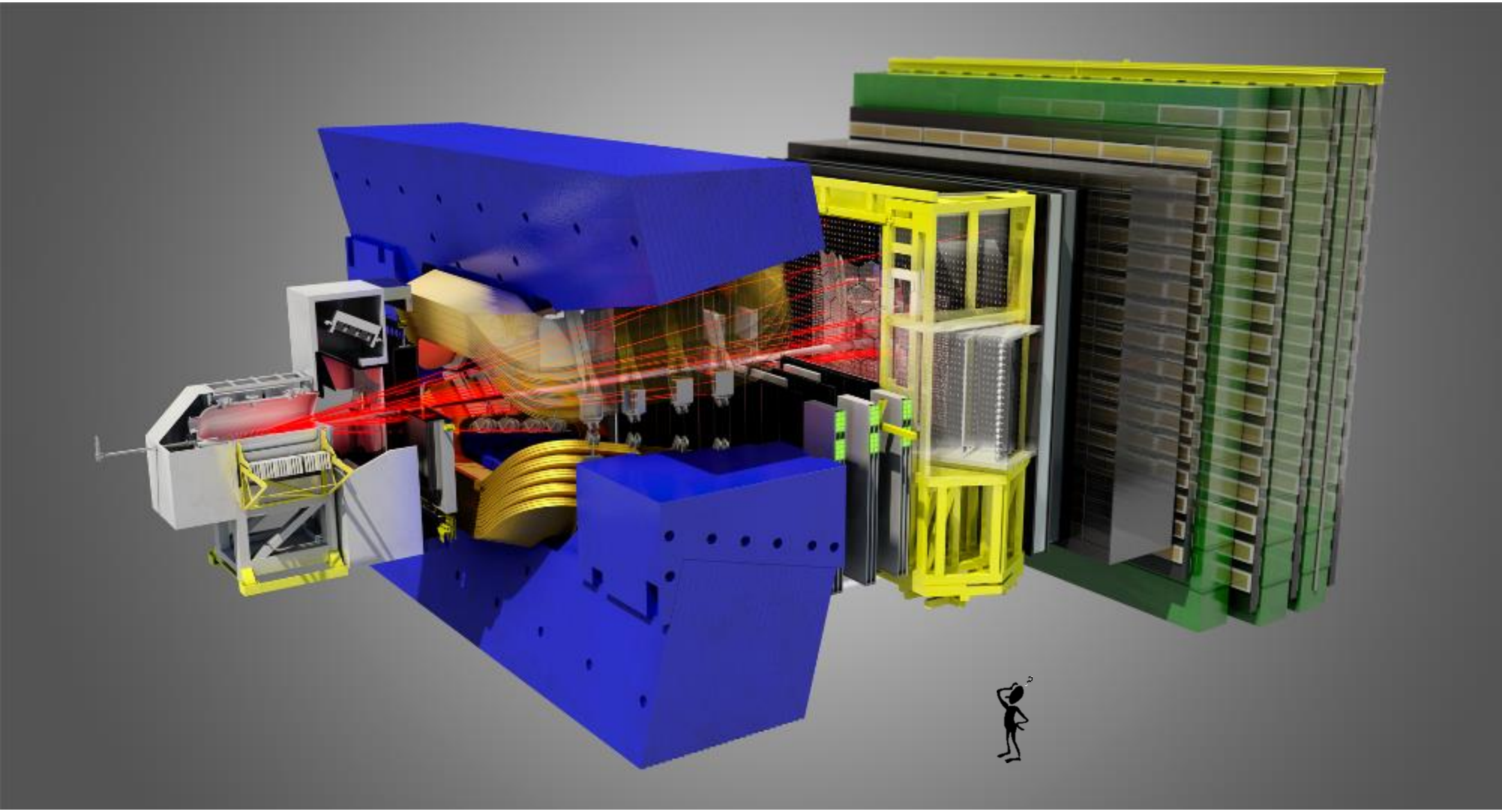
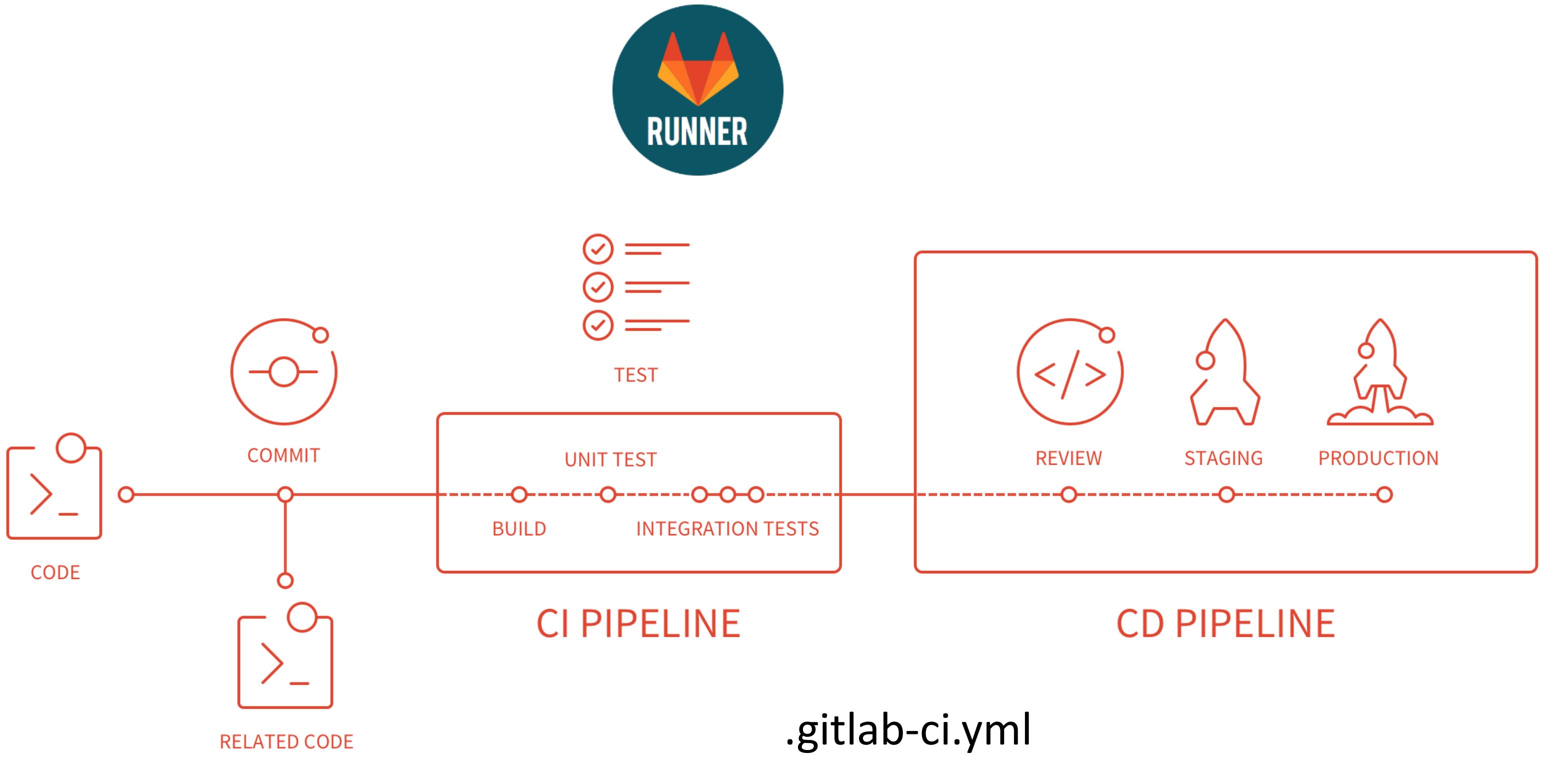
GitLab-CI for FPGA development at LHCb

