


A new backend for Distributed RDataFrame using AWS Lambda

25 Feb 2021

Jacek Kuśnierz
Maciej Malawski

AGH University of Science and Technology
Kraków, Poland 

Presentation plan

1. Inspiration for the project
2. AWS Lambda, Spark and PyRDF
3. Project Structure Overview
4. Demo
5. Technical Highlights

How the idea was born

1. I already worked at CERN in TOTEM experiment, and there I met ROOT
2. Inspirations for the project:
 - a. Spark ROOT
 - b. PyRDF
 - c. Swan Totem Helix Nebula
 - d. PyWREN



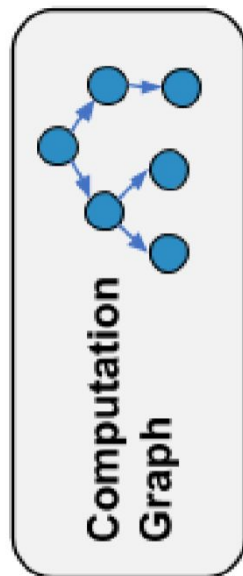
Why we did that

1. An obvious objective: bachelor thesis.
2. Less obvious objective:
 - a. we have tool for distributed computing - PyRDF
 - b. so far the investigation went into managed infrastructure
 - c. we want to try unmanaged distributed backend
3. I work @CS job, so it was easier to use tech known to me (AWS Lambda).

PyRDF

```
PyRDF.use(backend)
d = RDataFrame(dataset)
f = d.Define(...)
    .Define(...)
    .Filter(...)

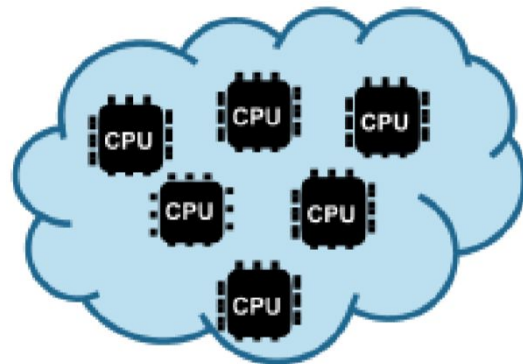
h1 = f.Histo1D(...)
h2 = f.Histo2D(...)
```



Local



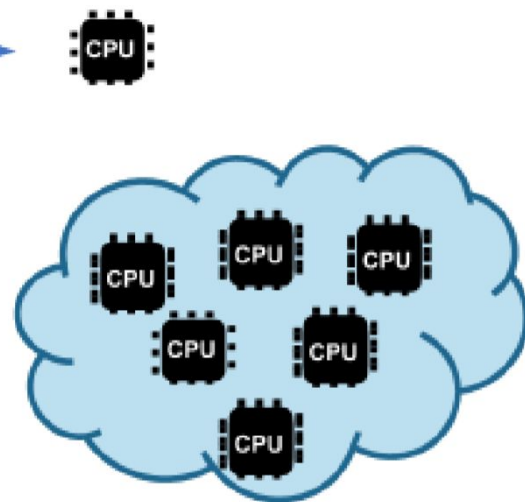
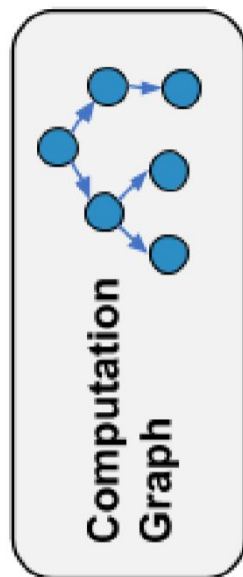
...



