VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

HO CHI MINH UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

CAPSTONE PROJECT REPORT

BUILDING AN INTEGRATED DATABASE FOR SHORT TERM FORECASTING SYSTEMS IN HYDRO-METEOROLOGY

Major: Computer Science

THESIS COMMITTEE:	**Council 9**
SUPERVISORS:	**Lê Hồng Trang**
	Trương Quỳnh Chi
REVIEWER:	**Trương Quỳnh Chi**
	—o0o—
STUDENT 1:	**Trần Hà Tuấn Kiệt (2011493)**
STUDENT 2:	**Nguyễn Đức Thuỵ (2012158)**

Ho Chi Minh City, 5/2024

Declaration

We guarantee that this research is our own, conducted under the supervision and guidance of Assoc. Prof. Dr. Lê Hồng Trang. The result of our research is legitimate and has not been published in any form prior to this. All materials used within this research are collected by ourselves, by various sources and are appropriately listed in the references section. In addition, within this research, we also used the results of several other authors and organizations. They have all been aptly referenced. In any case of plagiarism, we stand by our actions and are to be responsible for it. Ho Chi Minh City University of Technologytherefore is not responsible for any copyright infringements conducted within our research.

Ho Chi Minh City, May 20, 2024

Team Members

Trần Hà Tuấn Kiệt

Nguyễn Đức Thuy

Acknowledgments

We, the research group consisting of two members, would like to send our thanks toward everyone who has contributed to the success of this thesis.

First and foremost, we would like to express our gratitude to Assoc. Prof. Dr. Lê Hồng Trang . The dedication and extensive knowledge of our supervisor not only guided us through the challenges of the project but also helped us develop research skills and critical thinking.

We also want to extend special thanks to all the friends, colleagues, and family who supported us throughout the research process. The input and opinions of everyone enriched the content and quality of the project.

Last, but certainly not least, we appreciate each other - research partners accompanying us in every step of the project. Our collaboration and joint contributions have resulted in a product that we take pride in.

We believe that this project marks a significant step in our development and could not have been achieved without the support and contributions of everyone involved.

Abstract

In this document, we introduce a comprehensive proposal to integrate, coordinate, and monitor meteorological and hydrological data in Vietnam. The proposed system can serve as a foundation for building a national database specializing in meteorological information.

To clarify our proposal, we have built a comprehensive pilot version, in which we have simulated the process of collecting and transmitting data from the weather station at Nha Be. At the same time, we focused on the process of converting and organizing data storage to ensure transparency and optimal performance in information processing.

This research is a manifestation of significant efforts in optimizing the management of meteorological data in Vietnam. The proposed system not only addresses the challenges of integration and coordination but also lays the foundation for a robust national database, serving the diverse needs of research and applications in the field of meteorology.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

In recognizing the need for National Meteorological Services (NMHSs) to improve their climate data and monitoring services, there is a deliberate focus on those aspects of climate data management wishing to make the transition to a modern climate database management system and, just as important, on what skills, systems and processes need to be in place to ensure that operations are sustained. In the context of the ever-growing complexity of climate change, the task of creating an integrated database for short-term forecasting systems in the fields of meteorology and hydrology poses a considerable challenge. The question at hand is how we can optimize the management of weather information from multiple sources and store it efficiently in a database. This optimization is crucial to ensure the provision of synchronized and high-quality information to support forecasting systems.

1.2 Motivation

Information about the weather has been recorded in manuscript form for many centuries. The early records included notes on extreme and, sometimes, catastrophic events and also on phenomena such as the freezing and thawing dates of rivers, lakes and seas, which have taken on a higher profile with recent concerns about climate change.

Specific journals for the collection and retention of climatological information have been used over the last two or three centuries (WMO 2005). The development of instrumentation to quantify meteorological phenomena and the dedication of observers to maintaining methodical,

reliable and well-documented records paved the way for the organized management of climate data. Since the 1940s, standardized forms and procedures gradually became more prevalent and, once computer systems were being used by NMHSs, these forms greatly assisted the computerized data entry process and consequently the development of computer data archives. The latter part of the twentieth century saw the routine exchange of weather data in digital form and many meteorological and related data centers took the opportunity to directly capture and store these in their databases. Much was learned about automatic methods of collecting and processing meteorological data in the late 1950s, a period that included the International Geophysical Year and the establishment of the World Weather Watch. The WMO's development of international guidelines and standards for climate data management and data exchange assisted NMHSs in organizing their data management activities and, less directly, also furthered the development of regional and global databases. Today, the management of climate records requires a systematic approach that encompasses paper records, microfilm/microfiche records and digital records, where the latter include image files as well as the traditional alphanumeric representation.

Before electronic computers, mechanical devices played an important part in the development of data management. Calculations were made using comptometers, for example, with the results being recorded on paper. A major advance occurred with the introduction of the Hollerith system of punch cards, sorters and tabulators. These cards, with a series of punched holes recording the values of the meteorological variables, were passed through the sorting and tabulating machines enabling more efficient calculation of statistics. The 1960s and 1970s saw several NMHSs implementing electronic computers and gradually the information from many millions of punched cards was transferred to magnetic tape. These computers were replaced with increasingly powerful mainframe systems and data were made available online through developments in disk technology.

Aside from advances in database technologies, more efficient data capture was made possible through the mid-to-late 1990s with an increase in automatic weather stations (AWSs), electronic field books (i.e. on-station notebook computers used to enter, quality control and transmit observations), the Internet and other advances in technology. Not surprisingly, there are a number of trends already underway that suggest there are many further benefits for NMHSs in managing data and servicing their clients. The Internet is already delivering greatly improved data access capabilities and, providing security issues are managed, we can expect major op-

portunities for data managers in the next five to ten years. In addition, Open Source7 relational database systems may also remove the cost barriers to relational databases for many NMHSs over this period.

1.3 Project Goal

It is essential that both the development of climate databases and the implementation of data management practices take into account the needs of the existing, and to the extent that it is predictable, future data users. While at first sight this may seem intuitive, it is not difficult to envisage situations where, for example, data structures have been developed that omit data important for a useful application or where a data centre commits too little of its resources to checking the quality of data for which users demand high quality.

In all new developments, data managers should either attempt to have at least one key data user as part of their project team or undertake some regular consultative process with a group of user stakeholders. Data providers or data users within the organization may also have consultative processes with end users of climate data (or information) and data managers should endeavour to keep abreast of both changes in needs and any issues that user communities have. Put simply, data management requires awareness of the needs of the end users.

At present, the key demand factors for data managers are coming from climate prediction, climate change, agriculture and other primary industries, health, disaster/emergency management, energy, natural resource management (including water), sustainability, urban planning and design, finance and insurance. Data managers must remain cognizant that the existence of the data management operation is contingent on the centre delivering social, economic and environmental benefit to the user communities it serves. It is important, therefore, for the data manager to encourage and, to the extent possible, collaborate in projects which demonstrate the value of its data resource. Even an awareness of studies that show, for example, the economic benefits from climate predictions or the social benefits from having climate data used in a health warning system, can be useful in reminding senior NMHS managers or convincing funding agencies that data are worth investing in. Increasingly, value is being delivered through integrating data with application models (e.g. crop simulation models, economic models) and so integration issues should be considered in the design of new data structures.

1.4 Project Scope

This project focuses on Ho Chi Minh City, a sprawling metropolis with a distinct climate that significantly impacts the daily lives of its residents. To provide the most valuable weather insights, we will implement a two-pronged approach.

Firstly, We will conduct research and collect data from the meteorological and hydrological monitoring stations in the city to ensure diversity and representation of local weather conditions. At the same time, we will proceed with preprocessing and standardizing the data to ensure accuracy and consistency of the dataset. Based on what we have collected, we will populate the data to proper formats and structures.

Secondly, this project entails the development of a comprehensive data platform. This platform will integrate a central database, designed to efficiently store and manage weather information from diverse sources. By consolidating these disparate data streams, we aim to create a unified and dependable resource for weather data. This platform is anticipated to significantly enhance the workflow of end-users and will be built with scalability in mind to accommodate future growth in data volume and user demand.

1.5 Stakeholders

1.5.1 The National Center for Hydro-meteorological Forecasting

The National Center for Hydrometeorological Forecasting (NCHMF), abbreviated for "Trung tâm Dự báo khí tượng thuỷ văn quốc gia" in Vietnamese, is an organizational unit under the General Department of Meteorology and Hydrology, Ministry of Natural Resources and Environment[4]. The National Hydro-Meteorological Forecasting Center has several crucial missions, including the establishment and presentation of standards and technical regulations for meteorological and hydrological forecasting, the operation of the national forecasting and warning system, monitoring and reporting on weather conditions and climate change, issuing and disseminating forecast bulletins and warnings, and participating in international meteorological agreements. Additionally, the center is responsible for conducting research, application, and technology transfer related to forecasting and warning, and implementing administrative reform and anti-corruption measures. These key missions contribute significantly to the center's

role in ensuring public safety and providing timely essential meteorological and hydrological information.

1.5.2 Nha Be Weather Radar Station

The Nha Be weather radar station is located on Nguyen Van Tao Street, Long Thoi Commune, Nha Be District, Ho Chi Minh City. This radar station has been in operation since 2004 and is managed by the Southern Region Hydro-Meteorological Center. Its mission is to monitor and supervise weather phenomena within a radius of 480 km from the center of Ho Chi Minh City, and to provide warnings and forecasts for dangerous weather conditions such as storms, tropical depressions, and thunderstorms within a radius of 300 km. Additionally, the station serves the purpose of forecasting rain and heavy rain for the Ho Chi Minh City area within a radius of approximately 120 km.

The Nha Be weather radar station is of the Doppler DWSR-2500C type, operating on the C-band frequency. It is manufactured by the U.S. company EEC and has the capability to scan signals within a radius of 300 km. In addition to being the forecasting center for the entire Southern region and Ho Chi Minh City, the Nhà Bè radar station also has the task of predicting rain and providing data for the flood control center of Ho Chi Minh City.

1.6 Solutions

Our product, the Weather Data Platform (WDP), is designed to be a one-stop shop for harnessing the power of weather data. It caters to the specific needs of a wide range of users, including meteorologists, academics, researchers, and developers from various fields. Whether you're a freelancer, a large corporation, or a non-profit organization, WDP can be your central hub for integrating, analyzing, and visualizing weather data – and much more.

1.7 Achievements and Contribution

Chapter 2

Theoretical basis

2.1 Database system and database management system

2.1.1 Traditional Database Systems

Database systems perform vital functions for all sorts of organizations because of the growing importance of using and managing data efficiently. A database system consists of a software, a database management system (DBMS) and one or several databases. DBMS is a set of programs that enables users to store, manage and access data. In other words, the database is processed by DBMS, which runs in the main memory and is controlled by the respective operating system.

A database is a logically coherent collection of data with some inherent meaning and represents some aspects of the real world. A random assortment of data cannot be referred to as a database. Databases draw a sharp distinction between data and information. Data are known facts that can be recorded and that have implicit meaning. Information is data that has been organized and prepared in a form that is suitable for decision-making. Shortly information is the analysis and synthesis of data. The most fundamental terms used in the database approach are identity, attributes and relationships. An entity is something that can be identified in the user's work environment, something that the users want to track. It may be an object with a physical or conceptual existence. An attribute is a property of an entity. A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Database Management System is a general-purpose software system designed to manage

large bodies of information facilitating the process of defining, constructing and manipulating databases for various applications. Specifying data types, structures and constraints for the data to be stored in the database is called defining a database. Constructing the database is the process of storing data itself on some storage medium that is controlled by the DBMS. Querying to retrieve specific data, updating the database to reflect changes and generating reports from the data is the main concept of manipulating a database. The DBMS functions as an interface between the users and the database ensuring that the data is stored persistently over long periods, independent of the programs that access it [10]. DBMS can be divided into three subsystems; the design tools subsystem, the run-time subsystem and the DBMS engine.

The design tools subsystem has a set of tools to facilitate the design and creation of the database and its applications. Tools for creating tables, forms, queries and reports are components of this system. DBMS products also provide programming languages and interfaces to programming languages. The run-time subsystem processes the application components that are developed using the design tools. The last component of DBMS is the DBMS engine which receives requests from the other two components and translates those requests into commands to the operating system to read and write data on physical media [3].

The database approach has several advantages over traditional file processing in which each user has to create and define files needed for a specific application. In these systems' duplication of data is generally inevitable causing wasted storage space and redundant efforts to maintain common data up-to-date. In the database approach data is maintained in a single storage medium and accessed by various users. The self-describing nature of database systems provides information not only about the database itself but also about the database structure such as the type and format of the data. A complete definition and description of database structure and constraints, called meta-data, is stored in the system catalog. Data abstraction is a consequence of this self-describing nature of database systems allowing program and data independence. DBMS access programs do not require changes when the structure of the data files is changed hence the description of data is not embedded in the access programs. This property is called program-data independence. Support of multiple views of data is another important feature of database systems, which enables different users to view different perspectives of databases depending on their requirements. In a multi-user database environment, users probably have access to the same data at the same time as well as they can access different portions of the

database for modification. Concurrency control is crucial for a DBMS so that the results of the updates are correct. The DBMS software ensures that concurrent transactions operate correctly when several users are trying to update the same data.

Using a DBMS also eliminates unnecessary data redundancy. In the database approach, each primary fact is generally recorded in only one place in the database. Sometimes it is desirable to include some limited redundancy to improve the performance of queries when it is more efficient to retrieve data from a single file instead of searching and collecting data from several files, but this data duplication is controlled by DBMS to prohibit inconsistencies among files. By eliminating data redundancy inconsistencies among data are also reduced [3]. Reducing redundancy improves the consistency of data while reducing the waste in storage space. DBMS allows data sharing to the users. Sharing data often permits new data processing applications to be developed without having to create new data files. In general, less redundancy and greater sharing lead to less confusion between organizational units and less time spent resolving errors and inconsistencies in reports. The database approach also permits security restrictions. In a DBMS different types of authorizations are accepted to regulate which parts of the database various users can access or update.

2.1.2 A Modern Approach to Data Systems

In contemporary computing environments, the dominance has shifted towards data-intensive applications, deviating from the traditional emphasis on compute-intensive tasks. The limiting factor for these applications seldom resides in the sheer computational power of the CPU; rather, the primary challenges typically revolve around the magnitude of the data, its intricate structures, and the rapidity with which it changes. Unlike compute-intensive operations that heavily rely on processing speed, data-intensive applications, dealing with extensive datasets, intricate data structures, or swiftly evolving information, necessitate adept strategies for storage, retrieval, and manipulation. Consequently, effectively addressing the multifaceted dynamics of data becomes paramount, highlighting the imperative for sophisticated data management and processing techniques to optimize performance in the face of these intricate challenges.

