Akif Quddus Khan

Master's thesis

2022

Master's thesis

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Akif Quddus Khan

Smart Data Placement for Big Data Pipelines with Storage-as-a-Service Integration

May 2022

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Smart Data Placement for Big Data Pipelines with Storage-as-a-Service Integration

Akif Quddus Khan

Masters - Applied Computer Science

Submission date:	May 2022
Supervisor:	Ahmet Soylu
Co-supervisor:	Dumitru Roman, Mihhail Matskin, Radu Aurel Prodan

Norwegian University of Science and Technology
Department of Computer Science

Smart Data Placement for Big Data Pipelines with Storage-as-a-Service Integration

Akif Quddus Khan

May 31, 2022

Abstract

Big data pipelines are developed to process big data and turn it into useful information. They are designed to support one or more of the three big data features commonly known as the three Vs (Volume, Velocity, and Variety). With big data pipelines, massive volumes of data may be extracted, transformed, and loaded, unlike with smaller data pipelines. The implementation of a big data pipeline includes several aspects of the computing continuum such as computing resources, data transmission channels, triggers, data transfer methods, integration of message queues, etc., making the implementation process difficult. The design gets even more complex if a data pipeline is coupled to data storage, such as a distributed file system, which comes with additional challenges such as data maintenance, security, scalability, etc. On the contrary, many cloud storage services, such as Amazon *Simple File Service* offer nearly infinite storage with great fault tolerance, giving possible solutions to handle big data storage concerns. Thus, moving data to cloud storage/storage-as-a-service (StaaS) will move the extra overhead of data redundancy, backup, scalability, security, etc. to the cloud service provider, which makes the implementation process of a big data pipeline relatively easier.

The work presented in this thesis aims to 1) realize big data pipelines with hybrid infrastructure, i.e., computation on a local server but integration with storage-as-a-service; 2) develop a ranking algorithm to find the most suitable storage facility in real-time based on the user's requirements; 3) develop and entail the use of a domain-specific language to deploy a large data pipeline using StaaS. A novel architecture is proposed to realize the big data pipeline with hybrid infrastructure. In addition, an evaluation matrix is proposed to rank all available storage options based on five parameters; cost, physical distance, network performance, impact of server-side encryption, and user weights. Further, to simplify the deployment process, a new domain-specific language has been developed with an extensive vocabulary to cover all major aspects of the cloud continuum in general, and cloud storage in particular. This thesis proves the effectiveness of cloud storage by implementing the big data pipeline using the newly proposed architecture. Moreover, it justifies the importance of individual parameters involved in the evaluation matrix, such as cost, physical distance, network performance, and the impact of server-side encryption by the execution of a number of experiments. It also talks about different ways that the new ranking algorithm or evaluation matrix can be shown to work.

Acknowledgements

I would like to express my appreciation to everyone who stood by my side and encouraged my efforts to complete my thesis.

My deepest appreciation goes to my main supervisor, Ahmet Soylu, and co-advisors Dumitru Roman, Mihhail Matskin, and Radu Aurel Produan for their superb assistance and input, particularly in the form of asking the necessary questions to steer my thesis in the correct track. Through my supervisors, I was introduced to a group of amazing and highly skilled, including Nikolay Nikolov and Amir Payberah, from different research institutions (SINTEF Digital and KTH Royal Institute of Technology), who had the patience to facilitate me throughout the overall process of my thesis, from the times when my ideas were scattered to the finished version. Their counsel was quite beneficial, and it frequently helped me overcome obstacles and improve the thesis.

Lastly, but by no means least, I must express my deepest thanks to my parents and my siblings, Ata ul Razzaq Khan and Afia Farkhanda, for their unwavering support, encouragement, and love during my years of education and the process of researching and writing this thesis. Without them, this achievement would not have been possible. Thank you.

Contents

Figures

Tables

Code Listings

Part I

Preamble

Chapter 1

Introduction

The exponential expansion of digital data sources has the potential to transform every aspect of society and human life. To achieve this impact; however, data must be evaluated swiftly for extracting insights that may be used to drive decision-making. One way to process data efficiently is through data pipelines using all the available local and remote computing infrastructure. Yet, typical systems that rely on data transfer to distant data centers for processing are no longer feasible. Rather than that, it is critical to develop innovative methods for efficiently using distributed computing infrastructure and services for big data pipelines. Specifically, the combination of resources and services smoothly in the cloud and at the edge, as well as throughout the data channel as needed [1].

This thesis outlines our concept for providing a strategy for processing big data pipelines in the computing continuum, i.e., establishing a hybrid architecture for big data pipelines. In this case, hybrid computing combines compute on a local server or cloud server with storage-as-a-service integration. Moreover, the assessment of available cloud storage providers and choosing the most suitable one according to the customer's needs. This necessitates a one-of-a-kind solution for data pipeline design and a method for real-time data placement with unknown data volumes, availability, location, and data security constraints. Figure 1.1 shows the complete setup of the approach in the computing continuum. It can be that there is a data pipeline server (on-premise or on the cloud), connected with a data placement method for finding the most suitable cloud storage option in real-time. Once cloud storage is integrated with the data pipeline server, the complete benefits of cloud storage can be utilized, such as data redundancy, backup, and security.

1.1 Problem description

The development of a big data pipeline is a complicated task. It involves many elements such as compute resources, data communication medium, triggers, data transfer mechanism, integration of message queues, etc. The architecture becomes even more complicated with a data pipeline is attached to data storage, such as distributed file system. Data storage comes up with additional issues. These issues

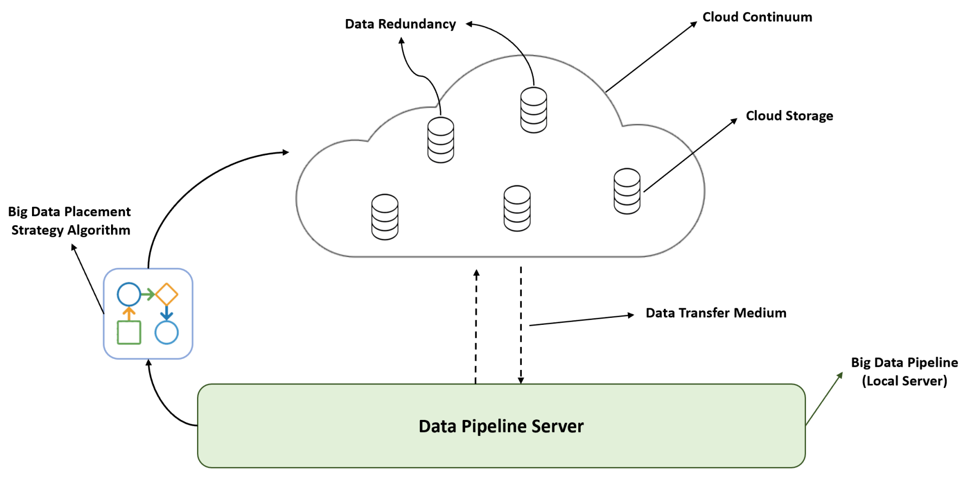


Figure 1.1: Big data pipeline in computing continuum

are created by the amount, velocity, and variety of Big Data. Storing big data on traditional physical storage on-premise is troublesome as hard disk drives (HDDs) regularly fail, and specific data protection measures, e.g., RAID, are not efficient [2].

Cloud storage systems (e.g., Amazon S3, Elastic Block Store, or EBS) offer nearly infinite storage with great fault tolerance, giving possible solutions to handle big data storage concerns. However, moving to and hosting big data on the cloud is expensive given the scale of data volume [3]. Principles and algorithms, including the spatiotemporal patterns of data consumption, need to be created to establish the data's analytical value and its preservation datasets by balancing the expense of storage and data transfer with the quick accumulation of big data [4]. In addition, the pace of big data necessitates the storage systems to scale up fast, which is challenging to do with standard storage systems. That is why it is important to either select a suitable storage facility before data placement, or move the compute step closer to the data during data processing.

In short, when it comes to big data pipelines, local storage is expensive, hard to maintain, comes with additional challenges like data availability, data security, and backup. Furthermore, it increases the complexity of the big data pipelines.

1.2 Justification, motivation, and benefits

This thesis aims to remove local storage and integrate storage-as-a-Service (StaaS) with big data pipelines. Big data pipelines need large storage, and local storage is limited and expensive, harder to manage, and has additional overheads of data backup, security, and availability. For local storage, hardware must be purchased for start-up and maintenance. Although it provides complete control over the resources, organizations are responsible for it's security. In addition to that, organizations are responsible for hardware maintenance and updates.

Cloud storage provides the most flexibility. Organizations determine where to run their applications. Even the physical hardware is not on premise, the organizations control security, compliance, and legal requirements. StaaS provides a viable solution in terms of cost, security, automation, accessibility, syncing, sharing, collaboration, data protection, and disaster recovery. Following are some key benefits of cloud storage:

- High availability
- Fault tolerance
- Scalability
- Global reach
- Customer latency capabilities
- Agility
- Predictive cost considerations

In terms of cost, cloud storage has operational expenditure, rather than capital expenditure. Capital Expenditure or CapEx is the up-front spending of money on physical infrastructure. Costs from CapEx have a value that reduces over time. On the other hand, operational expenditure or OpEx is the spending and billing of services as needed. Expenses are deducted in the same year. Cloud storage providers operate on a consumption-based model, which means that end users only pay for the resources that they use. Whatever they use is what they pay for. It provides better cost prediction. Prices are for individual resources and services, and most importantly, billing is based on actual usage.

Figure 1.2 shows an overall comparison of different level of services provided in cloud computing. It clearly shows the amount of overhead being transferred to the cloud service provider if an organization or user opt for cloud storage.

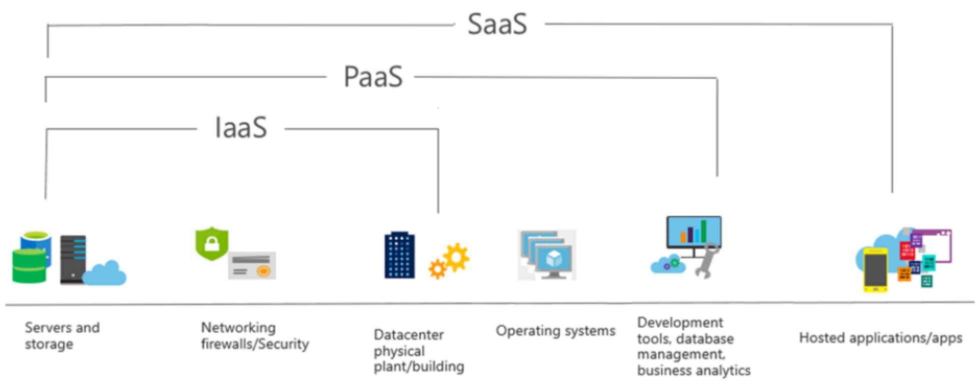

SaaS
PaaS
IaaS
Servers and storage
Networking firewalls/Security
Datacenter physical plant/building
Operating systems
Development tools, database management, business analytics
Hosted applications/apps

Figure 1.2: Comparison of resources and setup cost for IaaS, PaaS and SaaS [5]

Figure 1.3 shows a shared responsibility model i.e a detailed comparison of management responsibilities between On-premises compute and storage resources and different levels of cloud services.

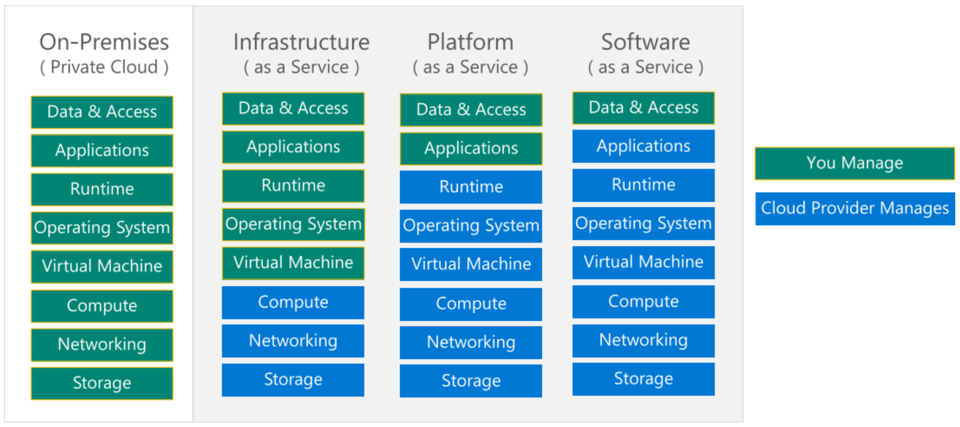


Figure 1.3: Shared responsibility model [6]

By using cloud storage or StaaS, full potential of cloud storage can be utilized. Moreover, it increases the simplicity of the big data pipeline.

1.3 Research objectives

Following are the research objectives of this thesis:

- integration of storage-as-a-service(StaaS) with big data pipelines to overcome technical and architectural challenges;
- to develop an effective evaluation criteria and method for ranking the available options so that an informed choice can be made;
- to develop a domain-specific language with an ability to deploy a big data pipeline with integration to StaaS.

1.4 Research questions

To achieve the objectives identified in the Section 1.3, the following questions need to be answered:

1. How StaaS is better than Local Storage for a big data Pipeline?
   - In terms of Data Availability?
   - In terms of Security and Reliability?
2. How big data pipelines can be deployed using StaaS?
3. How can we rank different cloud storage options based on the user requirements?
4. How can StaaS options be incorporated with DSL for big data Pipelines?

1.5 Contributions

In the light of the problems and challenges identified in the preceding sections, to advance state of the art, the research project presents a novel architecture, supported by a proof-of-concept implementation, focusing on overcoming the previously mentioned shortcomings. The findings of this research project can aid the big data pipeline designers and other related stakeholders to exploit the full potential of the compute servers and the benefits of cloud/edge storage by deploying a data pipeline in a hybrid cloud environment.

1.5.1 Big data pipeline architecture

This thesis proposes a novel architecture for big data pipelines. Following the logic outlined in Section 4.1, conventional data pipelines are installed either on a local computer or in the cloud continuum, with a single cloud service provider serving as the backbone. This work presents a hybrid architecture for big data pipelines. More details about the data pipeline architecture is presented in Chapter 6.2.

1.5.2 Ranking criteria and method

Existing data placement mechanisms are either platform dependant, too generic or are not related to big data pipelines as explained in Section 4.2. Hence, the thesis aims to identify a set of criteria/parameters for cloud storage selection and to develop a comprehensive method, which we call evaluation matrix (EM), to calculate and find the best available storage facility from multiple cloud service providers such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform. The parameters for the EM currently contain distance, cost, impact of server-side encryption, and network latency. The multi-criteria decision making/analysis (MCDA) is employed for this purpose.

1.6 Thesis outline

The rest of the thesis is arranged as follows:

Chapter 2: Methodology This chapter briefly describes the process of scientific research in general and computer science and technology research in particular. The major part of the chapter describes the complete research process which is employed for this thesis, starting with formulating the *Problem Statement*, to textitImplementation, and finally *Evaluation and Results* of the proposed solution, along with the methods such as literature study, interviews, and quantitative and qualitative analysis.

Chapter 3: Current technological landscape This chapter covers the technical terms involved in this research project. It gives a bigger picture of the big data pipeline in the computing continuum. It starts with the importance of big data and big data pipelines. Then it defines other technical terms such as computing continuum, cloud continuum, fog and edge computing. The last section explains storage-as-a-service (STaaS), data locality, and a big data pipeline, and how a domain-specific language can be used to deploy big data pipelines.

Chapter 4: Related work This chapter discusses the specifics of the relevant literature. It is divided into three sections, each section for each research question. The first section explains the benefits of cloud storage in big data pipelines. The second section explains the different techniques for large data placements. The last section covers the existing relevant domain-specific languages and how STaaS can be integrated with a DSL.

Chapter 5: Problem analysis This chapter starts with an overview of the problem that this thesis aims to resolve. Then it presents the justification of the selected parameters. It describes the selected parameters such as different storage tiers offered by each cloud storage provider, pricing structure for the network usage, and different types of data encryption methods, along with the chosen cloud storage options for each parameter. Lastly, it explains multi-criteria decision analysis and the selected VIKOR method for the ranking algorithm.

Chapter 6: Smart data placement solution design This chapter starts with the brief introduction of the preliminary research work on the big data pipeline architecture, domain-specific language, and effects of incorporating data locality. Furthermore, it has five further sections. The first describes the proposed novel architecture; the ranking/evaluation method; the software design for its sub-component; and the software design for the evaluation matrix. The last section contains the design details for the new extended domain-specific language.

Chapter 7: Smart data placement solution implementation This chapter contains the implementation details of all the software components, such as finding the physically closest storage facility, software package for cost estimation, measuring the network performance, and studying the impact of server-side encryption on the performance. It also has the implementation details for the evaluation method and algorithm. The last section of this chapter presents the implementation details for the domain-specific language to deploy a big data pipeline with STaaS integration. Relevant code snippets and links to the code repositories are also presented in this chapter.

Chapter 8: Experiments and results Experiments are designed around four parameters: distance, cost, network performance, and server-side encryption, as well as the method of evaluation. As a result, this chapter describes each experiment in depth, including the technical setup and results.

Chapter 9: Discussions and conclusion The last chapter of the thesis sums up the advances to the state of the art that have been made throughout the course of the project. The final section of the thesis outlines possible avenues for future research.

Chapter 2

Methodology

The objective of this chapter is to present an overview of the research process that was used throughout this project work, as well as to situate it within a broader framework of scientific research methodologies in general. Starting with a very broad review of scientific research methodologies, the chapter sets this project work within the context of the history of computer science and technology research.

2.1 Scientific Research Method

When hypotheses are scrutinized, they may need to be modified or discarded in light of the new findings. This is how scientific inquiry distinguishes itself from other forms of inquiry. This section summarizes some of the most fundamental components of scientific study and then situates this endeavor within the framework of computer science and technology research.

2.1.1 Scientific Research

Deductive and inductive procedures are two of the most common methodologies used in scientific inquiry. It is possible to deduce new facts using deductive approaches, which are based on the use of logical reasoning. A theory and a collection of premises may be used to draw conclusions [7]. When we use inductive approaches, we use observations to find principles that could be extended into rules in order to develop theories that enable us to accurately anticipate the outcomes of our experiments. To distinguish itself from non-sciences like religion and pseudoscience, scientific research is prepared to examine ideas and discard them if they don't fit the facts. Even if evidence contradicts a theory, it doesn't imply that it must be discarded right away. Ad hoc hypotheses may be used to "rescue" any hypothesis. When fresh data challenges an existing theory, new hypotheses may be proposed to alter its assumptions or conclusions [7].

2.1.2 Computer Science & Technology Research

Computer science research is usually technology-oriented, in contrast to many other disciplines of study. Computer science is intended to create largely technological innovations, such as procedures, techniques, algorithms, or products [8]. Additionally, computer science integrates theory with real-world applications, but often discusses theoretical issues using idealized perspectives of processing. For example, time and memory needs for an algorithm are often expressed in terms of abstract fundamental operations, whereas memory requirements are typically expressed in terms of the algorithm's upper limits of complexity. This enables us to think about and compare algorithms theoretically (e.g., in terms of time and memory complexity) in a manner that is independent of the algorithms and their input. However, when those same algorithms are implemented or used in a real-world situation, their behavior may vary significantly. For example, the distribution of data in the actual world often differs from the worst-case situations examined in theory, and there is typically a significant cost difference between an expected basic operation and its practical implementation in a machine. Technology research, on the other hand, is essentially an applied science in which ideas and models from several disciplines of study are applied to real-world situations [9]. For instance, in technological research, focusing only on one technique from a collection of choices while disregarding the others might be both adequate and sensible. Generally, a comparable strategy would not be accepted from other research viewpoints unless a compelling scientific basis for not investigating other viable options could also be offered.

2.2 Research Process

To achieve the expected goals stated in Section 1.4, we have followed a combination of qualitative and design science research approaches. Figure 2.1 shows the complete process of the research project from start to finish. Starting from describing the problem statement, research question formulation, literature survey, implementation, and evaluation. Moreover, the overall work follows the design science research guidelines. It is an inherent problem-solving process with a series of guidelines that ensure proper understanding and knowledge of the design problem and its solution. Also, it relies upon the rigorous method in both solution building and evaluation. Details for each task and information about the methods used to perform that task is as follows:

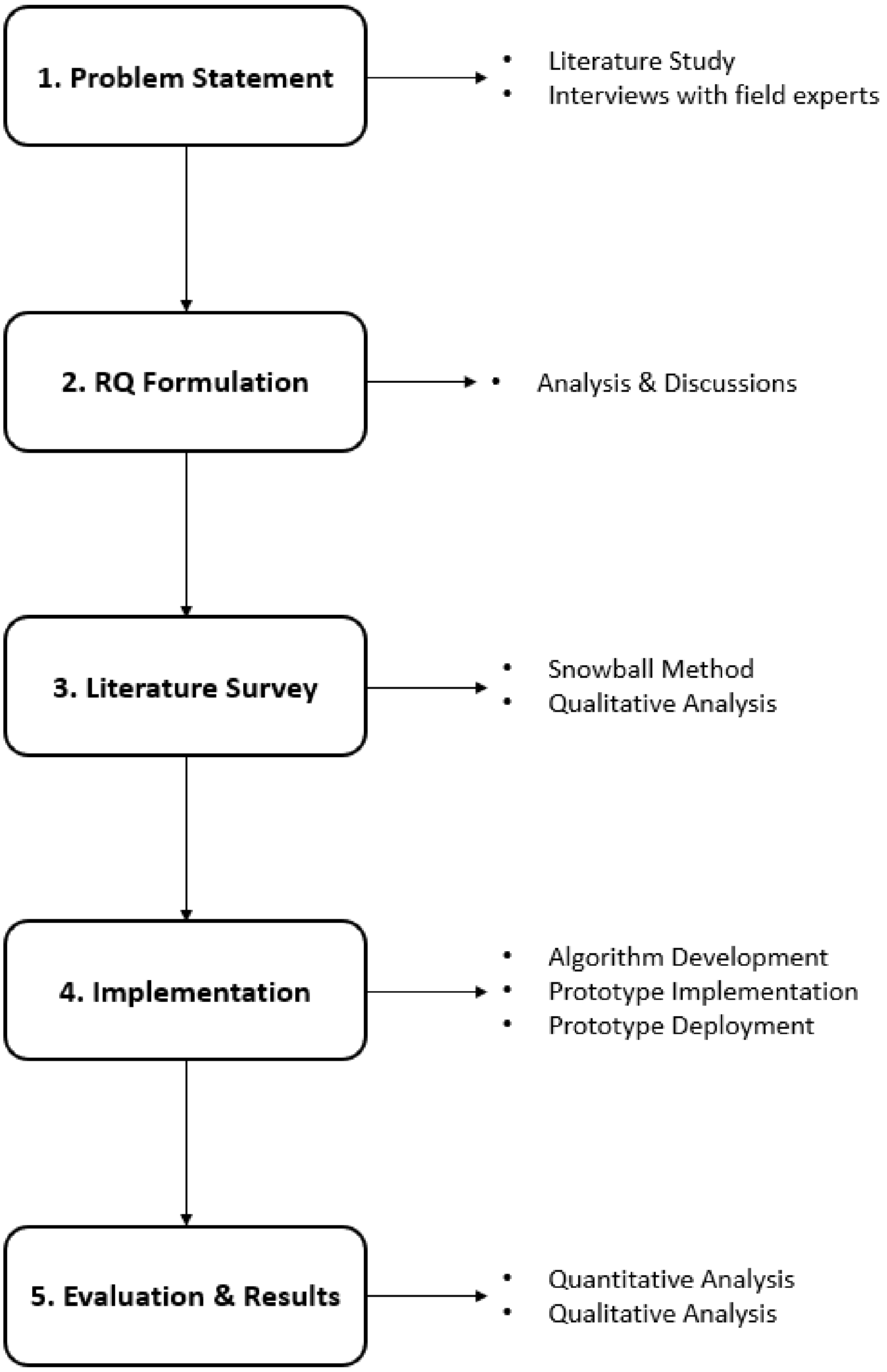


Figure 2.1: Complete research process for the thesis along with the used methodologies.

2.2.1 Formulation of Problem Statement

The process started by narrowing the research topic to big data pipelines and data locality based on interest. Interviews were conducted (Semi-structured interviews) with the field experts to discuss the possible research area related to these topics. Based on the feedback received from the field experts, I studied relevant literature to find out suitable papers and case studies to validate the problem and find out possible enhancements that can be done to the existing state of the art. The first processing step is based on thoroughly studying and understanding the case and getting all the relevant information on it, both from relevant literature and from experts in the field. The 1st step is further divided into two parts for efficient execution.

Interviews with field experts: This sub-step consisted of carrying out detailed but comprehensive 1v1 interviews with renowned experts in the field [10]. This step was important to gain significant insight into various perspectives regarding the case, get knowledge from the field experts, and find out about the current research trends and possibilities.

Literature Study: This sub-step involved the thorough study of relevant literature [11]. In this case, from various sources, mainly from scientific journals. We reviewed the literature from an analytical and usability perspective. Literature study is an integral part of any research project. It is used to discover existing research and studies in the same subject area. It is also important to uncover the research gaps and potential areas to add novelty. So that is why literature is done in this project as well.

2.2.2 Formulation of Research Questions

This step consisted of two parts, extensive analysis and discussions. The data collected from the first step was then discussed with the project supervisors to get their insights. The analysis was important to get a good birds-eye view of the information and resources at disposal and to decide how to utilize them effectively in this research process [12]. The discussions part included detailed discussions with our peers and knowledgeable supervisors about this case. Through the discussions, we broadened our view on this case and refined the research questions.

Literature Survey This step was very important because it consisted of us researching and compiling all the previous published literature relevant to this case that we found usable [13]. The presence of previously published literature related to our case served as a guideline and inspiration. Furthermore, we were also able to analyze the weaknesses in those research papers and avoid them independently. We gathered the relevant literature by using citations and references on papers as a guide to discover more. The **snowball method** is a strategy of gathering literature by employing a key publication on your subject as a starting point, then use the bibliography supplied on the main page to seek other resources on the subject [14]. In this project, the snowball method is used to search and accumulate scientific papers or simply for data generation because we have a wide range of keywords involved in this project. Moreover, we employed **qualitative analysis** techniques for data analysis since the data at hand was all textual.

2.2.3 Implementation

The step contains implementing a proof-of-concept prototype that covers the most important functionality included in the system architecture. It includes 1) The formulation of the evaluation matrix based on the careful analysis of the collected data 2) The development of the evaluation matrix into a testable software program. 3) The deployment of the new proposed architecture for big data pipelines. The script is developed in Python, whereas the data pipeline is deployed on the cloud continuum. The implementation also contains the development of a Domain-specific language meta-model and its compiler.

2.2.4 Evaluation & Results

After working on developing the algorithm and implementing and successfully deploying the prototype, the last step was evaluating and compiling the results. I first carried out a detailed **qualitative analysis** and compiled the results using the different methods. Qualitative analysis is carried out to analyze how much the proposed solution fulfills the user's non-functional requirements, such as security policies. **Quantitative analyses** are used to compare structured data such as processing time, network latency, etc. Lastly, we analyzed all the evaluations we had gathered and compiled them into the final results.

Part II

Background

Chapter 3

Current technological landscape

This chapter will cover the technical terms involved in this research project. Figure 1.1 shows the complete setup of the project in the abstract. The big data pipeline is set up in a computing continuum that contains cloud continuum, local compute servers, and data transfer medium. A big data pipeline can be deployed either on an Edge or on a local server depending on the business and use case requirements. For data storage, this research project aims to exploit the benefits of cloud storage of Storage-as-a-Service in the cloud continuum. Data transfer protocols such as FTP and HTTP will be used for data transfer between the pipeline server and the StaaS. Data Placement Strategy algorithm will be deployed on the data pipeline server to evaluate all available storage facilities. Each technical term is explained in detail below.

3.1 Big data

Data is considered an asset nowadays and it can be purchased and sold on data markets [15]. Data markets enable harnessing vast kinds of data in large volumes to boost the value of the end outcomes. Although there is no formal definition of "big data," both the academic and industrial communities commonly describe large data using a number of "Vs." [16]:

1. **Volume**: refers to the raw data size, which can range from hundreds of gigabytes to petabytes.
2. **Velocity**: in tandem with the enormous amount of data, the fast pace at which it is being created is also a distinguishing trait for big data. Systems need to be built to scale up to meet the throughput needs.
3. **Variety**: Variety in big data refers to any structured or unstructured data produced by humans or machines.

These features apply to the technological elements of processing large data, and designing solutions that can meet these difficulties is hard and costly. Creating the algorithms and tools that help extract value from data adds to the complexity and cost of producing complete big data solutions. Consequently, another V, the value the solution creates, which needs to balance the high cost, is typically mentioned as a crucial aspect of big data. A latest study [17] demonstrates that the diversity and the pace of big data play a more essential influence compared to the volume when measuring innovation performance.

Big data has quickly gained popularity because it enables the exploitation of data to discover hidden trends that provide important insights to drive corporate decisions and aid research. It has been successfully applied in a wide variety of fields, including marketing, social network analysis, healthcare, and finance. The widespread use of big data patterns stems from the idea that with a large amount of data, many real-world occurrences can be better understood and predicted.

3.2 Big Data Pipelines

Big data pipelines are developed to process the big data explained in the previous section and turn it into useful information. Big data pipelines are data pipelines designed to support one or more of the three big data features. *The term "data pipeline" describes a set of processes that performs a variety of transformations, such as data enrichment and data duplication* [18]. Compared to smaller data pipelines, the volume of data processed by a big data pipeline is vastly greater. Massive volumes of data may be extracted, transformed, and loaded using big data pipelines. In what follows, we describe two key concepts concerning big data pipelines.

Steps in big data pipelines Steps as shown in Fig 3.1 (as Compute Step A, Compute Step B and Compute Step C) are the atomic unit of pipelines. Steps are logical units that may be applied to data (e.g., cleaning text of the HTML and JavaScript code). Using a basic generalization, each step receives data as input from a data store and processes it before pushing the outputs back to a data storage.

Data Communication Medium Processed data travels over the computer network using the data transmission medium as a conduit. The inputs and outputs of a step are coupled to such channels. The results of one stage must be used as the input for the following step in the pipeline. Control messages such as notifications and triggers for the other steps are also passed through the communication medium. Some common examples include KubeMQ, ZeroMQ etc.

3.3 Computing Continuum

This work presented in this thesis aims to deploy big data pipelines in the computing continuum, i.e., integration of cloud resources to data pipelines to exploit the full potential of data pipelines. Continuum competitors make decisions throughout the computing continuum and across the company in order to build an integ-

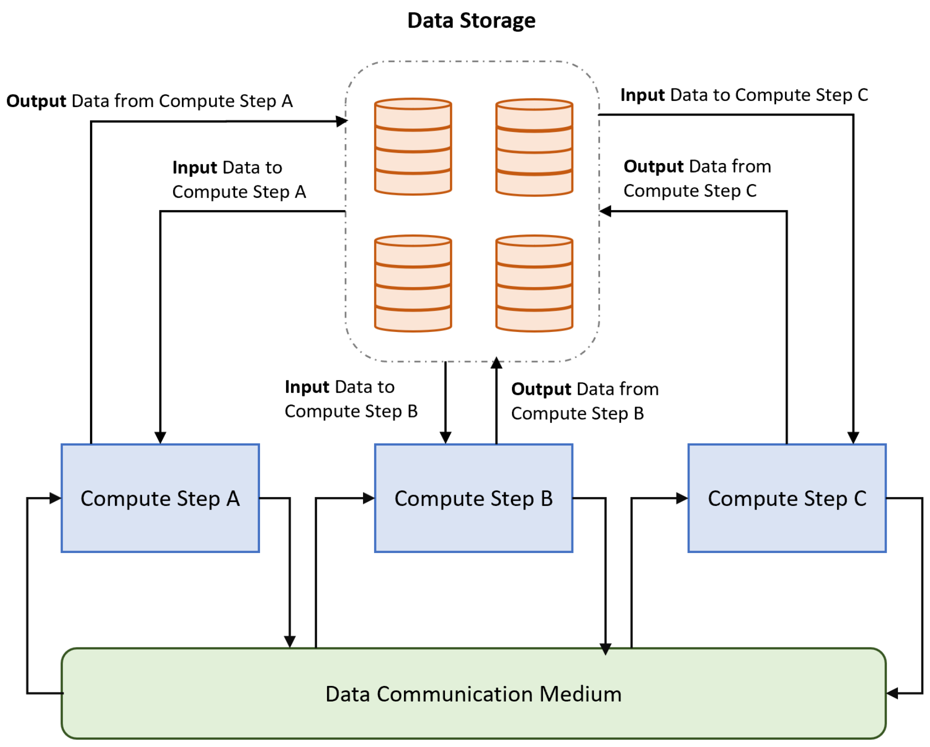

Data Storage
Output Data from Compute Step A
Input Data to Compute Step A
Input Data to Compute Step C
Output Data from Compute Step C
Input Data to Compute Step B
Output Data from Compute Step B
Compute Step A
Compute Step B
Compute Step C
Data Communication Medium

Figure 3.1: Big data pipeline architecture example

rated technology and capabilities foundation capable of supporting the business's ever-changing demands. They recognize that the computing continuum is a collection of capabilities that range from public to edge computing — and everything in between. This continuum includes various types of ownership and location (public to private to hybrid to co-location, multi-cloud, and edge), which are all dynamically supported by next-generation connectivity technologies like 5G and software-defined networks [19].

Historically, the term "cloud" was used to refer to both public cloud computing and shared data center computing. However, most businesses today utilize a mix of public, private, and edge clouds — with minimal interaction between them. As a consequence, innovation, data, and best practices developed in one part of the business do not benefit other parts, limiting the company's overall value-generating potential. The computing continuum, which is defined as "from public cloud to edge cloud and everything in between," encompasses a diverse set of capabilities and services that are readily linked by cloud-first networks and governed by complicated standards. Additionally, the collection of technologies varies by who owns them and their location, which might range from inside the company to off-site [19].

3.3.1 Cloud Computing

The preceding section highlights how difficult and expensive it is to manage big data activities. Big data's characteristics result in problems at all levels of the technological stack: to process big data, vast raw network, storage, memory, and computation resources are required at the infrastructure level. These resources are frequently provided by a distributed system of multiple machines. Software frameworks that can properly exploit existing resources and manage the ever-changing needs of big data operations must be established on a regular basis at the platform level.

At an application level, algorithms operating on the above-described platforms need to be designed and coupled to extract value from the data. Applications can also assist the engagement with a big data solution (e.g., visualization tools used by corporate leaders to examine the findings provided by a big data solution) (e.g., visualization tools used by business executives to analyze the results produced by a big data solution). The concept of cloud computing paves the way for accessible, economical, efficient big data processing through scalability, elasticity, pay for what you use approach.

3.3.2 Fog Computing

Fog computing [20], also referred to as fog networking or fogging [21], is a network architecture that utilizes edge devices to execute considerable amounts of computation (edge computing), storage, and communication locally before sending them across the Internet backbone to the rest of the world.

Fog computing is a type of distributed computing in which a large number of "peripheral" devices are connected to a central server. (The term "fog" refers to the edge or periphery of the cloud.) Instead of sending all of this raw data to cloud-based servers for processing, fog computing aims to do as much processing as possible on computing units co-located with data-generating devices, resulting in processed data being transferred rather than raw data and lower bandwidth needs. Processing locally rather than remotely reduces the latency between input and response, which is a substantial gain, because the processed data is likely to be requested by the same devices that provided it. This isn't a new concept: special-purpose hardware has long been utilized to reduce latency and relieve strain on a CPU's resources in traditional computing settings.

3.3.3 Edge Computing

Data localisation is a fundamental factor for any edge computing solution because one of the key aims of the edge computing paradigm is to minimize the amount of data carried from the network edge to the cloud, allowing for lower latency scenarios.

Data pipeline deployments in edge computing are ideally suited for scenarios where the producer of data is mostly centralized: for example, a web server recording and processing the network traffic for further analysis (clickstream data) [22]. However, with the growth of ubiquitous computing, vast volumes of data are created by devices (sensors, personal devices, automobiles, etc.) at the edge of the network. With the number of such devices rising at a quick pace, the old approach for making use of centralized cloud resources becomes untenable due to the high costs of transporting data from the edge of the network to the cloud and tighter latency requirements.

3.3.4 Storage as a Service (StaaS)

Storage as a Service (STaaS) is the practice of storing data on public cloud storage facilities. When you can match data types to cloud storage solutions, using STaaS is more cost-effective than building private storage infrastructure. Benefits of cloud storage or Storage-as-a-Service(STaaS) are explained in detail in Section 4.1.

Organizations that use STaaS often store and back up their data on a public cloud rather than on-premises. Public cloud storage can also be used for STaaS with a variety of alternative storage strategies. Backup and restore, disaster recovery, block storage, SSD storage, object storage, and bulk data transfer are all examples of these types of storage systems. In the event of data loss, the cloud serves as a disaster recovery site for data that has been safely backed up. Data from virtual computers can be protected and replicated as part of disaster recovery (VMs). Customers can use block storage to create low-latency I/O volumes. Typical applications for SSD storage include read/write and I/O intensive tasks. Storage systems utilized in data analytics, disaster recovery and cloud computing have a tendency to be slow.It is possible to produce and configure saved data fast with the aid of cold storage. Transferring large amounts of data will necessitate the usage of disks and other technology. [23].

3.4 Data Locality

In a distributed system, data locality is critical for data processing performance. It refers to bringing computing closer to the data, which is frequently less expensive and faster than transferring data closer to the calculation. Due to the nature of dealing with enormous amounts of data, it is necessary to pool the resources (network, RAM, CPU, and disk) of numerous machines in a **distributed system**. A desirable property of distributed systems is that they hide the complexity of the distributed nature of the resources behind a single interface, making the entire system appear to be a single entity (e.g. cloud storage systems like Amazon S3), making individual hosts backing the distributed systems more difficult to leverage. The fact that a CPU may only execute data stored in the same machine's memory is a key invariant of modern computer design. As a result, data transfer between devices has become an essential part of every big data system. Apart from standard communication protocols that rely on the operating system's network stack (e.g. TCP/IP-based protocols), more efficient protocols have been designed for use cases where latency is a critical factor. RDMA (Remote Direct Memory Access) [24] is one such protocol that enables the transport of data stored in the memory of one computer to another using specialized network cards without impacting the CPU or the operating system kernel.

Network traffic and the latency associated with data transmission between computers may have an impact on the overall cost and performance of a big data processing due to the large volume of data. As a consequence, the capability of the work to be distributed in a way that minimizes data transfers (data locality) has been explored and implemented into a number of studies. Even for centralized deployment systems (such as cloud deployments), data locality has been shown to be effective in reducing costs and execution times [25, 26]. For instance, Apache Spark [27] may leverage Hadoop File System (HDFS) metadata and prior knowledge about the outputs of previous operations to minimize data transfer. Locality of data is just one factor that might be considered while scheduling operations. Other concerns, such as load distribution and heterogeneity of available resources among nodes, must be balanced against data locality in order to perform the operation efficiently [28]. [29] and [30] provide complex scheduling algorithms for balancing data transmission reduction and load distribution.

3.5 Domain-Specific Languages

A 'Domain-Specific Language', or 'DSL', is a computer language that has been specialized for one specific domain, as its name suggests. It is established upon a language that serves as a host, such as C, Javascript, Java, etc. The host language offers a jointed structure or syntax and connotations to represent notions, ideas, concepts, patterns, and behaviors within a specific domain. Maximilien et al. [31] stated in their summary of this broad topic that the designing or implementation of a DSL usually provides help towards attaining top-level linguistic constructs, natural and straightforward syntax, the convenience of programming, code genesis, and terse code. Within the research societies and forums of programming languages and software engineering, DSLs serve as an integral tool and method to diminish the complications in programming along with enhancing productivity, without compromising on efficiency[32]. On the bigger picture, the fundamental goal of DSLs is to attain the notion of metaprogramming, which realizes the idea of a computer program that is designed to produce, study, interpret, analyze and modify other programs, and go as far as transforming[33] itself while executing the tasks mentioned above. An Internal DSL is a DSL that requires itself to be incorporated within the host language. An External DSL is a DSL that has its own syntax and does not need to be embedded in another language. [34]. One of the most significant benefits of DSLs is that, because they are specialized to a given application domain, they may take advantage of domain expertise to increase productivity and efficiency. [35].

It is well acknowledged that creating custom DSLs and the model-driven workflows that go with them is a difficult endeavor that should be approached in an iterative manner [36]. As a result, domain-specific languages can be used for a wide range of purposes. They can be utilized in a variety of ways and by a variety of persons. Some DSLs are intended for programmers and so are more advanced, whereas others are intended for non-programmers who want to use less geeky notions and terminology.

