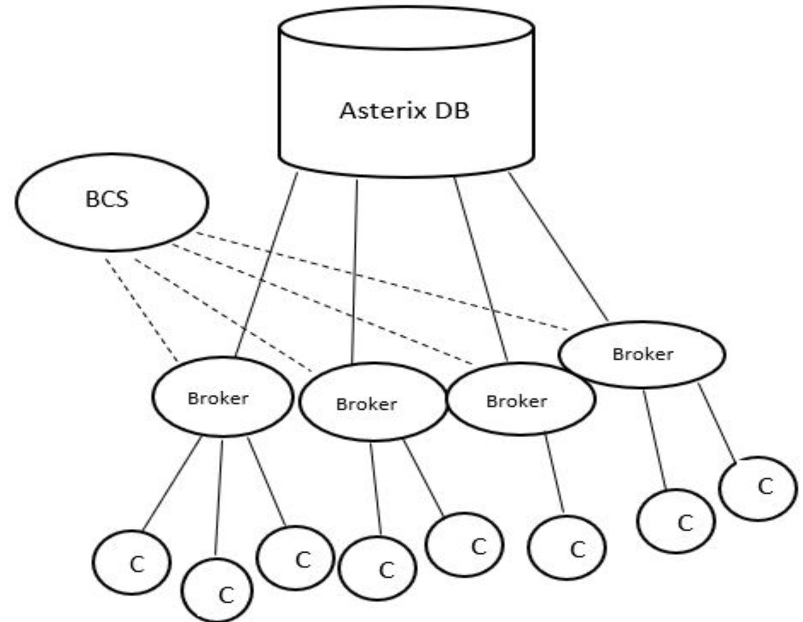

**Motivation**: Containerization of Broker Nodes in Big Active Data system and implementing them in cloud.

**Reason:**

- Kubernetes provides a container-centric management environment.
- Orchestrates computing, networking, and storage infrastructure on behalf of user workloads.
- Ensures availability and load balancing in the cloud cluster.

