Big Data Analysis using Cloud Technology

March 21, 2025

Abstract

This project demonstrates the use of cloud technologies to scale the ATLAS HZZ analysis workflow, which was originally designed for sequential execution on a single machine. By leveraging containerisation with Docker, task distribution with RabbitMQ, and automated container management with Kubernetes, the analysis is parallelized and distributed across multiple compute resources. This approach significantly enhances scalability and efficiency, making it capable of processing large volumes of data. The final results are served via a web interface, ensuring ease of access and user-friendliness.

The code for this project can be found at https://github.com/nicolaegues/hzz_cloud_tech

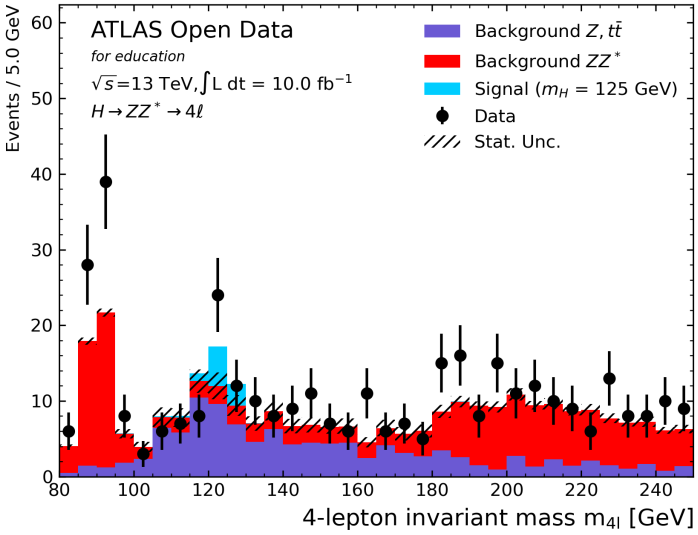

Figure 1: The final plot produced by the HZZ analysis notebook, showing the distribution of the 4-lepton invariant mass of events in ATLAS experimental data (Data), and Monte Carlo simulated data (Signal and Background(s)

1 Introduction

The ATLAS experiment is a large-scale particle physics detector at the Large Hadron Collider (LHC) at CERN, designed to study high-energy proton-proton collisions. Its goal is to explore the fundamental forces and particles of the universe, search for new physics, and investigate phenomena such as the Higgs boson, a fundamental particle that was discovered by the ATLAS in 2012 [2].

To detect such rare particle interactions, the petabytes of data generated by the LHC collisions need to be processed and analysed in an efficient way. (sentence about what cern currently does cloud). This project makes use of ATLAS Open Data and an example data processing workflow [1] to demonstrate how cloud technologies can be leveraged to scale and optimise the processing of such large datasets.

The ATLAS example analysis notebook used in this project focuses on the discovery of the Higgs Boson, specifically the HZZ decay channel where the Higgs first decays into two Z bosons, each of which decays into a pair of two leptons, resulting in a final state characterized by four leptons - the **signal**. To pick up the signal, one part of the analysis thus involves filtering the data to match the lepton type and charge of these Higgs decay end-products.

There are, however, various other background processes that can lead to the same final four leptons. To distinguish the signal from this background (and thereby successfully detect Higgs decay events), the notebook plots the filtered events based on their 4-lepton invariant mass 1. If Higgs decay events are present, they should contribute to a peak in the histogram at around 125 GeV, corresponding to the Higgs boson mass - this is also observed in the plot. In addition to the experimental ATLAS data, this analysis is also performed for Monte Carlo (MC) simulation data obtained from randomised simulations of particle collisions according to the Standard Model. The MC data includes data for both the Higgs decay signal and the background processes (figure 1), serving as a theoretical benchmark for comparison with the real data.

Importantly, however, the original analysis notebook processes the data sequentially on a single machine. This does not scale well to larger datasets due to the limited processing power and the fact that every event is processed one at a time, which makes the workflow computationally expensive and time-consuming.

