Exploring CI/CD Tools and Workflows for Research and HPC Software Projects

CEEC

Centre of Excellence in Exascale CFD

Profile

- Michael Zikeli | michael.zikeli@fau.de
- PhD student in RSE
- CI/CD Maintainer
- Friedrich-Alexander-University Erlangen-Nuerenberg (FAU)
- Chair for System Simulation (LSS)
- Software:
    - **waLBerla** (Simulation Framework)
    - HyTeG (Simulation Framework)
    - ExaStencils (Code Generator)
    - pystencils and lbmpy (Code Generator)

CX in HPC

Continuous Integration (CI)

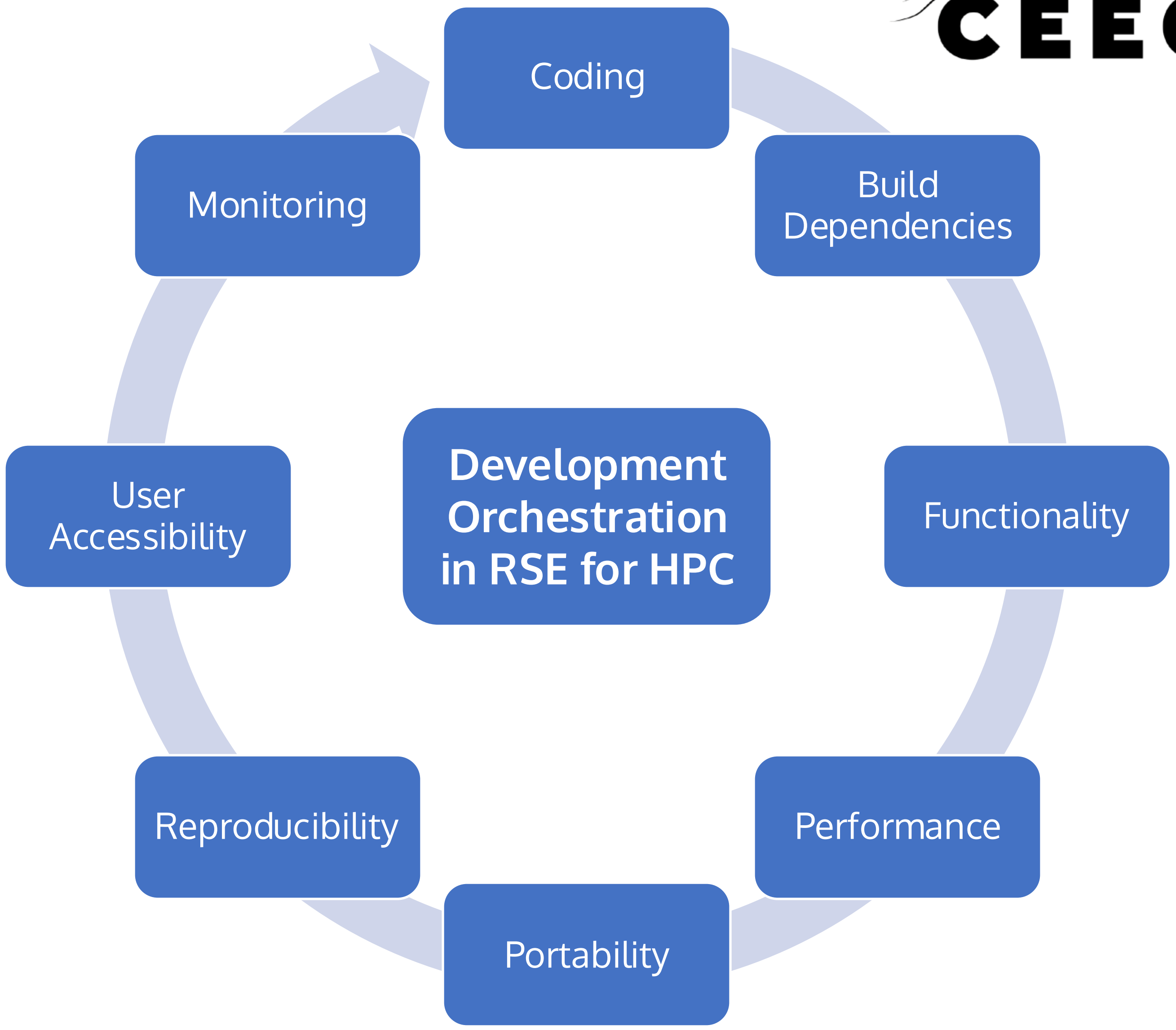
- Automatically **build & test every change**
- Ensures correctness across compilers, architectures, and systems

Continuous Delivery / Deployment (CD)

- Automatically **produce and ship** reproducible builds and environments
- Ensures software availability

Continuous Benchmarking (CB)

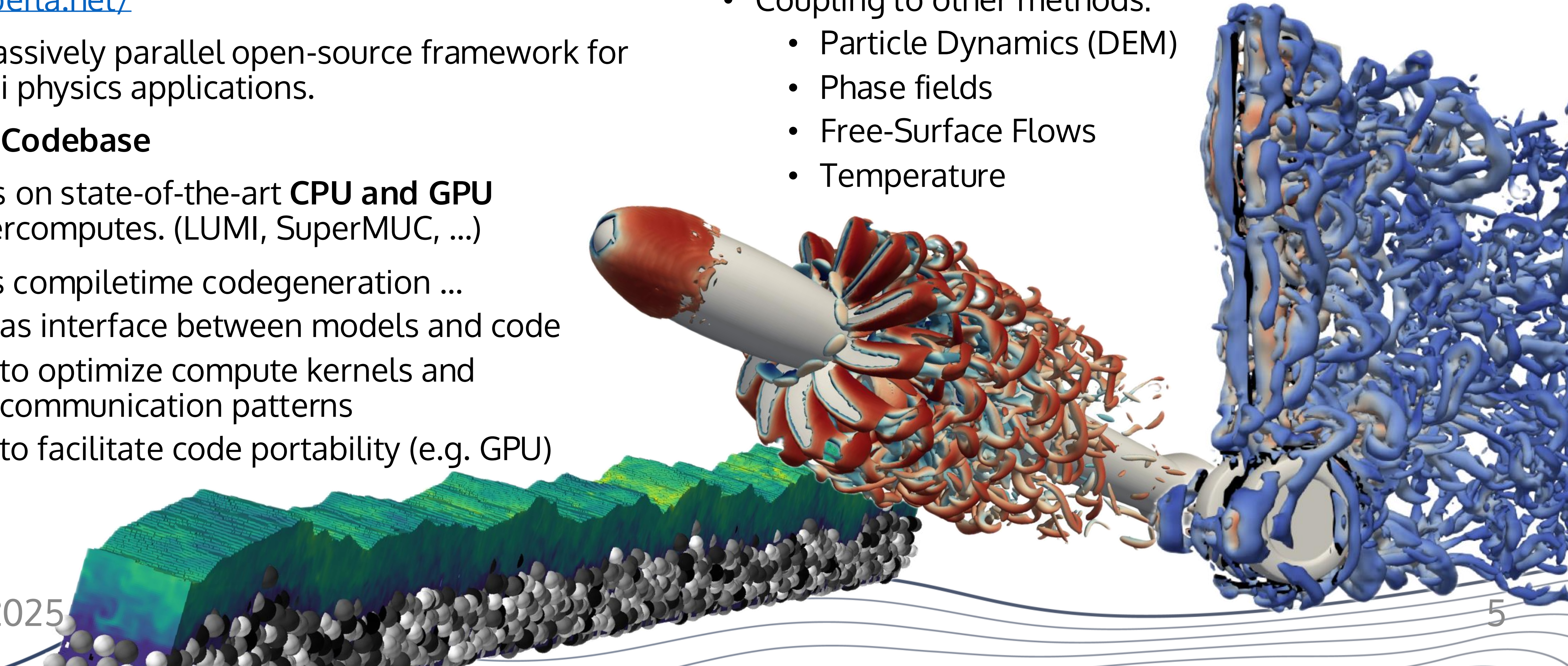
- Automatically measure and **track performance over time**
- Validates **scaling behaviour**



Showcase: waLBerla

- **w**idely **a**pplicable **L**attice **B**oltzmann from **Erla**ngen (**waLBerla**)
- walberla.net/
- A massively parallel open-source framework for multi physics applications.
- **C++ Codebase**
- Runs on state-of-the-art **CPU and GPU** supercomputes. (LUMI, SuperMUC, …)
- Uses compiletime codegeneration …
  - as interface between models and code
  - to optimize compute kernels and communication patterns
  - to facilitate code portability (e.g. GPU)

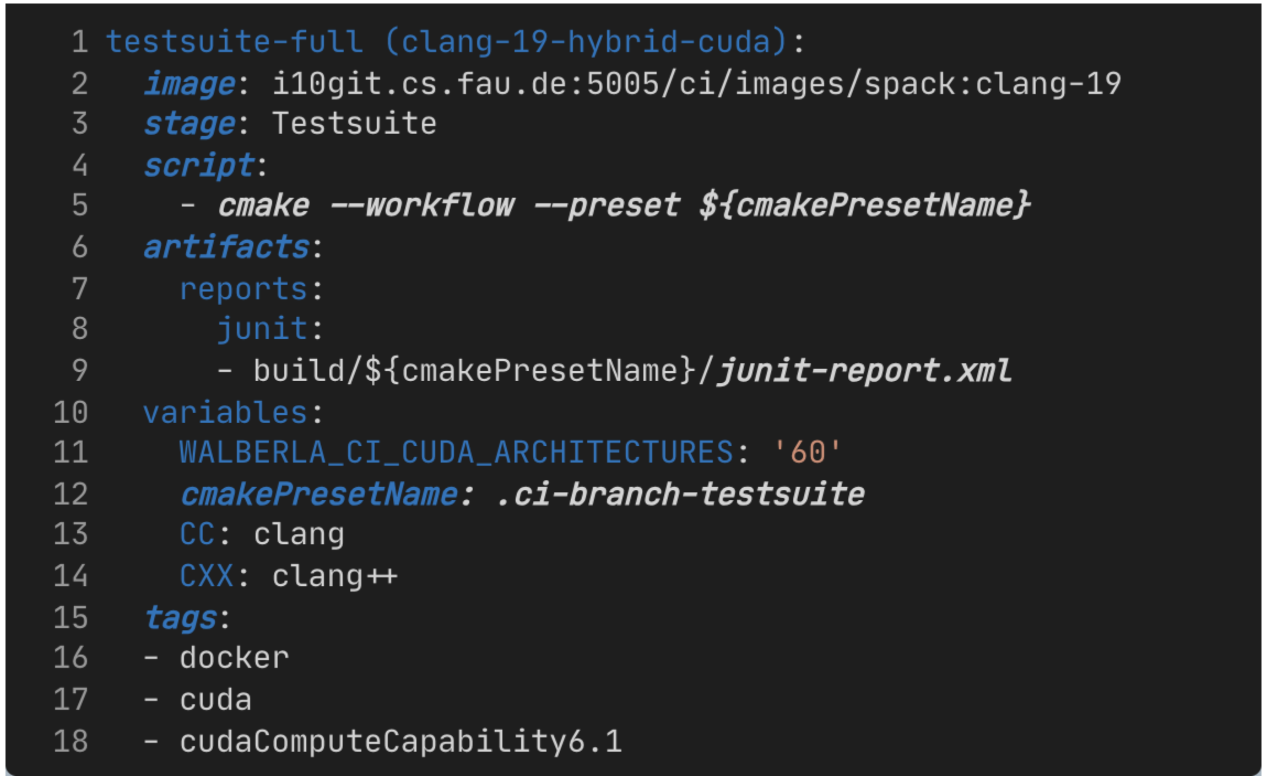
- Focuses on **CFD** Domain
  - Lattice Boltzmann Method (**LBM**)
  - Coupling to other methods:
    - Particle Dynamics (DEM)
    - Phase fields
    - Free-Surface Flows
    - Temperature



Showcase: waLBerla-CI

- **Codebase** at self hosted **GitLab**
  - i10git.cs.fau.de/walberla/walberla
- **CI pipeline orchestrator** is **GitLab CI**
  - **YAML** based Pipeline configurations
  - GitLab CI **schedules CI-Jobs** from pipeline
  - **Each Job** defines:
    - Its execution environment as **Image**
    - Hardwae requirements per **Tags**
    - Specific build flags
    - **Artifacts** to keep after execution
- **GitLab Runner executes** queued CI-Jobs
  - Runners are taged with their specs
    - OS: **Linux** or **MacOS**
    - GPU: **NVIDIA** for Linux
    - CPU ISO: **X86** Linux or **ARM** MacOS

CI Job definition example:



```yaml
testsuite-full (clang-19-hybrid-cuda):
  image: i10git.cs.fau.de:5005/ci/images/spack:clang-19
  stage: Testsuite
  script:
    - cmake --workflow --preset ${cmakePresetName}
  artifacts:
    reports:
      junit:
        - build/${cmakePresetName}/junit-report.xml
  variables:
    WALBERLA_CI_CUDA_ARCHITECTURES: '60'
    cmakePresetName: .ci-branch-testsuite
    CC: clang
    CXX: clang++
  tags:
  - docker
  - cuda
  - cudaComputeCapability6.1
```

waLBerla – **CI Pipeline**

CI pipeline is **triggered** by every code **„push“**.
BUT some jobs only run on specific branches:

Branch

Commits to **feature branches**

High execution **frequency**

Small test space

Saves resources

MR

Commits to branches with **open merge** requests

Large test space

Used as **Acceptance Test**

Stable

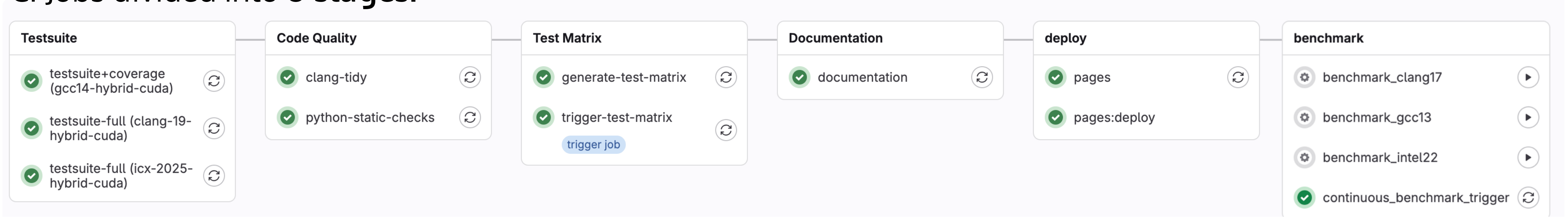
Commit to **main branch**

Regression Test

+ Performance Regression

Updates **GitLab Pages**
Documentation
Coverage report

CI Jobs divided into **6 stages:**



Testsuite
- ✓ testsuite+coverage (gcc14-hybrid-cuda)
- ✓ testsuite-full (clang-19-hybrid-cuda)
- ✓ testsuite-full (icx-2025-hybrid-cuda)