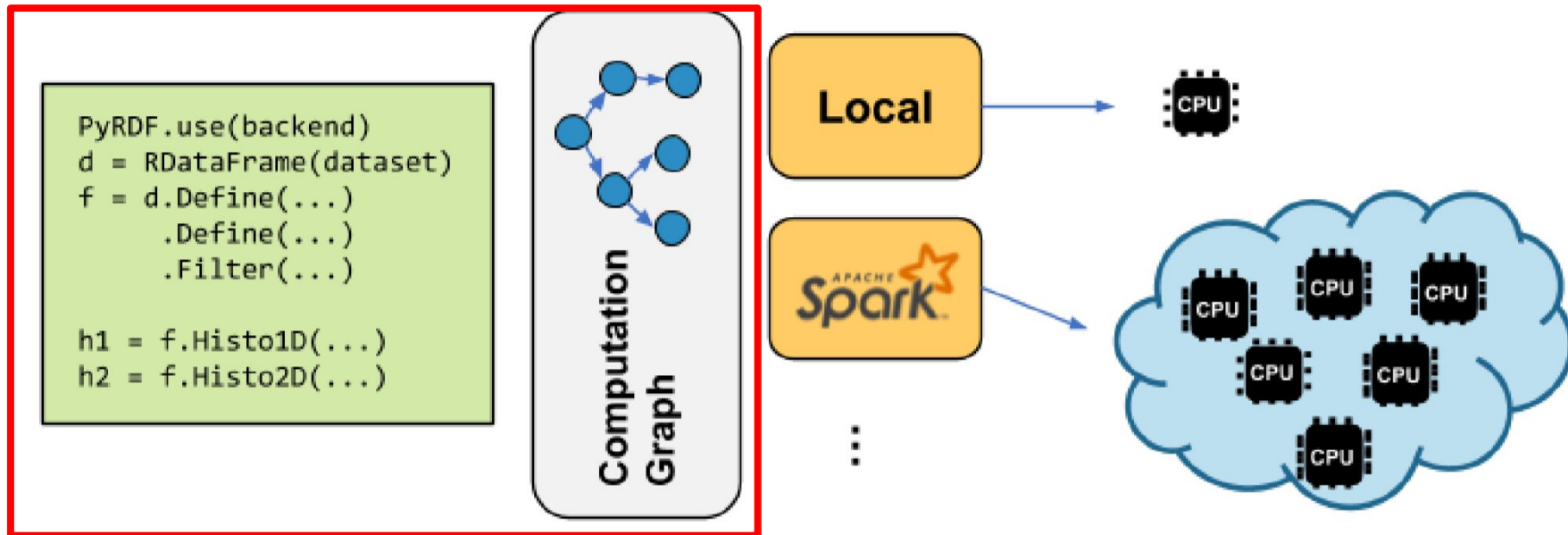
PyRDF

```
PyRDF.use(backend)
d = RDataFrame(dataset)
f = d.Define(...)
    .Define(...)
    .Filter(...)

h1 = f.Histo1D(...)
h2 = f.Histo2D(...)
```

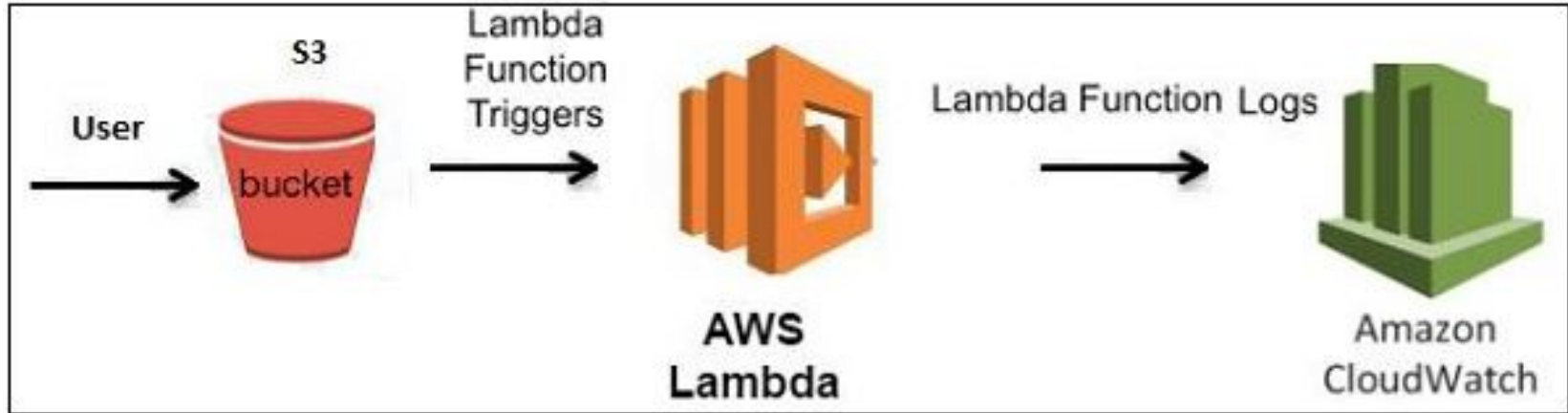


Blackboxing PyRDF



I took all things beside backend for granted and froze them as 'dependencies'.

