DevOps, Docker and Observability

Version 1.2.0 (2024-03-10)

CRI

CENTRE DES RESSOURCES INFORMATIQUES

-- Cyril zarak Duval, root CRI/ACU 2020

Introduction

Generic information about the course

Why should you listen to the course ?

- ➔ Being able to code is nice, without the production is useless
- ➔ State of the art notions
- ➔ Having the rights tools makes you more proficient
- ➔ Subject somewhat difficult
- ➔ Getting a decent grade

Who am I ?

- ➔ Why this slide ?
  - ◆ To understand my profile and my point of view
- ➔ EPITA 2020 - CRI/root ACU
  - ◆ ~8 years of real DevOps experience
  - ◆ I know EPITA and students' POV
- ➔ "DevOps engineer" title, in fact more SRE / Ops oriented
  - ◆ Not a dev

Plan

1. DevOps concepts
2. DevOps tools
    a. YAML
    b. Docker
    c. Docker compose
3. Git workflow
4. Observability

What is DevOps ?

Surprisingly, it’s both Dev and Ops

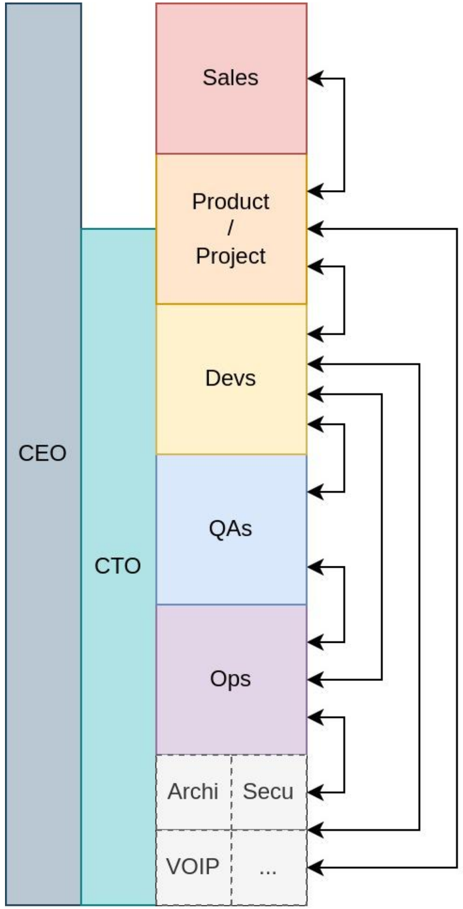
DevOps

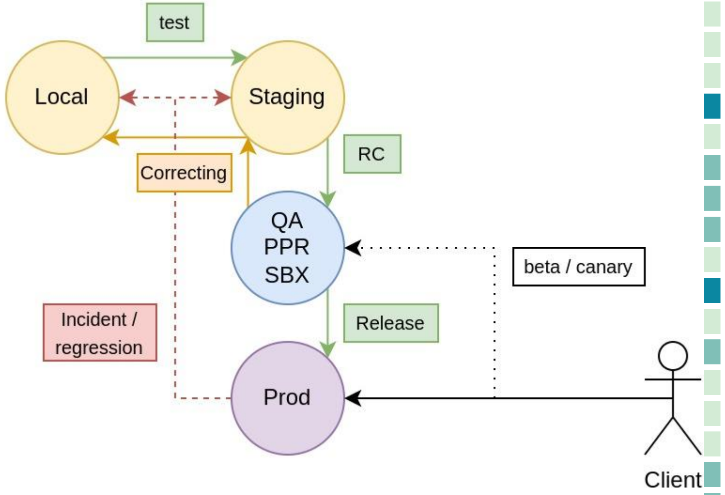
- Contraction of Development and Operations
- Development:
  - Creating code, applications, ...
  - A will to introduce changes
  - Work env: your laptop, your IDE
- Operations:
  - Ensuring services are working and available (including aforementioned applications)
  - A will to not break anything (somewhat reluctant to change)
  - Work env: servers/VMs, a text editor and ssh

DevOps

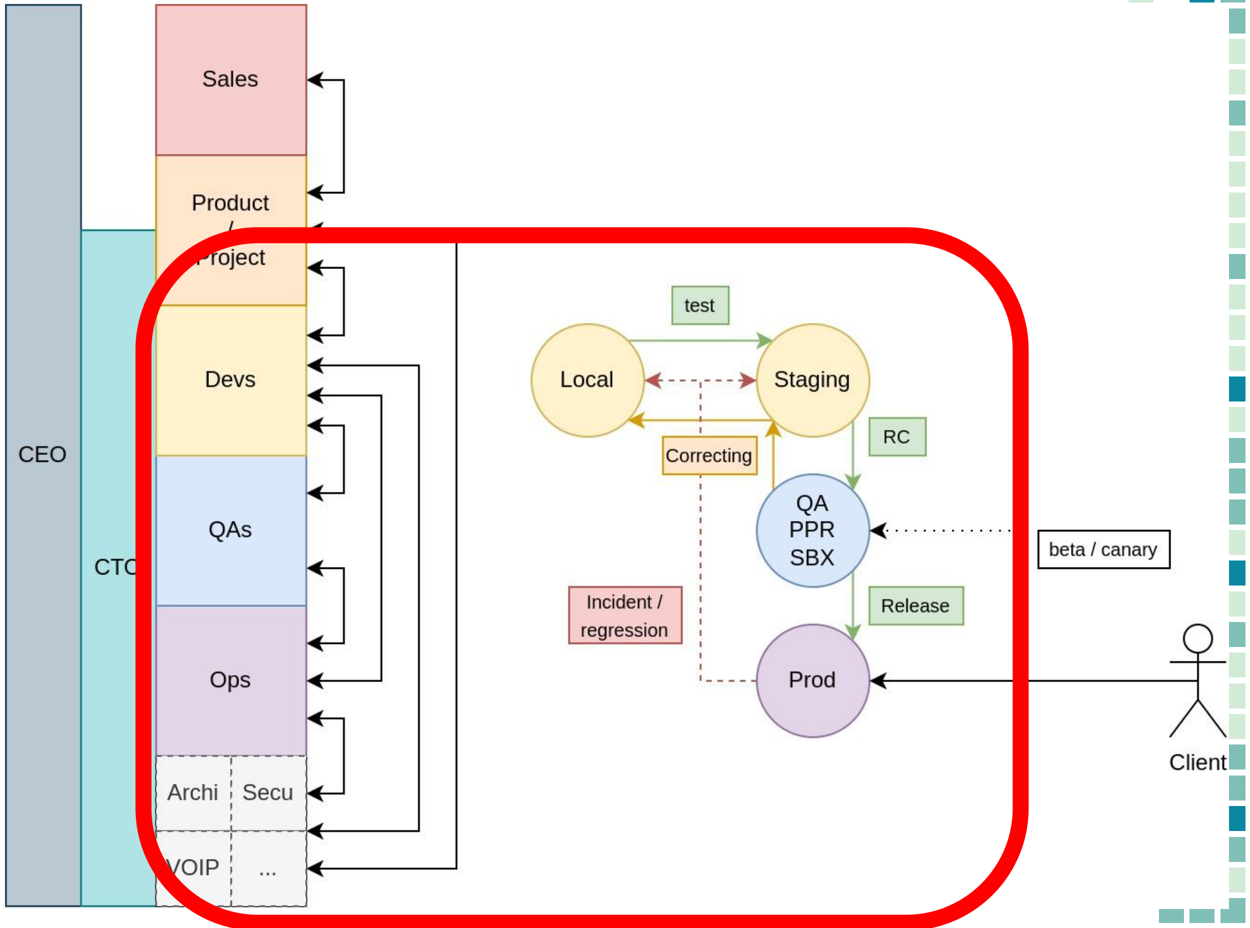
- ➔ DevOps is a set of tools and practices
- ➔ Aim to reduce – even remove – friction between Devs and Ops
- ➔ Devs are doing a bit of Ops
- ➔ Ops are doing a bit of Dev
- ➔ DevOps is not really a job per-se
- ➔ Some DevOps tools are so popular they became new standards

Devs and Ops relation





DevOps scope



What issues are solved by DevOps?

Why is everyone so psyched by DevOps ?

DevOps - Dev POV

- ➔ Devs work on their laptop
  - ◆ Their OS
  - ◆ Their libraries
  - ◆ Their environment
  - ◆ Their hardware specs
  - ◆ Their network stack
  - ◆ ...
- ➔ How to ship that to production servers seamlessly ?
- ➔ What are the meaningful differences ?

DevOps - Dev POV

- ➔ How does a Dev introduce architectural changes ?
- ➔ How to add a new dependency ?
- ➔ How to avoid “but it worked on my laptop :’(“ ?

But also …

- ➔ How to reduce time-to-market ?
- ➔ How to easily understand years of dev without you ?
- ➔ How to work with coworkers ?
  - ◆ That will be different from you

DevOps - Ops POV

- How to deploy Dev's application ?
- How to make sure it's working well ?
- How to know what the dependencies are ?
- How to upgrade things without breaking anything ?
- How to avoid differences between prod and dev env ?
- How to handle config/secrets ?

- How to easily understand years of ops without you ?
- How to work with coworkers ?
  - That will be different from you

DevOps - Solutions

- Devs must have some knowledge on how things are deployed
- Devs should have some control on ops side
  - Limited, scoped, supervised …
  - They aren’t ops
  - Control should be application-oriented
- Ops should be involved in the dev process
  - Not necessarily by coding directly
  - Focus on dependencies

How does DevOps solve those issues?

Surely, it isn’t magic

DevOps - Concepts

- Devops concepts are built around "12 factors"
- Goals of those 12 factors:
  - Be as declarative as possible
  - Understand interactions between app and system
  - No divergence between dev and prod
  - Try to be platform agnostic
  - Be able to scale up/down

DevOps - 12 factors - focus

- ➔ I. Codebase
  - ◆ One codebase tracked in revision control, many deploys
- ➔ II. Dependencies
  - ◆ Explicitly declare and isolate dependencies
- ➔ III. Config
  - ◆ Store config in the environment
- ➔ VII. Port binding
  - ◆ Export services via port binding
- ➔ X. Dev/prod parity
  - ◆ Keep development, staging, and production as similar as possible

DevOps - Practical aspect

- 12 factors is not the magical solution to everything
  - It gives good advices
  - Generic concepts
- DevOps is about mentality and tools
  - Let's look at the tools to understand the mentality

Let’s talk about configuration

Let me introduce you to a new job: YAML engineer

Configuration

- Article 3 of the 12 factors
  - strict separation of config from code
- What does it means ?
- Softwares must not include configuration within the code
- Configuration shall not even be in the code VCS
  - But could and should be stored in another VCS
    - Be careful not to store any secrets in plaintext in a VCS however
- Configuration will be different between dev, staging and prod(s)

What is configuration ?

- ➔ What is configuration ?
  - ◆ Everything removing genericity
  - ◆ Everything that can change from an env to another
- ➔ Examples:
  - ◆ Address of the database
  - ◆ IP/port to bind to
  - ◆ Log-level
  - ◆ Path(s) to store data
  - ◆ ….

How to provide config ?