The EP/LBC (Online) group

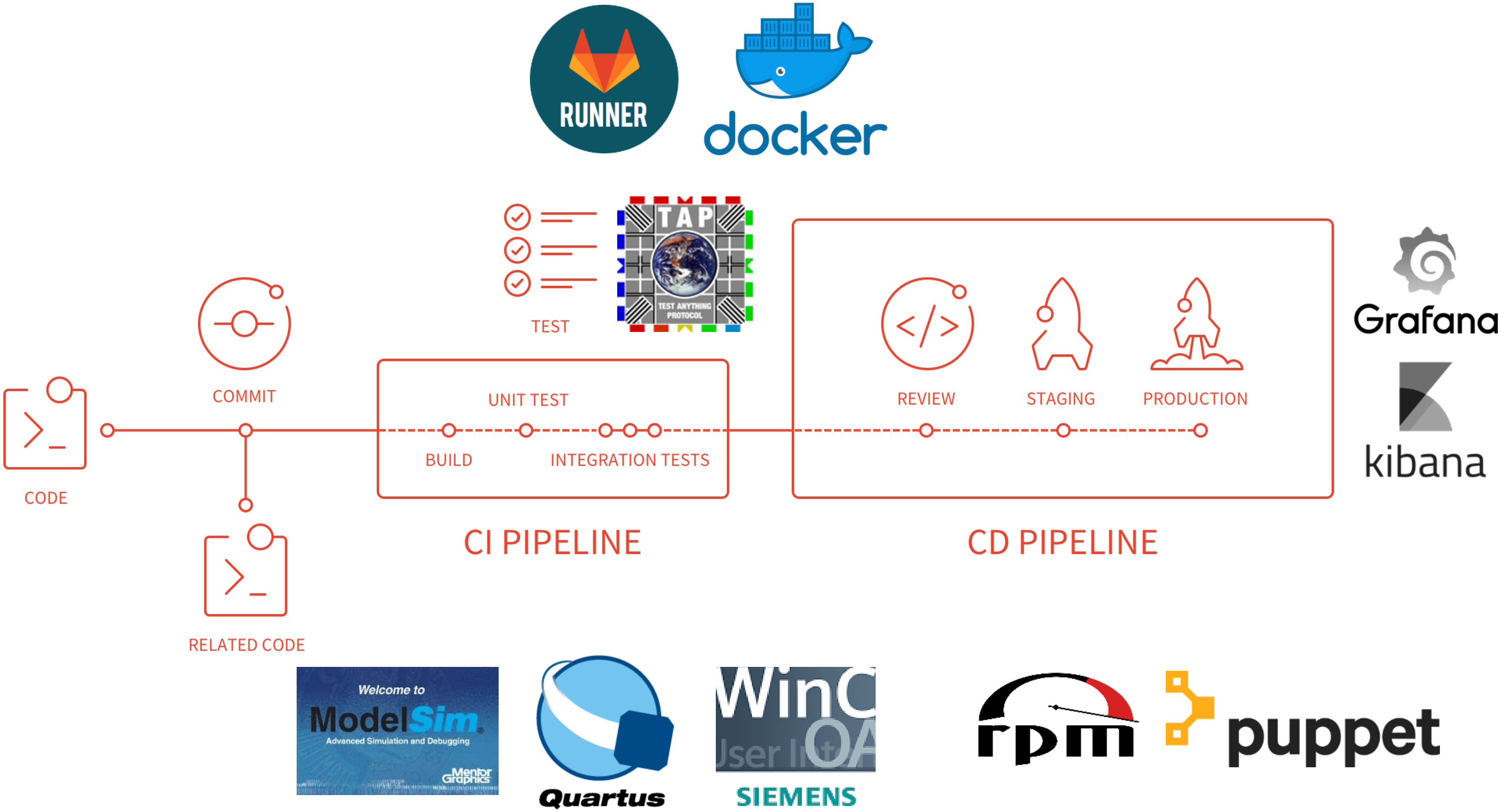


GitLab-CI (Continuous Integration)



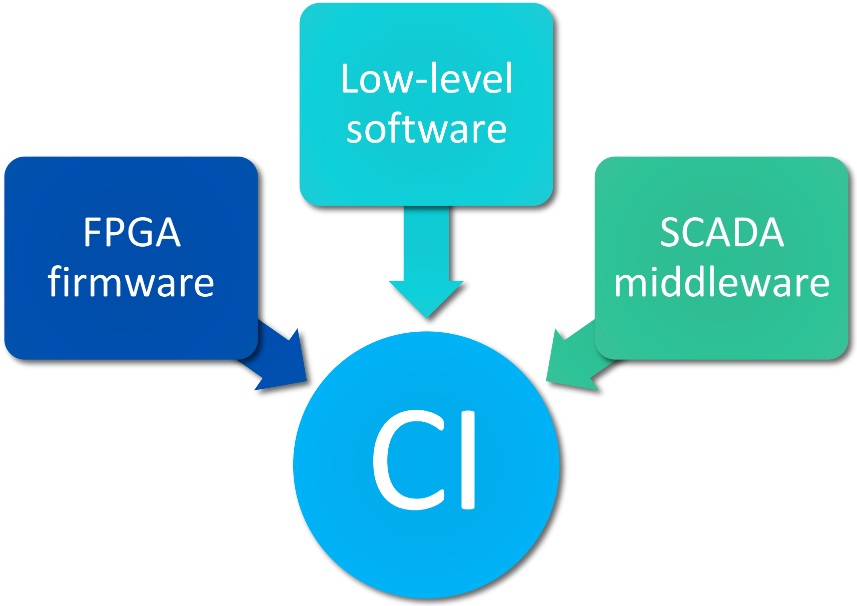
.gitlab-ci.yml

GitLab-CI at LHCb (FPGA development only)



