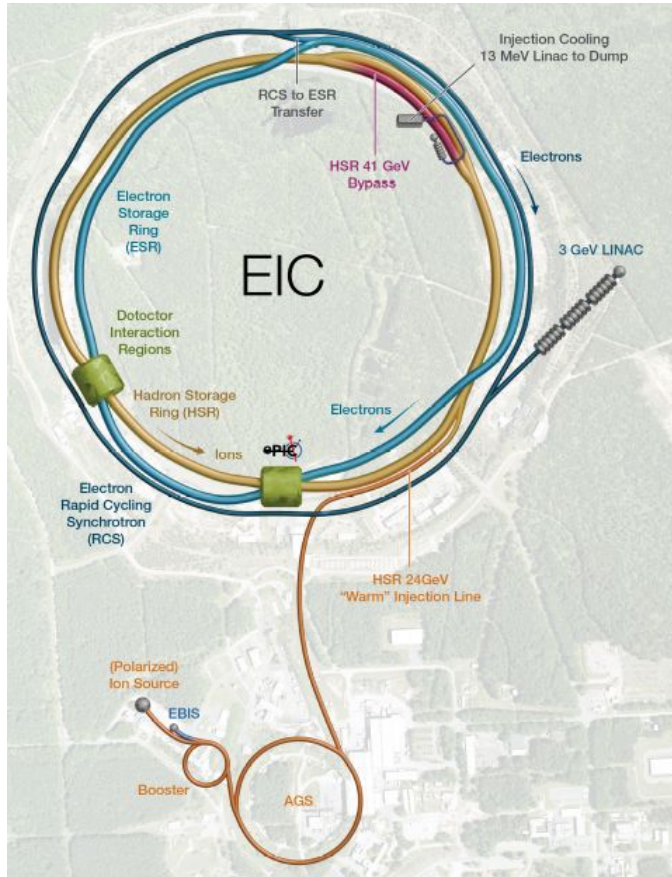# Cache Rules Everything Around Me
Building/testing full-stack containers on each commit

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

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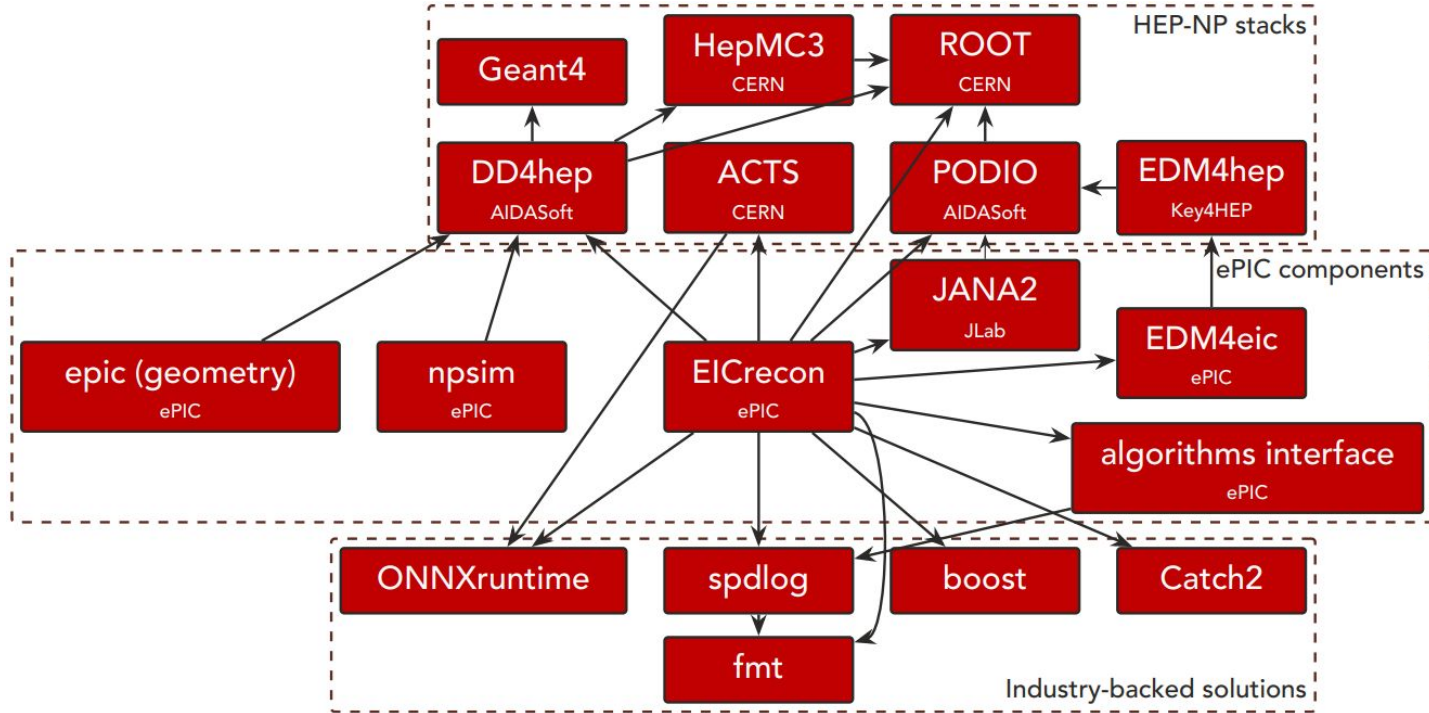
A **modular software stack** with interdependent components requires more than CI/CD testing of individual components. We need **full-stack** testing, validation, benchmarking.

