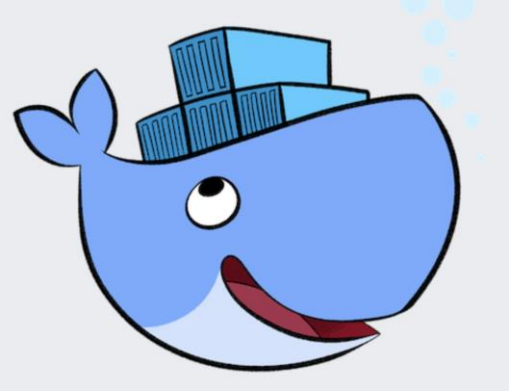
Docker

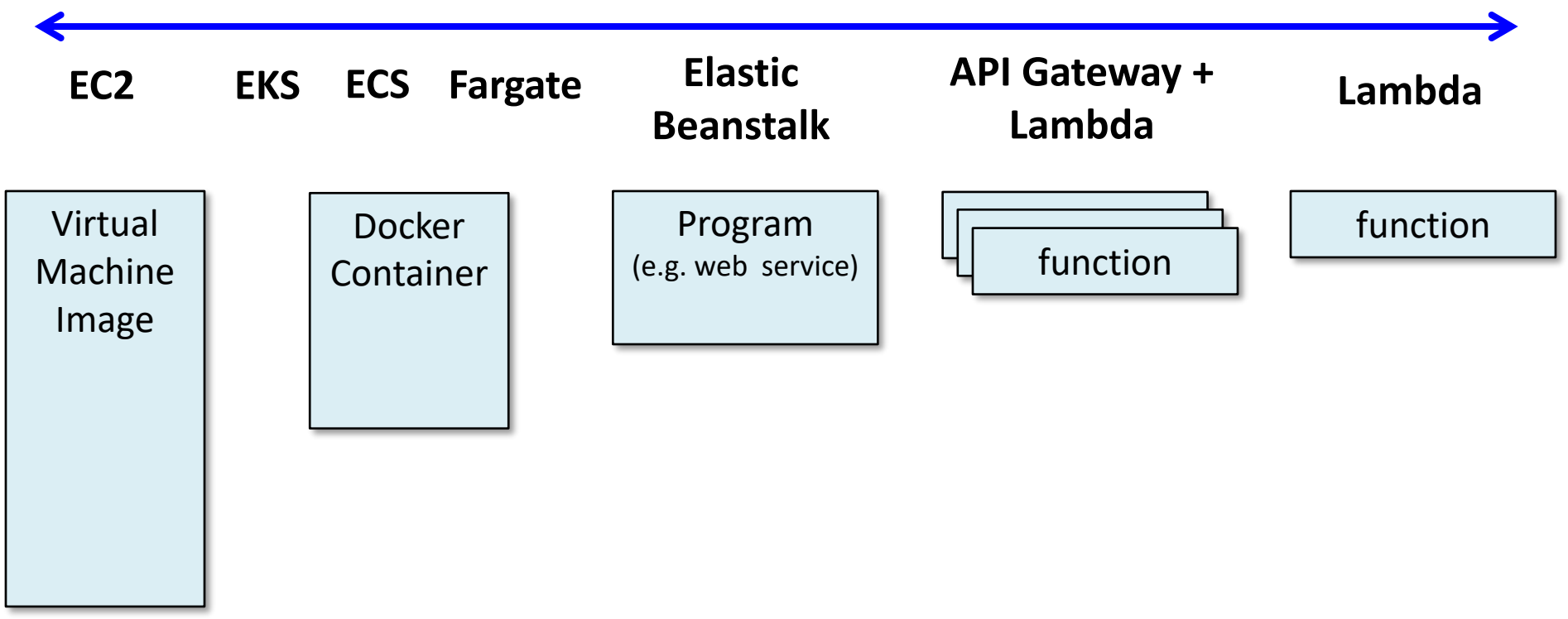
- What is Docker
- Dockerfiles, images, containers
- Examples outside AWS
- Examples within AWS

