ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING

Department of Electrical, Electronic and Information Engineering
"Guglielmo Marconi" DEI

Master's Degree in Automation Engineering

Master Thesis in

Image Processing And Computer Vision

A prototype of the data quality pipeline of the Online Observation Quality System of ASTRI-Mini Array telescope system

Candidate:
Luca Castaldini

Supervisor:
Prof. Luigi Di Stefano
Co-supervisor:
Dr. Andrea Bulgarelli
Dr. Nicolò Parmiggiani

Academic Year
2021–2022
Third Session

Contents

1 Introduction **5**

1.1 Gamma-ray Astronomy 6

1.2 Cherenkov Telescopes 13

1.3 ASTRI Mini-Array 15

2 Online Observation Quality System **22**

2.1 SCADA Context 24

2.2 Cherenkov Camera Context 25

2.3 OOQS use cases and requirements 28

3 OOQS-Pipeline development **33**

3.1 Programming Languages 33

3.1.1 C++ 33

3.1.2 Python 34

3.1.3 C++ vs Python 35

3.2 Frameworks 35

3.2.1 DQ-Pipe 35

3.2.2 Slurm 39

3.3 Tools/libraries 40

3.3.1 Apache Kafka 40

3.3.2 Apache Avro 42

3.3.3 HDF5 Library 45

3.3.4 GSL Library 47

3.4 Storage Systems 49

3.4.1 MySQL 49

3.4.2 MongoDB 50

3.5 OOQS-Pipeline Architecture 51

3.5.1 Kafka Consumer 53

3.5.2 DQ-Analysis 62

3.5.3 DQ-Aggregator 74

4 Deployment and Performance Evaluation **80**

4.1 Test Data and Environment 80

4.2 Performance 82

Conclusions **94**

Abstract

Gamma-ray astronomy investigates the physics of the universe and the characteristics of celestial objects through gamma rays. Gamma-rays are the most energetic part of the electromagnetic spectrum, emitted in some of the brightest events in the universe, such as pulsars, quasars, and supernova remnants. Gamma rays can be observed with satellites or ground-based telescopes. The latter allow to detect gamma rays in the very high energy range with the indirect Cherenkov technique. When highly energetic photons enter Earth's atmosphere, they generate air showers, cascades of particles whose fast motion produces elusive flashes of blue Cherenkov light in the sky.

This thesis discusses the research conducted at the Astrophysics and Space Science Observatory of Bologna in collaboration with the international project, guided by INAF, for ground-based gamma-ray astrophysics, ASTRI Mini-Array. The focus is on the Online Observation Quality System (OOQS), which conducts a quick look analysis during the telescope observation. The Cherenkov Camera Data Quality Checker is the OOQS component that performs real-time quality checks on the data acquired at high frequency, up to 1000 Hz, and with a total bandwidth of 148MB/s, from the nine Cherenkov Cameras. The thesis presents the implementation of the OOQS-Pipeline, a software prototype that receives scientific packets from a Cherenkov Camera, performs quality analysis, and stores the results. The pipeline consists of three main applications: Kafka-Consumer, DQ-Analysis, and DQ-Aggregator. The pipeline was tested on a server having similar performance as the ones of the Array Observing Site, and results indicate that it is possible to acquire the maximum data flow produced by the cameras. Overall, the thesis presents an important contribution to the ASTRI Mini-Array project, about the development of the first version of the OOQS-Pipeline, which will maximize observation time with quality data passing the verification thresholds.

Chapter 1

Introduction

The general context of this thesis project regards gamma-ray astronomy, which investigates the physics of the universe and the characteristics of celestial objects.

In this introductory chapter, we will explore the field of *gamma-ray astronomy* and its general concepts. We will delve into the history of gamma-ray space observatories, comparing their capabilities and observing energy range. Through this comparison, we will show the trend of growing performance in gamma-ray observatories over time and why ground-based observations are an important counterpart. Afterward, we will discuss the importance of the *Cherenkov effect* in ground-based gamma-ray astronomy. This effect is crucial in the indirect observation of gamma rays. With a comprehensive understanding of the general concepts besides *Imaging Air Cherenkov Telescopes* (*IACTs*), we aim to provide a solid foundation for the research conducted in this thesis on the **ASTRI Mini-Array** [1] Cherenkov telescope.

This thesis project aims to deal with the data management and elaboration of data produced from cameras for astronomical purposes in a ground-based gamma-ray observatory. To achieve this objective a prototype of software consisting of a *data elaboration pipeline* has been developed. This software is capable of performing in real-time the following activities: data collection, data quality analysis, and storage of the results on a database.
The obtained outcome will be exploited to signal various telescope software systems, as well as the operator, about abnormal activities and trigger their reaction to promptly correct anomalies.

In the following chapters, we deepen the various aspects regarding this project.
In Chapter 1, we describe the ASTRI Mini-Array project and its scientific objectives.
In Chapter 2 we present the ASTRI Mini-Array on-site software system which man-

ages the operation of the telescope during the observations.
In Chapter 3 we discuss the implementation of the pipeline, presenting all the principal tools, libraries and services used, and finally our software solution.
In Chapter 4 we show the result of the tests performed on the pipeline, providing performance metrics.

1.1 Gamma-ray Astronomy

Modern astronomy is defined *multi-messenger*[2] due to the coordinated interpretations of different signals:

- *Cosmic Rays* (*CRs*), they are constituted from electrically charged particles, mostly protons but also atomic nuclei. CRs are produced in galactic and extragalactic sources and are deflected by electromagnetic fields of other sources, losing any information about their origin.

- *Photons*: They are the basis of astronomy and are detected at different wavelengths, from radio to gamma rays, each providing information about the different physical processes that produce them. They are important because being neutral they are not deflected by magnetic fields while preserving the direction of the source.

- *Neutrino*: by weakly interacting with matter they can reach us from the depths of the Cosmos bringing fundamental information about the most remote astrophysical sources and the most powerful mechanisms of particle acceleration. Similarly to photons, they are neutral, and their arrival direction indicates the location of the production sites.

- *Gravitational Waves*: directly observed only since 2015, are the new frontier of astrophysics. They allow us to study the characteristics of incredible phenomena such as the fusion of neutron stars or black holes that occur in the depths of the Universe.

Gamma-ray astronomy is the field involved in the astronomical observation and study of *gamma-rays*. Gamma rays are highly energetic photons with energies greater than 100 keV[1], which represents the most extreme portion of the electromagnetic spectrum. The electromagnetic spectrum ranges from radio waves, which

[1] eV: *electron volt*. It is a unit of energy commonly used in atomic and nuclear physics. It is equal to the kinetic energy gained by an electron when the electrical potential at the electron increases by one volt. It is equal to $1\,\mathrm{eV} = 1.602 \cdot 10^{-19}\,\mathrm{J}$.

have the lowest energies, through visible optical light, which has higher energies, to gamma rays, which have the highest energies.
Gamma radiation can be distinguished from x-rays by its higher energy and shorter wavelength; in gamma-ray astronomy, we study energies ranging from a few hundred keV to several PeV. Gamma-ray photons travel in straight lines, which enables an accurate determination of their origin. In astronomy, gamma rays are emitted by various sources such as supernova remnants, pulsars, active galactic nuclei, and Gamma-Ray Bursts (GRBs). The highest energy photon that has been detected is in the range of 1.4 PeV [3]. GRBs, on the other hand, are powerful and intense bursts of gamma rays that are released for a short period, typically lasting from a few milliseconds to a few minutes. GRBs are some of the brightest and most energetic events in the universe [4]. *GRB-221009A* is a recent rare event with very high energy photons up to 18 TeV and isotropic-equivalent energy estimated at $\approx 5.9 \cdot 10^{54}$ ergs.[2] To make a comparison, the Sun delivered power is about $3.8 \cdot 10^{33}$ ergs/s. If the sun emitted the same energy as GRB-221009A, it would take about $4.9 \cdot 10^{20}$ years. The collection of the event was complicated by the saturation effects of multiple gamma-ray instruments due to the relatively low distance of the source [5]. The Swift team[6] revealed the exceptional rarity of such an event, for its intrinsic luminosity and very nearby to GRB standards. Only $1/10^4$ of the total acquired GRBs have total energy so high and refactoring based on the distance and same luminosity, it is estimated that only one event every 1000 years can happen so nearby and bright.

Gamma-rays are produced by several kinds of processes, One of the main ones associated with high-energy gamma-rays is *particle acceleration*. It consists of the interaction between particles like charged nuclei, electrons, or positrons and several targets like interstellar gas densities, and magnetic or radiation fields. In this situation, particles can be accelerated to relativistic speeds. When these particles collide with other particles, they can produce gamma-ray emissions. A *relativistic outflow* is a common situation of particle accelerations constrained in wind or jets. These outflows are generally observed in gamma-ray source classes, such as AGNs and GRBs. The accretion of a disk of material around black holes can lead to the formation of a collimated jet of plasma orthogonal to the disk. The study of such collimated flows is important to determine the laws that govern these enigmatic objects.
The observations of gamma rays could also give us the chance to indirectly ob-

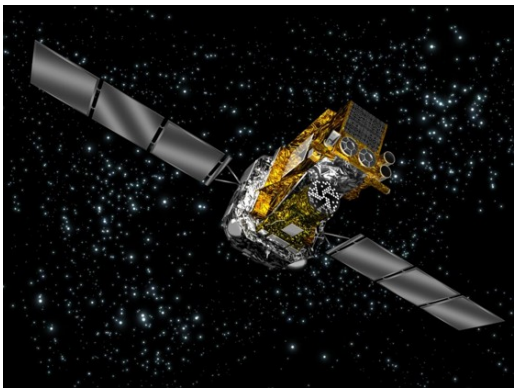
[2] *erg*, the unit of energy or work in the centimeter-gram-second system of physical units used in physics. It is defined as $\mathrm{g} \cdot \mathrm{cm}^2/\mathrm{s}^2$. $1\,\mathrm{erg} = 10^{-7}\,\mathrm{J}$. $1\,\mathrm{erg} \approx 624\,\mathrm{GeV}$.

serve signatures of *Dark Matter* (*DM*). The search for DM particles is a relevant topic in high-energy gamma-ray astronomy, coupled with experiments conducted in accelerators on Earth [4].

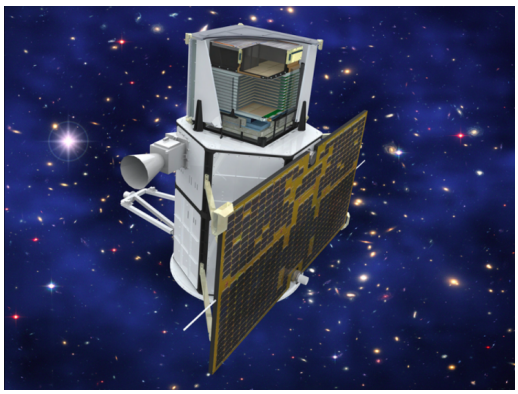
Space Based gamma-ray astronomy

(a) Fermi Gamma-ray Large Area Space Telescope – 2001

(b) Swift Gamma Ray Burst Explorer - 2004



(c) INTErnational Gamma-Ray Astrophysics Laboratory (INTEGRAL) - 2002



(d) Astrorilevatore Gamma a Immagini LEggero (AGILE) - 2007

Figure 1.1: Modern space based gamma-ray observatories

The atmosphere is a powerful screen for gamma rays and a minimization of the screen thickness is required for direct observation. The first observations about gamma rays were performed in the 50s. The energy range of 0.2 MeV to 400 MeV has been explored in the high atmosphere employing balloons able to reach an altitude of 25 km above the sea level, and successively by rockets equipped with Geiger counters [7]. Space-based missions carried out several breakthroughs in the high energy Universe. Satellites like *Explorer XI* [8] and *OSO-3* [9] launched in the 60s discovered the interaction between the Earth's Atmosphere and CRs and the interaction between galactic and extra-galactic sources. *Vela* satellites [10] detected

for the first time a GRB, despite Vela's mission being for military operations. The *Compton Gamma-ray Observatory* (CGRO)[11], launched in the 1980s, took data from 1991 until 2000 and was equipped with four instruments able to map the entire sky. These instruments covered six decades of the electromagnetic spectrum, ranging from 30 keV to 30 GeV, including the Burst And Transient Source Experiment (BATSE), the Oriented Scintillation Spectrometer Experiment (OSSE), the Imaging Compton Telescope (COMPTEL), and the Energetic Gamma-Ray Experiment Telescope (EGRET). The instruments on Compton provide 10 times the sensitivity of previous observatories, along with greatly improved angular resolutions and great timing capabilities. There are several space-based gamma-ray observatories active nowadays. Swift[12] and INTEGRAL are monitoring the sky for lower energies, from 10 keV to a few MeV. INTEGRAL[13] has been launched in 2002 and is dedicated to the fine spectroscopy and imaging of celestial gamma-ray sources. The Swift Gamma-Ray Explorer has been launched in 2004, and it is designed to make prompt multi-wavelength observations of GRBs and their afterglow.
A new era for space-based observations starts in 2007 with *AGILE* (Astrorilevatore Gamma a Immagini LEggero) and *Fermi Gamma-ray Large Area Space Telescope* in 2008[14]. AGILE[15] is a scientific mission of the Italian Space Agency (ASI). It is equipped with several subsystems that form the Gamma-Ray Imaging Detector(GRID) that is used for the observations in the 30 MeV − 50 GeV energy range. The major successes of the AGILE mission are the discovery of new pulsars, the confirmation of the micro-quasar Cygnus X-3 (in conjunction with Fermi), and other gamma-ray emissions from the Milky way and Terrestrial Gamma-ray Flashes [16]. Fermi[17] is for the study of electromagnetic radiation emitted by celestial bodies in the energy range between 8 keV − 300 GeV (gamma-rays), the observatory includes two scientific instruments: Large Area Telescope (LAT) and the Gamma-Ray Burst Monitor (GRB), which can scan about the 70% of the sky at any instant. Fermi made several important discoveries like the observations of new gamma-ray pulsars and *Terrestrial Gamma-ray Flash* (TGF). In 2017, Fermi detected a short GRB from two colliding neutron stars, while scientists at the National Science Foundation's Laser Interferometer Gravitational-wave Observatory (LIGO) detected gravitational waves coming from the same source [18]. This GW event was observed with instruments during the following days[19]. In the same year, the detection of the Neutrino detected from IceCube[20] (IceCube-170922A) which triggered an extensive campaign of observations from multiple instruments, such as Fermi, revealed an electromagnetic emission from blazar TXS 0506+056, at the same time neutrino arrived. These events started the so-called "*multi-messenger era*" [21]

The **TeV gamma-rays** are rare observable events but are widely produced in intensive particle acceleration interacting with the gas and radiation fields and are a typical contribution to the high bolometric luminosity of young Supernova Remnants, Pulsar Wind Nebulae, compact Binary Systems, and many other galactic and extra-galactic sources. The instruments installed on AGILE and Fermi provide effective detection of gamma-rays above 1 GeV. However, beyond 10 GeV, detected gamma-ray fluxes are generally very faint, and this instrumentation is not able to provide significant information in the range above 100 GeV (**Very High Energy** VHE). At the same time, the exploration of the VHE range is encouraged by the presence of several photons with energies between a few tens of GeV up to 0.1 TeV during the observations of bright GRBs [22]. Unfortunately, to observe efficiently this domain, the instrument's surface area increases dramatically, making space-based observatories for VHE unfeasible due to the moderate weight and space available on launch vehicles and high costs. An alternative method for observing these energies is based on the Cherenkov Radiation [23].

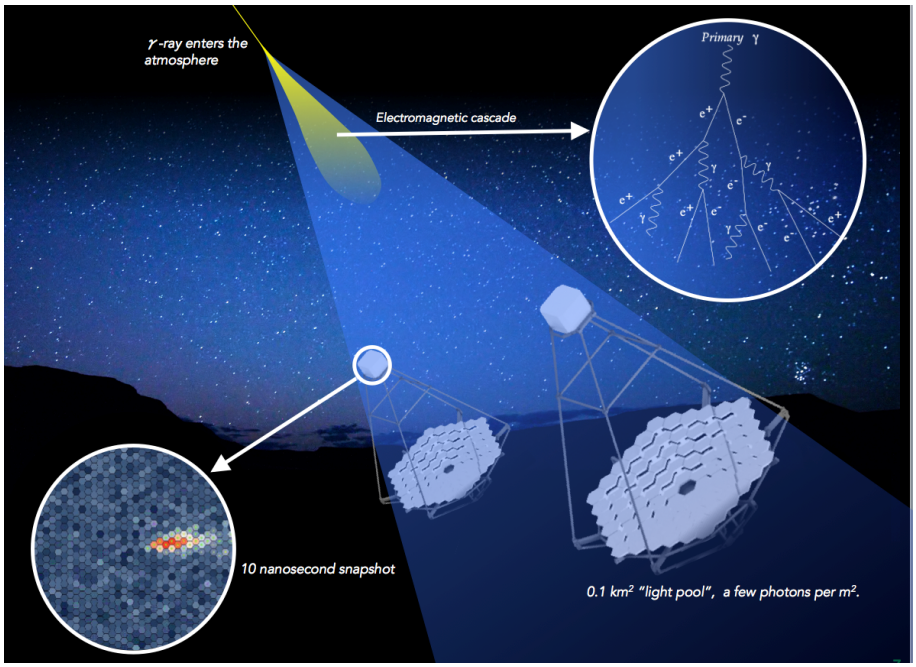


Figure 1.2: CTA Telescopes collecting the electromagnetic cascade that caused a high energy gamma ray entering the atmosphere. Picture from CTA's website (3)

Cherenkov Effect in Gamma Ray Astronomy

Cherenkov radiation is electromagnetic radiation that is produced by charged particles moving faster than the speed of light across a material. It manifests as a cone

of *UV/optical blue light* that is produced when a charged particle travels through a target material faster than the speed of light in that target [24].
The emission angle produced by the Cherenkov effect is represented in Fig: 1.3. A particle travels in a medium with speed v_p

$$\frac{c}{n} < v_p < c \tag{1.1}$$

where c is the speed of light in a vacuum and n is the refractive index of the medium.
We define the ratio between the speed of the particle and the speed of light as

$$\beta = \frac{v_p}{c} \tag{1.2}$$

The Cherenkov light will have a resulting speed equal to

$$v_{Ch} = \frac{c}{n} \tag{1.3}$$

The left corner of the triangle represents the location of the particle when it meets the medium; the right corner represents its position at some instant of time t.

The emission angle θ is given by the ratio between the position of the Cherenkov wave at time t, given by $x_{Ch} = \frac{c}{n}t$ and the position of the particle at time t, meaning $x_p = \beta ct$, that results in:

$$\cos\theta = \frac{x_{Ch}}{x_p} = \frac{1}{n\beta} \tag{1.4}$$

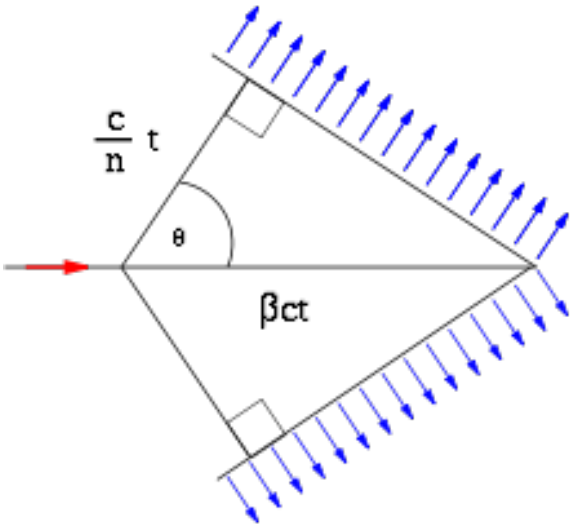


Figure 1.3: The geometry of the Cherenkov radiation shown for the ideal case of no dispersion

When a particle, which could be a proton, a nucleus, an electron, a photon, or more rarely a positron, a phenomenon called **Extensive air shower** takes place.

In this high-energy collision, primary particles release energy and split into sub-particles. The air shower is the result of a *nuclear cascade* process composed of successive interactions of sub-particles and air nuclei until all the energy contained in the primary particles is dissipated. The particles produced in each interaction can be unstable and decay, while other particles interact again with air nuclei and propagate downwards although with lower energy. Some of the produced particles in this cascade are neutral pions, which decay almost instantly, generating two gamma rays. These gamma rays initiate the *electromagnetic cascades* that produces photons, electrons, and positrons constituting the electromagnetic component of the *Extensive Air Shower* (EAS) [25]. The particle multiplication proceeds until the energy of the produced particles become so low that the multiplication stops. Depending on the primary energy and nature of the initial incident particle, the shower could stop at a high altitude, or it can reach the ground.

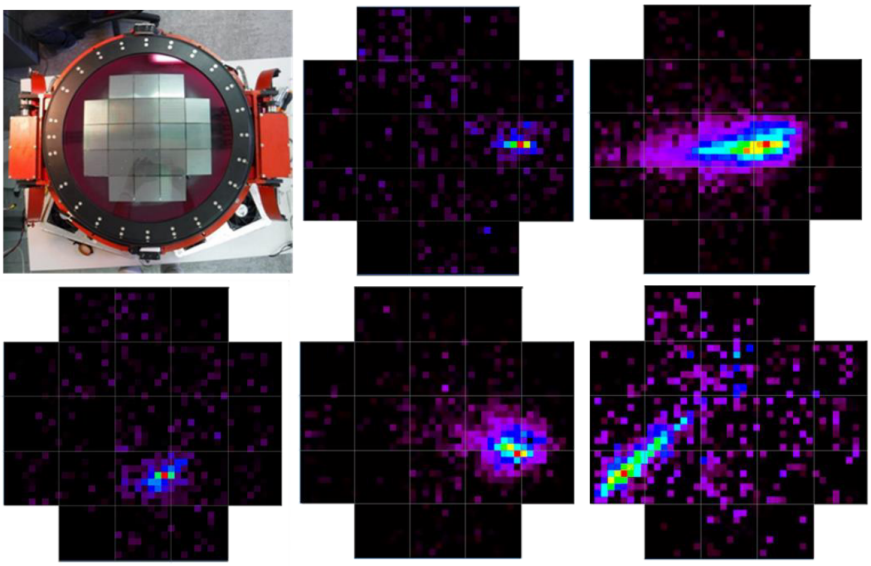


Figure 1.4: The ASTRI-Horn Cherenkov Camera, composed of 21 PDMs, and some acquisitions of gamma-ray air-showers. We can see the background given by cosmic-rays air showers. Picture taken from [26]

In the context of gamma rays observations, when *gamma rays* with energy greater than 1 TeV enter the atmosphere, they generate great electromagnetic cascades: the highly relativistic particles cause a flash ($\sim$ 2 ns) of UV-blue Cherenkov light, which propagates in a cone with an opening angle of $\sim$ 1 deg. The resulting circle of projected light on the ground, at $\sim$ 2000 m above sea level, typically has a radius of about 120 m.[27]. However, also nucleons and heavy nuclei coming from CRs can produce air showers, and this constitutes an irreducible background in the detection of gamma rays. CRs are highly isotropic and uniform in time, despite their statistical

fluctuation, we can identify gamma rays from point sources as spatial distributions of temporal distributions[25]. We can see an example of the phenomenon in Fig. 1.4.

1.2 Cherenkov Telescopes

In *ground-based gamma-ray astronomy*, observations of air showers it is performed by the **Imaging Air Cherenkov Telescopes** (IACTs) which perform night observations of the sky. For the detection is sufficient a large optical reflector equipped with a fast optical camera. The dimension of the telescopes can be large. For example, consider a telescope composed of a 10m diameter refractor, a multichannel camera composed of pixels of about 1/4 degree, and a field of view of 3 degrees. It can collect a primary gamma-ray of energy greater than 100 GeV across ground-level distances as large as 100 m, providing an effective detection area of 10^4 m^2.
The registered Cherenkov light image will have a proportional number of photons to the energy absorbed in the atmosphere. The orientation of the image, correlated with the arrival direction of the gamma-ray photon and its shape, contains enough information to determine the origin of the primary particle [23]. The use of two or more telescopes situated approximately 100 m apart for **stereoscopic observations** of air showers offers a significant improvement in background rejection (factor of 100) and excellent angular resolution (better than 0.1°) and energy resolution (better than 15%).
The benefits of stereoscopic observations of air showers, including effective rejection of hadronic showers and improved angular and energy resolution, demonstrate the need for IACTs to be an array of telescopes. This arrangement allows for more accurate and efficient detection of gamma rays, making IACT arrays a crucial tool in the study of high-energy astrophysics [23]. The Whipple 10 m reflector[28] was built in 1968 and was the first large purpose-built IACT. Thereafter came the HEGRA[29] array the Cherenkov Array at Themis (CAT)[30] telescope. These were the pioneers of the first generation of atmospheric IACTs. The current generation of IACTs, including the High Energy Stereoscopic System (H.E.S.S.) [31], the Major Atmospheric Gamma Imaging Cherenkov (MAGIC) telescopes [32], and the Very Energetic Radiation Imaging Telescope Array System (VERITAS) [33], have followed in their footsteps.

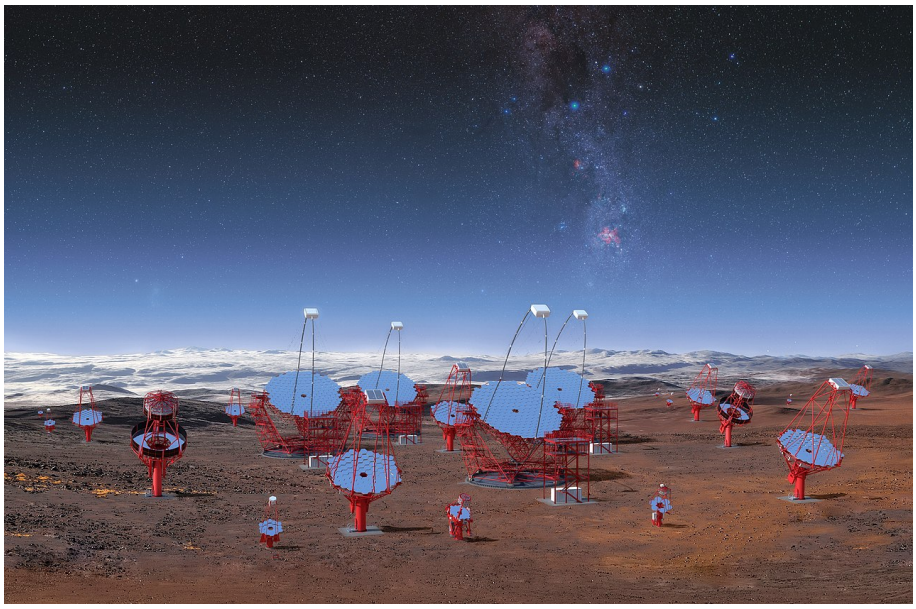


Figure 1.5: An initial render of Cherenkov Telescope Array southern hemisphere observatory (Chile, Atacama Desert. It is composed of four LSTs in the middle, surrounded by MST and various prototypes of SST, from right to left: SST-1M, ASTRI, GCT [34]. Picture from CTA's website (3)

Cherenkov Telescope Array

The *Cherenkov Telescope Array* (*CTA*)[35] is a next-generation observatory for ground-based gamma-ray astronomy[3]. For the first time in the VHE field, this observatory will be driven by proposals and will have open access to data. This observatory aims to build and operate more than 60 IACTs of various classes. The structure of CTA consists of an array of telescopes spread over several square kilometers, allowing for a wide field of view and improved sensitivity. The array is divided into two sites, one in the northern hemisphere and one in the southern hemisphere, providing full sky coverage.

CTA will make use of three types of telescopes, Large-Sized Telescopes (LST) with a diameter of up to 23 meters, Medium-Sized Telescopes (MST) with a diameter of 12 meters, and Small-Sized Telescope (SST) with a diameter of 4 meters. CTA will explore a wide high-energy range based on the telescope's different sensitivities [36]. SST design is based on the ASTRI prototype "ASTRI-Horn" and other telescopes [27].

