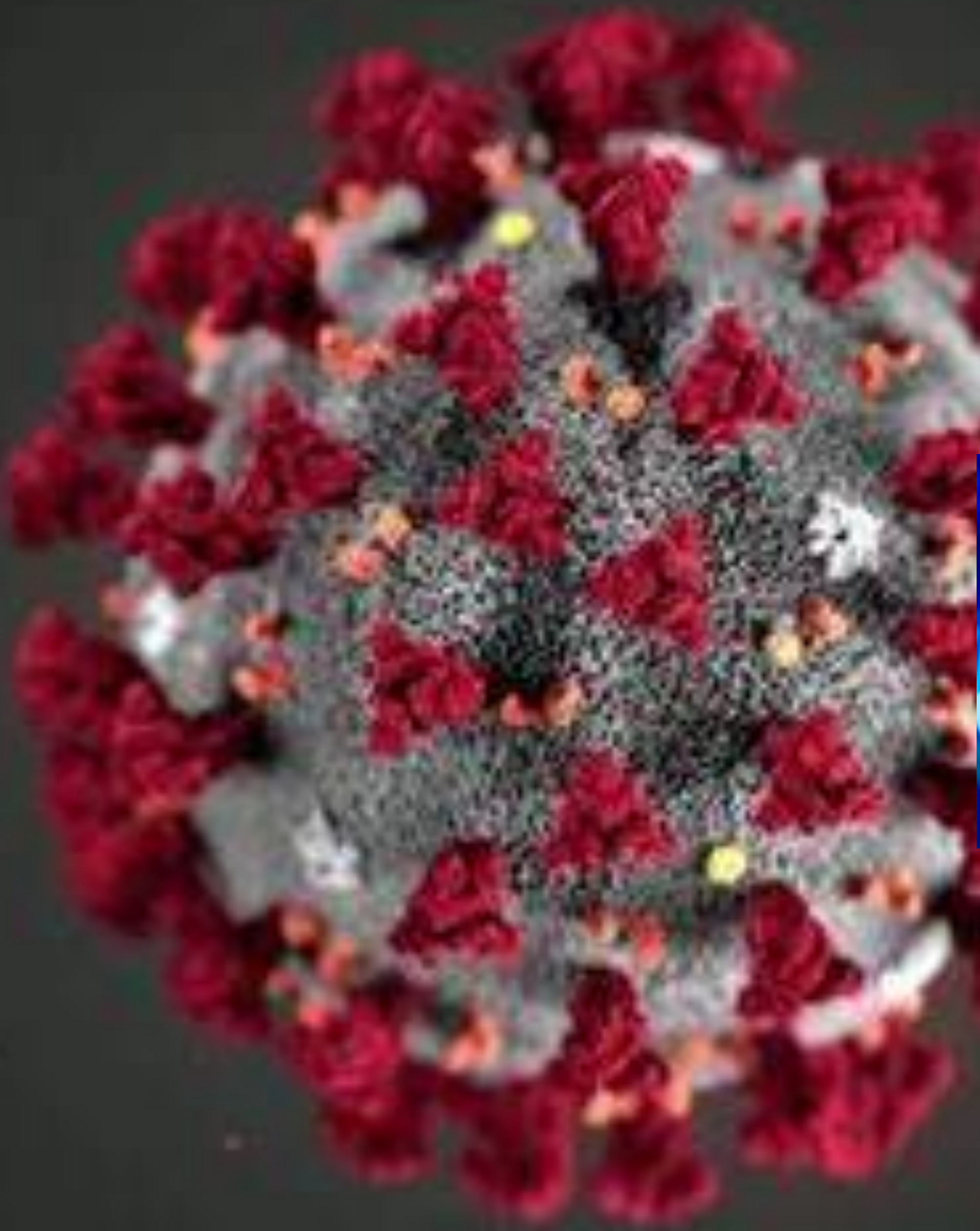
ServiceX In ATLAS

G. Watts (UW/Seattle)

Really aggravated with myself for not being able to join you!



