SERVICE ORIENTED ARCHITECTURES

ACIT3855 – WINTER 2024

AGENDA – LESSON 11

- Quiz 10
- Assignment 2
- Topics:
  - Scaling in Docker Compose
  - Reverse Proxy and Load Balancing
  - Deployment Patterns
  - Balanced Consumers
- Lab 11 – Final Lab Demo

QUIZ 10

- Online on D2L, Open Book
- Your have 15 minutes to complete it
- We'll review any "problem" questions at the end

COURSE SCHEDULE

Week	Topics	Notes
1	• Services Based Architecture Overview • RESTful APIs Review	Lab 1
2	• Microservices Overview • Edge Service	Lab 2, Quiz 1
3	• Database Per Service • Storage Service (SQLite)	Lab 3, Quiz 2
4	• Logging, Debugging and Configuration • Storage Service (MySQL)	Lab 4, Quiz 3
5	• RESTful API Specification (OpenAPI) • Processing Service	Lab 5, Quiz 4
6	• Synchronous vs Asynchronous Communication • Message Broker Setup, Messaging and Event Sourcing	Lab 6, Quiz 5
7	• Deployment - Containerization of Services *Note: At home lab for Monday Set*	Lab 7, Quiz 6 (Sets A and B) , Assignment 1 Due
8	• Midterm Week	Midterm Review Quiz
9	• Dashboard UI and CORS	Lab 8, Quiz 7
10	• Spring Break	No Class
11	• Issues and Technical Debt	Lab 9, Quiz 8
12	• Deployment – Centralized Configuration and Logging	Lab 10, Quiz 9
13	• Deployment – Load Balancing and Scaling *Note: At home lab for Monday Set*	Lab 11, Quiz 10 (Sets A and B)
14	• Final Exam Preview	Quiz 10 (Set C), Assignment 2 Due
15	• Final Exam	

REMAINING WORK

- Lab 11 – Reverse Proxy, Load Balancing and Scaling
  - Setting up a Reverse Proxy with NGINX and scale up your Receiver Service
  - Demo due before the end of next class
- Assignment 2 – Developing a Event Logger Service
  - Connects to a new Kafka topics
  - Other services – Receiver, Storage and Processing connect to the same new topic and publish event log messages
  - Consumes the event log messages from the other services
  - Stores the event log messages in a datastore (JSON or SQLite)
  - Has a simple GET endpoint
  - Must be containerized, deployed and running with your other services
  - Due for demo by end of the last class

SCALING

- Having multiple instances of our services to high number of concurrent requests
- For example:
  - Multiple Receiver services to handle a large number of incoming events
  - Multiple Storage services to handle processing a large number of received events

- Ideally we can dynamically scale up and down to respond to changes in load to save costs, especially for Cloud deployments

SCALING – DOCKER COMPOSE

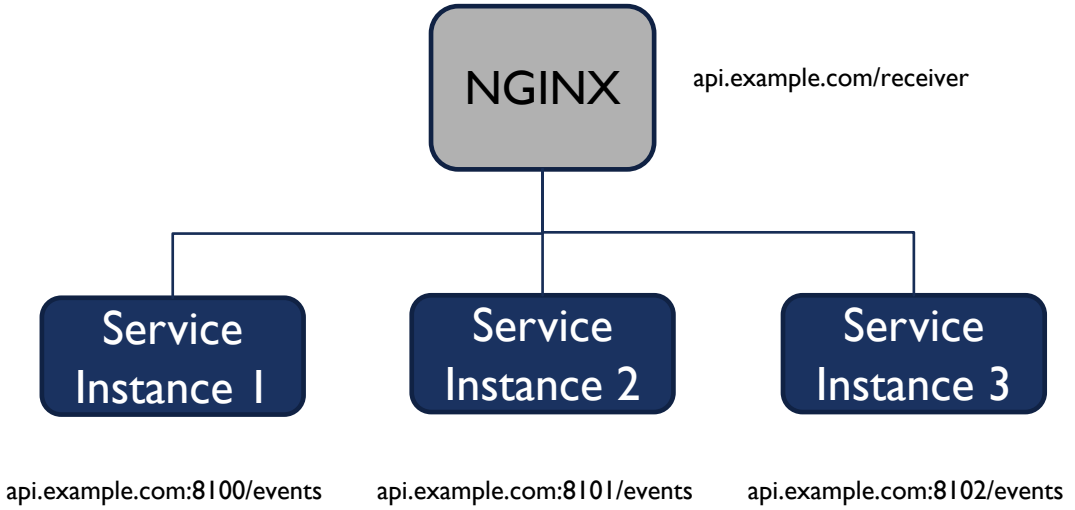
- Docker Compose lets you manually set the scaling of individual services
- On initial start-up
  - docker-compose up -d --scale storage=2 --scale receiver=3
  - Instantiates 1 additional storage container (storage_1 and storage_2)
  - Instantiates 2 additional receiver container (receiver_1, receiver_2, receiver_3)
- To adjust scaling run docker-compose up -d again
  - docker-compose up -d --scale storage=3 (increases to 3 storage containers)
  - docker-compose up -d --scale receiver=2 (decreases to 2 receiver containers)

SCALING – RESTFUL APIS

- If we scale services with a RESTful API, we now have multiple endpoints for that API based on the port on which the service is running. For example:
  - http://api.example.com:8100/events
  - http://api.example.com:8101/events
  - http://api.example.com:8102/events
- We can use a reverse proxy and software load balancer to reduce this to one endpoint (on a different port, in this case the default port 80):
  - http://api.example.com/receiver/events

