Karlsruher Institut für Technologie

Institut für Experimentelle Teilchenphysik

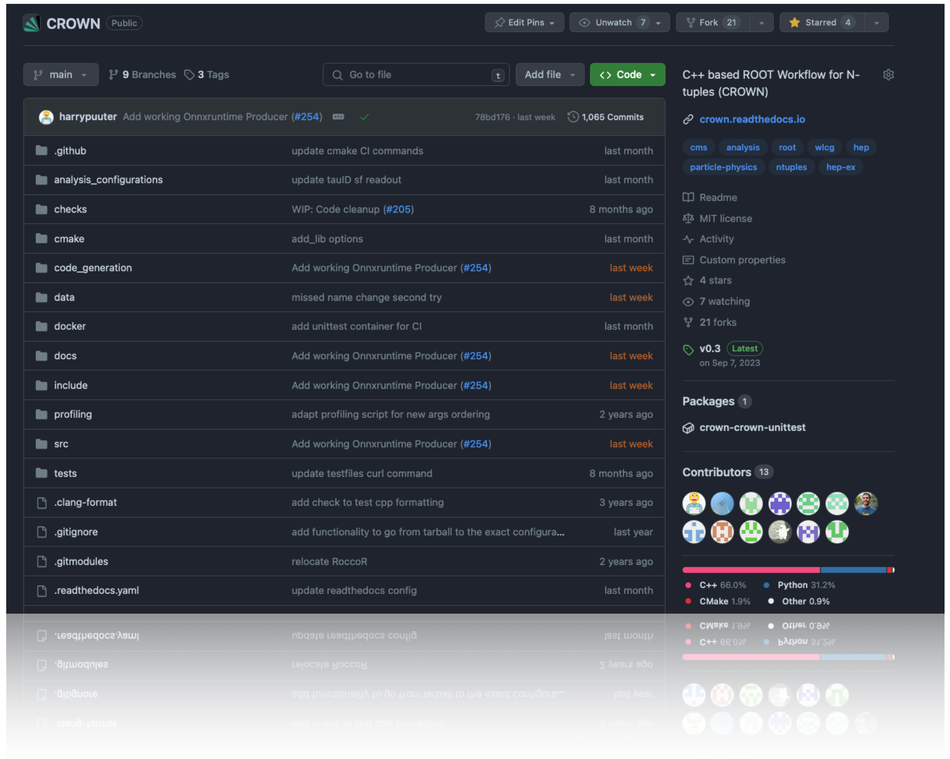
The CROWN Framework

167th ROOT PPP Meeting

Sebastian Brommer, Nils Faltermann, Olha Lavoryk, Moritz Molch, Artur Monsch, Ralf Schmieder, Nikita Shadskiy

CROWN NTuple Framework

- Convert CMS NanoAOD into analysis NTuples
- Focus on efficient and fast processing with minimal dependencies
- Automatic handling of systematics, optimised disk usage of outputs
- Friend tree support
- Scaling via workflow tool and HTCondor batch system



Why analysis NTuples ?

In the last years, we found that splitting the analysis into two steps has some advantages

- NTuples can be used by multiple people, provided by one person
- Defined state of selection / corrections
- Most changes can be made on the level of the Histogram Framework
- Expensive calculation of high level variables can be done later
- Smaller file inputs for the histogram framework → faster runtime and faster analysis turnarounds