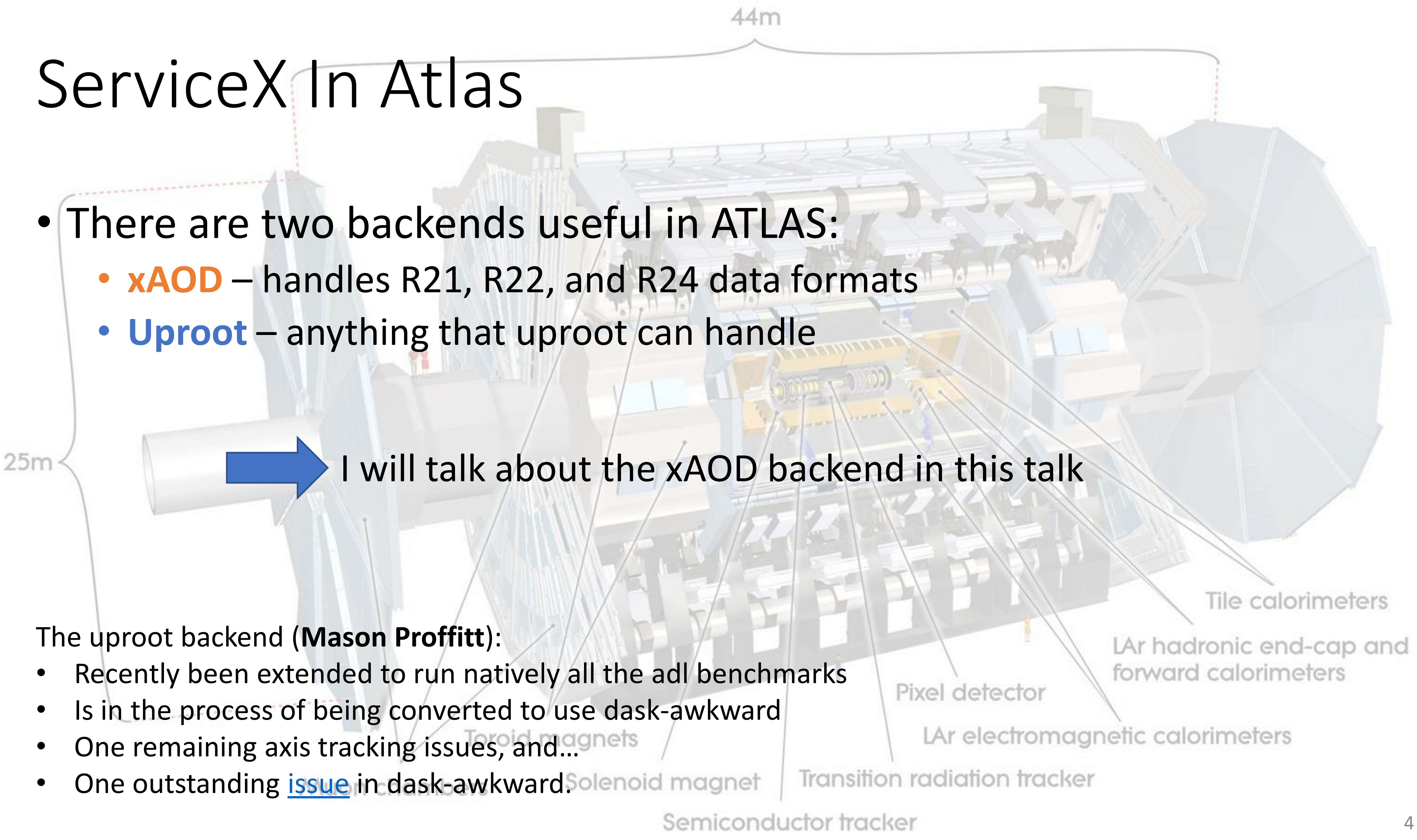
Please say hi to Tal and Alex!

Don’t Worry! Be Happy!!!

I have great views from my hotel…

I don’t have to leave for days!

