Octavio Horacio Iacoponelli

TOWARDS DYNAMIC PARTITIONING FOR MEASUREMENTSET DATA FORMAT

Bachelor's Thesis

Supervised by
ENRIQUE MOLINA GIMÉNEZ

GRAU D'ENGINYERIA INFORMÀTICA

UNIVERSITAT ROVIRA i VIRGILI

Tarragona
June 2025

Summary

This thesis presents the viability research, design, and implementation of a dynamic processing plugin for astronomical MeasurementSets (.ms), enabling efficient partitioning and parallel execution. Initially developed independently, the plugin was later integrated into Dataplug and applied within the EXTRACT project. It allows cloud-native workflows to process large datasets efficiently, reducing bottlenecks in ingestion and calibration. Deployed in a Kubernetes execution backend, it achieves high speedups and demonstrates that modular, scalable pre-processing of astronomical data is both feasible and impactful for next-generation observatories.

Resum

Aquesta tesi presenta la recerca de viabilitat, el disseny i la implementació d'un *plugin* de procesament dinàmic per a fitxers MeasurementSets (.ms) astronòmics, que permet particións eficients i execució paral·lela. Desenvolupat inicialment de manera independent, el plugin es va integrar posteriorment a Dataplug i es va aplicar dins del projecte EXTRACT. Facilita que els fluxos de treball natius al núvol processin volums massius de dades de manera eficient, reduint colls d'ampolla en la ingesta i la calibració. Executat en un backend de Kubernetes, aconsegueix grans millores de rendiment i demostra que el preprocesament modular i escalable de dades astronòmiques és factible i rellevant per als observatoris de nova generació.

Resumen

Este trabajo presenta la investigación de viabilidad, diseño e implementación de un *plugin* de procesamiento dinámico para archivos astronómicos MeasurementSets (.ms), que permite una partición eficiente y ejecución en paralelo. Desarrollado inicialmente de forma independiente, el *plugin* fue posteriormente integrado en Dataplug y aplicado dentro del proyecto EXTRACT. Permite que flujos de trabajo nativos en la nube procesen grandes volúmenes de datos de manera eficiente, reduciendo cuellos de botella en la ingestión y calibración. Ejecutado sobre un backend de Kubernetes, logra mejoras significativas en rendimiento y demuestra que el preprocesamiento modular y escalable de datos astronómicos es viable y de alto impacto para observatorios de nueva generación.

Contents

List of Figures

1 Introduction

This thesis encompasses the research, design, and development of a programmatic method for a **dynamic approach** to the pre-processing of the MeasurementSet (MS) (.ms) format, a complex directory-based format used mainly in astronomical research [1].

A dynamic approach that allows us to work with smaller fractions of a larger MeasurementSet enables faster parallel processing of data and even makes it possible to handle large datasets within the cloud infrastructure - an option that was previously unfeasible due to memory limitations of remote workers such as AWS Lambdas [2] or similar serverless and function-as-a-service execution backends. Furthermore, it allows for the adaptation of fragment sized based not only on technical constraints but also on both economic and time considerations.

This capability is particularly relevant to the EXTRACT [3] project—an European-funded, open-source, data-driven distributed platform designed to facilitate the development of data mining workflows. In particular, dataset partitioning proves crucial in the TASKA [4] use case, which addresses the challenge of processing massive datasets acquired in near real time. Specifically for astronomical data, solar radio burst observations from the NenuFar telescope in France alone produce over 56TB of raw data per day.

Within the EXTRACT project and its use cases, Activity C of TASKA focuses on providing a Python cloud-based fast-paced data calibration and imaging pipeline [5]. The goal of this activity is to provide a modular and accessible workflow that integrates multiple existing scientific tools into reusable components, usable by both experts and non-specialist—thus **democratizing access** to solar radio data. A working prototype has already been developed, supporting a three-step process: data ingestion, calibration, and imaging. In this thesis, we focus primarily on the data preparation stage, as it remains the most inefficient and resource-demanding step.

Efficient pre-processing of MeasurementSets is a key enabler for analysis in radio astronomy, particularly as next-generation observatories are expected to generate petabyte-scale data streams. The methods developed in this thesis aim to contribute towards the construction of **cost-effective, native, scalable pipelines** capable of handling such volumes of data.

1.1. Project Objectives

The primary objective of this thesis is to research, design, and evaluate a flexible and efficient method for pre-processing astronomical data in MeasurementSet format. This

includes the development of a dynamic (on the fly) strategy that enables scalable data handling and parallel processing in a cloud-based environment.

The result of this process will be an application that is portable and easy to set up. In our case, the goal is to produce a plugin(or add-on) for an already existing project, Dataplug—a client-side framework that enables the partitioning of various types of scientific data stored in object storage[6]. Dataplug supports read-only pre-processing and 0-cost dataset re-partitioning, making it ideal as it avoids re-writing and duplication of datasets.

These objectives are justified by the growing need to process increasingly large volumes of astronomical data, within certain time and resource constraints. Traditional static or non-flexible approaches to data handling are no longer sufficient, specially when dealing with such rapid and large data streams. By addressing current bottlenecks in data-preparation stages, this thesis contributes directly to the goals of the EXTRACT initiative, particularly Activity C of the TASKA use case, which focuses on fast-paced, cloud-based data calibration and imaging. The proposed solution is intended not only to improve computational efficiency, but to also enhance accessibility and re-usability of scientific workflows for a broader community of researchers. It is designed to be sufficiently general to support most, if not all, standard MeasurementSets produced in recent years, and is not limited exclusively to the TASKA-C pipeline.

From a personal perspective, my motivation to undertake this task stemmed from my involvement in the in-house research group, CLoudLab, which was already working on the EXTRACT project. This environment exposed me to the technical challenges associated with larger scale data processing in a more scientific context. I have interests in optimization and handling of large datasets, particularly in distributed and cloud-based systems. This project also presented a compelling research and technical challenge, as it was not clear from the outset wether a viable and efficient solution was even feasible. This uncertainty made the problem all the more engaging and aligned well with my academics and professional interests.

1.1.1. Formative objectives

From an academic standpoint, the primary formative objective is to gain hands-on experience with applied research in a real-world context. By researching, designing and developing a solution, then integrating it as a full-feature plugin within an existing framework, I am aiming to expand and deepen my knowledge of how to work within the constraints and design patterns of pre-existing systems—an essential skill in both academic research and industry-level software development.

This thesis also allowed me to further explore the fundamentals of scientific research: formulating hypotheses, experimenting with solutions, and validating results through various measurable outcomes. Working on an open-ended technical problem has helped me strengthen skills like critical thinking, autonomy, and problem-solving.

Moreover, by participating with an actual scientific data-processing framework and contributing meaningfully to it, and by consequence a larger data-analysis project, I have the opportunity to bridge the gap between theoretical knowledge and practical appli-

cation—something especially valuable for a future career in either academia or industry.

1.2. Context and Motivation

Scientific and industrial fields are undergoing an intense transformation driven by the exponential growth in data volume, velocity, and variety. This is particularly evident in data-intensive domains like astronomy and climate science, where static or traditional processing methods are increasingly inadequate. Although data storage is becoming cheaper and cloud computing capabilities continue to improve, bandwidth remains a limiting factor, and the cost of processing large-scale datasets is still significant. Meanwhile, the demand for greater accuracy continues to rise in parallel with technological advancement. As a result, scalable, modular, parallelizable and efficient workflows have become essential rather than optional.

In this context, the EXTRACT project aims to support trustworthy, accurate, fair, and energy-efficient data mining workflows for extreme data. These workflows must handle massive, real-time datasets with low latency, high throughput, and efficient resource usage. One of the core principles of EXTRACT is the seamless integration of cloud, edge, and HPC technologies into a unified compute continuum. This enables dynamic adaptation to infrastructure constraints, from low-latency edge computing to high-throughput HPC environments. The platform also introduces advanced orchestration, monitoring, and security features under a single abstraction.

EXTRACT is validated through two real-world use cases, including the previously mentioned TASKA, which processes astronomical data from thousands of radio telescopes for near real-time solar activity analysis. This use case requires handling massive amounts of raw data, up to tens of terabytes of data daily, with data preparation—restructuring raw MeasurementSet files for calibration and imaging—being one of the current bottlenecks. My thesis addresses this challenge by developing a dynamic, scalable pre-processing component for MeasurementSet data.

1.3. Scope and Limitations

This project focuses on the dynamic (on-the-fly) pre-processing of MeasurementSet files, specifically those produced by the NenuFar telescope within the EXTRACT project. While tailored and tested for this use case, the solution is designed to be generalizable to other types of MeasurementSets. Since the MeasurementSet format is a deep and extensible specification, some extra work may be needed in the future to adapt our solution to new, specific cases.

The system that I implemented is optimized for cloud-based infrastructures, where scalability and parallelism are key. Performance on monolithic or local machines may be suboptimal, as this was not a design priority.

The tool is intended to function as a component within a larger data processing pipeline, and its integration depends on the constraints and design of both the hosting framework

(Dataplug) and the pipeline itself. For broader usability, a **minimal viable product** (MVP) should be defined first at the framework level, allowing any pipelines to adapt accordingly.

Finally, while the implementation is tested within the TASKA use case, the broader aim is to build a free, reusable and accessible tool that can serve the astronomic community beyond this specific scenario.

1.4. Document Structure

This document is structured into six main chapters, each building upon the previous to guide the reader from the project context to its implementation and evaluation.

Chapter 1: Introduction Introduces the research problem, objectives, and motivation behind the project. It outlines the relevance of dynamic partitioning of MeasurementSets within the broader context of astronomical data processing and its role in the EXTRACT project, with specific focus on the TASKA-C use case.

Chapter 2: General Description Presents the conceptual and technological background required to understand the problem space. It introduces key technologies such as serverless computing, object storage, and the Lithops framework, as well as the scientific context surrounding Casacore and the MeasurementSet format.

Chapter 3: Requirements Describes the engineering aspects of the project, including functional and non-functional requirements, constraints, and high-level design decisions.This chapter formalizes the technical foundation needed for a robust implementation.

Chapter 4: Implementation Details the system architecture, design logic, and development process. It is split into two major parts: the initial prototype built to validate core ideas in isolation, and its subsequent integration into the Dataplug framework. Code excerpts and figures are included to support explanations.

Chapter 5: Evaluation Evaluates the solution from two perspectives: data correctness and system performance. It compares the system's behavior using both small and large MeasurementSet files, testing its consistency, scalability, and efficiency in distributed environments like those provided by the EXTRACT project.

Chapter 6: Conclusions Summarizes the contributions of the work, reflects on the outcomes of the evaluation, and outlines possible future improvements and research directions.

2 General description

This thesis explores the viability, design, and implementation of a software solution for the logical partitioning and pre-processing of **MeasurementSet** files, a standard file format used in radio astronomy. Its primary goal is to enable efficient and scalable data preparation workflows that are fully compatible with cloud infrastructures, facilitating large-scale processing and analysis.

The solution is conceived as a flexible tool that can partition large datasets into smaller, manageable, fully functional fragments, allowing them to be ingested more effectively into distributed pipelines such as TASKA-C, one of the key use cases within the EXTRACT project. By enabling on-the-fly partitioning based on technical constraints or resource availability, the component addresses one of the least optimized phases in current astronomical data workflows.

Ultimately, it is intended to support scientists and engineers dealing with extreme data volumes, by offering a versatile tool that can pre-process data in read-only matter, therefore avoiding data duplication and large downloads and uploads.

2.1. Background

To understand the design decisions and relevance of this project, it is essential to review several foundational concepts found in modern data processing. This background section briefly introduces in an orderly manner the core technologies and architectural paradigms that enable scalable, distributed workflows in cloud environments—such as serverless computing, container orchestration, and object storage. It also presents the specific tools and formats used in radio astronomy.

2.1.1. Cloud Computing: General Overview

Cloud computing has fundamentally changed how computational resources are consumed and managed. Its evolution has moved from private, dedicated servers to virtualized infrastructure and services, culminating in today's highly abstracted models such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Function as a Service (FaaS) within serverless architectures [7].

Originally, organizations relied on dedicated hardware, managing everything from scalability to maintenance. Virtualization enabled the shift to IaaS, offering flexible, on-demand

access to computing resources. PaaS further abstracted infrastructure, allowing developers to focus on application logic. Most recently, FaaS has enabled developers to run event-driven functions without provisioning or managing servers—billed only for execution time and consumed resources.

Modern cloud computing is defined by three core principles:

-Scalability: the system's capacity to handle varying workloads by provisioning additional resources as needed.

-Elasticity: dynamic, often automatic adjustment of resources to align with demand.

-Pay-as-you-go: users are billed only for the resources they actually consume.

In this landscape, the cloud-native approach has emerged as a key design philosophy. Cloud-native applications are built for distributed environments, typically as microservices, packaged in containers, and managed via orchestration platforms such as Kubernetes. They often leverage serverless models like FaaS to increase modularity, efficiency, and resilience.

Key enabling technologies include:

-Containers, which encapsulate applications and their dependencies in lightweight, portable units.

-Orchestration, which automates deployment, scaling, and management of containers [8].

-Serverless / FaaS, which abstracts infrastructure entirely, triggering code execution on demand with automatic scaling and granular billing.

These developments form the foundation for hybrid and distributed computing paradigms, including edge computing and high-performance computing (HPC), where cloud-native components can be integrated with localized or specialized compute resources.

2.1.2. Serverless Architecture and Function-as-a-Service (FaaS)

Serverless computing is a cloud execution model in which developers deploy code in a transparent environment, without managing underlying infrastructure. Unlike traditional models, where applications run on reserved servers or containers, serverless abstracts server provisioning, scaling, and maintenance, enabling a focus purely on logic and events.

A common implementation of the serverless paradigm is Function-as-a-Service (FaaS), where discrete functions are triggered by events and executed on demand. Popular platforms include AWS Lambda, Google Cloud Functions, and Azure Functions, each offering a relatively fine-grained scalability and resource-based billing [9, 10].

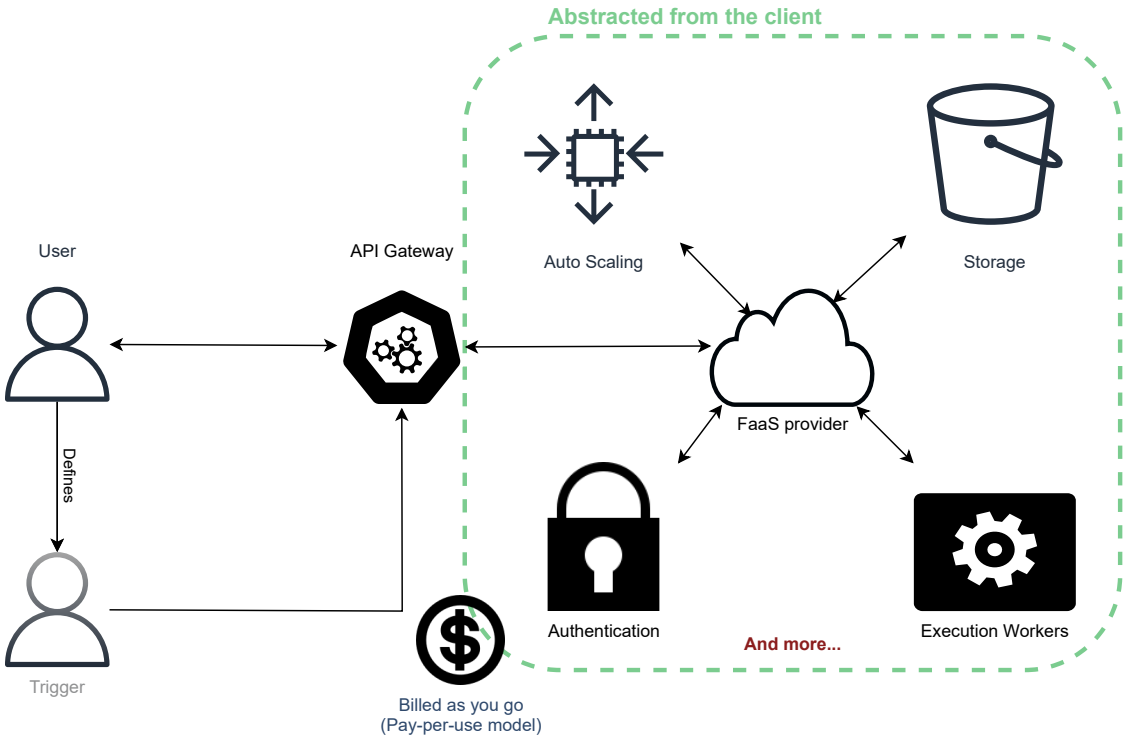

Figure 2.1: Function-as-a-Service (FaaS) model: an overview

Key advantages of the FaaS model include:

-Automatic scalability, as functions scale seamlessly with workload.

-Cost-efficiency, since users are only billed for actual function execution time, with no charges during idle periods.

-High parallelism, allowing many instances of the same function to run concurrently without manual configuration.

However, this model is not perfect and some limitations still remain [11]:

-Cold starts, which cause latency when a function is invoked after being idle.

-Resource constraints, as functions typically have time, memory, and CPU execution limits.

-Difficulty with calculating costs, as functions execution times can vary depending on the use case

Among other, more specific drawbacks that are heavily discussed in works where they may matter.

Serverless and moreover FaaS fits naturally within distributed architectures, particularly in event-driven and microservice-based systems. It simplifies backend logic in workflows, facilitates loosely coupled services, and enhances modularity—making it suitable for modern cloud-native and edge-integrated systems.

2.1.3. Containers and Orchestration: Docker and Kubernetes

While serverless architectures offer near complete abstraction from infrastructure, many cloud-native systems require extra control and flexibility. This need is addressed through containerization, which is a method for packaging applications together with their dependencies in lightweight, portable units.

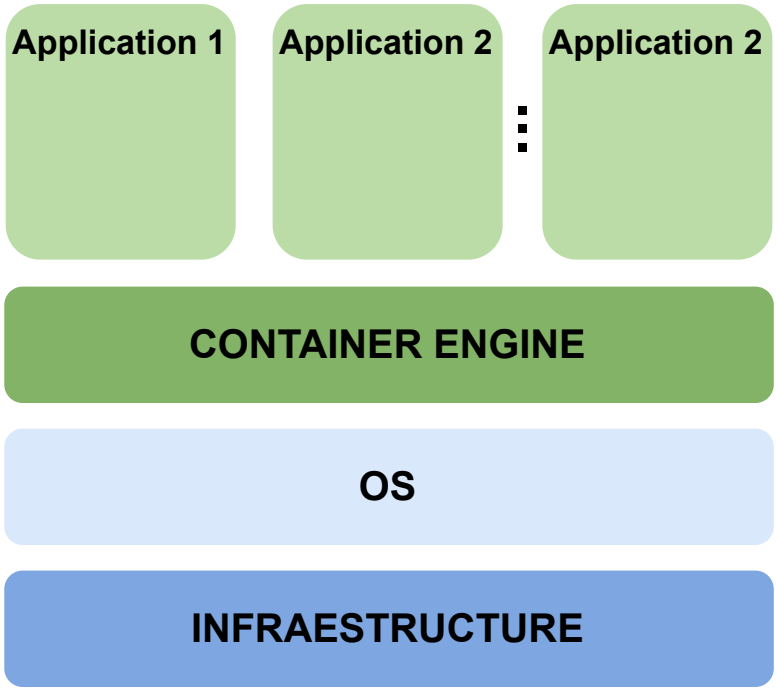

Figure 2.2: Containerization: Container engines work at the application level, allowing for fast and lightweight machines.

Docker is the most widely used platform for building and running containers. It allows developers to create consistent environments across development, testing, and production stages by using layered, version-controlled images [12]. Containers, unlike virtual machines, share the host operating system kernel, resulting in a lower overhead and faster startup times.

However, running containers at scale—especially in distributed systems—requires orchestration. This is where Kubernetes becomes essential. Developed by Google and now maintained by the CNCF, Kubernetes automates the deployment, scaling, networking, and management of containerized applications [13].

Kubernetes introduces concepts such as:

-Pods: the smallest deployable unit, often a single container or a tightly coupled group.

-Deployments: declarative definitions of application state, enabling rolling updates and rollback.

-Services: abstractions that expose groups of Pods via stable networking and load balancing.

Combined, Docker and Kubernetes form a powerful foundation for cloud-native development—offering high portability, scalability, and resilience for microservices and hybrid workloads.

2.1.4. Distributed Storage and Amazon S3

Modern cloud applications often require storage systems that are scalable, durable, and accessible across distributed environments. Distributed object storage addresses this by storing data as immutable objects, accessible via APIs, and replicated across nodes for fault tolerance and availability.

Amazon Simple Storage Service (S3) is the most widely adopted object storage platform [14]. It offers virtually unlimited capacity in the form of buckets, boosting high durability, and HTTP-based access, making it foundational to cloud-native architectures. A notable feature for our project is byte-range retrieval [15], which allows clients to read specific portions of an object—essential for efficient access to large datasets without needing to download or process the entire file.

In private or hybrid environments, MinIO provides a high-performance, S3-compatible alternative. It is cloud-native, lightweight, and if needed can be integrated seamlessly with Kubernetes, while retaining full compatibility with the S3 API [16].

These storage systems support parallel and partial access to objects, enabling multiple concurrent reads and writes, thus allowing us for distributed processing of data.

2.1.5. Lithops: A Serverless Computing Framework

Lithops is a Python-based serverless computing framework designed for massively parallel data processing over cloud object storage systems like Amazon S3 or MinIO [17]. It allows users to run thousands of functions concurrently across serverless platforms, virtual machines, or local environments—making it ideal for big data workloads.