Full-Stack Testing, Validation, Benchmarking Workflow
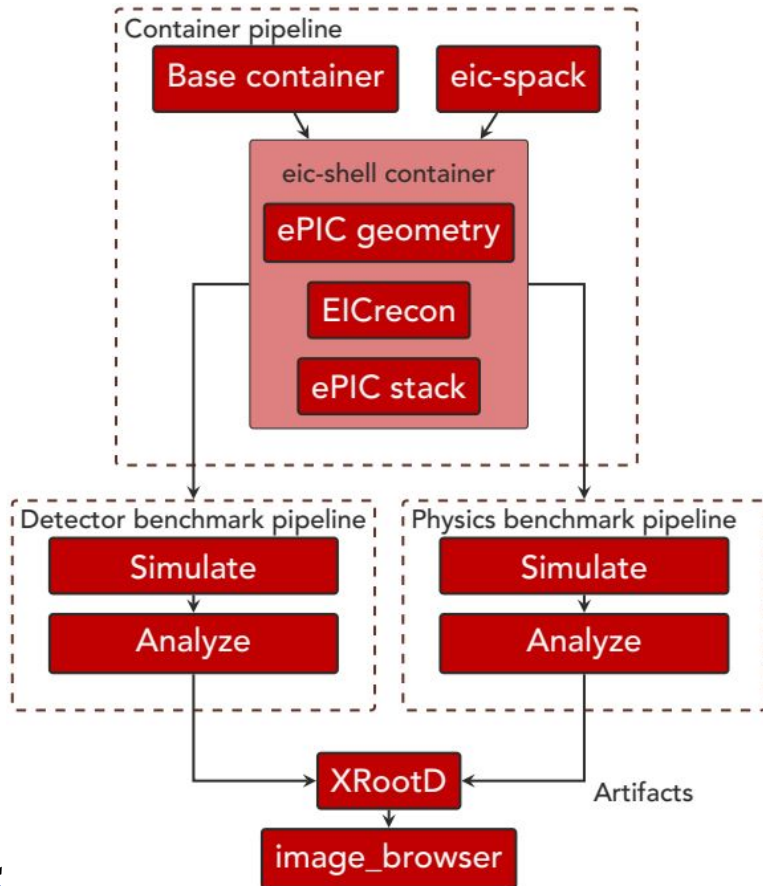
This workflow is run on every commit in any component developed by ePIC, and on every patch/upgrade of external components.