ServiceX In Atlas



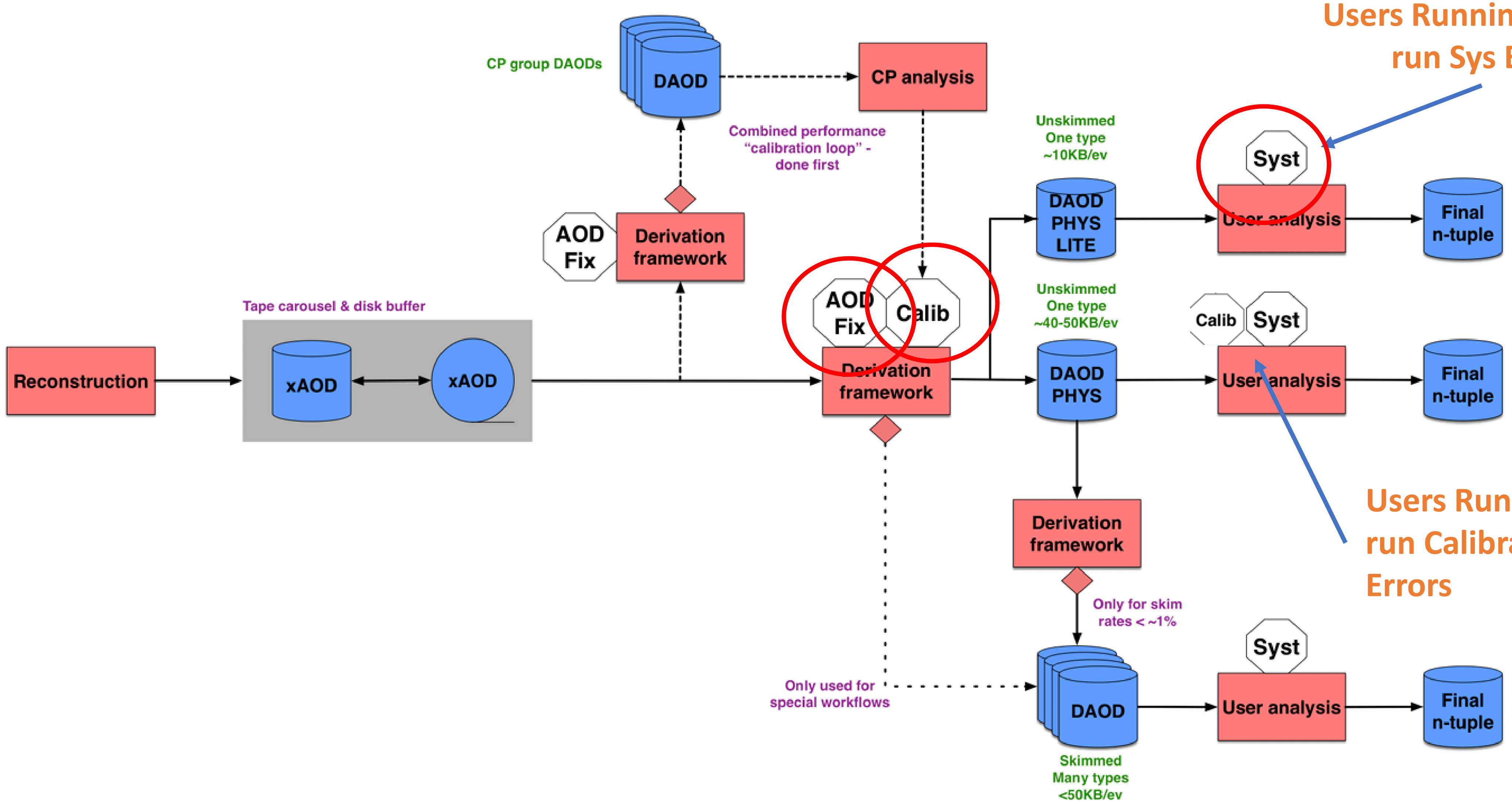
- There are two backends useful in ATLAS:
  - **xAOD** – handles R21, R22, and R24 data formats
  - **Uproot** – anything that uproot can handle

I will talk about the xAOD backend in this talk

The uproot backend (**Mason Proffitt**):

- Recently been extended to run natively all the adl benchmarks
- Is in the process of being converted to use dask-awkward
- One remaining axis tracking issues, and…
- One outstanding issue in dask-awkward.

ATLAS Analysis Data Workflow



Users Running Analysis run Sys Errors

Or direct analysis

Users Running Analysis run Calibration and Sys Errors

Versions

Software Release You Extract Your Analysis Data → “AOD Fix”

Calibration & Systematics Tags → Custom to your analysis, data period, and “time”

(Important) Versioning

Reconstructed Software Release	Defines the Data Format and EDM, and after-reco bug fixes	**Release Series**: • 21 – for Run 2 • 22&24 for Run 3 • Not compatible • Many sub releases for analysis (R22 has ~250 sub-releases with bug fixes, features – new one every few weeks)
Corrections Applied (Corrections Versions)	Defined by the Combined Performance (Object) groups	**Released 'in-time' with conferences:** • Calibration files, via cvmfs and https • Configured by job configuration • Systematic Errors are currently tightly tied to this system

