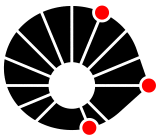

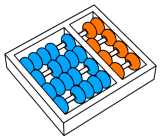

A Framework for running DASF applications with Kubernetes and Argo

Ramon Galate Baptista Ribeiro *Edson Borin*

Relatório Técnico - IC-PFG-23-66
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Framework for running DASF applications with Kubernetes and Argo

Ramon Galate Baptista Ribeiro, r205112@dac.unicamp.br
Advisor: Edson Borin, borin@unicamp.br

Abstract

The present report details the development of a framework for the construction of self-contained pipelines for data processing, using Dask and Argo for the generation of these pipelines. This structure benefits from Dask Kubernetes Operator for adaptive scaling, automatically adjusting according to demand and complexity of tasks. Implemented on the Kubernetes platform, this framework ensures scalability, flexibility, and optimization of resources.

The system was designed to cater to a wide range of applications, being especially relevant in areas such as seismic data analysis and ETL (Extraction, Transformation, and Loading) processes. Thanks to the efficient integration between Dasf, Dask, Argo, RapidsAI, and Kubernetes, it is possible to handle workloads of various sizes and intensities, enhancing the processing and analysis of large volumes of data.

In a scenario where data production is growing exponentially, the need for robust and scalable solutions like this becomes essential. This project is an important step towards scalable solutions, paving the way for future innovations in the field of large-scale data processing.

Contents

1 Introduction

The development of data processing and machine learning applications is generally based on a recurring flow of code development and testing with data sets. For this interactive development flow to be effective, it is important that the developer has quick access to test results. Therefore, development and most tests are usually carried out on dedicated workstations (*e.g.*, laptops and/or desktop computers) and with small samples of the data sets. Once the code functionalities have been validated with the small data sets, the code is migrated to a high-performance computing system and executed with the large data set. This execution can take hours or even days, depending on the size of the data set or the waiting time in the computational system's queues. Finally, once the system finishes processing, the developer analyzes the results and can return to the interactive flow to make adjustments to the code or implement new functionalities. Figure 1 illustrates these two development flows.

One of the challenges faced in this type of development is the adaptation of the code used in interactive development flow so that it can be executed in a scalable manner on a high-performance computing system. Often, this requires the implementation of specialized routines for accelerators (such as GPUs) and the adaptation of the code for distributed execution on multiple computational nodes. Therefore, various frameworks have emerged to allow the developer to implement and test the application on a workstation and then scale the execution of the application to high-performance computational systems with little effort.

DASF is a tool being developed by the Dicovery lab at UNICAMP that has the purpose of enabling the processing of seismic data in a scalable manner.

Although DASF allows the implementation of complex applications, it is often of interest to combine DASF applications with other applications in a workflow. In this context, the DASF application acts as a task, consuming data produced by other tasks (*e.g.*, other applications) and producing data for other tasks. In particular, there is interest in executing DASF applications as tasks of the Argo workflow manager [1]. This workflow manager simplifies the integration of multiple applications through the execution of tasks in specialized

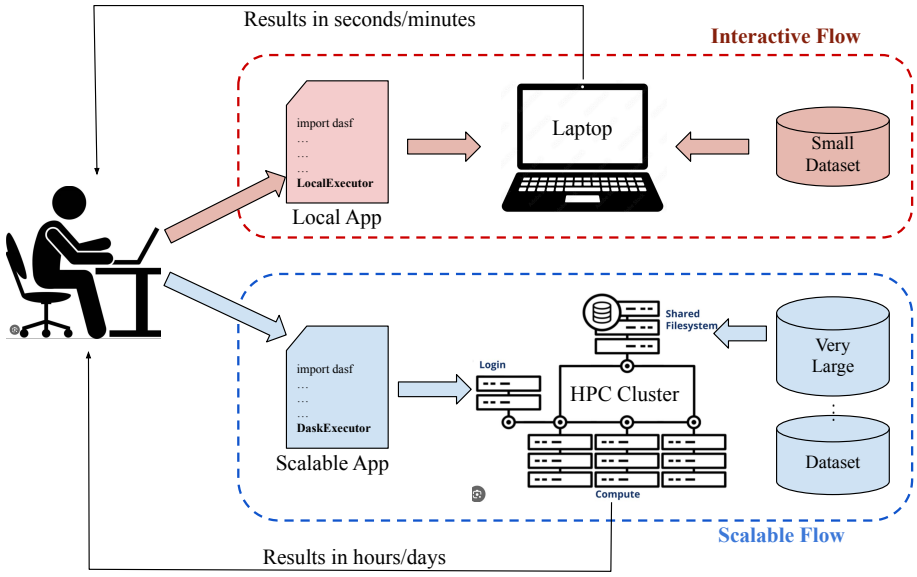

Figure 1: Interactive and scalable data processing and machine learning applications development cycles.

containers, which allows the specialization of the environment (such as the installation of libraries and/or adjustments of environment variables) for each task.

In its standard form, Argo executes each task in a container, however, to scale the execution of tasks based on DASF applications, it is important that this type of task is executed in a cluster of computational nodes, *i.e.*, in multiple containers. Therefore, it is necessary to implement tools for the configuration and management of DASF task execution in the Argo system.

In this work, we propose the implementation of a framework to facilitate the scalable execution of DASF tasks in Argo workflows. For this purpose, we designed a system that utilizes Argo, Pulumi, Dask, DASF, Hera, and Kubernetes. As a result, the framework has simplified the execution of scalable DASF tasks in Argo, requiring only a few lines of code from the developer.

The remainder of this report is organized in the following way:

Section 2 presents the fundamental concepts and tools essential for understanding the framework. Section 3 goes over the functional and non-functional requirements that our solution had to meet to be considered a success. Section 4 details the Argo-DASF Framework, discussing its overview, the creation of custom Docker images for DASF, the integration of Infrastructure as Code, and specific components like the Dask Operator and storage module. Section 5 focuses on the validation and results of the project, demonstrating the effectiveness and efficiency of the Argo-DASF framework in real-world applications. Section 6 covers the project development process, encompassing the stages from requirements gathering to the final report submission. Finally, Section 7 offers conclusions, summarizing the achievements, challenges, and the overall impact of the project.

2 Fundamental Concepts and Tools

The following sections present the fundamental concepts and tools used in our framework.

2.1 Workflows

A workflow [15], in the context of computing and data processing, is essentially a series of tasks that are executed in accordance with a set of procedural rules. These tasks are often represented as nodes in a graph, with the edges indicating dependencies between them. This structure not only defines the set of tasks but also imposes an order on their execution. Figure 2 illustrates a dependency graph for tasks.

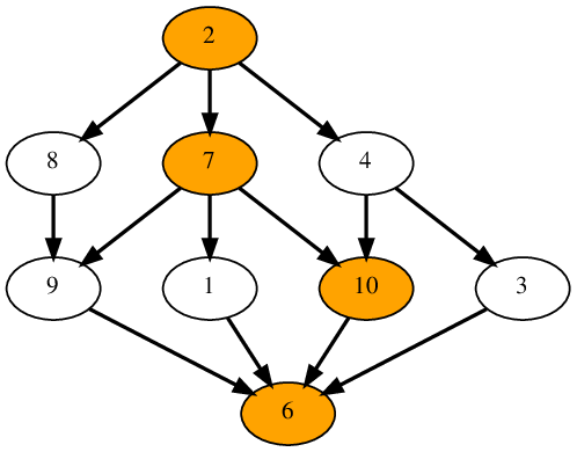

Figure 2: Example of a dependency graph for tasks

Key Terms and Concepts

- *Tasks:* These are the individual units of work or operations within a workflow. Each task performs a specific function or process.
- *Dependencies:* Dependencies refer to the relationship between tasks, where the initiation or completion of one task is contingent upon the initiation or completion of another.
- *Parallel Execution:* Tasks that do not have dependencies among themselves can be executed in parallel. This is managed by the workflow scheduler, which optimizes the execution flow for efficiency.
- *Containers:* Containers are lightweight, standalone, executable packages that include everything needed to run a task: code, runtime, system tools, system libraries, and settings.
- *Computational Nodes:* These are the physical or virtual machines where the containers, hence the tasks, are executed. They provide the necessary processing power.

- *Storage:* Systems like Network File System (NFS) are often used to store data that is used or generated by the workflow tasks.

Workflows are crucial in automating complex sequences of tasks, especially in fields like data science, software development, and systems administration. They help in orchestrating and automating the processes, making them more efficient, reliable, and scalable.

By understanding these basic concepts, you can better grasp how workflows function and how they are crucial in the context of computational tasks and data processing. This understanding is essential, especially when discussing more advanced topics like workflow optimization and the integration of specific technologies like containerization and distributed computing in workflow execution.

A note about terminology: In the literature, the terms workflows and pipelines are often used interchangeably, since they define the same concept. For clarity, in this project, we reserve the pipeline term to refer to Dask and DASF pipelines, while the workflow term will be dedicated to refer to Argo Workflows. Both represent a series of tasks that use a dependency mechanism to define which task to be performed before the other. But in the context of this work, Argo workflows are a series of tasks where a task can be a DASF pipeline. So from here on out, whenever we mention a workflow we are specifying workflows in the context of Argo Workflows and pipeline for DASF/Dask pipelines.

2.2 Kubernetes

Kubernetes [10], often abbreviated as K8s, is an open-source platform designed to automate deploying, scaling, and operating application containers. It groups containers that make up an application into logical units for easy management and discovery.

The key components of Kubernetes are:

- Pods: The smallest deployable unit that can be created and managed in Kubernetes. Each Pod represents a single instance of a running process in a computing cluster. When comparing to Docker containers, a container is a standalone unit of software encapsulating an application's code and dependencies, whereas a pod in Kubernetes is a group of one or more containers with shared storage and network resources, operating as a single entity.

- Services: An abstraction which defines a logical set of Pods and a policy by which to access them. This is often used to expose a set of Pods as a "network service" .

- Volumes: A way to persist data in a pod.

- Namespaces: Used to organize objects (pods, volumes, services...) in a cluster and provide a way to divide cluster resources between multiple users.

2.3 Operator Architecture

Kubernetes Operators are a method of packaging, deploying, and managing Kubernetes applications. An operator takes human operational knowledge and encodes it into software that is more easily packaged and shared with consumers.

1. *Custom Resource Definitions (CRDs):* Operators extend Kubernetes by introducing new object types (objects with a certain configuration that defines them) called custom resources. CRDs allow users to create and manage these custom resources just like they would do with standard Kubernetes objects. Imagine you're creating a new application that requires tracking of specific data not covered by standard Kubernetes resources. With CRDs, you can define a 'Book' resource with properties like 'author', 'title', and 'publishDate'. Once defined, you can use Kubernetes API to manage your 'Book' resources, similar to how you manage built-in resources like Pods.

2. *Controller:* The core of an operator is its controller. This controller watches the Kubernetes API for changes to specific resources and takes action to drive the current state towards the desired state.

3. *Focus of Operator Architecture:* The operator architecture focuses on automating complex, application-specific operations. It extends the Kubernetes API and its functionalities, allowing for more sophisticated deployment and lifecycle management of applications. The operator enables developers to encode domain knowledge for specific applications into Kubernetes extensions.

By leveraging the Kubernetes Operator architecture, organizations can automate and simplify many aspects of deploying and managing cloud-native applications, thus enabling a more efficient and scalable infrastructure.

