Docker and AWS Elastic Container Services (ECS)

SwissLife Workshop - 5.7.2022

Version: 1.0.3

Agenda

- Container (Docker) refresh
- AWS Fargate Foundations (managed Containers)
- HA/Scaling of Instances (LBs & ASGs)
- Container Security
- Datapersistence
- Terraform Examples

Goals of the Course

- getting a refresh / intro to containers
- Hands-On Time
- use of containers
- use Terrafrom to deploy some containers

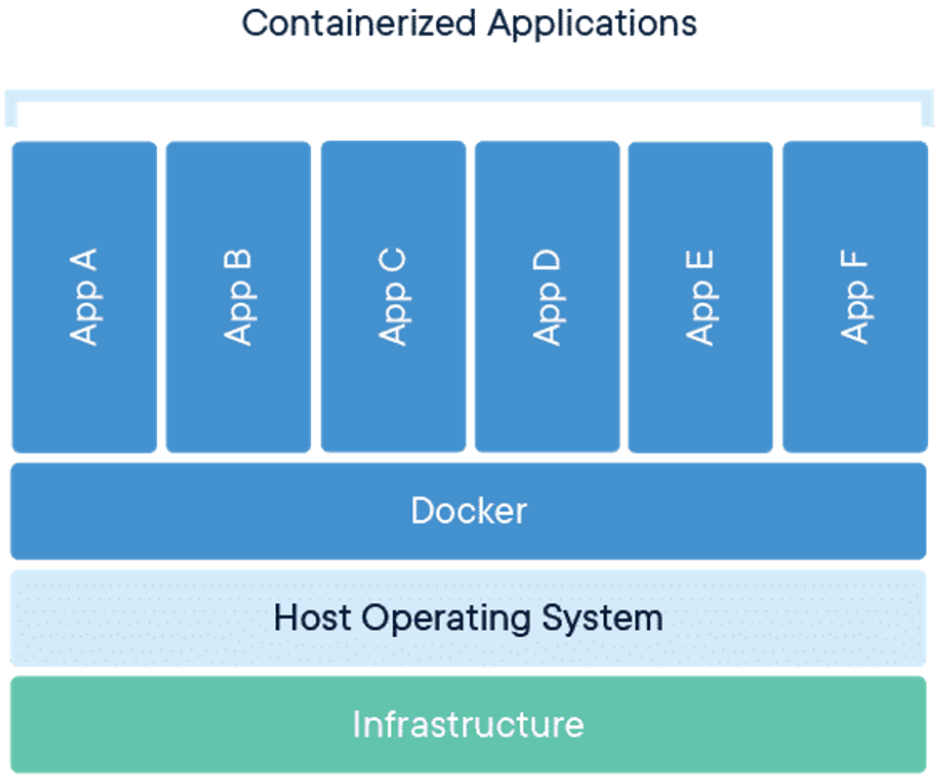
container refresh

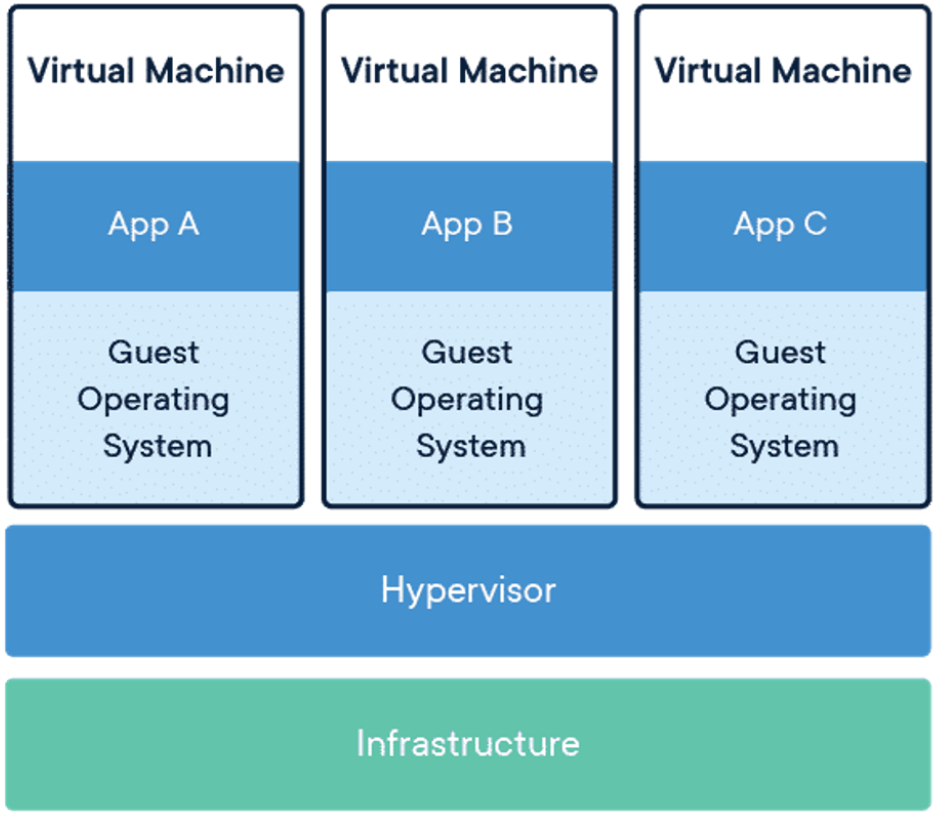
Containers provide a standard way to package your application's code, configurations, and dependencies into a single object.

