CLOUD COMPUTING APPLICATIONS

Docker Containers

Prof. Reza Farivar

Dockerfile

- ***Imperative*** method to create an image
- Example

```
FROM ubuntu:18.04
RUN \
  apt-get update && \
  apt-get install -y --no-install-recommends zip unzip openjdk-8-jdk expect && \
  apt-get autoremove -qq -y --purge && \
  apt-get clean && \
  rm -rf /var/cache/apt /var/lib/apt/lists

COPY myExecutableFile.sh /exec/myExecutableFile.sh

RUN chmod a+rwx -R /exec/

ENTRYPOINT ["./exec/myExecutableFile.sh"]
```

Running Docker Containers

- `docker container run alpine echo "Hello World"`

- First execution:
  - Will load the alpine image from Dockerhub

```
[→  ~ docker container run alpine echo "hello world"
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
ba3557a56b15: Pull complete
Digest: sha256:a75afd8b57e7f34e4dad8d65e2c7ba2e1975c795ce1ee22fa34f8cf46f96a3be
Status: Downloaded newer image for alpine:latest
hello world
```

- 2nd and afterwards:
  - Just runs

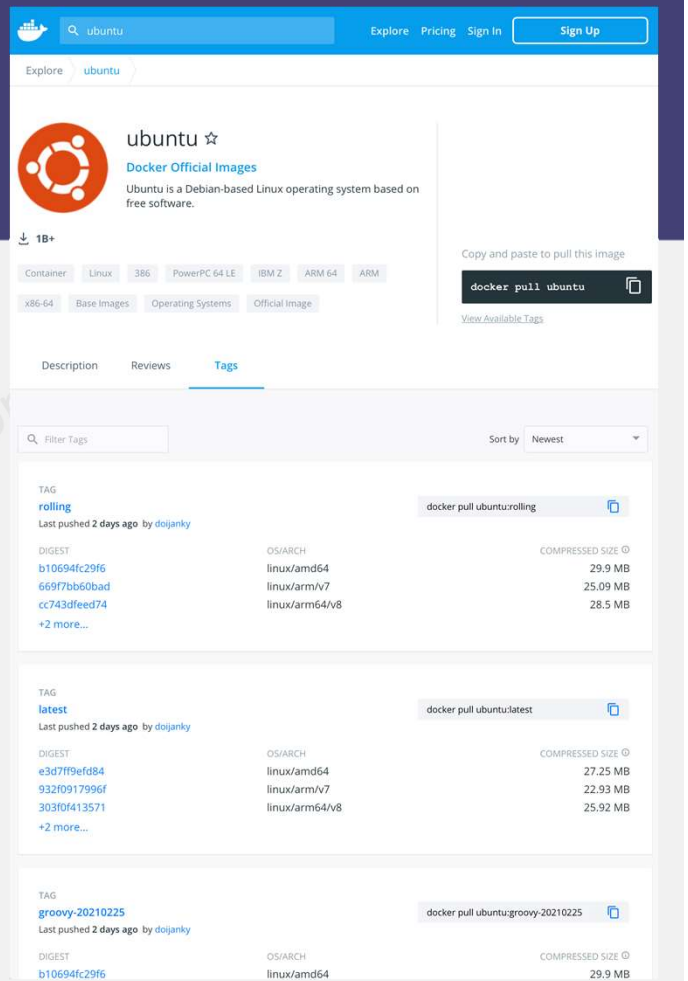
Daemon mode, Exec

- `docker container run alpine ping 127.0.0.1`
  - `64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.697 ms`
  - `64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.091 ms`
  - `64 bytes from 127.0.0.1: seq=2 ttl=64 time=0.090 ms`
  - `^C`
- `docker container run -d alpine ping 127.0.0.1`
  - `F64d6c10ec3f42646c254e8a935a472d7bc771dfcb2c6311c39616ac57e5f002`
- `docker container ls`
  - `CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES`
  - `f64d6c10ec3f  alpine  "ping 127.0.0.1"  19 seconds ago  Up 18 seconds  zen_noyce`
- `➜  ~ docker container exec -it f64d6c10ec3f /bin/sh`
- `/ # ps`
  - `PID  USER  TIME  COMMAND`
  - `1 root 0:00 ping 127.0.0.1`
  - `8 root 0:00 /bin/sh`
  - `14 root 0:00 ps`

Ubuntu in Docker

- `docker container run -it --name cca ubuntu:groovy-20210225 /bin/sh`

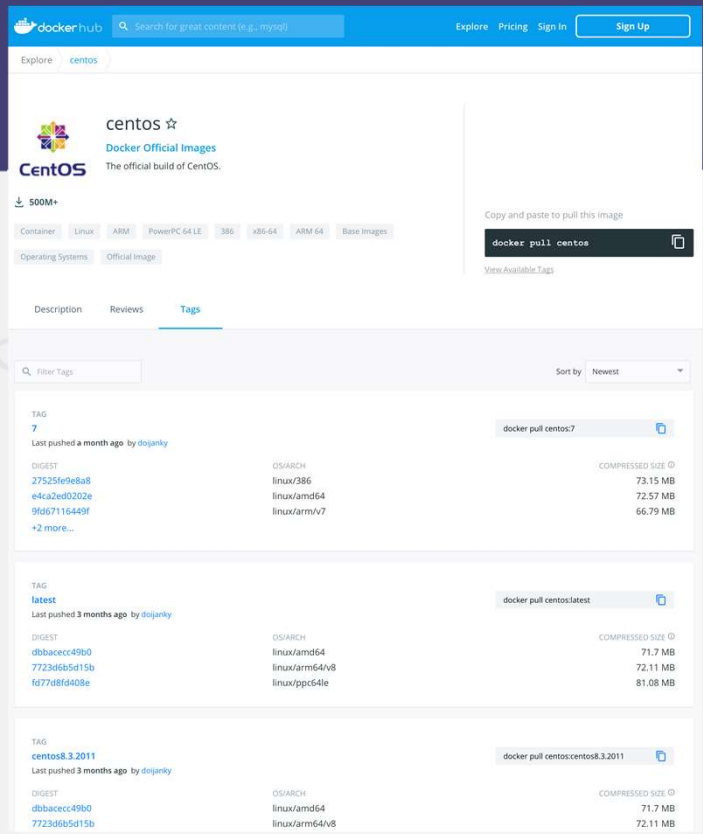
- `root@11d34305e17c:/# uname -a`
  - `Linux 11d34305e17c 4.19.121-linuxkit #1 SMP Tue Dec 1 17:50:32 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux`
- `root@11d34305e17c:/# cat /etc/os-release`
  - `NAME="Ubuntu"`
  - `VERSION="20.10 (Groovy Gorilla)"`
  - `ID=ubuntu`
  - `ID_LIKE=debian`
  - `PRETTY_NAME="Ubuntu 20.10"`
  - `VERSION_ID="20.10"`
  - `HOME_URL="https://www.ubuntu.com/"`
  - `SUPPORT_URL="https://help.ubuntu.com/"`
  - `BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"`
  - `PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"`
  - `VERSION_CODENAME=groovy`
  - `UBUNTU_CODENAME=groovy`



CENTOS in Docker

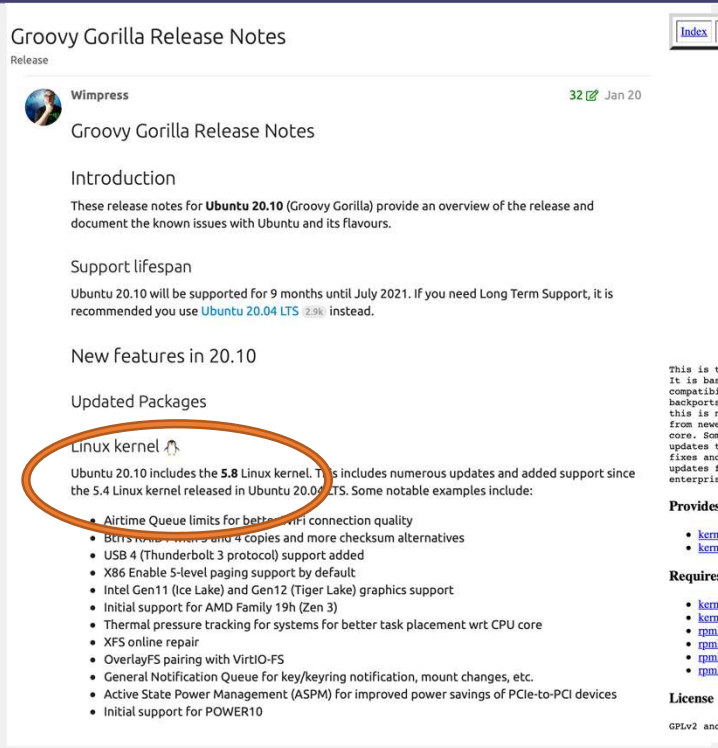
- `docker container run -it --name cca2 centos:latest /bin/sh`

- `[root@4a637cf86a9a /]# uname -a`
  - `Linux 4a637cf86a9a 4.19.121-linuxkit #1 SMP Thu Jan 21 15:36:34 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux`
- `[root@4a637cf86a9a /]# cat /etc/os-release`
  - `NAME="CentOS Linux"`
  - `VERSION="8"`
  - `ID="centos"`
  - `ID_LIKE="rhel fedora"`
  - `VERSION_ID="8"`
  - `PLATFORM_ID="platform:el8"`
  - `PRETTY_NAME="CentOS Linux 8"`
  - `ANSI_COLOR="0;31"`
  - `CPE_NAME="cpe:/o:centos:centos:8"`
  - `HOME_URL="https://centos.org/"`
  - `BUG_REPORT_URL="https://bugs.centos.org/"`
  - `CENTOS_MANTISBT_PROJECT="CentOS-8"`
  - `CENTOS_MANTISBT_PROJECT_VERSION="8"`
- `[root@4a637cf86a9a /]# cat /etc/centos-release`
  - `CentOS Linux release 8.3.2011`



Kernel Versions

- Note: The Groovy Gorilla version of ubuntu is supposed to come with Linux Kernel 5.8
- Centos 8.3 is supposed to run Kernel 4.18.240
- But we see it is running with 4.19.121 in Docker Desktop 3.2.3
  - Engine 20.10.5
  - This is the host Linux kernel, shared with all the containers
  - bzImage and initramfs



Groovy Gorilla Release Notes

Release

Wimpress 32 Jan 20

Groovy Gorilla Release Notes

Introduction

These release notes for **Ubuntu 20.10** (Groovy Gorilla) provide an overview of the release and document the known issues with Ubuntu and its flavours.

