Locality-Aware Workflow Orchestration for Big Data

Andrei-Alin Corodescu
University of Oslo
Oslo, Norway
andreaic@ifi.uio.no

Nikolay Nikolov
SINTEF AS
Oslo, Norway
nikolay.nikolov@sintef.no

Akif Quddus Khan
Norwegian University of Science and Technology
Gjøvik, Norway
akif.q.khan@ntnu.no

Ahmet Soylu
Oslo Metropolitan University
Oslo, Norway
ahmet.soylu@oslomet.no

Mihhail Matskin
KTH Royal Institute of Technology
Stockholm, Sweden
misha@kth.se

Amir H. Payberah
KTH Royal Institute of Technology
Stockholm, Sweden
payberah@kth.se

Dumitru Roman
SINTEF AS
Oslo, Norway
dumitru.roman@sintef.no

ABSTRACT

The development of the Edge computing paradigm shifts data processing from centralised infrastructures to heterogeneous and geographically distributed infrastructure. Such a paradigm requires data processing solutions that consider data locality in order to reduce the performance penalties from data transfers between remote (in network terms) data centres. However, existing Big Data processing solutions have limited support for handling data locality and are inefficient in processing small and frequent events specific to Edge environments. This paper proposes a novel architecture and a proof-of-concept implementation for software container-centric Big Data workflow orchestration that puts data locality at the forefront. Our solution considers any available data locality information by default, leverages long-lived containers to execute workflow steps, and handles the interaction with different data sources through containers. We compare our system with Argo workflow and show significant performance improvements in terms of speed of execution for processing units of data using our data locality aware Big Data workflow approach.

CCS CONCEPTS

• **Information systems** → *Computing platforms*; **Data management systems**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Big Data workflows; data locality; software containers

MEDES '21, November 1–3, 2021, Virtual Event, Tunisia

ACM ISBN 978-1-4503-8314-1/21/11…$15.00
https://doi.org/10.1145/3444757.3485106

ACM Reference Format:
Andrei-Alin Corodescu, Nikolay Nikolov, Akif Quddus Khan, Ahmet Soylu, Mihhail Matskin, Amir H. Payberah, and Dumitru Roman. 2021. Locality-Aware Workflow Orchestration for Big Data. In *International Conference on Management of Digital EcoSystems (MEDES '21), November 1–3, 2021, Virtual Event, Tunisia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3444757.3485106

1 INTRODUCTION

In recent years, harnessing large data sets from various sources has become a pillar of rapid innovation for many domains such as marketing, finance, agriculture, and healthcare [3]. The Big Data domain has evolved rapidly, and new challenges have arisen at different levels of the technological stack, from the complex business logic to the infrastructure required to process the ever-increasing volume, velocity, and variety of data. Working with Big Data is a complex process involving collaboration among a wide range of specialisations (such as distributed systems, data science, and business domain expertise) [4, 5, 21]. Handling such complexity naturally comes with an increased cost, and the value extracted from the data must offset this cost.

Big Data workflows formalise and automate the processes that data goes through to produce value by providing necessary abstractions for workflow definitions and efficiently leveraging underlying hardware resources. Big Data workflows usually integrate various data sets and leverage different programming languages or technologies to process data. Therefore, a desirable feature of a Big Data workflow system is to orchestrate workflows in a technology-agnostic fashion, both in terms of data integration and processing logic. On the other hand, approaches based on software containers emerged to create and execute workflows using processing steps in line with these considerations. While it is beneficial to leverage software containers to better separate concerns in a Big Data workflow system, higher-level abstractions come with a performance penalty; thus, it becomes more relevant to ensure the system performs as efficiently as possible.

Traditionally, Cloud service providers have been the standard solution for working with Big Data. However, Cloud services are

inherently centralised in a small number of locations (i.e., data centres) worldwide. Moreover, with the advent of the Internet of Things (IoT), significant amounts of data are generated at Edge networks [10]. With the data processing happening on geographically distributed systems across Edge and Cloud resources, it is crucial to reduce the delay and cost of transferring data over the network. Transferring massive data to the Cloud is an expensive task that may incur latency, which makes low-latency scenarios unfeasible. To address these issues, the Edge computing paradigm [9] aims to complement Cloud computing by leveraging hardware resources situated closer to the Edge of the network to offload processing, reducing transfer cost, and satisfying the latency requirements. However, existing solutions are mainly designed for Cloud-only workloads, making them unsuitable or inefficient for workloads spanning both the Cloud and edge.

To this end, we propose a novel architecture for container-centric Big Data workflow orchestration systems that makes containerised Big Data workflow systems more efficient on Cloud and Edge resources. Our proposed approach considers data locality, leverages long-lived containers (i.e., containers whose life-cycle is not tied to a particular data unit) to execute workflow steps, and handles the interaction with different data sources through containers. We provide an evaluation of the proposed approach by comparing it with a similar existing solution, Argo workflow, through an experiment to demonstrate the benefits of our approach. Through the experiment, we show that by considering data locality, our proposed system significantly improves the performance in terms of data processing speed (up to five times better).

The rest of the paper is structured as follows. Section 2 provides the relevant background, while Section 3 discusses the related work. Section 4 presents our approach and Section 5 describes its implementation. Finally Section 6 provides an evaluation, while Section 7 concludes the paper.

2 BACKGROUND

The high velocity of the data, combined with the large volume, mandates the processing to happen efficiently and cost-effectively to produce value that outweighs the costs. Therefore, execution time and bandwidth usage are two indicators that are often measured in Big Data systems and determine the feasibility of a solution. In this respect, in what follows, we introduce the essential techniques and concepts used in this study to reduce the execution time and bandwidth usage for Big Data workflows.

