

Machine Learning Operations for fast, online inference



NextGen
Next Generation Triggers

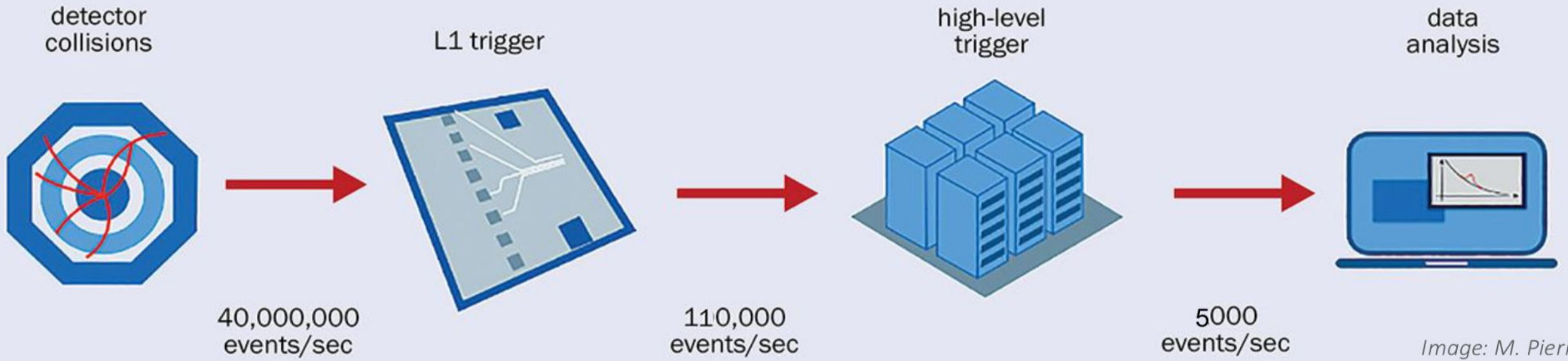
Welcome

- All domains are witnessing a shift from heuristic algorithms to Machine Learning
 - These are trained rather than explicitly programmed, which brings about a unique set of challenges we will discuss today
- Machine learning deployment into heterogeneous compute environments is on the rise

One noticeable example is High Energy Physics

- As the High-Luminosity LHC approaches and ML adaptation grows, attention is shifting towards the operational demands of deploying Machine Learning at large scale into the Level 1 Trigger system.
- We are looking to collaborate with the FastML community and take off-the-shelf solutions where possible!

Introduction to CMS Trigger



The CMS experiment at the LHC deploys a two-step trigger system to filter a 40 MHz proton-proton collision rates down to 100 KHz for offline analysis.

The Level 1 Trigger (L1T) ^[1] filters **99.75%** of collision events
If we don't identify interesting events in trigger, we lose them forever!

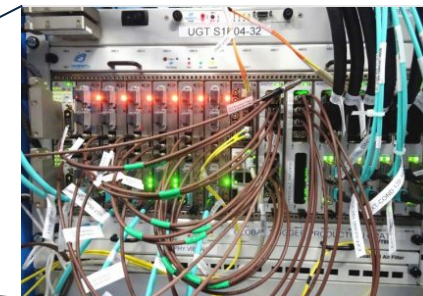
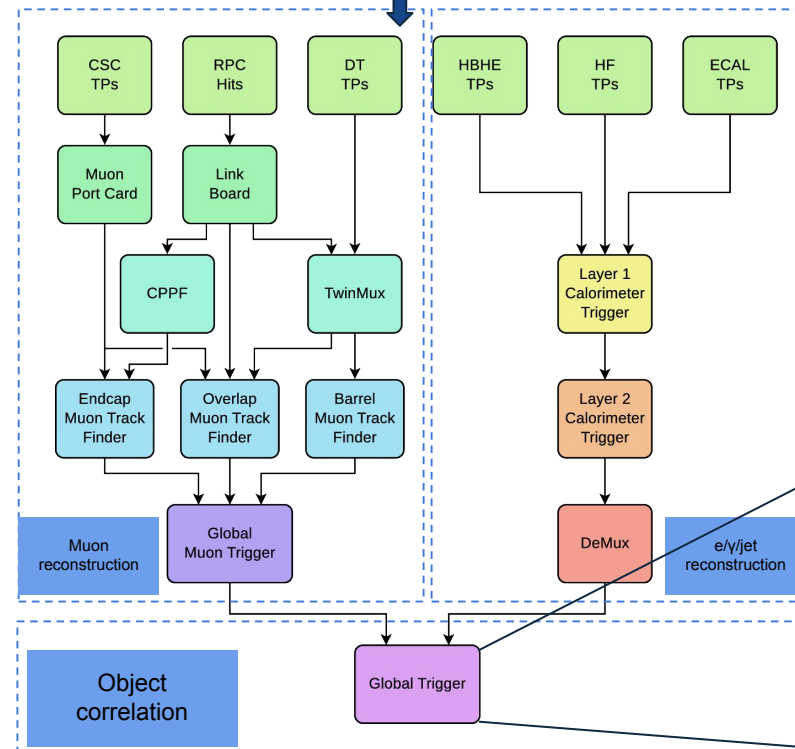
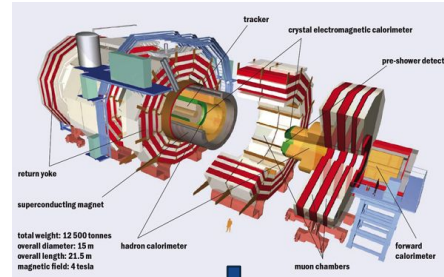
Introduction to Level 1 Trigger

The L1T receives condensed event information in the form of trigger primitives.

Dedicated subsystem modules reconstruct physics objects from varying detectors and/or regions.

L1T is an all FPGA design hosted on custom boards interconnected via GB/s optical links.