2.4 Argo

Argo [1] is an open-source platform that provides tools for orchestrating tasks in Kubernetes environments, making it possible to execute, for example, workflows, CI/CD pipelines, and batch computations. It was specifically designed to leverage Kubernetes resources, optimizing container orchestration and task automation. One of the main features of Argo is its operator for Kubernetes, the Argo workflow operator. This operator extends the Kubernetes API, adding new types of custom resources, thus allowing Kubernetes to understand and manage Argo workflows natively. This significantly simplifies the management and execution of complex tasks and workflows in a Kubernetes cluster.

One of the biggest advantages of this integration is the ability to create an endpoint for submitting self-contained pipelines. An endpoint, in this context, is essentially a specific URL or a network address where external systems can send requests to interact with the integrated service. This endpoint allows users to define and submit all the necessary elements for the execution of a pipeline in a unified form. This includes task definitions, dependencies, and data.

The creation of such an endpoint significantly streamlines the process of integrating with external systems, automations, and facilitates the rapid iteration and testing of new workflows. Users can leverage this feature to make quick and efficient changes without needing extensive modifications to the existing infrastructure, emphasizing the flexibility and scalability of the system.

For interaction with the Argo workflows operator, we use the Hera library, an Argo workflows client in Python, which allows direct integration with the notebook/environment of DASF (Python).

Essentially, Argo is a tool designed for executing workflows composed of tasks, where each task is generally carried out by a container or a Custom Resource Definition (CRD). We aim to implement a framework to facilitate the scalable execution of DASF pipelines as tasks within Argo workflows. This approach will enable each task to be executed in a cluster of containers using CRDs, rather than just a single container. This methodology significantly enhances the scalability and efficiency of processing, leveraging the distributed nature of container clusters to optimize the performance of DASF applications within the Argo workflow environment.

2.5 Infrastructure-as-Code

Infrastructure as Code (IaC) is a key practice in the realm of DevOps and modern cloud computing that involves managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. This approach enables developers and system administrators to automatically manage, monitor, and provision resources through code, rather than using a manual process.

IaC brings several advantages, such as:

1. Consistency and Standardization: By defining infrastructure through code, it ensures that the environment is consistent and reduces the chances of discrepancies or manual errors.

2. Version Control: Infrastructure configurations can be versioned, audited, and tracked using the same version control systems used for source code.

3. Efficiency in Deployment: IaC allows for rapid deployment and updating of infrastructure, enabling faster and more agile responses to changes in business requirements or technology.

4. Documentation: The code itself serves as documentation, showing exactly what the infrastructure looks like at any point in time.

Some well-known tools and technologies in the field of Infrastructure as Code include:

- Terraform: A tool that allows you to define infrastructure in a high-level configuration language.

- Pulumi: An open-source tool that allows you to define infrastructure in a high-level configuration language.

- Ansible: An open-source automation tool for software provisioning, configuration management, and application deployment.
- Chef: A powerful automation platform that transforms infrastructure into code.
- Puppet: A software configuration management tool which automates the provisioning and management of infrastructure.

For this project, we have chosen Pulumi, Helm, and the Hera API. The following sections describe these tools.

2.5.1 Pulumi

Pulumi [12] is an open-source infrastructure as code platform that allows developers and infrastructure operators to define and provision cloud resources using traditional programming languages, instead of markup languages or specialized scripts. Pulumi stands out for its ability to integrate with various cloud providers and tools, offering a unified experience. Pulumi was chosen for this project due to its flexibility, ease of use, and the power it offers for complex infrastructure setups. It has the following properties:

1. **Clear Definition:** By using popular programming languages such as Python, TypeScript, Go, among others, developers can define complex infrastructures with the flexibility and power that these languages offer.
2. **Complete Automation:** With Pulumi, the entire infrastructure, from creating Kubernetes clusters to configuring Argo and additional integrations, can be automated, ensuring consistency and reducing human errors.
3. **Reproducibility:** One of the major advantages of using IaC is the ability to reproduce the exact same infrastructure in different environments or regions. This facilitates the processes of testing, staging, and production, and allows different teams to work in harmony.
4. **State Management:** Pulumi manages the state of the infrastructure, allowing for incremental updates, rollbacks, and detailed previews of proposed changes before they are applied.

In summary, with Pulumi, it is not only possible to create all the necessary infrastructure automatically, but also to ensure that this infrastructure is easily reproducible in various contexts, promoting agility, consistency, and reliability to the project.

2.5.2 Helm

Helm [8] is a package management tool for Kubernetes, allowing users to define, install, and update Kubernetes applications simply and consistently. Helm charts are packages of instructions for Kubernetes, containing predefined and configurable resource definitions. One of the main utilities of Helm charts is the ability to define and manage CRDs (Custom

Resource Definitions) and other complex structures, enabling simpler and more modular management of Kubernetes resources.

Here are some ways in which Helm and Pulumi can be used together to facilitate infrastructure creation:

1. **Direct Integration:** Pulumi has direct integration with Helm, allowing developers to leverage existing Helm charts and customize them as needed, all within the Pulumi ecosystem.

2. **Simplified Configuration:** Using Helm charts, the definition of CRDs and other Kubernetes resources becomes much more simplified and modular. Pulumi can then be used to adjust and personalize these definitions as per the specific needs of the project.

3. **Automation:** Pulumi can automate the deployment and management of Helm charts, making the process of configuring and deploying resources even more seamless and integrated.

4. **Version Management:** Since Helm charts are versioned, it is possible to use different versions of a chart for different environments or phases of the project, ensuring consistency and reproducibility. Pulumi, in turn, can manage and orchestrate these different versions efficiently.

In summary, the combination of Helm and Pulumi provides a powerful and flexible approach to defining, configuring, and managing complex infrastructures in Kubernetes, simplifying the process and ensuring efficiency and reliability. Both the Kubernetes Dask operator, which creates the KubeClusters (Dask Clusters created in a Kubernetes environment), and the Argo Operator, which creates the DASF containers responsible for executing the DASF tasks, have Helm charts that we can use to create this desired structure.

2.6 Hera Client

The Python Hera [9] client for Argo Workflows is a tool developed to simplify the construction and submission of workflows in the Argo ecosystem. This Python SDK (Software Development Kit) aims to make accessing and using Argo project resources easier and more accessible for everyone.

Here are some features of Hera that made us choose it for our development:

- **Abstraction of Configuration Details:** The tool abstracts workflow configuration details, allowing users to focus more on workflow logic than on its technical configuration. This is achieved by maintaining terminology consistency with Argo while removing native Kubernetes YMAL templating elements, making it easier for users already familiar with Argo to understand and use Hera.

- **Construction of DAGs:** Hera is particularly useful in constructing DAGs, as it simplifies the process of defining workflow steps and dependencies. By using Hera, users can focus on business logic and workflow, rather than worrying about the syntax and structure of YAML.

- **Abstraction of Traditional YAML Templates:** One of the main challenges with Argo is the need to write and maintain complex YAML templates. Hera addresses this challenge, allowing users to define workflows in Python, a more familiar and easier-to-use language for many developers. This not only simplifies the workflow creation process but also makes maintaining and tuning workflows more accessible.

- **Expanding Accessibility to the Argo Ecosystem:** By providing a more user-friendly and less technical interface, Hera makes the Argo ecosystem more accessible to a broader audience, including those who may not have a strong technical background in Kubernetes infrastructure and configuration.

In summary, the Hera Python framework for Argo Workflows is a significant tool for those looking to simplify and streamline the process of building and managing complex workflows in Kubernetes environments, especially for those who prefer working with Python over YAML.

To exemplify the improved readability and ease of use of Hera when compared to the classical Argo workflows in pure YMAL, let us consider Listings 1 and 2. The first one illustrates how to create a diamond DAG workflow in Hera, while the second one shows how to create the same workflow using the Argo pure YMAL interface. Notice that the Hera interface allows a more compact and simpler definition when compared to the Argo one.

```python
from hera.workflows import (
    DAG,
    Workflow,
    script,
)

@script(add_cwd_to_sys_path=False, image="python:alpine3.6")
def echo(message):
    print(message)

with Workflow(generate_name="dag-diamond-", entrypoint="diamond") as w:
    with DAG(name="diamond"):
        A = echo(name="A", arguments={"message": "A"})
        B = echo(name="B", arguments={"message": "B"})
        C = echo(name="C", arguments={"message": "C"})
        D = echo(name="D", arguments={"message": "D"})
        A >> [B, C] >> D
```

Listing 1: Example of a diamond DAG in Hera

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: dag-diamond-
spec:
  entrypoint: diamond
  templates:
  - dag:
      tasks:
      - arguments:
          parameters:
          - name: message
            value: A
        name: A
        template: echo
      - arguments:
          parameters:
          - name: message
            value: B
        depends: A
        name: B
        template: echo
      - arguments:
          parameters:
          - name: message
            value: C
        depends: A
        name: C
        template: echo
      - arguments:
          parameters:
          - name: message
            value: D
        depends: B && C
        name: D
        template: echo
    name: diamond
  - inputs:
      parameters:
      - name: message
    name: echo
    script:
      command:
      - python
      image: python:alpine3.6
      source: 'import json

        try: message = json.loads(r''''''{{inputs.parameters.message}}'''''')
    )

        except: message = r''''''{{inputs.parameters.message}}''''''

        print(message)'
```

Listing 2: Example of a diamond DAG in pure Argo YMAL

2.7 The DASF Framework

DASF [3], or the Accelerated and Scalable Framework, is an approach for computing Machine Learning algorithms in Python using Dask and RAPIDS AI, specifically designed to address the challenges posed by deep learning techniques and data handling in batch formats. This innovative framework stands out in the current landscape of machine learning tools by offering unique capabilities and features.

The core concepts in DASF are:

1. *Distributed Data Processing:* Unlike existing tools, DASF introduces the capability to process data in a distributed manner across various computing nodes. This approach not only enhances scalability but also optimizes the utilization of computational resources.

2. *GPU Utilization:* DASF leverages the maximum computational power of GPUs, a feature not commonly found in many existing machine learning tools. This utilization significantly accelerates data processing, making DASF a highly efficient framework for complex computations.

3. *Standard API:* To ensure ease of use, DASF adopts a Standard API similar to the popular scikit-learn. This design choice minimizes the learning curve for new users and facilitates seamless integration into existing machine learning projects.

4. *Extensibility to New Devices:* DASF supports easy extensibility to various devices, including GPUs. Users can thus employ the framework across different hardware setups without worrying about compatibility or implementation details.

5. *Multi-Node Scalability:* The integration of Dask construct graphs in DASF allows for scalable, distributed computing across multiple nodes. This feature is particularly beneficial for large-scale projects requiring extensive computational resources.

DASF applications are Python applications composed of one or more DASF pipelines. A **DASF pipeline** is a graph in which nodes are **DASF operators** and edges indicate the data flowing between DASF operators. A **DASF operator** is a Python object designed to perform a task, which can be user-defined or imported from a set of operators available on the DASF framework. DASF applications can create and execute one or more DASF pipelines. These pipelines can be executed locally (on the same machine running the DASF application), or remotely, on an HPC cluster[1], for example.

Listing 3 shows a basic DASF application running an application locally with no GPU.

```
1 from dasf.pipeline import Pipeline
2 from dasf.ml.cluster import KMeans
3 from dasf.ml.cluster import SOM
4 from dasf.pipeline.executors import DaskPipelineExecutor
5 from dasf.transforms import Normalize, PersistDaskData
6
```

[1]Section 2.7.1 introduces Dask and Section 2.7.2 discusses how DASF employ Dask to execute DASF pipelines on remote HPC clusters.