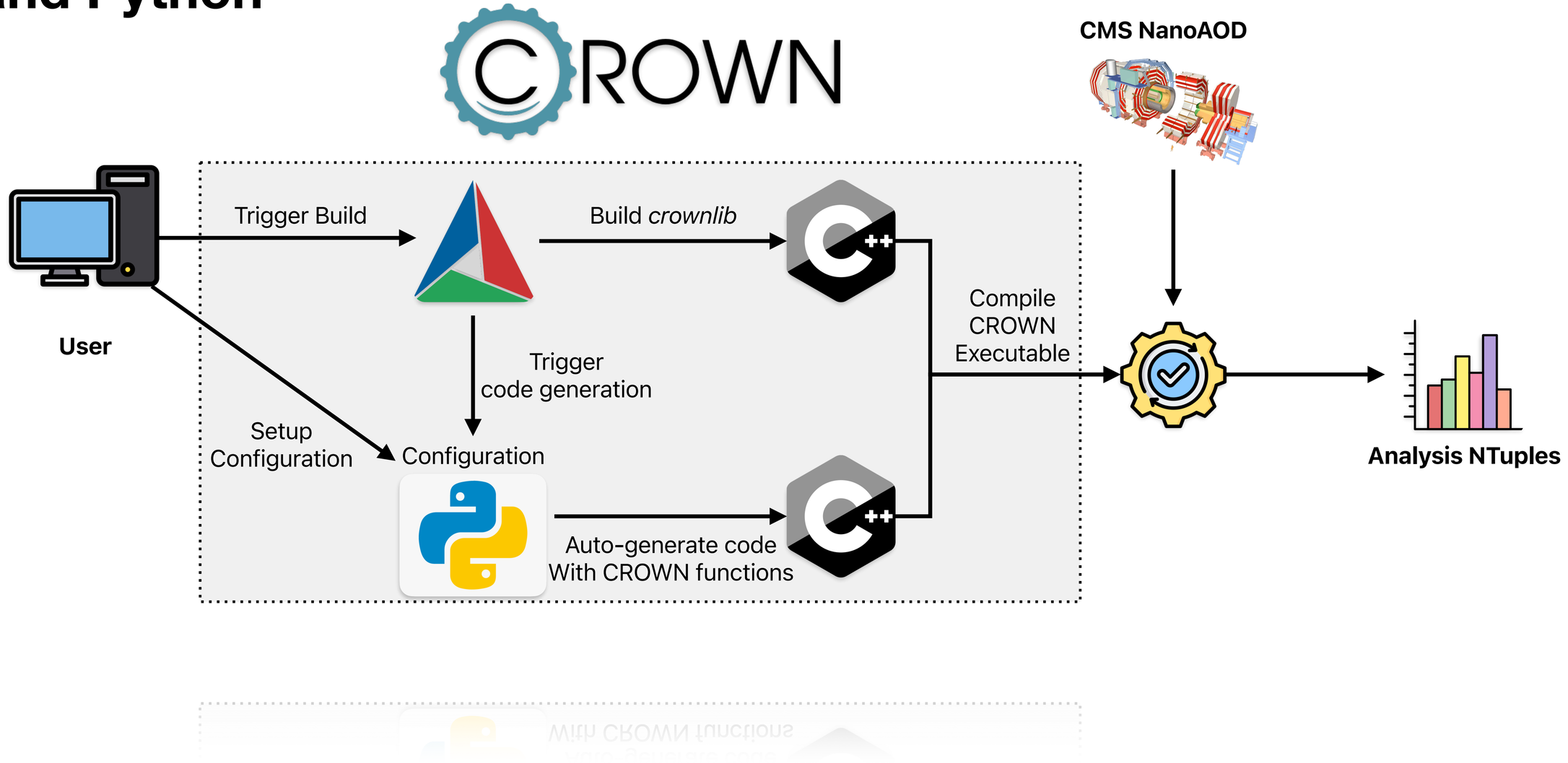
CROWN Usage

- Used by the KIT CMS Group since ~2 years
- O(10) analysis within CMS use CROWN
- Listed as “Supported Framework” by the CMS analysis tool group (link)
- 1065 commits and 204 pull requests by 13 Contributers

Core Principles and Basic building blocks of CROWN

Core Principles

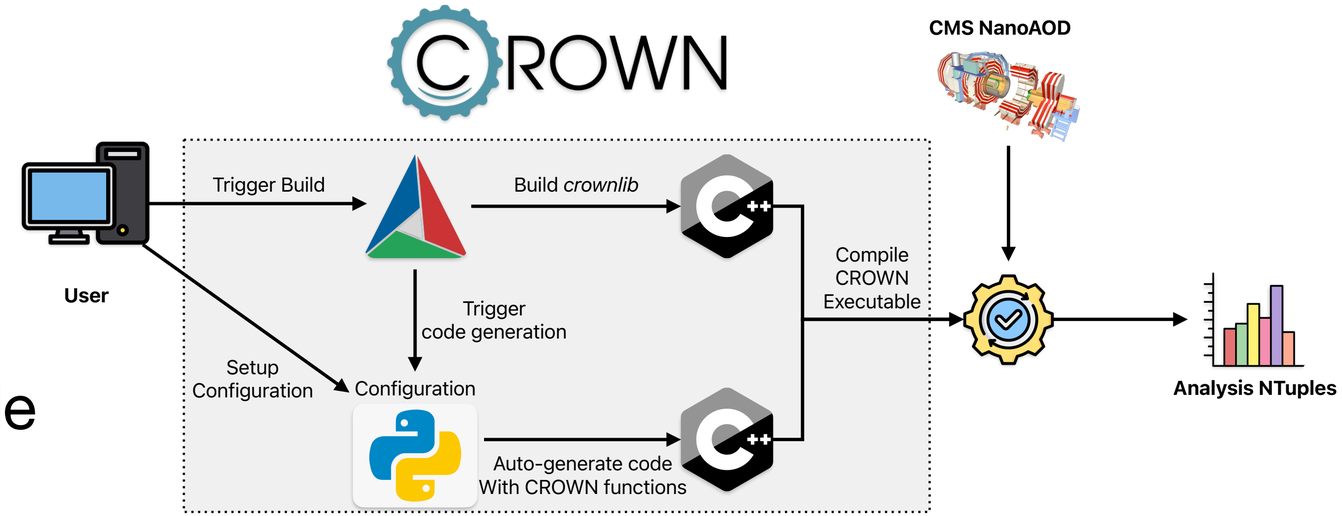
C++ and Python



Core Principles

C++ and Python

- Utilize **python as a configuration language** to auto-generate C++ code
- Combine simple C++ functions into **one large RDataFrame**
- Executables have **simple interface**
- Multiple executables for different samples, and eras
- Focus on validation of user configuration **before** the compilation




```
> ./config_ttbar_2018 outputfile.root /path/to/inputfiles/*.root
```

Quantities and Producers

Basic Building Blocks

- **Quantity** objects to track inputs and outputs
- **Producers** to calculate new quantities and **Filters** to filter events
- **ProducerGroups** to organise Producers easier

```
MET_phi = NanoAODQuantity("PuppiMET_phi")
MET_pt = NanoAODQuantity("PuppiMET_pt")

met_p4 = Quantity("met_p4")
met = Quantity("met")
```

```
BuildMetVector = Producer(
    name="BuildMetVector",
    call="lorentzvectors::buildMet({df}, {input}, {output})",
    input=[
        nanoAOD.MET_pt,
        nanoAOD.MET_phi,
    ],
    output=[q.met_p4],
    scopes=["global"],
)

MetPt = Producer(
    name="MetPt",
    call="quantities::pt({df}, {output}, {input})",
    input=[q.met_p4],
    output=[q.met],
    scopes=["global"],
)
```

```
MetBasics = ProducerGroup(
    name="MetBasics",
    scopes=["global"],
    subproducers=[
        BuildMetVector,
        MetPt,
    ],
)
```

C++ Functions

Basic Building Blocks

- Identical base pattern for all CROWN functions (df as first argument, df as return object)
- Opt for simple & generic functions
- No just-in-time compilation
- Designed to be shared among different analyses