- How to provide configuration ?
  - env variables
    - Recommended for simple cases, but limited
    - Only Key-Value
  - Config files
    - A bit more complex to setup/provide but advanced
    - Allow lists, mapping, etc
- How to write a config file ?
- Introducing to the most well known format in DevOps: YAML

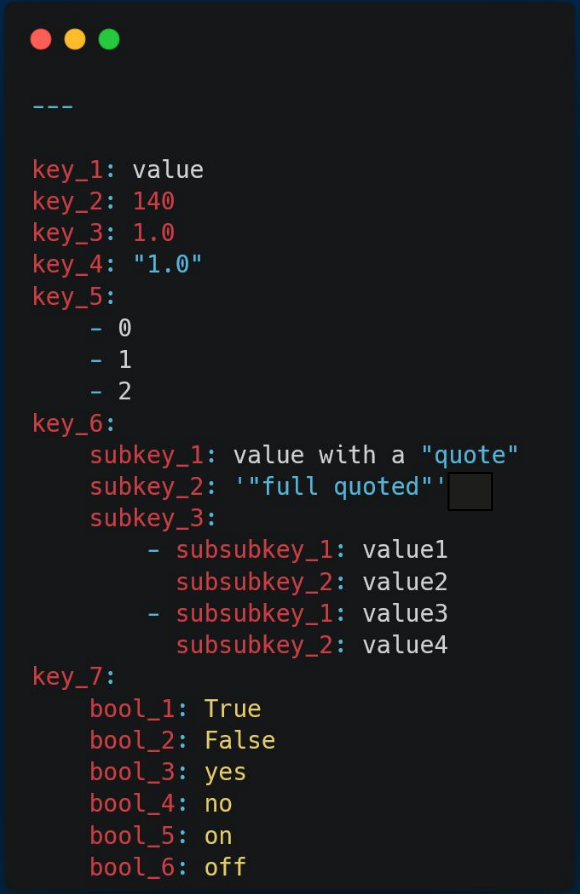
YAML

- ➔ YAML = Yet Another Markup Language
- ➔ Used for Docker-compose, Gitlab-CI, Kubernetes, Helm charts, Ansible, Elasticsearch, Argo, Harbor, …..
- ➔ A superset of JSON to make it more human readable
  - ◆ Meaning that JSON is valid YAML
  - ◆ Even small JSON snippets embedded within a YAML file
- ➔ .yml or .yaml extension
- ➔ Libs in every language to parse it

YAML – the simple way

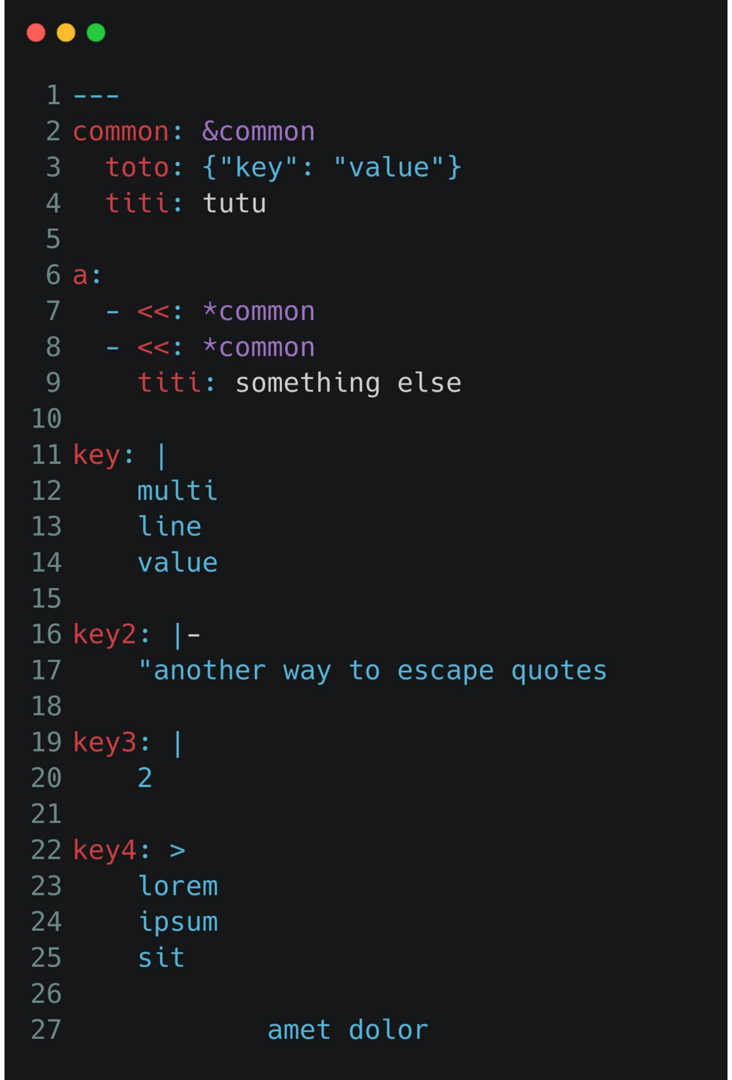
- YAML is a key-value format
- key is a string, value can be :
  - string
  - int
  - boolean
  - list
  - mapping
- Indentation is important
- Quoting can be used

YAML – the simple way

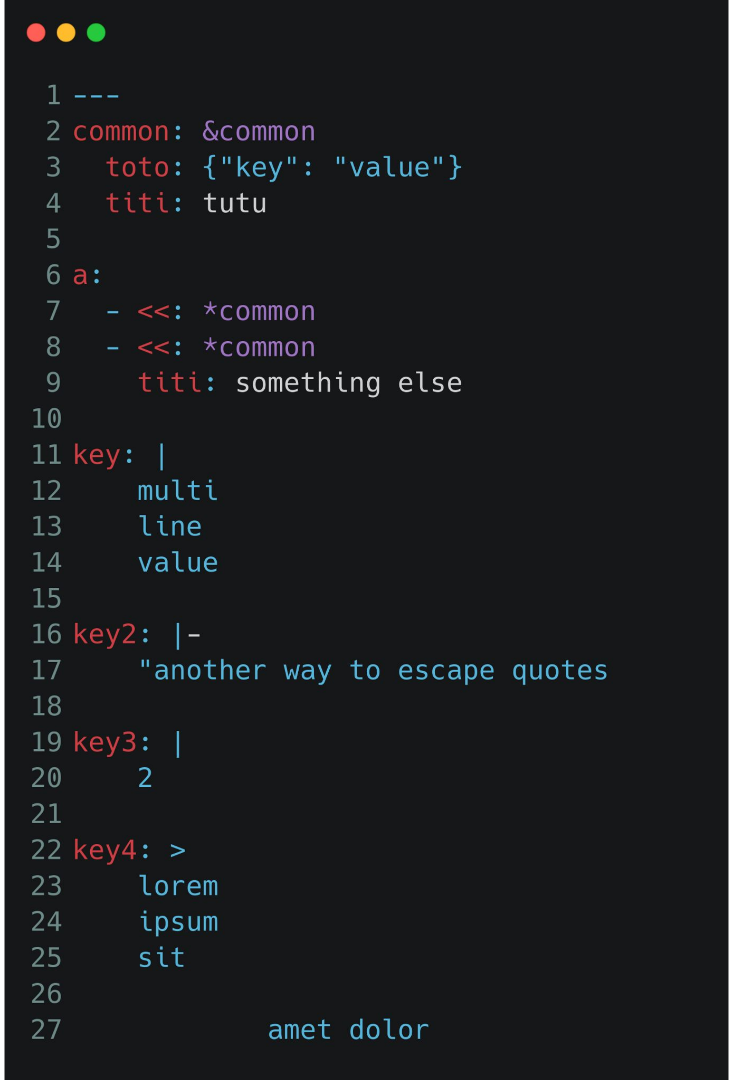


```yaml
---

key_1: value
key_2: 140
key_3: 1.0
key_4: "1.0"
key_5:
    - 0
    - 1
    - 2
key_6:
    subkey_1: value with a "quote"
    subkey_2: '"full quoted"'
    subkey_3:
        - subsubkey_1: value1
          subsubkey_2: value2
        - subsubkey_1: value3
          subsubkey_2: value4
key_7:
    bool_1: True
    bool_2: False
    bool_3: yes
    bool_4: no
    bool_5: on
    bool_6: off
```

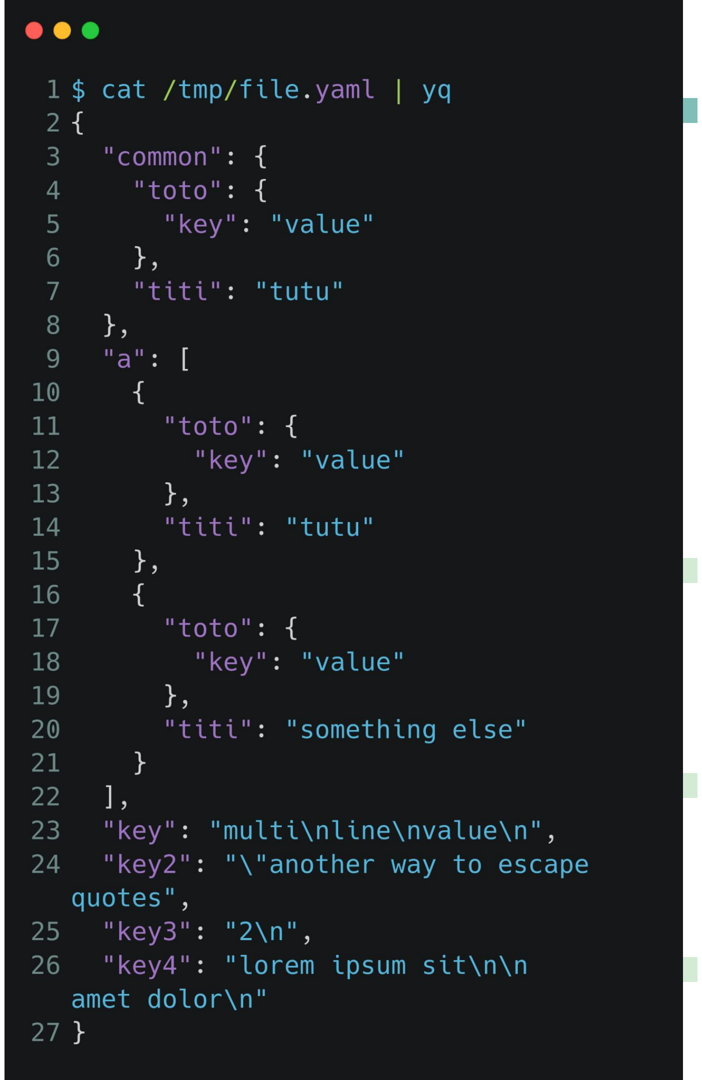
YAML – Advanced features



```yaml
---
common: &common
  toto: {"key": "value"}
  titi: tutu

a:
  - <<: *common
  - <<: *common
    titi: something else

key: |
    multi
    line
    value

key2: |-
    "another way to escape quotes

key3: |
    2

key4: >
    lorem
    ipsum
    sit

            amet dolor
```



```yaml
---
common: &common
  toto: {"key": "value"}
  titi: tutu

a:
  - <<: *common
  - <<: *common
    titi: something else

key: |
    multi
    line
    value

key2: |-
    "another way to escape quotes

key3: |
    2

key4: >
    lorem
    ipsum
    sit

            amet dolor
```



```
$ cat /tmp/file.yaml | yq
{
  "common": {
    "toto": {
      "key": "value"
    },
    "titi": "tutu"
  },
  "a": [
    {
      "toto": {
        "key": "value"
      },
      "titi": "tutu"
    },
    {
      "toto": {
        "key": "value"
      },
      "titi": "something else"
    }
  ],
  "key": "multi\nline\nvalue\n",
  "key2": "\"another way to escape quotes",
  "key3": "2\n",
  "key4": "lorem ipsum sit\n\n amet dolor\n"
}
```

YAML - tools

- ➔ yq is a useful tool to parse yaml in bash
- ➔ yamllint detects linting errors, inconsistencies and warn you about possible misuage

YAML - pitfalls

- YAML by its nature can mislead users
- Here are some points to be careful on:
  - JSON being valid YAML syntax
  - Strings can be unquotted
    - Be careful with numbers
  - The Norway problem
    - Case insensitive booleans in [Yes, True, On, Y]
  - Indentation
  - Mixup of “complex” types (mapping of lists of mappings, …)

YAML - some help

➔ https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html

Let’s dive in Docker

Docker with an example

Let’s have a look first at the top level

Quick docker overlook

- ➔ I want to run a webserver quickly
- ➔ I don't really know in details any
- ➔ I don't want to mess with the things installed on my computer
  - ◆ Libraries, general packages …
- ➔ I just need it for some time and then forget about it
- ➔ Maybe I'll need it again in some months

With docker

What are Docker and containers ?

Let’s try to understand with more or less details

What is Docker ?

- ➔ Docker is a container engine
- ➔ It allows you to:
  - ◆ Create images
  - ◆ Start containers from those images
  - ◆ Manage containers
  - ◆ Exchange images

What are containers and images ?