AWS Lambda



1. Serverless
2. Unmanaged
3. Cloud Native
4. Good integrations
5. Severe limitations

AWS Lambda - limitations

1. Function memory: 128 MB to 10240 MB
2. Timeout: 900 seconds (15 minutes)
3. Concurrency - default 500-3000, upgradable on request
4. Invocation payload: 256 KB async
5. Deployment package: 50 MB size, max 250 unzipped

PyWren

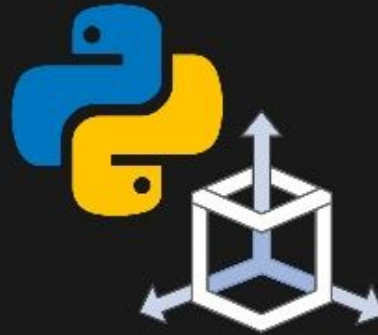
PyWren—“Microservices and Teraflops”

Run **existing Python code** at massive scale via AWS Lambda in parallel functions

Serializes local Python code and sends it to AWS Lambda for **massive parallel execution** with Amazon S3 as the intermediary

Receive **results back to your local machine** or other data stores for further analysis

pywren.io



AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws
amazon



Berkeley
UNIVERSITY OF CALIFORNIA

AWS Lambda vs Apache Spark

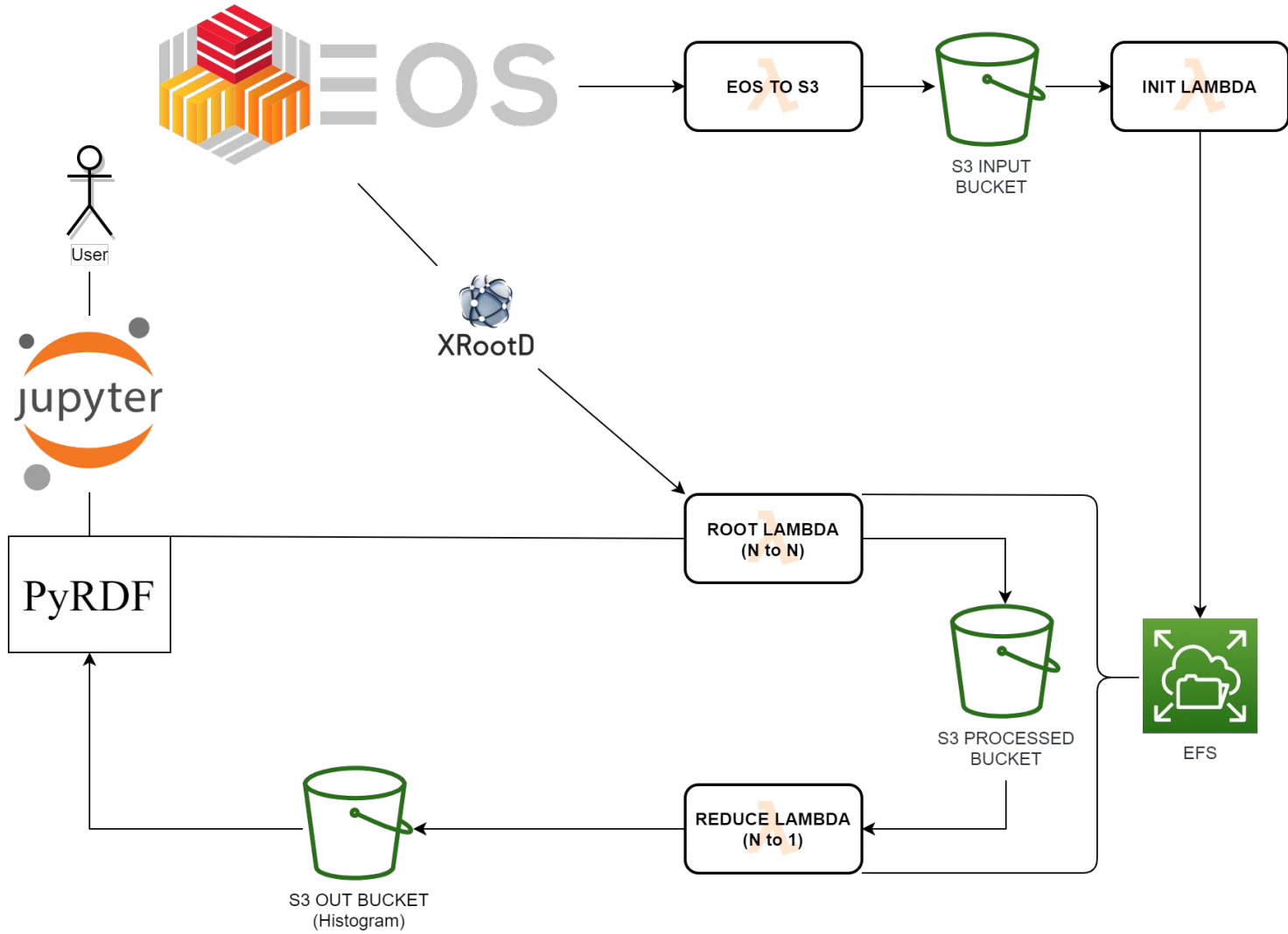
Lambda:

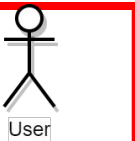
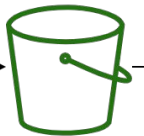
1. Unmanaged
2. Default environments are very small, very small amount of software by default
3. Not much software installation allowed
4. No good libraries for handling it (PyWREN is a working example, but that's not much)

Spark:

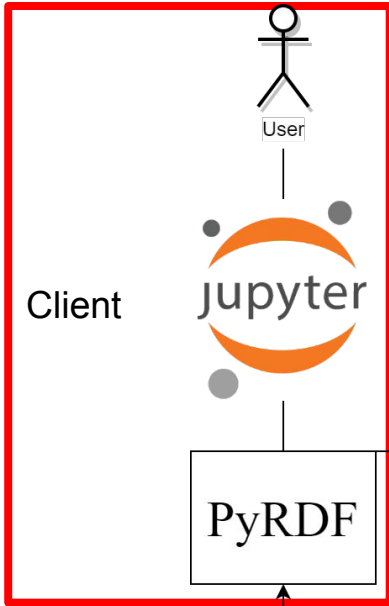
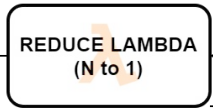
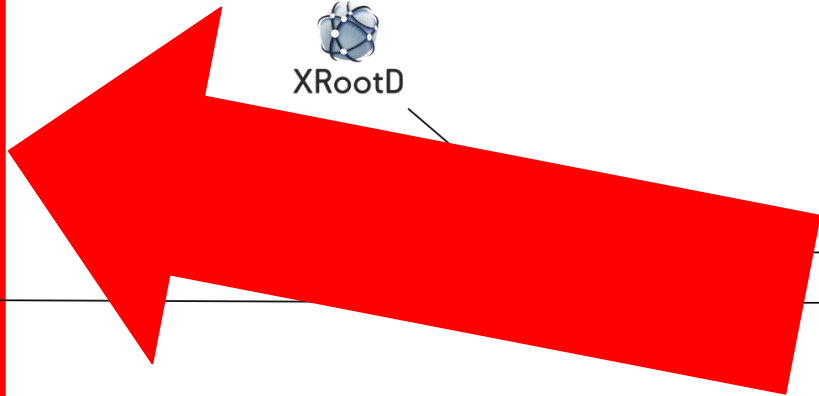
1. Managed
2. Environments are anything you can imagine
3. Allows plugging in lots of code
4. Good working libraries
5. Software can be written specifically for it, with very simple interfacing.

Structure Overview





XRootD



S3 OUT BUCKET
(Histogram)

S3 PROCESSED
BUCKET

EFS

REDUCE LAMBDA
(N to 1)

INIT LAMBDA

EOS TO S3

S3 INPUT
BUCKET

User

jupyter

PyRDF

Client

XRootD

S3 PROCESSED
BUCKET

EFS

REDUCE LAMBDA
(N to 1)

S3 OUT BUCKET
(Histogram)

INIT LAMBDA

EOS TO S3

S3 INPUT
BUCKET

User

jupyter

PyRDF

Client

XRootD

Client side

In order to make AWS Lambda to work with PyRDF, I needed to do:

1. packing Python ROOT objects.
2. Check if infrastructure is online.
3. Calling Lambdas with packed (pickled) objects as strings.
4. Receive and reduce the results (temporary solution).

