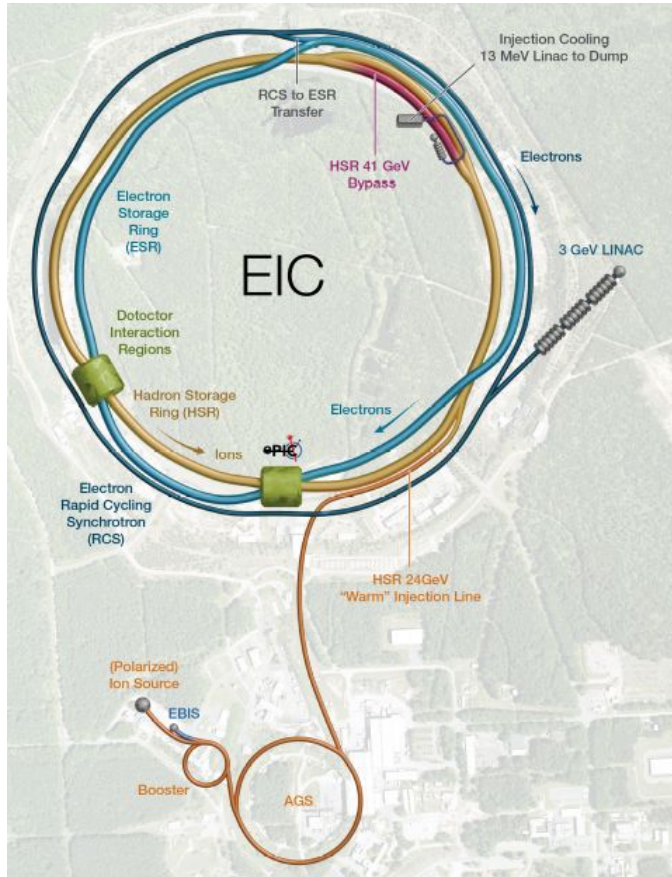


# Cache Rules Everything Around Me

Building/testing full-stack containers on each commit

**Wouter Deconinck (U. Manitoba)**  
**for the ePIC Collaboration**  
**CHEP 2024 – October 22, 2024**

# EIC/ePIC Detector: Software and Computing for the 2030s



Build forward-looking team of user-developers to ensure the long-term success of the EIC/ePIC scientific program in software & computing.

- Focus on **modern scientific computing practices**
- Strong emphasis on **modular orthogonal tools**

Integration with HTC/HPC, CI workflows, and enable use of data-science toolkits.

Avoid “not-invented-here” syndrome, and instead leverage cutting-edge CERN-supported software components where possible.

- Build on top of mature, well-supported, and actively developed software stack.
- Share support burden with upstream where possible

Actively work with other software stakeholders to help develop and integrate community tools for all EIC collaborations.

# EIC Software Statement of Principles

## EIC SOFTWARE: Statement of Principles

- 1 We aim to develop a diverse workforce, while also cultivating an environment of equity and inclusivity as well as a culture of belonging.**
- 2 We will have an unprecedented compute-detector integration:**
  - We will have a common software stack for online and offline software, including the processing of streamed data and its time-ordered structure.
  - We aim for autonomous alignment and calibration.
  - We aim for a rapid, near-real-time turnaround of the raw data to online and offline productions.
- 3 We will leverage heterogeneous computing:**
  - We will enable distributed workflows on the computing resources of the worldwide EIC community, leveraging not only HTC but also HPC systems.
  - EIC software should be able to run on as many systems as possible, while supporting specific system characteristics, e.g., accelerators such as GPUs, where beneficial.
  - We will have a modular software design with structures robust against changes in the computing environment so that changes in underlying code can be handled without an entire overhaul of the structure.
- 4 We will aim for user-centered design:**
  - We will enable scientists of all levels worldwide to actively participate in the science program of the EIC, keeping the barriers low for smaller teams.
  - EIC software will run on the systems used by the community, easily.
  - We aim for a modular development paradigm for algorithms and tools without the need for users to interface with the entire software environment.

- 5 Our data formats are open, simple and self-descriptive:**
  - We will favor simple flat data structures and formats to encourage collaboration with computer, data, and other scientists outside of NP and HEP.
  - We aim for access to the EIC data to be simple and straightforward.
- 6 We will have reproducible software:**
  - Data and analysis preservation will be an integral part of EIC software and the workflows of the community.
  - We aim for fully reproducible analyses that are based on reusable software and are amenable to adjustments and new interpretations.
- 7 We will embrace our community:**
  - EIC software will be open source with attribution to its contributors.
  - We will use publicly available productivity tools.
  - EIC software will be accessible by the whole community.
  - We will ensure that mission critical software components are not dependent on the expertise of a single developer, but managed and maintained by a core group.
  - We will not reinvent the wheel but rather aim to build on and extend existing efforts in the wider scientific community.
  - We will support the community with active training and support sessions where experienced software developers and users interact with new users.
  - We will support the careers of scientists who dedicate their time and effort towards software development.
- 8 We will provide a production-ready software stack throughout the development:**
  - We will not separate software development from software use and support.
  - We are committed to providing a software stack for EIC science that continuously evolves and can be used to achieve all EIC milestones.
  - We will deploy metrics to evaluate and improve the quality of our software.
  - We aim to continuously evaluate, adapt/develop, validate, and integrate new software, workflow, and computing practices.