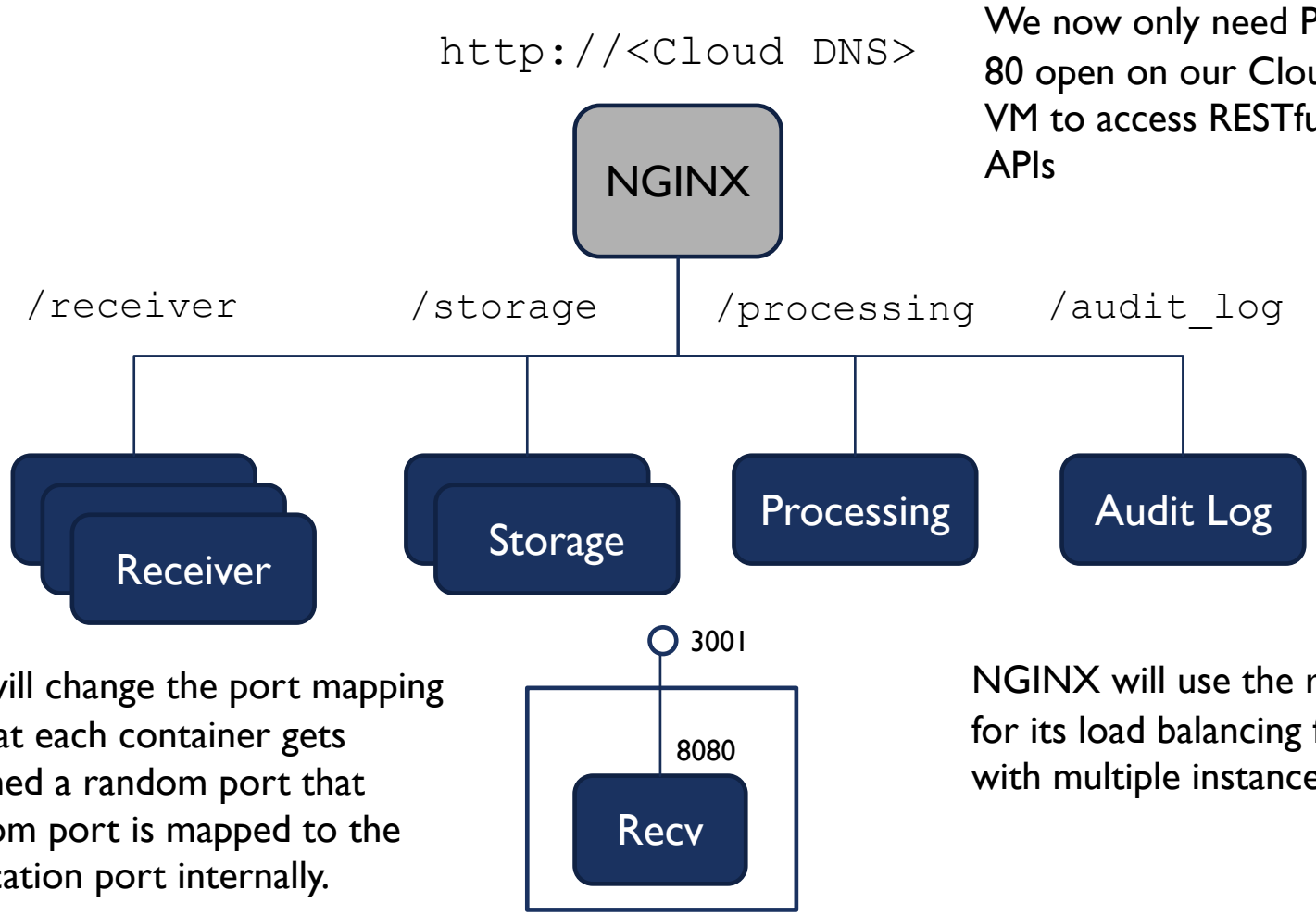
NGINX – REVERSE PROXY AND LOAD BALANCING

- NGINX is a web application server, similar to Apache
- It can act as a reverse proxy, proving a single endpoint that maps to multiple services based on the context path. For example:
  - /receiver to map to the receiver service(s)
  - /storage to map to the storage service(s)
  - /processing to map to the processing service(s)
  - /audit_log to map to the audit_log service(s)
- It can also act as a software load balancer when there are multiple instance of the same service using various algorithms, like Round Robin



NGINX – LAB CONFIGURATION

We will create a Docker Network to enable communication between NGINX and the services internally.



We now only need Port 80 open on our Cloud VM to access RESTful APIs

We will change the port mapping so that each container gets assigned a random port that random port is mapped to the application port internally.

NGINX will use the random ports for its load balancing for services with multiple instances.