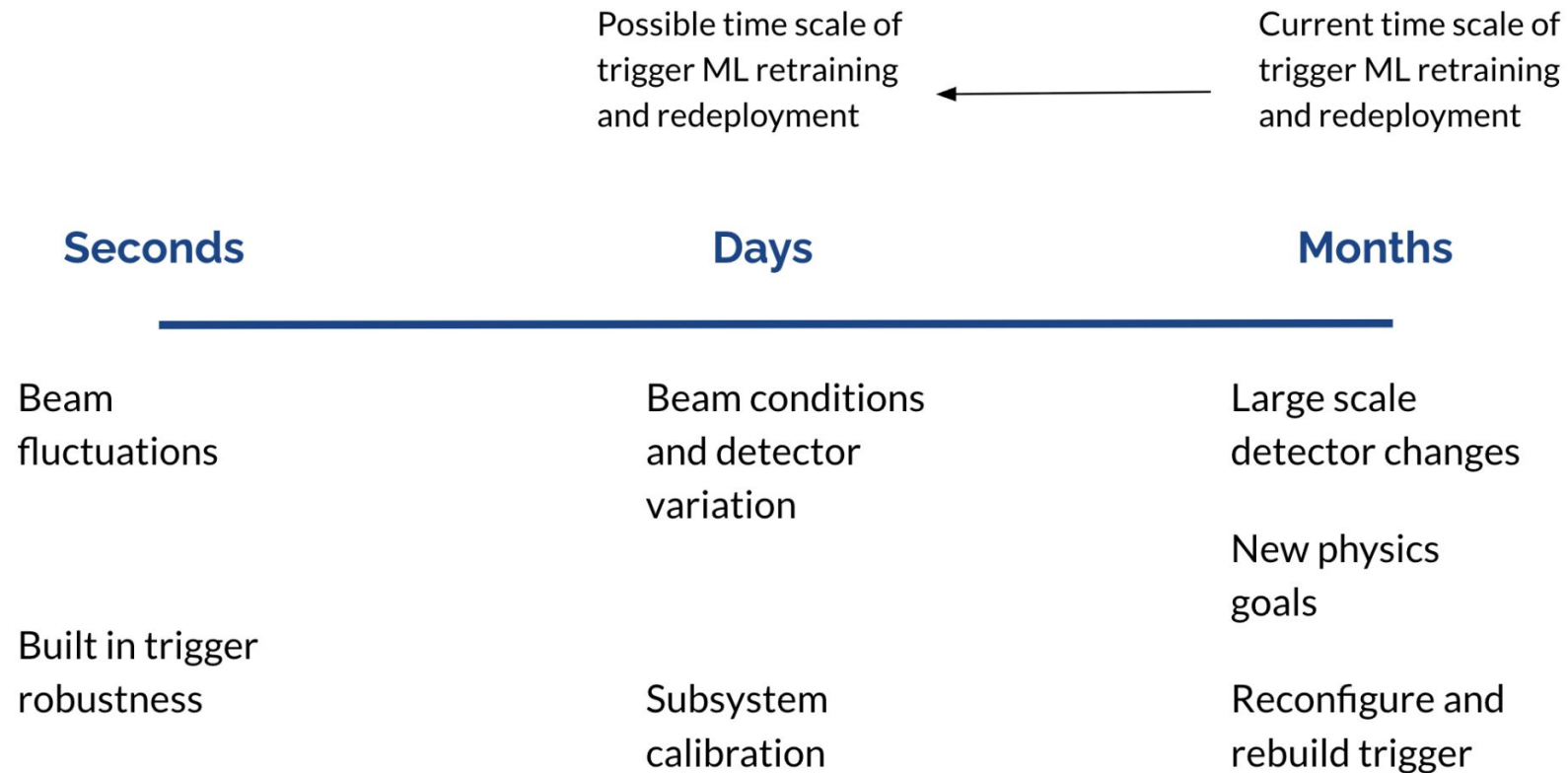
The final L1T decision (L1T accept/reject) is propagated to the Data Acquisition System (DAQ).



6 MP7 [2] cards hosting Xilinx Virtex 7 FPGA

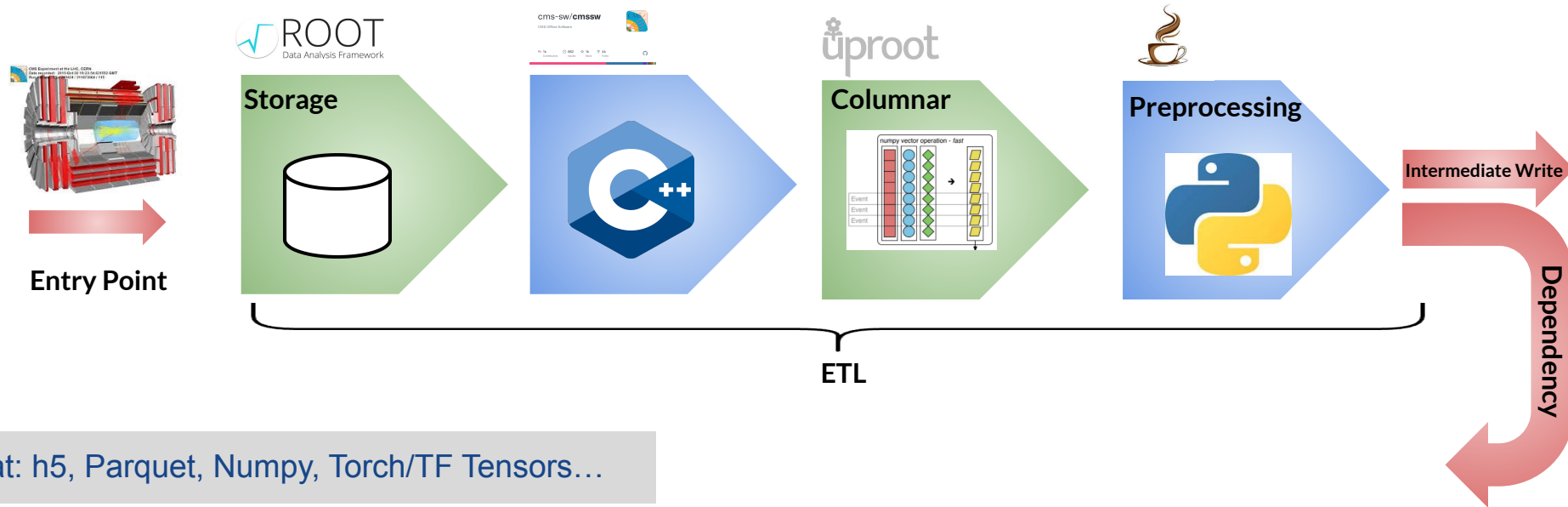
From Months to Days (to seconds?)

