ATLAS Data Analysis with Distributed Cloud Processing

Thomas Smith | 1700 words
github.com/do22555/Cloud-Comp-Proj

This report details the re-engineering of an existing, unscalable ATLAS Data Analysis script running in a Jupyter environment, to be run in a scalable, distributed, container-based Docker environment. The goal of the assignment was to construct a proof-of-concept application which can be scaled from a PC to a large cluster using cloud technology. This solution automatically installs the ATLAS Open data environment inside the containers, downloads the required data, and makes use of distributed analysis across multiple virtual nodes using many workers with RabbitMQ acting as a distributed task scheduler. The resulting data is aggregated to produce graphs which can be downloaded locally. The system meets all three project aims and it can demonstrate performance increases with more parallel workers.

1. Introduction

High-energy physics analyses often rely on large datasets and complex processing pipelines that must be scaled beyond a single machine. The ATLAS experiment creates Petabytes per year, after the necessary algorithms are run to reduce unnecessary data. This is stored in a federated grid-based computing system and requires 174 worldwide institutions to analyse the data generated. [1] The original ATLAS four-lepton analysis notebook is designed for educational use and therefore cannot do distributed computing natively.

The solution uses Docker Compose to provision an automated ATLAS Open Data environment, RabbitMQ for distributed task scheduling, and multiple worker containers to process ROOT files concurrently. The design aims to demonstrate practical scalability, fault tolerance, and modularity while preserving the underlying physics logic of the original notebook within practical limitations. The resulting framework provides a portable and extensible proof-of-concept for large-scale scientific data analysis on cloud infrastructure.

2. Physics Analysis

For the purposes of this assignment, the focus on the cloud-native execution of an analysis.

Several practical constraints of containerised execution meant the notebook's logic could not be transplanted unchanged. ATLAS OpenMagic (AOM) dependency on coffea pulled in incompatible versions of uproot, awkward, and boost-histogram, causing failures when using Hist, .to_dict(), or ATLAS-specific vector operations. Consequently, the physics was changed in some places to make use of compatible modules. Specific changes have been omitted for the sake of brevity.

The cloud application follows the notebook logic as closely as possible, and in most places follows it verbatim. Exceptions occur where AOM causes conflicts. The analysis was partially rewritten to reply solely on: numpy histograms, awkward arrays for vectorisation, vector for four-momentum constructions and simple .npz serialisation for inter-container communication. This affected the results of the experiment slightly (see **Sect. 5**).

3. System Architecture

Each container performs a single well-defined role, while RabbitMQ provides reliable message passing. ***(RabbitMQ also appears to be extremely robust/fault-tolerant; during one test, a worker crashed but I was pleased to see the rest of the workers take on the extra load and complete an identical analysis).*** The workers can be scaled indefinitely for concurrent processing, see **Fig. 1**.

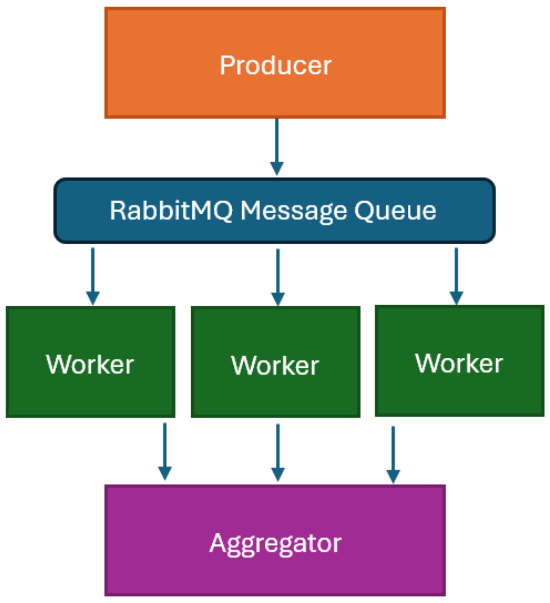


Fig 1: An overview of the cloud architecture.