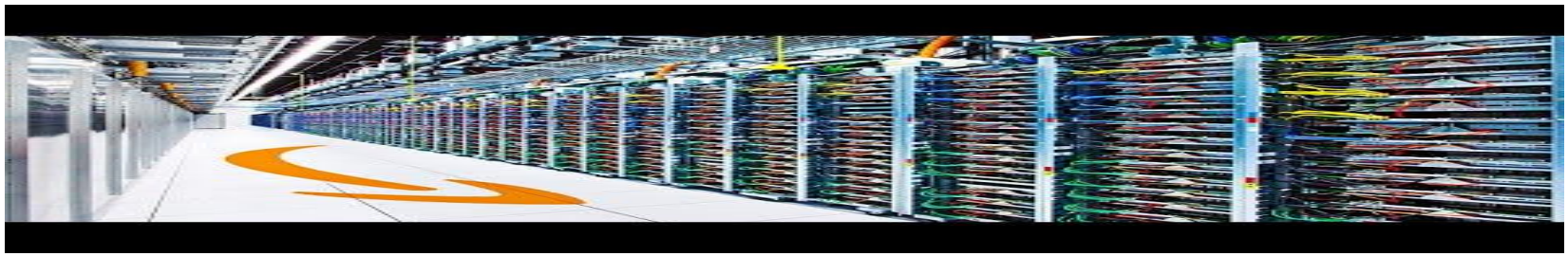
Docker

- **What is Docker?**
  - Docker is an open platform for developing, shipping, and running applications
  - Docker allows you to run software by installing just two things:
    - Docker desktop
    - Docker image

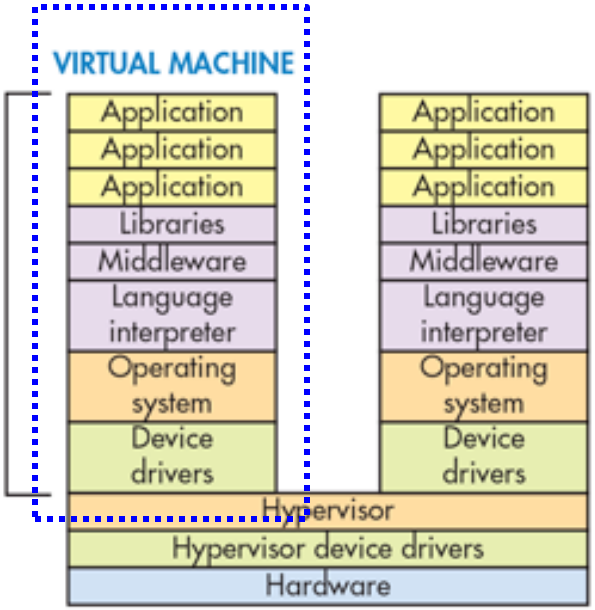


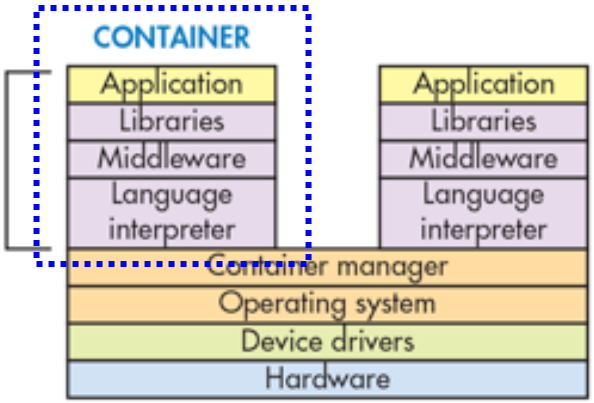
Software packaging options

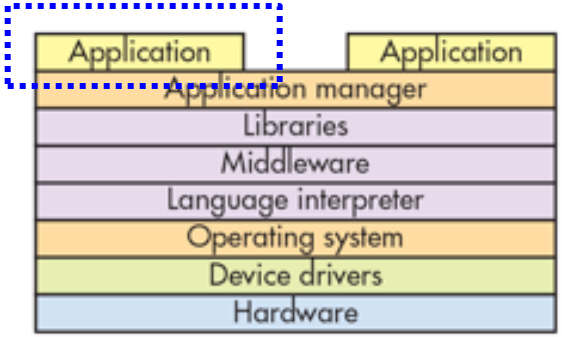




Software packaging trade-offs







VIRTUAL MACHINES

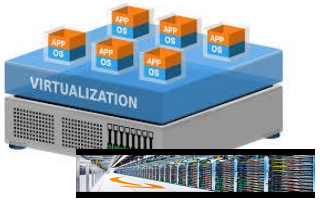
- ✓ Complete control
- ✓ Consistent environment
- ✓ HW-enforced separation/security
- ✗ High space overhead (VM size)
- ✗ Slow cold start since booting OS (=> slower to scale horizontally)

CONTAINERS

- ✓ Consistent runtime environment without overhead of a full VM
- ✓ Smaller in size => faster cold start (=> faster to scale horizontally)
- ✗ Less configurable than a VM
- ✗ Less secure? (SW-enforced)
- ✗ Apps cannot "see" each other, but can "feel" each other (sharing RAM, CPU)