Support lifespan

Ubuntu 20.10 will be supported for 9 months until July 2021. If you need Long Term Support, it is recommended you use Ubuntu 20.04 LTS instead.

New features in 20.10

Updated Packages

Linux kernel

Ubuntu 20.10 includes the **5.8** Linux kernel. This includes numerous updates and added support since the 5.4 Linux kernel released in Ubuntu 20.04 LTS. Some notable examples include:

- Airtime Queue limits for better WiFi connection quality
- Btrfs RAID 1 with 3 and 4 copies and more checksum alternatives
- USB 4 (Thunderbolt 3 protocol) support added
- X86 Enable 5-level paging support by default
- Intel Gen11 (Ice Lake) and Gen12 (Tiger Lake) graphics support
- Initial support for AMD Family 19h (Zen 3)
- Thermal pressure tracking for systems for better task placement wrt CPU core
- XFS online repair
- OverlayFS pairing with VirtIO-FS
- General Notification Queue for key/keyring notification, mount changes, etc.
- Active State Power Management (ASPM) for improved power savings of PCIe-to-PCI devices
- Initial support for POWER10

Index | index by Group | index by Distribution | index by Vendor | index by creation date | index by Name | Mirrors | Help | Search

kernel-4.18.0-240.el8 RPM for x86_64

From CentOS 8.3.2011 BaseOS for x86_64 / Packages

Name: kernel	Distribution: Unknown
Version: 4.18.0	Vendor: CentOS
Release: 240.el8	Build date: Fri Sep 25 22:04:44 2020
Group: System Environment/Kernel	Build host: kbuilder.bsys.centos.org
Size: 0	Source RPM: kernel-4.18.0-240.el8.src.rpm
Packager: CentOS BuildSystem <http://bugs.centos.org>	
Url: http://www.kernel.org/	
Summary: The Linux kernel, based on version 4.18.0, heavily modified with backports	

This is the package which provides the Linux kernel for CentOS.
It is based on upstream Linux at version 4.18.0 and maintains kABI compatibility of a set of approved symbols, however it is heavily modified with backports and fixes pulled from newer upstream Linux kernel releases. This means this is not a 4.18.0 kernel anymore: it includes several components which come from newer upstream linux versions, while maintaining a well tested and stable core. Some of the components/backports that may be pulled in are: changes like updates to the core kernel (eg.: scheduler, cgroups, memory management, security fixes and features), updates to block layer, supported filesystems, major driver updates for supported hardware in CentOS, enhancements for enterprise customers, etc.

Provides

- kernel
- kernel(x86-64)

Requires

- kernel-core-uname-r = 4.18.0-240.el8.x86_64
- kernel-modules-uname-r = 4.18.0-240.el8.x86_64
- rpmlib(CompressedFileNames) <= 3.0.4-1
- rpmlib(FileDigests) <= 4.6.0-1
- rpmlib(PayloadFilesHavePrefix) <= 4.0-1
- rpmlib(PayloadIsXz) <= 5.2-1

License

GPLv2 and Redistributable, no modification permitted

Multi Stage Dockerfile

- Multi stage allows us to not include unnecessary intermediate layers
  - ```
    FROM alpine:latest AS build
    RUN apk update && apk add --update alpine-sdk
    RUN mkdir /app
    WORKDIR /app
    COPY . /app
    RUN mkdir bin
    RUN gcc test.c -o bin/test

    FROM alpine:latest
    COPY --from=build /app/bin/test /app/test
    CMD /app/test
    ```
- The resulting image will be 4MB instead of ~200MB
    ```

Image registries

- Docker Hub
- Google
  - https://cloud.google.com/container-registry
- Amazon AWS Amazon Elastic Container Registry (ECR)
  - https://aws.amazon.com/ecr/
- Microsoft Azure
  - https://azure.microsoft.com/en-us/services/container-registry/
- Red Hat
  - https://access.redhat.com/containers/
- Artifactory
  - https://jfrog.com/integration/artifactory-docker-registry/

Full Namespace of Images

- <registry URL>/<User or Org>/<name>:<tag>
  - public.ecr.aws/nginx/nginx:latest
  - public.ecr.aws/datadog/agent:latest
  - myregistry.azurecr.io/marketing/campaign10-18/email-sender:v2

CLOUD COMPUTING APPLICATIONS

Docker Swarm

Prof. Reza Farivar

Container Orchestration

- Many application consist of multiple components, that need to be distributed on more than one machine
- Using containers, we can have each component running in its own container
- Thousands of pre-built components available on public registries

Docker vs. Swarm