To address these limitations, the workflow is restructured to allow for parallel processing and distribution across multiple compute resources. The key idea hereby is to divide the analysis into smaller, individual tasks that can be processed concurrently by multiple "worker" containers, with the aim of creating a system capable of scaling to much larger datasets.

2 Methods

2.1 Containerisation

A container is an isolated environment that encapsulates everything an application needs to run, including all dependencies, configurations and libraries. This ensures consistency and reproducibility regardless of the computing infrastructure the application is run on - be it local machines, cloud nodes, or HPC clusters.

In the context of this project, containers are particularly valuable because they allow for easy scaling and distribution of the ATLAS HZZ analysis workflow. If more data needs to be processed, more containers can just be launched dynamically to balance the workload when needed.

2.2 Task Distribution and Communication

The analysis is containerised by dividing it into three main stages. The first stage is handled by a dedicated task manager container, responsible for splitting the dataset into smaller batches. This is done by collecting the URLs of the data samples and determining start and end indices within each sample to ensure the workload distribution is more balanced, as the event count per sample varies drastically. Each "task" that is sent off is thus a dictionary containing the URL to the assigned data sample, as well as the start and end indices that specify the subset of events to be processed.

The second stage is executed by several worker containers, each of which receives a task and performs the analysis independently - which mainly includes event filtering and invariant mass calculations. A final container then gathers the results from all workers and plots the final invariant mass distribution as seen in fig. 1.

To carry out the distribution of tasks and final re-gathering of the results, the system employs the message broker software RabbitMQ, which allows the manager container to publish the tasks to a message queue instead of sending them to the workers directly. The worker containers then retrieve tasks from this queue as they are made available. Once they've performed their analysis, they publish the results to a second results queue which the collector container is subscribed to. The collector waits until it has received all results before it proceeds with the plot generation.

There are various benefits to using this message-queueing system. Firstly, RabbitMQ ensures the computing resources (such as CPU and memory) are efficiently used, since the tasks are not sent to specific workers but are dynamically distributed to whichever worker is free to process them at that moment. This avoids overloading some workers while others remain inactive, balancing the workload better. Additionally, it's a very robust method, given that the task of a failed worker remains in the queue and can be picked up by another worker. Importantly, RabbitMQ also allows for the system to scale up if needed, as it automatically handles the distribution of tasks to any number of workers, making sure they are used efficiently.

2.3 Container Management via Kubernetes

Kubernetes is a platform that manages the deployment, scaling and operation of containers. Configuration files (in YAML format) are set up to describe all of the resources needed in the system and their relationships. The manager, worker, and collector containers, as well as RabbitMQ, can be deployed with a single command (kubectl apply), with Kubernetes automatically managing everything defined in those configuration files after they have been applied. This includes deciding which node in a cluster should run a given container based on available resources, or how to allocate the resources if it's being run locally, for instance. Additionally, if a container crashes, Kubernetes automatically takes care of restarting it or rescheduling it to another node.

The deployment process is automated through a combination of the following steps:

1. The Docker images for the manager, worker, and collector containers are built using docker-compose - a tool to automate this process by specifying a compose configuration file.

2. The RabbitMQ service is deployed, and the kubectl wait command is employed to ensure it is ready to handle communication between the manager and worker containers - as a premature deployment of the other containers would