Containers run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment.

Whether you deploy locally on your laptop or to production, the experience will remain the same.

container vs. vm

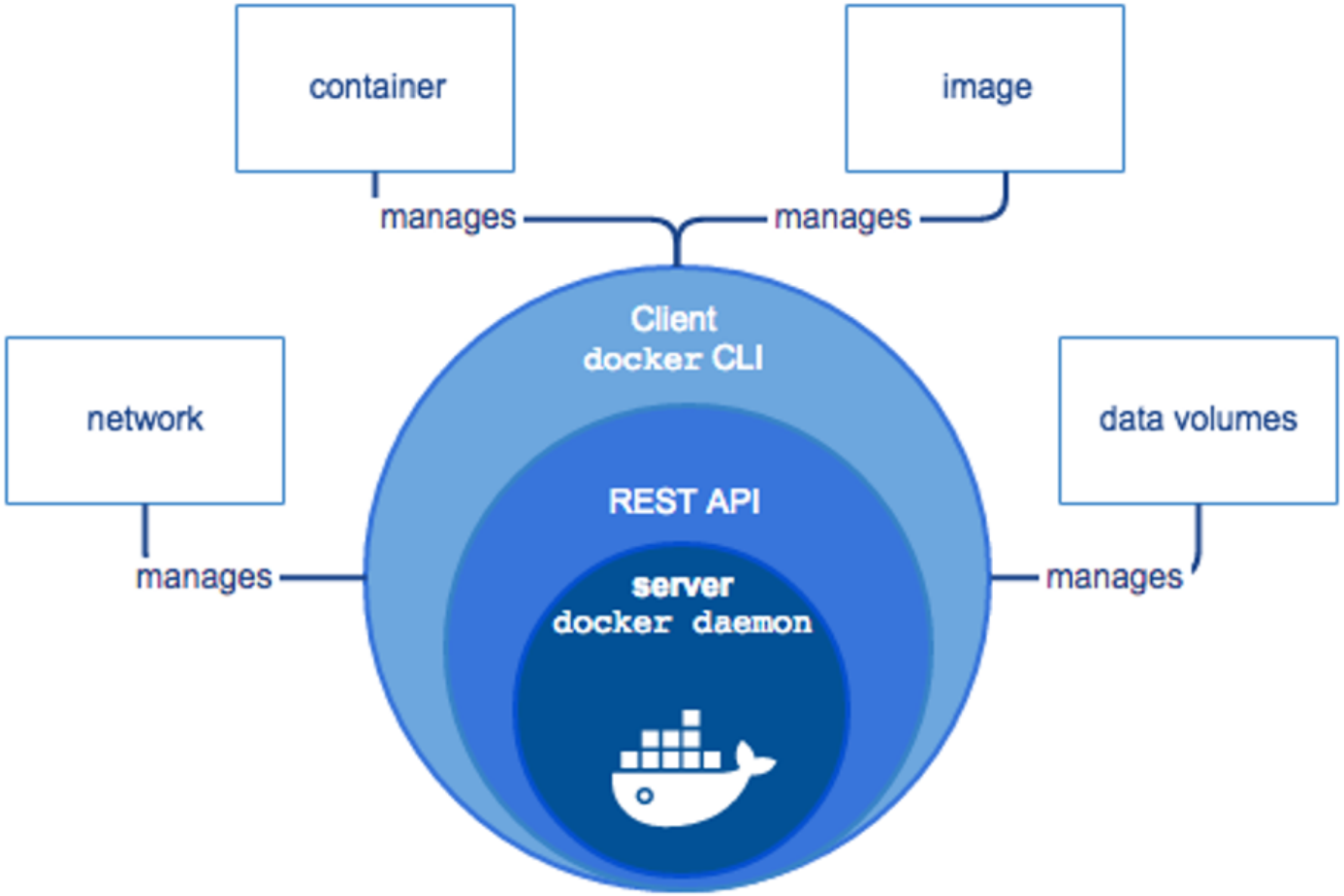




Docker refesh

- Open Source since March 2013
- License Apache 2.0
- commercial support durch Docker Inc.
- Docker was developed in GO
- Initally a wrapper for LXC
- Support for Windows Server 2016

Docker Architecture



container standard is OPEN

- breakup the All-in-one-Package
- OCI - open container initiative
- alternative runtimes like CRI-O, rkt, ...
- `containerd` as the runtime from Docker

Docker quick-install and Test

- Create an EC2 (t2.medium) Instance
- Install Docker the "easy"-way

```
$ curl -fsSL https://get.docker.com/rootless -o get-docker.sh
```

```
$ sh get-docker.sh
```

- Test Docker install

```
$ docker run -p 80:80 nginx
```

- Test the webpage

```
$ curl http://localhost
```

Deploy docker on Amazon Linux 2

- create EC2 t3.medium instance
- install the software:

```
$ sudo yum update -y
$ sudo amazon-linux-extras install docker -y
$ sudo service docker start
$ sudo systemctl enable docker
$ sudo usermod -a -G docker ec2-user
$ sudo docker info
```