The "Statement of Principles" represent guiding principles for EIC Software. They have been endorsed by the International EIC community. For a full analysis, see LINC.



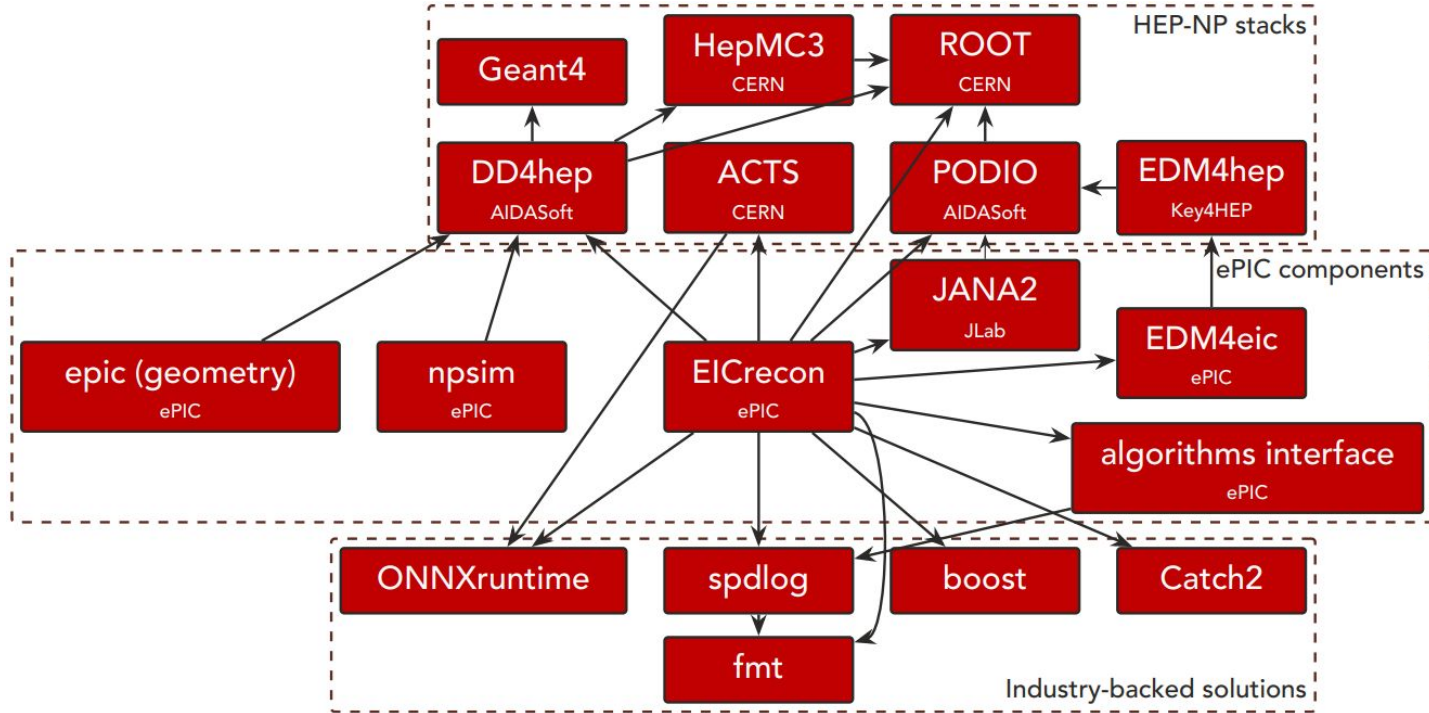
<https://eic.github.io/activities/principles.html>

## Key principles for this talk

- **User-centered design:** aim to give every user and use case an identical environment. Implementation: containers built from same packages definitions using spack.
- **Continuous deployment:** aim for always production ready stack, with testing and validation before deployment. Components calver or semver, with entire stack calver. Monthly production campaign sprint targets, rapid feedback on large event samples.