DOCKER NETWORKING

- We will be creating an internal Docker Network on which NGINX can communicate with the services. In docker-compose.yml:

```
networks:
  api.network:
```

- Then for each service:

```
 receiver:
    image: receiver
    ports:
      - "8080"
    networks:
      - "api.network"
```

- And for NGINX:

```
ports:
  - "80:80"
networks:
  - "api.network"
```

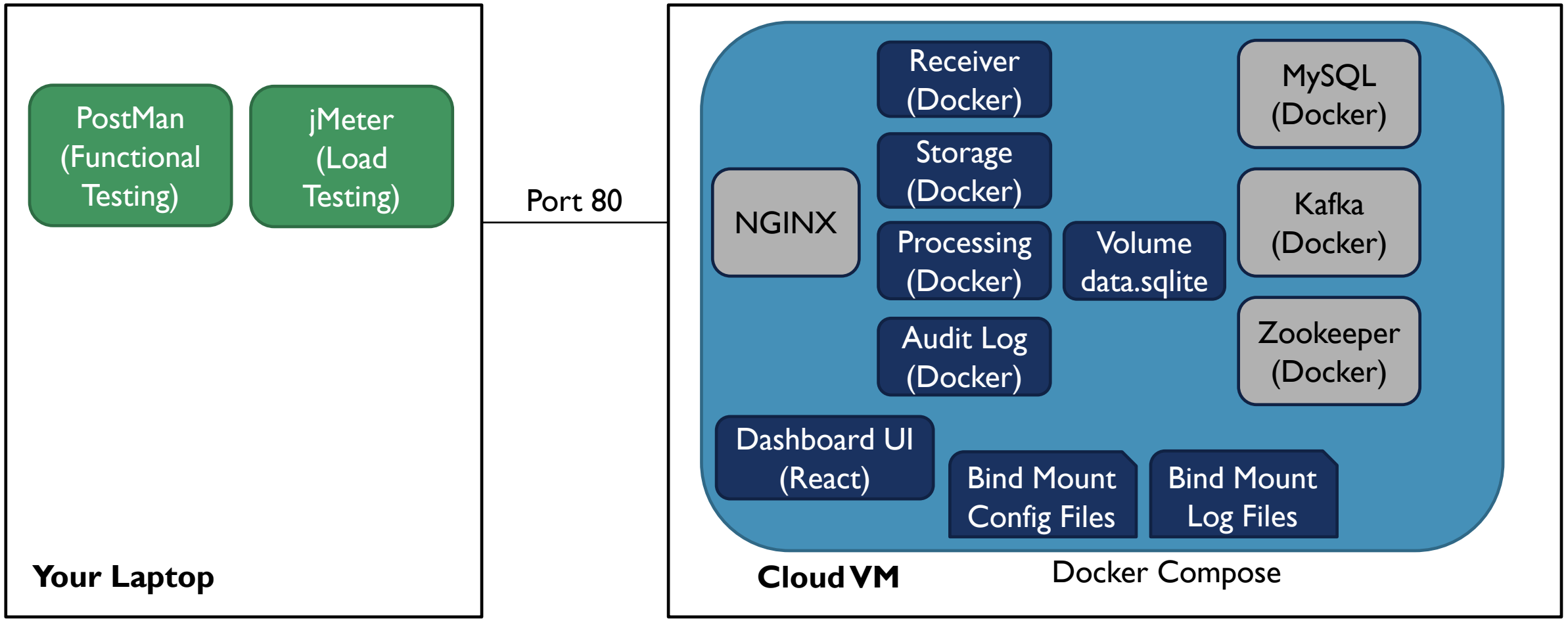
NGINX CONFIGURATION

```
user nginx;
# can handle 1000 concurrent connections
events {
    worker_connections   1000;
}
# forwards http requests
http {
        # http server
        server {
              # listens the requests coming on port 80
              listen 80;
              access_log  off;
              # / means all the requests on /receiver will be forwarded to receiver service
              location /receiver {
                # resolves the IP of receiver using Docker internal DNS
                proxy_pass http://receiver:8080;
              }

              ...

              location /audit_log {
                proxy_pass http://audit_log:8110;
              }
        }
}
```