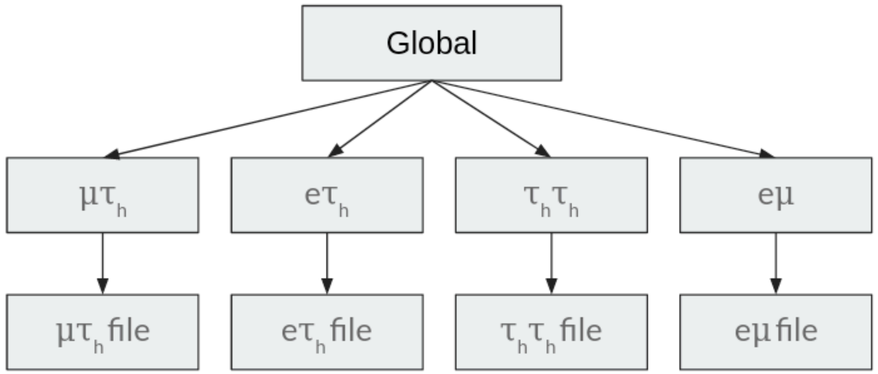
```
BuildMetVector = Producer(
    name="BuildMetVector",
    call="lorentzvectors::buildMet({df}, {input}, {output})",
    input=[
        nanoAOD.MET_pt,
        nanoAOD.MET_phi,
    ],
    output=[q.met_p4],
    scopes=["global"],
)
```

```
ROOT::RDF::RNode buildMet(ROOT::RDF::RNode df, const std::string &met_pt,
                          const std::string &met_phi,
                          const std::string &outputname) {
    auto construct_metvector = [](const float &pt, const float &phi) {
        // for Met, eta is zero
        auto met = ROOT::Math::PtEtaPhiEVector(pt, 0, phi, pt);
        // cast Met vector to a ROOT::Math::PtEtaPhiMVector to make latter
        // functions easier to use
        return (ROOT::Math::PtEtaPhiMVector)met;
    };
    return df.Define(outputname, construct_metvector, {met_pt, met_phi});
}
```

Python Configuration

Basic Building Blocks

- The whole configuration is stored in a python object
- **Parameters** are used to set cut values, working points etc.
- **Scopes** for different final states, one output file per scope




```
configuration = Configuration(
    era,
    sample,
    scopes,
    shifts,
    available_sample_types,
    available_eras,
    available_scopes,
)
```

```
configuration.add_config_parameters(
    "global",
    {
        "min_muon_pt": 10.0,
        "max_muon_eta": 2.4,
        "max_muon_dxy": 0.045,
        "max_muon_dz": 0.2,
        "muon_id": "Muon_mediumId",
        "muon_iso_cut": 0.3,
    },
)
```

Python Configuration

Basic Building Blocks

User defines all **Producers** to be run and all **Quantities** to be added to the output

Gives CROWN **complete knowledge**

- on what to run
- on what to modify (see next slides)
- to validate the user config before compiling and running
- to optimise configuration and producer ordering
- Minimize “magic” functions covering special cases

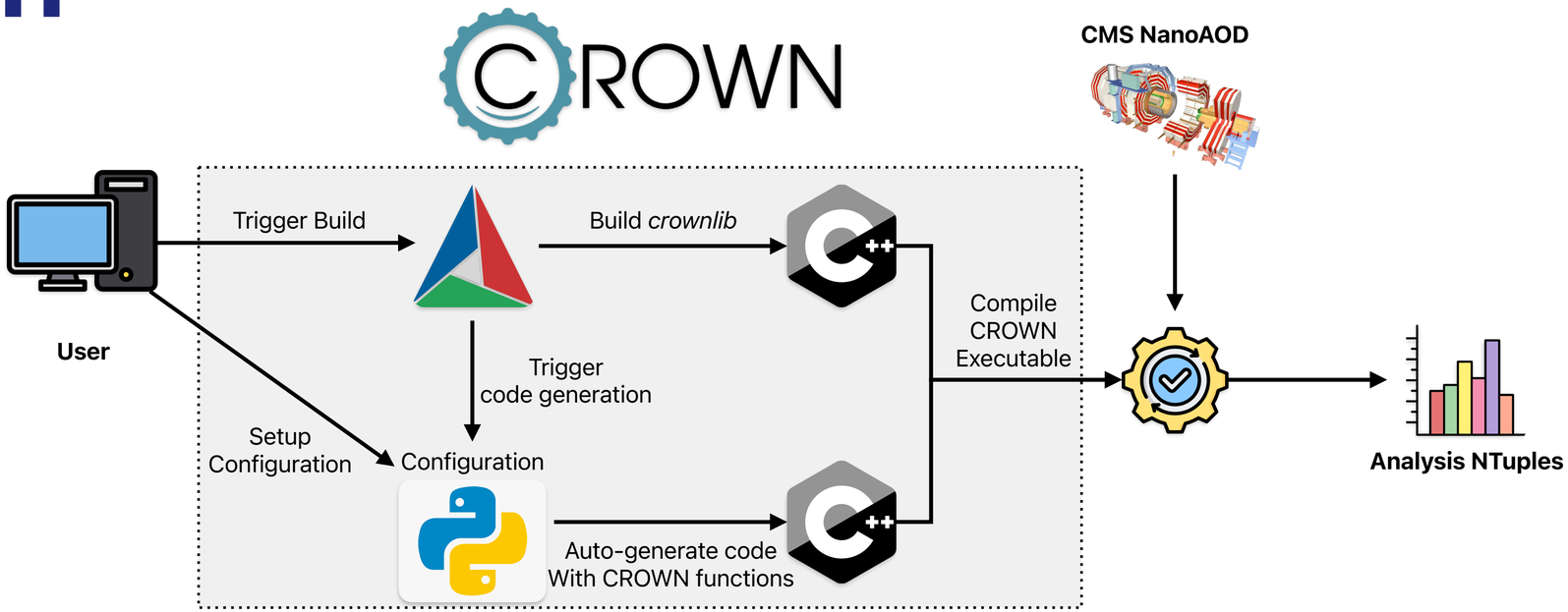
```
configuration.add_producers(
    "global",
    [
        met.MetBasics,
    ],
)

configuration.add_outputs(
    scopes,
    [
        nanoAOD.run,
        q.met,
    ],
)
```

```
configuration.optimize()
configuration.validate()
configuration.report()
return configuration.expanded_configuration()
```

Code Generation

Code Generation

- Based on configuration, C++ code is generated, one file per defined producer
- Steering via cmake with extensive report on things like unused parameters, size of RDataFrame etc.
- Code generation is fast (4-5 s for a RDataFrame with 15k Defines)
- Compile times are reasonably fast (2-3 min for a 15k Defines RDataFrame with 20 cores)




```
# setup build
> cmake ../ -DANALYSIS=tau -DCONFIG=config -DSAMPLES=ttbar
-DERAS=2018 -DSCOPES=mt,et,tt,em,ee,mm -DSHIFTS=All -DTHREADS=8
# build with make
> make install -j 20
# run CROWN executable
> ./config_ttbar_2018 outputfile.root /path/to/inputfiles/*.root
```

