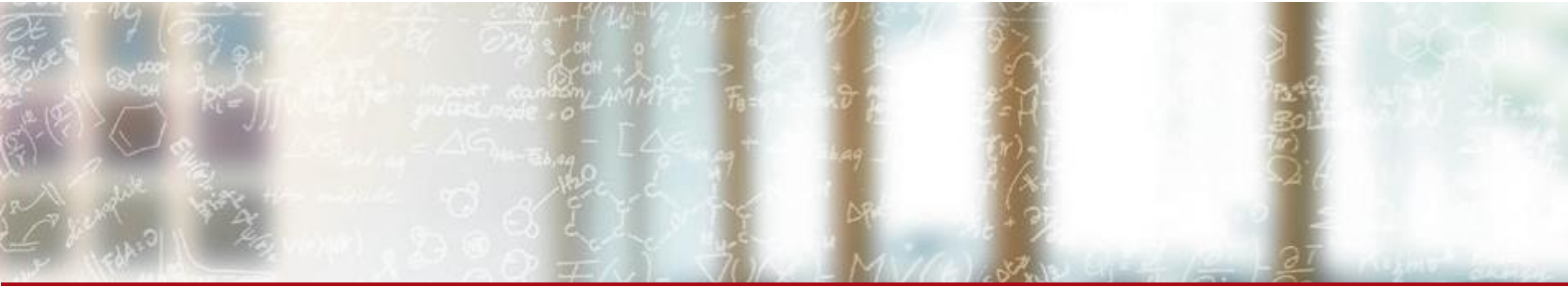
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Containerized CI/CD on Alps

Theofilos Manitaras, CSCS
Andreas Fink, CSCS
October 4, 2023

Table of Contents

Creating container images

Creating container images

Creating container images is straightforward:

```
include:
  - remote: 'https://gitlab.com/cscs-ci/recipes/-/raw/master/templates/v2/.ci-ext.yml'

stages:
  - build

build software:
  extends: .container-builder
  stage: build
  variables:
    PERSIST_IMAGE_NAME: $CSCS_REGISTRY_PATH/hello_world:01-$CI_COMMIT_SHORT_SHA
    DOCKERFILE: ci/docker/Dockerfile
```

Needed for Image Building

Mandatory variables

You can find extra information and options regarding building container images here

Testing container images

Testing container images

After building a container image, it's essential to test/run it on the target system(s):

```
.
.
test software:
  extends: .container-runner-daint-mc
  stage: test
  image: $CSCS_REGISTRY_PATH/hello_world:01-$CI_COMMIT_SHORT_SHA
  script:
    - /build/hello
  variables:
    SLURM_NTASKS: 4
    SLURM_JOB_NUM_NODES: 1
    SLURM_LABELIO: 1
    USE_MPI: 'YES'
```

Selecting the corresponding runner

Container command to execute

Variables controlling slurm allocation & Sarus parameters

You can find extra information and options regarding running containers here

Embracing multi-stage builds & reusing images

Multi-stage builds

Multi-stage builds are an essential mechanism during the image building:

- Multiple FROM statements are used in a Dockerfile
- Artifacts can be copied from one stage to another
- Allow creation of lightweight images containing only the essentials to run

```
# Start from "devel" image containing all the necessary components for building
FROM docker.io/nvidia/cuda:11.2.2-devel-ubuntu20.04 as builder
.
.
# Build binary <my_binary>
.
.
# Build the lightweight runtime image
FROM docker.io/nvidia/cuda:11.2.2-runtime-ubuntu20.04

COPY --from=builder <source path to binary> <target path>
```

Image splitting (1/2)

A typical pattern emerging when using container images is the following:

- Image components e.g third party libraries, change at a much slower pace than the actual software being developed
- Building an image with all the components from scratch might take a long time, wasting computing resources

A solution to the above is to split the image in two:

1. An image containing the build environment, that can be updated on demand
2. An image that only builds the software to be tested/shipped, using the base image as it's starting point

Image splitting (2/2)

An example pipeline is given here:

```
build base:
  extends: .container-builder-dynamic-name
  stage: build_base
  variables:
    WATCH_FILECHANGES: ci/docker/Dockerfile.base
    DOCKERFILE: ci/docker/Dockerfile.base
    PERSIST_IMAGE_NAME: $CSCS_REGISTRY_PATH/base/hello_world_base

build software:
  extends: .container-builder
  stage: build
  variables:
    PERSIST_IMAGE_NAME: $CSCS_REGISTRY_PATH/hello_world:03-$CI_COMMIT_SHORT_SHA
    DOCKERFILE: ci/docker/Dockerfile
    DOCKER_BUILD_ARGS: '["BASEIMG=$BASE_IMAGE"]'
```

