ServiceX: A distributed, caching, columnar data delivery service

B. Galewsky[A], R. Gardner[B], M. Neubauer[A], J. Pivarski[C], L. Gray[E]
I. Vukotic[B], G. Watts[D], M. Weinberg[B]

A. University of Illinois at Urbana-Champaign, B. The University of Chicago,

C. Princeton University, D. University of Washington, E. Fermilab

HSF DAWG -- DOMA Access joint meeting

3/23/2020

Delivering data to new analysis platforms

First introduced in Feb 2018 whitepaper: delivery of data from lakes to clients (insulate from upstream systems, remove latencies, reformat, filter & possibly accelerate)

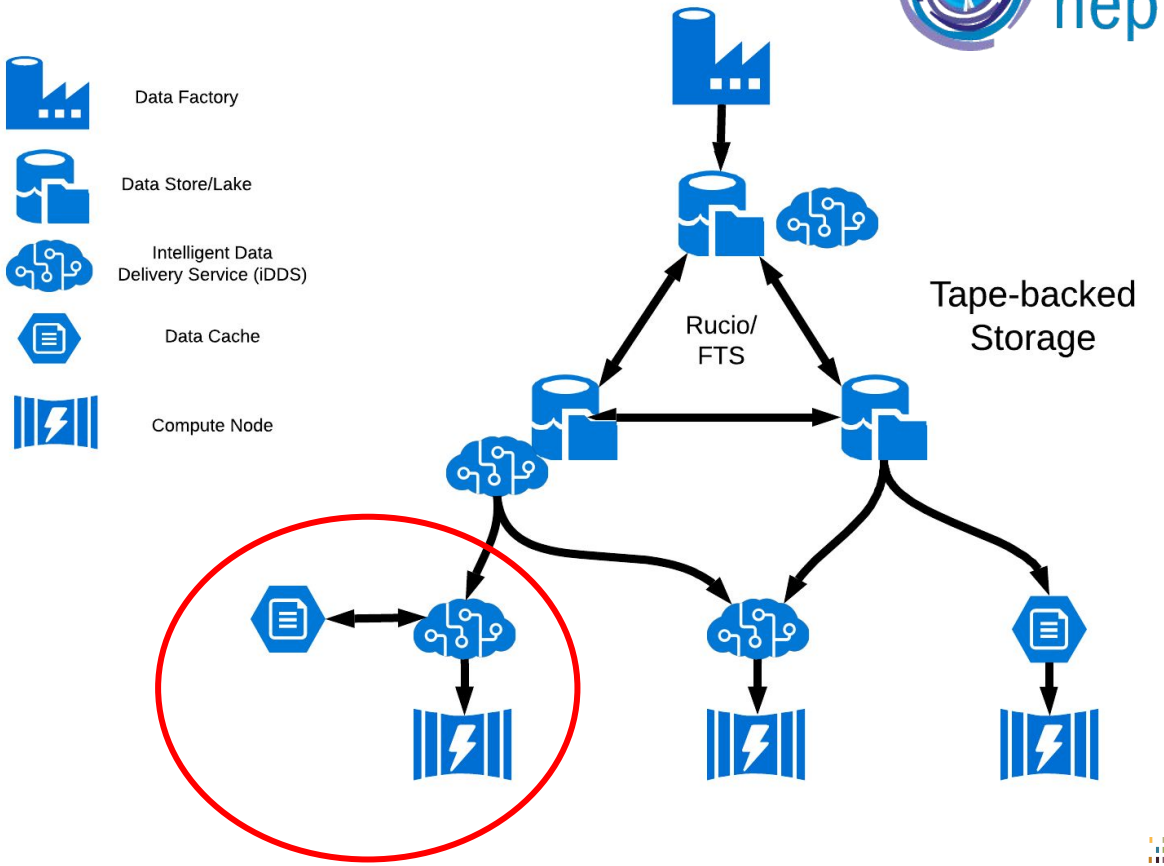
ServiceX focuses on integration with Rucio and reformatting for pythonic tools & **endstage analysis systems**

DOMA/AS groups interested in ***R&D*** for data delivery for analysis of columnar and other data formats

Supports multiple input types (xAOD, flat ntuples, ...) and common data mgt (Rucio, XCache)

Utilize industry standard tools (GKE, on-prem Kubernetes, Helm, Kafka, Redis, Spark, ...)