2.1 Data Locality

Data locality refers to the approach of moving computation closer to the data, which is typically cheaper and faster than moving data closer to the computation. The nature of working with Big Data mandates the resources (network, memory, CPU, disk) of multiple machines to be pooled together in a distributed system. A desirable characteristic of distributed systems is to hide the complexities of the distributed resources behind a single interface, such that the entire system appears as a single entity (e.g., Cloud storage systems such as Amazon S3). However, it makes it more challenging to leverage individual hosts backing the distributed systems. For example, a fundamental invariant of the current computer architectures is that a CPU can only work with data present in the memory of the same machine.

Consequently, data movement across machines becomes an integral part of any Big Data system. Besides traditional communication protocols that rely on the operating system network stack (e.g., TCP/IP-based protocols), latency is critical for many use cases. To this end, more efficient protocols have emerged. For example, RDMA (Remote Direct Memory Access) [17] is a protocol that allows the transfer of data stored in the memory of one machine to another without involving the CPU or the operating system kernel through specialised network cards.

As the quantity of data is significant, the network traffic and the associated latency of transferring data between machines can influence the overall cost and performance. Even for solutions targeted at centralised deployment (such as Cloud deployments), data locality has proven to be effective in reducing the cost and execution times [8, 16]. For example, Apache Spark [18] leverages the information provided by the Hadoop File System (HDFS) and knowledge about outputs of previous work to minimise the data transfer.

One of the core motivations of the Edge computing paradigm is reducing the amount of data transferred from the Edge of the network to the Cloud and supporting lower latency scenarios, making data locality a primary concern for any Edge computing solution. However, data locality is only one aspect that can be taken into account when scheduling work. Other aspects such as load distribution and heterogeneity of the available resources on different nodes need to be balanced together with data locality to perform the work effectively [14]. There exist studies proposing advanced scheduling strategies to balance the reduction of data transfer with load distribution (e.g., [6, 19]).

2.2 Inter-component Communication Optimisation

Separation of concerns and delegating responsibilities to different components have numerous benefits; however, the communication between components may introduce other performance and efficiency overhead due to message serialisation and transfer through potentially slow mediums. For example, what was performed as a simple method invocation in a monolithic solution can be turned into a REST API call for a solution where components are separated.

The choice of communication protocols directly impacts performance and bandwidth utilisation as different protocols provide different guarantees related to data transmission (e.g., TCP ensures ordered, lossless transmission but requires multiple round trips to establish connections and exchange data, while UDP is faster but also unreliable). Different protocols introduce additional overhead by injecting required data (e.g., HTTP headers). Techniques such as compression and binary serialisation help reduce the size of the payload. There exist studies exploring the possibility of using RDMA-backed memory channels to support fast and efficient inter-container communication (e.g., [1]). Apart from the bandwidth utilisation and speed of a particular protocol, the contract defining the communication between two entities (message format, content, and semantics) plays a significant role in facilitating the integration between components. Defining and enforcing a contract for the communication between the components allows for

the implementations of the components to be decoupled from one another.

2.3 Lifecycle Management of Containers

Software containers present themselves as a lightweight virtualisation alternative solution to traditional hypervisor-based virtualisation; however, there is still a cost associated with starting up and shutting down containers as needed. Life-cycle management is one of the aspects considered to have an impact on workflow execution time and it is suggested that the fastest option is re-using containers to process multiple units of data [20] (i.e., long-lived containers).

For many use cases, the execution time of the work delegated to a particular container is high, thus making the overhead of instantiating containers negligible. However, especially in Edge computing environments, the available resources are limited and the amount of data processed at a given point in time on a single host is reduced. Furthermore, with data being constantly processed in small parts (or even streamed), there is a need for this processing to happen as quickly as possible to achieve the desirable throughput. In such cases, the overhead of setting up and tearing down containers can quickly add up and become a significant bottleneck for the performance of the solution.

2.4 Integration with Data Management Solutions

One of the pillars of Big Data processing is being able to reason over and process heterogeneous data sets together in a unified manner. These data sets can be stored using different technologies and the interaction with these technology requires complex logic in itself. Thus, Big Data workflow systems should facilitate easy integration with different data management solutions, such as databases, file systems, Cloud storage, and web services.

3 RELATED WORK

In this section, we provide a review of the existing solutions according to the following criteria: (i) ability to incorporate data locality in the orchestration process, (ii) support for container lifecycle management, and (iii) the ease of integration with any data management solution.

- Snakemake [13] is a workflow orchestration tool that supports wrapping individual steps in containers, and different data solutions can be integrated into workflows by extending the Snakemake codebase. However, there is no support for controlling where the computation happens (data locality).
- Kubeflow[1] is a workflow orchestration tool oriented at machine learning-related workflows. The only storage supported is Minio (a Cloud-native, open-source alternative to the S3 Cloud storage). It offers no support for data locality.
- Makeflow [2] is a workflow orchestration tool able to orchestrate workflows on a wide variety of infrastructures. However, it does not have any built-in support for different data management systems or data locality features.
- Pachyderm[2] is another machine learning workflow orchestration solution, but the only supported storage system is the Pachyderm file system, a distributed file system built to offer additional functionalities to Pachyderm.
- Pegasus[3] is a workflow orchestration solution that supports containerised steps and leverages the location of the processed files to schedule the steps on the same host. However, its data management is limited to file systems.
- Airflow[4] is one of the most popular data workflow orchestrators and supports the execution of the workflows on a Kubernetes cluster. It is also possible to control where instances of steps are created. However, this needs to be set when the workflow is defined, making it unable to capture dynamically changing requirements. Integration with different data management solutions is possible by extending the Airflow code with providers, thus limiting it to Python implementations only.
- Argo workflows[5] is a workflow orchestration solution natively built on Kubernetes and supports data locality through a set of mechanisms. Similar to the Airflow solution, different data management solutions can be integrated, but require changes and integration with Argo code libraries.