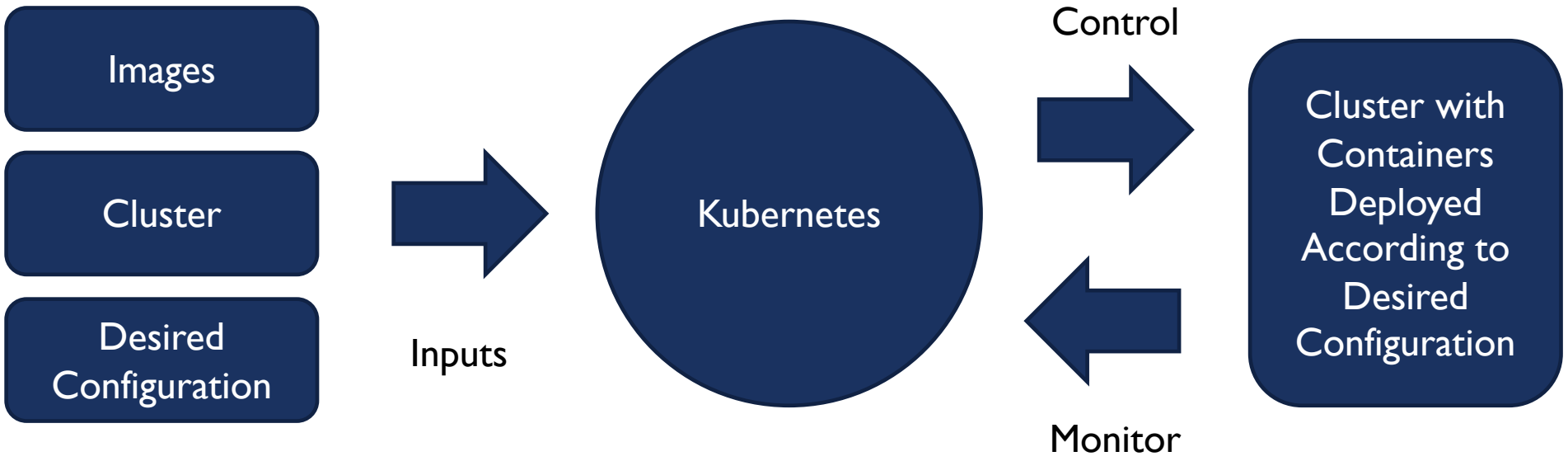
LAB 11 TEST ENVIRONMENT



WHAT IS KUBERNETES

- **Container orchestration** automates the deployment, management, scaling, and networking of **containers**.
- Kubernetes (or k8s or OKD) is probably the most popular container orchestration engines
- Much more dynamic than Docker and Docker Compose:
  - Service discovery and load balancing built in
  - High-availability deployments
  - Can be deployed over a cluster of VMs
  - Automatic scaling and self-healing
  - Secret management

CONTAINER ORCHESTRATION - KUBERNETES



A **Kubernetes cluster** is a set of node machines (physical or VMs) for running containerized applications.

CONTAINER ORCHESTRATION - KUBERNETES

Inputs:

- Images – Docker images from an artifact repository.
- Cluster – Group of physical servers or VMs. At a minimum, a Kubernetes cluster contains a worker node and a master node.
- Configuration – Yaml file that declares the configuration of the applications to be deployed.

Control:

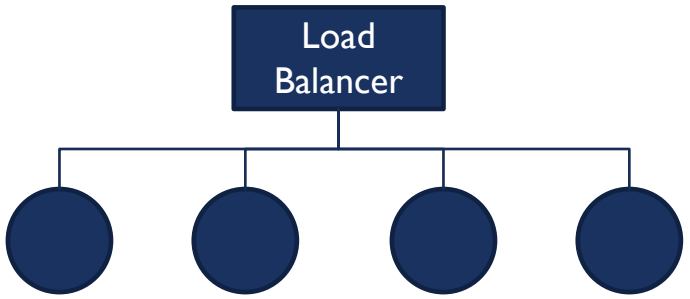
- Kubernetes deploys the images as containers to the cluster in accordance with the configuration.
- kubectl is the Command Line Interface to control a Kubernetes cluster

Monitor:

- Kubernetes monitors the containers on the cluster and adjusts as necessary. This is typically done by the master node.
- These adjustments may include:
  - Restoring the cluster back to the desired configuration if there is a fault (i.e., an application or node goes down)
  - Scaling applications up or down based on load in accordance with the parameters in the configuration

KUBERNETES - MANIFEST

```
apiVersion: v1
kind: Service
metadata:
  name: event-receiver-service
spec:
  selector:
    app: event-receiver
  ports:
  - protocol: "TCP"
    port: 9000
    targetPort: 8080
  type: LoadBalancer
```