- ➔ A container is “kinda” like a virtual machine
- ➔ A container is not a universal definition
  - ◆ We’re talking about linux containers in this course
- ➔ A container is essentially a process (and its sub-processes if any) which is isolated
- ➔ A container is ephemeral by design
- ➔ An image (in docker/OCI) is the source of a container
  - ◆ From an image you can create multiple containers
  - ◆ Each container is created from an image
  - ◆ See the relation like class/object in OOP

Container vs VM

- ➔ VM uses CPU mechanisms (+ bits of hypervisor)
- ➔ VM needs its own kernel
- ➔ VM can be of different architecture (x86, ARM, RISC-V, …)
  - ◆ Virtualization, paravirtualization, emulation
- ➔ Host (hypervisor) doesn’t have much access in the VM
  - ◆ i.e. can’t see natively its process, load, etc
- ➔ Container is simply a linux process isolated with kernel mechanisms
- ➔ Host has full access on the container

Container vs VM

- ➔ VM needs to be setup with RAM amount, CPU count, disk, etc
- ➔ Container is a process. You can limit resources but not mandatory
- ➔ Containers are lighter:
  - ◆ No kernel
  - ◆ Faster to start
  - ◆ Can even run without an OS
- ➔ Containers are less secure
- ➔ Containers can’t run everything (i.e. no windows on linux)
- ➔ Containers are ephemeral by nature

Container vs regular process

- What makes a process a container ?
- Isolation of:
  - Filesystem
  - Other running processes
  - Users
  - And also: network, mountpoints, UTS (hostname), ...
- Can also have limitations (CPU, memory, etc)
- No clear way of identification
  - No “container id” or anything provided by the kernel

Why do even need containers ?

Don’ t only take my word, but there are useful

Why do we use containers ?

- ➔ Control your OS (it’s in the image)
  - ◆ No dependency issue from a laptop to a server: everything is in the image
  - ◆ Can have multiple libs in parallel (in different images)
- ➔ Common interface to build and run applications
- ➔ Share easily the images
  - ◆ The app and all its dependencies
- ➔ Version control
- ➔ Isolation

Why do we use containers ?

- ➔ Cheap
  - ◆ Quick to build
  - ◆ Quick to start
  - ◆ No overhead (unlike VMs)
- ➔ No difference between your laptop, dev server and prod server
- ➔ Follow the 12 factors principles

How does Docker work ?

Let’s have a look at the daemon and the CLI

How does Docker work ?

- ➔ Docker works with a daemon: dockerd
- ➔ dockerd manages everything
- ➔ The user can contact dockerd in multiple ways
  - ◆ UNIX socket, TCP, …
- ➔ Most people use the Docker CLI client, the command docker
- ➔ The docker command will contact the docker daemon to execute the user inputed command

How does Docker work ?

- ➔ The Docker daemon manages running, stopped containers, but also images, volumes, etc…
- ➔ If the daemon is not running, or you don't have the permissions to contact it, you might get some error like
- ➔ Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/json": dial unix /var/run/docker.sock: connect: permission denied

Let’s get started with docker

Creating containers

Docker CLI - run containers

- ➔ To run a container with docker, we use docker run
- ➔ To check for running containers, we use docker ps
- ➔ Let’s check docker common operations with containers:
    - ◆ [pull, start, stop, ps, image ls, exec]()

How to build docker images ?

Stop using and start creating

What is a docker image ?

- ➔ We said that docker containers are created from docker image
- ➔ Like an instance from a class, an object and a template
- ➔ What defines an image then ?
- ➔ What's inside an image ?

What is a docker image ?

- ➔ A docker image is essentially a combination of a few things:
  - ◆ A filesystem
    - The “main” binary of the image
    - Its libraries, essential files, …
    - Some other binaries
      - Dependencies
      - Utilities
    - All sets of files deemed worthy of being shipped in the image

What is a docker image ?

- ➔ A docker image is essentially a combination of a few things:
  - ◆ A filesystem
  - ◆ Some metadatas :
    - How it was built
    - What commands to run by default
    - Some environment variable to set
    - ...
- ➔ Docker images are actually more generic and are OCI Images

Dockerfile

- ➔ Docker images are built with a Dockerfile*
- ➔ It's a recipe-like config file
- ➔ It has multiple kind of instructions
  - ◆ FROM selects the docker image to start from
  - ◆ RUN let you run arbitrary shell commands
  - ◆ …
    - More details to come with the practicum

* there are other ways not to be mentioned in this course

Dockerfile

- A Dockerfile starts by a source docker image
  - FROM instruction
- Let’s say you want to create an image based on debian
  - FROM debian:11
- Everything from the image stated in FROM will be imported
- The rest of the commands will create another image on top of the initial FROM

Dockerfile

- ➔ Each instruction will perform some modifications on the image
  - ◆ Add a file
  - ◆ Run a command
  - ◆ Set some variable
  - ◆ ….
- ➔ Once they are all successfully executed, a new image is built

Image, tags, repository

- A docker image is defined by a hash (sha256)
- But it’s not convenient for most people
- So a name can be set on a hash for references purposes
- But because a name could have multiple version, we can append a tag
  - debian:11, python:3, nginx:alpine, …

Image, tags, repository

- ➔ To be shared, images need to have a name that includes a registry
- ➔ Default registry: docker.io
- ➔ Default directory: library
- ➔ When referencing an image debian:12, in fact its *real* name is docker.io/library/debian:12

Docker image and build workflow

Really make the difference between building and running

Workflow

1. Find a starting appropriate image for your project
2. Find a correct tag for the image
3. Write a Dockerfile that starts from said image
4. `docker build` to create the image
5. Check information about the image
6. Create one or multiple container(s) from said image with `docker run`

How to find a good image ?

- What does define a good image for a project ?
- It depends on what you need
- 95% of projects has a main dependency
  - For example if it's a python application, its dependency is … python
  - If it's a nodejs application, it depends on node
  - Maybe you simply need a generic OS, like Debian
- Find a clean image following your dependency
  - Official, supported, well-used, clean

How to find a good image ?

- The base image will be used in its entirety to build your new image
- Don’t pick an image unnecessarily too big
- Most images are an OS
  - Debian, Ubuntu, Alpine
- Alpine remains the smaller OS there is
  - If possible and applicable, tend to use Alpine
- But how to choose the OS of an image ?

How to find a good image ?

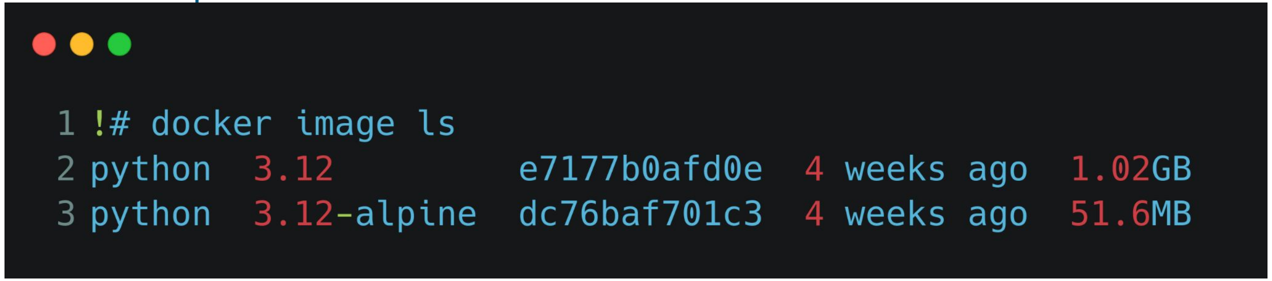
- Let’s say we are building a python application
- The best image seems to be “python”
- It’s official, well maintained, up to date and well used
- But which OS do I end-up with ?
- Let’s check it ourself !
- docker run --rm --entrypoint cat python /etc/os-release
- But how do I change ?

How to find a good image ?

- Part of selecting a good image is also about selecting the proper tag
- The python image comes with hundreds of tags
- Tags are way to change the version of python needed
- python:3.9, python:3.11, ...
- But they're also commonly used to change the flavor of the OS underneath
- python:alpine, python:3.11-alpine

How to find a good image ?

- It is a good practice to select an image with an explicit tag
- Keeping latest is often problematic and lead to issues
- Expliciting the tag makes the build reproductible
- Choosing the right tag is also important
- If you only need python and don't care about the underlying OS for example:



```
!# docker image ls
python  3.12          e7177b0afd0e  4 weeks ago  1.02GB
python  3.12-alpine   dc76baf701c3  4 weeks ago  51.6MB
```

Having a look at containers mechanisms

What does it look like ?

Container isolation

- Isolation is done via 2 syscalls:
  - chroot(2)
  - namespaces(7)
- chroot:
  - Change the root directory for a process
  - Prevent the process from accessing anything not in its root
  - Example

Container isolation - chroot

- Changing a process root directory means preventing it from accessing host libraries
  - /usr/lib for example might be needed and then provided
- A good way to control installed libraries and their version
- Needs to provide an “OS” in the chrooted directory
  - Needed binaries, libs, FHS, …
  - Tends to make a container VM-ish

Container isolation - namespaces

- Other syscall namespaces(7)
- Create a namespace of a kind for a process (and its children)
- Kind of namespace:
  - Network
  - Mount
  - PID
  - User
  - ...
- Hierarchical approach

Container isolation – namespaces

- ➔ Example of network namespace
- ➔ Example of PID (and user) namespace

Container limitation

- ➔ A container shall be limitable
- ➔ Like VM : allow max resources
  - ◆ Avoid CPU burst, OOM, ...
- ➔ Linux mechanism: cgroups

Cgroups (v2)

- Linux mechanism to add process in a control group
- Control groups allow to set limits on various resources
  - Limits are hierarchical, a sub cgroup cannot exceed its parent limits
- 2 versions of cgroups:
  - v2 used on modern systems
  - v1 still widely used
- Exposed as a pseudo filesystem
  - Check mount(1) output

Cgroups (v2)

- ➔ [Cgroups example with cpuset cgroup]

Container isolation – how to share ?

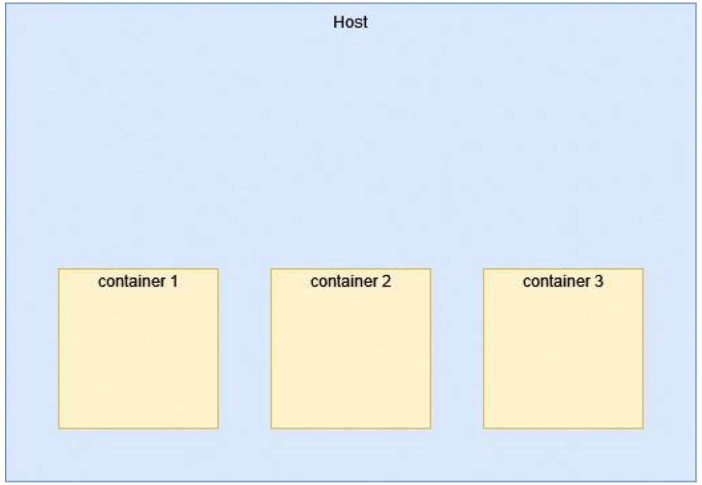
- What if you need to share a directory ?
  - Ephemeral containers aren't suitable for persistent data
- What if your container must be network accessible ?
- Docker offers way to share resources
  - Let's have a look at its CLI

Understand implications of such isolation through network

Maybe a schema will help ?