Code Generation

Repo containing a complete version of CROWNs autogenerated code (link)

Folder Structure

```
├── config_ttbar_2018.cxx
├── include
│   ├── ee
│   ├── em
│   ├── et
│   ├── global
│   │   ├── [...]
│   │   ├── MetBasics.hxx
│   ├── mm
│   ├── mt
│   └── tt
└── src
    ├── ee
    ├── em
    ├── et
    ├── global
    │   ├── [...]
    │   ├── MetBasics.cxx
    ├── mm
    ├── mt
    └── tt
```

Main File

```
// [...]
#include “[…]/include/global/MetBasics.hxx”
// [...]
auto df13_global = MetBasics_global(df12_global, onnxSessionManager, correctionManager);
// [...]
```

Auto-generated function calls

```
ROOT::RDF::RNode MetBasics_global (ROOT::RDF::RNode df0, OnnxSessionManager &onnxSessionManager,
CorrectionManager &correctionManager) {
    […]
   auto df2 = lorentzvectors::buildMet(df1, "PuppiMET_pt", "PuppiMET_phi", "met_p4");
    auto df3 = lorentzvectors::buildMet(df2, "PuppiMET_ptUnclusteredDown", "PuppiMET_phiUnclusteredDown",
   “met_p4__metUnclusteredEnDown");
    auto df4 = lorentzvectors::buildMet(df3, "PuppiMET_ptUnclusteredUp", "PuppiMET_phiUnclusteredUp",
   "met_p4__metUnclusteredEnUp");
    auto df5 = quantities::pt(df4, "met_uncorrected", "met_p4");
    auto df6 = quantities::pt(df5, "met_uncorrected__metUnclusteredEnDown", "met_p4__metUnclusteredEnDown");
    auto df7 = quantities::pt(df6, "met_uncorrected__metUnclusteredEnUp", “met_p4__metUnclusteredEnUp");
    […]
    return df19;
}
```

Advanced Features

Python Configuration

Advanced Building Blocks

- **Rules** are used to modify the configuration for different types of samples (e.g. sample-specific corrections)
- **Modifiers** to change parameters based on samples or eras
- **Systematic Shifts** are support for different types of shifts:
  - Different Producers
  - Different configuration parameters
  - Different input quantities

```
configuration.add_modification_rule(
    scopes,
    AppendProducer(producers=event.TopPtReweighting, samples="ttbar"),
)
```

```
configuration.add_config_parameters(
    ["et", "mt", "tt"],
    {
        "tau_sf_file": EraModifier(
            {
                "2017": "data/jsonpog-integration/POG/TAU/2017_UL/tau.json.gz",
                "2018": "data/jsonpog-integration/POG/TAU/2018_UL/tau.json.gz",
            }
        ),
    },
)
```

```
configuration.add_shift(
    SystematicShift(
        name="vsEleBarrelDown",
        shift_config={("et", "mt"): {"tau_sf_vsele_barrel": "down"}},
        producers={("et", "mt"): scalefactors.Tau_2_VsEleTauID_SF},
    )
)
```

Python Configuration

Advanced Building Blocks

- **Shifts** are propagated through the whole configuration
- All producers affected by a shift automatically also produce shifted quantities
- Shifted quantities added to output and identified via

  <quantity_name>__<shift_name>

```
configuration.add_shift(
    SystematicShiftByQuantity(
        name="metUnclusteredEnDown",
        quantity_change={
            nanoAOD.MET_pt: "PuppiMET_ptUnclusteredDown",
            nanoAOD.MET_phi: "PuppiMET_phiUnclusteredDown",
        },
        scopes=["global"],
    ),
    exclude_samples=["data", "embedding", "embedding_mc"],
)
```

```
auto df2 = lorentzvectors::buildMet(df1, "PuppiMET_pt", "PuppiMET_phi",
"met_p4");
auto df3 = lorentzvectors::buildMet(df2, "PuppiMET_ptUnclusteredDown",
"PuppiMET_phiUnclusteredDown", "met_p4__metUnclusteredEnDown");
```

Additional Features

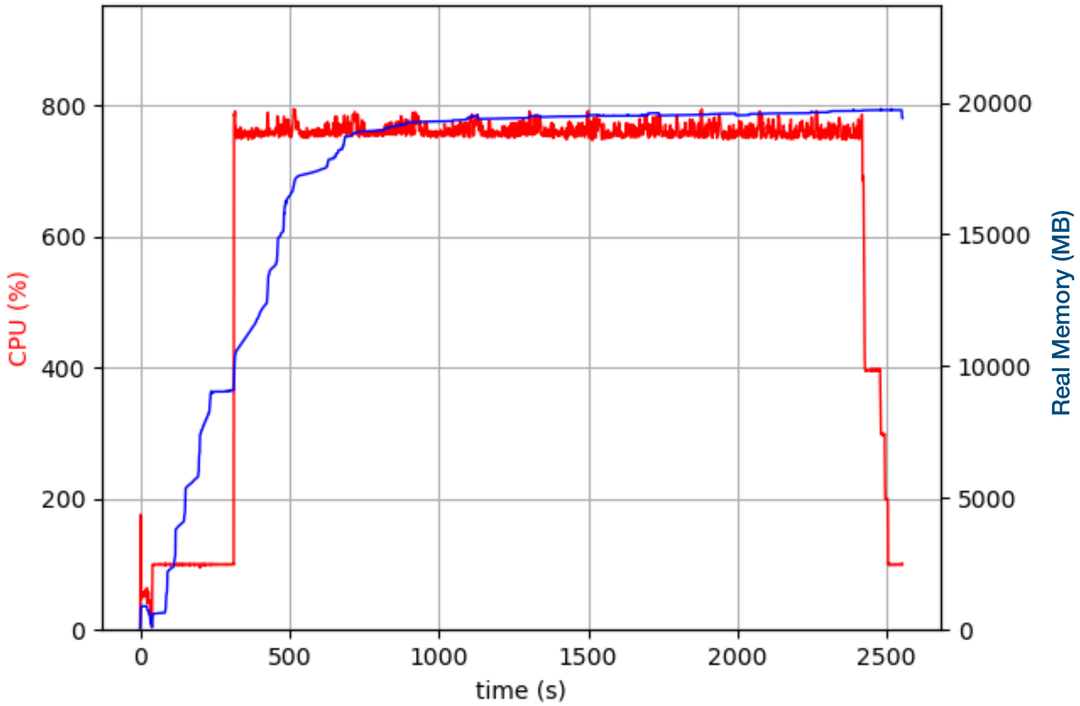
- Support for the production of FriendTrees (requires a CROWN NTuple as input) including a continuation of automatic systematics tracking
- Support for the production of FriendTrees with multiple input trees (aka a CROWN NTuple and some FriendTrees)
- Support for CMS *correctionlib* the standard tool in CMS standard for providing corrections
- Support for *onnxRuntime* for ML inference
- CROWN builds contain all information to get the exact config and code that was used (e.g. to investigate mistakes in NTuples)