```
# define a dataset
train_set = ...
# create the pipeline executor and objects
dask = DaskPipelineExecutor(local=True, use_gpu=False)
normalize = Normalize()
kmeans = KMeans(n_clusters=3, max_iter=100)
som = SOM(x=1, y=3, input_len=2, num_epochs=100)
# As we want to reuse the data after the pipeline execution, we need to
    persist the data
persist_kmeans = PersistDaskData()
persist_som = PersistDaskData()

# create the pipeline and adds the operators to it
pipeline = Pipeline("A KMeans and SOM Pipeline", executor=dask)

pipeline.add(normalize, X=train_set) \
        .add(kmeans.fit_predict, X=normalize) \
        .add(som.fit_predict, X=normalize) \
        .add(persist_kmeans, X=kmeans.fit_predict) \
        .add(persist_som, X=som.fit_predict) \
        .visualize()

pipeline.run()
```

Listing 3: Example DASF application defining a pipeline and operators

In summary, DASF emerges as a cutting-edge framework tailored for machine learning applications, particularly beneficial for handling large datasets and exploiting the full potential of modern computational resources like GPUs. Its user-friendly approach, coupled with its powerful and scalable architecture, makes it an attractive choice for both novice and experienced practitioners in the field of machine learning. It is a framework designed at the Discovery Lab at UNICAMP to facilitate the design and implementation of scalable machine learning code for large datasets, that scale well on high-performance computers.

More specifically, it offers an API (Application Programming Interface) that aims at abstracting away the complications introduced by designing code for distributed memory systems or GPUs, which are common requirements for scalability on high-performance computing (HPC) clusters.

In this way, users can design, debug, and explore the application on desktop-like computers using only a fraction of the large dataset and, finally, process the whole dataset on a large computing system with little or no code refactoring effort.

2.7.1 Dask

Dask [4] is a parallel computing library that allows the execution of data science operations in a distributed and parallel manner [13]. It is flexible and optimized for interactive and computationally intensive tasks. By breaking down large datasets into small pieces and processing them in parallel, Dask speeds up processing time and enables operations on datasets that exceed the system's RAM memory.

One of the main advantages of Dask is its easy integration with existing Python ecosystem libraries, such as Pandas and NumPy, allowing users to continue using the tools and techniques

they are familiar with, but with the additional capacity for scale and performance that Dask offers.

Dask's ability to execute data science operations in a distributed and parallel manner is facilitated by its use of a Dask cluster. This cluster is a flexible, scalable computing system that enhances Dask's efficiency in handling large-scale data processing and scientific computing tasks. It operates by breaking down large datasets into smaller chunks and processing them concurrently across multiple computational units, significantly accelerating processing time and enabling operations on datasets larger than the system's RAM. The main components of a Dask cluster include:

- Dask Scheduler: The heart of the Dask cluster. The scheduler manages task distribution and orchestrates the execution of tasks across Dask workers. It monitors their progress and schedules tasks to optimize resource usage and computation time.
- Dask Workers: These are the computational units of the cluster. Dask Workers execute the tasks assigned to them by the scheduler. Each worker can process multiple tasks simultaneously, depending on its resources (like CPU cores and memory).
- Client: This is the user-facing interface to the Dask cluster. Through the client, users submit tasks (in the form of Dask graphs) to the cluster. The client communicates with the scheduler, which in turn coordinates the execution of these tasks across the available workers.
- Task Graph: This is a data structure that represents the tasks to be executed. It is a directed acyclic graph where nodes represent computational tasks, and edges represent data dependencies between these tasks. The task graph is generated based on the computation the user wants to perform and the chunks extracted from the dataset.
- Dashboard: Dask provides a real-time web dashboard that offers valuable insights into the cluster's performance and resource utilization. This includes visualizations of task progress, memory usage, and other metrics.

Dask clusters can be set up in various environments, including single machines, local networks, and cloud platforms. They are highly scalable, allowing for the dynamic addition or removal of workers according to computational needs. This makes Dask clusters particularly useful for handling varying workloads and large datasets.

2.7.2 Scaling DASF Applications on HPC clusters

A **DASF executor** is the module responsible for executing a DASF pipeline. The default DASF executor is the "Local Executor", which executes the DASF pipeline locally, *i.e.*, on the same machine that is executing the DASF application. As discussed in the previous section, Dask enables large-scale computations to be distributed across multiple machines, enhancing the efficiency of applications, especially in HPC environments and Kubernetes clusters. The DASF "Dask Executor" allows the user to execute the DASF pipelines on HPC clusters, relying on a Dask scheduler and Dask workers. Figure 3 illustrates these concepts.

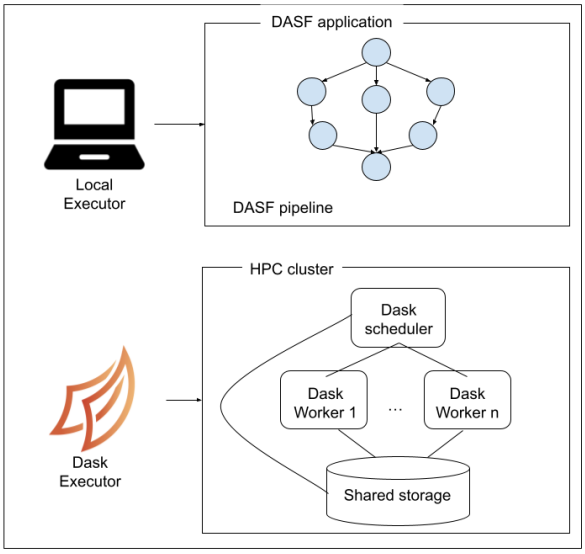

Figure 3: Execution of a DASF application on a local machine and on a HPC cluster.

HPC Clusters in Cloud Environments

Scaling DASF applications on High-Performance Computing (HPC) clusters, particularly in cloud environments, necessitates a strategic approach to leveraging the resources provided by cloud providers. These resources form the backbone of an HPC infrastructure that's optimized to meet specific computational requirements.

For the creation of these elements in a HPC Cluster, these are some of the essential components of Cloud-Based HPC Infrastructure:

- Virtual Machine Type and Image: Selecting the right virtual machine type and image is critical for performance alignment with the workload.
- Storage Solutions: Determining the appropriate storage capacity and type is vital for efficient data handling and accessibility.
- Network Configuration: Choosing suitable network adapters is essential for effective data communication within the cluster.
- Geographical Considerations: The location (region and availability zone) of these resources can impact factors like latency and data governance.

As discussed by Borin and Napoli [2], the management of these cloud resources can be approached in three primary ways:

- Web-Based GUIs: User-friendly interfaces, like the AWS Web Console, are straightforward for initial learning and deployment. However, they become less practical with increasing scale and complexity.

- Infrastructure as Code (IaC): IaC emerges as a superior method for managing larger or multiple infrastructures. It leverages automation tools for better organization, change tracking, error management, and reduction in overall management workload.
- Automation Tool Selection: The effectiveness of IaC heavily relies on the chosen automation tools, which should be evaluated based on usability, scalability, and error handling capabilities.

Kubernetes Cluster Execution

- DASF applications can also be executed on Kubernetes clusters using the Dask Kubernetes Operator.
- This setup allows Dask to dynamically scale up or down by adding or removing workers in response to the workload, all managed within a Kubernetes environment.
- The Dask operator on Kubernetes facilitates easy deployment and management of Dask clusters, making it an efficient choice for cloud-native environments.

In conclusion, DASF applications, thanks to their integration with Dask, offer flexibility in terms of deployment environments. They can run efficiently on a local machine, scale up to leverage HPC clusters, or even operate within a Kubernetes-managed cloud environment, making them versatile for various scientific computing needs.

2.8 Dask clusters on Kubernetes

In the dynamic realm of Kubernetes, the concept of an 'operator' plays a pivotal role, especially in deploying sophisticated applications like Dask clusters. The Dask Kubernetes Operator, a fine embodiment of this concept, stands as a testament to the evolving synergy between Kubernetes and complex data processing tasks.

At its heart, the operator comprises two fundamental elements: a data structure for articulating what needs to be deployed (in this case, a Dask cluster), and a controller that orchestrates the deployment itself. This duality of structure and action forms the essence of the operator.

In the world of Kubernetes, these data structures are known as Custom Resource Definitions (CRDs). They are the cornerstones of extending Kubernetes' capabilities, allowing us to introduce new, custom-designed resource types into the Kubernetes API landscape.

Each CRD is a blueprint, a declaration of intent, transforming our vision of a Dask environment into a tangible Kubernetes reality.

2.8.1 Dask Kubernetes Operator

The Dask Kubernetes Operator [6] is an innovative tool designed for managing Dask workloads within a Kubernetes environment.

However, deploying and managing Dask clusters in a dynamic and containerized environment like Kubernetes can be complex.

This is where the Dask Kubernetes Operator comes in. It acts as a bridge between Dask and Kubernetes, streamlining the deployment and management of Dask clusters. With the operator, users can easily launch Dask clusters, scale them up or down based on the workload, and handle their lifecycle within the Kubernetes ecosystem.

Key features of the Dask Kubernetes Operator include:

1. *Automated Cluster Management:* It automates the deployment of Dask clusters, managing their lifecycle and scaling them as per computational requirements.

2. *Custom Resource Definitions (CRDs):* The operator uses CRDs to define Dask clusters. Users can specify the size, configuration, and other parameters of their Dask clusters through these CRDs.

3. *Seamless Integration:* It integrates deeply with Kubernetes, allowing Dask to leverage Kubernetes' native features like auto-scaling, load balancing, and self-healing.

4. *Resource Optimization:* By intelligently scaling clusters, the operator ensures efficient use of resources, reducing costs and improving performance.

5. *User-Friendly Operations:* It simplifies complex operations, making it easier for users to deploy and manage Dask within Kubernetes without requiring in-depth knowledge of either Dask or Kubernetes internals.

In summary, the Dask Kubernetes Operator is a powerful tool for anyone looking to harness the power of Dask in a Kubernetes environment. It not only simplifies the deployment and scaling of Dask clusters but also optimizes resource utilization, making it an essential tool for efficient data processing and analysis in cloud-native ecosystems.