Code Quality
- ✓ clang-tidy
- ✓ python-static-checks

Test Matrix
- ✓ generate-test-matrix
- ✓ trigger-test-matrix (trigger job)

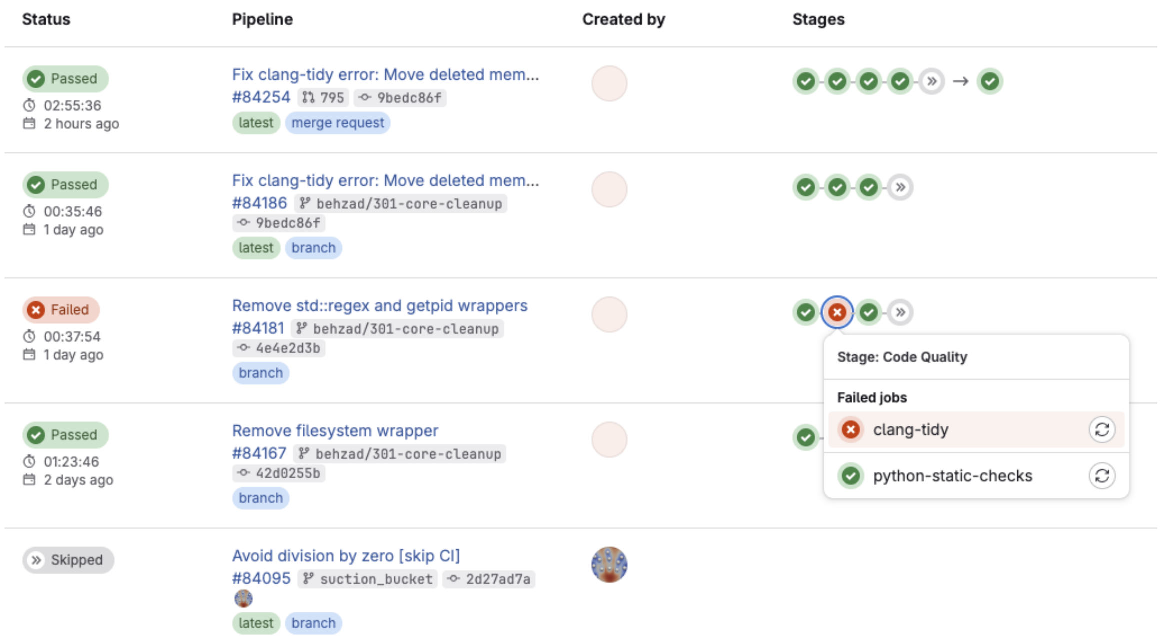
Documentation
- ✓ documentation

deploy
- ✓ pages
- ✓ pages:deploy

benchmark
- benchmark_clang17
- benchmark_gcc13
- benchmark_intel22
- ✓ continuous_benchmark_trigger

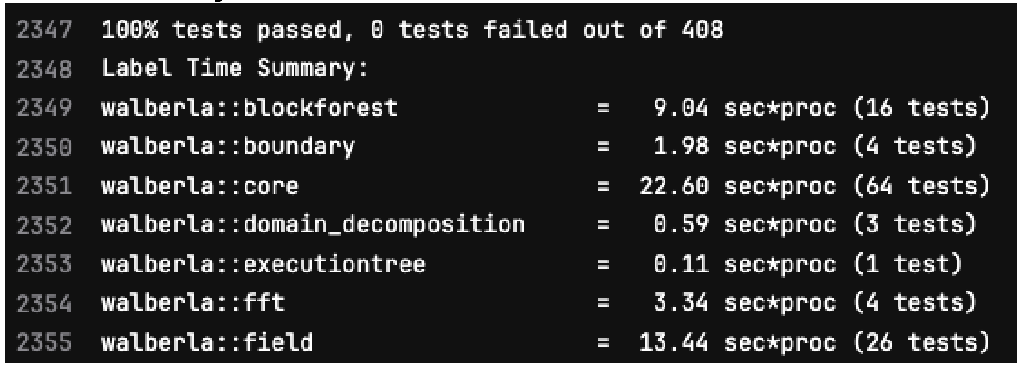
CI-Stage – **Testsuite**

- **Frequent small test set** to catch errors during coding
    - Executes on **all branches**
- Framework is built with different
    - **Compilers**: gcc, clang, icx
    - **Build Parameters**: e.g. CUDA, MPI, OMP, AVX, …
- All CI Tests are performed for build configuration
    - Unit, Integration, Regression, …
- Direct Feedback from GitLab GUI + Mail Notification
    - More details stored in build artifacts.



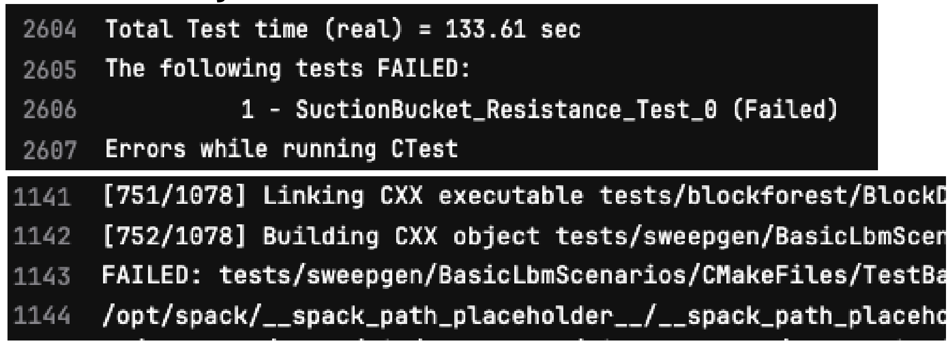
Status	Pipeline	Created by	Stages
Passed 02:55:36 2 hours ago	Fix clang-tidy error: Move deleted mem... #84254 795 9bedc86f latest merge request		
Passed 00:35:46 1 day ago	Fix clang-tidy error: Move deleted mem... #84186 behzad/301-core-cleanup 9bedc86f latest branch		
Failed 00:37:54 1 day ago	Remove std::regex and getpid wrappers #84181 behzad/301-core-cleanup 4e4e2d3b branch		Stage: Code Quality Failed jobs clang-tidy python-static-checks
Passed 01:23:46 2 days ago	Remove filesystem wrapper #84167 behzad/301-core-cleanup 42d0255b branch		
Skipped	Avoid division by zero [skip CI] #84095 suction_bucket 2d27ad7a latest branch		

TestSuite-Job-Passed



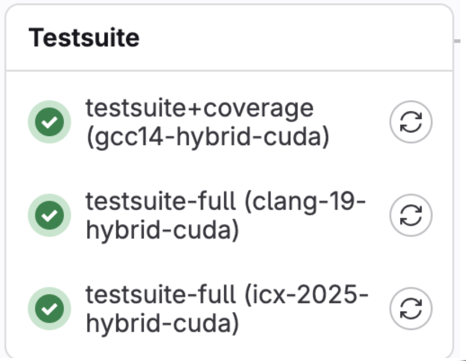
```
100% tests passed, 0 tests failed out of 408
Label Time Summary:
walberla::blockforest             =   9.04 sec*proc (16 tests)
walberla::boundary                =   1.98 sec*proc (4 tests)
walberla::core                    =  22.60 sec*proc (64 tests)
walberla::domain_decomposition    =   0.59 sec*proc (3 tests)
walberla::executiontree           =   0.11 sec*proc (1 test)
walberla::fft                     =   3.34 sec*proc (4 tests)
walberla::field                   =  13.44 sec*proc (26 tests)
```

TestSuite-Job-Failed



```
Total Test time (real) = 133.61 sec
The following tests FAILED:
          1 - SuctionBucket_Resistance_Test_0 (Failed)
Errors while running CTest
```

```
[751/1078] Linking CXX executable tests/blockforest/BlockD
[752/1078] Building CXX object tests/sweepgen/BasicLbmScen
FAILED: tests/sweepgen/BasicLbmScenarios/CMakeFiles/TestBa
/opt/spack/__spack_path_placeholder__/__spack_path_placeho
```

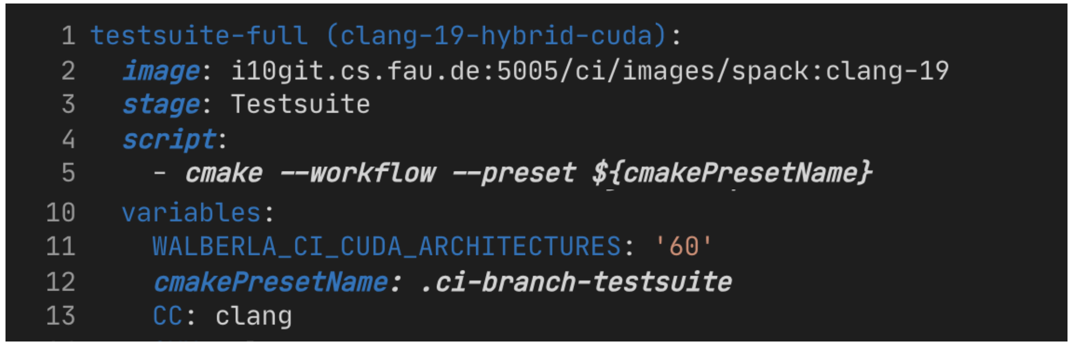


Testsuite

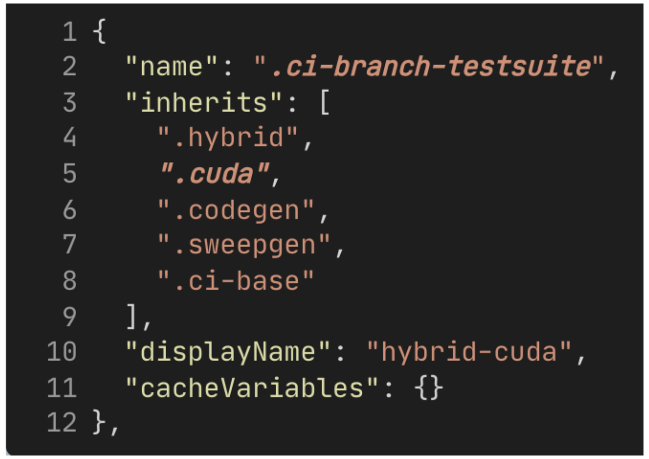
- testsuite+coverage (gcc14-hybrid-cuda)
- testsuite-full (clang-19-hybrid-cuda)
- testsuite-full (icx-2025-hybrid-cuda)

CMake Presets & Fragments

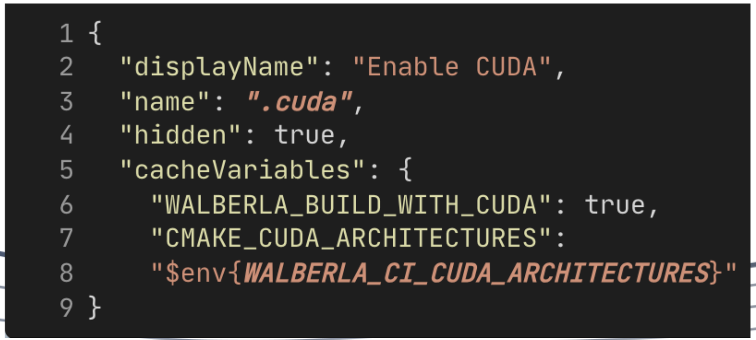
- waLBerla uses CMake Presets
  - to specify the **build configuration**
  - **Alternative** to
    - **`ccmake`**
    - passing **`-W<cacheVariable>`**
  - Clean build config definition
    - Allows generation of CMakePresets**.json**
    - Eases **Reproducibility**
- We define **common configurations** as "**Fragments**"
  - Reduces code duplications in presets
  - Fasilitates cleaner Presets
  - Fragment features include:
    - **Enabling external libraries** like OpenMesh, Metis, etc.
    - **Setting the build type**; e.g., Release, Debug
    - **Choosing your build tool**; e.g., Ninja, Unix Make
    - **Enabling support** for **GPU**, Python, MPI, and more



```yaml
testsuite-full (clang-19-hybrid-cuda):
  image: i10git.cs.fau.de:5005/ci/images/spack:clang-19
  stage: Testsuite
  script:
    - cmake --workflow --preset ${cmakePresetName}
  variables:
    WALBERLA_CI_CUDA_ARCHITECTURES: '60'
    cmakePresetName: .ci-branch-testsuite
    CC: clang
```