```
$ docker container create \
 --name my-nginx \
 --publish 80:80 \

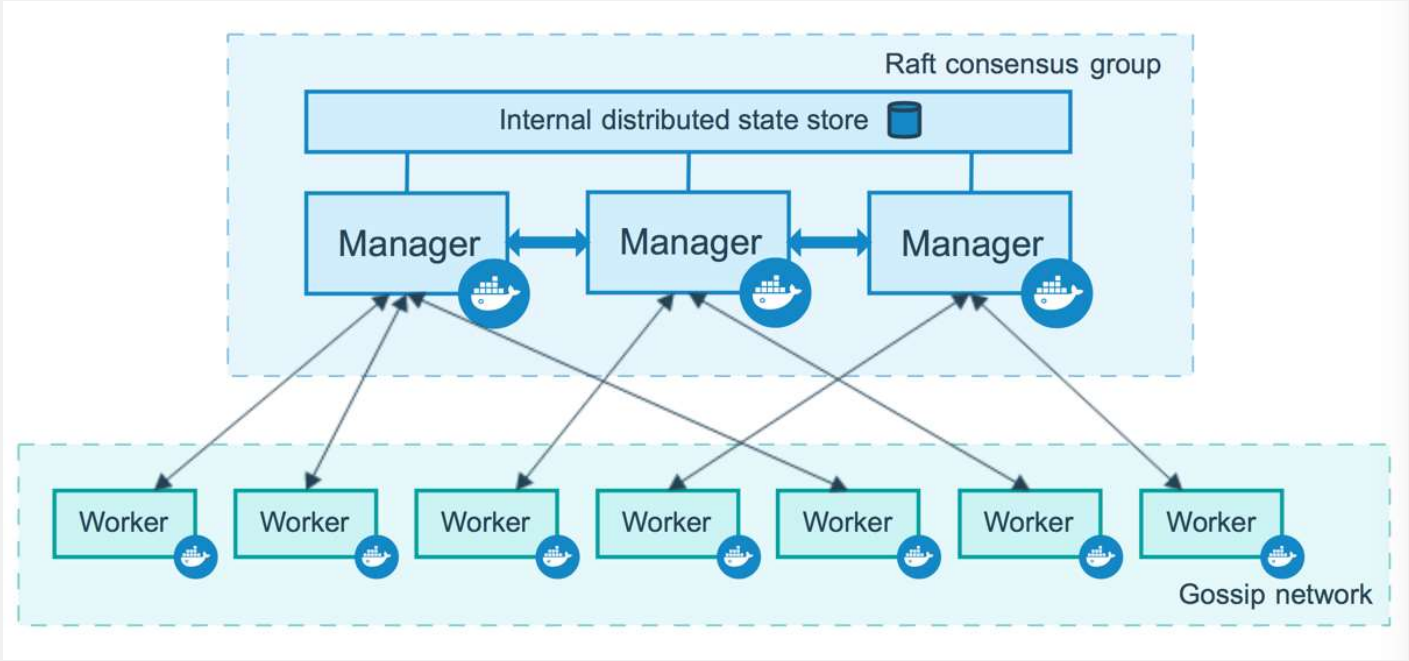
 --network nginx-net \
 nginx
```

```
$ docker service create \
 --name my-nginx \
 --publish target=80,published=80 \
 --replicas=5 \
 --network nginx-net \
 nginx
```

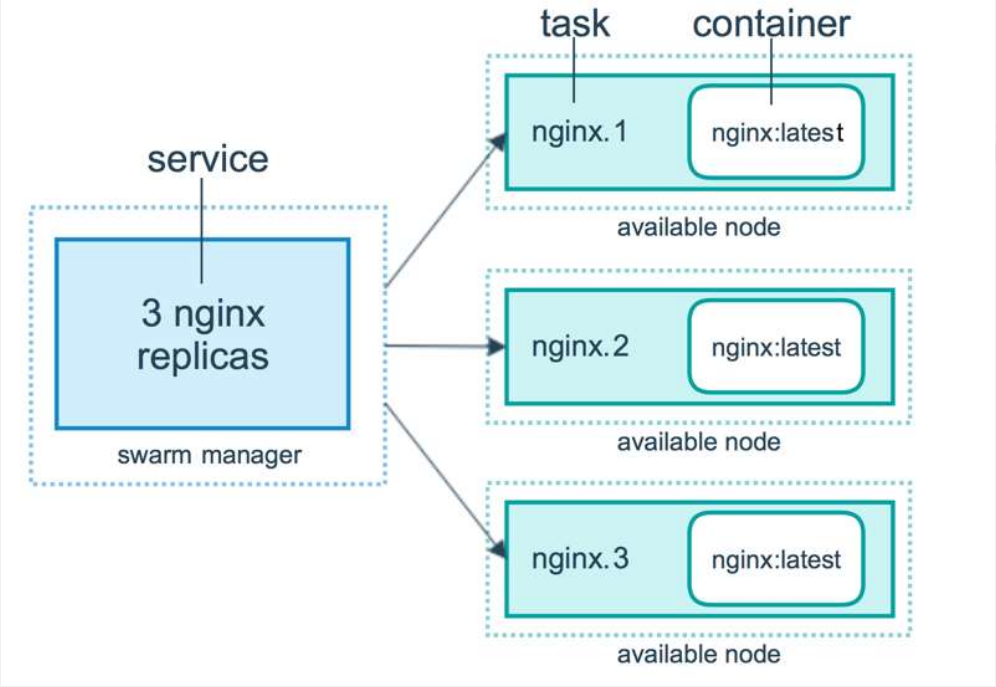
Swarm Services

- Swarm services use a *declarative* model, which means that you define the desired state of the service, and rely upon Docker to maintain this state.
- State
  - image name and tag
  - how many containers (tasks) in the service
  - ports exposed to clients outside the swarm

Nodes on Docker Swarm



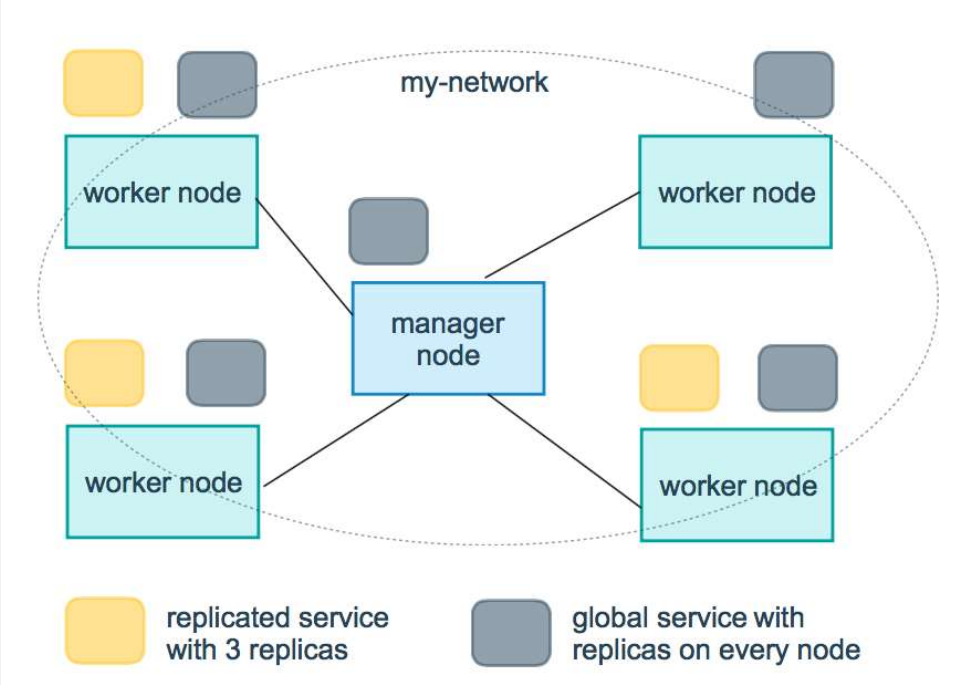
Services on Docker Swarm



Swarm Task States

- Docker lets you create services, which can start tasks.
- A service is a description of a desired state, and a task does the work.
- Work is scheduled on swarm nodes in this sequence
  1. Create a service by using `docker service create`.
  2. The request goes to a Docker manager node.
  3. The Docker manager node schedules the service to run on particular nodes.
  4. Each service can start multiple tasks.
  5. Each task has a life cycle, with states like `NEW`, `PENDING`, and `COMPLETE`

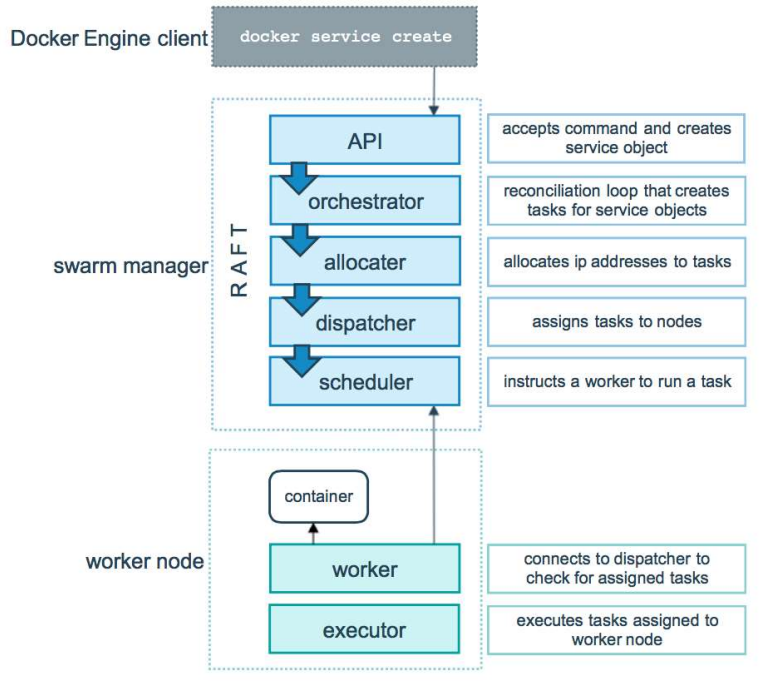
Replicated and Global Tasks



Example: Run a three-task Nginx service on 10-node swarm

- `$ docker service create --name my_web --replicas 3 --publish published=8080,target=80 nginx`
- Three tasks run on up to three nodes.
- You don't need to know which nodes are running the tasks; connecting to port 8080 on **any** of the 10 nodes connects you to one of the three nginx tasks.
  - Routing mesh

Tasks and Scheduling



CLOUD COMPUTING APPLICATIONS

Docker Networking

Prof. Reza Farivar

Docker Networking

- Pluggable, using drivers
  - Bridge
  - Host
  - Overlay
  - macvlan
  - None

Docker Bridge Networking

- `$ docker network create my-net`
-
```
$ docker create --name my-nginx \
--network my-net \
--publish 8080:80 \
nginx:latest
```
- connects Nginx container to the `my-net` network
- Any other container connected to the `my-net` network has access to all ports on the `my-nginx` container, and vice versa.
- It ***also*** publishes port 80 in the container to port 8080 on the Docker host, so external clients can access that port
- User-defined bridge networks provide DNS resolution

Docker Overlay Network Driver

- creates a distributed network among multiple Docker daemon hosts
- sits on top of (overlays) the host-specific networks
- Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container
- In a swarm, two overlay networks play a role
  - `ingress`, which handles control and data traffic related to swarm services.
  - User-defined overlay networks
    - `$ docker network create -d overlay my-overlay`

User-defined Overlay Networks

- Swarm services connected to the same overlay network effectively expose all ports to each other
- Encryption
  - All swarm service management traffic is encrypted by default
    - using the AES Algorithm in GCM mode
  - To encrypt application data: `--opt encrypted`
- Subnet CIDR
  - ```
    $ docker network create \
     --driver overlay \
     --subnet=10.11.0.0/16 \
     --gateway=10.11.0.2 \
     --opt encrypted \
     my-overlay
    ```
    ```

CLOUD COMPUTING APPLICATIONS

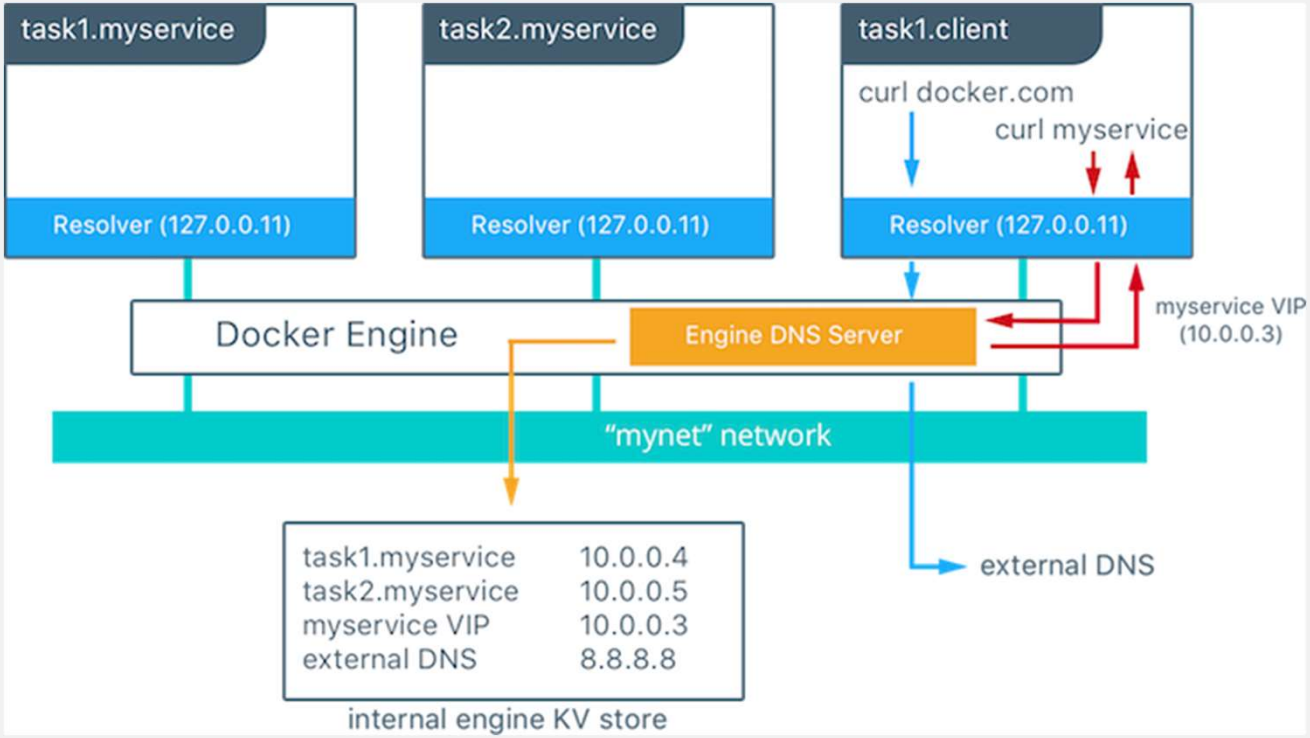
Service Discovery and Internal Load Balancing

Prof. Reza Farivar

Service Discovery

- User-defined networks provide DNS service
  - User-defined Bridge networks
  - User-defined overlay networks
- For most situations, you should connect to the service name, which is load-balanced and handled by all containers (“tasks”) backing the service.
- To get a list of all tasks backing the service, do a DNS lookup for `tasks.<service-name>`

DNS


task1.myservice
task2.myservice
task1.client
curl docker.com
curl myservice
Resolver (127.0.0.11)
Resolver (127.0.0.11)
Resolver (127.0.0.11)
Docker Engine
Engine DNS Server
myservice VIP
(10.0.0.3)
"mynet" network
task1.myservice 10.0.0.4
task2.myservice 10.0.0.5
myservice VIP 10.0.0.3
external DNS 8.8.8.8
internal engine KV store
external DNS

Internal Load Balancing in Swarm Services

- From one service to another
- This feature is automatically enabled once a Service is created.
- When a Service is created, it get a virtual IP address right away, on the Service's network.
- When a Service is requested the resulting DNS query is forwarded to the Docker Engine, which in turn returns the IP of the service, ***a virtual IP***.
- Traffic sent to that virtual IP is load balanced to all of the healthy containers of that service on the network.
- All the load balancing is done by Docker, since only one entry-point is given to the client (one IP).