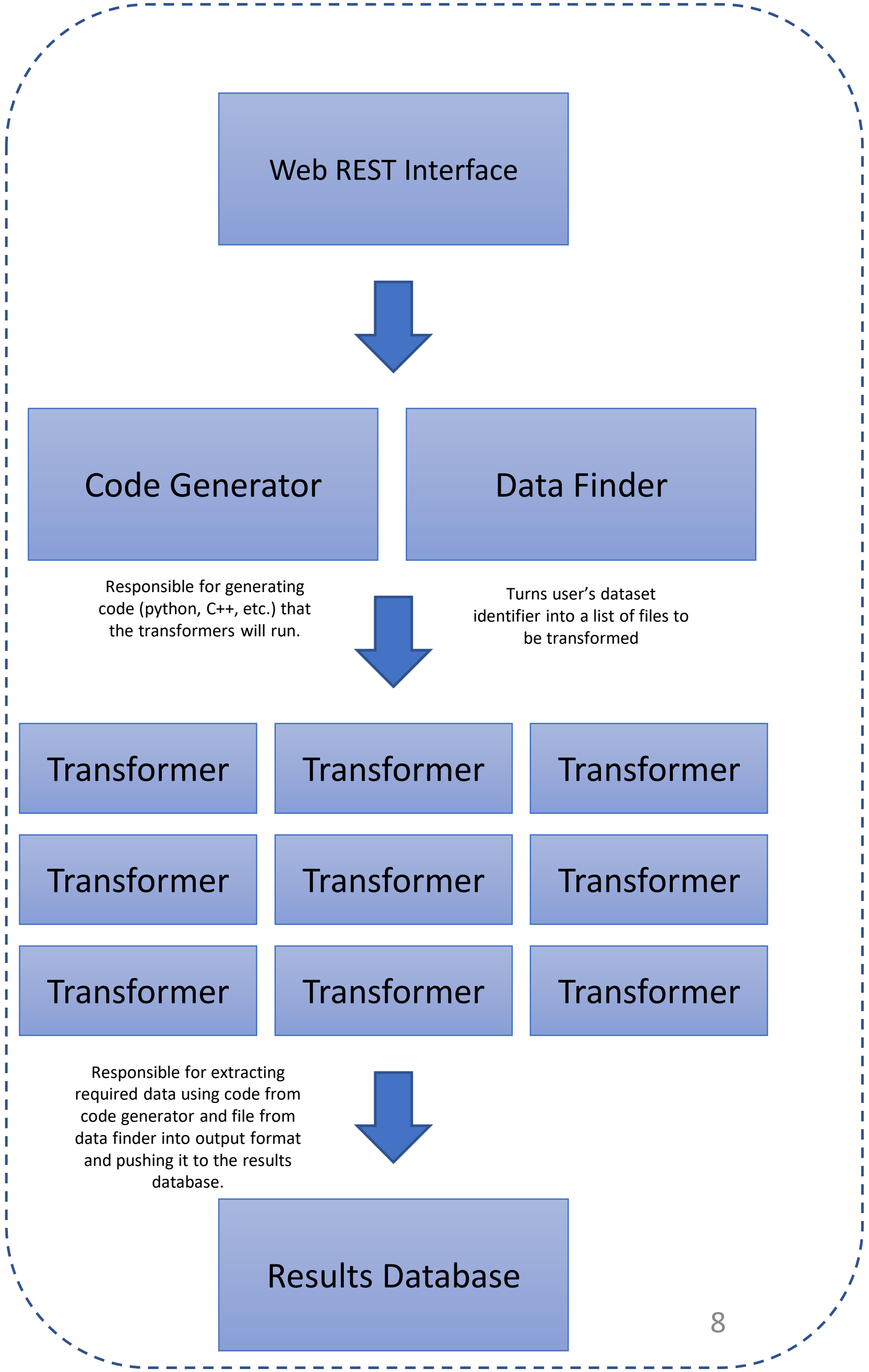
ServiceX

1. User must be able to specify what “version” of everything they want

2. Transformers must be running **correct and expected** versions and load corrections properly

<u>One other tricky aspect</u>

- ATLAS objects don’t always return pointers or C++ objects or references
- Since no assumptions can be made, the func_adl translator needs to know basic C++ type information.
- Currently use ROOT type system to understand ATLAS EDM… so works very poorly for templated functions!



Web REST Interface

Code Generator

Data Finder

Responsible for generating code (python, C++, etc.) that the transformers will run.

Turns user’s dataset identifier into a list of files to be transformed

Transformer

Transformer

Transformer

Transformer

Transformer

Transformer

Transformer

Transformer

Transformer

Responsible for extracting required data using code from code generator and file from data finder into output format and pushing it to the results database.

Results Database

What does a proper configuration look like?

- **Complete job options** are transmitted down
- Along with **type information**
- Gives the user a lot of flexibility on how they extract the data (to much for beginners!)