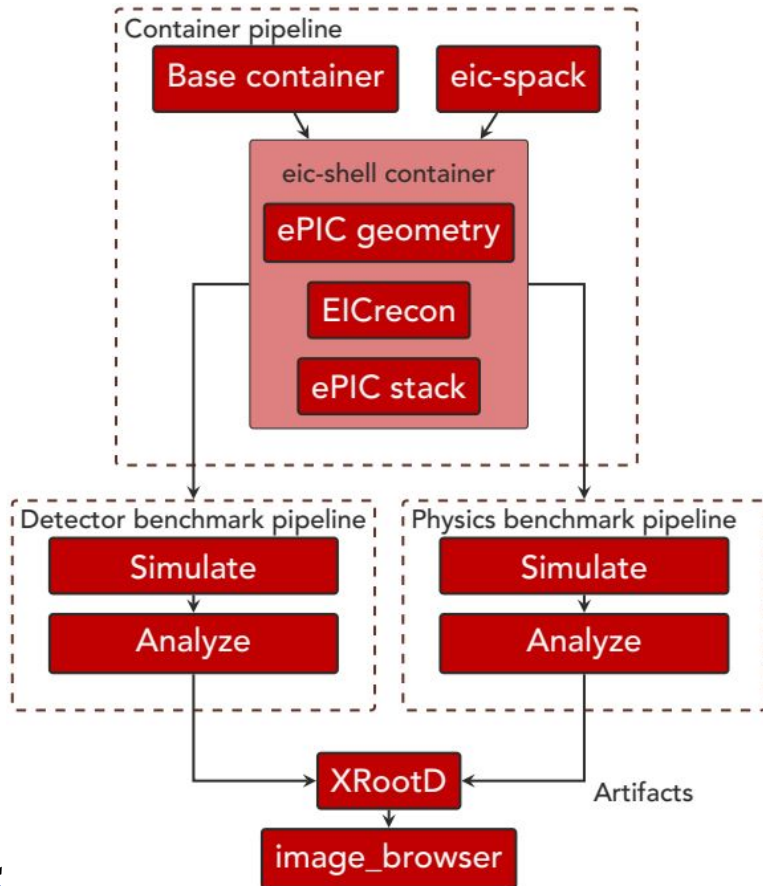


# The ePIC Software Stack and Key Dependencies: Modular Components



A **modular software stack** with interdependent components requires more than CI/CD testing of individual components. We need **full-stack** testing, validation, benchmarking.

# Full-Stack Testing, Validation, Benchmarking Workflow



**This workflow is run on every commit** in any component developed by ePIC, and on every patch/upgrade of external components.

Requires simulations and reconstruction, with **sufficiently granular caching** of intermediate files to speed up workflow.

Three key phases:

- **Build** temporary container, push to registries (tag is sequential id)
- **Run** testing, validation, benchmarks inside the temporary container
- **Publish** testing results to XRootD, where visualizer pulls artifacts for comparison (Rucio being rolled out).

# Full-Stack Testing, Validation, Benchmarking Workflow

## Code change on GitHub

✔ Lfhcal geoupdate #4147

### Summary

#### Jobs

- ✔ xmllint-before-build
- ✔ list-detector-configs
- ✔ build (gcc, g++)
- ✔ build (clang, clang++)
- ✔ xmllint-after-build
- ✔ check-geometry-configs
- ✔ check-tracking-geometry
- ✔ convert-to-gdml
- ✔ convert-to-tgeo
- ✔ convert-to-step
- ✔ dump-constants (epic\_craterl...
- ✔ dump-parameter-table (epic\_c...
- ✔ check-overlap-tgeo (m, epic\_c...
- ⊙ check-geometry-full
- ✔ check-overlap-geant4 (epic\_cr...
- ✔ check-overlap-geant4-fast (ep...
- ✔ check-overlap-geant4-fast (ep...
- ✔ generate-prim-file (epic\_crate...
- ✔ npsim-gun (pi, epic\_craterlake)
- ✔ npsim-gun (e, epic\_craterlake)
- ✔ npsim-dis (5x41, 1, epic\_crater...

## Benchmarking on Gitlab

Project: eic/detector\_benchmarks: pr/no\_detector\_build

✔ Passed Wouter Deconinck created pipeline for commit: e18a199e 3 hours ago, finished 1 hour ago

Trigger team 185 jobs 83 minutes 6 seconds, queued for 2,741 seconds

Group jobs by Stage Job dependencies

Stage	Job
benchmarks	bench_b0_tracker
	bench_backgrounds_eocal_backwards
	bench_backgrounds_eocal
	bench_barrel_eocal
	bench_ditch
	bench_eocal_gaps
	bench_em_cal_barrel_electrons_scan
	bench_em_cal_barrel_photons
	bench_em_cal_barrel_pi0
	bench_em_cal_barrel_pion_rejection
	bench_em_cal_barrel_pions
	bench_erich
	bench_hcal_barrel
	bench_hcal_barrel_scan
	bench_insert_muon
collected	collect_results_backgrounds
	collect_results_backgrounds_eocal
	collect_results_barrel_eocal
	collect_results_barrel_hcal
	collect_results_eocal_gaps
	collect_results_eocal
	collect_results_insert_muon
	collect_results_insert_neutron
	collect_results_material_scan
	collect_results_psd
	collect_results_tracking_performance
	collect_results_tracking_performance_cam
	collect_results_tracking_performance_dis
	collect_results_zdc
	collect_results_zdc_lambda
deploy	deploy_re

## Presentation Interface

Home Physics Detector CI TDR Contact

Plot Type: All

Material Scan: eta, phi=0

Material Scan: eta=2.5, phi=0

Material Scan: eta=-2.5, phi=0

Material Scan: Whole Detector

Material Scan: Central Tracking

▲ Plots on this page are automatically generated and are not approved for use in presentations or other documents.