Reproducible, portable deployments



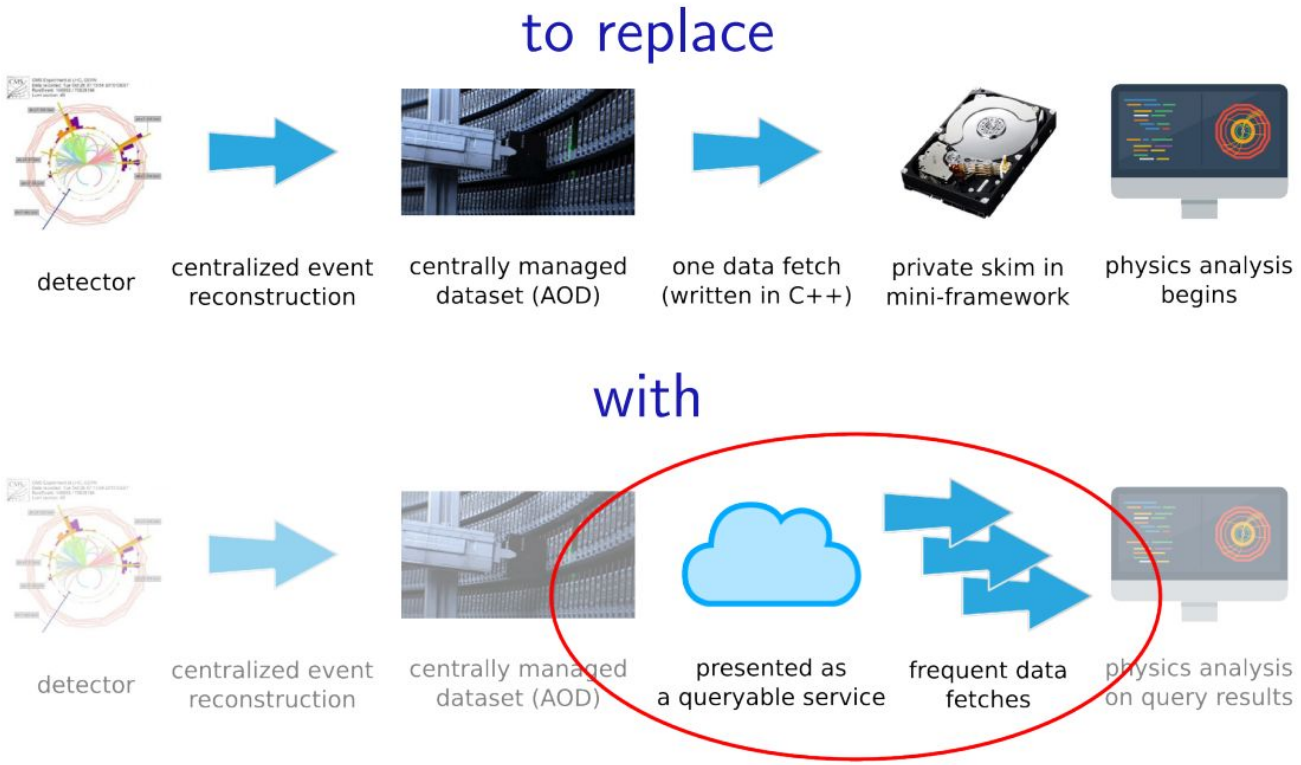
Classic analysis workflow

xAOD → DAOD → flat ntuples → skimmed ntuples → histograms/plots

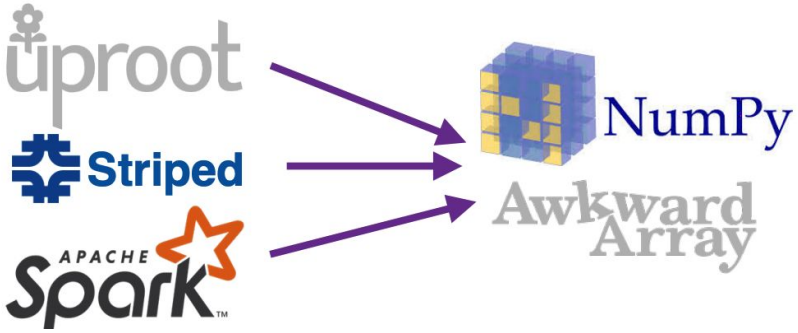
- First two formats are prescribed, but enormous variation in after that

Standard analysis might have "primary ntuple"

- Write ntuplization code to dump xAOD into flat trees with specialized objects
- Submit jobs to HTCondor by hand
- Primary ntuple then skimmed/trimmed; some data replicated (multiple times)
- Selections/cutflows baked into analysis
- Adding new variables means throwing previous skim, replicating everything