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

Benchmarking on Gitlab

Presentation Interface
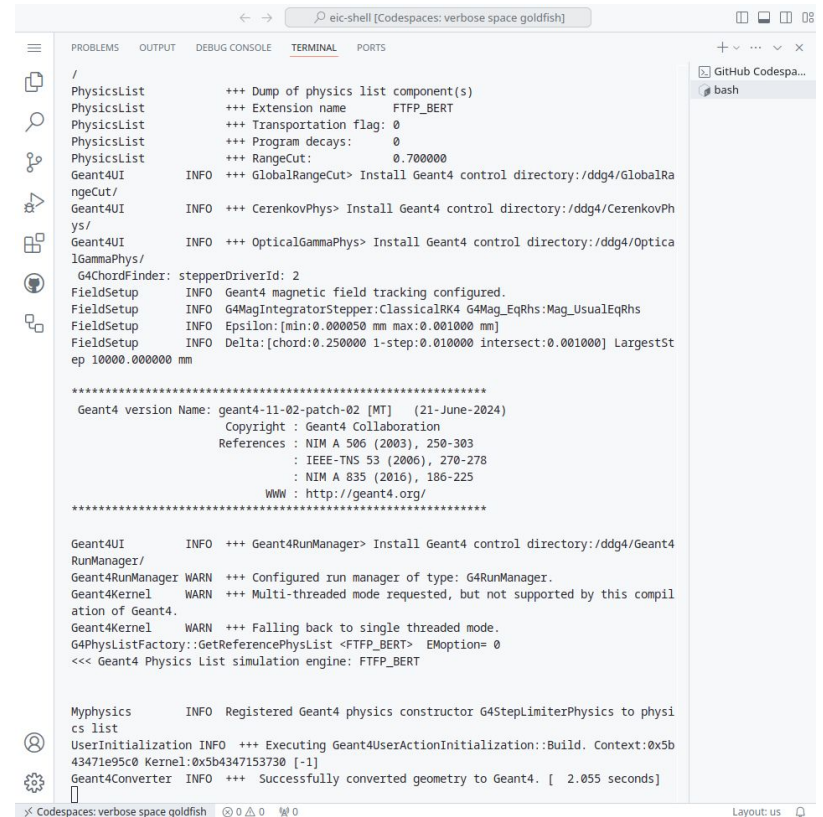
Containers at the Core of the EIC/ePIC Computing Approach

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

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 (no interactivity)

```
spack:
  include:
  - ../concretizer.yaml
  - ../packages.yaml
  specs:
  - dd4hep -ddeve
  - ...


dd4hep:
    require:
    - '@1.30'
    - +ddg4 +ddcad +hepmc3 +utilityapps
    - any_of: [+ddeve, -ddeve]
```

ghcr.io/eic/eic_xl (full interactivity)

```
spack:
  include:
  - ../concretizer.yaml
  - ../packages.yaml
  specs:
  - dd4hep +ddeve
  - ...
```