Why should we amalgamate these diverse elements within the overarching label of data systems? Recent years have witnessed the emergence of a plethora of novel tools for data storage and processing, each meticulously optimized for an array of distinct use cases [16]. Consider,

for instance, datastores that concurrently function as message queues (e.g., Redis) or message queues equipped with database-like durability assurances (such as Apache Kafka). The demarcation lines between these categories are progressively fading, reflecting a landscape where boundaries are increasingly ambiguous.

Moreover, a growing number of applications now present challenges of such magnitude or diversity that a solitary tool is no longer sufficient to fulfill all its data processing and storage requisites. Instead, the workload is deconstructed into tasks amenable to efficient execution by individual tools. These disparate tools are then intricately interwoven using application code, offering a nuanced and adaptable approach to the multifaceted demands of contemporary data management and processing.

In navigating the complex landscape of application development, the quest for reliability, scalability, and maintainability unveils a challenging yet essential journey. As we delve into the intricate patterns and techniques that permeate various applications, we embark on an exploration to fortify these foundational pillars in the realm of software systems.

Ensuring reliability involves the meticulous task of ensuring systems function correctly, even when confronted with faults. These faults may manifest in the hardware domain as random and uncorrelated issues, in software as systematic bugs that are challenging to address, and inevitably in humans who occasionally err. Employing fault-tolerance techniques becomes imperative to shield end users from specific fault types.

Scalability, on the other hand, necessitates the formulation of strategies to maintain optimal performance, particularly when facing heightened loads. To delve into scalability, it becomes essential to establish quantitative methods for describing load and performance. An illustrative example is Twitter's home timelines, which serve as a depiction of load, and response time percentiles providing a metric for measuring performance. In a scalable system, the ability to augment processing capacity becomes pivotal to sustaining reliability amidst elevated loads.

The facet-rich concept of maintainability essentially revolves around enhancing the working experience for engineering and operations teams interacting with the system. Thoughtfully crafted abstractions play a crucial role in mitigating complexity, rendering the system more adaptable and modifiable for emerging use cases. Effective operability, characterized by comprehensive insights into the system's health and adept management methods, also contributes to maintainability.

Regrettably, there exists no panacea for achieving instant reliability, scalability, or maintainability in applications. Nonetheless, discernible patterns and recurring techniques emerge across diverse application types, offering valuable insights into enhancing these critical attributes.

Data Transformation:

Data Transformation is a vital part of the data flow process, where data changes to meet specific requirements of the process or system. There are two main directions for implementing data transformation: batch processing and real-time processing.

In batch processing mode, data is processed in batches, often scheduled for processing at predefined intervals. This is suitable for tasks requiring the processing of large and complex datasets without an immediate response.

On the contrary, real-time processing is the method of processing data as soon as it arrives, without waiting for a large amount of data to accumulate. This is often preferred in applications demanding low latency, such as real-time event processing.

ETL:

ETL, an acronym for Extract, Transform, Load, stands as a fundamental methodology indispensable for orchestrating the seamless movement of data within storage ecosystems. This three-step process plays a pivotal role in shaping the lifecycle of data.

Firstly, in the "Extract" phase, data is sourced from diverse origins, ranging from databases to files and online services. This initial step lays the groundwork by retrieving relevant information from the varied reservoirs of data.

Subsequently, in the "Transform" phase, the extracted data undergoes a metamorphosis to align with the specific requirements of the target system. This transformative stage encompasses tasks such as data cleansing, format conversions, and even the computation of novel indices, ensuring the data is refined and tailored to suit its intended purpose.

Finally, the "Load" phase marks the culmination of the ETL process. At this juncture, the meticulously transformed data finds its destination, being loaded into the designated storage system. This storage system typically takes the form of a data warehouse or data lake, serving as the repository for the refined and purpose-adapted data. In essence, ETL encapsulates a

systematic and indispensable approach to managing the intricate journey of data within the expansive realm of storage systems.

Data Pipe:

A data pipe is an essential concept in deploying effective data flow. Built on the idea of automating the movement and transformation of data, data pipes serve as powerful workflow streams.

Through the use of data pipes, large data volumes can be processed flexibly and efficiently. Tasks such as error handling, performance monitoring, and even deploying new transformations can be automated, minimizing manual intervention and enhancing system stability.

Data Orchestration:

Data Orchestration is the intricate process of coordinating and managing multiple data processes, workflows, or services to achieve specific outcomes. At its core, it involves the meticulous definition and management of workflows, determining the sequential order, and dependencies among various data processes. Task execution is finely tuned through controllers that schedule and orchestrate tasks at optimal times and in a precise sequence, ensuring the desired outcomes are achieved. Controllers play a pivotal role in managing dependencies between tasks, orchestrating the execution of tasks only when their dependent tasks are successfully met. Robust orchestration systems provide comprehensive tools for monitoring workflow progress and logging pertinent information, offering insights to address and rectify any potential issues. Moreover, controllers optimize performance by strategically breaking down complex tasks into multiple subtasks, executing them in parallel to enhance overall system efficiency.

2.2 Basic of meteorology

Meteorology is the scientific study of the atmosphere that focuses on weather processes and forecasting. It involves the study of the behavior, dynamics, and physical properties of the earth's atmosphere and how these factors affect the weather and climate. Meteorologists use various tools and techniques, including weather satellites, radars, and computer models, to predict weather conditions and to understand the complex interactions within the atmosphere.

Meteorology is important for several reasons, such as predicting weather to warn of severe conditions, understanding climate change, and aiding in decision-making for agriculture, aviation, and other industries dependent on weather.

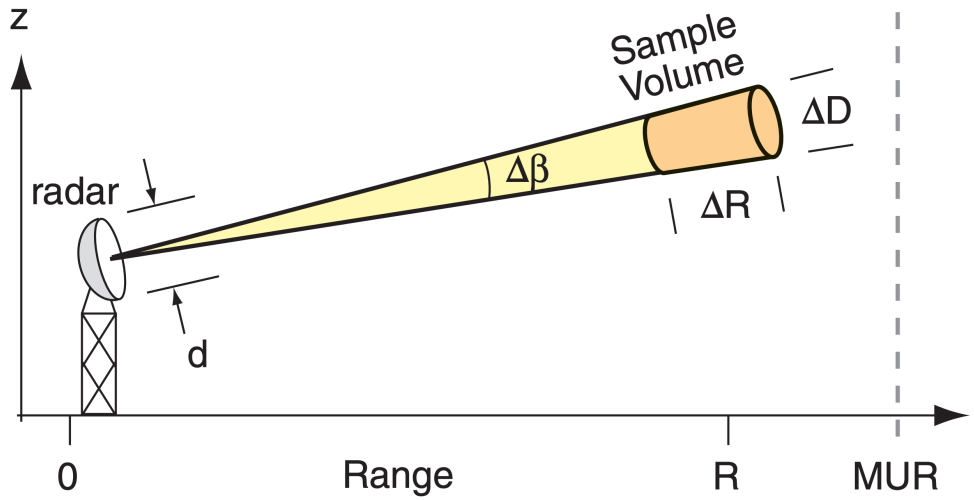


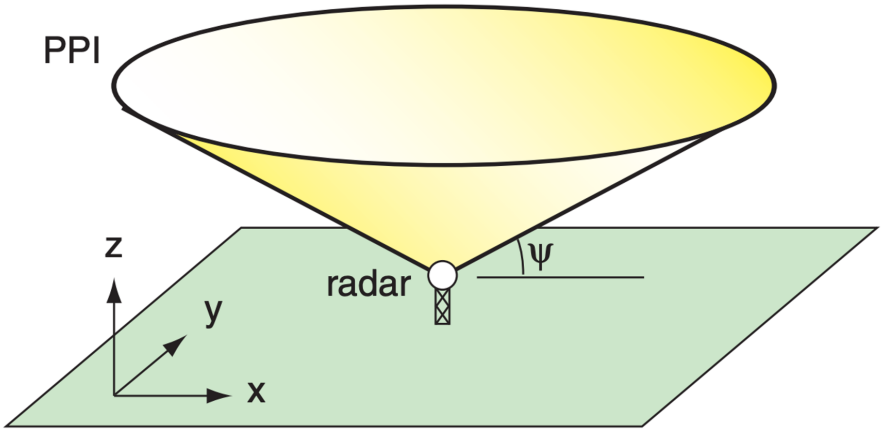
Figure 2.1: *A typical meteorological radar [17]*

2.2.1 Basic terminologies

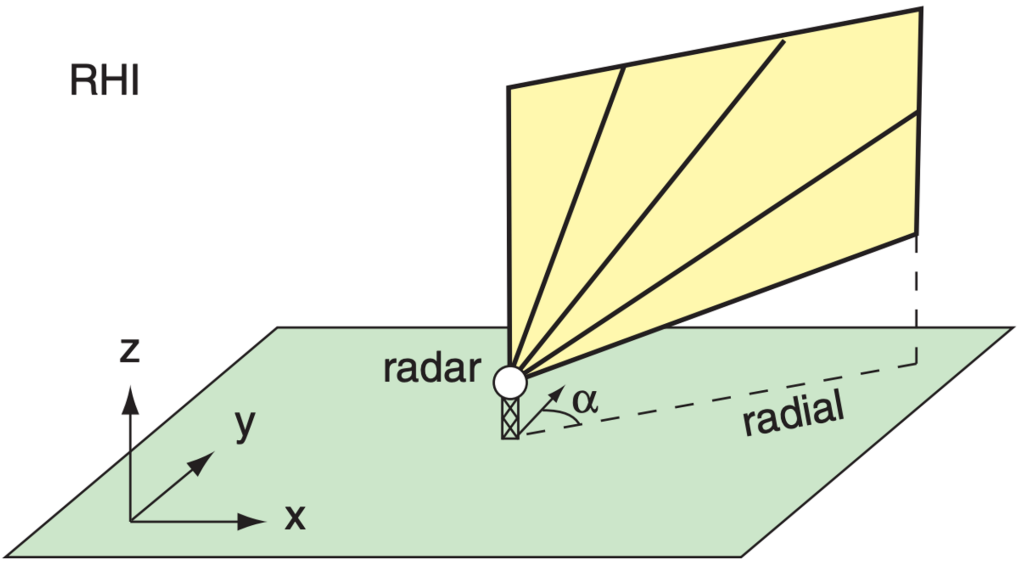
2.2.1.1 Weather Radar

Weather radar, short for weather surveillance radar, is a type of radar system used to detect and monitor precipitation, as well as other atmospheric phenomena such as the movement of severe weather systems. It plays a crucial role in meteorology and helps meteorologists track and forecast weather conditions. Normally, weather radars are programmed to scan in an azimuth of 360^o. For every round, the radar will scan at a different altitude. It usually takes about four to ten minutes for the radar to complete a full scan.

For PPI representation, the radar will scan the entire azimuth, but only at a certain altitude. The final result would be similar to a map on a flat surface. For RHI, in contrast, the radar retains the azimuth but increases in altitude. The collected result gives viewers more details about the height and sizes of a meteorologist event.



(a) *Plan-Position Indicator - PPI - [17]*



(b) *Range Height Indicator - [17]*

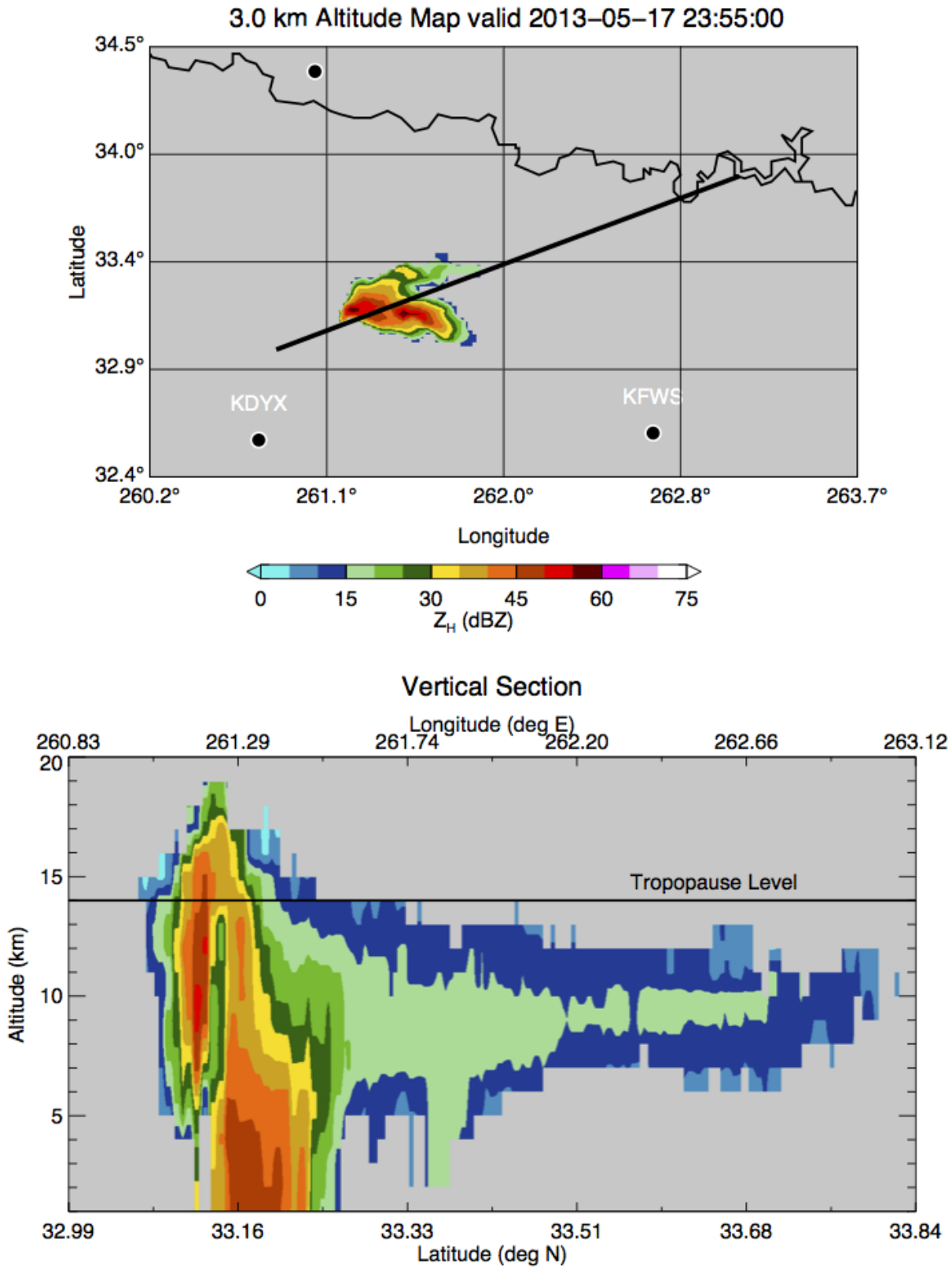


Figure 2.3: *Comparing the result between PPI and RHI - [2]*

2.2.1.2 Radar equation and Reflectivity

At a certain point in time, weather radar will emit a short pulse of radio wave ($\Delta t = 0.5 - 10\mu s$). Depending on the density of free molecules in the air (water vapor, smoke, ...), the energy of this wavelength will be partially absorbed. The wavelength intensity that the radar receives will be less than the intensity of the original wave. This ratio is expressed through **The radar equation** [17]:

$$\left[\frac{P_R}{P_T}\right] = [b] \cdot \left[\frac{|K|}{L_a}\right]^2 \cdot \left[\frac{R_1}{R}\right]^2 \cdot \left[\frac{Z}{Z_1}\right]$$

Which, the variables of the equation include:

- $|K|$ unitless:
  - $|K|^2 \approx 0.93$ for droplets
  - $|K|^2 \approx 0.208$ for ice crystal
- $R(\text{km})$: distance from the radar to the target
- $R_1 = \sqrt{Z_1 \cdot c \cdot \Delta t / \lambda^2}$: ratio of distance
- Z: Radarś reflectivity
- $Z_1 = 1\ \text{mm}^6\ \text{m}^{-3}$: Radar's unit reflectivity

From the radar equation, we can derive the formula for reflectivity:

$$\text{dBZ} = 10\left[\log\left(\frac{P_R}{P_T}\right) + 2\log\left(\frac{R}{R_1}\right) - 2\log\left|\frac{K}{L_a}\right| - \log(b)\right]$$

Meteorologists are usually interested in this number because it is proportional to the amount of precipitation.