All the considered solutions leverage short-lived containers as part of the orchestration - a container is created to execute work and destroyed as soon as the processing completes as these solutions target mostly batch processing scenarios. In terms of data locality specification, Argo offers the most expressive feature, as it leverages the full feature set offered by Kubernetes. However, by default, the limitation of having to specify data locality at workflow (introduced with the analysis of Airflow) definition time applies to Argo as well. Argo offers a mechanism through which respective outputs of processing steps can be used to modify the parameters (for data locality in this case) of subsequent steps in the workflow, allowing for dynamic data locality configurations at run-time. However, such an approach would require additional logic to be injected into the processing step. Pegasus, although limited in terms of data locality features, does handle data locality implicitly, without the need to modify the workflow definition. In contrast, for both Argo and Airflow, while offering more expressive data locality features, the workflow definition has to capture these details, thus breaking the separation of concerns principle.

4 PROPOSED SOLUTION

We propose an approach based on a workflow system architecture covering the run-time considerations of Big Data workflows based on the separation of concerns principle. The proposed architecture has three main layers:

(1) **Control layer:** it is responsible for the execution of workflows concerning their definitions (e.g., correct step sequencing and data being processed). The main component of the control layer is the *orchestrator*.

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

[2] https://github.com/pachyderm/pachyderm
[3] https://pegasus.isi.edu
[4] https://airflow.apache.org
[5] https://argoproj.github.io/projects/argo

(2) **Data layer:** it collectively refers to all the components involved in data handling (e.g., storage and retrieval of data, and moving data between hosts to make it available to compute steps that require it). The layer includes the *data store* component, referring to the technology used to store data (e.g., distributed file system and Cloud storage) and the *data adapter* component, serving as an interface between the data store and the other components in the workflow.
(3) **Compute layer:** it refers to the processing logic contained in the steps used in the workflow. The compute layer is composed of multiple *compute steps*, and by sequencing them together, they form a workflow. It is also possible that multiple instances of the same compute step type run in parallel.

The components of the different layers can be spread across multiple hosts, and the orchestrator serves as the coordinator of the centralised architecture. The use of a centralised architecture is motivated by leveraging data locality when executing Big Data workflows requires knowledge about the entire system (e.g., component physical placement) and the data that flows through it (the physical location where data is stored). The centralised architecture greatly simplifies the acquisition, management, and usage of this information. Data are organised into discrete, indivisible, and independent units when passing through the framework. These represent the units of work at both an orchestration level and individual step level. Each unit is processed independently of the others, and multiple units can be processed in parallel across different steps. This model, where the execution of the workflow is handled at the data unit level, can provide significant performance benefits as opposed to models where steps are executed synchronously (all outputs of the previous step have to be available to start the next one) [7, 15].

Whenever a new unit of data is available in the system, the orchestrator is notified, and it passes on the notification to a computation step to be processed. The compute step may produce one or more outputs from the input, each being a new data unit that continues to flow through the system independently of the others. Organising the work in independent units enables tasks to be distributed across all the available resources, allowing the proposed solution to scale horizontally (increase the processing power by adding more hosts in the distributed system). Figure 1 depicts a high-level overview of the three layers and their interactions. The following sections present each layer in greater detail.

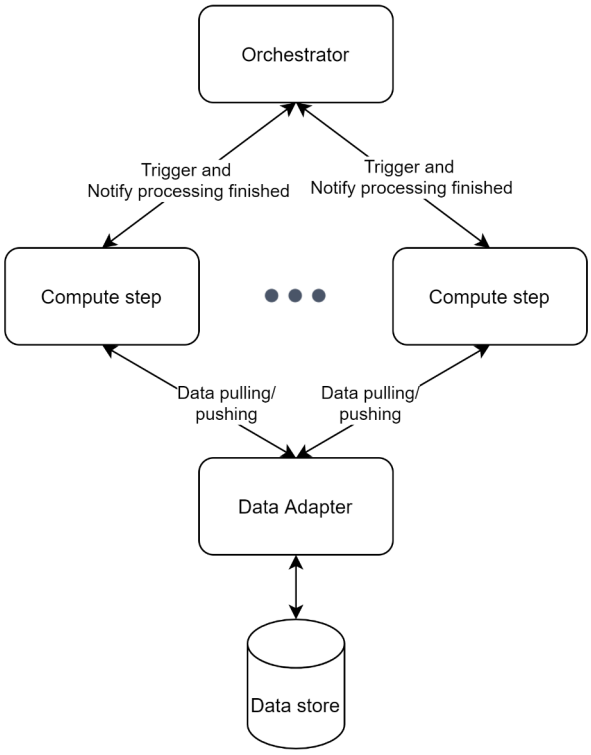

Figure 1: High level overview of the design components.

4.1 Control Layer

Upon receiving a notification indicating new data are available to be processed, the orchestrator needs to determine what type of compute step needs to be invoked for the current data unit, according to the workflow specification. For example, in a workflow consisting of three sequential steps, the data units outputted by the second step need to be passed to an instance of the third step.