Client side - packing functions

```
def ProcessAndMerge(self, mapper, reducer):  
    """Performs map-reduce using AWS Lambda...."""  
  
    ranges = self.build_ranges()  
  
    def encode_object(object_to_encode) -> str:  
        return str(base64.b64encode(pickle.dumps(object_to_encode)))  
  
    # Make mapper and reducer transferable  
    pickled_mapper = encode_object(mapper)  
    pickled_reducer = encode_object(reducer)
```


Client side - init check

I wanted to check that the infrastructure was ready, so I put a simple check.

```
# Check for existence of infrastructure
s3_output_bucket = ssm_client.get_parameter(Name='output_bucket')['Parameter']['Value']
if not s3_output_bucket:
    self.logger.info('AWS backend not initialized!')
    return False
```

Client side - invoking workers

```
def invoke_root_lambda(client, root_range, script):
    payload = json.dumps({
        'range': encode_object(root_range),
        'script': script,
        'start': str(root_range.start),
        'end': str(root_range.end)
    })

    return client.invoke(
        FunctionName='root_lambda',
        InvocationType='Event',
        Payload=bytes(payload, encoding='utf8')
    )

# Invoke workers with ranges and mapper
call_results = []
for cluster_range in ranges:
    call_result = invoke_root_lambda(lambda_client, cluster_range, pickled_mapper)
    call_results.append(call_result)
```

Client side - sample reducer call

```
# Get names of output files, download and reduce them
filenames = s3_client.list_objects_v2(Bucket=processing_bucket)['Contents']
accumulator = pickle.loads(s3_download_file(filenames[0]))

for filename in filenames[1:]:
    file = pickle.loads(s3_download_file(filename))
    accumulator = reducer(accumulator, file)

# Clean up intermediate objects after we're done
s3_resource.Bucket(processing_bucket).objects.all().delete()
return accumulator
```