The CTA is supported by the CTA Consortium, including 1200 members from more than 200 institutes in 31 countries, and it is actually under construction. CTA's north observatory (*CTAO-North*) is located on the existing site of the Instituto de

[3]CTA website: https://www.cta-observatory.org/

Astrofisica de Canarias on the island of La Palma. CTA's south observatory (*CTAO-South*) is less than 10 km southeast of the European Southern Observatory's (ESO's) existing Paranal Observatory in the Atacama Desert in Chile. In Fig 1.5 we can see a graphical representation of the base.
The scientific goals of CTA are diverse, including the study of active galactic nuclei, gamma-ray bursts, and dark matter. CTA will also search for evidence of new physics beyond the standard model and provide crucial data for our understanding of the high-energy universe. With its advanced technology and wide field of view, CTA has the potential to make groundbreaking discoveries in the field of astrophysics.
CTA aims to capture the elusive cascades of gamma-ray photons produced when high-energy gamma rays hit the Earth's atmosphere [37].
Due to the absolute rarity of the air showers, CTA uses two distinct observatories to maximize the probability of collecting these events. There is an asymmetric distribution of telescopes between the two observatories. In the CTAO-North, the array size is more limited than in the CTAO-South, and it will focus more on the energy range between 20 GeV and 5 TeV. On the other hand, the CTAO-South has a unique observation point of view of the central region of the Milky way and it will cover the energy range from 150GeV to 300TeV. In essence, the planned increase in the number of telescopes for CTA will lead to significant advancements in the observatory's capabilities. This is due to the enhanced sampling of individual showers and the ability to capture the complete light pool for a greater number of showers. These improvements will result in superior performance compared to current IACTs [27].

1.3 ASTRI Mini-Array

ASTRI (**Astrofisica con Specchi a Tecnologia Replicante Italiana**) **Mini-Array**[4][1] is a project from *INAF* (*Istituto Nazionale di AstroFisica*). It consists of an array of nine innovative IACTs that have evolved from the dual-mirror ASTRI Horn telescope[38]; in Fig. 1.6 is shown the telescope array at the observing site.

The scientific objective of ASTRI Mini-Array (MA) is to exploit the imaging atmospheric Cherenkov technique to measure the energy, direction, and arrival time of gamma-ray photons crossing Earth's atmosphere from astrophysical sources. There are several Imaging Atmospheric Cherenkov Telescopes able to perform high-energy observations, like *HESS*, *MAGIC* or *VERITAS*, but ASTRI MA aims to achieve a great level of sensitivity in the almost unexplored energy range of 1-300 TeV. We

[4]ASTRI-MA website: http://astri.me.oa-brera.inaf.it/en/

Figure 1.6: A graphical representation of the ASTRI Mini-Array Telescope system at the Teide Astronomical Observatory, on Mount Teide in Tenerife (Canary Islands, Spain). Picture from [1]

can see the graphical representation of an air shower in Fig. 1.7.
This technique requires an array of optical telescopes with a diameter of 4 meters and must be located at an altitude greater than 2000 m; the telescopes will be installed at the Teide Astronomical Observatory, on Mount Teide ($\sim$ 2400 m a.s.l.) in Tenerife (Canary Islands, Spain). Fig. 1.8 is shown a top view of the observing site. To ensure efficient and safe operations, the ASTRI MA must be remotely operated, with no human presence on-site during observations. A data center will be located on-site to provide a quick analysis and quality checks of the data, while the main ASTRI Data Center in Rome will provide data archiving, processing, simulation, and science user support. The high-speed networking connection at Teide will allow for seamless delivery of data to the main data center, reducing the need for storage devices on-site. [39]
The telescopes will have reflecting mirrors focusing the Cherenkov UV-optical light produced by atmospheric particle cascades (air-showers), initiated by the primary gamma-ray photons entering the atmosphere, onto cameras with a very fast response. Most of the collected data will come from a large number of charged primary CRs that initiated air-showers, which constitute a noise to be tackled. ASTRI-MA is also designed for performing Stellar Intensity Interferometry. Each telescope will be

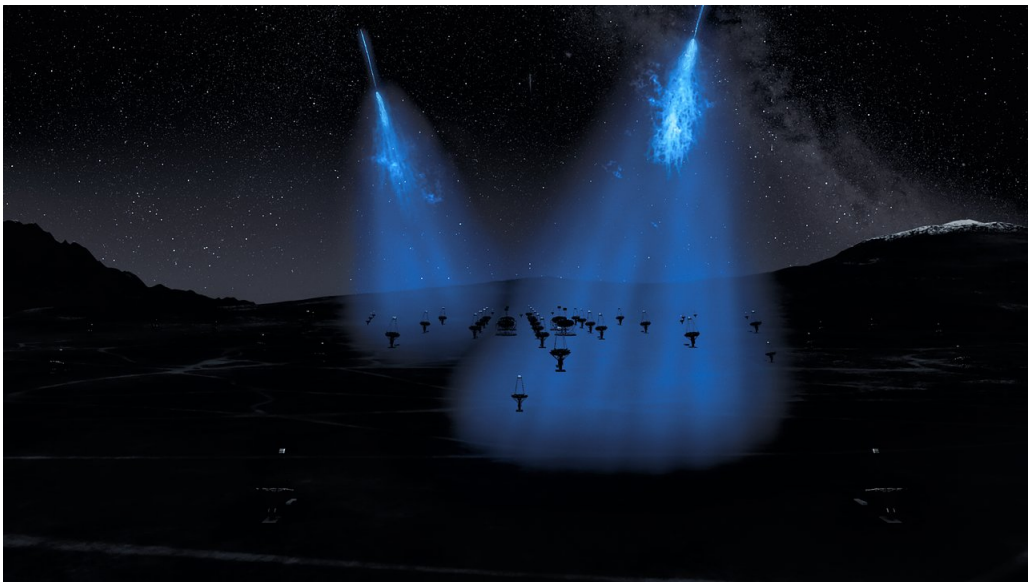


Figure 1.7: A representation of an air-shower during the night. It is an elusive phenomenon that lasts only a few nanoseconds. Picture from CTA's website (3)

equipped with an Intensity Interferometry module. By exploiting the ASTRI layout, it is possible to obtain details of the surface of bright stars and their surroundings. The background recorded during Cherenkov light observations will be also analyzed to provide direct measurements of the CRs.

We are interested in describing some scientific aspects of the *VHE Cherenkov observations* with ASTRI-MA. These can be summarized with :

- Arrival of the air shower: an IACT is enlightened by Cherenkov light. Light is reflected and focused onto a multi-pixel ultra-fast electronic camera. Several signals at the detector plane then arise and the image is acquired if these are above a certain threshold provided by the acquisition algorithm type.

- The camera acquires the shape of the image produced by the shower. The projected air shower on the detector plane shows an elliptical shape, representing the longitudinal and lateral shower development, with its major axis pointing toward the nominal source projected onto the camera.

- Simultaneous observations: several IACTs record the shower at the same time: this means that a stereoscopic observation is possible. IACTs must operate as a single system, generating a trigger event only when more than two telescopes are above a certain threshold. then, only the telescope interested in the event will acquire data.

- Camera images are then stored for subsequent scientific analysis.

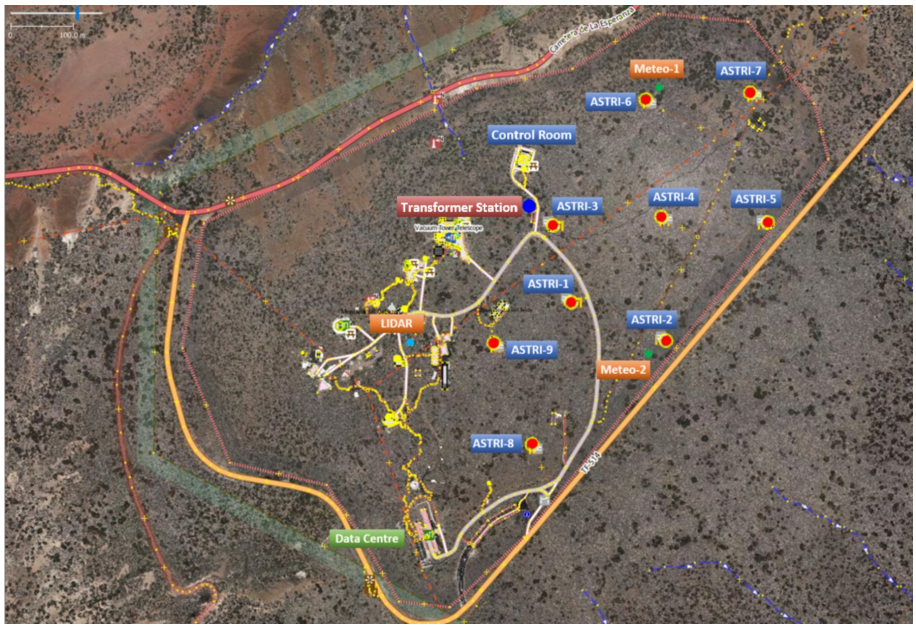


Figure 1.8: The final layout for the ASTRI mini-array Observing site. The figure shows the final positions for the 9 telescopes, the position of the two meteorological towers, the LIDAR, the transformer station and the local control room, and the data center. Picture from [1]

- Cherenkov light is emitted also by *charged cosmic rays*. By exploiting certain processing algorithms, we can determine whether the event collected by a telescope was induced by a CR or by a gamma ray. Thanks to the *stereoscopic observation*, we can improve our reconstruction capabilities, and obtain the impact point of the cascade, in addition to the height at which it reached the maximum, and also the incoming direction of the primary particle that produced the shower. This information highly improves the energy reconstruction and the background event rejection (charged CRs) of the 99%.

The stereoscopic technique is why the IACTs array has success and represents the future in ground-based gamma rays observations [27].

A Telescope is composed of the mechanical elements indicated in Fig. 1.10. Other interesting and innovative features of the ASTRI Mini-Array telescopes are:

- the *Dual Mirror Schwarzschild-Couder* configuration. The primary mirror M1 is composed of 18 aspheric panels with different curvatures, and M2 is an aspheric monolithic panel. Mirror panels are mounted on active supports that allow a fine calibration of the orientation of each segment[38].
  This optical system is not affected by spherical or comatic aberrations of traditional designs. Moreover, the second mirror de-magnifies the image, improving the field of view.

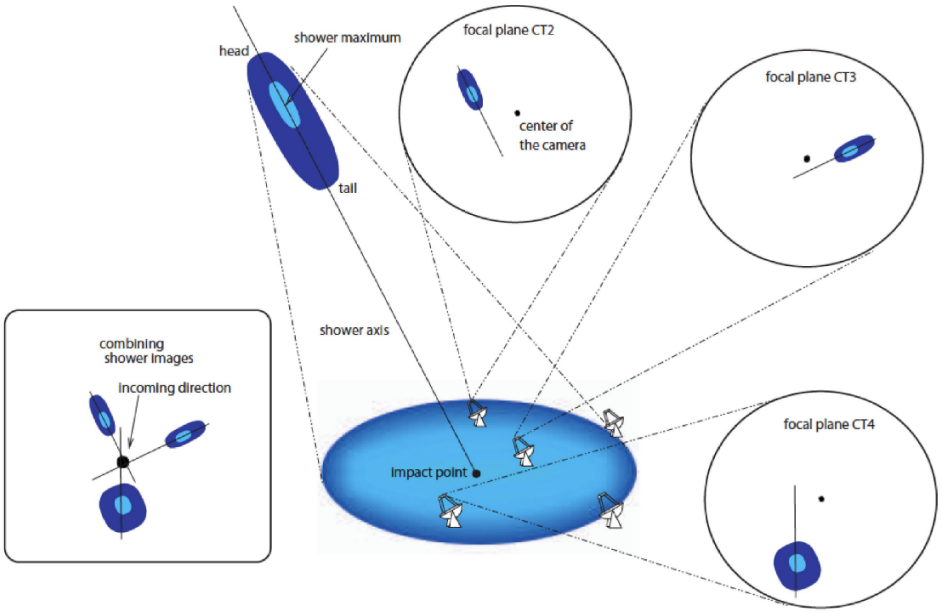


Figure 1.9: Schematic of the measuring principle of stereoscopic observations. The simultaneous observation of different viewing angles of the Cherenkov light emitted by the air shower produces different images that are used to improve the reconstruction of the incoming direction of the atmospheric shower and, eventually, the sensitivity of the whole system [40]. Picture from [27]

- the *new silicon photon-multipliers, SiPM*. SiPMs are excellent sensors and compared to photon-multiplier tubes they show an improved single photon resolution, a higher detection efficiency, low bias voltage, and no damage when exposed to ambient light;

- the *ASTRICAM Silicon photo-multiplier Cherenkov Camera*. It is composed of 37 Photon Detection Modules, PDMs, a mechanical unit containing SiPMs, and an electronic control unit. Each PDM is a flat array of 8 by 8 logical pixels. The camera covers a 9.6 deg full field of view [41].

According to preliminary Monte Carlo simulations, the sensitivity of a point source for nine telescopes is higher than that of HESS above 10 TeV, up to 100 TeV. The SST array planned for CTA south will make better in sensitivity terms, but the energetic range 1-300 TeV is covered as well. See the graph in Fig. 1.11, which represents the minimum detectable energy flux with an observation of 50 h of the ASTRI Mini-Array versus H.E.S.S and CTA-South observatory (the lower is better). The ASTRI mini-array will be able to study in great detail relatively bright (a flux of few$\times 10^{-12}\, erg\, cm^{-2} s^{-1}$ at 10 TeV) sources with an angular resolution of a few

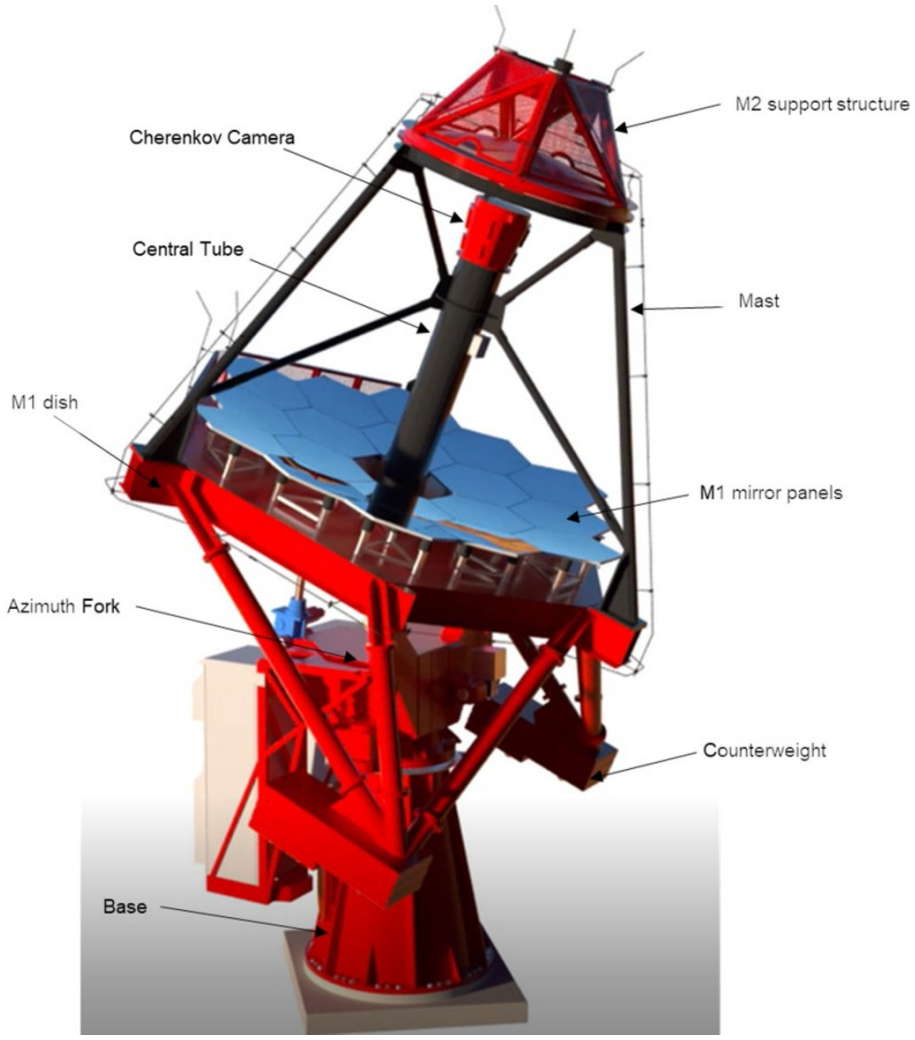


Figure 1.10: A 3D model of the ASTRI telescope, with component nomenclature, picture from [1]

arcmins[5] and an energy resolution of about 10–15 % [27].

The observations of the telescope are decided by a *night observing schedule*. A Cherenkov observation is composed by a series of successive short observations called *Observing Blocks*, which consist in pointing the source of interest or a region of points around it. The duration of such Observing Block will have typical duration of 30 min, and it must be carefully designed to avoid predictable image noises sources, for example, the presence of the moon or planets in the field of view. Moreover, the observing schedule can be changed during the an active observation, if an external scientific alert[6] is received from other observatories[42]. The ASTRI-MA telescope,

[5] *arcsec*: unit of measure used to quantify angles in the sky. An arcsecond is defined as 1/3600 th of a degree and a 1/60th of an *arcmin*. The use of arcseconds in astronomy is essential for measuring the apparent sizes and positions of celestial objects in the sky.

[6] A science alert is an urgent communication from one observatory to other observatories that a relevant astrophysical event is occurring in the sky

due to the presence of a high-performance camera, can produce a consistent amount of data. To continue the observations correctly, the telescope must perform an **online quick look analisys** on the acquired data to determine data quality status information at the telescope level and check-pointing precision and accuracy using some useful acquired data [43].
In the following chapter, we will focus more on the ASTRI-MA software architecture and quick-look analysis aspects.

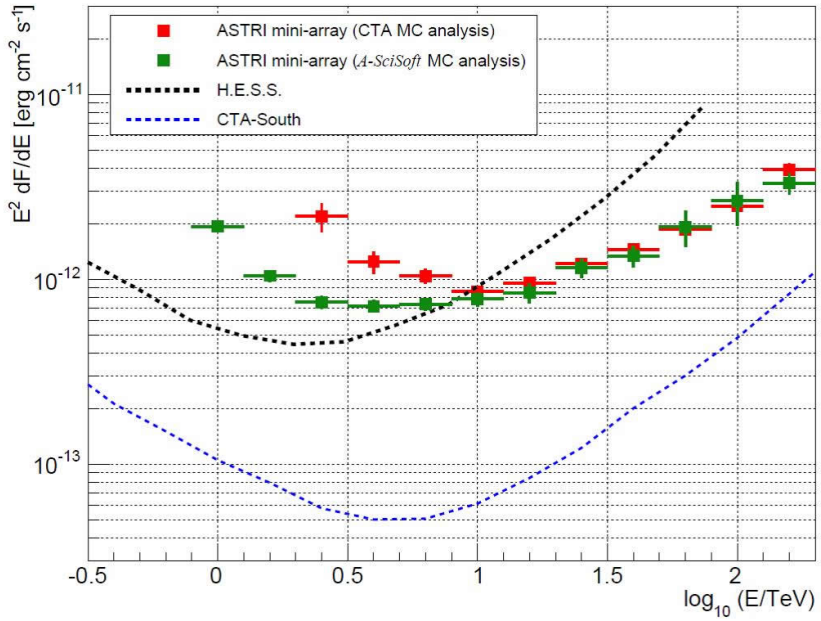


Figure 1.11: Preliminary results for the differential sensitivity of the ASTRI mini-array to point sources (5σ, 50 h, 5 bin/dex). Picture from ASTRI MA's website (4).

Chapter 2

Online Observation Quality System

ASTRI MA Software Systems (*MASS*) is developed in a variety of software components. **MASS** consists of two geographically distinct parts: the *on-site* one, running at the ASTRI Observing Site at Teide, and the *off-site* part running at the ASTRI Data Center in Rome. The on-site software is called *Supervisory Control And Data Acquisition*, **SCADA**), and it controls all the operations carried out at the observing site, like data acquisition, control and monitoring the telescopes, and handling alarms.
Our pipeline application constitutes an implementation view of a component of the **Online Observation Quality System** (**OOQS**). OOQS is a SCADA subsystem that aims to execute quality checks in real-time on the data acquired by the Cherenkov cameras and intensity interferometry instruments.
In this chapter, we provide a general view of all the SCADA sub-systems which interface with OOQS, and all the OOQS internal components involved in processing the Cherenkov camera data. This is essential for describing the pipeline's systems interconnections and behavior in the context of Cherenkov data processing.

The architectural approach used by ASTRI Team is the 4+1 view architectural model, which is the visualization of the system from different viewpoints with the support of the UML[1] diagrams (see Fig. 2.1):

- the *use case view* describes the system's interaction with entities developing use cases. A use case is a list of actions or event steps that describes how an actor and a system interact to accomplish a task. The actor could be a person or another piece of hardware or software;

[1] https://www.uml.org/

- the *functional* or *logical view*, is a system decomposition in functional blocks, describing the global information flow based on the analysis of use cases and data models;
- The dynamic aspect of the system is addressed by the *process view*.
- the *implementation* or *development view* is a detailed design of the implemented system;
- the *physical* or *deployment view* deals with the system engineer's point of view. The physical view is more concerned with the system's physical layer and connection, while the deployment view deals with the allocation of computing resources on physical nodes.

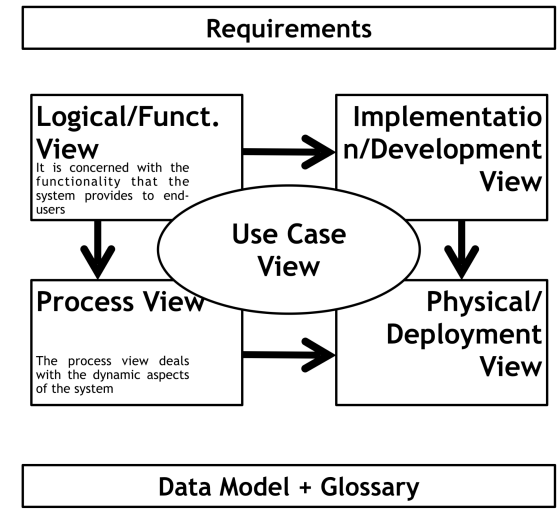


Figure 2.1: Illustration of the 4+1 Architectural View Model with requirements, data model, and glossary to complement the information. Picture from [39]

The *use case*, *logical/functional*, and *process views* are used to define the scope and the main functions of the software system.
To simplify the understanding of the ASTRI MASS, a definition of the top-level requirements is provided in *"ASTRI Mini-Array Top Level Use Cases*" [43], with a common glossary and high-level definition of the ASTRI data model. The use cases, coupled with the functional view, provide a complete description of the *functional* requirements of the software. The functional decomposition described in the *"ASTRI Mini-Array Top Level Software Architecture"* [39] has been used to develop the whole product breakdown structure of the software system, used to manage interfaces and to define the specification tree, i.e. the definition of the hierarchical

relationship of all technical aspects of the software system and is the basic structure to perform requirements traceability.
Then, concerning each subsystem, an initial definition of the requirements that the subsystem must satisfy has been discussed and it is provided in the Use Case Documents, and a specific functional view is provided.
The subsystems of interest for our context are a subset of the ASTRI Mini-Array Software main systems:

- **SCADA System**: this software controls all the operations carried out at the MA-Observatory.
- **Archive System**: it serves as the central storage for all essential and permanent information related to the ASTRI Mini-Array. This includes Observing Projects, observation plans, raw and processed scientific data, device monitoring data, past, present, and future MA system configurations, and records of all operations and schedules. The main archive of interest is the **Quality Archive**, which stores the Cherenkov and intensity interferometry observation quality checks during the observation.

2.1 SCADA Context

SCADA is responsible for controlling and monitoring the observing site system, site service system, and safety and security system installed at the Array Observing Site. All operations at the Array Observing Site are under the control of the SCADA system. SCADA has a Central Control System, which interfaces and communicates with all hardware and software installed at the site. It is responsible for the execution of the telescope observations.
SCADA is usually supervised by the operator but performs the operations in an automated way. It provides scientific data, logging, monitoring, alarms, and online observation quality information to help assess data quality during the acquisition.
This system supports the day and night *Observation Execution* and maintenance phases. The main functions we are interested in are:

- **Online Observation Quality System**: Receive and analyzes the Cherenkov camera packets, and focus on ongoing problems and the status of the observations. This is the principal subsystem we are interested in describing and developing in this thesis project.
- **Array Data Acquisition System**: acquires Cherenkov Camera data and

Stellar Intensity Interferometry Instruments. It provides the input data to the OOQS;

- **Central Control System**: is responsible for coordinating and managing the various subsystems involved in telescope operations. It receives, verifies, and executes the observation plans to control the telescopes and other subsystems. The Central Control System is also supported by the *control systems* and *collectors*. The control systems are responsible for controlling, monitoring, and managing the status of various subsystems such as telescopes, atmosphere characterization assemblies, calibration systems, and alarms. The collector subsystems are responsible for monitoring and determining the alarms and status of environmental devices, the Information and Communication Technology (ICT), and the power system.

- **Logging System**, **Monitoring System** and **Alarm System** monitor the overall performance of the systems through the acquisition of environmental, monitoring, and logging points and alarms from instruments and generates status reports or notifications to the Operator;

- the **Operator Human Machine Interface** (**HMI**), which serves as the user interface for the Operator. The HMI includes an Operator Logbook, which allows the operator to save logs of the observations made during the night.

2.2 Cherenkov Camera Context

The ASTRI Cherenkov camera [26] is composed of electronics designed to detect Cherenkov signals, generate triggers, perform digital conversion of the signals, and transmit the data to the camera server. A Cherenkov camera is composed of 2 main parts:

- *Cherenkov camera focal plane*: it is composed of 37 *Photon Detection Modules*, **PDMs**. Each of them consists of an 8×8 pixel matrix of SiPM sensors, a Front End Electronics board, and a Field Programmable Gate Array board (FPGA).

- *Back End Electronics* (BEE): it is a powerful FPGA, and is the main elaboration unit of the camera interfacing the 37 PDMs. It is responsible for the creation of several data packets based on the acquired images in real time.

A PDM is a complete fast signal processing chain to acquire, condition, perform a digital conversion, and dispatch to the BEE each SiPM's pixel signal. Each SiPM

pulse is adapted in terms of amplitude, and generates a trigger signal that must be managed by the PDM.
The PDM's processing chain is then composed of three separate sub-chains to produce the *high gain*, *low gain*, and *trigger* outputs. Each SiPM pulse is then amplified utilizing two different conditioning chains, producing high gain and low gain output signals. These channels differ in the pre-amplification level of the signals and are set to obtain two different dynamic ranges of photo-electrons. High gains and low gains outputs are stored in 64×2 analog memories. Moreover, a dedicated acquisition chain can produce a trigger output signal based on a low gain or a high gain amplified SiPM pulse and compare it with the appropriate threshold. There are 64 digital trigger outputs for each PDM to realize the trigger signal.
When a given number of contiguous pixels within a PDM will have a signal above a given photo-electron threshold, we have a so-called ASTRI **camera trigger**. The camera trigger algorithm presented is a *topological* type. When the camera triggers, analog outputs are read and then converted one by one in digital values from an Analog to Digital Converter (ADC), producing the *high gains* and *low gains ADC values*.