3.6 Example Big Data Pipeline

To help the understanding of big data operations that can span resources spanning edge and cloud deployments, a fictitious scenario influenced by actual world patterns is used. It is assumed that the data is stored from multiple streams, which will be further processed in the big data pipeline. It can be seen in Figure 3.2, data from multiple streams is gathered and stored in a cloud storage provider. A data communication medium is setup between the cloud storage and the data pipeline server. For data transfer, either HTTP or FTP protocols can be used. A message queue is implemented on the data pipeline server for inter-step communication. The important thing to note here is that the integration can either be done with single cloud storage or multiple. For example, it is possible to store data collected from various streams in the storage facility by Amazon, but the data after processing is stored in Azure, or Google cloud storage.

An example big data pipeline, shown in Figure 3.2, is composed of the following steps:

1. **FetchURL:** This step will fetch URLs and will validate them using HTTP protocol. Once the URL is validated, it will set a flag against the URL in the Database. The operations involving in this step are not entirely storage related.
2. **Fetch Articles:** This step will download only the validated URLs and will fetch/scrap articles from each of the URL in a sequence. Again, HTTP protocol is used for fetching articles from the RSS feed.
3. **Filter Articles:** The operation involves in this step are entirely storage related. It fetches the articles in the chunk of one thousand each, cleans the data by stripping out all the HTML tags and then stores the processed text data back to the storage.

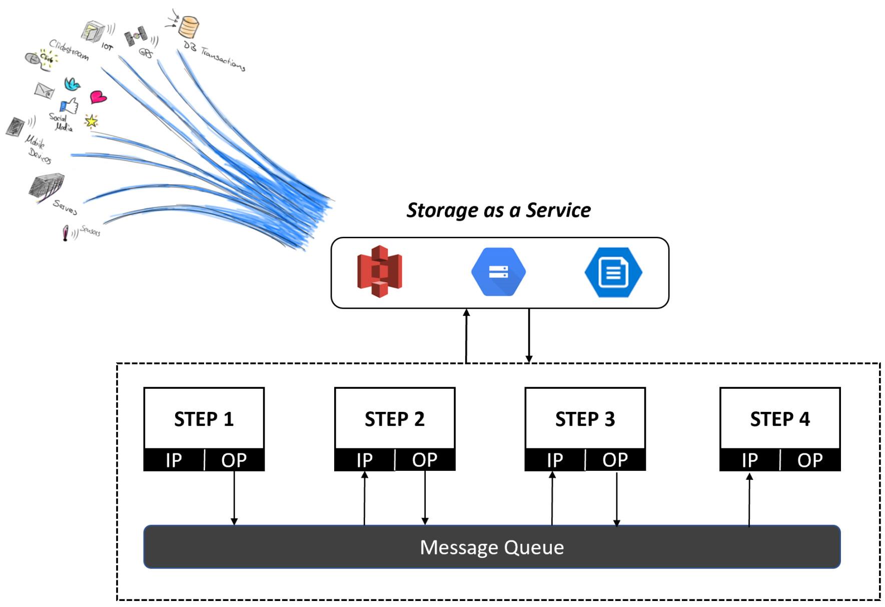


Figure 3.2: Example big data pipeline

Chapter 4

Related work

Storage as a service is quickly becoming the preferred approach for all small and medium-sized enterprises. This is due to the fact that storing data online rather than locally provides a slew of benefits for professional users.

The scientific community has extensively acknowledged the necessity to use cloud computing to execute scientific workflows/pipelines [37]. Many studies investigated and proved the viability of employing cloud computing for deploying big data pipelines in terms of both cost [38] and performance [39]. Abouelhoda et al. propose Tavaxy, a system that enables seamless integration of the Taverna system with Galaxy processes based on hierarchical workflows and workflow patterns [40]. Tavaxy provides an interface for creating a cluster on the AWS cloud and using it to perform processes. Wang et al. [41] present early results and experiences in enabling interaction between Kepler SWFMS and the EC2 cloud. Antonio et al. [42] talks about Hybrid multi-cloud storage systems and different data transfer techniques in general; however, it is not discussed with respect to big data pipelines.

These approaches discuss the benefits and possibilities of deploying big data pipelines in cloud infrastructure. But they do not discuss the possibility of hybrid big data pipelines. In addition to that, since they focus on one cloud provider each, the possibility of hybrid big data pipeline implementation with multi-cloud platforms has not been discussed.

4.1 Storage as a Service advantages in terms of Data Availability, Integrity, and Security

The explosion of data traveling through the network is being caused by the rapid expansion in the number of connected devices, including sensors, mobile, wearable, and other Internet of Things (IoT) devices. Gosh and Grolinger [43] presented a technique to combine Edge and Cloud computing in order to process data closer to the edge node. In this project, they carried out feature learning operation with deep learning and found out that on the edge, data can be reduced by upto 80% without any significant loss in accuracy. Ricardo et al. [44] developed a service for StaaS in Multi-Cloud Environment. In this approach, they did not focus on the performance, but the confidentiality and integrity of the data, i.e how it can be enhanced using Storage as a Service.

The cloud is an example of a centralized infrastructure system that stores data, runs application logic, and handles data and analytics duties while being located far away from end users and data sources. Because of their huge compute capabilities, excellent scalability and durability, a pay-per-use pricing mechanism, and cheap initial cost, they are very advantageous [45]. The purpose of Edge Computing (EC) is to reduce network traffic by moving computing closer to data sources rather than farther away. As an example, mobile edge computing (mobile EC) offloads computation and data storage to the edge (for example, base stations) in order to minimize network latencies, reduce battery consumption, enhance user experience, and add location awareness [46] has been intensively researched. When it comes to content delivery and caching, EC is particularly well suited, since it is particularly effective for applications that need very low latency [47].

Data Availability

The accessibility of your data is critical to your business's profitability and reputation among customers. If clients are unable to access your online presence due to a lack of data availability, they are likely to visit the website of a rival. While the great majority of cloud service providers use effective data backup and recovery solutions, backing up and restoring data is just a part of what you genuinely want. Data availability is a distinct area of focus since it is necessary before any storage can occur [48]. Cloud service providers offers services in multiple regions. A region is made up of one or more data-centers that are near together. It gives you the flexibility and capacity you need to cut down on client delay. With a full compliance solution, it also protects data residency. For Azure cloud, region pairs are at least 300 miles apart. Cloud service providers does automatic replication data. They also provide prioritized region recovery in the even of outage. Furthermore, in case of system update, updates are rollout sequentially to minimize downtime [49].

Azure cloud and AWS both have several availability zones to improve data availability. Within the same region, they are physically independent data-centers. Each data-center has its own power, cooling, and networking systems. Private fiber-optic networks link them together. Availability zones guard against data-center failure-related downtime. Figure 4.1 shows the concept of availability zones in an Azure region.

Data Redundancy

Data redundancy is the process of making several copies of your data, systems, and equipment in order to ensure that if your cloud service becomes damaged or

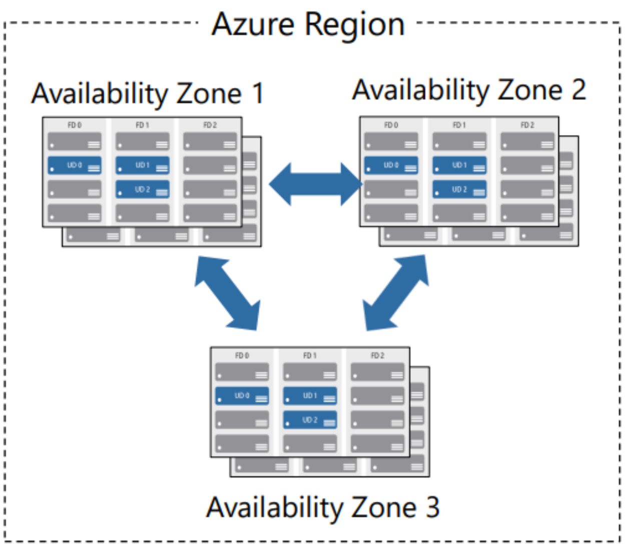


Figure 4.1: Availability zones in a region [49]

unavailable, you will have rapid and safe access to backup copies of your information [50].

Failures are a possibility in any virtual system but are more prevalent in cloud-based systems. Additionally, periodic maintenance, upgrades, and scheduled downtime are required. If you deal with a reputable cloud storage provider, you can be certain that your data is being stored on several computers and in line with any industry requirements (for example, some businesses may be required to have up to three copies of all data). The issue is more sophisticated than just duplicating data. Redundancy ensures that your company's data may be accessed at any time, regardless of whether an issue was anticipated or occurred unexpectedly [50].

StaaS Security

Additionally, the nature of these risks varies between STaaS providers and companies or individual customers, ranging from simple platform hardening and antimalware efforts to multitenancy. Customers can create vulnerabilities that a CSP has little control over [51].

Multitenancy

Processing and storage resources are separated from the underlying hardware in a cloud environment and made available in virtual pools via virtual machines (VMs) or containers. On the same physical server, many virtual machines and containers may operate. Your data and applications often share bare-metal resources with those of other customers. This is referred to as multitenancy since several tenants or customers share physical resources. Vulnerabilities in the workloads of another tenant may expose your workloads to hazard.

Workload segregation is critical to resolving the issue. While virtual machines and containers are automatically isolated, additional hardware-based precautions may also be beneficial.Intel® Software Guard Extensions (Intel® SGX), for example, is designed to construct trusted memory enclaves inside a platform to protect and safeguard data both in transit and during use. [52].

Customer Vulnerabilities

The main security issue for STaaS clients will be determining who within their company has read-only vs. read and write–level data access. Because CSPs have limited control over who has access to their customers' devices, it's critical to be wary of assaults like email phishing schemes that could compromise your point of entry. Another layer of security can be added by using strong passwords, two-factor authentication, and other recommended measures. [52].

Provider Practices

In STaaS, the cloud storage provider (CSP) who manages the cloud storage environment is primarily responsible for cloud security. The CSP is responsible for addressing vulnerabilities at both the hardware and software levels, as well as managing the human component by ensuring that all staff responsible for maintaining the cloud infrastructure are trustworthy and follow best practices. In this case, the consumer should educate themselves and be able to ask pointed queries when selecting a CSP [52].

4.2 Data Placement Techniques for Big Data

Zhang et al. [53] describes BerryStore, a distributed object storage system suited for the storing of huge quantities of little files. In a large web application, file sharing generates a large number of requests. BerryStore is built to manage these requests. The essential insight is that extraneous disk operations should be avoided when reading metadata. When it comes to small file storage, the fault-tolerant BerryStore system outperforms the more traditional NAS-based approach because to its substantially better throughput. Additionally, with the addition of a distributed coordinate controller, the system gains scalability, concurrency, and fault tolerance. The proposed mechanism does not provide any support for integration with the big data pipelines. In addition to that, *A Dynamic Data Placement Strategy for Hadoop in Heterogeneous Environments* [54], *An Improved Data Placement Strategy for Hadoop* [55] and *Improving MapReduce performance through data placement in heterogeneous Hadoop clusters* [56] are all data placement techniques for Hadoop.

Yuan et al. [57] researched the unique characteristics of scientific cloud operations and proposed a clustering data placement strategy capable of dynamically moving application data across data centers based on dependencies. Simulations on their cloud workflow system SwinDeW-C shown that our data placement strategy may significantly reduce data traffic during the execution of the process. The build-time algorithm reduces the amount of data acquired, while the run-time algorithm ensures a balanced distribution of data and may minimize data transportation costs via data reallocation, even when fixed-location data exists in the system. The suggested system is heavily platform dependant since it only works for Hadoop.

Er-dun et al. [58] addresses the issues connected with scientific workflows in the cloud computing environment, specifically the load balancing of datacentres. After examining the storage capacity of data centers, data transit patterns, and datacentre loads, they came with a workable data placement strategy using a genetic algorithm. In compared to other data placement strategies, the genetic algorithm-based data placement methodology performs well in terms of data center load balancing and data movement volume. In order to arrive at a better job placement strategy, further research on how to minimize task execution times and integrate them with scheduling tasks is required. While the proposed technique is effective, it does not address any functional requirement from the owners or developers of big data pipelines. It is limited to the number of datasets and the number of movements.

4.3 How can StaaS be integrated with DSL for big data pipelines?

Ivens et al. [59], from the University of Waterloo, carried out an extensive survey on the domain-specific languages for big data in the field of machine learning. This survey covers seven languages released in the literature or being used in the domain of Machine Learning in Big Data, and described them using a classification created from publications about DSL found in the literature. The major goal of this survey is to provide detailed knowledge to the domain experts so that they can make a sound decision while choosing the language. Moreover, in terms of the commonly used construction steps/process to develop a new DSL, Tomaz et al. [60] provides a preliminary study on different implementation techniques/approaches of domain-specific languages. It discusses the advantages and disadvantages of these approaches, but it does not provide any solid outcome that which technique is the best. This paper states the empirical results on the productivity of end-user, which is measured based on the number of lines of code needed to describe the DSL. It also discusses the error-reporting and debugging support in each implementation. Although both studies are focused on DSLs related to big data workflows, but none of them provides much information on the domain-specific languages for workflow specification in cloud infrastructure and date storage methods.

DSLs that are currently being used have a very limited scope. For example:

1. A DSL developed by Fazle Rabbi and Wendy MacCaull for Workflow Development [61] is focused more on providing an easier way to query data from the storage system. It has only one key element that is directly relevant to the workflow definition and that is dynamic task scheduling. It takes two parameters *delay* and *duration*. Duration represents the time for each step or task, whereas delay is the time between two tasks.
2. Mathias Slawik and Axel Küpper [62] developed a domain-specific language for business big data workflow in a cloud environment. They focused mostly on the challenge of connecting the workflow to the cloud. Figure 4.2 shows an example of vocabulary definition, whereas the Figure 4.3 shows the service description example. When this DSL provides an efficient way to solve the challenge of connecting to the cloud, it does not provide any vocabulary to actually define the individual workflow steps and their flow.
3. A few addresses the challenges such as scheduling [63] and connecting to cloud [64], but they do not cover any cloud infrastructure elements in their vocabulary.
4. DSLs that describe big data workflows covers elements such as containers, communication medium and scheduling [65], but the meta-model does not have any element to describe the storage system, network, ports and build parameters, and resource provisioning/management.

Mathias Slawik and Axel Küpper [62] developed a domain-specific language for business big data workflow in a cloud environment. They focused mostly on the challenge of connecting the workflow to the cloud. In order to resolve that problem, they provided the vocabulary to the DSL to generate a model based on the type of service the user needs, i.e Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Software as a Service (Saas). Figure 4.2 shows an example of vocabulary definition, whereas the Figure 4.3 shows the service description example such the workflow needs to be connected to Google Drive and the type is SaaS. While this DSL provides an efficient way to solve the challenge of connecting to the cloud, it does not provide any vocabulary to actually define the individual workflow steps and their flow.

```
type :cloud_service_model
cloud_service_model :saas
cloud_service_model :paas
cloud_service_model :iaas
service_properties do
   string :service_name
   cloud_service_model
end
```

Figure 4.2: Vocabulary definition example

Aravind et al. [63] in their paper proposed techniques of scheduling big data workflow tasks in the cloud environment. They explain how the processing time and resources can be divided into multiple steps to minimize the cost. F# Workflow DSL [64] provides an efficient way to describe a big data workflow. It gives

```
service_name "Google␣Drive␣for␣Business"
cloud_service_model saas
```

Figure 4.3: Service description example

the ability to create logical steps and their connection, but it does not cover any cloud infrastructure elements in its vocabulary. Nikolov et al. [66] suggested a DSL for scalable execution of large data operations using software containers. They put a lot of emphasis on using a domain-specific language to define workflows. A communication medium and triggers can be used to design a workflow stage. However, there is no element in the meta-model that describes the workflow's storage system. Network, ports, and build parameters can also be supplied because the DSL outputs a YAML file for cloud orchestration tools. Furthermore, it lacks the ability to provision and manage resources.

Part III

Smart Data Placement Solution for Big Data Pipelines

Chapter 5

Problem Analysis

This chapter analyses the problem addressed in this thesis in two folds, namely in terms of criteria to evaluate storage service options and method to rank these options with respect to these criteria. In this respect, firstly, we provide a description and analysis of different available cloud storage service options and their pricing structures, followed by justification for the selection of a particular cloud storage option used as an input for the development of the ranking algorithm. Secondly, for ranking different cloud storage providers based on the selected criteria, the use of multi-criteria decision analysis (MCDA) method is discussed. Hence, the chapter also provides a detailed description and justification of the chosen method.

Collecting information about different available options will help in understanding the problem better, i.e., to choose the best among available cloud storage options. Because in order to rank different cloud storage services based on the user requirements, it is important to collect and then normalize data about the offered services and their pricing structures. For example, to estimate the total cost, it is required to take into account not just the volume of data to be stored but also the cost of network usage in terms of the concrete number of READ and WRITE operations as well as the amount of data transferred in Gigabytes. Moreover, each cloud storage provider has multiple options for storage services. For example, Azure has three different tiers, such as the hot tier, the cool tier, and the archive tier. Google Storage, on the other hand, has exactly the same structure for storage options, but with different names, which are standard, coldline, and archive. Amazon follows the same pattern but with different names. Similarly, network usage is further categorized into egress and ingress. When it comes to the selection of a storage facility, the user is given the option of multiple regions and availability zones. For data normalization, it is important to select common services from each of the providers. For example, each service provider has several storage tiers and different pricing models for each tier. So, one tier is selected from each storage provider with similar characteristics, and then data is collected for the evaluation model. This chapter contains brief details about the storage service providers and their relevant offered services, as well as justification for the selected services.

In what follows various criteria for cloud storage options are discussed and for the development of a ranking algorithm, the following steps have been identified:

- get details about the different available storage tiers;
- comprehend the role of network utilization in cloud storage and its implications for cost;
- understand the different available encryption methodologies and identify the most suitable for this problem;
- study the roles of availability zones and geographical regions so that an informed choice can be made at the time of deployment.

5.1 Justification for the selected parameters

A total of four parameters were selected for the evaluation of different cloud storage providers, which are cost, physical distance, network performance, and impact of server-side encryption. Cost is selected because it is an important parameter while selecting any service from a cloud storage provider. There are studies that rank cloud services based on their cost ([67], [68]). Physical distance is selected while keeping in mind the principle of data locality. The closer the data, the better the performance is. So finding the closest storage facility is one of the problems this thesis resolved. Physical distance is calculated as a straight line between two points on the map, but the network infrastructure between two servers is not necessarily a straight line. It can be a mesh of wires, and there is a great possibility that the length of the actual network line is far longer than the actual physical distance between two servers. To overcome this challenge, a new parameter, network performance, is selected. The fourth parameter is the impact of server-side encryption. It has been selected based on the unstructured interviews and discussions with the field experts.

5.2 Storage tiers

5.2.1 Microsoft Azure Storage Cloud

Azure storage offers a variety of access tiers. Access tiers for Azure Storage include [69]:

- **Hot tier:** Data that is frequently accessed or modified is best stored in an online tier. The storage costs for the Hot tier are the highest, but access costs are the lowest.
- **Cool tier:** An online tier designed to store data that is accessed or modified only on a rare occasion. It is recommended that data in the Cool tier be kept for at least 30 days. In comparison to the Hot tier, the Cool tier has lower storage costs but higher access costs.
- **Archive tier:** Data that is rarely accessed and has flexible latency requirements, on the order of hours, can be stored in an offline tier. A minimum of

180 days should be kept in the Archive tier.

Azure storage capacity limits are set at the account level, rather than according to access tier [69]. You have the option of maximizing capacity usage in one tier or distributing capacity across two or more tiers.

5.2.2 Amazon Web Services

Amazon offers two file storage tiers: S3 (Simple Storage Service) and EBS (Elastic Block Store). For the most part, the main difference between them is in what they can be used for [70].

- **EBS:** EBS (Elastic Block Store) is only accessible when mounted to an EC2 (Elastic Computing Cloud) instance. To an EC2 instance, the volume mounted with EBS appears exactly like a hard disk partition on the server. With any file system, it can be used like a hard drive for storing data and writing or reading files. EBS has a standard limit of 20 volumes, each of which can hold 1 TB of data. A limitation of EBS is that it cannot be used concurrently by multiple instances. Once an instance mounts it, no other instance may use it. EBS data reads and writes almost instantly.
- **Simple Storage Service (S3):** In contrast, S3 can be used outside of EC2 as well. Retrieving files from an S3 bucket is as simple as using HTTP or BitTorrent protocols. S3 is a popular choice for many websites because it is accessible to HTTP clients, such as web browsers. S3 still has the upper hand when it comes to total storage capacity. S3 has a standard limit of 100 buckets, each of which can hold an unlimited amount of data. S3 can store multiple images of its contents, allowing it to be used concurrently by multiple users. A fascinating byproduct of this capability is something known as 'eventual consistency'. With S3, changes are not immediately written, which means that if you write something, it may not be the data returned by a read operation.

In short:

- EBS is only compatible with EC2 instances, whereas S3 is compatible with non-EC2 instances.
- Even though the EBS looks like a volume that can be mounted, the S3 needs extra software to read and write data. EBS can store less data than S3.
- A single EC2 instance can use EBS, but multiple instances can share S3.
- Typically, S3 has write delays, whereas EBS does not.

5.2.3 Google Cloud Storage

Google cloud storage offers three different tiers [71]:

- **Standard:** Standard tier does not have a minimum storage duration, and this type is also used in this thesis. Objects of other types can be deleted before they have been stored for the minimum duration, but at the time the

object is deleted, replaced, or relocated, you are charged as if the object has been stored for the minimum duration.
- **Coldline:** Data stored in Coldline Storage must be kept for a minimum amount of time.
- **Archive:** Data stored in Archive Storage must be kept for a minimum amount of time.

Selected tiers: For this thesis, hot tier from Azure, S3 from Amazon, and standard storage tier from Google is selected as they are most frequently used tiers with similar characteristics.

5.3 Network usage

Network usage is the amount of data read from or moved between the buckets. When Cloud Storage sends data via Egress, it's represented in HTTP response headers. An example of egress is reading data or metadata from a Cloud Storage bucket. HTTP requests sent to Cloud Storage are represented by Ingress. An example of ingress is the writing of data or metadata to a Cloud Storage bucket [71].

5.3.1 Data Egress

The term "data egress" refers to the act of sending data from a network to another. Data egress can be as simple as sending emails or transferring files to a cloud service or another location. An organization's network security might be jeopardized when sensitive data is sent to an unapproved destination [72].

Examples of common channels for data egress include:

- Email
- Web uploads
- Cloud storage
- FTP/HTTP transfers

5.3.2 Data Egress vs. Data Ingress

Data egress refers to outbound traffic from a network, whereas data ingress refers to traffic that originates outside the network and enters the network [73]. One way to think about it is as the amount and type of traffic that leaves a network and enters the outside world.

Egress filtering is the process of keeping tabs on outbound traffic for signals of malicious activity, such as phishing. It is possible to prevent sensitive data from being lost if malicious behavior is suspected or discovered. Egress filtering can also limit egress traffic and reject efforts to egress large volumes of data at once [72].

5.3.3 Azure network usage

The total cost of Microsoft Azure cloud storage is dependent on the following :

- Volume of data stored per month.
- Quantity and kinds of activities carried out, in addition to any costs associated with data transport [74].
- Data redundancy option selected.

5.3.4 Amazon network usage

Amazon s3 pricing depends on three things:

- To save things in the buckets of the S3 service [75].
- Requests and data retrievals. Amazon imposes fees for every requests made on the objects and buckets in your S3 storage. The expenses of an S3 request are determined by the type of request being processed and the number of requests that are being processed [76].
- Data transfer: Users of Amazon S3 storage are also responsible for paying for any bandwidth entering and exiting Amazon S3, with rare exceptions [77].

5.3.5 Google network usage

When you use cloud storage, you'll have to pay fees for the operations you carry out. A Cloud Storage operation is any action that modifies or retrieves data about buckets or objects. Class A, Class B, and free operations are all subclasses of one another [71]. Network usage charges apply for egress and are divided into the following cases:

- Network egress to other Google Cloud Storage buckets or services
- Specialty network services, when egress uses certain Google Cloud network products.
- When egress is out of Google Cloud or between continents, the item General network usage is used.

Selected options: Class A operations are the only ones considered in this thesis for Google. For Azure, no redundancy (LRS) and international data transfer rates are selected. For Amazon, the S3 storage type is chosen because it doesn't have any redundancy and has fast international data transfer rates.

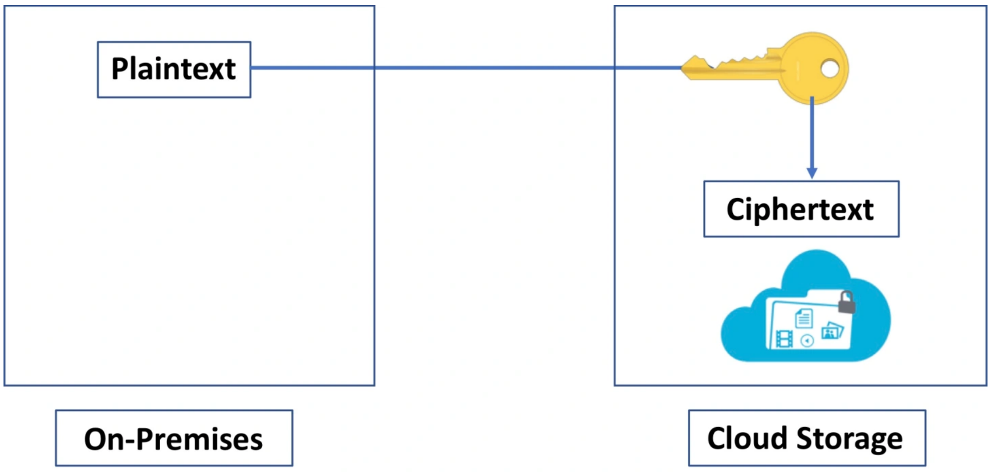


Figure 5.1: High-level diagram showing the process of Server-side encryption between On-premise servers and cloud storage using a server-managed key.

5.4 Server-side encryption

To prevent data from being read by unauthorized parties, the object storage service uses server-side encryption. Server-side encryption is offered by all three providers, but implementation details differ, particularly in the management of keys. Figure 5.1 shows the general scenario of the process of data being encrypted using a server-side encryption. Whereas the Figure 5.2 shows the same process for server-side encryption for aws s3 bucket.

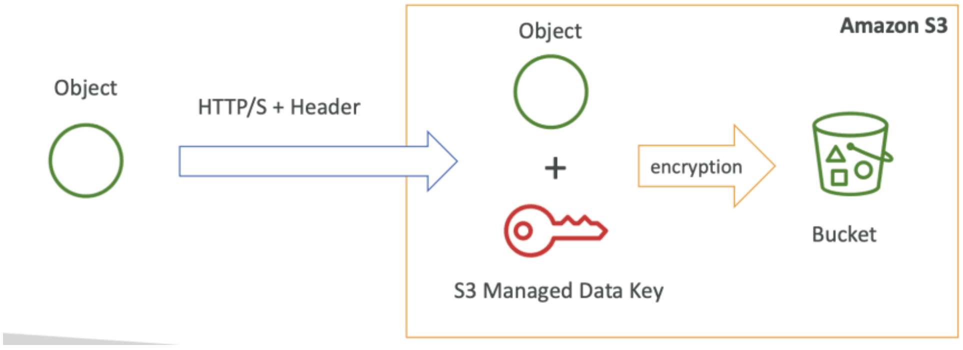


Figure 5.2: High-level diagram showing the process of Server-side encryption when an object is uploaded to AWS S3 bucket using HTTP protocol [78].

5.4.1 AWS S3 server-side encryption

There are many alternatives for both server-side and client-side encryption on Amazon S3. Server-side encryption is enabled by default for all items that are uploaded to S3. AES-256 with Galois Counter Mode (GCM) is used by AWS for all symmetric key encryption activities, on both the server and client sides. GCM authenticates encryption by adding a unique identifier to the ciphertext that ensures that the encrypted data has not been tampered with [78]. Except when the customer provides the encryption key, envelope encryption is utilized on both the client and server sides. For server-side encryption, Amazon S3 supports three options:

- Amazon S3-managed keys (SSE-S3)
- AWS Key Management Service (KMS) managed keys (SSE-KMS)
- Customer-provided keys (SSE-C)

SSE-S3 is the only storage and management solution that can handle both KEKs and DEKs simultaneously. The service takes care of all aspects of key management, including the routine rotation of keys, so there is no need for human participation in any of these processes. To accomplish this, S3 uses an AWS-managed Key Management Service (KMS) [78].

The steps involved in the encryption process for SSE-S3 are as follows:

1. The Amazon S3 server receives the data.
2. A one-time-only, one-of-a-kind Data Encryption Key is produced by the S3 service (DEK)
3. The DEK is utilized to encrypt the data that is uploaded.
4. After that, the DEK is encrypted with a KEK that is controlled and kept secure by the S3 service.
5. While the plaintext form of the DEK is removed from memory, the encrypted version of it is saved alongside the ciphertext data as metadata.

The decryption workflow is as follows:

1. Amazon S3 retrieves the encrypted DEK for the requested object and decrypts it using the associated KEK
2. S3 decrypts the ciphertext object using the decrypted DEK and then deletes the key from memory
3. The decrypted object is downloaded to the requesting client or application

5.4.2 Azure storage server-side encryption

Azure by Microsoft Blob Storage is a Microsoft Azure storage service. For the sake of this post, Block Blob Storage is the closest thing to Amazon S3 and Google Cloud Storage. Although Azure has nearly as many options for data encryption and key management as AWS, the details are frequently difficult to discover and frequently absent. [78].

Client-side encryption can be enabled by default for all objects uploaded to Azure, however both server-side and client-side encryption are supported. Storage Service Encryption refers to server-side encryption when it comes to Azure blob storage. Encryption of data or content is performed using AES-256 symmetric keys, whereas Key Encryption Key (KEK) encryption is performed using either symmetric or asymmetric keys, depending on who is creating and managing the keys [78]. Azure supports both types of encryption.

Storage Service Encryption supports using a KEK that is either:

- Microsoft's own internal key management infrastructure is used to administer the storage service.
- Customers store and manage their keys in Key Vault, which is a service from Azure that helps them store and manage their keys.

The encryption workflow for Storage Service Encryption is as follows:

1. Data is uploaded to Azure Blob Storage
2. Using a cryptographic library, Azure Blob Storage generates a one-of-a-kind one-time Content Encryption Key (CEK)
3. The uploaded data is encrypted using the CEK
4. It is then necessary to encrypt the CEK using an existing RSA public KEK, which can either be located on-site at the storage facility or in Azure Key Vault.
5. CEK information is retained alongside the encrypted ciphertext data; plaintext CEK is removed from memory.

The decryption workflow is as follows:

1. As soon as a user requests a piece of data, Azure Blob Storage receives the encrypted DEK and transfers it to the storage services' internal key management service or to Azure Key Vault.
2. The private key associated with the KEK is used to decrypt the CEK and send it back to Azure Blob Storage.
3. The plaintext CEK is used to decode the data.
4. When a client requests encrypted data, Azure Blob Storage transmits the decrypted data to the client.

Selected techniques: Each cloud storage provider gives multiple options for data encryption. They include server-side encryption and client-side encryption. Furthermore, for server-side encryption, different options are available, such as encryption with server-managed keys and client managed keys. Each option not only has an effect on the performance, complexity of the software but also on the cost. For this thesis, **server-side encryption** with **server-managed keys** option is selected as it requires minimal work from the user side and has no extra cost.

5.5 Availability Regions

Cloud data centers are located in *Availability regions*. In terms of latency, solution offerings, and costs, different regions offer a variety of options. Large service providers have availability zones all over the world. The term *Availability zone* refers to a specific geographic area in which a single data center is located. There is no single data center that is shared by multiple availability zones; instead, each zone has its own collection of data centers [79].

At the time of the data pipeline deployment, a user is required to select the region for storing the data. The new distance finding algorithm and the cloud storage ranking algorithm will give region as an output.

5.6 Multi-criteria Decision Analysis

Our decision-making process can benefit greatly from Multi-Criteria Decision Analysis, or MCDA. It is best suited to situations when a decision must be made between two or more choices. As an effective decision support tool, it helps us focus on what matters most, is logical and consistent, and is simple to use [80]. MCDA is an effective tool that can be utilized to solve complex challenges. It is particularly applicable to addressing circumstances that are represented as a choice among possibilities. It has all the elements of a useful decision assistance tool: it helps us focus on what is vital, is sensible and consistent [81]. At its core MCDA is useful for:

- Dividing the decision into smaller, more understandable parts
- Analyzing each part
- Integrating the parts to produce a meaningful solution

Since we are also dealing here with multiple alternatives of cloud service providers and multiple options on the storage facilities of each of these providers, MCDA is suitable for this project. MCDA enables groups communicate about their decision opportunity (the problem to be solved) in a way that allows them to consider the values that each regards as relevant when making a choice together. It also gives people a unique opportunity to think about and discuss complicated trade-offs between various options. As a result, it encourages people to re-examine their thoughts and revise their conclusions [80].

MCDA problems are comprised of five components:

1. Goal
2. Decision maker or group of decision makers with opinions (preferences)
3. Decision alternatives
4. Evaluation criteria (interests)
5. Outcomes or consequences associated with alternative/interest combination

For this project, VIKOR method is selected based on the generalised framework developed by Jankowski et. [82]. It is an innovative tool for choosing the MCDA approach that is most appropriate for the decision issue. The framework provided a methodological and practical framework for selecting appropriate MCDA procedures for a given decision situation. A collection of 56 available MCDA methods were analyzed, and a hierarchical set of method features and a rule base were created as a result. This analysis, principles, and modeling of uncertainty in the decision issue description enabled the development of a framework to support the selection of an MCDA approach for a given decision-making situation. The practical studies showed that the methods suggested by the proposed methodology were similar to those used by experts in real-life situations. The VIKOR procedure has the following steps:

5.6.1 VIKOR Steps

Step 1.

Determine the best f_i^* and the worst $f_i^\wedge$ values of all criterion functions,

$$i = 1, 2, ..., n;$$

$$f_i^* = max(f_{ij}, j = 1, ..., J),$$

$$f_i^\wedge = min(f_{ij}, j = 1, ..., J),$$

if the i-th function is benefit;

$$f_i^* = min(f_{ij}, j = 1, ..., J),$$

$$f_i^\wedge = max(f_{ij}, j = 1, ..., J),$$

if the i-th function is cost.

Step 2.

Compute the values S_j and R_j,

$$j = 1, 2, ..., J,$$

by the relations:

$$S_j = sum[wi(f_i^* - f_{ij})/(f_i^* - f_i^\wedge), i = 1, ..., n],$$

weighted and normalized Manhattan distance;

$$R_j = max[wi(f_i^* - f_{ij})/(f_i^* - f_i^\wedge), i = 1, ..., n],$$

weighted and normalized Chebyshev distance; where wi are the weights of criteria, expressing the DM's preference as the relative importance of the criteria.

Step 3.

Compute the values

$$Q_j,_j = 1, 2, ..., J,$$

by the relation

$$Q_j = v(S_j \breve{\ } S^*)/(S^\wedge - S^*) + (1 - v)(R_j - R*)/(R^\wedge - R^*)$$

where

$$S^* = min(S_j, j = 1, ..., J), S^\wedge = max(S_j, j = 1, ..., J),$$
$$R^* = min(R_j, j = 1, ..., J),$$
$$R^\wedge = max(R_j, j = 1, ..., J);$$

and is introduced as a weight for the strategy of maximum group utility, whereas 1-v is the weight of the individual regret.These techniques may be imperiled by v = 0.5, and in this case, v is adjusted as = (n + 1)/ 2n (from v + 0.5(n-1)/n = 1) because the criterion (1 of n) associated to R is also included in S.

Step 4.

Sort the options by the values S, R, and Q, starting with the lowest value. The end result is three ranking lists.

Step 5.

Propose as a compromise solution the alternative A(1) which is the best ranked by the measure Q (minimum) if the following two conditions are satisfied: C1. "Acceptable Advantage": Q(A(2) – Q(A(1)) >= DQ where: A(2) is the alternative with second position in the ranking list by Q; DQ = 1/(J-1). C2. "Acceptable Stability in decision making": The alternative A(1) must also be the best ranked by S or/and R. This compromise solution is stable within a decision making process, which could be the strategy of maximum group utility (when v > 0.5 is needed), or "by consensus" v about 0.5, or "with veto" v < 0.5). If one of the conditions is not satisfied, then a set of compromise solutions is proposed, which consists of: - Alternatives A(1) and A(2) if only the condition C2 is not satisfied, or - Alternatives A(1), A(2),..., A(M) if the condition C1 is not satisfied; A(M) is determined by the relation Q(A(M)) – Q(A(1)) < DQ for maximum M (the positions of these alternatives are "in closeness").

The obtained compromise solution could be accepted by the decision makers because it provides a maximum utility of the majority (represented by min S), and a minimum individual regret of the opponent (represented by min R). The measures S and R are integrated into Q for compromise solution, the base for an agreement established by mutual concessions.

Chapter 6

Smart Data Placement Solution Design

6.1 Preliminary work

In [66] (see appendix also), we proposed a solution for specifying big data workflows at a high abstraction level, which allows for the separation of design and runtime components and still allowing for scalable process execution. The approach is based on usage of software container technologies, message-oriented middleware (MOM), and a domain-specific language (DSL) that enables highly scalable workflow execution and abstract workflow formulation. Based on the results of our experiments, we show that the proposed technique may be applied in practice to the definition and scalability of large-scale data operations. Our method was evaluated against Argo Workflows, a well-known tool in the field of big data workflows, in terms of scalability, and a qualitative assessment of the suggested DSL and overall approach was provided in comparison to the current literature. The proposed DSL's design stresses the separation of concerns between workflow's structural features and its execution. As a result, we believe that the DSL should be used by non-technical users and so should avoid difficult structures. Therefore, in the creation of the language, another method is to introduce linguistic ideas only when they are absolutely necessary [83]. As a result, the DSL's initial version only includes notions that are relevant to our real-world use cases. The meta-model of the prosed DSL is shown in Figure 6.1.

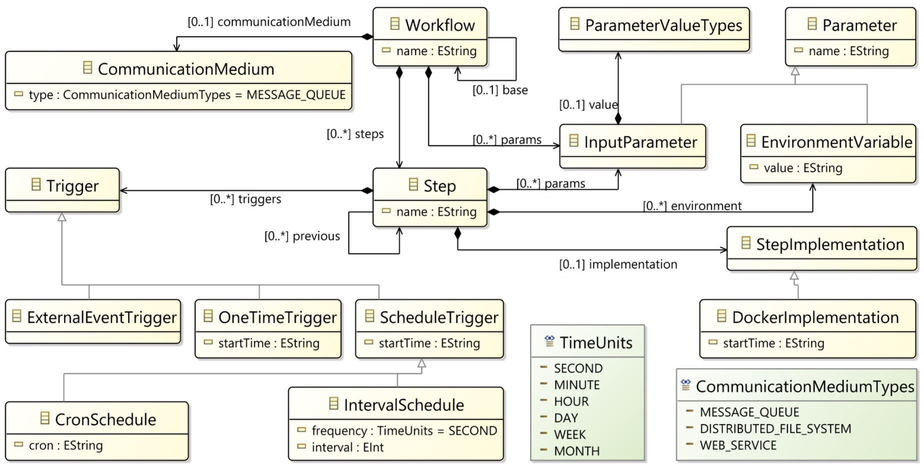


Figure 6.1: Meta-model for the proposed DSL for representing big data pipelines [66]

In [84] (see appendix also), we have proposed a unique architecture and a prototype implementation for container-centric big data workflow orchestration systems. Data locality is taken into consideration using a flexible model that takes into account the physical distance between hosts dispersed across the computing continuum in the method proposed. Long-lived containers in our system are better suited for processing small, frequent data units, yet they are reused to process many units instead. Furthermore, it extends the principles of isolating processing stages in different containers to the data management element of big data processes. As such, the logic needed to communicate with data management systems is encased in containers, giving the same benefits as for processing logic (technology agnostic solution, isolation, lightweight, etc.). Hence the importance and viability of 1) software containers, 2) domain-specific languages, 3) data-locality has been tested and demonstrated.

6.2 Novel architecture

Both studies explained in Section 6.1 proposed an architecture for a big data pipeline with a shared file volume setup on the same server as of the compute steps. Steps are encapsulated in software containers and a message queue is implemented for communication between different steps. The big data pipeline architecture design idea is shown in the Figure: 6.2. The data is stored on a local storage, hence no specific data transfer medium is required. Trigger model is also proposed to start and stop the processing for each compute step.