Columnar data R&D efforts

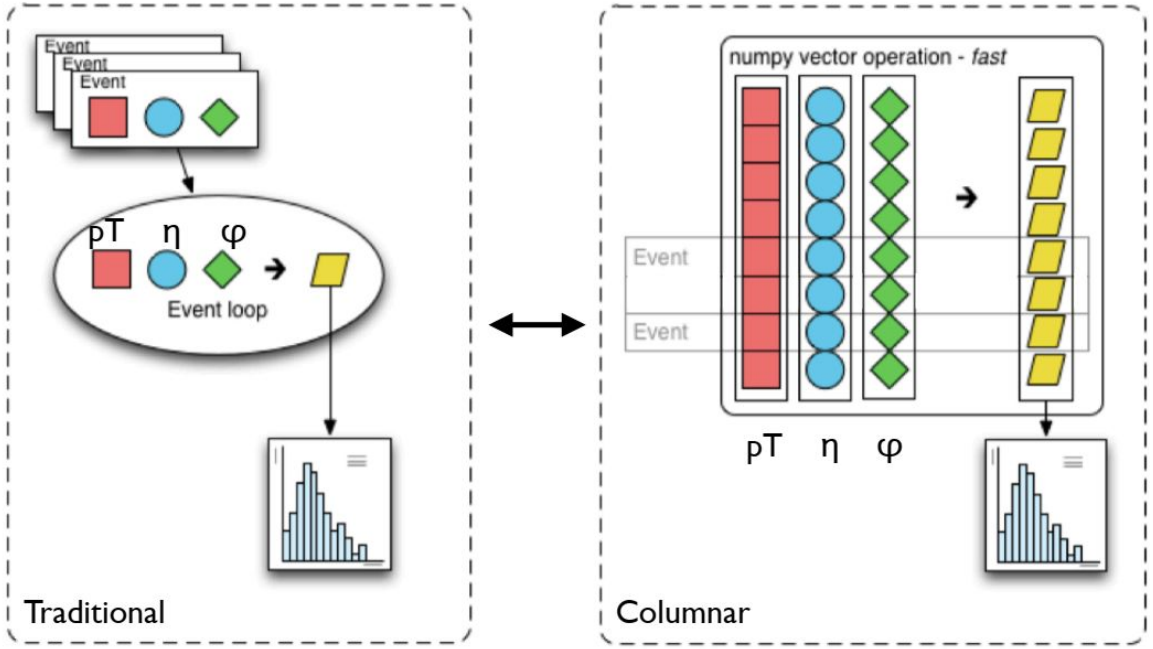


Recast data so attributes of physics objects grouped into contiguous columns, rather than grouping by event and then object

- Much more efficient for processing!
- Updating event content (or corrections) can be done by adding columns to existing data
- Can cache only necessary data for computation;
  No longer need to load entire event in memory

However, this is a significant change for analyzer

- New syntax can be simpler, more expressive
- Imagine analysis code with no for() loops…



ServiceX goals

Adding components to the intelligent data delivery ecosystem. Input-agnostic service to enable on-demand data delivery.

Tailored for **nearly-interactive,** high-performance **array-based analyses**

- Provide uniform interface to data storage services; users don't need to know how or where data is stored
- Capable of **on-the-fly data transformations** into variety of formats (ROOT files, HDF5, Arrow buffers, Parquet files, ...)
- Pre-processing functionality: Unpack compressed formats, filter events in place, project data columns

Support for columnar analyses. Start from any format, extract only needed columns

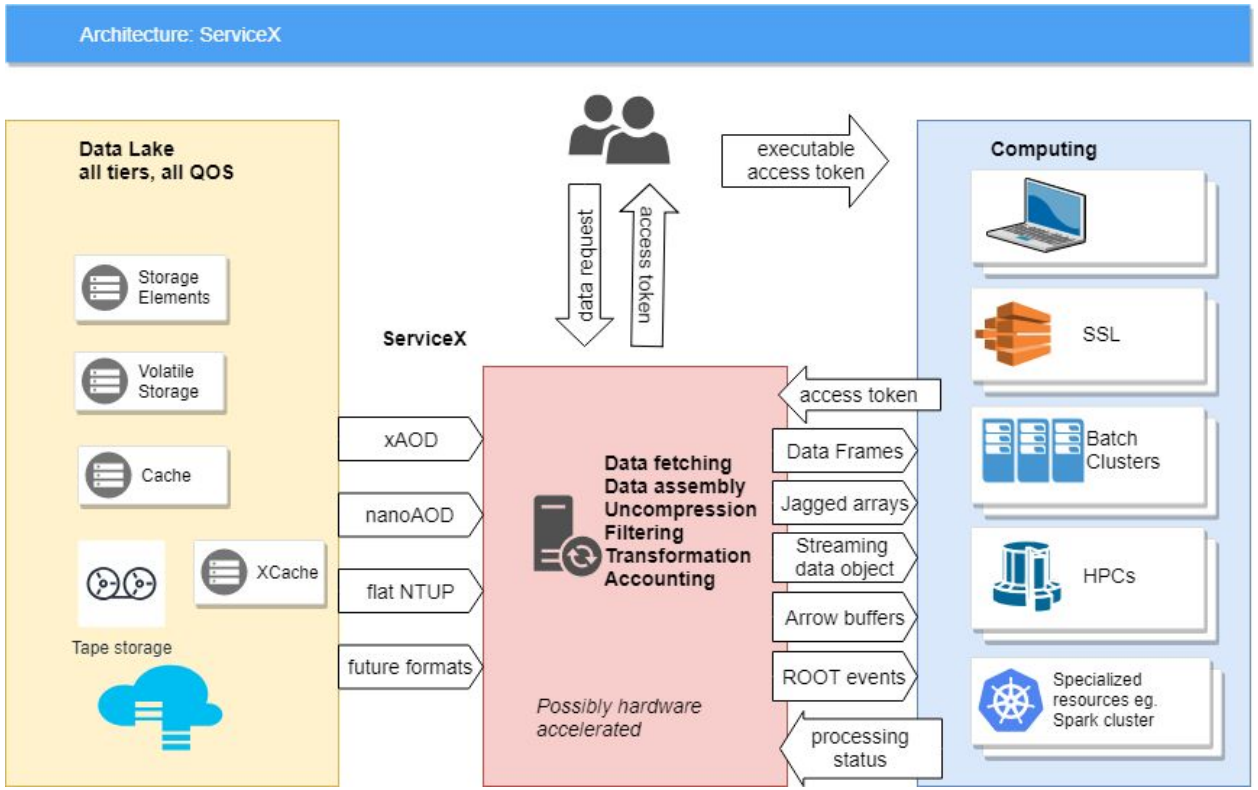
ServiceX components

Users specify needed events/columns and desired output format

- Use metadata tags (real/sim data, year, energy, run number, …)
- Any required preselection

ServiceX

- Queries backend (Rucio) to find data
- Gives unique token to identify request
- Access data from storage through XCache
- Validates request: Check on small sample, determine message size
- Extract requested columns, perform data transformations
- Send output to message broker for analysis