ADAS [44] is the system responsible for the data acquisition from the camera, through a component called "Camera Data Acquisition". BEE can produce data in .raw format called R0, which includes different types of packets. the R0 format consists of a binary stream of a fixed size, that cannot be interpreted without specific software. The ADAS is in charge of the R0 pre-processing, producing packets in a format called DL0, in which data are translated into alphanumeric quantities ready to be processed.
The packets are of fixed size and can be classified as *notification*, *scientific*, and *periodic*. The packet structure contains specific fields to recognize the packet type to manage it properly. Each packet type is identified by the "Packet Type" and "Packet SubType" [45].
For our purposes, we need to know that the packet is composed of the Packet Header and the Packet Data Field.
The *Packet Header* contains the following pieces of information:

- *telescopeID*: an integer from 1 to 9 identifying which of the telescopes generated the packet.
- *Packet Type* and *Packet SubType*.
- *Source Sequence Counter* (SSC): this field contains an incremental value that counts the number of packets generated by the BEE camera for each type and

subtype.

- *TimeTag*: the time tag refers to the acquisition time of the first sampling for periodic packets, and the time of the trigger camera for scientific packets.

The *Packet Data Field* is specific for each packet type and subtype and can be of the following types:

- **Scientific Packet S(2,2)**; this kind of scientific packet contains the camera events. R0 packet has a size of about 13 KB, while the DL0 has a dimension of 16384 Bytes due to the field conversion into 64-bit computer data models. These packets contain the pulse heights in the function of the ADC channel sampling that occurred at the skylight pulses. The maximum event rate for each camera is $f_{S22,\,max} = 1000\,Hz$, and the DL0 data rate is about 16 MB/s for each telescope, and about 148 MB/s considering the nine telescopes. In the Packet Data Field of these scientific packets, we can find:
  - *Data Field Header Extension*: contains additional information about the data acquisition, such as:
    - *Pixel trigger discriminator threshold* and *Pixel trigger lower level discriminator threshold* (in photo electron units);
    - *Trigger type*: it is the PDM trigger algorithm, where 0 means topological, and 1 means majority;
    - *Trigger configuration*: defines the neighborhood number if the trigger type is 0, otherwise, it defines the number of the majority if the trigger type is 1;
  - *Source Data Field*: it contains an array of 37 fields, each one for a specific PDM of the camera. A PDM data field is composed as follows:
    - *pdmVal*: it is equal to 1 if the PDM is enabled and running, otherwise it is equal to 0;
    - *PDM trigger enable*: it is 1 if it is enabled, otherwise it is equal to 0;
    - *triggered PDM*: it is 1 if it is true, otherwise it is equal to 0;
    - *pdmID* number: is a number between 1 and 37 indicating the specific PDM;
    - *high gain pixel*: it is an array of 64 elements containing the ADC data related to the signal of the high gain pixel acquisition chain;
    - *low gain pixel*: it is an array of 64 elements containing the ADC data related to the signal of the low gain pixel acquisition chain;

In the Listing 3.1, we can see the S(2,2) structure defined in our C++ code.

- Two **Variance data packets**: it is a kind of periodic packet containing information about the camera status and the observation context. They have a size of 9.5 KB, while the event rate is between 1 to 10 Hz; thus they generate packets at a low rate. Packets are produced by randomly sampling the pixels' values without a trigger. Values are packed separately for the high-gain (**VAR(10,2)**) and low-gain (**VAR(10,3)**) chains and they share the same data structure.

2.3 OOQS use cases and requirements

The purpose of the OOQS is to perform data quality analysis during the telescope observations and provide quick look results to the Operator and other SCADA-subsystems. OOQS must then perform real-time analysis, and in case of abnormal conditions, it provides feedback to other subsystems to take corrective actions. Moreover, it notifies the Operator if the data acquired by the telescopes is not consistent with the quality requirements [39]. Fig. 2.2 shows the UML diagram representing the system context of the OOQS, and we provide an explanation of the OOQS interfaces with the other subsystems:

- *Central Control System*: controls start and stop of the OOQS operation; moreover, OOQS sends to the Central Control System the telemetry data to notify *failures* detected by the quality checks.

- *ADAS*: sends to the OOQS the raw data (DL0) acquired during Cherenkov or intensity interferometry;

- *Alarm System*: OOQS sends to the Alarm System the telemetry data to notify *anomalies* detected by the quality checks. When the alarm system receives a notification from OOQS, it evaluates the possibility of sending an alarm to the HMI, in such a way that the operator can take an action towards, for example stopping the telescope array.

- *Monitoring System*: OOQS sends to the Monitoring System the information about the status of the OOQS software components.

OOQS can perform quality checks on the data acquired during Cherenkov Observations and the Stellar Intensity Interferometry Instrument (SI^3) observations. We are interested in describing the operative mode of OOQS during Cherenkov observations. According to the OOQS use cases described in *"ASTRI Mini-Array Online*

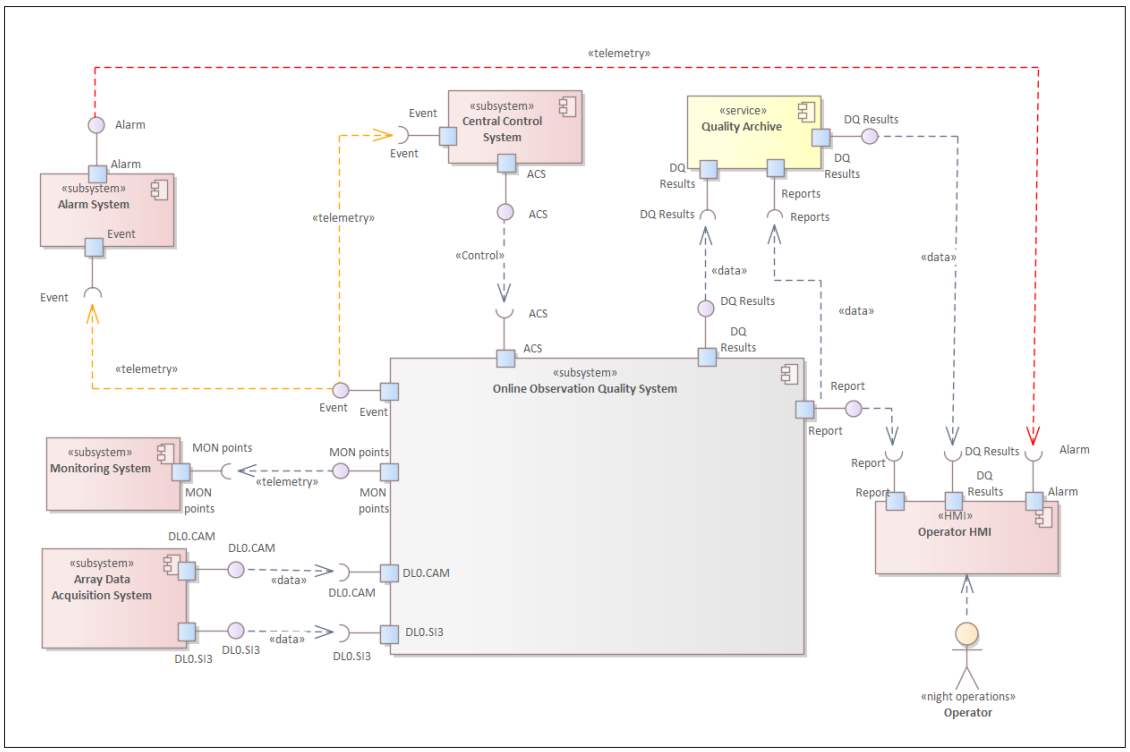


Figure 2.2: The system context diagram of the OOQS. Picture taken from [46]

Observation Quality System Use Cases" [47], we can identify a list of components that compose the OOQS:

- *OOQS Master*: the component that manages the OOQS components' life-cycle;
- *OOQS Manager*: it is the central component of the system, responsible for managing the execution of quality checks during both Cherenkov and Intensity Interferometry Observations. This component is connected to the central control system, alarm system, and monitoring system;
- *Cherenkov Camera Data Quality Checker* (**CCDQC**): performs the analysis for Cherenkov observations and it is the main component we are interested in developing;
- *SI*^{3} *Data Quality Checker* (*SI3DQC*): performs the analysis for intensity interferometry observations;
- **Quality Archive**: the results of the OOQS are stored in the Quality Archive, available in the on-site data center. The Quality Archive data are also summarized in *reports* available to the Operator through the HMI, for further investigation.

OOQS's components must satisfy their functional and performance requirements. During the OOQS verification and validation process, the quality assurance team will check that all the requirements will be satisfied.

OOQS Master and OOQS Manager are *Alma Common Software* (*ACS*) components [48]. ACS is a framework useful to develop distributed software: it collects common patterns and implements them in ACS components. It is compatible with several programming languages such as C++, Python, and Java.

OOQS Master controls the life cycle of the OOQS, and it is interfaced with the Central Control System. The Central Control System can send OOQS requests to start or stop all the OOQS components. Additionally, it monitors the OOQS component status and manages the exception that can occur during component execution. There exists a total of nine OOQS Masters, one for each telescope.
OOQS Manager supervises the software that executes the quality checks, and we can find nine instances also for this component. An OOQS Master controls an OOQS Manager, and each of them can instantiate a CCDQC, or a SI3DQC component, based on the observation type.
Quality checks are then performed individually on each telescope-acquired data. OOQS Manager it has an interface with the alarm system and with the central control system, which are used when an abnormal condition is detected. Additionally, it has also an interface with the monitoring system, so that it can communicate the status of the software components.

CCDQC is the main component that we developed for this thesis project. We can see a diagram of its functionalities in Fig. 2.3. The CCDQC has two software components: the *Science Data Quality Checker* and the *Variance Evaluator*. These two components analyze the data packets generated by the Cherenkov camera: the scientific S(2,2), and the two variance data packets, VAR(10,2) and VAR(10,3). The ADAS sends these data through different *Kafka* topics. Kafka is a publish/subscribe event streaming platform, to store and process events in a highly scalable and fault-tolerant manner; we describe it in Section 3.3.1. How data is stored in Kafka follows a data serialization model called *Avro*. Avro is an open-source data serialization system that guarantees excellent performance for this application; we describe it in Section 3.3.2. ADAS takes the DL0 and Avro serializes it, finally, a Kafka producer, one for each kind of data [49], inserts it as a payload in a Kafka message on a Kafka topic, completing the delivery.
The OOQS implements the same number of consumers to receive and deserialize the data packets, to perform the analysis.
The major data throughput that OOQS must handle is given by S(2,2) packet, which

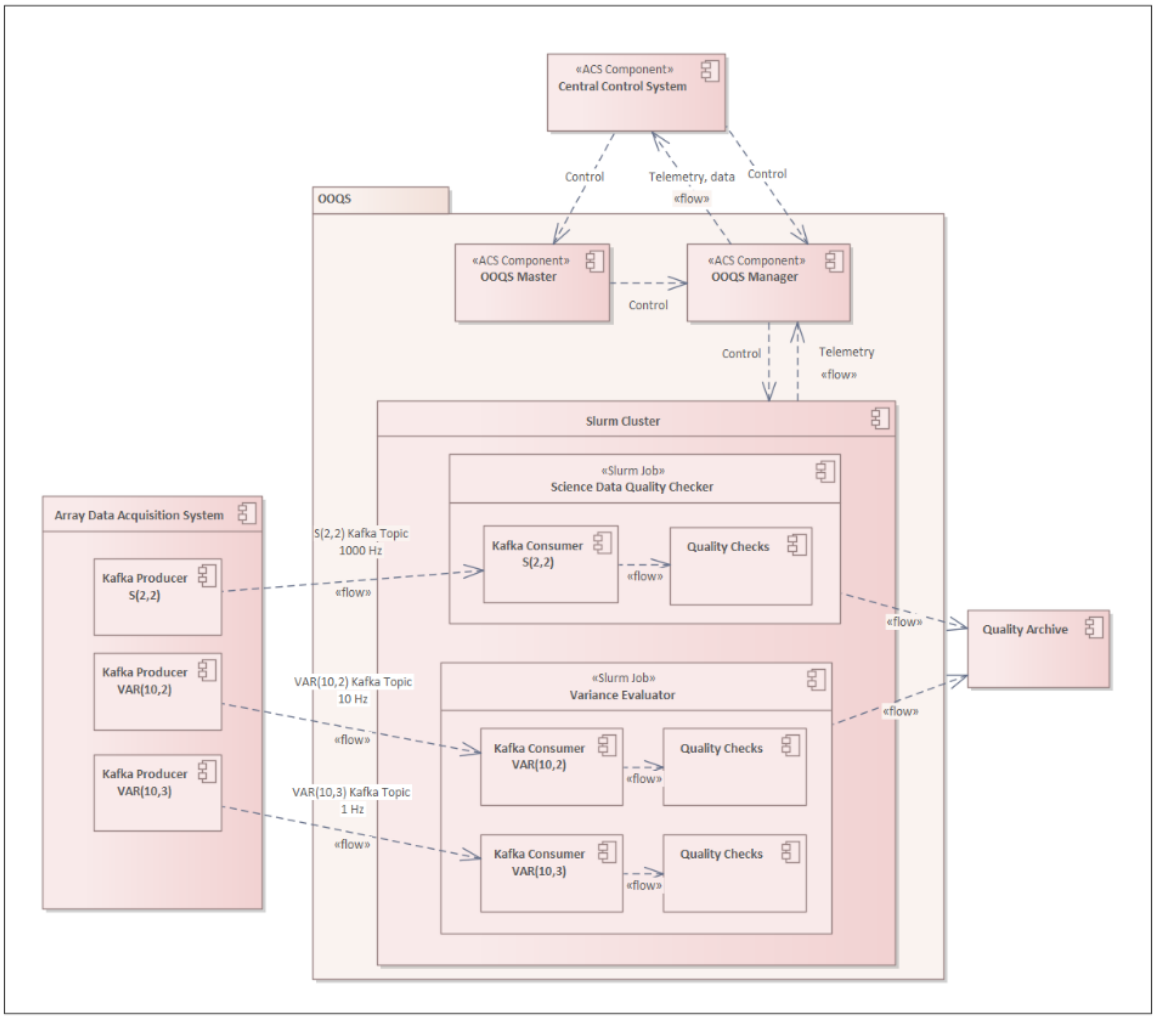


Figure 2.3: Illustration of the Cherenkov Camera Data Quality Checker component and its interfaces. Central Control System instantiates the component when a Cherenkov observation is requested. Picture from [46].

is characterized by the highest data rate acquisition among all the Cherenkov camera packets. CCDQC performance, following the requirements described in Section 2.2, must satisfy the acquisition of 1000 S(2,2) packets in a second (about 16 MiB/s). The VAR packet has a size of about 9.5 kB, but the event rate is between 1 and 10 Hz; thus, they do not constitute a challenging data rate.
The main data quality checks performed on the S(2,2) packets are:

1. calculate the histograms of the trigger number for each camera and each camera's PDM;

2. calculates the histogram of the times between two consecutive triggers for each camera;

3. check that the histogram values are inside a predefined range;

4. check that the pixel ADC values of each Cherenkov camera are inside a predefined range;

5. sample the data to obtain one camera image per second.

6. execute checks 1, 2, and 4 with the calibrated data obtained with predefined calibration coefficients;

If abnormal conditions are found during the data quality checks, the Cherenkov Camera Data Quality Checker sends a notification to the target subsystems described above and prepares a report that is saved into the Quality Archive. The failures are sent to the Central Control System.
The main data quality checks performed on the VAR(10,2) and VAR(10,3) are:

1. aggregate all the VAR data from the start of the observation for each camera;

2. calculate the ratio between the high-gain and low-gain of each camera PDM;

3. check if the pointing deviation and the point spread function (PSF) size are inside the nominal range;

4. sample the data to obtain one camera image per second.

The main purpose of the VAR analysis is to check if a pointing correction is necessary during observations. For this purpose, a specific technique has been developed for ASTRI [50], and the possible corrections can be applied in real-time or during the following observing run [46]. In the next chapter, we will further provide explanations of the implementation of the OOQS pipeline, and the developed framework to receive data from Kafka, perform analysis, and store results.

Chapter 3

OOQS-Pipeline development

3.1 Programming Languages

3.1.1 C++

The **C++** programming language[51] was created by Bjarne Stroustrup and his team at Bell Laboratories (AT&T, USA) to aid in the efficient and object-oriented implementation of simulation projects.
The earliest versions of C++, which date to 1980, were originally referred to as "C with classes". As its name suggests, this programming language was derived from C, in which the increment operator is represented by ++.

C++ is a hybrid programming language that combines object-oriented language with functionalities belonging to the C language, such as:

- *Universally usable modular programs*: modular and reusable code can save time and reduce errors when developing complex applications.
- *Efficiency*; C++ programs are compiled into machine code, which can be executed directly by computers' processors, resulting in shorter execution times with respect to interpreted languages.
- *Portable programs for various platforms.*

C++ includes various concepts of object-oriented programming which are:

- *Data abstraction*: C++ supports the creation and manipulation of objects that contain both data and the functions that operate on that data.
- *Data encapsulation*: aims at controlling access to objects, C++ provides also low-level control over system resources, such as memory allocation and hardware access.

- *Inheritance*: create classes able to inherit characteristics of parent classes
- *Polymorphism*: implementation of instructions that can have varying effects during program execution[52]

Unlikely other programming languages like Python and Java, C++ lacks built-in memory management tools. This requires the programmer to manually manage memory allocation and deallocation which can be challenging as well as prone to possible errors.

3.1.2 Python

Python is a high-level, general-purpose programming language that was created in the late 1980s by Guido van Rossum. Since then, it has become one of the most widely-used programming languages in the world, thanks to its simplicity, versatility, and ease of use [53].

Python is an interpreted language, which means that it doesn't need to be compiled before it can be run. This can make it more convenient for developers who want to write and test their code quickly. Python's interpreter can be used interactively, which is a useful feature for testing short snippets of code or exploring the language's features.
Another key characteristic of Python is its strong emphasis on code reuse and modularity. Python provides a wide range of built-in modules, which can be used to perform a variety of tasks, such as file I/O, networking, and regular expressions. Python also supports the creation of user-defined modules, which can be shared and reused in other programs. This makes it easier to write complex programs and maintain large codebases.

Python, like C++, is an object-oriented programming language, which means that it provides support for object-oriented programming concepts, such as encapsulation, inheritance, and polymorphism. Python is also a dynamically-typed language, which means that variable types are determined at runtime, rather than at compile-time. This can make it easier to write and modify code since variables can be used without being explicitly defined.
Python has a wide range of uses, such as:

- Web Development
- Data Science and Machine Learning
- Automation

- Scientific Computing
- Game Development

3.1.3 C++ vs Python

C++ and Python are two popular programming languages with distinct features and advantages. C++ is a compiled language that is known for its performance, while Python is an interpreted language that is popular for its simplicity and ease of use. In terms of syntax, C++ is a statically typed language that requires explicit variable declarations, while Python is dynamically typed and allows for more flexible variable declarations. C++ is typically used for system programming, game development, and other high-performance applications, while Python is often used for data science, machine learning, and web development.
Overall, C++ is a powerful language that provides low-level control over computer resources, while Python is a user-friendly language that allows developers to quickly and easily prototype ideas and test concepts.

3.2 Frameworks

3.2.1 DQ-Pipe

The DQ-Analysis and DQ-Aggregator pipeline applications are based on the *DQ-Pipe*, a Python framework developed by INAF. DQ-Pipe is a general framework to build automatic pipeline analyses, providing a variety of different objects to define the job that each single process constituting the pipeline must accomplish, the input and output entities. DQ-Pipe implements an object to define a trigger mechanism to perform a new job, one of the possibilities is based on the detection of new files in the specified file system paths. DQ-Pipe can work with *Slurm*, a workload manager able to optimize the load on the OOQS server cluster. An XML configuration file specifies the operations that an instance of the DQ-Pipe must perform. We extended the functionalities of the DQ-Pipe so that it can be a suitable coordinator for our applications.

In Fig. 3.1 is shown an XML file template that describes a typical configuration for an OOQS pipeline, it contains the following elements:

- the output directory of the logging files,
- the parameters of a MongoDB connection (or MySQL),

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rta_dq_pipe_config>

    <logging dir="$RTADQPIPE_TEST_DATADIR/logs_dir" level="DEBUG" />

    <dqpipelines>

        <dqpipeline type="dq_analysis_ASTRI" id="camera_batch" dqchain_id="camera_batch_analisys"
                nthreads="0" mode="slurm" >
            <input_data  type="h5" input_dirs="$CONSUMER_BATCH_OUTPUT" file_pattern="*" reading_string=""
                join_with="" join_keys="" filter_column="" filter_value=""/>
            <output_data type="h5CXX" output_loc="$DQ_CXX_OUT" overwrite=""  suffix="" />
            <process_call process_dir="$DQ_CXX_AN" process_name="/DQ-Analysis.lnx" />
        </dqpipeline>

        <dqpipeline type="dq_aggregator_ASTRI" id="camera_aggregator" dqchain_id="camera_pipe_aggregation"
                nthreads="0" mode="slurm">
            <input_data  type="h5" input_dirs="$DQ_CXX_OUT" file_pattern="*" reading_string=""
                join_with="" join_keys="" filter_column="" filter_value=""/>
            <output_data type="mongodb" output_loc="dq_batch_results" overwrite="" suffix=""/>
            <process_call process_dir="" process_name="" />
        </dqpipeline>