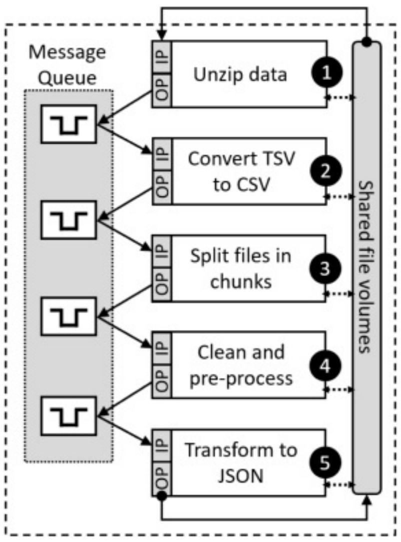


Figure 6.2: Prototype big data pipeline [66]. Shared file volume is setup on the same server as of the compute steps. Steps are encapsulated in software containers and a message queue is implemented for communication between different steps.

In this thesis, we propose an improved and extended architecture as shown in Figure 6.3 where the compute steps are still encapsulated in software containers with a communication medium attached, the local storage is replaced with hybrid cloud storage as a service. For this thesis, we have considered and evaluated three cloud storage providers: Amazon, Azure, and Google.

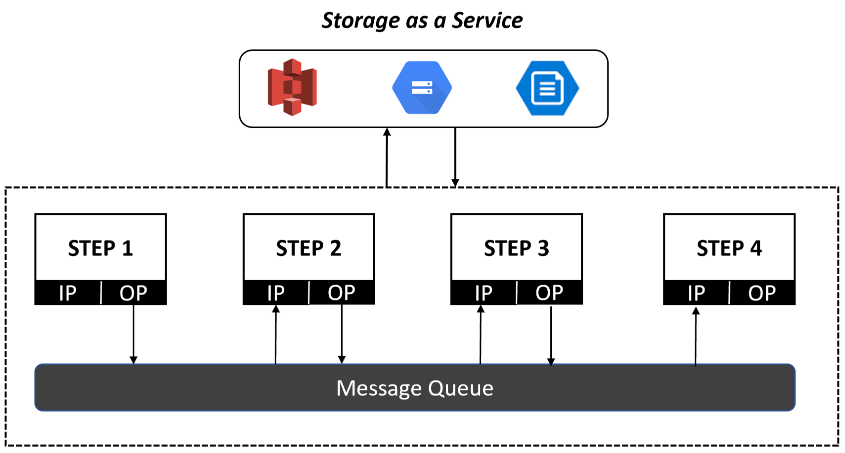


Figure 6.3: Big data pipeline proposed architecture with integration to Storage as a Service (IP: Input, OP: Output). Steps are encapsulated in software containers and a message queue is implemented for communication between different steps. Local storage is replaced with the cloud storage or STaaS.

6.3 Ranking method

There are several parameters that affects the choice of cloud storage such as the cost of storage space, security, performance and the location. These parameters are inter-dependant on each other. So there is a possibility that the cloud service provider that fulfills security requirements does not have the best network performance, or the one with closest to the data pipeline server is more expensive. Another scenario involves more functional requirements. Cost structure for cloud storage is not simple. So a data pipeline with an extensive data transfer operations and relatively less storage requirements, in this case, a cloud storage with the least expensive storage does not necessarily have the least expensive bandwidth as well.

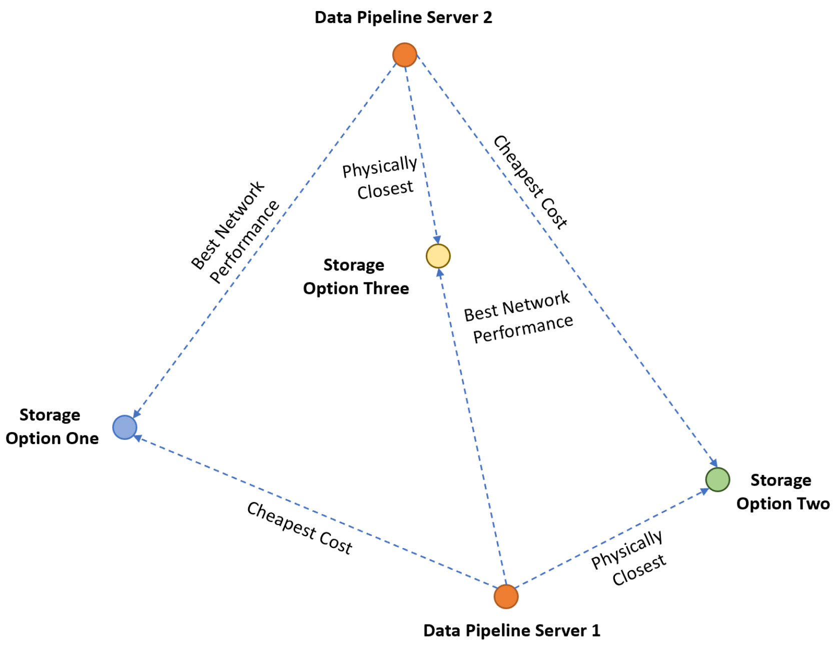


Figure 6.4: Relation between cost, network performance and physical Distance. Two data pipeline servers and three cloud storage options, each with it's own pros and cons.

The focus of the proposed method is mainly on data locality, that is, smart data placement to achieve maximum performance output. But different locations with different cloud storage providers have different costs. In addition to that, all data centers have different network infrastructure, so that network performance can vary between different data centers. Figure 6.4 shows the inter-dependency between different parameters. It shows a hypothetical scenario where there are two data pipeline server and three different data storage options, and each storage option has different positive affect. The challenge is to decide which criteria is best suited to the situation and user requirements. Hence, VIKOR, a Multi-criteria decision analysis method is used to rank different scenarios based on the weights set by the user. For this very evaluation model, four different parameters are selected in addition to the user weights. The parameters are as follows:

- Physical Distance
- Cost
- Network Performance
- Server-side encryption

The cost is further dependant on multiple parameters which are storage space, network usage, number of read and write operations. The model involves a comprehensive implementation of each parameter. Following are the utilisation details for all the parameters and the MCDA model.

6.3.1 Physical Distance

Data locality is a great approach to address the *bottleneck* problem of data having to travel through slow networks. Data do not need to circulate between a VM and remote hosts all the time as necessary data sets are available locally. Also, it keeps the I/O operations within a single physical node. Thus, *data locality* allows to avoid the processing overhead of the network stack, thereby considerably lowering latency because much fewer data require transferring via network. In order to do that, a software tool is developed to find out the physically closest storage facility available to the data pipeline server. The distance is calculated in real time. The tool analyses:

- two hundred and sixteen (216) storage facilities of Amazon Web Services including Amazon Gov Cloud

and

- two hundred and eighty-three (283) Azure cloud facilities.

It calculates the physical distances in Kilometers (KM) between the data pipeline server and the storage servers based on the longitude and latitude.

Pseudo Code

The pseudo code for the software program is as follows:

Algorithm 1 Psuedo Code for finding closest storage facility

```
1: procedure FindPipelineServerLocation
2:     serverIP ← IP Address of the data pipeline serverstring
3:     return $Long$ ← Longitude pipeline server, $Lat$ ← Latitude pipeline server
4: end procedure
```

Algorithm 2 Psuedo Code for finding closest storage facility

```
1: $AWSRegionsDetail \leftarrow$ Details of all AWS regions
2: **repeat**$r \in \mathcal{R}$
3:
4:     **procedure** FINDLONGLATOFEACHINSTANCE
5:         **return** *Long.* ← Longitude of instance, *Lat.* ← Latitude of instance
6:
7:         **procedure** FINDDISTANCE
8:             *LongPS* ← Longitude of pipeline server
9:             *LatPS* ← Latitude of pipeline server
10:            *Long* ← Longitude of AWS instance
11:            *Lat* ← Latitude of AWS instance
12:            **return** $distance$
13:        **end procedure**
14:        **if** *shortest()* **then**
15:            $FetchRegionDetails()$
16:        **end if**
17:
```

Algorithm 3 Psuedo Code for finding closest storage facility

```
1: $AzureRegionsDetail \leftarrow$ Details of all Azure regions $AzureRegion \in AzureRegionsDetail$
2:
3: **procedure** FINDLONGLATOFEACHINSTANCE
4:     **return** *Long.* ← Longitude of instance, *Lat.* ← Latitude of instance
5:
6:     **procedure** FINDDISTANCE
7:         *LongPS* ← Longitude of pipeline server
8:         *LatPS* ← Latitude of pipeline server
9:         *Long* ← Longitude of AWS instance
10:        *Lat* ← Latitude of AWS instance
11:        **return** $distance$
12:    **end procedure**
13:    **if** *shortest()* **then**
14:        $FetchRegionDetails()$
15:    **end if**
16:
```

Software Flow Chart

Figure 6.5 shows the software flow chart for the software component to find the physically closest storage facility for AWS, Azure, and Google. The software first finds out the longitude and latitude of the data pipeline server; after that, it finds the longitude and latitude of each storage server and calculates the physical distance between the two points.

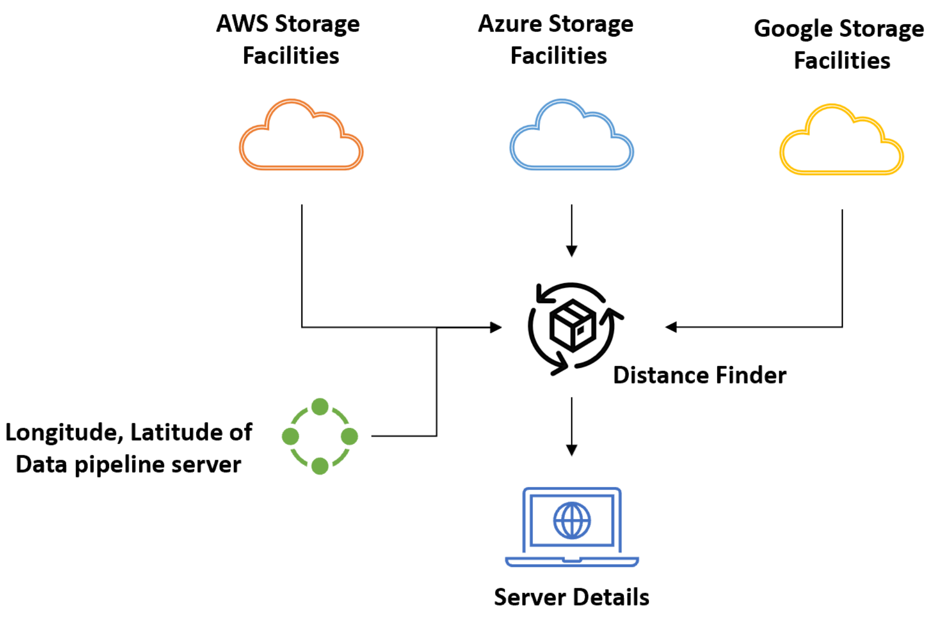


Figure 6.5: High-level design of a software component for physical distance calculation. After finding out the longitude and latitude of data pipeline server, it is then compared with each of the storage facility provided by Amazon, Azure and Google. Data is fetched for the cloud service providers using REST API. After calculation, the server details with the shortest distance is given as an output.

6.3.2 Cost of storage service

The data can be stored in the cloud without the need for on-premise storage infrastructure. As long as you have an internet connection, you can access the information stored in the cloud at any time and from any location.Despite the fact that cloud storage's primary selling point is its low cost, it's crucial to know exactly what those fees are and how to avoid an unexpected bill. Cloud storage costs might vary widely depending on a company's needs. Everything from retrieval frequency and storage capacity to network bandwidth affects the cost.

In simple words, cost is calculated based on four variables. The storage space itself, the Out Bandwidth and the number of READ and WRITE operations. When it comes to the cost of cloud storage, it's not just about the price per gigabyte (GB). Additionally, there are fees involved with moving data between the cloud and on-premise systems. In many services there are two costs: one per-gigabyte cost each time servers in different domains communicate with each other, and another per-gigabyte cost to transfer data over the internet.

Table 6.1 shows the data for Azure Cloud Storage and AWS S3. The cost is for EU West region. Data has been arranged so that the price can be compared easily between different services.

Data Preparation for Cost Calculation: Next step involved in the process is data normalization. Normalization is the process of organizing data, and to arrange and prepare the data based on which a software script can be developed to calculate the total cost of the cloud storage based on the user requirements. Normalized data is shown in Table 6.1.

Cost Calculation: This step involves the development of the software script to calculate the estimate cost of the storage service based on the user requirements. The input parameters are as follows: Requirements[Space](GB), Requirements[Bandwidth](GB), Requirements[WRITE](1000), Requirements[WRITE](1000). The result is the estimate of total cost in US dollars. The process is shown in Figure 6.6. The output received from this step would one of the integral parameters of the new proposed model. The script was tested for different set of user requirements and output was analysed in Section 8.3.

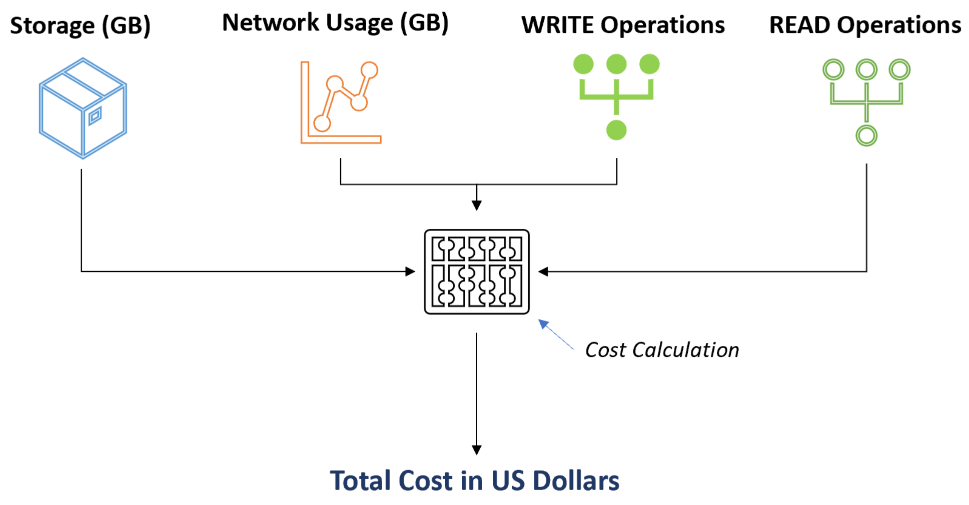


Figure 6.6: High-level design of a software component for cost estimation. User-requirements in terms of storage, bandwidth, READ and WRITE operations in given as an output, and an estimate for the total cost in US dollars is given as an output.

	AZURE (EU West)		**AWS (EU Stockholm)**
Space TB	Price Per GB		Price Per GB
0-50	0.02		0.023
51-500	0.0188		0.022
>500	0.018		0.021
Bandwidth IN	Price Per GB		Price Per GB
ALL	0.00		0.00
Bandwidth OUT	Price Per GB	Bandwidth OUT	Price Per GB
0-5	0.0	0-5	0.09
6 – 15	0.08	0-10	0.09
16 – 55	0.065	11 - 050	0.085
56 - 155	0.06	51 - 150	0.07
156 - 500	0.04	>150	0.05
Operations	Price Per 10,000		Price Per 10,000
WRITE	0.07		0,05
READ	0.006		0,04

Table 6.1: Cost Data Collection

6.3.3 Network Performance

Customers' perceptions of the quality of a network's service have a direct impact on network performance. Due to the variety of network types and architectures, a wide range of performance evaluation methods are available. In order to assess the overall quality of a data-center's network, a collection of network statistics is examined and evaluated. It is a quantitative technique that assesses and specifies the degree of performance of a network under consideration. End-user perception of network performance is the primary metric used to assess network performance (i.e. quality of network services delivered to the user). Generally speaking, network performance is assessed by examining the statistics and metrics from the following network components [85]:

- Network bandwidth or capacity refers to the amount of data that may be sent through a network.
- Network throughput is defined as the amount of data that is successfully carried through a network in a given amount of time.
- Network delay, latency, and jittering are all terms used to describe any network problem that causes packet transport to be slower than normal.

Need for testing network performance:

In accordance with the concept of data locality, the performance of the storage facility that is physically closest to the data pipeline server should be improved. Each datacenter, on the other hand, has a unique network infrastructure and set of capabilities. Furthermore, the physical distance between two points is a more permanent characteristic than the network's performance. As a result, while physical distance can provide an indication of which facility is likely to provide the best performance, the actual network performance test can provide assurance in this regard as well.

Software flowchart

Figure 6.7 shows the software chart for testing the network performance. The simple software tool uploads data to the cloud storage and then downloads the data to the local storage. In the third step, it takes the data from local storage to cloud storage again.

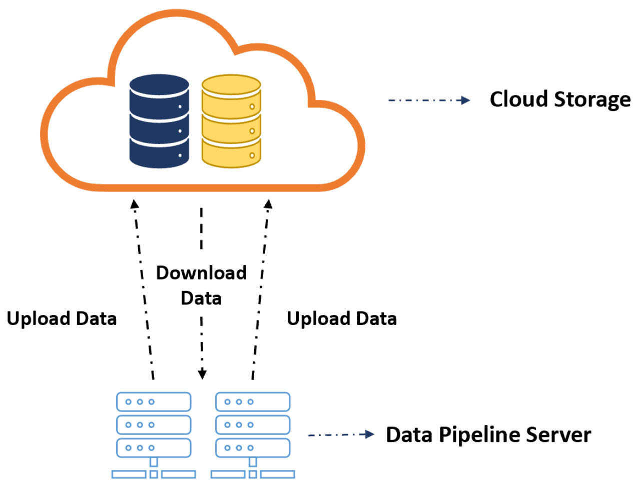


Figure 6.7: High-level design of a software component for testing the network performance. Data is uploaded, downloaded and then uploaded again from a data pipeline server to a cloud storage. Total execution time is calculated for the these operations.

6.3.4 Server-side Encryption

Server-side encryption is the encryption of data at its destination by the application or service that receives it. Cloud storage providers encrypt the data at the object level as it writes it to disks in its data centers and decrypts it for the user when it is accessed. As long as the request is authenticated and access permissions are properly defined, there is no difference in the way to access encrypted or unencrypted objects. For example, if objects are shared using a presigned URL, that URL works the same way for both encrypted and unencrypted objects. For this master thesis, server-side encryption is selected only with platform managed keys to transfer the overhead to the provider side as much as possible. Hence, no additional workload on the consumer side. A set of experiments are designed to study the affect of server-side encryption on the performance.

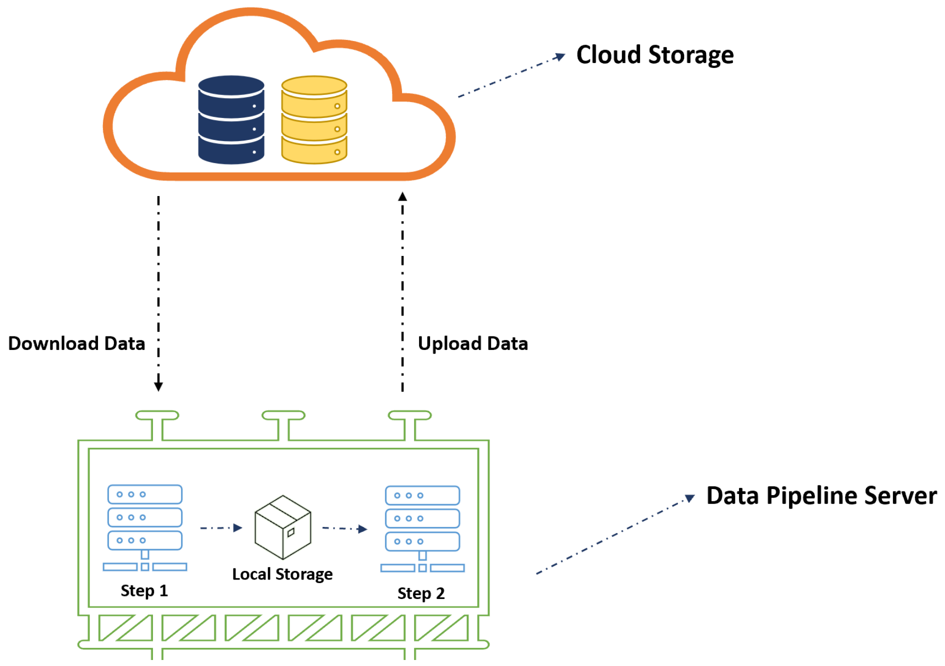


Figure 6.8: High-level design of a software component to test the impact of server-side encryption on performance.

High-level software design

Figure 6.8 shows the high level design for the software component to test the impact of server-side encryption on performance. The data is stored in the cloud storage and encrypted using server-side encryption. Two steps data pipeline is deployed on a separate compute instance. First step downloads the data from the server and stores in temporarily on a local storage. The second step reads the data from the local storage and uploads it back to the cloud storage. Some of the noticeable features on this design are as follows:

1. The concept of intermediate storage is introduced in this software component.
2. Use of relatively bigger data chunks to correctly observe the impact of decryption on performance while retrieving data.

Two separate storage buckets are deployed in the same physical region for each storage provider. Server-side encryption is enabled for one bucket whereas on the other bucket, data is stored without any encryption.

6.4 Evaluation Matrix

We are using Multi Criteria Decision Analysis (MCDA) for comparing different cloud service providers. VIKOR is the MCDA method used in this work. The VIKOR method is a multi-criteria decision making (MCDM) or multi-criteria decision analysis method. It was originally developed by Serafim Opricovic to solve decision problems with conflicting and non-commensurable (different units) criteria, assuming that compromise is acceptable for conflict resolution, the decision maker wants a solution that is the closest to the ideal, and the alternatives are evaluated according to all established criteria. VIKOR ranks alternatives and determines the solution named compromise that is the closest to the ideal.

Matrix development

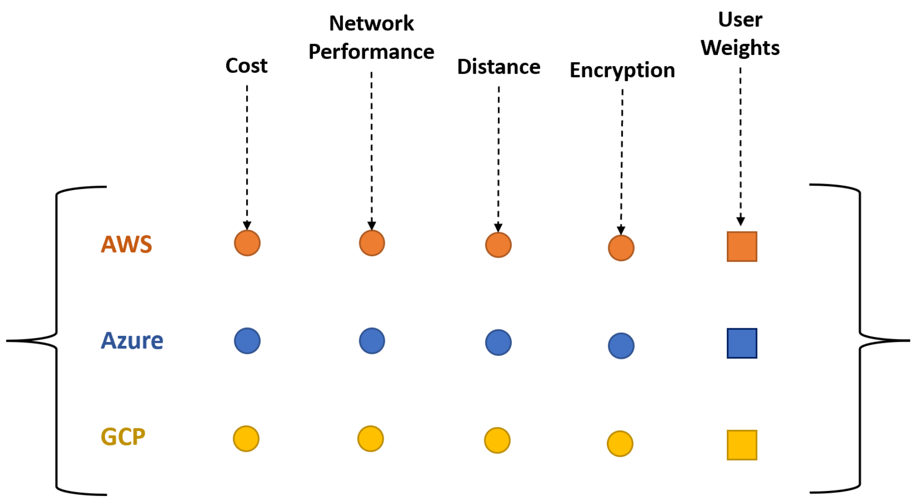


Figure 6.9: Evaluation matrix. Three rows and five columns. Each row is allocated to each cloud service provider that is Amazon, Azure and Google. The column represents the parameters such as cost, network performance, distance, impact of server-side encryption and user-weights.

Figure 6.9 shows the evaluation matrix for the proposed model. Values for the parameters are calculated from the steps explained in Section 6.3 above. Cost is calculated based on the storage space, bandwidth, read and write operations. Network performance is calculated in real-time with an experiment explained in Section 6.3.3. Distance is also calculated in real-time with the newly developed software tool explained in Section 6.3.1. Server-side encryption is implemented on the stored data in each cloud storage provider and it's affect is tested on the performance. The last column shows the weights set by the user. As explained earlier, the requirement could be different for each data pipeline. So the weights column provides an opportunity to set the priorities in the requirements.

High-level software design

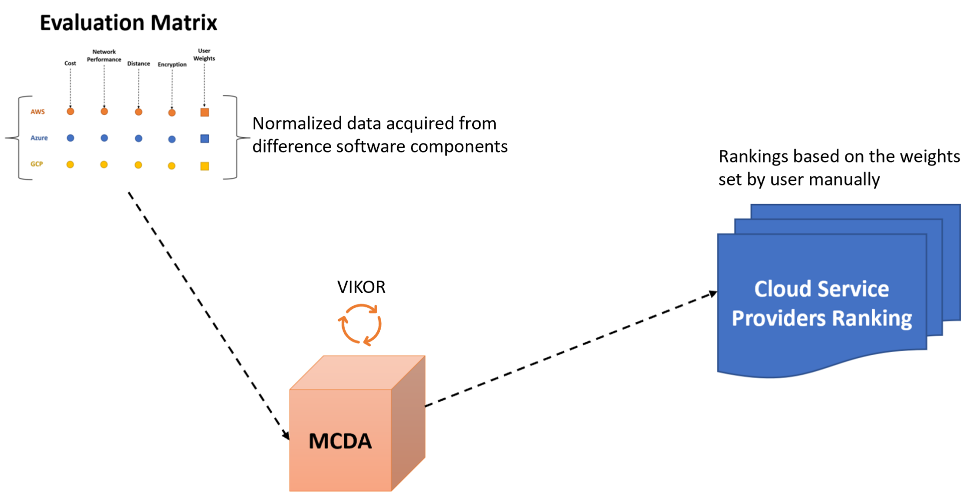

Evaluation Matrix
Cost
Network Performance
Distance
Encryption
User Weights
AWS
Azure
GCP
Normalized data acquired from difference software components
VIKOR
MCDA
Rankings based on the weights set by user manually
Cloud Service Providers Ranking

Figure 6.10: Multi-criteria Decision Analysis Flow Chart. An MCDA algorithm is implemented, the data in the form a normalized matrix is given as input, output shows the ranking of cloud service providers based on user-weights.

Figure 6.10 shows the flowchart for the proposed model. The evaluation matrix explained above is given as an input to the MCDA VIKOR algorithm, and the output is the ranking of the cloud service providers.

6.5 Domain-specific Language (DSL) Meta-model

The DSL core meta-model, depicted in Figure 6.11 and Figure 6.12, is technically represented as an ECore model, organized into several smaller packages. Each of these packages reflects a specific aspect of our DSL. In what follows, a comprehensive explanation of these packages is presented. It is extended version of DSL meta-model proposed in [66]. Noticeable elements include, flexible class structure to integrate large number of third party applications, support for security and monitoring packages. Most importantly, ability to integrated cloud storage in added in the DSL vocabulary. In addition to that, the DSL has now enhanced vocabulary to integrate a specific type of storage option such as MYSQL or NoSQL from a specific cloud storage. To further ease the usability, the user has an option to either specify the container image path for security, communication medium, monitoring etc. If a user specifies a particular package, the compiler would take that into account at the time of deployment, otherwise it would install the standard publicly available packages. To improve the usability of the class objects, single class for commands and environment variable is developed and reused at workflow level, step level, and even at single packages level such as security and storage.

6.5.1 Core Concepts

In this subsection, a brief overview of the main concepts of our meta-model is presented. Few of the main concepts are Workflow, Steps, Communication Medium and Storage.

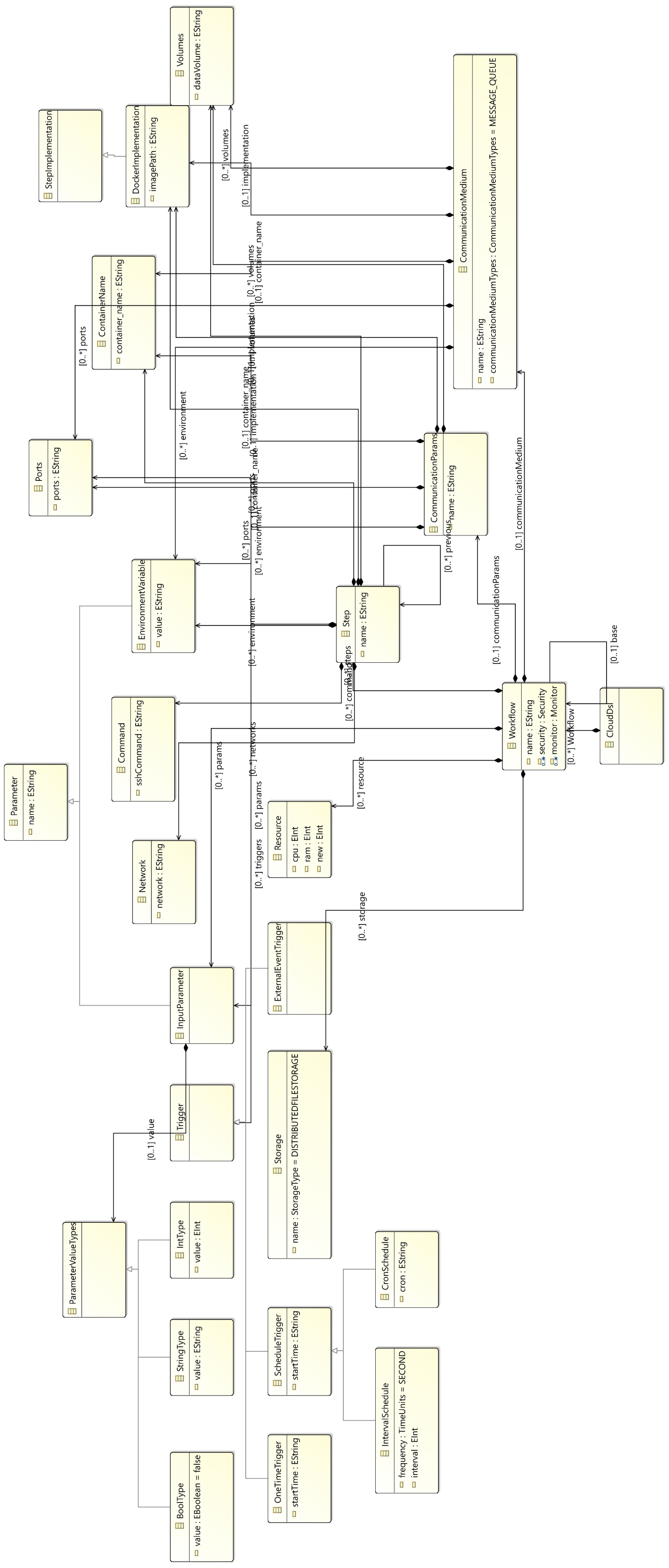


Figure 6.11: Domain-specific language main meta-model

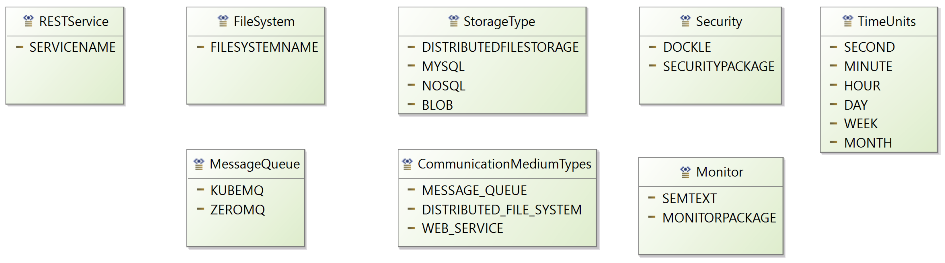


Figure 6.12: Domain-Specific Language Data Types

Sub-pipeline/workflow

In the designed DSL, each set of steps can be defined as a sub-workflow(pipeline); then, each sub-pipeline can be used as a separate step in other pipelines. Therefore, a sub-pipeline is a sequence of steps within a pipeline. By using sub-pipeline, we can reuse common processes without re-creating them for each pipeline. The CompositeStep entity represents the concept of sub-pipeline in the pipeline system. This class contains many different important characteristics of a data pipeline/-workflow. All the parameters that are need to be defined at the pipeline level are included in this class. Those are as follows:

- Steps
- CommunicationMedium
- Storage
- Security
- Monitor

Step Containerization

In the proposed design, each step is wrapped in a container. By doing so, an isolated virtual environment is provided for each step, which includes all the required dependencies of the provisioned tools. The *Step* class has the following important parameters:

- Trigger
- EnvironmentVariables
- InputParameters
- Command
- Network

Communication Medium

The data exchange mechanism is an essential element that binds the pipeline steps together by allowing them to pass data. The communicationMedium represents the mechanism in which workflow steps exchange data. For example, a pipeline step can pass data to another step using a message queue. The meta-model for the communication medium is shown in Figure 6.13. *CommunicationMedium* is a sub-class of *Workflow* object, as the communication medium would be defined at the workflow level. This very sub-class has further three parameters and a sub-class for for the *CommunicationMediumTypes*. There is a possibility to add three different kinds of communication medium: 1) Message Queue, 2) Distributed File System, and 3) Web Service. *MessageQueue* further has option for multiple packages such as ZeroMQ, KubeMQ but those are implemented in the generator class, not in the meta-model of the language Furthermore, pre-defined docker images can be override by defining the path in the *ContainerName* parameter. Similarly, *Ports* parameters can be used in the similar fashion as well.

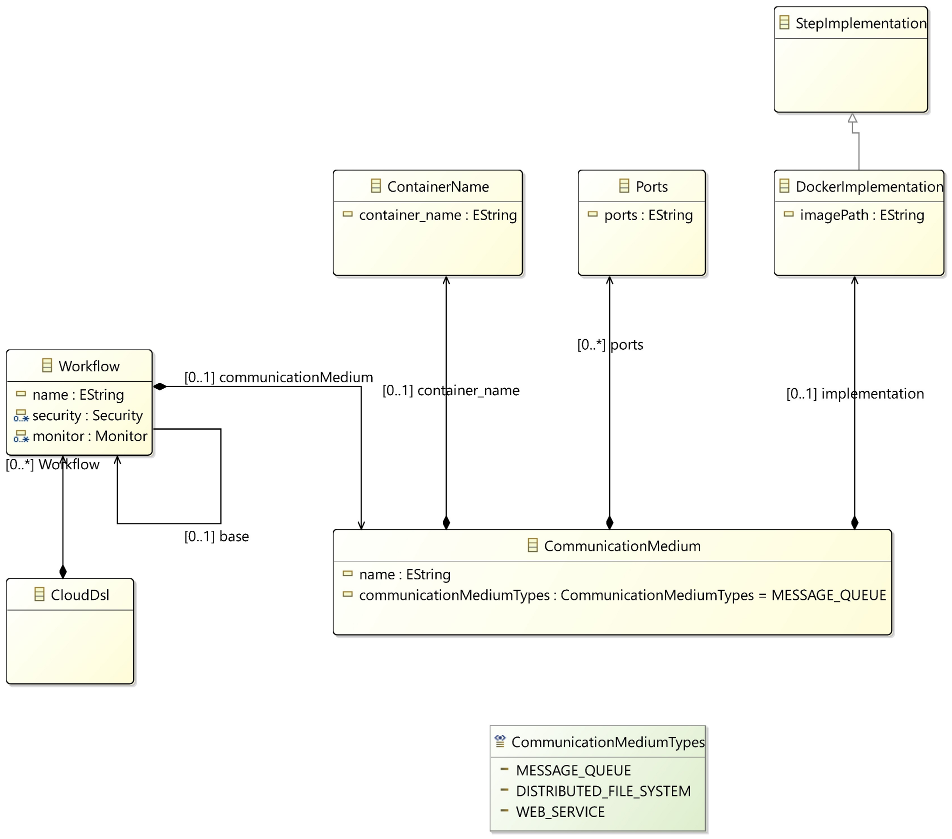


Figure 6.13: Communication Medium

Storage

The *Storage* package, depicted in Figure 6.12, is designed to manage and establish a connection to different types of data sources. So far, four different types of data sources are considered: DistributedFileStorage, MySQL, NoSQL and Blob. In addition, for each of these types, a specific type of credential is considered and are defined in the *EnvironmentVariable* package. Storage can be defined at two levels, workflow level and the step level.

Security

The *Security* package, shown in Figure 6.12, is designed to install and manage security packages for the data pipeline. The security protocols and package will be defined at the pipeline level, but will be deployed across the data pipeline at the step level. The option to override these protocols at the step level is also under consideration. Security options/packages can be defined at two levels, workflow level and the step level. If defined at the workflow level, it will be implemented across the workflow on all steps, but if defined separately at step level as well, it will override the configuration defined at workflow for that particular step.

Monitor

The *Monitor* package, shown in Figure 6.12, is designed to install and manage the software tool for big data pipeline monitoring. At this point, *SEMTEXT* is integrated and tested for the pipeline. Monitoring options/packages can be defined at two levels, workflow level and the step level. If defined at the workflow level, it will be implemented across the workflow on all steps, but if defined separately at step level as well, it will override the configuration defined at workflow for that particular step.

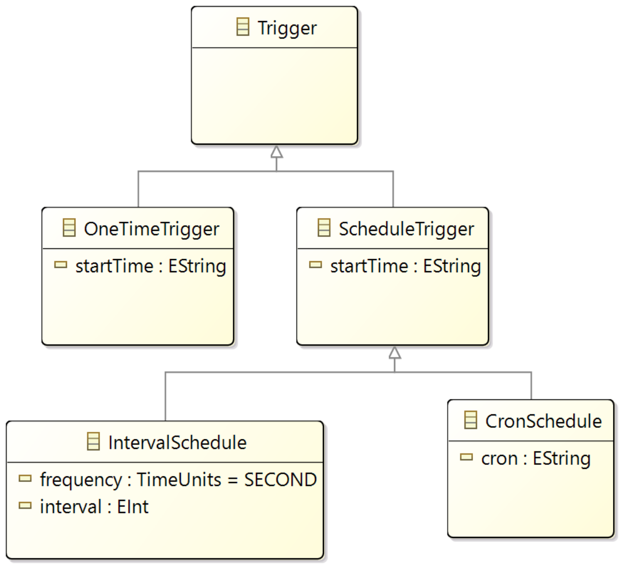


Figure 6.14: Trigger package

6.5.2 Trigger meta-model

Trigger, the package structure depicted in Figure 6.14, represents how the execution of a step instance is instantiated. Each step can potentially have zero or more trigger mechanisms. With the help of the trigger concept, the execution of each step can be triggered from a schedule that runs in a fixed interval of time, or it can be configured to run only once (e.g., during the initialization of the workflow). Execution can also be triggered by an external event (e.g., invocation from a REST API or availability of input data in a message queue).

6.5.3 Environment Parameters meta-model

The environment requirement meta-model, depicted in Figure 6.15, can capture the user functional and non-functional requirements, including but not limited to hardware, quality, and security ones. This meta-model includes the top-level Execution Requirement concept; each step can potentially have zero or one Execution Requirement, each of which can contain one or more parameters.

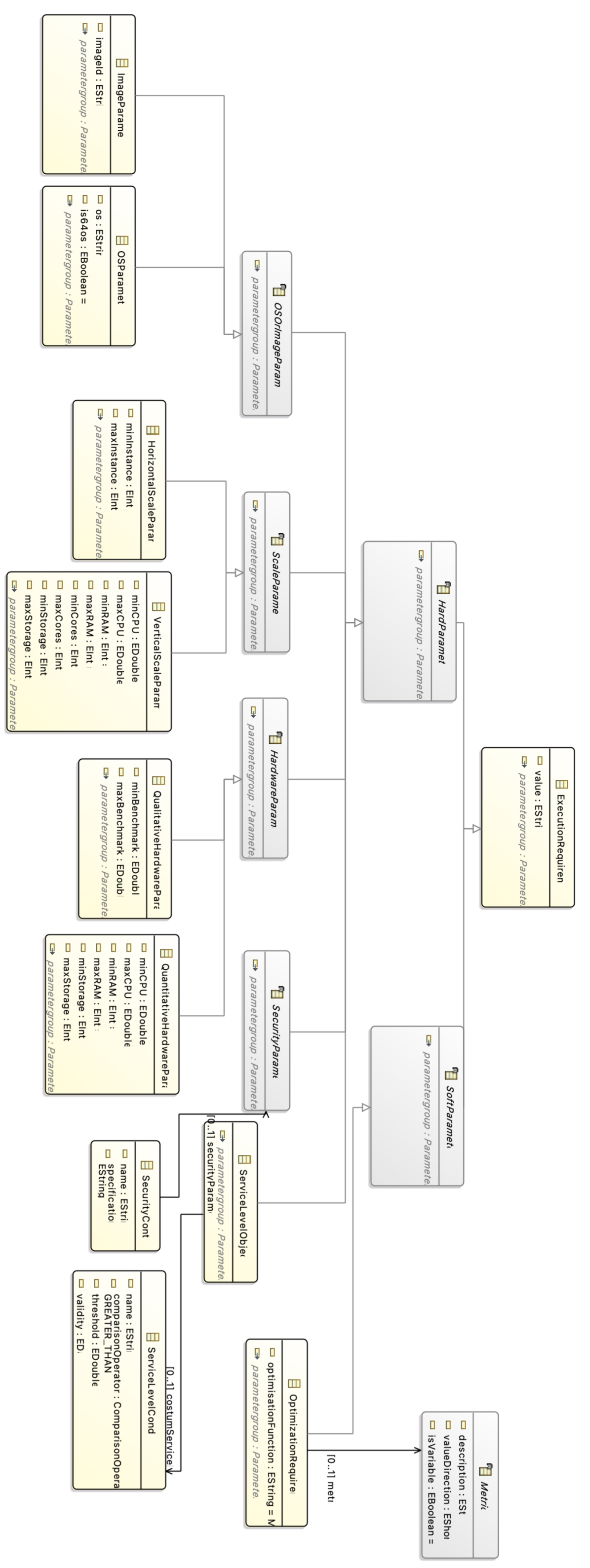


Figure 6.15: Environment Requirements meta-model

Chapter 7

Smart Data Placement Solution Implementation

The project involves the development of four software components, which are as follows: physical distance, cost calculation, network performance, encryption, and MCDA.