Value (dBZ)	Weather
-28	Haze
-12	Clear air
25 - 30	Dry snow / light rain
40 - 50	Heavy rain
75	Giant hail

Table 2.1: *Relation between reflectivity and precipitation - Stull [17]*

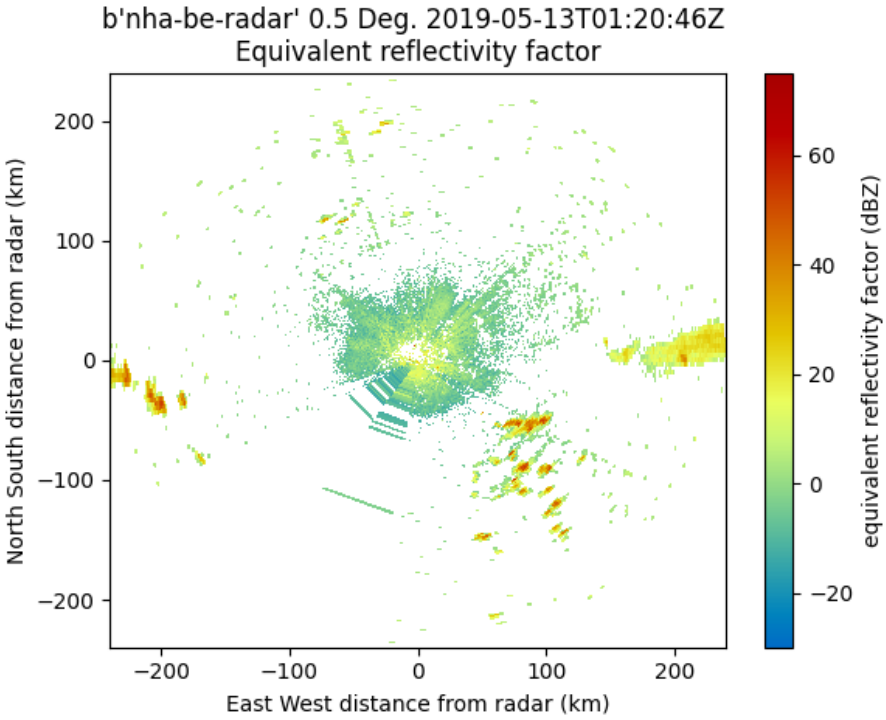


Figure 2.4: *Reflectivity from Nha Be radar*

2.2.1.3 Radial velocity

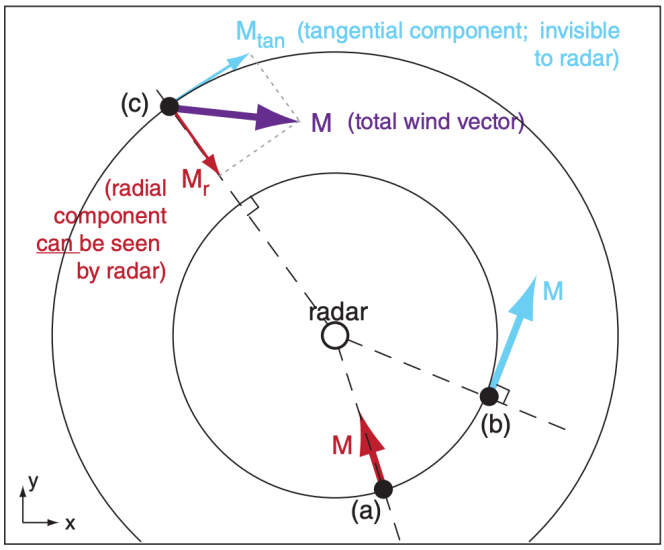


Figure 2.5: *Illustration for the velocity situations that a Doppler radar can observe. (a) When the wind direction at point M coincides with the radius of the circle centered at the radar, the radar can determine the velocity at this point. (b) When the wind direction is tangent to the circle, the radar cannot determine the velocity. (c) Analyzing the wind direction at M into two perpendicular velocities, the radar can only determine the velocity vector along M_r. - Stull [17]*

When the radio waves from these Doppler radars propagate to the molecules in the air, the displacement of these particles causes a phase shift between the transmitted and received signals. Radars rely on this information to calculate the wind velocity at various points in space.

2.2.2 Radar Products

2.2.2.1 Echo Top Height (ETH)

An echo top, in radar meteorology, signifies the highest altitude at which precipitation particles are detected. Essentially, it helps us pinpoint the maximum elevation angle where the intensity of precipitation, quantified by reflectivity, surpasses a predetermined threshold. This information is crucial for understanding the vertical extent of precipitation systems and their potential impact.

Overview

The modified ETH algorithm [9], can be applied to a NEXRAD PPI volume scan by following these steps:

1. **Finding Echo Top Height:**

   Locate the maximum elevation angle, denoted by θ_b, where the reflectivity, Z_b, surpasses a predefined echo-top reflectivity threshold, Z_T (e.g., 0 dBZ, 18 dBZ). If θ_b is not the highest elevation scan available in the volume, obtain the reflectivity value, Z_a, at the subsequent higher elevation angle, θ_a. Employ the following equation to calculate the echo-top height, θ_T:

$$\theta_T = \frac{(Z_T - Z_a)(\theta_b - \theta_a)}{Z_b - Z_a} + \theta_b$$

2. **Handling Highest Elevation Scan:**

   If θ_b coincides with the highest elevation scan accessible, set $\theta_T = \theta_b + \beta/2$, where β represents the half-power beamwidth. This scenario arises when:

   - Far from the radar: Higher elevation scans possess shorter ranges compared to a baseline "surveillance" scan.
   - Very close to the radar: The highest elevation scan fails to capture the cloud's peak.

   Under these circumstances, θ_T corresponds to the top of the beam containing reflectivity greater than or equal to Z_T. In essence, the traditional echo-top computation is adopted when no data exists from higher elevation scans.

   This mathematical formulation can be expressed in LaTeX as:

$$\theta_T = \begin{cases} \frac{(Z_T - Z_a)(\theta_b - \theta_a)}{Z_b - Z_a} + \theta_b & \text{if } \theta_b \text{ is not highest elevation scan} \\ \theta_b + \frac{\beta}{2} & \text{if } \theta_b \text{ is highest elevation scan} \end{cases}$$

   This equation incorporates a conditional statement to account for the two scenarios based on the availability of data from higher elevation scans.

2.3 Common Data format in meteorology

2.3.1 SIGMET data format - raw format (Vaisala)

Vaisala, a Finnish company renowned for its expertise in environmental and meteorological instrumentation, has developed the RAW format, also known as SIGMET [11], as a specialized storage format for organizing data output from its radar devices. This format represents a meticulously designed system tailored to efficiently manage the voluminous data generated by radar scanning sessions.

Distinctive features characterize the architecture of the RAW format. Notably, the file content is partitioned into discrete blocks, each precisely sized at 6144 bytes. This particular sizing aligns with historical conventions rooted in the main storage capacities of older tape devices, reflecting a pragmatic approach to data organization and storage optimization. Furthermore, the RAW format typically consolidates data from multiple radar scanning sessions within a single file, fostering data aggregation and accessibility.

Within each block of the RAW format, data records are meticulously structured, ensuring efficient utilization of storage space. In instances where space within a block remains unoccupied after accommodating data records, padding with additional zeros is employed, maintaining structural integrity and facilitating streamlined data retrieval processes.

The adoption of the RAW format confers several notable advantages, as elucidated by empirical analyses and industry insights [18]. Firstly, the format demonstrates inherent compatibility with a diverse array of tape types, a legacy consideration owing to the historical prevalence of tape-based storage systems. Despite the evolution of storage technologies, tapes persist as cost-effective alternatives, underscoring the ongoing relevance and practicality of the RAW format in contemporary data management contexts. Moreover, the block-oriented architecture of the RAW format facilitates robust error recovery mechanisms at the storage system level, enhancing data reliability and resilience against potential hardware failures.

Despite its merits, concerns persist regarding the alignment of the storage structure delineated by the RAW format with the corresponding mappings on hard drives and tape devices. Addressing these concerns necessitates meticulous attention to compatibility and interoperability considerations, ensuring seamless data interchangeability across heterogeneous storage environments.

In essence, the RAW format epitomizes Vaisala's commitment to innovation and excellence in the domain of environmental instrumentation. Its tailored design and inherent advantages render it a formidable tool for efficiently managing and organizing radar data, thereby empowering researchers and practitioners in meteorology and related fields with robust data management capabilities.

2.3.2 NETCDF data format - Network Common Data Form

The Network Common Data Form (NetCDF) [14] represents a pivotal file format meticulously engineered for the purpose of accommodating multidimensional scientific data. Embedded within the framework of the NetCDF library system are a variety of binary formats, each strategically contributing to the overarching flexibility and scalability of data management within scientific domains. These formats, which have evolved over iterations of the NetCDF library system, are emblematic of its adaptability to diverse data requirements and computational environments.

```
→  titan2023 ncdump -h radar.nc
netcdf radar {
dimensions:
        time = UNLIMITED ; // (1748 currently)
        range = 1198 ;
        sweep = 5 ;
        string_length = 32 ;
```

Figure 2.6: *Radar information in NETCDF format. The total dimensions of the dataset are 2975, grouped into 4 distinct labels.*

Foremost among these formats is the Classic Format, which has been a mainstay since the inception of the NetCDF library and continues to serve as the default option for file creation. Over time, subsequent releases of the NetCDF library system have introduced additional formats, each with distinct features tailored to specific demands. The 64-bit Offset Format, for instance, emerged with version 3.6.0, addressing the need for expanded variable and file sizes beyond the limitations of its predecessors.

In a similar vein, the integration of the netCDF-4/HDF5 Format with the release of version 4.0 represents a milestone in the evolution of the NetCDF ecosystem. By harnessing the capa-

bilities of the Hierarchical Data Format version 5 (HDF5), this format bridges the gap between the NetCDF and HDF5 data models, offering users enhanced capabilities such as support for unlimited dimensions and substantially larger file sizes. Furthermore, the incorporation of the HDF4 SD Format and the CDF5 Format underscores the NetCDF ecosystem's commitment to interoperability and compatibility with parallel processing frameworks.

Central to the design philosophy of NetCDF formats is the concept of self-description, wherein comprehensive metadata, including attribute name/value pairs and data array structures, are encapsulated within the file header. This design not only fosters platform independence but also facilitates seamless data exchange and interoperability across diverse computational environments.

To illustrate the versatility and efficacy of the NetCDF format, consider a practical scenario involving the storage of meteorological parameters, including temperature, humidity, pressure, wind speed, and direction, within NetCDF files. This exemplifies the format's capacity to accommodate complex, multidimensional datasets characteristic of scientific research domains, thereby empowering researchers with a robust and flexible tool for data management and analysis.

Furthermore, it is noteworthy that NetCDF Classic and 64-bit Offset Formats have garnered international recognition as standards endorsed by the Open Geospatial Consortium (OGC), underscoring their role as reliable and interoperable solutions for geospatial data management [13]. This validation not only speaks to the robustness and reliability of the NetCDF format but also underscores its global scalability and compatibility with established standards and practices in the scientific community.

2.4 Related technologies

2.4.1 Apache Airflow™

Apache Airflow™ stands as an open-source platform designed to manage data flow within systems associated with data. In the face of the escalating challenge of data pipeline management, Airflow emerges as a comprehensive solution, automating and optimizing data-related workflows effectively [1].

Airflow not only aids in defining and managing the start and end times of each data pipeline

but also provides precise and detailed monitoring of the results of each task. This becomes particularly crucial when ensuring the integrity and reliability of the processed data.

With the ability to discern complex relationships between tasks through the Directed Acyclic Graph (DAG) model, Airflow empowers administrators with tighter control and flexibility in handling workflow processes. Its robust integration with logging systems facilitates detailed activity tracking, assisting in issue resolution and ensuring that every process aligns with expectations.

Simultaneously, the scheduling flexibility makes Airflow an excellent tool for time and resource management. Its strong integration with various data sources and extensibility through plugins allows Airflow to meet diverse needs in data processing and task automation.

Apache Airflow not only delivers robust performance but also brings flexibility and optimal technical features to data processing workflows. With its time management capabilities, powerful logging integration, scheduling flexibility, and scalability, Airflow stands as the top choice for enhancing performance and control in data processing workflows.

2.4.2 Kubernetes

Kubernetes, an open-source system for managing and deploying highly flexible applications in cloud and data center environments, has evolved into one of the most widely adopted tools in the field of Information Technology [8]. Originally developed by Google and later transferred to the Cloud Native Computing Foundation (CNCF), Kubernetes aims to automate the deployment, scaling, and management of containerized applications, alleviating the burden on developers and system administrators. The platform offers a unified foundation for deploying, scaling, and managing containerized applications across multiple servers.

Kubernetes operates based on key concepts such as Pods, Services, ReplicaSets, and various other abstractions, creating a flexible environment for application deployment and management. This fosters an environment where developers can easily build applications, and system administrators can efficiently maintain them.

Beyond supporting traditional deployment models, Kubernetes paves the way for innovative strategies like Continuous Deployment (CD) and Microservices. With the ability to automate many aspects of the development and deployment process, Kubernetes plays a crucial role in constructing and sustaining complex, flexible, and scalable systems.