Docker Compose orchestrates services with minimal user intervention; Compose manages service creation, dependency, networking, and volume mounting. [2] The entire system is started through a sequence of simple commands:

```
docker compose up -d rabbitmq
docker compose up -d --scale worker=4
docker compose run --rm producer
docker compose run --rm aggregator
```

[ReadMe.md],

which are then managed by a YAML file [3]

.

All three services rely on a master script "analysis_common" for commonly-used functions to keep the containers light and limit the number of imports required. Aggregator imports several variables; worker imports the method process_file which contains logic to open ROOT file, apply notebook-style final-analysis logic, and return a dict with a pure-numpy histogram; producer imports build_samples to build the AOM dataset dictionary like in the notebook. "analysis_common" is the longest script at 382 lines long, and is by far the most similar to the original notebook.

3.1. Producer

The producer container is responsible for task discovery and distribution. It uses AOM to retrieve the list of dataset URLs, then publishes one message per ROOT file into a durable RabbitMQ queue. The producer exits after queueing tasks, keeping the architecture lightweight. The producer could be considered a bottleneck given it is limited by serial downloading. ATLAS already has systems for handling huge amounts of data, making this aspect outside the scope of the project.

```
for url in file_list:
      msg = {"file_url": url, "sample":
                                sample_name}
      ch.basic_publish(
          exchange="",
          routing_key=QUEUE,
          body=json.dumps(msg),
          properties=pika.BasicProperties(de
                                livery_mode=2),
      )
      print(f"[producer] Queued {url}")
```

[Line 27, producer.py].

3.2. Workers

Workers are microservices, executed in parallel. They pull one file, process it with uproot + awkward, and store the result. Each worker performs this loop:

1. Pull a task from RabbitMQ round-robin allocation.
2. Download and process a single ROOT file using uproot + awkward-array.
3. Extract 4-lepton momenta and compute invariant masses.
4. Write a small .npz histogram chunk into the cloud file /data/output volume.
5. Acknowledge the task to RabbitMQ.

Each worker writes one file per input ROOT file (as opposed to a shared histogram) as this was easiest to parallelise: workers will not attempt to write to the same file at once and reprocessing a file just overwrites it. Workers acknowledge tasks only after saving output, meaning crashed workers return their tasks to the queue automatically.

3.3. Aggregator

The aggregator performs the final reduction phase: loading all .npz chunks, summing histograms, computing errors, and producing and storing the plots.

Due to the relatively low-computing power this requires, it is not a huge bottleneck compared to the processing done by the workers. Real-time updating is extremely complicated and outside the scope of the project.

The graph generation is the same as in the notebook. The styles, colours, scale, content etc. is copied and pasted verbatim.

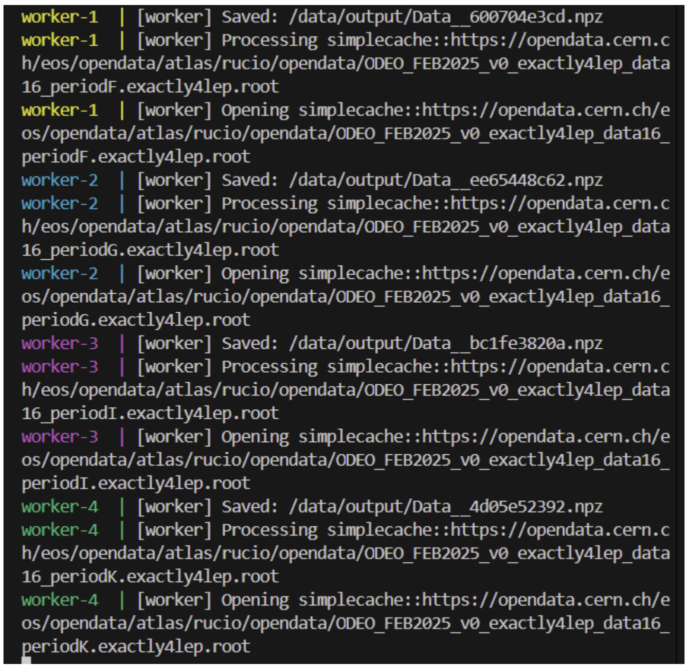
4. Scalability Analysis

The way the workers are set up means that they enjoy a theoretically limitless horizontal scalability. The performance of the program is limited only by:

- The external dataset download rate, as mentioned in **Sect. 3.1**. This can be considered outside the scope of the task, nevertheless file downloads become the limiting factor beyond "dozens" of files (for which we are working at current scale).
- The single queue, which means large numbers of workers could be bottlenecked by RabbitMQ's single-queue throughput. Clustered RabbitMQ, cloud message brokers or several more exotic solutions could circumvent this.
- Aggregation working as a single thread (previously mentioned in **Sect. 3.3**) will throttle

at much higher workloads, e.g. if thousands of histograms exist.

One could also argue that the choice of python, an interpreted language, is also suboptimal when compared to compiled solutions such as C, but this is also outside this task's scope.



```
worker-1  | [worker] Saved: /data/output/Data__600704e3cd.npz
worker-1  | [worker] Processing simplecache::https://opendata.cern.c
h/eos/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data
16_periodF.exactly4lep.root
worker-1  | [worker] Opening simplecache::https://opendata.cern.ch/e
os/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data16_
periodF.exactly4lep.root
worker-2  | [worker] Saved: /data/output/Data__ee65448c62.npz
worker-2  | [worker] Processing simplecache::https://opendata.cern.c
h/eos/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data
16_periodG.exactly4lep.root
worker-2  | [worker] Opening simplecache::https://opendata.cern.ch/e
os/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data16_
periodG.exactly4lep.root
worker-3  | [worker] Saved: /data/output/Data__bc1fe3820a.npz
worker-3  | [worker] Processing simplecache::https://opendata.cern.c
h/eos/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data
16_periodI.exactly4lep.root
worker-3  | [worker] Opening simplecache::https://opendata.cern.ch/e
os/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data16_
periodI.exactly4lep.root
worker-4  | [worker] Saved: /data/output/Data__4d05e52392.npz
worker-4  | [worker] Processing simplecache::https://opendata.cern.c
h/eos/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data
16_periodK.exactly4lep.root
worker-4  | [worker] Opening simplecache::https://opendata.cern.ch/e
os/opendata/atlas/rucio/opendata/ODEO_FEB2025_v0_exactly4lep_data16_
periodK.exactly4lep.root
```

Fig 2: Log demonstrating several workers in action simultaneously after running the command: `Docker compose logs -f worker`

5. Result

The physical analysis has been partially successful. Due to the efforts made to circumvent AOM package requirements, in the places where the logic was rewritten, the physics does appear to be different. ***(I am confident this is due to my poor physics rather than lack of understanding of cloud computing).*** Referencing **Fig. 3**, a near-perfect analysis of the final stack is shown with minor discrepancies. Another issue arises with the architecture of the program: in the subsequent figure, referencing example 2, the graph mirrors the stack, rather than the mid-point analysis shown in the notebook due to the way the data is collated. To achieve a more accurate replication, the architecture of the code would have to change – this could require an aggregation stage at every point where the notebook displays a graph. This is overly complex and does not demonstrate further understanding of cloud computing.

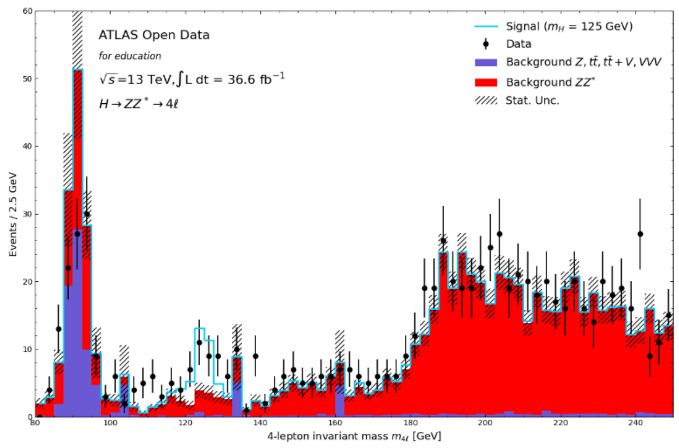


Fig 3: The final stack is near-identical to the Notebook's plot.