ServiceX implementation

System designed to be modular

- Can switch out modules to transform different types of input data, swap schedulers, …

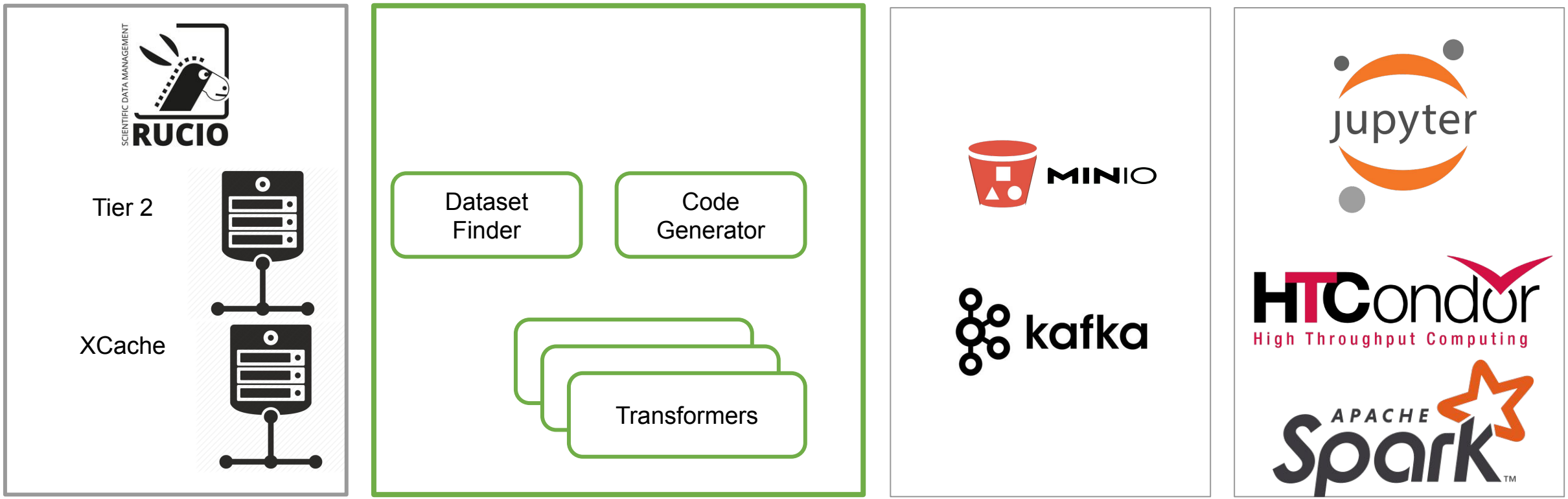
Implemented as central service in Kubernetes cluster on Scalable Systems Lab (SSL) cluster

- Easy to deploy: Just use Helm chart to define plugins to run
- Service can be deployed on variety of systems, including individual laptops
- **Reproducible pattern** for deployment on Kubernetes clusters (e.g. **Tier2s**, **institutional k8s T3**?)

Composed of multiple deployments: REST API server, DID finder, transformer, message broker

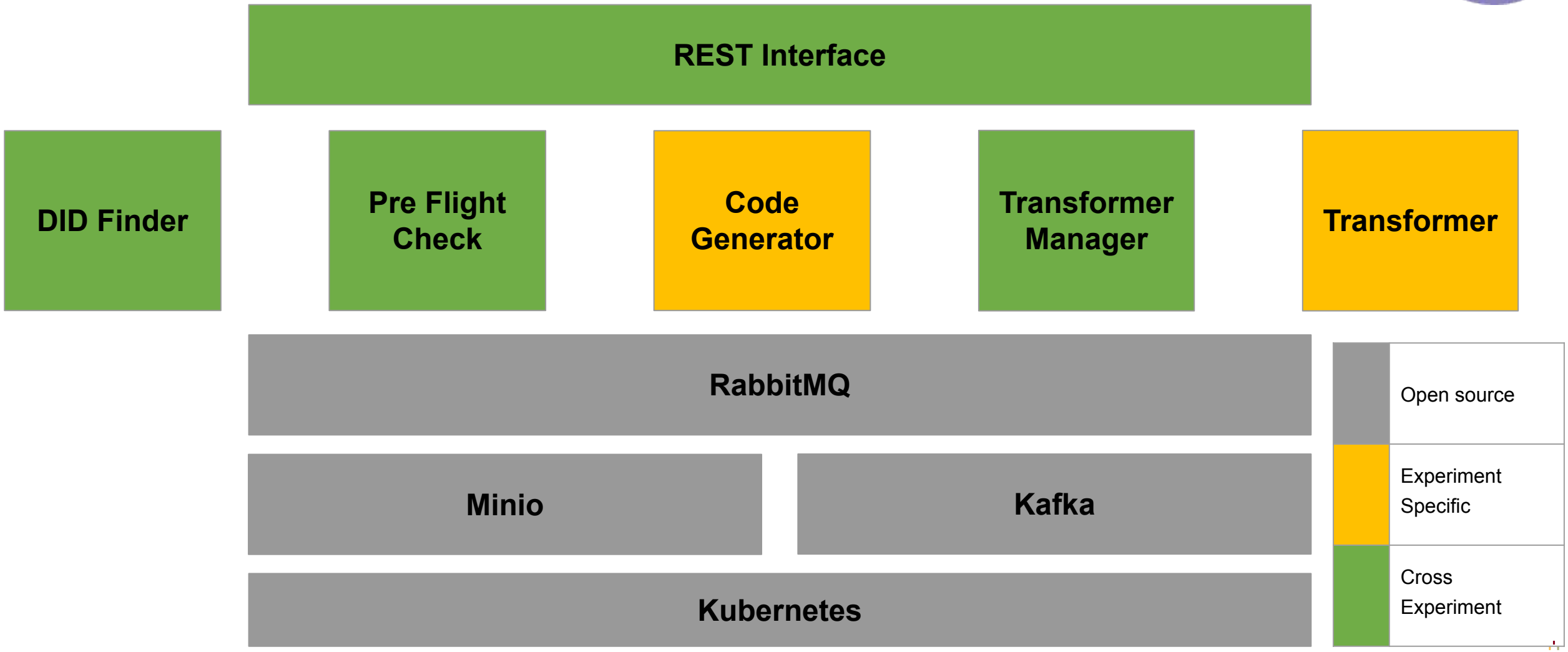
- API server: Manages requests via RabbitMQ with Postgres DB
- DID finder: Queries data lake via Rucio, writes ROOT files to XCache
- Transformer: Takes input files from XCache, outputs in various formats (ROOT files with flat trees, Awkward arrays, Parquet, Arrow tables, …)
- Kafka manager: Receives input from producer (transformer) and makes topics available to consumers (analysis jobs)