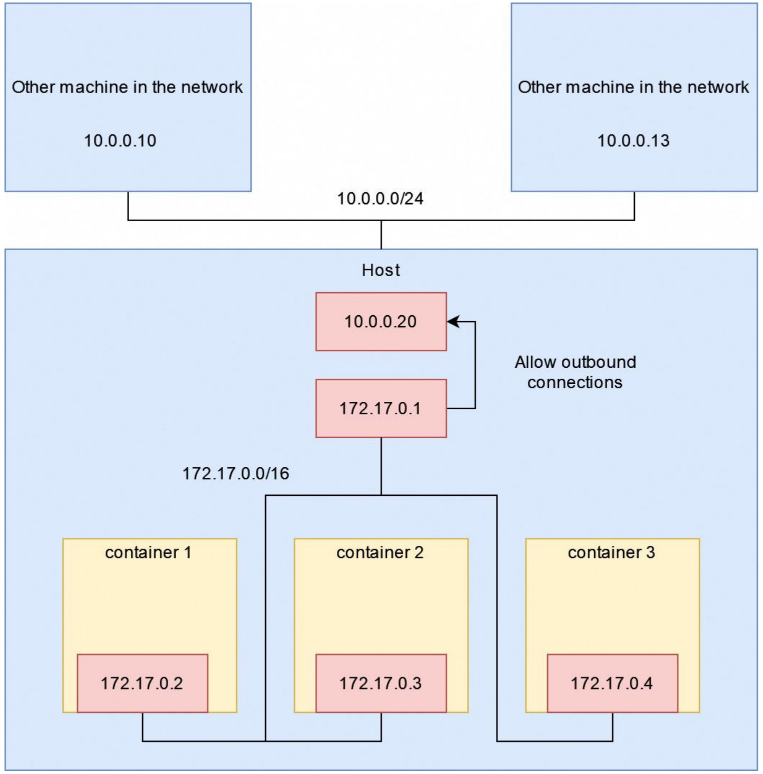
Container isolation boxes

- Most important part of container isolation to understand is the box model
- The host is the bigger box, and contains the rest
- Asymmetrical relation



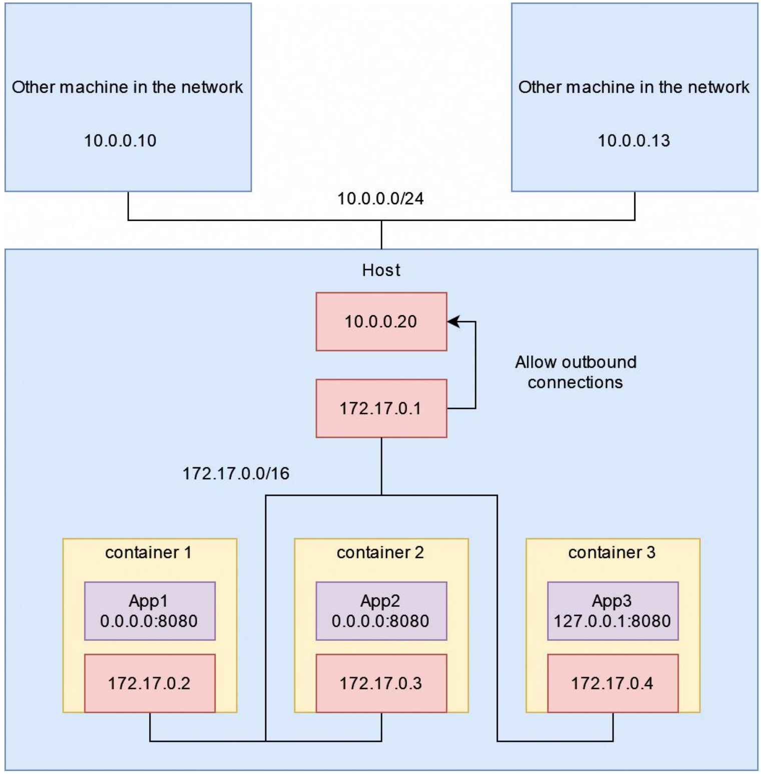
Container isolation boxes – network

- For network aspect, docker creates a subnet by default
- Each container is put in this default subnet
- Allow to access internet
  - But by default not accessible from outside
- Can communicate with each others



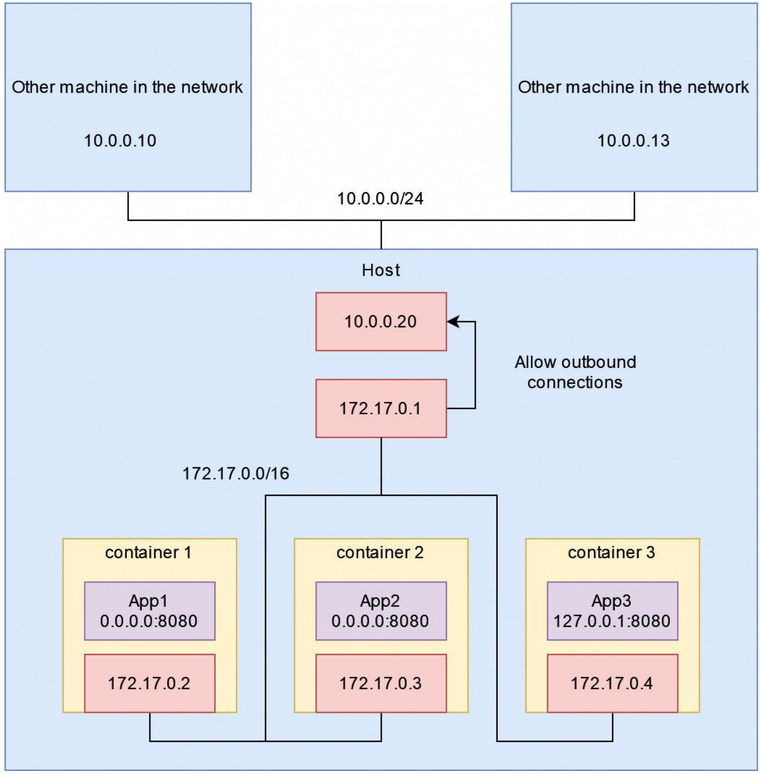
Container isolation boxes – network

- Let's have some network services listening and awaiting connections: App1, 2 & 3
- Listening on IP:port
  - 0.0.0.0:8080
  - 0.0.0.0:8080
  - 127.0.0.1:8080
- Listening on IP 0.0.0.0 means listening on all IP addresses



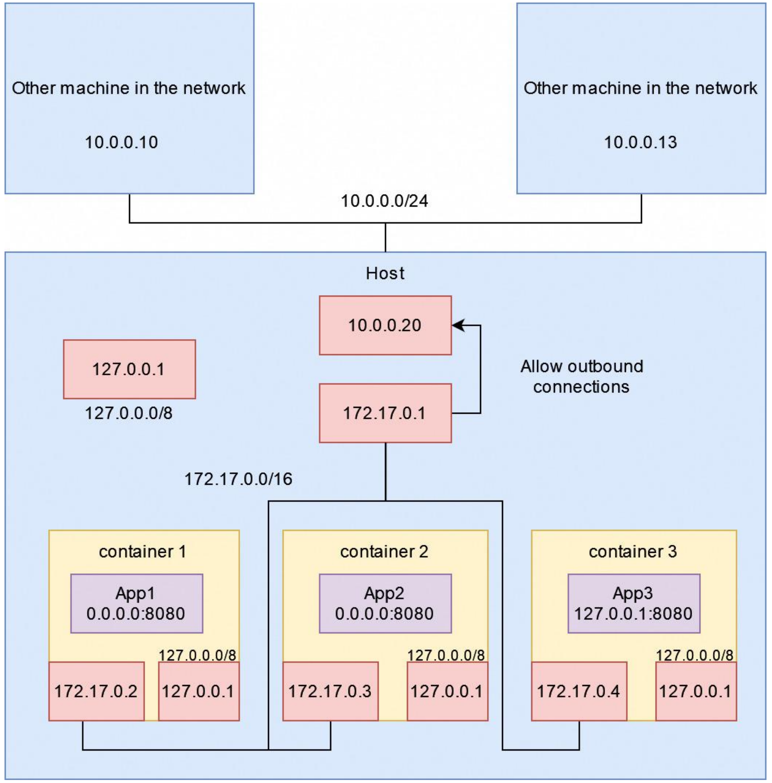
Container isolation boxes – network

- App1 and App2 don't step on each others toes
  - Different containers
  - Different IP addresses
  - They can both listen on port 8080
- Can 10.0.0.10 reach 172.17.0.2:8080 ?
- Can the host reach App3 ?
- Can container1 reach App3 ?



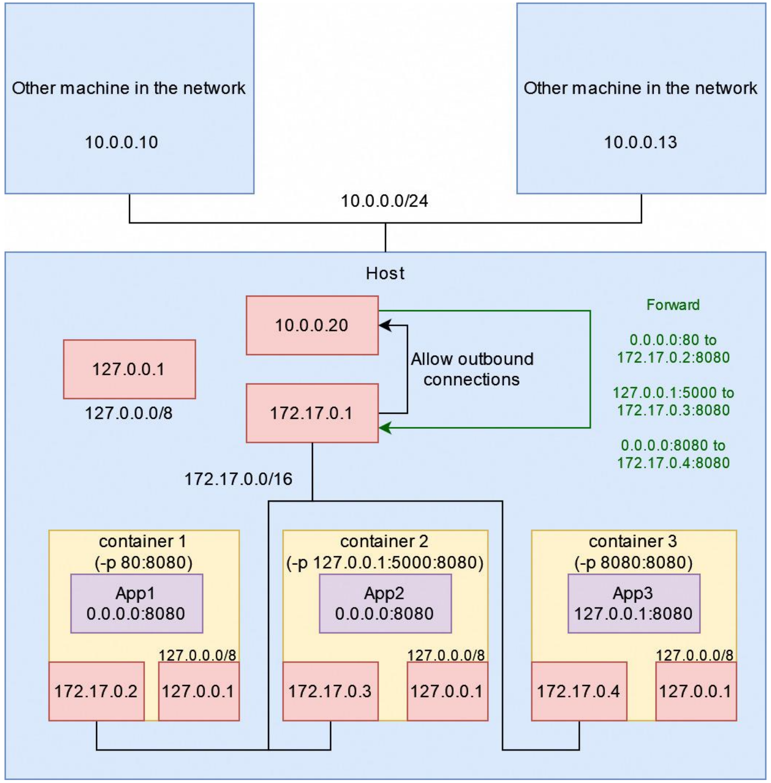
Container isolation boxes – network

- ➔ “En fait l’histoire est plus complexe”
- ➔ Each container and the host have their own localhost (127.0.0.0/8) subnet
- ➔ To expose App1 (or App2) publicly, a link must be made between 10.0.0.20 and 172.17.0.2 (172.17.0.3)



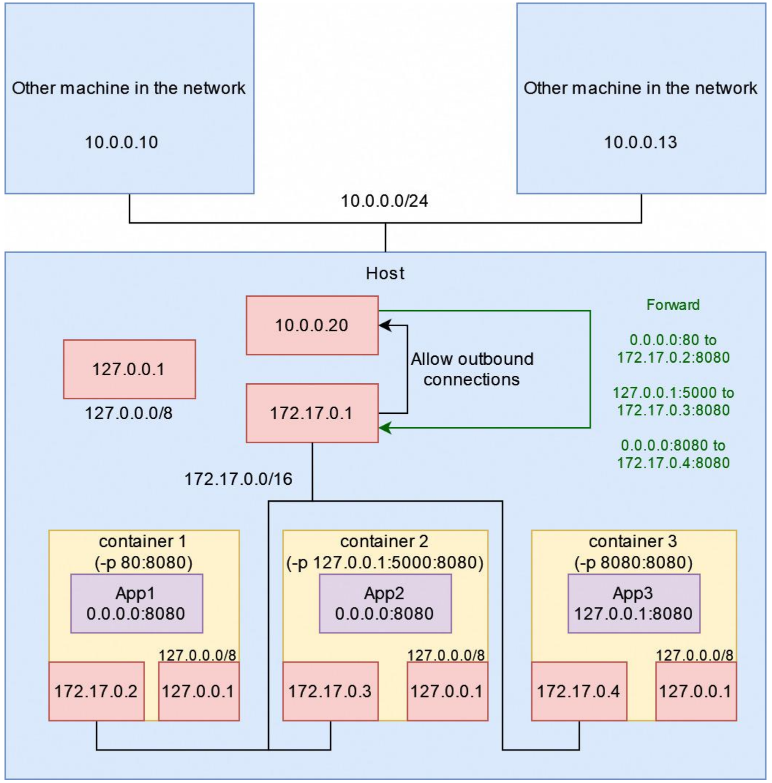
Container isolation boxes – network

- ➔ Running the containers with -p to expose ports (docker run -p) allow external connections and mapping
- ➔ App1 is reachable from 10.0.0.10 on 10.0.0.20:80
- ➔ App1 is reachable from Host on 127.0.0.1:80, 10.0.0.20:80 or 172.17.0.2:8080



