Build and Deploy pipelines at Deutsche Telekom

Table of contents

* General principles of the CICD
* Implementation of the CICD pipelines
* Docker images builds with kaniko
* Vulnerabilities detection in Docker images
* Build dependencies management
* Caching build Java / Maven

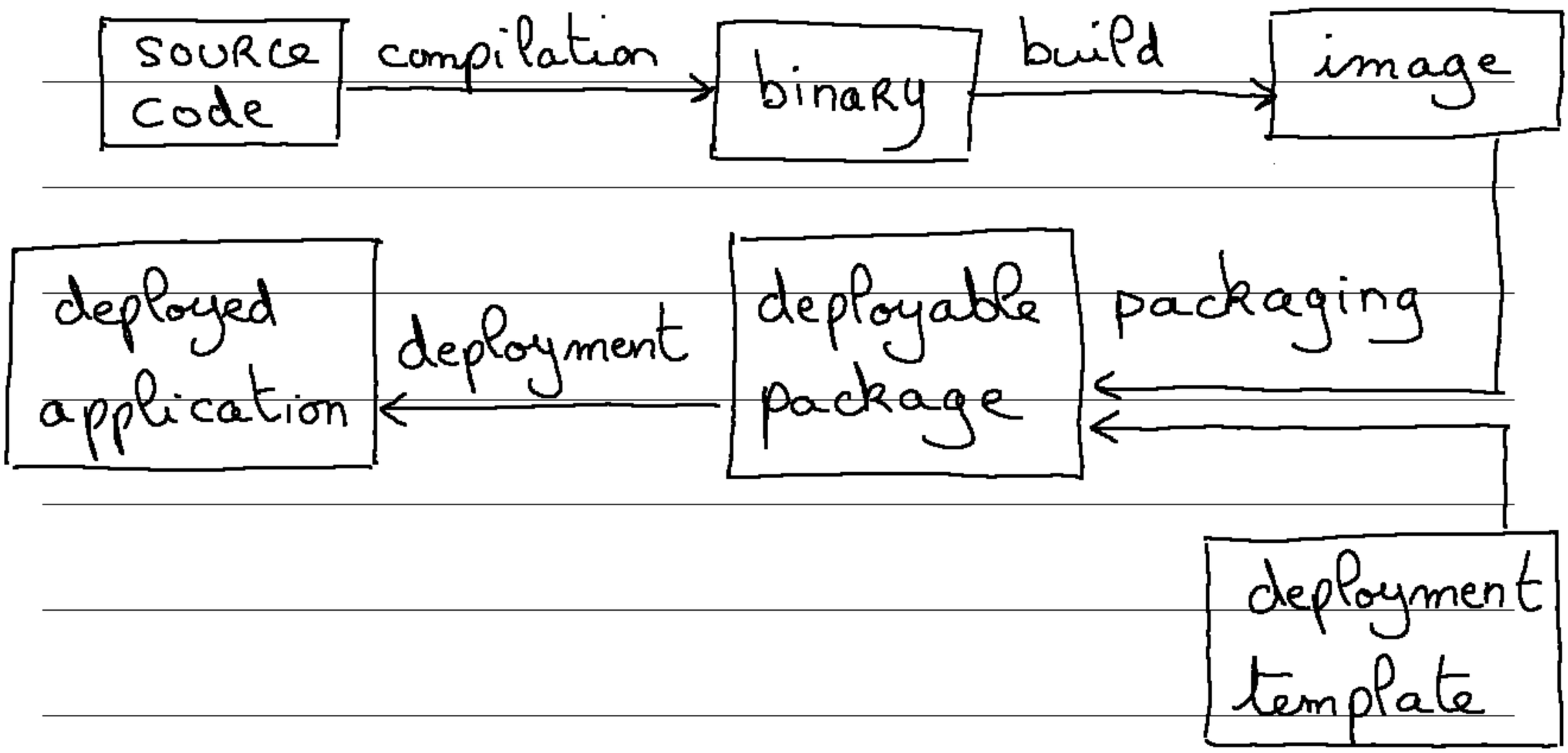
General principles of the CICD

4 distincts stages :