    </dqpipelines>

    <databases>
        <database type="[mongodb]||[mysql]" hostname="[IP]" port="[TCP-PORT]"
                username="[USER]" password="[PASSWORD]" database="[QUALITYARCHIVE]"> </database >
    </databases>
</rta_dq_pipe_config>
```

Figure 3.1: the conf.xml configuration file for the OOQS pipeline. It creates a dqpipeline process for a DQ-Analysis application and one for the DQ-Aggregator application. The interfaced database is MongoDB.

- a *dqpipeline process* including:
  - the input data directory where the application read the HDF5 files,
  - the output data directory,
  - the external process directory and filename,
  - id field for the log filename,
  - mode field which specifies if processes are managed via Slurm or not,
  - the application, or algorithm, selected to process the data.

By creating multiple *dqpipeline* processes, it becomes possible to establish a data processing workflow, comprising a sequence of processes executing various successive operations on data. To enable this mechanism, it is essential to interconnect two *dqpipeline* processes in such a way that the input directory of the second *dqpipeline* corresponds with the output directory of the first *dqpipeline*.

In Fig. 3.2 are described the components involved in the creation of a *dqpipeline* process. The entry point of the pipeline is DQPipelineController, it instances a DQPipeBuilder object using the method start. DQPipeBuilder reads an XML configuration file and builds one or more pipelines defined as described in the file.
An instance of a DQPipeline requires an input "*dataSource*", that defines the incoming data to be processed and an "*outputHandler*" defines where to store the

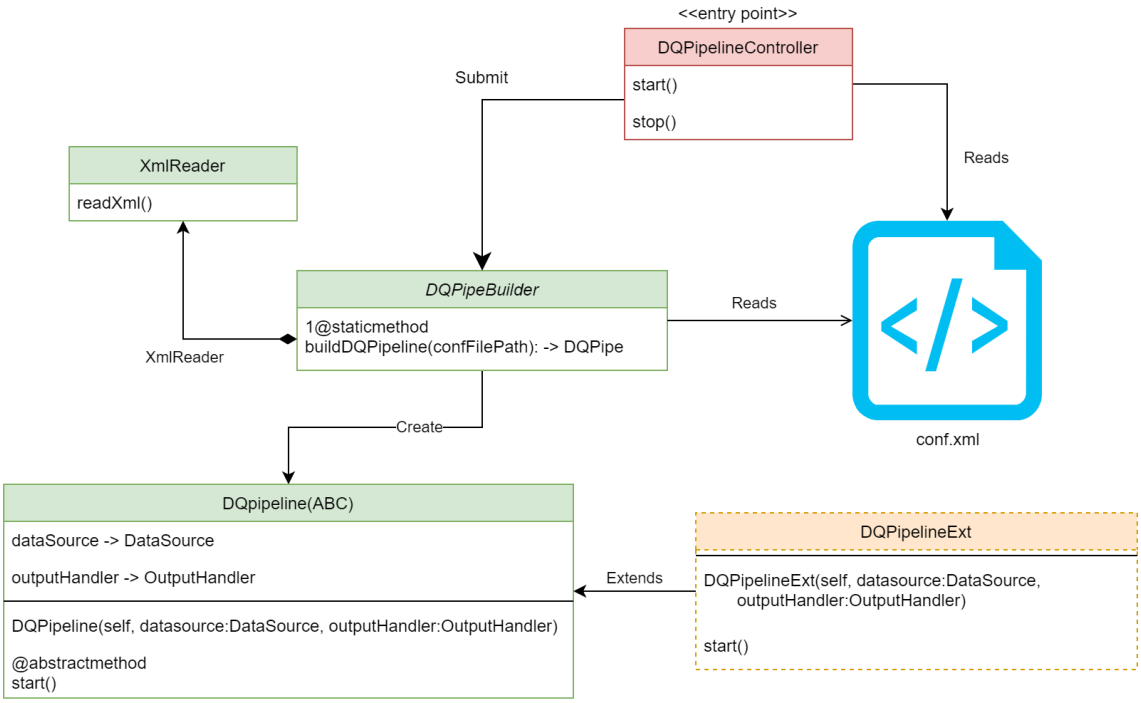


Figure 3.2: The class diagrams describing the relationships of the entities involved in the instantiation of a DQ-Pipeline Component

results of the elaborated data. Thanks to DQ-Pipe flexibility, the DQPipeline class can be extended to perform the desired user-defined tasks. The Fig. 3.3 shows the class structure of DataSource for handling files/datatypes. DataSource is an object connected to the dataSource member of a DQ-Pipeline instance.
FileSystemDS implements a *watchdog* library [1] that notifies when a new file is created on a specific directory. The watchdog events are specialized by FileSystemDS class, providing some useful synchronization tools to start the dqpipeline's job when some specific files, such as the **Ok files**, are detected. The Ok file is an empty file that ends with a ".ok" extension, and carries the same name of another file that a dqpipeline instance must elaborate. Further explanations on these file are present in our application development, in Sec. 3.5.1 and 3.5.2. FileHandler is an abstract class, describing the interface to manage different kind of user defined files. For instance, the read() method will describe the in an eventual instanced FileHandlerExt. The Fig. 3.4 shows the class structure of the OutputHandler. OutputHandler is an interface connected to the outputHandler member of a DQ-Pipeline instance. The output handler can be extended in a variety of ways, for instance to write data in a specified format via a FileHandler implementation, can connect to a database where to write a data via an OutputToDB implementation, or simply provides a file system

[1] Watchdog API: https://python-watchdog.readthedocs.io/en/stable/

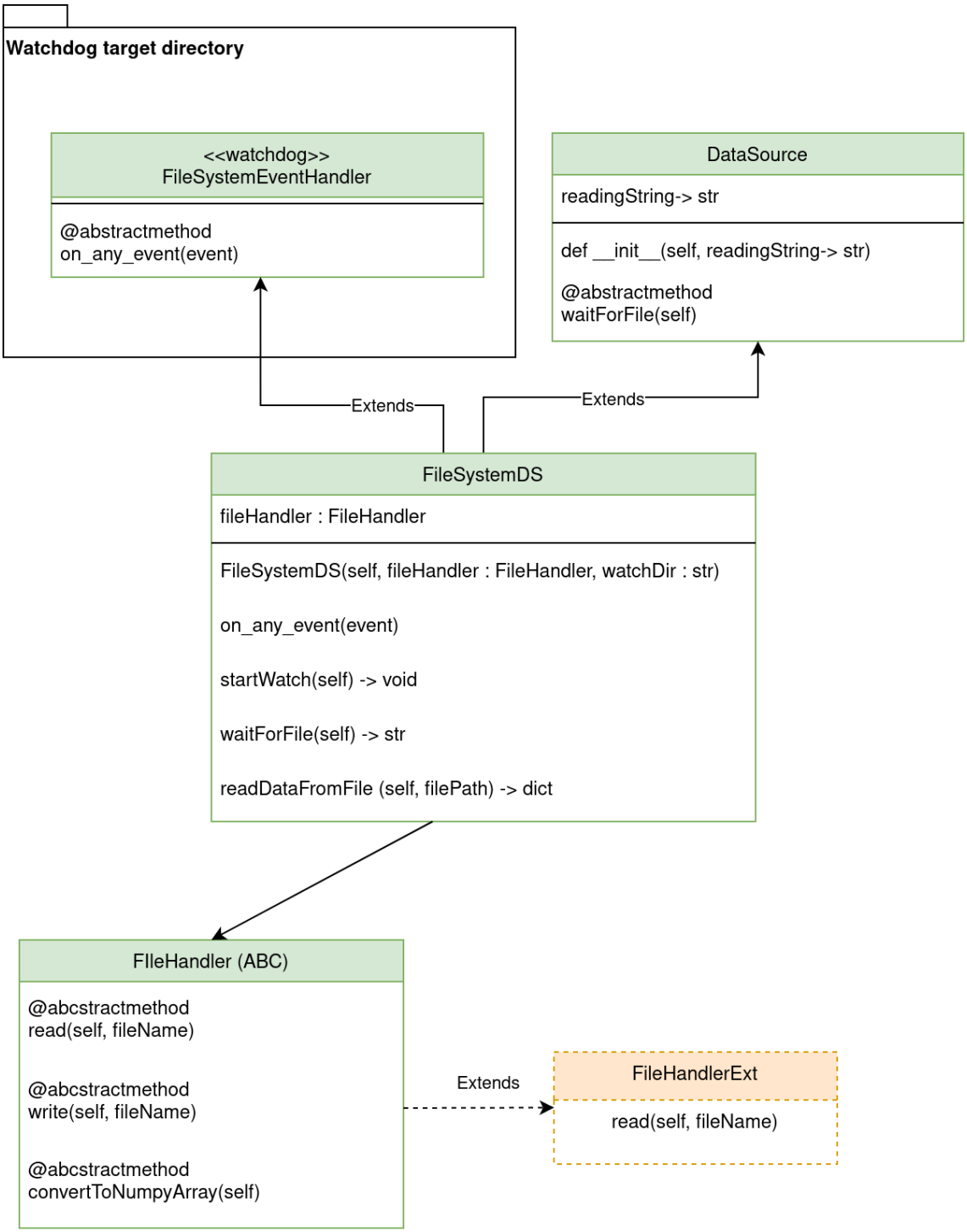


Figure 3.3: The class diagrams describing the characterization of the DataSource class

path where to write files with an OutputToFileSystem. The specific file format or path is specified by the user in the configuration file. In Fig. 3.5 are shown classes used for utility purposes:

- PipeloggerConfig is a logger implemented as Singleton;
- RepeatThread creates a concurrent thread task that executes periodically;

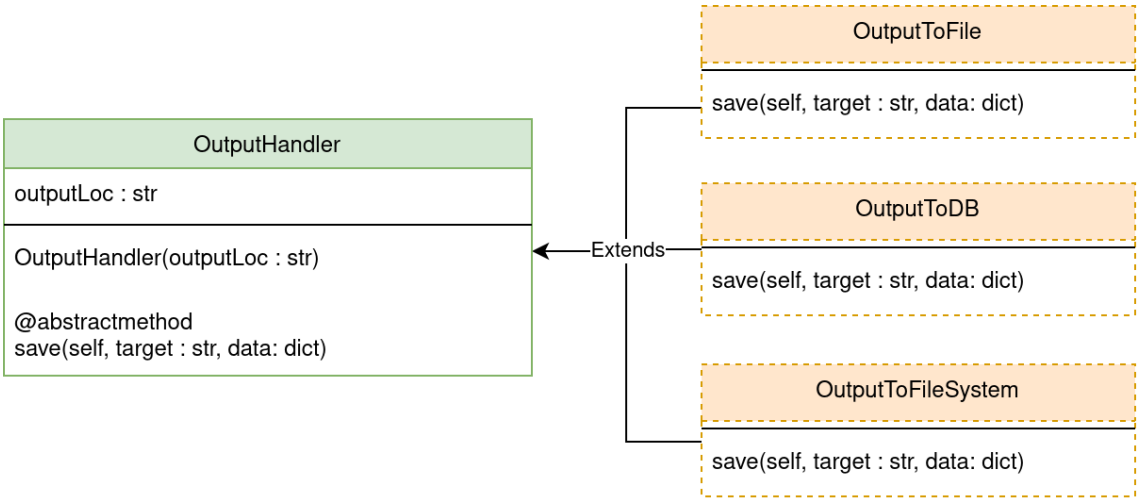


Figure 3.4: The class diagrams describing the characterization of the OutputHandler class

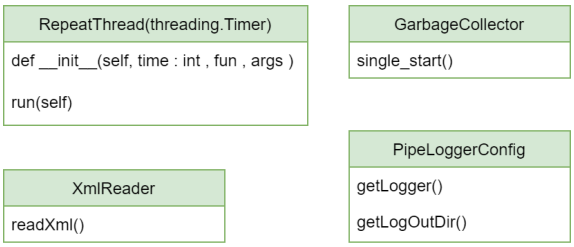


Figure 3.5: The DQ-Pipe utility classes

- GarbageCollector deletes data files that are already processed and not required;
- XmlReader for configuration file parsing.

3.2.2 Slurm

Slurm[2] is a job scheduler and resource manager used in high-performance computing environments. It is designed to efficiently allocate and manage resources, including computing nodes, memory, and other hardware resources, to enable the execution of large-scale scientific computations and data analysis tasks [54].
Slurm supports a wide range of job types, including parallel jobs that span multiple nodes, interactive jobs for user interaction with the computing environment, and array jobs that execute a large number of similar tasks. It also includes advanced features such as checkpointing, accounting and reporting, and workload management.
Slurm uses a hierarchical architecture that enables administrators to manage resources at multiple levels, including the cluster level, node level, and job level. This

[2]https://www.schedmd.com/index.php

allows for fine-grained control over resource allocation and scheduling, ensuring that jobs are executed in a timely and efficient manner [55].

At the Array Observing Site, there are multiple servers composing the on-site Computing Cluster. The OOQS instances, with all their software components, will be deployed as Docker containers in the Computing Cluster, along with other subsystems such as Monitoring System, Alarm System, Archive, etc.[46].
Distributing the workload among all these heterogeneous jobs is an important aspect of the implementation of our pipeline software solution that can be tacked with Slurm.
If DQ-Pipe is configured to work with Slurm, all the processes, can be scheduled as Slurm jobs in the Computing Cluster. To achieve this, the *DQPipelineController* will create *sbatch* file that will submit the *DQPipeBuilder* on Slurm. The sbatch script must specify all the dependencies, such as all the file system paths to execute the pipeline itself and the external processes that it must handle.

3.3 Tools/libraries

3.3.1 Apache Kafka

Before discussing the implementation of the Kafka Consumer, let us present briefly how Kafka works, the main problems that it can address, and the potential of Kafka in our project.

Apache Kafka is a *publish/subscribe messaging system* having a single centralized system that allows for publishing generic types of data. It is often described as a "*distributed streaming platform*". A file system or database commit log is designed to provide a record of all transactions so that they can be replayed to consistently build the state of a system. Data within Kafka is stored durably and ordered so that it can be read deterministically.

The unit of data within Kafka is called "*message*": it is an array of bytes that does not have a specific format or meaning inside Kafka. A message can have a key, which is a byte array of metadata, to manage the message's writing to different partitions.
Messages in Kafka are categorized into "*topics*", which are analogous to folders inside the file system. Topics can be split into several *partitions*, where each partition corresponds to a single log in the commit log structure of Kafka. Messages are written in Kafka in an append-only fashion and are read in order from the beginning to the end, but the reading in temporal order can be guaranteed only in the case

of a single partition topic. On the other hand, partitions allow Kafka to be simply scalable and provide redundancy in the system.

"Kafka clients" are users of the system and can be of two basic types:

- **Producers** create new messages, which in general will be produced on a specific topic. A producer can specify in which partition a message must be written or it can balance messages in all the partitions of a topic.

- **Consumers** read the messages created by producers. A consumer can subscribe to one or more topics and will read the messages in their production order. A consumer keeps track of the messages that have already been consumed via an "*offset*" of the messages, which is an incremental integer metadata value. Kafka adds to each message a unique offset in a given partition, in such a way that the consumer can start or stop its operations without losing track of the last consumed message. Consumers working on a certain topic must belong to the same "*consumer group*" to make sure that each partition is only consumed by one member at a time. We can manage large topics, by scaling horizontally the consumers. Moreover, if a consumer fails, the remaining members of the group will re-balance the partitions to take over the missing member.

Stream processing with Kafka

Kafka is suitable for several use cases, such as activity tracking, messaging, metrics, logging, commit log, and **Stream processing**.
A data stream is an abstraction representing an infinite and ever-growing data set. The data set is unbounded because, over time, new records keep arriving. Moreover, event streams are an ordered set of immutable data records, and it is desirable to be able to replay the whole, or at least a part, of the event stream. This is required to correct errors, try new methods of analysis, or perform audits [56].

Our Kafka application

Our application is a stream processing application with soft real-time requirements. It must be able to read non-stop, for a theoretically infinite time, the unread messages on the topic or wait for new messages, identifying transaction errors on the messages, and committing the offset to the server broker.
There are three Kafka topics, relating respectively to the Camera scientific packets S(2,2) and the Variance packets VAR(10,2) and VAR(10,3). Each CCDQC will have three Kafka Consumers, each of them assigned to a Kafka topic, which contains the

messages including a scientific or a variance packet produced by a specific telescope. In other words, there will be nine different topics for each kind of data, for a total of 27 topics.
Up to now, we developed the consumer for a single telescope, that can receive from a single partition topic relative to the scientific data of that telescope [56].

C++ Kafka API

The C++ API for implementing a Kafka consumer is called cppkafka, which is a C++ wrapper from Mattias Fontanini[3], built on top of librdkafka, the official C library for the Apache Kafka protocol.
We are interested in an example of a basic consumer program, available in the API documentation which makes use of the following tools:

- A *Configuration* object allows easily configuring your consumers/producers. A configurator typically stores information on how to connect to the Kafka server, like IP address and the TCP-IP port, which commit mode, and how many messages download together from the topic. We are also interested in defining some *callbacks* to have extra feedback about the connection status with a topic.

- A *TopicConfiguration* object: we must insert the topic on which the message will be consumed and the offset configuration.

- The *Consumer* object, which provides us the methods to consume messages, subscribe to the topics, commit consumed messages, and so on.

Further information on the configuration of Kafka can be found on the librdkafka's documentation[4].

3.3.2 Apache Avro

Kafka producer configuration includes mandatory serializers for several datatypes, such as strings, integers, and byte arrays, but we would require to serialize more complex datatypes records.
Apache **Avro**[5] is a language-neutral data serialization format created by Doug Cutting to provide a way to share data files with a large audience. *Avro serializer* is a recommended alternative to manage the Cherenkov Camera packets due to their

[3]CppKafka Docs: https://github.com/mfontanini/cppkafka
[4]librdkafka Docs: https://docs.confluent.io/platform/current/clients/librdkafka/html/index.html
[5]https://www.baeldung.com/java-apache-avro

rich data structure, using a schema to perform serialization and deserialization. The Avro schema is usually specified in JSON, and the serialization is usually to binary files, although a JSON serialization can be handled [56].
Avro supports two types of data:

- *Primitive* type: Avro supports all the primitive types. We use a primitive type name to define a type of a given field. For example, a value that holds a String should be declared as "type": "string" in Schema.
- *Complex* type: Avro supports six kinds of complex types: records, enums, arrays, maps, unions and fixed.

There are mainly four attributes for a given Avro schema:

- *Type*: describes the type of Schema that can be can be either primitive or complex type.
- *Namespace*: describes the namespace to which the given schema belongs.
- *Name*: the name of the Schema.
- *Fields*: tells the fields associated with a given schema. Fields can be either primitive or complex type.

In Avro terms, a *writer* performs the data serialization, namely it encodes data, while a *reader* performs data deserialization, so it decodes data.

Our Avro application

In our application, the producer encodes data in a binary format. Binary encoding does not include field names, self-contained information about the types of individual bytes, or field or record separators leading to smaller serialized data. On the other hand, a reader always wholly relies on the schema used to encode.
To Deserialize these data in a C++ application, we can rely on the AVRO C++ API[6].

Between the several functionalities that AVRO C++ provides, we focus on the followings:

- *Code generator*: avrogencpp is a one-time use C script that, starting from a .json file representing the desired AVRO schema, creates a C++ struct, so a language-dependent representation of the schema and the functions to encode and decode those structs. In the Listing 3.1 we can see the struct defining the S(2,2) scientific package.

[6]https://avro.apache.org/docs/1.11.1/api/cpp/html/

- *Custom binary decoder*: decodes the binary Avro data and is used at runtime.

Listing 3.1: The S(2,2) DL0 packet (Avro generated). int32_t is a typedef of a standard C++ signed integer

```
namespace S22 {
enum triggerTypeEnum {
    TOPOLOGICO,
    MAJORITY,
};

if (true)
struct PDMBlock {
    bool pdmVal;
    bool triggerEnabled;
    bool triggered;
    int32_t pdmID;
    std::vector<int32_t > highgains;
    std::vector<int32_t > lowgains;
    PDMBlock() :
        //struct init
};