Our environment

- FPGA firmware
  - Multiple repositories (submodules)
  - Languages: VHDL, Verilog, TCL
  - Toolchain: Questa, Quartus (proprietary)
  - CI: shell runners
- Low-level software
  - Multiple repositories (independent)
  - Languages: C, C++, Python
  - Toolchain: GCC, Python…
  - CI: shared runners
- SCADA middleware
  - One repository
  - Languages: CTRL (Siemens proprietary)
  - Toolchain: PVSS (proprietary)
  - CI: dind? [wip]
- Each has its own CI pipelines, but in the end we also have to test the integration of the different pieces working together

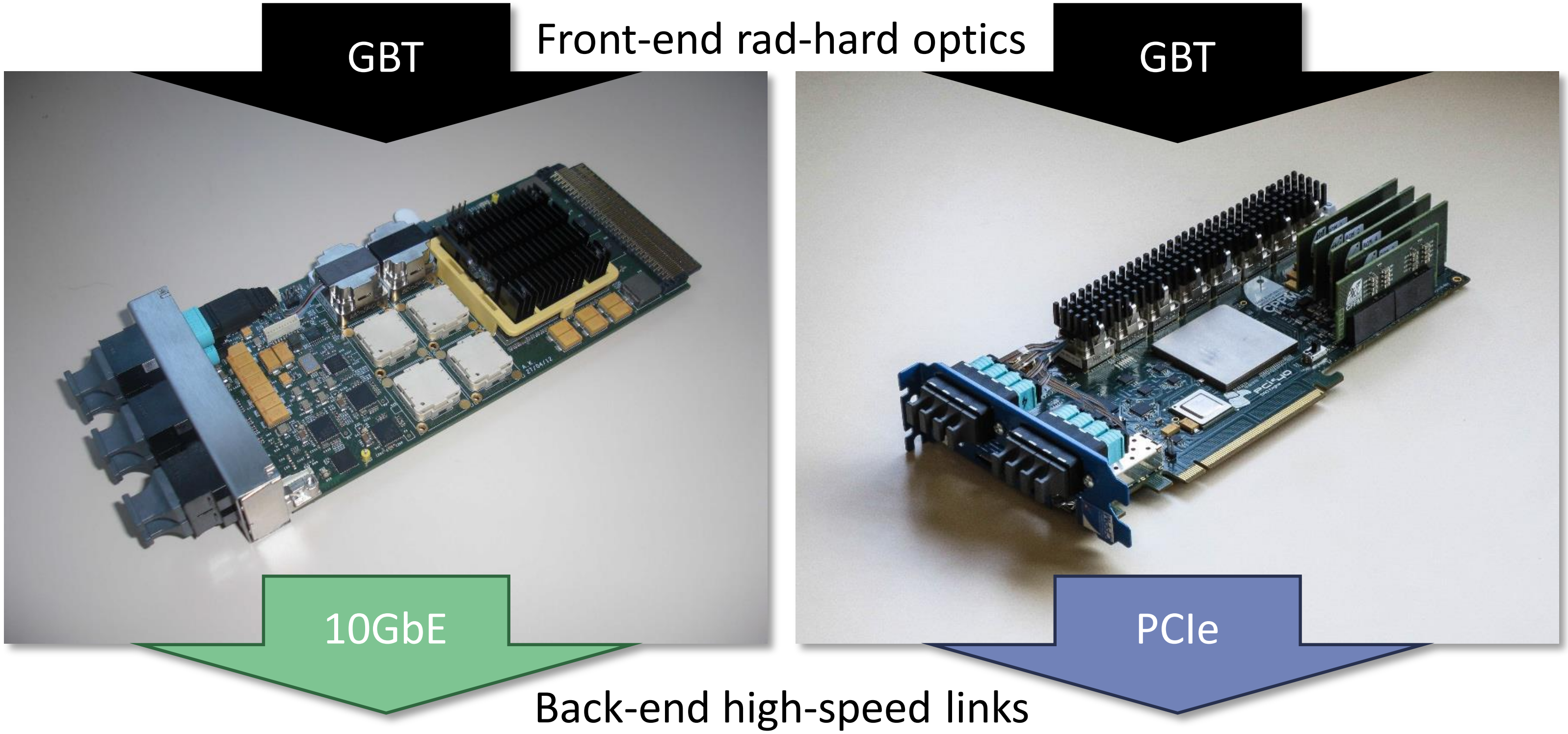


Target hardware

We receive data from the detectors (proprietary rad-hard protocols over optical fiber), process it and emit it in a "COTS-friendly" format.

We have to produce different firmware (sometimes more than one) for each device and for each sub-detector.