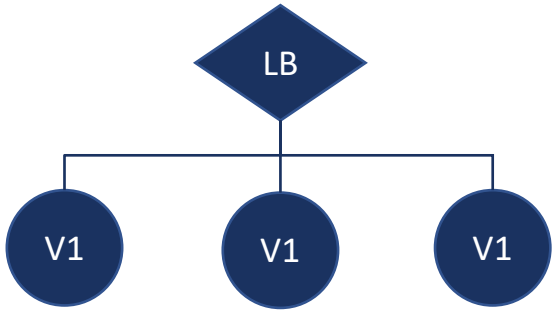

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: event-receiver
spec:
  selector:
    matchLabels:
      app: event-receiver
  replicas: 4
  template:
    metadata:
      labels:
        app: event-receiver
    spec:
      containers:
      - name: event-receiver
        image: <dockerhub username>/eventreceiver:latest
        ports:
        - containerPort: 8080
```

KUBERNETES DEPLOYMENT STRATEGIES

Recreate	A **Recreate** deployment terminate the old version and release the new one.
Rolling Update	**Rolling deployments** (or **ramped** deployment) are a pattern whereby, instead of **deploying** a package to all servers at once, we slowly **roll** out the release by **deploying** it to each server one-by-one. In load balanced scenarios, this allows us to reduce overall downtime.
Blue/Green	A **Blue/Green deployment** is a way of accomplishing a zero-downtime upgrade to an existing application. The "**Blue**" version is the currently running copy of the application and the "**Green**" version is the new version. Once the **green** version is ready, traffic is rerouted to the new version.
Canary	**Canary** Release is the technique that we use to "softly" **deploy** a new version of an application into Production. It consists of letting only a part of the audience get access to the new version of the app, while the rest still access the "old" version one.