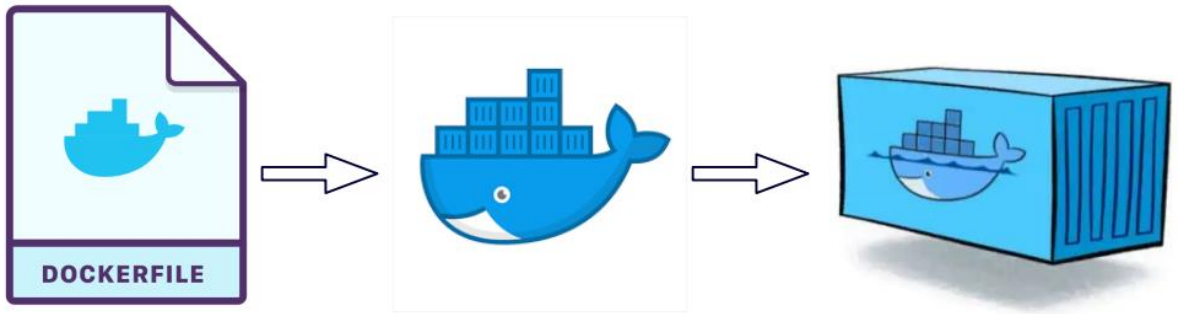
SERVERLESS

- ✓ Easiest to code and deploy
- ✓ Automatic scaling
- ✗ Longer latency (response time)
- ✗ <15 min