CLOUD COMPUTING APPLICATIONS

Docker Ingress Network and Routing Mesh

Prof. Reza Farivar

Swarm Overlay Networks

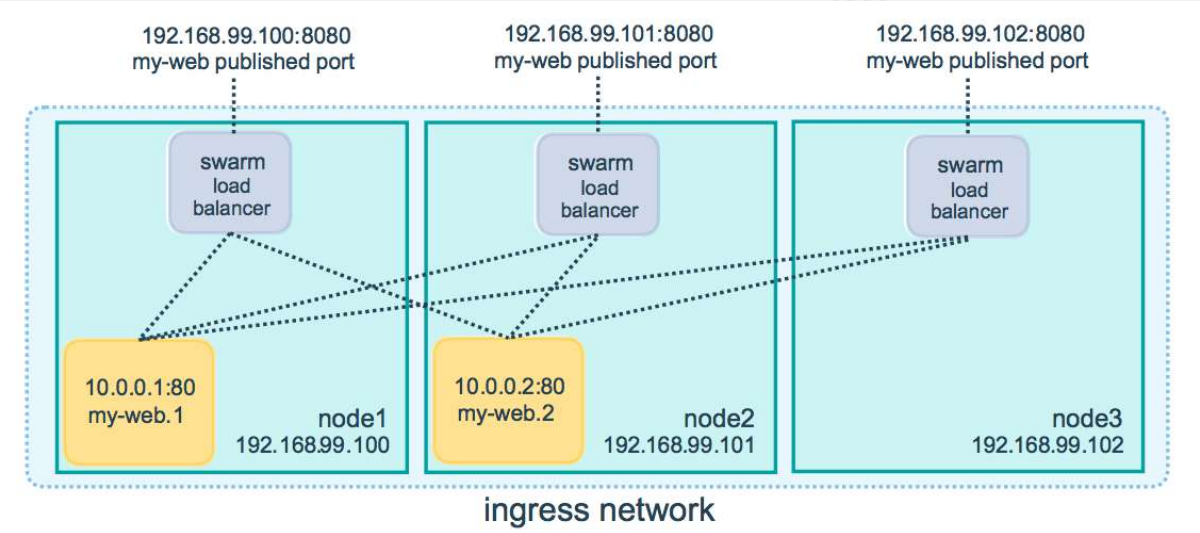
- Recall bridge networks on one host:
  - Containers connected to the same user-defined bridge network effectively expose all ports to each other.
  - For a port to be accessible to containers or non-Docker hosts on different networks, that port must be *published* using the -p or --publish flag.
- On a multi-host Docker Swarm
  - Swarm services connected to the same overlay network effectively expose all ports to each other
  - For a port to be accessible outside of the service, that port must be *published* using the -p or --publish
- By default, swarm services which publish ports do so using the ***routing mesh***

Docker Swarm Routing Mesh

- By default, swarm services which publish ports do so using the routing mesh
- When you connect to a published port on any swarm node **(whether it is running a given service or not)**, you are redirected to a worker which is running that service, transparently
- Effectively, Docker acts as a load balancer for your swarm services.
- Services using the routing mesh are running in *virtual IP (VIP) mode*.

Published ports

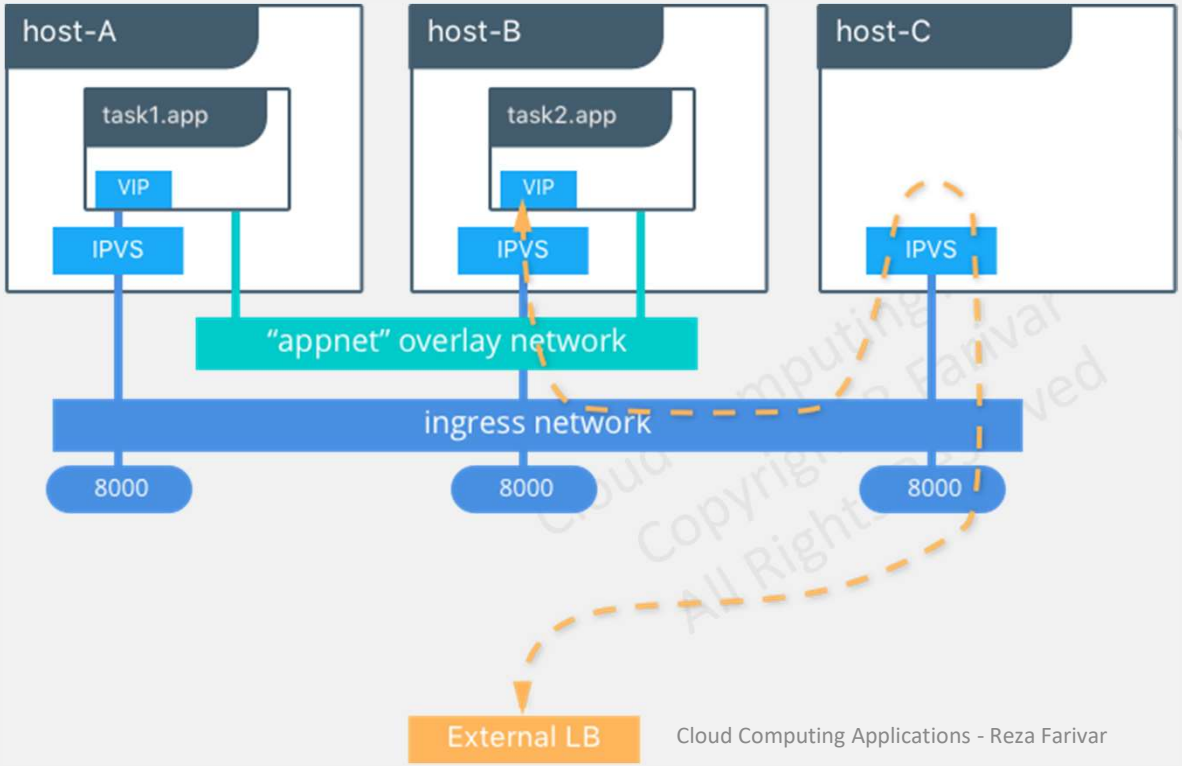
- When you create a swarm service, you can publish that service's ports to hosts outside the swarm
  - Routing mesh



ingress Overlay Network

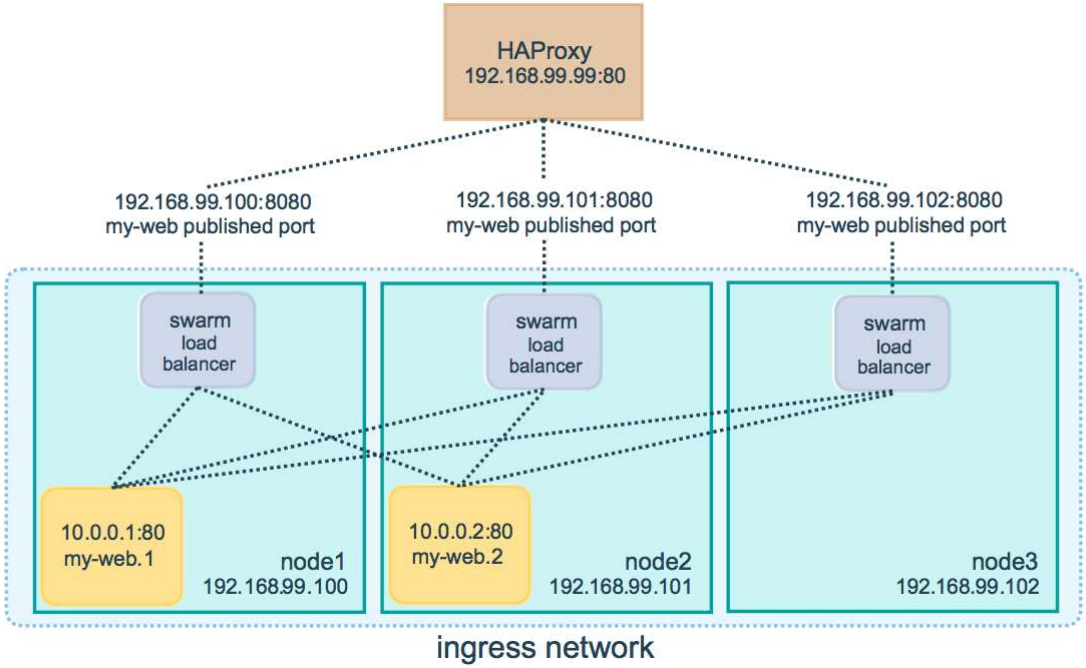
- The **ingress** network plays an important role in enabling published ports over the swarm
  - When you create a swarm service and do not connect it to a user-defined overlay network, it connects to the ingress network by default
- For a port to be accessible ***outside*** of the swarm service, it must be *published* using the `-p` or `--publish` flag on `docker service create` or `docker service update`
- Map TCP port 80 on the service to TCP port 8080 on the routing mesh, and map UDP port 80 on the service to UDP port 8080 on the routing mesh.
  - `-p 8080:80/tcp -p 8080:80/udp`
  - `-p published=8080,target=80,protocol=tcp -p published=8080,target=80,protocol=udp`