Timelines for deploying Machine Learning into the hardware triggers



From Christopher Brown

Deployment pipeline

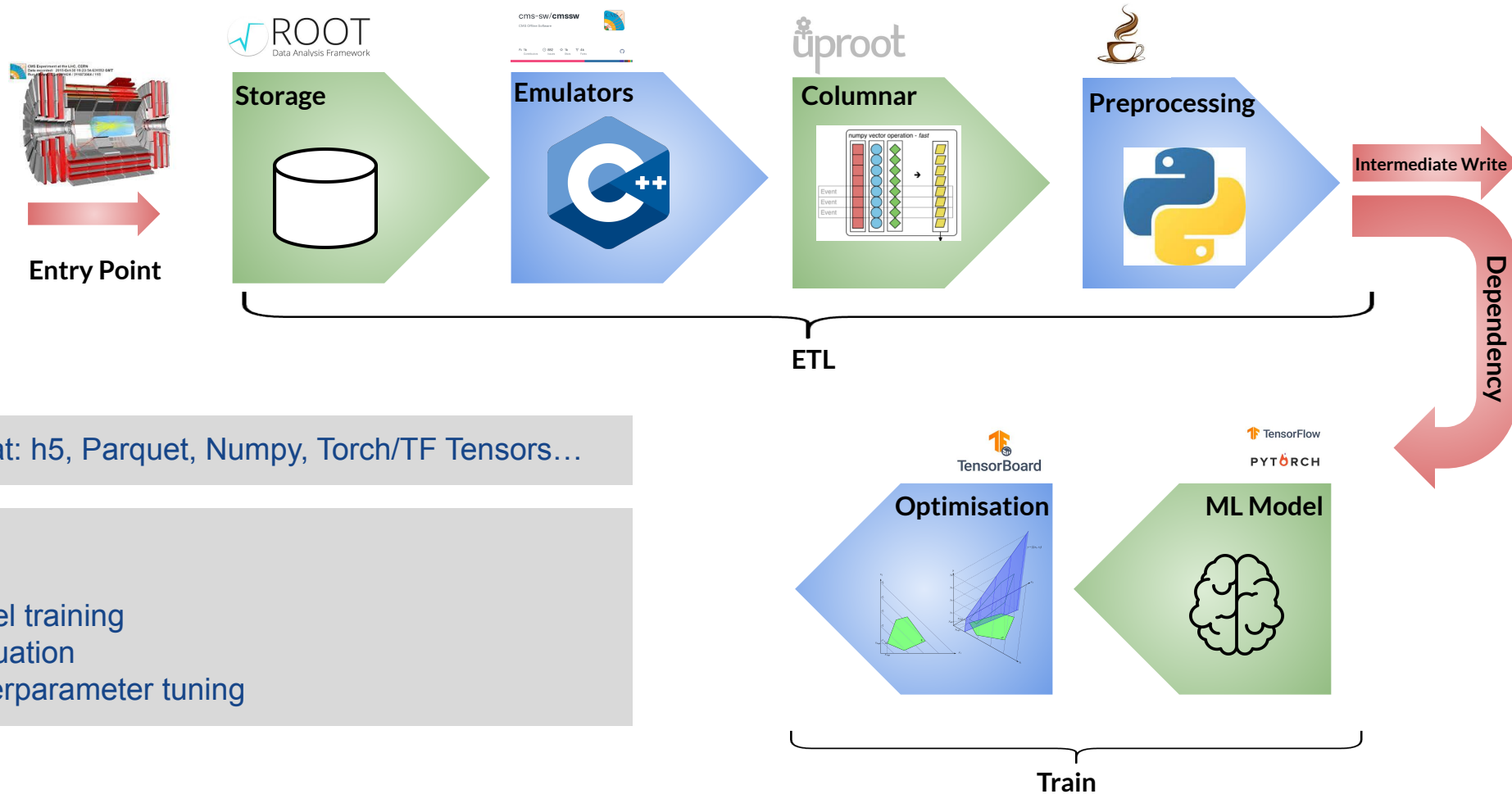


Data format: h5, Parquet, Numpy, Torch/TF Tensors...

Pre-processing

- ROOT unpacking
- Vectorisation
- Data transforms

Deployment pipeline



Data format: h5, Parquet, Numpy, Torch/TF Tensors...

Training

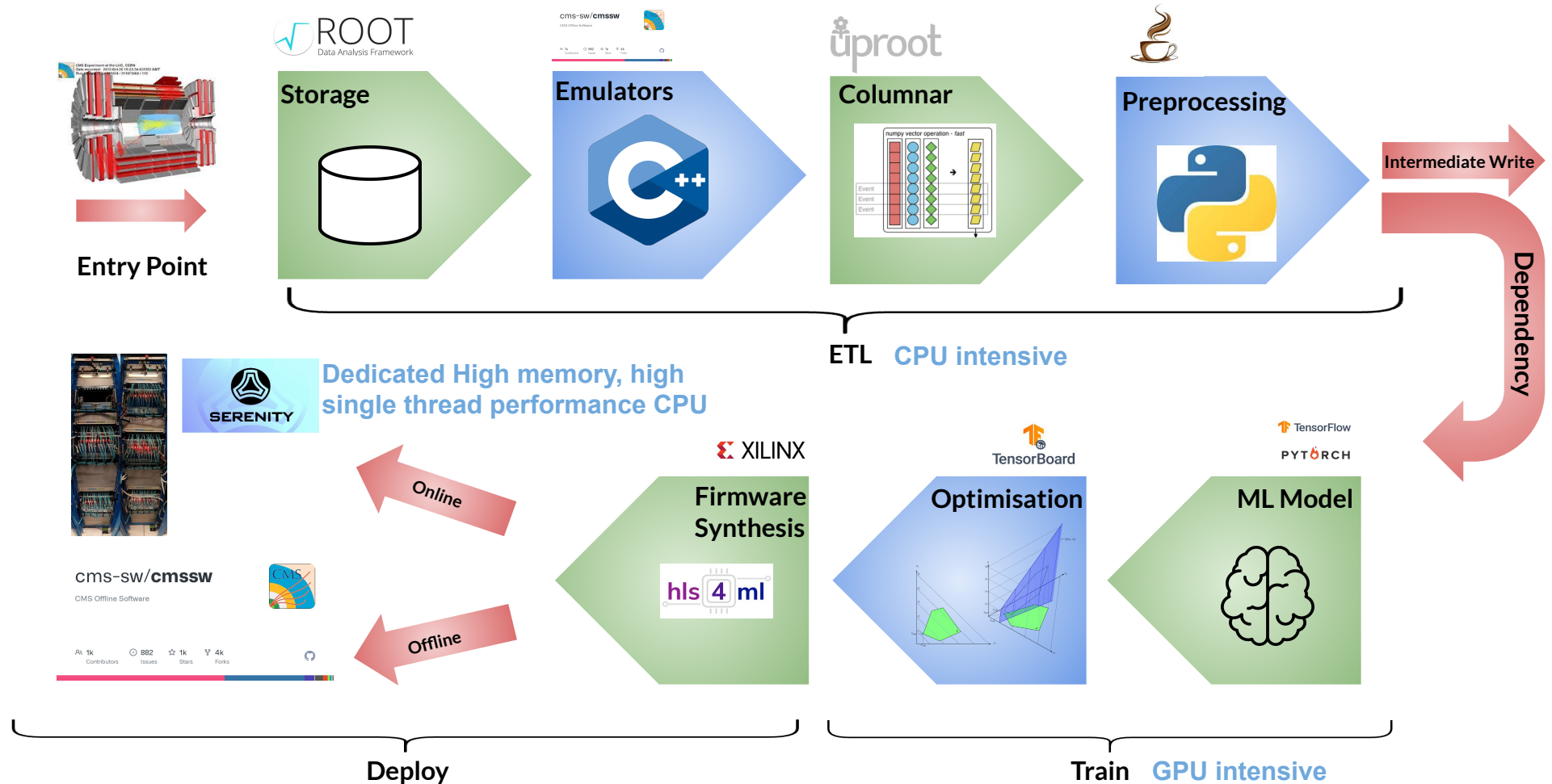
- Model training
- Evaluation
- Hyperparameter tuning

Deployment pipeline

Data format: h5, Parquet, Numpy, Torch/TF Tensor, ROOT, pattern file (raw binary)