Performance

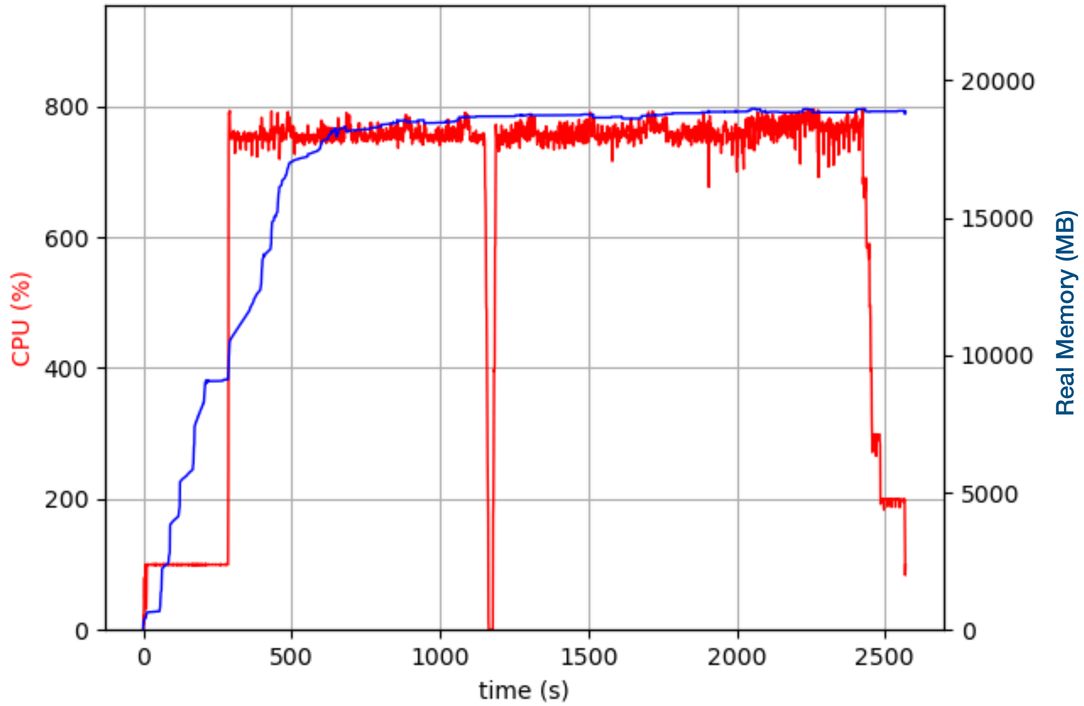
Performance

Dataframe 15.8k Defines // 8 Threads // ROOT 6.30
Input 14,289,000 Events // 28 GB
Output 6 Final States // 20 GB

