Scalable Data Analysis of Higgs Boson Decay Using Distributed Systems in Kubernetes

Herbert S Rowan
Special Core in Software Engineering and High Performance Computing for Scientists, School of Physics, University of Bristol

This report details a containerized and Kubernetes-deployed distributed system for analyzing ATLAS Open Data, focusing on the $H \to ZZ \to 4\ell$ decay channel. Using modern data science libraries, RabbitMQ message queuing, Docker containerization, and Kubernetes orchestration, the workflow achieved scalability, reproducibility, and efficiency. The system processed data in parallel across multiple services, reconstructed the invariant mass of the Higgs boson ($\sim 125\,\mathrm{GeV}$), and demonstrated the feasibility of analyzing $H \to ZZ \to 4\ell$ decay events at scale.

INTRODUCTION

The ATLAS experiment at the Large Hadron Collider (LHC) produces petabytes of data annually, enabling studies of the Higgs boson and other particle physics phenomena. This project analyzes the $H \to ZZ \to 4\ell$ decay channel using a distributed system designed for scalability, leveraging Kubernetes for orchestration, RabbitMQ for inter-service communication, and Docker for containerization. Together, these tools ensure the system is reproducible, fault-tolerant, and capable of handling large-scale data analysis.

OVERVIEW OF KEY TECHNOLOGIES

Docker: Containerization for Reproducibility

Docker is a platform that uses containerization to package applications and their dependencies into lightweight, portable units called containers. Containers ensure that the application runs consistently across different environments by bundling all required libraries, dependencies, and configurations.

For this project:

- Each stage of the workflow (data acquisition, processing, simulation, aggregation, and visualization) is encapsulated in its own Docker container
- Containers ensure that the workflow is reproducible, as the same environment is guaranteed on any machine or cloud platform
- The lightweight nature of containers allows multiple services to run in isolation on the same host without interference

RabbitMQ: Message Queuing for Inter-Service Communication

RabbitMQ is an open-source message broker that enables asynchronous communication between distributed services using a publish-subscribe model. Services communicate by sending and receiving messages through queues, ensuring decoupled and fault-tolerant workflows.

For this project:

- RabbitMQ facilitates communication between services. For example, the data acquisition service notifies the data processing service when files are ready for processing
- By decoupling the services, RabbitMQ ensures scalability. Each service can operate independently, processing messages at its own pace
- The message queue provides a buffer, enabling the system to handle bursts of tasks or failures gracefully. If one service crashes or slows down, messages are stored in the queue and processed later
- This architecture improves fault tolerance and simplifies debugging by isolating individual service failures

Kubernetes: Orchestrating Distributed Systems

Kubernetes is an open-source platform for managing containerized applications in distributed environments. It automates deployment, scaling, and management of containers, ensuring high availability and efficient resource utilization.

For this project:

- Kubernetes manages the deployment of all services (Docker containers) across a Minikube cluster
- Kubernetes Persistent Volumes (PVs) allow data to be shared between services (e.g., between data acquisition and processing)

- Horizontal Pod Autoscaling (HPA) enables dynamic scaling of services based on CPU or memory usage, ensuring efficient handling of increased workloads
- Kubernetes ensures fault tolerance by automatically restarting failed Pods or rescheduling them on available nodes

How Docker, RabbitMQ, and Kubernetes Work Together

The integration of these tools forms a scalable, distributed, and fault-tolerant system:

- **Docker**: Encapsulates each stage of the workflow, ensuring reproducibility and consistency
- **RabbitMQ**: Coordinates communication between services, enabling asynchronous and decoupled task execution
- **Kubernetes**: Orchestrates the entire system, automating container deployment, scaling, and recovery

For example:

1. The data acquisition container downloads ROOT files and notifies RabbitMQ when they are ready for processing
2. Kubernetes schedules the data processing container, which retrieves messages from RabbitMQ and processes the data
3. Results are passed to the aggregation container and, finally, to the visualization container, which generates plots

This modular design ensures scalability, fault tolerance, and reproducibility.