Throughout the system, there can be multiple instances of the same compute step type. The orchestrator is responsible for choosing one of the instances to perform the processing of the data. Instead of relying on traditional load balancing algorithms (such as round-robin, random, and least connections), the orchestrator needs to employ a custom routing decision algorithm that takes the number of variables as input. The orchestrator decides about the routing based on available information, such as data locality, current load, varying resource availability, cost, and existing policies. This list is not exhaustive and presents only a subset of potential aspects that can be considered when making a routing decision. Optimising for cost, performance while ensuring all the explicit requirements of the workflow are met makes the routing problem a complex, multi-variable optimisation problem, with no clear best decision. A trade-off in one or more areas is necessary.

The inputs to the routing decision need to be acquired and made available by the module responsible for the decision. Depending on the volume and velocity of the units of data flowing through the orchestrator, the calculation of the routing decision needs to be efficient to avoid spending a significant amount of time routing each unit. This means that some of the inputs may have to be precalculated or estimated asynchronously, as acquiring information about all the possible targets synchronously may incur a significant performance cost.

4.2 Data Layer

The ability to share data between the nodes hosting the compute step instances is a fundamental requirement of the proposed solution approach. This allows the steps to pass data from one another as part of the workflow execution. All the interactions with the data storage happen through the data adapter in the proposed architecture, serving as an intermediary between the data storage technology and the orchestration components. The data adapter

model hides the complexities and particularities of interacting with the data storage by separating the logic into a dedicated component that exposes a simplified interface for external communications.

Separating data handling concerns from the compute step creates a modular architecture that facilitates and encourages the re-usability of both components in multiple workflows. For example, a compute step processing images can be re-used to process images from multiple data sources, and the same data solution can be re-used with different compute steps. In addition to re-usability, separating the compute steps from the data adapter allows for more flexibility in terms of technologies chosen to implement either of them. For example, a compute step can be implemented in Python while the data adapter can be written in Java, and the two components can still communicate.

Apart from interacting with the underlying storage, the data adapter is also responsible for providing the other components in the system with information about the physical localisation of the data handling to support locality-aware work scheduling. The data locality model employed to communicate the localisation information needs the following characteristics:

(1) It needs to apply to both the data units flowing through the system and the compute step instances, as the information it captures is used to route data units to be processed on compute step instances in proximity.
(2) The localisation information needs to apply to resources throughout the computing continuum, and a distance measure needs to be determinable for any two localisations.
(3) The data locality model needs to be granular enough to capture host-level information.
(4) As different data storage solutions have different capabilities of exposing information about where data is stored, the data locality model should support reserved values to indicate parts of the information are missing.

4.3 Compute Layer

The compute steps follow a simple execution model since both the orchestration and data handling logic being handled by external components:

(1) Compute steps are served a unit of data as input (provided by the data adapter).
(2) The processing logic is applied in the compute step.
(3) The processing logic can produce one or more outputs, picked up by a data adapter and resulting in notifications for the orchestrator.

This architecture gives complete freedom to execute any logic that can be run in containers as the implementation of the compute step has no restrictions over what the processing logic can do with the input data. Neither the input nor the output data are typed.

4.4 Extension Model

The current architecture opts for the container-based extension model as it aligns better with the architecture requirements. The container-based extension model becomes apparent when diving deeper into the architecture of a compute step (Figure 2). A compute step is logically composed of a framework agent and processing logic, running in a separate container. The framework agent container is responsible for coordinating the execution in the context of a single step (retrieving the input data, triggering the processing logic, handling the output data). It effectively hides the complexities related to the orchestration and acts as the intermediary between the data and compute components, thus allowing them to have simplified interfaces they need to adhere to, dedicated only to their function (handling data or processing it), as described:

(1) The framework agent needs to accept requests from the orchestrator to process a unit of data.
(2) Based on the instructions received from the orchestrator, it reaches out to the data adapter to retrieve the input data.
(3) Once the input data is accessible to the container hosting the processing logic, the agent needs to send a request to trigger the computation.
(4) The output data is passed to the data adapter, and the orchestrator is notified that new data has become available.

By leveraging the container-based extension model, the microservice inspired architecture combines the code contributed by the user (i.e., data adapter, business logic, and data store) and the framework provided components (i.e., orchestrator and framework agent) to allow the definition and execution of customisable workflows. By injecting two components implementing simple interfaces (one for data handling and one for processing logic), the framework can orchestrate the execution workflows composed of steps implemented in different programming languages or technologies and leverage different data storage solutions. The data handling and processing logic are completely separated from one another, allowing the two components to be created and evolve independently, thus helping with the separation of concerns between the stakeholders involved in creating workflows. The proposed architecture provides a way for the framework users to inject data, processing logic and a workflow definition, combining the two elements.

5 IMPLEMENTATION

We implement our proposed solution as a proof of concept using Docker and Kubernetes. The code of the proposed solution, along with the associated Dockerfiles and Kubernetes YAML files to deploy the model to Kubernetes clusters, are publicly available, under the MIT license[6].