Copying and processing local files



Reading files via xrootd

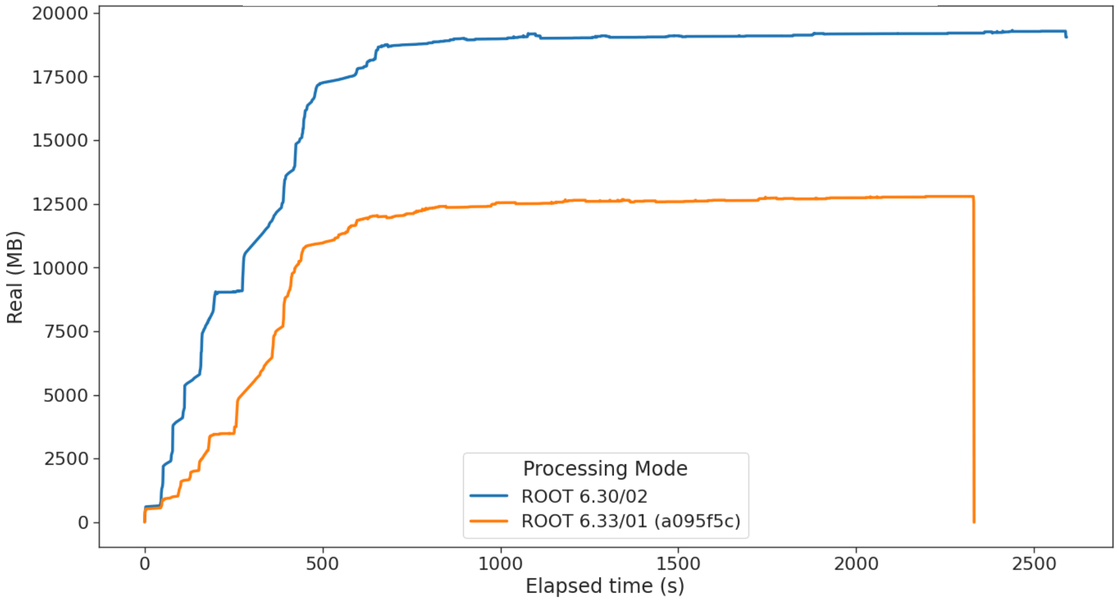


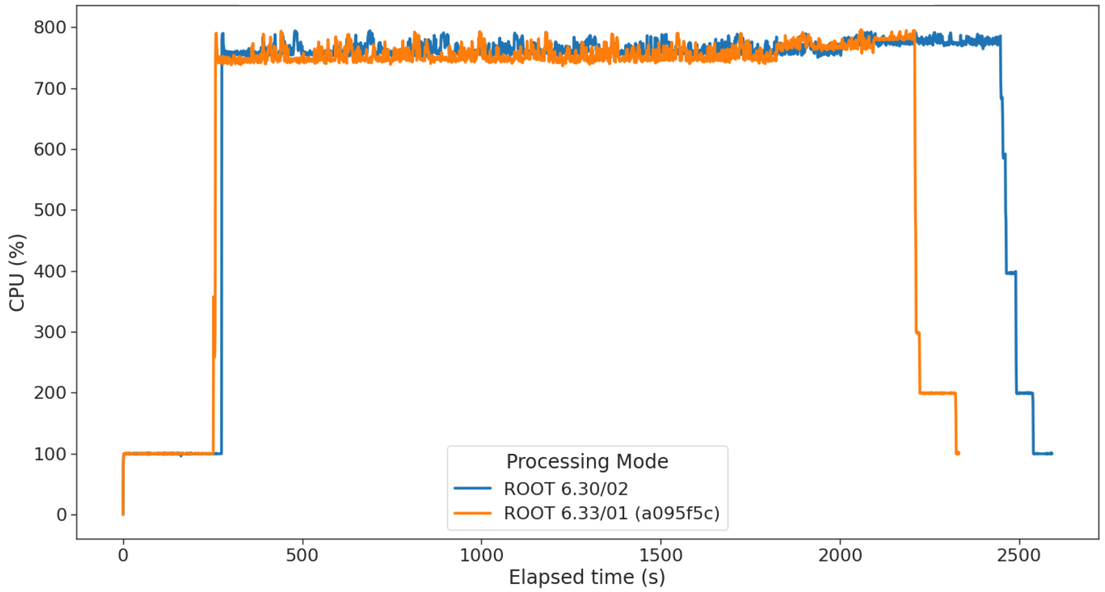
Processing 5600 Events/s into 6 final states including all systematics required for analysis

Performance

With recent master changes

Dataframe 15.8k Defines // 8 Threads
Input 14,289,000 Events // 28 GB
Output 6 Final States // 20 GB



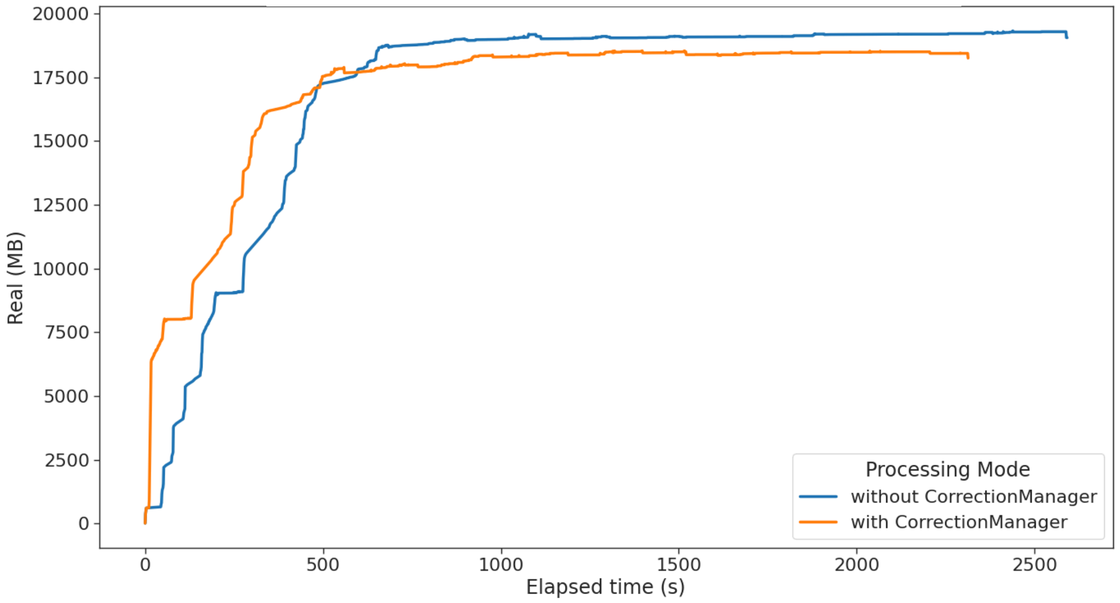


Current ROOT master significantly lowers the memory footprint (by about 30% in this example), while reducing the runtime by 10-15%

Performance

CorrectionManager

Dataframe 15.8k Defines // 8 Threads
Input 14,289,000 Events // 28 GB
Output 6 Final States // 20 GB