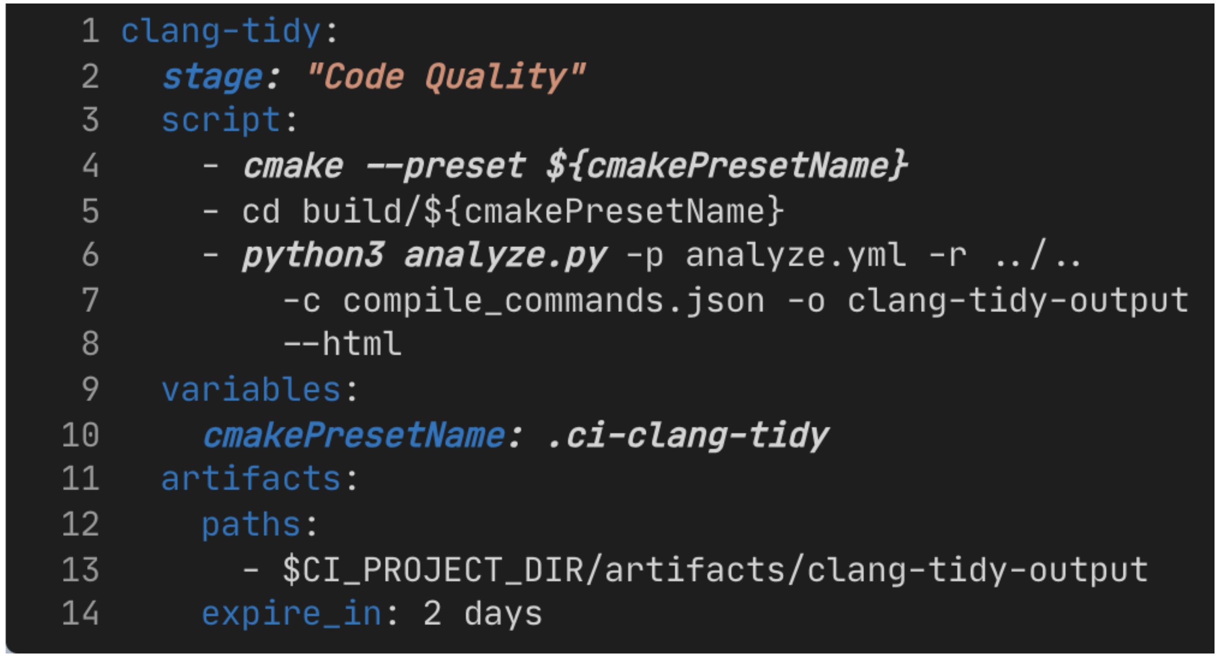
```json
{
  "name": ".ci-branch-testsuite",
  "inherits": [
    ".hybrid",
    ".cuda",
    ".codegen",
    ".sweepgen",
    ".ci-base"
  ],
  "displayName": "hybrid-cuda",
  "cacheVariables": {}
},
```



```json
{
  "displayName": "Enable CUDA",
  "name": ".cuda",
  "hidden": true,
  "cacheVariables": {
    "WALBERLA_BUILD_WITH_CUDA": true,
    "CMAKE_CUDA_ARCHITECTURES":
    "$env{WALBERLA_CI_CUDA_ARCHITECTURES}"
}
```

CI-Stage – **Code Quality**

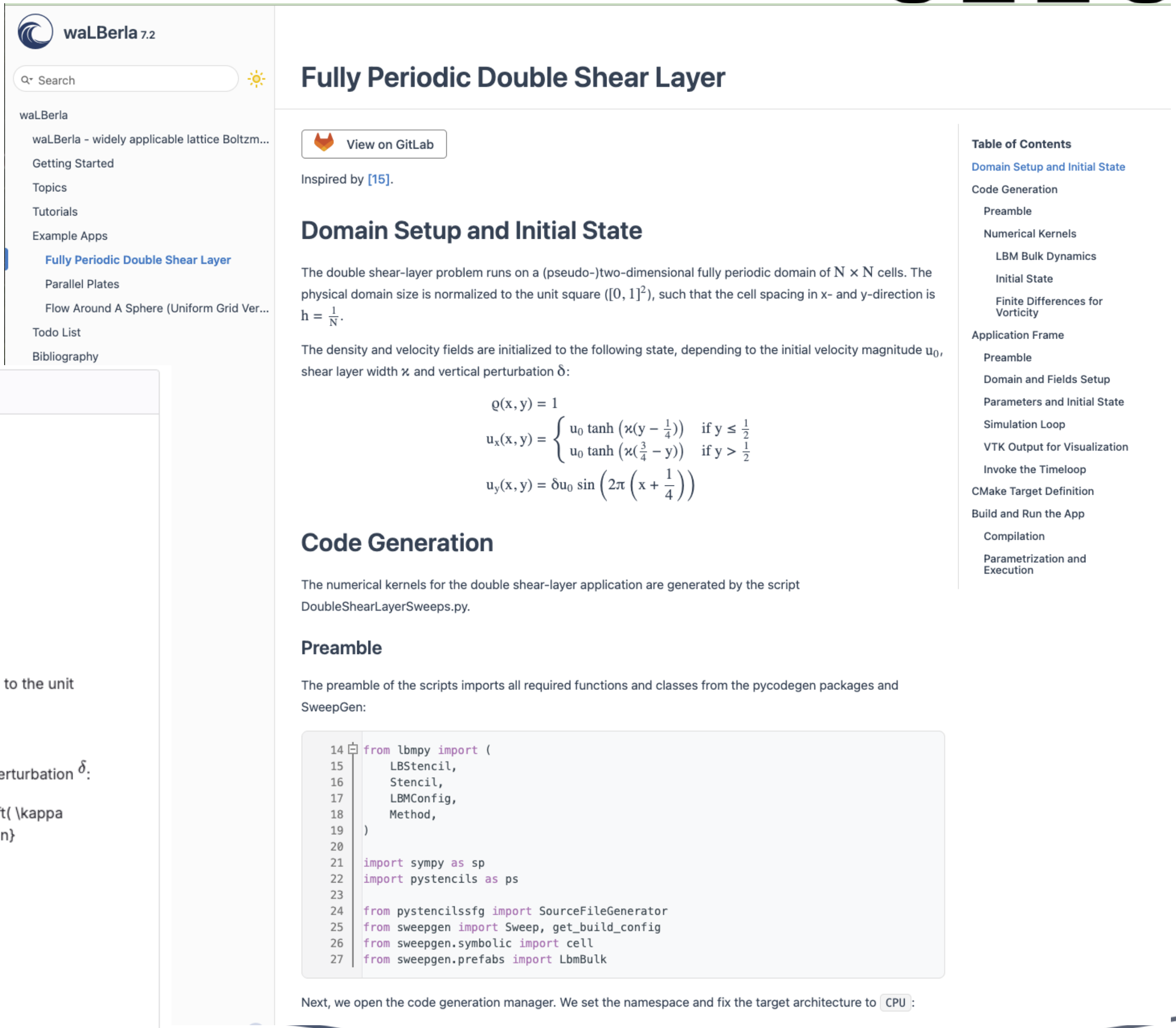
- **Statis Analysis Tests**
  - **Clang-tidy** for C++ **linting**
    - Helped to modernize waLBerla to C++20
  - We use own clang-tidy setup
  - Linting for python codes
    - Flake8 + MyPy for python linting
    - Executed with Cmake and Nox



```yaml
clang-tidy:
  stage: "Code Quality"
  script:
    - cmake --preset ${cmakePresetName}
    - cd build/${cmakePresetName}
    - python3 analyze.py -p analyze.yml -r ../..
        -c compile_commands.json -o clang-tidy-output
        --html
  variables:
    cmakePresetName: .ci-clang-tidy
  artifacts:
    paths:
      - $CI_PROJECT_DIR/artifacts/clang-tidy-output
    expire_in: 2 days
```

CI-Stage – **Documentation**

- Build in every pipeline and stored as artifact
- **Updated in deploy stage** (Stable branch)
  - **Hosted via GitLab Pages**
- Uses **Doxygen and Markdown** as sources
  - Can use **README.md** files for define pages



waLBerla 7.2

Search

waLBerla
- waLBerla - widely applicable lattice Boltzm...
- Getting Started
- Topics
- Tutorials
- Example Apps
  - Fully Periodic Double Shear Layer
  - Parallel Plates
  - Flow Around A Sphere (Uniform Grid Ver...
- Todo List
- Bibliography

Fully Periodic Double Shear Layer

View on GitLab

Inspired by [15].

Domain Setup and Initial State

The double shear-layer problem runs on a (pseudo-)two-dimensional fully periodic domain of $N \times N$ cells. The physical domain size is normalized to the unit square ($[0, 1]^2$), such that the cell spacing in x- and y-direction is $h = \frac{1}{N}$.

The density and velocity fields are initialized to the following state, depending to the initial velocity magnitude u_0, shear layer width $\varkappa$ and vertical perturbation δ:

$$\begin{aligned} \varrho(x, y) &= 1 \\ u_x(x, y) &= \begin{cases} u_0 \tanh\left(\varkappa(y - \frac{1}{4})\right) & \text{if } y \le \frac{1}{2} \\ u_0 \tanh\left(\varkappa(\frac{3}{4} - y)\right) & \text{if } y > \frac{1}{2} \end{cases} \\ u_y(x, y) &= \delta u_0 \sin\left(2\pi\left(x + \frac{1}{4}\right)\right) \end{aligned}$$

Code Generation

The numerical kernels for the double shear-layer application are generated by the script DoubleShearLayerSweeps.py.

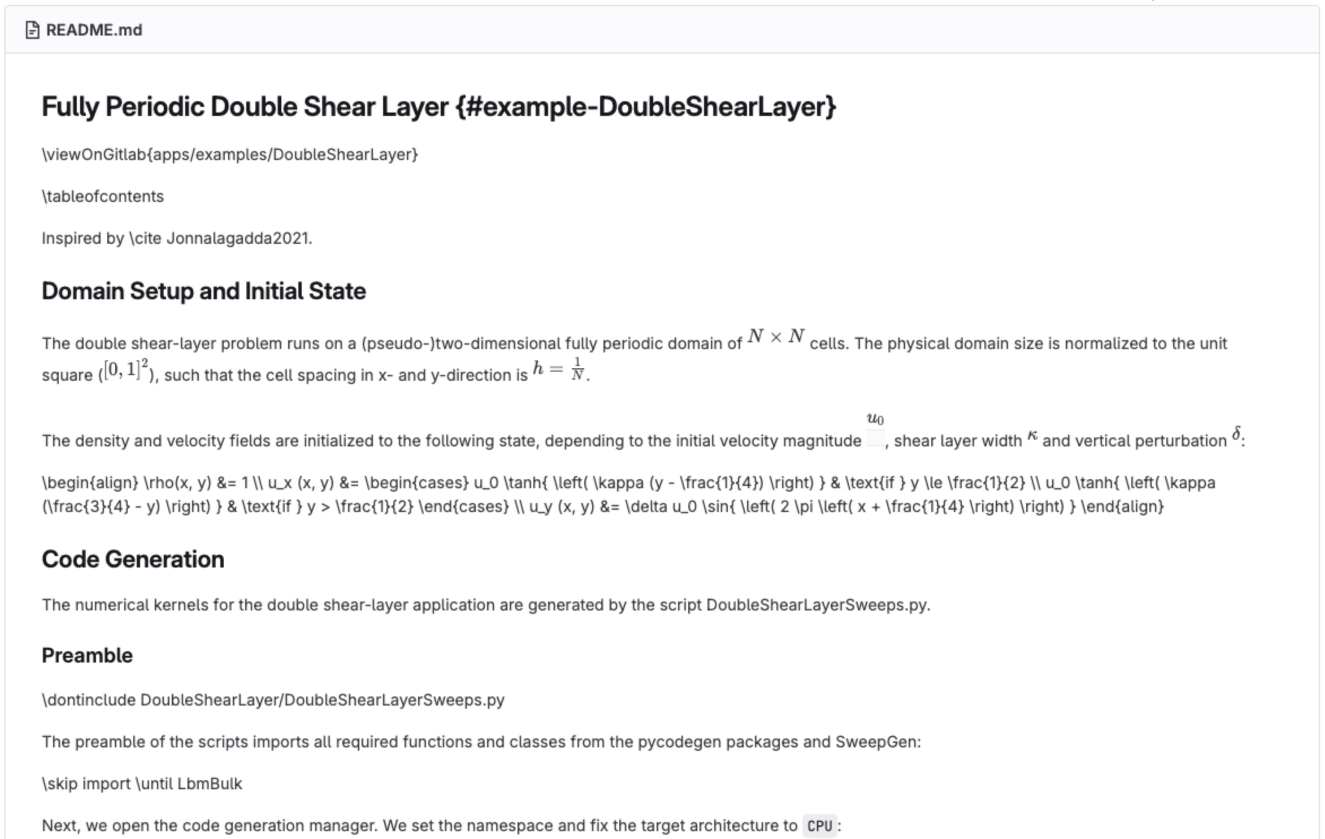
Preamble

The preamble of the scripts imports all required functions and classes from the pycodegen packages and SweepGen:

```python
from lbmpy import (
    LBStencil,
    Stencil,
    LBMConfig,
    Method,
)

import sympy as sp
import pystencils as ps

from pystencilssfg import SourceFileGenerator
from sweepgen import Sweep, get_build_config
from sweepgen.symbolic import cell
from sweepgen.prefabs import LbmBulk
```

Next, we open the code generation manager. We set the namespace and fix the target architecture to `CPU`:

Table of Contents
- Domain Setup and Initial State
- Code Generation
  - Preamble
  - Numerical Kernels
    - LBM Bulk Dynamics
    - Initial State
    - Finite Differences for Vorticity
- Application Frame
  - Preamble
  - Domain and Fields Setup
  - Parameters and Initial State
  - Simulation Loop
  - VTK Output for Visualization
  - Invoke the Timeloop
- CMake Target Definition
- Build and Run the App
  - Compilation
  - Parametrization and Execution



README.md

Fully Periodic Double Shear Layer {#example-DoubleShearLayer}

\viewOnGitlab{apps/examples/DoubleShearLayer}

\tableofcontents

Inspired by \cite Jonnalagadda2021.

Domain Setup and Initial State

The double shear-layer problem runs on a (pseudo-)two-dimensional fully periodic domain of $N \times N$ cells. The physical domain size is normalized to the unit square ($[0, 1]^2$), such that the cell spacing in x- and y-direction is $h = \frac{1}{N}$.

The density and velocity fields are initialized to the following state, depending to the initial velocity magnitude u_0, shear layer width κ and vertical perturbation δ:

\begin{align} \rho(x, y) &= 1 \\ u_x (x, y) &= \begin{cases} u_0 \tanh{ \left(\kappa (y - \frac{1}{4}) \right) } & \text{if } y \le \frac{1}{2} \\ u_0 \tanh{ \left(\kappa (\frac{3}{4} - y) \right) } & \text{if } y > \frac{1}{2} \end{cases} \\ u_y (x, y) &= \delta u_0 \sin{ \left(2 \pi \left(x + \frac{1}{4} \right) \right) } \end{align}

Code Generation

The numerical kernels for the double shear-layer application are generated by the script DoubleShearLayerSweeps.py.