struct S22 {
    int32_t telescopeID;
    int32_t type;
    int32_t subType;
    int32_t ssc;
    int32_t year;
    int32_t month;
    int32_t day;
    int32_t hours;
    int32_t minutes;
    int32_t seconds;
    bool validTime;
    int32_t timeTagNanosec;
    int32_t pixelTriggerDiscriminatorThreshold_pe;
    int32_t pixelTriggerLowelDiscriminatorThreshold_pe;
    triggerTypeEnum triggerType;
    int32_t triggerConfig;
    std::vector<PDMBlock > PDMs;
    S22() :
        //struct init
};
[...]
}
```

3.3.3 HDF5 Library

HDF5 is a high-performance data management and storage suite[7]. HDF5 supports n-dimensional datasets and each element in the dataset may itself be a complex object. It is useful in general for high-performance I/O operations and runs on several computational platforms, providing API for different programming languages.
HDF5 comes with:

- a *file format* for storing HDF5 data,
- a *data model* able to logically organize and access the HDF5 data from an application
- the *software*, such as libraries, language interfaces, and tools, for working with this format.

The HDF5 data model contains the building blocks to perform data organization and specification in HDF5 files. An HDF5 file, which is an object itself, is a sort of container that holds a variety of heterogeneous data objects or datasets. The datasets can be constituted from a great variety of elements such as images, tables, graphs, or even documents, such as PDF or Excel files.
The two primary structures that we find in an HDF5 file are *groups* and *datasets*: groups are entities similar to folders, while datasets include data and metadata, aiming at describing the data itself. HDF5 are then organized in a file system fashion, having a root group that can contain other groups or datasets.
A variety of objects are available to describe datasets, like **datatypes**, **dataspaces**, and **properties**. **Datatypes** describe the individual data elements in a dataset. Of particular interest for our application are:

- *Native Datatypes*: the native datatype of the plaftorm.
- *Derived Datatypes*: these are data types that are created or derived from the pre-defined datatypes. An example of a commonly used derived datatype is a string of more than one character. Nested compound datatypes are also derived types, of which we can see an example in Fig: 3.6.

Dataspaces consists of the layout of the data elements in the dataset. They define the dimensionality of the dataset through the rank number and the size for each

[7]The HDF5 website: https://www.hdfgroup.org/

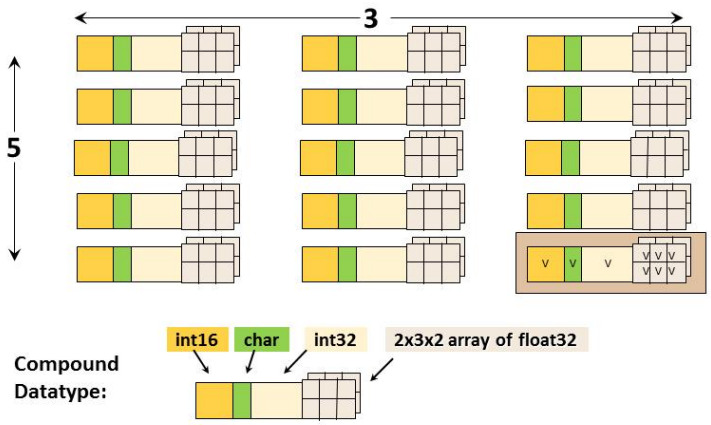


Figure 3.6: Example of a *compound dataset*. The compound datatype is a structured data, composed of native datatypes such as a 16-bit integer, a char, a 32-bit integer, and a $2 \times 3 \times 2$ array of 32-bit float. Image from HDF5-website (7)

dimension which can be either fixed or unlimited. Dataspace has a dual role in HDF5: it represents the spatial information (logical layout) of a dataset stored in a file, and it describes an application's data buffers and data elements participating in I/O. For instance, we can define a dataset as a matrix of size 2×3 by setting a dataspace Rank $= 2$ and Dimension $= (2, 3)$. Each element of the matrix will store data coherent with the dataset's datatype.
The **property** we are interested in for our purpose are *contiguous* and *chunked* dataset: contiguous means that data are stored physically adjacent to each other, while chunked means that data are divided into subsets. The advantage of chunked datasets is the increased access time speed for I/O operations on parts of the dataset.

HDF5 C++/Python APIs

To share data between the applications we developed for the OOQS-Pipeline, we used the HDF5 library, creating writing and reading functions for two different files. To manage the HDF5 files in the C++ applications, we used the HDF5 C++ wrapper, that simplifies the operations with respect to the HDF5 C library[8]. Instead, to manage the HDF5 files in the Python applications, we used PyTables[9], which is built on top of the HDF5 library. It supports NumPy[10] operations and provide an object-oriented interface with several critical parts using C extensions. Datasets in

[8]HDF5 APIs:https://docs.hdfgroup.org/hdf5/develop/index.html
[9]PyTables' API: https://www.pytables.org/
[10]NumPy (short for "Numerical Python") is a Python library that is widely used for scientific computing and data analysis. It provides a powerful N-dimensional array object, along with many other useful functions for working with arrays, matrices, and numerical data.

PyTables are treated as NumPy arrays, while groups can be accessed as dictionaries. Some functionalities we are interested in while using HDF5 in C++ are the followings:

- *file constructor*: we must set an appropriate name for the target data file and create or open it via the appropriate file constructor;
- *group constructor*: we must define several groups which will hold our datasets;
- *dataspace constructor*: it will define the dimension and shape of the datasets in the file and the memory;
- *datatype constructor*: it will define which kind of data will be inserted in the dataset. Datatypes can be simple, such as integers 32-bit or double (float 64-bit) or they can be H5 datatypes such as arrays of native datatypes, or **compounds**;
- *compound datatypes constructor*: in C++, we must define each member of our compound datatype. Starting from a C struct, we must define each component of the struct as a member of the datatype specifying a name, a size of the data, and a datatype for each component;
- *dataset constructor*: it defines the dataset, based on the specified dataset and datatype. It is a method for a file or group node to create the dataset in the appropriate file position;
- *methods for writing or reading datasets*: to perform an I/O operation, we must specify the buffer containing data be to written or to be read, the native datatype of the dataset, the dataspace instance for both memory and file dataset. The buffer size must be greater or equal to the dataset dimension. Moreover, we can define an "hyperslab" for the dataspace, so that we can write/read just a part of the dataset;
- methods for releasing the allocated resources for the opened file, and also for the datasets and dataspaces opened.

3.3.4 GSL Library

The *GNU Scientific Library* (*GSL*) is a numerical library for C and C++[11]. The library provides a wide range of mathematical routines such as random number generators, special functions, and least-squares fitting. There are over 1000 functions

[11] https://www.gnu.org/software/gsl/doc/html/usage.html

in total with an extensive test suite.
We are interested, in developing some quality checks exploiting the library's efficiency. Up to now, we have only been required to implement a histogram handler. In the OOQS quality checks context, histograms are useful to understand the ADC values distribution of the camera pixels; ADC values are described in the Sec. 2.2. OOQS will raise an anomaly if ADC values are outside of a known range, or a defined distribution.
In the scientific analysis context, Camera usually acquires night sky images and, as discussed in the introductory chapter, cosmic rays initiated air-showers disturb the acquisition of gamma ray initiated air-showers. Checking the histogram bins levels during observation, we could perform a background extraction, which is useful for further preliminary elaborations such as image binarization, aiming at detecting bright sources in the sky. A histogram is defined by the following struct:

- size_t n: the number of histogram bins (columns);
- double * range: it is a pointer to an array of n+1 elements containing the ranges of the histogram bins;
- double * bin: the number of elements per bin is stored in an array of n elements pointed by bin. The bins are floating-point numbers, so you can increment them by non-integer values if necessary.

Histograms provide a convenient way of summarizing the distribution of a set of data. The i-th bin is defined between two range elements, so that

$$\text{bin}[i] \;=\; \left\{ z \in \mathbb{R} \;:\; \text{range}[i] \leq \; x \; < \text{range}[i+1] \right\} \quad \forall\, i = 1,\, 2,\, \ldots,\, \text{n}$$

Let X be a random variable that maps a sample space Ω in real values.

$$X : \Omega \Rightarrow \mathbb{R}$$

Bins are a collection of the realization of a random variable $x \;=\; X(w)$ for some $w \in \Omega$, such that:

$$\text{bin}_{\text{count}}[i] \;=\; \#x \in \text{bin}[i] \quad \forall\, i = 1,\, 2,\, \ldots,\, \text{n}$$

To use histograms in a GSL application, we must define the number of bins and the bin range array. We are interested in several functions to operate with histograms:

- histogram allocation: a function that takes the number of bins n, and histogram struct h and allocates the right memory to h with a malloc() operation.

Array range in the histogram h will be allocated with $(\mathrm{n}+1) \times$ sizeof(double) bits and number of elements per bin with $n \times$ sizeof(double) bits;

- set uniform range: a function that, given a histogram h, and the min and max extremes of the linear range, builds the range array linearly spaced;
- histogram increment: it assigns to the correct bin of the histogram h, every new value x;
- histogram add: given two similar histograms $h1$ and $h2$, it performs a bin-wise addition saving the resulting number of counts in $h1$;
- functions able to free histogram h memory space.

3.4 Storage Systems

A database is a collection of information organized in a way that can be easily accessible, managed and updated. In order to implement the insertion into a storage system of the data quality results provided by the OOQS-Pipeline, an integration of a database is essential. In addition, it can be interesting to compare the performance of relational and non-relational database when testing them with OOQS data.

3.4.1 MySQL

MySQL is a popular open-source Relational DataBase Management System (RDBMS) that is widely used in web applications and software development. It provides a robust and reliable platform for storing, organizing, and retrieving data using a Structured Query Language (SQL). MySQL fully satisfies *ACID*[12] requirements for transaction-safe RDBMS: *atomicity* is handled by storing modified rows in a memory buffer and writing to disk after the transaction commits. *consistency* is ensured by logging mechanisms and locking mechanisms, while server-side semaphore variables and locking mechanisms manage *isolation* mechanisms. *Durability* is maintained by a binary transaction log file and recovery from hardware failures is relatively straightforward by using backups in combination with the log. MySQL is also high scalable, and ease of use, making it a preferred choice for small to large-scale applications. It offers various features such as transactional processing, data security, and replication capabilities, making it suitable for enterprise-level applications as

[12]ACID refers to the four key properties of a transaction: atomicity, consistency, isolation, and durability.

well. MySQL also offers a vast range of tools and integrations, making it easy to manage and maintain databases.

In our application, due to the variety of different analysis and data, a flexible object like a JSON can be used to store all the quality results. One advantage of using JSON in a MySQL database is that it allows for more flexible schema design. Python applications can store JSON encoded dictionaries into a MySQL table, that can be fetched as a string or as a JSON data.
However, there are also some disadvantages to using JSON in a MySQL database. One potential issue is that queries that rely heavily on JSON functions can be slower than traditional SQL queries. Additionally, it can be harder to enforce data integrity and perform complex queries on data stored in JSON format.

3.4.2 MongoDB

MongoDB is a popular NoSQL database management system that uses a document-oriented data model to store and manage data. Unlike traditional relational databases, MongoDB is designed to be flexible and scalable, making it a great choice for modern applications that require dynamic and rapidly-changing data structures. MongoDB's document model allows for nested structures and dynamic schema, meaning data can be stored without a defined structure and can evolve over time without compromising performance. MongoDB's distributed architecture allows it to handle large volumes of data and support high levels of concurrency. Additionally, MongoDB provides a variety of features such as replication, sharding, and indexing, making it a powerful tool for building and scaling modern applications.

In the context of our application, MongoDB is able to manage Python dictionaries natively with its flexible document data model, so that a comparison with MySQL is interesting from the performance point of view. MongoDB uses a flexible document data model based on the BSON format (Binary JSON), which is very similar to JSON but supports additional data types like binary data and date objects.
Using BSON in a MongoDB database can provide several advantages over using plain JSON. For example, BSON is more compact than JSON, which can lead to smaller database sizes and faster read and write performance.

3.5 OOQS-Pipeline Architecture

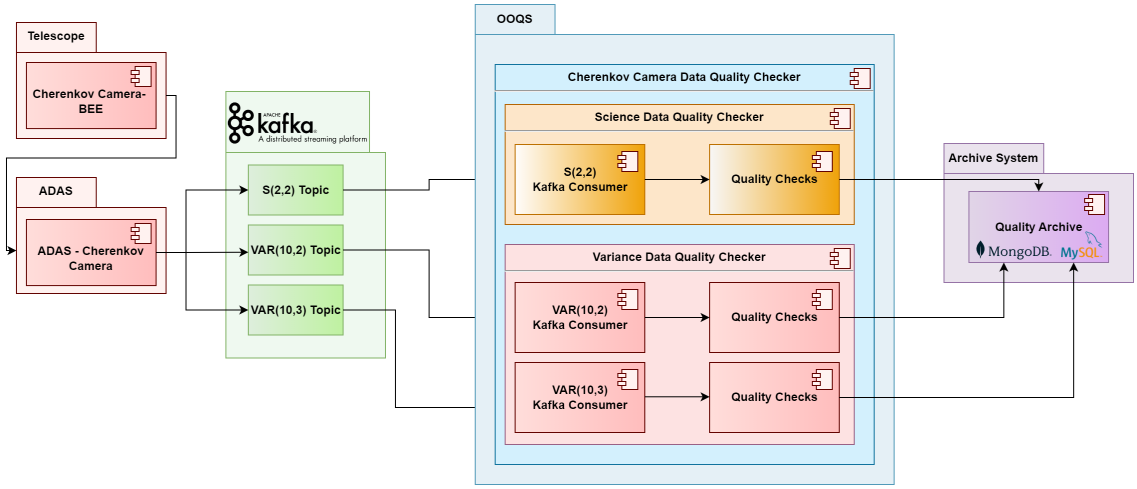


Figure 3.7: The OOQS system diagram, and the Mini-Array Software subsystems interconnections of interest in the OOQS-Pipeline development.

The component diagram in Fig 3.7 shows the ASTRI Mini-Array components that are meaningful for the application developed in this thesis. In this project, we focus on the acquisition Chain of the **Cherenkov Camera packets** from a single Cherenkov Camera. Cherenkov Camera has its BEE devices that produce several kinds of data packets [45]. ADAS is in charge of acquiring these packets through the Cherenkov Camera Data Acquisition module [46]. Subsequently, the Communication Step Between ADAS and OOQS takes place.

The Cherenkov Camera Data Quality Checker is the main topic of this thesis. CCDQC, as discussed in the OOQS chapter, aims at performing quality checks on the acquired Cherenkov camera images in near real-time with the observations.
To implement it, a data quality pipeline has been developed and tested. A data quality pipeline is software able to perform a sequence of elaboration processes: it takes some input data, elaborates it, and returns output data, which is the input for the next process step.

The OOQS pipeline is defined by some components according to the CCDQC structure. The data flow across components is built employing HDF5 files containing groups of data, or data results. In the OOQS quality checks, we are not only interested in analyzing a single camera event, but it is also important to visualize a certain interval of time or events or the history of the current observation. The pipeline is composed by three different *deamons*[13] applications:

[13] A daemon is a computer program that runs continuously in the background, performing various tasks as required. Daemons are often used to provide services or functionality to other programs or users, such as handling network requests, managing print jobs, or monitoring system resources.

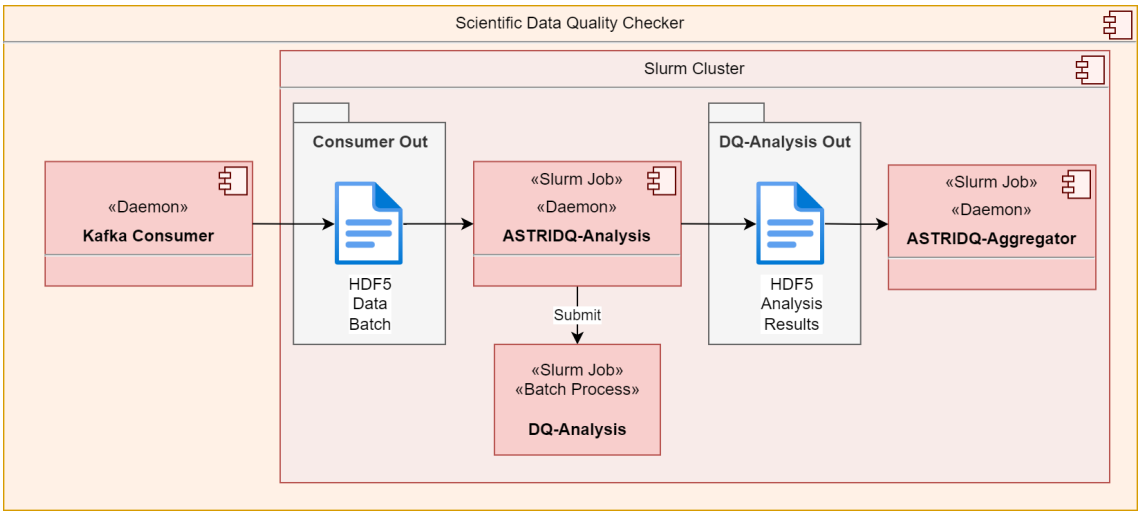


Figure 3.8: The Scientific Data Quality Checker Pipeline

- **Kafka Consumer**: it collects data from a given Kafka topic, and it writes an HDF5 file containing the data acquired;
- **Data Quality Analysis**: it performs quality checks on a certain group of data; It submits an *external batch process*[14] every time new data are available. This process can be a Slurm job if the pipeline is executing on a Slurm Cluster, or a bash process on the host machine that is currently executing the DQ-Analysis application;
- **Data Quality Aggregator**: it performs data aggregation on the results of the quality checks to reach the target level of data aggregation requested, then it stores the aggregated results in a database of the Quality Archive.

To synchronize the execution of the Analysis and the Aggregation, the DQ-Pipe framework has been exploited. The DQ-Pipe instantiates several pipelines in parallel, with the possibility of distributing the workload in the Slurm Cluster. This framework can be configured using XML files with several types of analyses. This framework holds in general for both the CCDQC sub-components: the **Scientific Data Quality Checker** (**SDQC**) and the **Variance Evaluator**; in Fig. 3.8 is shown the implementation of the OOQS pipeline for the SDQC component.
Given the high speed and the high amount of scientific data S(2,2), the development of the SDQC component is more relevant from the point of view of compliance with real-time constraints. However, due to the amount and variety of quality checks

[14]A batch process is a program that executes a series of tasks or jobs in a specific sequence, without user intervention. These tasks may involve processing large amounts of data, generating reports, or performing calculations. Batch processes are typically scheduled to run at a specific time or when certain conditions are met, such as the availability of resources.

that the two components must perform, only some use cases of the SDQC have been developed and tested. We note also that in order to acquire and process data from the whole telescope array, then a set of nine OOQS-Pipeline working with each Camera acquisition chain is then required.

3.5.1 Kafka Consumer

We start now by describing the first building block of the OOQS pipeline.
Kafka consumer is a C++ application that manages the acquisition of the Cherenkov Packets S(2,2), VAR(10,2), and VAR(10,3) sent by ADAS through Kafka. For each type of packet, an Avro schema has been created, describing the fields of the original packet required by the OOQS. The Avro schema is created through files in JSON format. For each data, a Kafka topic is created.
We can describe Kafka Consumer providing a functional view. ADAS, in the Kafka context, represents the producer side and it has the following features:

- A packet is processed by the appropriate processor component, based on the data type.
- The fields of interest are extracted and serialized by converting them into Avro format using the Avro encoder.
- The serialized packet is inserted in a Kafka Message and loaded on the associated Kafka Topic, containing other data of the same type.

The Kafka Server manages the persistence of a certain amount of data on the disk, and it executes the requests of the active producers and consumers groups.[49].
The CCDQC, in the Kafka context, represents the consumer side and it has the following communication features:

- A certain CCDQC instance is associated with a specific data type and a specific Kafka consumer is linked to a specific Kafka topic.
- The consumer is persistently looking for new messages on the topic. If a new one is available, the consumer will acquire it from that topic.
- The acquired message contains the encoded packet, the field will be decoded and the original DL0 packet will be available.

Now we will focus on the development of the consumer for the SDQC.

In addition to these communication constraints, Kafka consumers must implement further logic to manage DL0 data, involving functions to write data, measure time, and get a unique time. The functional logic of the consumer is the following:

1. **Split data in batch** basing on a defined rule. Up to now, we have decided to split data based on packet acquisition time. All the messages on the Kafka topic are ordered by arrival time, and the messages contain packets that have been loaded in acquisition order. We have defined a *time window* of $T_{Batch} = 1\,s$ so that when we receive the first packet *pi* from the Kafka topic acquired at time t_{pi}, all the packets *pj* with $t_{pj} \leq t_{p0} + T_{Batch}$ are collected in the same batch, otherwise, a new batch is created, updating the initial time with $t_{pi} = t_{pj}$. An array of packets is then formed for each batch.

2. The **epoch time** reconstruction. In the data struct, the Time Tag represents the time at which the packet was acquired. Timetag is in nanoseconds, seconds, minutes, days, months, and years format; we must define each t_p from epoch to simplify the logic and keep it general. C++ implements some standard methods to get the total epoch time, however, we developed a custom function to reduce computational effort.

3. A method to tackle the **interruption of data flow**. Telescope does not produce packets at a fixed rate, so we cannot know a priori the number of packets that we are receiving. Moreover, there is the possibility of failure of the Kafka server or the internet connection to be taken into consideration.
   Using the time rule as described in 1, a batch can be closed if and only if we receive the packet belonging to the next data batch, as a consequence a timeout is required to close the batch in absence of new data. A timer class was defined with methods such as reset, and get_time to enhance the logic's robustness and allow the pipeline to elaborate all the data collected.

4. **Write the batch** in a file. Data packets are complex and we must use an efficient method to open them in other programs, that could also be written in different programming languages. We decided to rely on the HDF5 library which has API for different programming languages and can perform data operations.

5. **Write a DQ-Pipe synchronization file** after that the batch file has been closed. The DQ-Pipe process synchronization mechanism checks the creation of an *Ok file*, defined in Sec. 3.2.1, in order to avoid to start the next application job prematurely. In fact, the mechanism provided by the DQ-Pipe cannot know if our batch file, when appears in the file system, is still opened and modified by the Kafka Consumer. Checking directly for the creation of the batch file, would mean start to start, with an high probability, the DQ-Analysis pro-

cess on a corrupted file. Kafka Consumer then produces an additional file, named as the current batch filename but with a .ok extension, that dq-pipe will use to trace the newly created batch.

The consumer, for each data that it is acquiring, will convert the instant of the trigger of the camera in epoch time and will check if it belongs to the current batch. If it does, data is then appended in an array of packets, our batch. Otherwise, the batch is written in an HDF5 file, and the data is saved in the first position of the batch, then acquisition proceeds. We can visualize the logic in Fig. 3.9.

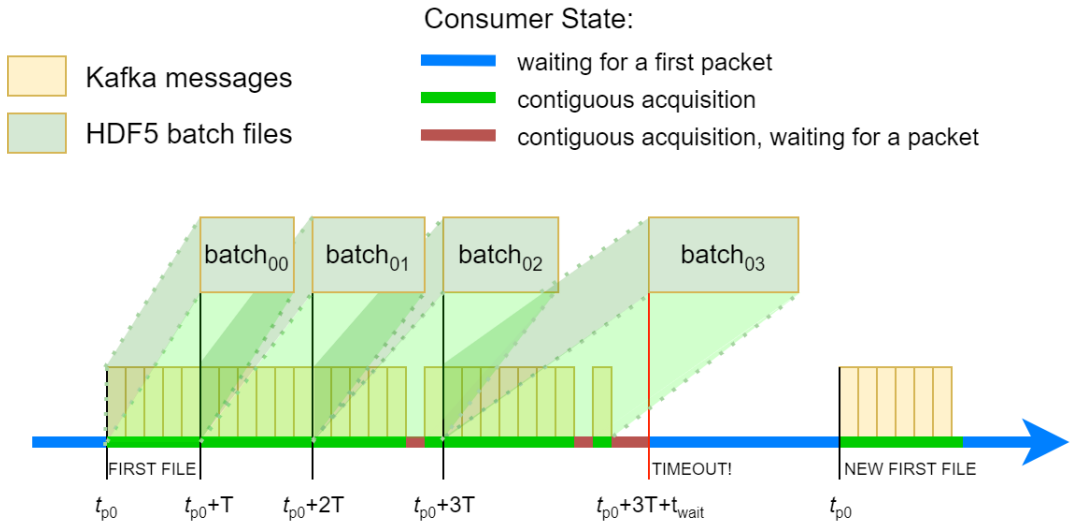


Figure 3.9: The time rule for collecting packets. When the Kafka consumer acquires a new message, it extracts the data packet, checks the packet's Time Tag, and closes the batch if it was acquired after T=T_{Batch} seconds from the first packet of the current batch. The batch is then stored in an HDF5 file and a new packet collection starts.
If packet flow from Kafka is interrupted, the batch is closed after a given amount of time.

Since Batch data is complex, an adaptation is necessary to write it in the HDF5 file. We divide packet fields into two distinct structures, then write them into two distinct compound datasets. Compound datasets are only compatible with C data types, in particular with POD structs[15]. Since the Avro generator defines all the packets with non-POD structures (they contain std::vector data that are not C types) then a data adaptation is required by splitting the packet's header information and the PDM block which contains scientific data. The final HDF5 layout we want to obtain can be defined as follows. We define the filename as "S22BatchX.Run.dl0.h5", where X is a string representing the batch sequence number, varying from 0 to m,

[15]POD stands for Plain Old Data - that is, a class (whether defined with the keyword struct or the keyword class) without constructors, destructors, and virtual members functions.

initialized with many "0" as the number of digits of m. Within the file, we define a group "/dl0/" in which we store two datasets "S22PacketHeaderTimeTag" and "S22ScientificData", as in Fig. 3.10. According to the S(2,2) packet maximum event rate of $f_{S22,\,Max} = 1000\,Hz$, we define then the following properties for S22PacketHeaderTimeTag dataset:

- Datatype: compound datatype derived from a struct S22_generalinfo.
  Referring to the S(2,2) DL0 description in 3.1, S22_generalinfo is a struct containing the variables from lines 20 to 35. For the sake of memory efficiency, the variable types in this struct are of the smallest byte size possible according to the S(2,2) R0 types. Appropriate datacasting is then required when initializing the buffer to be written in the dataset. The data type is then defined by inserting all the members of the struct in an HDF5 Compound datatype called "mtype1", with the insertMember method as explained in the HDF5 section.

- Dataspace: it is an array of N_packs elements, where N_packs, according to the time-window definition of 1 second, varies from 1 to 1000. For each element i in the array, we find the packet general information related to the i-th packet belonging to the batch.

Thereafter we define the following properties for the S22ScientificData dataset:

- Datatype: compound datatype derived from a struct S22PDMBlock.
  Referring to the S(2,2) DL0 description in 3.1, S22_PDMBlock is a struct containing the variables from lines 9 to 14. Instead of std::vectors in positions 13 and 14 there are two fixed-size arrays of "Num_pixels" elements, which are the number of pixels for each PDM equal to 64. As for S22_GenInfo, the variable types in this struct are of the smallest byte size possible. The data type is then defined by inserting all the members of the struct in an HDF5 Compound datatype called "mtype2".

- Dataspace: it is a matrix of dimension N_packs $\times$ PDM_size, where PDM_size is a fixed number and is equal to 37. According to the time-window definition of 1 second, the dataspace dimension varies from 1×37 to 1000×37. For each element i, j in the matrix, we find the j-th PDM block of the i-th packet belonging to the batch.

To handle the main program functionalities, Kafka Consumer provides also methods and code to be launched with a set of command line arguments mandatory to set the proper code functionalities. We must provide the code with the following information:

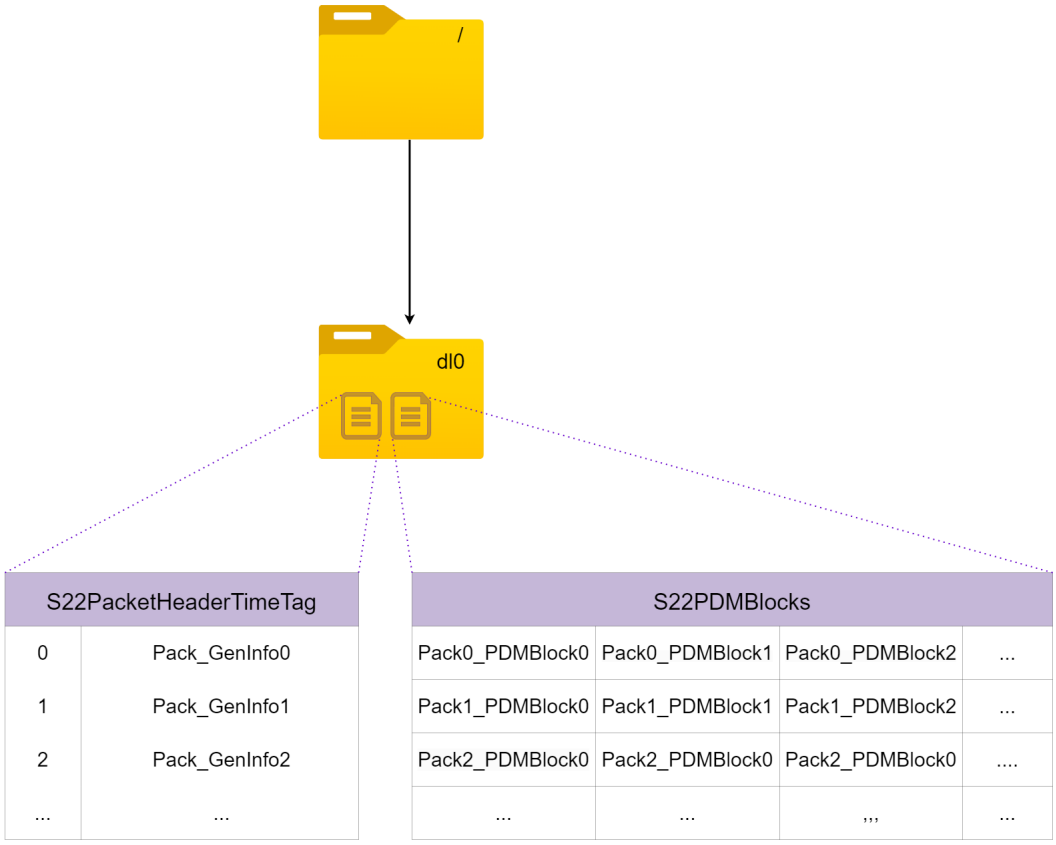


Figure 3.10: S22BatchX.Run.dl0.h5 file tree

- the output file system path where HDF5 batch files must be saved;
- the mandatory information to operate with Kafka: IP, port to configure the Kafka Consumer class, the Kafka topic name, and the groupID to assign the consumer to the target topic and consumer group;
- a flag for writing .ok files.

Besides the main program functionalities, Kafka-Consumer must also measure the code performances, providing the number of packets acquired, the packet acquisition frequency in terms of packets per second, the average packet processing time, and the average writing time of HDF5 files. The program must also manage some errors at run time by raising an exception and interrupting the program. Some examples of errors that can be managed are wrong command line arguments and errors in the creation of the HDF5 files, given by the system paths without user permission, and bad target data to write the declaration. The program must also handle process signals such as SIGINT to kill the program correctly.

Now we move further on providing a simplified implementation view of the most important aspects with some high-level pseudo code for the Kafka-Consumer to manage scientific data S(2,2). The program makes use of the following classes:

- **S22HDF5Handler**: the class S22HDF5Handler provides a method named S22HDF5Write, which writes the data provided in the S22batch array into an HDF5 file. The S(2,2) packets are provided as an array of pointers to S22 structs. The method takes the following arguments:
  - AcquiredPacket: a pointer to an array of S22 structs containing the data to be written in the HDF5 file;
  - _file_name: a character string that specifies the name of the output file;
  - N_Packs: an integer that specifies the number of packets to be written in the HDF5 file.

  The S22_GenInfo and S22_PDMBlock structures contain the data to be written in the HDF5 file. From AcquiredPacket, we allocate arrays of struct type S22_GenInfo of N_Packs elements and arrays of struct type S22_PDMBlock of N_Packs $\times$ N_PDMs elements, where the number of PDMs is 37. The method creates an HDF5 file, groups, and datasets to store the batch. Afterward, it creates the dataspace and the memory datatype objects to describe the layout of the data to be written into the datasets. Finally, it writes the S(2,2) packet data to the datasets in the HDF5 file.

- **OKFileHandler**: provides file handling functionality. The class has one private member variable, active, which is set to false by default but can be set to true using a constructor argument. If active is true, the file creation is enabled, so methods have some effect on the program.
  The class has one public member function called write_files which takes a const char* argument _file_name. write_files creates a new file with the extension ".ok" and returns true if the file creation was performed successfully.

- **myperformanceTimer**: it is a timer class for managing performance times in a program. It uses the steady_clock from the C++ standard library's chrono header to measure time. The class has several methods for implementing the temporized logic for closing the batches when the first file does not belong to the next batch. Such logic involves the use of the following methods:
  - reset, which resets the timer with a given timeout value;
  - sampletimer, which takes a sample of the elapsed time without stopping the timer. When the timer is elapsed, it returns 0.

We define the following functions:

- time_epoch: a function that calculates the epoch time given a packet that contains year, month, day, hours, minutes, seconds, and timeTagNanosec;
- HDF5_Write: a function that writes data to HDF5 files;
- NewBatchInit: a function that updates the starting and ending time of a new batch.

The code also defines the following constants and variables:

- timewindow: a constant variable to set the packets time window. We set it to 1 second so that packets are collected in batches of 1 second;
- NPACKS: the number of collectible packets before writing to the file, which is used to initialize the array which collected packets. It is set according to the time window: if timewindow=1, then NPACKS = timewindow $\times f_{S22,\,Max}$ = 1000.
- EMPTYTOPICTIMEOUTMS: the timeout in milliseconds to wait for data on the Kafka topic. It is used with the reset() method of myperformanceTimer;
- POLLTIMEOUT: the timeout in milliseconds for polling data from the Kafka topic;
- running: it is a boolean variable that indicates whether the program is running it can be turned to false by SIGINT;
- BatchEnd: it is a variable used to keep track of the absolute ending time of the current batch;
- s22hdf5: it is an instance of the S22HDF5Handler class used to write data to the HDF5 files;
- OkF: it is an instance of the OKFileHandler class used to write data to the HDF5 files;
- batchtimeout: it is an instance of the myperformanceTimer class used to implement the temporized logic for closing the batches when the first file does not belong to the next batch.

We can see the Kafka consumer main behavior in the Listing 3.2.

Listing 3.2: Kafka-Consumer pseudocode

```
1 #definition of some imported functions/classes (not ours)
```

```
2  from cppkafka import configuration, Consumer , Message
3  from avro.S22 import binary decoder
4  #define TIMEWINDOW 1 #The requested batch time window in seconds
5  #define MAXFS22 1000 #the S(2,2) packets max frequency
6  #define NPACKS (TIMEWINDOW * MAXFS22) # lenght of the array S22
       collection
7  define EMPTYTOPICTIMEOUTMS 1000 #ms, timeout to close batches
8  define POLLTIMEOUT 1000 #polling timeout
9
10 #define the array to create the batch
11 packetcollection[NPACKS]={0}
12
13 #convert utc local time to epoch in nanoseconds
14 time_epoch(*packet)
15
16 #build hdf5 filename, write packetcollection to hdf5, write ok file
       , return boolean
17 HDF5_Write(*s22hdf5, *okfileHandler, *packetcollection, NPacks,
       out_folder_path):
18
19 #put last message in packetcollection[0] and update time window
20 NewBatchInit(*packetcollection, last, *End):
21
22 Main (out_folder, okf_active, ip, port, topic, groupID):
23   #consumer init: creation of a configuration class
24   Configuration config=(ip, port, groupID, autocommit=false)
25   #cppkafka lib implements consumer
26   Consumer consumer(config)
27   Consumer.subscribe(topic)
28
29   ii=0 #index
30   jj=false #false=0, true=1 (circular index)
31
32   BatchEnd=0 #absolute time window end
33
34   Message msg[2] # save up to two kafka messages
35   #CLASSES instantiation
36   S22HDF5Handler s22hdf5 #to write batches in hdf5 files
37   OKFileHandler OkF(okf_active) #to write ok files
38   myperformancetimer batchtimeout #to close batches without waiting
         a new file
39
40   first=true
41   while true :
42
```

```
43      #check for a POLLTIMEOUT ms if there is a new message on kafka
44      msg[jj] = consumer.poll(POLLTIMEOUT)
45
46      if msg[jj] :
47        #New message:
48        # avro decoding and collect data in packet (binary decoder is
              specialized in decoding S22 packets)
49        packetcollection[ii]=binarydecoder(msg[jj])
50        if first :
51          first=false #first packet received
52          #timewindow definition: first packet time + timewindow [ns]
53          BatchEnd = time_epoch(packetcollection[0]) + TIMEWINDOW
54        #check for each collected packet, the time tag
55        CurrentPacketTime = time_epoch(&packetcollection[ii])
56        if CurrentPacketTime >= BatchEnd :
57          #packet out of timewindow, write the batch:
58          running = HDF5_Write(&s22hdf5, &OkF, &packetcollection[0],
                ii, out_folder_path)
59          #update first packet in the buffer and time window
60          NewBatchInit(&packetcollection[0], ii, &BatchEnd)
61          #communicate to kafka the last position consumed
62          #only when we write a batch
63          consumer.commit(msg[jj])
64          #initialize to 1
65          ii=1
66        else:
67          #update the packetcollection index
68          ii++
69          #trigger this counter so that we remember also the previous
                last message received (for consumer commit purposes)
70          jj = not jj
71        #reset the timer every the time a new packet is received
72        batchtimeout.reset(EMPTYTOPICTIMEOUTMS)
73      else:
74        #No new Messages:
75        # we need to close the file if packets flow stops
76        if (batchtimeout.sampletimer() == 0):
77          running = HDF5_Write(&s22hdf5, &OkF, &packetcollection[0],
                ii, out_folder_path)
78          #communicate to kafka the last position consumed
79          #only when we write a batch
80          consumer.commit(msg[not jj])
81          # New Acquisition can start:
82          first = true
83          ii = 0
```