- Makes the queries long (perhaps > 10K sometimes!)
- The process of translating a simple func_adl statement to this is complex and not easy to understand
- Some options has been exposed to make it easy… other options…

```
DEBUG:servicex.servicex:JSON to be sent to servicex: {'selection': '(call Select (call MetaData (call MetaData (call SelectMany (call MetaData (call MetaData (call MetaData (call
MetaData (call MetaData (call MetaData (call MetaData (call MetaData (call MetaData (call MetaData (call EventDataset \'bogus.root\') (dict (list \'metadata_type\' \'name\'
\'include_files\' \'container_type\' \'element_type\' \'contains_collection\' \'link_libraries\') (list \'add_atlas_event_collection_info\' \'Jets\' (list \'xAODJet/JetContainer.h\')
\'DataVector<xAOD::Jet_v1>\' \'xAOD::Jet_v1\' True (list \'xAODJet\')))) (dict (list \'metadata_type\' \'name\' \'script\') (list \'add_job_script\' \'sys_error_tool\' (list \'# pulled
from:https://gitlab.cern.ch/atlas/athena/-/blob/21.2/PhysicsAnalysis/Algorithms/JetAnalysisAlgorithms/python/JetAnalysisAlgorithmsTest.py\' \'# Set up the systematics
loader/handler service:\' \'from AnaAlgorithm.DualUseConfig import createService\' \'from AnaAlgorithm.AlgSequence import AlgSequence\' \'calibrationAlgSeq = AlgSequence()\'
"sysService = createService( \'CP::SystematicsSvc\', \'SystematicsSvc\', sequence = calibrationAlgSeq )" "sysService.systematicsList = [\'NOSYS\']" \'# Add sequence to job\')))) (dict
(list \'metadata_type\' \'name\' \'script\' \'depends_on\') (list \'add_job_script\' \'pileup_tool\' (list \'from AsgAnalysisAlgorithms.PileupAnalysisSequence import
makePileupAnalysisSequence\' \'\' \'# Use the sh object (sample Handler) to get the first tile and extract the filename\' \'# from it, which can then be used to fetch the MC
campaign. `calib.datatype`\' \'# should contain `data` or `mc`\' \'pileupSequence = makePileupAnalysisSequence(\' \'    "mc", files=sh.at(0).fileName(0)\' \')\'
\'pileupSequence.configure(inputName={}, outputName={})\' \'print(pileupSequence)  # For debugging\' \'\' \'calibrationAlgSeq += pileupSequence\') (list \'sys_error_tool\')))) (dict
(list \'metadata_type\' \'name\' \'script\' \'depends_on\') (list \'add_job_script\' \'corrections_jet\' (list \'jetContainer = "AntiKt4EMPFlowJets"\' \'from
JetAnalysisAlgorithms.JetAnalysisSequence import makeJetAnalysisSequence\' \'\' \'jetSequence = makeJetAnalysisSequence("mc", jetContainer)\'
\'jetSequence.configure(inputName=jetContainer, outputName=jetContainer + "_Base_%SYS%")\' \'jetSequence.JvtEfficiencyAlg.truthJetCollection =
"AntiKt4TruthDressedWZJets"\' \'try:\' \'    jetSequence.ForwardJvtEfficiencyAlg.truthJetCollection = (\' \'        "AntiKt4TruthDressedWZJets"\' \'    )\' \'except AttributeError:\' \'
pass\' \'\' \'calibrationAlgSeq += jetSequence\' \'print(jetSequence)  # For debugging\' \'\' \'# Include, and then set up the jet analysis algorithm sequence:\' \'from
JetAnalysisAlgorithms.JetJvtAnalysisSequence import makeJetJvtAnalysisSequence\' \'\' \'jvtSequence = makeJetJvtAnalysisSequence("mc", jetContainer, enableCutflow=True)\'
\'jvtSequence.configure(\' \'    inputName={"jets": jetContainer + "_Base_%SYS%"},\' \'    outputName={"jets": jetContainer + "Calib_%SYS%"},\' \')\' \'calibrationAlgSeq +=
jvtSequence\' \'print(jvtSequence)  # For debugging\' \'output_jet_container = "AntiKt4EMPFlowJetsCalib_%SYS%"\' \'# Output jet_collection = AntiKt4EMPFlowJetsCalib_NOSYS\')
(list \'pileup_tool\')))) (dict (list \'metadata_type\' \'name\' \'script\' \'depends_on\') (list \'add_job_script\' \'corrections_muon\' (list "muon_container = \'Muons\'" \'from
MuonAnalysisAlgorithms.MuonAnalysisSequence import makeMuonAnalysisSequence\' "muonSequence = makeMuonAnalysisSequence(\'mc\', workingPoint=\'Medium.NonIso\',
postfix = \'Medium_NonIso\')" \'muonSequence.configure( inputName = muon_container,\' "                    outputName = muon_container + \'Calib_MediumNonIso_%SYS%\' )"
\'calibrationAlgSeq += muonSequence\' \'print( muonSequence ) # For debugging\' \'output_muon_container = "MuonsCalib_MediumNonIso_%SYS%"\' \'# Output
muon_collection = MuonsCalib_MediumNonIso_NOSYS\') (list \'corrections_jet\')))) (dict (list \'metadata_type\' \'name\' \'script\' \'depends_on\') (list \'add_job_script\'
\'corrections_electron\' (list \'from EgammaAnalysisAlgorithms.ElectronAnalysisSequence import makeElectronAnalysisSequence\' "electronSequence =
makeElectronAnalysisSequence( \'mc\', \'MediumLHElectron.NonIso\', postfix = \'MediumLHElectron_NonIso\')" "electronSequence.configure( inputName = \'Electrons\'," "
outputName = \'Electrons_MediumLHElectron_NonIso_%SYS%\' )" \'calibrationAlgSeq += electronSequence\' \'print( electronSequence ) # For debugging\'
\'output_electron_container = "Electrons_MediumLHElectron_NonIso_%SYS%"\' \'# Output electron_collection = Electrons_MediumLHElectron_NonIso_NOSYS\') (list
\'corrections_muon\')))) (dict (list \'metadata_type\' \'name\' \'script\' \'depends_on\') (list \'add_job_script\' \'corrections_photon\' (list \'#TODO: Get photon
correcoutput_tau_container," \'    },\' \'    outputName = {\' "        \'electrons\' : \'Electrons_MediumLHElectron_NonIso_OR_%SYS%\'," "        \'photons\'   : \'Photons_OR_%SYS%\'," "
\'muons\'     : \'MuonsCalib_MediumNonIso_OR_%SYS%\'," "        \'jets\'      : \'AntiKt4EMPFlowJetsCalib_OR_%SYS%\'," "        \'taus\'      : \'TauJets_Tight_OR_%SYS%\'" \'    })\'
\'calibrationAlgSeq += overlapSequence\' \'# Output electron_collection = Electrons_MediumLHElectron_NonIso_OR_NOSYS\' \'# Output photon_collection = Photons_OR_\' \'#
Output muon_collection = MuonsCalib_MediumNonIso_OR_NOSYS\' \'# Output jet_collection = AntiKt4EMPFlowJetsCalib_OR_NOSYS\' \'# Output tau_collection =
TauJets_Tight_OR_NOSYS\') (list \'corrections_tau\')))) (dict (list \'metadata_type\' \'name\' \'script\' \'depends_on\') (list \'add_job_script\' \'add_calibration_to_job\' (list
\'calibrationAlgSeq.addSelfToJob( job )\' \'print(job) # for debugging\') (list \'corrections_overlap\')))) (lambda (list e) (call (attr e \'Jets\') \'AntiKt4EMPFlowJetsCalib_OR_NOSYS\')))
(dict (list \'metadata_type\' \'type_string\' \'method_name\' \'return_type\') (list \'add_method_type_info\' \'xAOD::Jet_v1\' \'pt\' \'double\'))) (dict (list \'metadata_type\'
\'name\' \'body_includes\') (list \'inject_code\' \'xAODJet/versions/Jet_v1.h\' (list \'xAODJet/versions/Jet_v1.h\')))) (lambda (list j) (call (attr j \'pt\'))))', 'result-destination': 'object-
store', 'result-format': 'root-file', 'chunk-size': '1000', 'workers': '20', 'codegen': 'atlasxaod', 'did': 'local_dataset'}
```