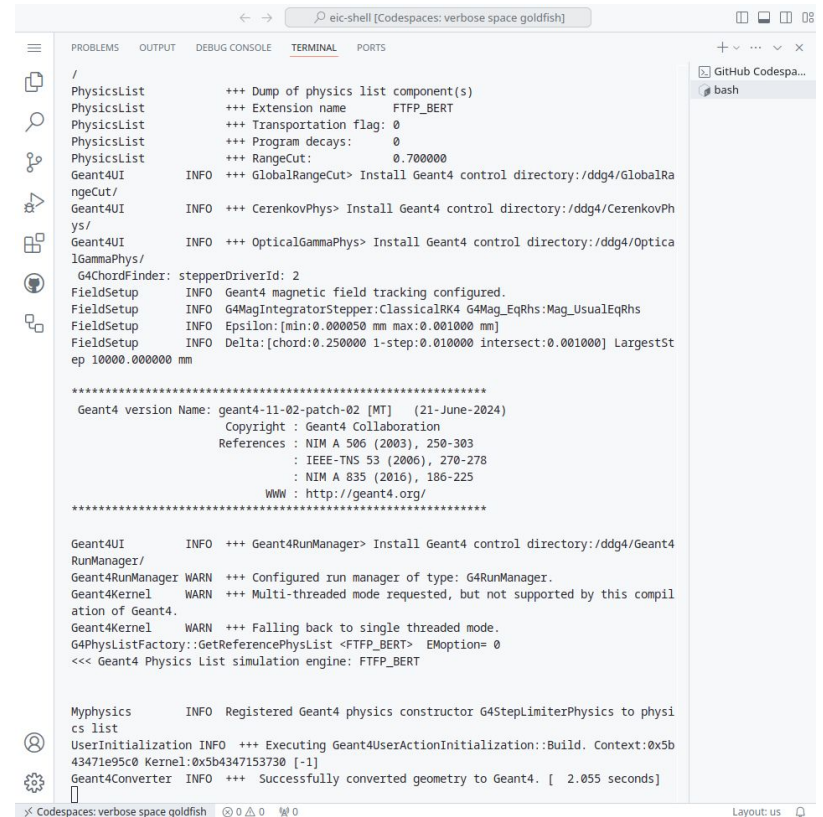
# Containers at the Core of the EIC/ePIC Computing Approach

User-centered design: Present a **consistent software environment** to users, no matter where they interact with the software stack

- Avoid separate distribution for interactive analysis computing
- Allow reproducing CI/CD issues on local development environments
- Allow reproducing production campaign errors on local system

Following all use **same container**:

- Interactive day-to-day use on
- Small farm simulations
- Collaboration campaigns
- CI/CD system
- GitHub Codespaces



```
eic-shell [Codespaces: verbose space goldfish]
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
/
PhysicsList      +++ Dump of physics list component(s)
PhysicsList      +++ Extension name      FTFP_BERT
PhysicsList      +++ Transportation flag:  0
PhysicsList      +++ Program decays:      0
PhysicsList      +++ RangeCut:            0.700000
Geant4UI         INFO +++ GlobalRangeCut> Install Geant4 control directory://ddg4/GlobalRangeCut/
Geant4UI         INFO +++ CerenkovPhys> Install Geant4 control directory://ddg4/CerenkovPhysics/
Geant4UI         INFO +++ OpticalGammaPhys> Install Geant4 control directory://ddg4/OpticalGammaPhys/
G4ChordFinder:  stepperDriverId: 2
FieldSetup      INFO Geant4 magnetic field tracking configured.
FieldSetup      INFO G4MagIntegratorStepper:ClassicalRK4 G4Mag_EqRhs:Mag_UsualEqRhs
FieldSetup      INFO Epsilon:[min:0.000050 mm max:0.001000 mm]
FieldSetup      INFO Delta:[chord:0.250000 1-step:0.010000 intersect:0.001000] LargestStep 10000.000000 mm

*****
Geant4 version Name: geant4-11-02-patch-02 [MT] (21-June-2024)
Copyright : Geant4 Collaboration
References : NIM A 506 (2003), 250-303
            : IEEE-TNS 53 (2006), 270-278
            : NIM A 835 (2016), 186-225
WWW        : http://geant4.org/
*****

Geant4UI         INFO +++ Geant4RunManager> Install Geant4 control directory://ddg4/Geant4RunManager/
Geant4RunManager WARN +++ Configured run manager of type: G4RunManager.
Geant4Kernel    WARN +++ Multi-threaded mode requested, but not supported by this compilation of Geant4.
Geant4Kernel    WARN +++ Falling back to single threaded mode.
G4PhysicsListFactory::GetReferencePhysicsList <FTFP_BERT> EMOption= 0
<<< Geant4 Physics List simulation engine: FTFP_BERT

Myphysics      INFO Registered Geant4 physics constructor G4StepLimiterPhysics to physics list
UserInitialization INFO +++ Executing Geant4UserActionInitialization::Build. Context:0x5b43471e95c0 Kernel:0x5b4347153730 [-1]
Geant4Converter INFO +++ Successfully converted geometry to Geant4. [ 2.055 seconds]
```

<https://github.com/codespaces/new/eic/eic-shell?quickstart=1>

# Spack for Container Building

**Spack:** A flexible package manager supporting multiple versions, configurations, platforms, compilers.

Spec language: variants `+`, compilers `%`, target archs `target=x86_64_v3`

**Environments** commonly used in NHEP:

```
spack env create eic_xl
spack add acts@33.1.0 +edm4hep %clang
spack concretize
spack install --jobs 64
spack buildcache push ghcr
```

Separation of package details from environment contents using configs.