7.1 Physical Distance

This software component is developed for calculating the physical distance between the data pipeline server and the data centers. Python is used as a programming language. The software script is further divided into multiple procedures, which are explained in the following sections. Figure 7.1 shows the flowchart of this software component, showing different procedures and their interaction with each other. Procedures are further explained in detail in the following sections.

7.1.1 Python libraries

The software program uses the following Python libraries:

- **requests:** It allows you to send HTTP/1.1 requests extremely easily. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, thanks to urllib3.
- **time:** Python has a module named *time* to handle time-related tasks.
- **json:** JSON (JavaScript Object Notation), specified by RFC 7159 (which obsoletes RFC 4627) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript 1). Python *json* exposes an API familiar to users of the standard library marshal and pickle modules.
- **geopy:** It is a simple and consistent geocoding library written in Python that provides access to GeoIP databases. It integrates with a variety of geocoding

providers, including Google, Bing, OSM, and many others.

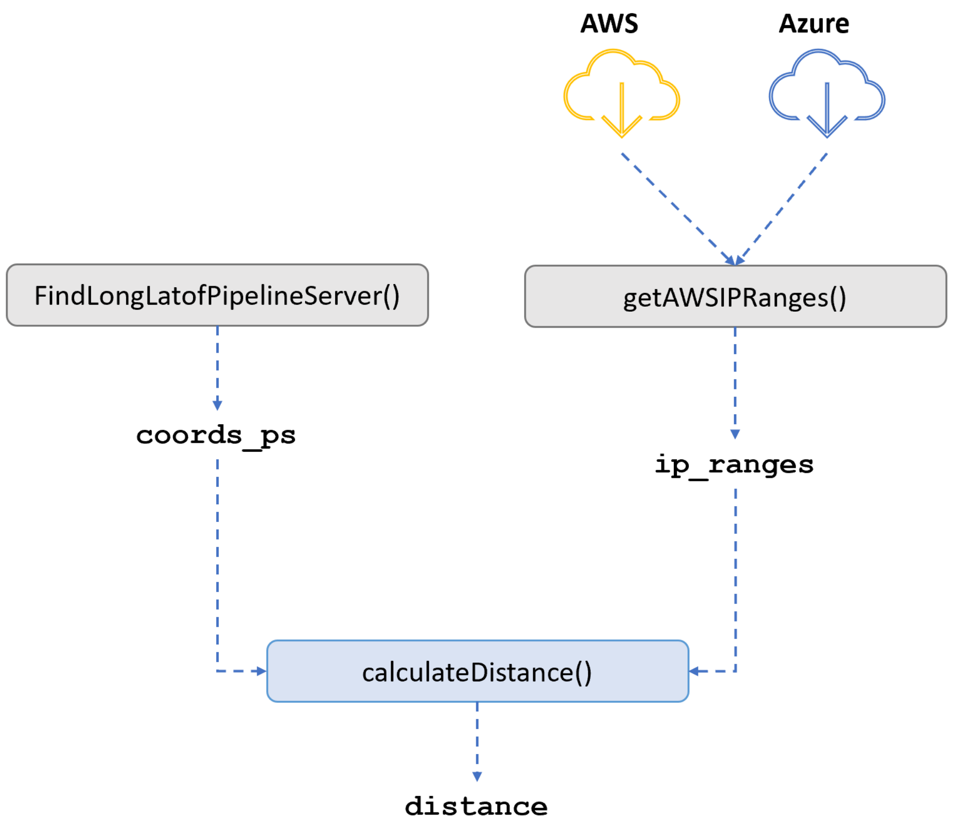


Figure 7.1: Flowchart for the software component to calculate the shortest physical distance.

7.1.2 Find Longitude & Latitude for Data Pipeline server

It is a simple procedure that finds out the longitude and latitude of the server on which the data pipeline itself is deployed. The values would be used further to calculate the distance. For this purpose, python **geocoder** library is used. The code snippet is shown in the listing 7.1. The *geocoder.ip*() function takes the string *me* as a parameter, stating that the longitude and latitude of the server on which this software component is being run are required. If it is not on the data pipeline server itself, the IP address can be passed as an input parameter.

Code listing 7.1: Python procedure for finding the longitude and latitude of the data pipeline server

```python
import geocoder
def FindLongLatofPipelineServer():
    g = geocoder.ip('me')
    return (g.latlng)
```

7.1.3 Fetch details for AWS storage servers

This procedure involves downloading the details of AWS Storage facilities in real-time from AWS IP Ranges [1]. Python **request** library is used to download and decode the JSON file.

Once the file is downloaded, the program loops through the json objects and filters out the IP addresses only for S3 servers. Listing 7.2 shows the code snippet for downloading the file from the AWS IP Ranges endpoint. The same code structure is used for both the procedure and the code that downloads and sorts the details for Azure storage servers.

Code listing 7.2: Python procedure for downloading the list of all AWS instances, including S3 in real-time from their official IP Ranges endpoint

```
import requests
def getAWSIPRanges():
    ip_ranges = requests.get('https://ip-ranges.amazonaws.com/ip-ranges.json'↩
     ).json()['prefixes']
    s3_ips = [item['ip_prefix'] for item in ip_ranges if item["service"] == "↩
     S3"]
    return s3_ips
```

7.1.4 Calculate distance

In order to find the longitude and latitude of the storage server from the IP, python **geopy** library is used. The procedure *findLongLatofAWSInstance()* in code listing 7.3 returns the longitude and latitude of an AWS S3 instance. whereas the procedure *calculateDistance()* calculates the actual distance in KMs between two sets of coordinates.

Code listing 7.3: Python procedure for finding the longitude and latitude of an AWS S3 instance and a procedure to calculate the distance between two sets of coordinates.

```
import geopy.distance

def findLongLatofAWSInstance():
    with urllib.request.urlopen("https://geolocation-db.com/jsonp/" + ip) as ↩
     url:
        d = url.read().decode()
        d = d.split("(")[1].strip(")")
        d = json.loads(d)
  coords_aws = (d['latitude'], d['longitude'])
  return coords_aws

```

[1] https://ip-ranges.amazonaws.com/ip-ranges.json

```
11 def calculateDistance(coords_aws, coords_ps):
12     distance = geopy.distance.vincenty(coords_aws, coords_ps).km
13     return distance
```

The same code pattern is followed for the Azure cloud. For Azure, the list of instances is downloaded in real-time from Azure IP Ranges [2]. Since it provides the list of all Azure services instances, the IP addresses are filtered out only for Azure storage. Furthermore, the IP ranges for the GCP are downloaded from GStatic [3]. The code for filtering is different from the one shown in listing 7.2 as the json file provided by Azure IP Ranges and GCP has a different schema.

The program checks for the shortest value returned from the procedure *calculateDistance* shown at line 11 in listing 7.3.

7.1.5 Code Repositories

Complete code for this software component is made available on GitHub [4]. Final version of the code has three versions:

1. Python file to run in a Windows or Linux environment (dependencies to be installed)
2. Python Notebook with a block of code to self install the dependencies
3. Set of files to build a docker image and run as a software container

7.2 Cost Calculation

This software component calculates the total estimate of the per month cost of using cloud storage in US dollars. It performs calculations for AWS S3 storage, Azure cloud storage, and Google cloud storage. This script is also written using the Python programming language. Since all cloud service providers work on a pay-as-you-go model, Table 6.1 shows the prices of each cloud storage provider divided on the basis of usage. The user requirements are passed as a 4x1 array, and the function returns a 6x3 array. The first column of *requirements* array has the total amount of storage required in GB. The unit GB is selected instead of TB to provide more flexibility to the user. The second column of this array specifies the required bandwidth in GB. The third and fourth columns of the *requirements* specify the number of write and read operations respectively.

7.2.1 Code Listings

Code listing 7.4: Software component written in python programming language to calculate to the estimated per month cost of a cloud storage based on user requirements

[2]https://www.microsoft.com/en-us/download/details.aspx?id=56519
[3]https://www.gstatic.com/ipranges/cloud.json
[4]https://github.com/akifquddus/MS-DataPlacementModel

```
#User Requirements
#requirements = [Storage, Bandwidth, WRITE, READ]
requirements  = [5000,    15000,     50000, 50000]

#total_cost   = [ProviderName, StorageCost, BandwidthCost, WRITECost, ↩
    READCost, TotalCost]
total_cost    = [
                    ['AWS',   0, 0, 0, 0, 0],
                    ['Azure', 0, 0, 0, 0, 0],
                    ['GCP',   0, 0, 0, 0, 0],
                ]

def calculateCost(requirements, total_cost):
    .
    .
    return total_cost

print(total_cost)

#SampleOutput
[
    ['AWS',   1150.0, 1350.0, 0.25, 0.2, 1465.5],
    ['Azure', 1000.0, 800.0,  0.35, 0.3, 900.65],
    ['GCP',   1000.0, 1720.0, 2.50, 0.0, 1822.5]
]
```

7.2.2 Code Repositories

Complete code for this software component is available on GitHub [5] and the published code has three versions:

1. Python file to run in a Windows or Linux environment (dependencies to be installed)
2. Python Notebook with a block of code to self install the dependencies
3. Set of files to build a docker image and run as a software container

7.3 Network Performance

Network performance refers to measures of the service quality of a network as seen by the customer. The performance may vary because each data center has a different network infrastructure. Hence, physical distance is not the only factor that can be used to measure the performance of cloud storage.

The software component to test the network is developed in the Python programming language. The simple program fetches articles from predefined online blogs and uploads the data in JSON format to the cloud storage.

[5] https://github.com/akifquddus/MS-DataPlacementModel

7.3.1 Python libraries

The program uses the following python libraries:

- **requests_html:** This library aims to make HTML parsing (for example, web scraping) as simple and straightforward as feasible [86].
- **time:** Python has a module named *time* to handle time-related tasks.
- **json:** JavaScript Object Notation (JSON) is specified by RFC 7159 (which obsoletes RFC 4627) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript 1) [87]. Python *json* exposes an API familiar to users of the standard library marshal and pickle modules [88].
- **boto3:** AWS services such as Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3) are created, configured, and managed using Python (Boto3) (Amazon S3). The SDK has both an object-oriented API and low-level access to Amazon Web Services.
- **azure.storage.blog:** Blobs are objects that can hold large amounts of text or binary data, including images, documents, streaming media, and archive data. You'll upload, download, and list blobs, and you'll create and delete containers. The Python SDK for blog storage helps to manage blobs using Python.

7.3.2 Code Listings

Code listing 7.5: Python code with two different procedures, get_source() to return the source code for the provided URL whereas the get_feed() returns a json object containing the RSS feed contents.

```python
def get_source(url):
    try:
        session = HTMLSession()
        response = session.get(url)
        return response

    except requests.exceptions.RequestException as e:
        print(e)
        return False

def get_feed(url):
    response = get_source(url)

    articles = []

    with response as r:
        items = r.html.find("item", first=False)

        for item in items:
            title = item.find('title', first=True).text
            pubDate = item.find('pubDate', first=True).text
            guid = item.find('guid', first=True).text
            description = item.find('description', first=True).text

            row = {'title': title, 'pubDate': pubDate, 'guid': guid, '↩
    description': description}
            articles.append(row)

    return articles
```

The code listing 7.5 shows two different python procedures. *get_source()* function takes the URL of the blog as an input parameter and returns the source code for the provided URL. *get_feed()* function also takes the URL of the RSS feed as an input parameter and uses the previously defined *get_source()* function to read the content, and then format it into a JSON object.

The software components also contain three different procedures to upload data to the s3 bucket, Azure blob storage, and Google cloud storage. There is no server-side or client-side encryption involved to make sure there are no anomalies in the experiment. In addition to that, all the data that is stored on the cloud storage is unique, hence no over-write operations are involved either. Listings 7.6 shows the procedures to upload JSON objects to AWS, Azure and Google Cloud.

Code listing 7.6: Python code with two different procedures, get_source() to return the source code for the provided URL whereas the get_feed() returns a json object containing the RSS feed contents.

```
# Upload JSON Object to AWS
def uploadAWS(article):
    object = s3.Object(BUCKET, article['title'])
    result = object.put(
        Body=(bytes(json.dumps(article).encode('UTF-8')))
    )

    res = result.get('ResponseMetadata')

    if res.get('HTTPStatusCode') == 200:
        return true
    else:
        return false

#Upload JSON Object to Azure
def uploadAzure(article):
    blob_client = blob_service_client.get_blob_client(container=CONTAINER, ↩
    blob=article['title'])
    data = json.dumps(article)
    res = blob_client.upload_blob(data,overwrite=True
    if res.get('HTTPStatusCode') == 200:
        return true
    else:
        return false)

#upload JSON Object to GCP
def uploadGCP(article):
    .
    .
    return true
```

7.3.3 Code Repositories

Complete code for this software component is available on GitHub [6] and the published code has three versions:

1. Python file to run in Windows or Linux Environment (Dependencies to be installed)
2. Python Notebook with a block of code to self install the dependencies
3. Set of files to build a docker image and run as a software container

7.4 Server-side Encryption

This section contains the implementation details for the software program to test the impact of server-side encryption on performance. There are three different components are setup to observe the stated purpose. System components are as follows:

- Storage Server
  - Amazon S3 Bucket (With and without server-side encryption)
  - Azure Cloud Storage (With and without server-side encryption)
  - Google Cloud Storage (With and without server-side encryption)
- Data Pipeline development

7.4.1 Python libraries

Following python libraries are used in this software program:

- **requests_html:** This library intends to make parsing HTML (e.g. scraping the web) as simple and intuitive as possible.
- **time:** Python has a module named *time* to handle time-related tasks.
- **json:** JSON (JavaScript Object Notation), specified by RFC 7159 (which obsoletes RFC 4627) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript 1). Python *json* exposes an API familiar to users of the standard library marshal and pickle modules.
- **boto3:** Python (Boto3) is used to create, configure, and manage AWS services, such as Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3). The SDK provides an object-oriented API as well as low-level access to AWS services.

[6]https://github.com/akifquddus/MS-DataPlacementModel

7.4.2 Cloud Storage

Storage services are setup on three different cloud service providers. Amazon web services, Azure cloud and Google cloud storage. On each of the provider, two different types of buckets are deployed, one with server-side encryption enabled and one without server-side encryption. Exactly same video data is uploaded to all the buckets.

7.4.3 Data pipeline

A simple two steps data pipeline is developed to observe the impact of server-side encryption on performance. Step 1 downloads data from the cloud storage and saves on the local storage. Step 2 in return uploads the data stored in local storage to the cloud storage and removes it from the local storage. For downloading and uploading, HTTP is used. HTTP is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, in this the data pipeline server.

7.4.4 Code Listings

Listing 7.7 shows the python procedure to download data from AWS s3 bucket in a sequential manner. The code is developed for the storage bucket with and without server-side encryption enabled. Software script is also written in the similar manner for Azure and Google cloud storage.

Code listing 7.7: Python procedure to download data from AWS s3 bucket. Same code is setup for bucket with encryption and without encryption

```python
s3client= boto3.client('s3',
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
    )

def download_cloud_data():
    for my_bucket_object in my_bucket.objects.filter(Prefix='videos_encrypted/'):
        key = my_bucket_object.key

        s3client.download_file(bucket, 'videos_encrypted/' + key, "tempvideos/" + key)

    return "Operation completed!"
```

Listing 7.8 shows the python procedure to upload data to AWS s3 bucket in a sequential manner and then subsequently deleting it from the local storage as well. The code is developed for the storage bucket with and without server-side encryption enabled. Software script is also written in the similar manner for Azure and Google cloud storage.

Code listing 7.8: Python procedure to upload data to AWS s3 bucket in a sequential manner and then subsequently deleting it from the local storage as well. Same code is setup for bucket with encryption and without encryption

```
s3client= boto3.client('s3',
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
    )

def download_cloud_data():
    for my_bucket_object in my_bucket.objects.filter(Prefix='videos_encrypted↩
    /'):
        key = my_bucket_object.key

        s3client.download_file(bucket, 'videos_encrypted/' + key, "tempvideos↩
    /" + key)


        body = open('tempvideos/' + key, 'rb')


        object = s3.Object(bucket, 'videos_new/' + key)

        result = object.put(Body=body)

        res = result.get('ResponseMetadata')

        if res.get('HTTPStatusCode') == 200:
            print('File Uploaded Successfully for ' + key)
        else:
            print('File Not Uploaded')
```

7.4.5 Code Repositories

Complete code for this software component is available on GitHub [7] and the published code has three versions:

1. Python file to run in Windows or Linux Environment (Dependencies to be installed)
2. Python Notebook with a block of code to self install the dependencies
3. Set of files to build a docker image and run as a software container

7.5 Evaluation Matrix

This section provides the implementation details for the evaluation matrix to rank different cloud service providers. For this purpose, the Python VIKOR method is used. The first step in the process is data normalization. As discussed in the previous sections, there are a total of four parameters: the physical distance in kilomet-

[7]https://github.com/akifquddus/MS-DataPlacementModel

ers; total cost estimation for each cloud service provider; network performance; and the impact of server-side encryption. The VIKOR method ranks the entity high with a higher value in the matrix, but in this situation, a higher value means poor performance. For example, 200 km in distance is better than 300 km. Similarly, the smaller the execution time, the better it is. So, to solve this problem, the values are multiplied by -1.

7.5.1 Python libraries

Following python libraries are used in this software program:

- **VIKOR:** Python *VIKOR* function implements the "VIseKriterijumska Optimizacija I Kompromisno Resenje" (VIKOR) Method. *VIKOR* returns a data frame which contains the score of the S, R and Q indixes and the ranking of the alternatives according to Q index.

7.5.2 Code Listings

Code listings 7.9 shows the python code to define a VIKOR object with data and data labels. On line number 6, a dataframe is defined for the VIKOR object. The first parameter is a multi-dimensional array with the data about the cloud service providers, such as cost, distance, network performance, and encryption impact. The second parameter is also an array with the names of cloud service providers. The third and last parameter is an array with labels such as cost, distance, time, etc.

Code listing 7.9: Python code to define a VIKOR object with data and data labels

```
 1 from scikitmcda.vikor import VIKOR
 2 from scikitmcda.constants import MAX, MIN, LinearMinMax_, LinearMax_, ↩
       LinearSum_, Vector_, EnhancedAccuracy_, Logarithmic_
 3
 4 vikor = VIKOR()
 5
 6 vikor.dataframe([
 7         [AWS_COST, AWS_DISTANCE, AWS_NETWORK, AWS_ENCRYPTION],
 8         [AZURE_COST, AZURE_DISTANCE, AZURE_NETWORK, AZURE_ENCRYPTION]
 9     ],
10     ["AWS", "Azure"],
11     ["Cost", "Distance", "Time"]
12 )
13 print(VIKOR.pretty_original())
```

Code listing 7.10 shows the code snippet to set the weights for each parameter. For example, on line number 1, the user has set the maximum weight for the network performance. The ranking is based on the user weights, which are set manually.

Code listing 7.10: Python code to set the weights manually based on the user's preference

```python
vikor.set_weights_manually([0.15, 0.05, 0.80])
topsis.set_signals([MIN, MAX, MAX])
topsis.decide()

print("NORMALIZED:\n", topsis.pretty_normalized())
```

Code listings 7.11 shows the code that analyzes the data and shows the results with user weights set manually. The results are explained in detail in Section 8.5.

Code listing 7.11: Python code to define a VIKOR object with data labels

```python
print("WEIGHTED:\n", topsis.pretty_weighted())
print("RANKING TOPSIS with", topsis.normalization_method , ":\n", topsis.↩
    pretty_decision())
```

7.5.3 Code Repositories

Complete code for this software component is available on GitHub [8] and the published code has three versions:

1. Python file to run in Windows or Linux Environment (dependencies to be installed)
2. Python Notebook with a block of code to self install the dependencies
3. Set of files to build a docker image and run as a software container

7.6 Domain-specific Language (DSL)

This section provides the implementation details for the newly proposed domain-specific language to deploy a big data pipeline. The language meta-model is developed using Xtext which is a framework for development of programming languages and domain-specific languages. To convert the language code written by user based on the new meta-model, a compiler is constructed using Xtend, which is is a flexible and expressive dialect of Java, which compiles into readable Java 8 compatible source code.

7.6.1 Code Listings

Code listing 7.12 shows the validated Xtext code to define a **Workflow** object with reference to the objects for communication medium, storage, security, and monitoring tools. The asterisk (textbf*) symbol determines if the object can be defined multiple times or just once. The question mark (**?**) specifies that the object is optional and that the code will compile without any errors even if the parameter

[8]https://github.com/akifquddus/MS-DataPlacementModel

is not defined. As you can see, a "big data workflow" or "big data pipeline" can have multiple steps and more than one way to store, protect, and keep track of the data.

Code listing 7.12: Xtext code to define a **Workflow** object with objects for communication medium, storage, security and monitoring tools.

```
Workflow:
  'workflow'
  name = ID
  ('extends' base = [Workflow])?
  '{'
    ('communicationMedium:' (communicationMedium = CommunicationMedium))
    ('communicationParams:' (communicationParams = CommunicationParams))
    ('parameters:' (params += InputParameter (',' params += InputParameter)*)⮐
        )?
    ('steps:' ('-' steps += Step ('-' steps += Step)*))
    ('storage:' (storage += Storage (',' storage += Storage)*))?
    ('security:' (security += Security (',' security += Security)*))?
    ('monitor:' (monitor += Monitor))?
    ('resource:' (resource += Resource (',' resource += Resource)*))?
  '}'
;
```

The code listing 7.13 displays the validated Xtext code for defining a textbf-Step object with references to objects for the trigger, implementation details, input parameters, ports, environment variables, volumes, commands, network, and the link to the previous step in the data pipeline/workflow. With trigger, a user can define the schedule by which a step will start the processing. It can either be manually triggered, event triggered, one time scheduled or cron scheduled. With the implementation object, implementation details can be added such as a link to the container image. Same as the **Workflow** object, the asterisk (*) symbol determines if the object can be defined multiple times or just once. The question mark (?) specifies that the object is optional and that the code will compile without any errors even if the parameter is not defined.

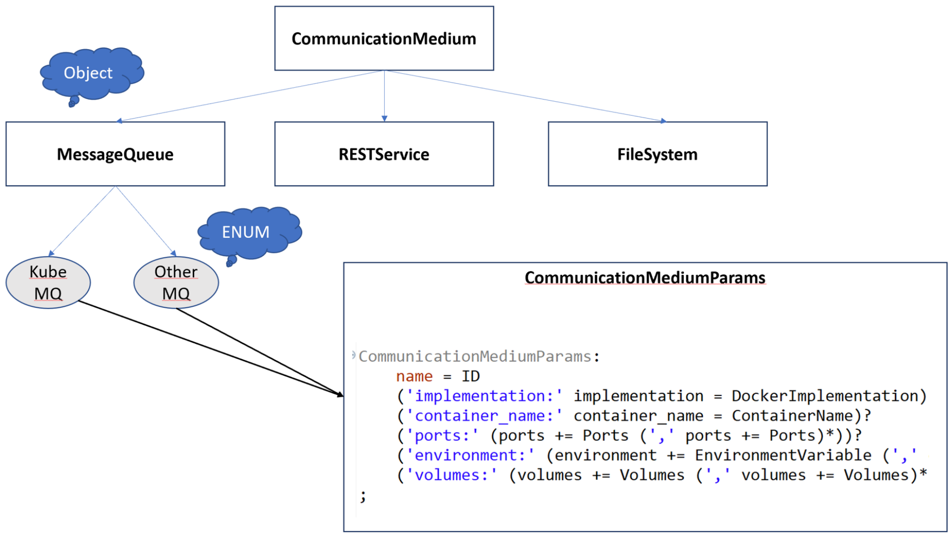
Code listing 7.13: Xtext code to define a **Step** object with objects for implementation, triggers, ports, volumes, command and other options.

```
Step:
  'step' name = ID
    ('triggers:' (triggers += Trigger (',' triggers += Trigger)*)+ )?
    ('implementation:' implementation = DockerImplementation)
    ('container_name:' container_name = ContainerName)?
    ('parameters:' (params += InputParameter (',' params += InputParameter)*)⮐
        )?
    ('ports:' (ports += Ports (',' ports += Ports)*))?
    ('environment:' (environment += EnvironmentVariable (',' environment += ⮐
        EnvironmentVariable)*))?
    ('volumes:' (volumes += Volumes (',' volumes += Volumes)*))?
    ('command:' (command += Command (',' command += Command)*))?
    ('networks:' (networks += Network))?
    ('previous:' ('none' |  (previous += [Step] (',' previous += [Step])*)))?;
```

The code listing 7.14 shows the validated Xtext code for defining a **CommunicationMedium** object with references to objects such as the type of communication medium, implementation details, ports, and environment variables. Line number 11 also shows an object for **CommunicationParam**. Figure 7.2 shows the proposed class diagram for implementing the communication medium, but during the development of the compiler, it resulted in a lot of redundant code. To solve the problem, a modified class diagram is proposed, which is shown in Figure 7.3. As can be seen, the objects for communication medium type have been changed to enum type. The new class model greatly optimized the code in the compiler.

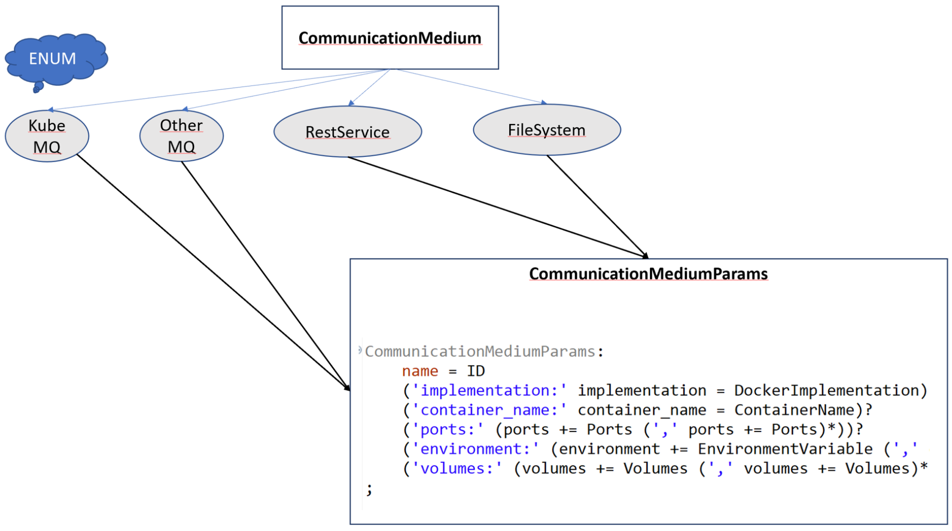
Code listing 7.14: Xtext code to define a **Communication** object with objects for implementation, ports, volumes and environment.

```
1  CommunicationMedium:
2    name = ID
3    'type:' communicationMediumTypes = CommunicationMediumTypes
4    ('implementation:' implementation = DockerImplementation)?
5    ('container_name:' container_name = ContainerName)?
6    ('ports:' (ports += Ports (',' ports += Ports)*))?
7    ('environment:' (environment += EnvironmentVariable (',' environment += EnvironmentVariable)*))?
8    ('volumes:' (volumes += Volumes (',' volumes += Volumes)*))?
9  ;
10
11 CommunicationParams:
12   name = ID
13   ('implementation:' implementation = DockerImplementation)?
14   ('container_name:' container_name = ContainerName)?
15   ('ports:' (ports += Ports (',' ports += Ports)*))?
16   ('environment:' (environment += EnvironmentVariable (',' environment += EnvironmentVariable)*))?
17
18 enum CommunicationMediumTypes:
19   MESSAGE_QUEUE | DISTRIBUTED_FILE_SYSTEM | WEB_SERVICE;
20
21 enum MessageQueue:
22   KUBEMQ | ZEROMQ;
23
24 enum RESTService:
25   SERVICENAME
26 ;
27
28 enum FileSystem:
29   FILESYSTEMNAME
30 ;
```



```
CommunicationMediumParams:
    name = ID
    ('implementation:' implementation = DockerImplementation)
    ('container_name:' container_name = ContainerName)?
    ('ports:' (ports += Ports (',' ports += Ports)*))?
    ('environment:' (environment += EnvironmentVariable (','
    ('volumes:' (volumes += Volumes (',' volumes += Volumes)*
;
```

Figure 7.2: Proposed class diagram for implementing communication medium.



```
CommunicationMediumParams:
    name = ID
    ('implementation:' implementation = DockerImplementation)
    ('container_name:' container_name = ContainerName)?
    ('ports:' (ports += Ports (',' ports += Ports)*))?
    ('environment:' (environment += EnvironmentVariable (','
    ('volumes:' (volumes += Volumes (',' volumes += Volumes)*
;
```

Figure 7.3: Revised class diagram for implementing communication medium.

The code listing 7.15 shows the validated Xtext code for **Trigger** model. There are three main types of triggers, **OneTimeTrigger**, **ScheduleTrigger** and **ExternalEventTrigger**. OneTimeTrigger has an option to specify a start time. ScheduleTrigger further has two types, **IntervalSchedule** and **CronSchedule**. With IntervalSchedule, specific time intervals can be defined. The CronSchedule works in a similar fashion to the server cron.

Code listing 7.15: Xtext code to define a **Trigger** object with three different options such as **OneTimeTrigger**, **ScheduleTrigger** and **ExternalEventTrigger**.

```
Trigger:
  OneTimeTrigger | ScheduleTrigger | ExternalEventTrigger;

OneTimeTrigger:
  {OneTimeTrigger}
  'one-time'
  (startTime = STRING)?;

ScheduleTrigger:
  IntervalSchedule | CronSchedule;

ExternalEventTrigger:
  {ExternalEventTrigger}
  'external-event';

IntervalSchedule:
  {IntervalSchedule}
  'interval-schedule'
  ('frequency:' frequency = TimeUnits)?
  ('interval:' interval = INT)?
  ('start-time:' startTime = STRING)?;

CronSchedule:
  {CronSchedule}
  'cron-schedule'
  ('start-time:' startTime = STRING)?
  ('cron:' cron = STRING)?;

enum TimeUnits:
  SECOND | MINUTE | HOUR | DAY | WEEK | MONTH;
```

The code listing 7.16 shows the code written in the new domain-specific language. It is an example code that specifies a workflow/data pipeline with a communication medium and two steps. This code is then further compiled into deployable YAML code.

Code listing 7.16: A code sample of the newly developed domain-specific language for the deployment of big data pipelines.

```
workflow CloudDataWorkFlow
{
  communicationMedium: MESSAGEQUEUE
  type: WEB_SERVICE
  communicationParams: restservices
    container_name: "kubemq"
    ports: '"8080:8080"', '"9090:9090"', '"50000:50000"'
    environment: KUBEMQ_HOST = "kubemq", KUBEMQ_TOKEN = "8297ae1b-d2bc-40b4-↩
         bf95-39a67162984d"
    volumes: "kubemqvol:/store"

```

```
steps:
  -step readzip_00
    triggers: one-time '24/06/2021 15:00', external-event
    implementation: docker-implementation image: "DOCKERPATH/ebw-movies-00-read-zip:latest"

    environment: STEP_NAME = 'readzip_00', KUBEMQ_HOST = 'kubemq'

    volumes: '/home/data/ebw-movies/00-read-zip/input:/in',
    '/home/data/ebw-movies/00-read-zip/work:/work',
    '/home/data/ebw-movies/00-read-zip/output:/out',
    '/home/data/ebw-movies/00-read-zip/sandbox:/sandbox'

    command: '"60"', '/sandbox', '/code', '/in', '/work', '/out', '"-"', '"MQ1"'
    networks: 'ebw-movies-ntk'

  - step unzip_01
    triggers: cron-schedule
      start-time:'2020-05-31T22:30:59'
      cron: '0 15,30,45 * ? * *'
    implementation: docker-implementation image: "DOCKERPATH/ebw-movies-01-unzip:latest"

    environment: STEP_NAME = 'unzip_01', KUBEMQ_HOST = "kubemq"

    volumes: '/home/data/ebw-movies/00-read-zip/output:/in',
    '/home/data/ebw-movies/01-unzip/work:/work',
    '/home/data/ebw-movies/01-unzip/output:/out',
    '/home/data/ebw-movies/01-unzip/sandbox:/sandbox'

    command: '"60"', '/sandbox', '/code', '/in', '/work', '/out', '"MQ1"', '"MQ2"'

    networks: "ebw-movies-ntk"
```

Part IV

Evaluation

Chapter 8

Experiments and Results

8.1 Experiment A - Novel Architecture

Using the above-mentioned system architecture presented in Figure 6.3, the following big data pipeline is designed. The pipeline is shown in Figure 8.1. A pipeline server is set up in New York (US East). A software script is developed to find the closest storage facility available. For the sake of this experiment, only AWS servers were evaluated. Two storage servers are set up, one in the USEast (suggested by the software package) region and another in the EU West region. Further details for this experiment are as follows.

8.1.1 Experiment Setup

Following are the technical details for this experiment:

- **Data pipeline server:** Newyork Region, 1 CPU, 2GB RAM, 50GB SSD Storage. Operation system: Ubuntu 20.04
- **AWS s3 bucket:** us-east-1 region and eu-west region.
- **Data pipeline:** Four steps data pipeline.
- **Data set:** Total 4MB. Data chunks: 3063. Each data chunk: Variable size.
- **Execution:** Serial and parallel execution of operations.

8.1.2 Minimal use of computing resources

Since the experiment was to study the role and impact of cloud storage in big data pipelines, to make sure the results are a true representation of the role of StaaS in the performance of a big data pipeline, simple textual data is used so that the least amount of time is spent on actual processing of the data. More on data transfer, data read, and data write operations. This way, the execution time of the data pipeline reflects the more precise role of the storage service being used.

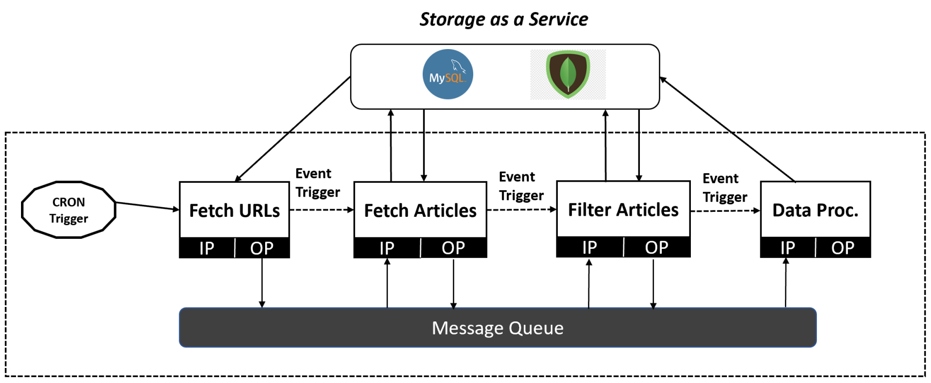


Figure 8.1: Big data pipeline for the feasibility study (IP: Input, OP: Output)

8.1.3 Use of very small data chunks

The dataset used in the experiments is articles from different online blogs and news websites, each occupying a few Kilobytes (KBs) of space on a disk. The use of bigger data chunks is avoided to increase the number of disk operations. In addition to that, a smaller data chunk means less use of computing power, as discussed in the previous section.

8.1.4 Results - Novel Architecture

The compute steps were run multiple times with both storage services separately in a serial and parallel manner. The results are shown in Figure 8.2 and Figure 8.3. Figure 8.2 shows the comparison of results of computing steps in serial execution between EUWest and USEast Storage. Whereas Figure 8.3 shows the comparison of pipeline execution in a serial vs. parallel manner. The important point to notice here is that the serial execution in the USEast region is faster than the parallel execution in the EUWest region.

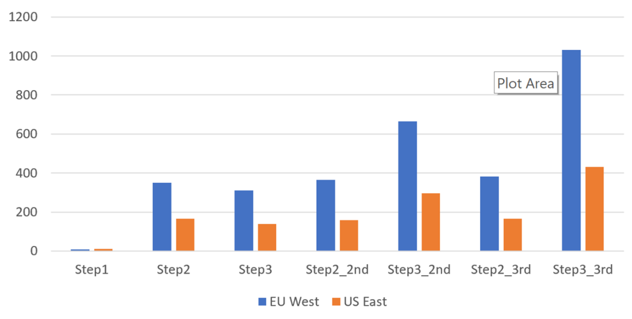


Figure 8.2: Comparison of results of computing steps in serial execution between EUWest and USEast Storage. The y-axis shows the number of seconds taken to finish each operation.

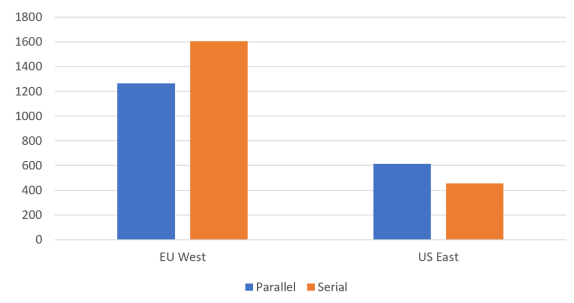


Figure 8.3: Serial vs Parallel execution of big data pipeline compute steps. The y-axis shows the number of seconds taken to finish each operation.

8.2 Experiment B - Physical Distance

To test the effect of the new software tool, a data pipeline is deployed on a server located in the USEast region. The pipeline performance is integrated with four different storage facilities. Two from Amazon Web Services and two from the Azure Cloud. Starting with the AWS S3, because the pipeline server is located in the USEast region, one can choose between the region pair *us-east-1* and *us-east-2*. But the software suggested the *us-east-1* region. In order to test the accuracy and effectiveness of our software tool, we setup S3 on both region pairs and tested the performance. During the test, a total of 3,601 WRITE operations were carried out, with a total size of 3.2 Megabytes (MB). To make sure there was no temporary downtime on the network, the operation was repeated three times.

8.2.1 Results with AWS S3

Figure 8.4 shows the difference in the performance of storage facilities located in the same region but different region pairs. With the results generated from the new software tool, an informed choice can be made to select the best region from the cloud service provider. As a further extension of the work discussed in 6.2, the test program was run in parallel. Figure 8.5 shows that the performance of the recommended region, which is *us-east-1* is clearly better than the AWS S3 storage facility located in *us-east-2*.

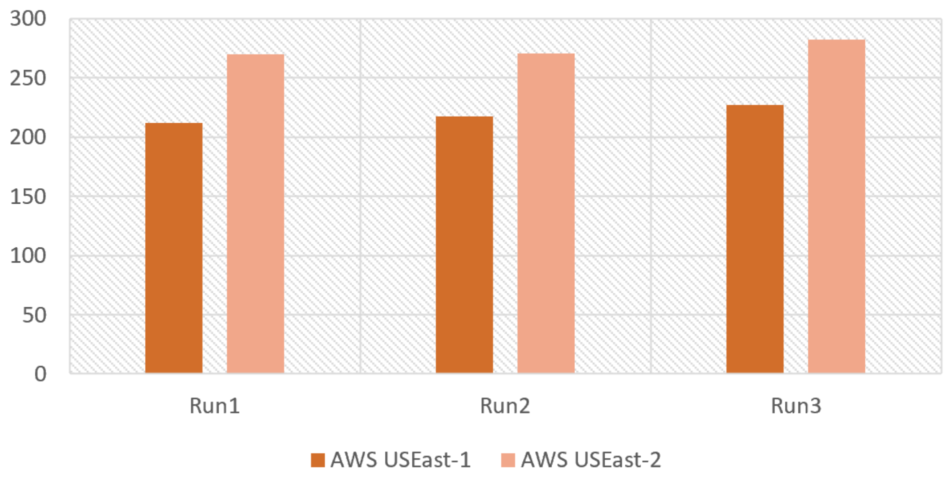


Figure 8.4: AWS region pair comparison in serial execution. The y-axis shows the number of seconds taken to finish each operation.