Preamble

\dontinclude DoubleShearLayer/DoubleShearLayerSweeps.py

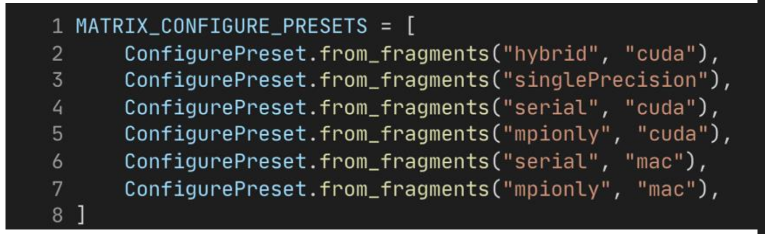
The preamble of the scripts imports all required functions and classes from the pycodegen packages and SweepGen:

\skip import \until LbmBulk

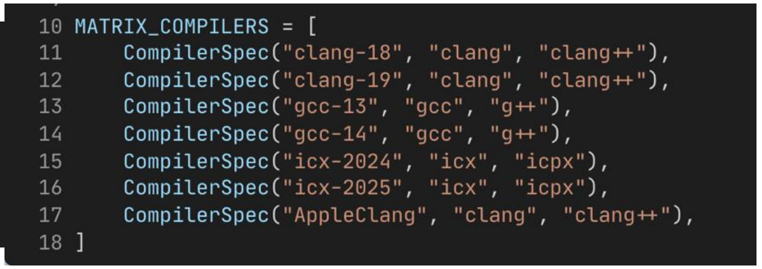
Next, we open the code generation manager. We set the namespace and fix the target architecture to `CPU`:

CI-Stage – **Testmatrix**

- **Extensive** test set
  - **Accaptence** and **Regression** for the **Stable** branch
  - More **compilers** and different **versions**
  - **Variation of build parameter** per compiler

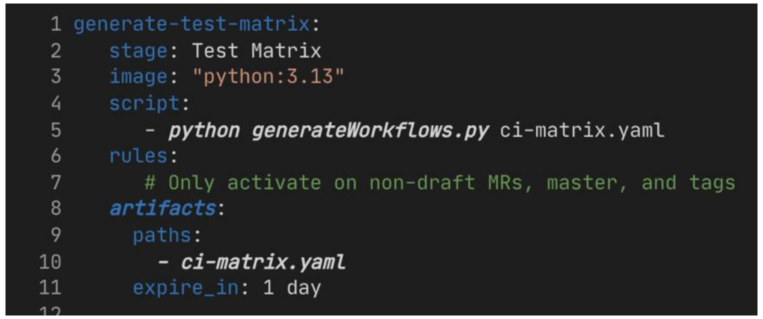


```
MATRIX_CONFIGURE_PRESETS = [
    ConfigurePreset.from_fragments("hybrid", "cuda"),
    ConfigurePreset.from_fragments("singlePrecision"),
    ConfigurePreset.from_fragments("serial", "cuda"),
    ConfigurePreset.from_fragments("mpionly", "cuda"),
    ConfigurePreset.from_fragments("serial", "mac"),
    ConfigurePreset.from_fragments("mpionly", "mac"),
]
```



```
MATRIX_COMPILERS = [
    CompilerSpec("clang-18", "clang", "clang++"),
    CompilerSpec("clang-19", "clang", "clang++"),
    CompilerSpec("gcc-13", "gcc", "g++"),
    CompilerSpec("gcc-14", "gcc", "g++"),
    CompilerSpec("icx-2024", "icx", "icpx"),
    CompilerSpec("icx-2025", "icx", "icpx"),
    CompilerSpec("AppleClang", "clang", "clang++"),
]
```

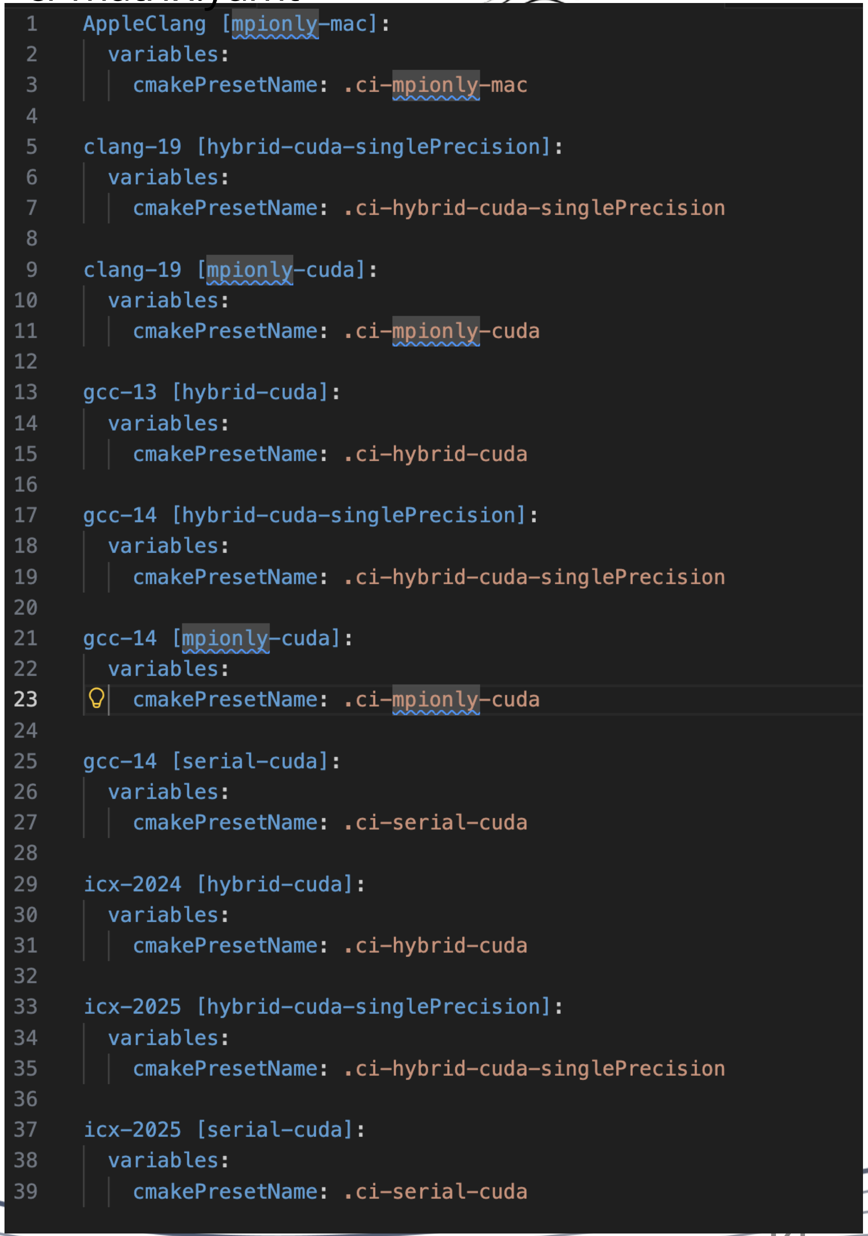
- Textmatrix Pipeline is **generated** as ci-matrix.yaml
  - Each job defines **specialized configuration Preset**
    - Presets generated as well
    - Clean association



```
generate-test-matrix:
  stage: Test Matrix
  image: "python:3.13"
  script:
    - python generateWorkflows.py ci-matrix.yaml
  rules:
    # Only activate on non-draft MRs, master, and tags
  artifacts:
    paths:
      - ci-matrix.yaml
    expire_in: 1 day
```

- **NOT** GitLab Matrix expression
  - Creates "full MxN-matrices"
- We don't want to test every combination

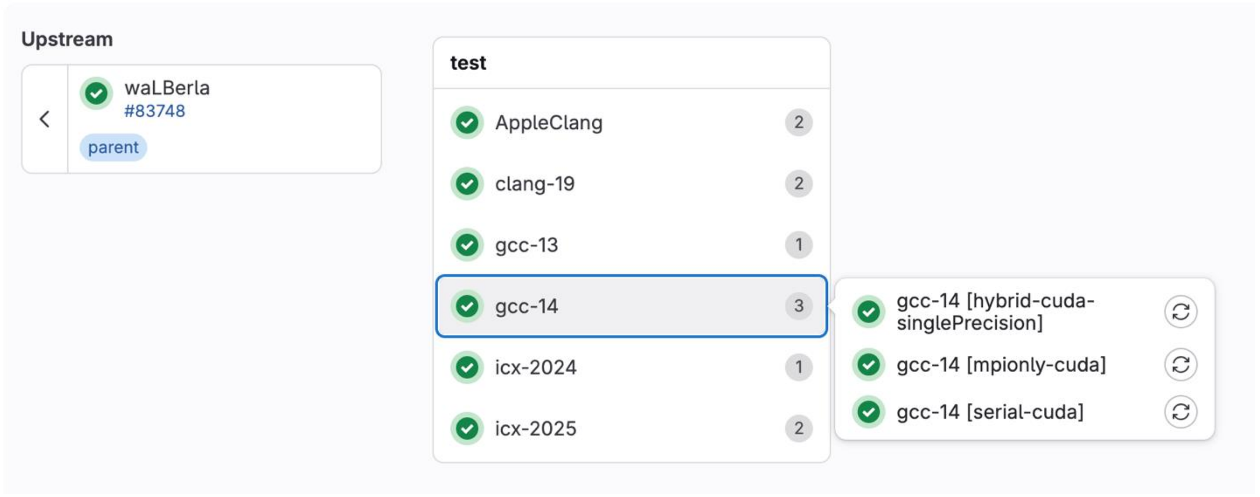
ci-matrix.yaml

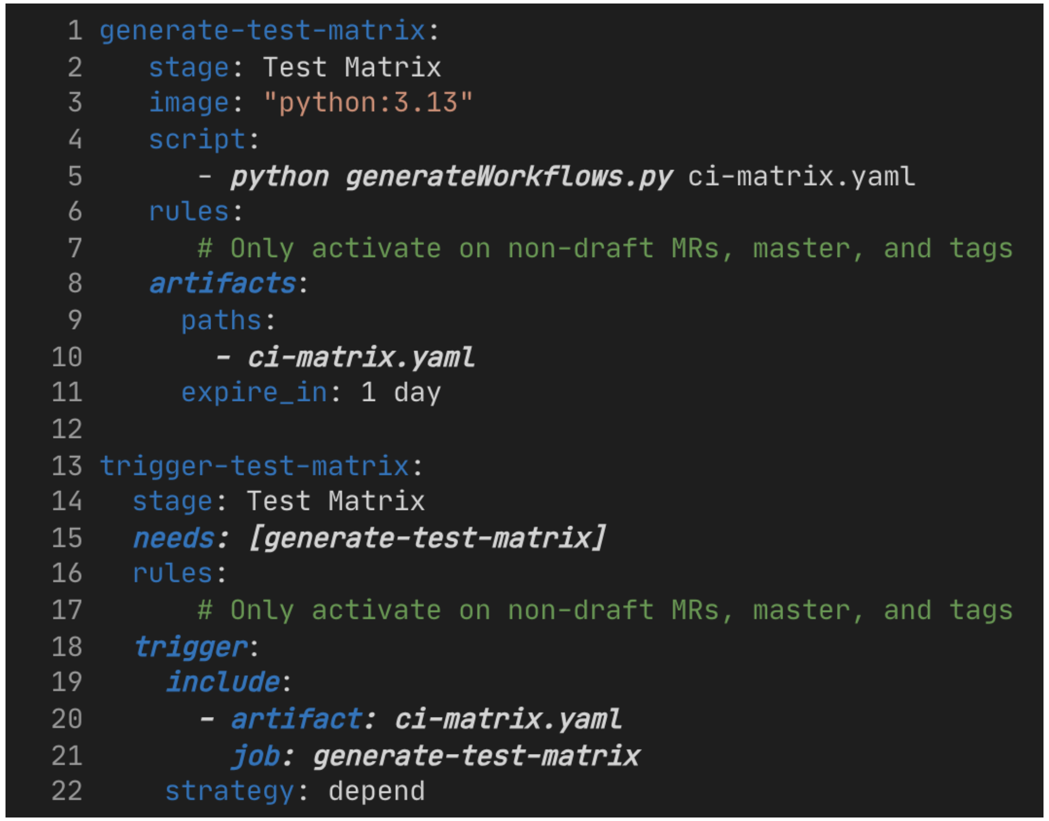


```
AppleClang [mpionly-mac]:
  variables:
    cmakePresetName: .ci-mpionly-mac

clang-19 [hybrid-cuda-singlePrecision]:
  variables:
    cmakePresetName: .ci-hybrid-cuda-singlePrecision

clang-19 [mpionly-cuda]:
  variables:
    cmakePresetName: .ci-mpionly-cuda

gcc-13 [hybrid-cuda]:
  variables:
    cmakePresetName: .ci-hybrid-cuda

gcc-14 [hybrid-cuda-singlePrecision]:
  variables:
    cmakePresetName: .ci-hybrid-cuda-singlePrecision

gcc-14 [mpionly-cuda]:
  variables:
    cmakePresetName: .ci-mpionly-cuda

gcc-14 [serial-cuda]:
  variables:
    cmakePresetName: .ci-serial-cuda

icx-2024 [hybrid-cuda]:
  variables:
    cmakePresetName: .ci-hybrid-cuda

icx-2025 [hybrid-cuda-singlePrecision]:
  variables:
    cmakePresetName: .ci-hybrid-cuda-singlePrecision

icx-2025 [serial-cuda]:
  variables:
    cmakePresetName: .ci-serial-cuda
```

CI-Stage – **Testmatrix**

- Realized as **Downstream Pipeline**
  - Downsteam pipeline **created during Upstream execution**
    - Pipeline can be adjusted to needs on demand
  - Cosmetical Advantage
    - Tidy representation in GitLab GUI
    - GUI groups similar jobs together
    - Prevents cluttering of main CI file
  - <u>Note</u>: Can trigger pipelines from external repositories



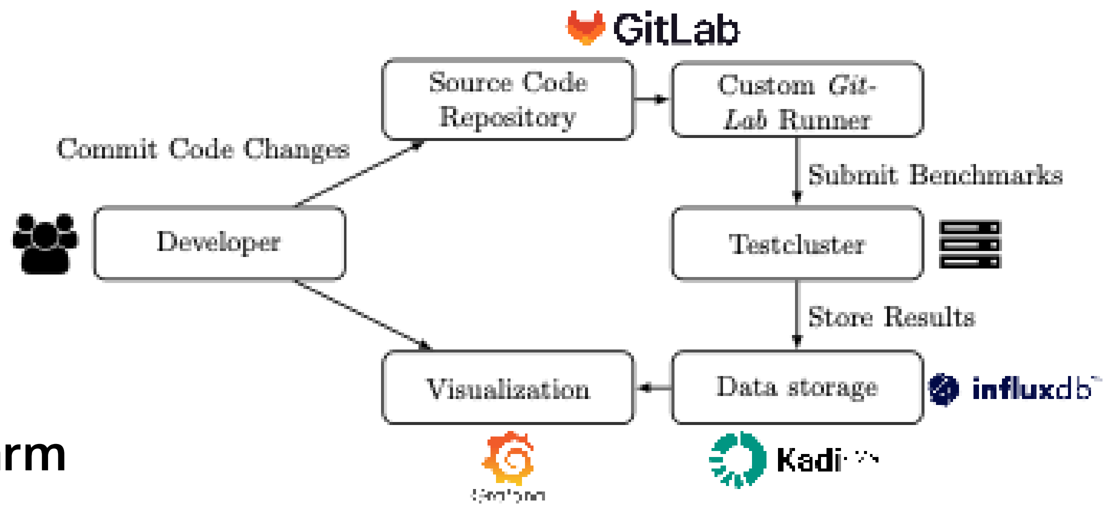


```yaml
generate-test-matrix:
   stage: Test Matrix
   image: "python:3.13"
   script:
      - python generateWorkflows.py ci-matrix.yaml
   rules:
      # Only activate on non-draft MRs, master, and tags
   artifacts:
     paths:
       - ci-matrix.yaml
     expire_in: 1 day

trigger-test-matrix:
  stage: Test Matrix
  needs: [generate-test-matrix]
  rules:
      # Only activate on non-draft MRs, master, and tags
  trigger:
    include:
      - artifact: ci-matrix.yaml
        job: generate-test-matrix
    strategy: depend
```