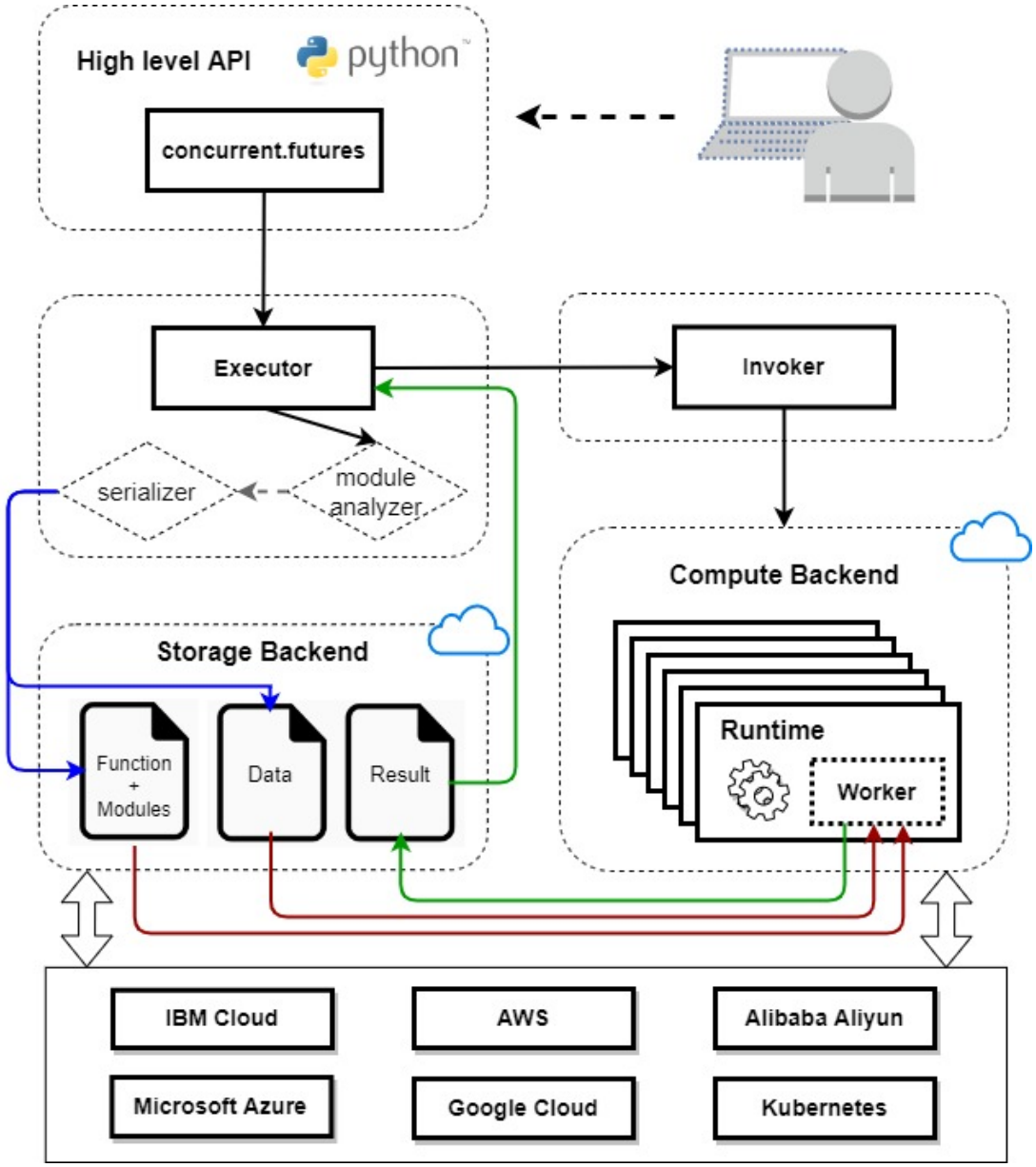

Figure 2.3: Lithops architecture diagram [18].Used for academic purposes.

Lithops plays a central role in our solution as the current main pipeline engine in the TASKA-C use case, where multi-cloud flexibility, configurability, and seamless integration with object storage are crucial. Its multi-backend architecture supports all major cloud providers and Kubernetes, where currently experiments are being run, allowing execution portability without code changes.

It supports three execution modes—localhost, serverless, and standalone—and offers two high-level compute APIs (Futures and Multiprocessing) along with native storage APIs.

In short, Lithops combines ease of use, cross-platform compatibility, and dynamic scalability, making it a powerful tool for orchestrating serverless data pipelines with minimal operational overhead.

2.1.6. Casacore and the MeasurementSet Format

Casacore is a C++ library developed to support radio astronomy data processing [19], especially the manipulation of Measurement Set (MS) data, which is the standard format for storing observational data in radio interferometry. An MS is a directory-based hierarchical format containing a main table and multiple sub-tables, which stores both data and metadata in a relational structure.

The Measurement Set format (v2.0) is defined to ensure flexibility, scalability, and rich metadata handling for complex observations of visibility and single-dish astronomical data. Although technically a collection of structured tables, it is treated as a single unit representing a calibrated or raw observation. This format also does not impose a restriction on file size, therefore files can easily stack up to GBs of data.

In this project, we interact with MS data using python-casacore, a Python binding to Casacore [20] that exposes the MS table interface, allowing custom processing and inspection via scripts, since the format allows multiple types of StorageManagers and virtualization logic that transform raw data into virtual columns. Additionally, we employ casatablebrowser, a lightweight Python GUI for visualizing MS contents outside of CASA.

These programs would prove essential to the development of a tool for pre-processing MS files and for testing them.

2.2. Current Context and Environment

Advances in radio astronomy have led to massive, complex datasets that demand scalable processing solutions. The EXTRACT project addresses this by enabling flexible, cloud-based data workflows. Its TASKA-C use case focuses on analyzing solar activity from NenuFAR telescope data, requiring efficient handling of 56TB of daily observations.

Processing these datasets—stored in the MeasurementSet (MS) format—poses challenges due to their size and structure. This section outlines the technological environment and the motivation for extending Dataplug with a plugin tailored for distributed MS processing.

2.2.1. Motivation and Problem Statement

Scientific data is often stored in an unstructured way, typically in large files and hosted in Object Storage systems. This approach allows researchers to easily access data without the need for extensive pre-processing or format transformation. In many ways, the flexibility of Object Storage - being able to store 'anything' as an object - is a significant advantage. However, this same flexibility can become a drawback, as it often leads to the storage of massive unconventional 'objects' ranging from gigabytes to terabytes in size.

In particular, for the **EXTRACT** project and the **Taska C** use case, we are dealing with up to **56 TB of daily readings** in the form of MeasurementSet files. This volume of data introduces several challenges to the current **TASKA-C** pipeline.

While a detailed technical review of the pipeline is beyond the scope of this project, it is important to outline its general structure. Originally, the entire pipeline was executed as a monolithic process—one that required a single instance to handle the full sequence from start to finish. This initial design consisted of five main stages: **rebinning**, **calibration**, **subtraction**, **applying calibration**, and **imaging**. To improve efficiency and reduce I/O overhead, these stages were later consolidated into three broader steps: **rebinning**, **calibration**, and **imaging**—laying the groundwork for a more modular and distributed solution. The final output of the pipeline is an astronomical image that scientists can later analyze.

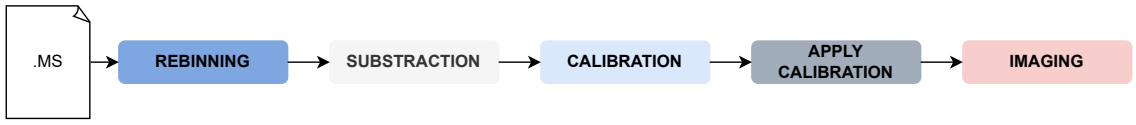

Figure 2.4: Taska C pipeline: Previous version.

Recent developments in the software have enabled the parallelization of four out of the five original stages. All steps except for imaging can now be executed in parallel, which significantly improves performance. However, the **imaging** stage remains a **monolithic reduce step**, and—at least for now—it cannot be parallelized.

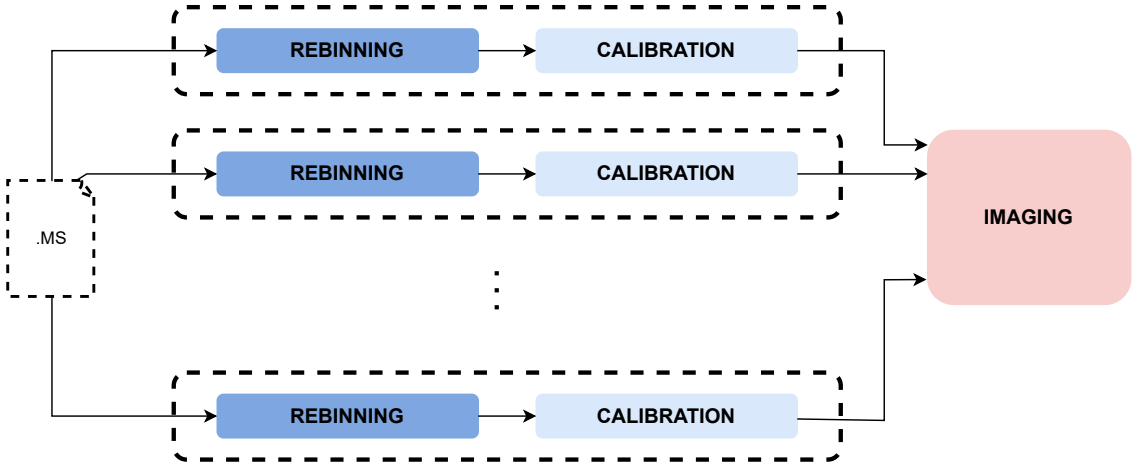

Figure 2.5: Taska C pipeline: Current version, where rebinning and calibration can be parallelized.

Those advances still leave us with two major bottlenecks:

1. **Imaging**: Mentioned above, this cannot currently be parallelized and lies outside the scope of this thesis.
2. **Data ingestion**: Equally important as imaging, if not more, given the ever-increasing size of datasets and the new need for distributed ingestion.

Today, scientific computing often benefits from remote or cloud-based execution environments. These allow researchers to pay only for the compute resources they use, avoiding

the need for expensive and quickly obsolete infrastructure. However, the current ingestion strategy presents significant limitations.

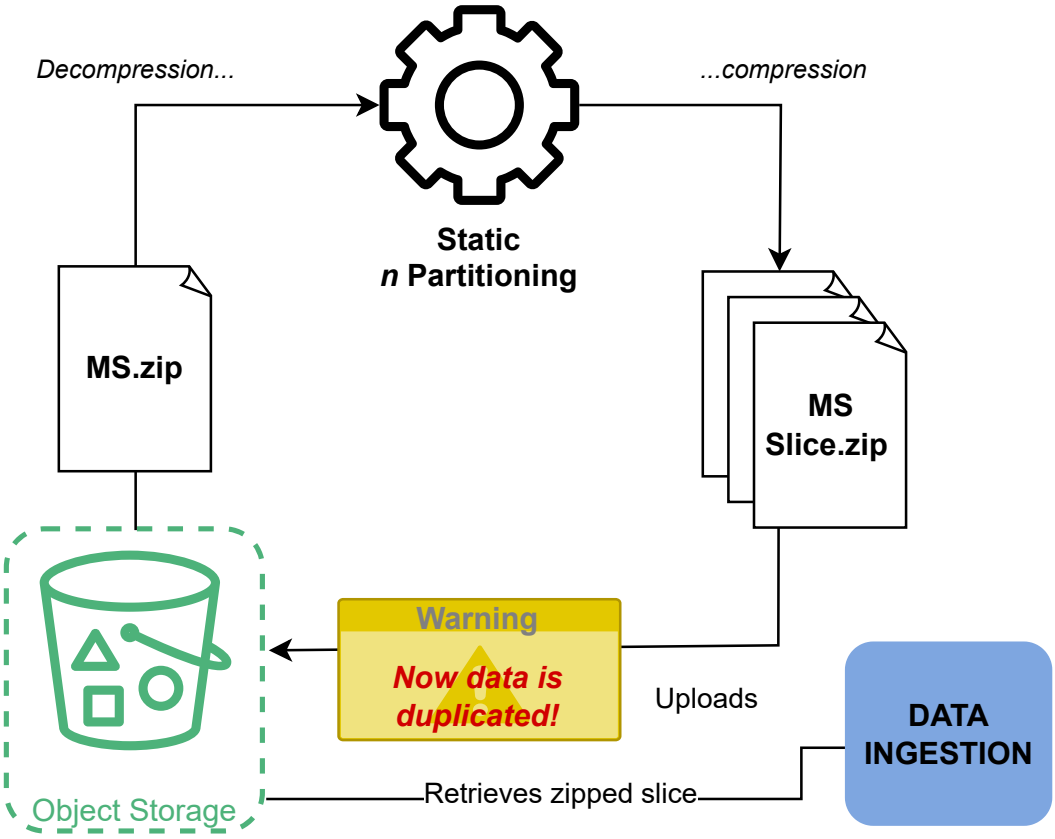

Figure 2.6: Current execution and ingestion logic: Data is duplicated and processing overhead is present.

In order to parallelize the pipeline effectively, the input data must first be fragmented. Currently, this fragmentation is handled in a **monolithic** manner: the entire file is downloaded, chunked locally, and then the resulting fragments are re-uploaded to object storage.

This approach introduces multiple issues:

- If fragmentation is executed by a cloud function (e.g., an AWS Lambda), we may hit memory limits. For instance, datasets larger than 9 GB would not fit in the memory-constrained environment (typically capped at 10 GB, some allowing up to 15GB).
- Even if memory were sufficient, the current logic requires downloading the full dataset, performing the chunking operation, and re-uploading the fragments. This not only duplicates the data but also adds unnecessary cost and latency.
- Furthermore, if the chosen chunk size turns out to be suboptimal, the entire process must be repeated from the beginning to generate new chunks.

These issues highlight the need for a more flexible and cloud-native solution. Ideally, we aim to **dynamically partition data**, minimizing—if not eliminating—the monolithic step in size-based fragmentation. Such a system would be more adaptive to cloud environments and improve both performance and cost-efficiency.

In particular, we need ingestion strategies that support **scalable**, **read-only**, and **cloud-native access** to partitioned data formats—especially for complex structures like MeasurementSets. These formats should be directly consumable by distributed systems without full pre-ingestion, allowing processing frameworks to retrieve only the necessary fragments on demand. Enabling this kind of access is essential for the scalability and future-readiness of data-intensive pipelines like Taska C.

2.2.2. Technological Environment Overview

This section provides a more detailed overview of the key technologies that support both the TASKA-C pipeline and the solution of the ingestion plugin. Understanding the technological environment is crucial, as the choice of tools and platforms directly impacts the system's scalability, performance, and ability to operate efficiently in a cloud-native setting. We will introduce each major component and explain its role within the architecture.

MeasurementSet Format and Casacore

Now, we need to explain how the **MeasurementSet (MS)** format works and why it may present a challenge.

A MeasurementSet is not a single file, but rather a structured, directory-based format used to store raw or processed data from radio telescopes. Conceptually, it resembles a **relational database**, where the main table contains core visibility data and several subtables contain metadata—such as antenna positions, feed information, spectral windows, and pointing data. These tables are internally represented using the **Casacore Table Data System (CTDS)** [21], which is highly flexible but not trivial to work with at a low level.

Each MS contains a set of predefined columns and sub-tables, some of which are required (e.g., the `DATA` column) and others optional, depending on the instrument or telescope. Moreover, observatories can define **instrument-specific extensions**—custom columns or sub-tables—which introduces further variation between MS's generated by different facilities.

One major architectural feature of the MS format is the use of **storage managers**. These define how table data is stored on disk and accessed in memory. Casacore supports several types: standard (plain), tiled (for chunked access), scalar (optimized for sparse updates), and combinations of these for specific access patterns. While this makes I/O efficient for astronomy workflows, it also adds complexity. Understanding how each MS is structured requires inspecting both table definitions and how storage is managed internally.

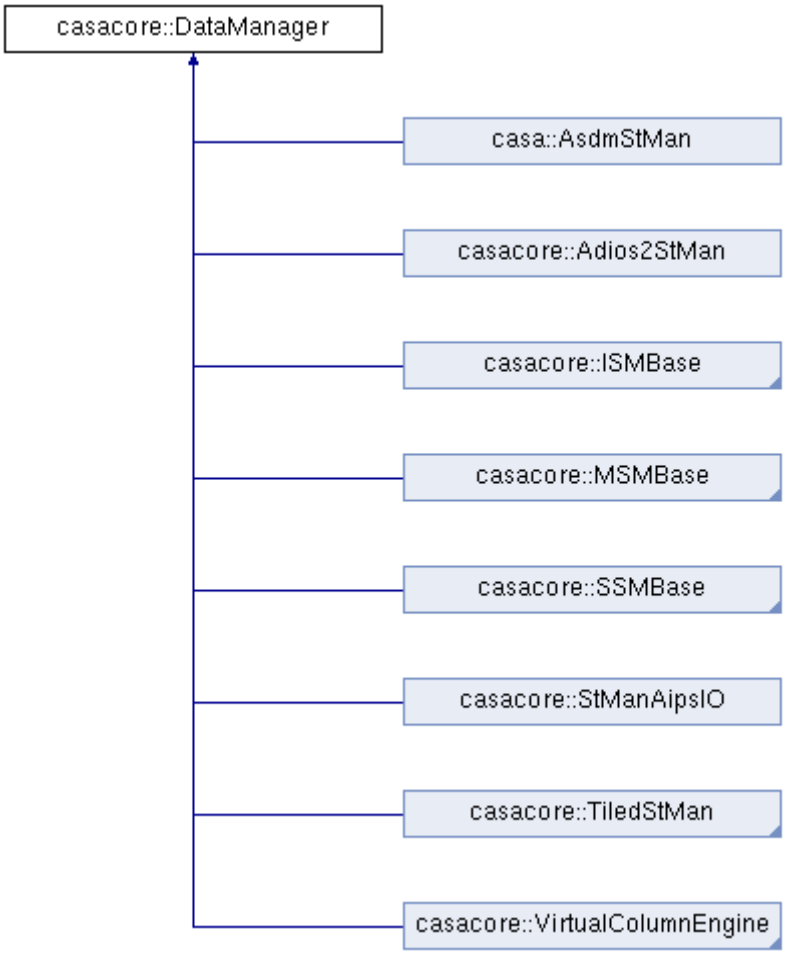

Figure 2.7: Excerpt from the documentation of `casacore::DataManager`, showing oficially supported DataManagers [22].

From a tooling perspective, working directly with the underlying files that compose a MeasurementSet is non-trivial. The internal format is not flat or well documented for manual parsing [23], and direct manipulation (by bypassing Casa logic) can easily lead to **data loss, inconsistencies, or corruption**. As a result, MS files are usually accessed through Casacore or a compatible high-level library.

This is where the **Casacore library** becomes essential. Casacore is a C++ library originally developed as part of the AIPS++ project, designed to provide infrastructure for working with MeasurementSets and other astronomy-related data models. It includes classes for reading/writing tables, managing metadata, accessing sub-tables, and interpreting units and coordinate systems via the AIPS++ **Measures** framework—which we will not explore in this project.

Casacore abstracts the MeasurementSet directory structure into a unified interface, enabling access to the main table and its sub-tables via methods like `ms.tableinfo()` or `ms.showstructure()`, which return metadata and structural details of the MS. While powerful, this abstraction adds some overhead: Casacore prioritizes **correctness, consistency, and flexibility**, so even simple operations may involve validating metadata, parsing keywords, or handling file locks.

To simplify integration in modern Python-based data pipelines, we use `python-casacore`, a Python binding that allows us to open and manipulate MeasurementSet tables directly from Python—accessing sub-tables, retrieving metadata, and performing queries—without needing to drop into C++.

In summary, while standardized, the MeasurementSet format imposes non-trivial constraints when building systems that aim to manipulate these datasets efficiently. Its directory-based layout, use of multiple storage managers, and reliance on sub-tables make low-level manipulation difficult and error-prone. Although our goal is to implement a **partitioning solution that minimizes dependency on Casacore**—favoring lightweight, direct access patterns—Casacore has nonetheless proven useful during exploration and prototyping, and remains partially integrated into the final version of our plugin. Notably, it enabled us to statically split an MS by subsetting specific columns (e.g., via `TIME`) in the main data table. This technique will become especially relevant when we describe how partitioning was integrated into the **TASKA-C** processing pipeline.

Lithops

Lithops provides a versatile and extensible foundation for executing distributed workloads with minimal effort from the user. In our use case, Lithops serves as the execution backend, but the interaction with it was largely *transparent*, thanks to its integration via **Flexecutor** [24], which handles orchestration and abstraction of the low-level execution details.

One of Lithops' strengths is its *flexible configuration model*. Through a simple YAML-based config file, users can switch between local and distributed execution environments by changing just a few keys. For example, setting the `backend` to `localhost` allowed us to to run small-scale tests, while switching to a `kubernetes` (or some other cloud solution) enabled full-scale cloud deployments—provided the infrastructure is properly set up.

In our deployment, **MinIO** is used as the object storage backend, fully supported by Lithops as a drop-in alternative to services like Amazon S3. Lithops natively integrates with MinIO, enabling efficient parallel access to input/output data during pipeline execution.

Another critical feature is Lithops' *log polling mechanism*, which periodically fetches logs from remote workers. This provides near-real-time insight into the progress of distributed jobs without requiring direct access to the execution environment.

Lithops also supports *custom Docker runtimes*, which is especially relevant in Kubernetes deployments. In these setups, each worker pod can launch a container with a custom runtime that includes all necessary dependencies for a specific function. This allows for precise control over the execution environment and ensures portability and reproducibility of results across platforms.

Overall, Lithops offers a robust and highly configurable compute layer that powers our serverless data pipeline, **Flexecutor**, that we will discuss in more detail later.

Object Storage and Kubernetes

Two essential components in our setup are MinIO, an S3-compatible object storage system, and Kubernetes, a container orchestration platform.

MinIO for Distributed Object Storage: MinIO serves as the primary data layer, being an analog to S3, providing efficient access to large scientific datasets through byte-range requests. This feature is critical for Flexecutor, where data is processed in slices and retrieved on-demand. The object store ALSO acts as a shared medium across stateless function invocations for Lithops, enabling scalable and storage-aware computation.

Kubernetes for Function Orchestration: Kubernetes underpins the Lithops execution model by managing the lifecycle of worker pods that instantiate custom Docker runtimes. These runtimes package domain-specific tools (e.g., DP3, WSClean) and are launched dynamically as part of Flexecutor pipelines. Kubernetes ensures isolation, scaling, and reliability.

Together, MinIO and Kubernetes form the infrastructural backbone of our serverless pipeline, aligning with the storage-centric and execution-flexible design required for scientific workloads.