Docker is used for building the container images for both the framework components (orchestrator and framework agent) and the examples of pluggable components (compute steps and data storage). The container images contain all the information needed by a container run-time to run the components. At run-time, the container orchestration solution uses the images to instantiate the different components as needed. Kubernetes is used to manage and orchestrate the deployment and communication between the components. Kubernetes is the industry standard for container orchestration and exposes abstractions over the hardware resources it manages. The following is a brief introduction to the main Kubernetes abstractions and features referenced in the remainder paper:

[6] https://github.com/alin-corodescu/MSc-workflows

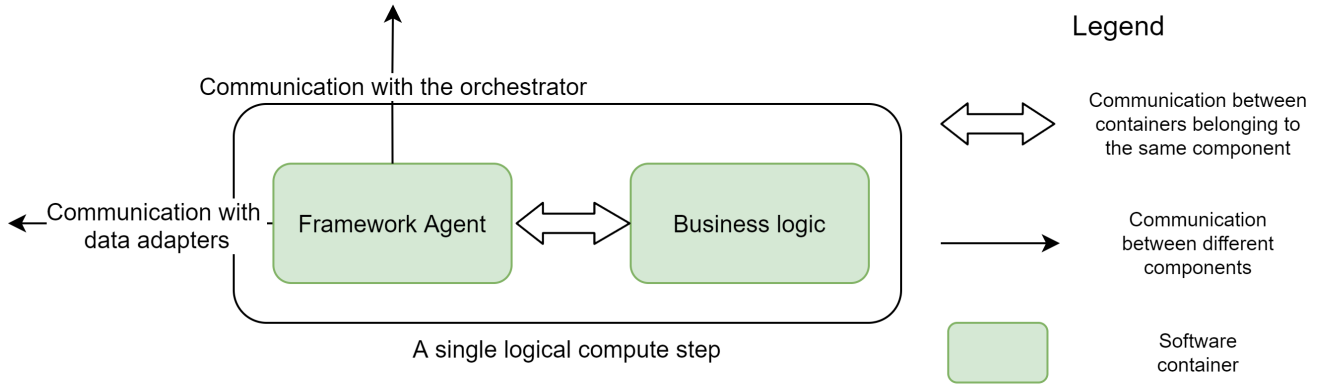

Figure 2: Detailed view of a compute step.

(1) **Nodes:** these are the abstraction Kubernetes uses for the hosts making up a cluster. These can either be physical or virtual machines.
(2) **Pods:** these are the smallest unit Kubernetes manages and deploys. A pod is a group of one or more containers, logically belonging together to perform a particular task.
(3) **Services:** these are a type of Kubernetes resource that helps connecting pods or expose functionalities outside the cluster. A selection criterion is used to identify the pods hosting the application the service exposes. The communication with the service is done through a designated IP address and port, with the request routing and load balancing between the pods being handled by Kubernetes.
(4) **DaemonSets:** these enable Kubernetes users to deploy an instance of a pod to every node in the cluster.
(5) **Labels:** All Kubernetes resources can be associated labels to help identify and distinguish resources serving different purposes (e.g., pods hosting different applications)
(6) **Volumes:** these are abstractions that allow the storage used by a container to be managed independently of the container. Volumes are made accessible by mounting them in a container.

The containers used to perform the steps are long-lived, and each can process multiple units of data. This reduces the overhead in performance incurred by creating and deleting containers constantly throughout the execution of a workflow. The current implementation opts for a point-to-point communication model. It allows for easier, explicit communication between two parties, making it more suitable for the deliberate routing decisions that take into account data locality. The gRPC framework is chosen to support the communication between different components. By leveraging the long-lived nature of the containers, the proposed implementation also attempts to re-use established connections to remote machines (connection pooling).

5.1 Orchestrator

A single instance of the orchestrator is deployed and made available to all the components in the cluster through a Kubernetes service. The single-replica strategy is needed because the central orchestrator is a stateful service, where the state is kept in memory. Setting up a multi-replica orchestrator is beneficial for performance, scalability, and resilience. However, it also requires external solutions to store, share, and synchronise the state among the replicas.

A workflow consists of a sequence of processing steps. Each step specifies a name that uniquely identifies the processing logic of the step. The orchestrator uses the name of the step to find pods hosting the specified processing logic through Kubernetes labels. Alongside the name, a data source and a data sink identifier are also part of a step specification[7]. The data source and sink identify the data storage solutions where the input and output data should be retrieved and stored. The orchestrator exposes another service responsible for orchestrating the execution of the workflow registered through the workflow definition service. The execution of the workflow is driven by the availability of data in the system, as opposed to a task-driven execution approach [12].

Request routing discussed earlier is a multi-objective problem. For the proposed implementation, request routing uses the distance and current load as inputs. The distance between the same hosts is considered zero. For all other cases, the orchestrator constructs a $N \times N$ matrix hosting the distances between each of the N zones (assuming there are N zones in total in the system). We presume that the distance between hosts within the same zone is higher than zero but lower than between two different zones. In the current implementation, the values are hard-coded in the matrix. Using a work tracker component, the orchestrator knows the number of concurrent requests on each pod. We propose two flavors of the selection algorithm.

The first one is the greedy approach: (i) the pods with several active connections higher than a configurable number (three by default) are eliminated, to avoid overloading a particular pod due to the uneven load balancing introduced by the data locality preferences; (ii) from the remaining pods, the one with the shortest calculated distance is selected (greedy selection); and (iii) if no pod remains, the request is added back into the queue to be processed at a later time.

The second approach is slightly modified and focuses more on spreading the load among the available resources. Instead of always choosing the closest pod, this variant spreads the load evenly at a

[7] In a simplified generalisation, steps receive data as input from a data source, processes it, and then push the outputs to a data sink.

zone level and falls back to a different zone[8], when the load on the pods passes a configurable threshold (three by default). The zone with available resources closest to where data are stored is chosen.

5.2 Framework Agent

Since the framework agent is logically coupled with a compute step, the two are always deployed together by leveraging the sidecar deployment pattern [11]. The agent and the compute step are separated in different containers, and they can communicate efficiently via local host calls. With pods being the atomic unit Kubernetes can manage, this deployment model guarantees that each pod hosting a compute step container also contains an agent container.