CI-Stage – **benchmark**

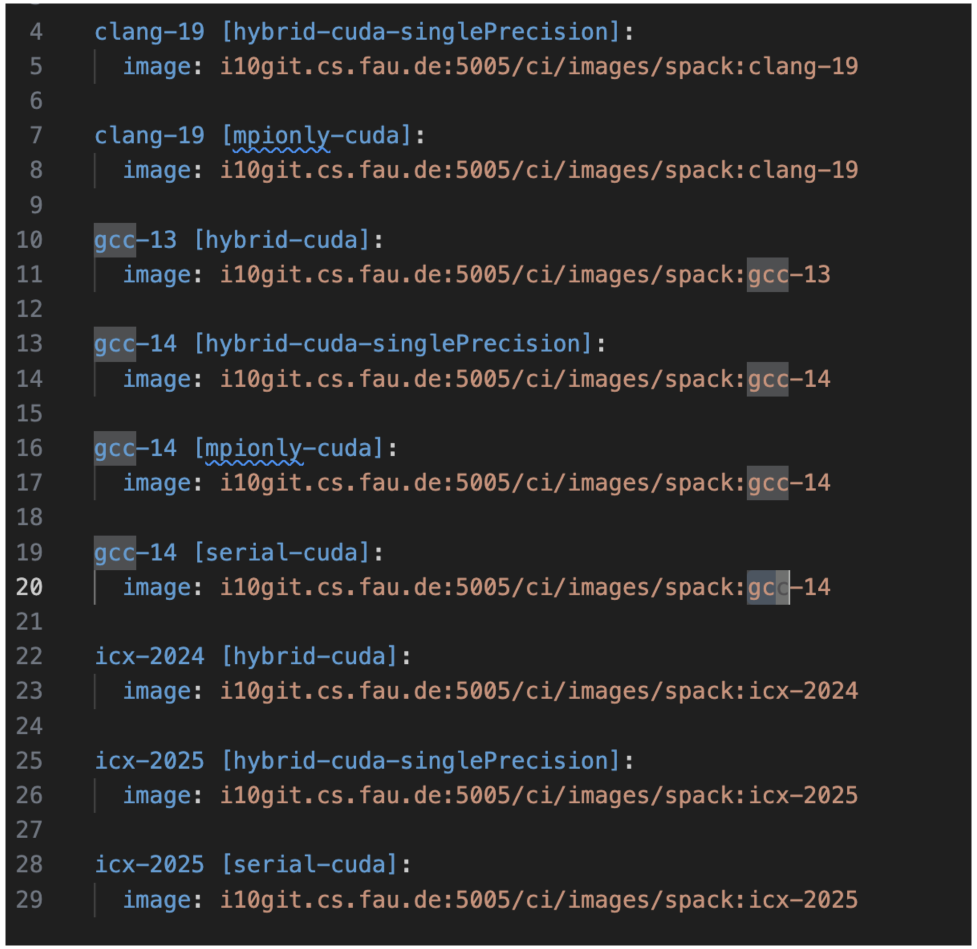
- waLBerla **CI triggers** waLBerla **CB pipeline** hosted at **external repository**
  - Only for **Stable branch**
  - Used for **Performance Regression**
- **CB pipeline** hosted at NHR@FAU
  - Jobs run on NHR **testcluster.**
    - Large architecutres variaty for CPU and GPU
    - Every node is different
      - ➔**No multi node / scaling tests**
  - NHR repository provides **HPC nodes** as **Runner swarm**
- Other HPC centers support HPC resources as Git Runners as well
- More information see:

  Alt, Christoph, et al. "A Continuous Benchmarking Infrastructure for High-Performance Computing Applications." *International Journal of Parallel, Emergent and Distributed Systems* 39, no. 4 (2024): 501–23.



CI-Images

- A **Image** is the **blueprint** for a container.
- Our CI pipeline is intended to **validate code changes**, **NOT environmental changes**
    - -> We run our CI jobs inside a **container**
        - Portable **Build- and Run Environment**
        - **Reproducible** (FAIR)
        - **Interoperatable** (FAIR)
        - For every CI job **waLBerla is freshly build**
            - Environment is reusable
- Almost every CI job needs a **different environment**
    - Need to manage **multiple images**
    - Images hosted in **GitLab Container Registry**
        - Like own "Docker Hub" within GitLab
        - Works for all OCI conform containers
        - CI **pulls** images directly from Container Registry
            - Accessible (FAIR)



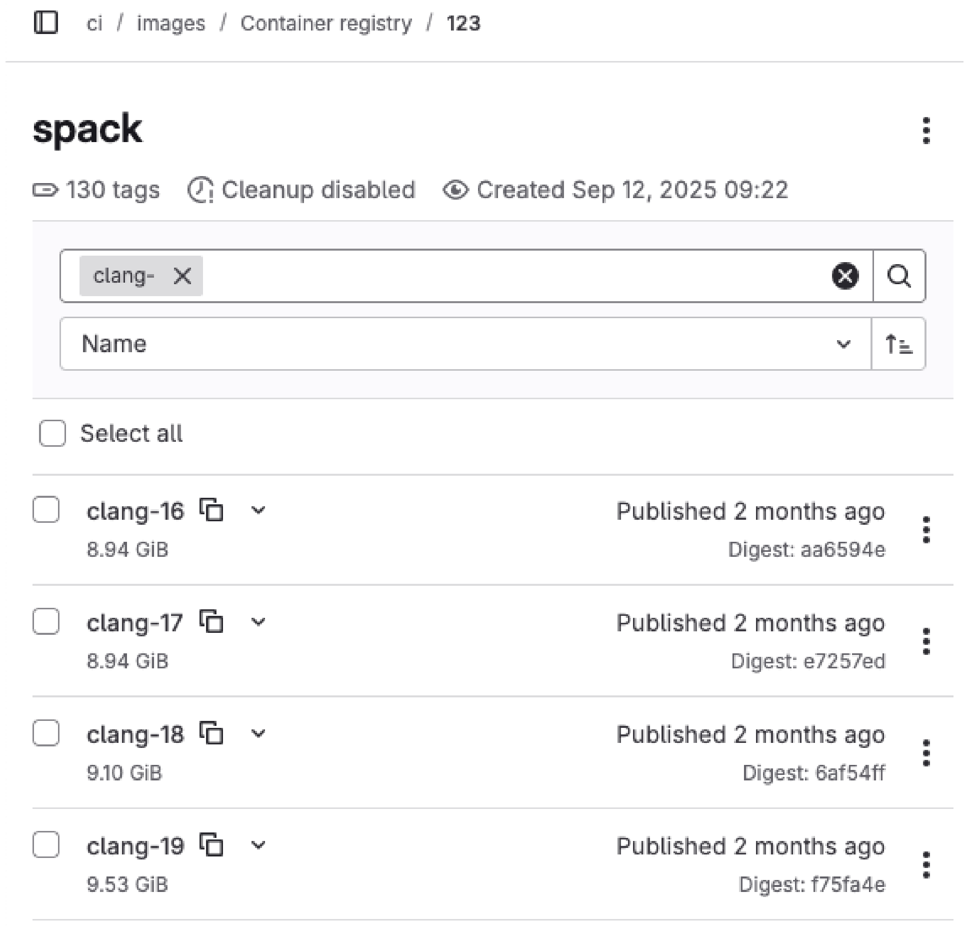
```
clang-19 [hybrid-cuda-singlePrecision]:
  image: i10git.cs.fau.de:5005/ci/images/spack:clang-19

clang-19 [mpionly-cuda]:
  image: i10git.cs.fau.de:5005/ci/images/spack:clang-19

gcc-13 [hybrid-cuda]:
  image: i10git.cs.fau.de:5005/ci/images/spack:gcc-13

gcc-14 [hybrid-cuda-singlePrecision]:
  image: i10git.cs.fau.de:5005/ci/images/spack:gcc-14

gcc-14 [mpionly-cuda]:
  image: i10git.cs.fau.de:5005/ci/images/spack:gcc-14

gcc-14 [serial-cuda]:
  image: i10git.cs.fau.de:5005/ci/images/spack:gcc-14

icx-2024 [hybrid-cuda]:
  image: i10git.cs.fau.de:5005/ci/images/spack:icx-2024

icx-2025 [hybrid-cuda-singlePrecision]:
  image: i10git.cs.fau.de:5005/ci/images/spack:icx-2025

icx-2025 [serial-cuda]:
  image: i10git.cs.fau.de:5005/ci/images/spack:icx-2025
```

Container Registry at LSS

- Containers maintained in an own repository
  - **i10git.cs.fau.de/ci/images**
  - Containers **created** directly in Repositorie's CI
    - Each container change gets an own version
      - **Findable** (**F**AIR)
  - Registry is **used for all C++ Frameworks** at LSS

- We use **Spack** to
  - **Manage** the Container Registry
  - Define the **Environment**
  - **Install Dependencies** (Compiles, Packages)
- Environments are **clearly defined** using Spack

Note: The Frameworks cannot be built with Spack not (yet)



ci / images / Container registry / 123

spack

130 tags · Cleanup disabled · Created Sep 12, 2025 09:22

clang-16	8.94 GiB	Published 2 months ago · Digest: aa6594e
clang-17	8.94 GiB	Published 2 months ago · Digest: e7257ed
clang-18	9.10 GiB	Published 2 months ago · Digest: 6af54ff
clang-19	9.53 GiB	Published 2 months ago · Digest: f75fa4e

CI-Images – **3 Steps**

- Creating waLBerla build environment in 3 CI-Stages:
  1. **Build BaseContainer**
     - Start from GPU vendor image
     - Prepare Spack
  2. **Install Dependencies**
     - Using **spack binary cache**
  3. **Create CI-Images**
     - **Generate** spack environments
     - Create Image **from spack environment**

```
spack-23-dockerimage:
  script:
    - docker login
    - docker build -f Dockerfile-Spack-23 .
    - docker push i10git.cs.fau.de:5005/ci/images/spack-23
  image: docker:latest
  stage: spack-base
  tags:
    - docker-docker
```

```
FROM nvidia/cuda:12.8.1-devel-ubuntu24.04
# Install minimal requirements
# Download Spack
RUN git clone https://github.com/spack/spack.git
# Make sure Spack is sourced in containers
SHELL ["spack-docker-shell"]
# Add GitLab Container registry to spack mirrors
RUN spack mirror add i10git
oci://i10git.cs.fau.de:5005/ci/images/spack
ENTRYPOINT ["spack-env"]
CMD ["spack-interactive-shell"]
```

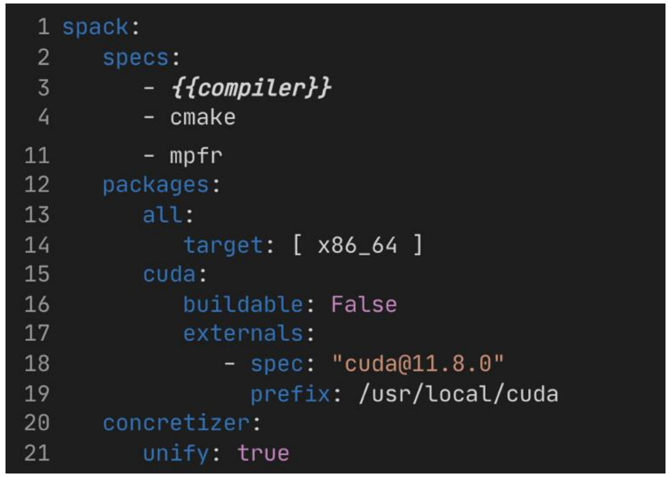
```
gcc:
  script:
    - env
    - spack install gcc@14.2.0 target=x86_64
    - spack buildcache push --unsigned i10git gcc@14.2.0
    - spack install gcc@12.4.0 target=x86_64
    - spack buildcache push --unsigned i10git gcc@12.4.0
    - spack buildcache update-index i10git
  image: i10git.cs.fau.de:5005/ci/images/spack-23
  stage: compiler-and-packages
  tags:
    - docker
```

```
packages:
  script:
    - spack install ccache mpfr ... target=x86_64
    - spack buildcache push i10git ccache mpfr ...
    - spack install metis+int64+real64 target=x86_64
    - spack install petsc+int64~metis+mumps target=x86_64
    ...
    - spack buildcache push i10git metis petsc ...
    - spack buildcache update-index i10git
  image: i10git.cs.fau.de:5005/ci/images/spack-23
  stage: compiler-and-packages
  tags:
    - docker
```