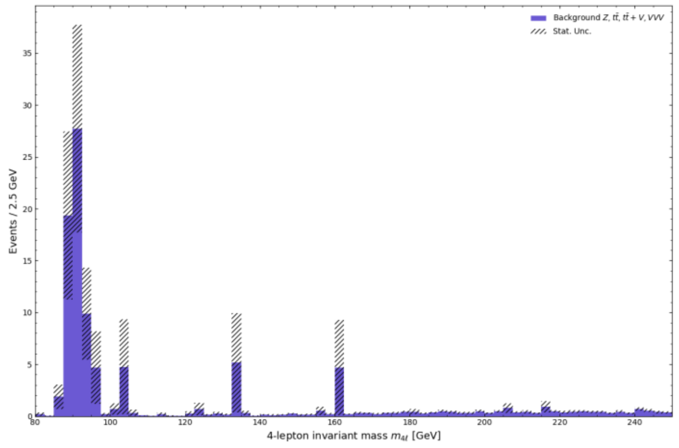


Fig 4: In example 2, the graph produced, using the same data, is different, due to the system's structure.

Parallelising the workers yielded a modest performance increase which showed a positive correlation between worker count and performance. With 40 tasks queued in total, the theoretical limit for workers is this number, however the bottlenecks in the system were found much earlier. These scripts were run on a Windows 11 machine running WSL, with a 2.1 GHz 12th Generation Intel Core i7 1260p mobile laptop CPU with 4 hyperthreaded performance-cores and 8 efficiency-cores (12 cores, 16 threads in total) with 8 GB RAM.

Worker Count	Worker runtime excl. overheads
1	10'54" (654 s)
2	8'53" (533 s)
4	5'31" (331 s)
8	5'28" (328 s)
16	5'31" (331 s)

6. Summary & Discussion

This project successfully re-engineered the original HZZ notebook into a distributed, containerised analysis pipeline using Docker Compose, RabbitMQ, and multiple worker containers. The results show that the final stacked invariant-mass distribution reproduces the notebook's output to good accuracy (**Fig. 3**) despite necessary simplifications imposed by package-compatibility constraints. I am disappointed with the inexact replication of the intermediate graphs. Achieving an exact replication would require a version-pinned environment identical to the ATLAS Jupyter notebook and aggregation stages matching each intermediate notebook plot**.** I would argue that neither of these features would demonstrate a better understanding of cloud computing.

A natural question is why the system uses Docker Compose rather than Docker Swarm or Kubernetes, which are designed for large-scale container orchestration. At the scale of the assignment, these technologies provide little benefit towards demonstration of understanding, but significant overheads in runtime and developmental complexity; Compose is simple to deploy and avoids the cluster-provisioning overhead of Kubernetes. Neither provide better reliability/fault-tolerance as these are already demonstrated through RabbitMQ's message broker (**Sect. 3**), but instead add features which are useful for large clusters. Admittedly, Kubernetes's batch-processing scheduler Kubernetes Jobs would be a valid proposal for further scaling. Other proposals for scaling include a distributed message broker (e.g. RabbitMQ cluster or cloud MQ) to negate the bottleneck in **Sect. 4** and moving towards object storage, beyond a shared volume. Another feature which should have been added was the automatic downloading of files and graphs to the local system; in theory, this can be done by modifying the YAML file.

Overall, the project shows that large-scale analyses can be distributed and that message-driven architectures such as RabbitMQ provide a robust foundation for scalable scientific computing. The result is portable, fault-tolerant, and capable of operating on any hardware, meaning it generalises directly to larger cloud or HPC deployments with minimal modification. The script shows a noticeable speed up with more workers.

References

[1] Particle Detectives: Physicists and Data - NYAS

[2] How Compose works | Docker Docs

[3] Docker Compose | Docker Docs

Notebook source:
notebooks-collection-opendata/13-TeV-examples/uproot_python/HZZAnalysis.ipynb at master · atlas-outreach-data-tools/notebooks-collection-opendata

Distributed coding project location:
do22555/Cloud-Comp-Proj
PLEASE READ README!