ServiceX in the IRIS-HEP ecosystem

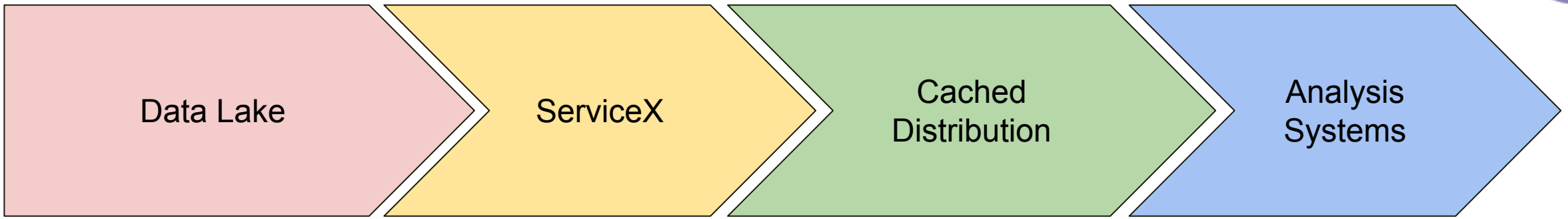


IRIS-HEP Scalable Systems Lab

ServiceX architecture



Connections to DOMA



ServiceX is part of DOMA's iDDS

- feeds data to downstream analysis systems
- enables data transformations developed in individual environments to be scaled up to production-based operations

ServiceX is being prototyped using IRIS-HEP's Scalable Systems Lab

- includes reproducible pattern for deployment
- entire project implemented as central service in Kubernetes cluster on SSL
- takes advantage of SSL infrastructure support to develop new features quickly.

ServiceX performance

- 10TB across 7794 files (~ 1.3GB/file)
- 10-column test: reads ~ 10% of file
  - Time for transformer to process file: ~ 13 seconds
  - ~ 1 second each for file access and output to MinIO/Kafka
  - Output size ~ 3MB (~ 400 reduction from input)
  - 400-transformer test completes in < 5 minutes
- 100-column test: reads ~ 30% of file
  - Total time to process file: ~ 31 seconds
  - ~ 1 second for file access (unless high load)
  - Output size ~ 38MB (~ 35 reduction from input)
  - 400-transformer test completes in < 25 minutes

ServiceX demo

Sample transform request including dataset to be transformed and output columns

Development version; some of these decisions will be hidden from the user

User receives unique request ID

Rudimentary updates on the progress of the transformation

To be augmented with status plots

ServiceX demo

This notebook illustrates the use of ServiceX to create a request for specific columns of data from a file stored in Rucio, and the reading of these columns to create analysis plots.

In [1]:
```
import requests
import tempfile
import pyarrow.parquet as pq
import pyarrow as pa
import awkward
from confluent_kafka import Consumer, KafkaException
import uproot_methods
from coffea import hist
import matplotlib.pyplot as plt
```

Transform the data

We start with the creation of the request inside the service. The user specifies the dataset to be transformed, along with the columns of interest. Here we transform a 700GB MC dataset of ~ 2 million Z -> ee events distributed across 17 files. The columns from this dataset are then streamed to Kafka, a message broker which makes them available to the user for analysis.

Meanwhile, the service returns a unique string that serves as the ID of the request. The user can use the request ID to get updates on the progress of the system and to identify their data in the message broker.