2.8.2 DaskJob

The DaskJob resource (CRD) in Kubernetes works in a way that's similar to other batch resources in Kubernetes. It's designed to complete a specific task or command. These are its main functionalities, as depicted in Figure 4:

- **Creates a Pod:** Just like other Kubernetes resources, DaskJob creates a Pod. This Pod is a small unit in Kubernetes that runs your task.

- **Runs a Command:** Inside this Pod, DaskJob executes the command you want to complete. This could be any data processing task you've set up.

- **Builds a Dask Cluster:** Alongside creating a Pod, DaskJob also sets up a DaskCluster. This cluster is a group of machines working together to process your data more efficiently.

- **Connects Pod to Cluster:** The job Pod is automatically configured to connect with this Dask cluster. This means the Pod can use the power of the entire cluster to complete its task faster and more efficiently.

- **Leverages Dask's Features:** By connecting to the Dask cluster, your job can handle larger datasets and more complex computations than a single Pod could manage on its own.

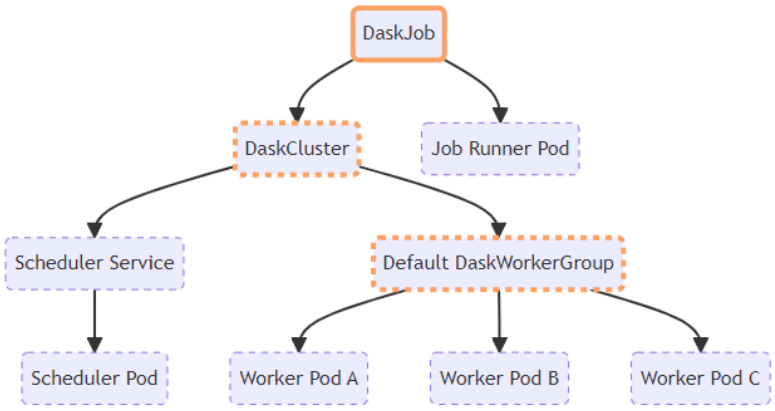

Figure 4: Structure of a DaskJob as show in the Custom Resources - Dask Documentation [5]

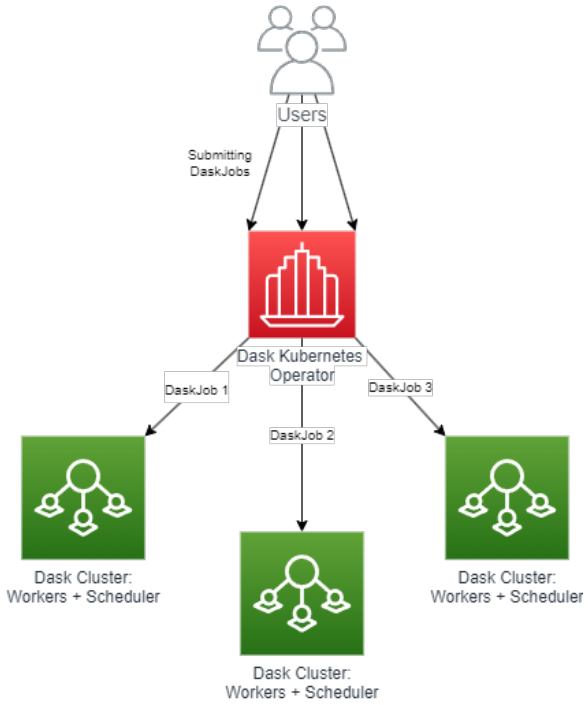

Figure 5: Example of a DaskJob submission to operator directly from the user

In essence, a DaskJob in Kubernetes not only executes your specified task but also smartly utilizes the capabilities of a Dask cluster to enhance the efficiency and scale of your

data processing tasks. It works as a stand alone component that can also receive tasks from users, as in Figure 5, although in our project we focus on its integration with workflows.

2.9 Storage module

An NFS (Network File System) storage system is ideal for environments where multiple clients need to read and write data simultaneously. It functions as a centralized file server that allows various workers or processes to access and modify the same files concurrently over a network.

NFS ensures data consistency and integrity through its file locking mechanism, even when multiple writes occur at the same time. This makes it especially useful in distributed computing contexts like ours were workers act concurrently computing values and saving checkpoints in-file, while maintaining efficiency and reducing data redundancy by storing information centrally.

3 Requirements and Choices

3.1 Requirements

The following functional requirements (FR) were defined:

- FR 1: The system must be able to implement a workflow process that uses Dask and DASF programs as operators;
- FR 2: The system needs to be capable of managing the execution of DASF tasks in multiple containers to ensure scalability;
- FR 3: The system must be able to distinguish between DASF and non-DASF tasks and treat them appropriately;
- FR 4: The system must be able to orchestrate the launch of multiple containers, including a scheduler container and multiple worker containers when a DASF task is executed for execution;
- FR 5: The system must be able to manage communication between containers, especially connecting workers to the scheduler when necessary;
- FR 6: The system must work with the Workflow retry mechanism to reuse intermediate results in multiple executions; and
- FR 7: The system must provide a Python API for new developers.

Besides the functional requirements, the following non-functional requirements (NFR) were defined:

Non-Functional Requirements (RNF):

- NFR 1: High availability and reliability;

- NFR 2: Optimized for quick execution and low computational cost;
- NFR 3: Security in deployment and execution processes; and
- NFR 4: The system should be resource-efficient, minimizing unnecessary container launches and ensuring proper destruction after task completion.

3.2 Choice of workflow framework

A workflow in this context, neat and concise,
Is a series of tasks, in order, precise.
It flows like a stream, from start to end,
With rules and paths on which it depends.
In each step it morphs, with data it dances,
Towards completion, it steadily advances.

When choosing the workflow framework for the project, the main frameworks studied were:

- **Argo Workflows:** A highly popular, Kubernetes-native engine known for its lightweight and scalable nature. It's designed for container environments and is cloud-agnostic, capable of running on any Kubernetes cluster.
- **KubeAdaptor:** This cloud-native engine integrates workflow systems with Kubernetes, offering flexible workflow definitions and an interface for optimized task execution sequences.
- **Kubeflow:** Focused on machine learning (ML) workflows, Kubeflow simplifies the deployment of ML tasks on Kubernetes, making them simple, portable, and scalable. It incorporates Argo Workflows for orchestrating these complex data science tasks.

After careful consideration, we have decided to utilize Argo for our workflow framework, specifically for building Dask workflows with the Custom Resource Definitions of the Dask jobs. This decision was primarily influenced by Argo's robust capabilities in deploying YAML CRDs for complete jobs. Argo's structure lends itself well to our requirements, offering both flexibility and efficiency in workflow management. Figure 6 shows an example of a Workflow with 7 tasks depicted by Argo Dashboard. The nodes indicate the tasks while the edges indicate the tasks dependencies. The green ticks indicate the successfully concluded tasks – all of them, in this example.

To complement our choice of Argo and enhance our Python-centric workflow, we have also decided to incorporate Hera's syntactic sugar. Hera's Python-friendly approach to defining and managing workflows aligns well with our team's expertise and our workflow's technical requirements, making it an invaluable addition to our toolkit.

In summary, Argo, complemented by Hera, provides the perfect blend of flexibility, power, and ease-of-use for our specific needs in building and managing workflows that contain DASF tasks. Its ability to handle complex task sequences with CRD and its superior Python integration make it the ideal choice for our project.

Figure 6: Example of a Workflow viewed in the Argo Dashboard.

4 The Argo-DASF Framework

As discussed in the previous section, DASF applications can easily migrate the execution of DASF pipelines from the local machine, *e.g.*, a laptop, to an HPC cluster by simply switching the DASF executor. For example, the DaskExecutor can be used to execute the DASF pipelines on a Dask cluster, composed of a Dask scheduler and multiple Dask workers.

Currently, the default way of executing DASF pipelines on high-performance computing cluster is to start a Dask scheduler and several Dask workers on the cluster and leverage the Dask backend to schedule the DASF application pipeline on the cluster. This process is usually performed manually, or by custom made scripts, designed for specific HPC clusters

This project aims to develop a framework to facilitate the creation of Kubernetes infrastructure for DASF applications, making it more accessible and scalable in cloud environments using infrastructure as code. More specifically, we are interested in scaling the performance of DASF applications that are embedded as Argo workflow tasks. To this end, we propose the Argo-DASF framework.

Figure 7 provides an overview of the Argo-DASF framework components.

The image shows a common interaction in this system:

1. The system admin creates the base infrastructure with the Argo/Dask operators and

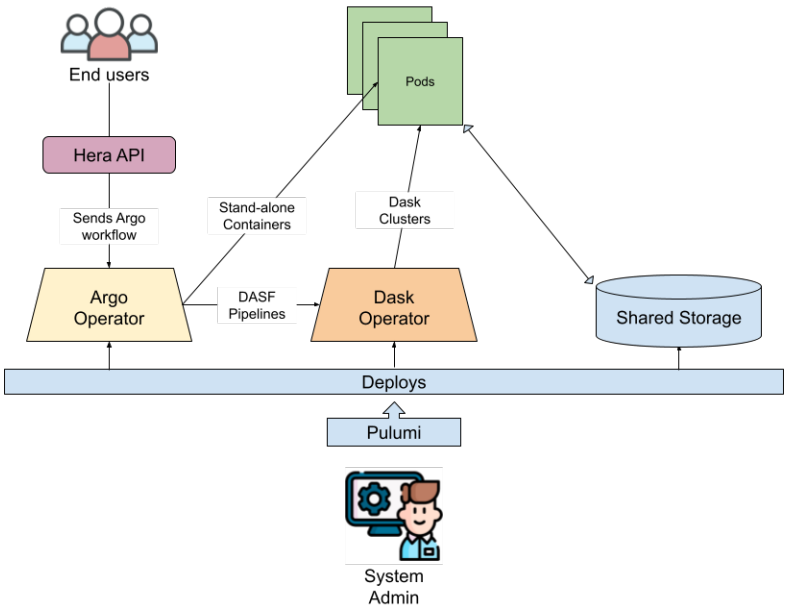

Figure 7: Argo-DASF overview.

the shared storage using Helm and Pulumi.

2. A user sends a workflow to be treated.

3. The argo operator creates containers to run commands and sends pipelines to the Dask Operator.

4. The Dask operator creates clusters that runs the pipelines using the shared storage.

The following sections present the Argo-DASF framework components.

4.1 Argo-DASF overview

The Argo-DASF framework is composed of 3 main components: The Argo Operator, the Dask Kubernetes Operator and an NFS storage system, it was inspired by a talk by Severin Ryberg [14] that talked about the power of Dask and Argo together.

Figure 8 illustrates how these components are used to scale the execution of DASF tasks[2] on Argo.

There are several ways of using the system. For example:

1. User submits a workflow through the Argo operator:

[2]DASF tasks are DASF applications embedded as Argo tasks.

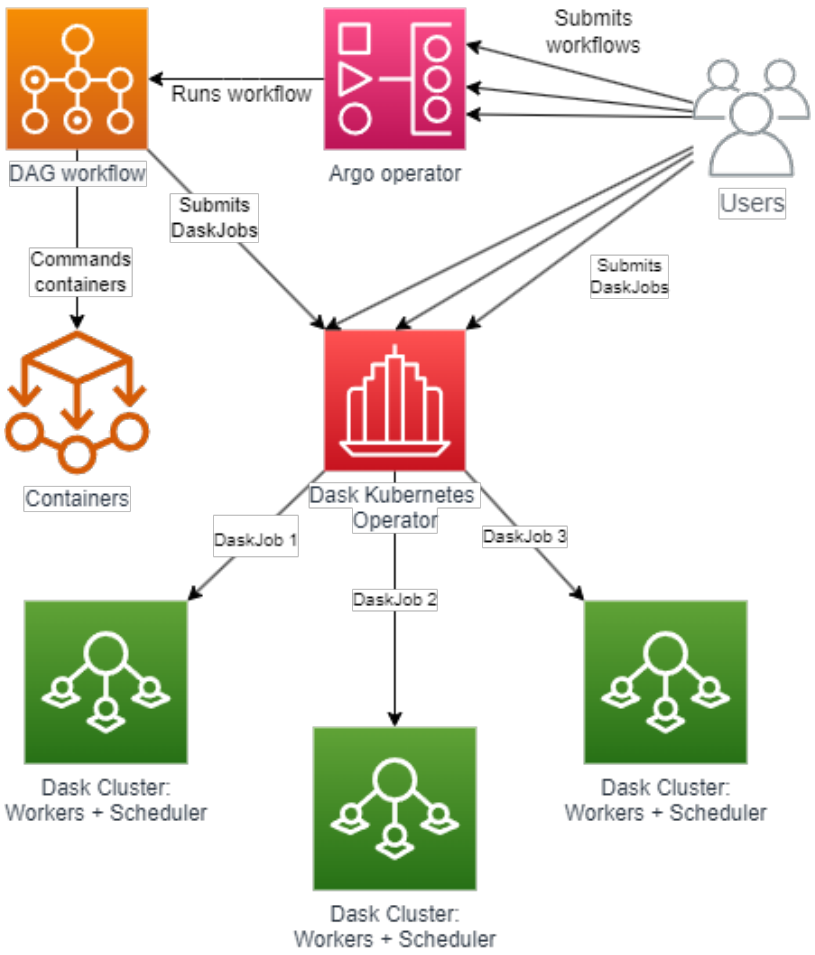

Figure 8: Scaling the execution of DASF tasks with Argo-DASF.

- The Argo operator sends a request to the Kubernetes API server to create a new workflow.
- The Kubernetes API server creates a new resource definition for the workflow and updates the namespace with the new resource, calling each step in the workflow in order.
- The Dask Operator watches for changes to the namespace and detects the creation of the new workflow resource that calls uppon it.
- The Dask Operator creates a new Dask job based on the workflow configuration and creates a Dask cluster for each job, with a set of controller, scheduler and workers.

2. Dask scheduler schedules the Dask job:

   - The Dask scheduler receives the job submission from the Dask Operator and

checks the availability of workers.

- If there are available workers, the scheduler assigns the job to one or more workers and starts executing the job.
- Otherwise, the scheduler waits for a worker to become available and then assigns the job to the worker.

3. Dask worker executes the Dask job:

   - The Dask worker receives the job assignment from the scheduler and starts executing the job.
   - The worker processes the data and produces output, which can be received and stored in a persistent volume claim (PVC).

4. Argo Workflows polls the status of the workflow:

   - Argo Workflows periodically polls the Kubernetes API server to check the status of the workflow.
   - When the workflow completes successfully, Argo Workflows retrieves the output from each step and passes it to the next step in the workflow as parameters.

5. User accesses the results of the workflow:

   - The user can access the results of the workflow through the Argo UI or by querying the Kubernetes API server directly.
   - These interactions enable the system to perform complex data processing tasks in a scalable and efficient manner, while providing a straightforward way for users to interact with the system and monitor the progress of their workflows.

4.1.1 Creating Dask Clusters

In addition to using Argo Workflows, users can directly interact with the Dask Operator from their local environment to submit Dask jobs and create Dask clusters. This can be useful for initial data exploration or for users who want to persist the cluster for continued study of the dataset. To do this, users can utilize one of the Pulumi stacks, which does not rely on the Argo Operator. Instead, users can write Python code to provision and manage their Dask clusters from Python.

For example, Listing 4 illustrates how to create a Dask Cluster in the Dask Operator programmatically with a Python application.