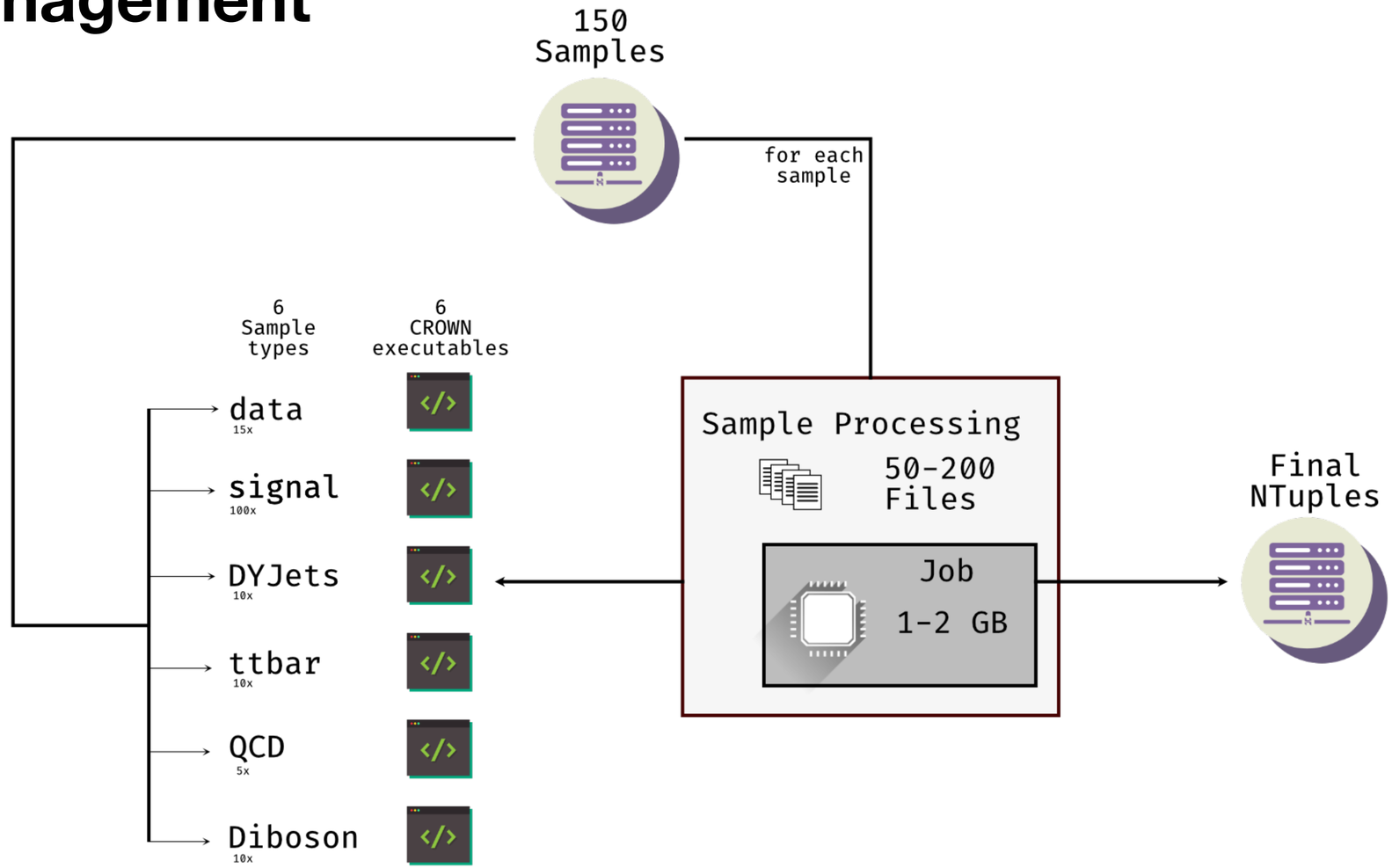

```
ROOT::RDF::RNode id(ROOT::RDF::RNode df, CorrectionManager &correctionManager,
                    const std::string &pt, const std::string &eta,
                    const std::string &year_id, const std::string &variation,
                    const std::string &id_output, const std::string &sf_file,
                    const std::string &idAlgorithm) {
    auto evaluator = correctionManager.loadCorrection(sf_file, idAlgorithm);
    auto df1 = df.Define(
        id_output,
        [evaluator, year_id, variation](const float &pt, const float &eta) {
            double sf = 1.;
            if (pt >= 0.0 && std::abs(eta) >= 0.0) {
                sf = evaluator->evaluate(
                    {year_id, std::abs(eta), pt, variation});
            }
            return sf;
        },
        {pt, eta});
    return df1;
}
```

New feature from yesterday (PR) significantly reduces setup time since corrections are only loaded ones and shared between functions, plus reduced memory usage

Workflow Management

Production for Analysis

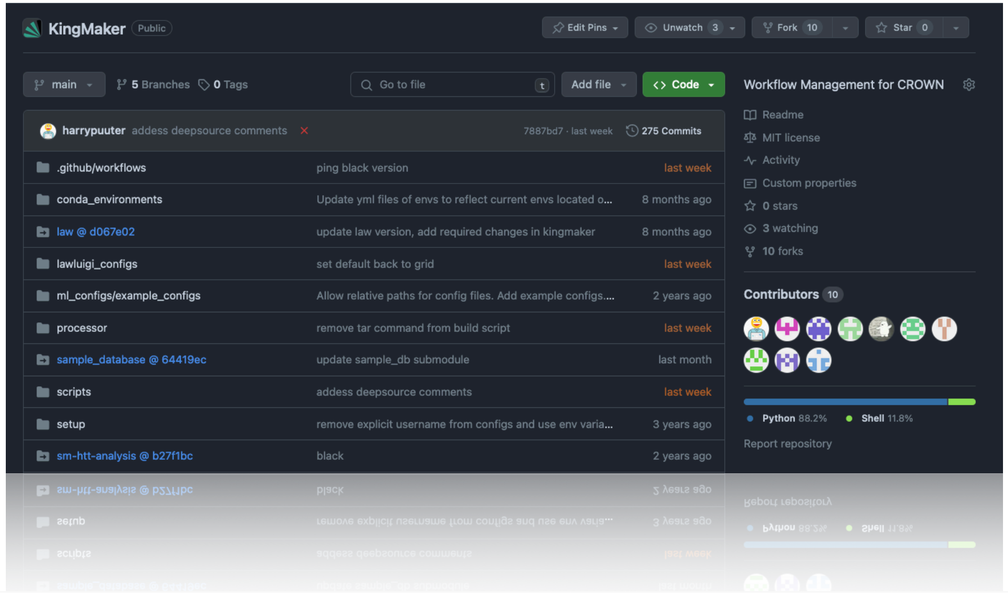
Workflow Management



Workflow management

Kingmaker

- To orchestrate CROWN, we use KingMaker, a law + Luigi workflow
- Organise samples with a sample database tool (link)
- Kingmaker takes care of building all required tarballs, submitting jobs to HTCondor, writing outputs to grid storage
- Allows for a turnaround cycle of < 1 day for large scale analyses (more than 100M events per era)