2.4.2.1 Kubernetes Components

Introducing essential concepts for managing and deploying applications, Kubernetes provides an effective and flexible environment. The main components of Kubernetes include Pod, ReplicaSet, Deployment, and Service.

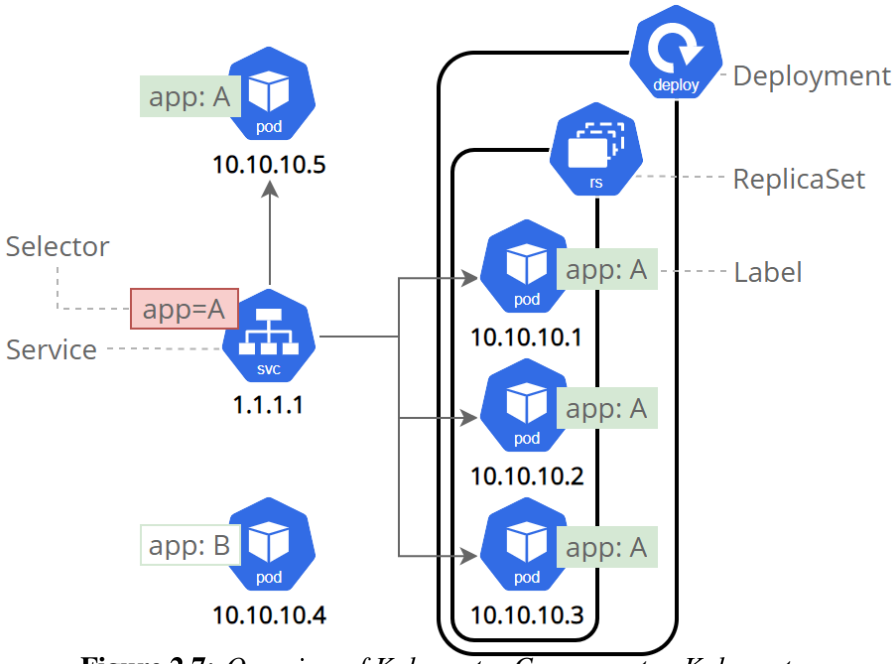


Figure 2.7: *Overview of Kubernetes Components - Kubernetes*

In Kubernetes, a **Pod** serves as the fundamental unit, representing a collection of containers that share a common workspace. Within the same Pod, containers collaborate by sharing network and storage resources, fostering interaction and enabling the construction of intricate applications.

The **ReplicaSet**, a crucial resource in Kubernetes, ensures a designated number of Pods operate in a specified manner. In the event of a Pod failure or shutdown, the ReplicaSet automatically initiates the creation of a new Pod to replace it. This mechanism ensures the application's stable state by guaranteeing a defined number of Pods are consistently operational.

For managing the deployment and updating processes of applications, **Deployment** is a key component in Kubernetes. It articulates the desired state of the application and orchestrates the updating of the ReplicaSet to achieve that state. Deployment provides versatile management capabilities, facilitating the deployment of new versions, rollbacks, and updates without disrupting the service.

The **Service** resource in Kubernetes furnishes an HTTP port to Pods, generating a unique

IP address and DNS name for a cluster of Pods. This enables seamless communication among applications within the cluster and with external environments. Service effectively simplifies the intricacies of handling multiple Pods and IP addresses, offering a straightforward means of accessing services within the Kubernetes environment.

Typically, large-scale systems leverage Kubernetes in their software development and deployment processes. This adoption brings several advantages, including efficient resource management and self-recovery capabilities. Kubernetes optimizes resource utilization, ensuring optimal performance and reducing waste. Additionally, it automatically addresses issues during operations, enhancing high availability.

However, the technology is not without its challenges, including a steep learning curve for beginners. Mastery of diverse knowledge areas such as computer networking and containerization is necessary. Moreover, deploying and maintaining Kubernetes demands significant resources, both in terms of personnel and hardware, particularly for smaller organizations.

2.4.2.2 High Availability in Kubernetes

High Availability is a crucial factor in the success of any system. In Kubernetes, High Availability is achieved through the combination of several features, including self-healing, load balancing, and auto-scaling.

First, Kubernetes uses **Deployment**, a type of **Controller** for managing replicas. Through Deployment, we can easily perform horizontal scaling, which is the process of increasing the number of replicas of a Pod. This mechanism ensures that the application can handle numerous requests without compromising performance. Moreover, in case of a Pod failure, a Deployment makes sure that a new Pod is created to replace it, ensuring the amounts of predefined replicas is always maintained.

At network layers, Kubernetes uses **Service** for communication between various components within or outside the cluster. Instead of directly accessing Pods, other components can access Services, which will redirect the request to the appropriate Pod. When Pods are replaced, the Service will automatically update the routing rules to ensure the request is sent to the correct Pod. Outside the cluster, Kubernetes also uses **Ingress** to manage external access to Services. Instead of specifying the direct node IP address, Clients can abstract it by using only the URL or hostname.

Finally, at the storage level, Kubernetes provides Persistent Volumes. By doing so, applications can be agnostic to the underlying storage infrastructure. This allows for easier management and scaling of storage resources. Not only that, depending on the provided **Storage Class**, Kubernetes makes sure that the data is replicated to multiple nodes, ensuring data availability in case of node failure.

To summarize, Kubernetes provides a robust set of features to ensure High Availability. By leveraging these features, we can build a highly available system that can scale with our load, while also being resilient to failures.

2.4.3 LROSE

LROSE (Lidar Radar Open Software Environment) is a project supported by the National Science Foundation (NSF) with the goal of developing common software for the Lidar, Radar, and Profiler community. The project operates based on the principles of collaboration and open source. The core software package of LROSE is a collaborative effort between Colorado State University (CSU) and the Earth Observing Laboratory (EOL) at the National Center for Atmospheric Research (NCAR) [7].

Originating from the need for a unified software environment for processing Lidar and Radar data in atmospheric science research [7], the project addresses complexities related to integrating data from various observation platforms, including Lidar, Radar, and Profiler. These components are designed to meet the specific needs of meteorologists and researchers working with remote sensor data.

LROSE is widely used in meteorological research, including studies related to cloud and precipitation processes, boundary layer dynamics, and other meteorological phenomena. The software supports the analysis of observation data collected from ground-based tools such as Lidar and Radar. LROSE seamlessly integrates with a variety of model and atmospheric analysis tools to optimize its capabilities. Researchers often integrate LROSE into numerical weather prediction models as well as other data assimilation techniques, creating a flexible and powerful system.

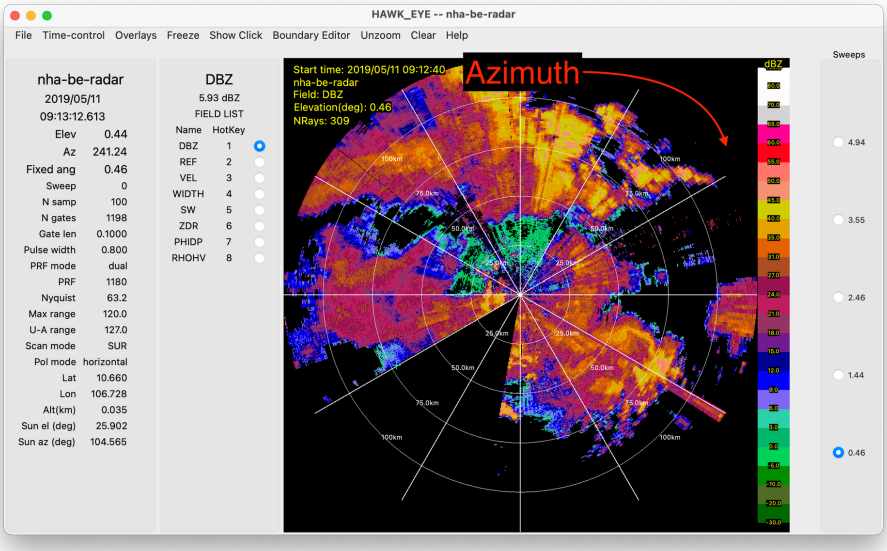


Figure 2.8: *Hawk Eye, Lidar and Radar visualization tool of LROSE*

The project actively encourages participation from a large scientific community, promoting the exchange of ideas, algorithms, and improvements for the software. Regular updates and contributions from users contribute to the continuous development and refinement of LROSE.

2.4.4 Py-ART

In the realm of meteorology, weather radar plays a crucial role in understanding and forecasting precipitation, cloud cover, and other atmospheric phenomena. Extracting meaningful information from this data requires specialized tools and techniques. Py-ART (the Python ARM Radar Toolkit) is an open-source Python library designed specifically for working with weather radar data.

Developed by the Atmospheric Radiation Measurement (ARM) Climate Research Facility, Py-ART offers a comprehensive suite of algorithms and utilities for researchers and atmospheric scientists. However, its user-friendly design and extensive functionalities make it valuable for anyone working with weather radar data, from meteorologists to students.

2.4.4.1 Core functionalities

Py-ART transcends its role as a simple data repository. It empowers users with a comprehensive suite of functionalities, transforming raw weather radar data into meaningful insights. At its core, Py-ART offers a robust toolbox for data manipulation, analysis, and visualization,

catering to the diverse needs of meteorologists, researchers, and anyone working with weather radar information.

The journey begins with seamless data access. Py-ART supports a wide range of file formats commonly used in atmospheric research. This versatility allows users to import data from various sources, eliminating compatibility hurdles and streamlining the workflow. Once the data is loaded, Py-ART's processing capabilities come into play. Real-world radar measurements are not without imperfections. Signal attenuation, caused by factors like distance and intervening obstacles, weakens the returning signal. Py-ART provides algorithms to correct for this attenuation, ensuring the accuracy of the retrieved information. Furthermore, raw radar data often contains noise – unwanted electrical disturbances that can distort the signal. Py-ART offers a variety of filtering techniques to remove this noise, resulting in a cleaner and more reliable dataset. Calibration, a crucial step in ensuring data integrity, is also made possible by Py-ART. By applying calibration techniques, users can account for systematic biases inherent in the radar system, leading to more precise measurements.

Once the data is cleaned and processed, Py-ART shines in its ability to visualize and analyze this information. It seamlessly integrates with popular scientific Python libraries like Matplotlib. This powerful combination allows users to create informative and visually compelling representations of the data. Imagine generating high-resolution reflectivity maps that paint a vivid picture of precipitation intensity across a region. Py-ART facilitates this by converting reflectivity data into maps, allowing for easy identification of areas with heavy rain or snowfall. Beyond maps, Py-ART enables the creation of vertical profiles, which depict how reflectivity and other parameters vary with altitude. This provides a detailed understanding of the vertical structure of storms and precipitation events. Analyzing these visualizations in conjunction with environmental data, which Py-ART can also incorporate, allows researchers to delve deeper into the atmospheric processes at play.

Py-ART's functionalities extend beyond basic visualization. It offers advanced analysis tools for tasks like feature detection and storm tracking. Imagine automatically identifying and tracking the movement of severe weather features like hailstorms or tornadoes within radar data. Py-ART's algorithms can accomplish this, providing crucial information for issuing timely weather warnings and protecting lives. Perhaps the most impactful functionality lies in quantitative precipitation estimation (QPE). By analyzing radar data and incorporating environmental factors,

Py-ART can estimate the amount of precipitation that has fallen over a specific area. This information is invaluable for flood forecasting, water resource management, and understanding overall precipitation patterns.

2.4.4.2 Benefits

Py-ART offers a range of benefits for working with weather radar data, making it a compelling choice for meteorologists and atmospheric scientists alike. Firstly, Py-ART is open-source and freely available, allowing anyone to access, download, and modify its code. This fosters collaboration and innovation within the atmospheric science community, as researchers can contribute improvements and share their work with others.

Additionally, Py-ART boasts cross-platform compatibility, functioning seamlessly on various operating systems including Windows, macOS, and Linux. This ensures wider accessibility and flexibility for users across different computing environments.

The modular design of Py-ART is another key advantage, allowing users to leverage specific functionalities they need for their radar data processing tasks. This modular approach enhances efficiency and adaptability, as users can tailor their workflow to suit their requirements.

Moreover, Py-ART provides extensive documentation and a rich collection of examples, which serve to ease the learning curve for new users. The availability of comprehensive resources empowers users to quickly familiarize themselves with the library's capabilities and effectively utilize its features for their research or operational needs.

2.4.5 Comparing between the tools of LROSE and PyART for Meteorology Data Processing

In meteorology, data processing is essential for interpreting and understanding atmospheric phenomena. While both tools offer robust functionalities for radar and lidar data processing, PyART stands out due to its seamless integration with Python, making it particularly powerful for researchers and developers who leverage Python's extensive scientific ecosystem. This section compares these two tools across several dimensions: functionality, ease of use, extensibility, community support, and performance, with a focus on PyART's advantages.

2.4.5.1 Functionality

PyART offers a comprehensive suite of tools and features for radar data analysis. It supports data input and output in multiple formats, including NetCDF, HDF5, and MDV, ensuring compatibility with various data sources. In terms of processing algorithms, PyART provides robust options for data correction such as dealiasing and attenuation correction, alongside filtering and moment calculation. For visualization, PyART integrates seamlessly with Matplotlib, enabling users to create detailed 2D and 3D visual representations like PPI (Plan Position Indicator) and RHI (Range Height Indicator) plots. Furthermore, PyART is built on the SciPy ecosystem, which facilitates easy integration with other Python scientific libraries such as NumPy, Pandas, and Scikit-learn, thus significantly enhancing its analytical capabilities.

On the other hand, LROSE-Core is also an extensive suite aimed at standardizing and enhancing radar and lidar data processing capabilities. Developed as an open-source initiative, it emphasizes interoperability and advanced processing techniques. The suite supports a wide range of radar and lidar data formats, including proprietary ones, making it highly versatile. LROSE-Core includes sophisticated signal processing algorithms such as dual-polarization processing, clutter filtering, and Doppler velocity analysis. For visualization, it offers advanced tools through its HawkEye and CIDD applications, which provide interactive and customizable plotting capabilities. Additionally, the toolset is designed to handle real-time data processing, which is crucial for operational radar networks.

2.4.5.2 Ease of Use - Learning Curve

PyART is designed with simplicity in mind, making it accessible to users with basic programming knowledge. The API is well-documented, and the library includes numerous examples and tutorials. Its seamless integration with Python's scientific stack (NumPy, XArray, and Matplotlib, ...) makes it a natural choice for researchers already familiar with these tools. Python's ease of use and readability further enhance PyART's appeal, allowing users to quickly prototype and deploy analysis workflows.