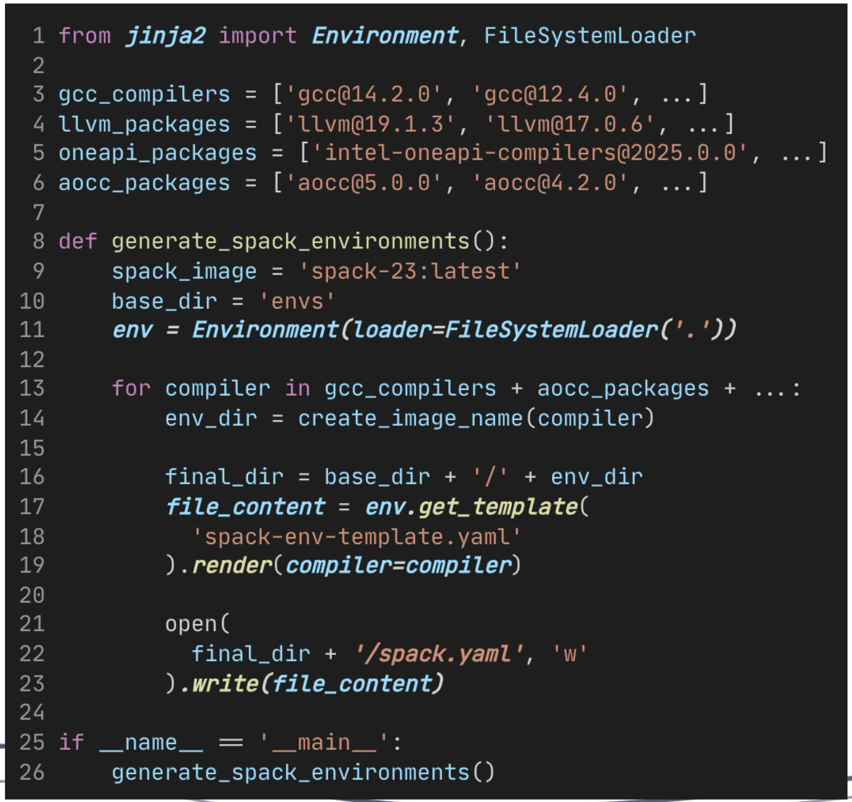
```
envs:
  script:
    - python3 ./generate_spack_environments.py
    - |-
      for ENV_DIR in created_environments
      do
        cd $ENV_DIR
        . /opt/spack/share/spack/setup-env.sh
        spack env activate .
        spack install
        # Append BaseContainer by environment
        spack buildcache push \\
        --base-image <BaseContainer> --tag $ENV_DIR \\
        i10git
        cd ..
      done
  image: i10git.cs.fau.de:5005/ci/images/spack-23
  stage: dev-images
  tags:
    - docker
```

CI-Image – **Generation**

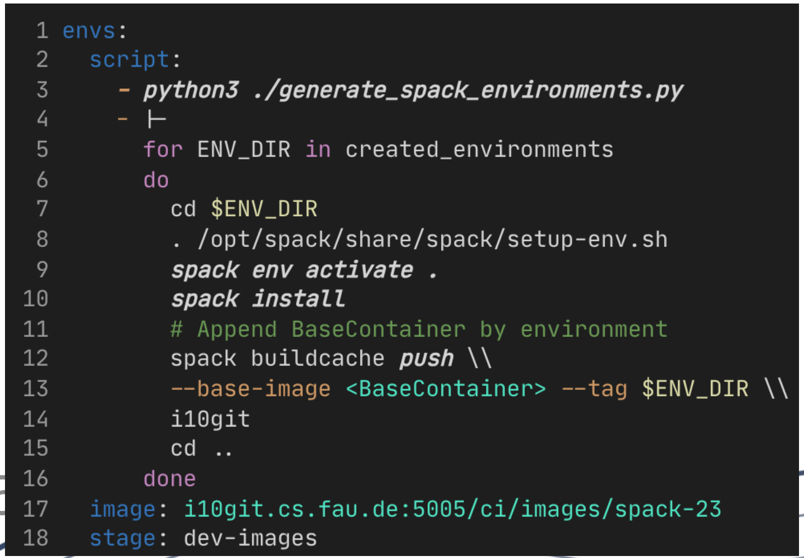
- Valid spack **environment file**
  - Mark required changes with **macros**
- Python environment **generation script**
  - **Define variations** in environment spec
  - Use **Jinja2 Templates** to **override macros**
  - Create environment **folder per image**
    - Each folder contains generated environment file
- CI loops over environment folders to create images



```yaml
spack:
  specs:
    - {{compiler}}
    - cmake

    - mpfr
  packages:
    all:
      target: [ x86_64 ]
    cuda:
      buildable: False
      externals:
        - spec: "cuda@11.8.0"
          prefix: /usr/local/cuda
  concretizer:
    unify: true
```



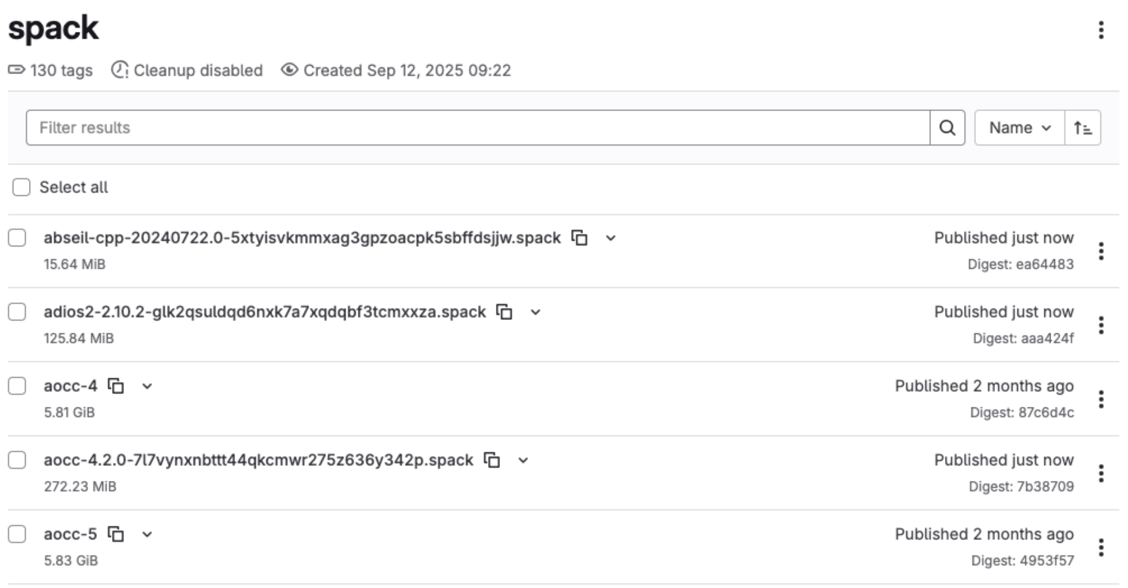
```python
from jinja2 import Environment, FileSystemLoader

gcc_compilers = ['gcc@14.2.0', 'gcc@12.4.0', ...]
llvm_packages = ['llvm@19.1.3', 'llvm@17.0.6', ...]
oneapi_packages = ['intel-oneapi-compilers@2025.0.0', ...]
aocc_packages = ['aocc@5.0.0', 'aocc@4.2.0', ...]

def generate_spack_environments():
    spack_image = 'spack-23:latest'
    base_dir = 'envs'
    env = Environment(loader=FileSystemLoader('.'))

    for compiler in gcc_compilers + aocc_packages + ...:
        env_dir = create_image_name(compiler)

        final_dir = base_dir + '/' + env_dir
        file_content = env.get_template(
          'spack-env-template.yaml'
        ).render(compiler=compiler)

        open(
          final_dir + '/spack.yaml', 'w'
        ).write(file_content)

if __name__ == '__main__':
    generate_spack_environments()
```



```yaml
envs:
  script:
    - python3 ./generate_spack_environments.py
    - |-
      for ENV_DIR in created_environments
      do
        cd $ENV_DIR
        . /opt/spack/share/spack/setup-env.sh
        spack env activate .
        spack install
        # Append BaseContainer by environment
        spack buildcache push \\
        --base-image <BaseContainer> --tag $ENV_DIR \\
        i10git
        cd ..
      done
  image: i10git.cs.fau.de:5005/ci/images/spack-23
  stage: dev-images
```

CI Images – **Spack Buildcache**

- **Accellerate** Spack installations
  - Already build binaries can be **stored in Build Cache**
  - **Lookup** binaries in cache before each build
    - If available binary **matches the spec**:
      - ➔Binary is **pulled instead of build**
- GitLab Container Registry can host Build Caches
  - **Same Registry** for CI-Images and Build Caches
- Note: Public build caches exist:
  - cache.spack.io



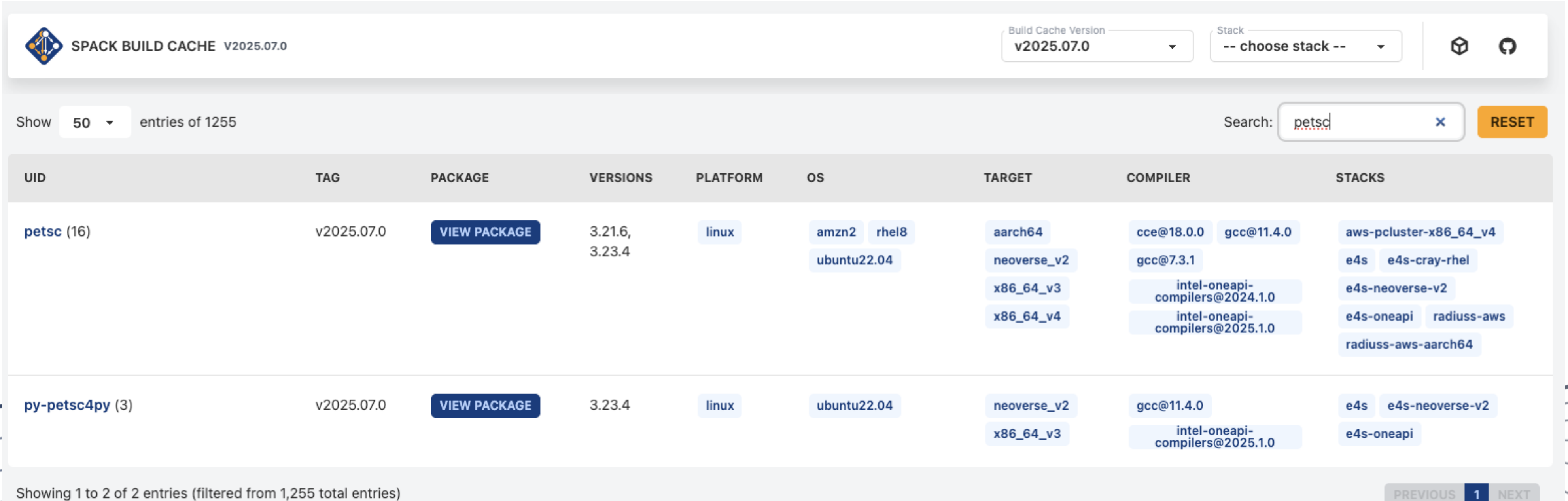
spack

130 tags · Cleanup disabled · Created Sep 12, 2025 09:22

Filter results · Name

Select all

Name	Size	Published	Digest
abseil-cpp-20240722.0-5xtyisvkmmxag3gpzoacpk5sbffdsjjw.spack	15.64 MiB	Published just now	Digest: ea64483
adios2-2.10.2-glk2qsuldqd6nxk7a7xqdqbf3tcmxxza.spack	125.84 MiB	Published just now	Digest: aaa424f
aocc-4	5.81 GiB	Published 2 months ago	Digest: 87c6d4c
aocc-4.2.0-7l7vynxnbttt44qkcmwr275z636y342p.spack	272.23 MiB	Published just now	Digest: 7b38709
aocc-5	5.83 GiB	Published 2 months ago	Digest: 4953f57



SPACK BUILD CACHE V2025.07.0

Build Cache Version: v2025.07.0 · Stack: -- choose stack --

Show 50 entries of 1255 · Search: petsc · RESET

UID	TAG	PACKAGE	VERSIONS	PLATFORM	OS	TARGET	COMPILER	STACKS
petsc (16)	v2025.07.0	VIEW PACKAGE	3.21.6, 3.23.4	linux	amzn2, rhel8, ubuntu22.04	aarch64, neoverse_v2, x86_64_v3, x86_64_v4	cce@18.0.0, gcc@11.4.0, gcc@7.3.1, intel-oneapi-compilers@2024.1.0, intel-oneapi-compilers@2025.1.0	aws-pcluster-x86_64_v4, e4s, e4s-cray-rhel, e4s-neoverse-v2, e4s-oneapi, radiuss-aws, radiuss-aws-aarch64
py-petsc4py (3)	v2025.07.0	VIEW PACKAGE	3.23.4	linux	ubuntu22.04	neoverse_v2, x86_64_v3	gcc@11.4.0, intel-oneapi-compilers@2025.1.0	e4s, e4s-neoverse-v2, e4s-oneapi

Showing 1 to 2 of 2 entries (filtered from 1,255 total entries)

PREVIOUS 1 NEXT

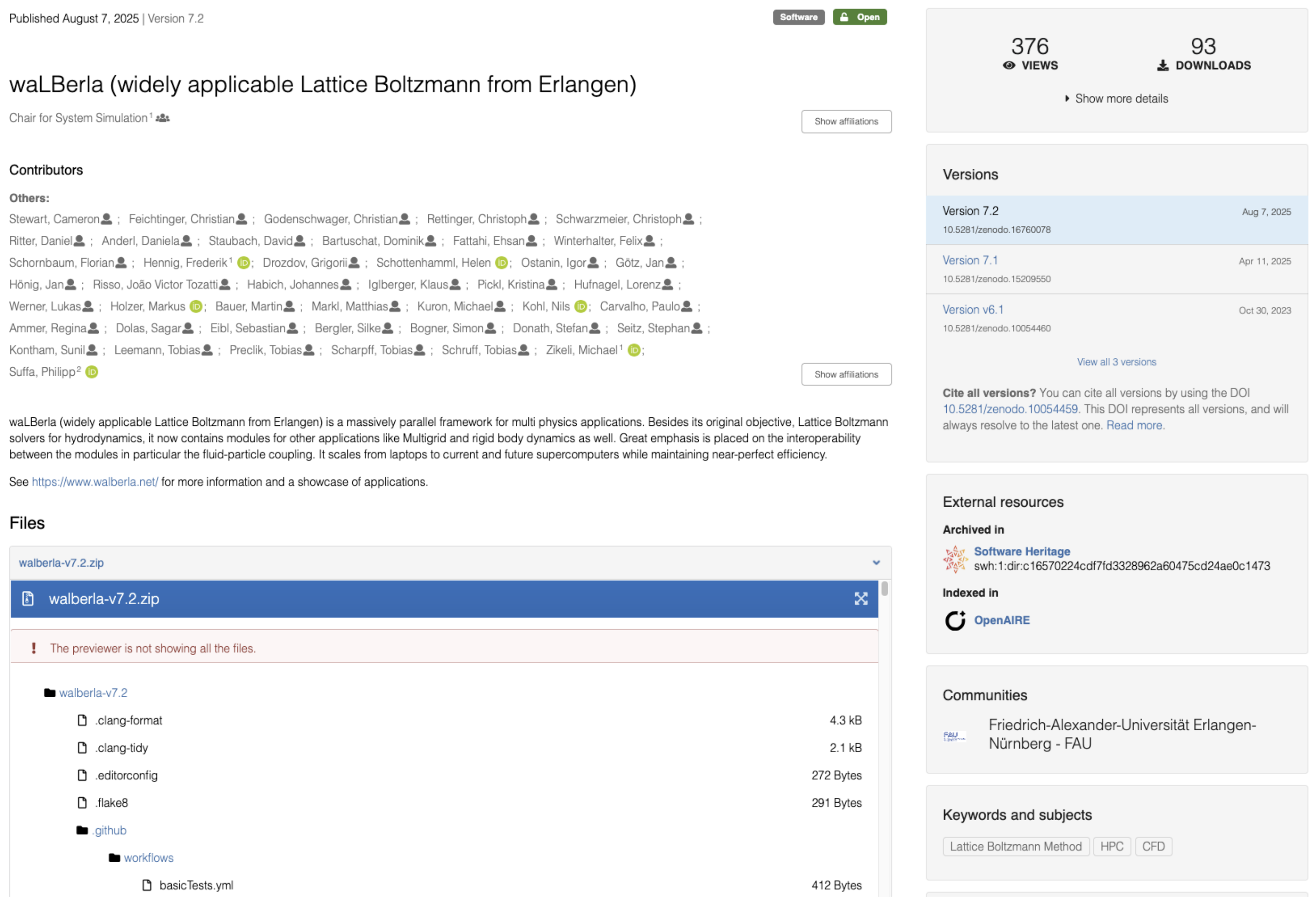
Deployment / Delivery

- **Current Deployment = Code Release**
  - **Manuel** Release via
    - GitLab Release for **Versioning**
    - **Zenodo** Publication for a **DOI**
      - Has hook for GitHub
  - **Automatic** Mirror to other Git Servers
    - For GitHub and EU Projects

- Binary Deployments
  - Container often not appropriate for HPC
  - Deploying a binary artefact is not enough
  - HPC codes need target optimizations

As hardware gets more powerful efficiency becomes even more important!