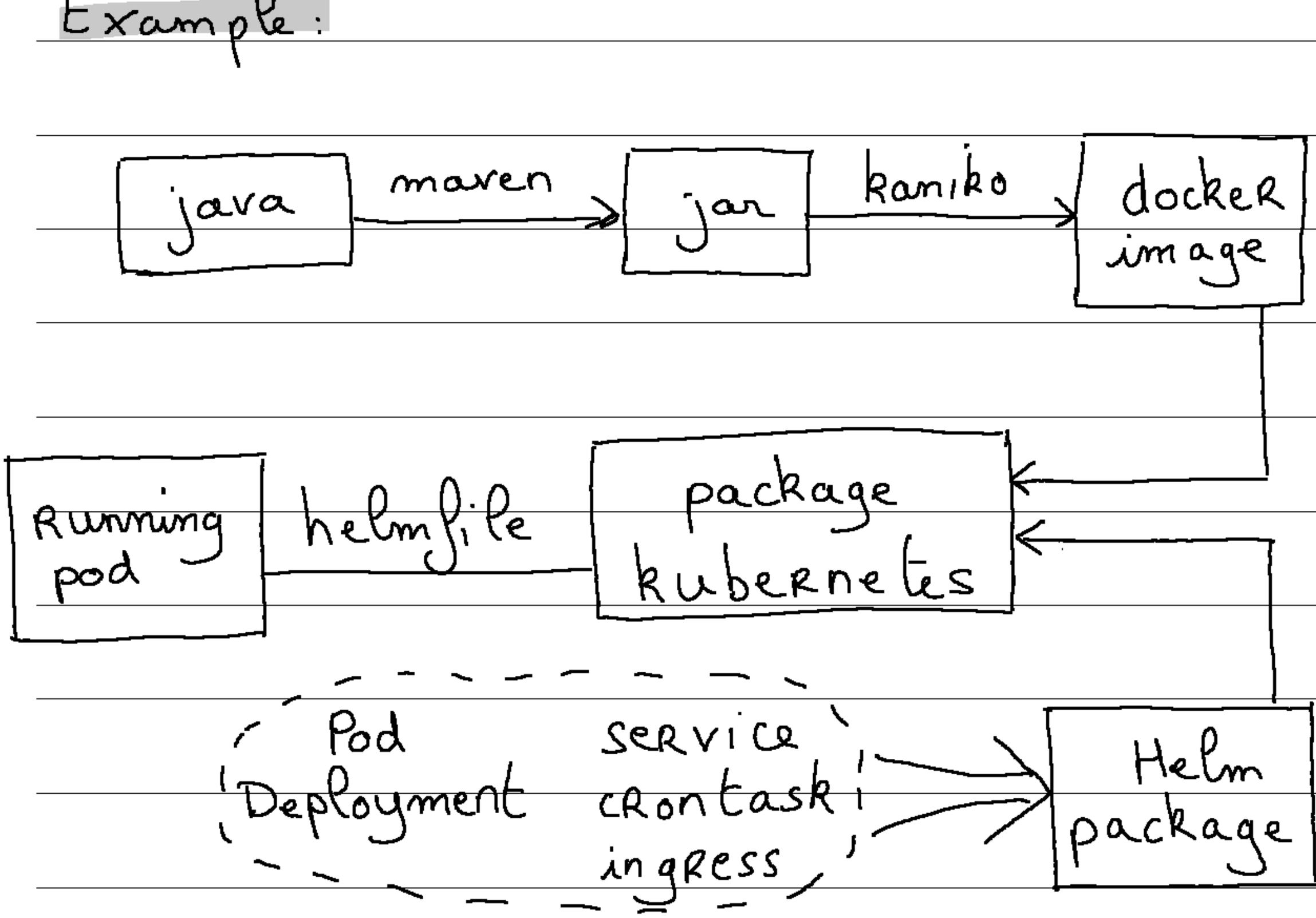
- development
- integration
- staging
- production

Dev	Int
Build Tests Deployment	Tag Deployment Tests
Staging	**Prod**
Promote tag Deploy Tests	Promote tag Deploy Use