The agents implement a service to expose a communication endpoint for the orchestrator. The request from the orchestrator contains the metadata needed by the data source to find the data the step should process, a unique identifier for the request (requestId), and the identifier of the data adapter to be used as a data sink. Upon receiving the request, the agent will perform the following operations:

(1) Forward the metadata to the data adapter specified in the request from the orchestrator.
(2) Once the data adapter ensures that the data is available at a path the compute step container can access, the compute step is invoked with the path to the file containing the data it needs to process.
(3) For each output emitted by the compute step, the framework agent will instruct the sink data adapter to register the output.
(4) Once the sink data adapter returns the metadata necessary to identify the newly added data, it is sent, together with the initial requestId, back to the orchestrator to notify that new data are available.

5.3 Data Adapter

In the current implementation, the data adapters only work with files (i.e., data retrieved from the storage solution is stored in a file, and only uploading data from a file to the storage solution is supported). Data adapters are deployed using the DaemonSet concept in Kubernetes. One pod for each type of storage adapter is deployed to every node of the Kubernetes cluster. The DaemonSet choice assumes the number of possibly different types of storage adapters is limited, and thus the overhead of running one pod of each type is negligible.

The distributed storage system used leverages the local storage of every node in the cluster to store the data being processed by the workflow. Besides reading and writing data, the interface also captures the localisation information about the data as part of the communication. Hard linking is used to leverage further the advantage provided by data locality. Hard linking is a faster operation than copying, and thus it further reduces the time spent on moving data between directories. The volumes are based on directories in the underlying node file system. Even though different directories are mounted as different volumes in pods, the resolution of hard links is delegated to the node file system. This allows the hard links to cross volumes mounted in different pods. By contrast, symbolic links operate by attempting to resolve a particular path, so they cannot cross different volumes unless all pods mount the same volume under the same path.

A data master was added as a standalone component to separate data handling from the orchestration components. While the current feature set is limited to locating files in the distributed storage system, it could potentially be extended to offer more functionalities (such as data lineage, sharing the same data across multiple workflows, and others). Like the orchestrator component, the data master runs in a single pod in the cluster, made available through a service. The state of the data master is stored in memory.

6 EVALUATION

Through the experiments, we compare our approach with Argo workflows. The test environment is set up on the Microsoft Azure Cloud using only Infrastructure-as-a-Service offerings (virtual machines and networking capabilities). The test environment is using Standard D2s v3 (two2 vcpus, 8GB memory) virtual machines, provisioned in three different Azure regions to mimic the geographical distribution of resources in a real Cloud and Edge topology. One virtual machine serves as the Kubernetes master node (and does not run additional pods). In addition to the master node, two virtual machines are configured as Kubernetes worker nodes in each region. The lightweight K3s[9] distribution of Kubernetes has been manually installed on each virtual machine. All machines are part of the same cluster.

We consider the following workflow to evaluate the architecture and implementation of the systems. There are four sequential steps, each accepting a file as input, randomly shuffling the bytes, and writing the shuffled result as output. The motivation for choosing an artificial workflow is the ability to capture and describe the behaviour of the solution in terms of universally applicable measures (e.g., bytes for data size). In contrast, a workflow processing particular data types (such as images) captures the specific behaviour better. However, the conclusions are harder to generalise because the data type is more restrictive (in terms of size, characteristics) than the artificial example.

For the first three steps, every machine in the cluster runs one instance of each step type. The final step only runs in the machines in the EastUS region, simulating a step that can only run on Cloud instances in a real scenario. Both the central orchestrator and the data master components are deployed on machines in the WestEurope region. Two additional supporting components are used when running experiments. The load generator is responsible for injecting data into the cluster and notifying the orchestrator that data is available. The load generator creates files containing random bytes, and it can trigger workflows for these files. The size and number of files to be injected into the cluster are configurable. The load generator also supports injecting data into two regions in parallel. The telemetry reader is responsible for gathering data on the execution of the workflow (e.g., time spent in different components, the quantity of data transferred between regions, and load spreading). The data is retrieved using the Jaeger API and is written in CSV files, which are later analysed through Python/Jupyter notebooks.

[8]The zone is a general term, meant to capture groups of hosts in close proximity to one another.

[9]https://k3s.io

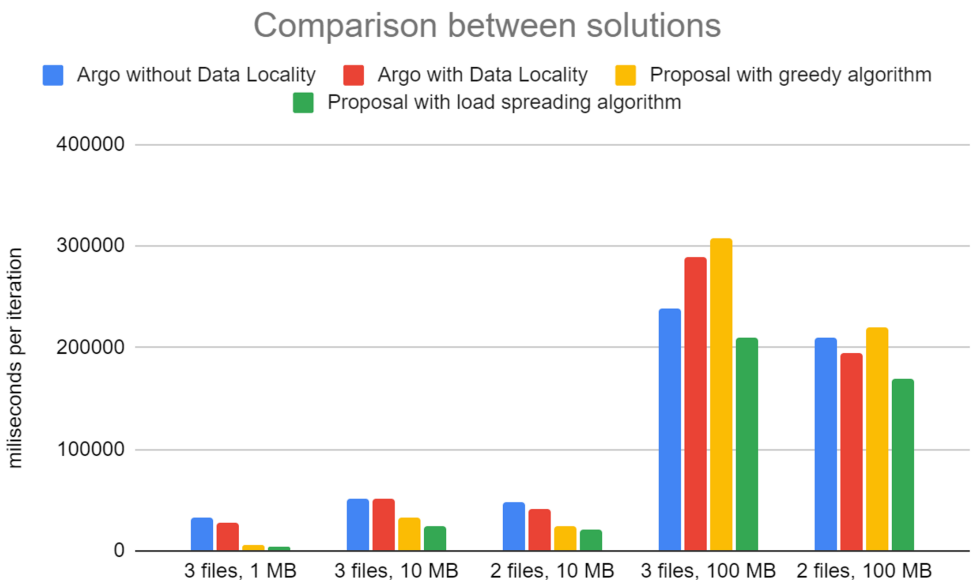

Figure 3: Proposed solution vs Argo.