➔Usually want to **build directly on target system**



Published August 7, 2025 | Version 7.2

Software | Open

waLBerla (widely applicable Lattice Boltzmann from Erlangen)

Chair for System Simulation[1]

Show affiliations

Contributors

Others:

Stewart, Cameron ; Feichtinger, Christian ; Godenschwager, Christian ; Rettinger, Christoph ; Schwarzmeier, Christoph ; Ritter, Daniel ; Anderl, Daniela ; Staubach, David ; Bartuschat, Dominik ; Fattahi, Ehsan ; Winterhalter, Felix ; Schornbaum, Florian ; Hennig, Frederik[1] ; Drozdov, Grigorii ; Schottenhamml, Helen ; Ostanin, Igor ; Götz, Jan ; Hönig, Jan ; Risso, João Victor Tozatti ; Habich, Johannes ; Iglberger, Klaus ; Pickl, Kristina ; Hufnagel, Lorenz ; Werner, Lukas ; Holzer, Markus ; Bauer, Martin ; Markl, Matthias ; Kuron, Michael ; Kohl, Nils ; Carvalho, Paulo ; Ammer, Regina ; Dolas, Sagar ; Eibl, Sebastian ; Bergler, Silke ; Bogner, Simon ; Donath, Stefan ; Seitz, Stephan ; Kontham, Sunil ; Leemann, Tobias ; Preclik, Tobias ; Scharpff, Tobias ; Schruff, Tobias ; Zikeli, Michael[1] ; Suffa, Philipp[2]

Show affiliations

waLBerla (widely applicable Lattice Boltzmann from Erlangen) is a massively parallel framework for multi physics applications. Besides its original objective, Lattice Boltzmann solvers for hydrodynamics, it now contains modules for other applications like Multigrid and rigid body dynamics as well. Great emphasis is placed on the interoperability between the modules in particular the fluid-particle coupling. It scales from laptops to current and future supercomputers while maintaining near-perfect efficiency.

See https://www.walberla.net/ for more information and a showcase of applications.

Files

walberla-v7.2.zip

walberla-v7.2.zip

! The previewer is not showing all the files.

```
walberla-v7.2
    .clang-format          4.3 kB
    .clang-tidy            2.1 kB
    .editorconfig          272 Bytes
    .flake8                291 Bytes
    .github
        workflows
            basicTests.yml 412 Bytes
```

376 VIEWS | 93 DOWNLOADS

Show more details

Versions

Version 7.2 10.5281/zenodo.16760078	Aug 7, 2025
Version 7.1 10.5281/zenodo.15209550	Apr 11, 2025
Version v6.1 10.5281/zenodo.10054460	Oct 30, 2023

View all 3 versions

Cite all versions? You can cite all versions by using the DOI 10.5281/zenodo.10054459. This DOI represents all versions, and will always resolve to the latest one. Read more.

External resources

Archived in

Software Heritage
swh:1:dir:c16570224cdf7fd3328962a60475cd24ae0c1473

Indexed in

OpenAIRE

Communities

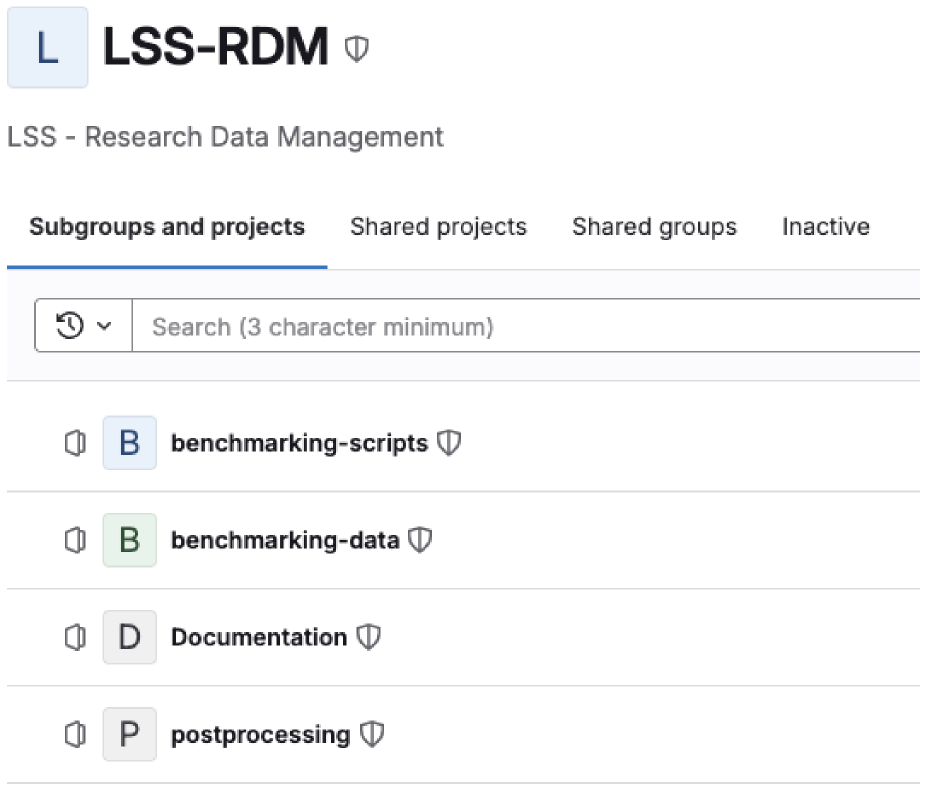
Friedrich-Alexander-Universität Erlangen-Nürnberg - FAU

Keywords and subjects

Lattice Boltzmann Method | HPC | CFD

RDM – Bridge to CD

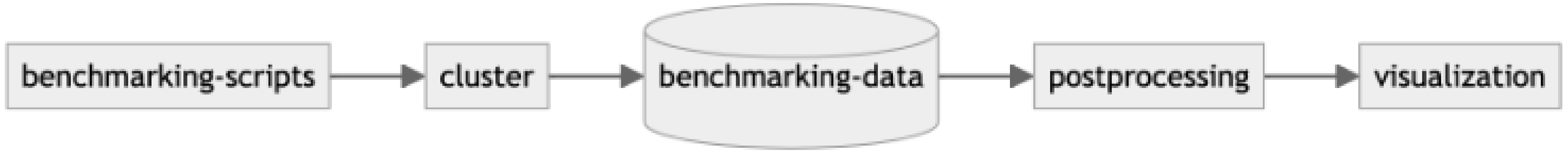
- Research Data Management (RDM) Repository
  - Internal first attempt for benchmark CD
  - Focus on FAIR Principles
  - Content of RDM-Collection
    - Build- and Run **Scripts**
    - Performance **Data**
    - Team **Guidbook** for systems
  - Requires manuel execution on site
    - **NO** Continous Deployment.



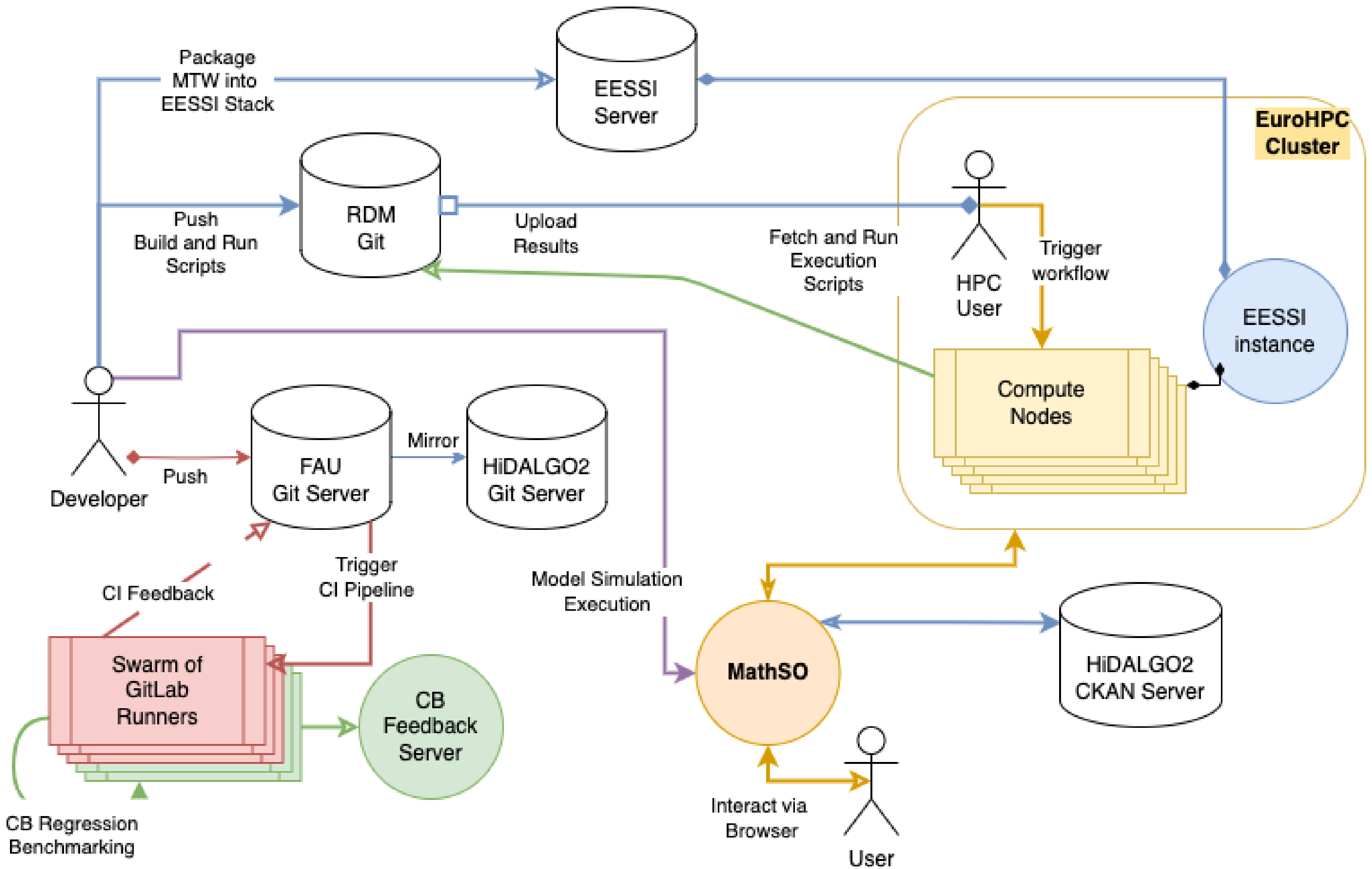
LSS-RDM

LSS - Research Data Management

- benchmarking-scripts
- benchmarking-data
- Documentation
- postprocessing



WaLBerla – **EuroHPC Landscape**



Tools – **Package Managers**

- **System Admin** oriented: E.g., RPM (Redhat) und Deb (Ubuntu, Debian)
- **User** oriented: Mostly **EasyBuild** and **Spack**; Other exist, e.g. Nix or pip for python, …

- Software deployment tools for complex **research software** (and HPC)
  - Build **scientific libraries** (MPI, HDF5, PETSc, etc.)
  - **Manage dependencies**, toolchains, updates, etc.
  - Enable **reproducibility** of software environments on **HPC clusters**
  - Provide environment **modules for users**
  - Facilitates concurrent version (multiple-version of same software on a system)

Note: Spack used in waLBerla to manage build dependencies in CI

- Not to deploy waLBerla itself (yet)
- Future developments will likely include Spack and EasyBuild

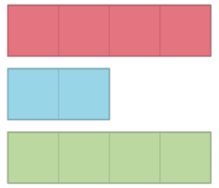
Compare – **Package Managers**

Spack

- By **LLNL** (**US**) | Internationally used
- **Flexible** dependency organization
  - E.g. ~fortran+Ofast **@gcc12 target=**x86_64
  - Convenient for **users** and **researchers**
  - Allows many options (Maybe too many?)
- Define Package as **Python classes**
  - Build options → **`variant`**
  - Dependencies→ **`depents_on`**
  - **Calls a function** for each parameter
- Example PETSC
  - >800 lines
  - > 40 variants + dep versions → ∞ Variations
- Pre-build binary stack
  - **Public Build Caches** (e.g. E4S)
- Main package repository
  - github.com/spack/spack-packages
  - Very Large Repository