2.2.3. Within the EXTRACT Project (TASKA-C Use Case)

The **EXTRACT** project aims to enable scalable and efficient data-intensive workflows across the compute continuum—edge, cloud, and HPC—through modular, serverless, and storage-aware architectures. By addressing challenges related to data mobility, parallel execution, and infrastructure abstraction, EXTRACT empowers scientific and industrial applications to process extreme volumes of data using flexible, cloud-native tools. A central demonstration of these capabilities is the **TASKA-C use case**, focused on real-time solar imaging with data from the NenuFAR radio telescope in Nançay, France.

The TASKA-C (**C**loud-based fast-paced data calibration and imaging.) pipeline exemplifies a data-intensive scenario in radio astronomy, where large and variable volumes of raw observational data—stored in the MeasurementSet (MS) format—must be processed rapidly to produce science-ready images of solar activity. These images are vital for studying low-frequency solar emissions, particularly for predicting the impact of coronal mass ejections on Earth.

To meet the requirements of this use case, EXTRACT integrates a function-oriented execution model powered by **Flexecutor**, an internal and unpublished tool developed at URV built on the Lithops serverless computing framework. This approach enables a fine-grained decomposition of the imaging workflow into independent tasks, dispatched in parallel across cloud-native infrastructures. It also incorporates logic for predictive performance based optimization.

Altogether, the TASKA-C use case not only showcases EXTRACT's potential for transforming scientific workflows but also lays the groundwork for reproducible, accessible,

and high-performance data processing pipelines that will be critical for the Square Kilometre Array (SKA). Through its modular design and serverless foundations, EXTRACT opens the door to broader adoption by non-specialists while enabling expert users to tailor workflows to complex, evolving research needs.

Introduction to Flexecutor: DAG-Based Orchestration over Lithops(REVISAR!)

Flexecutor is an orchestration framework built on top of Lithops, designed to execute scientific workflows expressed as Directed Acyclic Graphs (DAGs). Its main contribution is to bring **serverless smart provisioning** capabilities to Lithops—enabling intelligent and adaptive deployment of functions in the cloud, based on the profile of each pipeline stage. It also helps abstracting the representation of workflows, where each stage can have it's own input/output bindings, named *FlexIputs*, concurrency levels, parameters, resulting in a high-configurable and transparent tool.

Serverless Smart Provisioning

The serverless smart provisioning paradigm decouples the definition of workflows from the provisioning of computational resources. Instead of assigning static resource configurations, Flexecutor dynamically selects the optimal setup (e.g., number of workers, CPU, memory) for each stage based on performance models and user-defined constraints—such as performance, cost, and overall cost-effectiveness (modeled as the inverse of cost multiplied by the inverse of performance). This enables **elastic scaling**, performance-cost **trade-off exploration** using predictions and optimization and **transparent automation** of resource tuning, **without modifying the application code**.

State of the Art

Flexecutor's smart provisioning approach is grounded in recent advances in serverless orchestration and adaptive resource management. Notable related systems include:

- **Caerus:** A serverless analytics scheduler that uses fine-grained profiling and the NIMBLE algorithm to minimize cost and job completion time by modeling task dependencies and execution rates [25].
- **Orion:** An end-to-end autonomous driving system that integrates vision-language models and trajectory planning, aligning reasoning with action spaces to optimize decision-making performance [26].
- **Jolteon:** The main inspiration for Flexecutor. Jolteon combines white-box and black-box modeling to optimize serverless workflows under latency or cost constraints. It formulates a chance-constrained optimization problem and explores deployment plans using sampling and convexity techniques. Flexecutor adopts similar ideas for stage-level prediction, training, and dynamic configuration selection [27].
- **DITTO:** A framework for controlling pre-trained text-to-music diffusion models at inference time by optimizing noise latents, enabling training-free control over structure and style [28].

These systems share a common goal: automating complex resource and execution decisions to meet user-defined performance objectives. The result is an execution framework that adapts to context.

Flexecutor: Core Directives

Flexecutor relies on four main directives that manage profiling, modeling, prediction, and optimization of distributed workflows. These directives work together to enable intelligent, resource-aware execution.

1. **`profile()`: Gather Performance Data**
   Executes the workflow across different resource settings and collects performance metrics such as execution time and cost. This data is used to build performance models.
2. **`train()`: Build Predictive Models**
   Uses the data from profiling to train a performance model for each stage in the workflow. Training can be applied globally or per stage. (is this true? unsure)
3. **`predict()`: Estimate Execution Time**
   Predicts the expected runtime of each stage given a resource configuration using the trained models. Helps evaluate what-if scenarios without actual execution.
4. **`optimize()`: Find Optimal Resources**
   Searches the configuration space to determine the best combination of resource parameters that minimizes cost, time, or cost-effectiveness.

Flexecutor: Running the Workflow

A typical Flexecutor pipeline consists of:

- **Stages**: units of computation defined by a function and metadata (inputs, outputs, parameters),
- **DAG**: a directed acyclic graph defining stage dependencies,
- **DAGExecutor**: the component managing parallel execution using Lithops.

Stage connections use intuitive syntax:

```
rebinning_stage >> calibration_stage >> imaging_stage
```

Stages also specify input/output objects (`FlexInput`, `FlexOutput`) and may use broadcasting for common resources like Lua scripts or databases.

Each script must be decorated with `@flexorchestrator(bucket="...")`, which handles object storage I/O and initializes the orchestration context. The object store location is defined in Lithops configuration.

Flexecutor supports two chunking strategies:

- *Static chunking*: splits Measurement Sets (MS) by fixed time intervals, zips the chunks, and uploads them to storage.

- *Dynamic chunking*: with `Chunker(ChunkerTypeEnum.DYNAMIC)`, partitions are created at runtime based on the number of available workers, minimizing data movement. This was pending a Dataplug implementation, as of this thesis, this is now available and works as intended.

Flexecutor: Radio Interferometry pipeline

Our pipeline includes:

- **Rebinning (dp3)**: averaging and flagging visibility data,
- **Calibration**: applying gain solutions,
- **Subtraction**: removing strong sources,
- **Imaging (wsclean)**: producing the final image.

Artifacts such as logs, models, and output products are stored in structured directories (e.g., `profiling/`, `logs/`, `models/`), aiding reproducibility and debugging.

As mentioned before, although not fully tested yet, Flexecutor integrates Jolteon-like optimization strategies [27], enabling dynamic resource provisioning and cost/performance-aware tuning through Dataplug, something that before adding on-the-fly partitioning was not a viable option.

Flexecutor's DAG-based orchestration over Lithops yields a serverless, cloud-agnostic pipeline that scales seamlessly from local testing to cluster-scale execution.

Limitations of the Previous Pipeline

In the earlier implementation of the TASKA-C pipeline executed via Flexecutor, the rebinning stage required the MeasurementSet (MS) to be pre-fragmented into smaller, independently usable chunks to enable parallel processing. However, this process, was burdened by several inefficiencies.

First, the MS format,being a directory-based structure,necessitated either packaging into a single file before upload or downloading and reconstructing it locally. If the file was uploaded as-is, a pre-processing step was required to zip the contents beforehand. In some cases, the dataset had to be downloaded, zipped locally, and then uploaded again for later processing, introducing redundant data transfers and processing time.

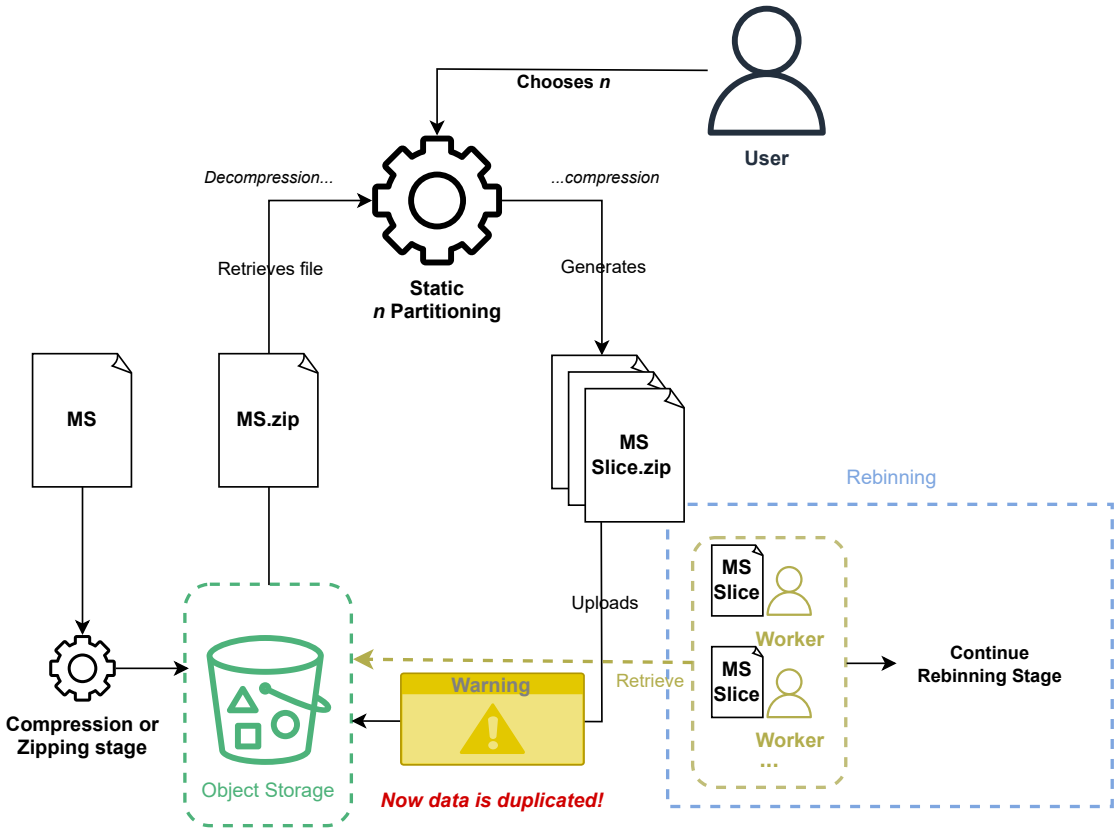

Figure 2.8: Taska C pipeline: Detail on the current ingestion stage and static partitioning

Once stored, fragmenting the MS for distributed processing introduced further complications. Partitioning relied heavily on the Casacore library, which was used to organize the dataset based on the `TIME` column and then split it accordingly. However, this operation was both memory-intensive and time-consuming. Casacore's virtualization layer adds some overhead, and crucially, it required access to the entire unzipped MS file in memory to operate—making partial or lazy evaluation impossible.

Moreover, each resulting slice had to be individually zipped and uploaded back to object storage. Since each slice includes necessary metadata to be independently functional, the combined size of all partitions often exceeded that of the original MS file. This not only duplicated storage usage, but also created significant I/O and compute overhead.

The pipeline's design also posed limitations for serverless computing. Function-as-a-Service (FaaS) platforms such as AWS Lambda typically impose memory limits—10GB in Lambda's case—that constrain processing and are often comparable across providers, meaning that large MS files could not be processed using cloud functions. This forced the use of more heavyweight options such as virtualization, containerization or local execution, which limited scalability, portability and cost-efficiency.

Finally, any miss-configuration—such as oversized slices—meant the entire pre-processing workflow (download, unzip, split, zip, and upload) had to be repeated from scratch. This inefficiency made it difficult and impractical to iterate or optimize partitioning strategies. It also hindered predictive resource provisioning and adaptive execution, both of which

are essential for achieving runtime efficiency in serverless or hybrid cloud workflows.

Summary of Existing Deficiencies

- Entire MeasurementSet must be downloaded and unpacked for processing.
- Static slicing introduces redundant uploads and duplicated storage.
- Repartitioning requires repeating the entire pre-processing pipeline.
- Partitioning logic is tightly coupled to Casacore, limiting flexibility.
- Serverless functions cannot process large MS files due to memory constraints.
- Smart provisioning is inefficient due to the lack of dynamic access.

2.2.4. Cloud-Native Partitioning with Dataplug and the Astronomy Plugin

To enable scalable and elastic processing of radio astronomy data in the cloud, our system leverages Dataplug, a lightweight and extensible framework for dynamic, cloud-native partitioning of unstructured scientific data stored in object storage systems.

Dataplug provides a read-only, pre-processing-based partitioning mechanism that builds small, decoupled indexes on top of immutable object storage files(Metadata). These indexes enable efficient, dynamic slicing of datasets without altering or copying the original data. The system is cloud-aware: it exploits S3-like cloud storage solutions support for HTTP GET Byte-range requests to perform high-throughput, parallel access to raw data objects, overcoming typical latency bottlenecks of object storage through massive parallelism.

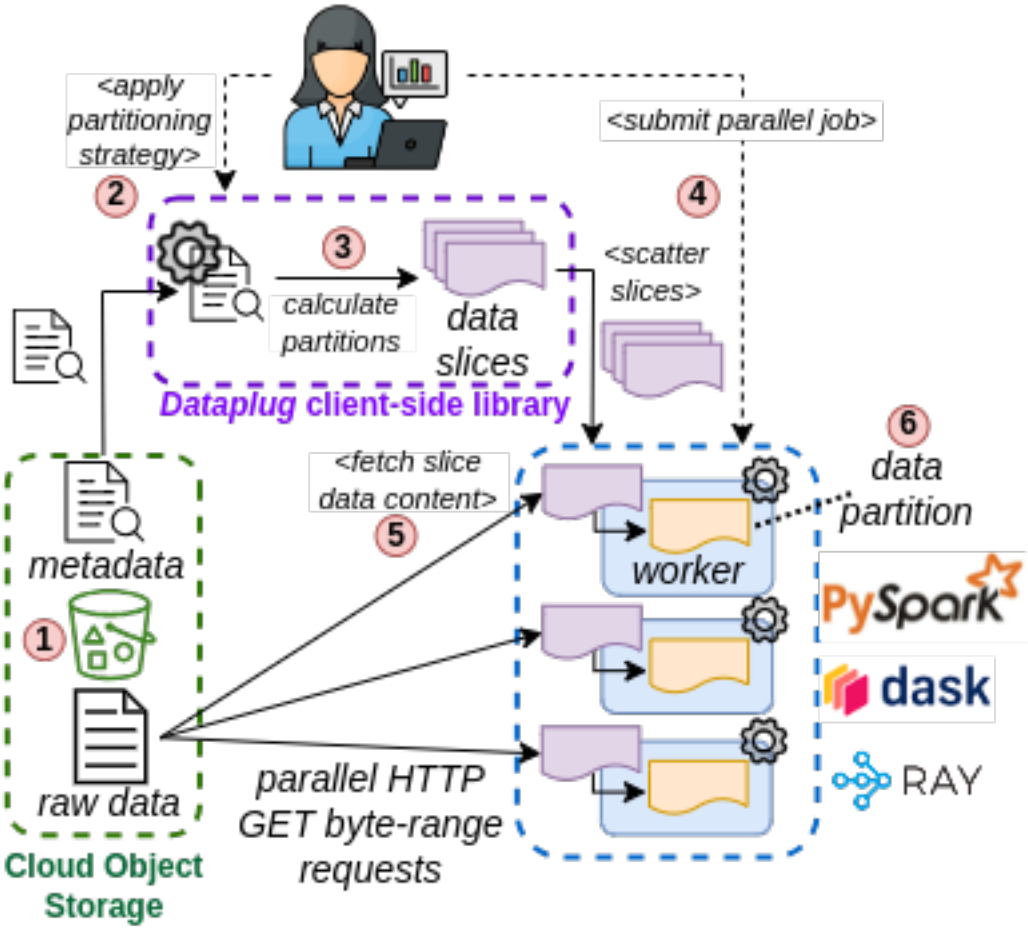

Figure 2.9: Dataplug framework architecture. Source: [29].

A central abstraction in Dataplug is the **data slice**—a lazily-evaluated partition of a dataset that is serializable and can be passed to distributed computing frameworks. Evaluation of a data slice occurs at runtime, triggering actual byte-range reads directly from S3. This deferred access model allows for extremely flexible and elastic data workflows without the need for intermediate formats or data materialization.

Dataplug supports a plug-in interface, making it adaptable to diverse scientific data formats. Existing plugins include support for:

Genomics: FASTA, FASTQ, VCF

Geospatial: LiDAR, Cloud-Optimized GeoTIFF (COG), Cloud-Optimized Point Cloud

Metabolomics: imzML

Generic formats: CSV, raw text

Each plugin implements format-specific logic for pre-processing and partitioning, translating high-level domain abstractions (e.g., genomic reads, raster tiles, point clouds) into addressable byte-ranges on object storage.

To extend this capability to radio astronomy, we developed a new Astronomic Plugin that supports the MeasurementSet (MS) format used by the CASA software suite. MS files are structured as directories containing nested tables and sub-tables, describing observational

metadata and visibility data in a relational layout. Partitioning these files efficiently requires an understanding of domain-specific groupings.

The plugin developed in this project leverages CASA's table system to extract partitioning metadata and build custom Dataplug indexes. Datasets can be partitioned at the row level, with granularity primarily determined by the number of data snapshots captured per second by the hardware. The resulting data slices reference only the relevant byte ranges of the corresponding MeasurementSet tables and sub-tables, enabling efficient, on-demand access and distributed processing directly from object storage. Currently, access is performed sequentially, but the design allows for straightforward parallelization if greater performance is required.

This integration allows us to retain the original MeasurementSet format throughout the workflow, avoiding costly and potentially lossy conversions, while enabling scalable, cloud-native access patterns. The resulting pipeline is more efficient and flexible, especially in distributed environments.

Modified Pipeline Behavior with Dataplug

By integrating Dataplug, the pre-processing step can now be offloaded to a worker node with direct access to object storage. Since only partial reads are required to extract partitioning metadata, the worker creates a lightweight index and template without downloading the entire dataset. This metadata is uploaded separately, eliminating the need for data duplication or transformation.

During execution, remote workers use these indexes to perform parallel HTTP GET byte-range requests directly to the raw MeasurementSet tables. This enables highly parallel, on-demand access without modifying the original data, significantly improving scalability and performance in cloud-based distributed processing.

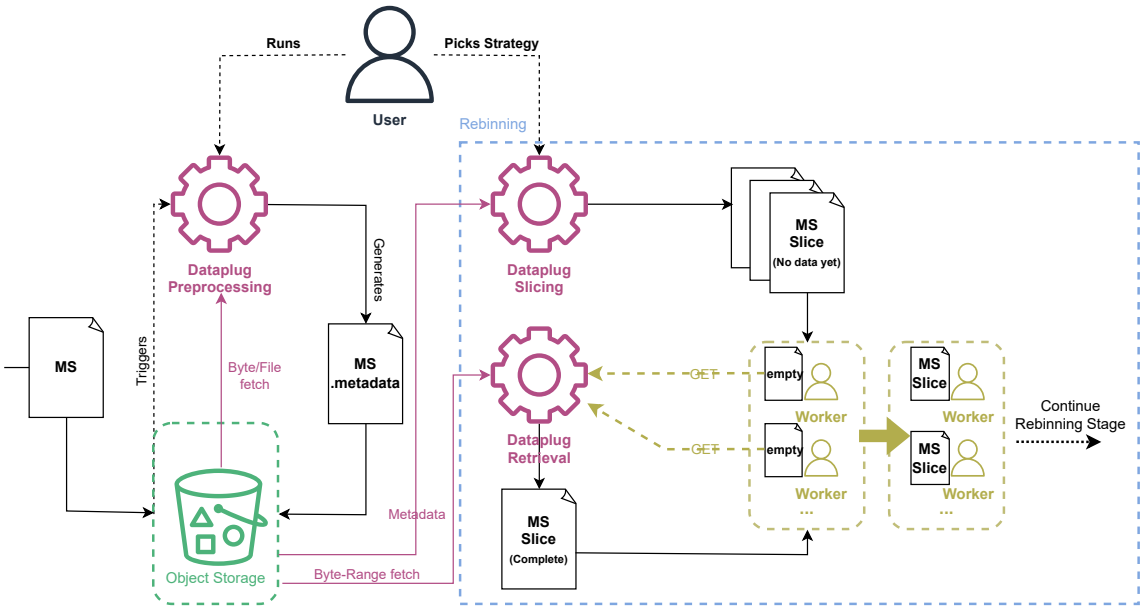

Figure 2.10: Taska C pipeline: New and improved workflow, now truly distributed and dinamically scalable.

2.2.5. From Prototype to Dataplug Integration

The core research effort of this project initially focused on whether a more dynamic approach to MeasurementSet partitioning was even feasible—especially one compatible with Dataplug. Dataplug was chosen early on due to its clean plug-in interface and sound architectural design, which served as a natural foundation to mirror during development, also, it was mainly developed by Aitor Arjona under one of URVs research teams, CloudLab[30].

Through multiple iterations, extensive testing, and detailed examination of both the CASACORE notes and MeasurementSet documentation, we developed an early working prototype. Tools such as the CASA Table Browser were used extensively to inspect structural integrity after partitioning and recombination. Byte-level comparisons were also conducted to pinpoint discrepancies in file layouts. This standalone prototype was invaluable—it allowed us to rapidly test partitioning logic in isolation, without having to embed directly into Dataplug's pipeline.

Once the prototype proved viable, we adapted it for full integration into Dataplug. This required a small core modification: enabling Dataplug to treat directory-based datasets as first-class data sources. Unlike flat files, directories in object storage are logical constructs rather than discrete key-value pairs, so support had to be explicitly added. From that point on, all further development, refinement, and bug-fixing occurred within Dataplug's framework.

The integration was relatively seamless, thanks to our prototype being modeled closely on Dataplug's principles. The remaining issues were primarily related to edge cases in MeasurementSet manipulation, which only surfaced during final pipeline testing. Once resolved, the plugin reached a complete and usable state—though further enhancements remain possible, as discussed in later sections.

Overall, this prototyping phase involved considerable work, much of which is hidden in the final Dataplug version. While the original prototype is not currently planned for release as a separate library, this remains a future option if broader reuse is deemed worthwhile.

2.2.6. Conclusion: Why This Plugin Matters

This plugin brings MeasurementSet handling into the realm of cloud-native, elastic data processing. By leveraging Dataplug's read-only, index-based access model, our approach avoids expensive data duplication and transformations, enabling true on-demand, distributed reads directly from object storage.

It removes the need for redundant uploads or monolithic pre-processing, allowing metadata generation to be performed independently and with minimal data access. This design not only improves performance and scalability but also paves the way for efficient smart provisioning and future extensions to other astronomy-specific data formats.