External Load Balancing in Swarm Services



External Load Balancer

- Using the routing mesh



Example HAProxy Config

```
global
        log /dev/log    local0
        log /dev/log    local1 notice
...snip...

# Configure HAProxy to listen on port 80
frontend http_front
   bind *:80
   stats uri /haproxy?stats
   default_backend http_back

# Configure HAProxy to route requests to swarm nodes on port 8080
backend http_back
   balance roundrobin
   server node1 192.168.99.100:8080 check
   server node2 192.168.99.101:8080 check
   server node3 192.168.99.102:8080 check
```

External Load Balancing w/o Routing Mesh

- set `--endpoint-mode` to `dnsrr`
  - the default value is `vip`
- No more a single virtual IP
- Docker sets up DNS entries for the service such that a DNS query for the service name returns ***a list of IP addresses***
  - Client connects directly to one of these
- You are responsible for providing the list of IP addresses and ports to your load balancer

Routing Mesh

- When you publish a service port, the swarm makes the service accessible at the target port on every node
  - regardless of whether there is a task for the service running on that node or not.
- This is less complex and is the right choice for many types of services

Host Mode

- You can publish a service task's port directly on the swarm node where that service is running.
  - Bypasses the routing mesh
  - Provides the maximum flexibility, including the ability to develop custom routing framework
- However, you are responsible for keeping track of where each task is running and routing requests to the tasks, and load-balancing across the nodes.
- use the `mode=host` option to the `--publish` flag

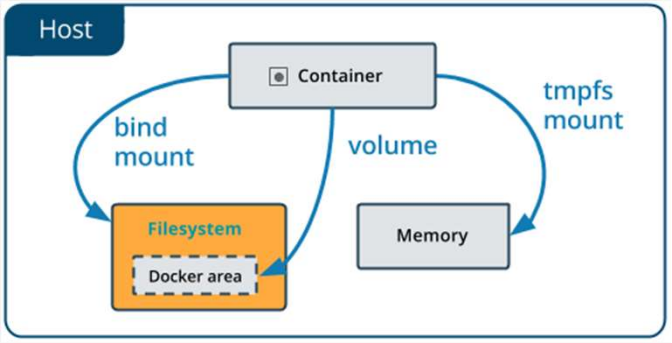
CLOUD COMPUTING APPLICATIONS

Docker Swarm

Prof. Reza Farivar

Data Volumes

- Recall that Docker containers are based on Unionfs
  - Multiple immutable (read-only) base layers
  - One read-write container-specific layer
- When a container is removed, the top layer is also removed
- To persist changes, and to access data outside the container, we need to mount an external storage location
- Three types of host to container mapping
  - Bind mount
  - Volume
  - tmpfs

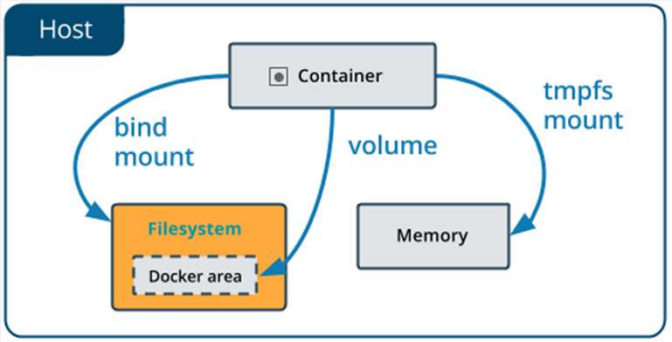


tmpfs

- tmpfs mounts are best used for cases when you do not want the data to persist either on the host machine or within the container
  - for security reasons
  - to protect the performance of the container when your application needs to write a large volume of non-persistent state data

Persistent Data Storage: Bind Mount

- When you use a bind mount, a file or directory on the *host machine* is mounted into a container
- The file or directory is referenced by its absolute path on the host machine



Docker Volume

- Persistent storage ***abstraction***
- Managed by Docker
- Will last after the container is removed
- Different drivers
  - Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
  - For local deployments, usually `local` driver
- For distributed applications (swarm)
  - Use an NFS and use `local` driver
  - Some drivers support writing files to an external storage system like NFS or Amazon S3
    - REX-Ray, CloudStor
    - `vieux/sshfs` volume drive

Mount Volume examples

- `$docker volume create my-vol`
- `$docker volume ls`
- `$docker volume inspect my-vol`
- `$docker volume rm my-vol`

- ```
  $ docker run -d \
      --name devtest \
      --mount source=my-vol,target=/app \
      nginx:latest
  ```
  ```

CLOUD COMPUTING APPLICATIONS

Docker Configs and Secrets

Prof. Reza Farivar

Docker Secrets

- A secret is a blob of data
  - E.g. password, SSH private key, SSL certificate
- Should not be transmitted over a network or stored unencrypted in a Dockerfile or in your application's source code
- use Docker *secrets* to centrally manage
- Docker secrets are only available to swarm services, not to standalone containers

Docker Secrets CLI

- `docker secret create`
- `docker secret inspect`
- `docker secret ls`
- `docker secret rm`
- `--secret` flag for `docker service create`
- `--secret-add` and `--secret-rm` flags for `docker service update`

Docker Secrets

- When you grant a newly-created or running service access to a secret, the ***decrypted*** secret is mounted into the container in an in-memory filesystem
- Location of the mount point within the container defaults to:
  - /run/secrets/<secret_name> in Linux containers
  - C:\ProgramData\Docker\secrets in Windows containers

Docker Configs

- Docker secrets are only available to swarm services, not to standalone containers
- Docker configs are only available to swarm services, not to standalone containers
- Configs operate in a similar way to secrets, except that they are not encrypted at rest and are mounted directly into the container's filesystem without the use of RAM disks.

Docker Config CLI

- `docker config create`
- `docker config inspect`
- `docker config ls`
- `docker config rm`

CLOUD COMPUTING APPLICATIONS

Docker Orchestration: Compose and Stack

Prof. Reza Farivar

Infrastructure as Code

- Imperative
  - Tell the system exactly how to do things
    - $ docker network create –d overlay my-network
    - $ docker container create --name web-service --publish 80:80 --network my-network nginx:latest
    - $ docker container create --name db-service --network my-network --volume /usr/bin:/usr/local/bin mysql:latest
    - …
- Declarative
  - Tell the system what you want to be achieved

Docker Compose

- The main tool by Docker for container orchestration
- Uses YAML file format by default
- The default file name is `docker-compose.yml`
- Declarative way
  - You tell docker-compose what your implementation looks like, it figures out the operations to make it happen

Imperative vs. Declarative

```
$ docker network create -d overlay my-network
$ docker container create --name web-service
--publish 80:80 --replicas 3 --network my-network
nginx:latest
$ docker container create --name db-service
--replicas 1 --network my-network
--volume /usr/bin:/usr/local/bin
mysql:latest
```