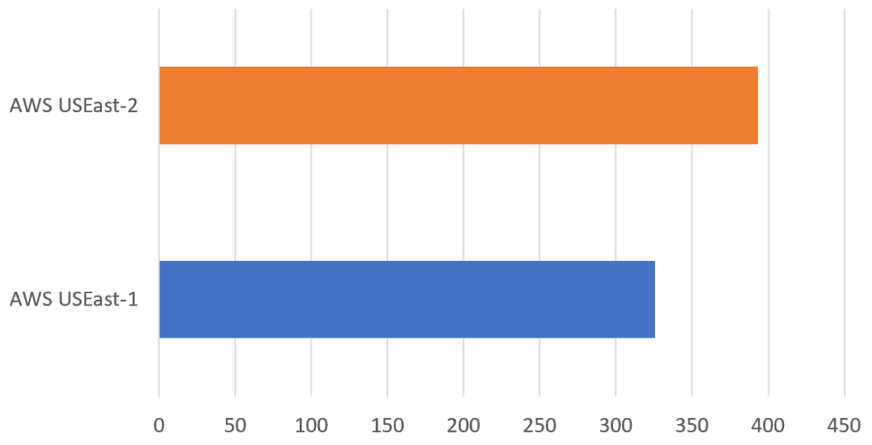


Figure 8.5: AWS region pair comparison in parallel execution. The x-axis shows the number of seconds taken to finish each operation.

8.2.2 Results with Azure Cloud

The same data pipeline as described in the previous section was then run with the StaaS integration of Azure Cloud. First, the results were obtained from the new software tool. The system suggested the USEast1 region of Azure Cloud. To test the accuracy of the program, the pipeline was deployed with integration of the USEast1 and USEast2 regions and run independently. Because Azure cloud offers multiple storage tiers, the hot tier was chosen for this experiment. Furthermore, the details of the operations and data size are as follows:

- WRITE Operations: 3061
- Data Size: 3.2 MB

The results for sequential execution are shown in Figure 8.6 and for parallel execution are shown in Figure 8.7.

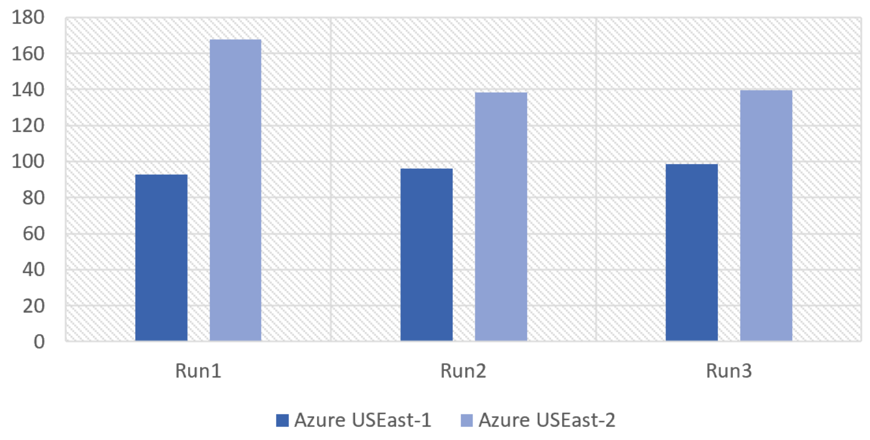


Figure 8.6: Azure region pair comparison in serial execution. The y-axis shows the number of seconds taken to finish each operation.

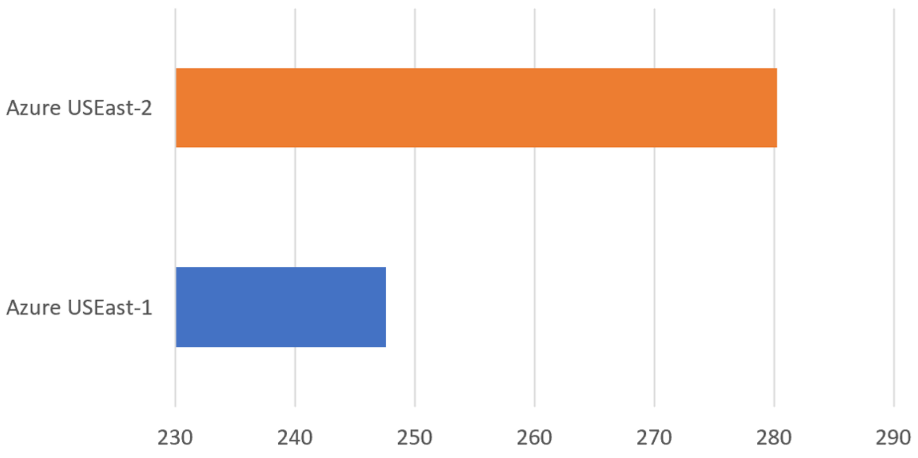


Figure 8.7: Azure region pair comparison in parallel execution. The x-axis shows the number of seconds taken to finish each operation.

8.2.3 Results with Google Cloud Storage

The same data pipeline as described in the previous section was then run with the StaaS integration with Google cloud storage. First, the results of the new software tool were obtained. The system suggested the us-east4 region of Azure Cloud. To test the accuracy of the program, the pipeline was deployed with integration of us-east1 and us-east4 regions and run independently. Because Google Cloud Storage offers multiple storage tiers, the standard tier was chosen for this experiment. Furthermore, the details for the operations and data size are as follows:

- WRITE Operations: 3061
- Data Size: 3.2 MB

The results for sequential execution are shown in Figure 8.8 and for parallel execution are shown in Figure 8.9.

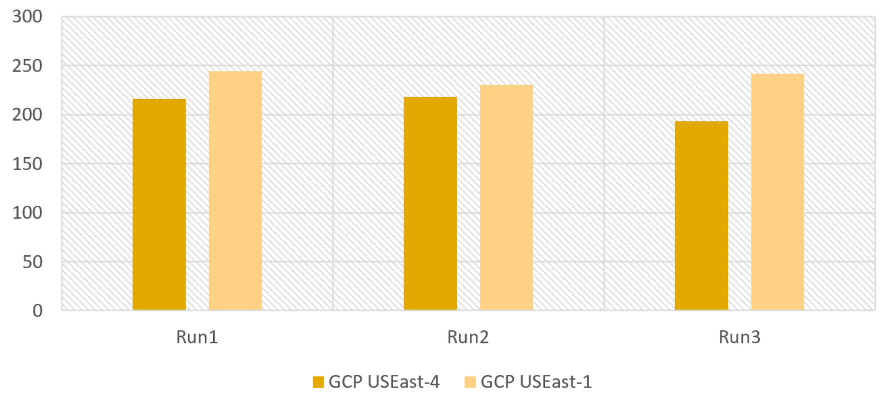


Figure 8.8: Google region pair comparison in serial execution. The y-axis shows the number of seconds taken to finish each operation.

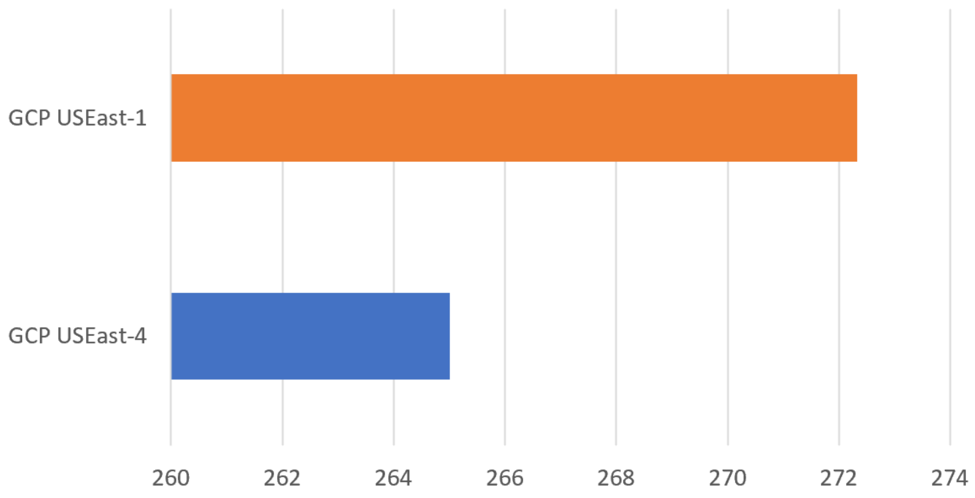


Figure 8.9: Google region pair comparison in parallel execution. The x-axis shows the number of seconds taken to finish each operation.

8.2.4 Conclusion

Hence, the experiments do not only approve the effectiveness of the new software tool, but as discussed earlier, along with the details of the storage region, the tool also provides the physical distance between the pipeline server and the storage server in KMs. This number will further be used in the evaluation matrix.

8.3 Experiment C - Cost of Cloud Storage

User requirements are given as an input to the newly developed software tool. Following are the parameters:

- StorageSpace in Gigabytes (GB)
- Bandwidth in Gigabytes (GB)
- Number of WRITE Operations
- Number of Read Operations

8.3.1 Case 1

User requirements are as follows:

```
Space = 5000 GB
Bandwidth = 15000 GB
WRITE Op. = 50000
READ Op. = 50000
```

Table 8.1 shows the cost of each cloud service provider, that is, AWS, Azure, and GCP in US dollars. There are separate columns for Space, Bandwidth, Write and Read operations, and then the fifth column shows the accumulated cost as well. It is clear from the estimation that Azure Cloud storage is the cheapest option, while Google Cloud storage is the most expensive. In addition to that, when WRITE operations are significantly expensive on GCP, READ operations are absolutely free.

	Cost of Space	**Cost of Bandwidth**	**Cost of Write Op.**	**Cost of Read Op.**	**Total Cost**
AWS	115	1350	0.25	0.2	1465.45
Azure	100	800	0.35	0.3	900.65
GCP	100	1720	2.5	0	1822.5

Table 8.1: Per month cost estimation of AWS, Azure and GCP for 5TB space, 15TB Bandwidth, 50 Thousand Write Operations and 50 Thousand Read Operations

Figure 8.10 shows the visual representation of cost comparison between different cloud storage providers based on the cost estimation provided by the software program. The color blue is allocated for Azure Cloud, Orange for AWS and Yellow for GCP. It is clear from the Figure 8.10 that the major cost is of the bandwidth, and Figure 8.11 shows that the bandwidth is cheapest on AWS while most expensive on GCP.

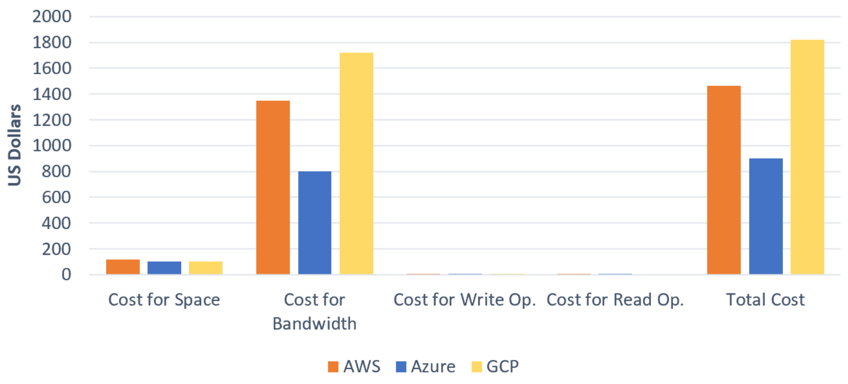


Figure 8.10: Per month cost estimation in US dollars of AWS, Azure and GCP for 5TB space, 15TB Bandwidth, 50 Thousand Write Operations and 50 Thousand Read Operations

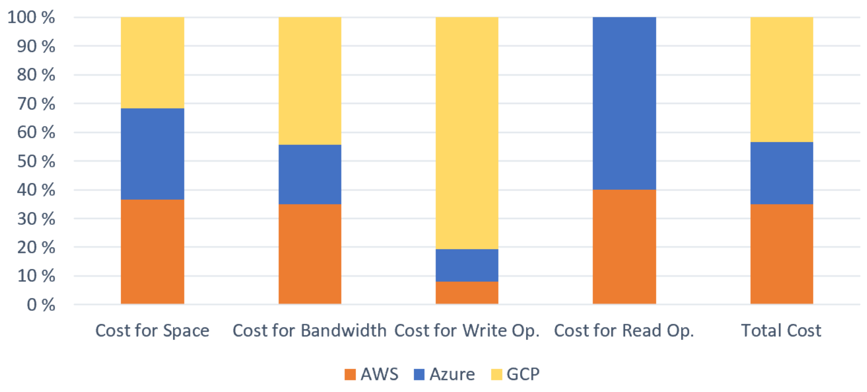


Figure 8.11: Per month cost comparison of AWS, Azure and GCP for 5TB space, 15TB Bandwidth, 50 Thousand Write Operations and 50 Thousand Read Operations.

8.3.2 Case 2

The same experiment is run with slightly different user requirements. Everything is kept the same except for the storage space. Updated user requirements are as follows:

```
Space = 50000 GB
Bandwidth = 15000 GB
```

```
WRITE Op. = 50000
READ Op. = 50000
```

Table 8.2 shows the cost estimation of each cloud service provider, that is, AWS, Azure, and GCP in US dollars and their comparison with each other. These values are based on the updated user requirements in which storage space has been increased from 5TB to 50TB.

	Cost of Space	**Cost of Bandwidth**	**Cost of Write Op.**	**Cost of Read Op.**	**Total Cost**
AWS	1150	1350	0.25	0.2	2500.45
Azure	1000	800	0.35	0.3	1800.65
GCP	1000	1720	2.5	0	2722.5

Table 8.2: Per month cost estimation of AWS, Azure and GCP for 50TB space, 15TB Bandwidth, 50 Thousand Write Operations and 50 Thousand Read Operations

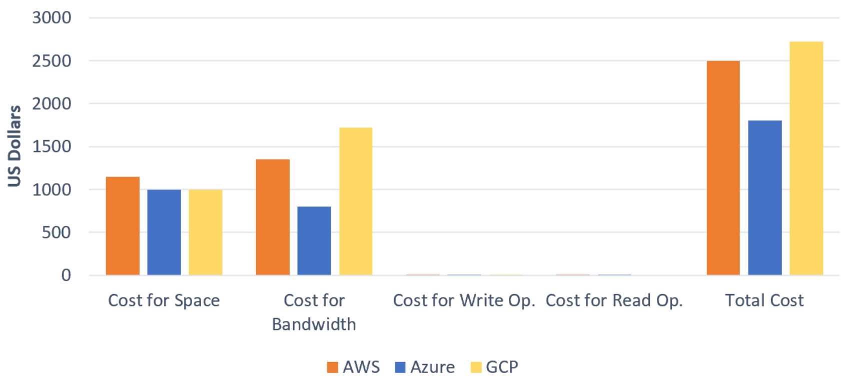


Figure 8.12: Per month cost comparison of AWS, Azure and GCP for 5TB space, 15TB Bandwidth, 50 Thousand Write Operations and 50 Thousand Read Operations.

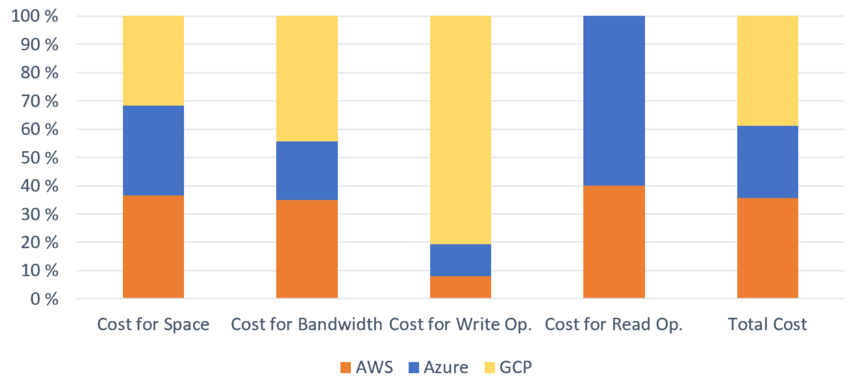


Figure 8.13: Per month cost comparison of AWS, Azure and GCP for 5TB space, 15TB Bandwidth, 50 Thousand Write Operations and 50 Thousand Read Operations.

8.3.3 Conclusion

A cost estimation is created based on two different requirements. The cost of storage based on the two test cases is similar for Azure and GCP, but it is slightly more with AWS. GCP has the highest price for bandwidth, whereas Azure has the lowest price. GCP has a significantly higher price for Write operations, but Read operations are completely free. In conclusion, bandwidth is proven to be an integral deciding factor in the total cost.

8.4 Experiment D - Network Performance:

Network performance relates to the customer's perception of a network's service quality. Because each network is unique in its nature and architecture, there are many different techniques to assess its performance. To assess the performance of the network in real-time, the algorithm deploys a sample data pipeline.

8.4.1 Experiment Setup

Following are the technical details for this experiment:

- **Data pipeline server:** Newyork Region, 1 CPU, 2GB RAM, 50GB SSD Storage. Operation system: Ubuntu 20.04
- **AWS s3 bucket:** us-east-1 region and us-east-2 region.
- **Azure cloud storage:** us-east-1 region and us-east-2 region. Server-side encryption with platform managed keys.
- **Data set:** Total 4MB. Data chunks: 3063. Each data chunk: Variable size.
- **Execution:** Serial execution of operations.

The reason for choosing a dataset with smaller data chunks and a large number of operations is to measure the network connection time for each operation.

The software tool not only executes READ operations; it also executes WRITE operations. The experiment is run three times, and the total execution time is recorded.

Two different AWS servers and two different Azure Cloud servers. The servers are selected based on the recommendation of the software tool explained in the Section 6.3.3. Since the data pipeline server is located in the NewYork region, there are two options for the storage server, us-east-1 or us-east-2. Based on the distance finder algorithm, us-east-1 is selected and termed as AWS Recommended and Azure Recommended. AWS Normal and Azure Normal are the names given to us-east-2.

8.4.2 Results & Conclusion

Figure 8.14 shows the result of the Azure and AWS storage. AWS in us-east-2 region took an average of 254 seconds, whereas Azure in us-east-2 region took an average of 195 seconds. For the storage servers in us-east-1 region, AWS took an average of 162 seconds, whereas Azure took an average of 88 seconds.

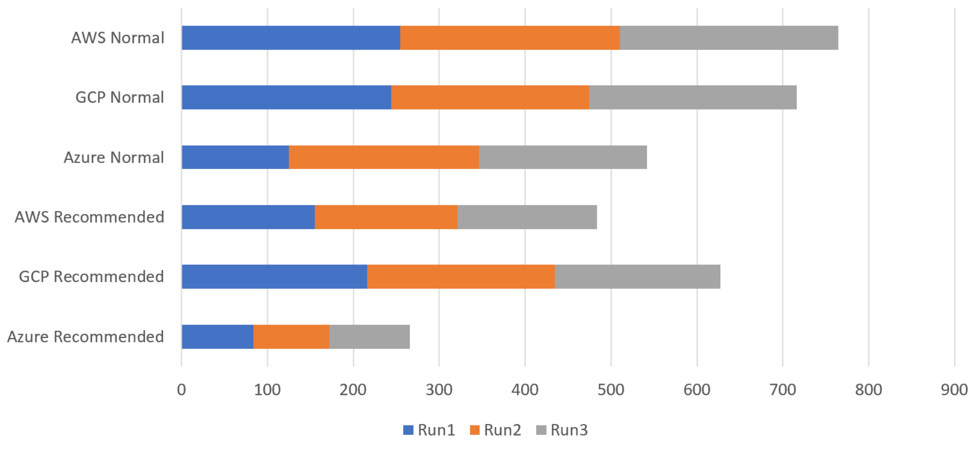


Figure 8.14: Network performance. The x-axis shows the number of seconds taken to finish each operation.

8.5 Experiment E - Server-side Encryption

Server side encryption relieves end users from the hassle of having to carry out the encryption themselves. Users can just upload files as is and then the cloud service provider will take care of encrypting them. But is there any trade-off with the performance in this case? To answer this question, this experiment is carried out. As explained in Section 7.4, storage servers are setup in the cloud with and without server-side encryption enabled. A data pipeline is deployed in the NewYork region.

8.5.1 Experiment Setup

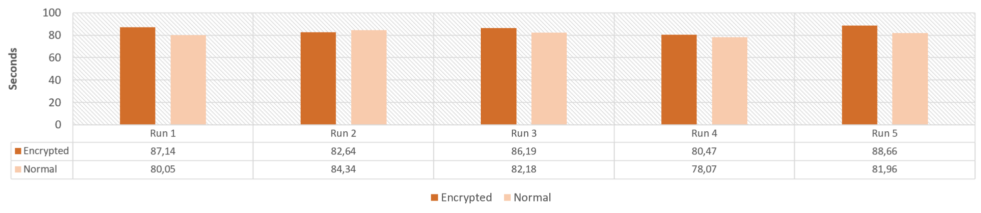
Following are the technical details for this experiment:

- **Data pipeline server:** Newyork Region, 1 CPU, 2GB RAM, 50GB SSD Storage. Operation system: Ubuntu 20.04
- **AWS s3 bucket:** us-east-1 region. Server-side encryption with s3 managed keys.
- **Azure cloud storage:** us-east-1 region. Server-side encryption with platform managed keys.
- **Data set:** Total 30GB. Data chunks: 100. Each data chunk: 30MB
- **Execution:** Parallel execution of data pipeline.

Unlike the data set used in other experiments, a comparatively larger dataset is used with bigger data chunks, so that the impact of real-time decryption can be observed.

8.5.2 Results - Azure s3 bucket

Figure 8.15 shows the comparison of the performance of a data pipeline with and without server-side encryption enabled. The data pipeline is executed five times. As it can be seen in the first execution of the data pipeline, with server-side encryption, it took 87.14 seconds to finish the process, whereas without encryption, it took 80.05 seconds. Hence the difference of 7 seconds for 30GB of data with 100 WRITE and 100 READ operations. Similarly, for the second execution, the difference is less than 2 seconds. Hence, it can be concluded that server-side encryption does impact the performance, but in most cases, that impact could be unnoticeable.



	Run 1	Run 2	Run 3	Run 4	Run 5
■ Encrypted	87,14	82,64	86,19	80,47	88,66
■ Normal	80,05	84,34	82,18	78,07	81,96

Figure 8.15: Comparison of performance with and without server-side encryption enabled with an AWS S3 bucket. The values in the table right beneath the bars show the actual value in seconds it took to finish the execution of the said operations.

Figure 8.16 shows a comparison of performance, instead of actual numbers, it shows a percentage. As it can be seen, the difference is quite small. Hence it can concluded that for AWS S3 storage, the server-side encryption does not have a noticeable impact on the performance.

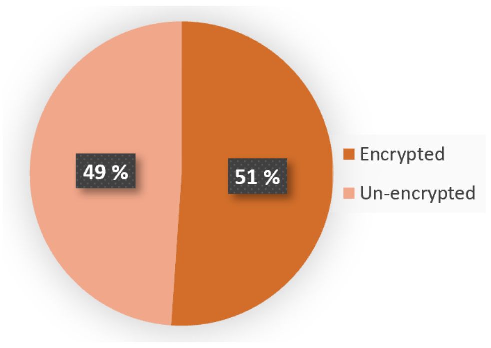
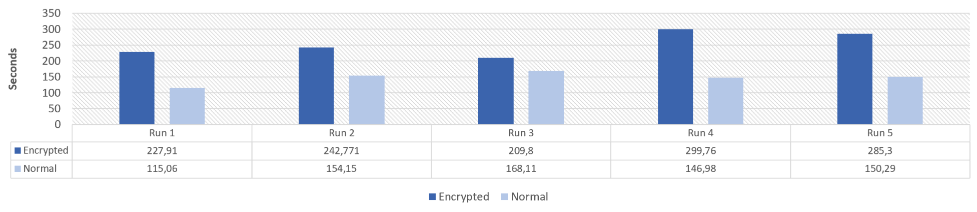


Figure 8.16: Comparison of performance with and without server-side encryption enabled with AWS S3 storage. When the same set of operations were run simultaneously, without server-side encryption enabled, it took 49% of the total time. Whereas, with the encrypted dataset, it took 51% of the time.

8.5.3 Results - Azure cloud storage

Figure 8.17 shows the comparison of the performance of a data pipeline with and without server-side encryption enabled on Azure cloud storage. The data pipeline is executed five times. As it can be seen in the first execution of the data pipeline, with server-side encryption, it took 227.91 seconds to finish the process, whereas without encryption, it took 115.06 seconds. Hence the difference of almost 113 seconds for 30GB of data with 100 WRITE and 100 READ operations. Similarly, for the second execution, the difference is less than 88.6 seconds. For a few iterations, the time consumption with server-side encryption is almost double that with no encryption. Hence, it can be concluded that for Azure cloud storage, server-side encryption does impact the performance and it is noticeable.



	Run 1	Run 2	Run 3	Run 4	Run 5
■ Encrypted	227,91	242,771	209,8	299,76	285,3
■ Normal	115,06	154,15	168,11	146,98	150,29

Figure 8.17: Comparison of performance with and without server-side encryption enabled with Azure cloud storage. The values in the table right beneath the bars show the actual value in seconds it took to finish the execution of the said operations.

Figure 8.18 shows a comparison of performance. Instead of actual numbers, it shows a percentage. As can be seen, the difference is quite significant when compared to what was observed for AWS. As a result, it can be concluded that, unlike AWS S3 storage, server-side encryption has a noticeable impact on performance for Azure cloud storage.

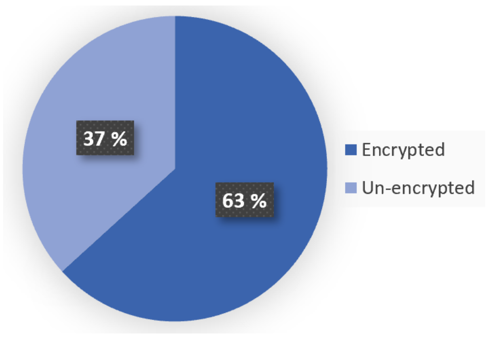
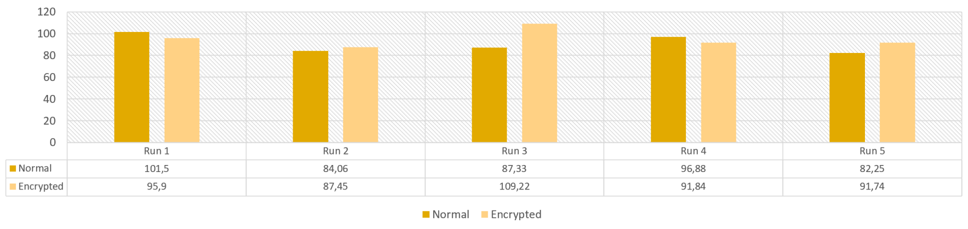


Figure 8.18: Comparison of performance with and without server-side encryption enabled with Azure cloud storage. When the same set of operations were run simultaneously, without server-side encryption enabled, it took 37% of the total time. Whereas, with the encrypted dataset, it took 67% of the time.

8.5.4 Results - Google cloud storage

Figure 8.19 shows the comparison of the performance of a data pipeline with and without server-side encryption enabled on Azure cloud storage. The data pipeline is executed five times. As it can be seen in the first execution of the data pipeline, with server-side encryption, it took 227.91 seconds to finish the process, whereas without encryption, it took 115.06 seconds. Hence the difference of almost 113 seconds for 30GB of data with 100 WRITE and 100 READ operations. Similarly, for the second execution, the difference is less than 88.6 seconds. For a few iterations, the time consumption with server-side encryption is almost double that with no encryption. Hence, it can be concluded that for Azure cloud storage, server-side encryption does impact the performance and it is noticeable.



	Run 1	Run 2	Run 3	Run 4	Run 5
■ Normal	101,5	84,06	87,33	96,88	82,25
■ Encrypted	95,9	87,45	109,22	91,84	91,74

Figure 8.19: Comparison of performance with and without server-side encryption enabled with Azure cloud storage. The values in the table right beneath the bars show the actual value in seconds it took to finish the execution of the said operations.

Figure 8.20 shows a comparison of performance. Instead of actual numbers, it shows a percentage. As it can be seen, the difference is quite high as compared to what has been observed for AWS. Hence, it can be concluded that for Azure cloud storage, unlike AWS S3 storage, the server-side encryption does have a noticeable impact on the performance.

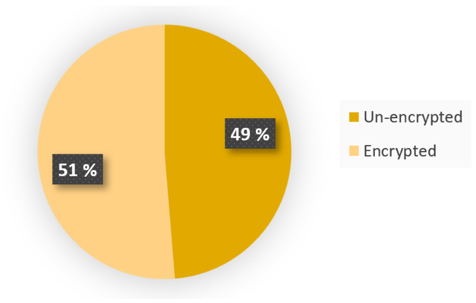


Figure 8.20: Comparison of performance with and without server-side encryption enabled with Azure cloud storage. When the same set of operations were run simultaneously, without server-side encryption enabled, it took 37% of the total time, whereas with the encrypted dataset, it took 67% of the time.

8.6 Experiment E - Evaluation matrix

There are a few ways of identifying the good and poor use cases, but it's more important that we understand the underlying workload characteristics, as many positives (or attractors) can still outweigh one or more negatives (detractors). Therefore, we should consider the application portfolio as a whole, as well as individually. Nobody wants to retain an active data centre for the sake of one server — the Pareto Principle and the assistance of a "palliative" care service provider are strong allies in this war.

8.6.1 User requirements / Attractors

The following are the user scenarios those are served well by the public cloud.

Case: Temporary Requirements

As the cloud uses a pay-as-you-go, utility-based pricing model, it is well-suited to short-term, transient workloads and projects. Example use cases: proof of concepts, pilots, application testing, product evaluations, labs and training environments, etc.

For these type of temporary applications, following user requirements have been setup:

```
Space = 1000 GB
Bandwidth = 15000 GB
WRITE Op. = 5000
READ Op. = 5000
```

Since the requirements are not clear or the resource consumption is not clear, equal weights are setup for all the parameters:

```
Cost = 25%
Distance = 25%
Network Performance = 25%
Encryption = 25%
```

Table 8.3 ranks the cloud service provider when a user puts equal weights for all three parameters that is 25% for cost, distance, encryption and network performance. As it can be seen, based on these input, GCP is the most suitable choice recommended by the algorithm, whereas, AWS and Azure are ranked second and third respectively.

Rank	Alternatives	Si	Ri	Qi
1	AWS	0.250378	0.150614	0
2	GCP	0.2803	0.25	0.529945
3	Azure	0.75	0.25	1

Table 8.3: Cloud service providers ranking for case 1. User weights are: network performance 25%, distance 25%, encryption 25%, and cost 25%.

Case: Highly variable workloads

Demand variability comes in two distinct flavours: predictable (seasonal, tidal, cyclical, etc.) and unpredictable. For example, month-end processing, on-season vs. off-season, morning vs. evening, Friday night, Christmas, etc. are all predictable surges in demand that can be planned around with processing capacity increased pro-actively according to a schedule or based on another trigger. Unpredictable, or unexpected demand, requires a reactive approach for scaling capacity and is triggered by a key event. The on-demand nature of the cloud, rapid elasticity, and a programmatic interface can be used to match supply with demand. This approach reduces the need for capacity planning, avoids the consequences from under-estimating peak loads, and avoids over-provisioning infrastructure that sits idle for long periods of time.

In this case, storage requirements are slightly increased as compared to the previous use cases. Bandwidth and WRITE and READ operations are increased significantly.

```
Space = 2000 GB
Bandwidth = 25000 GB
WRITE Op. = 10000
READ Op. = 10000
```

In terms of user weights, more focus is put on network performance.

```
Cost = 10%
Distance = 10%
Network Performance = 70%
Encryption = 10%
```

Table 8.4 ranks the cloud service provider when a user puts 70% weight on the network performance, 10% on the cost, and only 10% on the physical distance and 10% on encryption. It means, in this particular case, cost is a secondary factor, and network performance has clear priority over others. As it can be seen, based on these input, Azure is the most suitable choice recommended by the algorithm, whereas, GCP and AWS are ranked second and third respectively.

Rank	Alternatives	Si	Ri	Qi
1	Azure	0.3	0.1	0
2	AWS	0.451092	0.421718	0.451409
3	GCP	0.71212	0.7	0

Table 8.4: Cloud service providers ranking for case 1. User weights are: network performance 80%, distance 5% and cost 15%.

Case: High security, low scale/volume solutions

Although many customers fear the security of the cloud, there are many capabilities within AWS to restrict and monitor access to resources. For small-scale solutions that require a high-security perimeter, the cloud can provide an isolated, air-gapped sandbox that uses shared, highly available security apparatus that would be too costly to deploy in an on-premise or co-location environment.

For this scenario, the user requirements are similar to the first use case discussed in this chapter.

```
Space = 1000 GB
Bandwidth = 15000 GB
WRITE Op. = 5000
READ Op. = 5000
```

In terms of user weights, most weight is given to the the impact of encryption in terms of performance.

```
Cost = 5%
Distance = 5%
Network Performance = 10%
Encryption = 80%
```

Table 8.5 ranks the cloud service provider when a user puts 80% weight on the encryption, 10% on network performance, and only 5% on the cost and physical distance. It means, in this particular case, cheapest cost is the main goal, and performance and distance can be compromised if necessary. As it can be seen, based on these input, GCP is the most suitable choice recommended by the algorithm, whereas, AWS and Azure are ranked second and third respectively.

Rank	Alternatives	Si	Ri	Qi
1	AWS	0.0801982	0.0602455	0
2	GCP	0.10606	0.1	0.0426433
3	Azure	0.9	0.8	1

Table 8.5: Cloud service providers ranking for case 1. User weights are: encryption impact: 80%, network performance 10%, distance 5% and cost 5%.

Case: Dormant workloads

A dormant workload occupies no compute capacity and generates no network traffic, reducing the running costs to just storage. Re-animating such a workload through an API call provides some unique opportunities to minimize costs. Example use cases: test/development, UAT, unit and system testing, QA environments, cold disaster recovery sites, etc.

In this case, the storage are set to 5TB, whereas the bandwidth, WRITE and READ operations requirements are reduced significantly.

```
Space = 5000 GB
Bandwidth = 1000 GB
WRITE Op. = 2000
READ Op. = 2000
```

In terms of user weights, cost is given the highest weight, whereas the distance, network performance and encryption are given same weights.

```
Cost = 70%
Distance = 10%
Network Performance = 10%
Encryption = 10%
```

Rank	Alternatives	Si	Ri	Qi
1	GCP	0.11212	0.1	0.0098647
2	AWS	0.14817	0.0879241	0.0228776
3	Azure	0.9	0.7	1

Table 8.6: User weights are: cost: 70%, network performance 10%, distance 10% and cost 10%.

8.6.2 Conclusion

The research project has discussed and evaluated various user scenarios such as dormant workload, variable high demand, high security requirements etc. For each scenario, a user of user requirements were assumed and user weights were set. Furthermore, cloud server providers such GCP, AWS and Azure were ranked using the novel developed evaluation matrix model with MCDA algorithm.

Results are shown in Table 8.4, Table 8.5, Table 8.3 and Table 8.6. It can be seen that each public cloud has it's pros and cons.

Cost

If one is cheaper in terms of storage, the other one is cheaper in terms of bandwidth and network operations. For example, storage is comparatively expensive in GCP but it's READ operations are entirely free.

Physical Distance

Physical distance is a highly variable characteristic in this situation as the data pipeline server can be located anywhere. So no clear conclusion can be drawn for that.

Network performance

Network performance can vary from provider to provider and from data center to data center, as it is entirely related to the physical network structure of the data center.

Encryption

All cloud service providers use more or less the same process and exactly same key size for their server side data encryption and decryption. But it is interesting to see the variable impact of encryption on the performance by each provider.

User weights

When it comes to multi criteria decision analysis, the user weights are basically the deciding factors. As discussed earlier, each cloud service provider has its own pros and cons, and there are huge variations when they are combined with the user scenarios. So user weights are a good way to prioritize the requirements and then rank them accordingly.

Part V

Discussions & Future Work

Chapter 9

Discussions & Conclusion

The research work done in this thesis can be divided in the following artifacts:

1. A novel architecture for implementing big data pipelines with integration cloud storage.
    a. Hybrid-infrastructure i.e on-premise processing and on-cloud storage
    b. Use of more than one cloud storage instead of just one
    c. Use of local storage storage temporarily for inter-step data input and output.

   A big data pipeline is deployed and tested with all the above mentioned characteristics.
2. Evaluation criteria to rank different cloud storage options based on the user's requirements. Four parameters are selected, and software tools are developed accordingly. The results from each software tool are then put into a matrix (called an evaluation matrix) and used as input for the ranking algorithm (also called an evaluation method).
3. A domain-specific language is developed with extensive vocabulary incorporating not only cloud storage options, but also container security and monitoring packages.

The thesis verified four major novel concepts for big data pipelines by conducting these detailed experiments. 1) Use of Software Containers 2) the use of a domain-specific language for data pipeline deployment, 3) the effectiveness of the proposed pipeline architecture with StaaS integration, and last but not least, 4) the effectiveness of the new evaluation model as a whole as well as the promising results of software programs designed for individual parameters. In the following sections, each artifact with respect to the research questions are discussed in Section 4.1.

9.1 State of the art

RQ 1: How STaaS is better than local storage for a big data pipeline? A comprehensive literature survey is carried out that suggested that STaaS is better than local storage for big data pipelines in terms of capital expenditure, security, data backup, data redundancy, data availability and data migration. Detailed study has been presented in section

9.2 Novel architecture

RQ 2: How big data pipelines can be deployed using StaaS?

The proposed architecture reduces the complexity of the big data pipeline by moving storage to the cloud continuum. By doing so, a lot of extra overhead of data backup and security is also moved to the cloud service provider. In addition to that, cloud storage provides easy and on-demand scability in case more storage space is required. Also comes up with a flexible pay-as-you-go pricing plan.

Another purpose that we addressed is to construct pipelines stages into discrete containers so that step specific logic may be built using any programming language and make its deployment scalable. Hence, I utilised an approach in which a given big data pipelines is composed of separately produced pipeline stages. Containers for each pipeline stages are made using generic images. Moreover, depending on the workload, variable number of step containers may be supplied. Such strategy allows us to allocate more instances whenever speedier processing is desired. During pipelines operation, it was desired to have inter-step communication with a coordinated flow of data. This was solved by developing an external communication middleware solution. This middleware binds the execution of phases allowing them to pass data and form a pipelines. In addition, it standardises the link between the phases, allowing them to run in parallel without encountering a race issue.

Finally, we can say that the proposed big data pipeline solution allows us to separate the design and run-time parts of a particular pipeline, resulting in a scalable and race condition-free pipeline execution. Moreover, it promotes the cooperation of domain and technical specialists to produce a fine-tuned pipeline concept and composition.

9.3 Evaluation Matrix

RQ 3: How do we rank different cloud storage options based on the user requirements?

The research project has developed, discussed and evaluated various user scenarios such as dormant workload, variable high demand, high security requirements etc. using the newly proposed evaluation method. For each scenario, a user of user requirements were assumed and user weights were set. Furthermore, cloud server providers such GCP, AWS and Azure were ranked using the novel developed evaluation matrix model with MCDA algorithm. The solution provided a very clear recommendations regarding the cloud storage options.

9.3.1 Physical distance

Physical distance is a highly variable characteristic while selecting the storage facility as the data pipeline server can be located anywhere. So no clear conclusion can be drawn for that as of which provider has closest facility available.

However, the experiments proved the positive affect of physical distance within the same cloud storage provider. Whereas, it was an interesting observation to see variability of the result between different cloud storage providers. The experiments do not only approve the effectiveness of the new software tool, but along with the details of the storage region, the tool also provides the physical distance between the pipeline server and the storage server in KMs. This number is further used in the evaluation matrix.

9.3.2 Cost of cloud storage

If one storage provider is cheaper in terms of storage, the other one is cheaper in terms of bandwidth and network operations. For example, storage is comparatively expensive in GCP but it's READ operations are entirely free. So it entirely depends on the users' operational requirements.

In summary:

- The cost of storage based on the two test cases is similar for Azure and GCP but it is slightly more with AWS.
- GCP has the highest price for Bandwidth whereas the Azure has lowest price.
- GCP has significantly higher price for Write operations but Read operations are completely free.

In conclusion, bandwidth is proved to be an integral deciding factor in the total cost.

9.3.3 Network performance

Network performance can vary from provider to provider and from data center to data center, as it is entirely related to the physical network structure of the data center.

For example, two different servers on AWS as well as two different servers on Azure Cloud. The servers are selected based on the recommendation of the software tool explained in the Section 6.3.3. Since the data pipeline server is located in NewYork region, so there are two options for the storage server, us-east-1 or us-east-2. Based on the distance finder algorithm, us-east-1 is selected and termed as AWS Recommended and Azure Recommended. us-east-2 is termed as AWS Normal and Azure Normal. A clear performance difference can be observed in performance between the storage units from the same cloud storage provider. Whereas, it can be seen that even with a slightly larger physical distance, the storage units between different cloud storage providers can have better performance.

9.3.4 Server-side encryption

All cloud service providers use more or less the same process and exactly same key size for their server side data encryption and decryption. But it is interesting to see the variable impact of encryption on the performance by each provider.