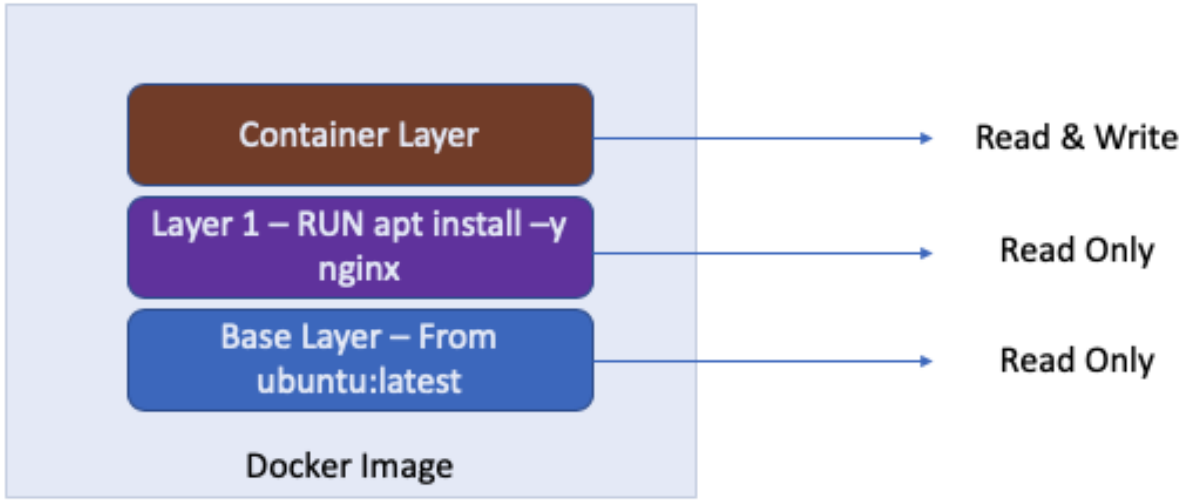
LAB

create a Docker server (console)

- create an ec2 `t3.medium` instance
- configure it for a `public` subnet
- install `Docker` to the machine
- test installation with
  `docker run hello-world`
- test installation with
  `docker run -p 80:80 nginx`
- access the webpage

Build your own container

- starting with a Base-Image
- customize you ressources
- build your image in multiple stages (12Factor-App)
- alternative Tools for building Images (Buildah, buildx, Packer)



Some Dockerfile commands

Command	Purpose
FROM	To specify the parent image.
WORKDIR	To set the working directory .
RUN	To install any applications and packages
COPY	To copy over files or directories from a specific location.
ENTRYPOINT	Command that will always be executed when the container starts.
CMD	Arguments passed to the entrypoint.
EXPOSE	To define which port to access

simple - Dockerfile example

```
FROM nginx:alpine
RUN rm -rf /usr/share/nginx/html/*
COPY index.html /usr/share/nginx/html
EXPOSE 80
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

multistage - Dockerfile example

```
# Build the AngularApp inside a container
FROM node:16-alpine as builder
COPY package.json package-lock.json ./
RUN npm install && mkdir /app-ui && mv ./node_modules ./app-ui
WORKDIR /app-ui
COPY . .
RUN npm run ng build -- --deploy-url=/sampleApp/ --prod

# Deploy the App to nginx-Container
FROM nginx:alpine
RUN rm -rf /usr/share/nginx/html/*
COPY --from=builder /app-ui/dist /usr/share/nginx/html
EXPOSE 4200 80
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

LAB

create your own image

- create a simple Website (index.html)
- create a `Dockerfile`
- use `nginx:1.20` as the base-image
- build image `webserver:1`
- test the new image

Registry - a place to store Images

- Docker
  - Registry (self-hosted)
  - hub.docker.com (hosted)
- 3rd Parties
  - GitLab
  - Artifactory
  - Nexus
  - AWS ECR

Amazon Elastic Container Registry (ECR)

- AWS managed container image registry
- secure, scalable, and reliable
- supports private repositories with resource-based permissions (IAM)
- use your tools to push, pull, and manage
    - Docker images
    - Open Container Initiative (OCI) images
    - OCI compatible artifacts

Features of Amazon ECR

- Lifecycle policies, managing the lifecycle of the images in your repositories. Define rules that result in the cleaning up of unused images.
- Image scanning helps in identifying software vulnerabilities in your container images. Each repository can be configured to scan on push. You can then retrieve the results of the image scan.
- Cross-Region and cross-account replication makes it easier for you to have your images where you need them.
- Pull through cache rules provide a way to cache repositories in remote public registries in your private Amazon ECR registry.

LAB

create & use a private ECR

- create an programmatic IAM account `ecr-admin`
- add policy `AmazonEC2ContainerRegistryFullAccess`
- create a private Registry `ecs-demo`
- tag the image `webserver:1` with the new ecr-server repo `ecs-demo:1.0`
- try to push it to the registry

AWS Fargate - Fundamentals (managed Container)

- highly scalable container service
- supports Docker containers
- run and scale containers on AWS
- eliminates the need to operate own clusters
- managed service to run and scale a cluster

LAB

EC2 vs. Fargate

- try to find 3 differnces
- Hints:
  - maybe usecases?
  - max. Memory?
  - ...

Terms of AWS ECS-Ecosystem

- Task Definitions
- Cluster
- Tasks and Scheduling
- Services

Task Definition

- think of it as a blueprint for your application.
- describes one or more containers that form your application
- is a text file (JSON)
- is a Set of Task definition parameters

JSON Example for Logfiles

```
...
"image": "nginx:1.20",
"memory": "1024",
"cpu": "512",
"containerDefinitions": [
    {
      "logConfiguration": {
        "logDriver": "awslogs",
        "secretOptions": null,
        "options": {
          "awslogs-group": "/ecs/ecs-test",
          "awslogs-region": "eu-central-1",
          "awslogs-stream-prefix": "ecs"
        }
      },
...
```

LAB

create a TaskDefintion

- create a TaskDefinition `ecs-demo`
- choose the `ecs-demo:1.0` image from the private ECR
- port `80` should be open
- allocate `0.5` CPU
- allocate `1GB` of RAM

Cluster

A cluster is basically the logical grouping of resources that your application needs. If you use Fargate launch type with tasks within clusters then Amazon ECS manages your cluster resources. If you use EC2 launch type, then your clusters will be a group of Amazon EC2 container instances that you manage.