Container isolation boxes – network

- ➔ App2 is not reachable from 10.0.0.10
- ➔ App2 is reachable from Host on 127.0.0.1:5000 or 172.17.0.3:8080
- ➔ App3 is reachable only* from container 3 on 127.0.0.1:8080



* actually can be reached from the Host by tricking quite a bit, but not covered by the course

A word about overlayfs

Understand this to build better images

Overlayfs

- Docker uses overlayfs to assemble images
  - Also to run containers on top of an image
- Overlayfs is interesting and a bit complex
  - Won’t go into details here
  - Basically uses layers
    - A layer contains all the files changed at a step
    - An image is built with multiple steps = multiple layers
    - A container adds a final layer on an image: the runtime diffs

Overlayfs

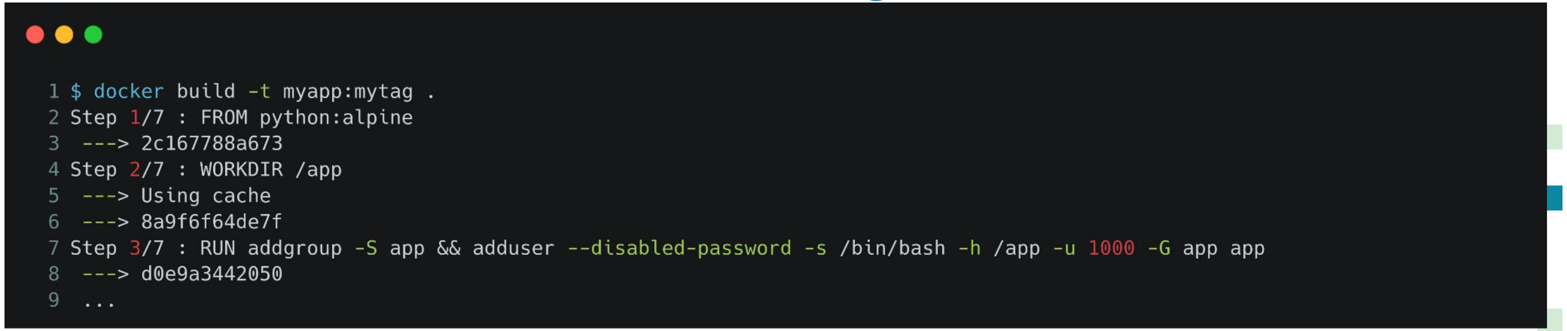
- Overlayfs layers
  - A layer contains all the files changed at a step
  - An image is built with multiple steps = multiple layers
- Many steps = Many layers
  - It is preferable to reduce as much as possible
  - Example:
    - RUN apt-get install -y vim
      RUN echo "syntax on" > ~/.vimrc
      ->
      RUN apt-get install -y vim && echo "syntax on" > ~/.vimrc

Overlayfs

- ➔ A layer that add a 1GiB file and layer that removes it after = 1GiB still
- ➔ A layer that both adds & removes = ~no space taken
  - ◆ Important to `apt-get install` and remove cache in the same layer
- ➔ 2 Images with common instructions creates the same layers
  - ◆ Until they diverge
  - ◆ Important to put the common instructions first
  - ◆ Then packages installation (heavy)
  - ◆ Then image-specific things

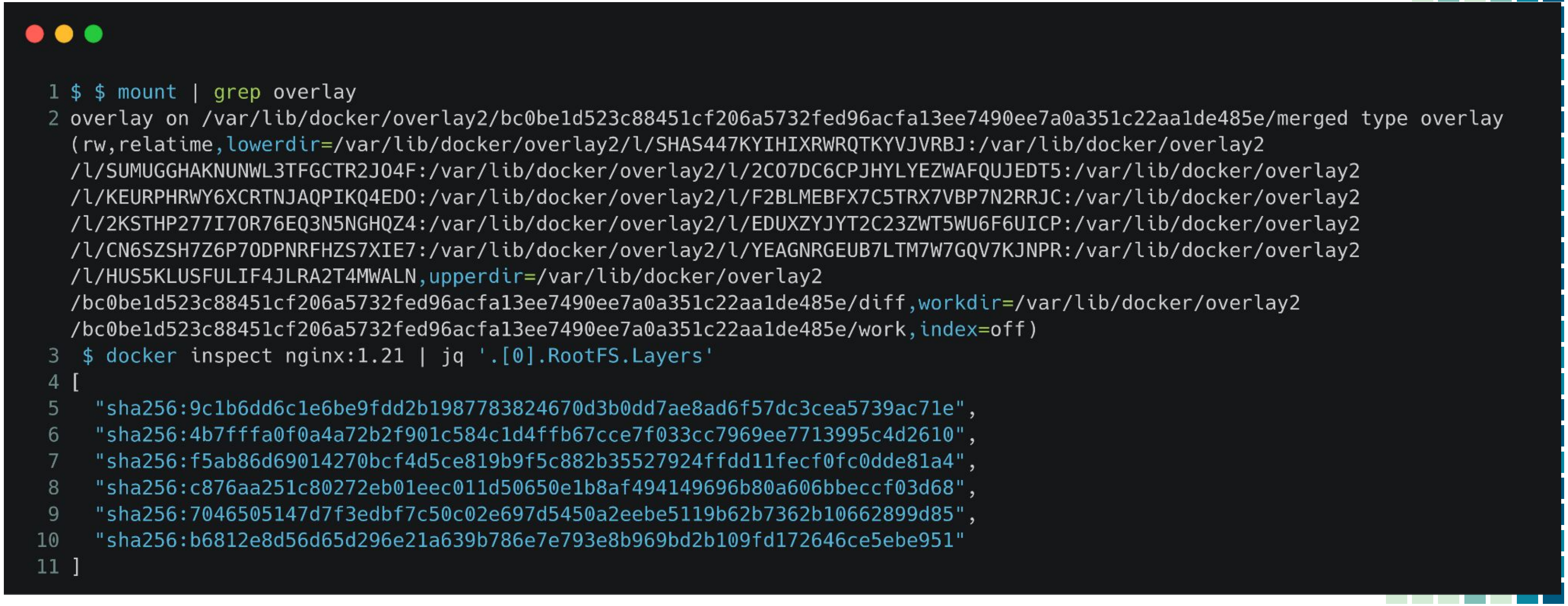
Overlayfs

- You can see layers when building images
  - They are designated by a hash
- When pulling
- With docker inspect
- With mount if a container is running



```
$ docker build -t myapp:mytag .
Step 1/7 : FROM python:alpine
 ---> 2c167788a673
Step 2/7 : WORKDIR /app
 ---> Using cache
 ---> 8a9f6f64de7f
Step 3/7 : RUN addgroup -S app && adduser --disabled-password -s /bin/bash -h /app -u 1000 -G app app
 ---> d0e9a3442050
 ...
```

Overlayfs



```
$ $ mount | grep overlay
overlay on /var/lib/docker/overlay2/bc0be1d523c88451cf206a5732fed96acfa13ee7490ee7a0a351c22aa1de485e/merged type overlay
(rw,relatime,lowerdir=/var/lib/docker/overlay2/l/SHAS447KYIHIXRWRQTKYVJVRBJ:/var/lib/docker/overlay2
/l/SUMUGGHAKNUNWL3TFGCTR2JO4F:/var/lib/docker/overlay2/l/2CO7DC6CPJHYLYEZWAFQUJEDT5:/var/lib/docker/overlay2
/l/KEURPHRWY6XCRTNJAQPIKQ4EDO:/var/lib/docker/overlay2/l/F2BLMEBFX7C5TRX7VBP7N2RRJC:/var/lib/docker/overlay2
/l/2KSTHP277I7OR76EQ3N5NGHQZ4:/var/lib/docker/overlay2/l/EDUXZYJYT2C23ZWT5WU6F6UICP:/var/lib/docker/overlay2
/l/CN6SZSH7Z6P7ODPNRFHZS7XIE7:/var/lib/docker/overlay2/l/YEAGNRGEUB7LTM7W7GQV7KJNPR:/var/lib/docker/overlay2
/l/HUS5KLUSFULIF4JLRA2T4MWALN,upperdir=/var/lib/docker/overlay2
/bc0be1d523c88451cf206a5732fed96acfa13ee7490ee7a0a351c22aa1de485e/diff,workdir=/var/lib/docker/overlay2
/bc0be1d523c88451cf206a5732fed96acfa13ee7490ee7a0a351c22aa1de485e/work,index=off)
 $ docker inspect nginx:1.21 | jq '.[0].RootFS.Layers'
[
  "sha256:9c1b6dd6c1e6be9fdd2b1987783824670d3b0dd7ae8ad6f57dc3cea5739ac71e",
  "sha256:4b7fffa0f0a4a72b2f901c584c1d4ffb67cce7f033cc7969ee7713995c4d2610",
  "sha256:f5ab86d69014270bcf4d5ce819b9f5c882b35527924ffdd11fecf0fc0dde81a4",
  "sha256:c876aa251c80272eb01eec011d50650e1b8af494149696b80a606bbeccf03d68",
  "sha256:7046505147d7f3edbf7c50c02e697d5450a2eebe5119b62b7362b10662899d85",
  "sha256:b6812e8d56d65d296e21a639b786e7e793e8b969bd2b109fd172646ce5ebe951"
]
```

docker compose

Docker cli limitations

- Docker cli is a bit limited for some cases
  - Lots of arguments
  - Needs to remember the arguments to restart/change/move the container
  - Can be considered config, but isn't in a config file
    - Against 12 factors

Docker compose

- docker compose translate a config file into docker commands
- Used to declare statically containers, networks, volumes, ...
- YAML format
- docker-compose.yml

Docker compose

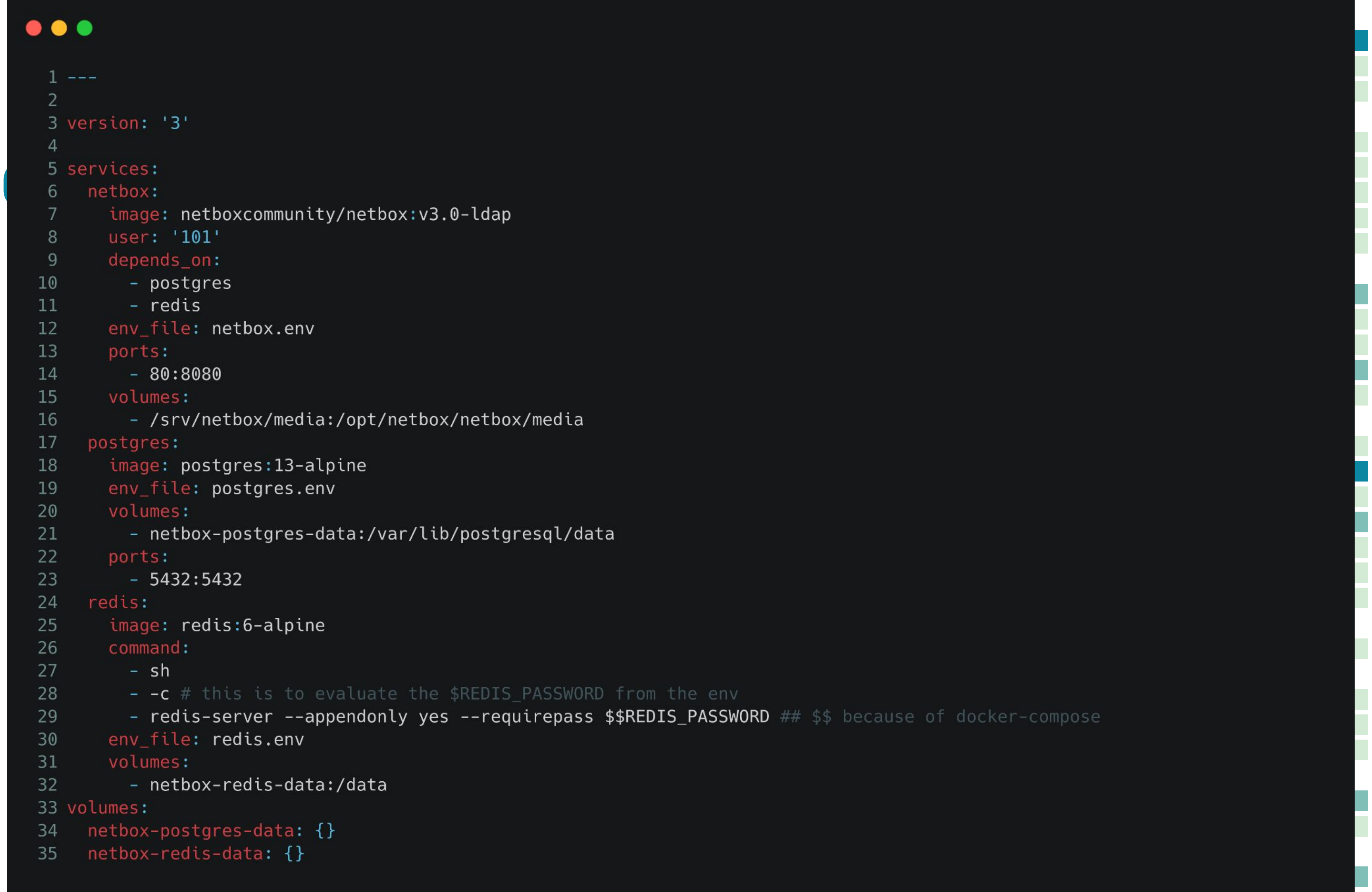
- ➔ docker compose up to create and run the containers
- ➔ docker compose down to stop and delete the containers
- ➔ docker compose start/stop to start/stop the containers
- ➔ docker compose logs to look at containers logs
- ➔ That's most of its CLI usage
- ➔ docker compose is not a daemon, just a "translator" that reads YML to convert it to docker commands