LROSE-Core, while powerful, has a steeper learning curve compared to PyART. It requires a more in-depth understanding of radar and lidar data processing principles. The installation process can be more complex, especially for users unfamiliar with building software from source. However, the detailed documentation and active community support can help mitigate these

challenges.

2.4.5.3 Extensibility

PyART's modular design makes it highly extensible. Users can easily incorporate their own algorithms or modify existing ones. Its reliance on Python ensures that it can leverage the extensive range of available libraries for additional functionality. This extensibility makes PyART particularly powerful, as users can integrate machine learning libraries such as TensorFlow or Scikit-learn to develop advanced predictive models directly within their radar data processing workflows.

LROSE-Core is also designed for extensibility, with a focus on providing a comprehensive platform for radar and lidar data processing. Its open-source nature allows users to contribute new algorithms and features. However, extending LROSE-Core may require more specialized knowledge in C++ and radar/lidar processing techniques, potentially limiting its accessibility compared to PyART.

2.4.6 ASP.NET Core

ASP.NET Core is a free, cross-platform, open-source framework for building modern, cloud-based, internet-connected applications. It is particularly well-suited for developing API servers that need to interact with databases, thanks to its robust set of features, high performance, and strong community support.

There are many reasons to believe that ASP.NET and C# can be a good candidate for building server in managing an integrated database system. The following sections explain some key features that Microsoft and this ecosystem provides when constructing the system.

2.4.7 Cross-Platform Support

One of the significant advantages of ASP.NET Core is its cross-platform support. You can run ASP.NET Core applications on Windows, Linux, and macOS. This flexibility allows you to choose the operating system that best fits your deployment environment, whether it is a local server, a cloud platform, or a hybrid setup.

2.4.8 Robust Security Features

Security is a critical aspect of any web application, especially those interacting with databases. ASP.NET Core provides built-in support for industry-standard authentication protocols and security features. It includes mechanisms to protect against common attacks such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). Additionally, it supports multi-factor authentication and external authentication providers like Google and Facebook.

2.4.9 Rich Ecosystem and Tooling

ASP.NET Core is part of the .NET ecosystem, which includes a vast array of libraries, tools, and frameworks. For database interactions, Entity Framework Core (EF Core) is a popular Object-Relational Mapper (ORM) that simplifies data access. It supports various databases, including SQL Server, SQLite, PostgreSQL, and MySQL.

2.4.10 Active Community and Open Source

ASP.NET Core is open-source and has an active community of developers. This means you can get quick answers to your questions on platforms like Stack Overflow and Microsoft Q&A. The project is hosted on GitHub, where you can report issues, contribute to the codebase, and access a multitude of community contributions.

2.4.11 MinIO

MinIO is an open-source, high-performance, distributed object storage system. It is designed to be fully compatible with the Amazon S3 API, which makes it an attractive option for developers and organizations looking to deploy scalable and cost-effective storage solutions. MinIO's simplicity, speed, and scalability make it suitable for a wide range of applications, from private cloud infrastructures to data lakes and large-scale data processing environments.

2.4.11.1 S3 Compatibility

One of the standout features of MinIO is its full compatibility with the Amazon S3 API. This means that any application designed to work with Amazon S3 can seamlessly interact with MinIO without any modifications. This compatibility extends to the majority of S3 features,

including bucket policies, multipart uploads, and presigned URLs, making MinIO a drop-in replacement for S3 in many scenarios.

2.4.11.2 High Performance

MinIO is optimized for high performance, with a particular focus on throughput and scalability. It can handle massive amounts of data with high efficiency, thanks to its use of advanced technologies such as erasure coding, bit-rot protection, and data compression. These features ensure that MinIO can deliver the performance required by demanding applications, such as big data analytics and machine learning workloads.

2.4.11.3 Scalability and Flexibility

MinIO is designed to scale horizontally, meaning that it can grow with your needs. You can start with a single server and expand to a large cluster of servers, all without sacrificing performance or reliability. This horizontal scalability is achieved through the use of a distributed architecture that allows MinIO to handle petabytes of data and billions of objects effortlessly.

2.4.11.4 Security

Security is a critical concern for any storage solution, and MinIO addresses this through a variety of mechanisms. It supports server-side encryption, ensuring that data is encrypted at rest. Additionally, MinIO offers support for Transport Layer Security (TLS) to encrypt data in transit. Access control mechanisms, including Identity and Access Management (IAM) policies, provide fine-grained control over who can access data and what actions they can perform.

2.4.11.5 Ease of Use

MinIO is designed to be easy to install and manage. It has a minimalistic design that reduces complexity and makes it straightforward to deploy and configure. The web-based management console provides a user-friendly interface for managing storage resources, while the command-line interface (CLI) offers powerful scripting capabilities for automation.

2.4.11.6 Interoperability

Many legacy systems and workflows still rely on SFTP for file transfer. By supporting SFTP, MinIO allows these systems to integrate seamlessly with modern object storage, providing a bridge between traditional file transfer methods and advanced cloud-native storage solutions.

Chapter 3

System Analysis and Design

3.1 Exploratory Data Analysis

This section describes the processes and steps that our team took to study the data.

3.1.1 Describe provided data

The provided data resides inside two parent directories:

For the `convert` directory, it seems like the data there has been converted from the other `UF` files. Compared with the original, these files do not store as many necessary fields, with many meteorologist features being left out or not included. For instance, fields like **Reflectivity**, **Mean doppler velocity** and **Doppler spectrum widtah** only appear in the `UF` files, but not in the converted ones.

```
{ 'time': { 'units': 'seconds since 2019-05-13T01:20:07Z', } ... 'fields': {
    'total_power': { 'units': 'dBZ', 'standard_name':
    'equivalent_reflectivity_factor', 'long_name': 'Total power', 'coordinates':
    'elevation azimuth range', 'data': masked_array( data=[ [37.0, 27.5, 37.5,
    ..., --, --, --], [50.0, 41.0, 31.5, ..., --, --, --], [45.0, 41.0, 29.5,
    ..., 0.0, 4.0, 7.0], ..., [35.0, 32.0, 21.0, ..., 3.0, --, --], [33.5, 27.5,
    20.0, ..., --, --, --], [41.0, 31.5, 27.0, ..., --, --, --] ], mask=[
```

```
8     [False, False, False, ...,  True,  True,  True], [False, False,
    False, ...,
9     True,  True,  True], [False, False, False, ..., False, False, False
    ], ...,
10    [False, False, False, ..., False,  True,  True], [False, False,
    False, ...,
11    True,  True,  True], [False, False, False, ...,  True,  True,  True]
    ],
12    fill_value=1e+20, dtype=float32 ), '_FillValue': -9999.0 }, ... } }
```

Listing 3.1: A sample of metadata extracted from UF file

As a result, we decided to use the `UF` files for our study, while temporarily ignoring the `convert` directory.

3.1.2 Comparing PyART and LROSE

From the advice of our professor, at first, we decided to learn about LROSE and try to open the provided data. LROSE is a set of multiple tools and libraries for working with Radar data output. For macOS, support for Homebrew makes it easier to install these tools for local usage. However, for other platforms, downloading and compiling the source code is required, which increases the complexity and time needed to set up the environment.

The reference for LROSE is not very extensive, with many commands requiring special configs and parameters that have not been written down in the documentation. As a result, it took us a lot of time to figure out how to use the tools and libraries. In the end, though, our team was able to open the provided data and visualize it using the LROSE tools, whose name was `HawkEye`.

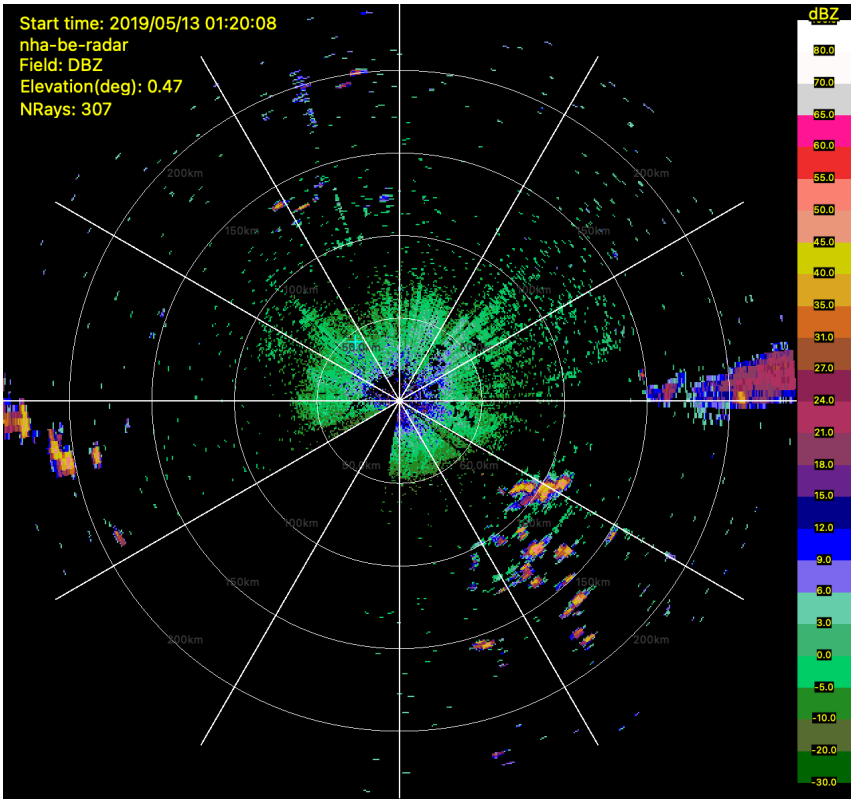


Figure 3.1: *Data visualization of Nha Be radar from HawkEye - Map of Vietnam*

During our investigation, we also encountered a different set of tools that can interact with the radar files, called PyART. When compared with LROSE-core, our team found that PyART contains many more advantages.

First, PyART is a Python library. This means that it is easier to install and use, especially for people who are already familiar with Python. Moreover, by supporting Python, our team can incorporate it into many popular ETL tools that use Python. Take Apache Airflow as an example. Instead of having to write a custom operator to call LROSE-core from the terminal, we can use a Task to perform the same action with PyART. Doing so would reduce much of the complexity of the workflow.

Secondly, with default configurations, PyART represents the data as an XArray [6] format, which is also a popular format for multidimensional data. As XArray is built on top of NumPy [5], this popular library can receive wider adoption when compared with many other formats. Integration with the existing library also means that we can use many other tools to work with the data. For instance, we can use Matplotlib to visualize the data; or use the library SciPy [19] to perform scientific calculations on the data. Because those tools require NumPy, we can use them directly with PyART without having to convert the data to another format.

Finally, PyART also has more comprehensive documentation when compared with LROSE. As newcomers to the subject of meteorology, we found that the documentation of PyART is much more accessible than LROSE. The documentation includes many examples and tutorials. Moreover, the documentation also contains many references to other resources, such as the book *Radar for Meteorological and Atmospheric Observations* [17]. Even though this is a subjective opinion, we believe that the documentation of PyART is much more helpful for further development.

With all the above ideas, our teams decided to use PyART for both the study and the development of the project.

3.1.3 Visualizing data using PyART

When installed, PyART also downloads another Python tool named Cartopy [12]. This library provides many useful features for our understanding of the data, such as the ability to plot the data on a map, or to convert the coordinates from one system to another.

Using these libraries, we can visualize our data directly from a Jupyter Notebook, similar to any other Python data visualization task. The area that the radar collects is a square, with the radar at the center. All of its length extends to about 300 kilometers in each direction.

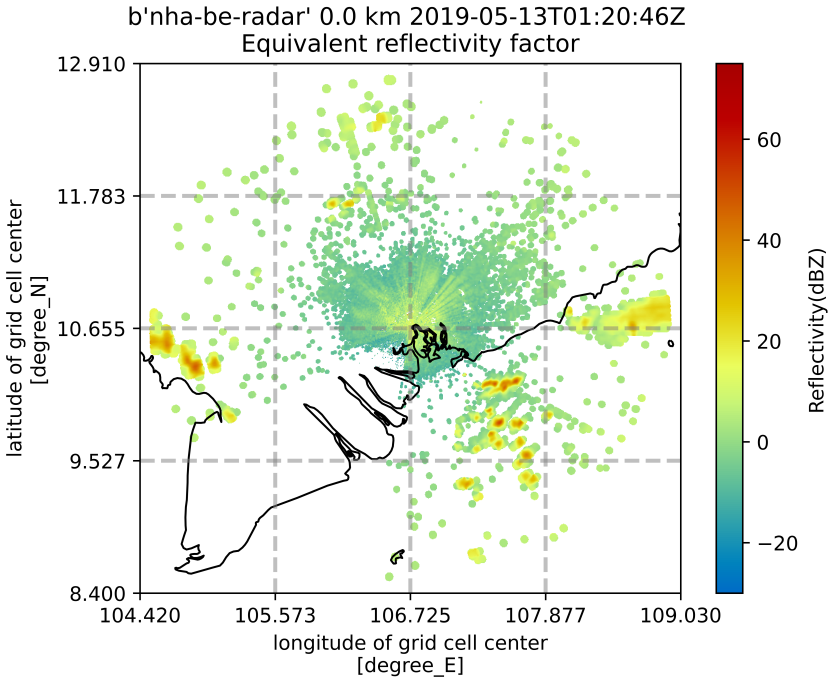


Figure 3.2: *Data visualization of Nha Be radar from PyART - Map of Vietnam - similar zoom to Figure 3.1*

3.2 Requirements

3.2.1 Functional Requirement

Data ingestion is the initial step where data is gathered from various sources, including **APIs** and **government data repositories**. This process is critical for accumulating the diverse datasets required for thorough analysis.

The next phase involves **data storage** and **management**. This may be carried out using databases or **data lakes**, providing robust solutions for handling extensive volumes of data effectively.

Data analysis and **visualization** are tailored specifically to support the needs of **meteorologists** and **researchers**. These tools are crucial for the detailed examination and interpretation of meteorological data.

Integration with third-party platforms and **APIs** enhances the system's functionality, enabling seamless interactions with other technological tools.

The system incorporates **user management** and **access control mechanisms** to ensure data security and accessibility only to authorized personnel, maintaining data integrity and confidentiality.

Lastly, the system includes **reporting** and **alerting functionalities** that are vital for communicating essential information to users, stakeholders, and decision-makers in a timely and efficient manner.

3.2.2 Non-Functional Requirements

Reliability is a critical component, requiring the platform to ensure high availability and stable operation consistently. It is also crucial to maintain **data integrity**, **security**, and **privacy** as top priorities to protect user information and system data.

Scalability needs to be addressed to handle increasing volumes of data. The system should have the ability to scale efficiently, leveraging **cloud-native technologies** and **containerization** such as Docker and Kubernetes, to facilitate easy scaling of infrastructure and applications.