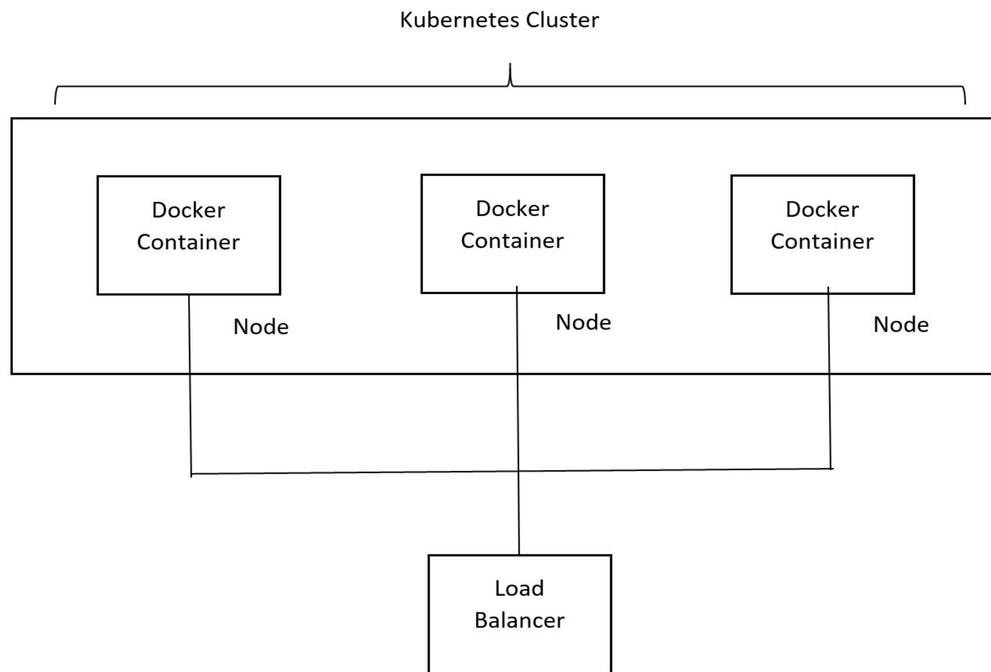
# Design Flow of the Broker Network

- BCS and Asterix DB started first
- Brokers are created
- Broker Registration with BCS
- Clients contacts BCS for Broker URL
- BCS informs Broker URL
- Client logs into Broker
- (Un)Subscribes randomly
- BCS periodically updates load
- Excess load = Migration to a new broker
- Client logs out of current broker
- Client logs into new broker with URL from BCS



# Design Specifics

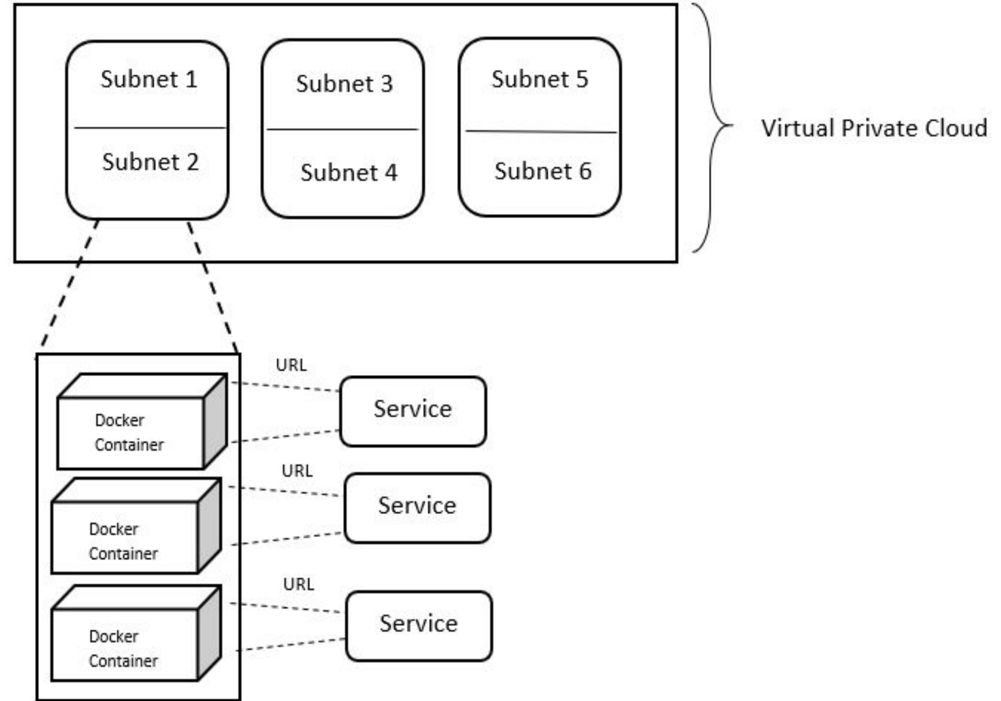
- Broker nodes are provisioned as separate docker containers (deployment).
- These docker containers collectively form the Kubernetes cluster.
- The runtime of docker containers is made available by exposing the deployment (NodePort/Load Balancer).



# Deployment of Service and its access

**Steps:**

- Creation of Virtual Private Cloud (EKS Cluster)
- Creation of subnets (different locations)
- Creation of Docker Containers
- Deployment of services
- Access Service via URLs



# Experiment Design

**Steps:**

- AWS-EC2 instance was used to build the docker container and Google Cloud Platform (Docker container registry) was used to house the docker images.
- Broker nodes as Docker containers are deployed as a replica set of three and are extended as services.
- The service is a load balancer which balances the load on the docker containers depending on the user access.
- Deployment and services are executed as .yaml files which include the specifications such as replica set, docker image URL, the port on which the docker container should run, port exposed to the outside world, type of service, and node selector.