AMC40 (Legacy) **PCIe40 (Production)**



GBT

Front-end rad-hard optics

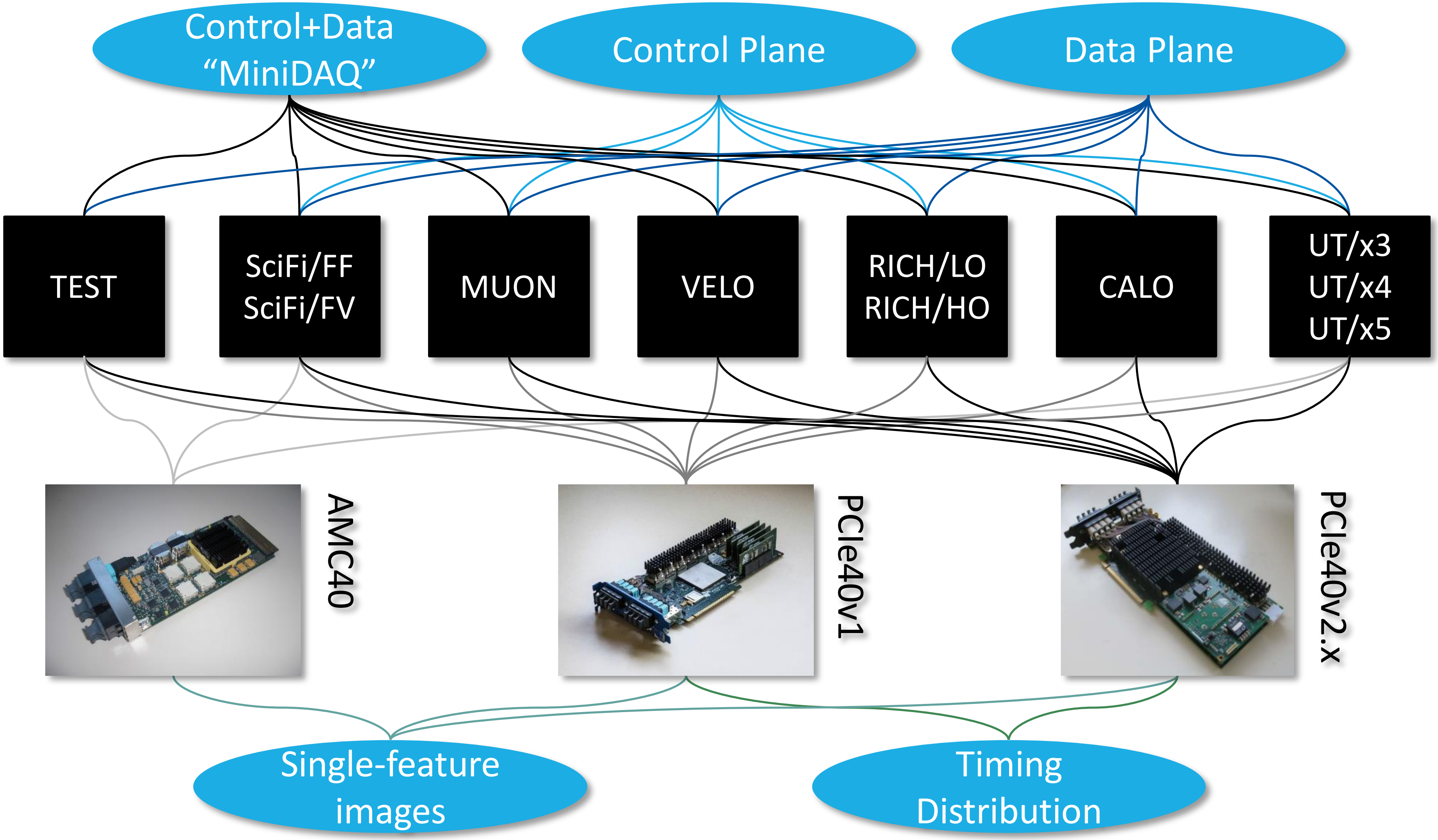
GBT

10GbE

PCIe

Back-end high-speed links

FW synthesis combinatorics



Control+Data “MiniDAQ”

Control Plane

Data Plane

TEST

SciFi/FF
SciFi/FV

MUON

VELO

RICH/LO
RICH/HO

CALO

UT/x3
UT/x4
UT/x5

AMC40

PCIe40v1

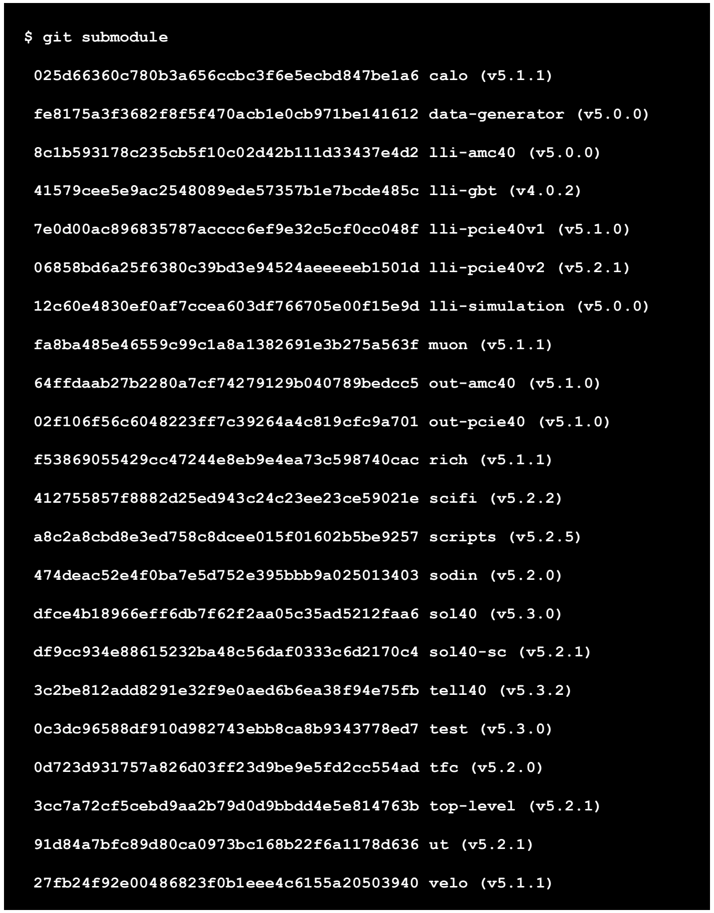
PCIe40v2.x

Single-feature images

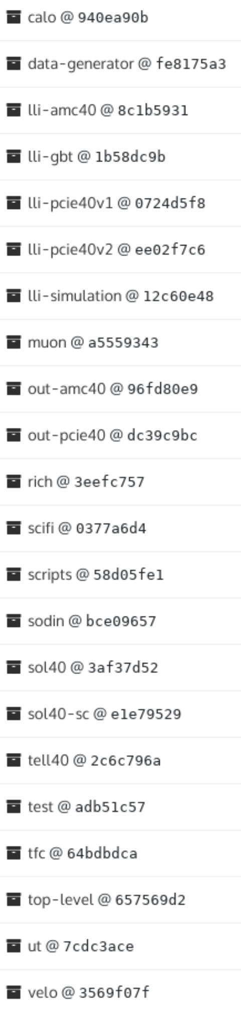
Timing Distribution

Firmware repository

- FPGA firmware split across different responsibility areas
- Different developers / institutes work in parallel on their respective component
- Each component is its own repository, versioned using git tags and semantic versioning
- One upstream repo tracks the global state of the firmware, each component is a git submodule
- Python scripts to manage submodules for users with little git experience
- CI runs EDA design flow against the upstream repo
- Submodules can trigger builds in the upstream repo for quick testing



```
$ git submodule
 025d66360c780b3a656ccbc3f6e5ecbd847be1a6 calo (v5.1.1)
 fe8175a3f3682f8f5f470acb1e0cb971be141612 data-generator (v5.0.0)
 8c1b593178c235cb5f10c02d42b111d33437e4d2 lli-amc40 (v5.0.0)
 41579cee5e9ac2548089ede57357b1e7bcde485c lli-gbt (v4.0.2)
 7e0d00ac896835787acccc6ef9e32c5cf0cc048f lli-pcie40v1 (v5.1.0)
 06858bd6a25f6380c39bd3e94524aeeeeeb1501d lli-pcie40v2 (v5.2.1)
 12c60e4830ef0af7ccea603df766705e00f15e9d lli-simulation (v5.0.0)
 fa8ba485e46559c99c1a8a1382691e3b275a563f muon (v5.1.1)
 64ffdaab27b2280a7cf74279129b040789bedcc5 out-amc40 (v5.1.0)
 02f106f56c6048223ff7c39264a4c819cfc9a701 out-pcie40 (v5.1.0)
 f53869055429cc47244e8eb9e4ea73c598740cac rich (v5.1.1)
 412755857f8882d25ed943c24c23ee23ce59021e scifi (v5.2.2)
 a8c2a8cbd8e3ed758c8dcee015f01602b5be9257 scripts (v5.2.5)
 474deac52e4f0ba7e5d752e395bbb9a025013403 sodin (v5.2.0)
 dfce4b18966eff6db7f62f2aa05c35ad5212faa6 sol40 (v5.3.0)
 df9cc934e88615232ba48c56daf0333c6d2170c4 sol40-sc (v5.2.1)
 3c2be812add8291e32f9e0aed6b6ea38f94e75fb tell40 (v5.3.2)
 0c3dc96588df910d982743ebb8ca8b9343778ed7 test (v5.3.0)
 0d723d931757a826d03ff23d9be9e5fd2cc554ad tfc (v5.2.0)
 3cc7a72cf5cebd9aa2b79d0d9bbdd4e5e814763b top-level (v5.2.1)
 91d84a7bfc89d80ca0973bc168b22f6a1178d636 ut (v5.2.1)
 27fb24f92e00486823f0b1eee4c6155a20503940 velo (v5.1.1)
```