In essence, the plugin closes the gap between the domain-specific structure of radio astronomy data and the general-purpose demands of modern cloud infrastructure—making scalable, elastic processing of raw MeasurementSet data both practical and future-ready.

3 Requirements

While this thesis is primarily research-driven with a practical implementation component, presenting the project's requirement is worthwhile, particularly as this phase took on a more engineering-oriented task.

3.1. Initial Requirements

The project consists of evaluation, design and implementation of a plugin capable of pre-processing Measurement Sets (MS), a standard file format utilized in radio astronomy. The tool aims to facilitate partitioning and metadata handling in order to improve data accessibility and processing efficiency.

Requirements are divided into:

- **Functional requirements:** define what the plugin must do.
- **Non-functional requirements:** define constraints such as performance or compatibility.

3.2. Scope

This work focuses on

- Reading and analyzing existing MS files.
- Extracting and managing metadata relevant to partitioning.
- Applying slicing strategies to split the data.
- Writing output in a consistent and functional way.

Out of scope: Extracting and re-implementing crucial casacore operations to a self-contained and reduced python environment in order to minimize the overhead that third-party software may provide.

3.3. Assumptions and Constraints

Assumptions:

- Input MS files are valid, follow the standard and have no uncommon or custom storage managers
- Sufficient metadata is available or derivable directly from casacore.

Constraints:

- Original MS files should not be modified.
- Processing must scale to large datasets.
- The solution must remain lightweight and portable.

3.4. Functional Requirements

- Load a Measurement Set and inspect its metadata.
- Clone a template MS structure without data for parallel use.
- Derive slicing strategies.
- Apply retrieval and generate new structured outputs (MS files).

3.5. Non-Functional Requirements

- **Efficiency:** must handle large datasets with minimal overhead.
- **Modularity:** implementation must allow easy extensions or integration.
- **Portability:** should run in standard Python environments.

3.6. Design Overview

To illustrate the overall design, the following diagrams are provided:

- **Figure 3.1:** Class diagram representing the main components and their relationships.
- **Figure 3.2:** Sequence diagram showing the flow for the slicing use case.

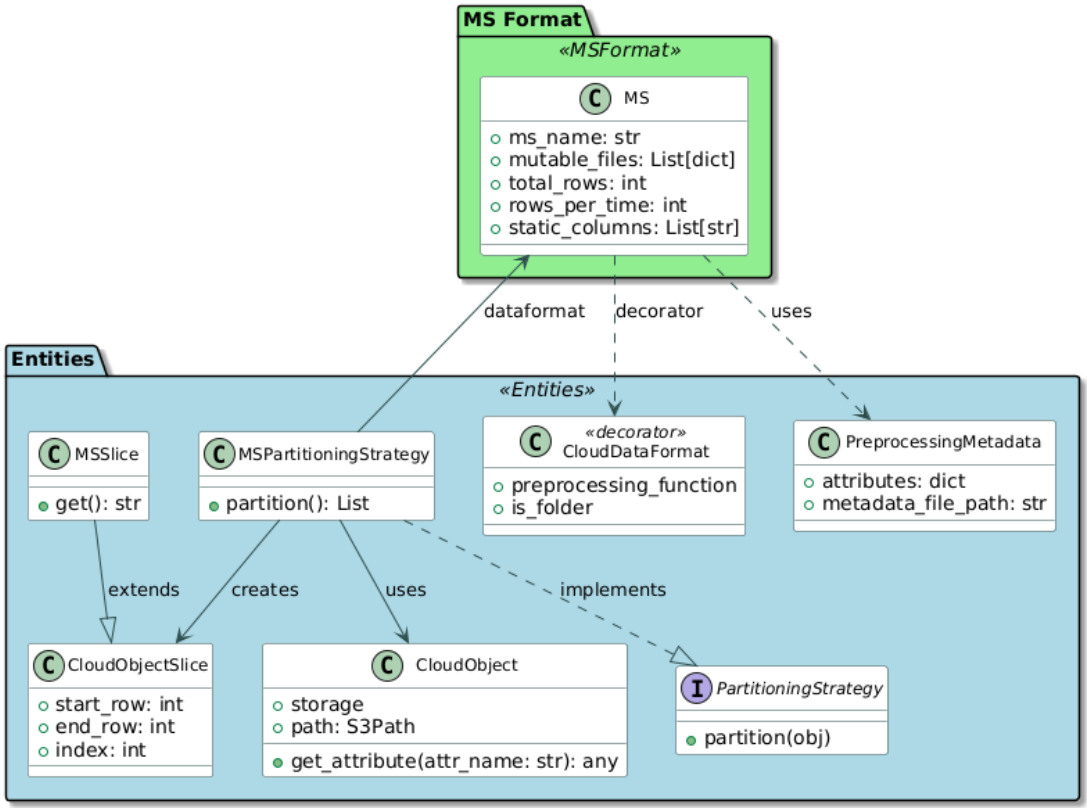

Figure 3.1: Class Diagram: A simplified view to how our plugin is structured

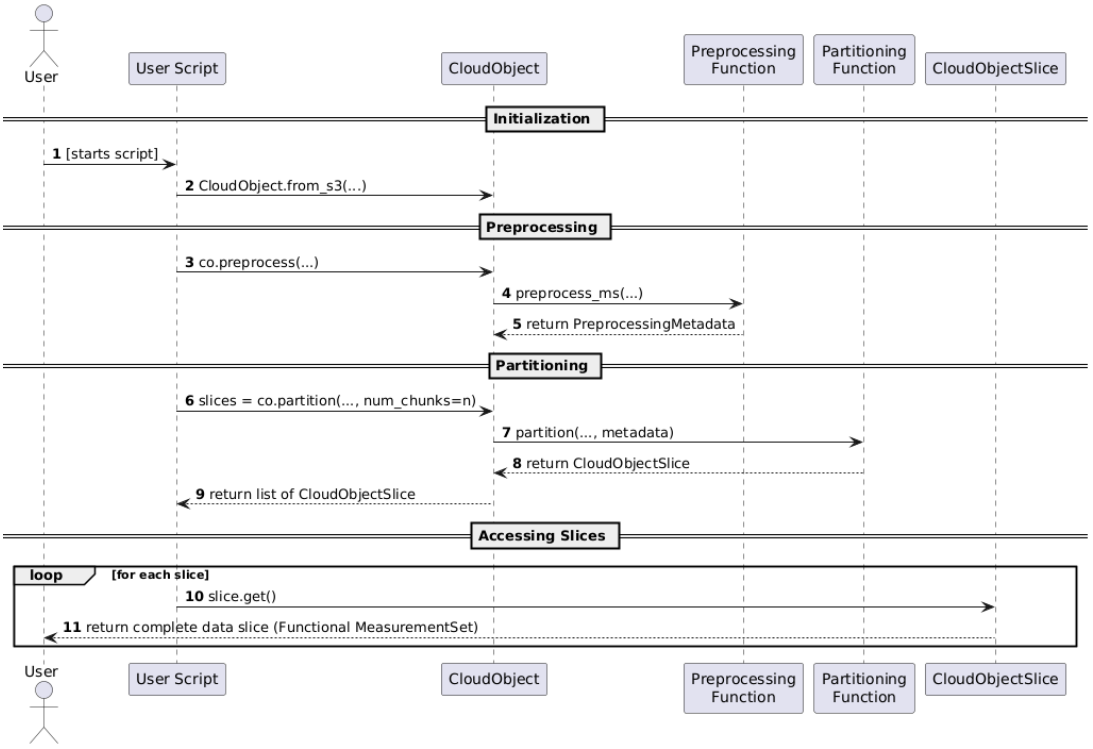

Figure 3.2: Sequence Diagram: Flow of execution of our plugin

4 Implementation

This chapter describes the technical implementation of our project, detailing the key design decisions, the logic behind its components, and how they interact. The objective is to walk through the development process in a way that is both illustrative and reproducible.

Where appropriate, we include code snippets and metadata examples to clarify how specific parts of the system work. These are presented in context, accompanied by explanations that justify our choices or highlight important behavior.

The chapter is structured in two main sections. First, we explore the standalone prototype, developed prior to full integration. This initial version was crucial for testing core concepts, validating assumptions, and defining the data flow in isolation. Next, we describe how this prototype was adapted and integrated into the Dataplug framework, completing the system's operational context and enabling it to work as part of a larger pipeline (TASKA-C).

4.1. Prototype: Precursor to Dataplug Integration

Before integrating any plugin into a production environment—in our case, particularly into the Dataplug framework—we first needed to design and implement a working prototype. This allowed us to validate our ideas in isolation and iterate quickly. In this section, we walk through the prototype's architecture and execution phases.

4.1.1. Overview

The prototype consists of a modular set of Python scripts, each responsible for a specific task in the pipeline. This compartmentalization avoids monolithic execution and improves maintainability and testability. It also allows for future parallelization and remote execution, aligning with the design patterns of Dataplug.

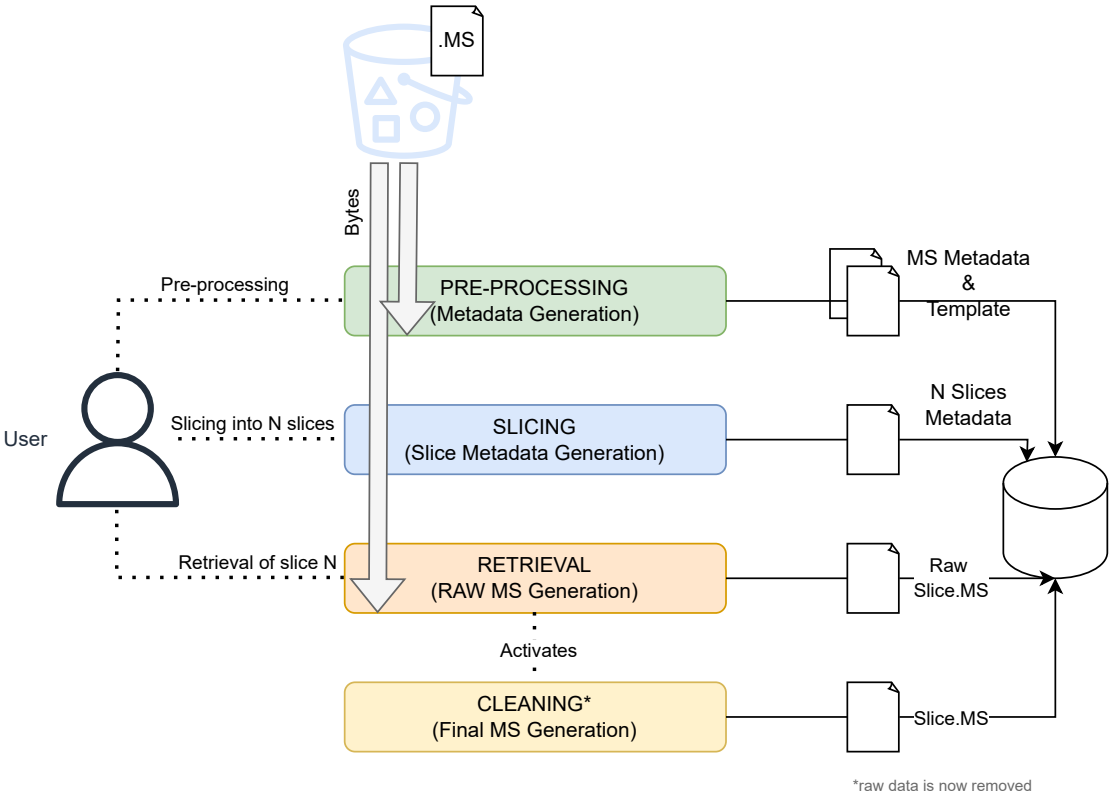

Figure 4.1: Prototype: Architecture overview.

4.1.2. Retrieval from Object Storage and Metadata Generation

This phase is the most file-intensive and acts as the foundation for the rest of the pipeline. The goal is to retrieve the minimal set of files required from an incomplete, yet workable Measurement Set (MS) and to generate accompanying metadata from it that will be later used for the slicing, retrieval, and reconstruction of the file slices.

Libraries and Tools

We rely on the following libraries:

- **boto3(botocore)**: low-level interface for accessing AWS-like object storage.
- **re (Regular Expressions)**: for analyzing the text containing the formats and extracting relevant information.
- **casacore (Python bindings):** for table introspection and metadata analysis of MS files.

File Categorization

We determined that for a viable solution, files in a Measurement Set should be divided into two categories, based on their extensions:

- **Immutable files:** smaller, static files (e.g., metadata headers) that are unlikely to change and can be reused without modification.
- **Mutable files:** larger, dynamic files containing binary data.

```
for obj in s3.list_objects(...):
    key = obj['Key']
    if key.endswith(criterion):
        open(local_file_path, 'wb').close() #Emtpy file
        ...
    else:
        s3.download_file(...) #Retrieve
```

Listing 4.1: Prototype: File classification (mutable vs immutable)

In our first function, we define this classification and retrieve from OS immutable files directly into a working folder—this folder becomes the base for our template MS file, and at the same time, can be further processed for more necessary metadata. Mutable files are recreated empty and locally.

For mutable files, we store only empty files, and we will later retrieve metadata describing their internal structure.

4.1.3. Metadata Extraction and Analysis

Once the template MS is in place, we analyze its contents using the `casacore` Python bindings. The `casacore.tables` module allows us to introspect table structures and extract meaningful metadata.

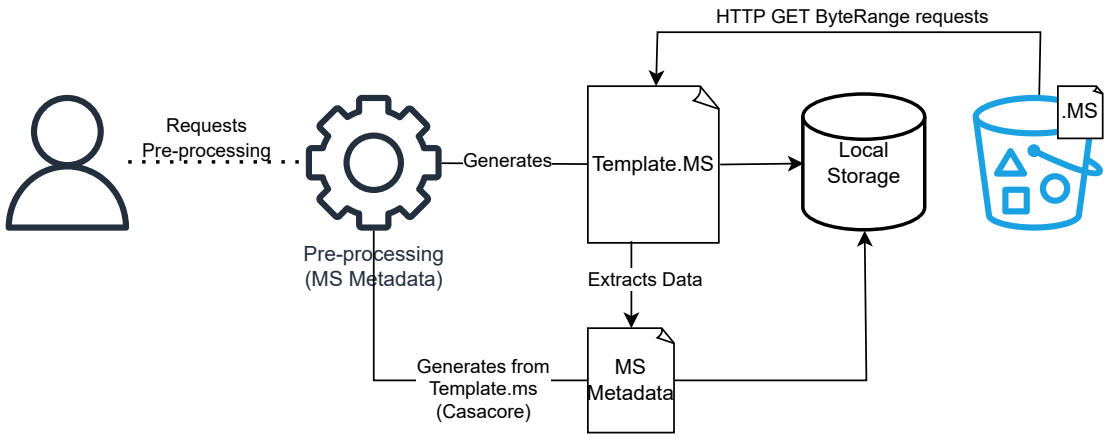

Figure 4.2: Prototype: Retrieval, Extraction and Analysis

We specifically make use of the `table.showstructure()` method, which reveals internal storage details such as:

- Data manager type
- Block size and bucket size
- Column types (binary, integer, etc.)
- Column names
- ...

```
structure = table(ms_path).showstructure()
for block in structure.split("\n\n"):
    if "TiledColumnStMan" in block:
        file = re.search(r"file=(\S+)", block)
        shape = re.search(r"shape=\[(.*?)\]", block)
        bucketsize = re.search(r"bucketsize=(\d+)", block)
        type = ...
```

Listing 4.2: Prototype: Gathering file Metadata)

This process is mostly text-based and uses regular expressions to determine data types and their storage configuration. Unfortunately, there is no native way in `casacore` to programmatically retrieve all types and their actual size, so this portion is currently hard-coded based on prior experimentation and official casa documentation. These types are needed for the exact calculation of internal sizes.

```
type_sizes = {"double": 8,"float": 4,"Complex": 8 ...}
```

Listing 4.3: File classification logic (mutable vs immutable)

Metadata Output

From this analysis, we generate a .JSON metadata file that describes the MS layout. This file includes:

- Column structure
- Expected data types
- Byte ranges for each slice

```
  "ms_name": "partition_1.ms",
  "mutable_files": [
    {
      "file_name": "table.f2_TSM0",
      "bucketsize": 196608,
      "block_size": 24
    },
    {
      "file_name": "table.f3_TSM0",
```

```
    ...
```

Listing 4.4: Prototype: Excerpt of generated MS metadata

The resulting metadata and the immutable MS template can be stored locally or uploaded to object storage, ideally in a separate metadata bucket (e.g., `<ms-name>.metadata/`) to keep concerns separated and facilitate remote execution.

This process is a execute-once function. Meaning that as long as it was correctly pre-processed, now we may slice many times and work with the data without the need to re-process the original file.

4.1.4. Slicing and Reconstruction

Another script in the prototype is responsible for performing data slicing based on the metadata created previously.

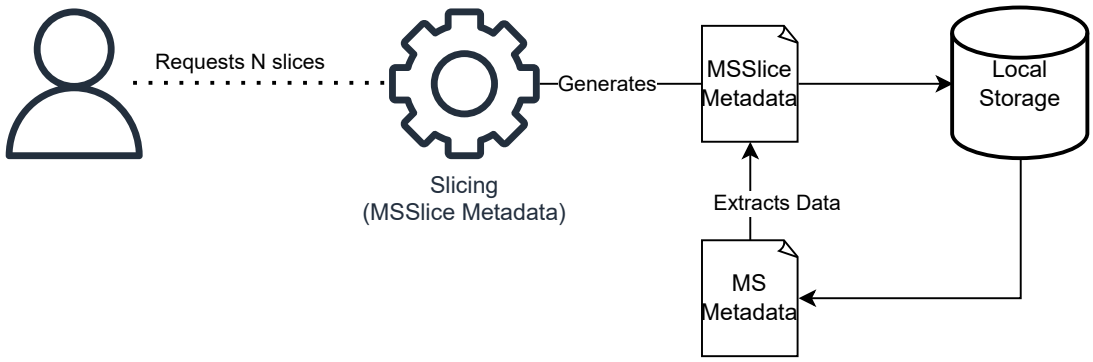

Figure 4.3: Prototype: Slicing

Slicing Strategy

Before slicing, the user defines how many parts the MS should be divided into. Using only the metadata, our system calculates the byte ranges for each slice and the padding required to match the bucket or block sizes.

```
bucket_size = file_info['bucketsize']
datablock_size = file_info['block_size']
...
actual_data_size = current_rows * datablock_size
if actual_data_size % bucket_size != 0:
    num_buckets = (actual_data_size // bucket_size) + 1
    padding = (num_buckets * bucket_size) - actual_data_size
else:
    padding = 0
```

Listing 4.5: Bucket and padding calculation snippet

This generates a metadata file that precisely defines each slice. If needed, the user can now re-slice the Measurement Set without regenerating the template or downloading any additional data—simply by using the existing MS metadata.

Data Retrieval

Each slice is then assembled by:

- Cloning the MS template structure.
- Retrieving byte-range chunks from object storage.
- Injecting data into the appropriate mutable files.

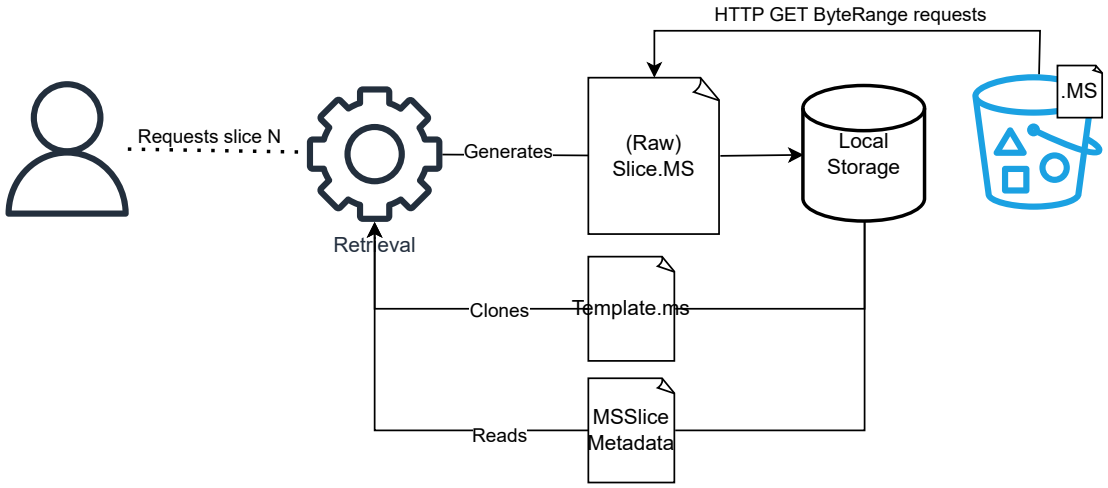

Figure 4.4: Prototype: Data Retrieval

If running remotely, the template and metadata would be fetched from object storage. Locally, we directly use the available copies, so we need to clone it before filling it.

```
def process_slice(bucket, ms_name, slice_number, metadata_file):
    template_path = "templates/template.ms"
    slice_ms_path = f"output/slice_{slice_number}.ms"

    clone_template(template_path, slice_ms_path)

    copy_byte_range_to_output_files(bucket, slice_number,
       ms_name, metadata_file, slice_ms_path)
```

Listing 4.6: Prototype: Creating a slice by cloning template and filling mutable files

Post-Processing

After slice completion, the MS structure is mostly correct—but row indices may still be inconsistent.