6.1 Comparison with Argo

We implement the workflow using the Argo workflow definition language. The data communication medium between the steps is Cloud storage (Azure BlobStorage), with one instance provisioned per region. The steps can read input data from any region but write output data to the region they are running in (e.g., a step in WestEurope can retrieve data from NorthEurope, but it always writes the output to WestEurope). Argo orchestration components (e.g., Argo Server) are deployed to worker nodes in the WestEurope region, similar to the proposed implementation. Files of different sizes are uploaded manually to the Cloud storage in WestEurope and NorthEurope region, and workflows are triggered from a client running outside the cluster, with these files as inputs.

In terms of data locality, we evaluate two different approaches. First, no data locality is captured in the workflow definition, allowing the steps to be assigned to nodes anywhere in the cluster. Second, using the node selectors to limit where step pods are instantiated, the processing is kept within the same region (e.g., files originating from WestEurope will continue to be processed in WestEurope as much as possible). By default, the orchestration component of Argo reacts to changes in step states (e.g., a step has finished) once every 10 seconds. This default value is unsuitable for workflows processing small amounts of data quickly, and for these experiments, we change it to one second, which is the minimum recommended value. The exact configuration used to deploy Argo workflows for the experiment is available in the GitHub repository of the solution.

6.2 Results

Figure 3 presents the average run-time of a workflow, given different data sizes and numbers of files to be processed in parallel. The X-axis denotes the number of files used in each of the two Edge regions, WestEurope and NorthEurope, along with their size. For example, "3 files, 1MB" indicates that three files of 1MB each are passed through the workflow from both WestEurope and NorthEurope in parallel (six files in total). The Y-axis is the time spent on the execution of the workflow, measured in milliseconds. The numbers on the Y-axis are averaged from a number of iterations.

The results show a low variance between different iterations. The chart compares the numbers from four runs of the same workflow: (i) in Argo, both with region-level data locality and without data locality, and (ii) in our solution, with the two flavors of the routing algorithm, greedy and load spreading. From experiments, we observe the followings:

(1) The first observation is made on the case when running with small data chunks that take little time to process, the proposed solution outperforms Argo workflows by a factor of five.
(2) As the data size grows, the time spent on executing the logic of the step increases, and the benefit of data locality reduce.
(3) Data from the second case (three files, 10MB) show that using data locality does not affect the case of Argo workflows. Furthermore, in the fourth case (three files, 100MB), using data locality in the workflow has a detrimental effect on performance. This phenomenon can be explained by a setup of the experimental testbed and the nature of the steps.
(4) The solutions that leverage data locality attempt to perform the work close to the data, while the other solutions spread the load evenly across the available machines. The gathered telemetry indicates that the steps of the example workflow are significantly slower when multiple instances are scheduled on the same host, as they are competing for resources.
(5) Considering that there are two worker nodes available in each region, processing three files in parallel results in at least two of the files to be processed on the same node when

data locality is enabled. This load distribution significantly offsets the reduction in data transfer time.

(6) When processing only two files of data sizes of 10MB and 100MB, the load can be better spread on the testbed topology (two files can be processed in parallel on the two host machines available in each region), and the benefit of data locality can be observed.

(7) On all cases, however, our proposed solution with the load spreading algorithm proved to be faster to execute the example workflow. However, the benefits varied, depending on the size of the data, from 500% (for the three files, 1MB) to roughly 20% (for the two files, 100MB case).

7 CONCLUSION

In this paper, we proposed a novel architecture and a proof-of-concept implementation for container-centric Big Data workflow orchestration systems. Our proposed solution enables the orchestration components to consider data locality, quantified using a flexible model that accounts for the physical distance between hosts spread across the computing continuum. Our solution is better suited for processing small, frequent data units by leveraging long-lived containers, but they are instead re-used to process multiple units. Furthermore, it extends the ideas behind isolating processing steps in separate containers to the data management aspect of Big Data workflows. As such, the logic needed to interact with data management systems is encapsulated in containers, providing the same benefits as for processing logic (technology agnostic solution, isolation, lightweight, etc.).

Future work can include extending the solution in multiple directions. The simplistic workflow definitions supported in the current form investigate potential performance improvements, but most real use-cases require complex constructs. A few examples include supporting direct acyclic graph-structured workflows, processing steps to receive input from more than one data source, and supporting aggregation over multiple data units. Both data and processing logic are isolated in different components, and the workflow definition language uses these components as building blocks. A potential direction is creating a marketplace-like ecosystem where data and processing logic are exchanged between parties in the form of these components. Regarding run-time concerns, a central piece of the proposed solution is the routing algorithm that takes into account load and data locality. Further improving the heuristic and adding more dimensions (such as matching node capabilities, e.g., needs to have a GPU with step requirements) can lead to better results.

ACKNOWLEDGMENTS

This work was partly funded by the EC H2020 project *"DataCloud"* (Grant nr. 101016835) and the NFR project *"BigDataMine"* (Grant nr. 309691).