Using the System - demo

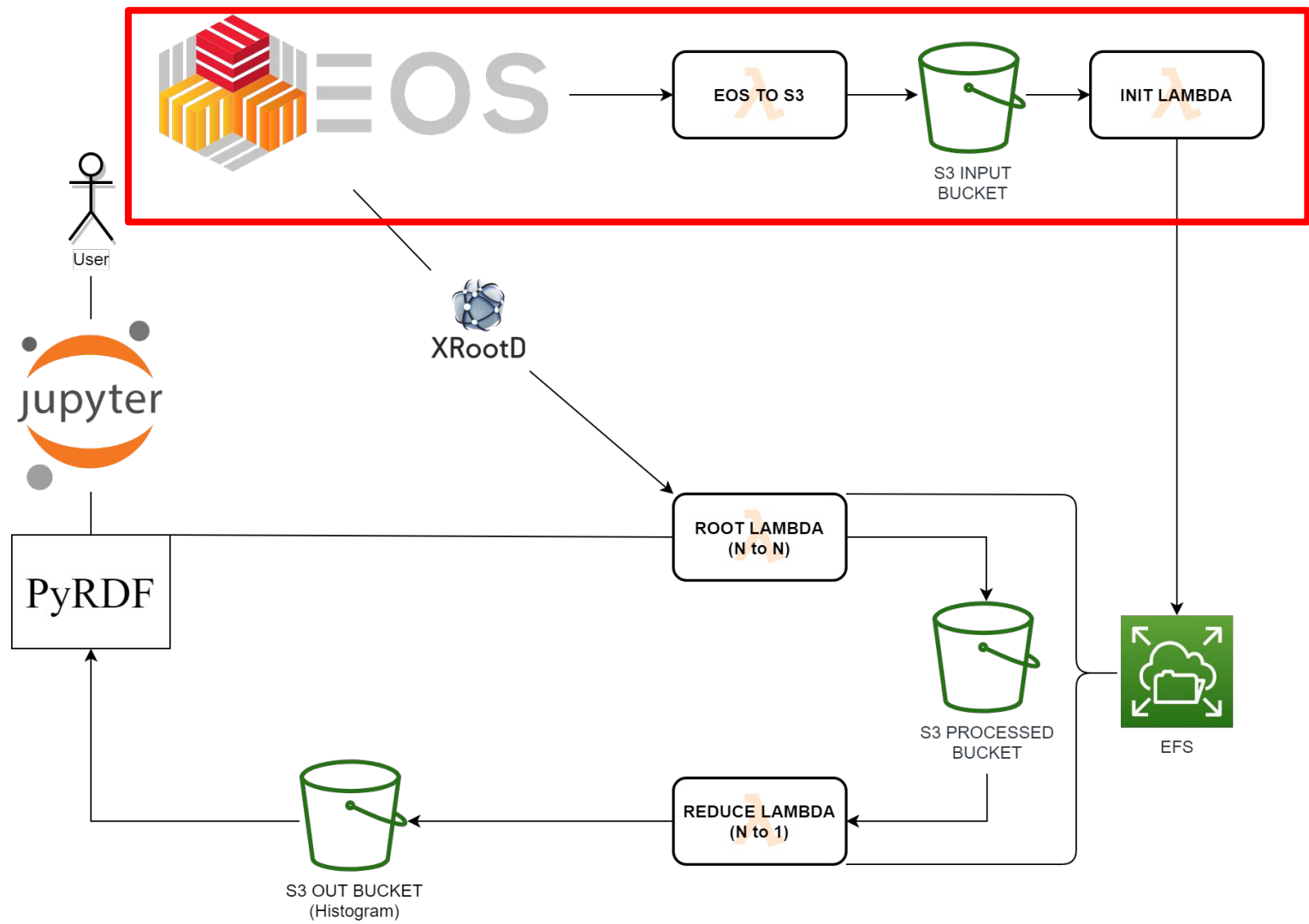
Technical Highlights

First: ROOT/docs issues dictating architecture

1. ROOT is big, very big - we have only 50MB deployment package for something about 200MB
2. I could use CVMFS, but had three issues:
 - a. Not sure if the compiled versions would work with AWS Linux, where Lambda runs
 - b. I have no idea how to authorize to CVMFS, no clear documentation
 - c. Even if I connected to CVMFS, I still have no proof that dependencies from Centos would work on AWS Linux
3. I still have no idea how to authorize to CERN from pure serverless solution

Solution: use EFS

1. No Docker implementation existed at the beginning of the project
2. EFS is just like NFS known from on-premises solutions
3. It allows to store arbitrarily big software
4. It can be attached to Lambda
5. I can then put my own ROOT compiled on AWS Linux Docker image