calo @ 940ea90b

data-generator @ fe8175a3

lli-amc40 @ 8c1b5931

lli-gbt @ 1b58dc9b

lli-pcie40v1 @ 0724d5f8

lli-pcie40v2 @ ee02f7c6

lli-simulation @ 12c60e48

muon @ a5559343

out-amc40 @ 96fd80e9

out-pcie40 @ dc39c9bc

rich @ 3eefc757

scifi @ 0377a6d4

scripts @ 58d05fe1

sodin @ bce09657

sol40 @ 3af37d52

sol40-sc @ e1e79529

tell40 @ 2c6c796a

test @ adb51c57

tfc @ 64bdbdca

top-level @ 657569d2

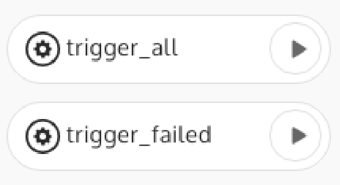
ut @ 7cdc3ace

velo @ 3569f07f

Firmware integration flow

DEVELOPER

1. Push changes to one or more components
2. Click a manual action in GitLab to build new changes (or wait for the nightly build)

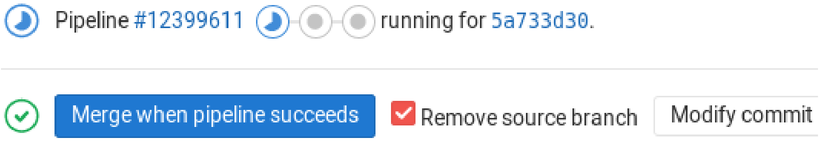


3. When satisfied, run a script that will generate tags and submit the new components upstream (a merge request is automatically created)

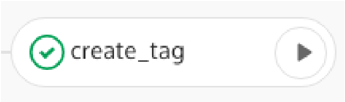
```
./scripts/git/release --minor --merge mycomponent/
```

MAINTAINER

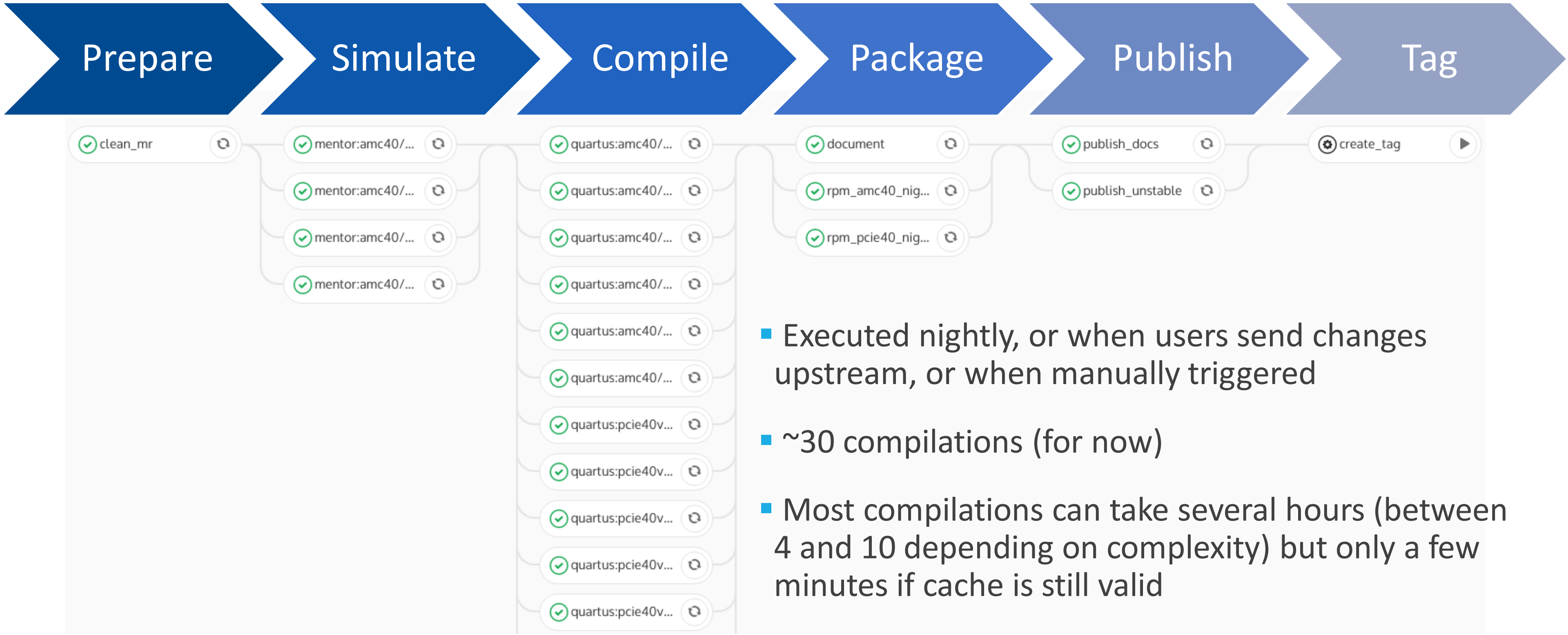
1. Receive a merge request (a pipeline is automatically started)
2. If the change is minor, can decide to automatically merge when the pipeline completes


Pipeline #12399611 running for 5a733d30.