In [2]:
```
response = requests.post("http://release-name-servicex-app.servicex:8000/servicex/transformation", json={
    "did": "mc15_13TeV:mc15_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.DAOD_STDM3.e3601_s2576_s2132_r6630_r6264_p2363
    "columns": "Electrons.pt(), Electrons.eta(), Electrons.phi(), Electrons.e()",
    "image": "sslhep/servicex-transformer:v0.1-prep",
    "result-destination": "kafka",
    "kafka":{
        "broker": "release-name-kafka:9092"
    },
    "chunk-size": 3000,
    "workers": 17
})

print("Request ID:", response.json()['request_id'])
request_id = response.json()["request_id"]
status_endpoint = servicex_endpoint+"/transformation/{}/status".format(request_id)
```

```
Request ID: 826ea2b2-0849-4790-9427-afe6498348eb
```

Get updates on the transformation

Once the request is sent, we can get information back on the status of the request. Note that the transformer begins running as soon as the first files from the dataset are found within Rucio; some of the information does not become available until all the files are discovered (e.g. the total number of files remaining).

In [9]:
```
status = requests.get(status_endpoint).json()

print("Request ID:                 ", status['request-id'])
print("Number of files processed:  ", status['files-processed'])
print("Number of files remaining:  ", status['files-remaining'])
if status['stats']:
    print("Number of events processed: ", status['stats']['total-events'])
    print("Size of files processed:    ", status['stats']['total-bytes'] / 1.0e9, "GB")
    print("Number of batches sent:     ", status['stats']['total-messages'])
    print("Total time:                 ", status['stats']['total-time'] / (n_workers * 60), "min")
```

```
Request ID:                 826ea2b2-0849-4790-9427-afe6498348eb
Number of files processed:  17
Number of files remaining:  0
Number of events processed: 1993800
Size of files processed:    0.2005268 GB
Number of batches sent:     666
Total time:                 1.2245098039215687 min
```

ServiceX demo

Output cached in message broker

Can be read out asynchronously, re-run

User analysis code

Time to go from xAOD or DAOD to analysis plot only a couple of minutes

Analyzing the output

At any time after the files have begun to transform, we may extract some of the data an analyze it. If the string corresponding to the `group.id` is not changed, subsequent reads will start from the last batch processed, to ensure there is no double-counting. (In order to read the entire output from the beginning, simply change the `group.id` to a different string.)

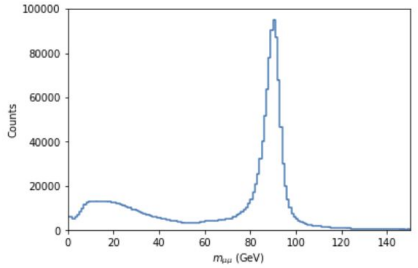
In [ ]:
```
conf = {'bootstrap.servers': broker_name, 'group.id': "foo1",
        'default.topic.config': {'auto.offset.reset': 'smallest'}}

c = Consumer(conf)

c.list_topics().topics[request_id]
c.subscribe([request_id])
timeout = 10.0  # Need a long timeout to allow for partition assignment
running = True
all_mass_hists = None
```

In [12]:
```
n_total_events = 0
while running:
    msg = c.poll(timeout=timeout)
    if msg is None:
        running = False
        continue
    if msg.error():
        raise KafkaException(msg.error())
    else:
        # Proper message
        buf = msg.value()
        reader = pa.ipc.open_stream(buf)
        batches = [b for b in reader]
        for batch in batches:
            arrays = awkward.fromarrow(batch)
            v_particles = uproot_methods.TLorentzVectorArray.from_ptetaphi(
                arrays['Electrons_pt'], arrays['Electrons_eta'],
                arrays['Electrons_phi'], arrays['Electrons_e']
            )
            v_particles = v_particles[v_particles.counts >= 2]
            diparticles = v_particles[:, 0] + v_particles[:, 1]

            mass_hist = hist.Hist('Counts', hist.Bin('mass', r'$m_{\mu\mu}$ (GeV)', 150, 0.0, 150.0))
            mass_hist.fill(mass=diparticles.mass/1000.0)

            if all_mass_hists:
                all_mass_hists = all_mass_hists.add(mass_hist)
            else:
                all_mass_hists = mass_hist

            n_total_events += len(arrays.tolist())
            # print("Number of events: " + str(n_events))
```

The output above is aggregated into a single histogram for plotting:

In [13]:
```
print(n_total_events)
fig, ax, _ = hist.plot1d(all_mass_hists)
plt.show()
```

```
1993800
```