Other talks on Spack at CHEP24:

[#303](#): Key4hep / [#409](#): Fermilab



Spack for EIC/ePIC containers:

- **System compilers** (gcc-12, direction clang-16 but fortran...)
- **Versions and variants** defined in package.yaml, used for multiple environments (xl, prod, cuda)
- **Unified concretization** (one version of installed direct deps, allow for multiple build deps which are garbage-collected away)
- Modified `spack containerize` to allow for:
  - Cherry-picks against spack repo
  - Addition of downstream repos
  - Two-track builds (see next slide)
  - Historical reasons (once missing functionality since been added)



# Spack for Container Building

[ghcr.io/eic/eic\\_prod](https://github.com/ghcr.io/eic/eic_prod) (no interactivity)

spack:

include:

- ../concretizer.yaml
- ../packages.yaml

specs:

- dd4hep -ddev
- ...

dd4hep:

require:

- '@1.30'
- +ddg4 +ddcad +hepmc3 +utilityapps
- any\_of: [+ddev, -ddev]

[ghcr.io/eic/eic\\_xl](https://github.com/ghcr.io/eic/eic_xl) (full interactivity)

spack:

include:

- ../concretizer.yaml
- ../packages.yaml

specs:

- dd4hep +ddev
- ...

Allow for specifying versions and variants in `packages.yaml`, then pick one allowed option in `spack.yaml`.

Additional options not yet in production: `include_concrete` to import full envs

# Increasing Development Speed: Speeding Up Container Builds

Desire to identify issues with validation and benchmarks rapidly.

- Requires rapid CI/CD turnaround time
- Start actual validation and benchmarks as soon as possible

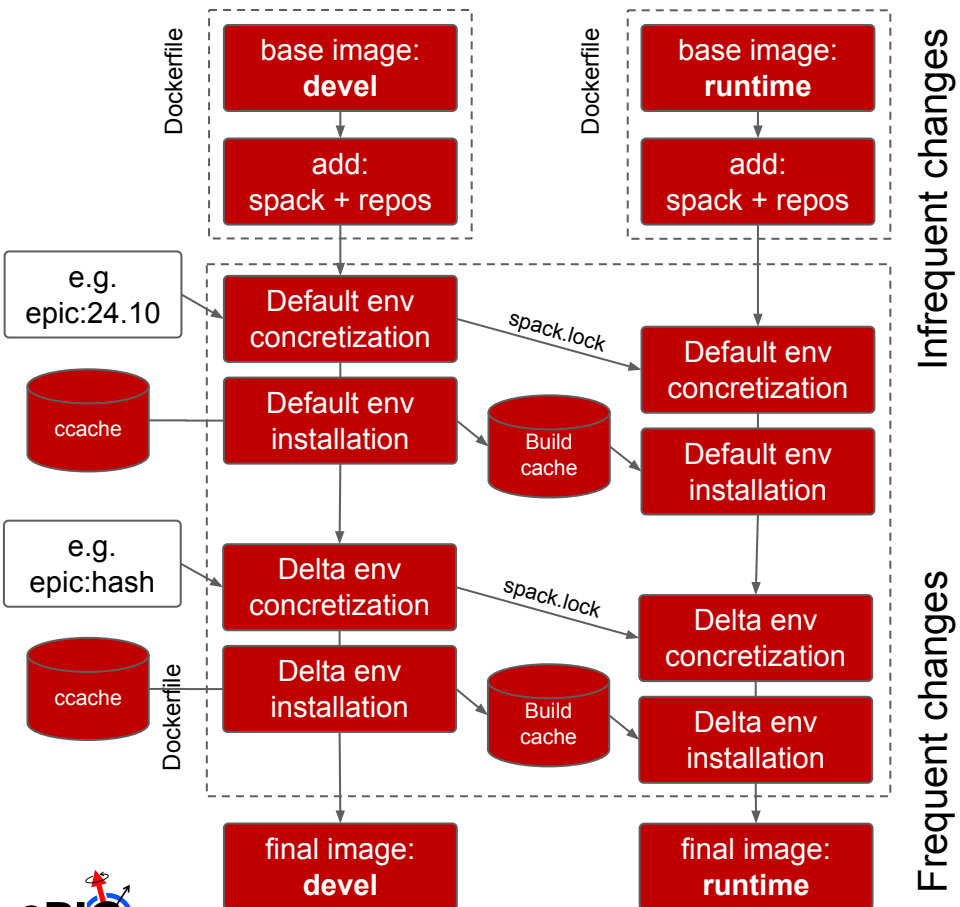
Speed of container builds is latency driver for start of benchmarking and validation

- Focus on speed up of container build is of primary importance

Strategy:

- **Spack build caches** for dependencies with unchanged hash (local, S3, OCI)
- **ccache** for consecutive builds of the same codebase (also for GitHub Actions)
  - Build times now dominated by CMake configuration step, not the actual build
- Container layer caching: aim for **small delta layers** between updates, when dependencies are unchanged, but also allow for infrequent dependency changes

# Spack for Container Building: Maximizing Layer Reuse, Devel vs. Runtimes



## Challenge:

- CUDA (or oneAPI) accelerated stack while maintaining **small runtime image** (spack does not currently have a CUDA runtime install that removes CUDA devel)
- Also: **unclear licensing** when not using upstream devel and runtime as base layer

## Approach:

- Devel track pushes to build cache; runtime track not allowed to build and can only pull from build cache.

Result: **consistent devel and runtime** container images (license-compliant).

# Reuse of Build Caches between Gitlab CI and GitHub Actions

Spack supports multiple binary build cache mechanisms: local, S3 buckets, OCI layers,... We use a combination of all of them (combined with `autopush` upon build).