# Experiment Results

```
C:\Users\shara\Desktop\project>kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
brokerdeployment1   3         3         3            3           1h

C:\Users\shara\Desktop\project>kubectl get services
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP                                                                 PORT(S)          AGE
brokerservice   LoadBalancer   10.100.159.25   a04761e3e8f0211e99c280257bc7e18a-1994244408.us-west-2.elb.amazonaws.com   5000:30537/TCP   33m
kubernetes      ClusterIP      10.100.0.1      <none>                                                                      443/TCP          140d
```

The BCS was first implemented as the Load Balancer. Using **kubectl get service** command we can get the port on which the load balancer is running along with the protocol used. It also shows that the brokers are managed as a Kubernetes cluster with the IP 10.100.0.1: 443.

```
C:\Users\shara\Desktop\project>kubectl get pods
NAME                                 READY   STATUS    RESTARTS   AGE
brokerdeployment1-5ddc79bdf7-q4lqq   1/1     Running   0          49m
brokerdeployment1-5ddc79bdf7-vkvh4   1/1     Running   0          49m
brokerdeployment1-5ddc79bdf7-wpl5g   1/1     Running   0          52m
```

The deployed broker nodes status can be seen that are running as Docker Containers. We can see how long they have been running, how many times they have restarted, etc.

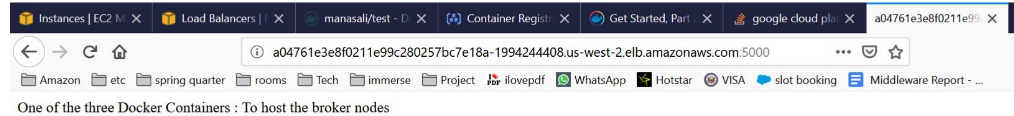
# Experiment Results Cont.

```
[root@ip-192-168-83-12 ec2-user]# docker ps
CONTAINER ID        IMAGE                                                              COMMAND                  CREATED
     STATUS              PORTS               NAMES
17d11e581516        b9e1c43c93da                                                       "/bin/sh -c 'pytho..."   45 minutes ago
     Up 45 minutes                           k8s_brokerdeployment1_brokerdeployment1-5ddc79bdf7-q4lqq_default_f43fbb53-8f01-11e
9-9c28-0257bc7e18ac_0
0abb896bd12b        602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/pause-amd64:3.1   "/pause"                 45 minutes ago
     Up 45 minutes                           k8s_POD_brokerdeployment1-5ddc79bdf7-q4lqq_default_f43fbb53-8f01-11e9-9c28-0257bc7
e18ac_27
75e8d6062674        b9e1c43c93da                                                       "/bin/sh -c 'pytho..."   About an hour a
go   Up About an hour                        k8s_brokerdeployment1_brokerdeployment1-5ddc79bdf7-vkvh4_default_f13940cc-8f01-11e
9-9c28-0257bc7e18ac_0
cd7c5802bc10        602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/pause-amd64:3.1   "/pause"                 About an hour a
go   Up About an hour                        k8s_POD_brokerdeployment1-5ddc79bdf7-vkvh4_default_f13940cc-8f01-11e9-9c28-0257bc7
e18ac_0
118b607d8324        gcr.io/middleware-243721/images                                    "/bin/sh -c 'pytho..."   About an hour a
go   Up About an hour                        k8s_brokerdeployment1_brokerdeployment1-5ddc79bdf7-wpl5g_default_7d58e11b-8f01-11e
9-9c28-0257bc7e18ac_0
9b434d67bdb5        602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/pause-amd64:3.1   "/pause"                 About an hour a
go   Up About an hour                        k8s_POD_brokerdeployment1-5ddc79bdf7-wpl5g_default_7d58e11b-8f01-11e9-9c28-0257bc7
e18ac_19
```

We can see the details of each running replica set of the 3 docker containers with the names of the containers, their creation time, their status, and on which port are they running, etc.