LAB

create an ECS Cluster (Fargate)

- create a cluster with name `ecs-cluster`
- use `Networking only`
- create a `new VPC` for this cluster
- enable `CloudWatch Container Insights`

Run Task

A task is the instantiation of a task definition within a cluster. After you have created a task definition for your application within Amazon ECS, you can specify the number of tasks that will run on your cluster.

Each task that uses the Fargate launch type has its own isolation boundary and does not share the underlying kernel, CPU resources, memory resources, or elastic network interface with another task.

LAB

create a Task

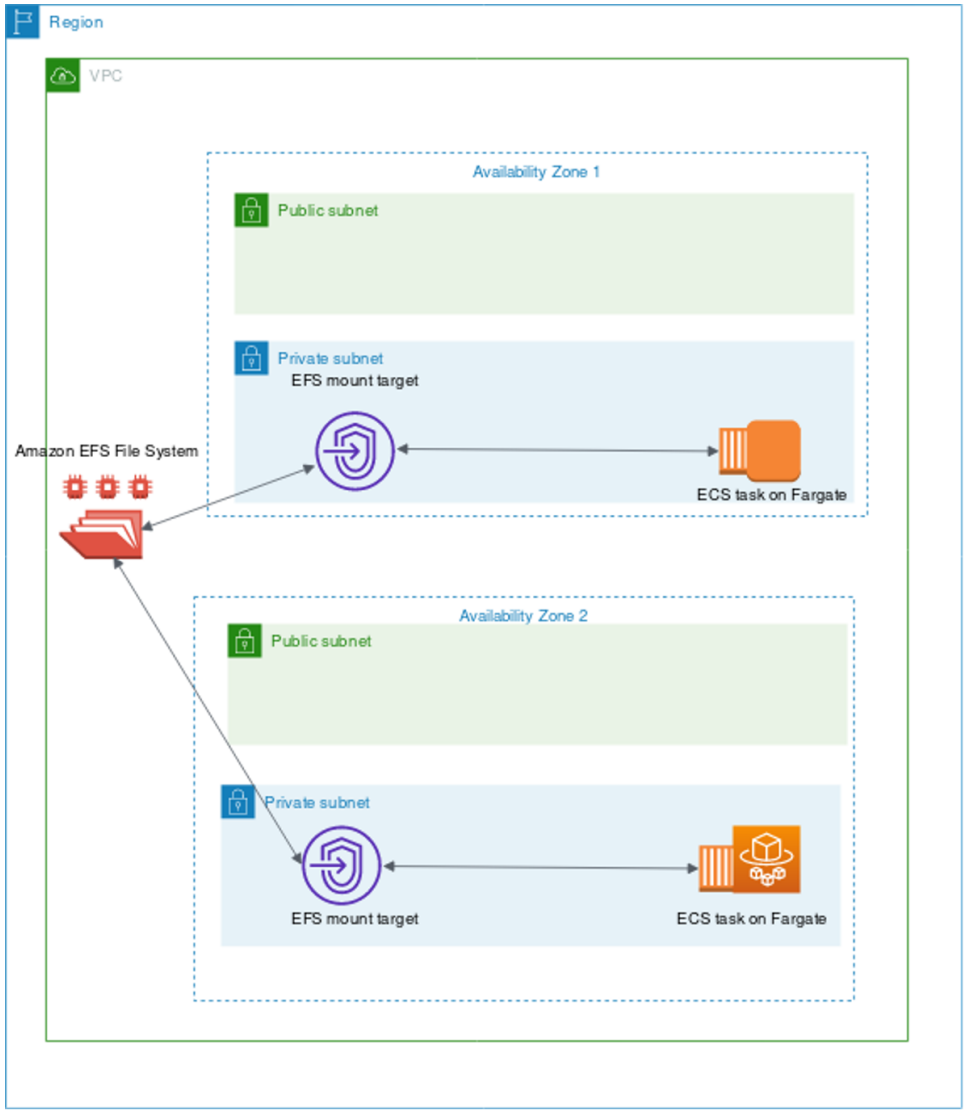
- launch type `fargate`
- OS type `Linux`
- use TaskDefinition `ecs-demo` with Revision `1`
- add both Subnets
- use security group `sg-webserver`
- Auto-assign public IP `enabled`
- check the App in a Browser

Datapersistence

- everything in a container is ephermal
- Default Container Directory `/var/lib/docker/container`
- Volumes are the native way of Docker to handle Data
- AWS has the option to use EFS/FSx as an alternative