Use to propagate BASE_IMAGE

Build when base Dockerfile changes

Pass BASE_IMAGE as build argument

Building for the target architecture

Building for the target architecture

Container images should be built with the target running system in mind:

- Crucial in order to generate binaries compatible with system specific futures (e.g accelerators, high speed interconnects, vector instructions)
- Relying on autodetection of system characteristics should be avoided since, the system where the image is built might be different from the target one
- CSCS offers a growing list of essential building blocks to facilitate the compatibility and performance of the resulting images:
  - Base images
  - CI pipeline templates
  - Helper scripts (e.g spack-install-helper)

Building for the target architecture example (1/2)

Different images can be built from scratch as follows:

```
FROM ubuntu:22.04 as builder

ARG TARGET=haswell
.
.
.
RUN mkdir -p /opt/spack/spack-env/ \
.
.
&& (echo "spack: " \
&&  echo "  packages: "\
&&  echo "    all:"\
&&  echo "      target: [${TARGET}]"\
&&  echo "  specs:" \
&&  echo "  - osu-micro-benchmarks@6 ^${MPI_SPEC}" \
.
.
.
```

Accept the architecture as a build argument

Use the architecture with Spack

Building for the target architecture example (2/2)

The corresponding pipeline looks as follows:

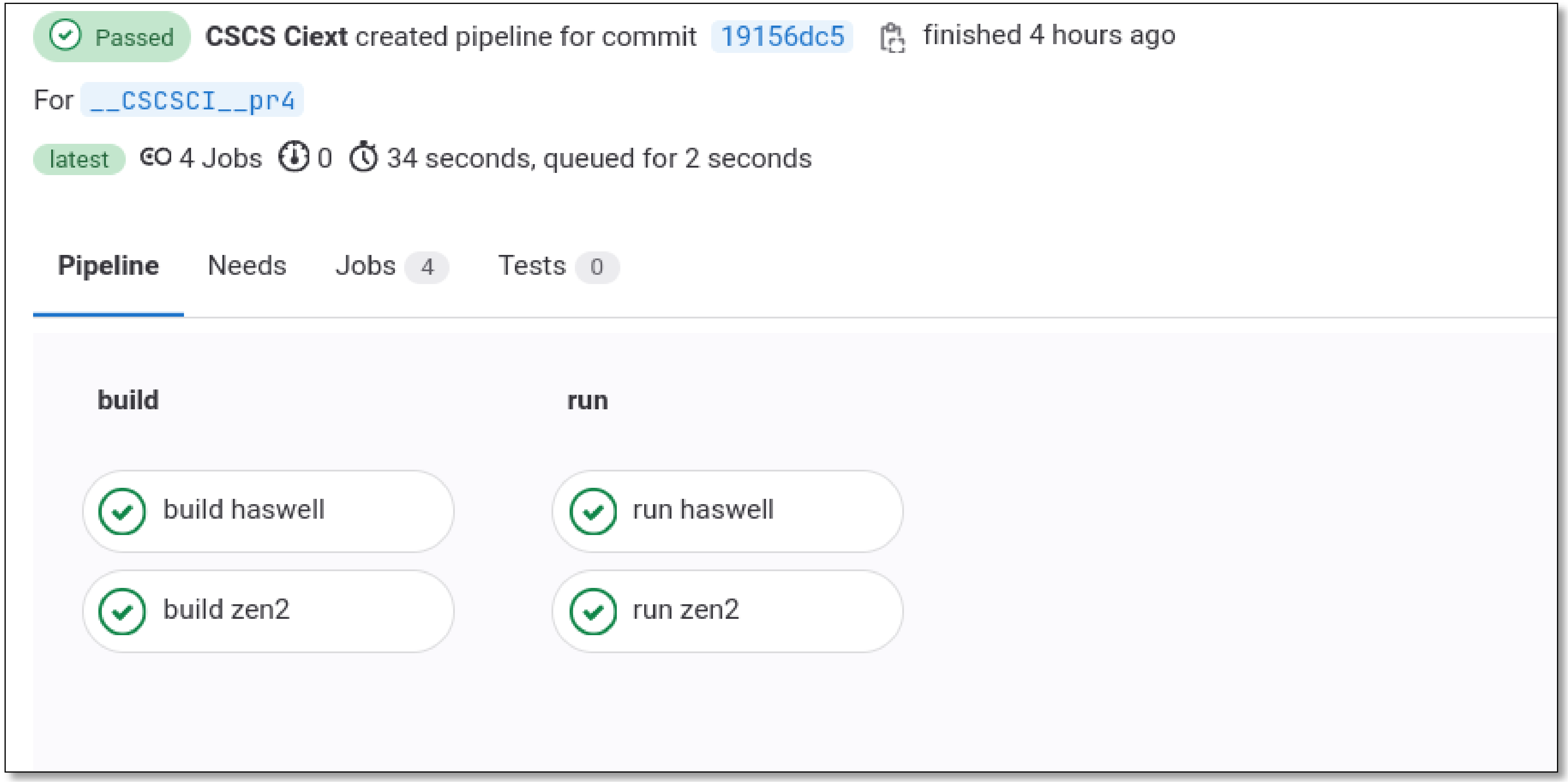
```
build haswell:
  extends: .container-builder
  stage: build
  variables:
    PERSIST_IMAGE_NAME: $CSCS_REGISTRY_PATH/mpich_osu:3.4.3_6.0_haswell
    DOCKERFILE: ci/mpi_osu/Dockerfile
    DOCKER_BUILD_ARGS: '["TARGET=haswell"]'

build zen2:
  extends: .container-builder
  stage: build
  variables:
    PERSIST_IMAGE_NAME: $CSCS_REGISTRY_PATH/mvapich2_osu:2_6.0_zen2
    DOCKERFILE: ci/mpi_osu/Dockerfile
    DOCKER_BUILD_ARGS: '["TARGET=zen2"]'
```