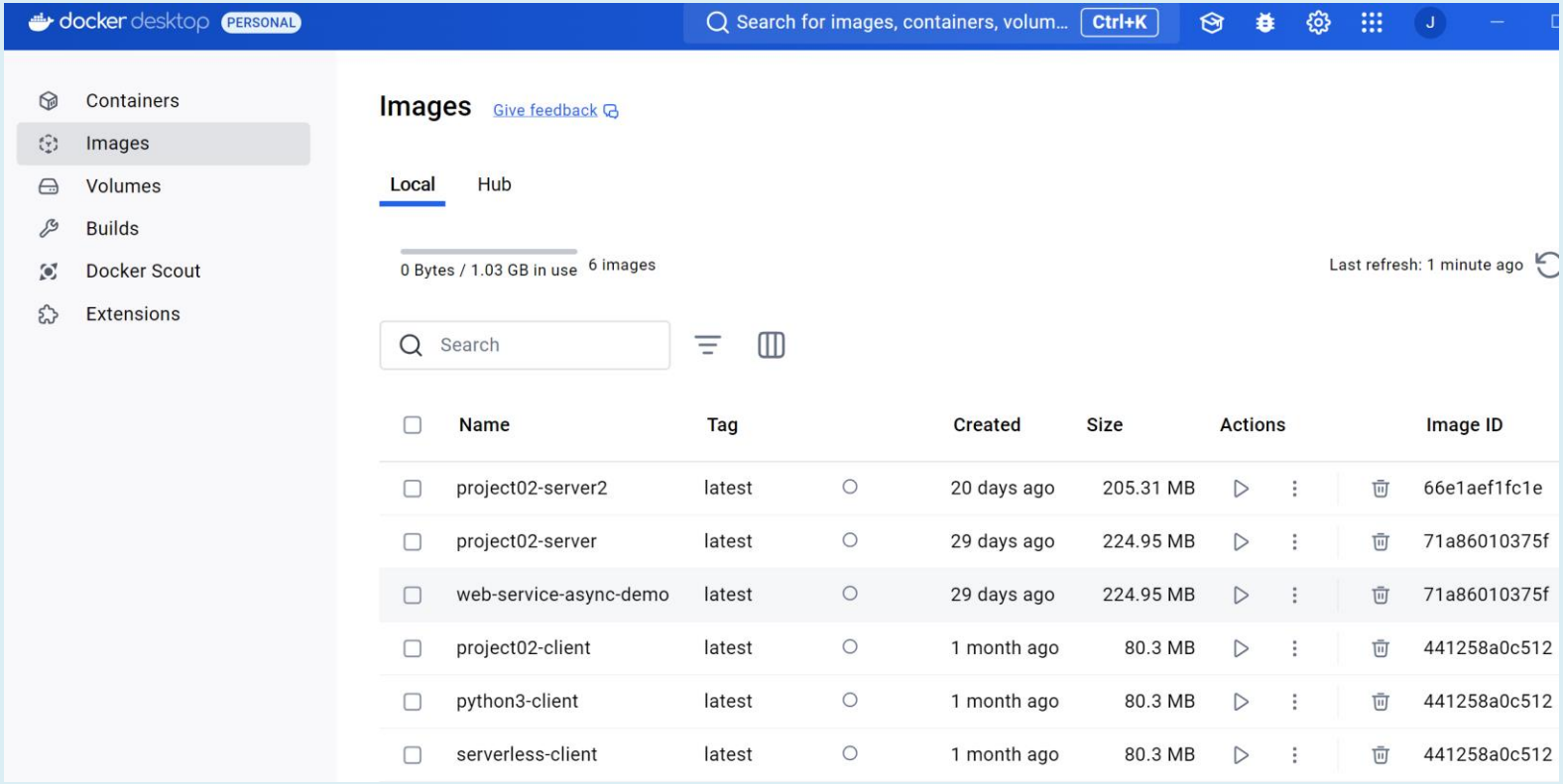
Overview

- A **Dockerfile** is a text document that contains the commands needed to build a **docker image**
- Think of a **docker image** as a snapshot of the software – a large file that can be stored and shared
  - ***Docker Hub*** *is the app store for docker images*
- A **docker container** runs a docker image



Images vs. Containers

- **Docker desktop** will show your images and containers…



Example #1

- Docker containers can run "interactively", allowing user to run & interact with the software:

```
hummel> ./docker-run.bash
project02-client> python3 main.py
** Welcome to PhotoApp v2 **

What config file to use for this session?
Press ENTER to use default (photoapp-client-config.ini),
otherwise enter name of config file>


>> Enter a command:
   0 => end
   1 => stats
   2 => users
   3 => assets
   4 => download
   5 => download and display
   6 => bucket contents
   7 => add user
   8 => upload
1
bucket status: success
# of users: 5
# of assets: 12

>> Enter a command:
   0 => end
   1 => stats
   2 => users
   3 => assets
   4 => download
   5 => download and display
   6 => bucket contents
   7 => add user
   8 => upload
```

Example #1 files

Dockerfile

```
FROM python:3.12.4-alpine3.20
#
# add bash to alpine Linux:
#
RUN apk update && apk upgrade
RUN apk add --no-cache bash
#
# turn off history file creation:
#
RUN echo "export HISTFILE=/dev/null" >> /etc/profile
#
# add a user (with no pwd) so we don't run as root:
#
RUN adduser -S user -G users -D
#
# install any additional python packages we need:
#
RUN pip3 install requests
RUN pip3 install jsons

```

docker-build.bash

```
#!/bin/bash
#
# Linux/Mac BASH script to build docker container
#
docker rmi project02-client
docker build -t project02-client .

```

remove existing image called that
make new one

docker-run.bash

```
#!/bin/bash
#
# Linux/Mac BASH script to run docker container
#
docker run -it -u user -w /home/user -v .:/home/user --network="host" --rm project02-client bash

```

we want access to the network

we want to interact w bash

-it is interactive

-v reach out to local filesystem, access them

Example #2

- Docker containers can run in the background to offer services, e.g. database server or web server

```
hummel> ./docker-run.bash
22908e6c767fb71fd4b4d9c0299dbf192e3ae4ec224cc1b74e9764ab4d1fb307
hummel>
hummel>
hummel>
hummel> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS
                    NAMES
22908e6c767f   docker-mysql   "docker-entrypoint.s…"   4 seconds ago   Up 3 seconds   3306/tcp, 33060/tcp, 0.0.0
.0:3307->3307/tcp   mysql
hummel>
hummel>
hummel>
hummel> docker stop mysql
```

mysql is not installed into my machine

Example #2 files

Dockerfile

```
FROM mysql:latest
#
# set root password for local execution:
#
ENV MYSQL_ROOT_PASSWORD=abc123
#
# NOTE: changing to port 3307 since I already have MySQL installed
# and running locally on its own. So this docker image is a second
# version.
#
ENV MYSQL_TCP_PORT=3307
#
# expose the port needed to connect to MySQL server:
#
EXPOSE 3307
#
# start server when container runs:
#
CMD ["mysqld"]

```

grab latest mysql image from docker hub

docker-build.bash

```
#!/bin/bash
#
# Linux/Mac BASH script to build docker container
#
docker rmi docker-mysql
docker build -t docker-mysql .