AWS EFS with ECS

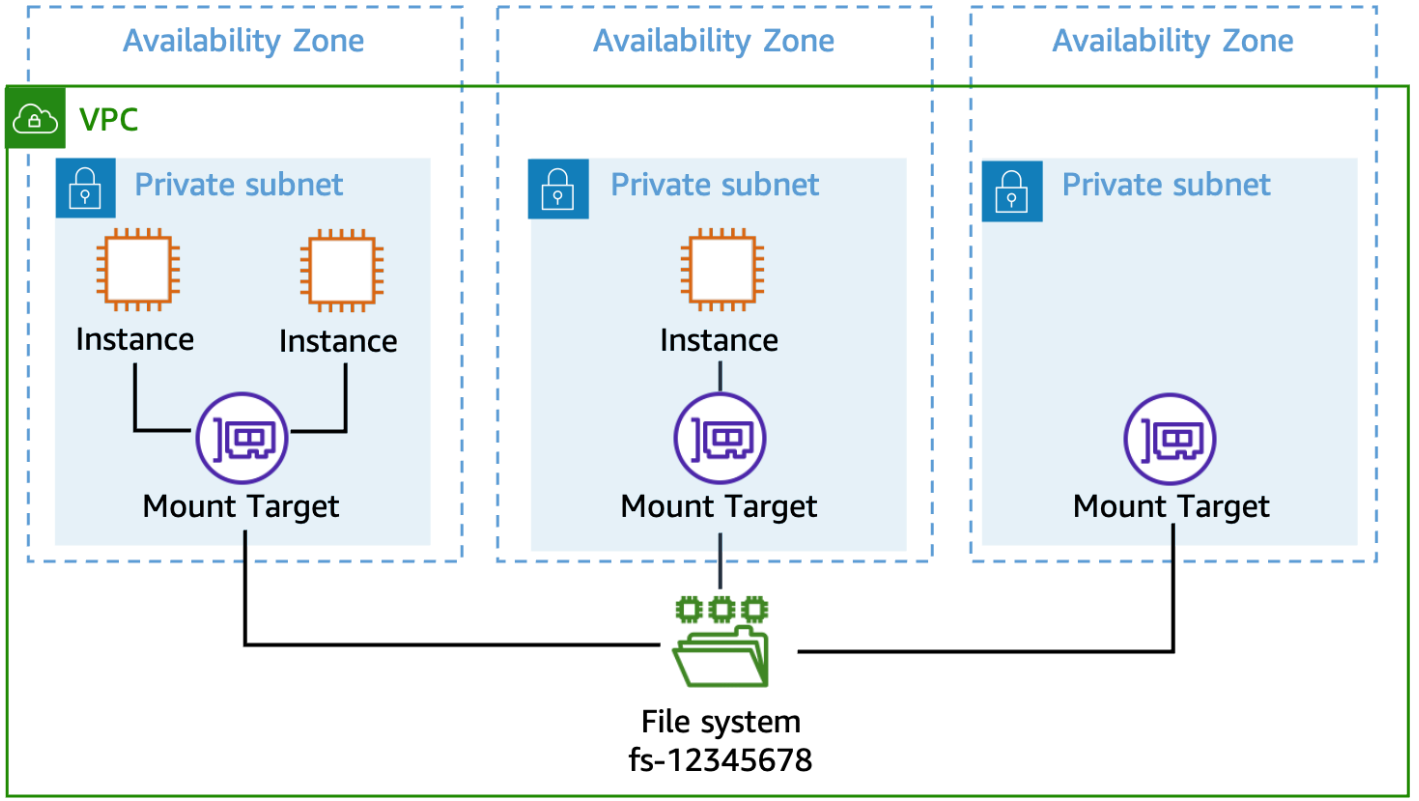
- persistent file storage for Tasks
- storage capacity is elastic (grows and shrinks automatically)
- EFS volumes are supported for Fargate and EC2
- NFS Driver "built-in" for Fargate



AWS Elastic File System

- scalable, elastic file system for Linux
- for use with AWS services and on-premises.
- Network File System (NFSv4 - 4.0/4.1)
- no action to expand the file system
- fully managed by AWS

AWS EFS Architecture



LAB

create & test AWS EFS

- Create security group `efs-access-ecs` with Inbound rule `NFS` and source `10.0.0.0/16`
- use EFS-Service and create file-system `efs-ecs` with custom-options
- add the security group `efs-access-ecs` to the mount targets
- create a new `ec2-instance`

```
$ sudo -i
$ cd /
$ mkdir /efs
$ mount -t nfs4 -o ...<command from EFS-Attach>
$ vim /efs/index.html
```

TaskDefinition Implementation

```
"mountPoints": [
    {
        "containerPath": "/usr/share/nginx/html",
        "sourceVolume": "efs-html"
    }
],
```

```
"volumes": [
    {
        "name": "efs-html",
        "efsVolumeConfiguration": {
            "fileSystemId": "fs-1324abcd",
            "transitEncryption": "ENABLED"
        }
    }
],
```

LAB

create a Task with EFS-Volumes

- use the efs-filesystem from the last lab
- create a new task revision
- add a mountPoints section for `/usr/share/nginx/html`
- add a volumes for EFS-Filesystem
  - Name: `efs-storage`
  - VolumeType: EFS
  - FilesystemID: <EFS-ID>
- run the Task
- test the EFS-Volume

ECS Service

When an ECS task is launched, you can assign it to a service that controls:

- how many tasks should be launched
- which version of the task should be launched
- how to scale up or down
- how to deploy updates
- utilize a loadbalancer