```
84    consumer.close()
85    return 0
```

3.5.2 DQ-Analysis

DQ-Analysis is the pipeline application aiming at performing the quality checks on the data managed by the OOQS.
The OOQS Use Case Document specifies the analyses that the CCDQC sub-components must perform on the S(2,2), VAR(10,2), and VAR(10,3) Camera Data. We are interested in developing a C++ application that can potentially produce all the analyses. Currently, we implemented only a subset of the quality checks expected for S(2,2) packets. The use cases that can be addressed are the following:

- *Use Case 3.1.2*: histograms with the number of triggers for each camera PDM using a sliding window with 10000 events. In total, they are required 37 PDM/camera $\times$ 9 telescopes = 333 Counters;
- *Use Case 3.1.6*: sample camera image at 1 Hz using not calibrated data (ADC values);
- *Use Case 3.1.8*: calculates the histogram for the ADC values event-by-event for the camera and all the PDMs. Create one histogram for the entire telescope observation and another with a sliding window of a configurable number of seconds.

In general, for the various use cases, it has been necessary to manage two different aspects. The first issue regards the fact that quality checks may require the definition of a sliding window operating on the number of events (generated packets) or on the observation time. The second issue concerns the generation of a result based on the entire observation.
The DQ-Analysis performs the quality checks on the smallest possible sliding window, then, through an aggregator application, the final use case results will be available.

To implement the minimum possible size of a sliding window, called a sub-sliding window, it may be necessary to open and process multiple batch files during the same computation if the number of events in a single file is not sufficient. The implementation of the time-based sliding window consists of opening a batch file, performing the use case on the available packets, and finally computing the right time window size by an aggregator.

Implementing the event-based sliding window requires more complicated logic: a daemon must continuously open all the new batch files, collect their variable number of packets in an array always available at run time, and compute the result based on the newest packets. A preliminary analysis suggested that a Python application is not performing enough for the analysis with a native Python class, by calling its methods in a specialized DQ-Pipe instance. For this reason, we decided to develop a C++ executable application that DQ-Pipe can submit as an external batch process.

The implementation of the DQ-Analysis can be summarized in the following functional steps:

1. **Open HDF5 Batch files**: a function that given the filename, opens the file, then saves the two datasets in two appropriate container objects that manage a variable size of packets. All the datatypes involved in the reading operations are defined as for the writing operation already described in the Sec. 3.5.1.

2. Define the **sub-sliding windows** on a *fixed number of events*, and on a *fixed time*. The minimum time-size sub-window Min_sub_time is set equal to the batch's time window.

   $$Min\_Sub\_Time = T_{Batch}$$

   The sub-event windows Min_Sub_Events, is defined as

   $$Min\_Sub\_Events \; = \; Max\_Batch\_Size = T_{Batch} \; \times \; f_{S22,\,max}$$

   otherwise, the quality checks will give inconsistent results due to the lack of information. The final sliding windows aggregation is performed by the DQ-Aggregator, described in Sec. 3.5.3. Further explanations about the sub-sliding window generation is shown in Fig. 3.11.

3. **Compute the analysis**: a function computing the Use Case exploiting the available packets has been implemented for each use case.

4. **Multi-file opening logic**: every time a new file is available, our application computes the quality check on a defined sub-sliding window, writes the result, and terminates. If a single batch file content is not sufficient for the quality checks, the process will open the penultimate received file to complete the time window on previous data. This mechanism can continue opening older files, up to a maximum number $Max\_OpenFiles$, that results to be compatible with the number of packets involved in the sub-sliding window computation.

5. **Write the DQ-Analysis results** in an HDF5 file: only the analyses carried out will be inserted on file.

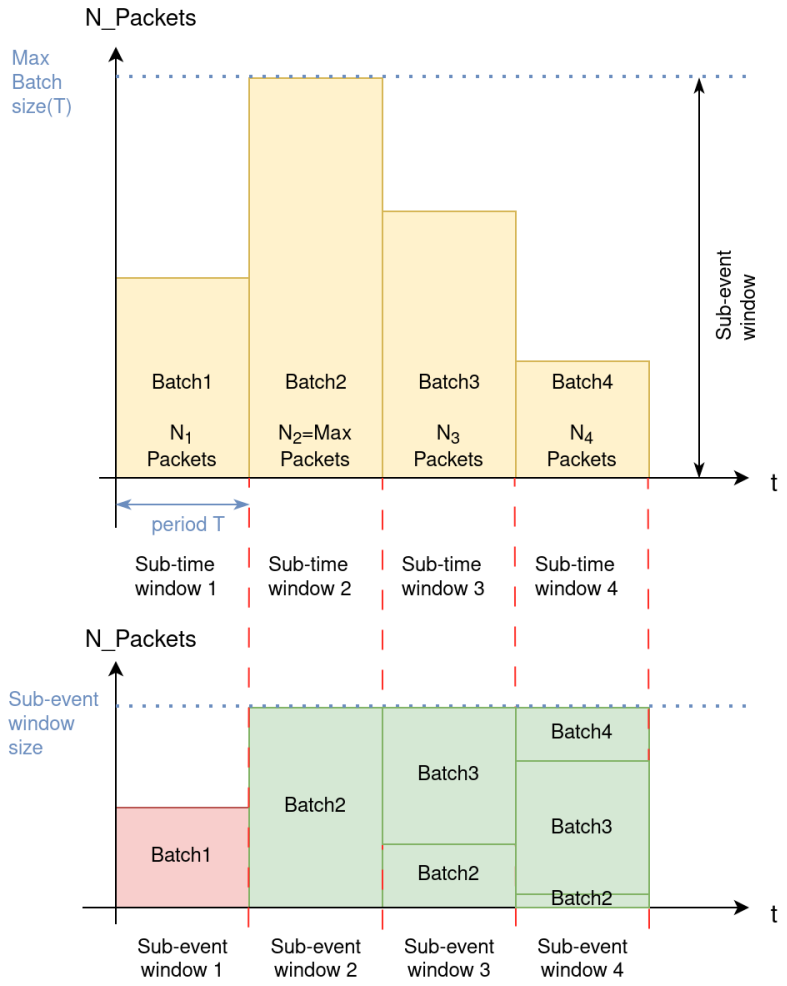


Figure 3.11: A representation of the sub-event and sub-time sliding windows. The minimum event sub-window size Min_Sub_Events, cannot be lower than the maximum number of packets that a batch file can contain, except for the initial instants of the observation. Moreover, Min_Sub_Events cannot be equal to a target Use Case sliding window size $Sliding\_Size$, since the application can occur in excessive performance degradation. A preferable trade-off is choosing the lowest divisor of the original time window size, higher than Max_Batch_Size. If more use cases define different sliding windows $Sliding\_Size\_i$, it is necessary to select the minimum sub-event window size according to the minimum common divider, such that:
$Min\_Sub\_Events = \gcd(Sliding\_Size\_1, Sliding\_Size\_2, \ldots, Sliding\_Size\_n)$,
and verifying: $Min\_Sub\_Events \geq Max\_Batch\_Size$.

6. **Write a DQ-Pipe synchronization file** after that the results file has been closed.

To handle the main program functionalities, DQ-Analysis provides also functions to fetch a set of command line arguments such as:

- the input file system path of the first HDF5 batch file to open,

- the output file system path where saving the HDF5 batch files,
- a flag to write .ok files.

Besides the main program functionalities, DQ-ADQ-Analysis must also measure the code performances, and provide the number of opened files and their opening time, the packet processing time, and the writing time of the HDF5 result file. The program must also manage some errors at run time, such as wrong command line arguments and errors in opening or creating the HDF5 files, by raising an exception and interrupting the program.

We provide now an implementation of the DQ-Analysis C++ application in our pipeline. Kafka-Consumer settings impose: $T_{Batch} = 1\,\mathrm{s}$, and $Max\_Batch\_Size = 1000\,\mathrm{Packets}$. Then, the sub-sliding windows, according to the use cases, can be defined as:

$$Min\_Sub\_Time = 1\,\mathrm{s}\ , \quad Min\_Sub\_Events = 1000\,\mathrm{Packets}$$

Moreover, we assume that the telescope's camera will acquire packets at a minimum rate of $Min\_Rate = 100$ packets per second due to several triggers generated from cosmic-rays initiated air showers; we then define $Max\_OpenFiles = Min\_Sub\_Events/Min\_Rate = 10$. A class called **S22AnHandler** has been developed of which we can see a diagram in Fig. 3.12

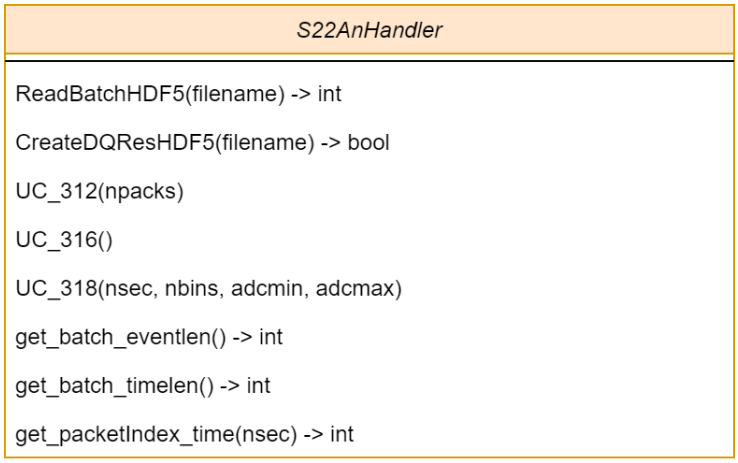


Figure 3.12: The S22AnHandler UML diagram

The class functionalities are the following:

- open a HDF5 batch file generated from the Kafka Consumer with the method *ReadBatchHDF5*;
- performs analysis requested from OOQS. For each analysis implemented, a method corresponding to a specific use case has been developed;

- store the results of the analysis in a new HDF5 file with the *CreateDQResHDF5* method.

We describe now the Use Cases already mentioned above and the operation requested for writing their result, assuming that the right number of HDF5 files to perform the sub-sliding windows has already been opened.

Use Case 3.1.2 implementation

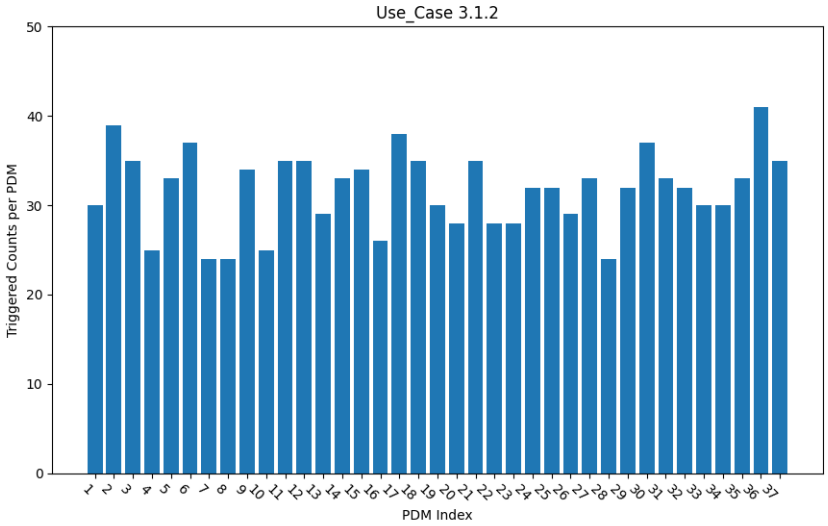


Figure 3.13: The count of the triggered PDMs of single camera, on an event window of 10.000 events, simulated data

We must check which PDMs are involved in the camera trigger generation. PDM blocks are stored in a 2D vector called S22PDMs, where S22PDMs[i][j] identify the j-th PDM of the i-th packet opened from a file. If a PDM has triggered a relevant event has been detected in the sky portion acquired by its SiPM sensors and the triggered flag is set to true. This use case is shown with random values in Fig 3.13. To allow the increment of the counter, then the PDM should necessarily be turned on, meaning pdmVal set to true, and be enabled in the algorithm performing the trigger generation, meaning triggerEnabled equal to true. uc_312_count is an array of 37 elements storing the PDM triggers of NPacks, where $NPacks$ is equal to the Min_Sub_Events. Listing 3.3 shows the pseudo-code.

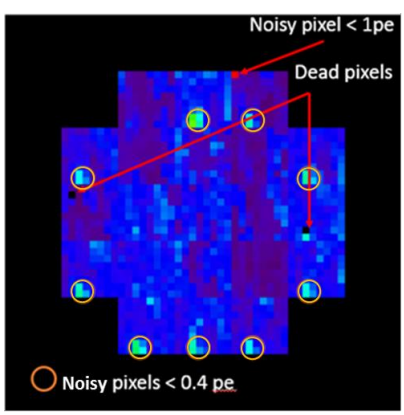
Listing 3.3: UC-312 pseudo code

```
1 UC_312(NPacks):
2     # Initialize uc_312_count array to zero
3     for i from 0 to PDMsize:
```

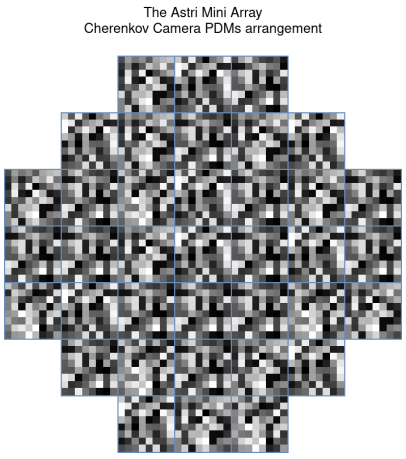
```
        uc_312_count[i] = 0

    # Iterate over each packet
    for Pack_i from 0 to NPacks:
        # Iterate over each PDM in the packet
        for PDM_i from 0 to PDMsize:
            # Check if the PDM is triggered and triggerEnabled is
                true
            if S22PDMs_vector2d[Pack_i][PDM_i].pdmVal is false AND
               S22PDMs_vector2d[Pack_i][PDM_i].triggerEnabled is
                  true AND
               S22PDMs_vector2d[Pack_i][PDM_i].triggered is true:
                  # Increment the count for the corresponding PDM
                  uc_312_count[PDM_i] = uc_312_count[PDM_i] + 1
```

To store the results of the quality check in the HDF5 file, we define a dataset as an array of 37 integer 32 bits elements in an "/Analysis/UC_312/" group.

Use Case 3.1.6 implementation



(a) A Camera image from ASTRI-Horn, we can notice the 21 PDMs composing the Cherenkov camera. Image from [26]



(b) An ASTRI Mini-Array Camera Image from ADC values, with 37 PDMs. Simulated data

Figure 3.14: A graphical representation of ADC values plotted following the PDMs focal plane pattern

A scientific Camera image, sampled at low frequency, can be easily plotted into an HMI; an example is shown in Fig. 3.14.
Every time that we open a new file , we select the first packet of the batch, copy the ADC high gains, low gains, and SSC values contained in S22PDMs, and copy them in a new structure having datatype cameraimage_1HZ. To store the result

of the quality check in the HDF5 file, we define a compound dataset with a single element in an "/Analysis/UC_316/" group, where the compound datatype is defined as cameraimage_1HZ datatypes.

Use Case 3.1.8 implementation

This use case aims in finding the distribution of the camera ADC values, for the purposes described in 3.3.4.
We define 37×2 GSL histograms to collect the high gains and low gains from the ADC values. The histograms are divided into a number of bins *nbins* in a linear range between two *adcmin* and *adcmax* values; these parameters are completely configurable for the sake of a flexible software solution. For instance, we set up the histograms parameters as $nbins = 100$, and assumed the camera ADC to produce values in the range $adcmin = 0$, $adcmax = 65536$, that is the maximum range for the ADC values in the R0 format (unsigned integer 16 bit). When real camera data will be available, a proper ADC range will be defined.
For each PDM, we assign each ADC value to the corresponding PDM histogram. We additionally define 2 GSL histograms with a linear range to aggregate the PDM histograms and obtain the total camera histograms. Listing 3.4 shows the pseudo-code.

Listing 3.4: UC-318 pseudo code

```
def UC_318(nsec, nbins, adcmin, adcmax):
    # Initialize histograms for ADC values of each PDM and total
       camera
    for i in range(PDMsize):
        uc_318_PDMlowgains[i] = gsl_histogram_alloc(nbins)
        uc_318_PDMhighgains[i] = gsl_histogram_alloc(nbins)
        gsl_histogram_set_ranges_uniform(uc_318_PDMlowgains[i],
           adcmin, adcmax)
        gsl_histogram_set_ranges_uniform(uc_318_PDMhighgains[i],
           adcmin, adcmax)

    uc_318_cumulativelowgains = gsl_histogram_alloc(nbins)
    uc_318_cumulativehighgains = gsl_histogram_alloc(nbins)
    gsl_histogram_set_ranges_uniform(uc_318_cumulativelowgains,
       adcmin, adcmax)
    gsl_histogram_set_ranges_uniform(uc_318_cumulativehighgains,
       adcmin, adcmax)

    # Retrieve the number of packets corresponding to nsec of
       acquisitions
```

```
15      NPacks = get_packetIndex_time(nsec)
16
17      # Iterate over each packet
18      for Pack_i in range(NPacks):
19          # Iterate over each PDM in the packet
20          for PDM_i in range(PDMsize):
21              # Iterate over each gain of the PDM
22              for gain_i in range(gains_size):
23                  # Assign the gain_i-th adc gain of the PDM_i-th PDM
                        to the corresponding histogram bins
24                  gsl_histogram_increment(uc_318_PDMlowgains[PDM_i],
                       float(S22PDMs_vector2d[Pack_i][PDM_i].lowgains[
                       gain_i]))
25                  gsl_histogram_increment(uc_318_PDMhighgains[PDM_i],
                        float(S22PDMs_vector2d[Pack_i][PDM_i].highgains
                       [gain_i]))
26
27      # Cumulate histograms to build the total camera histograms for
           both ADC lowgains and highgains:
28      for PDM_i in range(PDMsize):
29          gsl_histogram_add(uc_318_cumulativelowgains,
               uc_318_PDMlowgains[PDM_i])
30          gsl_histogram_add(uc_318_cumulativehighgains,
               uc_318_PDMhighgains[PDM_i])
```

The obtained histograms can be plotted as shown in Fig. 3.15 To store the results in the HDF5 file, we define 5 different datasets, in an "/Analysis/UC_318/" group:

- two datasets with a bi-dimensional dataspace of N_PDMs $\times$ *nbins* (37 $\times$ 100) elements, of double datatypes, for storing the PDM histograms for low gains and high gains,
- two datasets with a monodimensional dataspace of *nbins* (100) elements, of double datatypes, for storing the total camera histograms for low gains and high gains,
- a dataset with a monodimensional dataspace of *nbins* + 1 (101) element, of double datatypes, for storing the histogram ranges.

Main Code

The main code must provide a code for extracting the batch file number from the input file system path to generate an output file name consistent with it. Additionally, the batch number must be used to perform the iteration on the older files,

producing an older input file name.

Based on the defined sub-window sizes, we iterate on the batch input files the S22HDF5Read() method to reach the desired number of packets. If there are no more available files, for instance during the first observations, then the time window is adapted to perform the analysis on a lower number of packets. We can see the DQ-Analysis main code in the Listing 3.5.

Listing 3.5: DQ-Analysis pseudocode

```
from CCDQCAnHandler import S22AnHandler

MAX_OPENFILE = 10
MIN_SUB_EVENTS = 1000
MIN_SUB_TIME = 1
#UC-318 constants
NBINS = 100
ADCMIN = 0
ADCMAX = 65536

def get_batchnumber(file_path: string, file_ext: string,
    outfileheader: string) -> string:
     """Extracts batch number from file name"""
```

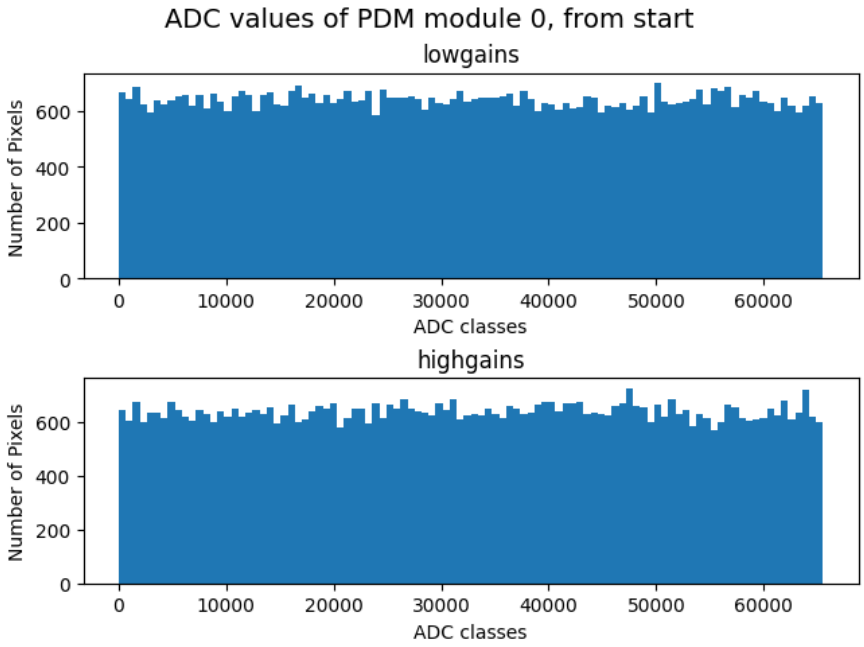


Figure 3.15: The high gains and low gains ADC histograms for the PDMID = 0, data aggregation on 1000 test packets. Histograms parameters: $nbins = 100$, $range = [0, 65536)$. Graph generated with a Jupyter notebook script to visualize the DQ-Analysis results. Simulated data

```
13 def get_filepath(file_path: string) -> string:
14     """Returns directory path from file path"""
15 def updateIn_file(In_file : string, number : int)
16     """Returns an updated InputFile name"""
17 main (In_file: string, Out_dir: string, OkFiles: bool):
18 
19     # extract the batch number from the input filename
20     file_ext = "Run.sim.dl0.h5"
21     outfileheader = "S22Batch"
22     outfileext = ".Run.sim.DQ.h5"
23     In_path = get_filepath(In_file)
24     # get the batch number
25     batchnumber = get_batchnumber(In_file, file_ext, outfileheader)
26     nn = int(batchnumber)
27     #build output file system path
28     Out_pathfile = Out_dir+outfileheader+batchnumber+outfileext
29 
30     S22handl = S22AnHandler() # instantiate the analysis class
31 
32     #set run-time sub-sliding windows
33     nevents_rt = MIN_SUB_EVENTS
34     nsec_rt = MIN_SUB_TIME
35     ii=0
36 #during file opening, check to reach the number of packet/ time
    required for the sub-sliding window requirements, and stop if
    too much files opened
37     while ((S22handl.get_batch_eventlen() < MIN_SUB_EVENTS
38             or S22handl.get_batch_timelen() < MIN_SUB_TIME)
39             and ii < MAX_OPENFILE):
40         # open a file and count the number of opened files with ii
41         ii = S22handl.S22HDF5Read(In_file)
42         if nn > 0:
43             updateIn_file(In_file, --nn)
44         else:
45             break # stop if no older files available
46     # handling nn = 0 or ii = MAX_OPENFILE reducing time-windows
47     if S22handl.get_batch_eventlen() < MIN_SUB_EVENTS:
48         nevents_rt = S22handl.get_batch_eventlen()
49     else:
50         nevents_rt = MIN_SUB_EVENTS
51     if S22handl.get_batch_timelen() < MIN_SUB_TIME:
52         nsec_rt = S22handl.get_batch_timelen()
53     else:
54         nsec_rt = MIN_SUB_TIME
55     # perform the analysis
```

```
56      S22handl.UC_318(nsec_rt, NBINS, ADCMIN, ADCMAX)
57      S22handl.UC_312(nevents_rt)
58      S22handl.UC_316()
59      S22handl.CreateDQResultHDF5(Out_pathfile) #Write result
```

This code is then compiled in an executable called "DQ-Analysis.lnx"

DQ-Pipe for the DQ-Analysis

We focus now on the DQ-Pipe objects that can execute an *external application*, such as the DQ-Analysis.lnx just described. A specialization of the DQ-Pipeline is then necessary, so an **ASTRI-DQAnalysis** class inherits DQ-Pipeline and defines the start() method. To implement the trigger action for starting a new process, the FileSystemDS extends the abstract DataSource class and provides a watchdog in the Kafka-Consumer Output folder, returning the file system path of the new HDF5 file as soon as the .ok file showing the same as the HDF5 file is available. FileSystemDS class is connected to the input of ASTRI-DQAnalysis. To provide an output of the DQ-Analysis process the OutputHandler class is extended with the OutputToFileSystemASTRI class, which defines its abstract methods. Additionally, a GarbageCollector class keeps cleaning the input folder by checking for files older than a specified date and time. The method single_start() is performed in a concurrent thread, with the RepeatThread class that invokes Run() periodically.
We can see the diagram with all the classes' interconnection in Fig. 3.16.

We now describe the behavior of the DQ-Pipe during the nominal operation as shown in the Listing 3.6. The **start**() method of ASTRIDQ-Analysis calls **startWatch**(), a method of *dataSource*: a *watchdog* is started in the input folder directory specified in the configuration file. A watchdog control can detect several events happening in this folder. We extended the watchdog functionalities in the FileSystemDS class, to detect the creation of new files ending with the ".ok" extension. Whenever the watchdog detects one of these specific files, it adds in a *queue* the file path, cropping the .ok extension, referring then to the homonymous file we wanted to detect. The queue is then returned to the start method with a *Python generator* **waitForFile**() which yields all the new events in the queue. If it is empty, it yields a NULL type.
The output directory is provided by *OutputHandler*, through the **getPath**() which returns a string of the desired file system path. Every returned file is then exploited to launch the external analysis process, which will analyze at least that detected file.
It can be submitted as a new *Slurm Job* on a computing cluster, or as a standard

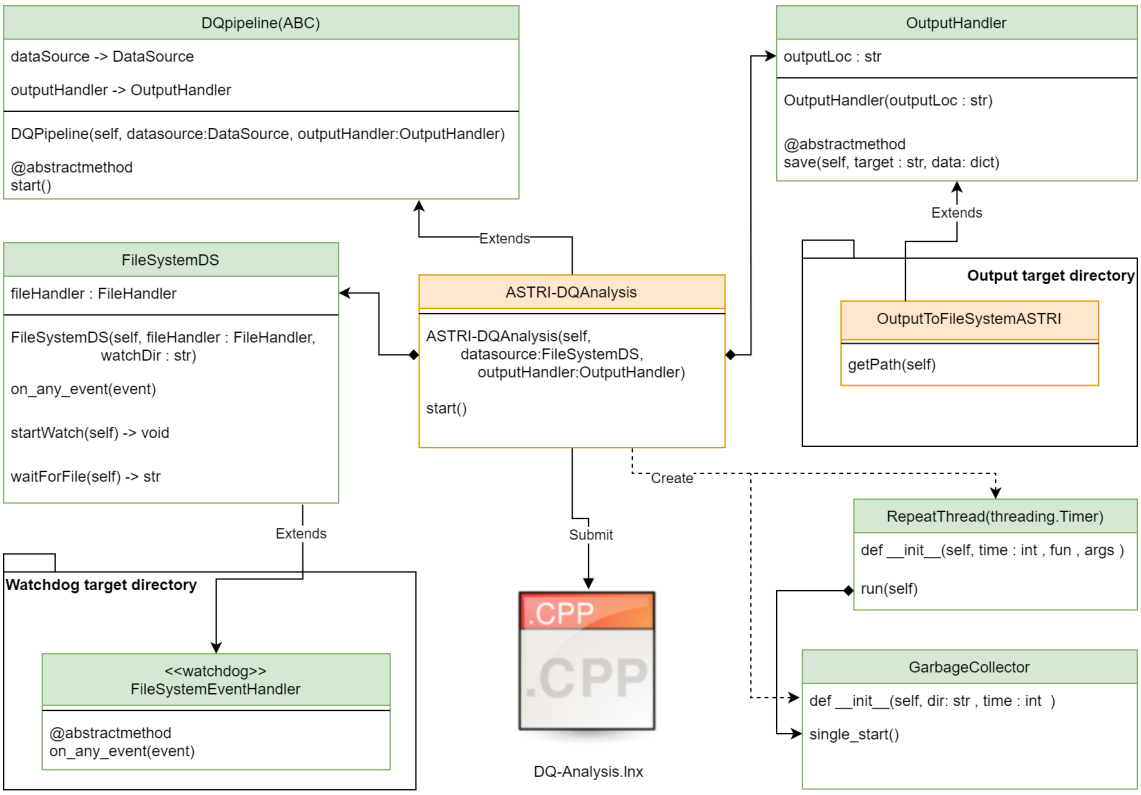


Figure 3.16: The DQ-Pipe specialization to run DQ-Analysis application

bash process. A function **Extprocess**() to format the command is then executed, including the program name, input file, and output directory. Alternatively, to define a new Slurm Job, we call **JobSlurm**() that writes a batch file specifying a name for the Slurm job, where the process must write output logs and errors, which partition use to execute the process and the command for executing the process as before. JobSlurm returns a *sbatch command* to run the sbatch file
The external process is launched with the **Popen**() method, a function of subprocess that creates a new child parallel process but does not wait for child termination.

Listing 3.6: ASTRIDQ-Analysis start method to submit external processes

```
1    def start(self):
2        #start a watchdog in the target directory
3        self.dataSource.startWatch()
4        #waitForFile returns an iterator on a queue of events
             detected on .ok files in the dataSource target dir
5        file_gen = self.dataSource.waitForFile()
6
7        for filePath in file_gen:
8            #filePath is the filesystem path of a new detected file
9            if filePath is not None:
```