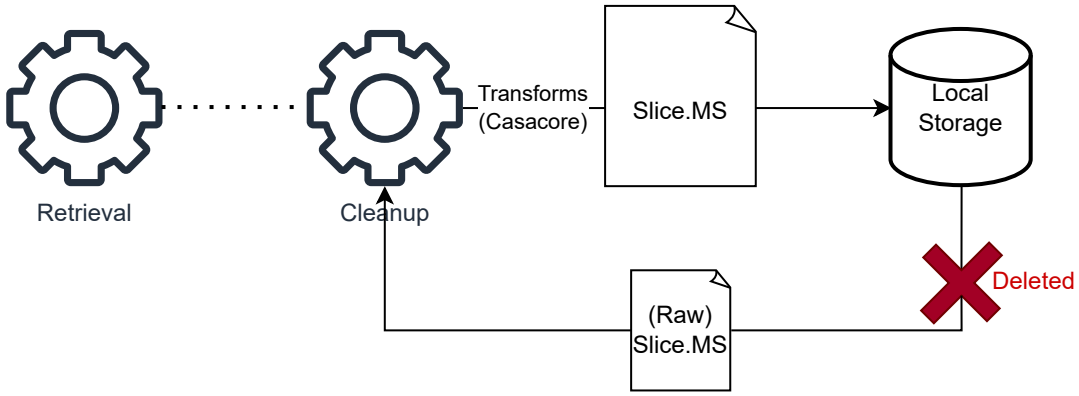

Figure 4.5: Prototype: Cleanup

To fix this, we use `casacore` to clone only the rows relevant to the current slice into a new cleaned MS and then we delete the non-adjusted MS:

```
ms = table(input_ms_path)

selection = ms.selectrows(list(range(num_rows)))
selection.copy(output_ms_path, deep=True)

return f"Measurement Set processed correctly, stored in:
   {output_ms_path}"
```

Listing 4.7: Prototype: Post-processing slice with casacore

4.1.5. Validation

Initial validation using `casatablebrowser` showed promising results—the prototype produced MS files structurally close to the originals. However, deeper tests during Dataplug integration and downstream pipelines like TASKA-C revealed issues related to data consistency and internal validation by `casacore`.

These issues were later addressed during full integration, but the prototype served as a crucial proof of concept. It demonstrated the viability of selective MS reconstruction from metadata and object storage, and provided a solid base for subsequent enhancements in Dataplug.

Deeper validation and testing will be performed on the next chapter.

4.2. Dataplug Plugin: Final Version

In this section, we describe the final implementation of our Dataplug plugin. While will try to avoid reiterating details from the prototype phase, we explicitly highlight the architectural changes, refinements, and additional features that led to a fully functional

integration. The section follows the logical structure of the plugin's execution life cycle, from object ingestion to slicing, retrieval of data and reconstruction.

4.2.1. Modifications to Dataplug Core

Dataplug is designed to support new plug-and-play format plugins without requiring changes to its core. However, our use case introduced a unique challenge: the need to treat entire `Measurement Set (MS)` directories as valid input units within an object storage backend.

In object storage systems (e.g., S3), folders are not native objects but are typically inferred from key prefixes. This complicates their recognition and handling, since they do not follow the classical key-value pattern. Therefore, we introduced some slight modifications to the `CloudObject` class, which serves as the generic interface for all data formats within Dataplug.

Among the standard entry points—`from_s3`, `from_bucket_key`, and `new_from_file`—only `from_s3` was adapted. This is the most common entry point and the one relevant to our cloud-native data processing flow. `from_bucket_key` was deemed inapplicable since folders do not possess unique keys, and `new_from_file`, while potentially adaptable, was deprioritized due to our focus on object-storage-centric workflows. Should a local-to-cloud pipeline be required in the future, adaptation remains feasible. In any case, it is preferable for these files to be stored in OS, so in a real application it would still be preferable to upload them first to OS and then use `from_s3`.

We introduced a new property, `is_folder`, to the `CloudObject`. This boolean is defined at the format level (i.e., `DataFormat.is_folder`) and used during metadata generation to distinguish directory-like formats, where before they needed a header to the file, now it can mark that the file being treated is a directory.We choose this because it was the most transparent solution to the user, since the developers of the plugin are the ones that determine how the `CloudObject` of the format is defined. This mechanism also avoids ambiguity—we believe there are no formats where the input can be both a directory and a file.

```
class CloudObject:
    def __init__(self,
                 ...
                 is_folder: bool = False):
        ...
        self._is_folder = is_folder
        ...
...
    def fetch(self):
        if not self._obj_headers:
            if self._is_folder:
                self._obj_headers = {
                    'Information': 'This is a folder-type format,
                       headers not applicable.'
                }
            else:
```

```
                self._fetch_object()
        ...
```

Listing 4.8: Final Version: Folder-aware CloudObject, in cloudobject.py

4.2.2. Preprocessing Flow and Format Integration

Dataplug abstracts preprocessing through a general wrapper method, `preprocess()`, implemented in `CloudObject`. This method orchestrates metadata bucket creation, lets the user choose one of the available preprocessing configurations, and delegates logic execution to the specific plugin's preprocessing routine.

Format Definition and Preprocessing

The MS format plugin implements a preprocessing function closely aligned with the logic explored during the prototype stage. Some of the differences and upgrades are the following:

- Improved handling of existing metadata—old files are removed to ensure consistency.
- Download logic now consistently segregates mutable and immutable files. Immutable files are left in the template path (`template.ms`), while metadata on mutable files is returned for reconstruction.
- A TAR archive is created from the `template.ms` directory to be embedded in the Dataplug metadata object path, as Dataplug allows any file to be part of the metadata.

```
for column in tiled_metadata:
    column_filename = column["filename"] + criterion
    for empty_file in empty_files_info:
        if os.path.basename(empty_file["name"]) ==
           column_filename:
            new_mutable = {
                "file_name": empty_file["name"],
                "ms_path": f"{ms_name}/",
                "real_size": empty_file["size"],
                "bucketsize": column["bucketsize"],
                "block_size": column["block_size"]
            }
            mutables_list.append(new_mutable)
```

Listing 4.9: Final Version: New File classification logic (mutable vs immutable), in ms.py

After the download phase, we proceed with tiled column analysis. As in the prototype, this involves parsing the structure of the MS to extract storage manager information. A new enhancement is the use of a `get_rows_per_time()` function, introduced to handle previously unseen inconsistencies in row grouping by timestamp—critical for ensuring compatibility with later pipeline stages (e.g., calibration and imaging, where unmatched time ranges resulted in empty rows).

The preprocessing function now also supports additional storage manager types that were not considered during the prototype phase, prompting the inclusion of broader compatibility logic. While the current implementation remains relatively sparse and generic, we have not pursued more in-depth logic at this stage. Tests have shown that most of this data tends to be relatively small, and it does induce an overhead on a later stage but we managed to overcome it. Nonetheless, the existing baseline offers sufficient flexibility to accommodate future optimizations.

```
for block in blocks:
    if "TiledColumnStMan" in block:
        # [ ... existing tiled column handling ... ]

    elif "StandardStMan" in block or "IncrementalStMan" in block:
        column_names = re.findall(
            r"^\s*([A-Z0-9_]+)\s+
            ?:Int|double|float|Complex|Bool)\b",
            block,
            re.MULTILINE
        )
        static_columns.extend(column_names)

    elif "StMan" in block:
        # [ ... same treatment as Standard ... ]
```

Listing 4.10: Final Version: Storage Manager compatibility logic, in ms.py

Metadata Dictionary Creation

During the pre-processing step, a dictionary is constructed to describe all files with mutable columns, and another one describing the immutable columns. This was one of the additions over the prototype that later ensures total file integrity. For each mutable file, we record the following:

- Filename and relative path
- Actual file size
- Bucket size and block size, as specified by its storage manager
- Data shape

In this same step, we also record additional data belonging to the entire MS, such as:

- Total rows of the MS
- Name of the MS
- Rows per time, meaning how many recordings are in a same time-frame.

```
tiled_metadata, total_rows, rows_per_time, static_columns =
   _analyze_tiled_columns(...)

mutables_list = []
```

```
for column in tiled_metadata:
    # [... fill in mutables list...]

return PreprocessingMetadata(
    attributes={
        "total_rows": total_rows,
        "mutable_files": mutables_list,
        "ms_name": ms_name,
        "rows_per_time": rows_per_time,
        "static_columns": static_columns
    },
    metadata_file_path=metadata_path + ".tar"
)
```

Listing 4.11: Final Version: Metadata that Dataplug will upload to OS, in ms.py

As with other scientific plugins, Dataplug handles the uploading of this metadata to OS, streamlining integration.

4.2.3. Format Decoration and Folder Recognition

The plugin is annotated using the class `@cloud_dataformat` decorator, where we specify `is_folder=True`. This ensures that the earlier-discussed logic within `CloudObject` correctly interprets the MS directory structure and handles dummy header metadata generation accordingly. We also put what pre-processing function will raw data take and we define here the name of the format that this plugin processes.

```
@CloudDataFormat(preprocessing_function=preprocess_ms,
   is_folder=True)
class MS:
    ...
```

Listing 4.12: Final Version: Definition for MS format, in ms.py

4.2.4. Partitioning Strategy

Partitioning logic is centralized within a dedicated strategy class, which defines the manner in which the MeasurementSet (MS) is sliced. Although multiple strategies were initially explored and are easily integrated into Dataplug, only one was ultimately retained, as it demonstrated the highest effectiveness in real-world scenarios. This strategy is designed to partition data based on groups of rows per timestamp, thus preserving temporal coherence throughout the dataset. The user specifies the desired number of slices, and the strategy attempts to distribute the data evenly by allocating an approximately equal number of bytes to each slice. This approach is primarily oriented around data size, rather than employing more complex partitioning schemes based on timestamp frames, fixed numbers of rows per file, or other structural metrics.

```
@PartitioningStrategy(dataformat=MS)
def ms_partitioning_strategy(cloud_object: CloudObject,
   num_chunks: int):
    ...
    for index in range(num_chunks):
        timestamps = base_timestamps + 1 if extra > 0 else
           base_timestamps
        if extra > 0:
            extra -= 1

        end = start + timestamps * rows_per_timestamp - 1
        slices.append(MSSLice(start, end, index=index))
        start = end + 1

    return slices
```

Listing 4.13: Final Version: Partitioning strategy, in ms.py

This approach dynamically returns `MSlice` objects, each representing a slice with the following attributes:

- Byte start and end offsets
- Index to identify slice position

The maximum number of slices is not predefined—it is dynamically determined during execution based on the size and structure of the input. Although this number could technically be computed in advance from the available metadata and stored as an additional key, we opted to keep this behavior opaque to the end user. The system will generate as many slices as possible when a user requests a high number; however, this may lead to inconsistencies when integrating with other programs, as they might expect more slices than can realistically be produced.

We consider this to be an acceptable trade-off, as testing with a relatively small MeasurementSet file (120MB) demonstrated that the system could generate up to 68 slices. This scaling behavior holds with larger inputs—for example, a 8,000MB (8GB) file allowed the creation of more than 6,500 slices. As a result, we believe that downstream programs processing such data are unlikely to request or benefit from dividing the input into such a high number of partitions. At this level of granularity, individual slice sizes fall in the 5–10MB range, which introduces significant overhead and leads to diminishing returns in terms of parallelization. The increased complexity of managing such small partitions—particularly with regard to I/O performance and coordination—often outweighs the performance gains from additional concurrency.

4.2.5. MSlice Execution and Reconstruction

Each `MSlice` object is executable independently. The slice begins by checking for a locally cached `template.ms`; if not found, it is downloaded and extracted from the metadata archive. This step is lightweight but introduces minor latency, particularly when executed on remote workers in distributed setups, where retrieval of metadata is mandatory. In

monolithic or local contexts, this extraction step may be skipped if the file is already available.

```
class MSSLice(CloudObjectSlice):
    ...
    def get(self):
        ms_name = self.cloud_object.path.key
        ...

        if not os.path.exists(metadata_dir):
            ...
            self.cloud_object.storage.download_file(
                meta_bucket, meta_key, template_path + ".tar"
            )
        ...
        _clone_template(...)
        _copy_byte_range(...)
        _cleanup_ms(...)
        ...
        return cleaned_sliced_path
```

Listing 4.14: Final Version: MSlice object, in ms.py

Once the template is confirmed to be present, we proceed to clone it. Although this may appear redundant, it is necessary in scenarios where a single worker may be able to processes multiple slices in sequence. Without cloning, one slice could overwrite or corrupt the shared template, introducing race conditions or data loss. By duplicating the template into a dedicated working directory, we ensure isolation and repeatability.

```
def _clone_template(template_path, output_path):
    if not os.path.exists(template_path):
        raise FileNotFoundError(f"[DATAPLUG] The template
           {template_path} does not exist.")
    ...
    print(f"[DATAPLUG] Cloned {template_path} to {output_path}")
```

Listing 4.15: Final Version: Cloning, in ms.py

After preparation, we invoke the `copy byte range` function. This is the core data movement routine, responsible for performing range-based retrieval from object storage. For each mutable object defined in the slice metadata, the function calculates the correct byte offsets, applies any necessary padding, and injects the raw content into the corresponding empty file within the cloned MS.

```
def _copy_byte_range(s3, bucket, ms_name, metadata, output_path,
   starting_row, end_row):
    for mutable in metadata:
        ...
        s3_range = f"bytes={start_byte}-{end_byte}"

        try:
```

```
            response = s3.get_object(Bucket=bucket,
               Key=f"{ms_name}/{file_name}", Range=s3_range)
            file_data = response["Body"].read()
        ...
        padded_len = ((len(file_data) + bucketsize - 1) //
           bucketsize) * bucketsize
        ...
        with open(target_file, "wb") as fout:
            fout.write(file_data + b"\x00" * (padded_len -
               len(file_data)))
```

Listing 4.16: Final Version: Copy Byte Range, in ms.py

This approach ensures that each slice reconstructs its portion of the MeasurementSet independently, without requiring knowledge of other slices. The reconstructed MS at this point is complete in structure but will still require post-processing to fully conform to expected formats, particularly regarding row consistency in some columns.

4.2.6. Final Cleanup with Casacore

After the initial reconstruction, the resulting MS file is technically usable but will contain some inconsistencies that, for some applications, may turn them unusable. These issues are addressed in the final cleanup phase. In a future version, this step could potentially be further optimized through reverse engineering, enabling more fine-grained, byte-level manipulation of some of the more complex Storage Managers. However, since in the files we have found and studied, these structures typically contain only a small percentage of the overall data—and given the significant effort already invested in reaching the current level of functionality and file size reduction—we consider this direction out of scope for now.

For now, we rely on the `casacore` suite to finalize the MeasurementSet. Casacore tools provide slower, yet robust utilities to interact with and modify MS files, particularly for operations that require awareness of the internal schema and the virtual tables that casacore generates on opening the files.

The cleanup routine uses Casacore to:

- Move static rows (from the ones defined as immutable columns) to the beginning of the file structure
- Correct the number of rows in the MS to reflect only valid, reconstructed data by trimming the present, but unpopulated rows.

```
def _cleanup_ms(input_ms_path, output_ms_path, num_rows,
   starting_range, static_columns=None):
    try:
        ms = table(input_ms_path, readonly=False)

        if starting_range > 0 and static_columns is not None:
            for colname in static_columns:
                ...
```

```
                desc = ms.getcoldesc(colname)
                ndim = desc.get('ndim', 0)

                if ndim == 0:
                    sliced = ms.getcol(colname,
                       startrow=starting_range, nrow=fixed_rows)
                    ms.putcol(colname, sliced, startrow=0)
                    continue
            ...
        ...
        return f"[DATAPLUG] Measurement Set processed correctly,
           stored in: {output_ms_path}"
    except Exception as e:
        return f"[DATAPLUG] Error MS: {str(e)}"
```

Listing 4.17: Final Version: New Cleanup, in ms.py

This stage was the final missing piece in transitioning from prototype to production-ready functionality. Without it, some MS files would pass early validation steps but fail at later pipeline stages such as calibration or imaging due to subtle mismatches in shape or metadata.

However, this stage also introduces the most significant overhead. Casacore's performance, especially during column-level operations, tends to degrade as the total number of rows increases. Casacore exhibits suboptimal I/O behavior for most of the Storage Managers, leading to long execution times when modifying large datasets.

To mitigate this, we hardcoded a limited set of critical columns to include in the cleanup step. These were empirically determined during testing as essential for preventing pipeline failure, versus the others that seemed to not affect end results. Althought this limits the generality of the plugin and may exclude edge cases that require broader column inclusion, the trade-off was necessary to ensure correctness under typical workloads. The overhead scales roughly linearly with the number of rows, so more parallel solutions will work better than those with small levels of parallelization.

```
for colname in static_columns:
                if colname not in ['FLAG_ROW', 'TIME',
                   'TIME_CENTROID']:
                    continue
```

Listing 4.18: Final Version: Small compromise, in ms.py

4.2.7. Conclusion

Upon a successful cleanup, the slice returns the path to a fully reconstructed and valid MeasurementSet. This output is now ready to be ingested by downstream components or stored for later use. In the TASKA-C pipeline, this step marks the end of the first part of the rebinning stage, after which calibration will happen and finally imaging.

5 Evaluation

This chapter describes the evaluation process of the developed system. The evaluation is divided into three main parts: tests for **data consistency and integrity**,tests for **Dataplug performance versus previous solutions** and**pipeline performance** when integrating Dataplug in a distributed environment.

For these tests, we primarily work with two datasets that represent the extremes in size and workload:

- **smallms.ms:** a subset extracted from a larger, complete MeasurementSet, previously generated by the static partitioning method used in the earlier version of the pipeline.
- **hugems.ms:** a complete MeasurementSet obtained directly from the `NenuFAR` telescope, part of the original dataset from the `extract` project. One file was selected at random, since all files follow a similar structure and format for this telescope.

Both datasets represent real use cases and serve to test the functional correctness as well as the scalability of the system under contrasting conditions.

5.1. Consistency and Integrity Tests

Before evaluating performance, it is essential to verify that data reconstruction is accurate. To this end, we conduct a series of tests designed to demonstrate that files can be:

- Partitioned into fragments by Dataplug
- Independently reconstructed from those fragments
- Compared against the original files without loss of relevant information

These tests are performed at three levels:

- **Visual:** The resulting file is inspected using standard visualization tools (`casatablebrowser`) to confirm that the MeasurementSet structure(columns) is preserved and the data(rows) is readable.
- **Via Casacore:** We compare columns directly as we iterate both MS, but this won't work for all columns.
- **Byte-level:** We perform direct comparisons between the original and reconstructed files using scripts that do byte level verification, since some metadata or auxiliary

structures may have minor differences (e.g., modification dates or internal paths), we choose to ignore those irrelevant differences.

Below is an example snippet showing how we reconstruct the MeasurementSet files with casacore and our byte-level comparison:

```
def concat_and_copy(ms_list, final_output_ms,
   concat_in_time=True):
    ...
    msconcat(ms_list, temp_virtual_ms, concatTime=concat_in_time)
    ...
    t = table(temp_virtual_ms, readonly=False)
    t.copy(final_output_ms, deep=True)
    t.close()
...
ms_list = [
    "slice_0.ms",
    "slice_1.ms"
]

final_output_ms = "hugems_concatenado.ms"
concat_and_copy(ms_list, final_output_ms)
```

Listing 5.1: Reconstructing an MS from slices

How we compare those columns using Casacore:

```
for colname in columns_to_compare:
        if colname in t1.colnames() and colname in t2.colnames():
            log_lines.append(f"Comparing column: {colname}")
            try:
                values1 = t1.getcol(colname)
                values2 = t2.getcol(colname)

                if values1.shape != values2.shape:
                    ...
                else:
                    diff_count = 0
                    for row in range(values1.shape[0]):
                        value1 = values1[row] if values1.ndim ==
                           1 else values1[row]
                        value2 = values2[row] if values2.ndim ==
                           1 else values2[row]
                        if value1 != value2:
                            log_lines.append(...)
                            diff_count += 1
```

Listing 5.2: Comparing with Casacore

Lastly, at the byte level, we first perform a fast hashing-based comparison using the MD5 algorithm to determine if any difference exists between the original and reconstructed files. If a mismatch is detected, we proceed with a deeper byte-by-byte analysis to identify the

exact positions and values of the differing bytes:

```
def compare_files_byte_by_byte(file1, file2):
    differences = []
    total_differences = 0

    with open(file1, "rb") as f1, open(file2, "rb") as f2:
        pos = 0
        while True:
            b1 = f1.read(1)
            b2 = f2.read(1)
            if not b1 and not b2:
                break
            if b1 != b2:
                differences.append((pos, b1.hex(), b2.hex()))
                total_differences += 1
            pos += 1

    return total_differences, differences
```

Listing 5.3: Comparing with Casacore

This set of tests validates that the partitioning and processing process preserves data fidelity and results in a functional MeasurementSet usable for downstream processing.

5.1.1. Results

We confirm that the files match in two ways. First, through visual inspection (which is omitted from this document, as it is difficult to convey results from `casatablebrowser`, and more precise methods are available). Second, through a Casacore-based comparison of table columns:

Column Comparison Report

Comparison of columns between:
MS1: `hugems.ms/`
MS2: `hugems_concatenated.ms/`

Comparing column: ARRAY_ID
No differences found.
Comparing column: DATA_DESC_ID
No differences found.
Comparing column: EXPOSURE
No differences found.
Comparing column: FEED1
No differences found.
Comparing column: FEED2
No differences found.
Comparing column: FIELD_ID
No differences found.
Comparing column: FLAG_ROW
No differences found.
...

Thirdly and finally, per-byte comparison:

Folder Comparison Summary

Comparing folders:
`/hugems.ms`
`/hugems_concatenated.ms`

DIFFERENT FILE: `table.info`
Difference found between the folders.

As we can see, only a single file differs: **table.info**. This is expected, as this file contains metadata about how the MeasurementSet was constructed. In our case, it now reflects that the data was reassembled from two slices. This difference is actually a positive indication—it confirms that the file was processed and successfully reconstructed. However, if required, this metadata can be overridden to match the original exactly, ensuring even this file remains identical.

5.2. Efficiency Gains from Dataplug MeasurementSet Partitioning

The first performance evaluation that we want to present, focuses on comparing the efficiency of the traditional static pre-processing and partitioning approach with the new dynamic method integrated with Dataplug. To ensure a fair comparison, we isolated these pre-processing steps and executed them independently of the rest of the pipeline.

This isolation helps eliminate external influences such as network latency, data transfer overhead, or remote execution variability.

All these following tests were carried out locally to further minimize external factors and provide a clean benchmark of raw partitioning performance. For this evaluation, we used the hugems.ms, selected for its large size and relevance as a realistic stress test for partitioning logic.

We tested four scenarios: splitting the dataset into 2, 4, 8 and 16 partitions. These configurations allow us to observe the performance impact of scaling the partitioning mechanism. This benchmark provides valuable insight into the efficiency gains of the dynamic approach.

Later sections will assess in more depth the impact of this improvement when fully integrated into the TASKA-C pipeline.