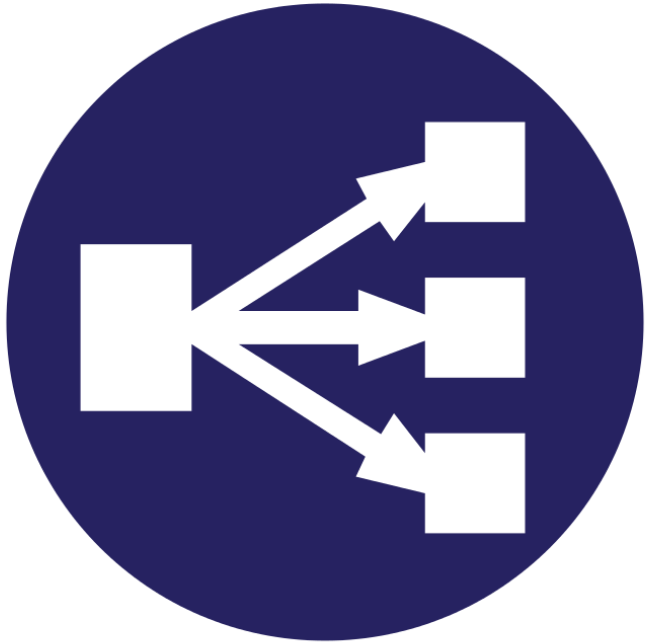
LAB

create a simple Service

- launch type `fargate`
- Number of tasks `2`
- Service name `ecs-service`
- add both Subnets
- Auto-assign public IP `enabled`
- no Loadbalancer (this time)
- ***test*** if the service recreates

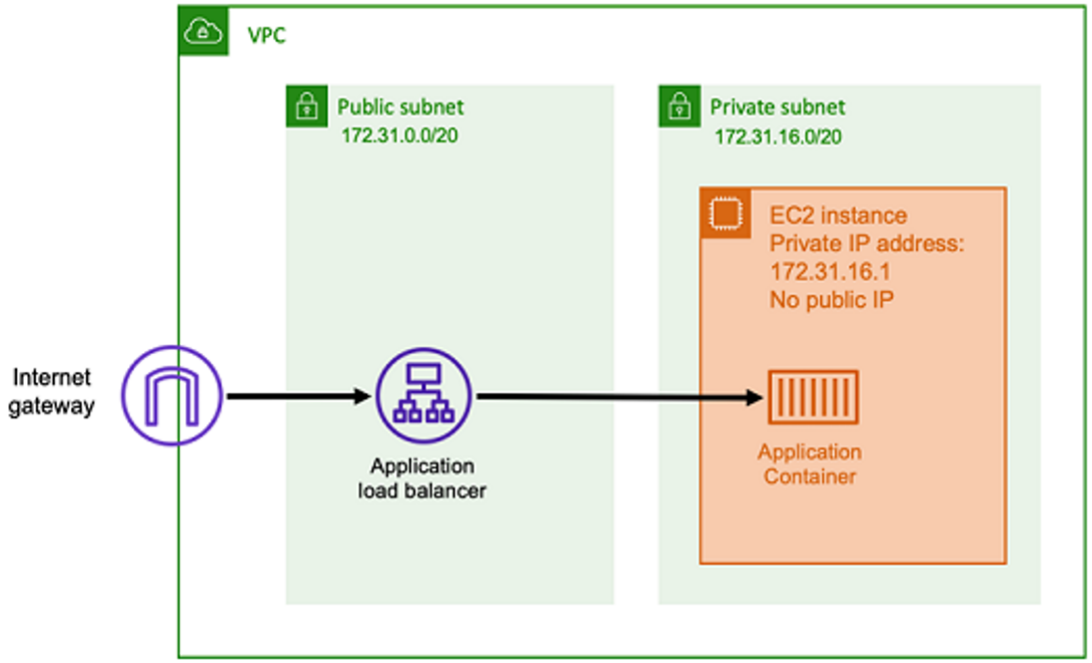
Amazon Elastic Load Balancer

- LB is a best practice for ingress-traffic
- it funnels requests via a Listener
- uses Healthcheck for Target-Groups
- AWS ELB scales by traffic (SLA!)