Build cache writing:

- Local build cache, accessed as Dockerfile build mount with `RUN --mount=type=cache, target=/var/cache/spack`
- S3 buckets on appliance at BNL (but phasing out)
- OCI layers on local Gitlab registry and on ghcr.io

Build cache reading:

- Local build cache and OCI on local registry are preferred since fastest
- OCI layers on ghcr.io are used for passing build cache products between sites

Seeding of build caches with cloud resources, e.g. buy a 32-core arm cluster for a day.



# Triggering of Gitlab CI Pipeline from GitHub Actions with Reporting Back

GitHub is main code repository (publicly accessible, is portfolio for user-developers).

Gitlab is main CI/CD site (dedicated compute resources), but limited to few users.

Challenge: How to synchronize between GitHub and Gitlab (without premium Gitlab)?

Approach:

- Trigger workflow from GitHub Actions with `eic/trigger-gitlab-ci`
  - Requires Gitlab pipeline trigger token on receiving repository as Actions secret
  - Can pass variables, e.g. `GITHUB_PR`, `GITHUB_SHA`, `PIPELINE_NAME`, ...
- Run Gitlab workflow and report back with webhooks to GitHub Actions API
  - Requires GitHub fine-grained token on receiving organization as pipeline secret
  - Report back success/failure to PR, and posts links to artifacts (e.g. geometry viewer)

Management concern: secrets (organization-wide on GitHub, but per repo on Gitlab)

# Hashicorp Bakefiles for Encoding of Gitlab vs. GitHub Differences

## Challenges:

- Gitlab CI can push build cache and containers to private local registry.
- Local developer has read-only access to public registries, and usually no local registry.
- Different configuration needed for different running scenarios, with no capacity within Dockerfile to resolve.
- Images pushed to multiple repo/tags.

## Original approach:

- Extensive use of `--build-args` with `docker build` commands that differ on Gitlab and GitHub.
- **Poor maintainability...**

## Bakefiles (`.hcl`) to control docker builds

- Env vars to control build:

```
args = [  
    SPACK_VERSION =  
        try(SPACK_VERSION, "v0.22.2")  
]
```
- Easy to specify outer products:

```
tags = [  
    for r in regs:  
        for t in tags:  
            ${r}/eic:${t}  
]
```
- Inheritance of optional targets:

```
target "default" {  
    inherits =  
        ["docker-metadata-action"]  
}
```

→ Single Dockerfile: `docker bake`

# Taking A Step Back: GitHub Actions for CI Testing of Individual Modules

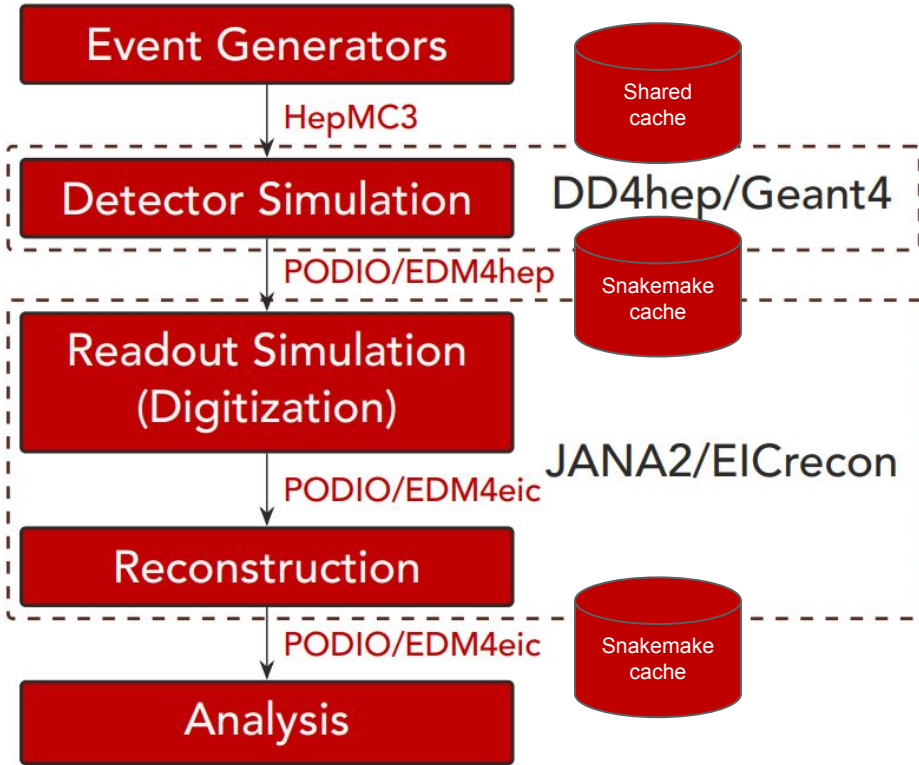
Before system integration testing, we also **still run CI tests in GitHub Actions**:

- But we use the container environments with all the dependencies installed at the versions used in the various production environments.
- Using the containers as published on cvmfs.
- For “nightly” versions (all current main/master branches; deployment each 5 hrs).
- For several released tags (backwards compatibility checks)

Only **after GitHub Actions checks complete** the Gitlab chain gets triggered.