For AWS, it can be seen, the difference is quite small. Hence it can concluded that for AWS S3 storage, the server-side encryption does not have a noticeable impact on the performance. When same set of operations was run simultaneously, without server-side encryption enabled, it took 49% of the total time. Whereas, with the encrypted dataset, it took 51% of the time. In case of Azure cloud, it can be seen, the difference is quite high as compared to what it has been observed for AWS. Hence it can concluded that for Azure cloud storage, unlike AWS S3 storage, the server-side encryption does have a noticeable impact on the performance. When same set of operations was run simultaneously, without server-side encryption enabled, it took 37% of the total time. Whereas, with the encrypted dataset, it took 67% of the time.

9.3.5 User weights

When it comes to multi criteria decision analysis, the user weights are basically the deciding factors. As discussed earlier, each cloud service provider has its own pros and cons, and there are huge variations when they are combined with the user scenarios. So user weights are a good way to prioritize the requirements and then rank them accordingly.

9.4 Domain-specific language (DSL)

RQ 4: How can StaaS options be incorporated with DSL for big data Pipelines?

Initially, it was planned to provide a new DSL to allow the high-level specification and technology-independent design of big data pipelines. This was managed by building a specially built DSL that allows high-level pipelines development and hides complicated technology specifics. The DSL gives an abstraction of essential concepts and parameters that are necessary to create a particular pipeline. This makes it conceivable to have a separation between pipelines definition and its execution.

The new DSL has a flexible class structure to integrate large number of third party applications, support for security and monitoring packages. Most importantly, ability to integrated cloud storage in added in the DSL vocabulary. In addition to that, the DSL has now enhanced vocabulary to integrate a specific type of storage option such as MYSQL or NoSQL from a specific cloud storage. To further ease the usability, the user has an option to either specify the container image path for security, communication medium, monitoring etc. If a user specifies a particular package, the compiler would take that into account at the time of deployment, otherwise it would install the standard publicly available packages. To improve the usability of the class objects, single class for commands and environment variable is developed and reused at workflow level, step level, and even at single packages level such as security and storage.

Chapter 10

Future Work

In this chapter, I will discuss the possibilities of enhancement in the proposed model and the newly developed domain-specific language.

Evaluation Matrix: This master thesis has proposed a novel evaluation model to rank cloud service providers based on 1) cost, 2) network performance, 3) physical distance, and 4) impact of server-side encryption. Following are the possible future enhancement for the evaluation matrix:

1. **Cost:** At this stage, the data for the cost is collected only for the storage in the Europe west region. Although the cost will not vary a lot, but it is a possibility to collect data for all available regions.
2. **Server-side encryption:** At this stage, server-side encryption with server managed keys is evaluated. Server-side encryption with client managed keys can also be evaluated. In addition to that, it's impact on the cost can also be considered.
3. **More parameters:** More parameters can be added into the evaluation matrix that play role in making the decision to select the cloud storage.

Domain-specific Language (DSL): In future, there is a clear possibility of new parameters, such as cost, bandwidth, and network latency. For the domain-specific language, the work can be extended in multiple directions now:

1. **Compiler construction:** At this stage, the language meta-model is is developed. Whereas, the compiler construction/development is in progress. The compiler or the *generator class* is used to generate YAML code based on the code written in this new language.
2. **Grammer enhancement:** By enhancing the DSL grammar to support more concepts from computing continuum.
3. **Graphical user interface:** Development of a graphical User Interface over the language compiler so that the user can deploy a big data pipeline by using a user-friendly, drag-and-drop interface.

4. **Evaluation:** In order to test the effectiveness of the new DSL, usability tests can be carried out.

Integration of evaluation matrix with the DSL: The language meta-model is designed to support the integration of cloud storage with the big data pipeline. In addition to that, multiple storage services can bed integrated with the data pipeline between different compute steps. But the integration of the new proposed evaluation model/matrix is missing. Hence, there is a possibility to integrate this model to the language itself to enhance the usability.

Bibliography

[1] M. AbdelBaky, M. Zou, A. R. Zamani, E. Renart, J. Diaz-Montes and M. Parashar, ‘Computing in the continuum: Combining pervasive devices and services to support data-driven applications,’ in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017, pp. 1815–1824.

[2] S. Robinson and R. Ferguson, ‘The storage and transfer challenges of big data,’ *MIT Sloan Management Review*, vol. 7, 2012.

[3] C. Yang, Y. Xu and D. Nebert, ‘Redefining the possibility of digital earth and geosciences with spatial cloud computing,’ *International Journal of Digital Earth*, vol. 6, no. 4, pp. 297–312, 2013.

[4] M. Padgavankar, S. Gupta *et al.*, ‘Big data storage and challenges,’ *International Journal of Computer Science and Information Technologies*, vol. 5, no. 2, pp. 2218–2223, 2014.

[5] J. Chen. ‘Azure fundamental: Iaas, paas, saas.’ (2020), [Online]. Available: `https://medium.com/chenjd-xyz/azure-fundamental-iaas-paas-saas-973e0c406de7` (visited on 04/05/2022).

[6] T. Lanfear. ‘Shared responsibility in the cloud.’ (2022), [Online]. Available: `https://docs.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility` (visited on 04/05/2022).

[7] H. Lunell, *Code generator writing systems*. Software systems research center [Inst. för informationsbehandling], Univ., 1983.

[8] R. L. Glass, V. Ramesh and I. Vessey, ‘An analysis of research in computing disciplines,’ *Communications of the ACM*, vol. 47, no. 6, pp. 89–94, 2004.

[9] I. Nordin, *Using Knowledge: On the Rationality of Science, Technology, and Medicine*. Lexington Books, 2017.

[10] S. Döringer, “the problem-centred expert interview’. combining qualitative interviewing approaches for investigating implicit expert knowledge,’ *International journal of social research methodology*, vol. 24, no. 3, pp. 265–278, 2021.

[11] A. Booth, A. Sutton and D. Papaioannou, ‘Systematic approaches to a successful literature review,’ 2016.

[12] S. K. Ratan, T. Anand and J. Ratan, 'Formulation of research question–stepwise approach,' *Journal of Indian Association of Pediatric Surgeons*, vol. 24, no. 1, p. 15, 2019.

[13] A. S. Denney and R. Tewksbury, 'How to write a literature review,' *Journal of criminal justice education*, vol. 24, no. 2, pp. 218–234, 2013.

[14] A. Brettle and M. Raynor, 'Developing information literacy skills in pre-registration nurses: An experimental study of teaching methods,' *Nurse Education Today*, vol. 33, no. 2, pp. 103–109, 2013.

[15] G. S. Ramachandran, R. Radhakrishnan and B. Krishnamachari, 'Towards a decentralized data marketplace for smart cities,' in *2018 IEEE International Smart Cities Conference (ISC2)*, Sep. 2018, pp. 1–8. DOI: `10.1109/ISC2.2018.8656952`.

[16] A. Ashabi, S. B. Sahibuddin and M. S. Haghighi, 'Big data: Current challenges and future scope,' in *2020 IEEE 10th Symposium on Computer Applications Industrial Electronics (ISCAIE)*, Apr. 2020, pp. 131–134. DOI: `10.1109/ISCAIE47305.2020.9108826`.

[17] M. Ghasemaghaei and G. Calic, 'Assessing the impact of big data on firm innovation performance: Big data is not always better data,' *Journal of Business Research*, vol. 108, pp. 147–162, 1st Jan. 2020, ISSN: 0148-2963. DOI: `10.1016/j.jbusres.2019.09.062`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0148296319305740` (visited on 01/05/2021).

[18] M. Smallcombe. 'What is Big Data Pipeline?' (2021), [Online]. Available: `https://www.xplenty.com/glossary/what-is-big-data-pipeline/` (visited on 04/05/2021).

[19] *The cloud continuum*, `https://www.architectureandgovernance.com/app-tech/the-cloud-continuum/`, Accessed: 2021-12-02.

[20] F. Bonomi, 'Connected vehicles, the internet of things, and fog computing. the eighth acm international workshop on vehicular inter-networking (vanet),' 2011.

[21] M. Abdelshkour. 'Iot, from cloud to fog computing.' (2015), [Online]. Available: `https://blogs.cisco.com/perspectives/iot-from-cloud-to-fog-computing` (visited on 02/12/2021).

[22] D. Koehn, S. Lessmann and M. Schaal, 'Predicting online shopping behaviour from clickstream data using deep learning,' *Expert Systems with Applications*, vol. 150, p. 113 342, 15th Jul. 2020, ISSN: 0957-4174. DOI: `10.1016/j.eswa.2020.113342`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0957417420301676` (visited on 03/05/2021).

[23] A. S. Gillis. 'Storage as a service (staas).' (2019), [Online]. Available: `https://www.techtarget.com/searchstorage/definition/Storage-as-a-Service-SaaS` (visited on 04/05/2022).

[24] C. Youmin, L. Youyou, L. Shengmei and S. Jiwu, 'Survey on RDMA-based distributed storage systems,' *Journal of Computer Research and Development*, vol. 56, no. 2, p. 227, 1st Feb. 2019, ISSN: 1000-1239. DOI: `10.7544/issn1000-1239.2019.20170849`. [Online]. Available: `https://crad.ict.ac.cn/EN/10.7544/issn1000-1239.2019.20170849` (visited on 02/05/2021).

[25] Y. Elshater, P. Martin, D. Rope, M. McRoberts and C. Statchuk, 'A study of data locality in yarn,' in *2015 IEEE International Congress on Big Data*, 2015, pp. 174–181. DOI: `10.1109/BigDataCongress.2015.33`.

[26] T. Renner, L. Thamsen and O. Kao, 'CoLoc: Distributed data and container colocation for data-intensive applications,' in *2016 IEEE International Conference on Big Data (Big Data)*, Washington DC,USA: IEEE, Dec. 2016, pp. 3008–3015, ISBN: 978-1-4673-9005-7. DOI: `10.1109/BigData.2016.7840954`. [Online]. Available: `http://ieeexplore.ieee.org/document/7840954/` (visited on 27/12/2020).

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, 'Spark: Cluster computing with working sets,' in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, Boston, MA: USENIX Association, 2010, p. 10.

[28] N. S. Naik, A. Negi, T. B. B.r. and R. Anitha, 'A data locality based scheduler to enhance MapReduce performance in heterogeneous environments,' *Future Generation Computer Systems*, vol. 90, pp. 423–434, 1st Jan. 2019, ISSN: 0167-739X. DOI: `10.1016/j.future.2018.07.043`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0167739X18308379` (visited on 02/05/2021).

[29] D. Zhao, M. Mohamed and H. Ludwig, 'Locality-aware scheduling for containers in cloud computing,' *IEEE Transactions on Cloud Computing*, vol. 8, no. 2, pp. 635–646, 1st Apr. 2020, ISSN: 2168-7161, 2372-0018. DOI: `10.1109/TCC.2018.2794344`. [Online]. Available: `https://ieeexplore.ieee.org/document/8259462/` (visited on 27/12/2020).

[30] E. H. Bourhim, H. Elbiaze and M. Dieye, 'Inter-container communication aware container placement in fog computing,' in *2019 15th International Conference on Network and Service Management (CNSM)*, Halifax, NS, Canada: IEEE, Oct. 2019, pp. 1–6, ISBN: 978-3-903176-24-9. DOI: `10.23919/CNSM46954.2019.9012671`. [Online]. Available: `https://ieeexplore.ieee.org/document/9012671/` (visited on 27/12/2020).

[31] E. M. Maximilien, H. Wilkinson, N. Desai and S. Tai, 'A domain-specific language for web apis and services mashups,' in *International Conference on Service-Oriented Computing*, Springer, 2007, pp. 13–26.

[32] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[33] T. Sheard and S. P. Jones, 'Template meta-programming for haskell,' in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002, pp. 1–16.

[34] R. D. Kelker, *Clojure for domain-specific languages*. Packt Publishing Ltd, 2013.

[35] M. Dayarathna and T. Suzumura, ‘A first view of exedra: A domain-specific language for large graph analytics workflows,’ in *Proceedings of the 22nd International Conference on World Wide Web*, ser. WWW ’13 Companion, Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, pp. 509–516, ISBN: 9781450320382. DOI: `10.1145/2487788.2487985`. [Online]. Available: `https://doi.org/10.1145/2487788.2487985`.

[36] Volter, ‘Md* best practices,’ *Journal of Object Technology*, vol. 8, no. 6, pp. 79–102, Sep. 2009, (column), ISSN: 1660-1769. DOI: `10.5381/jot.2009.8.6.c6`. [Online]. Available: `http://www.jot.fm/contents/issue_2009_09/column6.html`.

[37] Y. Zhao, X. Fei, I. Raicu and S. Lu, ‘Opportunities and challenges in running scientific workflows on the cloud,’ in *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2011, pp. 455–462. DOI: `10.1109/CyberC.2011.80`.

[38] E. Deelman, G. Singh, M. Livny, B. Berriman and J. Good, ‘The cost of doing science on the cloud: The montage example,’ in *SC’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Ieee, 2008, pp. 1–12.

[39] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer and D. Epema, ‘Performance analysis of cloud computing services for many-tasks scientific computing,’ *IEEE Transactions on Parallel and Distributed systems*, vol. 22, no. 6, pp. 931–945, 2011.

[40] M. Abouelhoda, S. A. Issa and M. Ghanem, ‘Tavaxy: Integrating taverna and galaxy workflows with cloud computing support,’ *BMC bioinformatics*, vol. 13, no. 1, pp. 1–19, 2012.

[41] J. Wang and I. Altintas, ‘Early cloud experiences with the kepler scientific workflow system,’ *Procedia Computer Science*, vol. 9, pp. 1630–1634, 2012.

[42] A. Celesti, A. Galletta, M. Fazio and M. Villari, ‘Towards hybrid multi-cloud storage systems: Understanding how to perform data transfer,’ *Big Data Research*, vol. 16, pp. 1–17, 2019.

[43] A. M. Ghosh and K. Grolinger, ‘Edge-cloud computing for internet of things data analytics: Embedding intelligence in the edge with deep learning,’ *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 2191–2200, 2020.

[44] R. Di Pietro, M. Scarpa, M. Giacobbe and A. Puliafito, ‘Secure storage as a service in multi-cloud environment,’ in *International Conference on Ad-Hoc Networks and Wireless*, Springer, 2017, pp. 328–341.

[45] H. El-Sayed, S. Sankar, M. Prasad, D. Puthal, A. Gupta, M. Mohanty and C.-T. Lin, 'Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment,' *IEEE Access*, vol. 6, pp. 1706–1717, 2017.

[46] E. Ahmed, A. Ahmed, I. Yaqoob, J. Shuja, A. Gani, M. Imran and M. Shoaib, 'Bringing computation closer toward the user network: Is edge computing the solution?' *IEEE Communications Magazine*, vol. 55, no. 11, pp. 138–144, 2017.

[47] Z. Xiong, Y. Zhang, D. Niyato, P. Wang and Z. Han, 'When mobile blockchain meets edge computing,' *arXiv preprint arXiv:1711.05938*, 2017.

[48] DSM. 'What data availability means and how your business can achieve it.' (2021), [Online]. Available: `https://www.dsm.net/it-solutions-blog/achieve-data-availability` (visited on 25/11/2021).

[49] George Soler. 'Cross-region replication in Azure: Business continuity and disaster recovery.' (2022), [Online]. Available: `%5Curl%7Bhttps://docs.microsoft.com/en-us/azure/availability-zones/cross-region-replication-azure%7D` (visited on 04/05/2022).

[50] C. Wray. 'What does redundancy mean in the cloud?' (2016), [Online]. Available: `https://www.rsaweb.co.za/what-does-redundancy-mean-in-the-cloud/` (visited on 25/11/2021).

[51] A. Bagaeen, S. Al-Zoubi, R. Al-Sayyed and A. Rodan, 'Storage as a service (staas) security challenges and solutions in cloud computing environment: An evaluation review,' in *2019 Sixth HCT Information Technology Trends (ITT)*, 2019, pp. 208–213. DOI: `10.1109/ITT48889.2019.9075097`.

[52] *Storage as a service (staas) is the practice of using public cloud storage resources to store your data. using staas is more cost efficient than building private storage infrastructure, especially when you can match data types to cloud storage offerings.* `https://www.intel.co.jp/content/www/jp/ja/cloud-computing/storage-as-a-service.html`, Accessed: 2021-11-25.

[53] Y. Zhang, W. Liu and J. Song, 'A novel solution of distributed file storage for cloud service,' in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, IEEE, 2012, pp. 26–31.

[54] C.-W. Lee, K.-Y. Hsieh, S.-Y. Hsieh and H.-C. Hsiao, 'A dynamic data placement strategy for hadoop in heterogeneous environments,' *Big Data Research*, vol. 1, pp. 14–22, 2014.

[55] L. Wei-Wei, 'An improved data placement strategy for hadoop,' *Journal of South China University of Technology (Natural Science Edition)*, vol. 1, p. 028, 2012.

[56] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares and X. Qin, 'Improving mapreduce performance through data placement in heterogeneous hadoop clusters,' in *2010 IEEE international symposium on parallel & distributed processing, workshops and Phd forum (IPDPSW)*, IEEE, 2010, pp. 1–9.

[57] D. Yuan, Y. Yang, X. Liu and J. Chen, 'A data placement strategy in scientific cloud workflows,' *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1200–1214, 2010.

[58] Z. Er-Dun, Q. Yong-Qiang, X. Xing-Xing and C. Yi, 'A data placement strategy based on genetic algorithm for scientific workflows,' in *2012 Eighth International Conference on Computational Intelligence and Security*, IEEE, 2012, pp. 146–149.

[59] I. Portugal, P. Alencar and D. Cowan, 'A survey on domain-specific languages for machine learning in big data,' *arXiv preprint arXiv:1602.07637*, 2016.

[60] T. Kosar, P. E. Martı, P. A. Barrientos, M. Mernik *et al.*, 'A preliminary study on various implementation approaches of domain-specific language,' *Information and software technology*, vol. 50, no. 5, pp. 390–405, 2008.

[61] F. Rabbi and W. MacCaull, 'T: A domain specific language for rapid workflow development,' in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2012, pp. 36–52.

[62] M. Slawik and A. Küpper, 'A domain specific language and a pertinent business vocabulary for cloud service selection,' in *International Conference on Grid Economics and Business Models*, Springer, 2014, pp. 172–185.

[63] A. Kashlev and S. Lu, 'A system architecture for running big data workflows in the cloud,' in *2014 IEEE International Conference on Services Computing*, 2014, pp. 51–58. DOI: 10.1109/SCC.2014.16.

[64] J. Novino. 'Microservices to workflows: Expressing business flows using an f# dsl.' (2019), [Online]. Available: https://medium.com/@jnovino/microservices-to-workflows-expressing-business-flows-using-an-f-dsl-d2e74e6d6d5e.

[65] Y. D. Dessalk, N. Nikolov, M. Matskin, A. Soylu and D. Roman, 'Scalable execution of big data workflows using software containers,' in *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, 2020, pp. 76–83.

[66] N. Nikolov, Y. D. Dessalk, A. Q. Khan, A. Soylu, M. Matskin, A. H. Payberah and D. Roman, 'Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers,' *Internet of Things*, p. 100 440, 2021.

[67] A. Li, X. Yang and M. Zhang, '{Cloudcmp}: Shopping for a cloud made easy,' in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[68] W. Zeng, Y. Zhao and J. Zeng, 'Cloud service and service selection algorithm research,' in *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, 2009, pp. 1045–1048.

[69] A. Wells. 'Hot, cool, and archive access tiers for blob data - azure storage.' (2022), [Online]. Available: `https://docs.microsoft.com/en-us/azure/storage/blobs/access-tiers-overview` (visited on 25/04/2022).

[70] B. Joan. 'Difference between amazon s3 and amazon ebs.' (2011), [Online]. Available: `http://www.differencebetween.net/technology/internet/difference-between-amazon-s3-and-amazon-ebs/` (visited on 25/04/2022).

[71] G. Cloud. 'Cloud storage pricing.' (), [Online]. Available: `https://cloud.google.com/storage/pricing` (visited on 25/04/2022).

[72] A. Shabtai, Y. Elovici and L. Rokach, 'Data leakage detection/prevention solutions,' in *A Survey of Data Leakage Detection and Prevention Solutions*, Springer, 2012, pp. 17–37.

[73] B. B. Gupta and O. P. Badve, 'Taxonomy of dos and ddos attacks and desirable defense mechanism in a cloud computing environment,' *Neural Computing and Applications*, vol. 28, no. 12, pp. 3655–3682, 2017.

[74] M. Factor, K. Meth, D. Naor, O. Rodeh and J. Satran, 'Object storage: The future building block for storage systems,' in *2005 IEEE International Symposium on Mass Storage Systems and Technology*, IEEE, 2005, pp. 119–123.

[75] Z. Daher and H. Hajjdiab, 'Cloud storage comparative analysis amazon simple storage vs. microsoft azure blob storage,' *International Journal of Machine Learning and Computing*, vol. 8, no. 1, pp. 85–9, 2018.

[76] Y. Tian, R. Babcock, C. Taylor and Y. Ji, 'A new live video streaming approach based on amazon s3 pricing model,' in *2018 IEEE 8th Annual computing and communication workshop and conference (CCWC)*, IEEE, 2018, pp. 321–328.

[77] P. Subhashini and S. Nalla, 'Data retrieval mechanism using amazon simple storage service and windows azure,' in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, IEEE, 2016, pp. 412–414.

[78] V. a. posts by Kenneth Hui, *Data Encryption in the Cloud, Part 4: Aws, Azure and Google Cloud*, [Online; accessed 2022-04-15], Mar. 2018.

[79] J. Johnson. 'What are availability regions and zones?' (2020), [Online]. Available: `https://www.bmc.com/blogs/cloud-availability-regions-zones` (visited on 25/04/2022).

[80] A. Ishizaka and P. Nemery, *Multi-criteria decision analysis: methods and software*. John Wiley & Sons, 2013.

[81] E. Triantaphyllou, 'Multi-criteria decision making methods,' in *Multi-criteria decision making methods: A comparative study*, Springer, 2000, pp. 5–21.

[82] J. Wątróbski, J. Jankowski, P. Ziemba, A. Karczmarczyk and M. Zioło, 'Generalised framework for multi-criteria method selection,' *Omega*, vol. 86, pp. 107–124, 2019.

[83] M. Patel, R. Sachan, A. Parihar and R. Sethi, 'Association between intima-media thickness of the carotid artery and risk factors for cardiovascular disease in patients on maintenance hemodialysis,' *International Journal of Scientific and Research Publications*, p. 2, 2012.

[84] A.-A. Corodescu, N. Nikolov, A. Q. Khan, A. Soylu, M. Matskin, A. H. Payberah and D. Roman, 'Locality-aware workflow orchestration for big data,' 2021.

[85] Techopedia. 'Network performance.' (2015), [Online]. Available: `https://www.techopedia.com/definition/30022/network-performance` (visited on 04/05/2022).

[86] M. Y. Ullah Khalid and M. Y. Ullah, *Monthly Roundup: February 2018 - Yasoob Khalid*, [Online; accessed 2022-04-25], Mar. 2018.

[87] V. K. Sharma, L. Murthy, K. Singh Saluja, V. Mollyn, G. Sharma and P. Biswas, 'Webcam controlled robotic arm for persons with ssmi,' *Technology and Disability*, vol. 32, no. 3, pp. 179–197, 2020.

[88] T. Oonuma. 'Python-pcl documentation.' (2018), [Online]. Available: `https://python-pcl-fork.readthedocs.io/en/rc_patches4/tutorial/` (visited on 25/04/2022).

Appendix A

Published work

A.1 Contribution to the published work

In *Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers* [66], I contributed to language grammar and conceptual design. In addition to that, I developed the compiler for the new DSL to convert the code written in the newly developed language into a deployable YAML code. The DSL presented in this project is an extension of DSL presented in [66]. In *Locality-Aware Workflow Orchestration for Big Data* [84], I contributed to doing the literature survey/compiling state-of-the-art for the paper, conceptual design, and evaluation. Moreover, I worked on deploying orchestration tools in the Kubernetes Environment on windows machines.

Internet of Things 16 (2021) 100440

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Research article

Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers

Nikolay Nikolov [a,*], Yared Dejene Dessalk [b], Akif Quddus Khan [c], Ahmet Soylu [d], Mihhail Matskin [b], Amir H. Payberah [b], Dumitru Roman [a]

[a] *SINTEF AS, Forskningsveien 1, 0373 Oslo, Norway*
[b] *KTH Royal Institute of Technology, Brinellvägen 8, 114 28 Stockholm, Sweden*
[c] *Norwegian University of Science and Technology — NTNU, Teknologivegen 22, 2815 Gjøvik, Norway*
[d] *OsloMet — Oslo Metropolitan University, Pilestredet 46, 0167 Oslo, Norway*

ARTICLE INFO

Keywords:
Big data workflows
Internet of Things
Domain-specific languages
Software containers

ABSTRACT

Big Data processing, especially with the increasing proliferation of Internet of Things (IoT) technologies and convergence of IoT, edge and cloud computing technologies, involves handling massive and complex data sets on heterogeneous resources and incorporating different tools, frameworks, and processes to help organizations make sense of their data collected from various sources. This set of operations, referred to as Big Data workflows, requires taking advantage of Cloud infrastructures' elasticity for scalability. In this article, we present the design and prototype implementation of a Big Data workflow approach based on the use of software container technologies, message-oriented middleware (MOM), and a domain-specific language (DSL) to enable highly scalable workflow execution and abstract workflow definition. We demonstrate our system in a use case and a set of experiments that show the practical applicability of the proposed approach for the specification and scalable execution of Big Data workflows. Furthermore, we compare our proposed approach's scalability with that of Argo Workflows – one of the most prominent tools in the area of Big Data workflows – and provide a qualitative evaluation of the proposed DSL and overall approach with respect to the existing literature.

1. Introduction

Massive amounts of data is being generated especially with the rise of Internet of Things (IoT) technologies creating new value creation opportunities through Big Data analysis. Accordingly, Big Data analysis has been a driving factor in revolutionizing major sectors, such as mobile services, finance, and scientific research. *Big Data workflows* are composed of multiple orchestrated steps, such as workflow activities that perform various data analytical tasks. They are different from business and scientific workflows since they are dynamic, process heterogeneous data, and are executed in parallel instead of a sequential set of scientific operators [1]. Although many organizations recognize the significance of Big Data analysis, they still face critical challenges when implementing data analytics into their process [2]. Firstly, multiple experts, ranging from technical to domain experts, need to be involved in specifying such complex workflows. Secondly, given the fact that IoT, Edge and Cloud technologies converge towards a computing continuum,

* Corresponding author.
E-mail addresses: nikolay.nikolov@sintef.no (N. Nikolov), yaredd@kth.se (Y.D. Dessalk), akif.q.khan@ntnu.no (A.Q. Khan), ahmet.soylu@oslomet.no (A. Soylu), misha@kth.se (M. Matskin), payberah@kth.se (A.H. Payberah), dumitru.roman@sintef.no (D. Roman).

https://doi.org/10.1016/j.iot.2021.100440
Received 14 February 2021; Received in revised form 27 July 2021; Accepted 29 July 2021
Available online 8 August 2021

workflow steps need to be mapped dynamically to heterogeneous computing and storage resources to ensure scalability [3,4]. Providing a scalable, general-purpose solution for Big Data workflows that a broad audience can use is an open research issue [2,4].

The challenges in devising an applicable generalized solution come from the fact that bottlenecks can occur on an individual workflow step level – for example, when the throughput of one step is lower than the others. Thus, scaling up the entire workflow does not address the scalability issues and needs to be done on the individual step level. This issue becomes worse by the fact that scalability needs to be organized and orchestrated over heterogeneous computing resources. Furthermore, scaling up individual steps introduces race conditions between step instances that attempt to process the same piece of data simultaneously. Another major challenge is achieving usability by multiple stakeholders as most Big Data processing solutions are focused on ad-hoc processing models that only trained professionals can use. However, organizations typically operate on specific software stacks, and getting experts in Big Data technology can introduce costs that are not affordable or practical. Even if an organization has the necessary technical personnel, data workflow steps pertain to specific domain-dependent knowledge, which is possessed by the domain experts rather than the data scientists who set up the data workflows. In this respect, this work aims to provide an approach that allows:

(a) Conceptualization of Big Data workflows using a domain-specific language (DSL) to support the high-level definition of complex data processing across multiple types of parameters, inputs, and outputs; and
(b) Scalable execution of Big Data workflows using software container technologies and message-oriented middleware solutions.

In this article, we present the design and implementation of a Big Data workflow approach based on the use of software container technologies, message-oriented middleware (MOM), and a DSL to enable highly scalable workflow execution and abstract workflow definition [5]. Our design allows for scaling up on the level of individual workflow steps on top of heterogeneous infrastructures while avoiding race conditions through a system of inter- and intra-step coordination. Furthermore, our container-based approach allows for the separation of concerns between the stakeholders by providing a flexible means of injecting domain-specific code involving any programming language and enabling the definition of workflows on a high level (i.e., without the step-specific code). Finally, the DSL allows easy specification of Big Data workflows by abstracting low-level technical aspects. We demonstrate our approach's applicability by implementing a prototype based on a real-world data workflow and multiple experiments showing satisfactory performance. Furthermore, a set of comparative experiments with Argo Workflows shows better performance due to concurrent workflow execution. A qualitative evaluation of the overall approach and the DSL with respect to the existing literature presents our approach's benefits. This article extends our previous work in [5] by providing (i) more insights and explanations about the motivation, approach and solution details, (ii) an extended presentation and analysis of the related work, (iii) an elaborate account and analysis of the requirements for enabling Big Data workflows on the Computing Continuum, (iv) a qualitative evaluation and discussion of the proposed DSL, and (v) a set of examples of real-life use cases of the approach.

The rest of the article is organized as follows. Section 2 sets the background, while Section 3 discusses related work. Section 4 describes the requirements and Section 5 presents the proposed approach. Section 6 discusses our proof-of-concept implementation based on the proposed design, Section 7 provides an evaluation, and, finally Section 8 concludes the article and discusses possible future work.

2. Background

In this section we briefly introduce technological background relevant in the context of scalable Big Data workflows execution.

2.1. Big data workflows

A Big Data workflow is the computerized modeling and automation of a process consisting of multiple orchestrated steps that perform various data analysis tasks [6]. In practice, most Big Data workflows are usually represented by a Directed Acyclic Graph (DAG) [7]. Various processing models can be applied for parallelizing data processing known as workflow data patterns [8]. In this context, Pipe and Filter (P&F) is a relevant architectural design to decompose a larger processing task into a series of smaller, separate processing steps (filters) that are connected by channels (pipes) [9]. The filters can then be integrated into a workflow, whereby each filter receives and sends data in a standardized way, thus implementing the "shared data passed by reference" pattern [8]. This pattern, given that different steps are loosely coupled, enables scalability at the workflow step level, but introduces the issue of handling concurrency control.

2.2. Message-oriented middleware

Achieving race-condition-free consistency and concurrency for scaling the homogeneous Big Data workflow steps requires using a synchronization mechanism across the different step instances. One approach for addressing such synchronization issues is to use Message-Oriented Middleware (MOM). MOM provides an infrastructure for loosely-coupled and asynchronous inter-process communication using messaging capabilities [10]. In a system integrated using MOM, a client can send messages to and receive from the other clients of the messaging system (without losses or message duplication) in a race-condition-free way through the use of a message queue. Thus, in the context of Big Data workflows, the middleware can act as a medium for communication, whereby step instances coordinate passing intermediate results through sending/receiving messages in MOM queues.

2.3. Container technology

A container is a packaged, standalone, deployable collection of program elements [11]. Containers provide an isolated virtual environment and include all the required dependencies of the provisioned tools. Docker is one of the most well-known platforms for organizing solutions based on container technologies. When executing multiple containers, a container orchestration system can manage their deployment, scaling, and networking. In particular, container technology is useful for deploying distributed scalable applications (such as Big Data workflows), as it provides transparent means for infrastructure management and easy scalability of individual application sub-components. Orchestration tools use a configuration file to define container images, network, and related deployment schemes of an application.

2.4. Domain-specific languages

DSLs are programming languages or specification languages that target a specific problem domain. DSLs are small, descriptive, and contain only the details needed for the desired domain. In general, there are two types of DSLs: internal and external [12]. An internal DSL is a specific form of Application Programming Interface (API) in a host general-purpose language (GPL). In contrast, an external DSL is a language that is parsed independently of the host GPL. Unlike GPLs, DSLs are limited in scope and cannot cover all aspects of a given problem. However, they are more effective than GPLs in providing expressiveness at the cost of generality. DSLs offer better domain-specificity and significantly improve collaboration between domain experts and developers [13].

3. Related work

In this section, related work is presented in terms of related scientific literature and commercial and non-commercial software tools. Aspects such as workflow resource scheduling and others, as described in [4,14], are complementary to our approach as the described concept uses container orchestration systems to specify them.

3.1. Related software tools

There is a large variety of Big Data workflow solutions that share similar design principles while fulfilling the needs of various groups of users and use cases. We carried out a comparative analysis of the most promising workflow tools (chosen based on their mass user base and relevance), including Pachyderm,[1] Apache Airflow,[2] Snakemake,[3] Apache NiFi,[4] Node-RED,[5] Argo Workflows,[6] NextFlow,[7] and Conductor.[8] An overview of the comparison is presented in Table 1 with respect to workflow type, usability for non-technical experts, run time container support, and generality of the solution.

We consider two main types of workflows — *scientific* and *general-purpose*. *Scientific* workflow approaches are built to be applied in homogeneous infrastructures, such as High-Performance Computing (HPC) clusters, where resources are shared between multiple workflows and dedicated algorithms are used to optimize job scheduling and resource allocation. On the other hand, *general-purpose* workflow tools are applicable in varying domains/scenarios and can be executed in heterogeneous computing infrastructures. In terms of *usability* for non-technical experts, we consider three levels of support — *easy*, *medium*, and *difficult*. We regard approaches that provide a dedicated DSL/UI that caters to domain experts as *easy*. If the approach requires the use of coding of the Big Data workflows in a specific programming language, we consider the approach of *medium* usability. On the other hand, if the approach requires coding and the use of approach-specific concepts and libraries in order to declare a workflow, we consider the approach to be *hard* in terms of usability to non-technical experts. With respect to *container support*, we classify the approaches on whether or not they support the use of container technologies (e.g., Docker) for encapsulating the entire Big Data workflow or individual steps. We note that although some of the tools, such as Apache NiFi, are packaged and deployable using container technology,[9] but we do not consider them to provide *container support*. We consider the solutions in terms of whether or not they are generic or catered for a specific vertical domain of knowledge. Thereby, we regard approaches that are applicable in any vertical domain as *generic* and vice versa. Finally, in terms of monitoring capabilities, we classify workflow tools by the availability of monitoring execution of the workflow. Depending on how the execution can be monitored, it can be further classified into *logging* and *runtime*. We regard those tools that provide real-time monitoring as *runtime*. If the monitoring is based on logs at the end of execution, we classify the tools as *logging*. Tools that mainly provide logging-based monitoring, which also support limited real time monitoring capabilities, are regarded as *logging and (partial) runtime*.

1 https://www.pachyderm.com.
2 https://airflow.apache.org.
3 https://snakemake.readthedocs.io.
4 https://nifi.apache.org.
5 https://nodered.org.
6 https://argoproj.github.io/argo.
7 https://www.nextflow.io.
8 https://netflix.github.io/conductor.
9 https://hub.docker.com/r/apache/nifi.

Table 1
Comparison matrix for tools supporting Big Data workflows.

Tool name	Workflow type	Usability	Container support	Generic solution	Monitoring
Airflow	General-purpose	Difficult	Docker	Yes	Logging and (partial) runtime
Argo	General-purpose	Difficult	Docker	Yes	Logging and (partial) runtime
Conductor	General-purpose	Difficult	No	Yes	Logging
Nextflow	Scientific	Medium	Docker	No	Runtime
NiFi	General-purpose	Medium	No	No	Runtime
Node-RED	General-purpose	Medium	No	No	Logging
Snakemake	General-purpose	Medium	Docker	Yes	Runtime
Pachyderm	General-purpose	Difficult	Docker	No	Runtime
Our approach	*General-purpose*	*Easy*	*Docker*	*Yes*	*Runtime*

Argo Workflows natively supports containers in workflows by implementing each step as a container. The workflow definition and automation are done by YAML templates based on a custom DSL.[10] The steps can be arranged either sequentially or in a DAG, making it possible to orchestrate and parallelize jobs. However, Argo Workflows does not have a middleware solution to handle inter-step communication, which may result in step instances running into race conditions when scaled horizontally. Additionally, individual steps cannot be scaled up in order to increase the workflow throughput (although they can be run in parallel).

Nextflow is a workflow framework based on the dataflow paradigm [15]. It uses a declarative processing model to execute parallel tasks and supports step-level scalability. Nextflow has built-in support for container technologies and the communication among processes is handled using channels, and asynchronous First-In-First-Out (FIFO) queues preventing race conditions. Nextflow provides a custom DSL to write complex workflows, which is an extension to Apache Groovy.[11] Nextflow does not provide a clear separation of concerns between workflow definition and implementation and relies on a specific software stack (through the DSL) for workflow step implementation.

Apache Airflow is a platform for the creation, scheduling, and monitoring of data workflows. Python scripts are used to describe DAG structured workflows. Airflow has a scheduler that executes workflows on a set of workers, but it lacks a mechanism to avoid race-conditions when scaled. Airflow has a rich user interface support allowing users to visualize workflows and monitor their execution. The workflow definition is done via programming. Pachyderm is a tool for managing data workflow and related input/output data that results in all the data workflows' reproducibility and scalability. Pachyderm is based on Docker and Kubernetes, and provides advanced features such as pluralization and incremental processing. Users need technical knowledge to define workflows.

Snakemake is a workflow system in which workflows are defined in terms of rules presenting the input–output conversion of files. Determining dependencies between rules, Snakemake automatically forms parallelizable DAG workflows. It provides a concise and readable DSL, an extension of Python programming language, for defining rules and workflow specific properties. It was initially developed for scalable workflows in bioinformatics. Apache NiFi is a project of Apache Software Foundations that allows automation of data flow between systems. It is based on a flow-based programming model for building scalable data workflows. Although it comes with a user interface, it requires technical knowledge to design workflows.

Node-RED is another flow-based programming workflow tool that is built based on Node.js run-time. It follows the event-driven and non-blocking model. It provides a Web-based visual editor for designing workflows, but it still requires technical knowledge from the user. Even though Node-RED was initially designed for Internet-of-Things (IoT) applications, it has evolved to develop various applications. Conductor is a workflow orchestration engine by Netflix that allows creating microservice-based business and process workflows. Workflows are composed of tasks that are executed by remote workers. A JSON-based DSL is used to define workflows.

3.2. Related studies

Many efforts have been made to use containers to address the challenges of scalability, resource provisioning, scheduling, orchestration, and data management of data workflows. Authors in [16] propose an approach for decomposing scientific workflows into micro-units that are containerized and contain sub-workflows that communicate through streaming middleware among each other and other applications and devices. This approach is specific to the domain of digital twins and addresses the job dependency, job scheduling, and streaming support issues, but does not provide means of independent scaling of steps and is domain-specific.

Another set of approaches relies on the use of container technology to deploy workers that execute jobs. Authors in [7] use Docker to deploy a software stack to homogenize the environment where tasks are run. However, their approach has limitations on the number of workflow step containers deployed on a single host and does not support long-running tasks (i.e., containers are shut down after executing). The approach in [17] uses containers to encapsulate workers that contain the workflow engine and execution environment but is applicable only in the context of workflows expressed in a specific stream-based dataflow DSL [18], which reduces applicability for general-purpose data workflows.

[10] https://argoproj.github.io/argo-workflows/fields.
[11] https://groovy-lang.org.

Another set of data workflow implementations in the area of HPC are built on top of the Shifter [19] framework. Authors in [20] use the framework to distribute jobs and entire workflows (encapsulated in a single container) over an HPC cluster. This approach does not support individual steps' scalability but views data workflows as entire units of work. Authors in [21] rely on Shifter for defining virtual HPC clusters to run jobs. Thereby, containers are used for creating worker nodes with the necessary HPC functionality and for managing the jobs, but their approach is not applicable for general-purpose data workflows.

One approach that comes close to the one described in this paper is presented in [1], whereby containers are used to wrap individual steps. The framework uses a TOSCA [22] to describe both the deployment and workflow steps, which containers can implement. However, the framework does not support dynamic workflows — tasks are executed in a sequential manner, whereby a task must finish for next to be deployed, which makes the approach not suitable for long-running workflows.

Finally, authors in [23] use Cloud orchestration for deploying distributed workflows in a similar manner as the one described here, although containers are not supported. However, the approach does not support run-time scalability and relies on the step definition to manage the input and output between the steps. Furthermore, the approach uses a DSL that provides no clear separation of concerns between the design- and run-time phases of data workflows.