What stuff can you currently (easily) control

The ATLAS configuration system gives users 1000000000000000000000000’s of options. You need zero when you start, and perhaps 20 by the time your analysis is done.**

Distil down to **only options needed**:

- Some of these are per-analysis choices
- Others have to do with default selection cuts, which have to be matched to corrections, and other things that enable “pit of success”

How this works

1. As the query is processed, metadata is added
2. The metadata controls the job options that configure the job
3. One can override almost anything, but making deep changes isn’t trivial!
4. Mostly, there is x10 more metadata than C++!

```
class CalibrationEventConfig:
    # Name of the jet collection to calibrate and use by default
    jet_collection: str
    # Name of the truth jets to be used for the jet calibration
    jet_calib_truth_collection: str
    # Name of the electron collection to calibrate and use by default
    electron_collection: str
    # The working point (e.g. xxx)
    electron_working_point: str
    # The isolation (e.g. xxxx)
    electron_isolation: str
    # Name of the photon collection to calibrate and use by default.
    photon_collection: str
    # The working point (e.g. xxx)
    photon_working_point: str
    # The isolation (e.g. xxxx)
    photon_isolation: str
    # Name of the muon collection to calibration and use by default.
    muon_collection: str
    # The working point (e.g. xxx)
    muon_working_point: str
    # The isolation (e.g. xxxx)
    muon_isolation: str
    # Name of the tau collection to calibrate and use by default.
    tau_collection: str
    # The working point (e.g. xxxx)
    tau_working_point: str
    perform_overlap_removal: bool
    # ** Data Type (data, MC, etc., used for pileup, jet corrections, etc.)
    datatype: str
    # ** Run calibrations by default (PHYSLITE vs PHYS)
    calibrate: bool
    # ** True if we can return uncalibrated (PHYSLITE doesn't)
    uncalibrated_possible: bool
```

Draft of “whats needed”

** slight exaggeration…

What I've Been Working On Recently…

Release 21 has been working well:
- Most queries up to now have all been based on R21/Run 2 data

Release 22
- This is the new part
- In particular, getting it to work correctly along side of R21
- R21 and R22 are subtly, but importantly, different

Release 24
- Given the way ATLAS is naming its releases, R24 is the new R22
- So, no extra work has to be done

In github, 3 packages:

- ATLAS Type Interpreter
    - Scans for all type information
    - Knows the scripts for all the releases
    - Produces one yaml file describing the whole release
- Type Package Builder
    - Uses the output yaml file to produce a python package.
    - Only R21 is on pypi right now
    - Waiting until full testing before releasing R22.
- Automated Tester and Producer
    - Working towards a CI that will build the python package given a release version
    - This is brand new – and reflects the fact that above two are too hard to use