```
version: "3.9"
service:
          web-service:
                    image: nginx:latest
                    ports: "80:80"
                    deploy:
                              replicas: 3
                    networks:
                              front-end
...
networks:
          front-end:
```

Compose Specification

- Compose Application Model
- Formalizes many of the concepts we have covered
  - Service
  - Network
  - Volume
  - Config
  - Secret
- *https://github.com/compose-spec/compose-spec/blob/master/spec.md*

Compose

- The most basic deployment model is docker-compose, running in one machine, and deploying multiple containers
- Docker Swarm is the distributed multi-host extension
- Same compose specification

Compose versions

- Version 1 was only for single host deployment
  - No Volumes, no Networks, no Build arguments
- Version 2
  - Support for Volumes, Networks and Build added
  - `depends-on` to indicate startup order
- Version 3
  - Targets both single host (docker-compose) and Swarm mode stacks
  - Added secrets

- https://docs.docker.com/compose/compose-file/compose-versioning

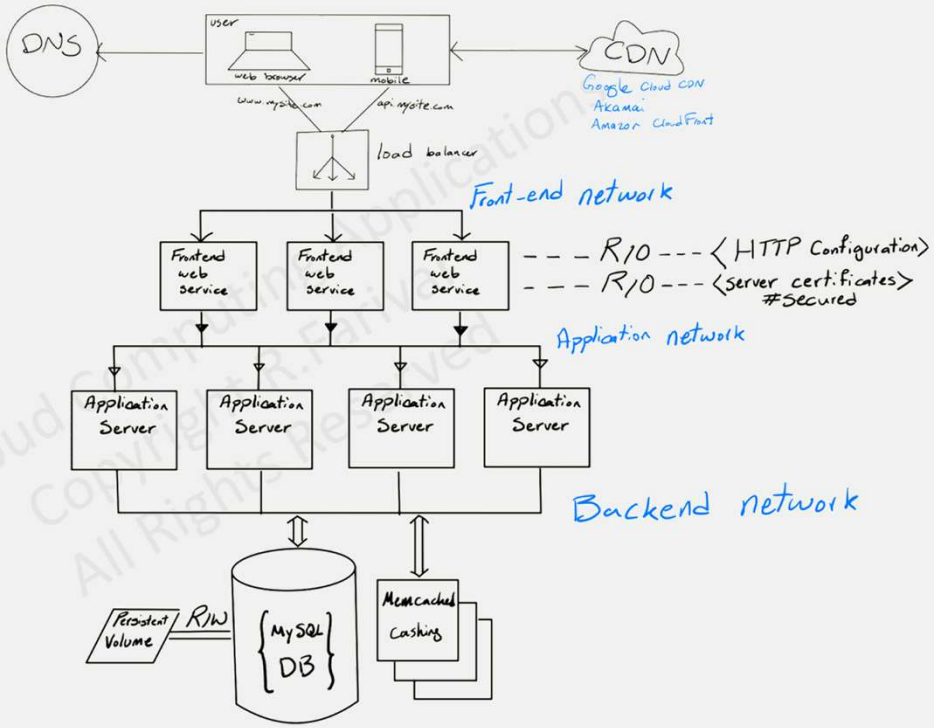
CLOUD COMPUTING APPLICATIONS

Example Architecture in Docker Swarm Stack

Prof. Reza Farivar

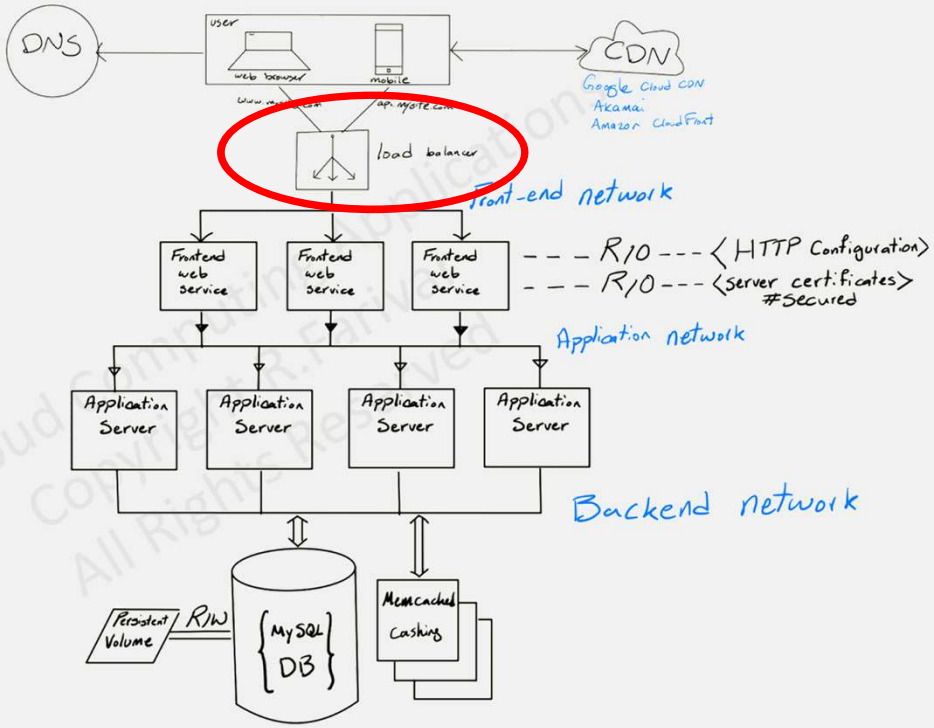
Example Architecture

- Classic 3-tier architecture



Example Architecture

- Classic 3-tier architecture



Example HAProxy Config

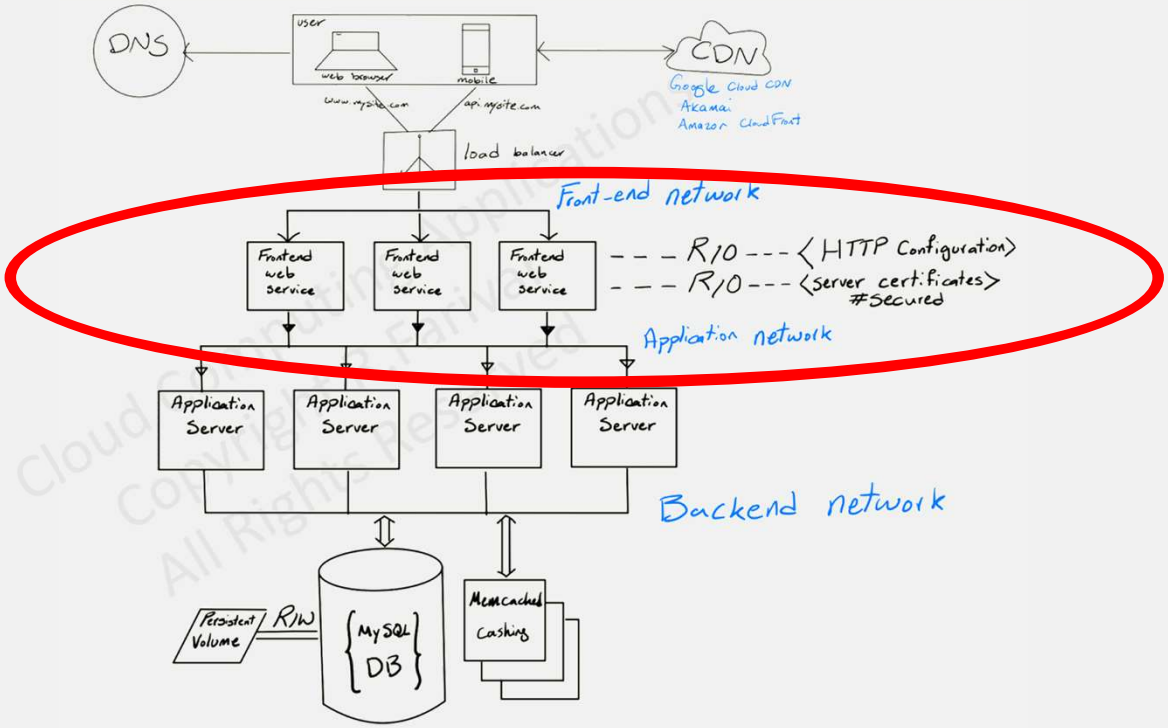
```
global
        log /dev/log    local0
        log /dev/log    local1 notice
...snip...

# Configure HAProxy to listen on port 80
frontend http_front
   bind *:80
   stats uri /haproxy?stats
   default_backend http_back

# Configure HAProxy to route requests to swarm nodes on port 8080
backend http_back
   balance roundrobin
   server node1 192.168.99.100:8080 check
   server node2 192.168.99.101:8080 check
   server node3 192.168.99.102:8080 check
```

Example Architecture

- Classic 3-tier architecture



Frontend web-app service

```
version: "3.9”
services:
    frontend:
        image: web-app
        deploy:
            replicas: 3
        ports:
            - "443:8043"
        networks:
            - front-tier
            - application-tier
        configs:
            - httpd-config
        secrets:
            - server-certificate
```

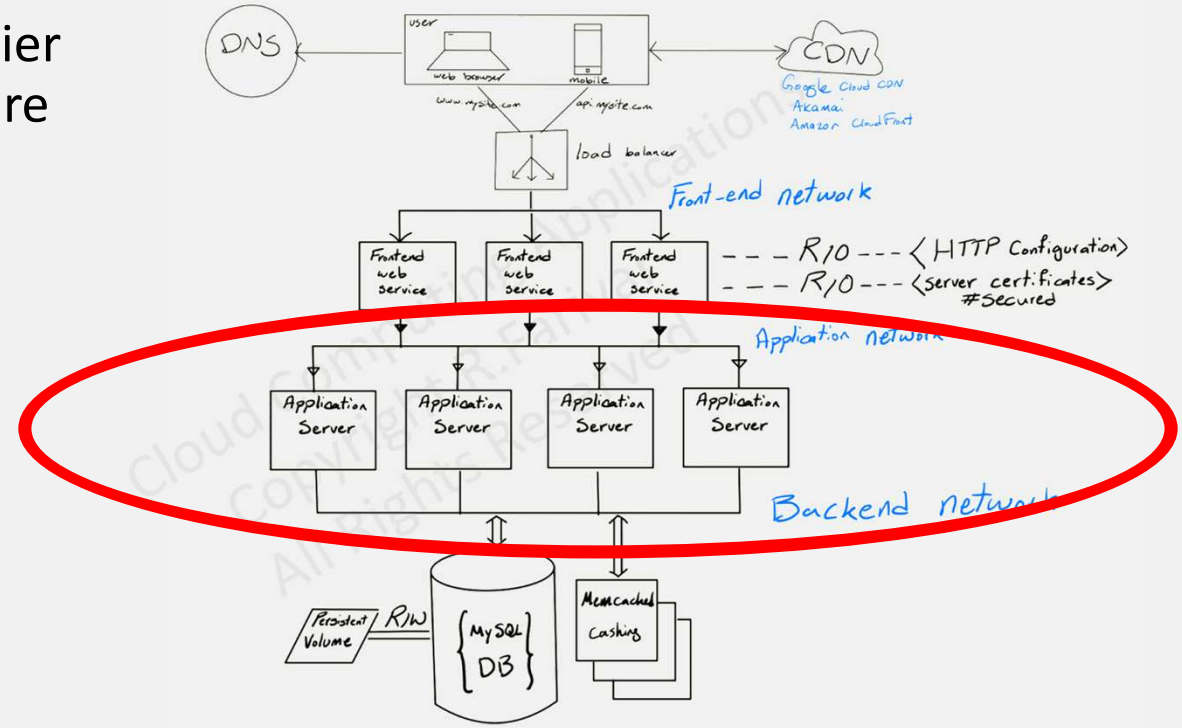
- *Deploy Specifies configuration related to the deployment and running of services*
- *Ignored by docker-compose*

- *Endpoint_mode: vip (default)*
- *Port Publishing*
- *Routing Mesh*
  - *Ingress Network*
- *Port 443 on **ALL** Swarm nodes is load-balanced into the 3 replicas*
  - *Swarm port*

https://github.com/compose-spec/compose-spec/blob/master/spec.md

Example Architecture

- Classic 3-tier architecture



Application Service

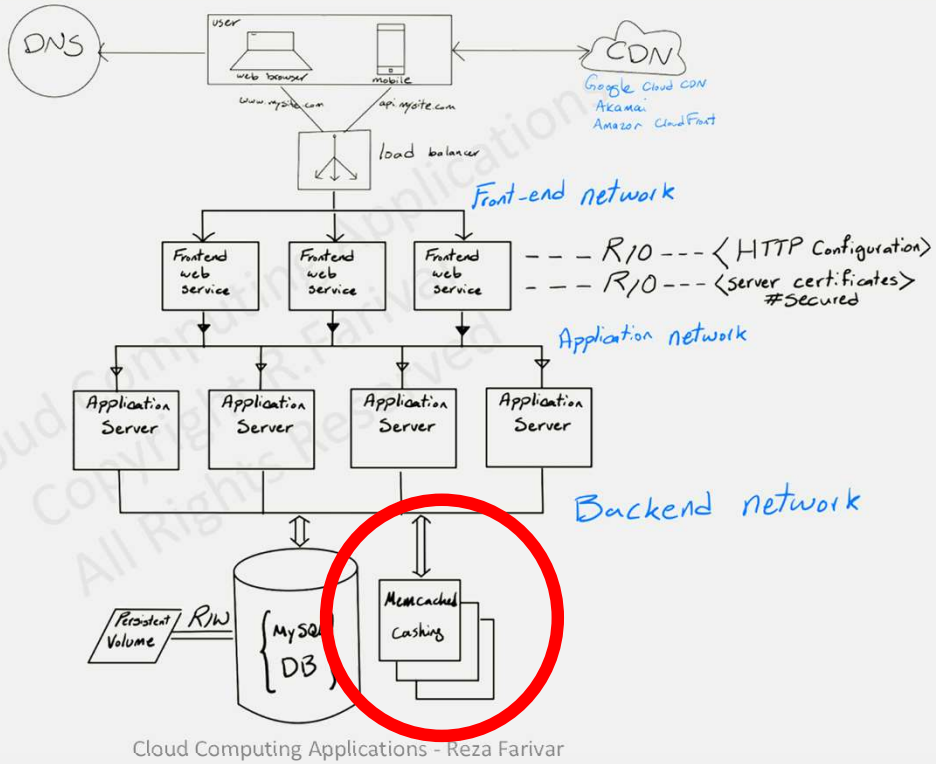
```
application:
        build:
                context: ./dir  #path to the build context
                dockerfile: Dockerfile-alternate
                args:
                        - buildno: 1
        image: application-logic
        deploy:
                mode: replicated
                replicas: 4
        networks:
                - application-tier
                - back-tier
```

Build is only used in docker-compose
Ignored in Swarm stacks

- *Default mode is replicated*
- *Global: exactly one container per swarm node*

Example Architecture

- Classic 3-tier architecture

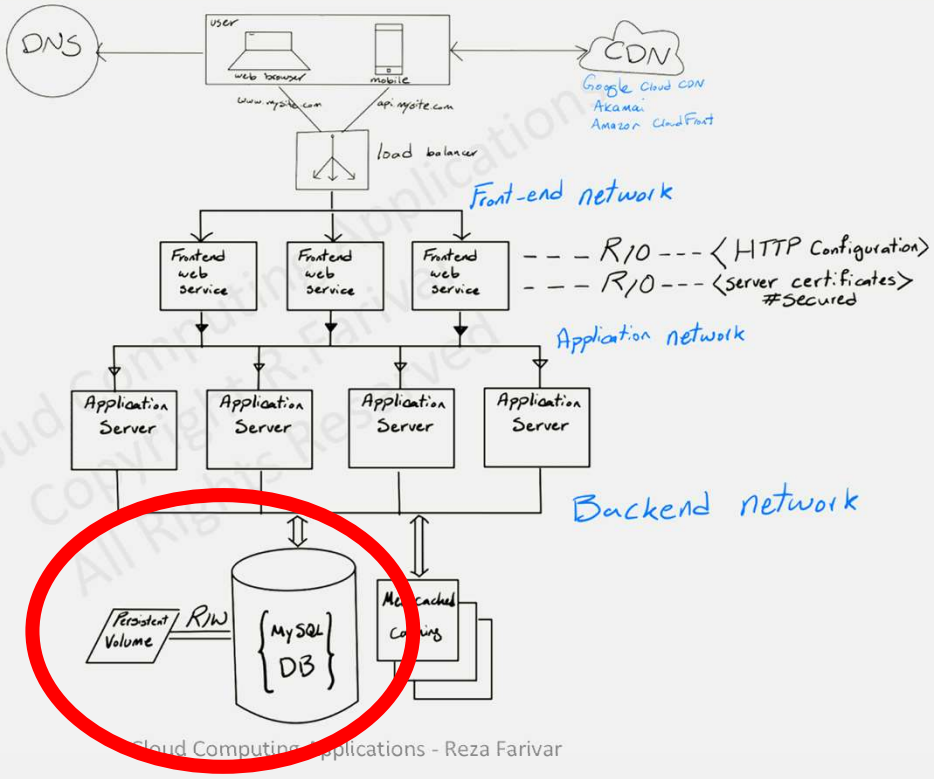


Cache Service