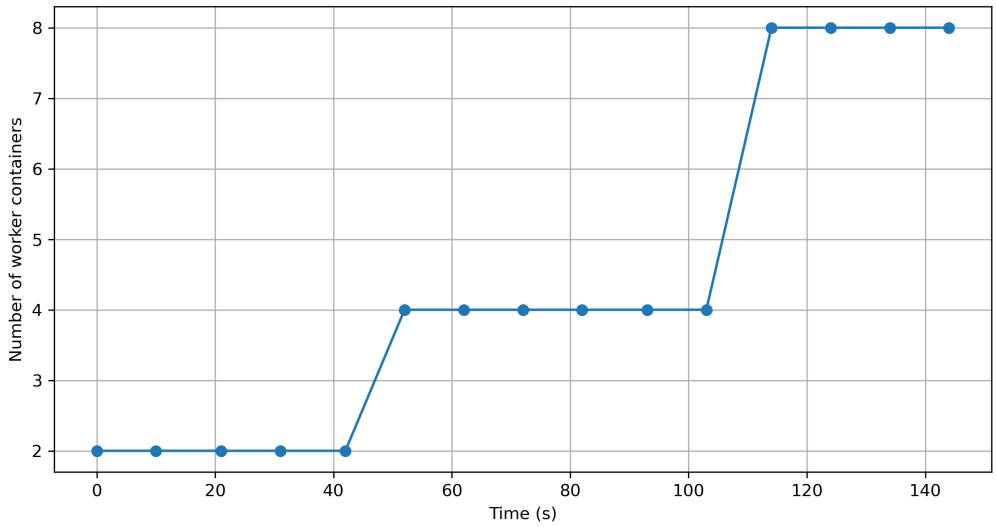

Figure 2: Plot showing the number of worker containers against the time passed since the deployment of the system

otherwise result in an error. This is enforced by defining a "Readiness probe" in RabbitMQ's configuration file, which ensures that at least a minute has passed until the status changes.

3. Once RabbitMQ is ready, the script applies the Kubernetes configuration files for the manager, workers, collector, web-server, and for the Persistent Volume (a piece of storage) as well as the Persistent Volume Claim (PVC). The latter two are implemented in order to store data that can persist even after the containers are stopped. The PVC ensures that other containers can read and write to this shared storage.

To ensure they only run once, the manager and collector containers are deployed as Jobs, which are marked as completed after they have been run once. The kubectl wait command is applied once again to track when the collector has finished.

4. After the collector container generates the final plot and stores it in the shared Persistent Volume (PVC), the webserver container, running an Nginx web server, makes the plot available over HTTP. When a user accesses the specified URL in their browser, the browser sends a request to the webserver container, which responds by serving the plot. Additionally, the plot is stored locally by tying the Persistent Volume to a designated directory, ensuring the plot persists even after the contaniners shut down.

5. Once the user has finished viewing the plot, the script provides an option to delete all Kubernetes resources (deployments, services, etc.) to clean up the local environment.

Most importantly, Kubernetes provides the "Horizontal Pod Autoscaling" tool, which here is implemented to automatically scale the number of worker containers based on CPU usage. Depending on the workload, the number of workers is thus scaled up or down, providing an efficient solution to cases where the dataset wants to be massively increased, for example.

3 Results

The system described above successfully demonstrates how a cloud-based containerised system can be leveraged to scale the ATLAS HZZ analysis workflow, turning the originally sequential, single-machine process into a parallel, distributed model.

The combination of the RabbitMQ queuing system and the tools provided by Kubernetes make this version highly scalable, as the HorizontalPodAutoscaling automatically increases or decreases the number of worker containers in response to the workload.

Figures 2 and 3 show Kubernetes' autoscaling in action and should be viewed alongside eachother, with the former displaying how the number of worker containers scale over time, and the latter showing the CPU utilization percentage over time. Initially, with a small number of workers, the CPU usage is high (250%), but as more workers are added,

