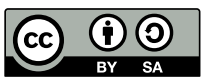
Cloud computing

Production-Ready and Advanced Features

Aleix Llusà Serra

Master's degree in Machine Learning and Cybersecurity for Internet Connected Systems
Universitat Politècnica de Catalunya
`https://epsem.upc.edu`

13 d'octubre de 2026



This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.

1 Best practices

2 Dynamic configuration and security

3 Execution control

4 Monitoring

Best practices

- Microservice orientation: one service per container. Yet the service may fork into multiple processes (e.g. the nginx web server)
- Infrastructure as code: all is build and deployed from files, do not run manual tasks
- Search docker hub (or others) for already developed images. You can base your changes from them, or at least copy the Dockerfile and modify for your needs
- Data: try to design stateless microservices, try to only concentrate critical data on DBMS services
- Think that any container can be shutdown at any time and recreated from zero (except data)

Best practices: images

What is better, an image for each microservice or a generic image adapted by volumes or environmental variables?

- Images are read-only (immutables). Containers should be too.
- If a container needs to be modified: 1) modify the image 2) redeploy
- If persistent data is needed use docker volumes
- Do not confuse data with configuration nor code
- Do not use volumes to adapt an image
- Build an image for each different microservice. Having the same for local/testing than for production, the better control you have.
- Regardless, you can design generic images too. Ease the future particularisation by new Dockerfiles.
- Efficiency? In what? Disk, security, downtime, automated changes, minimum error when deploying, changelog historic keeping...

Environmental variables

Reusable configurations aimed at different environments (local/testing/production). Do not overuse for avoid building an image.

compose.yml

```
services:
  web:
    image: own:latest
    [...]
    environment:
      - DEBUG=false
      - STATUS=production
      - PORT=8000
      - ROOT_EMAIL=me@upc.edu
```

Variable interpolation

https://docs.docker.com/compose/how-tos/environment-variables/variable-interpolation/

compose.yml

```
services:
  web:
    image: own:${TAG:-latest}
    [...]
    environment:
      - DEBUG=${DEBUG-false}
      - STATUS=production
      - PORT=8000
```

secrets

Sensitive environment variables: passwords, certificates, SSH keys

You need swarm for secrets (encrypted storage, encrypted network, per container tmpfs unencrypted)

compose.yml

```
services:
  web:
    [...]
    secrets:
      - my-secrets
    environment:
      PASSWORD_FILE: /var/run/secrets/my-secrets
secrets:
  my-secrets:
    file: ./example.secrets # first (or docker secret create
    #external: true  # latter
    #name: STACKNAME_my-secrets  # latter
```

About entrypoint and CMD

- A Dockerfile should specify at least one of CMD or ENTRYPOINT
- ENTRYPOINT when using the container as an executable
- CMD as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container

Dockerfile

```
FROM python:latest
CMD ["python"]
```

Dockerfile

```
FROM python:latest
COPY entrypoint.sh  /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
CMD ["start"]
```

About entrypoint and CMD

entrypoint.sh

```
#!/bin/bash
set -e
mkdir -p /srv/statics  # Initialise configuration
if [[ "start" == *"$1"* ]]; then
  exec python -m http.server ${PORT}
fi
if [[ "http11" == *"$1"* ]]; then
  exec python -m http.server --protocol HTTP/1.1
fi
# Custom
exec "$@"
```

healthckeck

Check container health by running a command inside the container.

See healthy status with `docker ps`

Dockerfile

```
FROM nginx:latest

RUN [...] apt install curl [...]

HEALTHCHECK --interval=30s --timeout=3s CMD curl \
   -f http://localhost/ || exit 1
```

Also healthcheck in compose.