4. Requirements

Commonly used data processing frameworks (such as Spark, Flink, Beam) are designed with ad-hoc processing models that technical experts on specific technology stacks can only use. The focus is on the programming and run-time aspects of workflows than the actual definition of workflows themselves. Even though some solutions, as discussed in related work, have demonstrated defining and executing Big Data workflows, they do not cater to the needs of domain experts. Solutions like Pachyderm, Snakemake, and Airflow are merely designed for technical experts, and the definition of workflows is done using high-level programming and scripting languages. To enable domain experts to participate in the process, some tools provide a user-friendly interface to define workflows. However, they either are made for a specific application domain (e.g., bioinformatics, computational chemistry, ecology, genomics, etc.) or require some level of technical knowledge to manipulate the data. Thus, although most approaches and tools use some form of DSL or UI, the abstraction level is not sufficient to allow for separation of concerns between definition and implementation, which is necessary to effectively involve domain experts in workflow definition.

Another major challenge in Big Data workflows is the dynamic mapping workflow steps to heterogeneous computing and storage resources to ensure scalability. This challenge comes from the fact that bottlenecks can occur on an individual workflow step level, e.g., the throughput of one step is lower than others. Thus, scaling up the entire workflow does not solve the scalability issues and needs to be done on the individual workflow step level. This issue is exacerbated by the fact that the scalability needs to be organized and orchestrated over heterogeneous computing resources. Achieving scalability in Big Data workflows has another dimension of challenges: exchanging data among workflow components and race conditions when multiple instances of workflow components try to modify a shared resource (e.g., a file) at the same time. Multiple workflow step instances can make up a large set of capabilities to deliver the workflow's needs. Some approaches and tools discussed in the related work attempt to address this issue through the use of containers and different types of middleware. However, no approach is able to unlock the full potential for achieving step-level scalability of workflows, which relies on workflow and step encapsulation.

Finally, in Big Data workflows, data need to be passed between the workflow components so that the communication overhead is minimal. A communication solution must decouple the communication between the steps to be scaled up while maintaining race-condition-free data access. Additionally, this communication module has to play a central role in determining data flow between workflow steps. Based on our analysis of the state of the art, we find there is no holistic Big Data workflow solution that can provide such a communication module along with high-level workflow definition and scalable execution.

Accordingly we extracted the following requirements for our Big Data Workflow solution:

(a) A workflow definition mechanism with a clear separation between design- and run-time aspects and not limited to a specific technology stack, application domain or ad-hoc processing models;
(b) A workflow run-time support that considers workflows as separate units, rather than as a single unit, for individual workflow steps; and
(c) A workflow enactment approach with event driven execution and support for race-condition-free parallel execution.

5. Proposed solution

In this section, we propose an approach for the workflow step design and inter-step communication. For the description of the Big Data Workflow we use the DSL from [24].

5.1. Overall architecture

The solution enables various stakeholders to be involved in the creation of Big Data workflows. The desired properties of the system are achieved by utilizing container and orchestration technologies and a DSL. The workflow system is composed of three components: *Workflow Modeling Manager*, *Deployment Service Runtime*, and *Data Storage/Sharing Ecosystem* (see Fig. 1).

The Workflow Modeling Manager is the central element for defining workflow steps and composing them in workflows. It comprises a set of tools and configurations that allow the formation of deployable Big Data workflows. The component uses models that provide

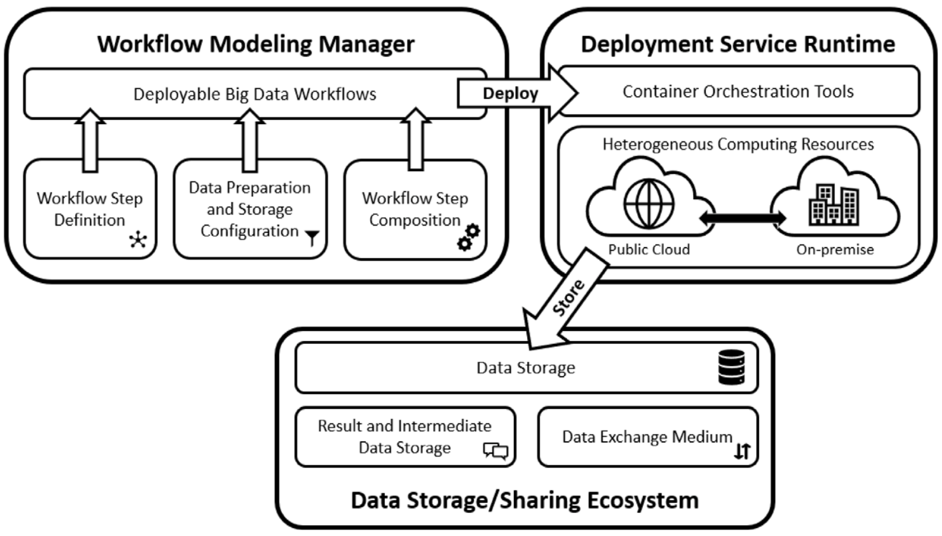

Workflow Modeling Manager
Deployable Big Data Workflows
Workflow Step Definition
Data Preparation and Storage Configuration
Workflow Step Composition
Deploy
Deployment Service Runtime
Container Orchestration Tools
Heterogeneous Computing Resources
Public Cloud
On-premise
Store
Data Storage
Result and Intermediate Data Storage
Data Exchange Medium
Data Storage/Sharing Ecosystem

Fig. 1. Overview of the our workflow system.

high-level descriptions of workflow steps and their dependencies. This component handles storage configurations, data preparation, and step-level data processing and transformation operations. The output of the component is a deployable data workflow. The *Deployment Service Runtime* is a component representing the collection of hybrid computing resources where workflows steps are deployed. As individual workflow steps are wrapped as containers, container orchestration tools play an important role in managing the heterogeneous resources and allowing workflows step containers to be deployed. The component also controls operations such as scaling and load balancing. *Data Storage/Sharing Ecosystem* is responsible for storing intermediate and output data and the data exchange mechanism during workflow execution. As workflow steps are deployed across heterogeneous distributed environments, the data exchange mechanism is an essential element that binds the workflow steps together by allowing them to pass data.

The system ensures the separation of design-time and run-time aspects of the workflows, i.e., workflow definition is done without considering the run-time execution. This allows having a separation of concerns among the involved stakeholders. Thereby, domain-experts can be responsible for extracting data processing requirements and structuring the high-level design of workflow steps. Technical experts provide the concrete programmatical implementations of the steps — for example, data scientists may provide workflow step-specific analytical models and data preparation code. Finally, DataOps experts are engaged in deploying and maintaining data workflows in production settings and monitoring data quality and related infrastructure status.

5.2. Big data workflow description

The DSL used in this work [24] for representing Big Data workflows is inspired by [25] and is shown in Fig. 2. The domain conceptualized by the DSL is container-based Big Data workflows, which is reflected in the choice of concepts. The main concepts of the metamodel are used to represent the different aspects of the *Workflow Modeling Manager* element shown in Fig. 1. *Workflow Step Definition* is implemented by the *Workflow* concept, which is comprised of a set of *Steps* in the DSL. Additionally, the element *Data Preparation and Storage Configuration* is implemented by the concepts *Parameter*, *Trigger* and *Communication Medium*, whereas the *Workflow Step Composition* element relates to the *Step Implementation* concept in the metamodel.

A *Workflow* is a sequence of steps that need to be executed in some order to process a set of data. It represents the conceptualized series of steps that perform different data ingestion, transformation, and analytics tasks. A given workflow can be defined by reusing another workflow. Besides the steps, a workflow is composed of a communication medium and a set of parameters. The *Steps* are the building blocks of a workflow, and each step corresponds to a single unit of data processing work in the workflow. The steps in a workflow are executed independently and are isolated from each other. Further, a step can have various options for its implementation and trigger mechanisms for executions.

A *Parameter* represents an input configuration value needed for the execution of a specific workflow instance. In addition to workflows, parameters can also be used to define configurations for workflow steps. The *Trigger* concept represents how the execution of a step instance is instantiated. In our metamodel, step execution can be triggered from a schedule that runs in a fixed interval of time, or it can be configured to run only once (e.g., during initialization of the workflow). Execution can also be triggered by an external event (e.g., invocation from a REST API or availability of input data in a message queue).

The *Step Implementation* is a concept that represents how the actual implementation of a workflow step is performed. A container-based implementation can be considered as an example of step implementation. In this way, the DSL provides explicit support for the container-based Big Data workflow approach described in the rest of this paper and additionally allows for other implementations that may not necessarily make use of containers. The *communication Medium* represents the mechanism in which workflow steps exchange data. A workflow step can pass data to another step using a message queue. Nevertheless, our approach allows that data exchange be performed using other means such as distributed file systems or Web services.

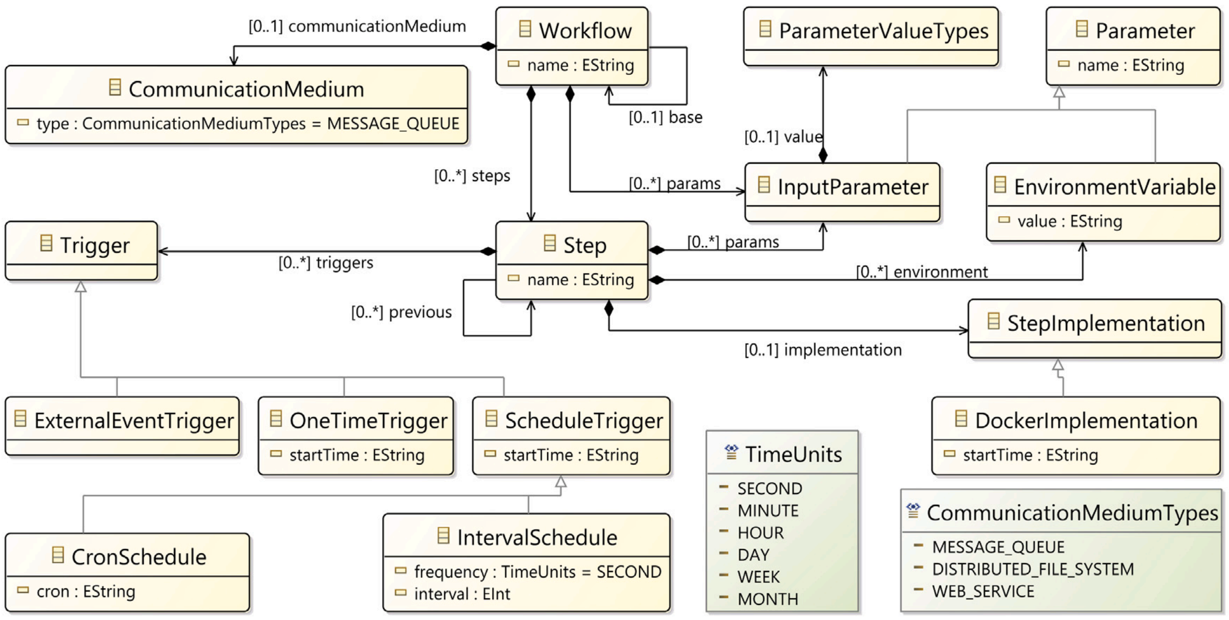


Fig. 2. Metamodel for the proposed DSL for representing Big Data workflows.

The design of the DSL emphasizes separation of concerns between the structural and implementation details of a workflow (e.g., through the clear separation between *Step* usage in workflow and *Step Implementation*). These aspects correspond to the domains of concerns of the two main groups of stakeholders — non-technical personnel (e.g., experts in a vertical domain) and technical experts (e.g., programmers). In that way, the DSL satisfies the requirement discussed in Section 4). Furthermore, we assume that the DSL should be used by non-technical experts and, therefore, should avoid complex constructions as much as possible. Therefore, another approach we apply in the language design is introduction of language concepts only if necessary. This means that only concepts used in our practical cases are supported in this first version of the DSL. Thus, constructs, such as loops and conditional steps have not been introduced into the DSL presented in this paper. The introduction of more advanced control structures is part of ongoing work in the context of the DataCloud project.[12]

5.3. Step design

Our approach takes advantage of container technology [26] in order to implement step encapsulation (in the DSL, this is reflected in the *Docker Implementation* sub-class of the *Step Implementation* concept shown in Fig. 2). The implementation of the approach implies wrapping workflow steps as containers and having step containers run independently of each other and in parallel. Moreover, step templates (container images) are downloaded once, and multiple instances of the same template can be deployed.

To enable the definition and deployment of workflow steps and their composition into a workflow, workflow steps are derived from a generic workflow step template supporting multiple programming languages. A workflow step can be prepared by customizing the generic template according to the need of the step and other settings. To achieve such flexibility, this template needs to be designed so that it is easy to introduce customization. The step template is composed of three main components: *Input Processing*, *Workflow Step Action*, and *Output Processing* (see Fig. 3).

The *Input Processing* is responsible for handling incoming data; this includes fetching data from remote sources (e.g., copying or downloading a file from shared file volumes and moving the data to the step workspace, where it will be processed). Based on the step's configuration, input data can be fetched once at a container startup or scheduled to poll for the data availability at a specific time interval. The Input Processing can be triggered when a piece of data is available at the source. The *Workflow Step Action* is a wrapper component for step-specific data processing code. This component allows the injection of custom code using different programming languages. The data fetched by the *Input Processing* component is processed using the step-specific code in the step. The *Output Processing* component is responsible for delivering the processed data to a specific destination (e.g., upload to a remote source or move it to a shared volume for further processing by next steps) and notifying that the processed data is available for the next steps. Output Processing also includes the clearing up of temporary and input data from the step workspace. Configuration and attributes of a workflow step can be expressed as parameters and injected at the deployment time. The step parameters are accessible only by the corresponding step, but workflow-level parameters can be defined as well.

[12] https://datacloudproject.eu.

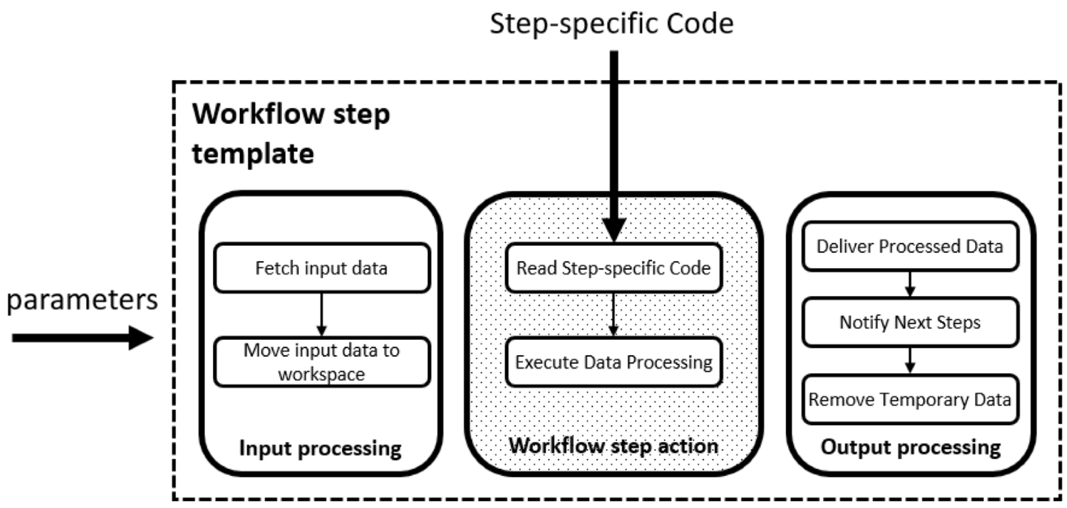


Fig. 3. Components of workflow step template.

5.4. Inter-step communication

The step design approach described in the previous section implements loose coupling of steps that comprise a workflow. However, in order to ensure that data between steps are transmitted consistently and correctly, consecutive steps need to communicate to notify each other of data availability. This communication corresponds to the *Fetch input data* and *Notify Next Steps* sub-processes of the *Input and Output processing* components in Fig. 3. In the proposed workflow approach, MOM serves as a medium for workflow step communication. Two inter-dependent workflow steps communicate by passing data through MOM without direct interaction. The sender step pushes data to the MOM so that it is consumed by the receiver step at any time after the data becomes available in the MOM. The two steps do not need to run simultaneously for interaction, ensuring temporal decoupling. The space decoupling is also achieved since none of the sequential steps needs to know the other nor how many other steps are in the workflow. Since workflow steps are loosely coupled, they can be scaled independently. Therefore, it is possible to assign more instances to bottleneck workflow steps that are, for example, more computationally heavy and reduce the overall processing time.

Specifically, message queues are used as a communication medium so that two inter-dependent workflow steps can share a queue to exchange data asynchronously. Both workflow step processes do not necessarily need to be running simultaneously to interact with the queue. Additionally, the messaging system is capable of providing an exactly-once message delivery guarantee. Furthermore, MOM-based communication ensures that the workflow does not run into race conditions. These can be seen in two scenarios:

- A workflow step that receives data does not access the data while the predecessor step is writing it. Message queues ensure that sending a message to a queue is independent of when it enters the queue and when the receiver reads it; and
- When a workflow step is scaled up, a step instance does not process a piece of data already being processed. This problem is avoided since a message is delivered to only a single step instance, i.e., data access is only for a specific instance.

The data patterns of the designed workflow are characterized according to the major workflow data patterns described in [27]. From a data visibility perspective, data elements are accessible from all workflow steps (e.g., data can be stored in shared file storage). Though a data element is accessible for all, it can only be used by a single workflow step instance at a time (e.g., when a reference of a file is transferred to the step from a message queue). Internally, data interaction happens only between two workflow step instances. Depending on the workflow step's purpose, it can also interact with an external source (e.g., by downloading a file from a remote source or invoking an external API endpoint). Data transfer is undertaken by the reference to the data element in some shared location (e.g., a file can be stored in a shared location, and its reference is exchanged over a message queue). In this case, data locking is not required; message queues restrict concurrent access to the data element. In terms of data-based routing, a given workflow step is executed whenever data are available at a network location.

6. Prototype implementation

To demonstrate the applicability of the proposed solution, we developed a prototype Big Data workflow (see Fig. 4) that implements the design choices described in Section 5 (available on GitHub[13] including a small anatomized sample of the used data). The prototype workflow comes from the domain of digital marketing (see [28] for a detailed description) and was chosen

[13] https://github.com/SINTEF-9012/ebw-prototype.

such that some of its steps have higher compute requirements than others. Those steps need to be assigned with more computing resources than other workflow steps to enable faster data processing. This makes it possible to demonstrate the applicability of step-level scalability. The prototype workflow demonstrates the implementation of Big Data workflows using container technology (encapsulating individual steps of the workflow), a messaging system (Message Queue), shared file system volumes (can be local or distributed file system), and includes the following steps: (i) extract tab-separated values (TSV) files from an archive file stored on a volume in a shared file system, (ii) convert TSV files to comma-separated values (CSV) files, (iii) split CSV files into smaller pieces if the number of rows in the files is above a certain number, (iv) clean and pre-process CSV files, (v) and convert tabular CSV files to JSON collections for further storage. The DSL description of the prototype workflow using an Xtext-based[14] grammar specification over our DSL model is given in Listing 1.

```
workflow prototypeWorkflow {
    communicationMedium: medium MESSAGE_QUEUE
    parameters: MQ_HOST = kubemq
    steps:
        - step unzip
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-00-unzip'
            environment:
                STEP_NAME='00-unzip'

        - step tsv2csv
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-01-tsv2csv'
            environment:
                STEP_NAME='01-tsv2csv'

        - step split
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-02-split'
            environment:
                STEP_NAME='02-split'

        - step transform
            triggers: external-event
            implementation:
                docker-implementation image: 'ebw-prototype-03-transform'
            parameters: tranformationJar = '/transformation/transformation.jar'
            environment:
                STEP_NAME='03-transform'

        - step toarango
            triggers: external-event
            implementation:
                docker-implementation image: '/ebw-prototype-04-toarango'
            parameters: tranformationJson = '/transformation/transformation.json'
            environment:
                STEP_NAME='04-toarango' }
```

Listing 1: The DSL description of the prototype workflow.

6.1. Inter-step communication

Asynchronous FIFO message queues are used to implement inter-step communication and the KubeMQ[15] messaging system was chosen for this purpose. KubeMQ provides multiple message queues with a guarantee of exactly-once message delivery. A communication link is established when a workflow step is configured with a message queue as its output channel, and the same queue is used as an input channel for another step. In this way, the latter step waits for the output of the former and its execution is triggered immediately when the shared queues have content. This communication mechanism enables the two step instances to run concurrently during execution while maintaining race-condition-free data access. Such workflow execution follows the P&F architecture, i.e., step instances as filters and message queues as pipes. In the example prototype, a containerized instance of KubeMQ is used to handle the communication between the steps and is made accessible to all workflow steps. Each consecutive workflow step is configured to share a message queue (except the first and last steps). Thereby, steps in the workflow use two message queues: one for retrieving information about available input data from the previous step and one for signaling that data have been made available for the next step. There is no direct link between two consecutive steps; instead, the message queue they share creates a logical connection.

[14] https://www.eclipse.org/Xtext/.

[15] https://kubemq.io.

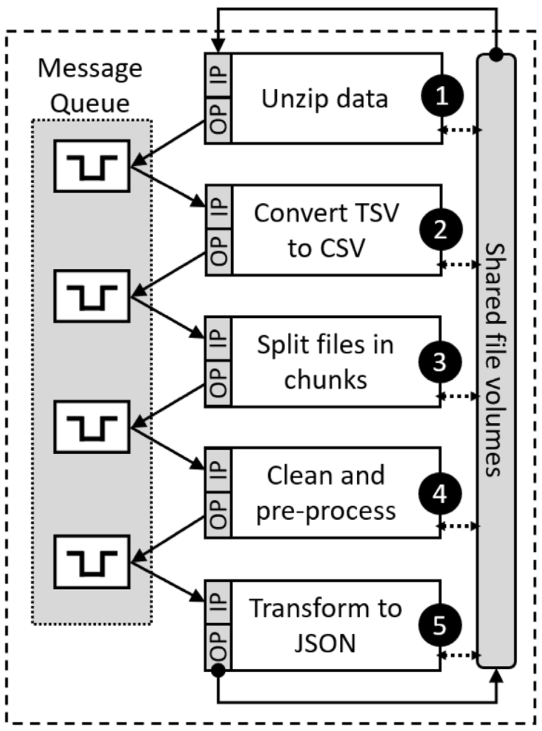


Fig. 4. Prototype Big Data workflow.

Even though the message queues in the prototype workflow serve as a communication mechanism, the actual data for processing are not stored in the message queues but are stored on shared volumes. Only references to the data are placed in the message queues. To organize the processing of the data, a step container allocates at least four different file storage volumes: *in*, *work*, *out*, and *sandbox*. When a file reference becomes available in the input message queue of a step container, it reads the file reference from the queue and accesses the file from its *in* volume. Then, the file is moved to the *work* volume for processing. When file processing is completed, the result is stored in the *out* volume, and the reference of the output file is published in the output message queue so that the next step can use it for further processing. The *sandbox* volume is used to store any files that are not processed successfully.

6.2. Container orchestration

The individual steps in a workflow are wrapped as Docker containers. The Docker images of the workflow steps are derived from a generic step template that implements the input processing, output processing, and communication logic. The template is modified by adding the installation script for any necessary software libraries in the Docker image building configuration and injecting relevant code scripts (step processing code) in a specific place in the template logic. Container orchestration systems provide effective means to deploy distributed applications across a heterogeneous cluster of resources. To use these tools, it is necessary to prepare a deployment configuration file either in JSON or YAML, which describes the location of the container images, network setups, and other configurations. In the case of Big Data workflows, the configuration needs to include the description of different workflow steps and the communication medium. Scalability and other constraints can also be stated in the configuration. When deploying a workflow, the orchestration tool will automatically schedule the deployment of each workflow step to a cluster and pick the right host, taking into account any stated requirements or constraints.

The prototype workflow deployment was done by composing individual workflow step Docker containers using a Docker-compose file. A workflow step description in the file includes shared volumes, environment values, and message queue assignments. The communication medium, KubeMQ, is also included as a separate service in addition to the workflow steps. Rancher[16] is used as an orchestration tool to deploy the workflow on a private cloud environment running Docker containers. In Rancher, the number of instances for each step can be defined in a separate YAML file. When this YAML file, together with the Docker compose file, is supplied, Rancher handles the deployment by taking the scalability requirement into account and finding the right host for each container.

[16] https://rancher.com.

Table 2
Summary of feature-based comparison.

Workflow tool	Step-level containerization	Communication mechanism	Parallelism	DSL for workflow	Separation of concerns
Airflow	Yes	Yes	No	Yes	Yes
Argo	Yes	No	Yes	No	Yes
Conductor	No	Yes	Yes	Yes	No
Nextflow	Yes	Yes	Yes	Yes	No
NiFi	No	Yes	No	Yes	No
Node-RED	No	No	No	No	Yes
Pachyderm	Yes	No	No	No	Yes
SnakeMake	Yes	No	Yes	No	Yes
Our approach	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*

6.3. Fault tolerance

To address faults in Big Data workflows, it is necessary to keep track of each file's status while it is being processed. Having this information stored in a structured way helps for monitoring and debugging purposes. To this end, a centralized logging functionality writing structured logs in JSON format is incorporated. These logs are stored in scalable message queues, such as Kafka [29], and contain important execution statuses, warnings, and errors. A centralized logging system can be a bottleneck when the system grows. This is because the logging unit gets overwhelmed by the flow of log data coming from scaled up workflow step instances. This slows down the overall performance of the workflow. However, our implementation is not significantly affected by this problem as the centralized logging approach is inspired by the state management system in well-known scalable platforms, such as Flink [30] and MillWheel [31]. In addition, the bottleneck on the logging mechanism can also be alleviated by using decentralized queuing systems. Errors in the common and step-specific scripts are handled differently by using a separate exception handler. This is done by checking exit codes of program calls and operating system control flow constructs. Using such constructs, important program calls in the scripts are bound together with an error handler function. The error handler will be invoked when the program exits with an error code and logs error information, including the file causing the error, the line number that caused the error, and the input file's name being processed.

To make the workflow more resilient to intermittent failures, we implemented retrying processing of failed input files. Whenever an error occurs, the input file is moved to the respective *sandbox* volume to be processed later or to be archived. Since the message queue implements exactly-once delivery, the reference of the file will not be available in the queue for retrials. In this case, all the failed files can be accessed directly from the sandbox without the need to access the message queue. When a step becomes free, i.e., when there are no more files in the message queue (or input directory for the first step), it tries to re-process files from the sandbox. With the current implementation, retrial is done only once.

7. Evaluation

In this section, we present the evaluation of the proposed solution. It includes a feature-based comparison of the workflow solution against existing workflow tools and performance experiments based on the prototype implementation.

7.1. Feature-based comparison

To compare the proposed workflow solution with other similar tools, we selected five main implementation features: (i) the ability to wrap workflow steps as containers (isolated units); (ii) communication mechanism between steps and their activation trigger; (iii) parallel execution of workflow steps; (iv) inclusion of DSL for workflow definition; and (v) separation of concerns. (see Table 2).

Granular containerization (i.e., step-level containerization) is not provided out of the box for Big Data workflows containing multiple steps. Hence, step-level scalability cannot be achieved when faster processing is needed for an individual workflow step. Argo Workflows, Nextflow, Pachyderm, and SnakeMake are instances of the few Big Data workflow tools supporting step-level containerization. Nextflow also has an additional feature that allows deploying the entire workflow as a single container. The majority of existing data workflow tools lack such a separate communication link and instead, steps in these tools are tightly coupled. Thereby, the logic that determines the flow of step execution is embedded as part of the step implementation. Even though Argo Workflows provides both sequential and parallel workflow step definitions, this type of communication mechanism is missing.

Apache NiFi and Nextflow use a queuing system for inter-step communication. Nextflow provides advanced data binding and publish/subscribe features. Conductor has a special type of workflow step (i.e., event step) to enable event-based dependencies for steps by publishing events internally or an external message queuing system like Amazon SQS. Apache Airflow uses a feature called XCom to communicate small messages between steps and larger data are exchanged using remote storage such as S3 and HDFS.

Workflow tools such as Apache Airflow, Conductor, Nextflow, and Apache NiFi provide parallel execution of step instances from different workflow steps. In Apache Airflow and Conductor, special operators are used to define a parallel set of workflow steps. Parallelism support in Argo Workflows is limited to all workflows in the system (i.e., it is not granular to a class of workflows, or steps within them).

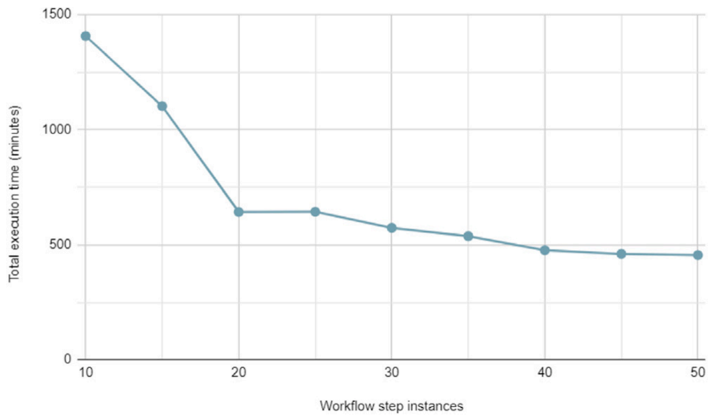


Fig. 5. Results of the horizontal scalability test.

Defining workflows in existing data workflow tools requires knowledge of general-purpose programming and scripting languages such as Java, Python, Scala, or R. Consequently, domain-experts face a significant learning curve to master these languages. The need for DSLs is indispensable in this regard. Only a few Big Data workflow tools, e.g., Argo Workflows, Conductor, Nextflow, and SnakeMake, support a custom-made DSL for workflow definitions.

Separation of concerns is not a focus in tools like Conductor, Nextflow and Apache NiFi. Hence, there is no mechanism for the separation of high-level workflow definition concerns from step-specific implementation and deployment details. Therefore, such tools do not ensure the separation of design- and run-time aspects of workflows.

7.2. Performance evaluation

We evaluated our workflow approach's horizontal scalability using the prototype workflow and compared it with the Argo Workflows. We used eight heterogeneous physical hosts configured to form our distributed testbed. Three of them have 12-core Intel CPUs and 64 GB RAM each; the other five — four-core AMD CPUs and 16 GB RAM each. The hosts are connected in a Gigabit Ethernet network, share a distributed file system, and run the Docker engine connected to Rancher.

7.2.1. Scalability evaluation

Our workflow solution allows us to scale individual workflow steps. Hence, it is worth investigating the impact of horizontal scalability on the performance of the workflow. Therefore, we designed an experiment to determine how the number of workflow step instances (i.e., containers) affects workflow execution performance. The prototype workflow, shown in Listing 1, was used for this experiment. We defined an increasing number of workflow step instances and measured the time it takes to complete processing input files. Input data size is kept constant over all iterations. The results of this experiment are shown in Fig. 5.

We performed the scalability experiment by processing approximately 100 GB of compressed TSV files in nine rounds. These are historical data from the use case described in [28] that comprises a large volume of extracts from the Google Ads platform[17] and made available for this experiment by a digital marketing company. For each round, we increased the number of workflow step instances by five (starting from 10), and the increased number was distributed among the steps based on the previous step execution time. Hence, the step that takes the longest time was allocated a higher number of instances. We stopped adding more step instances after nine rounds since we noticed the minimal effect in the last two rounds. Fig. 5 shows that with four times increase in the number of instances, the total execution time decreases from 1406 to 455 min, i.e., approximately a three-times decrease. There is a significant reduction of execution time up to 20 containers. However, from 25 containers upwards, the reduction in execution time gets smaller. This reduction in performance gains is due to resource bottlenecks with the addition of more instances. In our experimental setup, resources available to the containers are shared. They are split among the containers by availability time, size or processing power and increasing the number of instances beyond 50 does not improve the performance further. Even though such an arrangement helps to utilize the resources effectively, it can cause resource contention among the containers. As the number of instances increases, it leads them to compete for the resources. Further increase of instances can result in degraded performance due to CPU, memory, and I/O bottlenecks. Measuring the usage of such resources (CPU, memory, I/O, Network Receive/Transmit Throughput, etc.) would be interesting but is considered out of scope for this evaluation as the container orchestration system does not allow to easily change or customize them.

[17] https://ads.google.com/home.

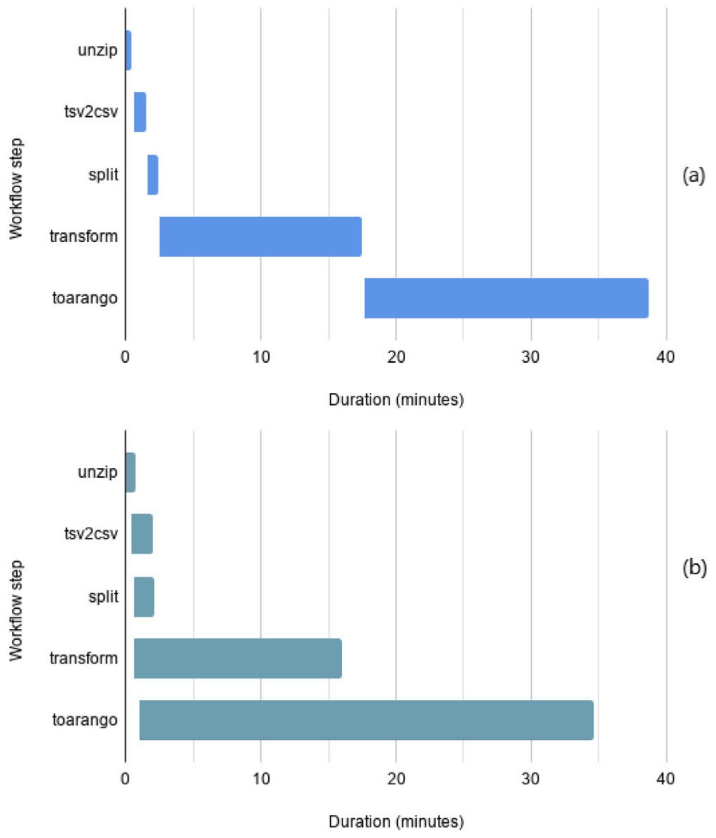


Fig. 6. Workflow execution times (a) using Argo Workflows and (b) using the proposed approach.

7.2.2. Comparison with Argo Workflows

The implemented workflow approach allows individual workflow steps to run independently. In this experiment, we investigated the effect of concurrent workflow execution on the workflow's performance by comparing our approach with another workflow tool that does not have built-in concurrency support. For the comparison, we chose Argo Workflows since it does not have built-in concurrency support. The composition of a workflow can be done quickly using its custom DSL that is similar to traditional YAML. Together with the step template invocator, the container template was used to compose the sequential workflow with a single instance assigned for each step. The experiment had two parts: one using fixed input data size, and the other is by using increasing input data size.

Experiment with fixed input data size. In this experiment, we compose workflows in both approaches and measure the execution time of each step with a fixed volume of input data. We measure each the start and end time of each step to observe the performance difference between concurrent and sequential execution modes. To achieve this, we used the workflow in Listing 1 and followed the same workflow composition as in the scalability evaluation from Section 7.2.1, but with a single container instance assigned to each step. A similar workflow structure was composed in Argo Workflows using the same step container images. Both workflows were executed using 100 MB of input data on a Kubernetes cluster (Argo Workflows runs only on Kubernetes). We measured the execution time for each step. To minimize systematic errors, we repeated each experiment 20 times and took the arithmetic mean of execution time to compare the two modes. The repetitions were done in a cool start fashion without caching result or intermediate data from previous rounds.

The sequential execution mode in Argo Workflows takes 38.7 min to complete, whereas the concurrent mode takes 34.6 min, which shows a slight performance gain when using our approach. Fig. 6(a), shows the execution time for each step in Argo Workflows. Since the workflow is in sequential execution mode, step container execution starts after the previous completes processing all the input files. In this mode, step containers are not initialized at the time of workflow submission, and initialization occurs right before step execution, which results in a short gap in between consecutive steps. The concurrent workflow execution based on our workflow implementation is shown in Fig. 6(b). We observe a slight performance gain due to the concurrent execution of steps. Concurrency is enabled by the P&F architectural design which allows workflow step container to continue processing the next input file right after passing the processed file to the next step. Moreover, concurrent step containers share the host machine's resources, which explains the longer execution time of the last step (compared to sequential mode). Additionally, we do not observe positive effects of the concurrency in the first three steps since their execution times are relatively short (they are completed in a couple of minutes).

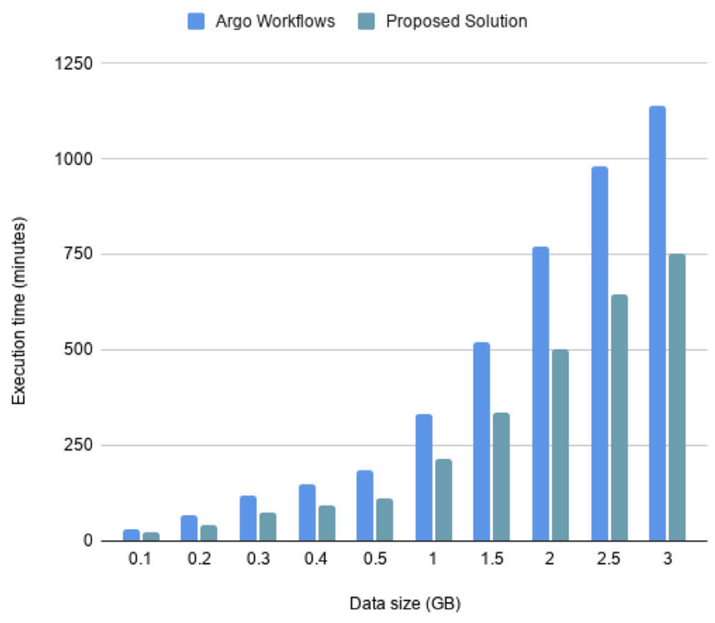


Fig. 7. Performance with increasing input size.

Experiment with increasing input data size. The previous experiment is repeated to evaluate which of the two workflow approaches performs better with increasing input data of compressed CSV files. To achieve this, we used the workflow from Listing 1 once again and measured execution time for both approaches using input data starting from 0.1 up to 3 GB. Fig. 7 shows the results of the experiment. It is evident from these results that our workflow approach performs better than Argo Workflows in all cases, with an approximate performance gain of 36%. However, we did not observe a significant performance gain when input data are further increased. This is because the last two steps are computationally intensive and they determine the overall speed of the workflow. Having a single container instance for each step, the addition of more input data does not bring any significant performance gain as the execution rate of steps is (relatively) constant independent of the size of the input data.

7.3. DSL evaluation

We evaluated the proposed DSL qualitatively in what follows from a development and quality perspectives from multiple dimensions.

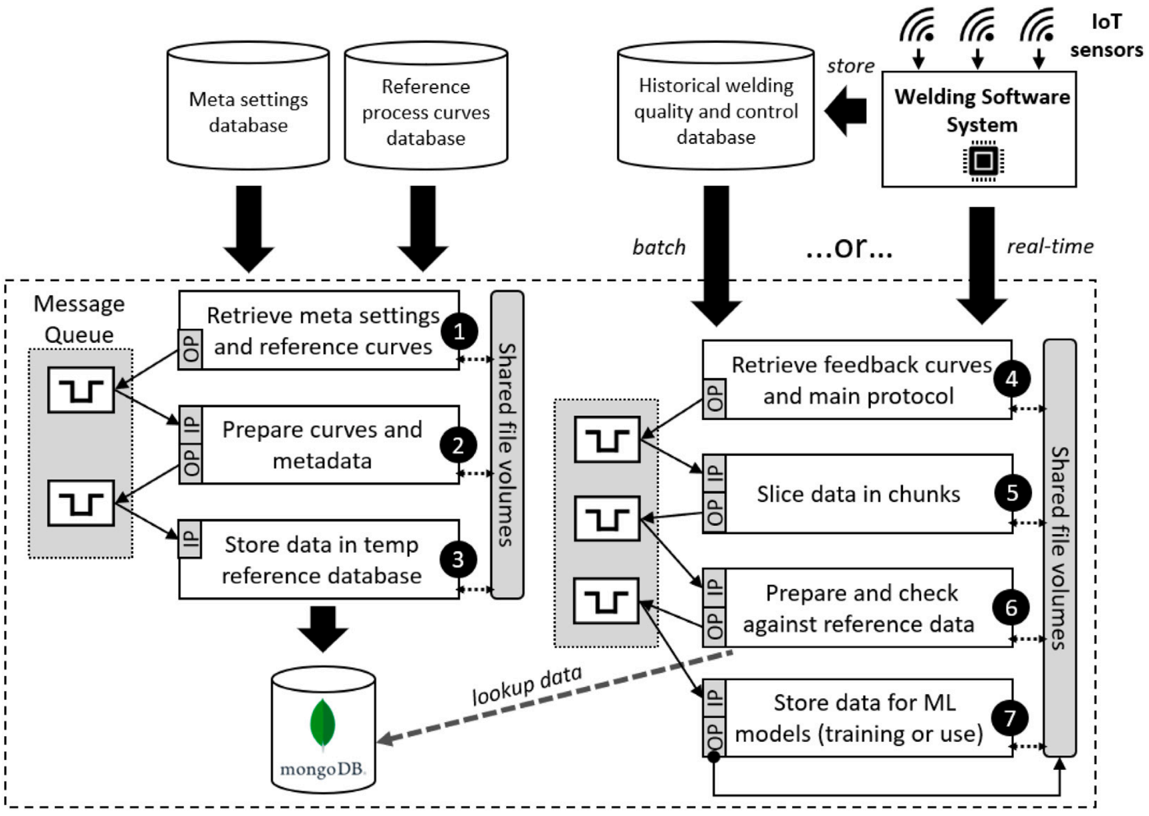
7.3.1. Development perspective

One of the benefits of using DSL over general-purpose programming language is to reduce code complexity and increase development efficiency. However, it is challenging to measure the level of code abstraction introduced by using DSL. A simple quantitative evaluation metric can be used by measuring and comparing the amount of code required to define a workflow with and without DSL usage [32]. The measurement can be expressed in lines of code (LOC). Another metric is to compare the number of concepts or technologies involved with and without the DSL. This metric shows the number of concepts abstracted by the DSL.