```
def create_dasf_cluster(
    name="cluster-spec-name",
    namespace=namespace,
    adapt=False,
    minimum=1,
    maximum=2,
    replicas=1,
    use_gpu=False,
```

```
    image_dask="dasf:cpu",
    **kwargs,
):
    print(image_dask)
    spec = make_cluster_spec(name=name, n_workers=replicas, image=image_dask
    )
    spec = adapt_worker_spec(spec, use_gpu=use_gpu)
    cluster = KubeCluster(
        namespace=namespace, custom_cluster_spec=spec, resource_timeout=300
    )
    # use adapt to scale the cluster up and down automatically, or scale to
    a fixed number of workers
    if adapt:
        cluster.adapt(minimum=minimum, maximum=maximum)
    else:
        cluster.scale(replicas)
    return cluster

def adapt_worker_spec(spec, use_gpu=False):
    # Modify the worker pod spec to use a GPU-enabled image and request GPU
    resources
    if use_gpu:
        spec["spec"]["worker"]["spec"]["containers"][0][
            "image"
        ] = "nvcr.io/nvidia/rapidsai/rapidsai-core:23.04-cuda11.8-runtime-
    ubuntu22.04-py3.10"
        spec["spec"]["worker"]["spec"]["containers"][0]["args"] = [
            "dask-cuda-worker",
            "$(DASK_SCHEDULER_ADDRESS)",
            "--rmm-pool-size",
            "10GB",
        ]
        spec["spec"]["worker"]["spec"]["containers"][0]["resources"] = {}
        spec["spec"]["worker"]["spec"]["containers"][0]["resources"]["limits
    "] = {}
        spec["spec"]["worker"]["spec"]["containers"][0]["resources"]["
    requests"] = {}
        spec["spec"]["worker"]["spec"]["containers"][0]["resources"]["limits
    "][
            "nvidia.com/gpu"
        ] = 1
        spec["spec"]["worker"]["spec"]["containers"][0]["resources"]["
    requests"][
            "nvidia.com/gpu"
        ] = 1
    return spec

# Create a Dask cluster
cluster = create_dasf_cluster(
    use_gpu=False,
    name=cluster_name,
    namespace=namespace,
    image_dask=image_dask,
    replicas=3,
```

```
)

dask = DaskPipelineExecutor(cluster=cluster, use_gpu=False)
```

Listing 4: How to create a DASF cluster in the Dask Operator programatically

This provides a lightweight and flexible way for users to interact with their Dask clusters, without requiring the overhead of a fully-fledged workflow manager like Argo for simpler tasks. This can be called from a python script or a Jupyter notebook with access to the Kubernetes configuration.

4.1.2 Creating DaskJobs from Argo

To bring DaskJobs into our Python workflow submission, we create a CRD in YMAL with a template mechanism that allows us to pass parameters to it from a Python script. For the DaskJob using CPU we declare it as illustrated in Listing 5.

```
apiVersion: kubernetes.dask.org/v1
kind: DaskJob
metadata:
  generateName: dask-jobs-
  namespace: {{namespace}}
spec:
  job:
    spec:
      serviceAccountName: {{serviceAccountName}}
      containers:
        - name: job
          image:  "{{image}}"
          imagePullPolicy: "IfNotPresent"
          args: {{formatted_args}}
          volumeMounts:
            - name: shared
              mountPath: /shared
      volumes:
        - name: shared
          persistentVolumeClaim:
            claimName: {{nfsname}}

  cluster:
    spec:
      worker:
        replicas: 2
        spec:
          containers:
            - name: worker
              image: "{{image}}"
              imagePullPolicy: "IfNotPresent"
              args:
                - dask-worker
                - --name
                - $(DASK_WORKER_NAME)
                - --dashboard
```

```
              - --dashboard-address
              - "8788"
            ports:
              - name: http-dashboard
                containerPort: 8788
                protocol: TCP
            env:
              - name: WORKER_ENV
                value: hello-world # We dont test the value, just the name
    scheduler:
      spec:
        containers:
          - name: scheduler
            image: "{{image}}"
            imagePullPolicy: "IfNotPresent"
            args:
              - dask-scheduler
            ports:
              - name: tcp-comm
                containerPort: 8786
                protocol: TCP
              - name: http-dashboard
                containerPort: 8787
                protocol: TCP
            readinessProbe:
              httpGet:
                port: http-dashboard
                path: /health
              initialDelaySeconds: 5
              periodSeconds: 10
            livenessProbe:
              httpGet:
                port: http-dashboard
                path: /health
              initialDelaySeconds: 15
              periodSeconds: 20
            env:
              - name: SCHEDULER_ENV
                value: hello-world
        service:
          type: ClusterIP
          selector:
            dask.org/cluster-name: simple-job
            dask.org/component: scheduler
          ports:
            - name: tcp-comm
              protocol: TCP
              port: 8786
              targetPort: "tcp-comm"
            - name: http-dashboard
              protocol: TCP
              port: 8787
```

```
89                targetPort: "http-dashboard"
```

Listing 5: YMAL template file with parameters for itegration with python and enable reusable configs

Where the "{{items}}" are parameters used in the job creation.

Note: if the workflow is going to run parallel jobs to the same namespace, the ports declared in the CRD must be different between the 2 parallel jobs so that the schedulers and workers do not overlap and crash. Or run 2 jobs on the same scheduler and workers with adaptive scaling.

4.1.3 RBAC roles and service accounts

RBAC (Role-Based Access Control) in Kubernetes (K8s) is a security mechanism for managing access and giving permissions to resources. It involves defining Roles and ClusterRoles to specify permissions for resources. RoleBindings and ClusterRoleBindings are then used to assign these roles to users, groups, or service accounts within specific namespaces or across the entire cluster, ensuring secure and controlled access.

To create clusters and CRDs defined in Dask, you need special permissions. These permissions are assigned to a service account (a user in the Kubernetes cluster) through a role. This setup ensures that only the service account with the correct role has the authority to create and manage these resources in your Kubernetes environment.

From the line in the previous section were we define:

$$serviceAccountName : \{\{serviceAccountName\}\}$$

We pass the name of the service account created in Pulumi defined in Listing 6, which will give the permission for a resource within the Kubernetes cluster to create other resources.

This is a characteristic of Kubernetes that enforces that no resource within the K8 cluster can create something without the proper authorization, this is done with the intent of guaranteeing security within the cluster.

```
export const createDaskServiceAccount = (
  namespace: k8s.core.v1.Namespace
): k8s.core.v1.ServiceAccount => {
  const daskRole = new k8s.rbac.v1.Role('dask-role', {
    metadata: {
      namespace: namespace.metadata.name
    },
    rules: [
      {
        apiGroups: ['kubernetes.dask.org'],
        resources: [
          'daskclusters',
          'daskworkergroups',
          'daskworkergroups/scale',
          'daskjobs',
          'daskautoscalers'
        ],
        verbs: ['get', 'list', 'watch', 'patch', 'create', 'delete']
```

```
      },
      {
        apiGroups: [''], // indicates the core API group
        resources: [
          'pods',
          'pods/status',
          'pods/log',
          'services',
          'poddisruptionbudgets'
        ],
        verbs: ['get', 'list', 'watch', 'create', 'delete']
      }
    ]
  });
```

Listing 6: Creating service account name for Argo Workflows and DaskJobs using Pulumi

4.2 Kubernetes modifications for GPU

Kubernetes is a powerful platform for data processing, offering significant advantages in scalability and reproducibility. For our case in particular, our main aim was to integrate DASF to this Kubernetes ecosystem. DASF is, at its core, a framework that uses Dask and RAPIDS.ai to make data pipelines using GPUs that handle data processing of huge datasets concurrently. For this, we use the native Kubernetes creation of Dask Workers that uses NVIDIA GPUs so that the environment can run these efficient workflows.

To enable the use of GPU on a Kubernetes environment that has access to NVIDIA GPUs, because dask-Kubernetes uses standard Kubernetes pod specifications, we can use Kubernetes device plugins and add resource limits defining the number of GPUs per pod/worker. Additionally, we can also use tools like dask-cuda for optimized Dask/GPU interactions, like shown in the Listing 7.

```
kind: Pod
metadata:
  labels:
    foo: bar
spec:
  restartPolicy: Never
  containers:
  - image: nvcr.io/nvidia/rapidsai/rapidsai-core:23.04-cuda11.8-runtime-
    ubuntu22.04-py3.10
    imagePullPolicy: IfNotPresent
    args: [dask-cuda-worker, $(DASK_SCHEDULER_ADDRESS), --rmm-pool-size, 10
    GB]
    name: dask-cuda
    resources:
      limits:
        cpu: "2"
        memory: 6G
        nvidia.com/gpu: 1 # requesting 1 GPU
      requests:
        cpu: "2"
```

```
        memory: 6G
        nvidia.com/gpu: 1 # requesting 1 GPU
```

Listing 7: Kubernetes Pod for a Dask Worker with the changes necessary for GPU integration

4.3 Argo workflows and Hera

The integration of Argo Workflows with Hera in our project represents a leap forward in orchestrating complex data processing tasks within a Kubernetes environment. The Python code in Listing 8 illustrates how we leverage these powerful tools to create and manage workflows efficiently.