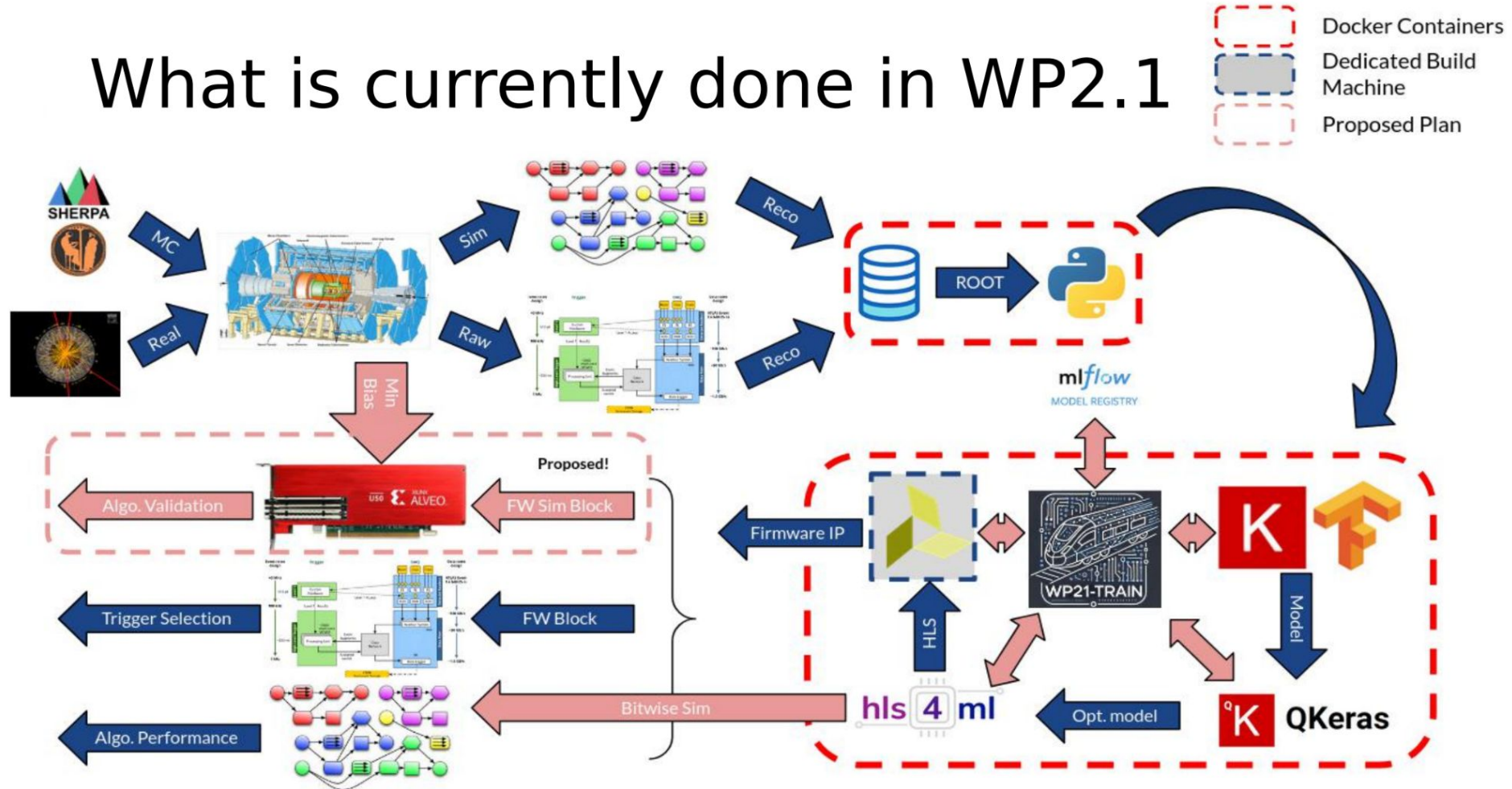
FW synthesis and validation

- Synthesise FW
- Validate HLS standalone
- Validate SW emulator
- Validate FW with test vectors



ATLAS Level0

What is currently done in WP2.1



Deployment challenges

Data

- On demand access to ML-compatible training data
- Data traceability, provenance & auditing throughout formats
 - Data generation and transaction traceability (how was this data generated and processed?)
i.e. departure from: (X_train.npy, X_test.npy)
- Removal of I/O bottlenecks and large variance in data formats in use
- Changing data schemas
- GPU underutilisation and bottlenecks associated with CPU, VRAM transfers

Firmware

- Limited re-configurable firmware options (to updates weights, don't rebuild the whole bitfile).
- Long synthesis runtimes for large models (i.e. CNNs)

Hardware

- Require flexible provisioning of required hardware when required
- Ephemeral scaling depending on workflow
- FPGA standalone testing

Software

- Compilation of HLS code within release scheduled software frameworks (e.g. CMSSW, ATHENA)
- Centralised deployment of models
- Synchronised version/dependency updates
- Workflow orchestration
 - in service of both R&D & Deployments
- Monitoring and versioning of deployed models
- Tracking models through software to firmware transitions

MLOps Workshop

Back in June, ATLAS & CMS got together for a day to discuss solutions to these challenges

- <https://indico.cern.ch/event/1543741/>

Data: Algorithms to have on-demand access to training data

- Standardised ML data formats across the project to reduce I/O bottlenecks
- GPU utilisation across the whole chain (NVIDIA Rapids for data pre-processing)
- Central production of “ML-friendly” datasets within experimental software (e.g. serialise SoA)
- Enable data tracking, traceability & audits of training data

Emulation: ML Models in the online-to-offline framework

- Define a model “object” which travels through all software and firmware transitions for clear model lineage
- Add ML models to centralised offline database with “Interval Of Validity” for bookkeeping
- Allow for just-in-time compilation / runtime interpreters when evaluating models converted into High Level Synthesis code

Training: Re-training & Calibrating models

- GPU capacity
- Robust training schemes (Lipschitz networks)
- Monitoring data drifts
- Strategies for continual learning



Data

Production of ML friendly datasets within experimental software frameworks (initial meetings with developers of SoA solutions have taken place)

Data versioning, traceability and metadata transfers with DVC

GPU utilisation for large-scale data processing with NVIDIA Rapids

Standardised data format throughout (e.g. Parquet)

New! RAPIDS Initialization Action
GPU Accelerated Data Science At Scale for Cloud Dataproc

The diagram shows a layered architecture for RAPIDS. At the bottom is ARROW, followed by CUDA. Above CUDA is RAPIDS, which includes CUDF, CUML, CUGRAPH, and CUDNN. On top of RAPIDS is PYTHON, which includes DEEP LEARNING FRAMEWORK. To the left of the layers is DASK. To the right of the layers are five icons with corresponding text: a square with a plus sign for 'Scaling out on any GPU', a target for 'Top Model Accuracy', a clock for 'Reduce Training Time', a gear for 'Hassle-free Integration', and code symbols for 'Open Source'. The NVIDIA logo is in the top right corner.

RAPIDS: Load data into VRAM once. Preprocess, train & release memory

```
$ gto history churn -r https://github.com/iterative/example-gtc
```