REFERENCES

[1] M Abranches, S Goodarzy, M Nazari, S Mishra, and E Keller. 2019. Shimmy: Shared Memory Channels for High Performance Inter-Container Communication. *USENIX Workshop on Hot Topics in Edge Computing (HotEdge)* (2019). https://par.nsf.gov/biblio/10107994

[2] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proc. of SWEET 2012*. 1–13. https://doi.org/10.1145/2443416.2443417

[3] Ardavan Ashabi, Shamsul Bin Sahibuddin, and Mehdi Salkhordeh Haghighi. 2020. Big Data: Current Challenges and Future Scope. In *Proc. of ISCAIE 2020*. 131–134. https://doi.org/10.1109/ISCAIE47305.2020.9108826

[4] Mutaz Barika, Saurabh Garg, Albert Y. Zomaya, Lizhe Wang, Aad Van Moorsel, and Rajiv Ranjan. 2019. Orchestrating Big Data Analysis Workflows in the Cloud: Research Challenges, Survey, and Future Directions. *Comput. Surveys* 52, 5 (2019). https://doi.org/10.1145/3332301

[5] Mutaz Barika, Saurabh Garg, Albert Y. Zomaya, Lizhe Wang, Aad Van Moorsel, and Rajiv Ranjan. 2019. Orchestrating Big Data Analysis Workflows in the Cloud: Research Challenges, Survey, and Future Directions. 52, 5 (2019), 95:1–95:41. https://doi.org/10.1145/3332301

[6] El Houssine Bourhim, Halima Elbiaze, and Mouhamad Dieye. 2019. Inter-container Communication Aware Container Placement in Fog Computing. In *Proc. of CNSM 2019*. 1–6. https://doi.org/10.23919/CNSM46954.2019.9012671

[7] Yared Dejene Dessalk, Nikolay Nikolov, Mihhail Matskin, Ahmet Soylu, and Dumitru Roman. 2020. Scalable Execution of Big Data Workflows using Software Containers. In *Proc. of MEDES 2020*. 76–83. https://doi.org/10.1145/3415958.3433082

[8] Yehia Elshater, Patrick Martin, Dan Rope, Mike McRoberts, and Craig Statchuk. 2015. A Study of Data Locality in YARN. In *Proc. of Big Data 2015*. 174–181. https://doi.org/10.1109/BigDataCongress.2015.33

[9] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. 2019. Edge computing: A survey. *Future Generation Computer Systems* 97 (2019), 219–235. https://doi.org/10.1016/j.future.2019.02.050

[10] D. Kimovski, R. Matha, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan. 2021. Cloud, Fog or Edge: Where to Compute? *IEEE Internet Computing* (in press) (2021). https://doi.org/10.1109/MIC.2021.3050613

[11] Philippe Martin. 2021. *Multi-container Pod Design Patterns*. Apress, Berkeley, CA, 169–173. https://doi.org/10.1007/978-1-4842-6494-2_13

[12] Ryan Mitchell, Loic Pottier, Steve Jacobs, Rafael Ferreira da Silva, Mats Rynge, Karan Vahi, and Ewa Deelman. 2019. Exploration of Workflow Management Systems Emerging Features from Users Perspectives. In *Proc. of Big Data 2019*. 4537–4544. https://doi.org/10.1109/BigData47090.2019.9005494

[13] Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B. Hall, Christopher H. Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O. Twardziok, Alexander Kanitz, Andreas Wilm, Manuel Holtgrewe, Sven Rahmann, Sven Nahnsen, and Johannes Köster. 2021. Sustainable data analysis with Snakemake. 10 (2021), 33. https://doi.org/10.12688/f1000research.29032.2

[14] Nenavath Srinivas Naik, Atul Negi, Tapas Bapu B.r., and R. Anitha. 2019. A data locality based scheduler to enhance MapReduce performance in heterogeneous environments. *Future Generation Computer Systems* 90 (2019), 423–434. https://doi.org/10.1016/j.future.2018.07.043

[15] Nikolay Nikolov, Yared Dejene Dessalk, Akif Quddus Khan, Ahmet Soylu, Mihhail Matskin, Amir H. Payberah, and Dumitru Roman. 2021. Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers. *Internet of Things* (in press) (2021), 100440. https://doi.org/10.1016/j.iot.2021.100440

[16] Thomas Renner, Lauritz Thamsen, and Odej Kao. 2016. CoLoc: Distributed data and container colocation for data-intensive applications. In *Proc. of Big Data 2016*. 3008–3015. https://doi.org/10.1109/BigData.2016.7840954

[17] Chen Youmin, Lu Youyou, Luo Shengmei, and Shu Jiwu. 2019. Survey on RDMA-Based Distributed Storage Systems. *Journal of Computer Research and Development* 56, 2 (2019), 227. https://doi.org/10.7544/issn1000-1239.2019.20170849

[18] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. of USENIX 2010*.

[19] Dongfang Zhao, Mohamed Mohamed, and Heiko Ludwig. 2020. Locality-Aware Scheduling for Containers in Cloud Computing. 8, 2 (2020), 635–646. https://doi.org/10.1109/TCC.2018.2794344

[20] Charles Zheng and Douglas Thain. 2015. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In *Proc. of VTDC 2015*. 31–38. https://doi.org/10.1145/2755979.2755984

[21] Baifan Zhou, Yulia Svetashova, Tim Pychynski, Ildar Baimuratov, Ahmet Soylu, and Evgeny Kharlamov. 2020. SemFE: Facilitating ML Pipeline Development with Semantics. In *Proc. of the CIKM 2020*. 3489–3492. https://doi.org/10.1145/3340531.3417436