A/B Testing	**A/B Testing** is the release a new version to a subset of users in a precise way (HTTP headers, cookie, weight, etc.). A/B testing is really a technique for making business decisions based on statistics. It does not come out of the box with Kubernetes. It is included here for completeness. It requires additional infrastructure such as Istio, Linkerd, Traefik, custom nginx/haproxy, etc)
Shadow	A new version is released alongside the old version. Incoming traffic is mirrored to the new version and doesn't impact the response.

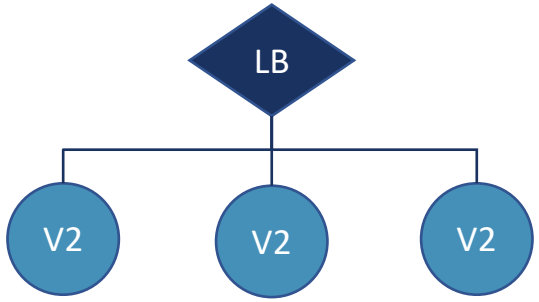
Recreate

Existing deployment V1 is terminated and deployment V2 is created.



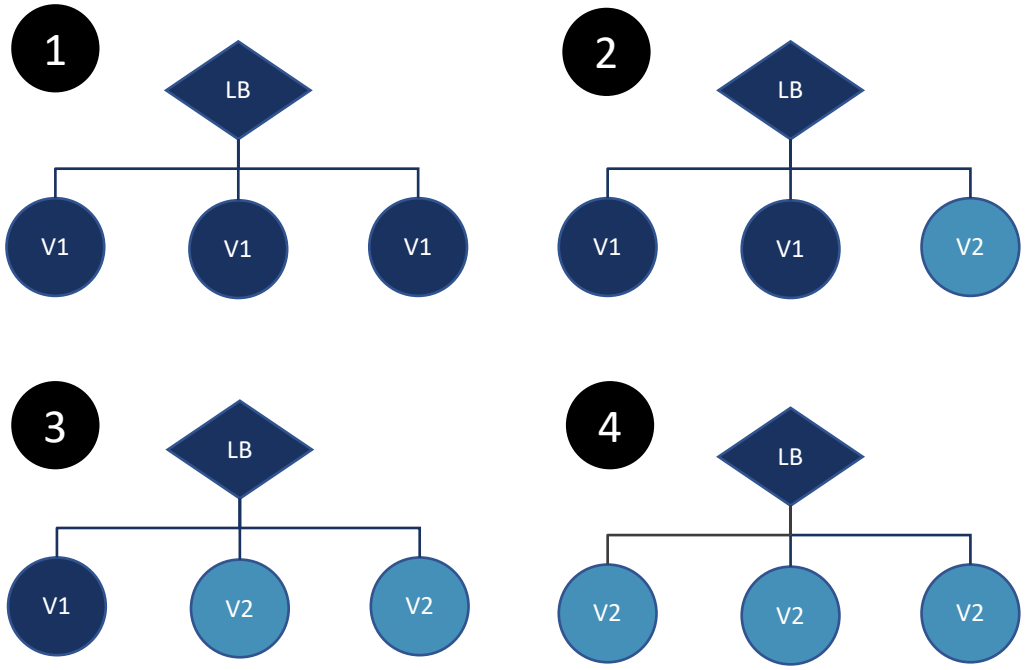
Terminate

Create



Rolling Update

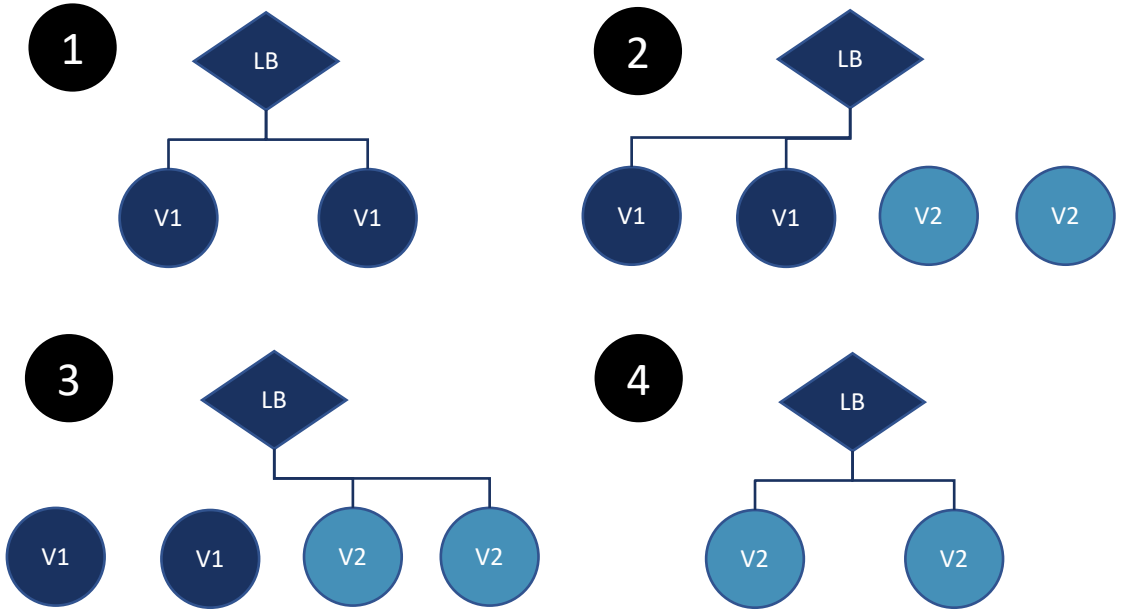
New deployment V2 gradually replaces existing deployment V1.



Blue/Green

Deployment V2 deployed in parallel with V1. Then traffic is switched over from V1 to V2.

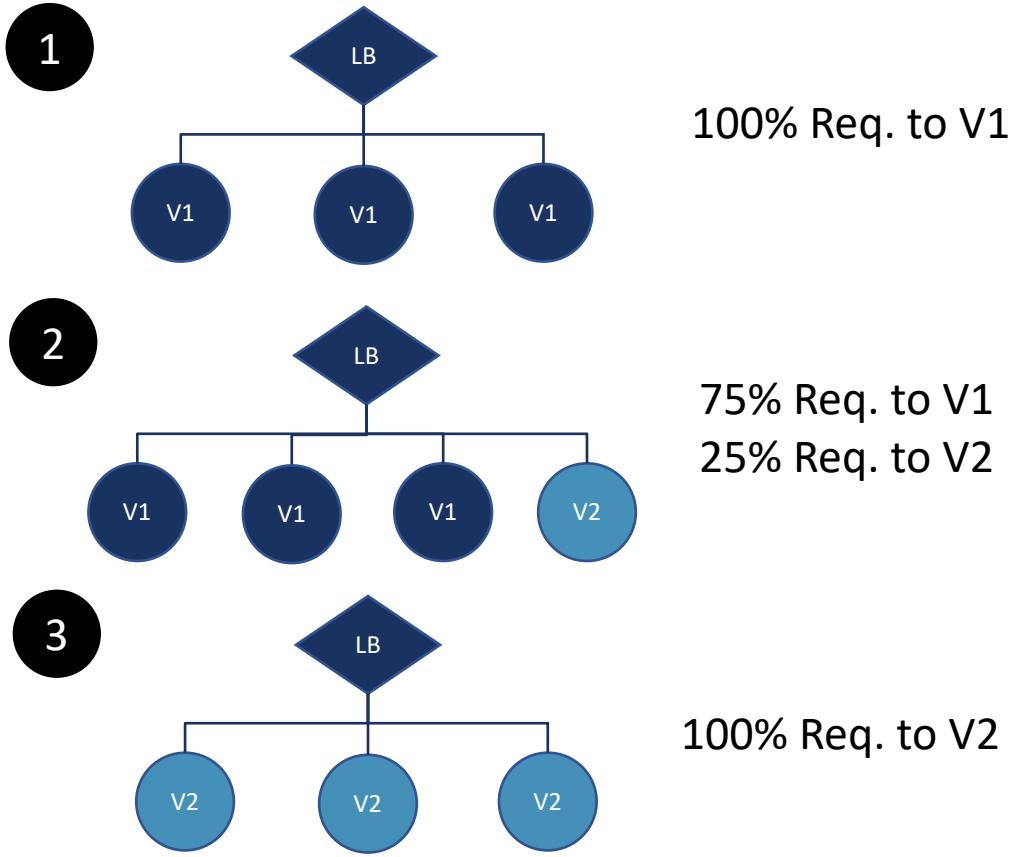
V1 is the Blue deployment, V2 is the Green deployment.



Canary

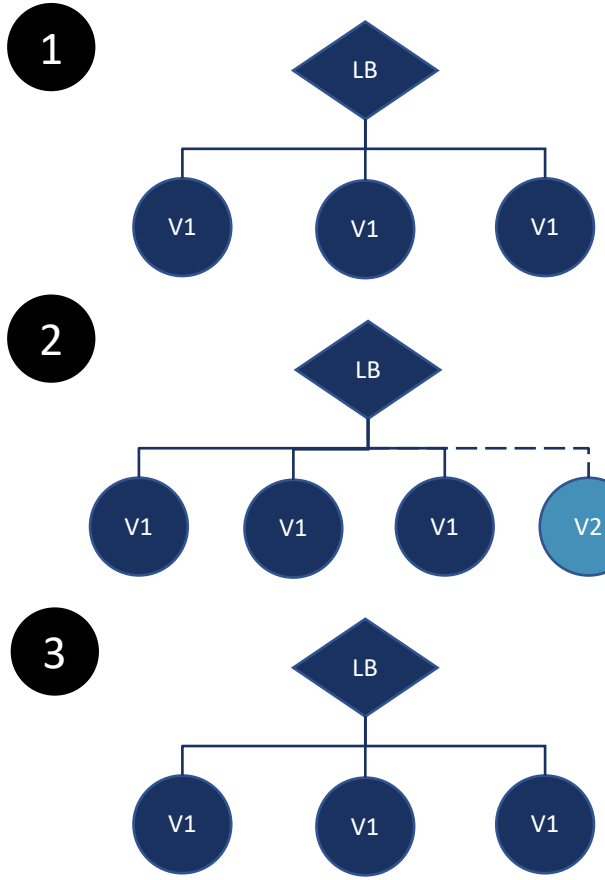
A V2 is deployed in parallel with V1 and a portion of traffic is redirected to V2.

When confident with V2 it can be scaled up, all traffic directed to V2 and V1 removed.



A/B Testing

A V2 is deployed as an experiment and filtered traffic is directed to V2 (based on a criteria). Data is collected and then V2 is terminated.

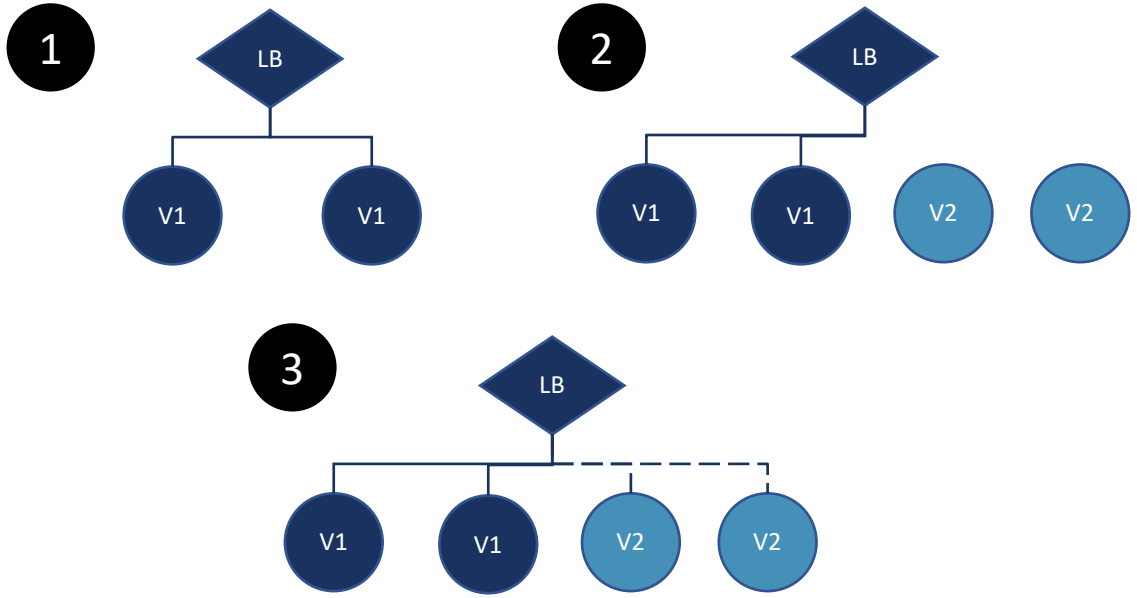


V2 is an experiment
Filtered Traffic is directed to V2

V2 is removed when experiment is complete

Shadow