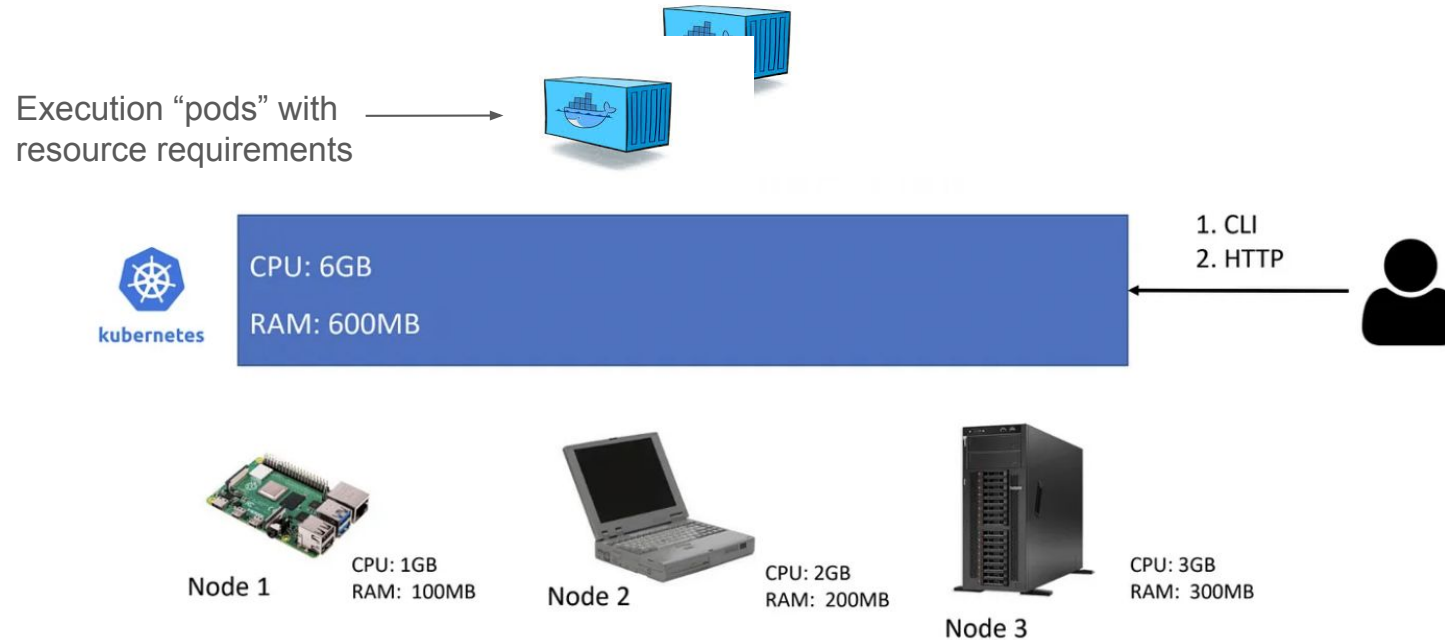
timestamp	artifact	event	version
2022-11-09 13:40:33	churn	assignment	v3.1.1
2022-11-09 13:40:33	churn	registration	v3.1.1
2022-11-08 09:53:53	churn	commit	v3.1.1
2022-11-07 06:07:13	churn	assignment	v3.1.0
2022-11-06 02:20:33	churn	assignment	v3.0.0
2022-11-04 22:33:53	churn	assignment	v3.1.0
2022-11-03 18:47:13	churn	assignment	v3.0.0
2022-11-02 15:00:33	churn	registration	v3.1.0
2022-11-01 11:13:53	churn	commit	v3.1.0
2022-10-28 23:53:53	churn	registration	v3.0.0
2022-10-27 20:07:13	churn	commit	v3.0.0

DVC example of artifact history (e.g. a data object)

Hardware

Big data processing, Model training, Firmware synthesis.

- Complete deployment pipelines consist of data processing, model training and firmware synthesis demands **heterogeneous** compute.
- Kubernetes is used to orchestrate the underlying hardware, provisioning resources dependent on pipeline component.



Firmware

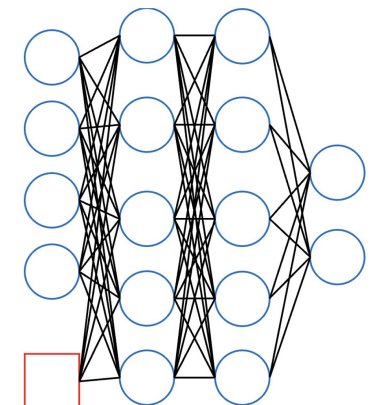
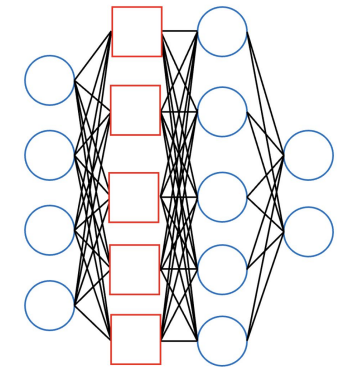
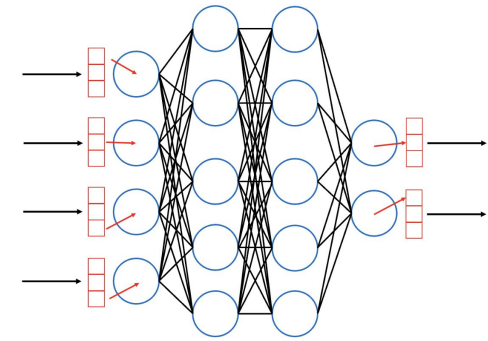
Want to avoid rebuilding bitfile for small model updates -> same architecture, same precision, same inputs

Calibration of ML models

Retrain a model off-chip, send model updates to FPGA to update a model

Reconfigurable FW

- Fully reconfigurable BDTs available with “Forest Processing Unit” take a latency and resource hit. Have to define a max depth and max number of trees beforehand
- Reconfigurable NNs would be implemented using BRAMs, would take a latency and resource hit and lose any optimisations from HGQ or da4ml like packages
- Other options are available:
 - Reconfigurable output threshold cut
 - Reconfigurable input scaling layers
 - Reconfigurable output scaling layers
 - Reconfigurable single layers -> scaling in a latent space
 - Regression of LUT addresses allowing output to be updated
 - Parameterised NNs with reconfigurable parameterisation



Software

Goal: standardised pipeline which is flexible to a variety of processing/training/deployment schemes

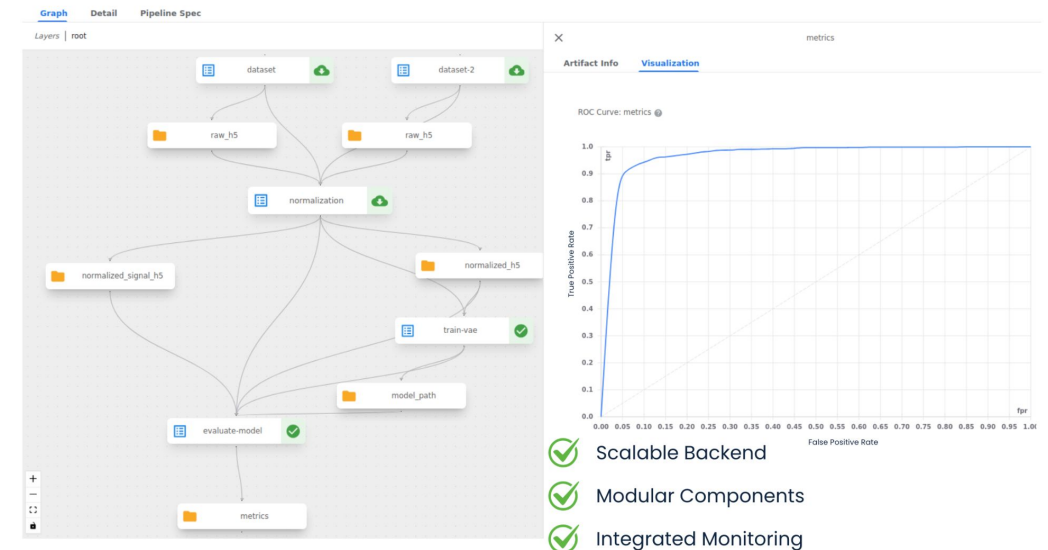
Approach: Provide a scaffolding for each pipeline component and leave the logic within to individual developers

Execute: Each component executed on a centralised Kubernetes platform

```
1 iris_data:  
2   filepath: data/01_raw/iris.csv  
3   load_args:  
4     sep: ','  
5   type: pandas.CSVDataset  
6  
7 iris_data_loaded:  
8   filepath: data/02_loaded/iris.csv  
9   save_args:  
10    index: False  
11    sep: ','  
12    type: pandas.CSVDataset  
13  
14 processed_iris_X_train:  
15   filepath: data/03_preprocessed/processed_iris_X_train.csv  
16   save_args:  
17     index: False  
18     sep: ','  
19     type: pandas.CSVDataset  
20  
21 processed_iris_X_test:  
22   filepath: data/03_preprocessed/processed_iris_X_test.csv  
23   save_args:  
24     index: False
```

```
shuttles:  
  type: pandas.ExcelDataset  
  filepath: data/01_raw/shuttles.xlsx  
  load_args:  
    engine: openpyxl  
  
preprocessed_companies:  
  type: pandas.ParquetDataset  
  filepath: data/02_intermediate/preprocessed_companies.pq  
  
preprocessed_shuttles:  
  type: pandas.ParquetDataset  
  filepath: data/02_intermediate/preprocessed_shuttles.pq
```