In terms of **performance**, the platform must offer rapid data processing and analysis capabilities to meet the demands of real-time decision making and large-scale data handling.

Usability is essential, with a focus on providing an intuitive and **user-friendly interface** that is tailored for different user groups to ensure ease of use and accessibility.

Security measures must include robust data protection and access control mechanisms to safeguard against unauthorized access and potential data breaches.

Compliance with applicable regulatory standards, data privacy laws, and compliance requirements is mandatory to ensure that the platform operates within legal constraints.

Lastly, expanding on **scaling technology and infrastructure**, the platform should adopt **cloud-native architectures** and containerization technologies to enable easy scaling. Utilizing auto-scaling capabilities and elastic resources provided by cloud platforms allows for dynamic adjustment of capacity based on demand. Additionally, implementing a **microservices architecture** and decoupled components will enable independent scaling of individual services or modules, enhancing overall system resilience and flexibility.

3.2.3 Data Requirements

The initial step in handling radar data involves a comprehensive **cleaning process**. This is crucial to remove any **noise** and extraneous data, ensuring the accuracy and reliability of the data. Cleaning radar files is essential as it preps them for more complex processing and analysis, which are critical for accurate meteorological assessments.

Once the radar data has been cleaned, it undergoes a sophisticated **processing routine**. Specialized algorithms analyze the data to extract meaningful patterns and information, structuring it into a format that can be easily utilized in further analyses. This step is vital for transforming raw data into actionable insights that can be relied upon for decision-making.

After processing, the data is stored as **blobs** in an object storage system, typically **Minio**. This method of storage is selected for its scalability and ease of access, which are necessary attributes for managing the voluminous data generated by radar systems. Blob storage also facilitates the efficient retrieval and management of data, supporting dynamic access patterns required by meteorological applications.

To further enhance data handling efficiency, a **caching mechanism** is implemented. This system stores recently accessed data temporarily, significantly reducing retrieval times and ensuring that frequently needed information is readily available. This feature is particularly important during critical situations, such as severe weather events, where timely data access is

crucial.

The processed and stored radar data is extensively used across various applications. Meteorologists rely on this data for accurate **weather forecasting** and **climate modeling**. It is also crucial for **emergency response teams** who depend on real-time data for effective decision-making during weather-related emergencies. Furthermore, the data supports a broad range of scientific research, aiding in the study of climate patterns and atmospheric conditions.

3.3 Data management

3.3.1 Data modeling

3.3.2 Blob storage

As an intermediary entity situated between the data producers, exemplified by the radar center, and the data consumers, represented by the Data Model and Machine Learning team, our paramount responsibility lies in ensuring that the data management infrastructure is meticulously designed to seamlessly accommodate the diverse and often contrasting requirements set forth by both parties.

The dataset, procured from the esteemed Nha Be radar center, adheres to the rigorous SIGMET format, as delineated in 2.3.1. This format encompasses a plethora of scalar values pertaining to metadata facets such as the radar's nomenclature, the modality of scanning employed, and the temporal stamp denoting the execution of the scan. Concurrently, this data corpus embodies a rich array of multidimensional metrics encapsulating various meteorological phenomena, including but not limited to reflectivity and energy profiles.

Collaborative engagements with the modeling cadre, predominantly comprising members from the distinguished teams led by Tu and Vinh, have endowed us with a nuanced understanding of their exigencies and prerequisites vis-à-vis data structuring paradigms. From the purview of these erudite teams, data representation assumes two principal manifestations. Firstly, there is a pronounced inclination towards encapsulating data in the form of images tailored to specific geographical demarcations within the Vietnamese domain. A quintessential exemplar of this utilization paradigm is elucidated in Figure 3.2, which encapsulates the data instrumental in honing the multimodal solutions. Alternatively, there exists a proclivity towards direct data consump-

tion in the form of NumPy Tensors, leveraging the expansive spectrum of NumPy-compatible APIs inclusive of those furnished by SciPy and XArray. Although ostensibly more efficient, this mode of data consumption has not garnered widespread adoption within the precincts of our collaborative endeavor.

Armed with an exhaustive comprehension of the requisites delineated by both stakeholders and guided by heuristic principles germane to the architecture of rudimentary data warehousing systems, we proffer a set of salient properties that our envisaged system ought to embody:

Primarily, the envisaged data repository must find its abode in an Object Storage medium, characterized by maximal interoperability with extant systems. This necessitates the exclusion of proprietary solutions such as Google Cloud Storage or Azure Storage Blob in favor of alternatives such as Hadoop Distributed File System (HDFS) [15] or any Object Storage infrastructure compliant with AWS S3 protocols. While HDFS presents itself as a viable candidate, the logistical intricacies entailed in deploying and administering a comprehensive HDFS cluster militate against prioritizing its adoption. Fortuitously, our discerning team has identified a panacea in the form of MinIO, an open-source Object Storage solution that not only aligns with the requisites of our project but also boasts compatibility with prevailing AWS S3 APIs.

The conundrum surrounding the optimal data format to be stored within the Object Storage reservoir has been a subject of fervent debate and contention amongst our learned colleagues. While proponents of denormalization advocate for its adoption to streamline data retrieval processes, empirical evidence underscores the efficacy of the RAW data format in compressing and storing multidimensional datasets, as attested by the diminutive file sizes characteristic of individual scan executions. Indeed, attempts to transmute this format into alternative structures, such as JSON, have yielded negligible dividends owing to the intrinsic complexity of the dataset. Moreover, with the ascendancy of multimodal Machine Learning paradigms, necessitating the simultaneous retrieval of diverse data fields to facilitate batch processing, the retention of all pertinent fields within a singular RAW file emerges as a prescient strategy poised to accommodate the evolutionary trajectory of our modeling endeavors.

In delineating a nomenclatural schema for the files ensconced within the Object Storage repository, precedence is accorded to a rudimentary ordering scheme premised on tenant identification and temporal demarcation. For the myriad utilization scenarios contemplated heretofore, the concatenation of the tenant identifier with a timestamp proves to be a judicious nomen-

clatural convention. Furthermore, the hierarchical organization of files within the Object Storage framework ought to be predicated on a delineation between tenant-specific directories and temporal partitions.

Regarding the temporal component of the nomenclatural schema, adherence to the ISO-8601 standard, a globally ratified temporal convention, is deemed imperative. The employment of the ISO-8601 timestamp format, typified by the concatenation of year, month, day, hour, minute, and second components devoid of any superfluous delimiters, serves to obviate compatibility issues with extant filesystems. Notably, the conspicuous absence of special characters such as hyphens and colons serves to circumvent prohibitions imposed by certain filesystems, such as Windows, on the usage of such characters within filenames.

3.3.3 Caching

3.4 System Architecture

In today's rapidly evolving technological landscape, building robust, scalable, and maintainable systems is paramount. A well-designed system architecture not only addresses these needs but also provides a foundation for future growth and adaptability. This section delves into the architecture that underpins our system, exploring the principles of microservices, the application of Clean Architecture, and the integration of various infrastructure components to create a cohesive and efficient solution.

3.4.1 Microservices

In essence, a microservices architecture decomposes a large software application into a collection of smaller, self-contained services. Each service plays a specific role and owns a well-defined business capability. These services are loosely coupled, meaning they interact through well-designed APIs and operate independently.

Microservices architectures offer a compelling approach to building software. By decomposing large applications into smaller, independent services, they unlock agility, maintainability, and fault isolation. Each service owns a specific business function and interacts with others through well-defined APIs. This allows for faster development cycles, easier maintenance of individual services, and the freedom to choose the best technology for each job.

However, this power comes with a price. Managing a distributed system with numerous services inherently increases complexity compared to a monolithic application. Communication between services adds overhead to the system's performance. Perhaps the most significant challenge lies in ensuring data consistency across these independent services, requiring meticulous design and implementation. Carefully considering these trade-offs is crucial before embarking on a microservices journey.

While microservices boast impressive agility and maintainability, adopting this architecture isn't without its complexities. Managing a distributed system with numerous services inherently increases complexity compared to a monolithic application. Furthermore, communication between these services adds overhead to the system's performance. Perhaps the most significant challenge lies in ensuring data consistency across these independent services, requiring meticulous design and implementation.

3.4.2 Clean Architecture

Microservices and Clean Architecture can complement each other to create a modular, maintainable, and scalable system. Microservices architecture promotes the development of small, independent services that can be deployed, scaled, and updated independently. Each microservice encapsulates a specific business capability or domain, reducing the overall complexity of the system.

Clean Architecture, on the other hand, advocates for a layered approach to software design, promoting separation of concerns and decoupling of components. By adhering to the principles of Clean Architecture within each microservice, developers can achieve a high degree of modularity, testability, and maintainability.

Clean Architecture separates the system into distinct layers, each with specific responsibilities. The Shared Kernel acts as the core, housing reusable components that benefit all layers. The Domain layer sits at the heart of the application, defining core entities and their behaviors. A common base class promotes code maintainability within this layer. The Application layer orchestrates request routing and establishes project-wide contracts for implementation in other layers. Clean Architecture emphasizes the testability of each layer through unit tests. Frameworks like xUnit simplify unit test creation.

The Infrastructure layer provides supporting services for the application. Cross-cutting con-

cerns like logging reside here. User registration, authentication, and authorization functionalities are handled by the Identity layer. Persistence takes care of data access using patterns like repositories and Unit of Work. The Web Framework layer manages configurations for the web application, while the Web API layer delivers functionality to the user. Finally, plugins bridge the gap between monolithic and microservices architectures by promoting modularity.

3.4.3 Recommended System Architecture

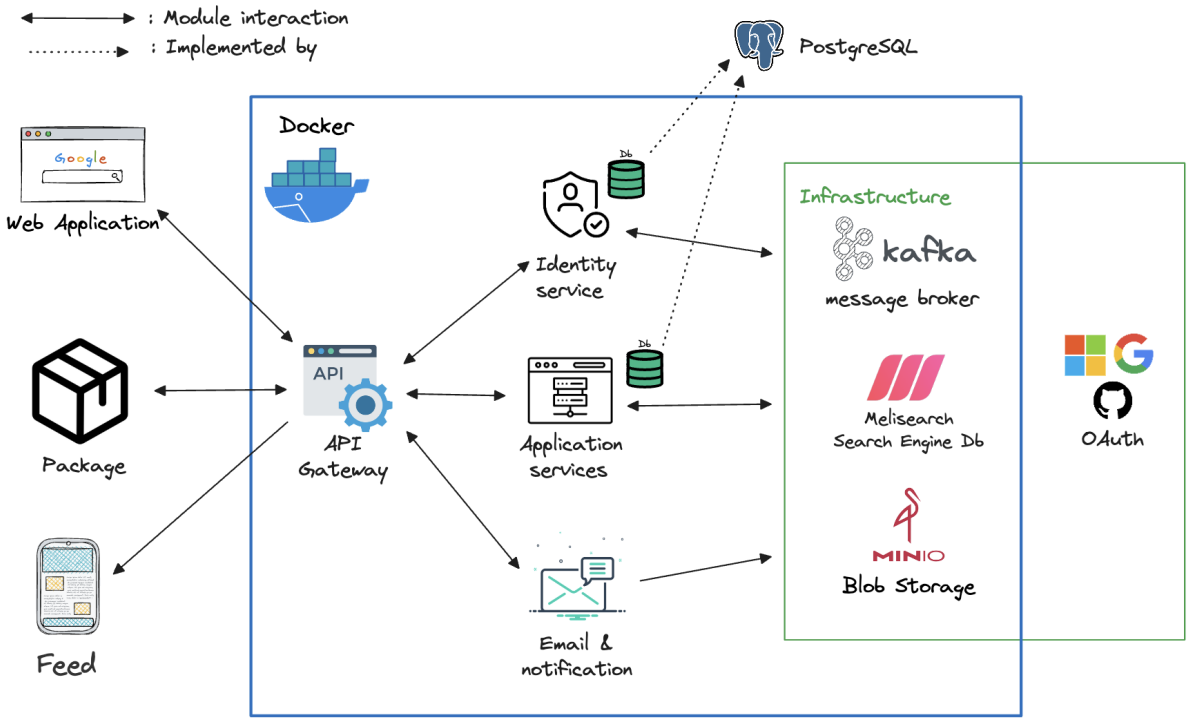


Figure 3.3: *System Architecture*

Based on the requirements set by stakeholders and after studying the existing system, the team proposes the design and implementation of a Weather Data Platform. Figure 3.3 illustrates the implementation of the system.

Central to this architecture is the use of Docker, which encapsulates the application's microservices in containers. This encapsulation ensures that each service can be deployed and scaled independently across different environments without compatibility issues. Docker not only facilitates easy deployment but also enhances the manageability and scalability of the ap-

plication services, making the infrastructure robust and flexible.

Within this architecture, a critical component is the Identity Service. This service is responsible for managing user authentication and authorization processes. It likely operates with a dedicated database that stores user credentials and session information, ensuring secure and efficient user access control. By centralizing identity management, the system can enhance security and simplify the integration of different services that require user identification and access control.

The Application Services form the backbone of the system's business logic. These services handle the core functionalities of the application, interacting with the Identity Service for authentication purposes and accessing dedicated databases to retrieve or store data. This separation of concerns allows for better maintenance and scalability of the application logic, enabling each service to be optimized and scaled according to specific demands without affecting the overall system performance.

An Email & Notification Service is integral to the architecture, managing communications with users. This service automates the sending of emails and other notifications, which are crucial for user engagement and timely communication. By having a dedicated service for this function, the system ensures that notifications are handled efficiently, maintaining high performance even under heavy loads of communication tasks.

The architecture is further supported by robust infrastructure components. PostgreSQL serves as the primary database management system, offering reliable and efficient management of structured data with its powerful SQL capabilities. Kafka, used as a message broker, ensures seamless and reliable data flow between different parts of the application, which is essential for maintaining data consistency and decoupling services. Meilisearch enhances the application's functionality by providing a fast and responsive search engine database, which significantly improves the user experience through quick search results. MinIO offers a high-performance solution for blob storage, managing unstructured data such as media files, backups, and logs, which supports the application's scalability and data management needs. Lastly, OAuth is employed to facilitate secure delegated access, allowing the application to authenticate users by integrating with external OAuth providers like Google, thus broadening the scope of user accessibility and security.

Each component of this architecture plays a vital role in ensuring the application is scalable,

modular, and resilient, employing a combination of state-of-the-art tools and technologies to meet a broad range of operational requirements efficiently.

Chapter 4

Implementation

4.1 Data Processing

The implementation phase of the team will be an extension to the current system, which is outlined in Figure 3.3.

In step 3, the team will set up a simple SFTP server. SFTP is a straightforward and widely used protocol, supported by numerous libraries and tools for communication based on this protocol. Additionally, compared to FTP, the mentioned protocol ensures security during data transfer. Depending on the permissions, the team may assist the observation station in constructing scripts to automatically forward processed files or allow manual file submission.