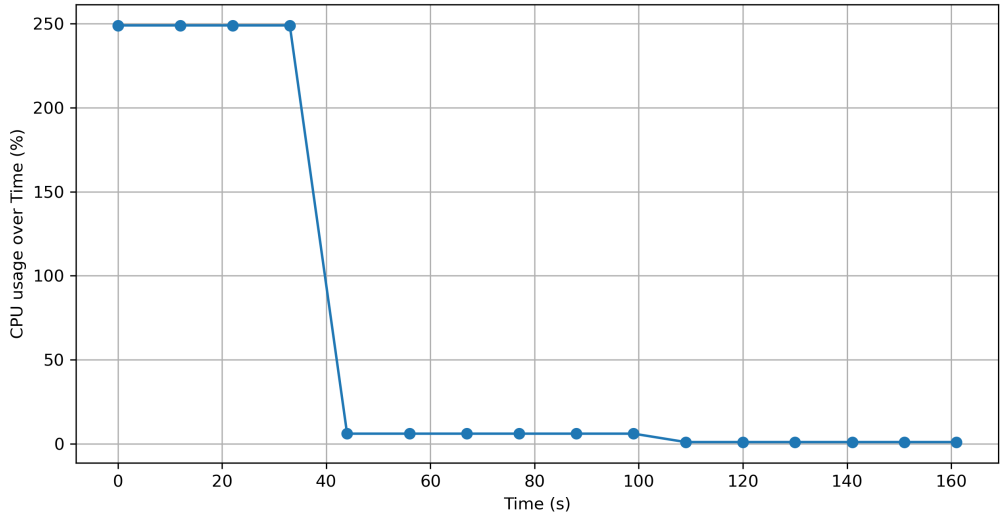

Figure 3: Plot showing the CPU usage percentage against the time passed since the deployment of the system

this decreases significantly, eventually stabilizing at around 1%. This is a clear indicator of the system's ability to distribute the workload efficiently across multiple containers.

Additionally, the web-server container, running an Nginx server, successfully serves the final plot over HTTP, reproducing the same figure as seen in figure 1.

4 Discussion

The Kubernetes-based solution presented in this report is inherently scalable due to its use of containerisation and the horizontal pod autoscaling tool, allowing it to handle much larger data volumes efficiently. If this solution were to be deployed on a large cluster of real or virtual machines, the analysis would have to be moved to a cloud provider such as Amazon Web Services (AWS) - where the containers would be distributed across multiple cloud nodes instead of running on a single machine. However, due to the isolated nature of the implemented cloud technologies, no changes to the workflow itself would be necessary - only to the infrastructure.

As more workers are added, however, the RabbitMQ messaging system could become a bottleneck, as the message queue could become overloaded with tasks waiting to be processed, and many messages sent at once could potentially lead to disk I/O bottlenecks.

Despite this, the system remains highly robust - thanks to the tasks persistence ensured by RabbitMQ and the self-healing capabilities of Kubernetes, whereby failed containers are automatically restarted and workloads rescheduled if needed.

Additionally, while the underlying technologies - Kubernetes, Docker, and RabbitMQ- introduce a level of complexity, this is largely counteracted by automation. The entire system can be setup, deployed, and cleaned up via a single command (that runs a bash script), making this solution highly accessible even to users with limited cloud computing experience.

Finally, the serving of the final plot via a web-interface enhances the user experience even further, as almost no interaction with the system is required.

5 Conclusion

In conclusion, the implementation of a Kubernetes-based, containerised system successfully transforms the ATLAS HZZ analysis into a scalable and efficient workflow. Despite potential bottlenecks in task queuing, the combination of RabbitMQ and Kubernetes leads to a very robust system that should, in theory, be capable of processing much larger

volumes of data.

References

[1] ATLAS Collaboration. *How to rediscover the Higgs boson yourself!* URL: https://github.com/atlas-outreach-data-tools/notebooks-collection-opendata/blob/master/13-TeV-examples/uproot_python/HZZAnalysis.ipynb.

[2] G. Aad et.al. "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". In: *Physics Letters B* 716.1 (2012), pp. 1–29. ISSN: 0370-2693. DOI: https://doi.org/10.1016/j.physletb.2012.08.020. URL: https://www.sciencedirect.com/science/article/pii/S037026931200857X.