- Containers read from cvmfs using `cvmfs-contrib/github-action-cvmfs`
  - Typically 20-30s overhead due to installation; speed-up through `apt` archive caching
  - No cvmfs cache caching yet in downstream workflows, unclear how much this would speed up; cvmfs still significantly faster than pulling entire container in each job since only part of the container is needed and cvmfs downloads selectively what is used
- GitHub Actions caching with `ccache`: 20 minute build in under 2 minutes (dominated by CMake config)

# Validation Workflows: Caching Intermediate Simulations with Snakemake



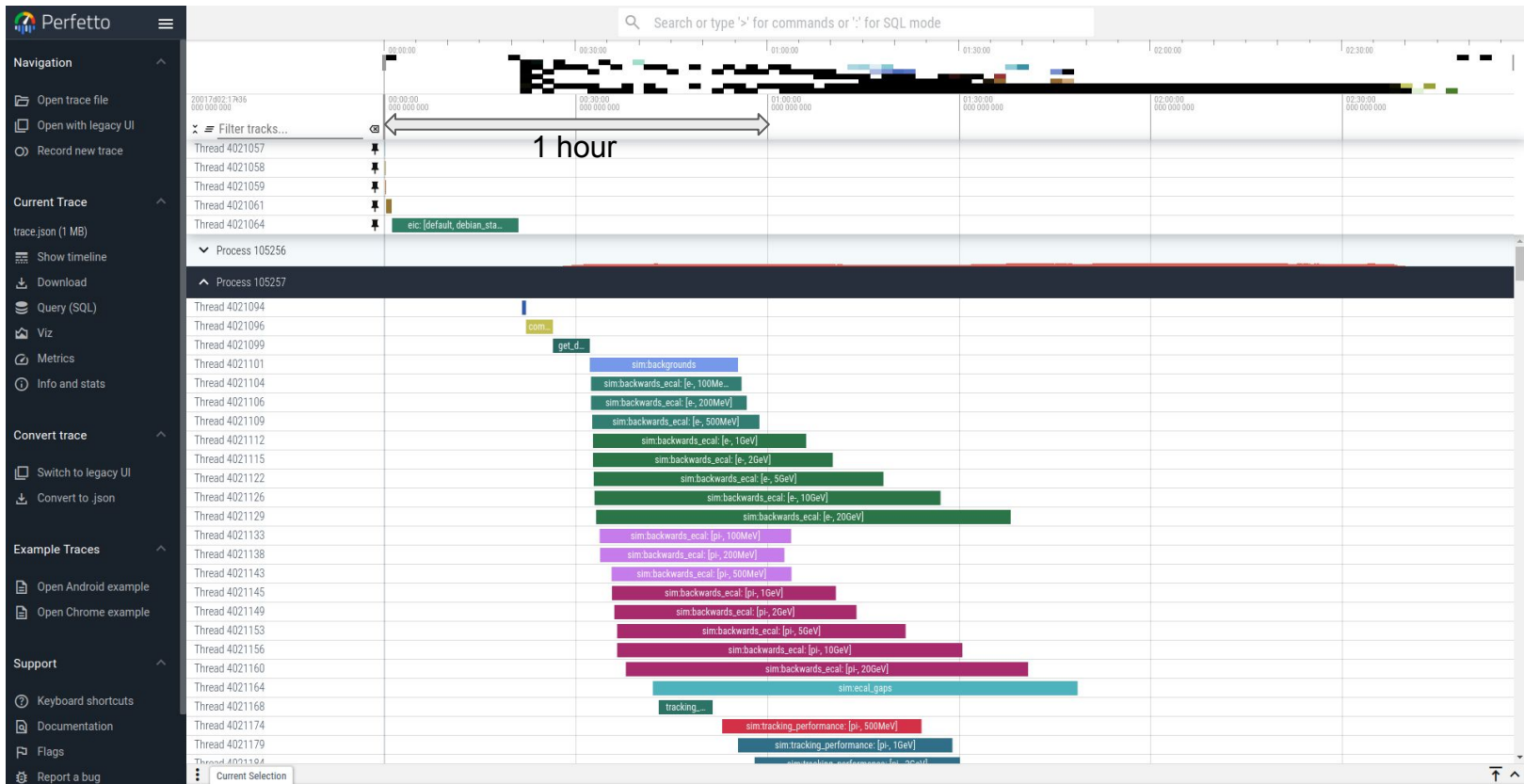
Validation and benchmarking jobs:

- Event generators: read from XRootD with local caching on directory shared between all jobs
- Detector simulation caching: Snakemake `--cache`, keyed on code, parameters and software environment
  - Rerun when detector geometry or DD4hep version changes, but not otherwise
- Event reconstruction caching: Snakemake `--cache`, keyed on configuration and software environment