WORKFLOW OVERVIEW

The analysis pipeline consists of five stages: data acquisition, preprocessing, Monte Carlo simulation, aggregation, and visualization. Each stage is encapsulated in a Docker container and orchestrated within a Kubernetes cluster. RabbitMQ facilitates reliable inter-service communication, while Persistent Volumes in Kubernetes ensure data persistence across services.

Data Acquisition

The data acquisition service downloads experimental and Monte Carlo (MC) datasets from the ATLAS Open Data repository. Python's `requests` library handles automated downloads, while `uproot` parses ROOT files. The service writes data to a shared Kubernetes Persistent Volume (`data-pvc`) to enable downstream processing.

Preprocessing and Event Selection

Preprocessing includes:

- Event selection using Awkward Arrays for efficient handling of jagged data
- Identification of lepton pairs ($\ell^+\ell^-$) and Z bosons based on:
    1. Opposite charge and same flavor (electrons or muons)
    2. $p_T > 20\,\mathrm{GeV}$ and $|\eta| < 2.5$
- Reconstruction of four-lepton events (4ℓ) to analyze $H \to ZZ \to 4\ell$

The Higgs boson invariant mass m_H is reconstructed using:

$$m_H = \sqrt{(E_Z^1 + E_Z^2)^2 - (\vec{p}_Z^1 + \vec{p}_Z^2)^2}$$

Monte Carlo Simulation

MC datasets simulate signal ($H \to ZZ$) and background processes ($Z + \text{jets}, t\bar{t}$). These datasets provide a baseline for distinguishing signal events from background.

Aggregation

Aggregated datasets from preprocessing and simulations are saved in Parquet format for efficient access and storage.

Visualization

The visualization service generates histograms of invariant mass distributions, highlighting the Higgs boson resonance around $125\,\mathrm{GeV}$. An example output is shown in Figure 1.

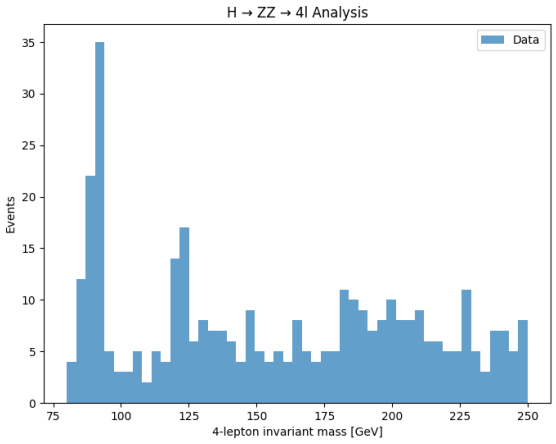

FIG. 1. Invariant mass distribution of $H \to ZZ \to 4\ell$ events. The Higgs boson resonance is visible around $125\,\mathrm{GeV}$. It's simple to edit the visualization script to obtain different plots such as plotting monte-carlo data.

SCALABILITY

The workflow supports distributed execution using Kubernetes and RabbitMQ. Tasks like event selection, invariant mass calculation, and aggregation are parallelized. This design is scalable to larger datasets and can be extended to cloud infrastructures.

Scaling Up Nodes Locally

To test scalability with multiple virtual nodes locally, you can set up a multi-node Kubernetes cluster using Minikube. This setup simulates a larger, distributed environment, which can handle more Pods and provide more resources for processing.

Once Minikube is running with multiple nodes, Kubernetes will automatically schedule Pods across the available nodes.

Additionally, you can scale your Deployments across nodes to observe the system's scalability.

You can also perform load testing using tools like `hey` to simulate real-world traffic and observe how Kubernetes handles the increased load by scaling Pods across the available nodes.

Potential Bottlenecks and Constraints

While Kubernetes and RabbitMQ provide scalability, there are potential bottlenecks and constraints that must be considered when scaling up the system:

- **Data Transfer and I/O**: As the number of nodes increases, the data transfer between services (e.g., between data acquisition, processing, and storage) may become a bottleneck. To mitigate this, services can be deployed on nodes with local storage and optimized for faster I/O, or cloud-based storage systems (e.g., AWS S3) can be utilized to provide high-speed access to datasets

- **Message Queues Overload**: RabbitMQ, while scalable, may encounter issues with high message throughput or long message processing times as more nodes are added. To avoid this, RabbitMQ clustering can distribute load across multiple queues, improving throughput and fault tolerance