3. If not, wait for pipeline to publish firmware RPMs, go to a development machine, run *yum install lhcb-pcie40-firmware-merge-<MR#>* and test it
4. Push fixes to merge request until it can be accepted
5. Once merged, click a manual action in GitLab to automatically create and publish a new tag



Firmware pipeline



- Executed nightly, or when users send changes upstream, or when manually triggered
- ~30 compilations (for now)
- Most compilations can take several hours (between 4 and 10 depending on complexity) but only a few minutes if cache is still valid
- Custom scripts to avoid a rebuild whenever possible (EDA tools have poor dependency tracking)

(30m ~ 20h)

Firmware lessons learned

- **Resources**
  Describe simulation and compilation resources available in terms of gitlab-runner tags,
  e.g. *lhcb-daq40-firmware-quartus161-32G, lhcb-daq40-firmware-cache, lhcb-daq40-firmware-publish*

- **Simulations**
  Pool resources from other institutes within the LHCb collaboration

- **Concurrent jobs**
  Use `rsync(1)` and `flock(1)` to prevent Quartus cache corruption

- **Long-running jobs**
  Write cron scripts to keep idle Quartus jobs alive (as gitlab-runner spontaneously terminates them after 1 hour)

- **Dependencies**
  Keep Quartus project reports, write scripts to compare git histories between rebuilds

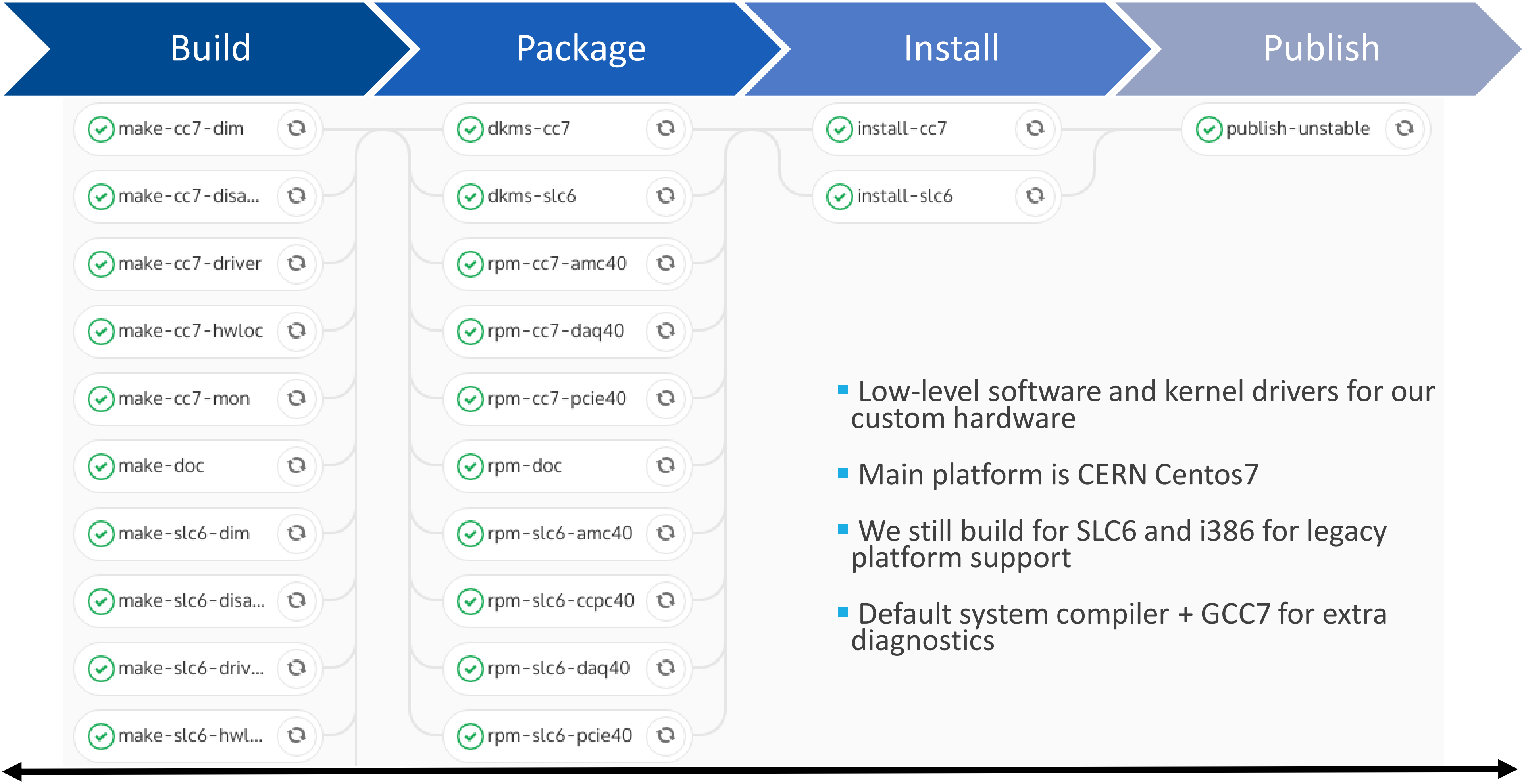
- **Merges**
  Automatically create MR based on branch name, reuse build cache after merge, fast-forward upstream merges

- **Branches**
  Branches with the same name across components are triggered together (using GitLab CI variables)

- **Deployment**
  Distribute firmware, WinCC components, software etc. as RPMs and write tools to install/program automatically

- **Traceability**
  Store git version information in the FPGA image automatically, and write tools to extract this info in the field

Software pipeline



Build:
- ✓ make-cc7-dim
- ✓ make-cc7-disa...
- ✓ make-cc7-driver
- ✓ make-cc7-hwloc
- ✓ make-cc7-mon
- ✓ make-doc
- ✓ make-slc6-dim
- ✓ make-slc6-disa...
- ✓ make-slc6-driv...
- ✓ make-slc6-hwl...

Package:
- ✓ dkms-cc7
- ✓ dkms-slc6
- ✓ rpm-cc7-amc40
- ✓ rpm-cc7-daq40
- ✓ rpm-cc7-pcie40
- ✓ rpm-doc
- ✓ rpm-slc6-amc40
- ✓ rpm-slc6-ccpc40
- ✓ rpm-slc6-daq40
- ✓ rpm-slc6-pcie40