Once files are uploaded to the SFTP server, the team utilizes Airflow to orchestrate all existing ETL workflows in the overall system. Currently, the team pauses with a single DAG to process data from the Nha Be observation station. Airflow monitors newly added files on our SFTP server and initiates the ETL process. The choice of Apache Spark is based on the volume and complexity of the data. If the data size per new SIGMET file is manageable with Python alone, without the need for Spark, it will not be employed in this step.

Meteorological data, upon reaching the team's infrastructure, will be bifurcated into two main streams: Metadata, such as creation date, size, timestamp, etc., will be stored in a traditional Relational Database Management System (RDBMS). Specifically, PostgreSQL is chosen due to its popularity and the team's familiarity. Storing metadata here facilitates rapid query responses without direct access to the raw data. Common queries may include:

- What timestamps are being recorded? (e.g., from 21/11/2023 to 17/12/2023)

- At timestamp x, what are the geographical coordinates of the radar?
- What fields of data are currently stored?

Additionally, the database acts as an index, quickly identifying the storage location of raw data.

For specific hydrometeorological data, the team finds it inefficient to store them directly in conventional DBMS. Simultaneously, storing data in files still maintains a reasonable overall size. Therefore, the team decides to separate the raw data and store it directly in files. This approach, combined with the previously mentioned indexes, accelerates the retrieval process.

To facilitate data queries for models, machine learning, AI, etc., the team will develop a simple backend server using Python's FastAPI at step 5. In step 6, the backend receives query data in REST API format, queries the metadata DB and data files, and returns the achieved results. During different training instances, Machine Learning entities can connect to this server to retrieve data.

It's worth mentioning that the entire system will be developed and operated in a containerized manner and will be deployed on the Kubernetes platform. This reflects the system's ability to maintain high availability and ease of solution maintenance. In this illustration, the team will deploy it on a cluster of Raspberry Pi-embedded computers.

Lastly, in step 7, the team proposes an additional consideration. If suitable, the team may build a DataLoader to swiftly serve other model-making groups. Considering the popularity of Pytorch in AI, the team will initially approach this platform.

4.2 System

4.2.1 Programming languages and Libraries

C# was selected for this project because of its widespread popularity in enterprise applications and its capability to operate across various operating systems following the release of .NET Core. This feature significantly enhances deployment flexibility, allowing the application to be utilized in diverse environments.

Additionally, we have incorporated ASP.NET Core specifically version 8 into our technology stack. By leveraging ASP.NET Core, we benefit from a robust, well-supported framework

that facilitates the development of scalable and secure web applications. It seamlessly integrates with C#, enabling us to utilize a consistent programming environment while also exploiting features such as dependency injection, a vast ecosystem of middleware, and a strong configuration system that is suited to modern web applications.

ASP.NET Core 8 brings forward improvements in areas such as minimized startup times, reduced memory footprint, and enhanced security features, making it an ideal choice for developing scalable and secure web applications. The choice also underscores our commitment to developing applications that are both efficient and future-proof, ensuring that they perform optimally on both Windows and non-Windows platforms. This alignment with .NET Core's cross-platform capabilities ensures that our project remains versatile and adaptable to the evolving technological landscape.

Optionally, we have integrated the Nuxt framework into our technology stack. Nuxt is a progressive Vue framework that is used for building more robust and versatile web applications. It simplifies the development process by handling various aspects of the web infrastructure, such as server-side rendering, static site generation, and automatic code splitting. This inclusion enriches our application's interactivity and user experience, providing a seamless and dynamic interface for users.

4.2.2 Command Query Responsibility Segregation

The Command Query Responsibility Segregation (CQRS) is an architectural pattern that distinctively separates the tasks of reading data (queries) and writing data (commands) within a software application. This separation splits responsibilities into two main components:

- **Command Side**: This component manages operations that modify the system's state. It handles incoming commands from clients or external systems, conducts validations, and updates the data store accordingly. This side is essential for maintaining the integrity and accuracy of data modifications within the application.
- **Query Side**: Dedicated to data retrieval, this component processes all read requests. It fetches data from the appropriate sources, ensuring that the information provided is accurate and reflects the current state of the data store.

Key advantages of employing the CQRS pattern are:

- **Scalability**: CQRS allows for the independent scaling of the read and write components based on their respective workloads, which can significantly enhance system performance.
- **Flexibility**: With the separation of concerns, different storage and optimization strategies can be applied to the reading and writing processes. This flexibility enables the use of the most appropriate tools for each function, optimizing efficiency.
- **Event-Driven Architecture Compatibility**: The use of CQRS often complements event-driven architectures, where changes in the system's state are captured and managed as events. This compatibility ensures that the architecture is dynamic and responsive to changes in business requirements.

4.2.3 Project Structure

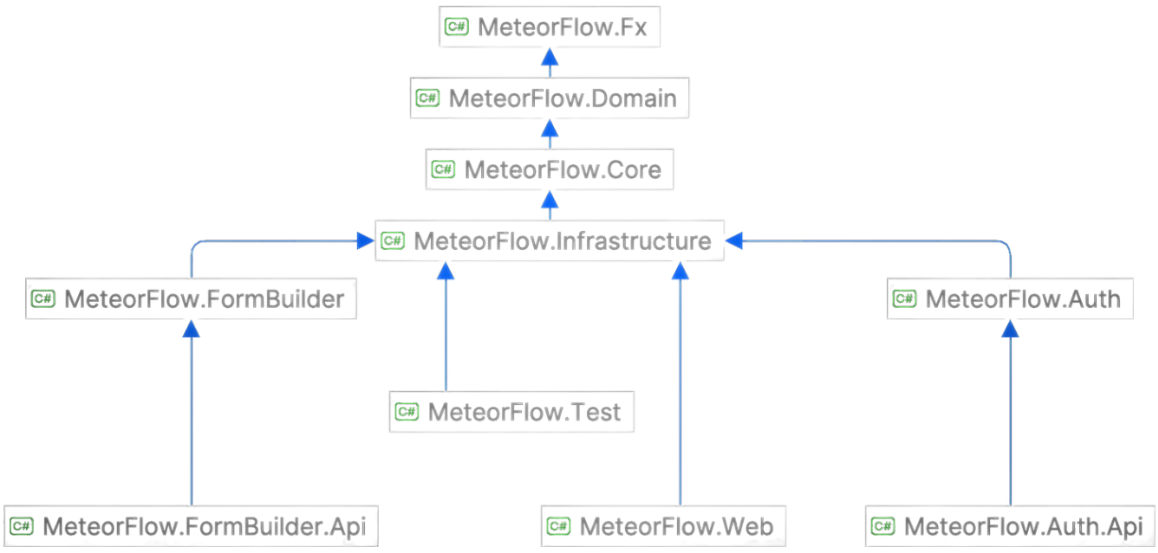


Figure 4.1: *Project Structure*

MeteorFlow is depicted as a modular framework comprising several interdependent components. Each library or module serves a distinct function, collectively supporting a robust and scalable application infrastructure. This section will delineate the roles and relationships of these components.

MeteorFlow.Fx enriches the application by introducing additional functionalities or user interface enhancements. These features, while not central to the primary business logic, signif-

icantly augment the application's overall capabilities, providing enriched user experiences and functional extensions.

MeteorFlow.Domain is tailored to articulate the business domain, encapsulating entities and rules essential to the application's domain logic. Its reliance on the Core module underscores the latter's foundational role within the architecture, affirming its influence over the domain-specific functionalities.

At the heart of the architecture, MeteorFlow.Core embodies the fundamental business logic and operations critical to the application. Its pivotal role is underscored by its influence on all peripheral modules, which depend on the Core for their foundational functionalities.

MeteorFlow.Infrastructure functions as the backbone for data access and manages cross-cutting concerns such as logging and caching. This module is crucial for the operational management of the application, interfacing seamlessly with the Core module to apply essential functionalities across various infrastructural tasks.

Built upon the MeteorFlow.Infrastructure, the Auth module specializes in security and authentication processes, managing user verification and credential handling. The module provides APIs to enable external access to their functionalities and seamlessly integrate with other systems and applications. Other modules currently or in the near future follow the same structure above to provide additional functionalities.

MeteorFlow.Web, primarily tasked with managing the application's API gateway, interacts with all other modules within MeteorFlow to ensure robust web operations and effective resource management. Its role is crucial in maintaining the integrity and performance of web-based services, underpinning the system's interaction with users and external systems.

4.3 API Specification

4.3.1 Core API

GET /api/core/definition

Tags	Definition
Description	Retrieves a list of all core definitions.
Responses	200: An array of core definitions returned successfully.

	401: Unauthorized access.

POST /api/core/definition

Tags	Definition
Description	Creates a new core definition.
Request Body	AppDefinitions schema.
Responses	201: Core definition created successfully. 400: Bad request if the data is invalid. 401: Unauthorized access.

GET /api/core/definition/{id}

Tags	Definition
Description	Retrieves a specific core definition by ID.
Parameters	id: UUID of the core definition.
Responses	200: Core definition returned successfully. 404: Core definition not found. 401: Unauthorized access.

DELETE /api/core/definition/{id}

Tags	Definition
Description	Deletes a specific core definition by ID.
Parameters	id: UUID of the core definition.
Responses	200: Core definition deleted successfully. 404: Core definition not found. 401: Unauthorized access.

POST /api/core/instance

Tags	Instance

Description	Creates a new instance.
Request Body	AppInstances schema.
Responses	201: Instance created successfully. 400: Bad request if the data is invalid. 401: Unauthorized access.

GET /api/core/instance/{id}

Tags	Instance
Description	Retrieves a specific instance by ID.
Parameters	id: UUID of the instance.
Responses	200: Instance returned successfully. 404: Instance not found. 401: Unauthorized access.

DELETE /api/core/instance/{id}

Tags	Instance
Description	Deletes a specific instance by ID.
Parameters	id: UUID of the instance.
Responses	200: Instance deleted successfully. 404: Instance not found. 401: Unauthorized access.

GET /api/core/setting

Tags	Setting
Description	Retrieves a list of all settings.
Responses	200: An array of settings returned successfully. 401: Unauthorized access.

POST /api/core/setting

Tags	Setting
Description	Creates a new setting.
Request Body	AppSettings schema.
Responses	201: Setting created successfully. 400: Bad request if the data is invalid. 401: Unauthorized access.

GET /api/core/setting/{id}

Tags	Setting
Description	Retrieves a specific setting by ID.
Parameters	id: UUID of the setting.
Responses	200: Setting returned successfully. 404: Setting not found. 401: Unauthorized access.

DELETE /api/core/setting/{id}

Tags	Setting
Description	Deletes a specific setting by ID.
Parameters	id: UUID of the setting.
Responses	200: Setting deleted successfully. 404: Setting not found. 401: Unauthorized access.

GET /configuration

Tags	FileConfiguration
Description	Retrieves the current configuration.

Responses	200: Configuration returned successfully. 401: Unauthorized access.

POST /configuration

Tags	FileConfiguration
Description	Updates the current configuration.
Request Body	FileConfiguration schema.
Responses	200: Configuration updated successfully. 400: Bad request if the data is invalid. 401: Unauthorized access.

4.3.2 Indentity API

GET /api/auth

Tags	Auth
Description	Retrieves authentication status based on the returnUrl parameter.
Parameters	returnUrl (string): The URL to return to after authentication.
Responses	200: Authentication status returned successfully.

POST /api/auth/login

Tags	Auth
Description	Logs in a user with provided credentials.
Request Body	LoginInfo schema: Includes username and password.
Responses	200: Login successful.

POST /api/users/{id}/passwordresetemail

Tags	Users
Description	Sends a password reset email to the user specified by ID.
Parameters	id (string, uuid): Unique identifier of the user.
Responses	200: Password reset email sent successfully.

PUT /api/users/{id}/password

Tags	Users
Description	Updates the password for the user specified by ID.
Parameters	id (string, uuid): Unique identifier of the user.
Request Body	PasswordSetter schema: Includes new password and confirmation.
Responses	200: Password updated successfully. 404: User not found.

POST /api/users/{id}/emailaddressconfirmation

Tags	Users
Description	Sends an email confirmation to the user specified by ID.
Parameters	id (string, uuid): Unique identifier of the user.
Responses	200: Email confirmation sent successfully.

4.4 Datastore

4.4.1 The purpose of storing data directly as raw file

In the comprehensive compilation acquired during the recent field excursion, an array of invaluable data was meticulously gathered, meticulously chronicled, and thoughtfully analyzed. Specifically, our focus gravitated towards the operational activities conducted within the radar center situated in the Nha Be district. Operating at the pinnacle of technological advancement, the radar center diligently executes periodic scans of the atmospheric conditions, facilitating

the meticulous documentation and exportation of a SIGMET file, which serves as a repository encapsulating a myriad of meteorological phenomena and atmospheric dynamics.

The SIGMET file, serving as the quintessential embodiment of scientific rigor and methodological precision, meticulously records a plethora of critical parameters and meteorological variables. Among the salient features meticulously delineated within this archival masterpiece are the discerning metrics of reflectivity and wind radial velocity. Reflectivity, a fundamental metric in radar meteorology, is an indispensable parameter delineating the intensity of electromagnetic waves returned to the radar antenna from various atmospheric particles, thereby furnishing crucial insights into the spatial distribution and intensity of precipitation within the monitored region. Furthermore, the wind radial velocity, an elemental component in meteorological analysis, delineates the rate and direction of atmospheric motion along the radial axis relative to the radar antenna. These metrics serve as a vital tool in elucidating the intricate dynamics of atmospheric circulation, enabling meteorologists to decipher prevailing wind patterns, identify regions of convective activity, and forecast the trajectory of severe weather phenomena with enhanced accuracy and precision.

Yet, amidst the effervescent tapestry of meteorological data acquisition and analysis, a conundrum of paramount importance arises: the optimization of data storage and retrieval mechanisms. It is within this crucible of inquiry that the proposition to store meteorological results directly within the file repository assumes a mantle of significance, heralding a paradigm shift in data management practices.

Indeed, the inherent multidimensionality of meteorological data, encapsulated within its tripartite tensor structure, underscores the imperative for a streamlined approach to data storage and retrieval. By eschewing aggregation and denormalization techniques, we advocate for a direct integration of results within the archival framework, thereby fortifying the foundations of data integrity and computational efficiency.

Moreover, the proposition to leverage the existing ETL (Extract, Transform, Load) system developed by Vaisala, as utilized by our esteemed National Center for Hydrometeorological Forecasting, emerges as a beacon of pragmatism and resourcefulness. In a landscape fraught with technological complexities and budgetary constraints, the utilization of pre-existing infrastructure represents a judicious allocation of resources, affording seamless integration and interoperability across disparate data management platforms.