```
from  hera.workflows import (
    Workflow,
    DAG,
    Parameter,
    Container,
    Resource,
    NFSVolume,
    models,
)
from .BaseConfig import BaseConfig
from  hera.shared import global_config
from .Utility import get_manifest

# define  argo configs
global_config.host = "http://127.0.0.1:2746"
global_config.token = ""

config = BaseConfig(
    namespace="example-namespace",
    service_account_name="example-sa",
    base_image="example-image",
)

with Workflow(
    generate_name="dag-target-",
    entrypoint="dag-target",
    namespace="dasf-da121b4a",
    arguments=Parameter(name="target", value="E"),
    volumes=[
        models.Volume(
            name="shared", persistent_volume_claim={"claim_name": "many-nfs-pvc"}
        )
    ],
) as w:
    # we can define containers that run commands
    sendData = Container(
        name="send-data",
        image="senddata:cpu",
        command=["sh", "-c"],
        args=["cp -r /f3.zarr /shared/"],
```

```
        volume_mounts=[
            models.VolumeMount(name="shared", mount_path="/shared"),
        ],
    )
    # as well as CRDs for daskjob
    dask_job = Resource(
        name="dask-job",
        action="create",
        service_account_name=config.service_account_name,
        volume_mounts=[
            models.VolumeMount(name="shared", mount_path="/shared"),
        ],
        manifest=get_manifest(
            namespace=config.namespace,
            image=config.base_image,
            serviceAccountName=config.service_account_name,
            container_args=[
                "python",
                "attribute_extractor.py",
                "-x",
                "128",
                "-y",
                "128",
                "-z",
                "-1",
                "-a",
                "envelope",
                "-r",
                "cpu",
                "-d",
                "/shared/f3.zarr",
                "-o",
                "/shared/output.zarr",
            ],
        ),
    )

    with DAG(
        name="dag-target",
    ):
        A = sendData(name="A")
        B = dask_job(name="B")

        # then we can define the graph dependency
        A >> B

w.create()
```

Listing 8: Python workflow example using the DaskJob template within the Hera API

Here the *get_ manifest* function is an abstraction created to facilitate the translation of CRDs into python parameterized YMAL code (defined previously), as such we can define templates for DaskJob for both CPU and GPU code, by only changing the image with the code responsible for the transformation, the correct YMAL template and the command that

triggers the action required.

Through this setup, we efficiently manage data processing tasks, leveraging the capabilities of both Argo Workflows and Dask within a Kubernetes ecosystem. This approach ensures scalable, reproducible, and well-orchestrated data processing workflows.

4.4 NFS storage system

The NFS system is designed to provide a flexible and scalable file storage solution for Kubernetes applications. It uses the OpenEBS Helm chart to deploy an NFS server and client, allowing multiple pods to share the same NFS volume.

At a high level, here's how the system works within our IaC code:

1. The createNFSVolume function creates a new Persistent Volume Claim (PVC) with the desired size and storage class. In this case, we use the openebs-kernel-nfs storage class, which is configured to use the OpenEBS NFS server.

2. The createNFSVolume function then deploys a Helm chart for the OpenEBS NFS server, using the nfs-provisioner chart. This chart installs the NFS server and its dependencies, including the OpenEBS driver.

3. The createNFSVolume function also deploys a second Helm chart for the Local PV Provisioner, using the localpv-provisioner chart. This chart provisions a local Persistent Volume (PV) that will be used to store the NFS data.

4. The testNfsPvc variable is created to represent the PVC that will be used to store the NFS data. This PVC is requested by the testNfsPod pod when it starts up.

5. The testNfsPod pod is created with a single container that runs the busybox image and sleeps for 3000 seconds. This pod serves as a placeholder to demonstrate how a pod can consume the NFS volume.

6. To scale the NFS system, you can simply increase the number of replicas of the testNfsPod pod, which will automatically request more PVCs and provision additional NFS volumes as needed.

In a production environment, this setup could be replaced with other distributed file systems like FSx for Lustre or EFS (in AWS), depending on the specific requirements of the application. However, for smaller to medium-sized clusters, this NFS system can provide a simple and cost-effective solution for shared file storage. As always, scaling considerations should be taken into account when designing a Kubernetes storage solution.

4.5 Infrastructure as Code

In the evolving landscape of cloud infrastructure, Infrastructure as Code (IaC) has emerged as a pivotal paradigm, and Pulumi stands at the forefront of this revolution, especially in the Kubernetes community. Pulumi, a modern IaC tool, transforms how we build, deploy, and manage cloud infrastructure by using familiar programming languages. Central to Pulumi's

power is the concept of 'stacks' – a mechanism that not only revolutionizes infrastructure deployment but also enhances its scalability, manageability, and reproducibility.

4.5.1 Functionality of Pulumi Stacks

Pulumi stacks are akin to individual, isolated instances of our infrastructure. Each stack represents a unique configuration state, making it possible to deploy multiple environments (like development, staging, or production) from the same code base. This multiplicity allows us to pass variables and manage configurations seamlessly across different environments, offering an unmatched level of flexibility and control. By leveraging different stacks, we can tailor the infrastructure to specific requirements or contexts without altering the underlying code, a testament to the adaptability and sophistication of Pulumi.

4.5.2 Modularization through Stacks

Our project harnesses the full potential of Pulumi stacks by dividing the infrastructure code into three main parts: 'Dask', 'Argo', and 'nfsVolume'. This modular approach underlines our commitment to a clean, organized, and functional infrastructure codebase.

- **Dask Stack:** This stack is the backbone of our Kubernetes environment, responsible for setting up and managing the Dask Kubernetes Operator. It lays the groundwork for deploying and orchestrating Dask clusters, ensuring a smooth and efficient integration within our Kubernetes ecosystem.
- **Argo Stack:** The Argo stack focuses on the deployment and configuration of a Argo Workflows endpoint. This segment of our infrastructure is crucial for managing complex workflows, enabling us to automate and streamline our data processing tasks with precision and reliability.
- **NFS Volume Stack:** The NFS Volume stack is dedicated to creating and managing network file system (NFS) volumes. These volumes are essential for shared data storage and accessibility across different pods and nodes, with simultaneous read/write, facilitating seamless data exchange and storage in our distributed environment.

Each stack segment is meticulously crafted, with variables defined in the respective Pulumi stack YAML files dictating the specific aspects of the infrastructure to be created. This methodical division allows us to not only maintain clarity and organization within our code but also to instantiate specific infrastructure components as needed, based on the variables provided. Listing 9 illustrates how to define a stack that uses all three parts.

```
config:
  dasf-iac:dasfImage: dasf:cpu
  dasf-iac:kubeconfig: 'false'
  dasf-iac:namespace: 'dasf'
  dasf-iac:portDasf: "8788"
  dasf-iac:portScheduler: "8786"
  dasf-iac:portWorker: "8787"
  dasf-iac:replicas: "2"
```

```
dasf-iac:dask: true
dasf-iac:argo: true
dasf-iac:nfsVolume: true
kubernetes:context: minikube
```

Listing 9: Example stack configuration in Pulumi to select the parts to use

For interacting with these stacks, here is a quick guide:

- **Selecting a Stack:** Use *pulumi stack select [stack-name]* to switch to the desired stack.
- **Creating a Stack:** Create a new stack with *pulumi stack init [new-stack-name]*.
- **Deploying Resources:** Deploy stack resources using *pulumi up*, which applies the desired state defined in your code to your stack.
- **Destroying Resources:** To remove all resources in a stack, use *pulumi destroy*. This command dismantles all the resources managed by the stack.
- **Deleting a Stack:** Finally, remove the stack itself with *pulumi stack rm [stack-name]*. Be cautious, as this is irreversible.

In essence, our use of Pulumi stacks use the fusion of modular coding practices with the robust capabilities of IaC, resulting in a powerful, scalable, and versatile infrastructure that is tailored to our project's unique demands.

5 Validation and insights

The project was executed in both a local and scalable environment. Locally, it utilized Minikube v1.31.2 with Docker v24.0.5 (linux/amd64), a lightweight Kubernetes implementation that creates a VM on the local machine as a Docker image containing a Kubernetes cluster. Minikube is ideal for development and testing scenarios, offering an accessible and efficient way to work with Kubernetes without complex infrastructure.

For scalability testing, Amazon Elastic Kubernetes Service (EKS) was used, with Kubernetes version 1.28. In EKS, we created a managed node group and integrated the cluster autoscaling controller, allowing for dynamic adjustment of resources based on demand. This setup enabled the project to be tested under realistic, scalable conditions, simulating actual deployment environments, minus deeper security compliance configurations.

In the local environment, specific resource allocations were made for optimal performance. The Kubernetes cluster's specifications were as follows:

- **CPU Allocation:** 5 CPUs, ensuring sufficient processing power for concurrent tasks without overburdening the host system.
- **GPU Allocation:** 2 GPUs, crucial for tasks requiring high computational capabilities, like data processing and machine learning workflows.

This configuration balanced performance with resource utilization, suitable for the project's limitations. For the nodes in the EKS, we used t3.medium instances for our workers and general infrastructure, for a more fine-grained control of scaling and worker size for different use cases, different managed node groups can be used in together.

Data for the project was sourced from the Unicamp Discovery Research Group, using two .zarr format datasets - one at 80 MB and another at 3 GB. The .zarr format is efficient for storing chunked, compressed, multidimensional arrays, ideal for handling large-scale data in Kubernetes environments.

These datasets were pivotal in the project, providing essential information for system integration and testing. The size variation of the datasets allowed assessment of the system's performance under different data loads in both the local Minikube and scalable EKS environments.

5.1 Experimental Setup and Methodology

The validation involved two primary methods:

1. **Argo Workflow with DaskJob:** This method entailed running the envelope data transformation within an Argo workflow, using a DaskJob that uses a DASF application that computes the envelope attribute for a seismic data. It represents a scenario of automated, large-scale data processing.

2. **Programmatic Dask Cluster Creation:** The second method involved programmatically creating a Dask cluster within the Dask operator, executed from a local PC through a Python script. This setup reflects a user-centric approach, allowing direct interaction and testing of data transformations of DASF pipelines before scaling to a system like item 1.

5.2 DASF adaptation for testing

To test our framework in conjunction with DASF, a few changes to the traditional DASF framework had to be made. The following Dockerfile snippets detail the setup of our DASF environment, crucial for ensuring compatibility with the Dask Kubernetes Operator and maintaining reproducibility through version-controlled requirements.

5.2.1 Base Dask Image Setup

The first Dockerfile establishes the base environment for our Dask deployment. The contents of this file is shown in Listing 10.

```
1 FROM ghcr.io/dask/dask:2023.10.0
2
3 ENV DEBIAN_FRONTEND=noninteractive
4 RUN echo "tzdata tzdata/Areas select Europe" | debconf-set-selections
5 RUN echo "tzdata tzdata/Zones/Europe select London" | debconf-set-selections
6
7
```

```
RUN apt update -y
RUN apt install -y --no-install-recommends \
        g++ \
        gcc \
        git \
        graphviz \
        openssh-client \
        python3-dev \
        xdg-utils \
        python3-pip && \
    rm -rf /var/lib/apt/lists/*

RUN pip3 install pip --upgrade

COPY dasf /dasf
COPY requirements.txt /requirements.txt
COPY README.md README.md

RUN pip3 install --no-cache-dir -r requirements.txt

WORKDIR /
```

Listing 10: Dockerfile for building a DASF image

In this Dockerfile, we start with the official Dask image ghcr.io/dask/dask:2023.10.0 to ensure compatibility with the Dask Kubernetes Operator. The environment is set to non-interactive to streamline the installation process. Timezone settings are configured for Europe/London, followed by updating and installing essential packages like g++, gcc, git, and others. We then upgrade pip and copy the necessary files (DASF, requirements.txt, README.md) into the container. The pip3 install command ensures that all dependencies are installed as per the requirements.txt file, and the –no-cache-dir option is used to keep the Docker image size minimal.

5.2.2 Seismic Lite Setup

The second Dockerfile extends the base image for seismic data processing. The contents of this file is shown in Listing 11.