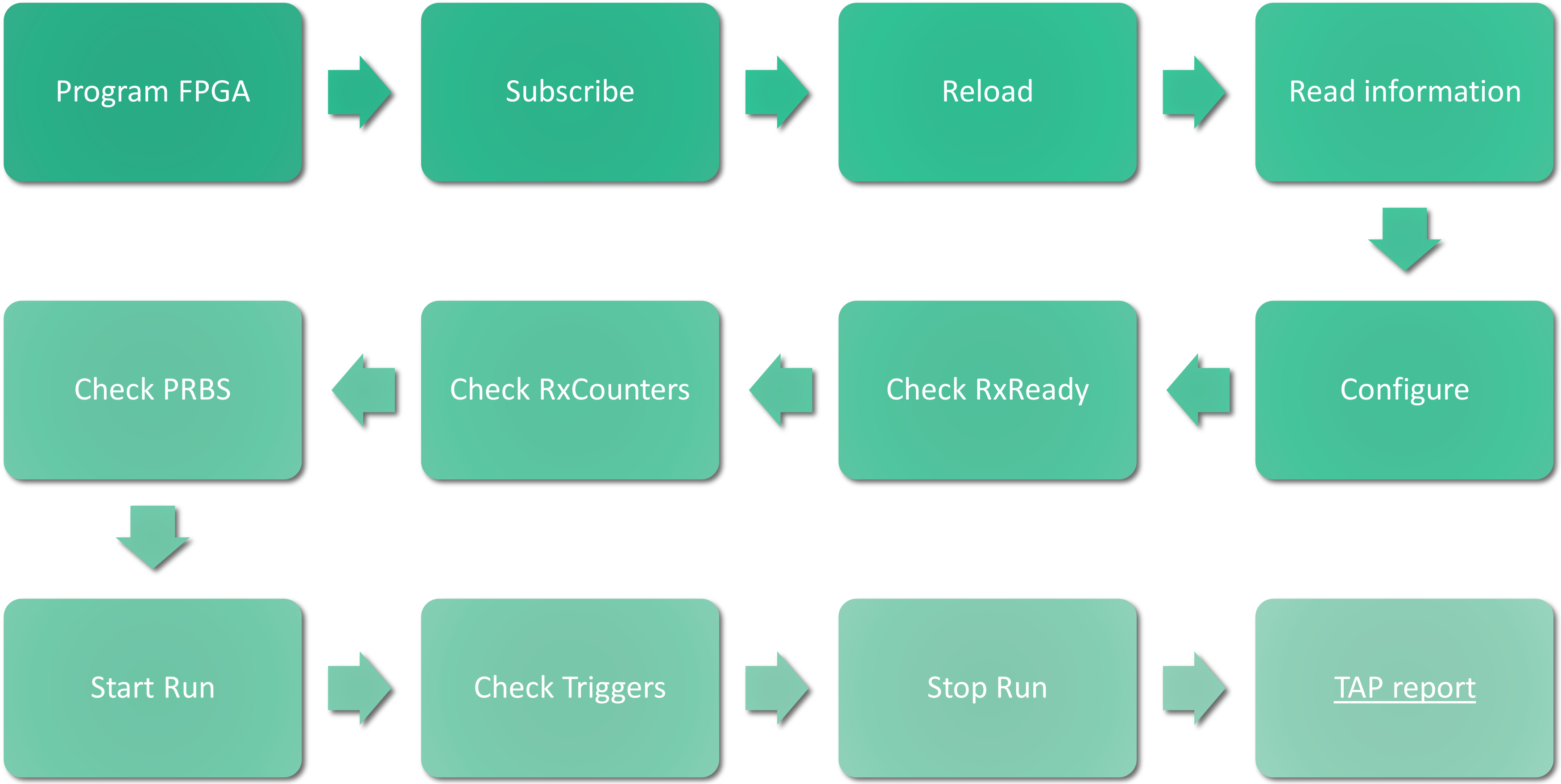
Install:
- ✓ install-cc7
- ✓ install-slc6

Publish:
- ✓ publish-unstable

(~10m)

- Low-level software and kernel drivers for our custom hardware
- Main platform is CERN Centos7
- We still build for SLC6 and i386 for legacy platform support
- Default system compiler + GCC7 for extra diagnostics

WinCC-OA "pipeline"



Program FPGA

Subscribe

Reload

Read information

Configure

Check RxReady

Check RxCounters

Check PRBS

Start Run

Check Triggers

Stop Run

TAP report

Final remarks

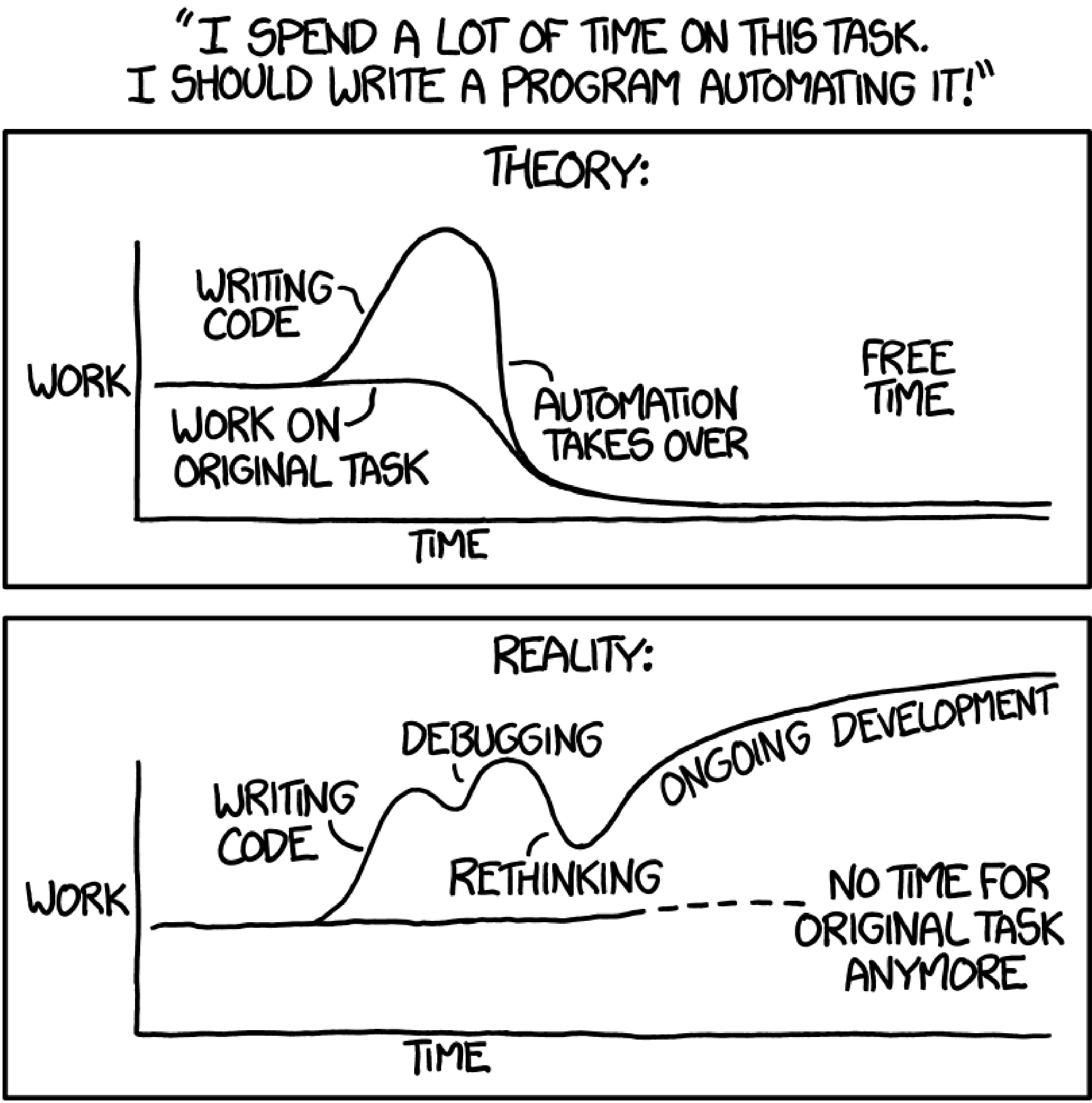
- Overall, our experience with GitLab has been very positive!
  - GitLab REST API is powerful enough to let us customize the CI flow to our needs
- I’m slowly converting our group to use GitLab CI
  - Some “devops” attitude is required, but we make the system as easy to use as possible
  - Some of these tools are still unfamiliar outside the domain of software engineering
  - Integrating with EDA and SCADA toolchains requires extra effort
- Work in progress
  - Integrate simulation output validator in pipeline
  - Automate hardware-in-the-loop integration tests
- Eventually
  - Deploy/rollback using GitLab environments
  - Publish metrics and monitor hardware performance in a live system (Grafana/Kibana)