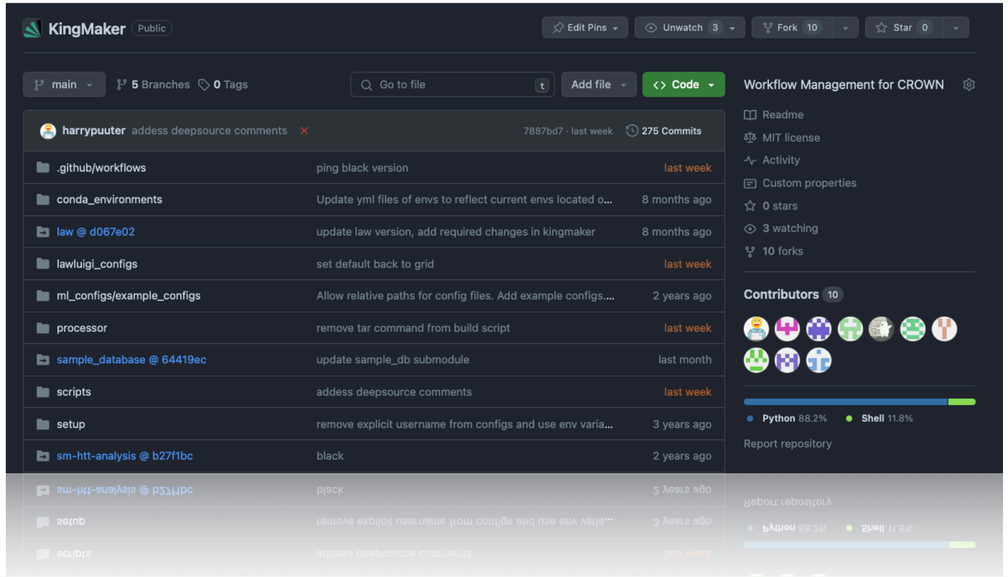



```
law run ProduceSamples --analysis tau --config config --sample-list
samples_18.txt -production-tag 2018_ntuples_v10 --workers 100 --scopes
mt,et,tt --shifts all
```

Workflow management

Kingmaker

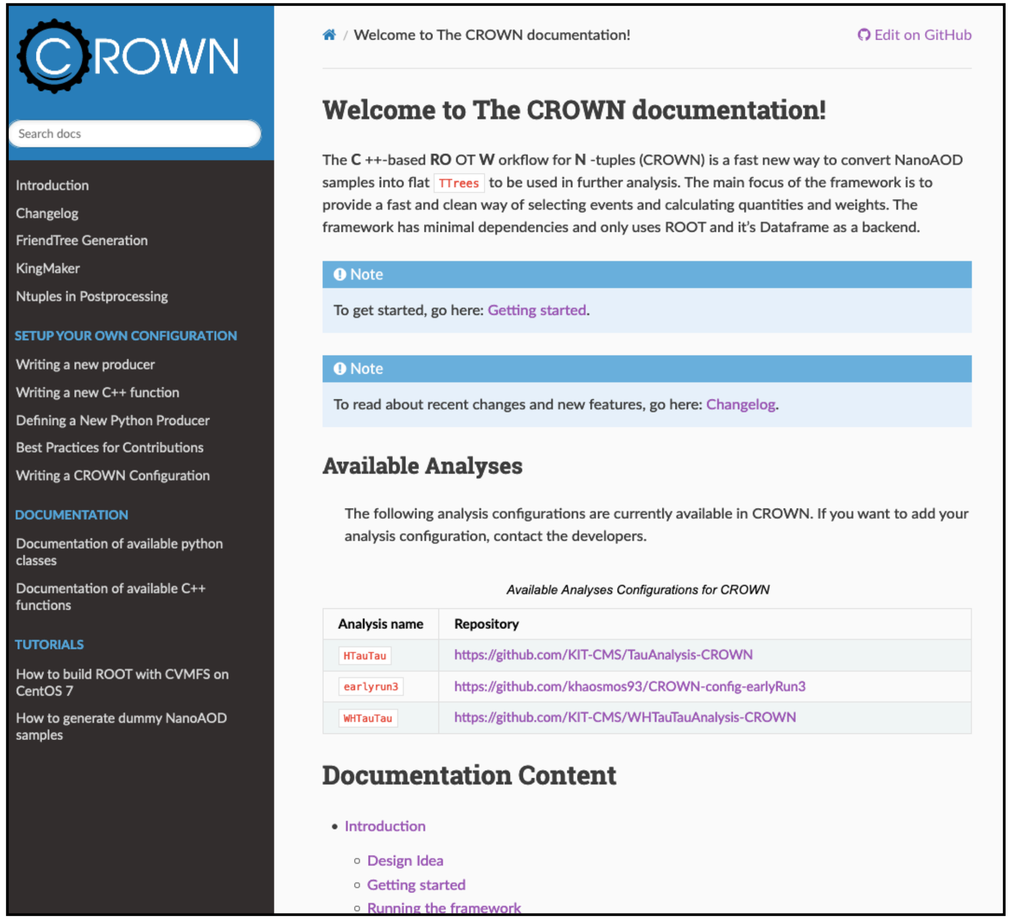
- Minimizes build times by reusing *`crownlib`* containing all CROWN C++ functions
- Supports FriendTree production, keeping track of all required inputs / outputs
- Friends are always run single-core with one input file per job to maintain the correct structure




```
law run ProduceSamples --analysis tau --config config --sample-list
samples_18.txt -production-tag 2018_ntuples_v10 --workers 100 --scopes
mt,et,tt --shifts all


law run ProduceFriends --analysis tau --config config --friend-config
fastmtt -sample-list samples18.txt -shifts all --friend-name
fastmtt_friends_v3 --production-tag 2018_ntuples_v10 --scopes mt,et,tt
--workers 100
```

Conclusion

- CROWN is the main NTuple framework for multiple analysis within CMS
- Law-based workflow manangement tool to run large-scale productions on a batch system
- Focused on performance and efficiency
- CROWN directly profits from all RDataFrame improvements



CROWN: https://github.com/KIT-CMS/CROWN
KingMaker: https://github.com/KIT-CMS/KingMaker
Documentation: https://crown.readthedocs.io/en/latest/index.html

Questions ?

Jitting for Snapshots

- Currently, the snapshot is the only fitted part of CROWN

```
Info in <[ROOT.RDF] Info /build/jenkins/workspace/lcg_release_pipeline/build/projects/ROOT-6.30.02/src/ROOT/
6.30.02/tree/dataframe/src/RLoopManager.cxx:803 in void ROOT::Detail::RDF::RLoop
Manager::Jit()>: Just-in-time compilation phase completed in 114.279227 seconds.
```

- In principle, CROWN would be able to also track datatypes of quantities consistently and provide this information
- C++ templating is limiting factor in this case, compile times explode and even crashes for very large templates