```
FROM dasf:cpu

# install the seismic stuff
COPY dasf_seismic /dasf_seismic
COPY requirements-seismic.txt /requirements-seismic.txt
RUN pip3 install --no-cache-dir -r requirements-seismic.txt

# add the code to run
COPY attribute_extractor.py /attribute_extractor.py

WORKDIR /
```

Listing 11: DASF seismic dockerfile based on the DASF CPU image

This Dockerfile builds on the base DASF image, tagged `dasf:cpu`. It focuses on installing specific packages for seismic data processing. Required seismic-related files and the `requirements-seismic.txt` are copied into the container, and the necessary Python packages are installed using pip3. Again, the `-no-cache-dir` option is used to minimize the Docker image size. Finally, the `attribute_extractor.py` script, the file that runs the test data pipeline is added.

Together, these Dockerfiles provide a comprehensive and efficient setup for our DASF-based data processing environment, ensuring compatibility between schedulers and workers, reproducibility, and optimized image sizes.

5.2.3 DASF modifications

To ensure that the DASF project could be integrated with the Kubernetes system some changes to the code had to be made. These changes are shown in Listing 12.

```
class DaskPipelineExecutor(Executor):
    """
    A pipeline engine based on dask data flow.

    Keyword arguments:
    address -- address of the Dask scheduler (default None).
    port -- port of the Dask scheduler (default 8786).
    local -- kicks off a new local Dask cluster (default False).
    use_gpu -- in conjunction with ‘local‘, it kicks off a local CUDA Dask
                cluster (default False).
    profiler -- sets a Dask profiler.
    protocol -- sets the Dask protocol (default TCP)
    gpu_allocator -- sets which is the memory allocator for GPU (default
    cupy).
    cluster_kwargs -- extra Dask parameters like memory, processes, etc.
    client_kwargs -- extra Client parameters.
    cluster -- a Dask cluster object.
    daskjob -- boolean to indicate if the executor is used in a DaskJob.
    """

    def __init__(
        self,
        address=None,
        port=8786,
        local=False,
        use_gpu=False,
        profiler=None,
        protocol=None,
        gpu_allocator="cupy",
        cluster_kwargs=None,
        client_kwargs=None,
        cluster=None,
        daskjob=False,
    ):
        self.address = address
        self.port = port

```