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
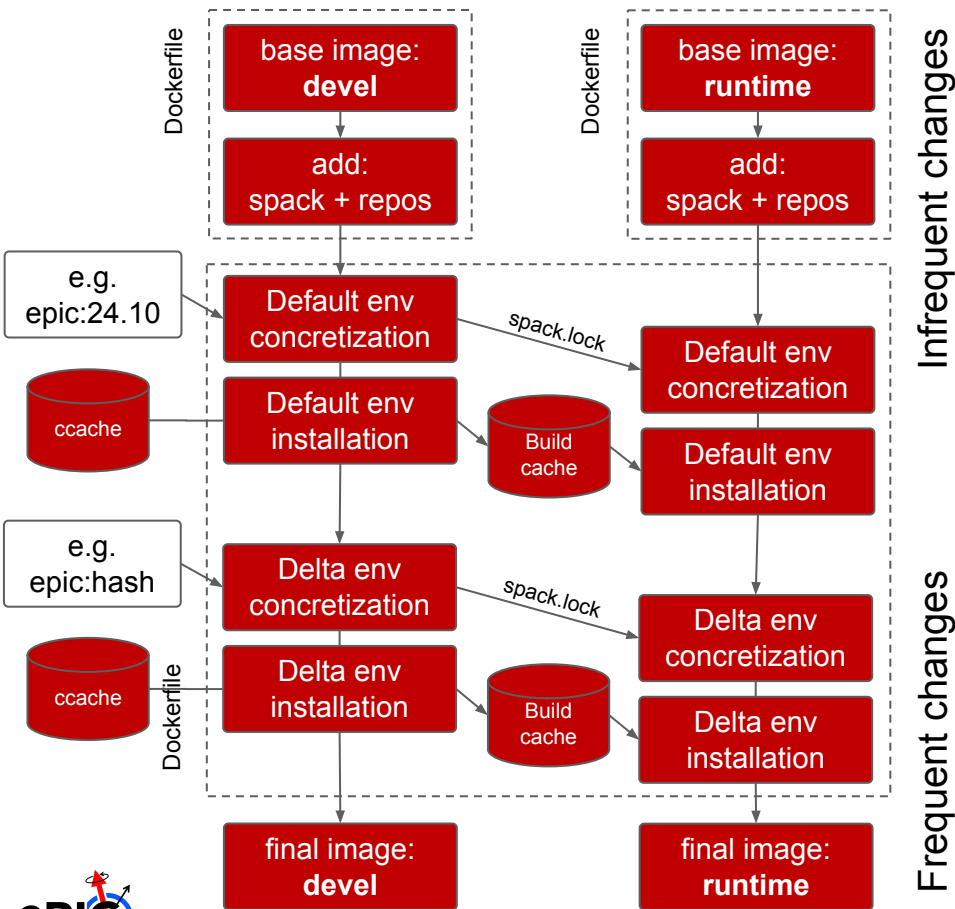
Challenge:

- 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
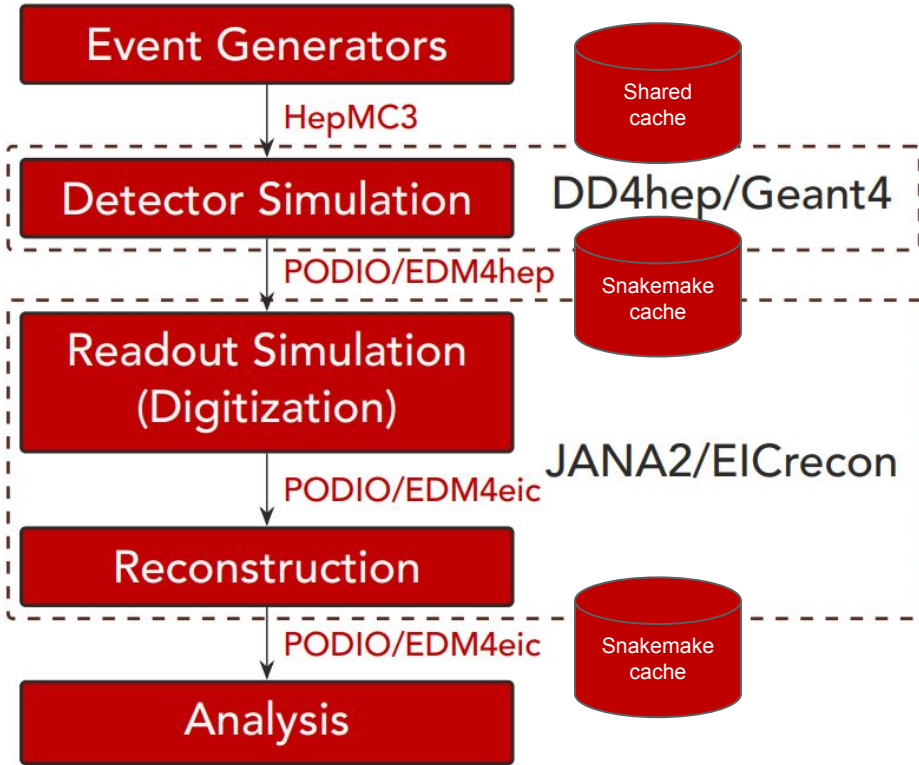
Validation and benchmarking jobs:

- 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
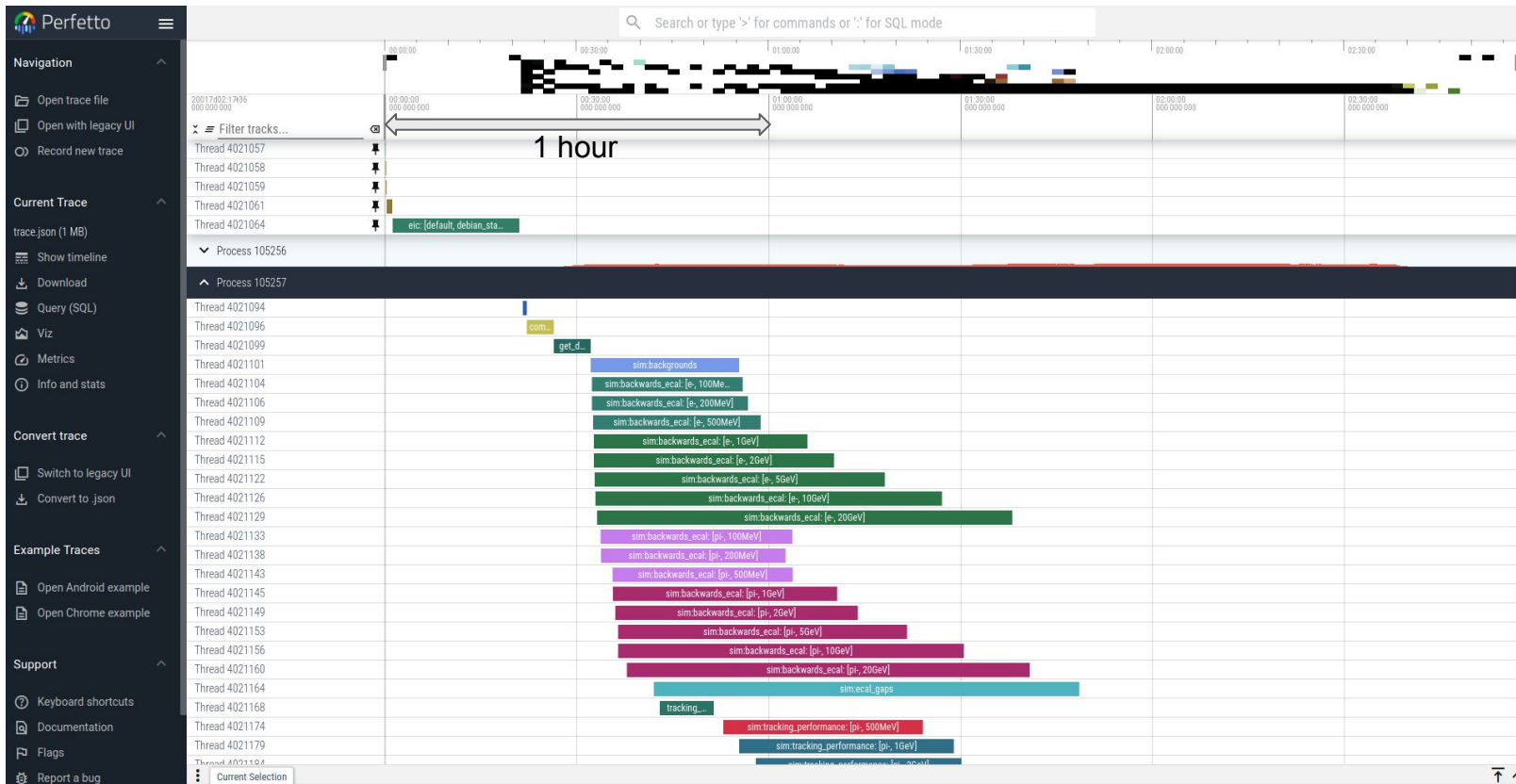
Container Build Latency (analysis benchmark changes): 8 minutes