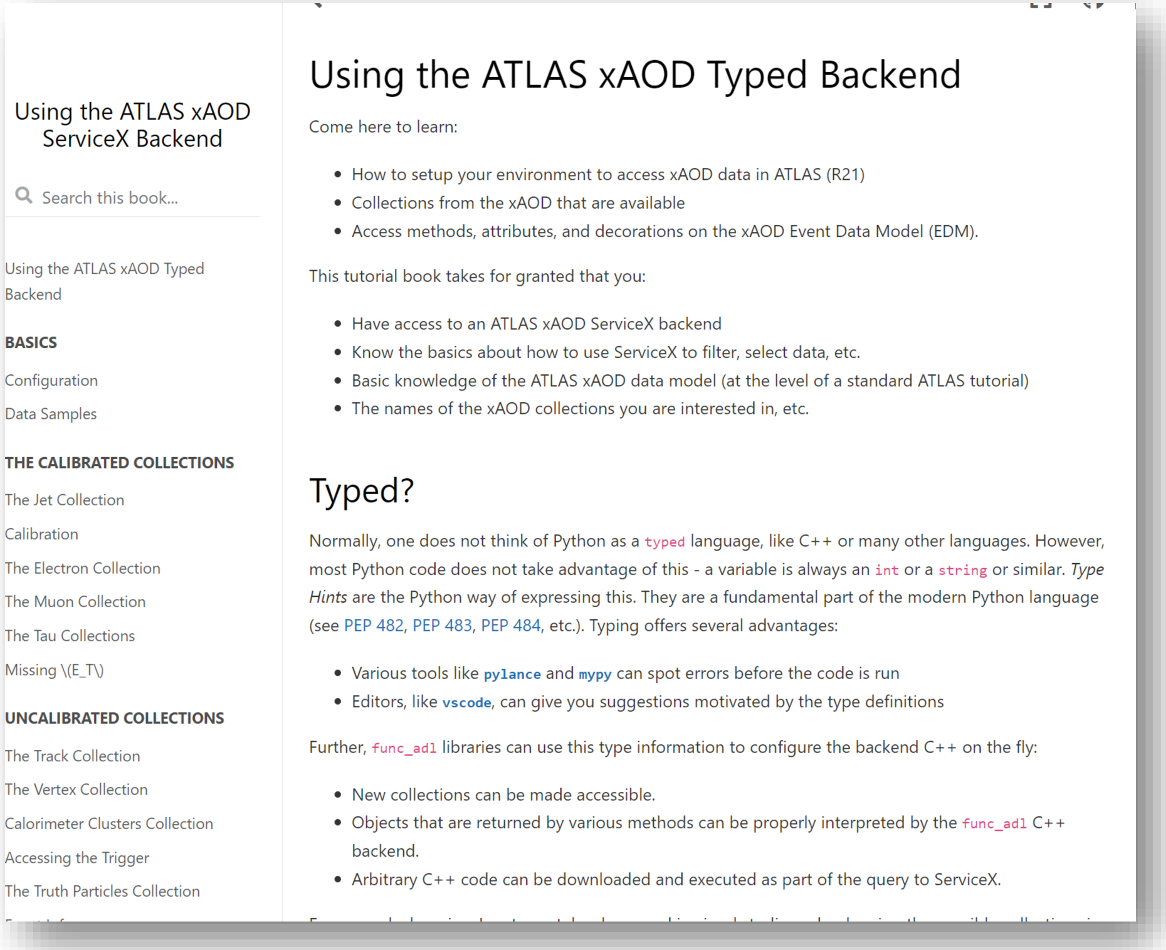
[Repo]
[Docs Served On the Web]

Documentation

How to:

- Access calibrated objects (jets, muons, etc)
- How to access uncalibrated objects (tracks, MC particles, etc.)

- Built as a Jupyter Book
- All code runs
- Includes instructions for building working venv
- Not updated for R22 (and physlite) yet.



Using the ATLAS xAOD ServiceX Backend

Search this book...

Using the ATLAS xAOD Typed Backend

BASICS

Configuration

Data Samples

THE CALIBRATED COLLECTIONS

The Jet Collection

Calibration

The Electron Collection

The Muon Collection

The Tau Collections

Missing \(E_T\)

UNCALIBRATED COLLECTIONS

The Track Collection

The Vertex Collection

Calorimeter Clusters Collection

Accessing the Trigger

The Truth Particles Collection

Using the ATLAS xAOD Typed Backend

Come here to learn:

- How to setup your environment to access xAOD data in ATLAS (R21)
- Collections from the xAOD that are available
- Access methods, attributes, and decorations on the xAOD Event Data Model (EDM).

This tutorial book takes for granted that you:

- Have access to an ATLAS xAOD ServiceX backend
- Know the basics about how to use ServiceX to filter, select data, etc.
- Basic knowledge of the ATLAS xAOD data model (at the level of a standard ATLAS tutorial)
- The names of the xAOD collections you are interested in, etc.

Typed?

Normally, one does not think of Python as a `typed` language, like C++ or many other languages. However, most Python code does not take advantage of this - a variable is always an `int` or a `string` or similar. *Type Hints* are the Python way of expressing this. They are a fundamental part of the modern Python language (see PEP 482, PEP 483, PEP 484, etc.). Typing offers several advantages:

- Various tools like `pylance` and `mypy` can spot errors before the code is run
- Editors, like `vscode`, can give you suggestions motivated by the type definitions

Further, `func_adl` libraries can use this type information to configure the backend C++ on the fly:

- New collections can be made accessible.
- Objects that are returned by various methods can be properly interpreted by the `func_adl` C++ backend.
- Arbitrary C++ code can be downloaded and executed as part of the query to ServiceX.