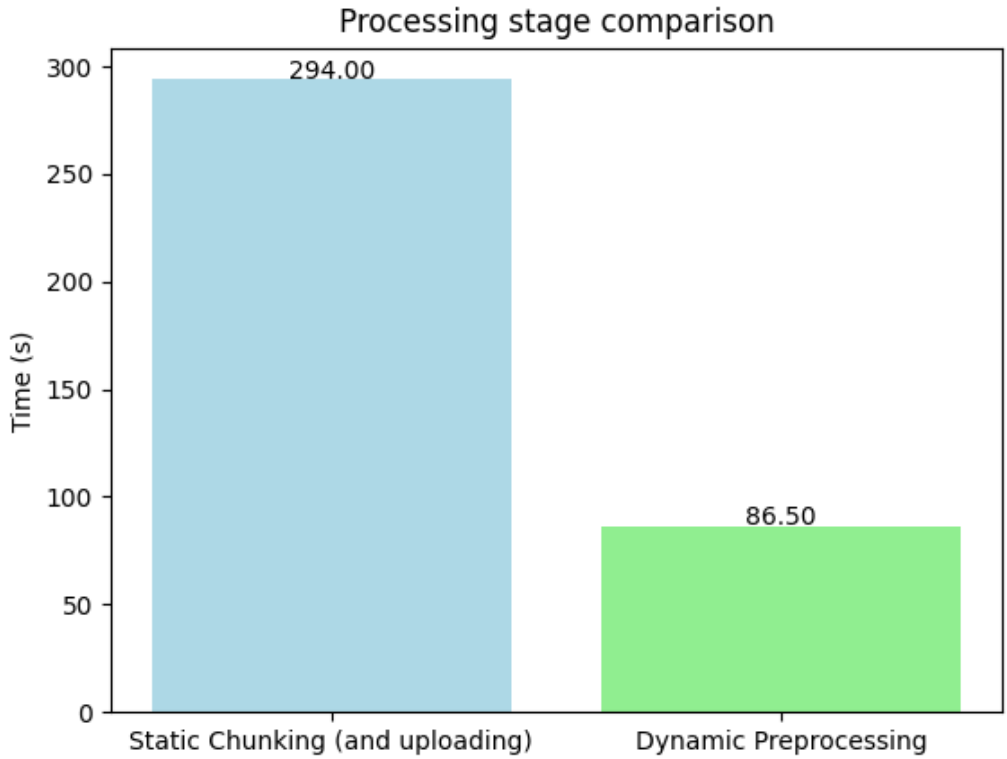

Figure 5.1: Comparison: Static Chunking vs. Dynamic Pre-processing hugems.ms

We observe the average processing time across all four dataset partitions. We can see a major improvement in the data preparation stage: total time from Static to Dynamic (Dataplug) drops from 294s to 86.5s — a **70.6%** reduction.

Although this isn't a strictly fair comparison—since some of the processing work is deferred to the execution phase—it still highlights the significant efficiency gains enabled by Dataplug.

It is also worth mentioning that only Dynamic Dataplug Preprocessing maintains linear

execution time. Static Chunking worsens as the number of files increases. Not only that, but we are in perfect data transfer conditions (local), so Static benefits even more than Dynamic from our enviroment.

A more realistic look at a real execution would be comparing how long it takes for data that is stored and unprocessed, to arrive as a functional slice to a single worker:

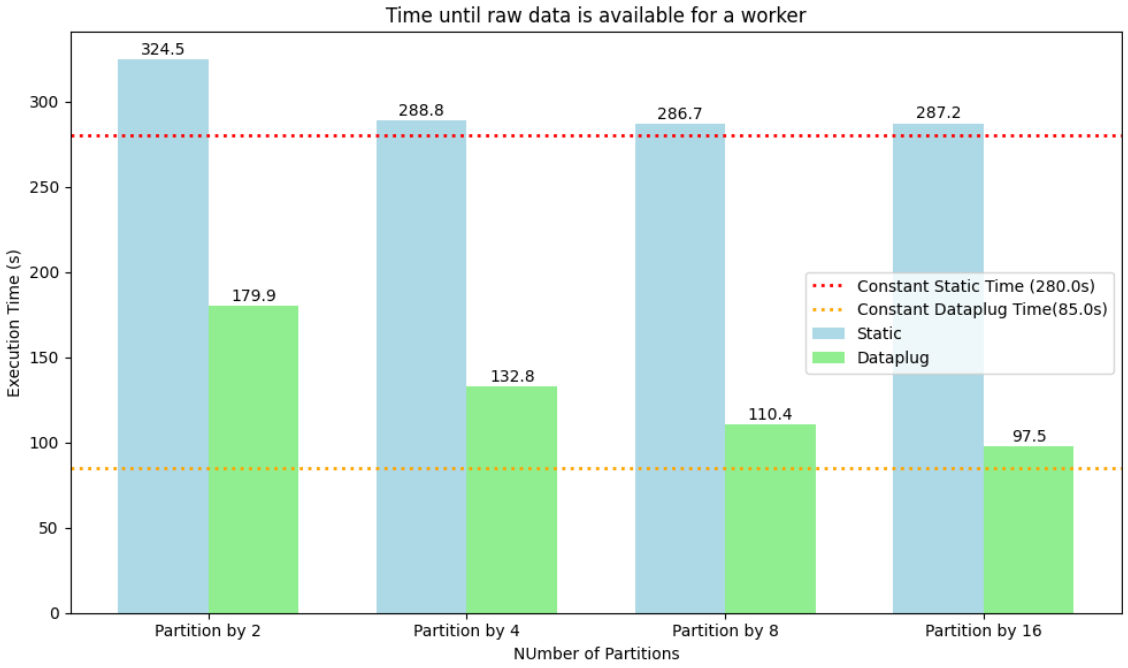

Figure 5.2: Comparison: Getting functional data to a worker

Even with trivially large partitions, such as the case of 2, a single worker using Dataplug has complete access to the data in 179.9s, compared to 324.5s with static chunking — a **44.6%** improvement. When going up to 16 partitions, the time drops to 97.5s vs. 287.2s, which is a **66.1%** decrease in overall time. This shows that even in worst-case, where casacore I/O bottleneck is at its worst, Dataplug significantly outperforms static partitioning by splitting the workload and leveraging optimized pre-processing and cleaning steps.

Finally, we highlight one of the most compelling scenarios that demonstrates the value of our solution: Dataplug's data fetching without the need for pre-processing.

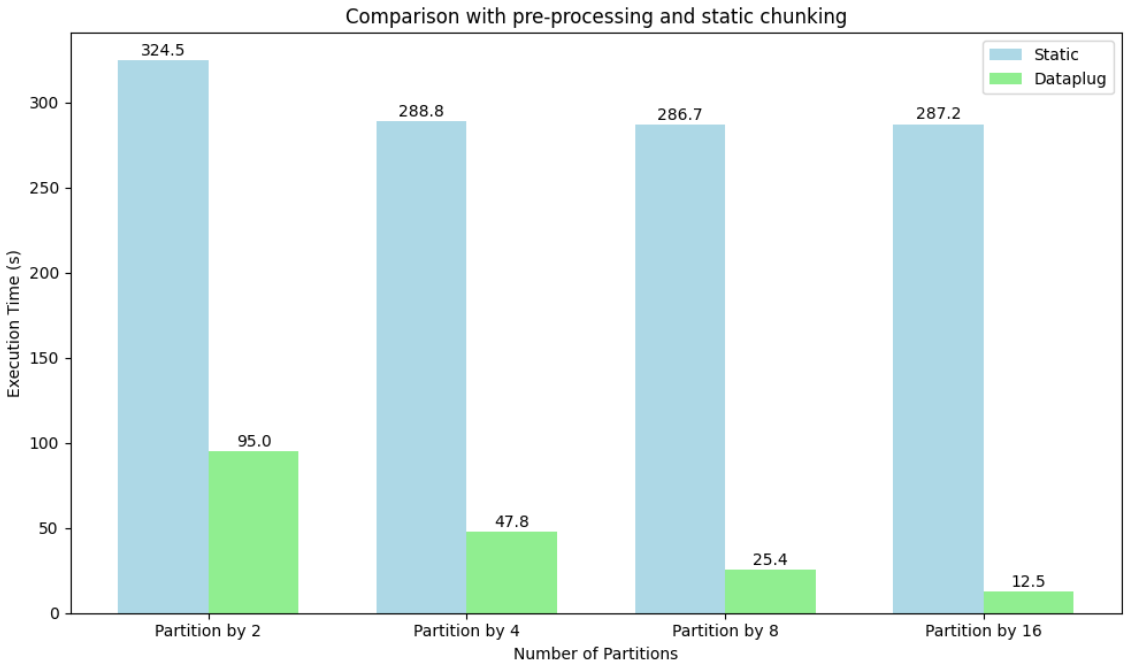

Figure 5.3: Comparison (Best Case): Static Chunking vs. Dynamic partitioning on hugems.ms

In cases where preprocessing is unnecessary—such as when partition metadata is already available from a prior run or generated during storage ingestion—the dynamic approach yields **dramatic time savings**. For instance, with just 2 partitions, total time untill data is available drops from 324.5s to 95s. With 16 partitions, the improvement is even more striking: from 287.2s to 12.5s, a **95.6%** speedup.

These results strongly reinforce the efficiency of dynamic partitioning—not only in raw execution time, but also in flexibility and reduced redundancy. When factoring in static partitioning overheads like full file downloads and higher per-worker memory usage, the advantages of our approach become even clearer

5.3. Astronomical Pipeline Performance Evaluation

This section would not have been practically feasible without the integration of Dataplug. While it is technically possible to execute the pipeline using only statically pre-chunked data, doing so would introduce significant pre-processing overhead—previously estimated at around 300 seconds per run. When multiplied across the test (over 60), this would have made the evaluation prohibitively time-consuming. In contrast, Dataplug enables on-the-fly dynamic chunking, delivering at least a **44.6%** improvement and achieving up to **95%** faster execution under comparable conditions—with potentially even greater gains at larger scales. This efficiency is one of the core motivations behind the project and highlights Dataplug's critical role in enabling scalable experimentation and flexible workflow design.

5.3.1. Methodology

This section outlines how Dataplug was integrated into the astronomy pipeline, the experimental setup, and the performance metrics collected. In addition, it discusses the rationale behind the chosen environment and addresses potential limitations.

Pipeline Integration. Dataplug had already been planned as a modular component of the TASKA-C pipeline to support dynamic preprocessing of MeasurementSets. Although it was not yet implemented during early development phases, the pipeline had been designed to recognize when to trigger dynamic partitioning, enabling seamless integration once Dataplug became available. Only minor modifications were required to adapt to the dynamic flow—primarily adjusting how the pipeline handled input files, which previously expected pre-zipped static partitions. Once integrated, Dataplug functioned as intended, requiring no additional changes for compatibility.

Experimental Setup. We tested the system using both small and large MeasurementSet files to evaluate performance under realistic workloads. The experiments simulated resource-constrained environments, with each worker limited to a single vCPU and 1GB of memory (some exceptions discussed later). This setup reflects common limitations in cloud-native or serverless deployments and is designed to stress-test the system's ability to handle large datasets efficiently.

Execution Environment. Although `Lithops` supports multiple compute backends, including local, AWS Lambda, IBM Cloud Functions, and Kubernetes,we selected Kubernetes for this evaluation. This choice was motivated by the availability of a dedicated Kubernetes rack within our research team's infrastructure, offering a controlled and consistent environment with sufficient computational resources. While this setup limits network bandwidth to the capabilities of the local rack, it remains suitable for assessing system performance in a distributed setting. We also acknowledge certain limitations inherent to Kubernetes-based deployments, such as potential cold-start latencies and trouble with large amounts of pods being deployed.

Metrics Collected. The following performance metrics were collected during our tests:

- **Data ingestion time:** Time taken for workers to fetch data.
- **Execution time:** Total time from trigger to completion of the processing phase.
- **Cold start time:** Time spent initializing containers and runtime environments. This data is later averaged out.
- **Upload time:** Duration required to store the results back to object storage.

To ensure reliability and account for performance variability—such as that introduced by Kubernetes scheduling or transient network conditions—each experiment was executed four times. Results were then averaged to provide a more stable and representative performance assessment.

For the `smallms` file, we evaluated partitioning into 2, 4, 8, 16, 32, 64, and 68 slices—the latter being the maximum number of meaningful partitions for this dataset.

For the `hugems` file, we tested partition counts of 4, 8, 16, 32, 64, and 100, which corresponds to the maximum number of workers supported by our current configuration. Although the theoretical maximum number of partitions for `hugems` exceeds 6,500, exploring such granularity was deemed unnecessary for this evaluation.

5.3.2. Results

Before presenting the results, we first address the rationale behind averaging and the decision to execute each test four times—particularly relevant for the `smallms` experiments.

Figure 5.4: Execution time variability across four runs for `smallms`.

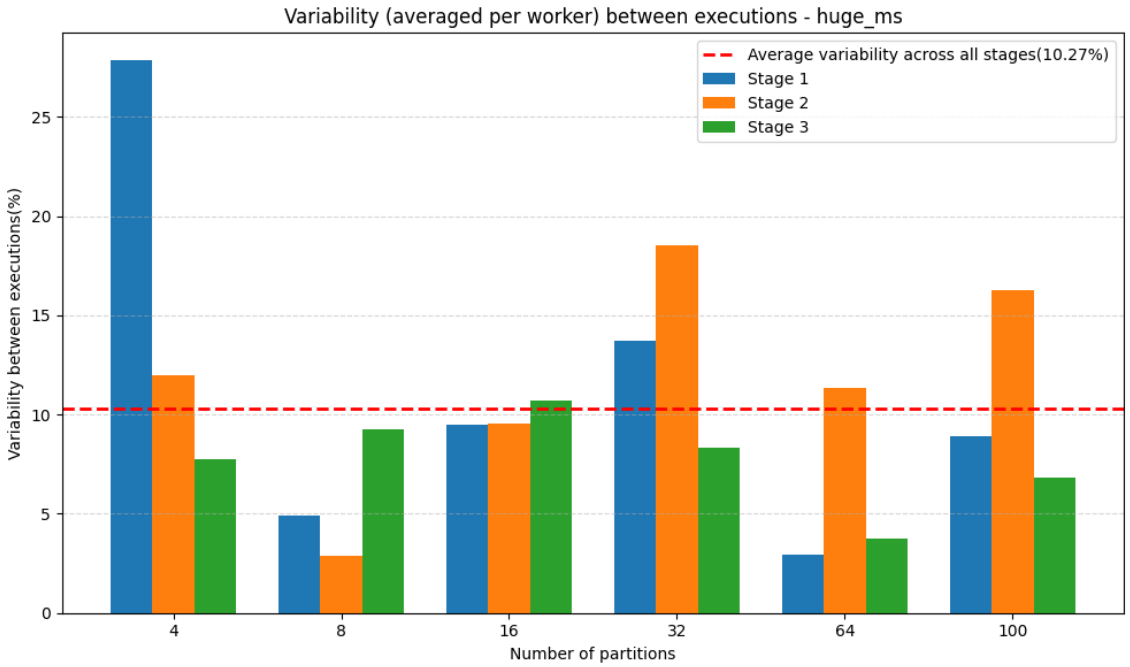

Figure 5.5: Execution time variability across four runs for `hugems`.

As illustrated in the figures above, the variability in execution times for `smallms` remains within an acceptable margin. However, for `hugems`, we observe greater variability—this is expected, as larger datasets demand more data transfer and place greater strain on the orchestration layer (in our case, Kubernetes).

Despite this, we determined that four executions per configuration provide a good balance between accuracy and practicality. Variability tends to remain within a 10% margin on average, which we consider acceptable for evaluating overall performance trends.

Evaluation of smallms

We begin evaluating each stage of the pipeline individually, starting with the *rebinning* stage. This is where Dataplug's dynamic partitioning logic is most involved, handling both data retrieval and the associated cleanup operations.

Rebinning stage:

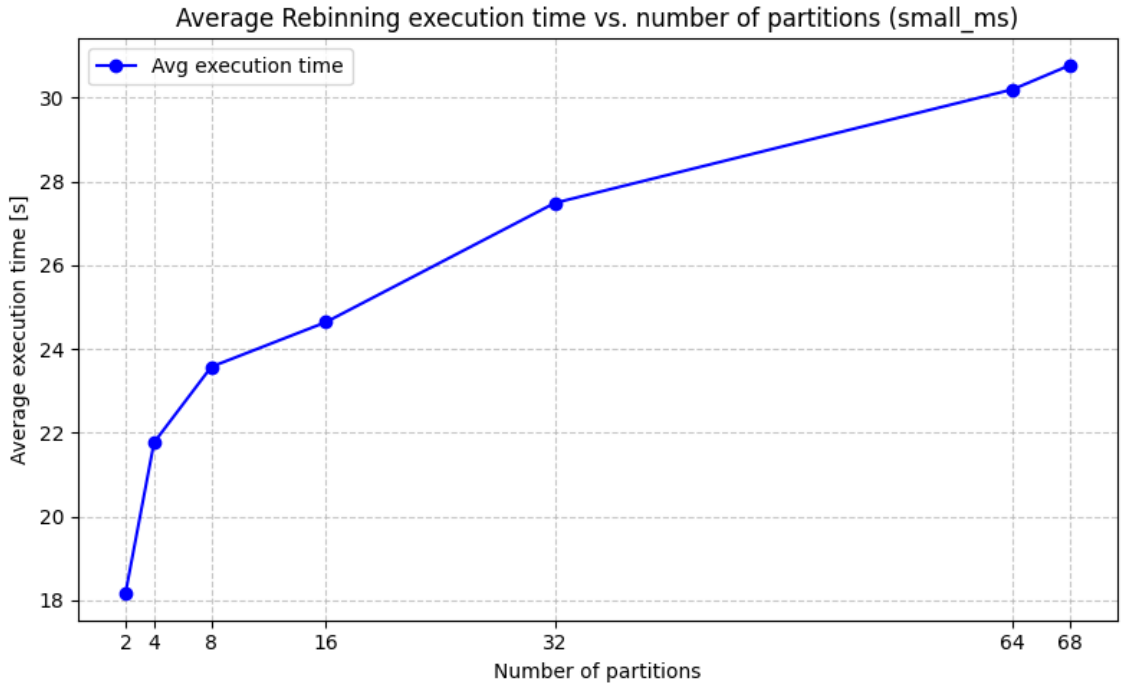

Figure 5.6: Average rebinning execution time vs. number of partitions (`smallms`).

As shown in Figure 5.6, increasing the number of partitions on a small MeasurementSet introduces overhead rather than benefits. This is expected, as certain operations (such as orchestration and reconstruction) do not scale linearly with data volume. For smaller datasets, the overhead of managing many partitions can outweigh the advantages of parallel execution.

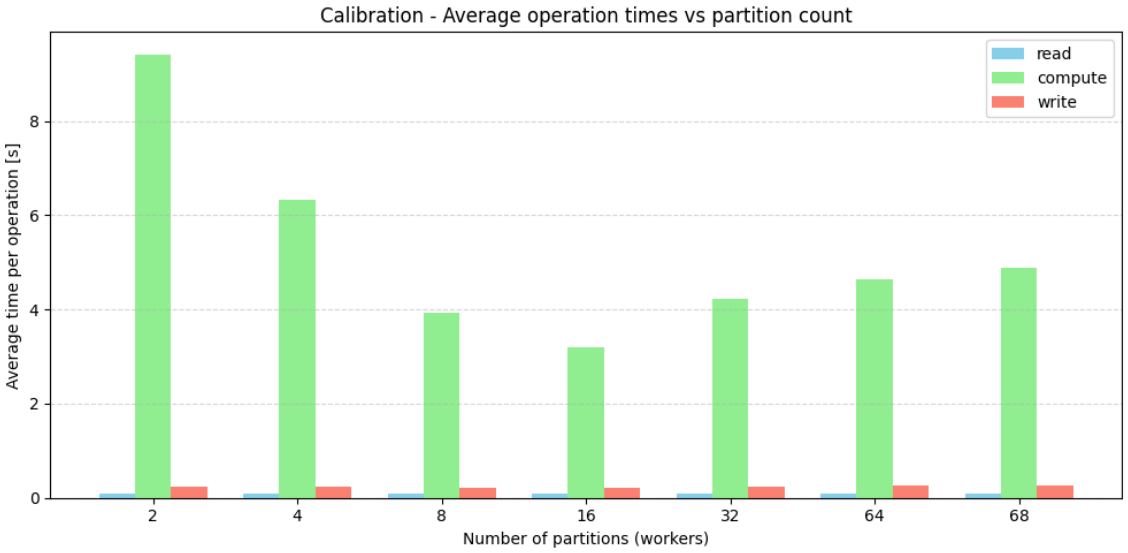

Figure 5.7: Breakdown of read, compute, and write times during rebinning.

In Figure 5.7, we compare the read, compute, and write times across partition counts. We observe that read time increases notably with more partitions, as reading many small

files results in inefficient data transfer. On the other hand, compute time shows modest gains from parallelization, peaking around 16 partitions. Write time remains relatively constant regardless of partition size, indicating that writing is not a bottleneck in this phase.

Calibration stage: Next, we evaluate *calibration*. This stage is highly dependent on the input data , making it theoretically well-suited for parallelism.