In summation, the decision to store meteorological data directly within the file repository, without resorting to further aggregation techniques, emerges as a testament to both pragmatism and foresight. By embracing this approach, we not only enhance the accessibility and usability of meteorological datasets but also lay the groundwork for a new era of scientific inquiry and meteorological prognostication, fortified by the pillars of technological innovation and methodological rigor.

4.4.2 Extracting data from each of the radar center

We have authored a script, albeit not yet integrated into the operational system, designed to facilitate the redirection of the SIGMET (Significant Meteorological Information) file from the data center to our preconfigured Storage Server.

Presently, the script is implemented in Python to ensure platform-agnosticism. Accompanied by a concise setup script, it will be poised for execution on the radar center's machinery.

MinIO has been employed as an unstructured data repository owing to its compatibility with the S3 (Simple Storage Service) protocol. As the system matures in the distant future, transitioning to AWS S3 should be straightforward. The underlying logic and API for interfacing with the storage infrastructure remain largely invariant during such a migration.

```
file_path = pathlib.Path(args.filename)

s3_client = boto3.client( "s3", endpoint_url=os.getenv("S3_HOSTNAME"),
    aws_access_key_id=os.getenv("S3_ACCESS_KEY_ID"),
    aws_secret_access_key=os.getenv("S3_SECRET_ACCESS_KEY"), ) _ =
    s3_client.upload_file(file_path, os.getenv("S3_BUCKET"),
    generate_new_name(file_path.name))
```

Listing 4.1: Part of the script for uploading data to Storage

underfull hboxes

Currently, we follow a designated nomenclature for naming data files of the RADAR objects: `<year><month><day>T<hour><minute><second>`. This system enhances the speed of data retrieval for specific times of the day. Utilizing S3 Prefix filtering, we can efficiently select multiple data points within a time frame that may vary from a second to an entire year.

For instance, to query data for a specific day (e.g., 2024-03-15), utilizing the UNIX path wildcard, we can express our query as:

```
ls 20240315T*
```

Analogously, the same query can be executed using the `boto3` library in Python:

```
def query_with_wildcard(bucket_name): s3_client = boto3.client('s3')

    response = s3_client.list_objects_v2( Bucket='nha-be-radar',
        Prefix='20240315T' )

    return [obj['Key'] for obj in response['Contents']]
```

Listing 4.2: Querying data in 2024-03-15

Moreover, although our naming convention deviates somewhat from the ISO-8601 standard for timestamp representation, the inclusion of the letter **T** aids in swiftly identifying datetime components within the naming structure.

One limitation of this naming convention is its inability to support wildcard querying for a central time component. Consider the scenario necessitating data retrieval for a specific hour daily. Utilizing the UNIX wildcard, this can be depicted as:

```
ls 202403*T09*
```

The utilization of an internal wildcard here implies the absence of an efficient mechanism for querying such objects presently. The current recourse involves maximal prefixing (e.g., `Prefix=202403`), followed by subsequent filtering in Python.

4.4.3 Data Explorer

Besides S3-compatible API for accessing the storage, our solution also provides some alternative ways for preview the data storage. This can be used for manual intervention (in case of failure), or simply for the radar center's operators to preview.

First, there is an online explorer that can be access from the web browser. As you can see from Figure 4.2, the web UI is very user-friendly and easy-to-use. With the use of IAM (shorts for Identity and Access Management) and **Policies**, the admin user of our entire system can restrict what a tenant (or a radar center) can see and what should be hidden.

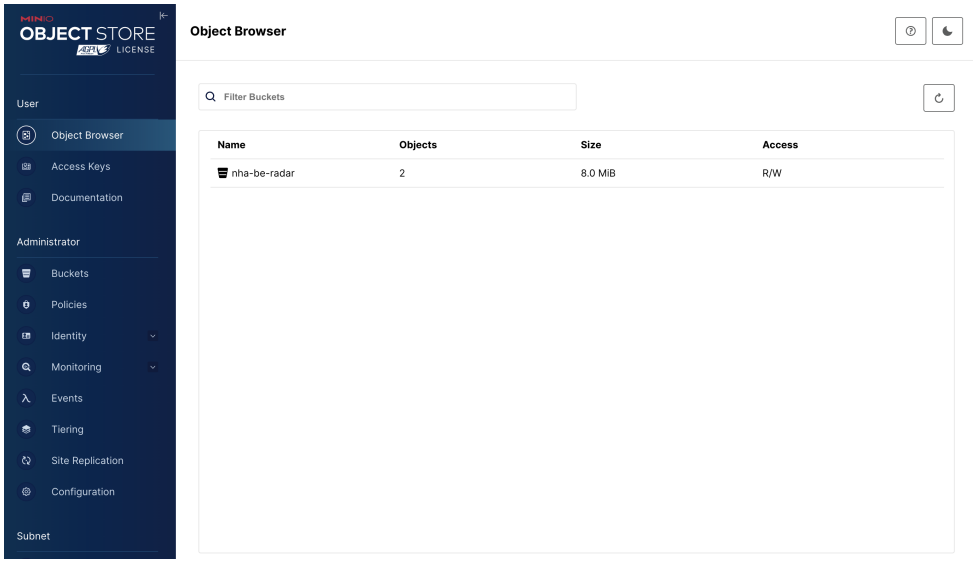


Figure 4.2: *MinIO web interface provides a clear and intuitive way to explore the storage*

Currently, as of the time of writing, our system has been self-hosted on a small Raspberry Pi server. The explorer has been set up with necessary load balancing, DNS and SSL/TLS registering correctly. Anyone that has been given proper permissions can visit `https://explorer.meteor-flow.com/` to see the server running. Figure 4.3 shows some of the files that our team has prepared beforehand.

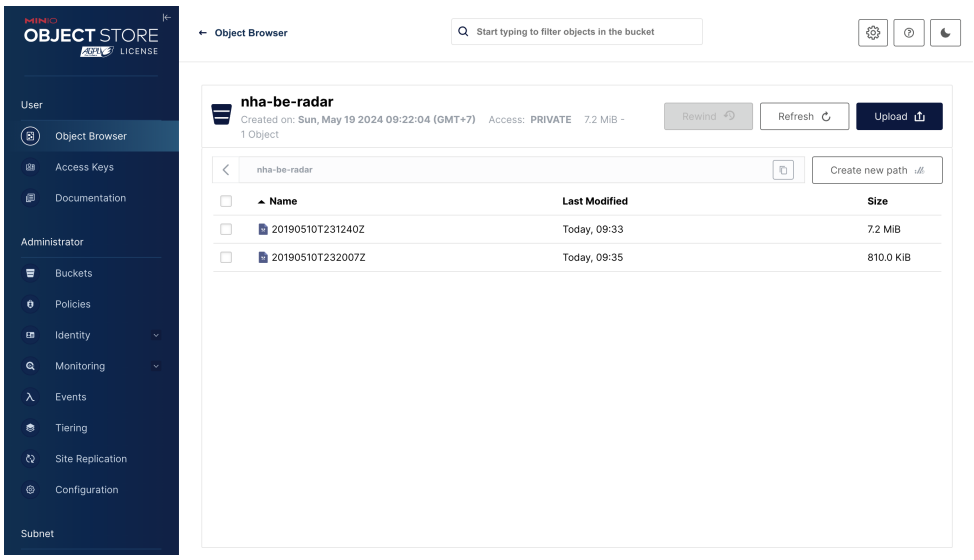


Figure 4.3: *Some radar files are currently being stored on the storage*

Not stopping at that, our system can also work with any other S3-compatible client. One notable example of this is Cyberduck. In the future, our team can research for more compatible protocols, such as FTP or SFTP.

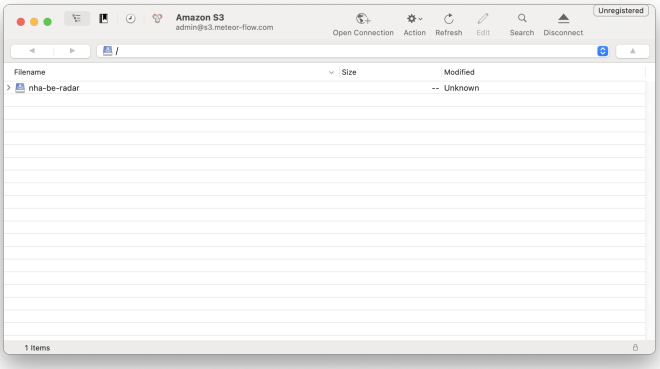


Figure 4.4: *Using Cyberduck to view data on the platform*

Chapter 5

Testing and Validation

5.1 Unit Test

5.2 Integration Test

During the Integration Test phases, our team makes sure that not only does our system run correctly at every part, but also that the system as a whole works as expected. To do that, we first start with a Test Plan.

5.2.1 MeteorFlow Test Plan

Project Name	MeteorFlow
Created Date	TBD
Release Date	TBD

Component	Start Time	Assignee	Notes
ETL files and REST API	TBD	Thuy Nguyen, Kiet Tran	Notes

5.2.2 MeteorFlow Test Report

Release Date: TBD

Chapter 6

Impacts

The "MeteorFlow" Weather Data Platform (WDP) represents an ambitious endeavor to create a centralized, comprehensive solution for managing and utilizing weather data. Designed to cater to meteorologists, scholars, researchers, developers, businesses, freelancers, and NGOs, this platform aims to transcend traditional weather data offerings and deliver a multifaceted, immersive weather data experience.

The strategic growth of the MeteorFlow involves several pivotal enhancements and extensions to its initial design:

Non-linear Data: We plan to extend the data integration capabilities of WDP to encompass a richer variety of non-linear, detailed, and multi-source data. This will enable users to gain deeper insights into environmental conditions and patterns, significantly enhancing the scope and utility of the weather data provided.

Interactive User Interface: Our goal is to transform the user experience from mere data retrieval to an interactive engagement with weather forecasts. The interface will be designed to stimulate user curiosity and facilitate active interaction with the data, making the exploration of weather conditions more engaging and informative.

Geospatial Information Connection: Integrating geographical information systems, WDP will provide a localized, contextual view of weather forecasts. This feature is crucial for users needing to understand the specific impacts of weather on their immediate environment, enhancing both personal and professional decision-making.

Performance Optimization: Ensuring that the platform operates quickly and smoothly under various conditions is a priority. This will enhance user experience and ensure that the platform remains reliable and efficient, even under high demand.

Data Security: We commit to enhancing the security measures within WDP to protect the integrity and confidentiality of the data. This is essential for maintaining user trust and ensuring that sensitive information remains secure against potential cyber threats.

Deployment and Maintenance: Following deployment, WDP will adhere to a structured update cycle to consistently deliver precise and dependable weather information. Regular updates and maintenance will help adapt to evolving user needs and incorporate the latest technological advancements.

Through these development directions, the MeteorFlow aims not only to serve as a repository of weather data but also as a dynamic, evolving platform that responds to and anticipates the needs of its diverse user base. This will significantly contribute to the field of meteorology and related disciplines by providing a reliable, innovative tool for weather data analysis and application.

Chapter 7

Conclusion

In this study, we have successfully built a Proof-of-Concept with high applicability, aiming to streamline the steps in the conventional workflow. This is a crucial step towards optimizing and enhancing operational efficiency in both research and practical contexts.

Our goal was to create a flexible system with high adaptability, helping to simplify complex steps in the workflow. In doing so, we not only contribute to increasing efficiency but also alleviate the workload pressure on personnel, creating favorable conditions for creativity and focus on core tasks.

We didn't just stop at developing the system but also proposed flexible deployment strategies, emphasizing easy integration into the current working environment of those involved in information gathering and weather forecasting tasks.

References

[1] Airflow. What is airflow?, 2020. URL https://airflow.apache.org/docs/apache-airflow/stable/index.html. Last accessed by 16/12/2023.

[2] casey. Using range height indicator scan of radar, 2017. URL https://earthscience.stackexchange.com/questions/7222/using-range-height-indicator-scan-of-radar. Last accessed by 17/12/2023.

[3] Ramez A. Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison Wesley, third edition, 1998.

[4] TRUNG TÂM DỰ BÁO KHÍ TƯỢNG THỦY VĂN QUỐC GIA, Jun 2203.

[5] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[6] S. Hoyer and J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017. doi: 10.5334/jors.148. URL https://doi.org/10.5334/jors.148.

[7] Brenda Javornik, Hector Santiago III, and Jennifer C. DeHart. The lrose science gateway: One-stop shop for weather data, analysis, and expert advice. In *Practice and Experience*

in Advanced Research Computing, PEARC '21. ACM, July 2021. doi: 10.1145/3437359.3465595. URL http://dx.doi.org/10.1145/3437359.3465595.

[8] Kubernetes. Overview, 2019. URL https://kubernetes.io/docs/concepts/overview/. Last accessed by 17/12/2023.

[9] Valliappa Lakshmanan, Kurt Hondl, Corey K. Potvin, and David Preignitz. An improved method for estimating radar echo-top height. *Weather and Forecasting*, 28(2):481 – 488, 2013. doi: 10.1175/WAF-D-12-00084.1. URL https://journals.ametsoc.org/view/journals/wefo/28/2/waf-d-12-00084_1.xml.

[10] G. Latisen and G. Vossen. *Models and Languages of Object-Oriented Databases*. Addison Wesley, 1998.

[11] Lrose. Radxconvert - lrose wiki, 2021. URL http://wiki.lrose.net/index.php/RadxConvert.

[12] Met Office. *Cartopy: a cartographic python library with a Matplotlib interface*. Exeter, Devon, 2010 - 2015. URL https://scitools.org.uk/cartopy.

[13] opengeospatial. Ogc standard netcdf classic and 64-bit offset. https://www.opengeospatial.org/standards/netcdf, 2011. Archived from the original on 2017-11-30. Retrieved 2017-12-05.

[14] Russ Rew, Glenn Davis, Steve Emmerson, Cathy Cormack, John Caron, Robert Pincus, Ed Hartnett, Dennis Heimbigner, Lynton Appel, and Ward Fisher. Unidata netcdf, 1989. URL http://www.unidata.ucar.edu/software/netcdf/.

[15] Vineet Sajwan, Varnita Yadav, and M. Haider. The hadoop distributed file system: Architecture and internals. *International Journal of Combined Research & Development (IJCRD)*, pISSN:2321–2241, 05 2015.

[16] Michael Stonebraker and Uğur Çetintemel. 'one size fits all': An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, April 2005.

[17] Roland Stull. Weather Radars, 12 2022. [Last accessed by 17/12/2023].

[18] *RAW Product Format - IRIS Programming Guide - IRIS Radar*. Vaisala, 2024. URL `https://ftp.sigmet.vaisala.com/files/html_docs/IRIS-Programming-Guide-Webhelp/raw_product_format.html`.

[19] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.