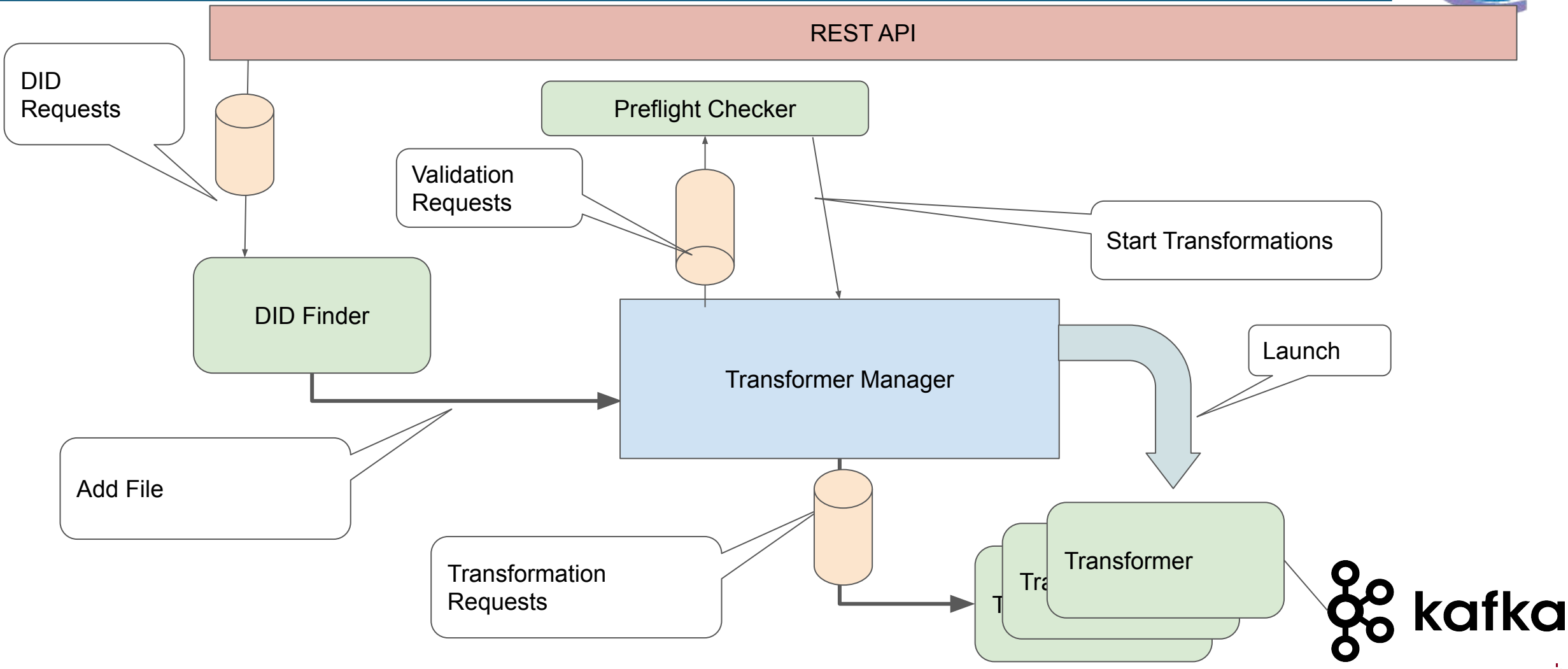


Current work

- Transformers now use compiled C++ to speed up read/write
  - Implementing via SQL-like declarative selection statements to permit in-place filtering for requests
- Service scales well with multiple large concurrent requests
- Sharing swapable components: transformers, message brokers (Kafka and Redis), …
  - Coordinating development of transformers with iDDS/ESS production use cases
- Adding multiple transform input formats
  - Available for xAOD, DAOD, flat TTree formats
- Stability testing: Ensure output robust vs randomly killing transformer nodes
- Ramping up to v1.0

Backup slides

ServiceX architecture



ServiceX running on SSL: RIVER 2

Services

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
release-name-0-external	NodePort	10.104.248.66	<none>	19092:31090/TCP
release-name-1-external	NodePort	10.102.224.207	<none>	19092:31091/TCP
release-name-kafka	ClusterIP	10.110.20.190	<none>	9092/TCP
release-name-kafka-exporter	ClusterIP	10.97.197.166	<none>	9308/TCP
release-name-kafka-headless	ClusterIP	None	<none>	9092/TCP
release-name-minio	ClusterIP	10.97.66.31	<none>	9000/TCP
release-name-postgresql	ClusterIP	10.96.46.224	<none>	5432/TCP
release-name-postgresql-headless	ClusterIP	None	<none>	5432/TCP
release-name-rabbitmq	ClusterIP	10.106.225.45	<none>	4369/TCP,5672/TCP,25672/TCP,15672/TCP,9090/TCP
release-name-rabbitmq-headless	ClusterIP	None	<none>	4369/TCP,5672/TCP,25672/TCP,15672/TCP
release-name-servicex-app	NodePort	10.102.57.95	<none>	8000:31973/TCP
release-name-zookeeper	ClusterIP	10.108.18.68	<none>	2181/TCP
release-name-zookeeper-headless	ClusterIP	None	<none>	2181/TCP,3888/TCP,2888/TCP