```
37          if not cluster_kwargs:
38              cluster_kwargs = dict()
39
40          if not client_kwargs:
41              client_kwargs = dict()
42
43          # If address is not set, consider local
44          local = local or (
45              address is None
46              and "scheduler_file" not in client_kwargs
47              and cluster is None
48          )
49
50          if daskjob:
51              self.client = Client()
52          elif cluster is not None:
53              self.client = Client(cluster)
```

Listing 12: Modifications made to the executor in the DASF executor to integrate with the Kubernetes operator

In the `dasf/pipeline/executors/dask.py` file we had to add a configuration update where we could pass the cluster itself to the Dask pipeline client or simply call the client if it is part of a DaskJob CRD (Lines 50-53 in Listing 12). This was specially made so that we could use the pipeline with a Dask operator called from a python client with access to the namespace of our Kubernetes deployment.

5.3 Code source and testing

In this section we will go over the necessary steps to run these tests and how to deploy yourself the infrastructure plus the workflow submission. The code can be found at:

https://github.com/discovery-unicamp/dasf-argo

5.3.1 Local Environment

The necessary steps to run the minikube local environment consists of :

1. Install minikube, pulumi and a container environment like Docker Desktop.
2. Install kubectl and set your local environment to use minikube.
3. Go inside the **iac**/ directory, from the root of the project, that contains the Pulumi Typescript code.
4. Run **pulumi up** to build the infrastructure after selecting the type of stack you wish to use locally.
5. Change into the **workflow_api**/ directory (from the root of the project) to access the workflows created.

6. Verify the configuration of the testing file **simple_example.py** and run it with **python simple_example.py**.

This process will begin the workflow that sets up the environment and that runs the DaskJob containing a DASF pipeline.

5.3.2 AWS Environment

The deployment structure of our EKS cluster[7] using Pulumi is organized into four distinct modules. Each module within our stack is responsible for creating specific resources in the cloud. These modules output values that are utilized by subsequent modules, ensuring a cohesive and efficient deployment process. This structure includes the following parts, as shown by Figure 9 :

1. **Cluster Module:** This module is responsible for creating a Virtual Private Cloud (VPC) and an Elastic Kubernetes Service (EKS) cluster. It also sets up a managed node group that will be utilized by the pods and controllers within our system. The VPC provides an isolated network environment for our EKS cluster, enhancing security and control with private and public subnets secured by NAT Gateways and Internet Gateways.

2. **Autoscaler Module:** It deploys the AWS Cluster Autoscaler controller. This controller dynamically adjusts the number of nodes in the cluster based on current needs, ensuring efficient resource utilization.

3. **Argo-DASF Module:** This module deploys the Argo-DASF framework within the cluster. It also creates an Elastic Container Registry (ECR) repository for storing Docker images and an NFS for shared data storage among pods. ECR provides a secure, scalable container image registry, while NFS offers a simple file storage for use with AWS Cloud services.

4. **Services Module:** This final module sets up the AWS Load Balancer Controller using Helm. This controller enables the creation of Application Load Balancers (ALB) to facilitate external user access to our framework in a secure manner with native Kubernetes ingresses. ALBs offer high availability, automatic scaling, and robust security features necessary for modern web-facing applications.

Each module is meticulously designed to ensure seamless integration and optimal performance of the overall EKS cluster deployment, tailored to meet the specific needs of our project.

To run the EKS cluster containing the project the process consists of:

1. Install AWS CLI and configure it with **aws configure** using the credentials from an IAM user with sufficient permissions for cloud provisioning.

2. Install pulumi.

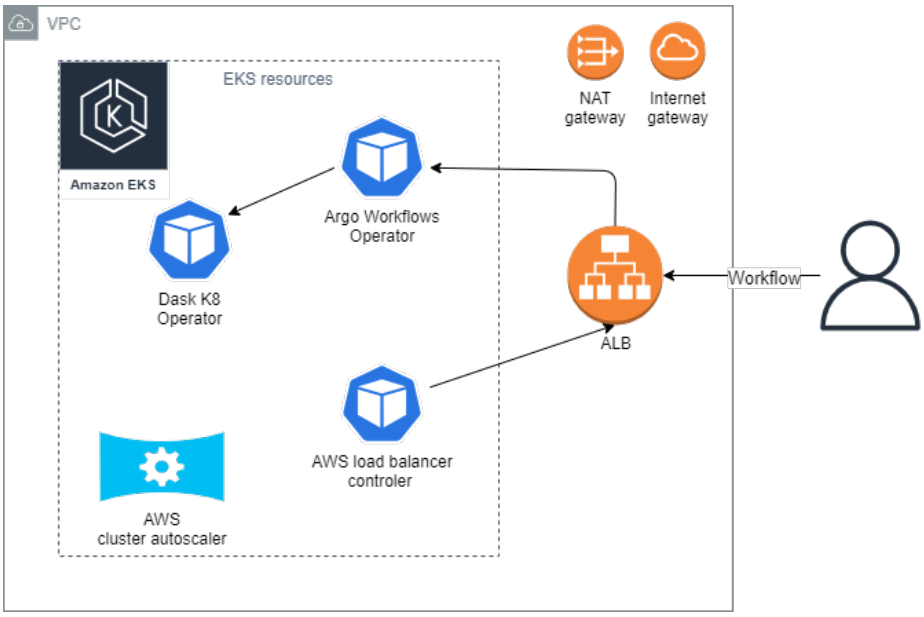

Figure 9: Overall structure of cloud infrastructure for testing

3. Go inside the **iac-eks**/ directory, from the root of the project, that contains the Pulumi Typescript code.

4. For each of the stack modules (cluster > autoscaler > argo-dasf > service, in order), run **pulumi up** to build the infrastructure and use/create the dev stack, this will build out the development infrastructure we created.

5. Change into the **workflow_api**/ directory (from the root of the project) to access the workflows created.

6. Verify the configuration of the testing file **simple_example_eks.py** and modify it to use the public DNS of the Application Load Balancer created by the infrastructure as code at **global_config.host** as well as the base configuration defined within, the alb url and the configurations will be outputted after the creation of the autoscaler, services and argo-dasf modules.

7. Run it with **python simple_example_eks.py**.

8. Enter the load balance url to verify the workflow progression within the argo workflows dashboard.

5.4 Results and Insights

The experimental outcomes demonstrated the framework's efficacy in executing the envelope data transformation across both approaches by succeeding in applying the transformation and dimensional changes expected from the transformation. Key insights include:

- The Argo workflow with DaskJob effectively managed to simplify the deployment DASF pipeline, confirming the framework's capability in orchestrating complex data workflows with less effort required from the end user. This approach is particularly suited for predefined and validated DASF pipelines, offering a distributed and cost-effective solution for large data sets. This insight was achieved from successfully running a simple python script containing the Listing 7, that created the workflow and ran the contained pipeline with a verified output file in the NFS storage.

- The programmatically created Dask cluster provided an accessible platform for individual users. It was run using the Listing 4, which created the Dask Cluster in the Kubernetes cluster and ran a pipeline like Listing 3. This Python script was called from a Jupyter notebook, as a testing method, similar to how data scientists work for quickly testing a theory, but with the power of the cluster behind the system.

  The data pipeline was viewed using the Dask Dashboard created with the Cluster and the data verified against the output data.

  This method is ideal for testing and verifying single DASF pipelines, offering simplicity and flexibility for exploratory data analysis and transformation tasks.

- The implementation that was built in EKS on AWS was a great show of the scaling capacity of our systems. With the addition of K8 cluster autoscalers to the EKS clusters we were able to see that our systems scaled as needed by growing the number of pods, and nodes when the number of pods reached a maximum.

  Regarding the speedup of our system from implementing this is a scalable manner in EKS, we saw that the time reduction was not too substantial in our test case. This was due to the fast nature of our test case, so the overall time decrease was minimal, the sum of the time that it took to initialize the pod was usually larger than that of the Daskjob itself, as show in Table 1, where for 5 workers we saw that the system performed a horizontal scaling of the number of instances/nodes rather than that of pods, which is a slower process and this had a bigger effect in the overall time than the task itself (usually took 3 to 4 seconds).

  About the NFS volume created with openEBS on the EKS system, when too many pods were mounted on the system at the same time (more than 10 pods, equivalent to 6 workers), the persistent volume claim created would crash by not being able to handle such a large number of concurrent accesses. This showed a shortcoming regarding the nfs stack and proved that implementing a cloud managed file sharing solution like EFS or Lustre is better for production environments.

5.5 Conclusions

The validation of Argo-DASF highlighted its versatility and power in processing seismic data. It proved capable of supporting automated workflows and individual, interactive data processing tasks, with a simplified workflow creation. This dual capability ensures that the framework can cater to diverse operational needs, from complex data analysis to individual

Number of workers	Time till completion (s)
1	15
1	18
2	16
3	14
4	14
5	30

Table 1: Time of completion for treating the zarr object, given the number of workers in the DaskJob.

research and development efforts, thus affirming its value in the realm of data processing and analysis.

The adaptation of the solution to EKS clusters and the use of the Kubernetes Cluster Autoscaler, proved the capability of our systems at being a scalable solution by horizontally scaling with nodes and pods to fit our workflows submitted to the Load Balancer that targeted our Argo workflows controller.

6 Project Development

This section outlines the development of the project, detailing the journey from requirement gathering to the final review and submission. We focus on selecting and implementing frameworks like Kubernetes and Pulumi, overcoming challenges in scalability and integration, and ensuring the system's correctness and efficiency, delving deeper into the specifics of framework implementation and the key challenges faced during this project.

1. **Requirements Gathering**: Focused on identifying the project's specific needs including data processing capabilities, scalability, and storage specifications.

2. **Survey and Selection of Frameworks**: Evaluated various frameworks like Kubeflow, Dask Distributed, *etc.*, based on performance, scalability, ease of integration, and community support.

3. **Framework Implementation**: Set up the Kubernetes environment, installed and configured Argo and Dask operators, created the NFS storage system, and addressed challenges with versioning and kubectl commands in Pulumi. More details concerning this are described in Section 6.1.

4. **Correctness and Scalability Testing**: Conducted tests to verify data workflow accuracy and system performance under different workloads, with limitations due to CPU and GPU constraints in my environment.

5. **Report Review and Submission**: Reviewed the report for accuracy and clarity, incorporating feedback from colleagues and experts, including Prof. Edson Borin and Alan Souza.

6.1 Framework Implementation

In the preliminary phase, we dedicated ourselves to hands-on experimentation with Kubernetes, utilizing kubectl commands made with the help of "The Kubernetes Book" [11]. This practical engagement was crucial for gaining a real-world grasp of Kubernetes functionalities and limitations.

Transitioning from theory to application, we embarked on constructing a robust infrastructure using Pulumi. This phase was characterized by the translation of kubectl commands into Pulumi scripts. The rationale behind this translation process was two-fold: Firstly, to ensure a deep-rooted understanding of the underlying mechanisms of Kubernetes orchestration and secondly, to establish a framework that ensures reproducibility and scalability.

This transition was not merely a change in tools but a strategic shift towards infrastructure as code (IaC), an approach that guarantees consistency, version control, and a high degree of automation. By leveraging Pulumi, we were able to encapsulate the complexity of Kubernetes commands into reusable, scalable, and maintainable code structures.

Finally, we integrated the Hera clients and the parameterized templates that we submit to the framework itself.

6.2 Main Challenges Through the Process

During the project development, several challenges were encountered:

- **Versioning and Configuration**: Updating the Dask image construction mechanism for effective communication in a Kubernetes environment.
- **Selection of Dask Deployment**: Determining the optimal approach for the Dask creation mechanism, which involved navigating through multiple deployment options.
- **Integration Complexities**: Ensuring seamless integration of various technologies like EKS, Hera, DASF, Kubernetes, Argo, Dask, and NFS into a cohesive system.

These challenges were instrumental in shaping the project's direction and contributed significantly to the learning and development process.

7 Conclusions

The main focus of the project was to develop an efficient system for managing complex and parallelizable data pipelines with DASF tasks. The designed system is organized in three main modules:

- NFS Volume: responsible for simultaneous reading and writing operations from multiple sources, vital for parallel processing in Dask pipelines.

- Argo Operator: responsible for managing and executing workflows in Kubernetes, simplifying task orchestration.
- Dask Operator: responsible for executing data processing pipelines in Kubernetes, optimizing parallel and distributed processing.

The system was also designed so dask pipelines can be executed in Ephemeral Workflows. In this context, Dask pipelines are implemented as jobs within Argo's ephemeral workflows, providing detailed control over each stage of the data process.

Moreover, the system leveraged the Python Hera client for workflow management, which facilitated the definition and exectuion of workflows, abstracting the complexity of Argo's YAML templates and allowing a greater focus on the logic of data pipelines.

Figure 10 shows the dependency in levels between each component of the project, where the elements are built using a bottom up approach.

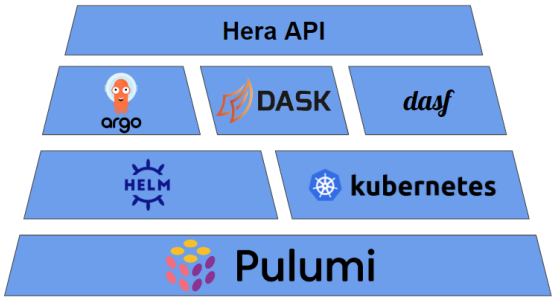

Figure 10: Project Dependency Structure

Throughout our comprehensive exploration of how to deploy DASF frameworks to Kubernetes, we have delved into various facets of deploying and managing cloud-native applications using Kubernetes, with a particular focus on data processing workflows using Dask and Argo Workflows.

- **Dask Kubernetes Operator:** We explored the Dask Kubernetes Operator in detail, highlighting its role in simplifying the deployment of Dask clusters within Kubernetes. This operator, with its use of Custom Resource Definitions (CRDs), exemplifies how Kubernetes can be extended to manage complex, data-intensive tasks efficiently.
- **Workflow Orchestration with Argo and Hera:** The integration of Argo Workflows, augmented by Hera, demonstrated how complex data processing tasks could be orchestrated and automated within a Kubernetes environment. This setup not only enhanced the efficiency of our workflows but also underscored the scalability and flexibility offered by Kubernetes in handling sophisticated workflows.
- **Infrastructure as Code with Pulumi:** The adoption of Pulumi for Infrastructure as Code (IaC) was a significant stride in our project. By segmenting our infrastructure into modular stacks - 'operator', 'Argo', and 'nfsVolume' - we achieved a high degree

of organization and clarity. This modularization, coupled with the ability to select, create, deploy, and delete stacks, showcased the agility and control we maintained over our infrastructure.

- **Pulumi Lifecycle Management:** The lifecycle management of Pulumi stacks – from creation to deletion – was instrumental in ensuring that our infrastructure deployment was both dynamic and responsive to our project's needs. The use of pulumi commands brought about a smooth workflow for deploying and dismantling infrastructure components as required.

In conclusion, this project stands as a testament to the synergy between Kubernetes, Dask, Argo Workflows, and Pulumi. Each component played a critical role in creating a robust, scalable, and efficient environment for data processing and workflow management. The lessons learned and the methodologies applied here not only achieved the project's immediate goals but also laid down a blueprint for future endeavors in the realm of cloud-native application development and management. This journey has been a vivid demonstration of the power and versatility of modern cloud technologies when harnessed with expertise and foresight.

7.1 Future work

As we look ahead, there are several areas of potential development and improvement that can further enhance the capabilities and efficiency of our system.

1. Enhanced Security Protocols: Continuous improvement of security measures is vital. Future work could involve integrating advanced security protocols and encryption methods to safeguard data and operations against emerging threats.

2. Cross-Platform Compatibility: Expanding the system's testing with various cloud providers and platforms would make it more versatile and user-friendly, catering to a broader range of use cases and environments.

3. User Interface and Experience Improvements: Developing a more intuitive and user-friendly interface will enhance the accessibility and usability of the system, making it more approachable for a wider range of users.

4. Test at max scale: Fully test a HPC environment with our configuration, *e.g.*, in AWS use cluster placement groups for its instances, EFA acceleration and Lustre for shared storage with high data dimensionality and complexity

5. Setup observability: The operators come with integration with Prometheus, so creating monitoring Dashboards with Grafana could come in handy to avoid problems down the development line.

These areas represent a roadmap for future enhancements, ensuring that the system remains cutting-edge, user-centric, and adaptable to the evolving needs of its users.

8 Acknowledgements

As this project reaches its end, I am filled with deep gratitude towards those who have contributed to its success and have supported me throughout this journey.

First and foremost, I extend my heartfelt thanks to my advisor, Edson Borin. His trust in entrusting me with such a fascinating and challenging project has been a great honor. His expertise and insight have been invaluable in guiding me through the complexities of this endeavor. His mentorship has not only shaped this project but also my professional growth and understanding.

Special thanks are due to Alan Souza, whose guidance was crucial in navigating the intricacies of NFS and Argo workflows. His knowledge and hands-on advice were instrumental in overcoming some of the most challenging aspects of the project.

I am also immensely grateful to the Unicamp Discovery Research Group. Their creation of such a vital open-source project and their decision to welcome me into their fold have been pivotal in the success of this work. Collaborating with them has been an enriching experience, allowing me to contribute to and learn from a project of significant value and impact.

Additionally, I must express my profound gratitude to the Dask community. The extensive and well-organized documentation provided by them has been a cornerstone in the successful deployment phase of this project. The clarity and depth of their documentation have greatly simplified complex concepts and processes, making it an invaluable resource.

Furthermore, the assistance provided by members of the Dask forum deserves special mention. Their prompt and insightful responses to queries I made and the supportive environment they foster have significantly eased the deployment challenges. The community's willingness to share knowledge and offer practical solutions has not only aided in the technical aspects of the project but also has been a source of encouragement and inspiration.

Last but certainly not least, I would like to express my deepest appreciation to my family and friends. Their unwavering emotional support and various other forms of assistance have been my backbone throughout this demanding period of balancing work and this project. Their belief in me and their encouragement have made my life more bearable and joyful in the face of challenges.

In summary, this journey, while filled with technical learning and professional development, has also been a testament to the power of collaboration, mentor-ship, and personal support. To everyone who has been a part of this journey, I am eternally thankful.

Bibliography

[1] *Argo Workflows*. https://argoproj.github.io/argo-workflows/. Argo Workflows is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. Argo Workflows is implemented as a Kubernetes CRD (Custom Resource Definition).

[2] Edson Borin et al. *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*. 1st ed. Computer Science, Computer Science (R0) eBook Packages. Springer Cham, 2023. ISBN: 978-3-031-29769-4. URL: https://doi.org/10.1007/978-3-031-29769-4.

[3] *Dasf core*. https://github.com/discovery-unicamp/dasf-core. Framework for computing Machine Learning algorithms in Python using Dask and RAPIDS AI.

[4] *Dask docs*. https://docs.dask.org/en/stable/. Dask is a flexible library for parallel computing in Python.

[5] *Dask Documentation - Custom resources*. https://kubernetes.dask.org/en/latest/operator_resources.html.

[6] *Dask Operator*. https://docs.dask.org/en/stable/deploying-kubernetes.html. The Dask Kubernetes Operator is a set of Custom Resource Definitions (CRDs) and a controller that allows you to create and manage your Dask clusters as native Kubernetes resources.

[7] *EKS Best Practices Guides*. https://aws.github.io/aws-eks-best-practices/.

[8] *Helm*. https://helm.sh/. The package manager for Kubernetes.

[9] *Hera*. https://hera.readthedocs.io/en/stable/. Hera is a Python framework for constructing and submitting Argo Workflows.

[10] *Kubernetes*. https://kubernetes.io/. Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

[11] N. Poulton and P. Joglekar. *The Kubernetes Book*. Nigel Poulton, 2019. URL: https://books.google.com.br/books?id=nN07zQEACAAJ.

[12] *Pulumi*. https://pulumi.com/. Open Source Infrastructure as Code.

[13] Matthew Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: Jan. 2015, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013.

[14] *Scheduled Scaling with Dask and Argo workflows*. https://dok.community/blog/scheduled-scaling-with-dask-and-argo-workflows/. This talk discusses the combination of these two worlds by showcasing a set-up for Argo-managed workflows which schedule and automatically scale-out Dask-powered data pipelines in Python.

[15] *Workflow definition*. https://datascientest.com/en/workflow-definition-and-advantages/.