```
Cache:
        image: Memcached:1.6
        deploy:
                replicas: 3
                placement:
                        max_replicas_per_node: 1
                resources:
                        limits:
                                cpus: ‘1.50’
                                memory: 8G
                        reservations:
                                cpus: '0.25’
                                memory: 500M
        networks:
                - back-tier
```

Example Architecture

- Classic 3-tier architecture

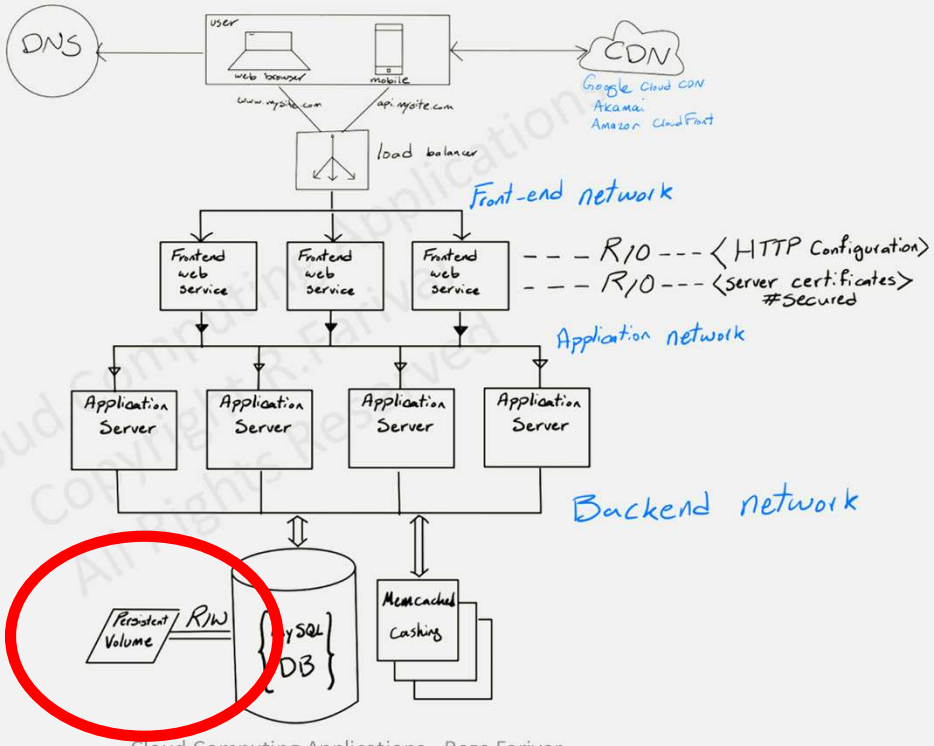


Backend Database Service

```
backend:
        image: example-registry.com:4000/mysql:8.0
        restart: on_failure
        environment:
                #MYSQL_ROOT_PASSWORD: example
                #.env
                MYSQL_ROOT_PASSWORD_FILE=/run/secrets/mysql-root
        volumes:
                - db-data:/etc/data
        networks:
                - back-tier
        Secrets:
                - mysql-password
```

Example Architecture

- Classic 3-tier architecture

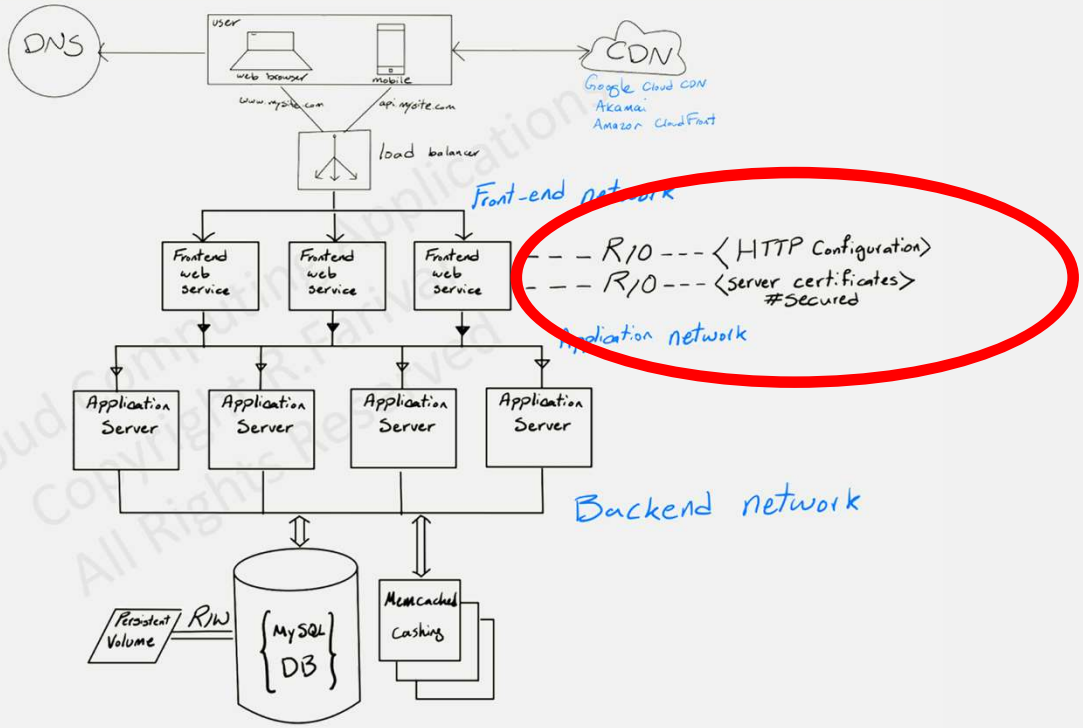


Volumes