Kedro for code and data type structures to standardise interfacing between components



Kubeflow as execution engine

Kedro

```
project-dir      # Parent directory of the template
├── conf         # Project configuration files
├── data         # Local project data (not committed to version control)
├── docs         # Project documentation
├── notebooks    # Project-related Jupyter notebooks (can be used for exp
├── src          # Project source code
├── tests        # Folder containing unit and integration tests
├── .gitignore   # Hidden file that prevents staging of unnecessary files
├── pyproject.toml # Identifies the project root and contains configurator
├── README.md    # Project README
└── requirements.txt # Project dependencies file
```

- Kedro tries to have clean code for data science
- Takes concepts from software engineering and applies them to machine-learning projects
- Everything is structured and the user only needs to implement “simple” functions for each step

Kedro

- Kedro uses yml based config files for each step
- Where is a dataset / model stored?
- Which pipeline needs what kind of model / dataset?

```
2 This is a boilerplate pipeline 'model_training'
3 generated using Kedro 0.19.14
4 """
5
6 from kedro.pipeline import node, Pipeline, pipeline # noqa
7 from .nodes import train_model
8
9
10 def create_pipeline(**kwargs) -> Pipeline:
11     return pipeline(
12         [
13             node(
14                 func=train_model,
15                 inputs=["processed_iris_x_train", "processed_iris_y_train"],
16                 outputs="train_model",
17                 name="train_model_node",
18             )
19         ]
20
```

```
processed_iris_y_train:
  filepath: data/03_preprocessed/processed_iris_y_train.csv
  save_args:
    index: False
    sep: ','
  type: pandas.CSVDataset

processed_iris_y_test:
  filepath: data/03_preprocessed/processed_iris_y_test.csv
  save_args:
    index: False
    sep: ','
  type: pandas.CSVDataset

train_model:
  filepath: data/04_models/trained_model.pkl
  type: pickle.PickleDataset

model_pred:
  filepath: data/05_validation/model_pred.pkl
  save_args:
    index: False
    sep: ','
  type: pandas.CSVDataset
```



default

Search

- feature_engineering
 - create_derived_features
 - create_feature_importance
 - create_static_features
 - joiner
 - feature_engineering.feat_der...
 - feature_engineering.feat_stat...
- ingestion
 - <lambda>
 - apply_types_to_companies
 - apply_types_to_reviews
 - apply_types_to_shuttles
 - combine_step
 - company_agg