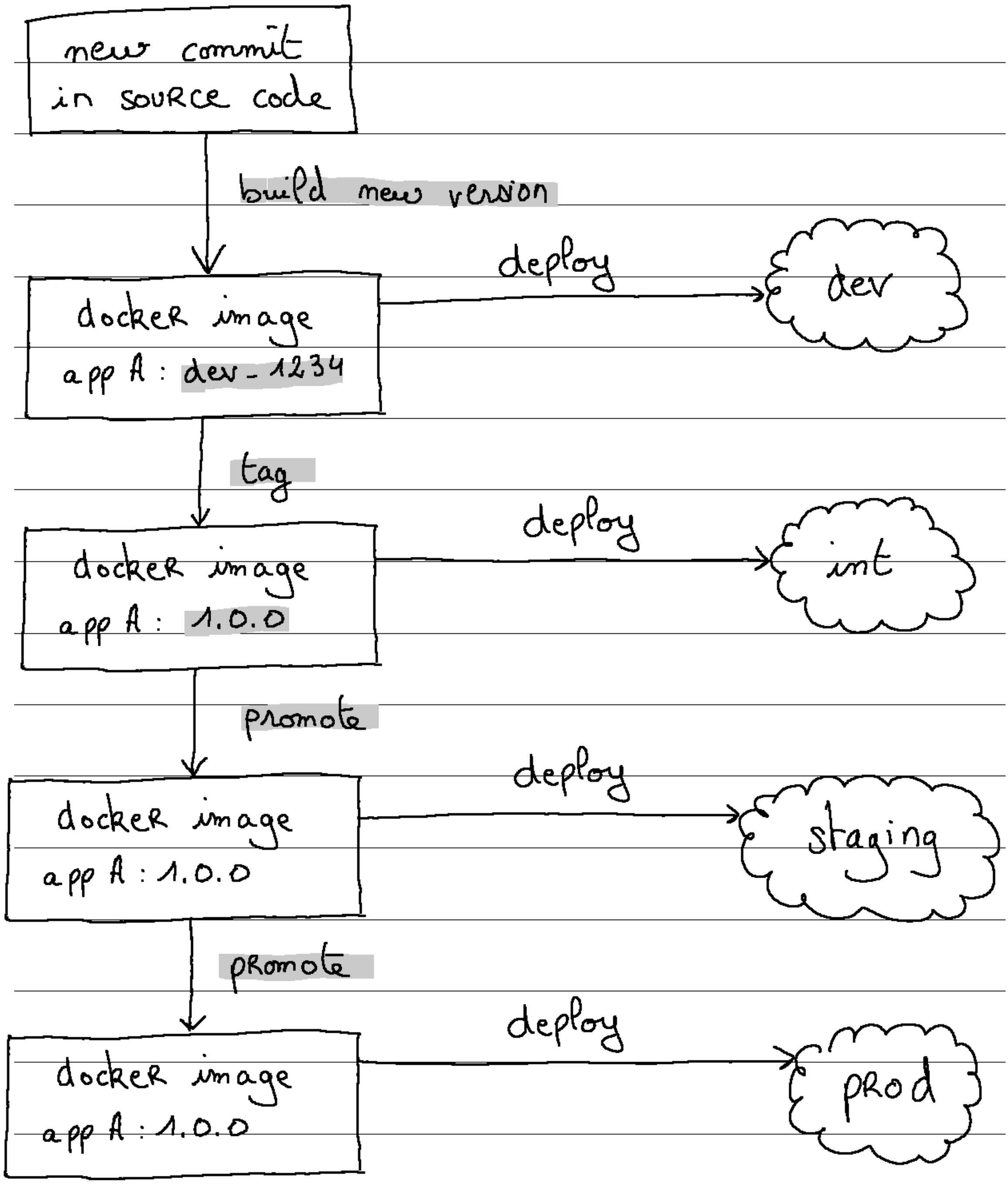
From source code to a deployed app



Example :



Deployment from dev to prod



Implementation of the CICD pipelines

- Using Gitlab as CI tool

- Using gitlab-ci.yml files to describe how the build should be done

- Implemented generic pipelines that can simply be included

- Content of a typical .gitlab-ci.yml :

```
include :
    - project : " cicd / gitlab - share "
      Ref : "master"
      file : " .gitlab - ci - build - image . yml "
```

Ci Pipeline Functionalities

Once included, the pipeline will :

- check validity of semantic versioning (tags)
- check if using docker images from allowed registry
- check if deployments define limits, Request, liveness & readiness probes
- generate dynamically corresponding build configs
- build docker image
- push docker image to Registry
- check vulnerabilities
- trigger downstream project
- update the version deployed (in dev / int)

Docker images builds with kaniko

. Read docker-compose file to get the list of docker images to build:
    * path to the dockerfile
    * image name
    * image tag
    * environment variables available during the build

. Creates dynamic nested child pipelines for each docker image to build

. Build each docker image with kaniko in kubernetes without needing any root access

Vulnerabilities detection in Docker images

During the build :

* Kaniko pushes images in Gitlab Registry
* Grype (or equivalent) runs to get the list of vulnerabilities of the docker image.