```
volumes:
        db-data:
                driver: flocker
                driver_opts:
                        size: "10GiB"
```

Example Architecture

- Classic 3-tier architecture



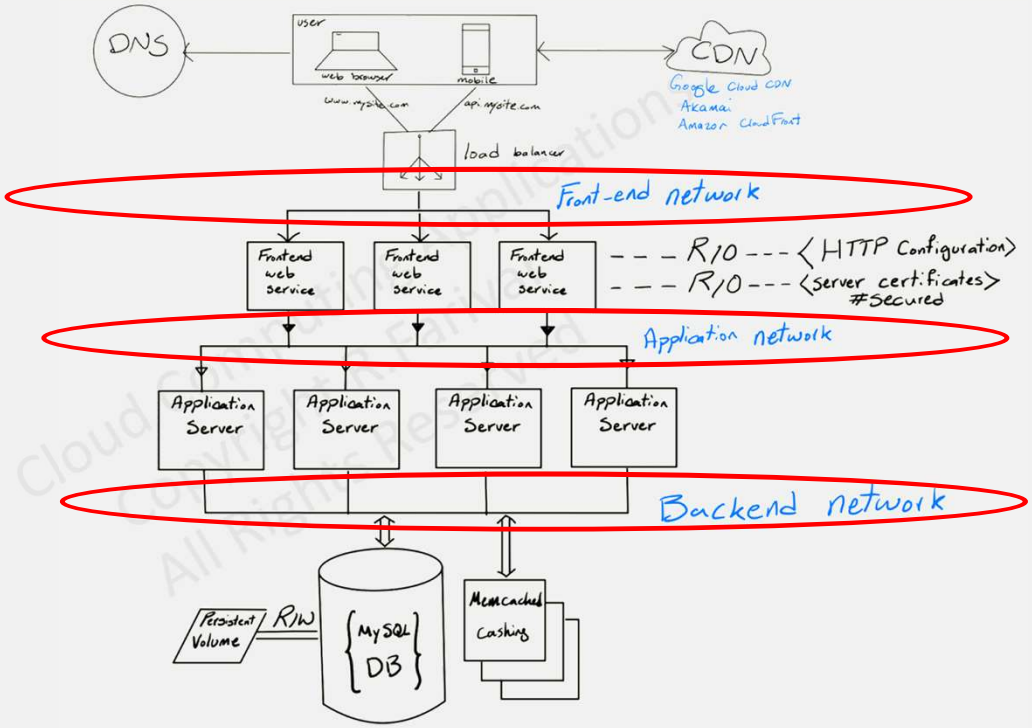
Configs and Secrets

```
configs:
        httpd-config:
                external: true
secrets:
        server-certificate:
                external: true
        mysql-password:
                file: ./my_secret_password.txt
```

it has already been defined in Docker, either by running the `docker secret create` *command or by another stack deployment*

Example Architecture

- Classic 3-tier architecture



Networks

```
networks:
        # The presence of these objects is sufficient to define
them
        front-tier:
        application-tier:
                driver: overlay
                ipam:
                        driver: default
                        config:
                                - subnet: 172.28.0.0/16
                                  ip_range: 172.28.5.0/24
                                  gateway: 172.28.5.254
        back-tier:
```

CLOUD COMPUTING APPLICATIONS

Docker on Amazon Elastic Container Service (ECS)

Prof. Reza Farivar

Amazon Elastic Container Service (ECS)

- Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service
- Containers either run on customer-managed EC2 instances, or on AWS Fargate
- Fargate: Serverless backend for container deployment
  - AWS manages resource provisioning for container instances
- Services such as Amazon Sagemaker and Lex internally run on ECS

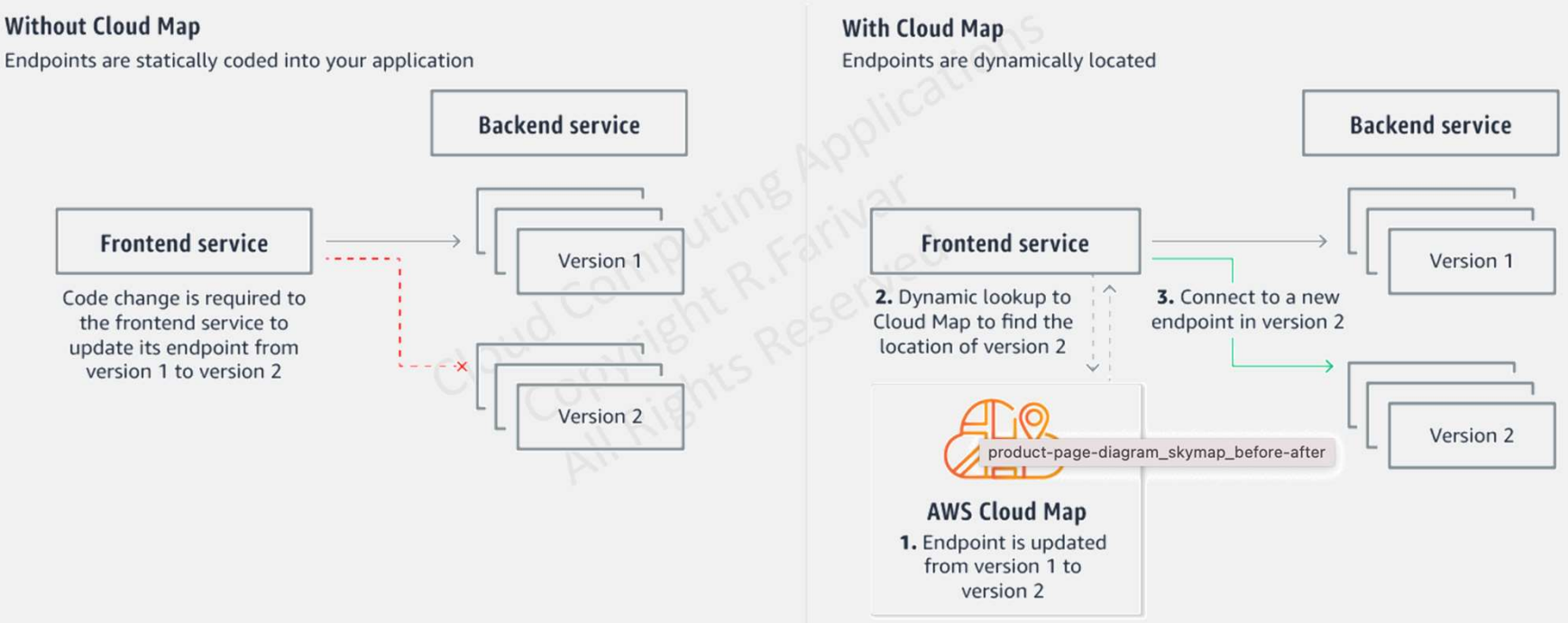
ECS Task Definition

- The Docker image to use with each container in your task
- How much CPU and memory to use with each task or each container within a task
- The launch type to use, which determines the infrastructure on which your tasks are hosted
- The Docker networking mode to use for the containers in your task
- The logging configuration to use for your tasks
- Whether the task should continue to run if the container finishes or fails
- The command the container should run when it is started
- Any data volumes that should be used with the containers in the task
- The IAM role that your tasks should use

Example AWS ECS Task Definition

```
{
   "containerDefinitions": [
      {
         "command": [
            "/bin/sh -c \"echo '<html> <head> <title>Ar
         ],
         "entryPoint": [
            "sh",
            "-c"
         ],
         "essential": true,
         "image": "httpd:2.4",
         "logConfiguration": {
            "logDriver": "awslogs",
            "options": {
               "awslogs-group" : "/ecs/fargate-task-de
               "awslogs-region": "us-east-1",
               "awslogs-stream-prefix": "ecs"
            }
         },
         "name": "sample-fargate-app",
         "portMappings": [
            {
               "containerPort": 80,
               "hostPort": 80,
               "protocol": "tcp"
            }
         ]
      }
   ],
   "cpu": "256",
   "executionRoleArn": "arn:aws:iam::012345678910:role/
   "family": "fargate-task-definition",
   "memory": "512",
   "networkMode": "awsvpc",
   "requiresCompatibilities": [
      "FARGATE"
   ]
}
```

Service Discover: AWS Cloud Map



Using Docker Compose File Syntax in ECS

- The `ecs-cli compose` and `ecs-cli compose service` commands allow you to create task definitions and manage your Amazon ECS tasks and services using Docker Compose files
- Additional ECS parameters specified for the container size parameters in a separate YAML file

Example

```
version: '3'
services:
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    logging:
      driver: awslogs
      options:
        awslogs-group: /ecs/cli/tutorial
        awslogs-region: us-east-1
        awslogs-stream-prefix: nginx
```

hello-world.yml

```
version: 1
task_definition:
  services:
    nginx:
      cpu_shares: 256
      mem_limit: 0.5GB
      mem_reservation: 0.5GB
```

ecs-params.yml

```
ecs-cli compose --project-name hello-world
--file hello-world.yml --ecs-params ecs-params.yml
--region us-east-1 create --launch-type EC2
```

Docker AWS ECS context

- Another method: Directly use docker
  - Set up an AWS context in one Docker command
    - `$ docker context create ecs myecscontext`
  - Use Compose files
    - `--context myecscontext`
    - `docker compose up --context myecscontext`
- Docker translates the compose application to CloudFormation template
  - Actual mapping:
    - https://github.com/docker/compose-cli/blob/main/docs/ecs-architecture.md

Native AWS Services

- ECS cluster for the Compose application
- Use the default VPC
  - A Security Group per network in the Compose file on the AWS account's default VPC
  - LoadBalancer to route traffic to the services
- Service Discovery (service-to-service load balancing) is handled by AWS Cloud Map
- Volumes are handled by instantiating EFS filesystems
- Secrets handled by AWS Secrets Manager

```yaml
services:
  webapp:
    image: ...
    secrets:
      - foo

secrets:
  foo:
    name: "arn:aws:secretsmanager:eu-west-3:1234:secret:foo-ABC123"
    external: true
```

Docker integration with Azure

- Docker is also integrated with Azure Container Instances
  - `$ docker login azure`
  - `$ docker context create aci myacicontext`
  - `$ docker --context myacicontext run -p 80:80 nginx`
- Volumes → Azure File Share
- Port mapping → Only symmetrical mapping 80:80
- Networks not supported
  - At least not until the recording of this video in 2021
  - Communication between services is implemented by defining mapping for each service in the shared `/etc/hosts` file of the container group.