```
10
11                  self.logger.debug(f"New file extracted from the
                        queue: {filePath}. Queue lenght: {self.
                        dataSource.files.qsize()}")
12                  # Check if build a sbatch file for slurm
13                  if self.opMode == "noslurm":
14                      command = self.Extprocess(self.process,
                            filePath, self.outputHandler.getPath())
15                  elif self.opMode == "slurm":
16                      command = self.JobSlurm(self.process, filePath,
                            self.outputHandler.getPath())
17                  # Submit process
18                  DQ_process = subprocess.Popen(command, shell=True)
```

3.5.3 DQ-Aggregator

The DQ-Aggregator is the last application constituting our pipeline and is responsible for the following aspects:

- Collect the DQ-Analysis quality checks results and operate on them to reconstruct the desired Sliding Window size;
- Format Data in a Specific dictionary object;
- Store the dictionary in a database to archive persistently. The database operation concludes the workflow on the batch.

The databases that we tested are MySQL, described in Sec. 3.4.1, and MongoDB, described in Sec. 3.4.2. To do so a Python application has been developed and its operation has been directly integrated into the *DQ-Pipe*, through a set of classes, shown in Fig. 3.17, that extend the framework's functionalities:

- **ASTRI-DQAggregator**: this class is an extension of the DQ-Pipeline, and the working principle is similar to the one already illustrated in Sec. 3.5.2 for the ASTRI-DQAnalysis class. ASTRI-DQAggregator implements the **start**() method, shown in the Listing 3.7, that instantiates some entities such as variables to declare the sliding-window size, counters, and numpy objects compatible with the Analysis Result datasets. Afterward, a watchdog observer in the DQ-Analysis output folder is started and it is used to perform aggregation operations every time a new file is detected. When a new result file is available, through the readDataFromFile_numpy method of dataSource we get the

content of the file stored in a dictionary containing numpy objects of the Quality Checks to be aggregated. With the collected data, the aggregation step is then performed and a database dictionary is formatted into a new dictionary that is passed to the OutputHandler, which is extended to load the data on a database specified in the XML file.

- **HDF5HandlerASTRI**: it is an extension of the abstract FileHandler and implements a method read_numpy() called with FileSystemDS, the dataSource of the ASTRI-DQAggregator.
- **OutputToMongoDBASTRI**: it provides a connection to a MongoDB database and defines a query to insert a new element in a collection. The stored data is composed of our aggregation result data along with the loading UTC instant.
- **OutputToMySqlASTRI**: it provides a connection to a MySQL database and defines a query to insert a new row in a table. The stored data is composed of our aggregation result converted into a JSON string, along with the loading UTC instant.

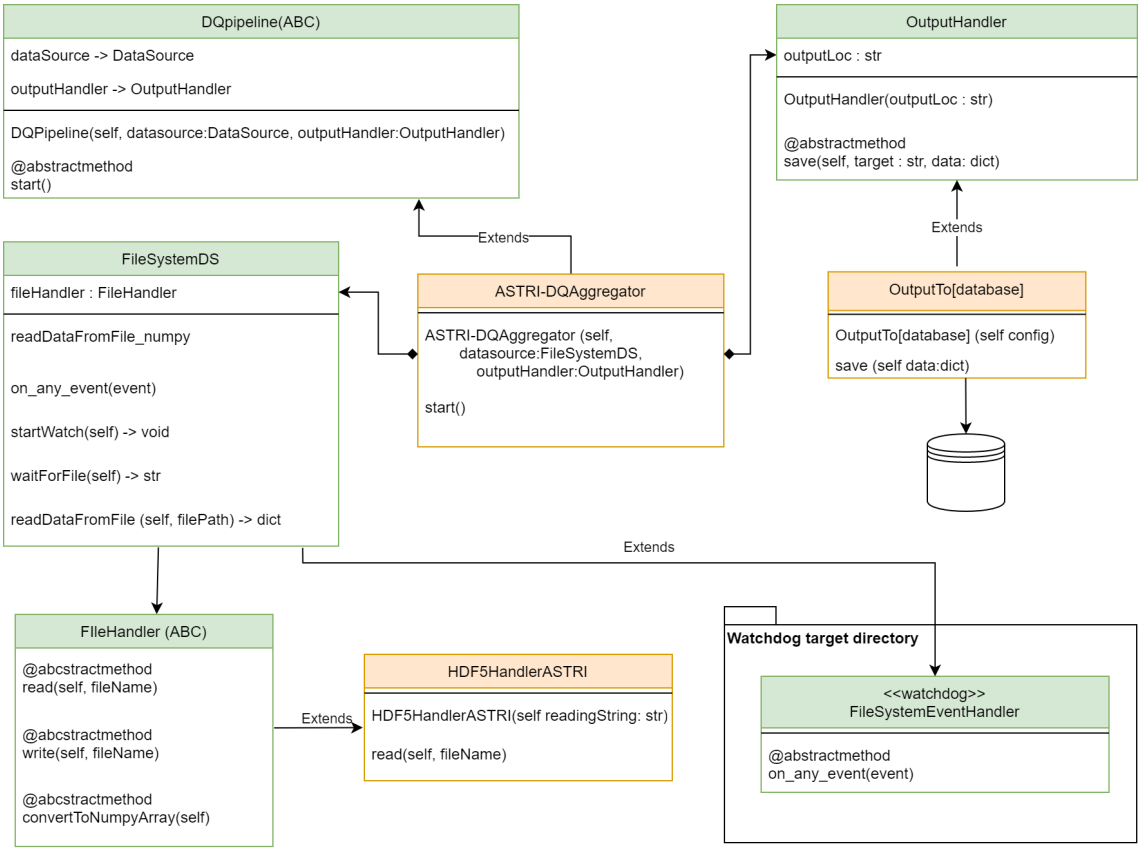


Figure 3.17: The DQ-Pipe specialization to perform the DQ-Aggregator application

To perform aggregation, the dictionary derived from a file is then stored in a list of fixed length *Analysis*, determined as a function of the maximum event windows and time windows to be performed. A circular array is then used to address the list keeping the indexes bounded. Being Max_OpenFile the dimension of the array, to increment the index i we perform:

$$i = (i + 1)\%Max\_OpenFile$$

where "%" is the module operation.
The aggregation level is specific to the use case. given the desired UCX$_sliding_size$ and knowing that each file contains a fixed UCX$_sub_window_size$ then we define:

$$UC\text{X}\_Nfiles = \frac{UC\text{X}\_sliding\_size}{UC\text{X}\_sub\_window\_size}$$

where X is the number of implemented Use Cases. Then, we define several of numpy objects UCX,Y$_Aggr$ initialized to zero, of size equal to a dataset Y that needs aggregation, belonging to the Use Case X.
To perform the *aggregation*, we increment UCX,Y$_Aggr$ with the last dataset arrived, stored in $Analysis[i]$.

$$UC\text{X,Y}\_Aggr \ += \ Analysis[i]['\text{X}']['\text{Y}'],$$

where the symbol += represent an increment operation $x \mathrel{+}= a \iff x = x + a$.
And to implement the *sliding window*, for each new file that arrives, we decrement UCX,Y$_Aggr$ by removing the oldest dataset not contributing anymore to the aggregation. We need to calculate the index basing on the UCX$_Nfiles$, and consistently with the circular address:

$$UC\text{X,Y}\_Aggr \ -= \ Analysis[i\_oldUC\text{X}]['\text{X}']['\text{Y}'],$$
$$\text{Where:} \quad i\_oldUC\text{X} = (i + Max\_OpenFile - UC\text{X}\_Nfiles)$$

where the symbol −= represent a decrement operation $x \mathrel{-}= a \iff x = x - a$.
In the initial moments of the run, there are no older files available in the list so decrementing the aggregation must not be performed until UCX_Nfiles are opened.

Listing 3.7: ASTRIDQ-Aggregator start method to perform the UC_318 sliding window

```
def start(self):
        ##Variable Declarations
        self.garbage_start()
        #watchdog start in DQ-Analysis output folder
        self.dataSource.startWatch()
```

```
        file_gen = self.dataSource.waitForFile()
        for filePath in file_gen:
            if filePath is not None:
                Analysis.insert(i_last,self.dataSource.
                    readDataFromFile_numpy(filePath))

                print(f"318analisys count: {i_last}:{max_open_file
                    -1}\n")
                UC_318_aggregatedPDM_SW=UC_318_aggregatedPDM_SW[:]+
                    Analysis[i_last]['UC_318']['PDM_Bins'][:]
                UC_318_aggregatedCamera_SW=
                    UC_318_aggregatedCamera_SW[:]+Analysis[i_last]['
                    UC_318']['Camera_Bins'][:]

                uc318i=self.update_circ_counter(i_last,
                    max_open_file, Nfile_318)
                print(f"UC318 sliding window {i_last}:{uc318i}\n")
                try:
                    UC_318_aggregatedPDM_SW=UC_318_aggregatedPDM_SW
                        [:]-Analysis[uc318i]['UC_318']['PDM_Bins'
                        ][:]
                    UC_318_aggregatedCamera_SW=
                        UC_318_aggregatedCamera_SW[:]-Analysis[
                        uc318i]['UC_318']['Camera_Bins'][:]

                except:

                    #the requested element of the list is not
                        available since the number of Analysis file
                        opened is less than Nfile_318
                    pass

                #Save UC aggregated:
                Aggregation_Result=dict()
                #UC312 aggregation to Aggregation_Result
                #UC316 to Aggregation_Result
                #UC318 is shown:
                Aggregation_Result['UC_318']['Bin_intervals'] =
                    Analysis[i_last]['UC_318']['Bin_intervals']
                for i in range(UC_318_aggregatedPDM_SW.shape[0]):
                    Aggregation_Result['UC_318']['PDM_Bins'].append
                        (dict( lowgains = UC_318_aggregatedPDM_SW[i
                        ,:,0].tolist(), highgains =
                        UC_318_aggregatedPDM_SW[i,:,1].tolist()))
                Aggregation_Result['UC_318']['Camera_Bins']=dict(
```

```
                    lowgains = UC_318_aggregatedCamera_SW[:,0].
                    tolist(), highgains = UC_318_aggregatedCamera_SW
                    [:,1].tolist())
35
36                  #Update Circular Array
37                  i_last=self.update_circ_counter(i_last,
                    max_open_file)
38                  ##load Aggregation_Result in the database
39                  self.outputHandler.save(Aggregation_Result)
```

To perform an insert query operation versus a Database, we divide the elements of the numpy dataset into simpler objects and define a dict containing all the updated analysis results. Single quality checks can be accessed by an associated key. The use of a dictionary is preferred when dealing with databases in Python, since dicts can be natively encoded into JSON, providing a string containing a description of the dictionary. We currently tested databases such as *MySQL 5.7* and *MongoDB 6*. MySQL can load a JSON string in a single column of a table as shown in the Listing 3.8, by fetching it as a string or a JSON object, depending on the data specified for the target column during the table creation. Along with the JSON data type, a set of SQL functions is available to enable operations on JSON values, such as creation, manipulation, and searching. These functionalities cannot be executed when JSON is loaded as a string in a single column, and to achieve a result similar to the one before, we should load each different quality check result in different columns, increasing the table dimension and the queries complexity. However, dealing with large JSON data can lead to performance issues, since MySQL does not support JSON indexing[16] directly.

Inserting JSON data into a MySQL table can potentially be slower than inserting traditional row-based data. In fact, JSON data is often larger and more complex than traditional data types. This can increase the amount of time it takes to write the data to disk.

MongoDB treats dictionary objects natively, transforming them into BSON objects, a subset on JSON data. BSON documents can be indexed for efficient querying using a wide range of indexing options. Morover MongoDB being a NoSQL database provide operations generally faster than relational databases like MySQL. Then, a comparison between the two databases capabilities can be considered of interest for

[16]JSON indexing is the process of organizing and storing JSON data in a way that allows for efficient and speedy querying. JSON indexing involves creating a data structure that allows for rapid retrieval of specific pieces of data within a JSON document. This is typically done by creating an index on one or more fields within the JSON data.

our pipeline application.

Listing 3.8: The OutputHandlerToMySql class to init the database connection and save a record

```
class OutputToMySqlASTRI(OutputHandler):

    def __init__(self, conf, outputLoc, dqchain_id ):
        super().__init__(outputLoc)

        self.table=outputLoc
        self.dqchain_id = dqchain_id
        self.mysqlHandler = MySqlHandler(conf)
        self.logger = PipeLoggerConfig().getLogger(__name__)

    def save(self, data):