```

docker-run.bash

```
#!/bin/bash
#
# Linux/Mac BASH script to run docker container
#
# NOTE: using port 3307 because the Dockerfile exposes
# that port since I already have MySQL running locally.
#
docker run -d -p 3307:3307 --name mysql --rm docker-mysql

```

-d means disconnect from terminal (run in background)

Docker in AWS

- Docker images are stored in ECR, and then executed via ECS, EKS, Fargate, or Lambda infrastructure

storing docker containers

Amazon ECR
(Elastic Container Registry)

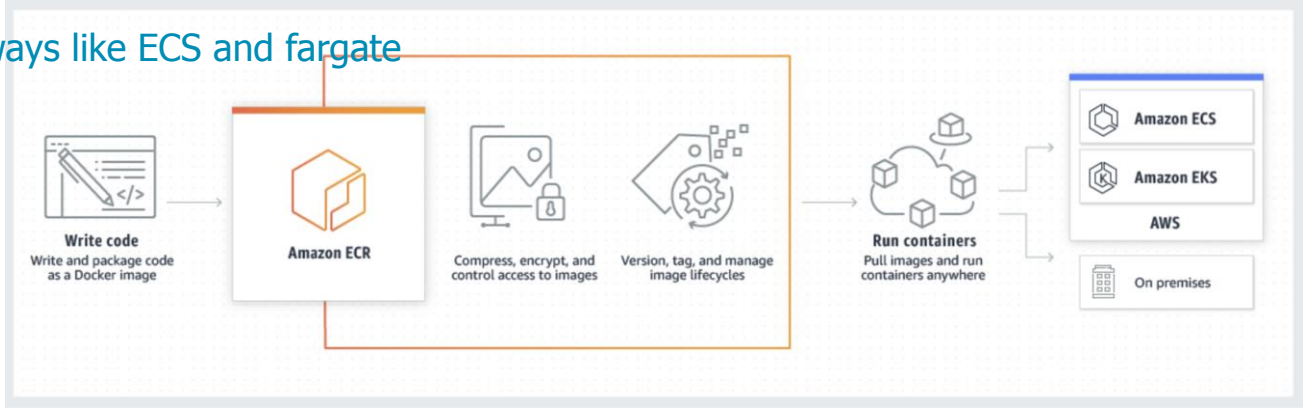
- ECR is **AWS' Docker container registry service** that allows developers to store, manage and deploy Docker container images.
- **Fully managed and integrated with ECS, EKS, Fargate, Lambda, etc.** making it straightforward to run containerized applications on AWS.

managing/controlling containers

Amazon ECS
(Elastic Container Service)

- ECS is a highly scalable, high-performance **container orchestration service**.
- **Used to launch and stop container-based applications** with simple API calls.
- Can be **launched on EC2 instances** (you manage the server) or on **AWS Fargate** (serverless).

once you
1. put image in ECR
then you can
2. run it in many ways like ECS and fargate



Example #3

- Let's run a lambda function using Docker:

lambda_function.py

```python
import json

def lambda_handler(event, context):
  try:
    #number1 = int(event['n1'])
    #number2 = int(event['n2'])

    print("call to add2...")

    params = event["queryStringParameters"]

    number1 = int(params["n1"])
    number2 = int(params["n2"])

    print("adding", number1, '+', number2)

    sum = number1 + number2

    print("sum:", sum)

    return {
      'statusCode': 200,
      'body': json.dumps(sum)
    }

  except Exception as err:
    print("**ERROR**")
    print(str(err))

    return {
      'statusCode': 500,
      'body': json.dumps(str(err))
    }

```

Docker files

Dockerfile

```
1  FROM public.ecr.aws/lambda/python:3.12
2
3  COPY lambda_function.py ${LAMBDA_TASK_ROOT}/
4
5  CMD [ "lambda_function.lambda_handler" ]
```

grab linux + python + lambda stuff in it from aws

copy my fxn into their thing

run my fxn?

docker-build.bash

```
1  #!/bin/bash
2  #
3  # Linux/Mac BASH script to build docker container
4  #
5  docker rmi lambda-add2
6  docker build -t lambda-add2 .
7
```

Push to ECR, load into Lambda, test

- Switch over to AWS:
  1. *Search for ECR, open ECR console*
  2. *Create a repository with same name as container image*
  3. *Select repository, view "Push command"*
  4. *Open terminal window on your laptop (AWS CLI must be configured --- we did this in project 01)*
  5. *Execute each of the "push" commands to push image --- skip the command that builds the image, we did that*
  6. *Switch to Lambda console*
  7. *Create new lambda function*
  8. *Select "Create from Image", and select image from ECR drop-down*
  9. *Name function and create*
  10. *Configuration tab: increase timeout*
  11. *Deploy*
  12. *Test --- logs are available from CloudWatch, not Lambda console*

That's it, thank you!