Final Word on Systematic Errors

1

ATLAS Systematic Errors have Event-Wide implications – they are not per-object

- The entanglement is triggered by overlap removal (x2) and the fact that things like Jet Systematics shift jets past various cuts.
- The experiment is hard at work understanding if both can be removed from the systematic error calculation

2

We threw out our systematic error calculation twiki about 10 years ago and built our current dual-use tools

- These have our EDM and the calculations deeply entangled
- Calibration & error constants are distributed by https and cvmfs
- Tools are well adapted to this infrastructure
- Calibration and errors are fairly sophisticated (often involving NN’s) and sometimes expensive even without EDM access
- Experiment is trying to simplify systematic errors without losing fidelity
- And support RDF, python, and our C++ frameworks (quad-use calibration tools?)

Systematic Errors & Service X

ServiceX

Has access to full fidelity systematic error calculation now

But…
* Not obvious yet how to feel the data to the end user…

Regardless, current ServiceX can handle systematic errors
- But only one per query!
- So this needs real work!

Using the current xAOD model there are two standard approaches:
- Create duplicate tree’s of the same data, with shifts for each of ~100 systematic errors
- Create one tree with new branches for each variation

Further ATLAS Context

HL-LHC Demonstrators

- To be used to help ATLAS decide where to invest resources for HL-LHC
- Outcomes of demonstrators is expected at end of summer 2024
- They don't have to be complete – but do need to show the value proposition
- Incorporated into the ATLAS Computing TDR which is to be written over the summer/fall

Because we do not yet have OpenData (but soon?)

Analysis Grand Challenge Demonstrator

We will **demonstrate the Python-ecosystem analysis workflow** from the IRIS-HEP Analysis Grand Challenge on **ATLAS internal Release 22 data** and on ATLAS analysis facilities.

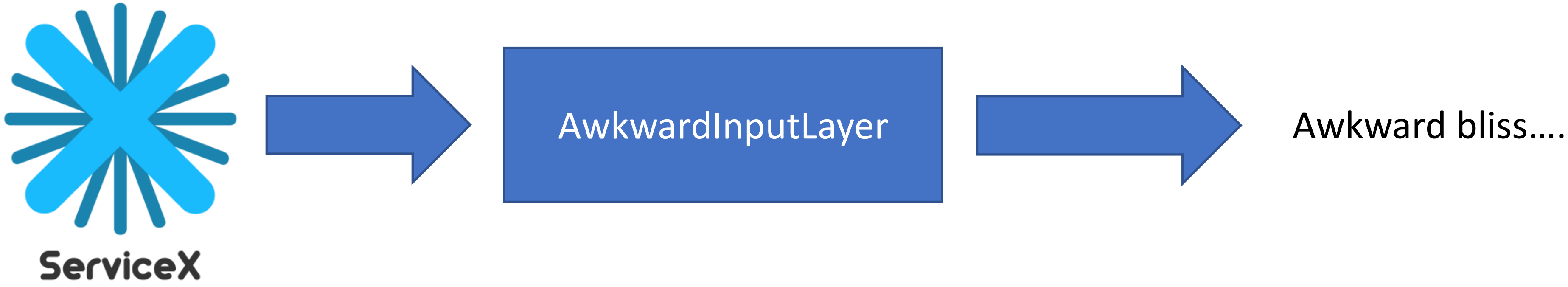
The IRIS-HEP Analysis Grand Challenge (AGC) seeks to demonstrate, at HL-LHC scales, the full analysis pipeline. Data starts in post-production format, PHYS or PHYSLITE or a DAOD for ATLAS. The end result is a limit or parameter measurement that usually comes from a statistical tool, like pyhf/cabinetry. The pipeline includes the calculation of systematic errors. The AGC is, on the one hand, an integrative effort to make sure tools work together and scale to the data loads we expect. It is also a testbed for the user interface.

The **ATLAS version of the AGC would use a "simple" benchmark analysis**, like a top quark cross-section or a simple Exotics search. Most importantly, it would be one that had been completed on Run 2 data (or was well understood on Run 3 data).

What about awkward-dask Integration

From Linsey’s talk this morning, this is an obvious integration we want to do…

ServiceX/func_adl is delayed execution by design…



ServiceX

AwkwardInputLayer

Awkward bliss….

Current Status:

- Prototype built
- Issues to work around:
    - Do not know # of input files at start of query
    - How to move operations between awkward and service to minimize data transferred/efficiency

Can fake the first query, and then cache answer for later…

No answer yet, but people keep telling me this is easy (also not crucial for v1)… ;-)

Conclusions

- Next Steps
  - Updating documentation with new versions
    - R22 & PHYS/PHYSLITE
  - Automating the Package Generation
  - Benchmark Analysis with no Systematic Errors
  - The addition of Systematic Errors
- Some issues…
  - (besides Tal’s talk!)
  - How to address ATLAS’s constant changing releases!