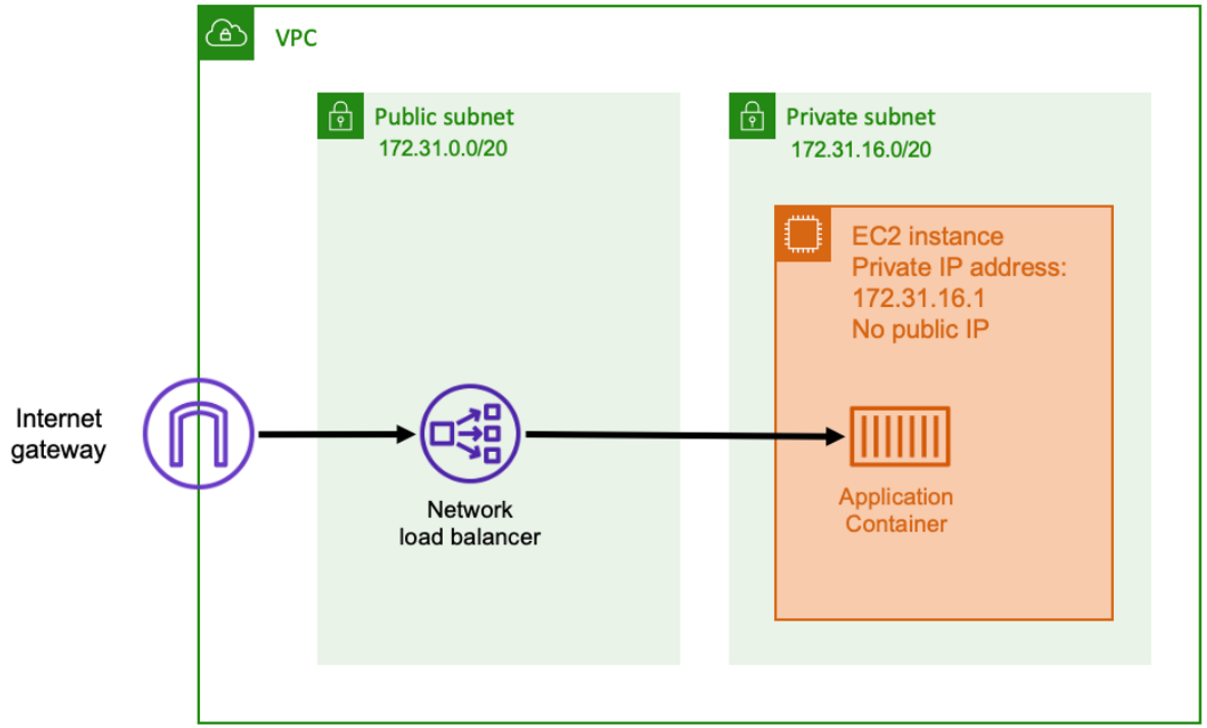
Application Loadbalancer

- an ALB functions at the seventh layer of the OSI model
- ALB is suitable for public HTTP services.
- A website or an HTTP REST API is a suitable for this workload.



Network Loadbalancer

- an NLB functions at the fourth layer of the OSI model.
- NLB suitable for non-HTTP protocols or scenarios where end-to-end encryption is necessary.
- NLB is best suited for applications that don't use HTTP.



How to configure a Loadbalancer

- First create a `Target Group`
- Type is `IP addresses` because of awsvpc
- create the `ALB` and attach the `TargetGroup`

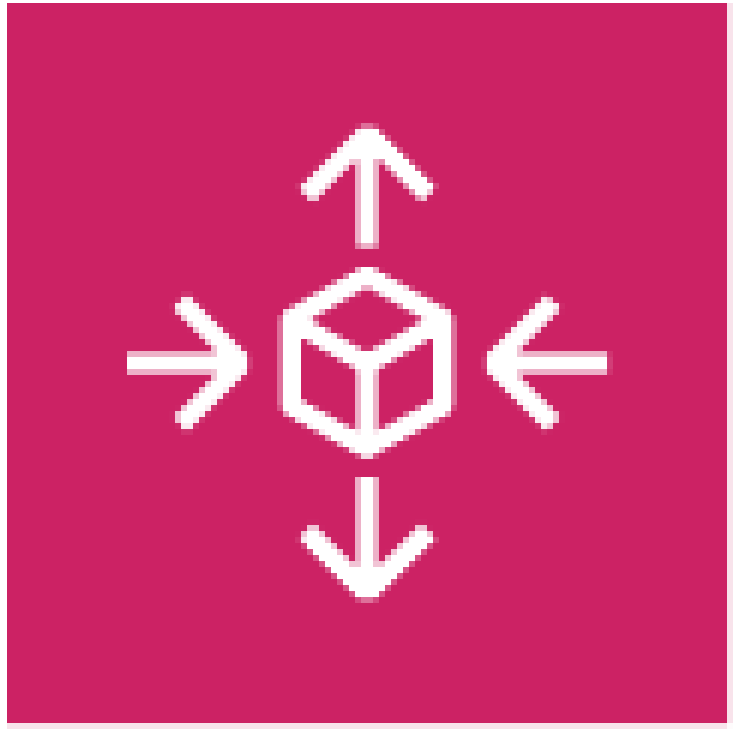
LAB

create a loadbalanced Service

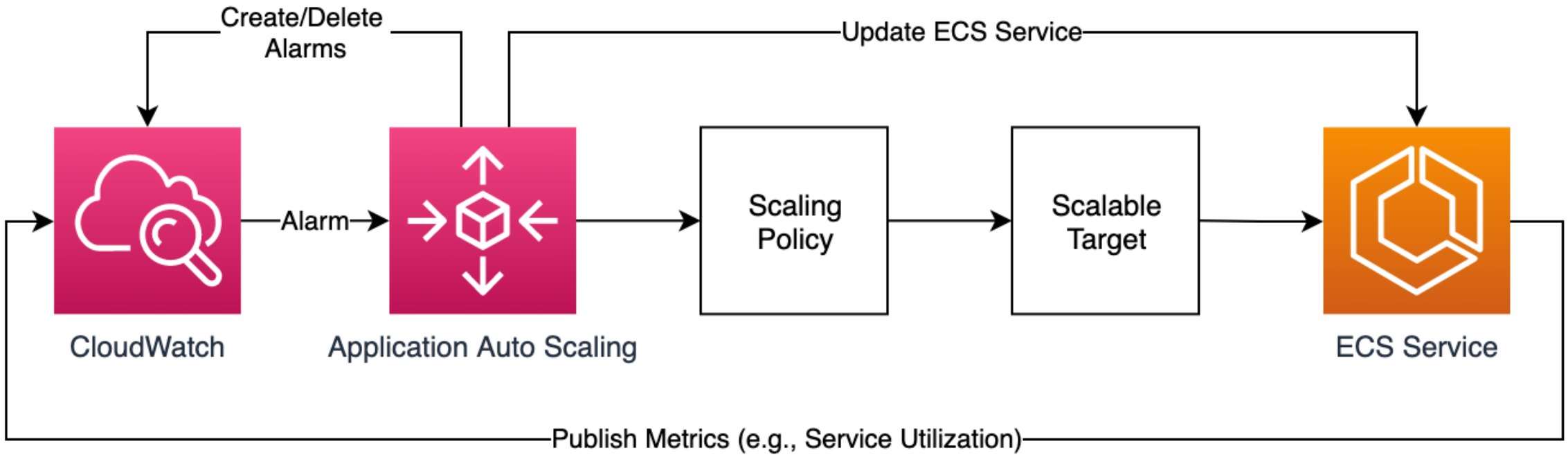
- EC2-Console
  - create a Target Group `tg-ecs-lb` with target type `IP addresses`
  - create an ALB with name `alb-ecs-demo`
- ECS-Settings like before plus:
  - create a new Service `ecs-service-lb`
  - add `ALB` to the service
  - `Test` ALB-Adress in Browser