Docker compose

- docker compose up to create and run the containers
- docker compose down to stop and delete the containers
- docker compose start/stop to start/stop the containers
- docker compose logs to look at containers logs
- That’s most of its CLI usage
- docker compose is not a daemon, just a “translator” that reads YML to convert it to docker commands

About docker-compose.yml file

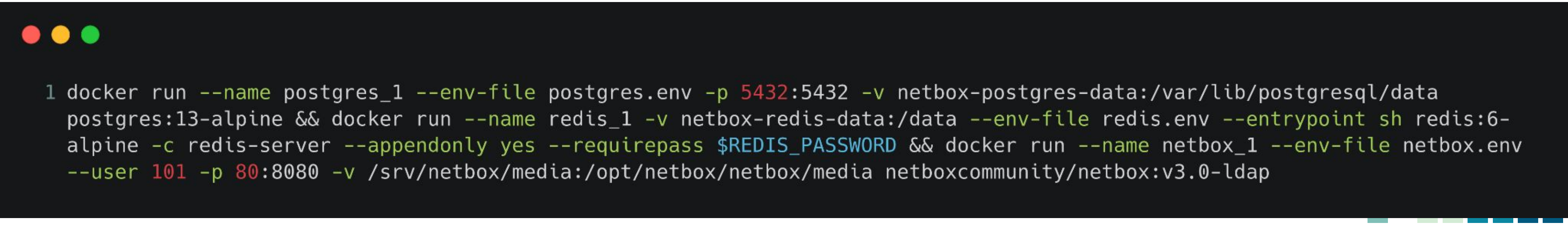
How to write this config file ?



```yaml
---

version: '3'

services:
  netbox:
    image: netboxcommunity/netbox:v3.0-ldap
    user: '101'
    depends_on:
      - postgres
      - redis
    env_file: netbox.env
    ports:
      - 80:8080
    volumes:
      - /srv/netbox/media:/opt/netbox/netbox/media
  postgres:
    image: postgres:13-alpine
    env_file: postgres.env
    volumes:
      - netbox-postgres-data:/var/lib/postgresql/data
    ports:
      - 5432:5432
  redis:
    image: redis:6-alpine
    command:
      - sh
      - -c # this is to evaluate the $REDIS_PASSWORD from the env
      - redis-server --appendonly yes --requirepass $$REDIS_PASSWORD ## $$ because of docker-compose
    env_file: redis.env
    volumes:
      - netbox-redis-data:/data
volumes:
  netbox-postgres-data: {}
  netbox-redis-data: {}
```

docker-compose.yml

➔ ~Equivalent to:



```
docker run --name postgres_1 --env-file postgres.env -p 5432:5432 -v netbox-postgres-data:/var/lib/postgresql/data
postgres:13-alpine && docker run --name redis_1 -v netbox-redis-data:/data --env-file redis.env --entrypoint sh redis:6-
alpine -c redis-server --appendonly yes --requirepass $REDIS_PASSWORD && docker run --name netbox_1 --env-file netbox.env
--user 101 -p 80:8080 -v /srv/netbox/media:/opt/netbox/netbox/media netboxcommunity/netbox:v3.0-ldap
```

Docker compose vs docker-compose

Docker Inc. likes to make things complex - they were founded by frenchs after all

Docker compose

- ➔ You might encounter 2 versions of docker compose, with 2 syntaxes:
  - ◆ docker-compose and docker compose
- ➔ docker-compose is the v1 of the docker/compose project
  - ◆ Old, was written in python, works well but not maintained anymore
- ➔ docker compose is the v2 of the docker/compose project
  - ◆ New, to be preferred. Don't use docker-compose anymore if you can

Git workflow and CI/CD

Git workflow and CI/CD

- ➔ You need to use a VCS to work efficiently with many people
  - ◆ Git is obviously the most popular VCS
- ➔ The way you work with git is called a git workflow
- ➔ A git workflow is a set of rules and best practices for a project or a team
  - ◆ Ex: don’t push on master branch directly

Git workflow and CI/CD

- Usual git workflow looks like this:
  - master/main/devel branch represents the project “stable” version
    - It’s the most important branch
    - One cannot push directly to it (protected)
    - The ability to merge is limited to maintainers
  - To add a new feature, one must create a branch

Git workflow and CI/CD

➔ Git workflow can also:
- Setup a git message format
- Allow/force/forbid to squash commits in a branch
- Choose between merge commits, fast forward or not, or rebases
- Enforce a branch naming convention
- Allow or not push force on feature branches
- ...

Git workflow and CI/CD

- ➔ Usually you want to assert code quality before merging it
- ➔ People review Merge/Pull Requests before merging them
  - ◆ Again, it depends on your git workflow
- ➔ People often make mistakes
- ➔ Continuous integrations (CI) can help but running your
  - ◆ e2e tests
  - ◆ Unit tests
  - ◆ Linter
  - ◆ ...

Git workflow and CI/CD

- ➔ CI can also try to compile your project to see if there are errors or warnings
- ➔ Make available build on release
- ➔ Push the new version somewhere
  - ◆ In this case it's called a CD: continuous deployment
- ➔ CI/CD are more or less the same: code to be executed with a git workflow
  - ◆ CI are tests to ensure quality
  - ◆ CD deploys automatically

Git workflow and CI/CD

- ➔ When to run your CI/CD depends on your git workflow
- ➔ Examples include:
  - ◆ On each commit
  - ◆ Manually
  - ◆ On tags
  - ◆ On specific branches
  - ◆ Based on the commit name
  - ◆ On merge requests
  - ◆ ...

An example of a CI: gitlab-ci

It’s already there, why not use it ?

Gitlab-CI

- ➔ Gitlab comes with a CI/CD system: Gitlab-CI
- ➔ In your project, create a .gitlab-ci.yml
- ➔ This file is a config file describing your CI/CD:
  - ◆ What to do
  - ◆ When to do it
  - ◆ How to do it
- ➔ In the project options, CI/CD options available to:
  - ◆ Provide variables (like keys for deployment)
  - ◆ Setup pipeline triggers
  - ◆ ….

Gitlab-CI and its integration

- ➔ Gitlab-ci in itself is a very powerful tool
  - ◆ Checkout the gitlab-ci.yml reference file to be convinced
- ➔ Its strength is also with the integration it comes with:
  - ◆ Secrets
  - ◆ Built-in container registry
  - ◆ Specific/shared runners
  - ◆ Badges: pipeline status, coverage, etc
  - ◆ Triggers
  - ◆ Scheduling
  - ◆ Cross-projects

Gitlab as a DevOps tool

- ➔ Gitlab provides services for DevOps in general
- ➔ On top of the previously mentioned:
  - ◆ Deployment
    - ● Tracking deployments
    - ● Listing platforms with types
    - ● Well integrated with CI/CD
    - ● Integration with Sentry
  - ◆ Release
  - ◆ Allow advanced Git workflows
  - ◆ Permission system

How to write a .gitlab-ci.yml

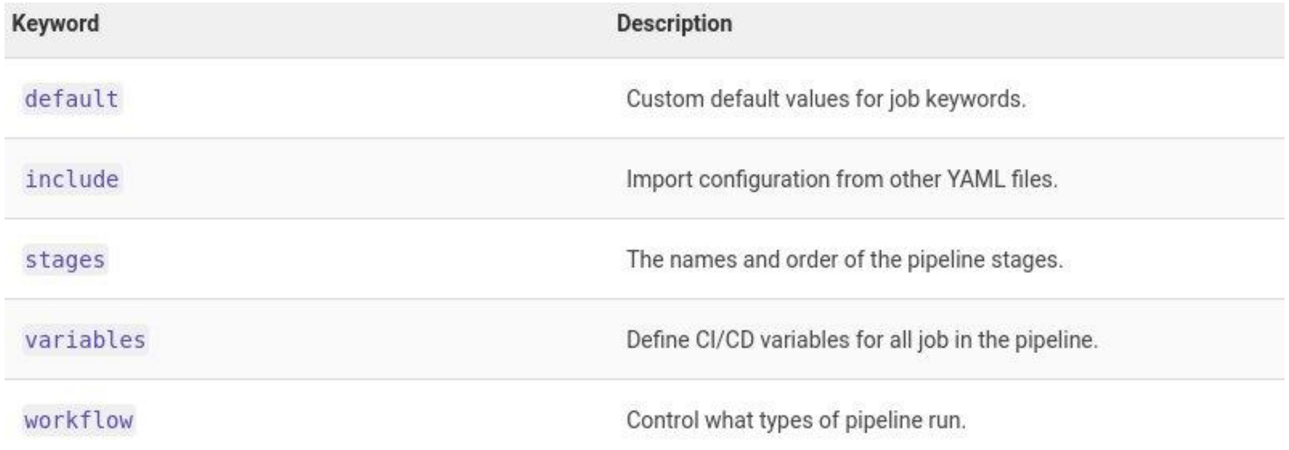
Don't get confused with jobs, stages, …

Gitlab-CI

- ➔ Gitlab-CI have a concept of pipeline
- ➔ A pipeline is a list of stages to execute in a specific order
- ➔ A stage is a list of jobs to execute in parallel
- ➔ You can put rules for which jobs/stage to run for a specific pipeline
- ➔ You can put rules to allow failure in a job to not fail the whole pipeline
- ➔ https://docs.gitlab.com/ee/ci/yaml/ is the reference

