ATLAS Data Analysis with Distributed Cloud Processing

Thomas Smith
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: automatic configuration, distributed computing and scalability.

1. Introduction

High-energy physics analyses often rely on large datasets and complex processing pipelines that must be scaled beyond a single machine. The original ATLAS four-lepton analysis notebook cannot do this.

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 could have 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**.

Compose orchestrates services with minimal user intervention; compose manages service creation, dependency order, networking, and volume mounting. 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.

3.1. 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].

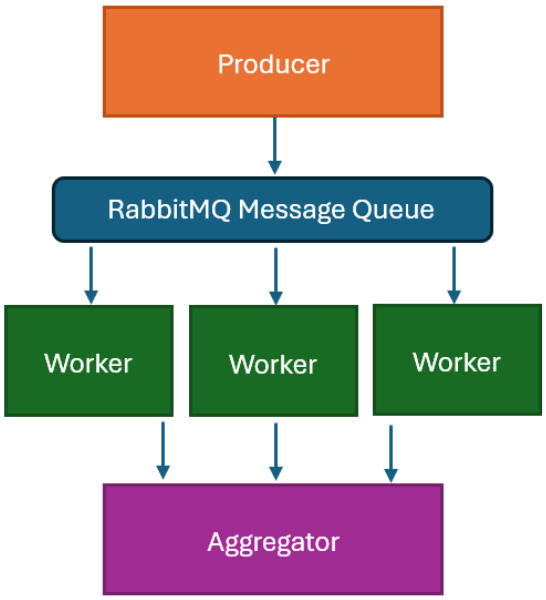

Fig 1: An overview of the cloud architecture.

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.

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, content etc. is copied and pasted verbatim.

4. Scalability Analysis

Scalability limited only by:

- external dataset download rate This can be considered outside the scope of the task.
- The single queue means large numbers of workers could be bottlenecked by RabbitMQ's single-queue throughput [AN ALTERNATIVE SOLUTION?]
- aggregator's single-process reduction [WHAT?]
- **Worker scaling is linear** for embarrassingly parallel workloads.
- RabbitMQ handles millions of queued tasks at cloud scale.
- Because each file is processed independently, you avoid:
    - shared state
    - race conditions
    - lock contention

Component	Bottleneck Explanation
Input data (OpenData HTTP server)	File downloads become the limiting factor beyond dozens/hundreds of workers.
RabbitMQ	Single queue becomes a choke point at very large scale → need clustered RabbitMQ or cloud message brokers.
Output Volume	NPZ files written to a shared Docker volume work fine locally, but in real clusters you'd use object storage (S3, MinIO, GCS).
Aggregator	Single-threaded → becomes too slow if thousands of histograms exist. Needs map-reduce/fan-in structure at very large scale.

5. Result

Example timings, say 1 worker, 2 workers, 4 workers, 8 workers.

Screenshot or embedded final Higgs peak plot
Confirmation that histogram matches notebook shape
Summarise what was achieved and how your system could be adapted to real LHC workloads.

Fig 2: Log demonstrating several workers in action simultaneously.

6. Discussion

Reliability and fault tolerance, discuss:

- RabbitMQ ensures tasks are not lost.
- Workers acknowledge tasks only after saving output.
- Crashed workers automatically return tasks to queue.
- Stateless workers → can be restarted at any time.

To scale further You can propose:

- Object storage instead of shared volume
- Distributed message broker (RabbitMQ cluster or cloud MQ)
- Batch-processing scheduler (HTCondor, Slurm, Kubernetes Jobs)
- Multi-level aggregation (tree reduction)

This demonstrates deep understanding.