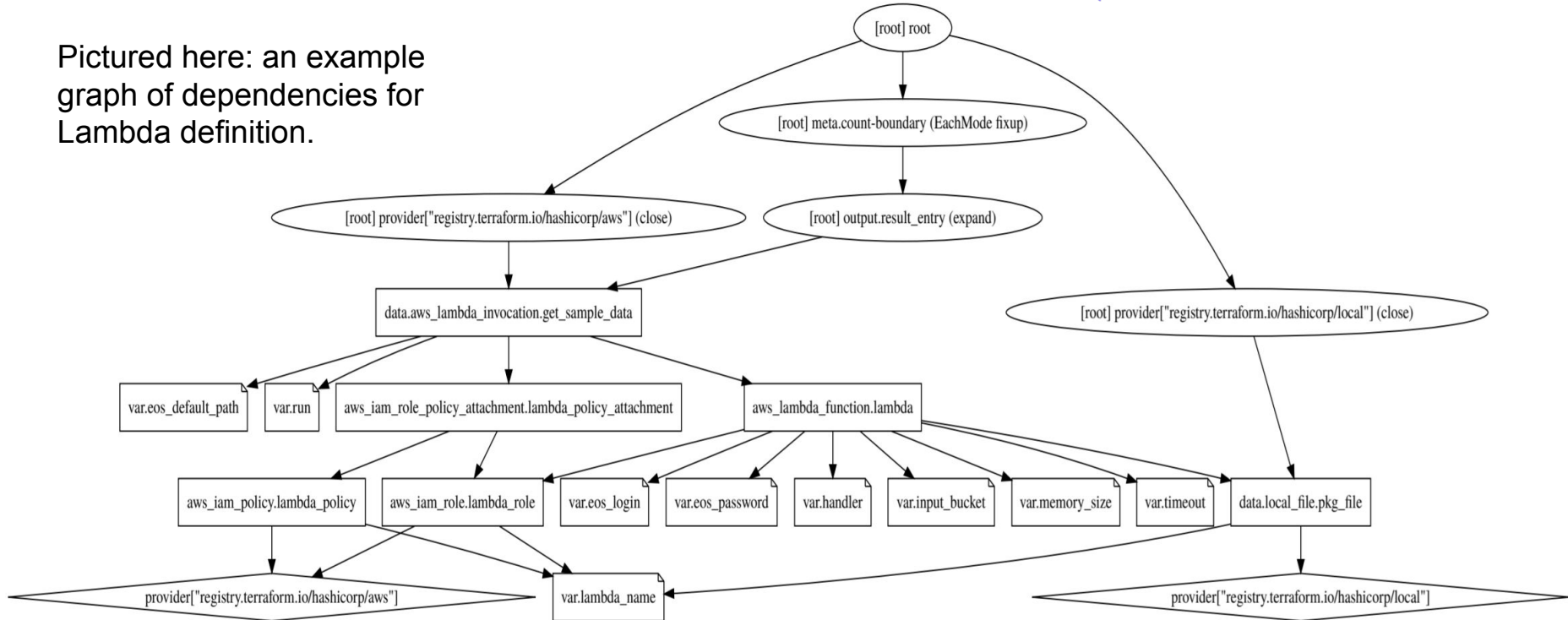


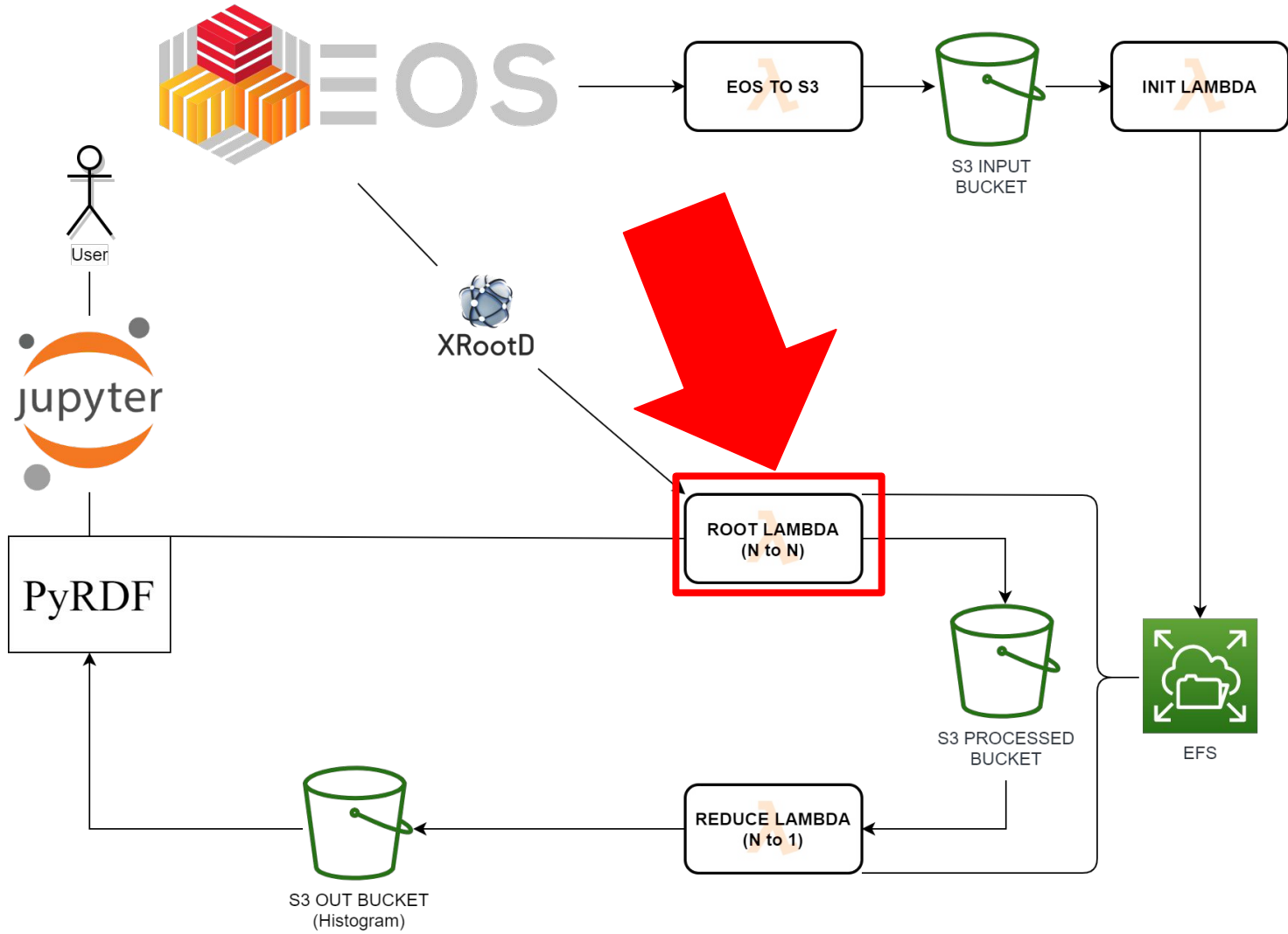
Terraform - Make for infrastructure



HashiCorp
Terraform

Pictured here: an example graph of dependencies for Lambda definition.





ROOT Lambda - loading the functions

```
range = base64.b64decode(event['range'][2:-1])
mapper = base64.b64decode(event['script'][2:-1])

glue = f"""
import pickle
mapper=pickle.loads({mapper})
range=pickle.loads({range})
hist=mapper(range)
pickle.dump(hist, open('/tmp/out.pickle','wb'))
"""

script_file = open('/tmp/to_execute.py', "w")
script_file.write(glue)
script_file.close()
```

ROOT Lambda



Lambda 1

Access Files

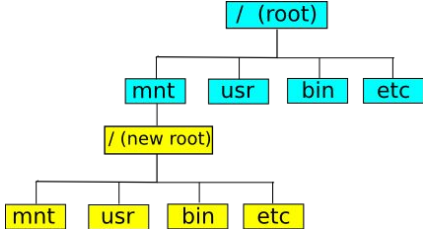


EFS

Access Files



Lambda 2



ROOT Lambda - running the analysis

```
result = os.system('''
    export PATH=/mnt/cern_root/chroot/usr/local/sbin:/mnt/cern_root/chroot/usr/local/bin:/mnt/cern_ro
    export LD_LIBRARY_PATH=/mnt/cern_root/chroot/usr/lib64:/mnt/cern_root/chroot/usr/lib:/usr/lib64:/
    export CPATH=/mnt/cern_root/chroot/usr/include:$CPATH && \
    export PYTHONPATH=/mnt/cern_root/root_install/PyRDF:/mnt/cern_root/root_install:$PYTHONPATH && \
    export roothome=/mnt/cern_root/root_install && \
    cd /mnt/cern_root/root_install/PyRDF && \
    . ${roothome}/bin/thisroot.sh && \
    /mnt/cern_root/chroot/usr/bin/python3.7 /tmp/to_execute.py
''')
s3.upload_file(f'/tmp/out.pickle', bucket, f'partial_{start}_{end}.pickle')
```

Results

- Proof of Concept works
- Looks promising despite 17s init time per lambda
- Shared file system on EFS - bottleneck
- High cost - new Docker Lambda implementation should solve it

Summary and future work