DIY (Do It Yourself)

You will need:

1. GitLab CI
   - Centrally provided by IT-CDA
   - Includes shared Docker runners
2. Your toolchain packaged in a way that GitLab can use (Quartus/Vivado/etc.)
   - Could be centrally provided by IT-CDA?
   - Or: build your own Docker images
   - Or: set up private runners on a physical or virtual machine (e.g. on CERN OpenStack)
3. A way to share artifacts across pipelines (if you want to reuse builds)
   - E.g.: some NFS/SMB filesystem running on CERN OpenStack
   - Or: GitLab CI provides an internally managed cache since version 9.0
     - We have no experience with it however, and our cache is currently around 200 GB
4. Patience to write and debug lots of scripts! (it pays off, eventually)
   - In addition to TCL, we use python a lot
     - Python-gitlab for GitLab API integration
     - GitPython to automate GIT repository manipulation

Thank you for your attention



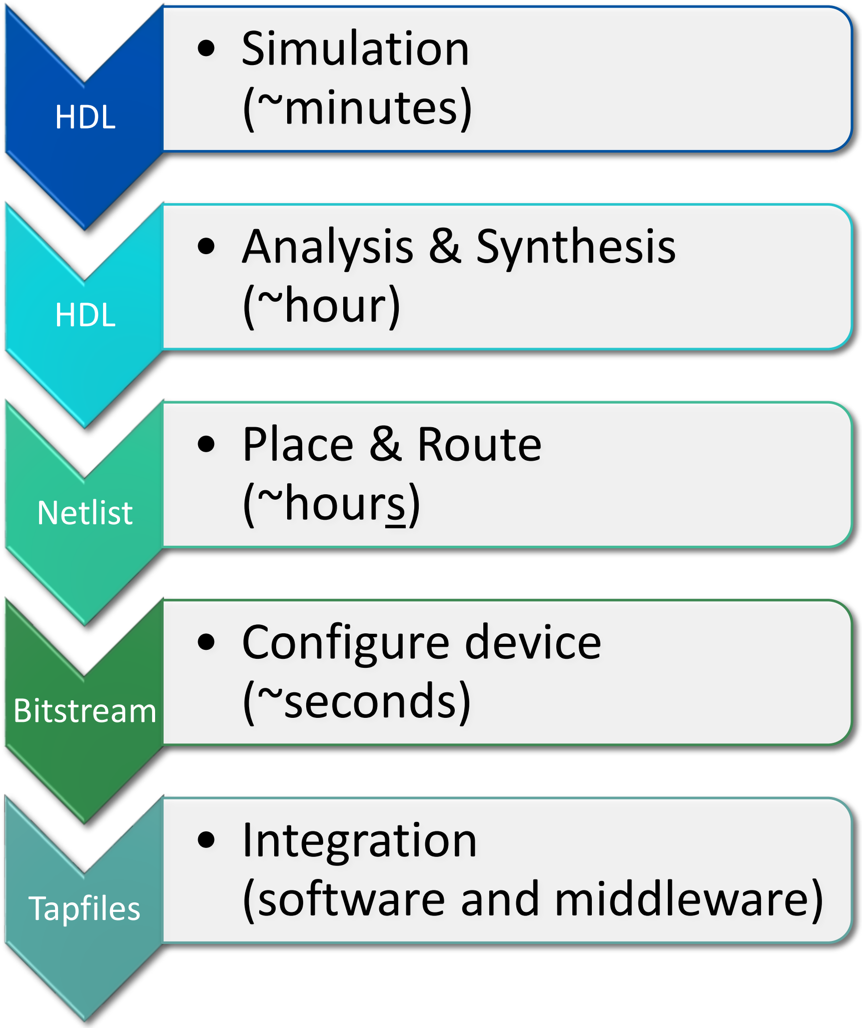
GitLab issues

1. https://gitlab.com/gitlab-org/gitlab-ce/issues/38265
   StuckCiJobsWorker wrongly detects, cancels 'stuck' builds when per-job timeout is more than an hour
   - Some of our jobs can easily require several hours
   - We have less machines than we have jobs
   - For now, we set up cronjobs running in background that use the GitLab API to find jobs that have been killed for inactivity and resuscitate them

2. https://gitlab.com/gitlab-org/gitlab-ce/issues/37356
   Relative submodule link to a nested project fails to resolve
   - Our firmware flow relies on submodules
   - A user should be able to click on a submodule and open the corresponding repository at the correct commit state
   - ~~It's already fixed on gitlab.com but not yet in the instance deployed at CERN~~

FPGA development 101

- Circuit is defined in a Hardware Description Language
- Simulator + testbench + input vectors reproduce behavior of device and allow some form of debugging
- Compiler toolchain maps specification into device-specific gate array configuration and interconnection
  - Lots of TCL scripts
  - 48 GB RAM!! (vendor recommendation)
  - As many GHz as you can afford
  - Still very time consuming
- Workarounds
  - Distribute different compilations to different machines
  - Reuse artifacts aggressively


HDL
• Simulation (~minutes)
HDL
• Analysis & Synthesis (~hour)
Netlist
• Place & Route (~hours)
Bitstream
• Configure device (~seconds)
Tapfiles
• Integration (software and middleware)

Firmware pipeline - stages

Jobs in “Prepare” stage

- create_mr
  - Users run a script to submit changes upstream
  - Pipeline picks up the new branch and creates a merge request automatically
- cache_master
  - Previous compilations are preserved and cached
  - Custom TCL script during FPGA synthesis checks project against git history for modifications
- clean_mr
  - Ensure the merged branch is deleted
  - Update cache for master branch with last compilation

Jobs in “Tag” stage

- create_tag
  - Manual job
  - When maintainer wants to create a new release, a tag is created according to our naming convention
- clean_tag
  - Remove project cache for tag once firmware has been released
  - Every cache amounts to tens of gigabytes