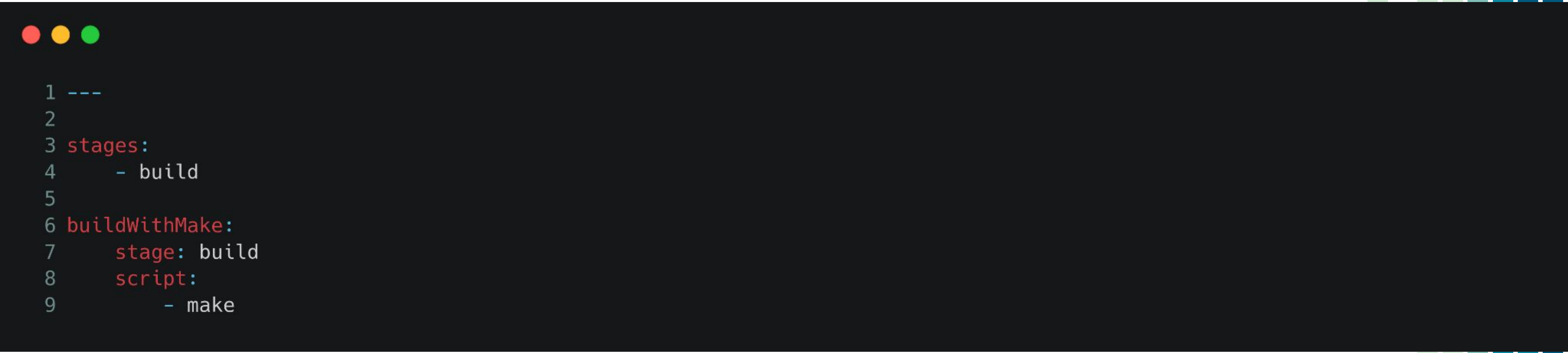
gitlab-ci.yml

➔ On the root of the YAML file you can put:
- A global keyword
- A job



Keyword	Description
`default`	Custom default values for job keywords.
`include`	Import configuration from other YAML files.
`stages`	The names and order of the pipeline stages.
`variables`	Define CI/CD variables for all job in the pipeline.
`workflow`	Control what types of pipeline run.

gitlab-ci.yml



```yaml
---

stages:
    - build

buildWithMake:
    stage: build
    script:
        - make
```

gitlab-ci.yml - job

- ➔ A job
  - ◆ has a name (its key on the root of the doc)
  - ◆ is in a stage
  - ◆ has a script to run

gitlab-ci.yml - job

- ➔ Job context is independant
  - ◆ Each job is run in a new environment
    - Except for the artifacts which remain
    - Artifacts are defined explicitly
  - ◆ The script is executed in a directory where the project is
    - It's provided as a git repo, you can do git operations
      - You can even commit from a CI/CD
  - ◆ Environment variables are provided with informations about the job
    - Git commit hash, git branch, repo URL, ...

gitlab-ci.yml - job

- ➔ How can you run a job in a new environment every time ?
  - ◆ Without being able to escape this environment
  - ◆ While being as deterministic as possible
  - ◆ While having a way to choose what will be in the env
    - ● Like packages, or even the OS
- ➔ If you don’t have a hint on how to implement that, go back to the first slide
- ➔ (sidenote: some people don’t use containers as a runner executor. It remains the most popular one though)

Observability

DevOps and observability

DevOps is also about visibility for everyone

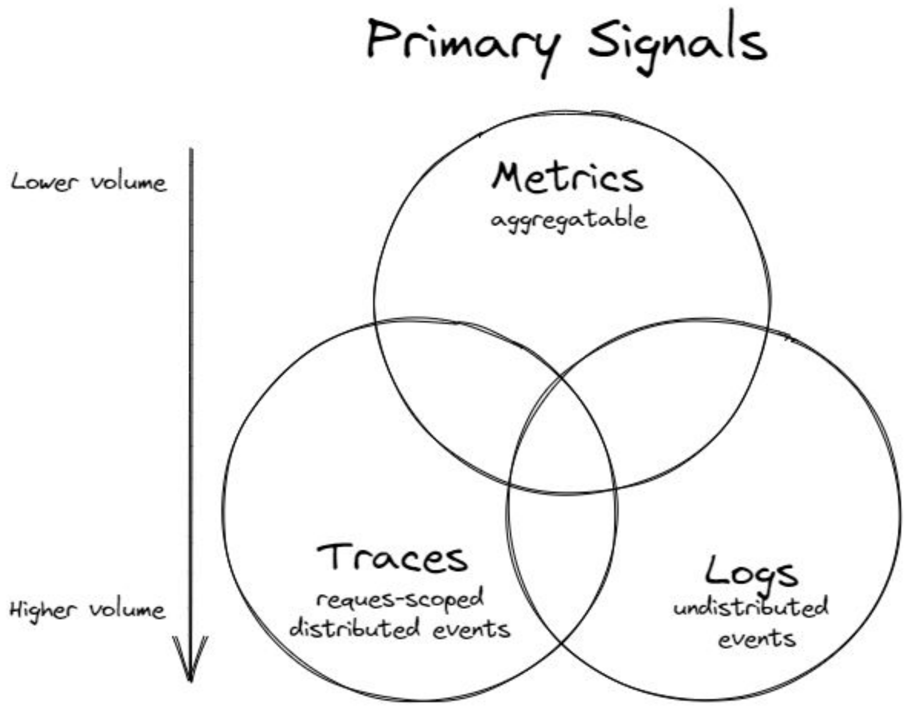
What is observability

- ➔ Once an application is in production, how does it behave ?
- ➔ Is it overloaded ?
- ➔ Is it working well ?
- ➔ Are there clients on it ?
- ➔ Are they facing errors ?
- ➔ Bugs ?
- ➔ If so, what kind ?
- ➔ How to investigate easily ?
- ➔ Shall we consider scaling up/down ?
- ➔ For an on-call ops, how to understand what's going on ?

What is observability

- ➔ A solution to all these questions are observability
- ➔ 3 pillars:
  - ◆ Metrics
  - ◆ Logging
  - ◆ Tracing
- ➔ Ops shall provide platforms to receive these signals
- ➔ Dev shall provide such observability in their apps
  - ◆ And if applicable, documentation about the observability
    - ● And the actions to take if any

What is observability



metrics

Who doesn’t love graphs, charts and histograms ?

Metrics

- Metrics is about exposing internal stats in a numerical form
- Metrics are meant to be aggregated
- Metrics are to be collected by an external tool
- Metrics are usually meant to be plotted
- 2 kinds of metrics:
  - Data already numeric
  - Data converted to be represented by numbers
- Metrics represent a state of a system at a given time
- Used to understand what is happening, not why

What is observability - metrics

- Most popular way of exposing metrics now:
  - Expose a HTTP route
    - Or HTTPS
    - /metrics
  - Prometheus format
- metric_name_unit{label1="value1", label2="value2"} value
  - i.e.:
    docker_containers_total{status="running"} 5
    docker_containers_total{status="stopped"} 1

What is observability - metrics

- ➔ What kind of metrics to expose ?
- ➔ Counter metrics
  - ◆ Only increasing counter
  - ◆ I.E:
    - ● Number of requests handled in total
    - ● Number of file read in total
    - ● ….

What is observability - metrics

- ➔ What kind of metrics to expose ?
- ➔ Gauge metrics
  - ◆ Single numerical value that can go up and down
  - ◆ I.E:
    - ● Number of videos being watched right now
    - ● Number of open files right now
    - ● Size of the event queue
    - ● ...

What is observability - metrics

- ➔ What kind of metrics to expose ?
- ➔ Histogram metrics
  - ◆ Samples of an observation divided in buckets
  - ◆ Meant to compute the φ-quantiles
  - ◆ I.E:
    - ● Amount of time taken to answer a request
    - ● Time taken writing data to cache
    - ● Size of a file uploaded
    - ● ….

What is observability - metrics

- ➔ What kind of metrics to expose ?
- ➔ Summary metrics
  - ◆ Like a Histogram but without the bucket sampling
  - ◆ I.E:
    - Amount of time taken to answer a request
    - Time taken writing data to cache
    - Size of a file uploaded
    - ...

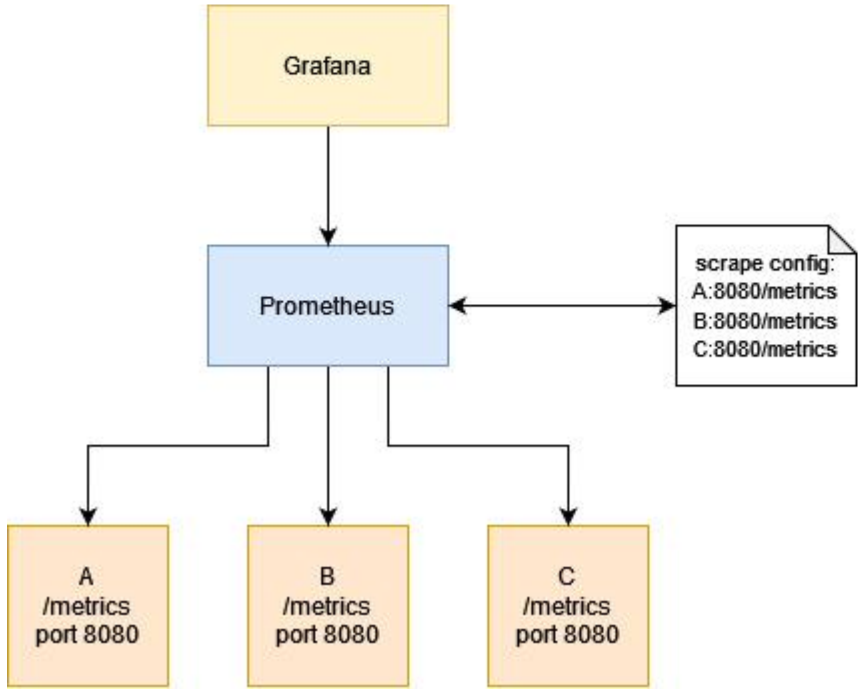
What is observability - metrics

- ➔ What kind of metrics to expose ?
- ➔ Info metrics
  - ◆ Hacking the metric system a bit to expose key-value info
  - ◆ I.E:
    - ● Version of the application
    - ● Build number
    - ● Hostname on which the app is running
    - ● …

What is observability - metrics

- ➔ Metrics are key to see what's going on
- ➔ We plot graph and we can visualize
- ➔ High level metrics (KPI) and low level
- ➔ Used for alerting
  - ◆ Ex: sudden drop of connected users
- ➔ Used for reporting
  - ◆ Ex: increasing amount of time taken to handle a request after an update
  - ◆ Ex: average user document size increasing over months

What is observability - metrics



logs

You logged things without even knowing it

What is observability - logs

- ➔ Logs are the most useful indication to understand what is going on in the app in details, with description
  - ◆ They don’t provide any global overview though
- ➔ Useful to get information about:
  - ◆ Understanding what’s going wrong
  - ◆ Which client/route/component is:
    - ● Used
    - ● Not working
    - ● Hammered
  - ◆ What is the app doing

What is observability - logs

- ➔ Logs can hold a lot of value
  - ◆ Even legal one, mind the GDPR for example
- ➔ 2 schools of thoughts about providing logs:
  - ◆ stdout/stderr (pull model)
  - ◆ syslog/elastic/… client (push model)
- ➔ **Logs must be structured**
  - ◆ syslog format
  - ◆ JSON
  - ◆ Homemade but consistent

What is observability - logs

- Why should logs be structured ?
- Useful to search for specific things
  - Logs will often be put in Loki, Elastic, ...
  - They provide query languages
    - Ex: {component="auth", severity="error"}
    - Ex: client_id: 10 AND route: "/login"
- Having a structure (and a consistent and documented one) is important
- GiB of logs to be generated: not read manually

What is observability - logs

- ➔ Logs shall have a severity level:
  - ◆ DEBUG
  - ◆ INFO
  - ◆ WARNING
  - ◆ ERROR
- ➔ Severity level must be configurable
- ➔ The amount of logs generated must be chosen carefully
- ➔ For DEBUG, don't care
- ➔ Starting from INFO, one must be wise
- ➔ Use a logging library

What is observability - logs

- What kind of logs can we have ?
- One think of application logging first
  - What is my application doing
  - Examples:
    - INFO User making a query on /api/v1/tickets
    - WARNING Cache is full. Dropping half its content
    - ERROR Exception uncaught while handling query
    - DEBUG read 17 bytes from file /var/run/myapp/fEcUs.tmp
    - FATAL Could not write data: disk is full

What is observability - logs

- ➔ What kind of logs can we have ?
- ➔ Security logs
  - ◆ Admin user changed its password
  - ◆ Deletion of xxx
  - ◆ New device logged-in
- ➔ Audit, system and infra logs are used for ops

tracing

Follow the trail of events in details

What is observability - tracing

- ➔ Logs are great to see some generic information about ongoing operation in the application
- ➔ They are not focused on a single user request
  - ◆ Single transaction
- ➔ Traces are meant to cover this case

What is observability - tracing

- ➔ What if you application throw an error while handling a client query ?
- ➔ You don’t want the whole app to crash for most cases
- ➔ Just return an error to the client
- ➔ You also need the error to be reported to you to fix it
- ➔ Logs ?
  - ◆ Stacktrace are multi-lines
  - ◆ They have their own context
  - ◆ Put in the logs some concise information usually
    - ex: “Can’t connect to DB”
    - ex: “Can’t find <...> for <...> via <...>”

What is observability - tracing

- Send the whole stacktrace and its context to another service
- Error tracking service
- Example: Sentry
- Regroup similar errors and plot their occurence
- Integrated with Gitlab to report bugs and regression
- Provide context
  - Browser used, account id, ... if useful
  - Crumbs
  - Runtime data
  - Alerts

What is observability - tracing

- Tracing also covers multi spans transactions for microservices architectures
- A trace is created for each transaction
  - Very high granularity
  - Often downsampled
- Used to understand interactions between services
  - Is the database slow ? microservice A, B or C ?
- A bit more difficult to put in place than logs & metrics
  - More specialized
- Check OpenTelemetry, Sentry, ZipKin, Tempo, ...