Filters

- Element types
- Nodes 21
- Datasets 30
- Parameters 6
- Tags



raw



intermediate



primary

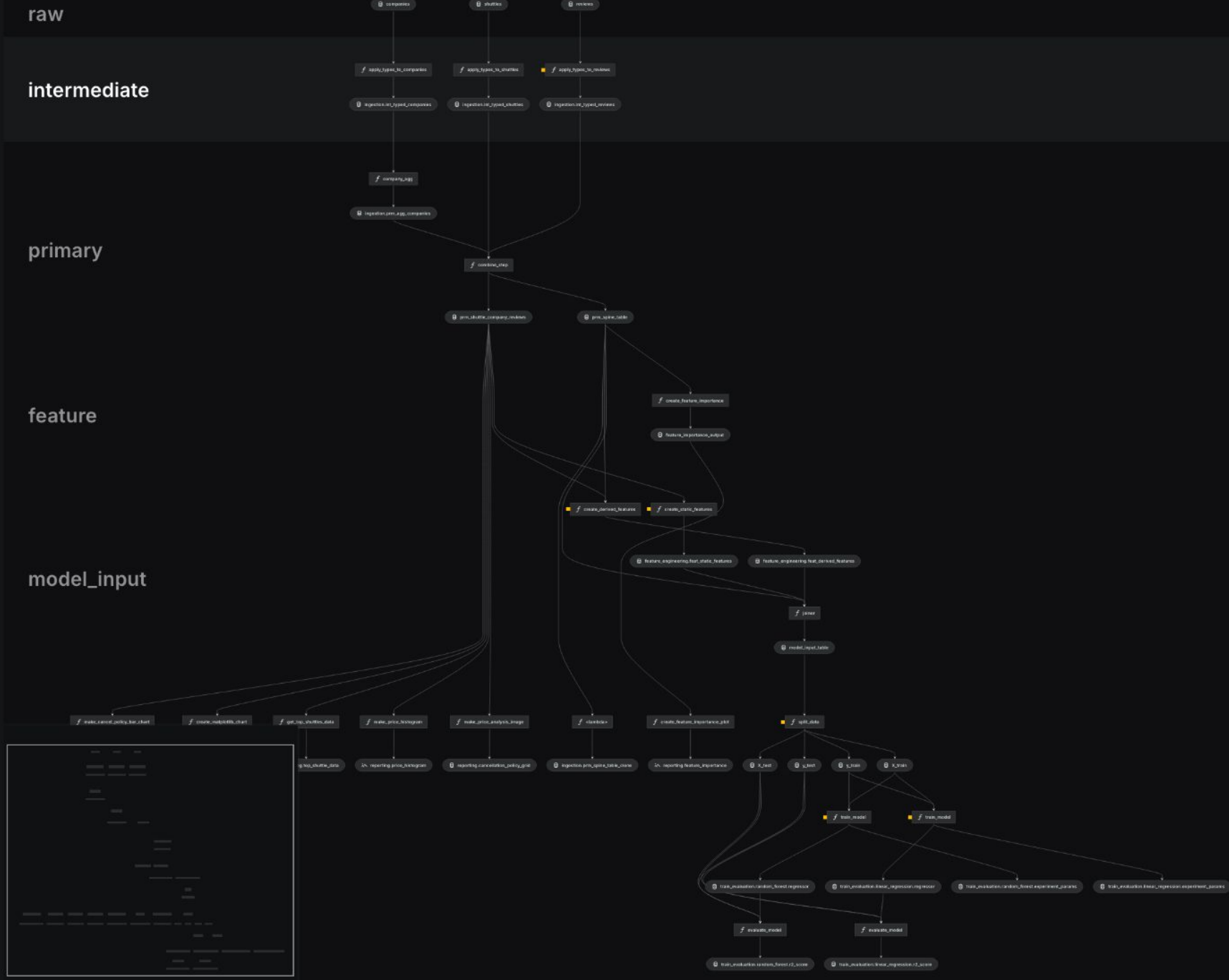


feature

model_input



25%



Kedro

```
load_data:
  stage: load
  script:
    - kedro run --pipeline=load_data
  artifacts:
    paths:
      - data/02_loaded/

preprocess_data:
  stage: preprocess
  script:
    - kedro run --pipeline=data_processing
  artifacts:
    paths:
      - data/03_preprocessed/

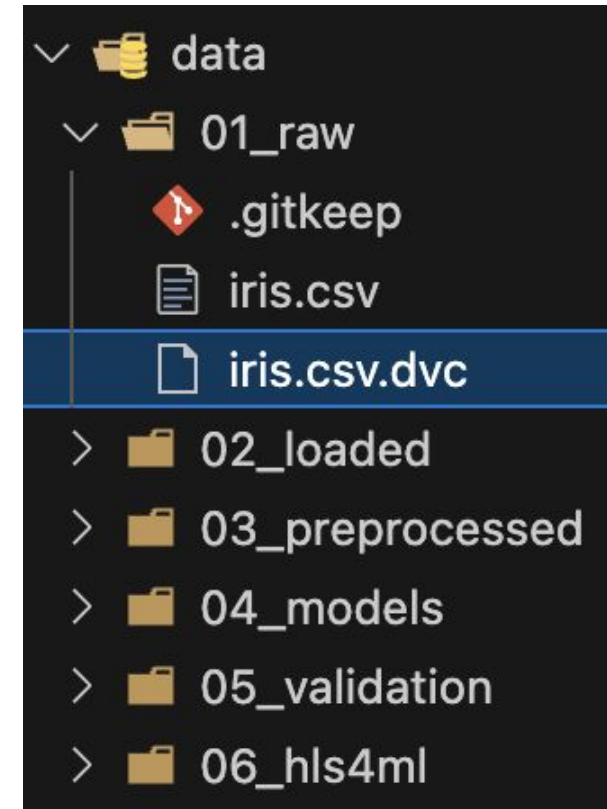
train_model:
  stage: train
  script:
    - kedro run --pipeline=model_training
  artifacts:
    paths:
      - data/04_models/
```

- Each pipeline step can be run separated
- Perfect to use this in gitlab CI

Kedro

- Model and datasets are versioned using <https://dvc.org/>
- Metadata can be added
- Tracking of input data (e.g. 100x root files) for the flat data used to train the model can also be checked against

```
outs:  
- md5: 04ed69383af337e9dabf934cbc8abc11  
  size: 3858  
  hash: md5  
  path: iris.csv
```



Software

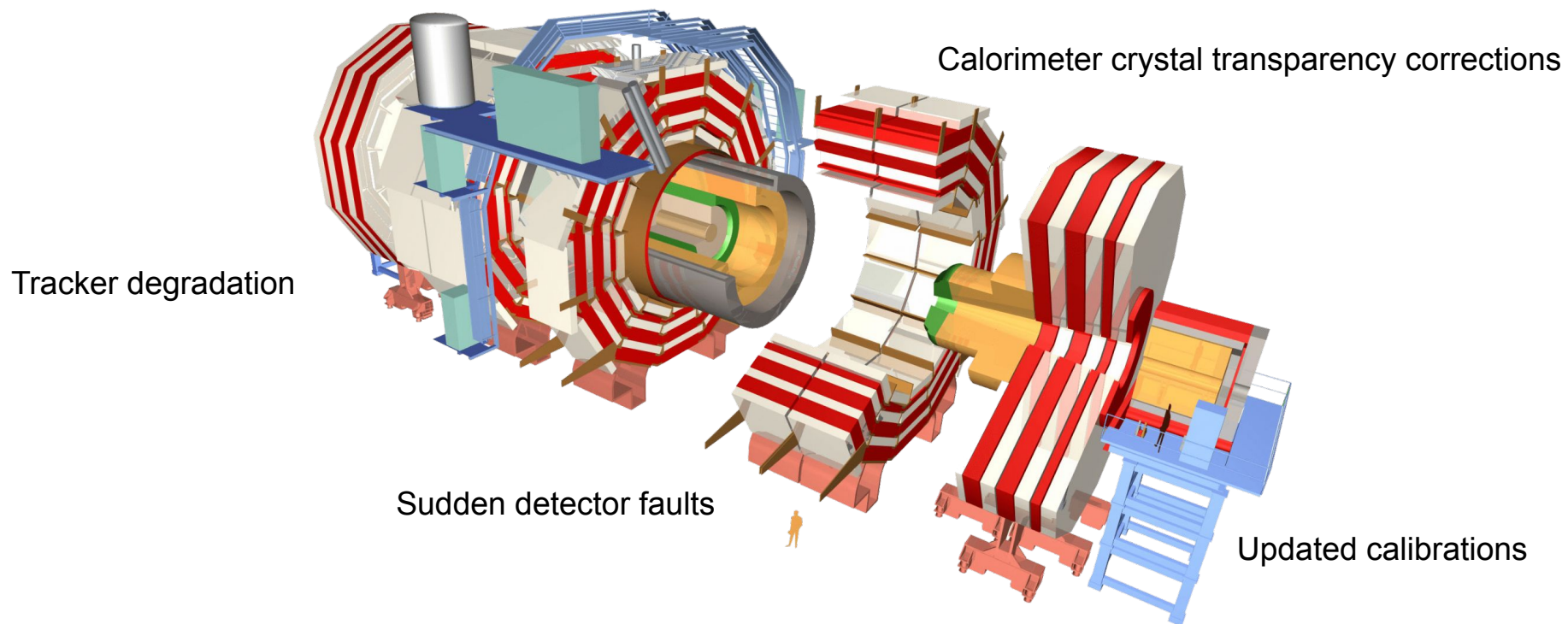
Some issues:

- complex software stack, C++, python, HLS (not standard in ML deployments)
- bespoke deployment needs (e.g. compile HLS within CMSSW)
- Ability to *replay* any component on demand (automatically fetch required artifacts from last successful run)

ML in Evolving Environments

The challenges of data drift.

- The CMS detector is an example of a changing environment;
 - changes can be sudden and unexpected or evolve as a function of time.
 - Models will encounter changing conditions w.r.t. training time.



Re-training (re-calibrating?)

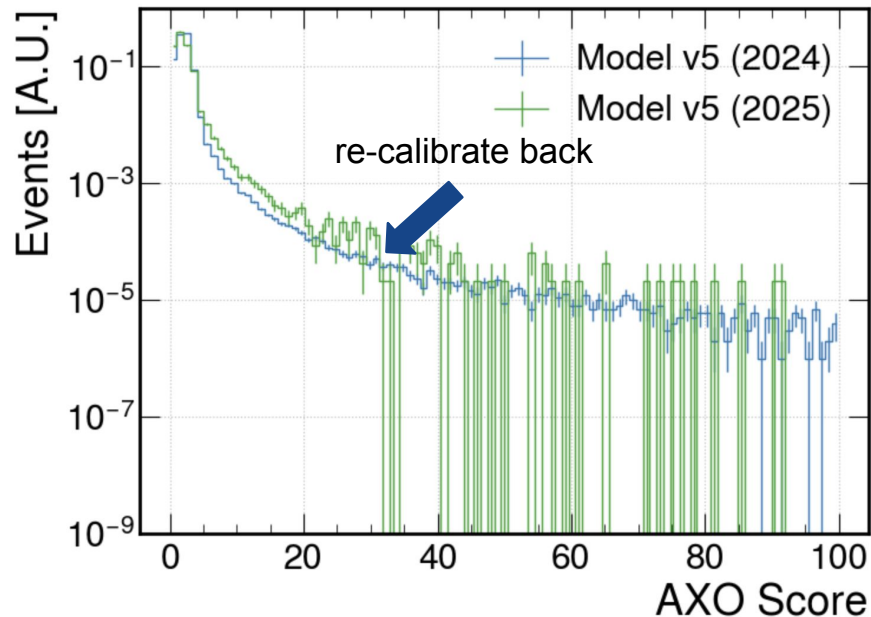
Data will drift over time (and/or unexpectedly). What are the options?