Container Autoscaling

- Application Autoscaling
- Scaling-Policy Fundamentals
- use Loadbalancer-Metrics with CloudWatch



AutoScaling 1/2



AutoScaling 2/2

Scalable Target

The auto scaling service will update the ECS service's desired task count based on the particular scaling policy

Scaling Policy

The scaling policy defines when a scaling event — either scale in or scale out — should occur. There are several types of scaling policies:

- Target Tracking policies
- Step scaling policies

Target Tracking policies

Out of box, three ECS Service Metrics are supported for ECS Application Auto Scaling.

- Average CPU Utilization
- Average Memory Utilization
- Request Counts Per Target

For example, we can set `Request Count Per target` at 1000, it adjusts the number of tasksto keep the `Request Count target` at 1000.

Step scaling policies

With step scaling, you choose scaling metrics and threshold values for the CloudWatch alarms.

Step scaling policies increase or decrease the current capacity of a scalable target based on a set of scaling adjustments, known as step adjustments.

- A lower bound for the metric value
- An upper bound for the metric value
- The amount by which to scale, based on the scaling adjustment type

Testing with a pre-build Taurus-Image

test-asg.yml

```
execution:
  - concurrency: 100
    ramp-up: 15s
    hold-for: 300s
    scenario: sample

scenarios:
  sample:
    timeout: 500ms
    requests:
    - http://<LB-Address>/
```

Start via Docker

```
docker run -it -v /root/perf:/bzt-configs blazemeter/taurus test-asg.yml
```

cutom Loadtesting via Taurus

Dockerfile

```
FROM python
RUN pip install bzt
RUN apt-get update && apt-get -y install default-jre-headless
RUN bzt -install-tools -o modules.install-checker.include=jmeter
ENTRYPOINT ["bzt"]
```

Build the Image

```
$ docker build -t bzt .
```

Run the Image

```
$ docker run -it bzt http://<Address>:<Port>
```

LAB

create a scaling Service

- Service `svc-asg-demo`
- TargetGroup `tg-asg-demo`
- Loadbalancer `alb-asg-demo`
- Set `Auto Scaling`
  Min. # of tasks `1` and max # `10`
- TargetTracking `plc-asg-demo`
  - `Request Count Per target`
  - TargetValue: `75.0`
  - Both Cooldown: `60`

Terraform Examples

LAB 00

create a VPC for ECS

- create a vpc `10.0.0.0/16`
- public subnet `10.0.1.0/24` , `10.0.2.0/24`
- private subnet `10.0.10.0/24` , `10.0.20.0/24`
- with 2 AZs `eu-central-1a` , `eu-central-1b`
- use a `nat-gateway`

LAB 01

create an ECR

- create a ECR `ecs-demo`
- Images should be `IMMUTABLE`
- create a Repo-Policy (optional)

LAB 02

create iam

- create a `ecsTaskExecutionRolePolicy`
- create a `ecsTaskExecutionRole`
- Security Group `allow-http`
  - allow `inbound` Port `80` from `all`
  - allow `outbound` Port `any` to `all`

LAB 03

create a sg for the container

- Security Group `allow-http`
  - allow `inbound` Port `80` from `all`
  - allow `outbound` Port `any` to `all`

LAB 04

create a TaskDefinition

- create a TaskDefinition `ecs-demo`
- image: `nginxdemos/hello`
- memory: `1024`
- cpu: `512`
- PortMapping: `80`

LAB 05

create an ECS-Cluster

- create a ECS `ecs-demo`
  - Enable `containerInsights`
- create a Task `hello-worl-app`
  - Type `FARGATE`
  - Public-IP `enabled`

LAB 06

create a SG for ALB/Service

- Security Group `alb-http`
  - allow `inbound` Port `80` from `all`
  - allow `outbound` Port `any` to `all`
- Security Group `allow-http-from-lb`
  - allow `inbound` Port `80` from `loadbalancer`
  - allow `outbound` Port `any` to `all`

LAB 07

create an ALB

- create a Target Group
- create a LoadBalancer
- create a Loadbalancer-Listener

LAB 08

create a TaskDefinition ALB

- create a TaskDefinition `hello-world-alb`

LAB 09

create a Service with ALB

- create a Service `svc-hello-world`
  - Number of Tasks: `2`
  - Type: `REPLICA`
  - Public-IP: `disabled`
- attach the ALB

LAB 10

create Scaling

- create an `Autoscaling-Target`
  - scalable Dimension: `"ecs:service:DesiredCount"`
  - scaling: `min 1 / max 10`
- create two Cloudwatch-Alarms
  - Metric: `CPUUtilization`
  - Namespace: `AWS/ECS`
  - Threshold: `70`
  - Actions: `scale policy up/down`