- **Pod Scheduling**: Kubernetes may struggle to efficiently distribute Pods across nodes if there are resource constraints. Configuring resource limits (CPU, memory) and node affinity rules can help ensure efficient resource allocation and balanced distribution

- **Network Latency**: As the number of nodes grows, network latency may impact inter-service communication. Ensuring that services with high communication requirements are co-located on the same node or in close proximity can mitigate this issue

To address these potential bottlenecks, it's crucial to monitor system performance, set resource limits appropriately, and optimize message handling using techniques such as RabbitMQ clustering.

Enhancements for scalability include:

- Horizontal Pod Autoscaling (HPA) dynamically adjusts the number of Pods based on CPU or memory usage

- Multi-node setups in Minikube simulate real-world distributed clusters

- RabbitMQ clustering distributes messages across multiple queues for improved load balancing

RESULTS

The analysis successfully reconstructed the Higgs boson invariant mass peak at $125\,\mathrm{GeV}$, consistent with Standard Model predictions. Figure 1 highlights the clear separation of signal and background distributions.

CONCLUSION

This project demonstrates a scalable, distributed workflow for high-energy physics analysis using Kubernetes, RabbitMQ, and Docker. By leveraging containerization and orchestration, the workflow achieves reproducibility, efficiency, and scalability, making it a robust solution for analyzing large datasets.

Future improvements include, integration with cloud platforms for larger-scale analyses and enhanced load balancing and fault tolerance through RabbitMQ clustering

APPENDIX

VALIDATION OF SUCCESSFUL EXECUTION

The successful execution of the distributed system was verified using terminal logs and generated output files. This section details the workflow through screenshots captured during the execution.

Building the Docker Image and Deploying the Kubernetes Cluster

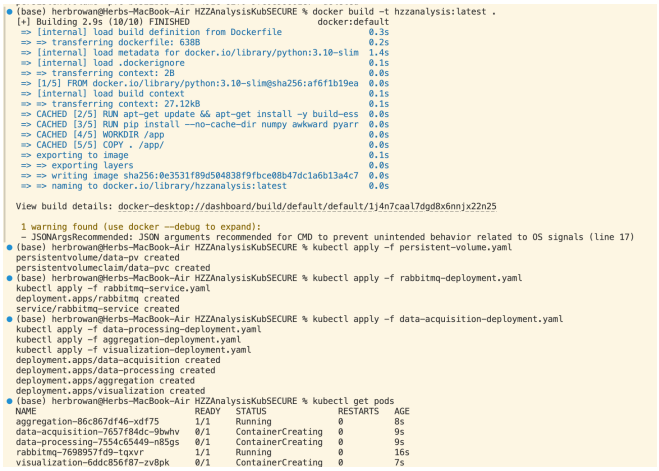

FIG. 2. Building the Docker image using the `docker build` command and deploying the Kubernetes cluster with `kubectl apply`.

Monitoring Kubernetes Pods

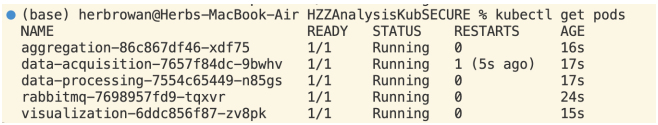

FIG. 3. Initial status of Kubernetes Pods after deployment, showing Pods transitioning from `ContainerCreating` to `Running` and `Completed` states.

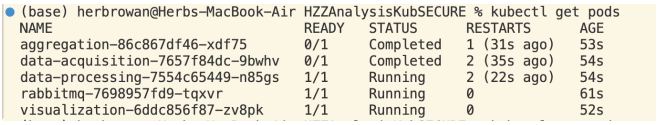

FIG. 4. Final state of all Kubernetes Pods, with completed data acquisition and aggregation tasks, and running RabbitMQ and visualization services.

Copying and Verifying Output Files

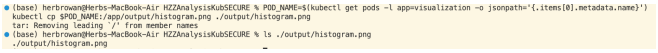

FIG. 5. Copying the generated histogram file from the visualization Pod to the local system using the `kubectl cp` command.