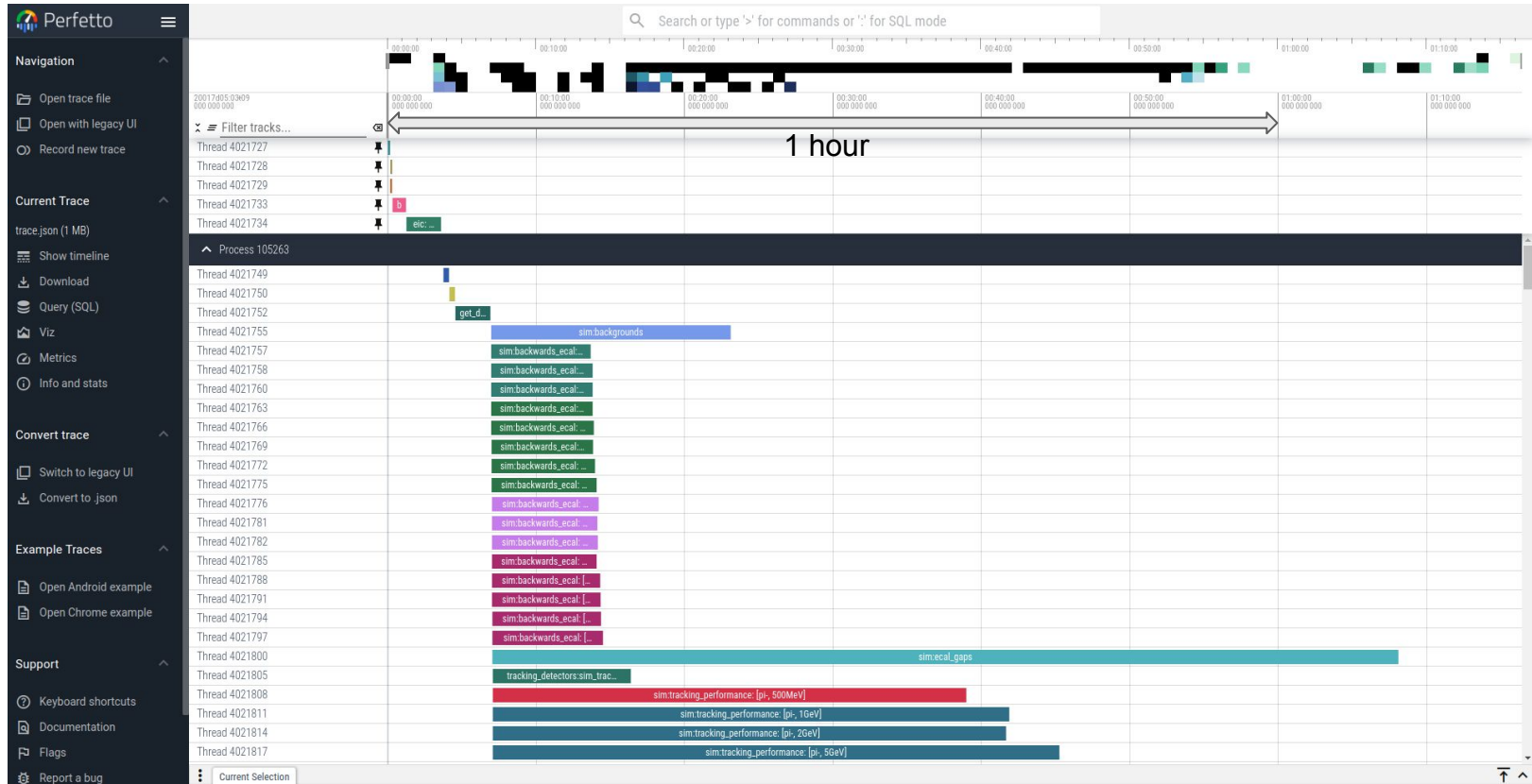
Opportunities for Further Improvements

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.