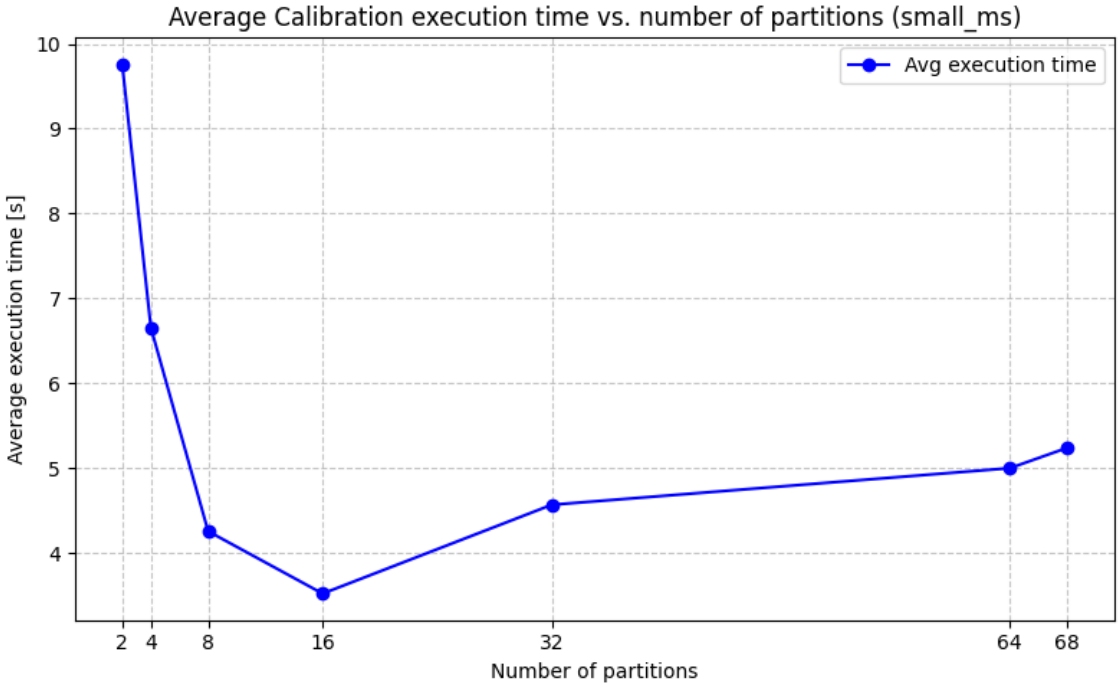

Figure 5.8: Average calibration time vs. number of partitions.

As shown in Figure 5.8, we observe a significant performance gain up to 16 partitions. Beyond that, overhead begins to outweigh the benefits, with 32 partitions performing worse than 16.

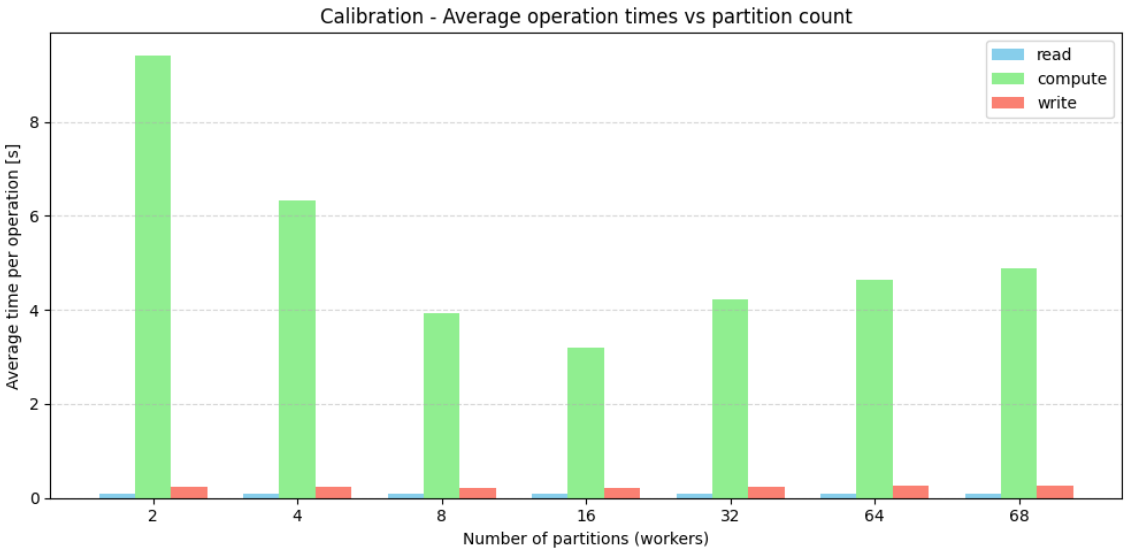

Figure 5.9: Breakdown of read, compute and write times during calibration.

Figure 5.9 confirms that both compute and write times degrade after 16 partitions, indicating diminishing returns from further parallelization.

Imaging stage: Finally, *imaging* is a monolithic step executed by a single worker.

Figure 5.10: Average imaging time vs. number of partitions.

Figure 5.10 shows a slight improvement at 4 partitions, likely due internal structure of the MS.

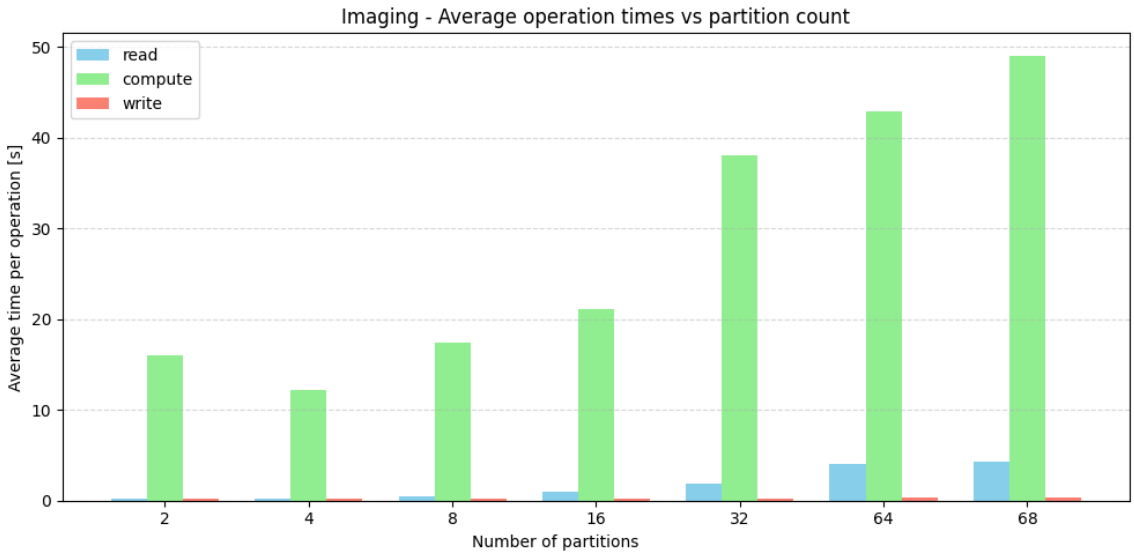

Figure 5.11: Breakdown of read, compute and write times during calibration

As seen in Figure 5.11, read and compute times worsen with more partitions—Casacore's I/O performs poorly with excessive virtual table movement.

Overall evaluation: We originally devised using `smallms` as a baseline to validate our setup and illustrate how parallelism becomes more beneficial as data size increases.

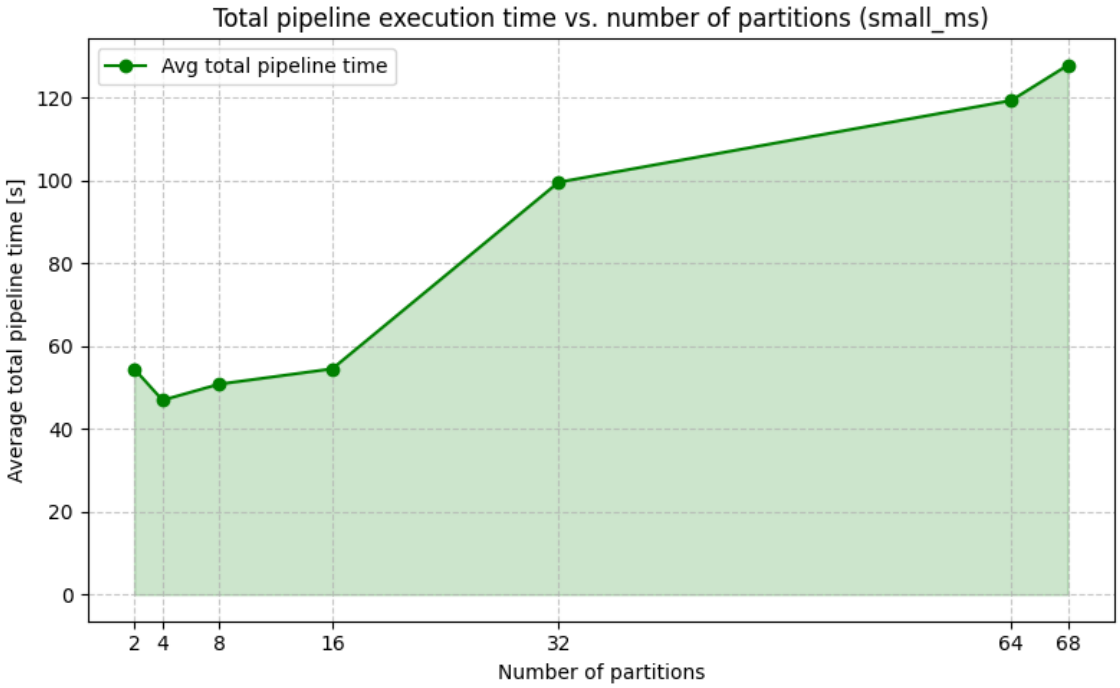

Figure 5.12: Overall execution time over number of partitions (`smallms`).

Although `smallms` is unrealistically small for production, we observe gains even at this scale—especially at 4 workers. However, beyond that point, overheads dominate, confirming the expected trade-off in parallelism.

Evaluation of hugems

We continue the evaluation using the same methodology, now applied to `hugems`.

Before proceeding, it is important to explain why we did not test with just 2 partitions. At this scale, 2 partitions always led to execution timeouts (set at 10 minutes) and would require increasing each worker's memory to 4GB, which we wanted to avoid. Instead, we start from 4 partitions—where we allow a 2GB memory configuration per worker—as a reasonable compromise.

Rebinning stage:

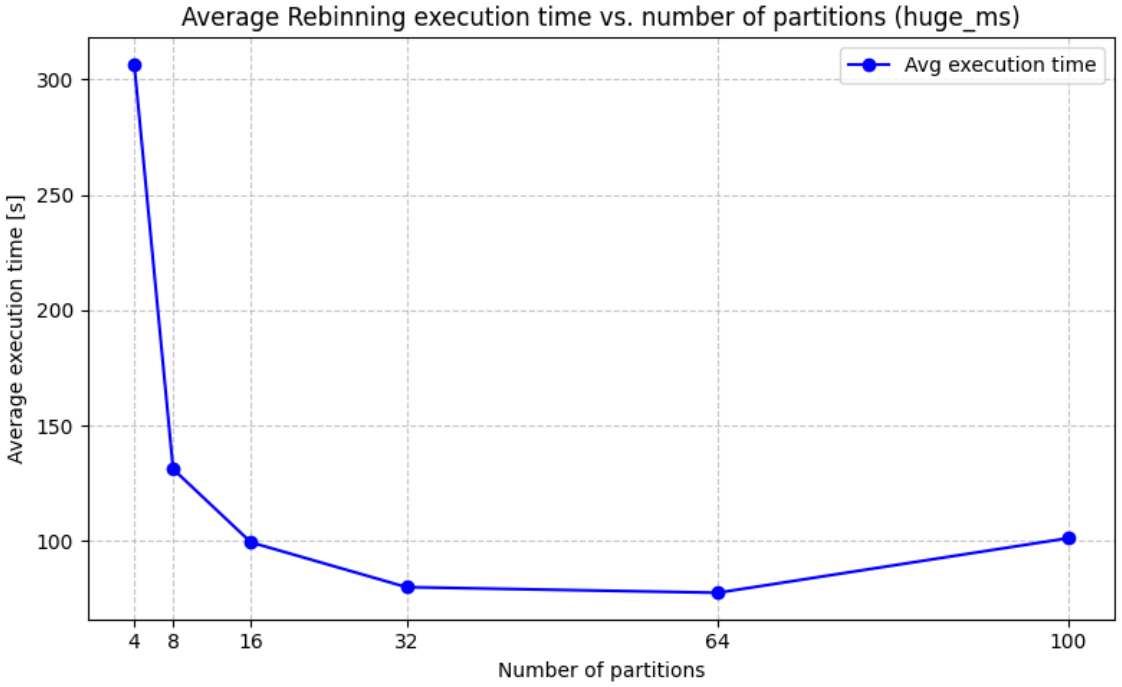

Figure 5.13: Average Rebinning execution time vs. number of partitions (`hugems`).

Figure 5.13 shows significant improvements from parallelization, peaking near 64 partitions—demonstrating the value of our approach.

Figure 5.14: Breakdown of read, compute, and write times during rebinning.

We can observe that compute time benefits most from parallelism. Read time also improves but starts showing overhead earlier (before 64 partitions), and beyond 64, both read and compute overheads increase, reducing the overall efficiency at 100 partitions.

Calibration stage: As seen previously with `smallms`, calibration should benefit the most from parallelism.

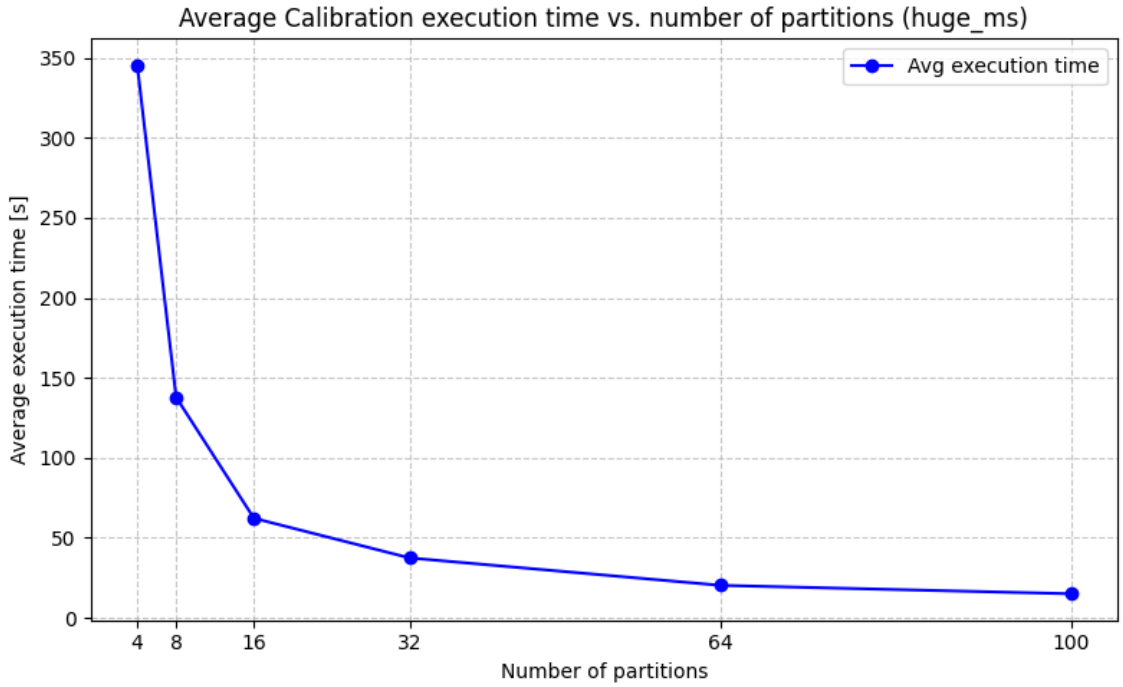

Figure 5.15: Average Calibration execution time vs. number of partitions (`hugems`).

We see consistent improvement even at 100 partitions, although the gains decrease.

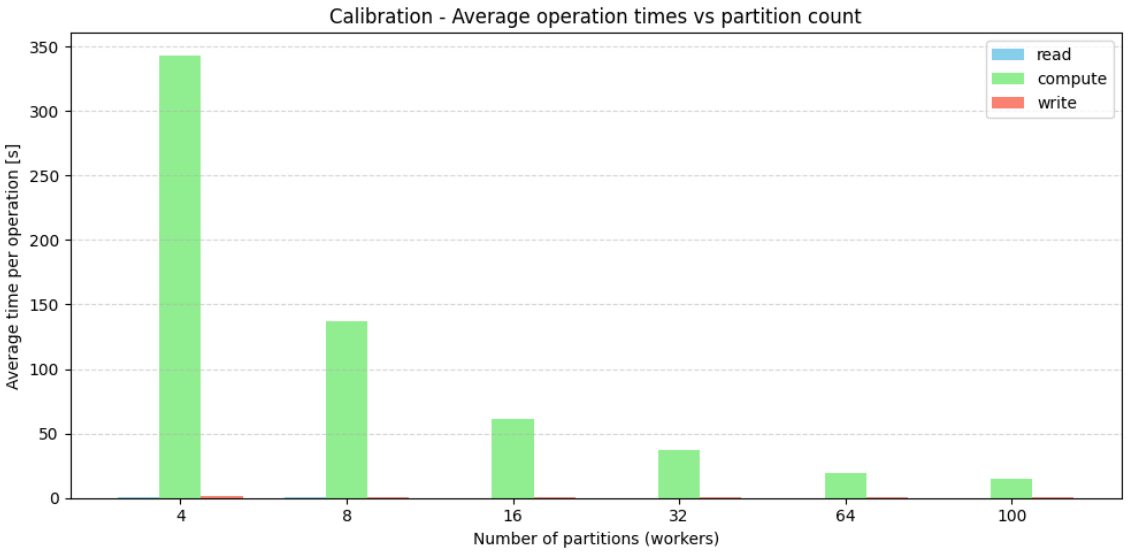

Figure 5.16: Breakdown of read, compute, and write times during calibration.

No major overhead emerges, confirming that calibration is the stage that benefits the most from parallel execution.

Imaging stage: As previously stated, this stage is monolithic and performs worse with increased parallelism due to I/O bottlenecks.

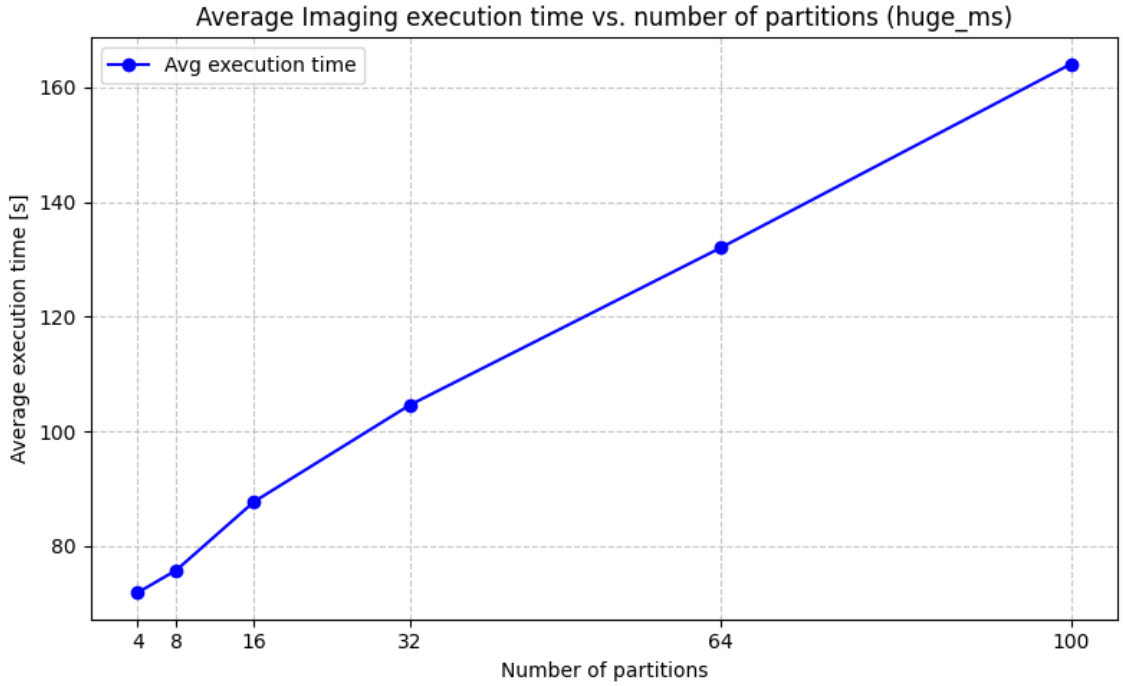

Figure 5.17: Imaging execution time vs. number of partitions.

As shown above, execution time increases almost linearly with the number of partitions.

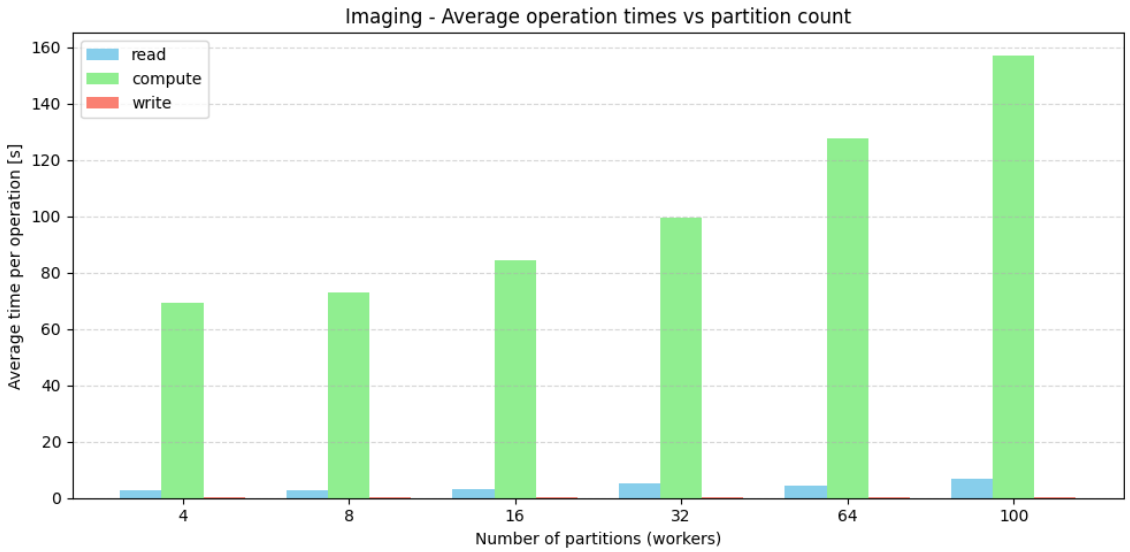

Figure 5.18: Breakdown of imaging stage (read, compute, write).

While performance is relatively acceptable up to 16 partitions, we see no real benefit beyond that. This confirms imaging as the main bottleneck in the pipeline.

Overall evaluation: We observe that up to a more than reasonable point (between 32–64 partitions), `Dataplug's` chunking significantly improves overall pipeline performance.