A connected element of observability: monitoring

Collect, store, observe, alert, report

Monitoring

- ➔ Monitoring is a subcategory of observability
  - ◆ Or another concept but tightly linked to observability
- ➔ Monitoring is about 5 points:
  - ◆ Collect
  - ◆ Store
  - ◆ Visualize
  - ◆ Alert
  - ◆ Report

Monitoring - Collecting

- ➔ The collecting component of a monitoring solution is in charge of getting the data
- ➔ Pull/Push
- ➔ Metrics-oriented mostly
- ➔ Examples:
  - ◆ Netdata
  - ◆ Node exporter/Prometheus
  - ◆ Metricbeats
  - ◆ Telegraf

Monitoring - Storing

- ➔ Metrics are representation of things happening at a given time
- ➔ Need to store them to have trends, history and evolution
- ➔ The volumetry can be quite high
  - ◆ Depends on the resolution (1s, 15s, 30s, 1m, ...)
  - ◆ Depends on the amount of metrics
    - ● Cardinality
- ➔ Use dedicated databases for this (often Time Serie DataBases)
- ➔ Example: Prometheus, Netdata, InfluxDB

Monitoring - Visualizing

- ➔ Once collected over time, metrics are best represented with graphs (visualization in general)
- ➔ Need to have real-time dashboards
- ➔ Graphs, histograms, plots, …
- ➔ Focus on the UI/UX
- ➔ Example:
  - ◆ Grafana
  - ◆ Netdata
  - ◆ Kibana
  - ◆ Chronograf

Monitoring - Reporting

- ➔ Reporting is most of the time forgotten and less popular
- ➔ Report once in a while (weekly, monthly, ...) what happened, and trends
  - ◆ To be read when needed, or from time to time
- ➔ Create a report (i.e. PDF file) with visualizations
- ➔ Used to be aware of non-immediate situation
- ➔ Often mangled with the visualization & alerting component

Monitoring - Alerting

- ➔ Observability & Monitoring are useful to see and try to understand what’s going on with your app/infra
- ➔ Used for post-mortems
  - ◆ Evidence of the issue
  - ◆ Provide tools for RCA to try to determine the RCE
- ➔ If one can understand through metrics an issue that happened, why not alert when it happens ?
  - ◆ Or even before if we can

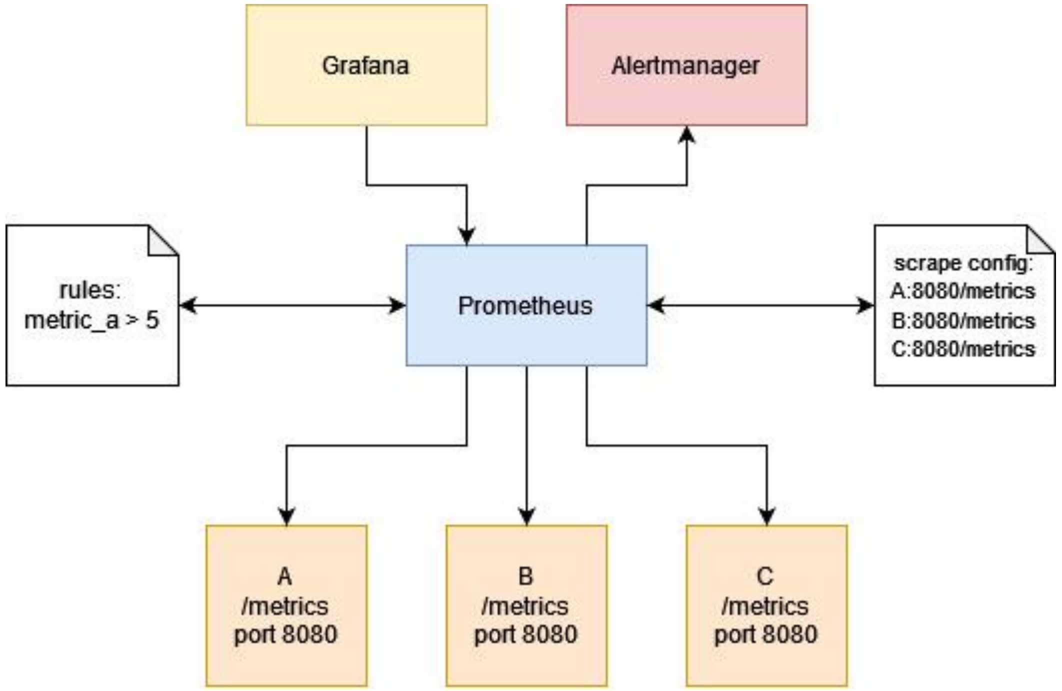
Monitoring - Alerting

- ➔ Alerting should be done first on high-level metrics
  - ◆ Number of clients
  - ◆ Number of videos being watched
  - ◆ Number of emails sent
  - ◆ Latency increasing
- ➔ Alerting can be done on low-level metrics with care
  - ◆ If high CPU but no impact on the client, is it an alert ?
  - ◆ 10% of disk left, is it the same if 1GiB left of 1TiB ?

Monitoring - Alerting

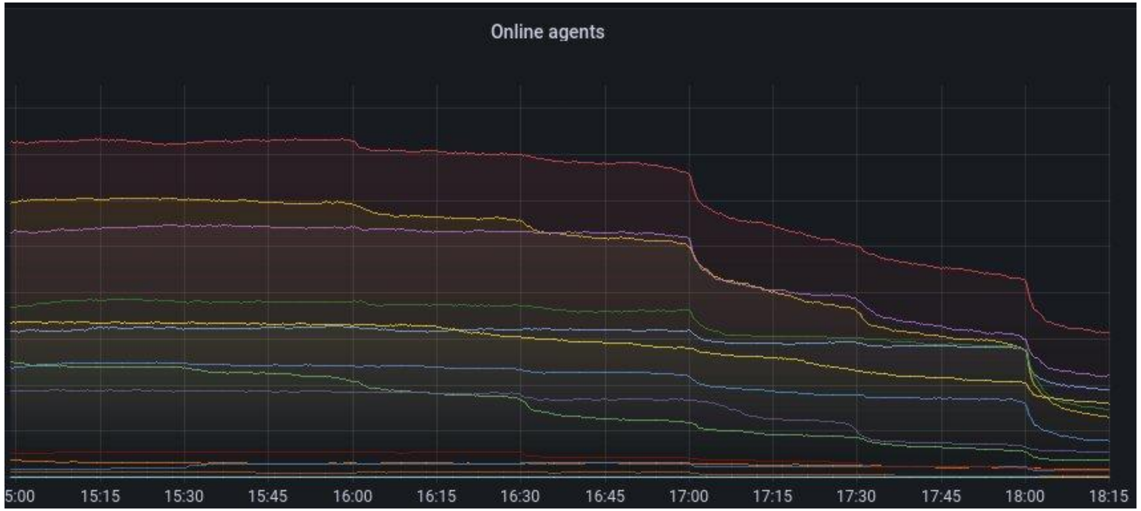
- Alerts should have multiple level of criticality
  - Think about whether to wake up an ops or not for example
  - Lowest level(s) can even be moved to reporting
- Too many alerts = alert fatigue
- Too many false positives = more chances to miss an important one

Monitoring –Alerting archi



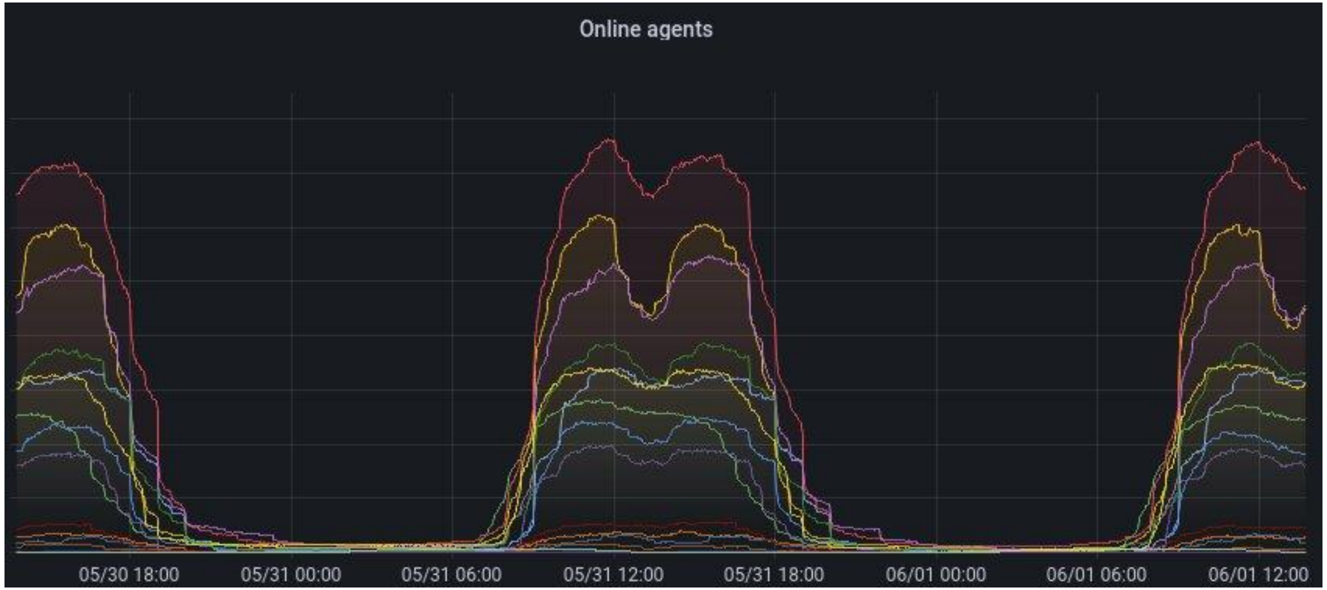
Monitoring - Alerting

➔ Be extra careful with alerting



Monitoring – Alerting

➔ Be extra careful with alerting



Monitoring - Alerting

- ➔ How to do proper alerting ?
- ➔ Many ask the question and few found the answer
- ➔ Appears to always be a balance between too many and too few alerts

Monitoring - Alerting

- ➔ Alerts should be derived from SLIs, taking into account the SLOs and sold SLA
  - ◆ Service Level Indicators - A quantifiable indicator of the level of service provided (often mistaken for KPIs)
  - ◆ Service Level Objectives - An objective set on a SLI about how much a SLI can fail
  - ◆ Service Level Agreement - What has been sold to the client in terms of disponibility

Monitoring - Alerting

- ➔ SLA is 100% - SLO, represents the % of availability
  - ◆ Also expressed in a number of 9
    - ● Three 9s means 99.9% of availability
  - ◆ Also expressed in allowed failure time per period of time
- ➔ 99.9% of SLA = 8.7h/y, 44min/m
- ➔ 99.99% = 52min/y, 4.3min/m
- ➔ 99.999% = 5.2min/y, 26.3s/m
- ➔ Alerts can use this budget of allowed failure to avoid false positives by working on the burn rate

Thanks !

Questions ?

Slides available on zarak.fr/

Contact: cyril@cri.epita.fr

zarak production#5492