- set up ROOT C++ JIT environment on AWS

Summary and future work

- set up ROOT C++ JIT environment on AWS
- integrated solution for PyRDF

Summary and future work

- set up ROOT C++ JIT environment on AWS
- integrated solution for PyRDF
- under certain conditions obtain not worse results than the current solution

Summary and future work

- set up ROOT C++ JIT environment on AWS
- integrated solution for PyRDF
- under certain conditions obtain not worse results than the current solution
- run arbitrary software with own environment on Serverless Platform

Summary and future work

- set up ROOT C++ JIT environment on AWS
- integrated solution for PyRDF
- under certain conditions obtain not worse results than the current solution
- run arbitrary software with own environment on Serverless Platform
- next step: move to Docker Lambda and test further

Summary and future work

- set up ROOT C++ JIT environment on AWS
- integrated solution for PyRDF
- under certain conditions obtain not worse results than the current solution
- run arbitrary software with own environment on Serverless Platform
- next step: move to Docker Lambda and test further

Thank you. Questions?

Special thanks to:

- Maciej Malawski, Associate Professor of AGH
- Supervisor of the Project
- Vincenzo Eduardo Padulano - PyRDF expert

Code available at: <https://github.com/CloudPyRDF>

contact: kusnierz@protonmail.com

This work was supported by the Polish Ministry of Science and Higher Education, grant DIR/WK/2018/13

The project was realised as part of Bachelor Thesis at AGH University of Science and Technology in Kraków.



Backup Slides

EOS Lambda