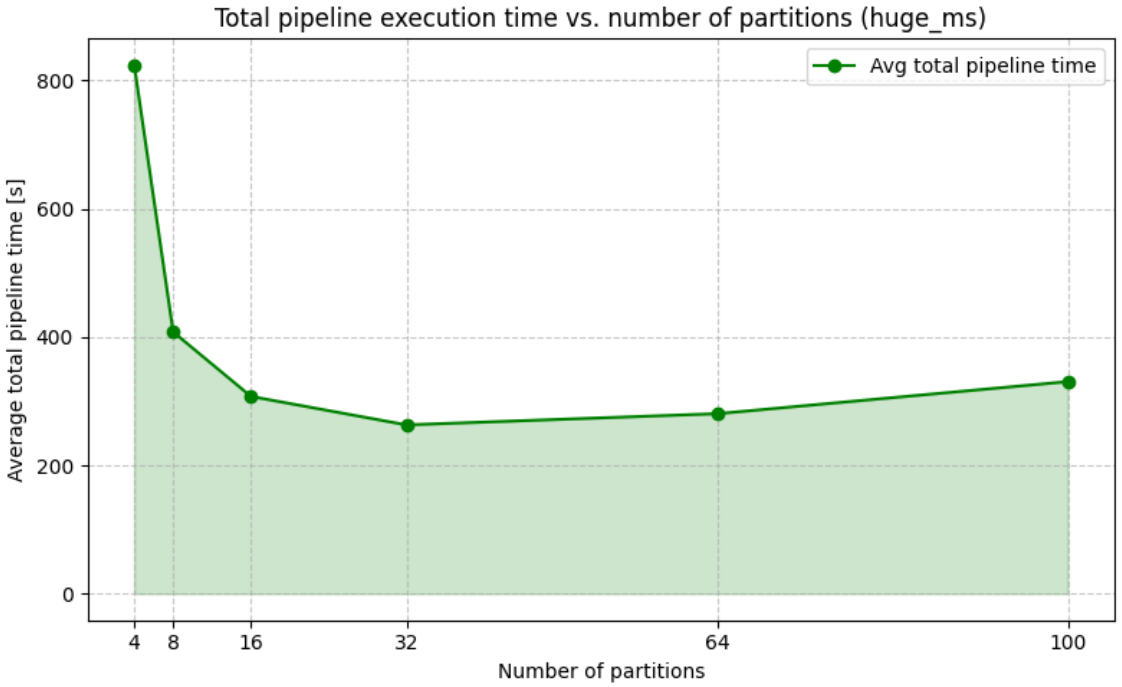

Figure 5.19: Overall execution time over number of partitions(`hugems`).

Even if 64 partitions are slightly slower than 32, they still allow data slices of roughly 8GB / 64 = 125MB (plus 25MB for immutable files), making it possible for very

lightweight machines (1vCPU, 150MB memory) to efficiently process astronomical data in parallel.

At our worst point (4 partitions), execution time is over 800s, while at the best (around 250s with 32 partitions), this represents a performance gain of nearly **69%**.

Final Evaluation

We will conclude with a comparison of total pipeline performance across partition counts, using a defined baseline.

For `hugems`, we couldn't use 2 or 1 partitions due to timeouts (10-minute execution cap). Therefore, we set a conservative baseline with 3 minutes for imaging and 10 minutes for the rest, representing a realistic upper bound. For `smallms`, we estimated a 60-second baseline, accounting for calibration's poor scaling—even on small files—due to column-based complexity.

Using these baselines, we calculate:

- **Parallel Speedup:** $S = \frac{T_{\text{baseline}}}{T_{\text{parallel}}}$, which measures how much faster the parallel version is.
- **Parallel Efficiency:** $E = \frac{S}{P}$, where P is the number of workers. This evaluates how effectively the parallel resources are used.

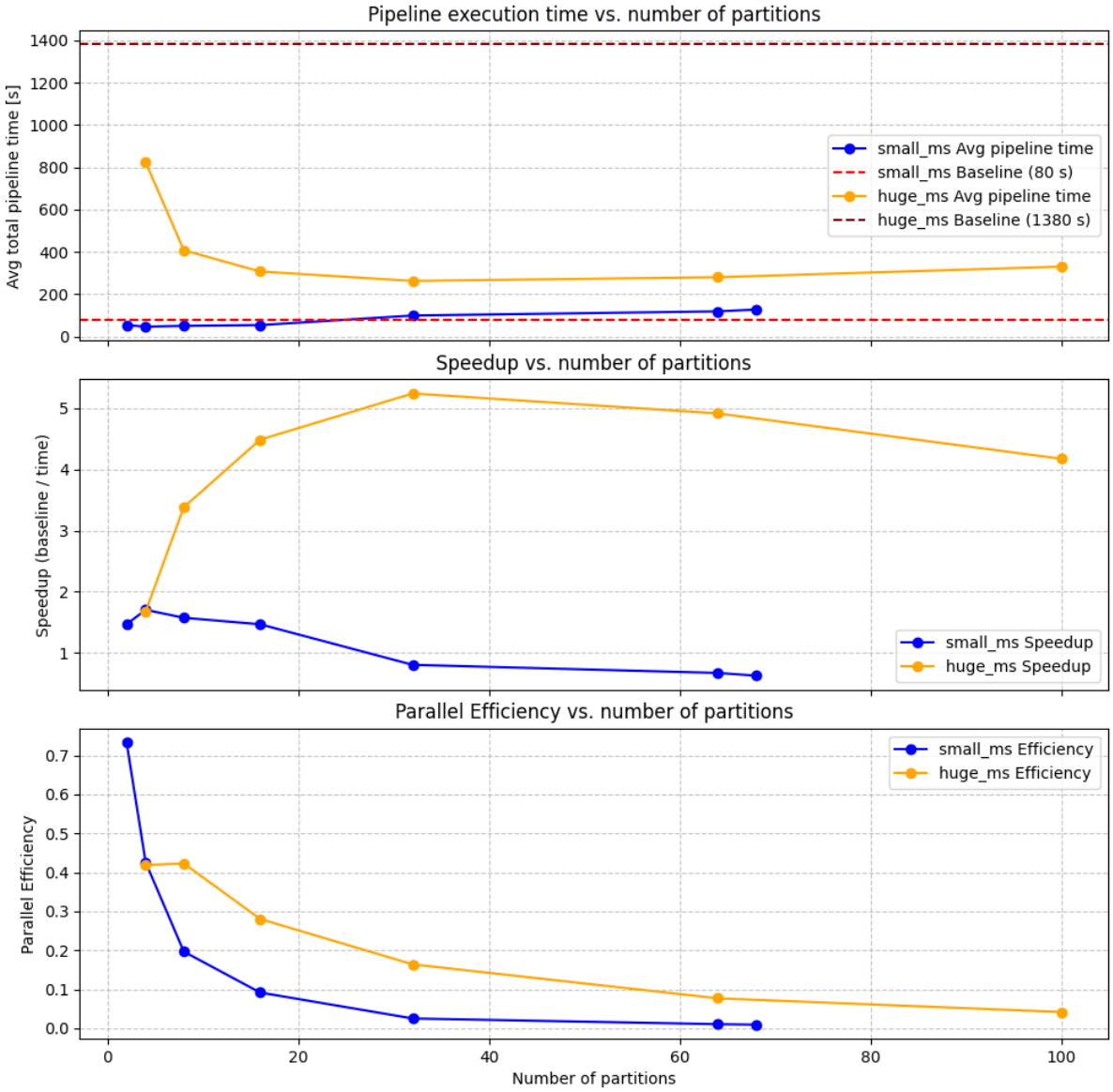

Figure 5.20: Time, speedup and efficiency across partition counts and files

We observe speedups up to $\mathbf{5}\times$, likely underestimated given that baseline executions timed out. Efficiency declines with more workers—especially for `smallms`—but remains more stable for `hugems`. While high efficiency typically implies better use of resources, we prioritize throughput: using more workers is acceptable even if it results in lower theoretical efficiency.

Additionally, the pipeline integrates adaptive resource management techniques inspired by modern orchestration systems (e.g., profiling, prediction, and dynamic configuration selection) as mentioned in previous chapters. Dataplug complements those , enabling the pipeline's ability to scale and optimize workflows without pre-processing again at each step, although a full evaluation of this modeling is beyond this thesis's scope.

Conclusions

In our experiments, we observed the best overall performance and speedup for larger MS at **32 partitions**, which yielded the fastest execution time across all configurations. While configurations with 64 partitions showed some additional gains in specific stages, they also introduced overheads—particularly in orchestration and I/O—which slightly reduced overall efficiency. We believe that with further fine-tuning, the true optimum may lie somewhere between 32 and 64 partitions, depending on workload characteristics and infrastructure conditions.

It is also worth noting that for smaller datasets, configurations with around 4 partitions tend to yield the most effective execution times, as the benefits of parallelism are quickly offset by coordination and data transfer overheads.

Crucially, these performance insights were only made possible thanks to Dataplug's dynamic ingestion mechanism, which allowed us to explore a wide range of partitioning strategies without the heavy cost of pre-generating static chunks. For example, at 32 partitions during the rebinning stage (see Figure 5.13), total runtime was approximately 100 seconds, with 20 seconds of compute, of which we attribute 10 to Dataplugs processing. In contrast, using static chunking would require around 300 seconds just to prepare the data, plus the same 100 seconds of execution(minus 10)—resulting in a total of around 400 seconds. This represents an almost 4× speedup (almost**75%** reduction in time), highlighting the significant efficiency gains enabled by our system.

5.3.3. Discussion

Summary of Key Findings

This evaluation demonstrates that integrating Dataplug into the astronomical data processing pipeline works seamlessly, requiring no major adjustments. It also validates the practicality and effectiveness of the Astronomics plugin we developed—capable of producing perfect slices, that can be reconstucted into identical copies of the original files through chunk concatenation. Compared to partitioning solutions that rely solely on Casacore (which demand deeper technical knowledge of the suite), Dataplug offers a more accessible, user-friendly effective approach for efficiently partitioning astronomical data.

Significant performance improvements were observed for large datasets with increasing parallelism. The calibration stage benefited the most from parallel execution, while the imaging stage showed no gains, as expected. Overall, speedups of up to **5×** were achieved, and large performance gains(in some cases, **up to 95%**) over less parallel solutions, demonstrating the effectiveness of our approach.

Interpretation and Implications

While large datasets clearly benefit from this approach, small datasets show limited improvement and can suffer from overhead costs. Thus, our solution is best suited for large-scale workloads, where its advantages are most pronounced.

Limitations

Several limitations should be considered. The Kubernetes rack used introduced variability and may not fully represent other environments. Overhead on casacore operations. Efficiency decreases as the number of workers grows, which can lead to resource waste. Finally, as long as the imaging stage is unchanged, it will remain a bottleneck.

Design and Infrastructure Considerations

The pipeline design balances flexibility and efficiency. The serverless model adds adaptability but can be inefficient for I/O-heavy or monolithic stages. Parallelism improves performance up to a point, beyond which overhead dominates and reduces gains.

Future Work

Exploring other distributed environments (e.g., AWS Lambda, Azure Functions) or CaaS platforms may offer benefits for data-heavy, non-CPU-bound stages like imaging. To address the imaging bottleneck, introducing an intermediary step for parallel data aggregation could help distribute Casacore I/O overhead. Reverse-engineering the remaining CASA functions within Dataplug may allow decomposition into core operations, minimizing overhead from column rearrangement of immutable files, further optimizing the data cleanup. Lastly, testing with other types of astronomical data will help validate and, if necessary, adapt the plugin for broader applicability, even if it's current design should theoretically accommodate every MS file.

5.3.4. Summary of Evaluation

The evaluation confirms the effectiveness of dynamic partitioning with *Dataplug* across multiple dimensions: performance, scalability, and data integrity.

- **Consistency and Data Integrity:** Files generated with *Dataplug* are byte-for-byte identical to the originals, except for `table.info`, which correctly reflects reconstruction metadata. This is expected and reversible, ensuring compatibility and data fidelity.
- **Flexibility and Redundancy Reduction:** Dynamic partitioning eliminates the need for pre-generated chunks, enabling fast iteration and reduced pre-processing and memory overhead. This flexibility is critical for scalable and exploratory workflows and gives way to modeling and prediction works.
- **Best-Case Efficiency:** In scenarios where preprocessing is unnecessary—e.g., partition metadata is already available—dynamic partitioning achieved up to a **95.6% speedup**. For instance, execution dropped from **287.2s** (static) to **12.5s** (dynamic) with 16 partitions.
- **Performance and Scalability:** The best performance on large MeasurementSets was achieved with **32 partitions**, delivering the fastest execution time across all

configurations. For smaller datasets, around **4 partitions** proved most efficient, as the benefits of parallelism were offset by coordination overheads beyond that point.

- **Speedup Against Baseline:** In the rebinning stage at 32 partitions, dynamic ingestion reduced total theorical runtime from at least **400s** (static preprocessing) to **100s**, yielding a **75% improvement** over the baseline.

These results validate the robustness, efficiency, and adaptability of our approach. Dataplug not only reduces execution time but also improves flexibility and reproducibility by lowering both computational and memory overheads in distributed data processing.

6 Ethical and Social Considerations

It is important to review this project under ethical, social, and environmental responsibilities. These dimensions are essential in ensuring that the outcomes of the work contribute positively not only in a technical sense but also in terms of fairness, sustainability, and respect for societal values.

The following subsections reflect on relevant issues of gender equality, environmental impact, social responsibility, and ethical practice as they relate to the project's context and implementation.

6.1. Gender Equality

This project focuses on astronomical data formats and cloud technologies applied to the processing of observational data from natural phenomena. As such, it does not involve human subjects or contexts where gender bias could arise. While the broader field of science and technology continues to face gender disparities, this particular work does not present scenarios where such issues are applicable.

6.2. Environmental Awareness

The improvements introduced in this project enable more effective optimization strategies, allowing users to select the most efficient execution paths—choices that are inherently more sustainable than less effective alternatives. Enhancements in computational efficiency and streamlined pre-processing help reduce both energy consumption and data transfer overhead. While maximizing performance can sometimes lead to increased energy use, this trade-off is left to the discretion of the users running the pipelines. Our work ensures that all conditions are in place to support environmentally conscious decisions.

6.3. Social Responsibility

The project is aligned with open science principles and the democratization of access to data and computational resources. It has involvement on the EXTRACT project, which promotes reuse and sharing of scientific knowledge through accessible and standardized infrastructures. This open approach may benefit institutions with limited technological

capacity and contributes to reducing digital divides. The role of technology as a lever for social transformation is recognized, and efforts were made to ensure that the outcomes of this project can be beneficial beyond the academic context.

6.4. Ethical Considerations

The project was developed with respect for academic and professional integrity. Third-party code, tools, and datasets were properly credited in accordance with licensing and intellectual property requirements. Although the work was mostly carried out individually, every external contribution or resource was acknowledged transparently.

Ethical and deontological principles from the field of computing were taken into account, particularly in decisions involving algorithmic transparency, and resource consumption. Care was taken to evaluate the long-term implications of technical choices, not only in terms of performance but also regarding fairness, reproducibility, and environmental impact.

All decisions were made with awareness of their potential consequences for users, systems, and broader societal contexts, in line with the expectations of responsible academic conduct within the university community.

7 Conclusions

This final project has been both professionally and personally fulfilling. In engineering, it is natural to approach problems with the goal of solving them—that's at the core of the discipline. However, unlike traditional engineering, research often lacks a clear or predefined methodology.

Working with MeasurementSets proved to be a significant challenge. The documentation is limited, the field is highly specialized, and there are few people to consult. I found myself trying to solve problems that even experts in the field either hadn't considered or assumed to be unresolvable, or not worth solving.

Participating in the European EXTRACT project and developing this plugin during my time at the CloudLab has provided me with valuable experience in both engineering and research. I've gained hands-on knowledge in working with a complex data format, in cloud technologies, and in distributed processing—and the results of my work are already being applied to ongoing projects, helping others streamline their workflows and process their data more efficiently.

This thesis does not go into detail about every aspect of my journey, as the project took longer than initially expected and frequent roadblocks required constant adaptation. Nonetheless, overcoming those challenges was part of the learning process, and continuing forward was deeply rewarding. If given the opportunity, I would like to further refine both the plugin and the pipeline.

In short, this has been an enriching experience and, in my view, an ideal way to conclude my Computer Engineering degree. It allowed me to engage with nearly every facet of the discipline. I would gladly take part in a project like this again and strongly encourage others to seek out similar opportunities.

Bibliography

[1] CASA Team. *Measurement Set Format Reference*. 2021. URL: https://casa.nrao.edu/Memos/229.html (page 6).

[2] Amazon Web Services. *AWS Lambda Quotas - Memory and CPU Allocation*. 2025. URL: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html (page 6).

[3] *EXTRACT – A distributed data-mining software platform for EXTReme data ACross the compute conTinuum*. 2022. URL: https://cordis.europa.eu/project/id/101093110 (page 6).

[4] EXTRACT Project Consortium. *Transient Astrophysics with a Square Kilometre Array Pathfinder (TASKA)*. 2024. URL: https://extract-project.eu/case-2/ (page 6).

[5] EXTRACT Project Consortium. *TASKA "C" Use Case: Fast-paced data calibration and imaging on the "C"loud*. Nov. 2024. URL: https://extract-project.eu/taska-c-use-case-fast-paced-data-calibration-and-imaging-on-the-cloud/ (page 6).

[6] Aitor Arjona, Pedro García-López, and Daniel Barcelona-Pons. «Dataplug: Unlocking extreme data analytics with on-the-fly dynamic partitioning of unstructured data». In: *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2024, pp. 567–576. DOI: 10.1109/CCGrid59990.2024.00069 (page 7).

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. «A view of cloud computing». In: *Commun. ACM* 53.4 (2010), pp. 50–58. DOI: 10.1145/1721654.1721672 (page 10).

[8] Red Hat, Inc. *What is container orchestration?* 2024. URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration (page 11).

[9] Microsoft Azure. *Azure Functions*. URL: https://azure.microsoft.com/es-es/products/functions (page 11).

[10] Google Cloud. *Cloud Functions Documentation*. 2025. URL: https://cloud.google.com/functions (page 11).

[11] Eric Jonas, Johann Schleier-Smith, Vinay Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Kevin Krauth, Joseph E. Gonzalez, and Ion Stoica. «Cloud Programming Simplified: A Berkeley View on Serverless Computing». In: *arXiv preprint arXiv:1706.03178* (2017). URL: https://arxiv.org/abs/1706.03178 (page 12).

[12] Docker Inc. *Docker Overview*. 2025. URL: https://docs.docker.com/get-started/docker-overview/ (page 13).

[13] The Kubernetes Authors. *Kubernetes Documentation: Overview*. 2025. URL: https://kubernetes.io/docs/concepts/overview/ (page 13).
[14] Amazon Web Services. *Amazon S3 Documentation*. 2025. URL: https://docs.aws.amazon.com/s3/ (page 14).
[15] Amazon Web Services. *Using Byte-Range Fetches in Amazon S3*. 2025. URL: https://docs.aws.amazon.com/es_es/whitepapers/latest/s3-optimizing-performance-best-practices/use-byte-range-fetches.html (page 14).
[16] MinIO, Inc. *MinIO S3 Compatibility*. 2025. URL: https://min.io/product/s3-compatibility (page 14).
[17] Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López. «Outsourcing Data Processing Jobs With Lithops». In: *IEEE Transactions on Cloud Computing* 11.1 (2023), pp. 1026–1037. DOI: 10.1109/TCC.2021.3129000 (page 14).
[18] The Lithops Team. *Lithops Architecture*. Accessed: 2025-04-14. 2023. URL: https://lithops-cloud.github.io/docs/source/design.html (page 15).
[19] casacore developers. *casacore: A suite of C++ libraries for radio astronomy data processing*. 2025. URL: https://casacore.github.io/casacore/ (page 16).
[20] python-casacore maintainers. *python-casacore: Python bindings to the casacore C++ library*. 2025. URL: https://pypi.org/project/python-casacore/ (page 16).
[21] Ger van Diepen. *NOTE 260 – CTDS File Formats*. https://casacore.github.io/casacore-notes/260.html. 2017 (page 19).
[22] NRAO. *casacore::DataManager Class Reference*. https://casa.nrao.edu/doxygen/classcasacore_1_1DataManager.html. Accessed: 2024-12-23 (page 20).
[23] casacore team. *Casacore Notes Index*. https://casacore.github.io/casacore-notes/. Accessed: 2025-06-02. Note 260, the one that defines tables, is marked as incomplete. 2025 (page 20).
[24] CloudLab. *Pursuing the Ultimate Serverless Experience*. https://extract-project.eu/pursuing-the-ultimate-serverless-experience/. July 2024 (page 21).
[25] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. «Caerus: NIMBLE Task Scheduling for Serverless Analytics». In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 653–669. ISBN: 978-1-939133-21-2. URL: https://www.usenix.org/conference/nsdi21/presentation/zhang-hong (page 23).
[26] Haoyu Fu, Diankun Zhang, Zongchuang Zhao, Jianfeng Cui, Dingkang Liang, Chong Zhang, Dingyuan Zhang, Hongwei Xie, Bing Wang, and Xiang Bai. *ORION: A Holistic End-to-End Autonomous Driving Framework by Vision-Language Instructed Action Generation*. 2025. arXiv: 2503.19755 [cs.CV]. URL: https://arxiv.org/abs/2503.19755 (page 23).
[27] Zili Zhang, Chao Jin, and Xin Jin. «Jolteon: Unleashing the Promise of Serverless for Serverless Workflows». In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 167–183. ISBN: 978-1-939133-39-7. URL: https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-jolteon (pages 23, 25).
[28] Zachary Novack, Julian McAuley, Taylor Berg-Kirkpatrick, and Nicholas J. Bryan. *DITTO: Diffusion Inference-Time T-Optimization for Music Generation*. 2024. arXiv: 2401.12179 [cs.SD]. URL: https://arxiv.org/abs/2401.12179 (page 23).

[29] CLOUDLAB URV. *Dataplug: Framework Architecture Diagram*. https://github.com/CLOUDLAB-URV/dataplug/blob/master/docs/framework-architecture.png. 2024 (page 28).
[30] CLOUDLAB. *CLOUDLAB — Cloud and Distributed Systems Lab*. https://cloudlab-urv.github.io/WebCloudlab/. 2025 (page 30).