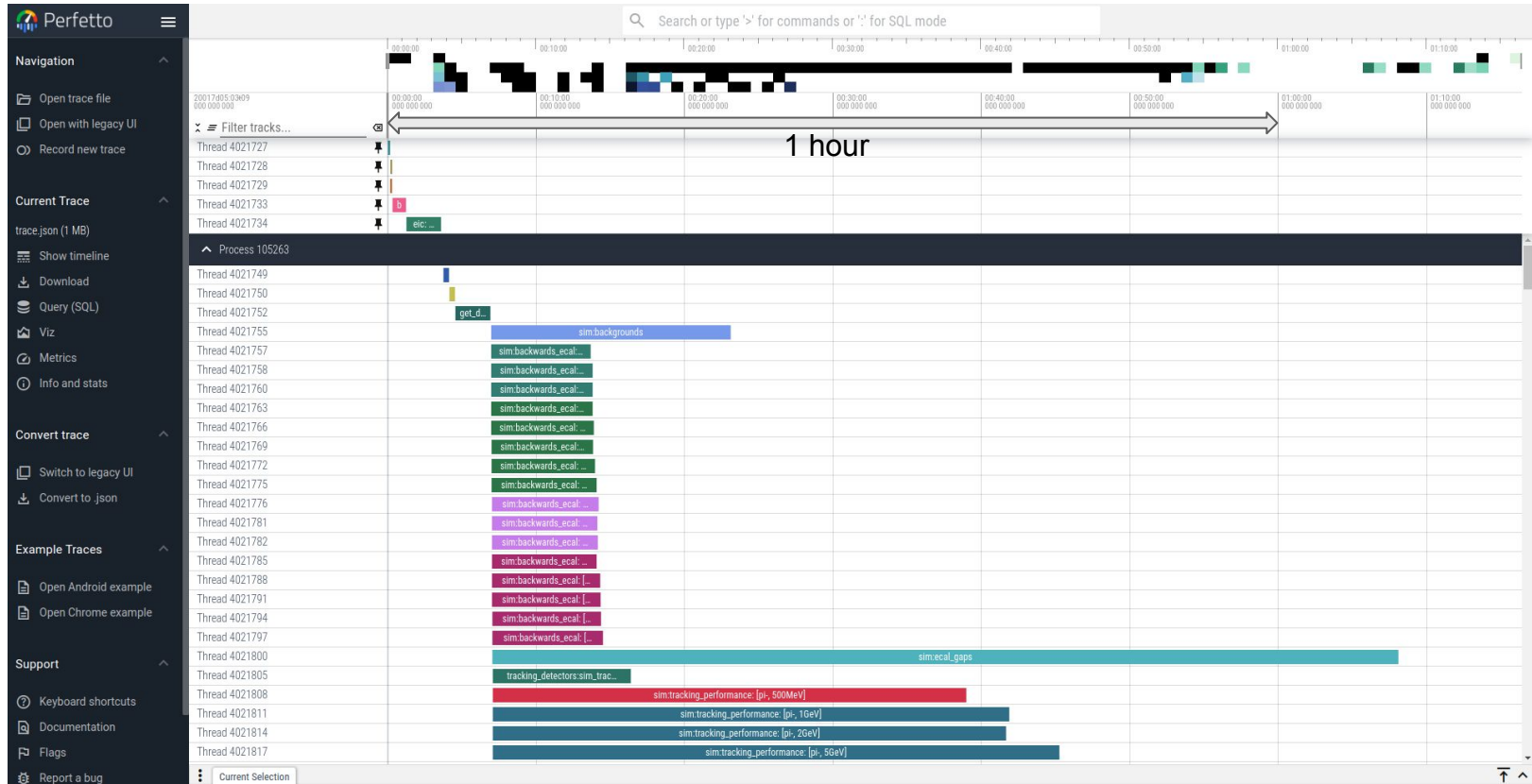
Combination of Snakemake and local caching



# Container Build Latency (reconstruction code changes): 20 minutes



# Container Build Latency (analysis benchmark changes): 8 minutes



# Opportunities for Further Improvements

During full rebuilds of packages (not build cache retrievals):

- `ccache` has sped up the build enough that we now spend most time in the CMake configuration stages... A challenge shared with the spack project and especially relevant on systems with high core-count.

During typical builds (one or few packages need rebuild, others are retrieved):

- About 20 minutes to concretize, build, install when only a few minutes of compilation is expected
- Separation of environment into
  - Layer with dependencies only
  - Layer (default) with latest released versions
  - Layer (delta) with current commit hash versions:  
reconcretization of only handful of packages in already constrained environment with `include_concrete`

# Summary

---

## Takeaways:

- **Integrated full-system testing of modular software environments** requires more than is possible in single project continuous integration tests.
- **Building full-stack containers** for integrated full-system testing, validation, and benchmarking **on every commit is possible**, but requires judicious use of caching strategies.

## Ongoing development:

- **Upstreaming** and documenting of customized functionality into spack (e.g. containers with custom repositories, two-track builds without layer copy).
- Use of **include\_concrete** and better spack environment layering to avoid the need to concretize full environments, only reconcretize changed components.



# Abstract

---

The ePIC collaboration is working towards the realization of the first detector at the upcoming Electron-Ion Collider. As part of our computing strategy, we have settled on containers for the distribution of our modular software stacks using spack as the package manager. Based on abstract definitions of multiple mutually consistent software environments, we build dedicated containers on each commit of every pull request for the software projects under our purview. This is only possible through judicious caching from container layers, over downloaded artifacts and binary builds, down to individual compiled files. These containers are subsequently used for our benchmark and validation workflows. Our container build infrastructure runs with redundancy between GitHub and self-hosted GitLab resources, and can take advantage of cloud-based resources in periods of peak demand. In this talk, I will discuss our experiences with newer features of spack, including storing build products as OCI layers and inheritance of previously concretized environments for software stack layering.