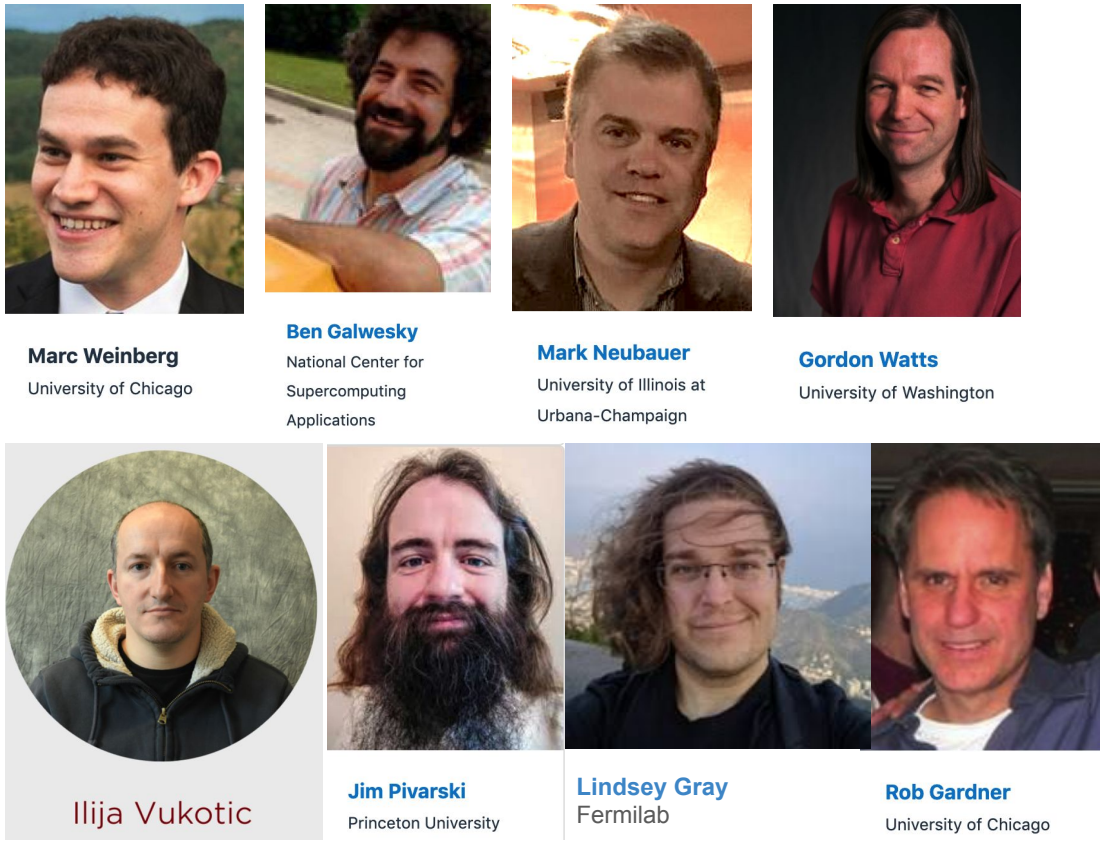
- Currently deployed on Kubernetes cluster on SSL RIVER 2
- Performance testing: Running with hundreds of transformers on cluster
- Has Prometheus monitoring dashboard

Workloads

```
C:\Users\ivukotic>kubectl get pods -n servicex
NAME                                                  READY   STATUS      RESTARTS   AGE
release-name-did-finder-b5b6bfb48-d2bnq               1/1     Running     2          14m
release-name-kafka-0                                  1/1     Running     3          17d
release-name-kafka-1                                  1/1     Running     1          17d
release-name-kafka-exporter-76cf9c7c69-nbzcq          1/1     Running     2          17d
release-name-minio-5c54b8648-b92n2                    1/1     Running     0          14m
release-name-postgresql-0                             1/1     Running     0          14m
release-name-preflight-cb454d56-f4tmv                 0/1     Error       7          14m
release-name-rabbitmq-0                               2/2     Running     0          14m
release-name-servicex-app-9b58465fc-q8xkm             1/1     Running     2          14m
release-name-test-topic-consumer                      0/1     Completed   0          2d23h
release-name-test-topic-consumer-v2                   0/1     Completed   0          2d23h
release-name-test-topic-consumer-v3                   0/1     Completed   0          2d23h
release-name-test-topic-create-consume-produce        0/1     Completed   0          17d
release-name-testclient                               0/1     Error       0          26h
release-name-zookeeper-0                              1/1     Running     1          17d
release-name-zookeeper-1                              1/1     Running     2          17d
release-name-zookeeper-2                              1/1     Running     1          17d
transformer-210b2510-0e16-4006-9055-d9addb51fcaa-xprdk  1/1     Running     0          13m
```

IRIS-HEP R&D efforts in DOMA & Analysis Systems

- DOMA/AS groups interested in ***R&D*** for data delivery for analysis of columnar and other data formats
- Supports multiple input types (xAOD, flat ntuples, …) and common data mgt (Rucio, XCache)
- Utilize industry standard tools (GKE, on-prem Kubernetes, Helm, Kafka, Redis, Spark, …)
- Reproducible, portable deployments



Loop-less array programming

Can do all kinds of stuff with optimized linear algebra computations

- Multidimensional slices of data
- Element-wise operations (e.g. `muons_pz = muons_pt * sinh(muons_eta)`)
- Broadcasting (e.g. `muon_phi - 2 * pi`)
- Event masking, indexing, array reduction, etc.

But we don't have a simple rectangular arrays

- Nested variable-size data structures everywhere in HEP
- Jagged arrays handle this with two 1D arrays:
  - First array contains long list of values, one per object
  - Second array contains breaks that give event boundaries

Loop-less array programming

- But this is shown to the user as a list containing lists of various lengths:

```
In [4]: import uproot
        f = uproot.open("HZZ-objects.root")
        t = f["events"]

In [5]: a = t.array("muoniso")      # muon isolation variable; multiple per event
        a
```

Event 1 Event 2 ...

```
Out[5]: <JaggedArray [[4.2001534 2.1510613] [2.1880474] [1.4128217 3.3835042] ... [3.7629452], [0.550810
        7], [0.]] at 7b229f313240>
```

The implementation is a façade: these are not millions of list objects in memory but two arrays with methods to make them *behave like* nested lists.

```
In [6]: a.offsets

Out[6]: array([    0,     2,     3, ..., 3823, 3824, 3825])

In [7]: a.content

Out[7]: array([4.2001534, 2.1510613, 2.1880474, ..., 3.7629452, 0.5508107, 0.        ], dtype=float32)
```

Open source technologies used