A V2 is deployed in parallel with V1 and traffic is directed to both V1 and V2. However, only V1 provides responses to users. So V2 is running in the "shadows".



Requests are also directed to V2 to evaluate its stability and performance.

DEPLOYMENTS WITH DOCKER COMPOSE

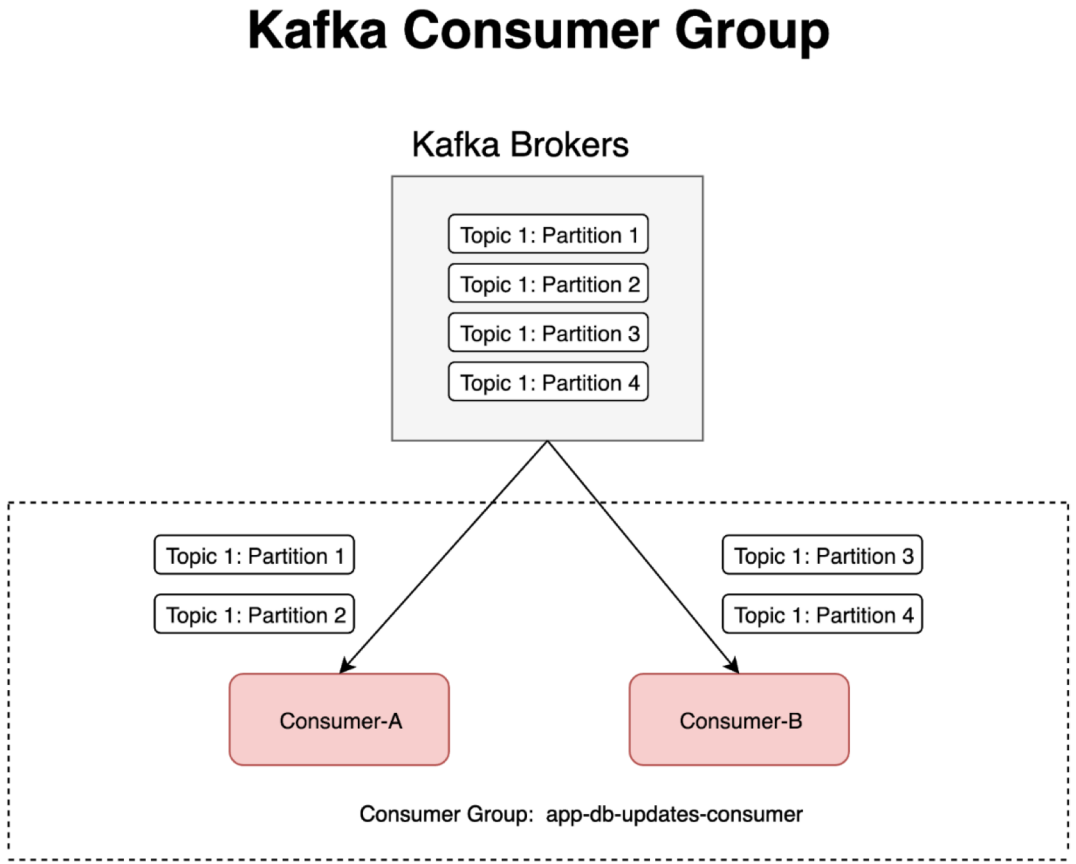
- When we make an update to one or more services used in a Docker Compose yaml file, we can take down all the services and then bring them back up with the new versions.
- If we just have updates to a couple of services (that are non-breaking to other services), we can just re-build the image and run "docker-compose up –d" again.
  - It will apply the **Recreate** deployment strategy
  - Bring down the existing container instances for the modified service
  - Then bring then back up again using the latest image

SCALING MESSAGE CONSUMERS

- Two Options
- Simple Consumer
  - All consumers receive all messages on a Topic and process them
  - This is good if each consumer performs a different task in response to the message
- Balanced Consumer
  - Each consumer receives messages from specific partition(s) on a Topic
  - A message is only received by one consumer in a Consumer Group
  - So messages aren't duplicated across consumers in the Consumer Group
  - Need at least as many partitions as consumers, otherwise additional consumers don't get any messages

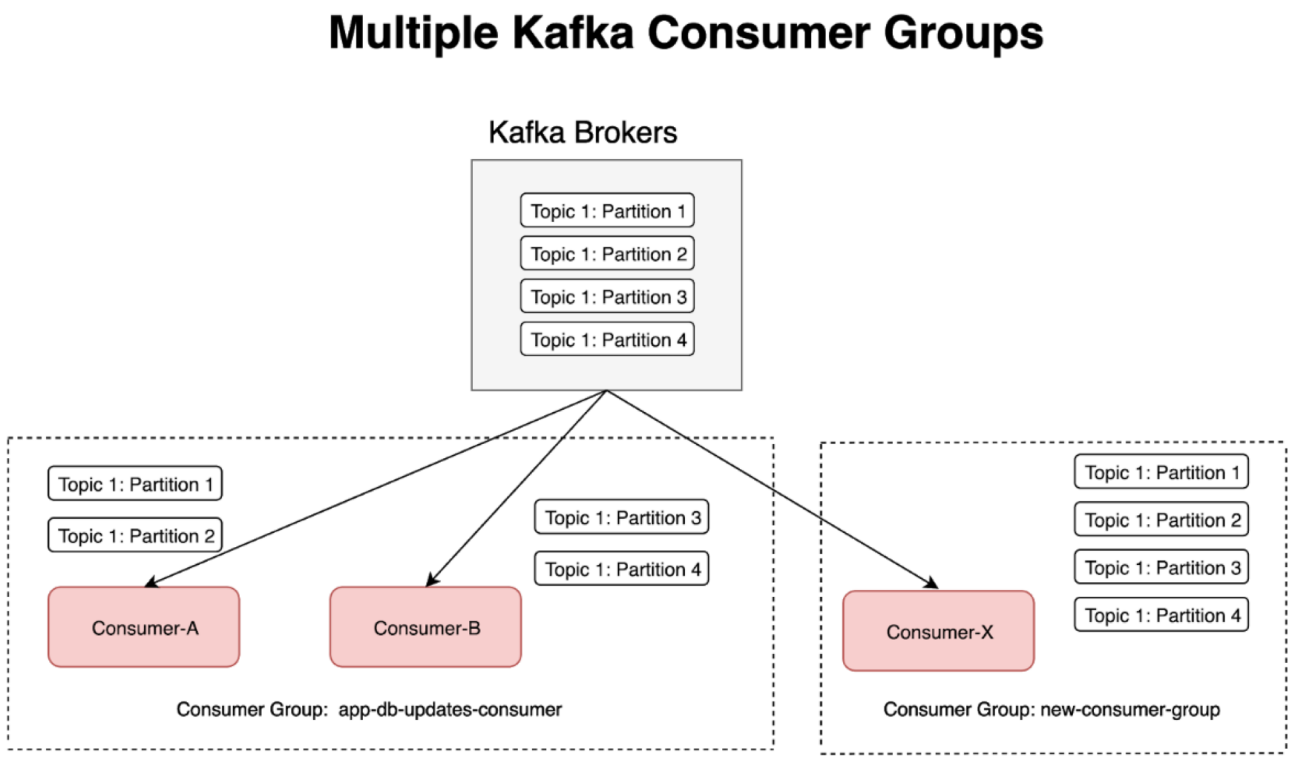
BALANCED CONSUMER PATTERNS

- With 4 partitions and 2 consumers, each consumer is allocated 2 partitions



BALANCED CONSUMER PATTERNS