        encodedData = json.dumps(data)
        batchtime= datetime.datetime.now()
        encodedBatchtime=batchtime.strftime("%Y-%m-%d %H:%M:%S")
        #inserting a new row in the table
        query = f"insert into {self.table} (json_data,batch_time)
            values ('{encodedData}','{encodedBatchtime}')"
        self.mysqlHandler.write(query)
```

Chapter 4

Deployment and Performance Evaluation

This chapter of the thesis focuses on the physical aspects of the deployment of the OOQS-Pipeline, presenting the testing environment that has been chosen to evaluate every application. Moreover, we focus on the performance analysis of the pipeline, which is developed to receive and elaborate a data stream in real-time, during the telescope acquisition. We explain the real-time constraints and discuss the high-frequency and large bandwidth of the data stream that the application is required to process. To demonstrate the software's performance, we present metrics measured by the application and relevant graphs that highlight the capabilities and limitations of our solution.

4.1 Test Data and Environment

The ASTRI Mini-Array architecture is designed to have nine Camera Servers, each hosting a separate ADAS instance installed on bare metal. These Camera Servers will acquire data from the nine telescopes through a direct point-to-point connection and send it to nine OOQS instances via Kafka. The OOQS instances, including all necessary software components, will be deployed as Docker containers within a Computing Cluster composed of multiple servers, shared with other sub-systems.
The Array Observing Site is under construction, and the real servers that will constitute the array observing site are not available up to now. Moreover, no ASTRI Mini-Array real camera packets are available. To implement and test our pipeline, then we have relied on a server available at the INAF computing center located in Bologna (Italy).

The on-site Computing cluster will be composed of 6 machines with a dual-socket

configuration, 20 core - 40 threads CPUs with a base clock of at least 2.3 GHz, 256GB DDR4 Ram, and available data storage of 6TB SSD. The Server will be connected with a fast 10 Gbit/s LAN which ensures great reliability. To simulate the environment in the context of our application, it has been used a machine with a dual-socket configuration, with 14 cores - 28 threads CPUs with a base clock of 2.6 GHz, 125GB DDR4 RAM that is sufficiently powerful to simulate a realistic environment to test our application.

The ADAS-OOQS interface

As discussed in Sec. 2.3, the ADAS-OOQS interface is based on Apache Kafka. We started the development by installing on the server a test environment that the ADAS team provided to us: a Linux CentOS7 Virtual Machine (*ASTRIVM*) with 6 CPU cores, 8 GB RAM, and about 60 GB HDD available space. This VM holds all the software to simulate the telescope subsystems involved in the ADAS-Kafka data stream; this software includes:

- The **Kafka server**, created with *Apache Kafka 2.8*, is accessible via the virtual machine's localhost and a defined port;
- The **Kafka Producer**, able to acquire R0 test packets, convert them in the Avro serialized DL0 data, and upload them as messages on a specified Kafka topic. This is a simplified and less powerful version of the final ADAS software;
- A Python application called **PacketFactory**, that can generate the S(2,2), VAR(10,2) and VAR(10,3) packets in the R0 format that Kafka Producer can manage. PacketFactory can generate a stream of a defined number of packets, containing information such as the telescopeID, packetType, and subType. The remaining packet fields are filled with random numbers. We cannot simulate scientific data, however, we modified PacketFactory to simulate also a consistent *TimeTag*, so that we can realize a logic based on the camera acquisition time.

Thanks to this environment, we can emulate the behavior of ADAS during a telescope observation. The Kafka-Consumer application has been developed with C++11, inside the ASTRIVM.
This environment has been used during the Kafka Consumer developments, and to test if the application can acquire persistently data from long observations.

Pipeline deployment

The DQ-Pipe framework, and the DQ-Analysis and DQ-Aggregator applications, have been developed within a Python 3.9 Anaconda environment containing all the required dependencies.
DQ-Pipe has been installed and tested outside the ASTRIVM, directly on the host machine, to separate the Kafka environment from the analysis one. To allow the data workflow from the Kafka-Consumer to the DQ-Analysis application, a shared folder has been created, and configured in both applications as described in the DQ-Pipe Section number 3.2.1 .
Databases are running in separate environments: *MySQL 5.7* is installed on the ASTRIVM, while *MongoDB 6.0* is installed on a Docker container, running on the server. Both ASTRIVM and MongoDB Container, are configured with Network Mode *Network Access Translation* NAT; to access the databases from the host machine, or the external network, a port forwarding versus the internal database ports has been configured.
Slurm 17.11 can manage the DQ-Pipe workload among 50 logic cores of the machine, in this context Slurm will manage: two DQ-Pipeline single core daemons and a third aperiodic job is given by the external DQ-Analysis application.

4.2 Performance

An OOQS-Pipeline instance will be responsible for receiving and processing data for each individual telescope camera, and for performing the work on the entire telescope array, a set of nine pipelines that work independently on the data generated by each camera is then needed. We now focus on evaluating the performance of a single instance of an OOQS-Pipeline. The pipeline is composed of a sequence of jobs performed on a stream of files containing the received packets. the pipeline is *event-driven*, meaning that each job execution is allowed by the presence of a new file. The overall data processing workflow that the pipeline can perform is as follows:

- Kafka Consumer will continuously receive the packets available on the Kafka topic, and according to a *fixed time window* of the telescope observation, it will divide the data into batch files. For instance, during a real observation, if the time window is fixed to one second, HDF5 files will be created every second containing the newest acquired data and the next application will execute almost every second.

- for every new batch file, the ASTRIDQ-Analysis pipeline will then *synchronously*

submit the DQ-Analysis process, so that must perform its job in a time compatible with the time window. The result is a new HDF5 file containing partial results. Synchronously means that only a single DQ-Analysis process can be started at a time, avoiding parallel execution. Verification of the timings involved in the processing time of the ASTRIDQ-Analysis is then required to assess if this solution is feasible.

- for every new result file, the ASTRIDQ-Aggregator pipeline, must perform its job in a time compatible with the time window. the result is then loaded into the target database.

The respect to timing constraints is then important to ensure that the pipeline will perform the analysis in real time. The applications must finish their job on the current batch data within the fixed time window before the next batch arrives. The time window can be considered as a simple time limit to respect, so that the application works correctly, respecting the maximum event rate and bandwidth imposed by the S(2,2) packets. The OS could schedule the pipeline's processes in a disadvantageous way to allow respect for the time window, and one or more applications could exceed it. However, DQ-Pipe inserts the files detected by the watchdog inside a queue, handling compensation for the eventual delays.

Kafka Consumer

The basic functionalities of the Kafka Consumer described in Sec. 3.5.1 are performed in a single thread, even if the application is multi-threaded. The consumer implementation requires three additional background threads created from the cppkafka library. These threads manage several aspects involved in the connection to the Kafka server, such as the acquisition of the messages from the Kafka topic, the message offset committing, and the generation of the so-called "heartbeat" signal, which is dispatched to the server until the consumer is active.

An initial Kafka Consumer implementation with Python suggested to us that is not possible to acquire the S(2,2) packets with a single-core Python application, since the maximum collectible event rate was about $\sim 600\,Hz$. We have identified that the function which caused the bottleneck in the program speed is the Avro deserialization, and we couldn't optimize that part of the code. We then decided to move to a C++ implementation.

We tested the Kafka Consumer C++ application with a simulation of an observation from a single telescope. This test is necessary to understand if the Consumer can handle the S(2,2) packet event rate and bandwidth. To simulate a high-frequency

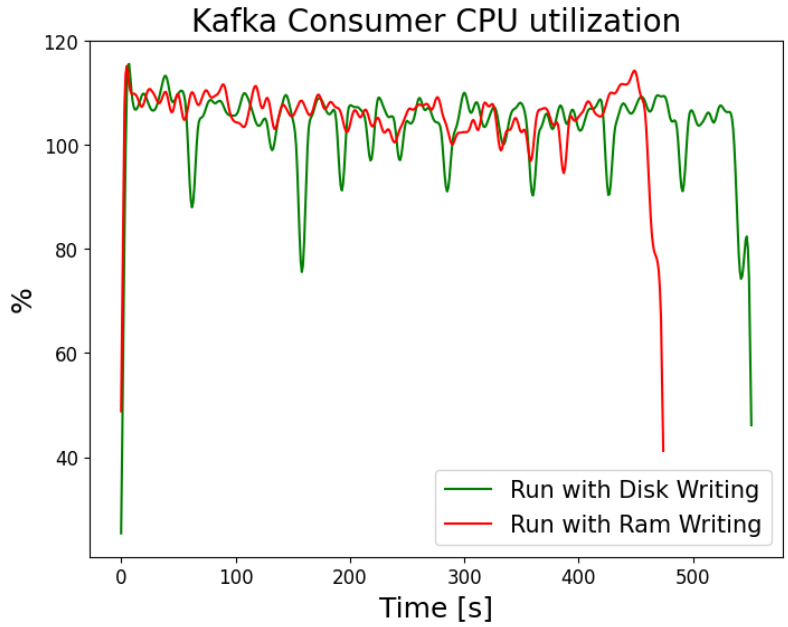


Figure 4.1: The Kafka Consumer CPU utilization during a long run

acquisition accurately, it's essential to pre-load all the desired packets into the Kafka Topic using the processor application provided by the ADAS team. This approach helps to reduce the number of active processes in the ASTRIVM during the entire observation simulation. Additionally, the processor can only produce at an event rate of 300Hz, which creates a bottleneck when acquiring packets at higher event rates.

The Observing run duration can be calculated as:

$$\text{N\_Packets} = \text{Topic\_bytesize}/\text{DL0\_bytesize}$$

$$t_{observation}[s] = \text{N\_Packets}/f_{packets}$$

A single DL0 packet is 16456 Bytes and the free space in the ASTRIVM is about 60 GB.

We conducted a *long-run test* with a data stream of 3.232.856 packets, produced with Time Tags to simulate a fixed 1000Hz frequency. The amount of packets consists of a size of $\sim$ 53 GB of data, and leads to a real Observation time of $\sim$ 55 min, which is about two times the duration of an *Observing Block*, as described in Sec 1.3. The Kafka Consumer will acquire packets no stop from the Kafka server, and it is set to shut down automatically as soon as packets on the topic are finished. It is of interest to compare two different runs: one with the consumer writing batch files in a File System directory mounted on Hard Disk and one writing files in a temporary file system on RAM.

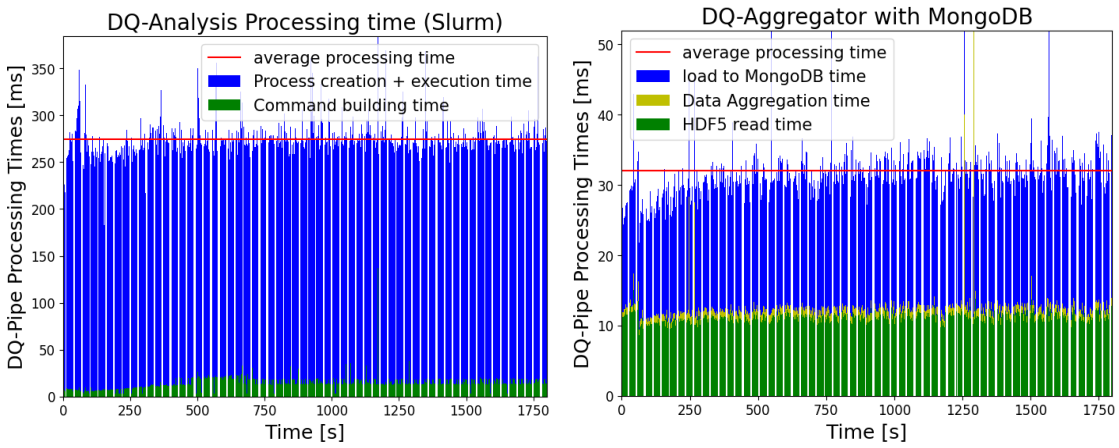
As we can see in fig. 4.1, we can derive two main pieces of information from this plot:

- Consumer terminates the run writing on the disk in 560 seconds (9 min and 30 s). Considering the number of packets which constitutes the observation, the maximum event rate that the consumer can potentially manage on this machine is 5800 Hz.
- Consumer terminates the run writing on the RAM in 480 seconds (8 min). Considering the number of packets which constitutes the observation, the maximum event rate that the consumer can potentially manage on this test machine is 6700 Hz, when mounting a temporary file system.

Moreover, the Kafka consumer CPU utilization, when acquiring data at a maximum speed, is lower than 120 %. We notice that the multiple threads constituting the program do not involve a CPU load on more than 2 physical Cores. Considering nine telescopes, we could be able to reduce the number of active processes by managing more than one telescope with a single Kafka Consumer instance, reducing then the number of resources allocated to manage the whole telescope array data stream. The high performance can be useful also to handle a hypothetical situation where the Network connection to the Kafka server goes down from the OOQS side during the observation, while ADAS continues to stream packets, which accumulate on the Kafka topic. When the availability of the network is restored, the consumer has good performance to recover all the messages published in a very short time, about 6/7 times faster than the real-time requirements. Kafka Consumer Batch Files are written every second, containing a maximum number of packets equal to 1000, and consequently a maximum size of 9.5 MB. With nine consumers actively writing data at the maximum event rate, then it is required a storage device with a minimum sequential write speed greater than 90 MB/s. The on-site Computing Cluster will mount SSDs that can satisfy this requirement. Overall, a single instance of Kafka Consumer can successfully respect the maximum S(2,2) packet frequency of 1000 Hz and it can acquire the data bandwidth required by an instance of the OOQS-Pipeline, of 16 MB/s. By instantiating nine Kafka Consumers it is then possible to acquire the packets generated from the whole telescope array respecting the maximum target bandwidth of 148MB/s.

OOQS-Pipeline

The Kafka Consumer now is limited in acquiring at the limit event rate of 1000 Hz, to test the workflow at the target maximum frequency.



(a) The ASTRIDQ-Analysis benchmarks during an Observation of 30 minutes

(b) The DQ-Aggregator loading to MongoDB the data analysis result as JSON

Figure 4.2: DQ-Aggregator benchmarks during an Observation of 30 minutes

In Fig.4.2a we can see columns representing the execution time of the DQ-Analysis application, for every new batch file that is detected. The overall process execution is composed of two phases:

- the part of the time depicted in green is given by the python code necessary to build the command to launch the external process. This time is negligible compared to the blue one.
- The part of the time depicted in blue is the amount of time required to submit and complete the execution of the external process "DQ-Analysis.lnx".

We can note that the average job execution of the DQ-Analysis is about 280 milliseconds when managing the workload using Slurm. The average job execution time is respectful of the batch time window of 1 second, and allows us to run the external process synchronously with the DQ-Pipeline work.
In Fig.4.2b we can see columns representing the execution time of the DQ-Aggregator application, for every new data quality result file that is detected. The overall process execution is composed of three phases:

- the part of the time depicted in green is given by the reading time of the current HDF5 file.
- The part depicted in yellow refers to the aggregation time, that is required to update the sliding windows iterating efficiently on the desired set of data results. This time is negligible compared to the other two parts.
- The blue part shows the time required to insert the final aggregated result on MongoDB.

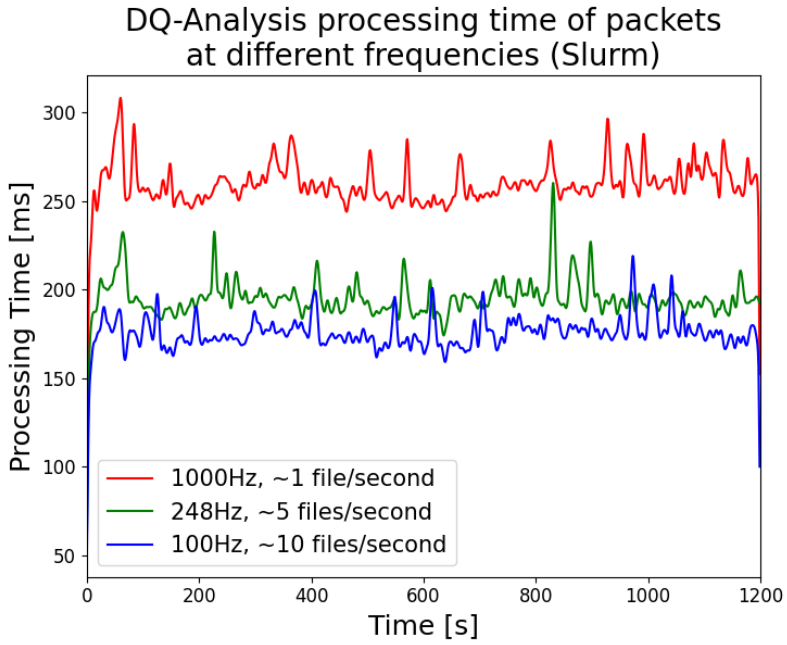


Figure 4.3: The time to perform the external DQ-Analysis C++ process on different amounts of packets per batch, Slurm only

We can note that the average job execution of the DQ-Aggregator is about 32 milliseconds when inserting data in a MongoDB collection. The average process execution time is respectful of the batch time window and allows us to conclude the workflow on every batch without delays. Overall, we can note that the pipeline can proceed continuously elaborating neatly all data batches with the data acquired from Kafka. At the moment, there are no obvious problems regarding the accumulation of delays by the applications that do the data processing.

Focusing on the external process timings, an interesting comparison is shown in the graph 4.3, which reports the time required for the external process "DQ-Analysis.lnx" to perform quality checks, in the presence of telescope observations that produce packets at different event rates. All the processes are executed on Slurm. Setting the time window of 1 second, in each of the reported runs, the DQ-Analysis application will always produce results based on the size of the sub-windows explained in Sec.3.5.2. We can note:

- in the case of the test at an event rate of 1000 Hz, the program only opens a single file at every process call to execute all the use cases; the average process execution time is about 260 ms.

- In the case of the test at an event rate of 248Hz, the program opens the last 5 files backward to execute use cases that require a fixed number of events, such as 3.1.2; the average process execution in this scenario is about 200 ms. For

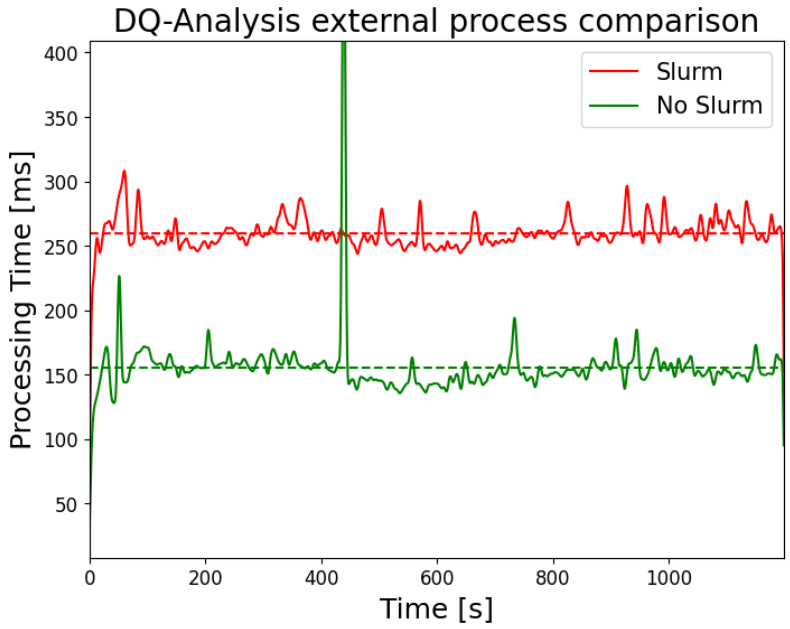


Figure 4.4: The time to perform the external DQ-Analysis C++ process, Slurm versus bash process

the other implemented use cases, only the last created batch is needed.

- In the case of the test at an event rate of 100 Hz, the program opens the last 10 files backward to execute the use cases; the average process execution in this scenario is about 175 ms.

Then, we can notice that the time involved to process data every second does not grow linearly to the event rate that characterizes each run; an explanation is then required. The use cases which are based on the sub-time window require to use, independently from the event rate of the run, only the packets contained in the latest arrived file; this involves a time proportional to the event rate and contributes to the overall processing time. On the other hand, the strategy of opening multiple files to reach the target sub-event window size, required for some implemented use cases, involves an increment of time necessary to open sequentially all the required files; this maps in the overall processing time, making it grow inversely proportional to the packet event rate. However, we remember that up to now we consider 100 Hz to be the minimum frequency to handle, and the graph reports the feasibility of a synchronous execution of the DQ-Analysis, in the event rate $[\,100\,,\,1000\,]$ Hz. We conclude the discussion for the DQ-Analysis reporting that the size of the HDF5 file containing the analysis results, considering the currently implemented use cases, has a size of 79.2 KB, independent from the time window selected.

In Fig.4.4, we compare scheduling an external process with Slurm versus launch-

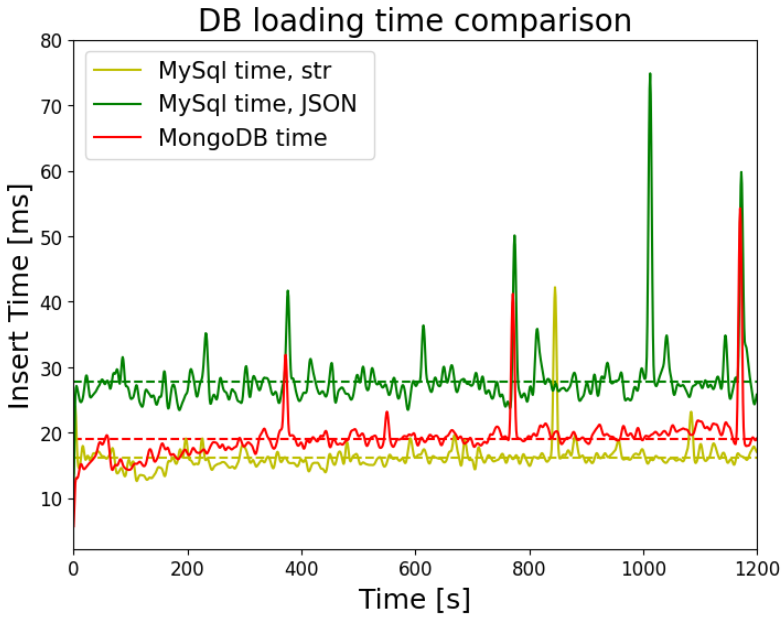


Figure 4.5: The DQ-Aggregator Database insert comparison

ing it as a standard bash process. The graph indicates that when using the same process and data, scheduling with Slurm takes an additional 100 ms compared to running the same processes as bash. This extra overhead may be unavoidable when running the pipeline by distributing jobs on a computing cluster. Therefore, it is important to take this into account when considering whether the analysis will take longer than the given time window. In such cases, the processing scheme should be modified to start the external processes asynchronously and leverage parallel execution.

Finally, Figure 4.5 displays a comparison of the time required by an insert query for two databases used in the pipeline. The data being uploaded to a database is a Python object occupying 450,KB in memory. Here are some key observations:

- When using MongoDB, the dictionary is converted in BSON automatically and upload takes on average 19 ms. Additionally, we report that the size of the data in the collection is around 130 KB.

- However, with MySQL, loading requires first converting the data into a JSON formatted string, which is a Python object of average size 95 KB. If this data is entered into a JSON field, loading takes 28 ms and the average row size is 150 KB. If the data is entered into a text field, the required time is reduced to 17 ms and the average row size of the table is now 103 KB.

Considering the current size of the aggregated data quality results, loading a JSON in MySQL as a string is the best solution if no queries are required on the fields of

the data analysis results. In comparison, an insertion in MongoDB takes up to 10% longer. In addition, the data size is the lightest, allowing the use of storage space in the most efficient way.
However, considering the analysis results loaded as JSON to perform queries on its single fields, MongoDB with the BSON represents the best solution: insertion of a JSON in MySQL takes up to 47% longer. Moreover, by creating new quality checks and including their result in the JSON object, as discussed in Sec. 3.5.3, loading such data on MySQL could lead to an inefficient lengthening of the time required by an insert query. In addition, the size of the data is lighter than that required in MySQL, allowing for more efficient use of storage space.

Conclusions

In this thesis, we discuss the research activities carried out during the internship at the Astrophysics and Space Science Observatory of Bologna (OAS): a branch of the Italian National Institute for Astrophysics (INAF), in collaboration with the researchers of ASTRI Mini-Array[1]. The ASTRI Mini-Array is an international collaboration for a ground-based project for astrophysics led by INAF with the partnership of the Instituto de Astrofísica de Canarias, Fundación Galileo Galilei, Universidade de São Paulo (Brazil), North-West University (South Africa) and University of Geneva (Switzerland).

This section provides a summary of the thesis and outlines the conclusions of our work, discussing also the future developments. This thesis work is divided into four parts.

The first part, discussed in Section 1, provides an introduction to the scientific context of the thesis project. We present the ASTRI Mini-Array project, an array of nine Imaging Atmospheric Cherenkov Telescopes (IACTs). These gamma-ray observatories are capable of detecting the Cherenkov blue light produced in air showers, the manifestation of high-energy photons entering the atmosphere.
In Section 2, we described the Supervisory Control And Data Acquisition (SCADA) system, which controls all the operations carried out on-site. SCADA is composed of various components, each performing specific operations during telescope observations. One such component is the Online Observation Quality System (OOQS), which is responsible for the conduction of real-time data quality checks on the Cherenkov Camera's output. The OOQS's primary function is to diagnose anomalies that may arise during observations, providing feedback to other SCADA subsystems and informing on-site operators of the telescope's condition. The goal is to ensure that the collected data meet the highest quality standards achievable by next-generation telescopes like ASTRI Mini-Array so that scientists can perform accurate analyses. The Cherenkov Camera is the primary instrument used for Cherenkov observations in the ASTRI Mini-Array, producing various types of packets with different information. The Array Data Acquisition System (ADAS) is

responsible for acquiring and pre-processing all camera packets before dispatching them to the OOQS.
The OOQS will receive the scientific and variance packet types. Variance data are two periodic data packets, with a frequency of 10 Hz and a size of about 10 KB. Scientific packets constitute the most challenging scenario, these are characterized by a variable frequency, with maximum peaks of 1000 Hz. Each scientific packet has a fixed size of about 16 KB, the maximum data flow is therefore about 16 MB/s for each telescope, for a total of 148 MB/s from the whole telescope array. The Cherenkov Camera Data Quality Checker (CCDQC) is the OOQS component performing the Cherenkov data quality checks in real time. It must implement a software solution able to receive the data packets from ADAS and perform the requested quality checks on each camera data, then store the results in the on-site Quality Archive.

In Section 3, we discussed the implementation of the OOQS-Pipeline, a software prototype that receives scientific packets generated from a Cherenkov Camera, performs quality analysis, and stores the results in a storage system. To elaborate data from the whole telescope array, a set of nine pipelines that work independently from each other, on each camera data, is then required. To facilitate a comprehensive understanding of the pipeline's operations, we provided a detailed overview of all the used software, libraries, tools, and storage systems.
As outlined in Section 3.5, an OOQS-Pipeline comprises three main applications. The first is the *Kafka-Consumer*, which serves as the interface between the pipeline and the Kafka Server. This application is responsible for receiving scientific packets that are loaded onto a Kafka Topic by ADAS. The Kafka-Consumer must receive and collect packets in batches, and then write them into an ordered stream of files for further processing.
The second and third applications in the pipeline are based on the *DQ-Pipe*, which is a framework developed by INAF. The DQ-Pipe provides a process synchronization mechanism that is based on the detection of new files within specified file system paths. It can also work with *Slurm*, which is the workload manager responsible for optimizing the load on the ASTRI on-site computing cluster. A configuration file is used to specify the operations that an instance of the DQ-Pipe must perform. We have extended the functionalities of the DQ-Pipe to make it a usable executor for the pipeline applications.
The second application of the pipeline is known as *DQ-Analysis*. It is responsible for carrying out a subset of the *OOQS Quality Checks* on the scientific data present in the batch files and generating new files containing the analysis results. DQ-Analysis is an external process executed whenever a new file is detected by the ASTRI-DQ-

Analysis: a new DQ-Pipe specialized class.
The third application is called *DQ-Aggregator*. Its primary function is to access the analysis result files, perform the necessary result aggregation, then upload the entire aggregated analysis result versus target *databases* such as MySQL, a relational database, and MongoDB, a NoSQL database. To do so, several specialization methods for the DQ-Pipe have been developed, such as the *ASTRI-DQ-Aggregator* class, and classes for opening the analysis results file and saving analysis results to the target database.

In section 4, it is described that the pipeline has been developed and tested on a server with performance similar to, or lower than, the one at the Array Observing Site. The Kafka-Consumer component was developed using highly specialized code to efficiently acquire and write HDF5 files while adhering to the maximum event rate of 1000Hz for scientific packets. Performance data indicate that it is possible to acquire the packets produced by a single camera at an event rate of up to 5800 Hz when writing files to HDD.
The performance of the DQ-Analysis and DQ-Aggregator applications is primarily measured during execution with Slurm. During a simulation of a real 30 minutes observation, the pipeline is capable of continuously and accurately processing all data batches acquired from Kafka, and efficiently inserting the aggregated data into all possible databases. Overall, the OOQS-Pipeline represents a feasible initial solution for implementing the final OOQS scientific pipeline.

The work described in this thesis contributes to the ASTRI Mini Array project, with the development of the first version of the OOQS-Pipeline, a software implementing a part of the OOQS: the Scientific Data Quality Checker of the CCDQC. The software will be installed by the end of June, at the Mini Array Observing Site, located at the Teide Astronomical Observatory, on Mount Teide in Tenerife (Canary Islands, Spain). The pipeline can receive and process data coming from a single Cherenkov Camera of the telescope array in real-time employing three different applications to realize the data processing workflow. The analysis results are then stored in a database.

Future developments

There are two possible future developments for the OOQS-Pipeline.
The first one involves testing and assessing the performance of the pipeline with the camera packets acquisition chain and the real Cherenkov Camera, which will be available in the coming months.
The second one concerns the development of the OOQS. The flexibility of the OOQS-Pipeline will allow for additional quality checks to be easily added to the DQ-Analysis application, although this will increase computation time. The DQ-Pipe and Slurm frameworks are capable of horizontally scaling the pipeline capabilities. Once all the quality checks are implemented, the OOQS will be able to generate feedback signals that must be delivered to other SCADA subsystems. In addition, the OOQS-Pipeline will be modified to implement the CCDQC sub-component for processing variance packets, known as the Variance Evaluator.

Bibliography

[1] S. Scuderi, A. Giuliani, G. Pareschi, et al. "The ASTRI Mini-Array of Cherenkov telescopes at the Observatorio del Teide". In: *Journal of High Energy Astrophysics* 35 (2022), pp. 52–68. ISSN: 2214-4048. DOI: https://doi.org/10.1016/j.jheap.2022.05.001. URL: https://www.sciencedirect.com/science/article/pii/S2214404822000180.

[2] Imre Bartos and Marek Kowalski. *Multimessenger Astronomy*. 2399-2891. IOP Publishing, 2017. ISBN: 978-0-7503-1369-8. DOI: 10.1088/978-0-7503-1369-8. URL: https://dx.doi.org/10.1088/978-0-7503-1369-8.

[3] Z. Cao, F.A. Aharonian, Q. An, et al. "Ultrahigh-energy photons up to 1.4 petaelectronvolts from 12 γ-ray Galactic sources". In: *Nature* 594 (2021), pp. 33–36.

[4] Stefan Funk. "Ground- and space-based gamma-ray astronomy". In: *Annual Review of Nuclear and Particle Science* 65.1 (2015), pp. 245–277. DOI: 10.1146/annurev-nucl-102014-022036.

[5] Tanmoy Laskar, Kate D. Alexander, Raffaella Margutti, et al. *The radio to Gev afterglow of GRB 221009A*. 2023. URL: https://arxiv.org/abs/2302.04388.

[6] Maia A. Williams, Jamie A. Kennea, S. Dichiara, et al. *GRB 221009A: Discovery of an Exceptionally Rare Nearby and Energetic Gamma-Ray Burst*. 2023. DOI: 10.48550/ARXIV.2302.03642. URL: https://arxiv.org/abs/2302.03642.

[7] P. Morrison. "On gamma-ray astronomy". In: *Nuovo Cim* 7 (1958), pp. 858–865. DOI: 10.1007/BF02745590.

[8] W. L. Kraushaar and G. W. Clark. "Search for Primary Cosmic Gamma Rays with the Satellite Explorer XI". In: *Phys. Rev. Lett.* 8 (3 1962), pp. 106–109. DOI: 10.1103/PhysRevLett.8.106. URL: https://link.aps.org/doi/10.1103/PhysRevLett.8.106.

[9] *High-energy cosmic gamma-ray observations from the OSO-3 satellite. - NASA technical reports server (NTRS).* DOI: doi:10.1086/151713. URL: https://ntrs.nasa.gov/citations/19730026226.

[10] W. C. Priedhorsky and S. S. Holt. "Long-term cycles in cosmic X-ray sources". In: *Space Science Reviews* 45.3 (1987). ID: Priedhorsky1987, pp. 291–348. ISSN: 1572-9672. DOI: 10.1007/BF00171997. URL: https://doi.org/10.1007/BF00171997.

[11] Neil Gehrels, Carl E. Fichtel, Gerald J. Fishman, et al. "The Compton Gamma Ray Observatory". In: *Scientific American* 269.6 (1993), pp. 68–77. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/24941727 (visited on 02/19/2023).

[12] David N. Burrows, J. E. Hill, J. A. Nousek, et al. *The swift X-ray telescope - space science reviews.* URL: https://link.springer.com/article/10.1007/s11214-005-5097-2.

[13] Winkler, C., T. J.-L. Courvoisier, Di Cocco, G., et al. "The INTEGRAL mission". In: *A&A* 411.1 (2003), pp. L1–L6. DOI: 10.1051/0004-6361:20031288. URL: https://doi.org/10.1051/0004-6361:20031288.

[14] A. De Angelis and M. Mallamaci. "Gamma-ray astrophysics". In: *The European Physical Journal Plus* 133.8 (2018), p. 324. ISSN: 2190-5444. DOI: 10.1140/epjp/i2018-12181-0. URL: https://doi.org/10.1140/epjp/i2018-12181-0.

[15] M. Tavani et al. "The AGILE Mission". In: *Astronomy & Astrophysical Journal* 502.3 (2009), pp. 995–1013. DOI: 10.1051/0004-6361/200810527. URL: https://doi.org/10.1051/0004-6361/200810527.

[16] A. Pellizzoni et al. "Discovery of new gamma-ray Pulsars with AGILE". In: *The Astrophysical Journal* 695.1 (2009), p. L115. DOI: 10.1088/0004-637X/695/1/L115. URL: https://dx.doi.org/10.1088/0004-637X/695/1/L115.

[17] W. B. Atwood, A. A. Abdo, M. Ackermann, et al. "The large area telescope on the fermi gamma-ray space telescope mission". In: *The Astrophysical Journal* 697.2 (2009), pp. 1071–1102. DOI: 10.1088/0004-637x/697/2/1071.

[18] I. Horváth, B. G. Tóth, J. Hakkila, et al. "Classifying GRB 170817A/GW170817 in a Fermi duration–hardness plane". In: *Astrophysics and Space Science* 363.3 (2018), p. 53. ISSN: 1572-946X. DOI: 10.1007/s10509-018-3274-5. URL: https://doi.org/10.1007/s10509-018-3274-5.

[19] B. P. Abbott, R. Abbott, T. D. Abbott, et al. "Multi-messenger Observations of a Binary Neutron Star Merger". In: *The Astrophysical Journal Letters* 848.2 (2017), p. L12. DOI: 10.3847/2041-8213/aa91c9. URL: https://dx.doi.org/10.3847/2041-8213/aa91c9.

[20] Thomas Gaisser and Francis Halzen. "IceCube". In: *Annual Review of Nuclear and Particle Science* 64.1 (2014), pp. 101–123. DOI: 10.1146/annurev-nucl-102313-025321. eprint: https://doi.org/10.1146/annurev-nucl-102313-025321. URL: https://doi.org/10.1146/annurev-nucl-102313-025321.

[21] Péter Mészáros, Derek B. Fox, Chad Hanna, et al. "Multi-messenger astrophysics". In: *Nature Reviews Physics* 1.10 (2019), pp. 585–599. ISSN: 2522-5820. DOI: 10.1038/s42254-019-0101-z. URL: https://doi.org/10.1038/s42254-019-0101-z.

[22] Lara Nava. "High-energy emission from gamma-ray bursts". In: *International Journal of Modern Physics D* 27.13 (2018), p. 1842003. URL: https://doi.org/10.1142/S0218271818420038.

[23] Frank M. Rieger, Emma de Oña-Wilhelmi, and Felix A. Aharonian. "TeV astronomy". In: *Frontiers of Physics* 8.6 (2013), pp. 714–747. ISSN: 2095-0470. DOI: 10.1007/s11467-013-0344-6. URL: https://doi.org/10.1007/s11467-013-0344-6.

[24] Boris M Bolotovskii. "Vavilov – Cherenkov radiation: its discovery and application". In: *Physics-Uspekhi* 52.11 (2009), p. 1099. DOI: 10.3367/UFNe.0179.200911c.1161. URL: https://dx.doi.org/10.3367/UFNe.0179.200911c.1161.

[25] M.V.S. Rao and B.V. Sreekantan. *Extensive Air Showers*. World Scientific, 1998. ISBN: 9789810228880. URL: https://books.google.it/books?id=t5UZ3mtJVzAC.

[26] Osvaldo Catalano, Milvia Capalbi, Carmelo Gargano, et al. "The ASTRI camera for the Cherenkov Telescope Array". In: *Ground-based and Airborne Instrumentation for Astronomy VII*. Ed. by Christopher J. Evans, Luc Simard, and Hideki Takami. Vol. 10702. International Society for Optics and Photonics. SPIE, 2018, p. 1070237. DOI: 10.1117/12.2314984. URL: https://doi.org/10.1117/12.2314984.

[27] Giovanni Pareschi, Giacomo Bonnoli, Stefano Vercellone, et al. "The mini-array of ASTRI SST-2M telescopes, precursors for the Cherenkov Telescope Array". In: *Journal of Physics: Conference Series* 718.5 (2016), p. 052028.

DOI: 10.1088/1742-6596/718/5/052028. URL: https://dx.doi.org/10.1088/1742-6596/718/5/052028.

[28] J. Kildea, R.W. Atkins, H.M. Badran, et al. "The Whipple Observatory 10m gamma-ray telescope, 1997–2006". In: *Astroparticle Physics* 28.2 (2007), pp. 182–195. ISSN: 0927-6505. DOI: https://doi.org/10.1016/j.astropartphys.2007.05.004. URL: https://www.sciencedirect.com/science/article/pii/S0927650507000746.

[29] R. Mirzoyan, R. Kankanian, F. Krennrich, et al. "The first telescope of the HEGRA air Cherenkov imaging telescope array". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 351.2 (1994), pp. 513–526. ISSN: 0168-9002. DOI: https://doi.org/10.1016/0168-9002(94)91381-1. URL: https://www.sciencedirect.com/science/article/pii/0168900294913811.

[30] A. Barrau, R. Bazer-Bachi, E. Beyer, et al. "The CAT imaging telescope for very-high-energy gamma-ray astronomy". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 416.2-3 (1998), pp. 278–292. DOI: 10.1016/s0168-9002(98)00749-9. URL: https://doi.org/10.1016%2Fs0168-9002%2898%2900749-9.

[31] R. Steenkamp. "H.E.S.S. - An Array of Gamma Ray Telescopes in Namibia". In: *African Skies* (2000).

[32] Petry D. *The Magic Telescope - prospects for GRB Research*. 1999. URL: https://doi.org/10.1051/aas:1999369.

[33] Badran H. Weekes T. C. and et al. *Veritas: The very energetic radiation imaging telescope array system*. 2002. URL: https://www.sciencedirect.com/science/article/abs/pii/S0927650501001529.

[34] Teresa Montaruli, Giovanni Pareschi, and Tim Greenshaw. *The small size telescope projects for the Cherenkov Telescope Array*. 2015. DOI: 10.48550/ARXIV.1508.06472. URL: https://arxiv.org/abs/1508.06472.

[35] The Cherenkov Telescope Array Consortium, B. S. Acharya, I. Agudo, et al. *Science with the cherenkov telescope array*. 2018. URL: https://arxiv.org/abs/1709.07997.

[36] Nicolò Parmiggiani. "Analysis methods and data management for real-time gamma-ray astrophysics." In: (2021).

[37] The Cherenkov Telescope Array Consortium, : H. Abdalla, et al. *Sensitivity of the Cherenkov telescope array for probing cosmology and fundamental physics with gamma-ray propagation*. 2021. URL: https://arxiv.org/abs/2010.01349.

[38] Enrico Giro, Rodolfo Canestrari, Pietro Bruno, et al. "The ASTRI-Horn telescope validation toward the production of the ASTRI Mini-Array: a proposed pathfinder for the Cherenkov Telescope Array". In: *Optics for EUV, X-Ray, and Gamma-Ray Astronomy IX*. Ed. by Stephen L. O'Dell and Giovanni Pareschi. Vol. 11119. International Society for Optics and Photonics. SPIE, 2019, 111191E. DOI: 10.1117/12.2530896. URL: https://doi.org/10.1117/12.2530896.

[39] Bulgarelli, Tosti, and Scuderi. *ASTRI Mini-Array Top Level Software Architecture*. URL: ASTRI-INAF-DES-2100-001.

[40] A. Daum, G. Hermann, M. Heß, et al. "First results on the performance of the HEGRA IACT array". In: *Astroparticle Physics* 8.1 (1997), pp. 1–11. ISSN: 0927-6505. DOI: https://doi.org/10.1016/S0927-6505(97)00031-5. URL: https://www.sciencedirect.com/science/article/pii/S0927650597000315.

[41] C Bigongiari, G Cusumano, F Di Pierro, et al. "Simulation of the ASTRI two-mirrors small-size telescope prototype for the Cherenkov Telescope Array". In: *Journal of Physics: Conference Series* 718.5 (2016), p. 052003. DOI: 10.1088/1742-6596/718/5/052003. URL: https://dx.doi.org/10.1088/1742-6596/718/5/052003.

[42] A. Bulgarelli, F. Lucarelli, G. Tosti, et al. "The Software Architecture and development approach for the ASTRI Mini-Array gamma-ray air-Cherenkov experiment at the Observatorio del Teide". In: *Software and Cyberinfrastructure for Astronomy VII*. Ed. by Jorge Ibsen and Gianluca Chiozzi. Vol. 12189. International Society for Optics and Photonics. SPIE, 2022, p. 121890D. DOI: 10.1117/12.2629164. URL: https://doi.org/10.1117/12.2629164.

[43] Bulgarelli, Tosti, and Scuderi. *ASTRI Mini-Array Top Level Use Cases*. URL: ASTRI-INAF-SPE-2100-001.

[44] Vito Conforti, Fulvio Gianotti, Valerio Pastore, et al. "The Array Data Acquisition System software architecture of the ASTRI Mini-Array project". In: *Software and Cyberinfrastructure for Astronomy VII*. Ed. by Jorge Ibsen and Gianluca Chiozzi. Vol. 12189. International Society for Optics and Photonics.

SPIE, 2022, 121890N. DOI: 10.1117/12.2626600. URL: https://doi.org/10.1117/12.2626600.

[45] Camera Team, S. Scuderi, and G. Pareschi. *AD8-ASTRI Camera BEE – Camera Camera Data Acquisition Interface Control Document of the ASTRI Cherenkov camera in the framework of the INAF program "Astronomia Industriale"*. URL: ASTRI-SPE-7350-001.

[46] N. Parmiggiani, A. Bulgarelli, L. Baroncelli, et al. "The online observation quality system software architecture for the ASTRI Mini-Array project". In: *Software and Cyberinfrastructure for Astronomy VII*. Ed. by Jorge Ibsen and Gianluca Chiozzi. Vol. 12189. International Society for Optics and Photonics. SPIE, 2022, 121892H. DOI: 10.1117/12.2629278. URL: https://doi.org/10.1117/12.2629278.

[47] N. Parmiggiani, Conforti V., A. Bulgarelli, et al. *ASTRI Mini-Array Online Observation Quality System Use Cases*. URL: ASTRI-INAF-SPE-9100-003.

[48] Gianluca Chiozzi, Birger Gustafsson, Bogdan Jeram, et al. "CORBA-based common software for the ALMA project". In: *Advanced Telescope and Instrumentation Control Software II*. Ed. by Hilton Lewis. Vol. 4848. International Society for Optics and Photonics. SPIE, 2002, pp. 43–54. DOI: 10.1117/12.461036. URL: https://doi.org/10.1117/12.461036.

[49] Valerio Pastore, Vito Conforti, Fulvio Gianotti, et al. "Array data acquisition system interface for online distribution of acquired data in the ASTRI Mini-Array project". In: *Software and Cyberinfrastructure for Astronomy VII*. Ed. by Jorge Ibsen and Gianluca Chiozzi. Vol. 12189. International Society for Optics and Photonics. SPIE, 2022, p. 1218924. DOI: 10.1117/12.2629922. URL: https://doi.org/10.1117/12.2629922.

[50] Simone Iovenitti, Giorgia Sironi, Enrico Giro, et al. "Assessment of the Cherenkov camera alignment through Variance images for the ASTRI telescope". In: *Experimental Astronomy* 53.1 (2021), pp. 117–132. DOI: 10.1007/s10686-021-09814-9. URL: https://doi.org/10.1007%5C%2Fs10686-021-09814-9.

[51] Ulla Kirch-Prinz and Peter Prinz. *A Complete Guide to Programming in C++*. 1st. Sudbury, MA: Jones and Bartlett Publishers, 2003.

[52] Robert Lafore. *Object-Oriented Programming in C++, Fourth Edition*. 4th. Indianapolis, IN: SAMS Publishing, 2018.

[53] Dave Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. 4th. Self-published, 2019. URL: `https://www.davekuhlman.org/python_book_01.pdf`.

[54] Slurm Workload Manager. *Job Scheduling and Resource Management System*. https://slurm.schedmd.com/overview.html. Accessed: February 25, 2023. 2021.

[55] Andy B Yoo, Mark A Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing* (2003), pp. 44–60.

[56] Neha Narkhede and Gwen Shapira adn Todd Palino. *Kafka: The Definitive Guide*. O'Reilly, 2017.