After pushing the image to the MTR (Magenta Trusted Registry) which is a rebranded version of Quay , vulnerability tests are run and the results are stored.

During the deployment :

* Get the list of images to deploy
* Request Quay API to retrieve the precomputed list of vulnerabilities
* Ignore each vulnerability marked as excluded
* Display the list to the user

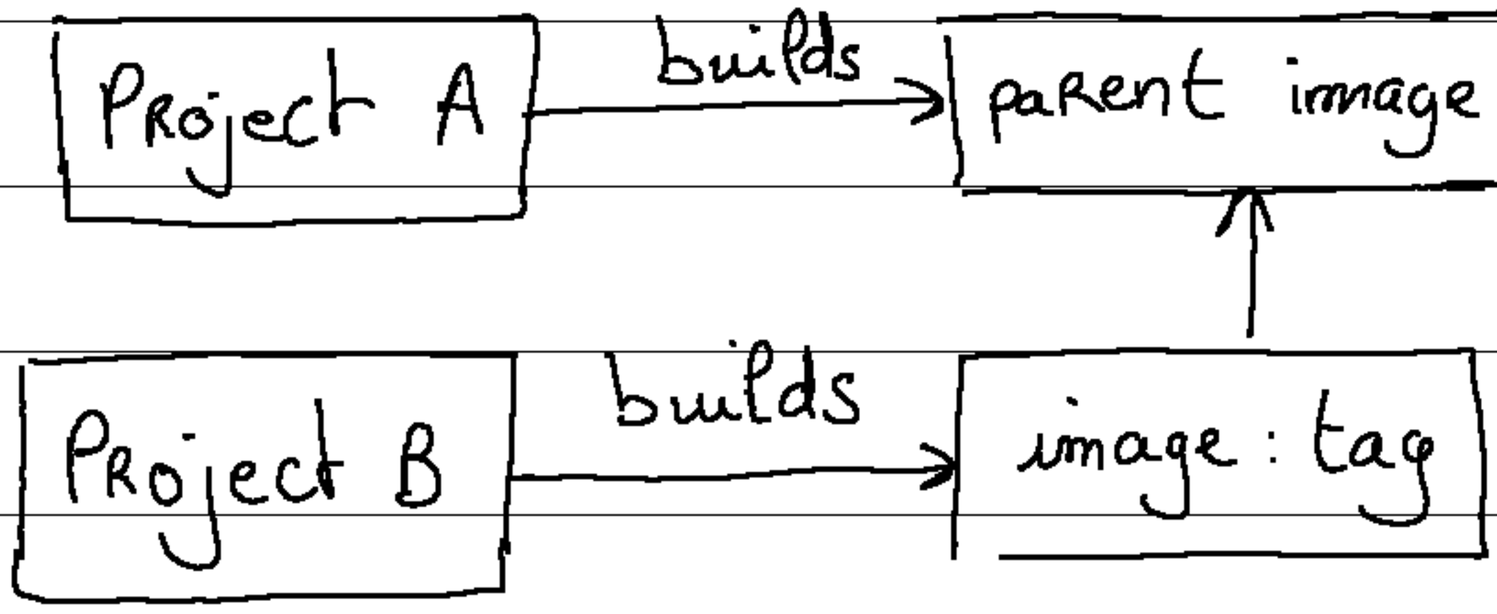
Build dependencies management

* Docker images can inherit from each others

```
FROM parent_img:tag

COPY <...>
RUN <...>
```

* Therefore, projects depend on each others



* Problem: If project A is rebuilt, project B should be rebuilt !

Build Dependencies Analyzer

* Python project looping over docker projects using Gitlab API

* For every single project, retrieve and analyze the Dockerfile

* Creates a tree of dependencies:
  - project to dockerfile
  - dockerfile to parent dockerfile

* Loop again over docker projects and add a metadata file with all child projects

* One of the last step of the build pipeline reads this metadata file and triggers a build of every single child project.

Caching build Java / Maven

. Automatically activated when a pom.xml is detected at the root of the project.

. One build step is dynamically added to pre-download all dependencies

. Maven cache directory (~/.m2) is then stored in GitLab cache.

. The main build step mounts and re-uses the cache

Thanks for

Listening

&

Special Thanks to

Julien Acroute for

his support & technical

Lead