- With 4 partitions and 2 consumers, each consumer in the **first consumer group** is allocated 2 partitions
- With 4 partitions and 1 consumer, the single consumer in the **second consumer group** is allocated all 4 partition
- The Consumer Group is a logical structure where the offsets across the partitions are stored
- A Balanced Consumer Group can also manage the partition allocations as the number of consumers in the group increases or decreases



SETTING UP A BALANCED CONSUMER

- In our Kafka service in docker-compose.yml

```
environment:
    KAFKA_CREATE_TOPICS: "events:2:1" # topic:partitions:replicas
```

The above specifies 2 partitions for Topic events

SETTING UP A BALANCED CONSUMER

- In our Storage service. Change the simple consumer to a balanced consumer:

```
consumer =
    topic.get_balanced_consumer(consumer_group='event_group',
                                zookeeper_connect=zookeeper,
                                reset_offset_on_start=False,
                                auto_commit_enable=True,
                                auto_commit_interval_ms=100)
```

This is a bit tricky to get setup and working. So it is an advanced improvement for Lab 11.

LAB 11 – REVERSE PROXY AND SCALING

Parts 1 to 3 – NGINX

- Setting up a reverse proxy
- Setting up a load balancer (for the Receiver Service)
- Scale up the Receiver Service and test it under high load

TODAY'S LAB

Today you will:

- Demo Lab 10
- Work on Lab 11 – Demo is due before the end of the next class