EasyBuild

- **European** Solution, By Gent University (**NL**)
- **Hierarchical** decency organization
  - One **dependency** chain **per eb-config** file
  - Convenient for **maintainer** and **admins**
  - Ready-to-use recipes; fewer decisions needed
- Define Package as EasyBuild configs (DSL)
  - Build options → **`toolchainopts`**
  - Dependencies→ **`dependencies`**
  - **Defines a lists** for each parameter
- Example PETSC
  - 46 lines
  - 3 build options & 10 PETSc.eb files
- Pre-build binary stack
  - **EESSI** (European) (based on CernVM-FS)
- Main package repository
  - github.com/easybuilders/easybuild-easyconfigs



Tools – **Benchmark Orchestration**

- Test / Benchmark orchestration tools for complex **research software** (and HPC)
    - *Initially developed* for Acceptance and Regression **tests of HPC systems**
    - Natively designed to work with workload managers like **SLURM**
    - **Abstraction** of test dependencies and execution
        - Require definition of at least: System, Build, Parameterset
    - **Spans Testmatirx / -pool** from set of parameters
    - Utilizes **data staging**; Own folders for execution and results per test
    - Eases **result collection**
    - Support **reproducibility** (workspace snapshotting, environment tracking)
    - One job per test; Risks sheduler problems; ~~Loved~~Loathed by system admins

- Common Tools **ReFrame** or **JUBE**
    - Both **European solutions**
    - Other exist, e.g. Pavilion2, Ramble, …

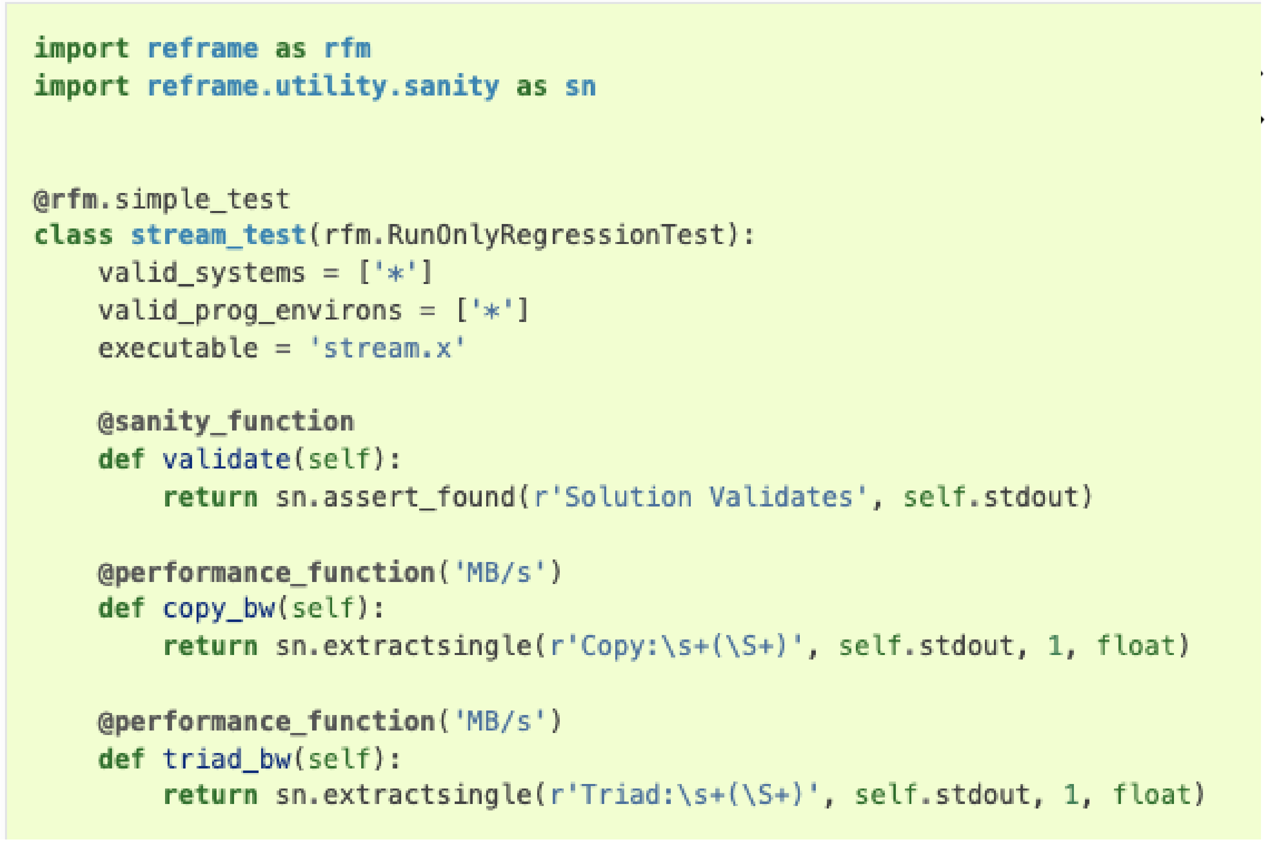
Note: Different CoE requre the use of different Benchmark Orchestrators

Tool – **ReFrame**

- By **CSCS** Zürich (**CH**)
- Uses **Python based DSL**
- Job divided in 5 stages; Most stages are required

Setup	Compile	Run	Sanity	Performance	Cleanup

- Co
  - Natively provides **result collection from logfiles** by defining regexes
  - Handles **complex execution graphs**
  - **Simple validation logic** for quick feedback
- Opinion:
  - **Stronger** and more **flexible**
    - You can do basically everything if you call your own functions correctly
  - Power to customize **can become a pitfall**
  - Convenient features quickly become **annoying** when modelling simple tasks
  - **Initial barrier** relatively high; Time-consuming first setup
    - **Nice documentation**

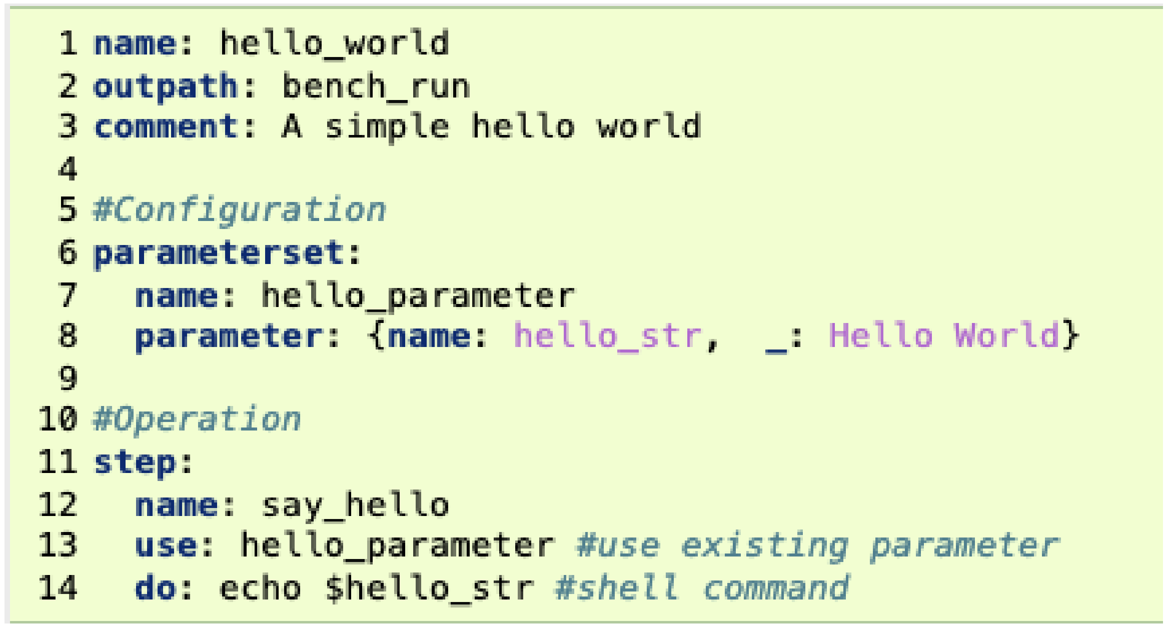


```python
import reframe as rfm
import reframe.utility.sanity as sn


@rfm.simple_test
class stream_test(rfm.RunOnlyRegressionTest):
    valid_systems = ['*']
    valid_prog_environs = ['*']
    executable = 'stream.x'

    @sanity_function
    def validate(self):
        return sn.assert_found(r'Solution Validates', self.stdout)

    @performance_function('MB/s')
    def copy_bw(self):
        return sn.extractsingle(r'Copy:\s+(\S+)', self.stdout, 1, float)

    @performance_function('MB/s')
    def triad_bw(self):
        return sn.extractsingle(r'Triad:\s+(\S+)', self.stdout, 1, float)
```

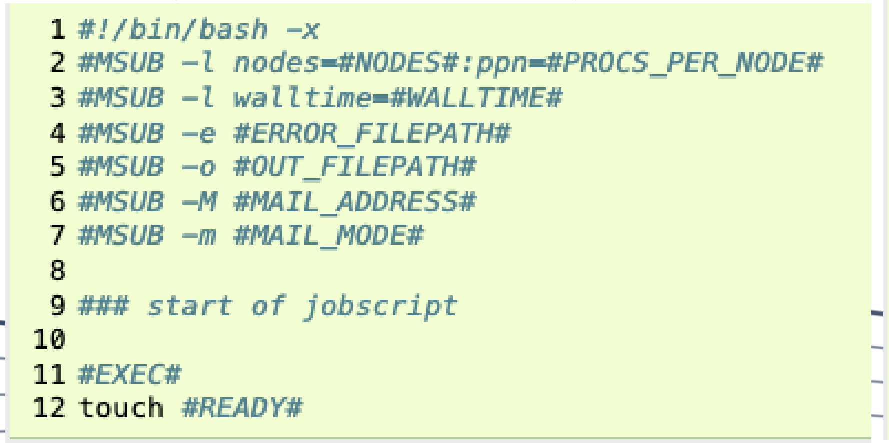
Tool – **JUBE**

- By **JSC** Jülich (**GER**)
- Uses XML or **YAML configs**
- **Scheduler** integration works with **template macros**

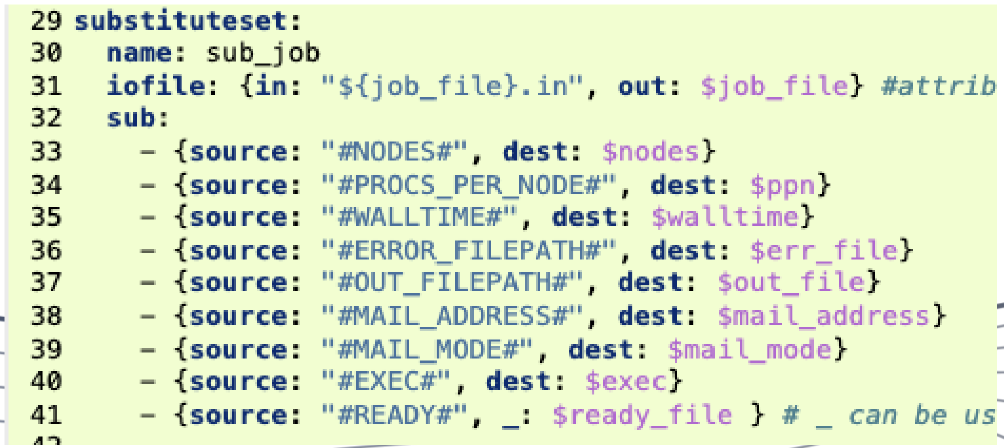
- <u>Opinion</u>:
  - Test definitions are **clear** and **easy** to understand
  - **Low initial barrier** for easy tasks
  - **Hard do model more complex tasks**
  - Good for waLBerla benchmarks
    - One binary per system and application
    - Variation of batch and parameter files required



```yaml
name: hello_world
outpath: bench_run
comment: A simple hello world

#Configuration
parameterset:
  name: hello_parameter
  parameter: {name: hello_str,  _: Hello World}

#Operation
step:
  name: say_hello
  use: hello_parameter #use existing parameter
  do: echo $hello_str #shell command
```



```bash
#!/bin/bash -x
#MSUB -l nodes=#NODES#:ppn=#PROCS_PER_NODE#
#MSUB -l walltime=#WALLTIME#
#MSUB -e #ERROR_FILEPATH#
#MSUB -o #OUT_FILEPATH#
#MSUB -M #MAIL_ADDRESS#
#MSUB -m #MAIL_MODE#

### start of jobscript

#EXEC#
touch #READY#
```



```yaml
substituteset:
  name: sub_job
  iofile: {in: "${job_file}.in", out: $job_file} #attrib
  sub:
    - {source: "#NODES#", dest: $nodes}
    - {source: "#PROCS_PER_NODE#", dest: $ppn}
    - {source: "#WALLTIME#", dest: $walltime}
    - {source: "#ERROR_FILEPATH#", dest: $err_file}
    - {source: "#OUT_FILEPATH#", dest: $out_file}
    - {source: "#MAIL_ADDRESS#", dest: $mail_address}
    - {source: "#MAIL_MODE#", dest: $mail_mode}
    - {source: "#EXEC#", dest: $exec}
    - {source: "#READY#", _: $ready_file } # _ can be us
```

Tools – Friendly **Execution Orchestration**

- MathSO
    - TODO copy content from D2.6
    - Add YouTube Link
    - Not reallly part of CX.
    - Allows the Orchestration of tasks on target clusters via web interface.
    - Developer can abstract program execution.
    - Users can simply insert their parameters and needs.
        - Knowledge of HPC or cluster computing not necessary.

Summary

Hosting Source Code	(GIT) -> **GitLab**, **GitHub**, Bitbucket, …
Building Software	**CMake + Presets**
CI-Tools	(**GitLab**) -> **GitLab CI**, Pages, **Container Registry** Note: Possible with GitHub as well
Package Managers	**Spack**, **EasyBuild**, Nix, Guix, pip, conda, npm, gem, gradle, …
Benchmark Orchestration Tools	**ReFrame**, JUBE, Pavilion, Benchpark/Ramble, …

There are many tools for all kinds of purpuses.

The result is often the same,
but they all comes in many flavours.

Take some time to try them
and reach out to compare.

Centre of Excellence in Exascale CFD

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Sweden, Germany, Spain, Greece, and Denmark under grant agreement No 101093393.

Co-funded by
the European Union

EuroHPC
Joint Undertaking

CEEC
Centre of Excellence in Exascale CFD

Thank you
for your attention!

CEEC
Centre of Excellence in Exascale CFD