- 1. Collect a new dataset and retrain the model (our initial approach with AXOL1TL).**
 - Each retraining changes the algorithm's characteristics, effectively selecting a new phase space.
 - The new behaviour must then be tested, validated, presented: very lengthy process.
- 2. Collect new dataset and re-calibrate the model to a reference distribution.**
 - The reference is already well understood in terms of trigger characteristics.
 - This can be done quickly to respond to sudden drifts in the data.

Re-training (re-calibrating?)

Going back.

- A strategy for aligning ML models exposed to evolving input data;
 - an updated loss function regularised by divergence from reference distribution (obtained at training time) and clipping mechanism to ensure stable gradients.
 - Largely inspired by reinforcement learning strategies (GPRO).



output (new model/old model) on new data

Kullback–Leibler divergence (KLD) between new model on new data and reference distribution (from original training)

$$L(\theta) = \sum_{i=1}^N \rho_{\delta}(s_{\theta}(x_i) - s_{\text{ref}}(x_i)) + \beta D_{\text{KL}}(p_{\theta}(s) \parallel p_{\text{ref}}(s)) \quad [7]$$

where $\rho_{\delta}(x) = \text{clip}(x, -\delta, \delta)$

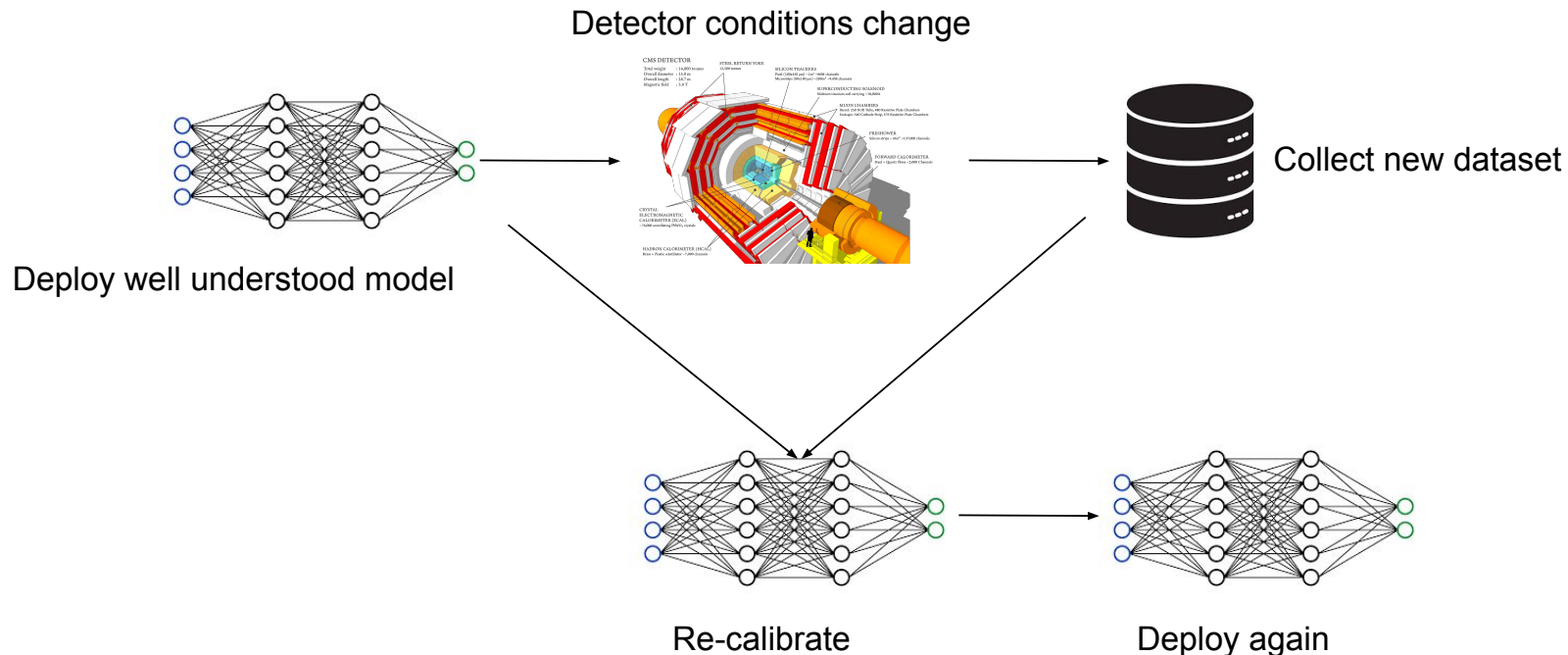
Clipped to a “trust region” where model can explore

↳ early stopping when KLD is minimal

Re-training (re-calibrating?)

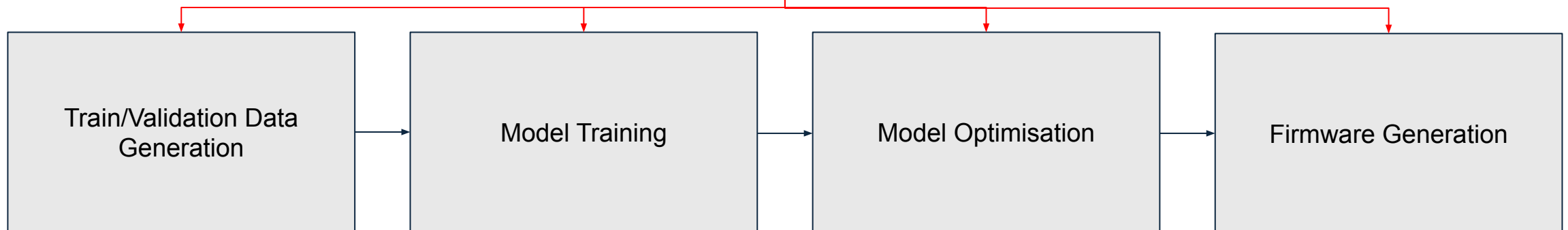
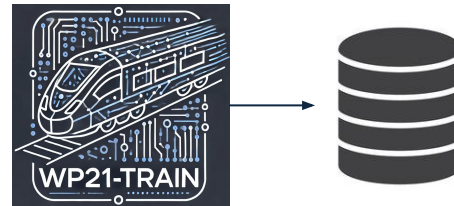
Continual Learning in 2026?

- Considered strategy for AXOL1TL during 2026 data taking;
 - deploy understood model under known detector conditions,
 - as conditions change, use new data to re-calibrate model to known distribution,
 - continue taking data without change in rates or efficiency losses.



Parameter tracking

- **Steps in MLOps pipelines sometimes disjoint**
 - ex: HLS4ML model with corresponding Keras/pyTorch
- **For WP2.1 we're developing the WP21_Train framework**
 - Easily installable [python](#) library
 - Creates meta-data from all the different steps in your pipeline
 - For the time being tracked features are fixed (based on WP2.1 needs) → ideally we want to have it configured via configuration file



Physics based optimisation

- **With WP2.1Train we extract all the information (incl. physics plots)**
 - Trigger rates and efficiency plots
 - P2 TDR drives what the minimum/maximum requirement is (comparing with heuristic equivalents)
- **Create spider/radar-plots with parameters of interest**
 - 2x stage approach
 - First we look at physics performance and ML parameters i.e. Trigger Rate, Energy cut, AUC, ...
 - Iterative process compressing model and tune hyper-parameters
 - Models that pass physics requirement (from TDR initially) go for synthesis
 - Resources, latency, power... checked for successful model to head into implementation