Build for haswell

Build for zen2

Build/Run pipeline

Final pipeline for different architectures



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Image labels

Labels in a Nutshell

Labels are a mechanism for adding useful metadata to a container image:

```
FROM  docker.io/nvidia/cuda:11.2.2-devel-ubuntu20.04 as builder

# Labels for builder image
LABEL "rfm.features"="osu-micro-benchmarks;mpi;serial;openmp;mvapich2"
LABEL "rfm.cc"="mpicc"
LABEL "rfm.cxx"="mpic++"
LABEL "rfm.ftn"="mpif90"
.
.
.
# Labels for runner image
FROM docker.io/nvidia/cuda:11.2.2-runtime-ubuntu20.04

LABEL "maintainer"="cscs-ci-demo"
LABEL "rfm.features"="osu-micro-benchmarks;mpi;serial;openmp;cuda"
LABEL "rfm.cc"="mpicc"
LABEL "rfm.cxx"="mpic++"
LABEL "rfm.ftn"="mpif90"
```

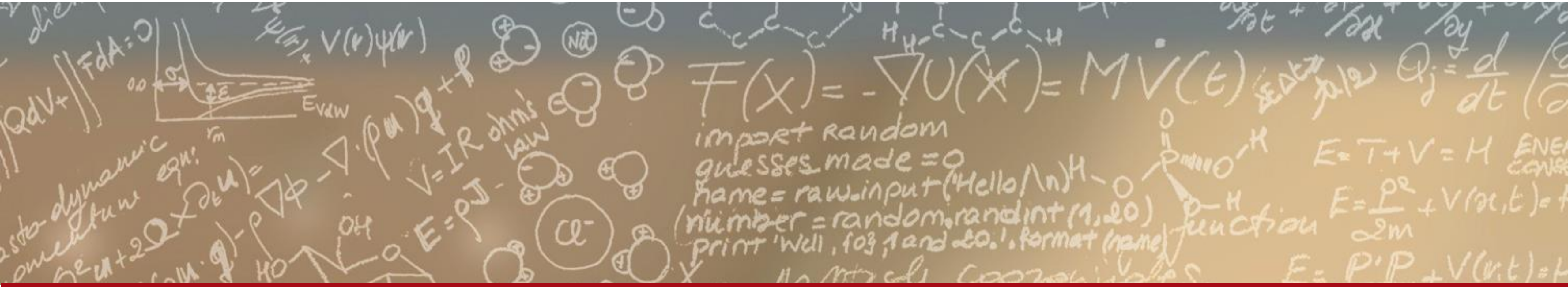
Accessing Labels

Labels can be easily accessed via image inspection, using Docker, skopeo, etc.

```
# Inspect using Docker
$ docker inspect <image_name>  --format="{{json .Config.Labels}}" | jq
{
  "maintainer": "cscs-ci-demo",
  "rfm.cc": "mpicc",
  "rfm.cxx": "mpic++",
  "rfm.features": "osu-micro-benchmarks;mpi;serial;openmp;cuda",
  "rfm.ftn": "mpif90"
}

# Inspect using Skopeo
$ skopeo inspect docker://<image_name> | jq '.Labels'
{
  "maintainer": "cscs-ci-demo",
  "rfm.cc": "mpicc",
  "rfm.cxx": "mpic++",
  "rfm.features": "osu-micro-benchmarks;mpi;serial;openmp;cuda",
  "rfm.ftn": "mpif90"
}
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Thank you for your attention.