# Experiment Results -Accessing the load balancer and Fault tolerance

**Accessing the load balancer** will direct the traffic to one of the docker containers provisioned.



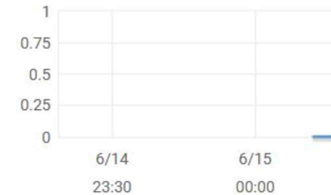

**Killing one of the docker containers** in a kubernetes cluster will automatically initiate another docker container with zero downtime supported from the elastic load balancer as well. The following snippet demonstrates this where the terminating node (52m) immediately leads to the spawning of the new broker deployment (15s).

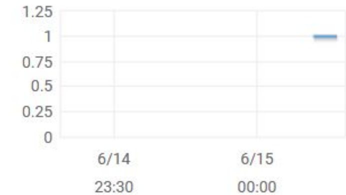
```
C:\Users\shara\Desktop\project>kubectl delete pod brokerdeployment1-5ddc79bdf7-q4lqq
pod "brokerdeployment1-5ddc79bdf7-q4lqq" deleted

C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>kubectl get pods
NAME                                 READY   STATUS        RESTARTS   AGE
brokerdeployment1-5ddc79bdf7-q4lqq   1/1     Terminating   0          52m
brokerdeployment1-5ddc79bdf7-rcqqt   1/1     Running       0          15s
brokerdeployment1-5ddc79bdf7-vkvh4   1/1     Running       0          53m
brokerdeployment1-5ddc79bdf7-wpl5g   1/1     Running       0          56m
```

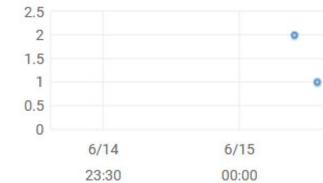
# Experiment Results -Performance Metric

**Load Balancing Metric:**

a. *Unhealthy Hosts*: The number of unhealthy (after exceeding unhealthy threshold) instances registered with the load balancer.
b. *Healthy Hosts*: The number of healthy instances registered with the load balancer. (passing the timely health check)
c. *Average Latency*: The total time elapsed, in seconds, from the time the load balancer sent the request to a registered instance until the instance started to send the response headers.
d. *Requests*: The number of requests completed or connections made during the specified interval (1 or 5 minutes).

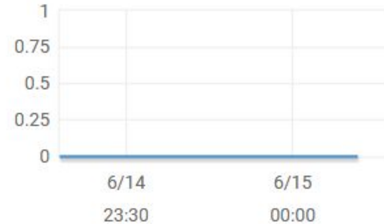


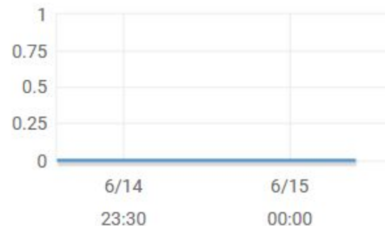



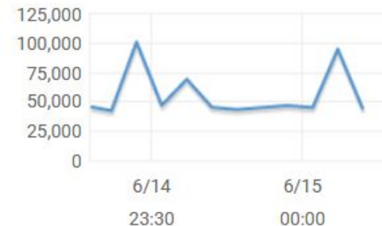


# Experiment Results -Performance Metric

The figure shows the performance metrics such as CPU utilization, disk operations, and network utilization of the chosen EC2 instance for deployment.







# Conclusion

In this project, We converted the broker nodes into docker containers and deployed and managed them as Kubernetes clusters.

- We explored how a distributed broker node can make use of cloud services to become more available and scalable.

- We also could implement load-balancing and fault tolerant among broker nodes.

# Future Work

We consider the following as a possible future work:

a. Migrate Asterix Database from its traditional deployment into AWS.

b. Explore other functions such as failover handling, migration handling, and handoff mechanisms among brokers.

c. Implement a decentralized intelligent handling mechanism that could predict failure and take necessary migrations and failover handling.

Thank You