For example, to implement the example workflow presented without using the DSL, it is necessary to follow some specific steps; however, the important work in this process is to compose all the workflow steps into a single deployment file. Then, assuming we have all the step images ready, it is required to set up the MOM and shared volumes. Also, input/output parameters, message queues, and environment variables must be adequately assigned for each workflow step. By using the DSL, composing a workflow requires fewer configurations since the DSL abstracts some concepts. For example, it is not required to set up MOM and assign message queues for workflow steps. Intermediate storage volumes and settings are not specified either. In other words, workflow definition using the DSL does not necessarily require knowing technologies like shared volume management, MOM, and message queues. To provide a quantitative comparison, we examined the LOC required to define workflow shown in Fig. 4 with and without the DSL. Without using the DSL, the Docker compose file has 133 LOC.[18] Whereas the same workflow can only be expressed using 36 LOC by the DSL on a higher level of abstraction — see Listing 1. The DSL provides the capability to define the example workflow in a short LOC because of the abstraction of key concepts and technologies needed to define a workflow.

To test the functional ability of our DSL in the context of an IoT scenario, we mapped a third-party data preparation workflow used at Bosch as described in [33]. The mapping of the workflow to our approach is shown in Fig. 8. The IoT scenario represents

[18] https://github.com/SINTEF-9012/ebw-prototype/blob/master/docker-compose.yml.



Meta settings database
Reference process curves database
Historical welding quality and control database
store
IoT sensors
Welding Software System
batch
...or...
real-time
Message Queue
Retrieve meta settings and reference curves 1
Prepare curves and metadata 2
Store data in temp reference database 3
Shared file volumes
Retrieve feedback curves and main protocol 4
Slice data in chunks 5
Prepare and check against reference data 6
Store data for ML models (training or use) 7
OP
IP
lookup data
mongoDB

Fig. 8. Mapping of approach to an IoT Big Data workflow about welding data preparation for machine learning.

a use case at Bosch[19] of quality monitoring for resistance spot welding. Data about the conducting of process are continuously generated at high velocity by sensors attached to the welding equipment and are managed by a welding software system. These data are stored in a database for and used by the software to signal for the current process status and potential issues. During the welding process, signals need to be generated at near-real time as any necessary response by the system or operators at the factory needs to be quickly addressed to avoid delays on the production lines. In addition to the sensor data from the welding machines, the use case involves reference data that provide information about target parameters of the equipment as well as the settings of the individual machines and welding programs. The overall goal of the work is to create a workflow for offline *batch* preparation of data for machine learning as well as online scenario for *real-time* monitoring using trained ML models.

The workflow consists of two parallel dependent sub-workflows that are involved in processing the different parts of the data. We use a gray shade to represent components specific to our approach that are necessary for the encapsulation of the workflow steps. Data transmission in this workflow and the rest of the examples in this section is done through shared file volumes of a (possibly distributed) file system, whereas the coordination of the steps (i.e., informing the next step of available data for processing) is performed using a Message Queue that is dedicated for each pair of steps. The output of each step is passed by the *Output Processing (OP)* script to the *Input Processing (IP)* script of the next step, both of which are part of the wrapper of the respective images that wrap the step implementations. The left side of Fig. 8 represents the sub-workflow for processing the reference datasets. In the first step, data are retrieved from the meta settings and reference process curves databases and sent for further processing. The curves and metadata are integrated according to a common data model and are also re-formatted and prepared to be referenced by the larger scale sub-workflow. After the preparation data are stored in a reference database (implemented using MongoDB[20]) where they can be accessed for lookups. The reference datasets are continuously but infrequently updated as new machine settings or reference data become available. The large-scale IoT data sub-workflow is displayed on the right side of Fig. 8. This sub-workflow processes either very large volumes of historical welding quality and control data from a database (in the offline setting), or direct signals from the welding software system in real-time (in the online setting). The data retrieval step (numbered as step 4) fetches the data from the respective source and sends it to a dedicated slicing step. In the offline scenario, this step is used to chunk the data in a specific way, whereby the data coming from the software system is packaged together with the exact sensor data that relates to it in chunks of a pre-determined size (this correspondence is needed during the preparation step). The chunk size is calculated so that when the workflow is scaled, there are enough hardware resources (esp. RAM and CPU) to process them are available. The chunks

19 https://www.bosch.com.
20 https://www.mongodb.com.

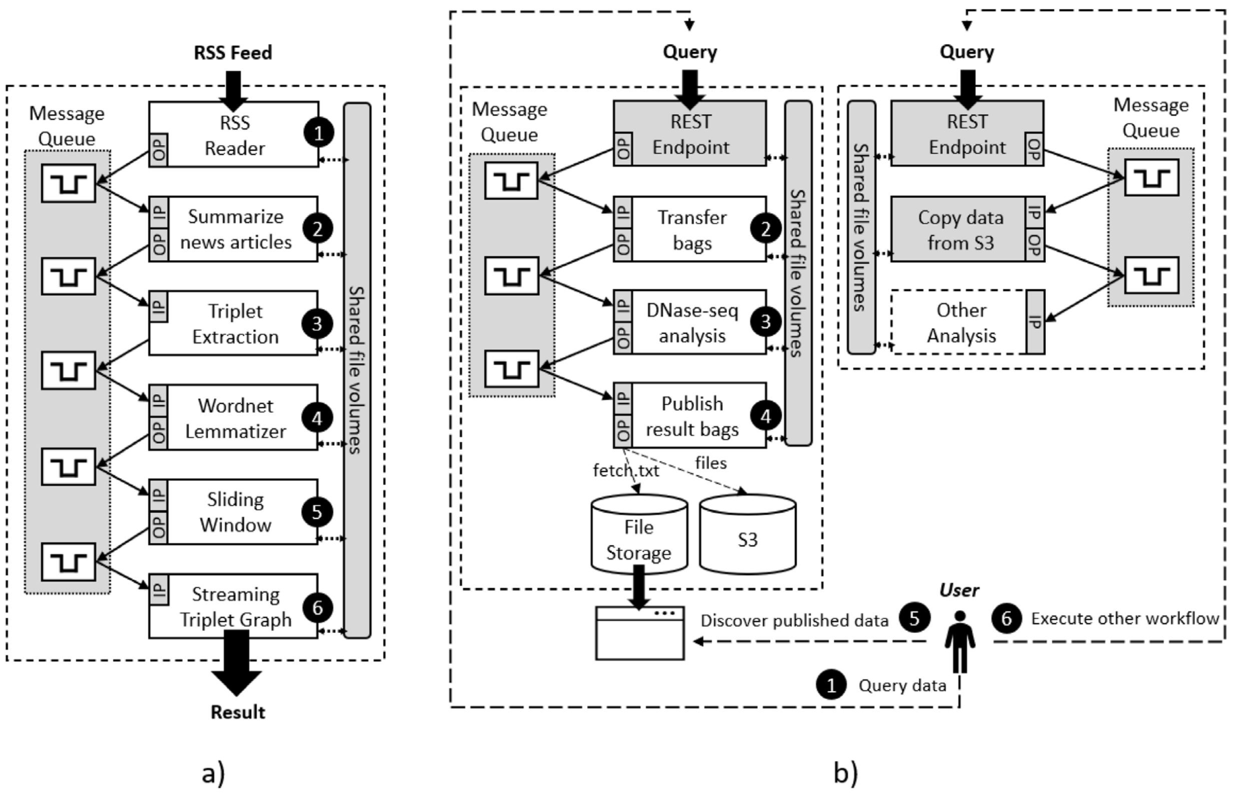


Fig. 9. Mapping of approach to existing Big Data workflows — (a) CloudFlows data mining and (b) ENCODE Data access workflows.

are then sent for preparation during which the data are integrated, reformatted and packaged together with the specific relevant reference data through a lookup from a reference database. The result is then stored on disk for further training of or use by the machine learning models for welding quality prediction.

Another case study we tried to express in terms of our DSL is of the workflow *CloudFlows, A data mining workflow platform* [34]. Fig. 9a shows a semantic triplet graph workflow built on the ClowdFlows framework from an RSS feed workflow.

The first step of the workflow implements an RSS Reader that takes an RSS feed URL as input. The second step summarizes news articles based on the generated read data from the first step. The text from the news feed is then passed to the next step of the workflow that performs triplet extraction. This step implements NLP techniques to extract *subject–verb–object* triples from the news articles. Step four of the workflow uses a WordNet[21] Lemmatizer on the resulting triples, and step five performs a sliding window that takes the number of triplets as input. Finally, the data window is passed to the Streaming triplet graph, which is the last step of this workflow.

Another case study is of *ENCODE Data Access* [35] shown in Fig. 9b. In the first step of the workflow, the user enters a REST-format ENCODE query or uploads an ENCODE metadata file that represents a dataset array corresponding to the so-called *bags* of data. The selected data are then transferred for analysis to the DNase-seq sub-workflow that implements a set of analyses on the data. After the analysis is performed, the result (also serialized into bags) is published on Amazon S3, and a metadata file (named "fetch.txt") is created and stored for serving to the users of the result. A user may then discover these new bags and perform further analysis using the same type of querying as in step 1, but with another analysis set.

7.3.2. Quality characteristics

We used the Framework for Qualitative Assessment of DSLs (FQAD) [36] that defines sets of quality characteristics to evaluate DSLs. The quality characteristics are mainly determined from ISO/IEC 25010:2011 standard but tailored for DSLs. FQAD is helpful for evaluating the requisite quality characteristics at the outset of DSL development, as well as assessing the end product of the DSL development process (i.e., the language). It defines a set of assessment goals and DSL characteristics to fulfill them, which are derived from the ISO/IEC 25010:2011 international systems and software standards model.

Several works in literature have used this framework to evaluate their DSL. For example, Florian et al. [37] used this framework to evaluate a DSL in the business domain for creating test case specification. They did not evaluate the test language itself but only the output and, therefore, concentrated only on the expressiveness, usability and productivity characteristics using the sub-characteristics of completeness, correctness, comprehensibility, reading flow, and reproducibility. Aleksandar et al. [38] employed

[21] https://wordnet.princeton.edu.

the full FQAD framework to perform a quality assessment of their DSL for modeling application-specific functionalities of business applications. Sadiq and Geylani developed DSML4DT - a domain-specific modeling language for device tree software [39]. For qualitative assessment of the language, they prepared a comprehensive questionnaire for the users. To prepare the scoring part of the questionnaire, they employed FQAD and customized with respect to DSML4DT specifications.

Below, we outline the evaluated FQAD characteristics for our DSL and the respective assessments.

- *Functional suitability*: This refers to the degree to which a DSL is completely developed. Functional suitability further has two sub-characteristics:
  - Completeness: The ability of a DSL to express all concepts and scenarios of the domain is termed as completeness. We successfully implemented four Big Data and IoT workflows from different domains with our DSL. The language was expressive enough to describe the workflows efficiently.
  - Appropriateness: The scale to which the DSL is appropriate for the particular applications of the domain is called appropriateness. While implementing the Encode workflow, we faced the challenge of incorporating all elements in one workflow. However, by breaking it down into two separate workflows based on the functionality and with additional REST services, we achieved the required functionality with our DSL.

  The functional suitability analysis shows that all the important functionality is included in the DSL. In other words, the DSL should not contain functionalities that are not part of the domain. In this regard, based on the workflows that were mapped to our DSL, we claim that the DSL implemented in this work covers the core functionality required to define a given data workflow.
- *Usability*: This aspect refers to the degree to which a group of users can use a DSL to achieve specific goals. A DSL must be as simple as possible to express the domain concepts and support its users. As presented, our DSL provides the option to define concise workflows with minimal technical knowledge.
- *Reliability*: It defines the characteristics of the language that help to produce reliable code. This includes functionalities to prevent errors and support for model checking. The DSL was implemented using the Eclipse environment that supports languages designed with precise semantics based on well-defined principles. Additionally, the Eclipse IDE also has essential features to debug and handle code errors. The reliability characteristics has two sub-characteristics:
  - Model Checking: This concerns whether the DSL reduces user error rates. In comparison to writing YAML files, our DSL-based approach has lower error rate. This is, on the one hand due to the conceptual schema that is defined using the Ecore metamodel[22] and Xtext grammar on top of it. When a user is defining a model of our DSL, the Xtext editor uses the Ecore definitions to check for erroneous values or concept instances and highlights incorrectly defined values.
  - Correctness: The term correctness concerns whether appropriate elements, as well as the correct relation between them, are provided, i.e., any unexpected interactions are prevented. Ecore alongside the Xtext framework allow for checking of the validity of each DSL model class instance and attribute value types thus ensuring the correct elements and relations are chosen by the users.
- *Maintainability*: This characteristic refers to the degree to which it is easy to maintain a DSL. A DSL needs to be easy to modify or introduce new concepts. Modularity falls under this characteristic as well. The DSL was designed based on the separation of concerns principle. This makes it easily understandable and maintainable.
  - Modifiability: This characteristic refers to the DSL's ability to incorporate new functionality by as little modification as possible. The modular approach of our DSL makes it easier to add new sub-vocabularies (which is also part of our future work) that can be used to describe Big data workflows with more details.
  - Low coupling: This means how discrete the elements of the DSL are. At this stage, each element of the DSL plays an integral part in describing a workflow. Therefore, the change in each element will have an impact on other elements.
- *Productivity*: This is mainly related to the number of resources required by a user to achieve specific goals. Productivity can be improved using our DSL because of two main reasons. First, the DSL incorporates high-level concepts of Big Data workflows. Hence, designing a workflow using the DSL is simplified and can be done quickly. Second, the language provides automatic generation of template workflow code and configurations. This saves a lot of time compared to when the process is done manually from the ground up.
- *Extensibility*: This characteristic expresses the degree to which a language has mechanisms for users to add new features. The language we have presented currently has a low degree of extensibility since it does not provide users a means (e.g., separate packages in the Ecore model) to extend the language with new functionalities.
- *Compatibility*: This characteristic measures degrees to which a DSL is compatible with the domain and development process. Our DSL was developed iteratively and modified to cover concepts relevant in the domain.
- *Expressiveness*: Expressiveness is defined as the degree to which a problem-solving strategy can naturally be mapped into a DSL program. The DSL was developed after a thorough domain analysis in which each concept has mapped to its corresponding metamodels to represent only core elements and does not cover complex structures such as cyclic workflows. The sub-characteristics of the expressiveness are as follows:

[22] http://www.eclipse.org/modeling/emft/search/concepts/subtopic.html.

  - Mind to program mapping: Concepts are designed appropriately and named so that they are intuitive in order to accommodate problem-solving tasks for the domain.
  - Uniqueness: Because of its simplicity, our DSL provides one and only one good way to express every concept of interest.
  - Orthogonality: Each construct in our Big Data workflow DSL is used to represent exactly one distinct concept in the domain.
  - Correspondence to important domain concepts: DSL constructs correspond to important domain concepts and the language does not include trivial domain concepts. Our DSL only includes highly relevant and non-trivial Big Data workflow concepts.
  - Conflicting elements: This refers to the absence of conflicts between the DSL elements. Our DSL concepts have no conflicts as each element serves a unique purpose for data workflows specification.
  - Right abstraction level: This refers to whether the DSL is at the correct abstraction level that it is not more complex or detailed than necessary. Our DSL provides only the most important elements of the domain at the moment. A few more elements are planned to be added to enhance the functionality — for example, concepts for aspects of data transmission and resource requirements.

- *Reusability*: This characteristic refers to how a language construct can be used in more than one language. The grammar specification of our DSL can be imported and reused in the Eclipse environment. Furthermore, the language is conceptualized so that some elements can be reused in the workflow definition. For example, a workflow step from one workflow can be copied and used as part of another workflow.
- *Integrability*: This characteristic measures how a DSL can be integrated with other languages and modeling tools. The DSL that we have provided can be exported as a plug-in within the Eclipse environment. Integrability of a DSL is largely dependent on the technical space in which it was defined. As shown in [40], the modelware technical space, which is the one used for defining our DSL, provides a high level of integrability both within the technical space and across technical spaces. Furthermore, the Eclipse modeling environment provides integrations with other languages and frameworks such as Javascript,[23] which increases the integrability of our DSL. Finally, as demonstrated in [40], the EMF framework provides automated migration of models through the Edapt framework.[24] Therefore, we conclude that the DSL has a high degree of integrability.

8. Conclusions

In this article, we described a Big Data workflow approach that allows specification of Big Data workflows at a high level of abstraction and enables separation of design- and run-time aspects while maintaining scalable workflow execution. Scalable workflow execution entails parallel data processing that requires workflow fragments to run separately on different computing resources. In the future, we plan to implement comprehensive provenance support. Another limitation to be addressed is the use of a centralized message-oriented communication, which could become a bottleneck for large-scale execution and is a single point failure. Decentralized communication media, Web services, or distributed file systems can potentially be used to address this issue. The DSL that is used in this work is limited to expressing core workflow structures. Thus, the experiments on scalable workflow execution can be performed using an extended DSL, e.g., including infrastructure requirements or with different workflow (sub-) structures. Finally, a visual language for workflow definition, composition, and run-time monitoring would be essential to support domain-experts' participation in workflow design.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partly funded by the EC H2020 projects *"DataCloud: Enabling The Big Data Pipeline Lifecycle on the Computing Continuum"* (Grant nr. 101016835) and *"COGNITWIN: Cognitive plants through proactive self-learning hybrid digital twins"* (Grant nr. 870130), and the NFR project *"BigDataMine"* (Grant nr. 309691).

References

[1] R. Qasha, et al., Dynamic deployment of scientific workflows in the cloud using container virtualization, in: Proc. of the CloudCom 2016, 2016, pp. 269–276.
[2] M. Barika, et al., Orchestrating big data analysis workflows in the cloud: Research challenges, survey, and future directions, ACM Comput. Surv. 52 (5) (2019).
[3] R. Ranjan, et al., Orchestrating big data analysis workflows, IEEE Cloud Comput. 4 (3) (2017) 20–28.
[4] R. Buyya, et al., A manifesto for future generation cloud computing: Research directions for the next decade, ACM Comput. Surv. 51 (5) (2018) 1–38.

[23] https://wiki.eclipse.org/JS4EMF.
[24] https://www.eclipse.org/edapt/.

[5] Y.D. Dessalk, et al., Scalable execution of big data workflows using software containers, in: Proc. of the MEDES 2020, 2020, pp. 76–83.
[6] A. Kashlev, et al., Big data workflows: A reference architecture and the DATAVIEW system, Serv. Trans. Big Data 4 (1) (2017).
[7] W. Gerlach, et al., Skyport - Container-based execution environment management for multi-cloud scientific workflows, in: Proc. of the DataCloud 2014, 2014, pp. 25–32.
[8] N. Russell, et al., Workflow data patterns: Identification, representation and tool support, in: Proc. of the ER 2005, 2005, pp. 353–368.
[9] C. Wulf, et al., Increasing the throughput of pipe-and-filter architectures by integrating the task farm parallelization pattern, in: Proc. of the CBSE 2016, 2016, pp. 13–22.
[10] E. Curry, Message-oriented middleware, in: Middleware for Communications, John Wiley & Sons, Ltd, 2005, pp. 1–28.
[11] N. Naik, Docker container-based big data processing system in multiple clouds for everyone, in: Proc. of the ISSE 2017, 2017, pp. 1–7.
[12] S. Junsawang, Y. Limpiyakorn, A domain specific language for scripting ETL process, in: Proc. of the WCSE 2017, 2017, pp. 239–243.
[13] M. Mernik, et al., When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[14] M. Adhikari, et al., A survey on scheduling strategies for workflows in cloud environment and emerging trends, ACM Comput. Surv. 52 (4) (2019) 1–36.
[15] D.C. Arvind, et al., The Tagged Token Dataflow Architecture, Technical Report, MIT Laboratory for Computer Science, 1984.
[16] A. Alaasam, et al., Scientific micro-workflows: Where event-driven approach meets workflows to support digital twins, in: Proc. of the RuSCDays 2018, 2018, pp. 489–495.
[17] R. Filgueira, et al., Asterism: Pegasus and dispel4py hybrid workflows for data-intensive science, in: Proc. of the DataCloud 2016, 2016, pp. 1–8.
[18] R. Filguiera, et al., dispel4py: A Python framework for data-intensive scientific computing, Int. J. High Perform. Comput. Appl. 31 (4) (2017) 316–334.
[19] L. Gerhardt, et al., Shifter: Containers for HPC, J. Phys. Conf. Ser. 898 (8) (2017) 082021.
[20] M. Belkin, et al., Container solutions for HPC systems: a case study of using Shifter on Blue Waters, in: Proc. of the PEARC 2018, 2018, pp. 1–8.
[21] L. Bryant, et al., VC3: A virtual cluster service for community computation, in: Proc. of the PEARC 2018, 2018, pp. 1–8.
[22] D. Palma, T. Spatzier, Topology and Orchestration Specification for Cloud Applications Version 1.0, OASIS Standard, 2013.
[23] P. Kacsuk, et al., The flowbster cloud-oriented workflow system to process large scientific data sets, J. Grid Comput. 16 (1) (2018) 55–83.
[24] Y.D. Dessalk, Big Data Workflows: DSL-based Specification and Software Containers for Scalable Execution, The Royal Institute of Technology, 2020.
[25] T. Fernando, et al., WorkflowDSL: scalable workflow execution with provenance for data analysis applications, in: Proc. of the COMPSAC 2018, 2018, pp. 774–779.
[26] C. Zheng, D. Thain, Integrating containers into workflows: A case study using makeflow, work queue, and docker, in: Proc. of the VTDC 2015, 2015, pp. 31–38.
[27] S. Migliorini1, et al., Pattern-Based Evaluation of Scientific Workflow Management Systems, Tech. Rep., Queensland University of Technology, 2011.
[28] V. Cutrona, et al., Semantically-Enabled Optimization of Digital Marketing Campaigns, in: Proc. of the ISWC 2019, 2019, pp. 345–362.
[29] J. Kreps, et al., Kafka: A distributed messaging system for log processing, in: Proc. of the NetDB 2011, 2011, pp. 1–7.
[30] P. Carbone, et al., Apache flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Tech. Committee Data Eng. 36 (4) (2015).
[31] T. Akidau, et al., Millwheel: Fault-tolerant stream processing at internet scale, Proc. VLDB Endowment 6 (11) (2013) 1033–1044.
[32] T. Wegeler, et al., Evaluating the benefits of using domain-specific modeling languages: An experience report, in: Proc. of the DSM 2013, 2013, pp. 7–12.
[33] B. Zhou, et al., SemFE: Facilitating ML pipeline development with semantics, in: Proc. of the CIKM 2020, 2020, pp. 3489–3492.
[34] J. Kranjc, et al., Clowdflows: Online workflows for distributed big data mining, Future Gener. Comput. Syst. 68 (2017) 38–58.
[35] K. Chard, et al., I'll take that to go: Big data bags and minimal identifiers for exchange of large, complex datasets, in: Proc. of the BigData 2016, 2016, pp. 319–328.
[36] G. Kahraman, S. Bilgen, A framework for qualitative assessment of domain-specific languages, Softw. Syst. Model. 14 (4) (2015) 1505–1526.
[37] F. Häser, et al., Is business domain language support beneficial for creating test case specifications: A controlled experiment, Inf. Softw. Technol. 79 (2016) 52–62.
[38] A. Popovic, et al., A DSL for modeling application-specific functionalities of business applications, Comput. Lang. Syst. Struct. 43 (2015) 69–95.
[39] S. Arslan, G. Kardas, DSML4DT: A domain-specific modeling language for device tree software, Comput. Ind. 115 (2020) 103179.
[40] N. Nikolov, et al., Integration of DSLs and migration of models: a case study in the cloud computing domain, Procedia Comput. Sci. 68 (2015) 53–66.

sensors

Article

Big Data Workflows: Locality-Aware Orchestration Using Software Containers

Andrei-Alin Corodescu [1], Nikolay Nikolov [2], Akif Quddus Khan [3], Ahmet Soylu [4,*], Mihhail Matskin [5], Amir H. Payberah [5] and Dumitru Roman [2,*]

1 Department of Informatics, University of Oslo, 0373 Oslo, Norway; alin.corodescu@gmail.com
2 SINTEF AS, Software and Service Innovation, 0373 Oslo, Norway; nikolay.nikolov@sintef.no
3 Department of Computer Science, Norwegian University of Science and Technology, 2815 Gjøvik, Norway; akif.q.khan@ntnu.no
4 Department of Computer Science, OsloMet—Oslo Metropolitan University, 0166 Oslo, Norway
5 Department of Computer Science, KTH Royal Institute of Technology, 114 28 Stockholm, Sweden; misha@kth.se (M.M.); payberah@kth.se (A.H.P.)
* Correspondence: ahmet.soylu@oslomet.no (A.S.); dumitru.roman@sintef.no (D.R.)

Abstract: The emergence of the edge computing paradigm has shifted data processing from centralised infrastructures to heterogeneous and geographically distributed infrastructures. Therefore, data processing solutions must consider data locality to reduce the performance penalties from data transfers among remote data centres. Existing big data processing solutions provide limited support for handling data locality and are inefficient in processing small and frequent events specific to the edge environments. This article 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. The proposed solution considers the available data locality information, leverages long-lived containers to execute workflow steps, and handles the interaction with different data sources through containers. We compare the proposed solution with Argo workflows and demonstrate a significant performance improvement in the execution speed for processing the same data units. Finally, we carry out experiments with the proposed solution under different configurations and analyze individual aspects affecting the performance of the overall solution.

Keywords: big data workflows; orchestration; data locality; software containers

Citation: Corodescu, A.-A.; Nikolov, N.; Khan, A.Q.; Soylu, A.; Matskin, M.; Payberah, A.H.; Roman, D. Big Data Workflows: Locality-Aware Orchestration Using Software Containers. *Sensors* **2021**, *21*, 8212. https://doi.org/10.3390/s21248212

Academic Editor: Haipeng Dai

Received: 7 November 2021
Accepted: 7 December 2021
Published: 8 December 2021

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 [1]. 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) [2–4]. Handling such complexity naturally comes with an increased cost, and the value extracted from the data must, thereby, offset this cost.

Big data workflows formalise and automate the processes that data go through to produce value by providing necessary abstractions for defining workflows and efficiently leveraging underlying hardware resources. In the context of big data workflows, we consider computing resources, such as processors (CPUs), memory, and storage, as relevant hardware resources (from now on referred to as just resources) among others, e.g., [5,6]. 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 manner, both in terms of data integration and processing logic. Accordingly, approaches based on software containers have emerged to create and execute workflows using processing steps in line with these considerations. While it is beneficial to leverage software containers, packaging and isolating applications for deployment, 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 [7]. With the data processing happening on geographically distributed systems across edge and cloud resources, reducing the delay and cost of transferring data over the network becomes crucial. Transferring massive amounts of data to the cloud is expensive and may incur latency, making low-latency scenarios unfeasible. To address these issues, the edge computing paradigm [8] aims to complement cloud computing by leveraging hardware resources situated closer to the edge of the network to offload processing, reduce transfer cost, and satisfy 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. In this context, in order to alleviate the aforementioned problems, this article addresses the following research questions:

- How can data locality be implicitly integrated with container-centric big data workflow orchestration?
- How can software containers be used to facilitate the interaction with different data management systems as part of big data workflows?
- How can containers encapsulating processing logic be used to improve the performance of big data workflows?

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 [9]. Our proposed approach considers (i) data locality, (ii) leverages long-lived containers (i.e., containers whose life-cycle is not tied to a particular data unit) to execute workflow steps, and (iii) handles the interaction with different data sources through containers. We compare our proposed approach with a similar existing solution, Argo workflows. The comparison shows that by considering data locality, our proposed approach significantly improves the performance in terms of data processing speed (up to five times better). We also present a set of experiments with different configurations analysing individual aspects affecting the performance of the overall solution.

The rest of the article is structured as follows. Section 2 gives the relevant context. Section 3 provides an analysis of the problem. Section 4 discusses related work. Section 5 presents the proposed solution and Section 6 describes its implementation. Finally, Section 7 provides experimental evaluation of the proposed solution, and Section 8 concludes the article and outlines the future work.

2. Background

Big data has quickly gained momentum as it allows the exploitation of data to uncover hidden trends that give valuable insights that drive business decisions and support research (e.g., [10,11]). Big data solutions have been successfully leveraged in a large number of industries. The general applicability of big data patterns stems from the fact that many real-world phenomena can be better understood and predicted given sufficient data. The characteristics, e.g., volume, velocity, and variety of big data, translate into challenges at all levels of the technology stack for big data processing:

1. At the infrastructure level , significant raw network, storage, memory, and processing resources are required to handle big data. Often these resources are provided by multiple machines, organised in a distributed system.

2. At the platform level, software frameworks that can effectively leverage the available resources and handle the ever-changing needs of big data operations need to be continuously developed.
3. At the application level, algorithms running on the previously mentioned platforms need to be devised and combined to extract value from the data. Applications can also facilitate the interaction with a big data solution (e.g., visualisation tools used by business executives to analyse the results produced by the solution).

Devising algorithms and tools that help process and extract value from the data further amplify the complexity and cost of building comprehensive big data solutions. Consequently, another V, the value the solution generates, which needs to offset the high cost, is often included as a central characteristic of big data.

2.1. Big Data Workflows

Raw data must be taken through a series of operations (e.g., cleaning, aggregation, and transformation) before producing valuable insights. While it is possible for each of these steps to be manually triggered and independently controlled in an ad-hoc manner, workflow orchestration tools facilitate the automation of the execution and sequencing of these operations, allowing reliable and reproducible execution of complex processes. Key concepts used in workflow orchestration [12] (see Figure 1) include:

- **Step:** Steps are the atomic units upon which workflows are defined. A step encapsulates business logic units that receive data as input from a data source, processes them, and then pushes the outputs to a data sink.
- **Workflow:** Individual steps can be linked together to form a workflow. A workflow is a linear sequence of steps with the semantics of executing the steps. Big data workflows are specified in more complex configurations. A widely used model stems from graph theory, which describes a workflow as a directed acyclic graph (DAG).
- **Data communication medium:** The distributed nature of big data processing warrants the existence of a data communication medium through which information can be exchanged between the components of the system. Thereby, both the inputs and the outputs of a step connect to such channels.
- **Control flow communication medium:** To be able to execute the workflows, control messages (e.g., triggering a step, notifying when a step has finished) have to be exchanged within the system through a control flow communication medium. Examples of such communication mediums include point-to-point network communication between components and message queues.

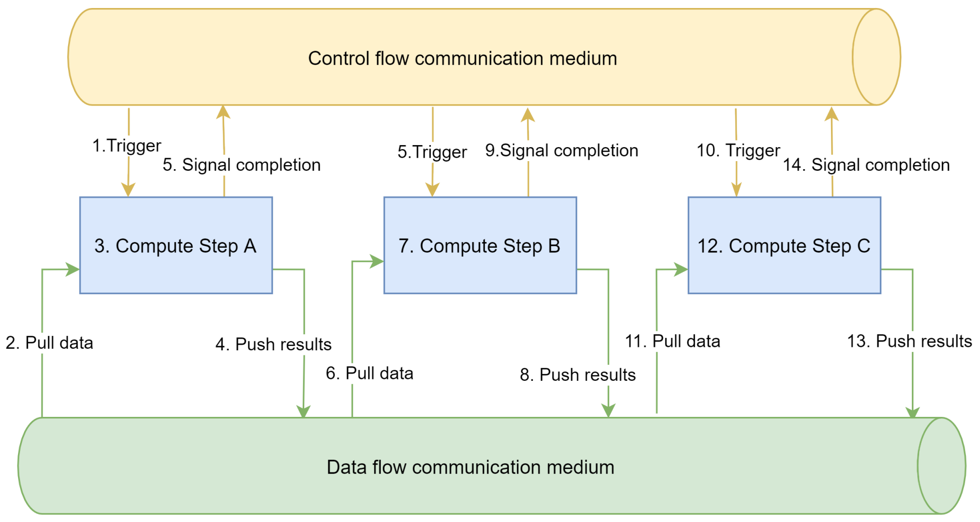


Figure 1. Workflow as sequence of steps with communication mediums.

Creating big data workflows is a complex process involving knowledge from multiple domains (hardware provisioning, cluster management, data handling, different processing steps, definition of workflows according to business needs, and orchestration). Delegating the different responsibilities to independent components allows easier development of both the orchestration frameworks and the workflows running on top of them, reducing the costs and making big data workflows more accessible.

2.2. Cloud and Edge Computing

Cloud computing paves the way for accessible, affordable, and efficient big data processing through scalable, elastic, and "pay-per-use" models. Cloud deployments are best suited for cases where the producer of data is mainly centralised. However, with the advancement of ubiquitous computing, tremendous amounts of data are produced by devices (e.g., sensors, personal devices, and vehicles) at the edge of the network. With the number of such devices increasing rapidly, the traditional model for using centralised cloud resources becomes infeasible. Edge computing [8] complements cloud computing by performing computations on resources physically located closer to the edge of the networks. In this respect, Computing Continuum [13] refers to all available resources for a system, from the edge of the network to the cloud.

Although the edge computing paradigm addresses some fundamental limitations of cloud computing, it also faces a different set of challenges [14]. Among others, these notably include:

- Hardware resources on edge devices exhibit different characteristics and capabilities compared to cloud resources (e.g., processor architectures, processor clock speeds, and amounts of memory). Therefore, software running on such devices has to be designed to consider these resource constraints. At the same time, edge deployments offer limited or no elasticity.
- Edge resources are geographically distributed, and the latencies can differ significantly depending on the distance between the communicating parties.
- Geographical distribution also raises logistical challenges, as these devices can be spread over wide areas and sometimes even in hard-to-reach locations, making provisioning and maintenance a significant challenge.
- The nature of edge resources also makes them prone to failures at the device level (hardware failures) or supporting infrastructure (network failures). Software solutions targeting edge deployments need to tolerate failures gracefully and, if possible, operate offline for extended periods.
- Security and privacy are two complex domains where edge computing plays a significant role. On the one hand, processing data closer to the source can make it easier to adhere to a certain jurisdiction and ensure better security and privacy. On the other hand, large-scale edge deployments are inherently more complicated to secure, mainly due to their geographical spread, and the risk of having devices compromised through physical interference is much higher than in a cloud-only setup.

2.3. Software Containers and Big Data

Software containers are standardised, isolated units of software that package everything required to run an application (https://www.docker.com/resources/what-container, accessed on 8 November 2021). They provide a lightweight and faster virtualisation alternative to hypervisor virtualisation [15,16]. Software containers exhibit a series of characteristics that make them applicable to a wide range of domains:

- Containers ensure the packaged software runs in complete isolation from other applications on the same operating system. Packaging all dependencies in a container can avoid challenging issues such as conflicting dependencies and complex prerequisite configurations.
- The low overhead introduced by containers allows many containers to be run efficiently on a single node, making them a good fit for resource-constrained environments.

- Software that can run in a container is not limited to a particular technology or programming language, allowing solutions leveraging software containers to orchestrate cooperation of components developed using different technologies easily.
- Containers are a widely adopted standard for software packaging, which translates into two major benefits. First, containerised software is easier to share and reuse across different environments. Second, containers can be used to move the execution of logic onto a distributed system's nodes ("function shipping").

Container orchestration solutions, such as Kubernetes (https://kubernetes.io/, accessed on 8 November 2021), simplify the deployment and management of highly distributed systems by creating abstractions for the underlying infrastructure and facilitating the interaction between the components of an application. Software containers are extensively leveraged in cloud environments [17] and, in some cases, containerised applications are referred to as cloud-native applications [18]. Several works also identify software containers as a feasible technology for resource-constrained edge environments [19–21]. In the context of big data, containers have been used to simplify the deployment and management of entire big data solutions or individual components. Leveraging containers in big data solutions can also lead to performance improvements when compared to the hypervisor-based virtualisation alternatives [22].

There is a distinction between the two strategies of using containers in big data solutions at a high level. First, software containers are leveraged to deploy and manage the components of a big data processing platform. Although this approach simplifies the deployment process, it does not influence the run-time aspects of the platform. Second, software containers are used as an integral part of the architecture and as a mechanism through which custom behaviour can be injected into the platform (e.g., data processing logic). Software containers have gained much traction in the field of microservices as they greatly simplify the management of highly distributed systems [23,24]. Such an architecture can provide some benefits, including modularity, loose coupling, and technology independence, which align with the needs of big data workflow systems.

3. Problem Analysis

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 given system. In the following, we introduce the essential concepts and techniques to reduce the execution time and bandwidth usage in big data workflows.

3.1. Data Locality

Data locality [25] refers to 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 (e.g., 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, this makes it more challenging to leverage individual hosts comprising the distributed systems. For example, a fundamental invariant of current computer architectures is that a CPU can only work with data present in the memory of the same machine.

Consequently, the movement of data across machines becomes an integral part of any big data system. With traditional communication protocols that rely on the operating system network stack (e.g., TCP/IP-based protocols), latency becomes critical for many use cases. To this end, more efficient protocols have emerged. For example, RDMA (Remote Direct Memory Access) [26] 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 volume of data is significant in the context of big data, 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 [27,28]. For example, Apache Spark [29] leverages the information provided by the Hadoop File System (HDFS) and knowledge about outputs of previous executions of jobs 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 considered when scheduling data processing tasks. 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 tasks effectively [30]. Studies exist that propose advanced scheduling strategies to balance the reduction in data transfer with load distribution (e.g., [31,32]).

3.2. 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, 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 the performance and bandwidth utilisation, as different protocols provide different guarantees related to data transmission (e.g., TCP ensures an ordered and lossless transmission but requires multiple round trips to establish connections and exchange data, while UDP is faster but less reliable).

Different protocols introduce additional overhead by injecting more data in transmission packets (e.g., HTTP headers). Techniques, such as compression and binary serialisation, help reduce the size of the payload. Furthermore, there exist studies exploring the use of RDMA-backed memory channels to support fast and efficient inter-container communication (e.g., [33]). 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 communication contract between components allows decoupling them from one another.

3.3. Lifecycle Management of Containers

Software containers are a lightweight virtualisation alternative to traditional hypervisor-based virtualisation; however, there is still a cost associated with starting up and shutting down containers on demand. Life-cycle management has a high impact on workflow execution time, and reusing containers to process multiple units of data (i.e., long-lived containers) is a way to improve it [34]. 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.

In edge computing environments, the available resources are limited, and less data can be processed on a single host at a given point in time. Furthermore, with data being constantly processed in small batches (or even streamed), there is a need for this processing to happen as quickly as possible to achieve the desired throughput. In such cases, the overhead of setting up and tearing down containers can quickly add up and become a significant bottleneck for the solution's performance.

3.4. Integration with Data Management Solutions

One of the pillars of big data processing is 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.

4. Related Work

In this section, we first present and discuss the related work in terms of existing published literature and then in terms of existing tools.

Regarding the published literature, Valerie et al. [35] discuss the effect of in-memory processing and data locality on neuroimaging data sets and show the importance and benefits of data locality. However, they do not propose any new system or orchestrator tool and only evaluate the performance of existing systems. Ching et al. [36] propose new techniques for implementing locality-aware virtual machines. They mainly focus on improving the performance of MapReduce programs in heterogeneous and hybrid cloud environments and propose a technique to enhance data locality using a virtual machine mapping technique. Thereby, the locality-aware technique balances workloads and reduces communication overheads at run-time, but it is only restricted to MapReduce applications. Therefore, it cannot be employed with the more general-purpose software containers. August and Christoph [37] present a method of extending smart containers for data locality-aware skeleton programming. They extend the existing SkePU skeleton programming framework to enhance the performance of sequences of transformations on smart containers. However, the framework does not provide support for orchestrating workflows or container lifecycle management.

Bu et al. [38] describe a task scheduling technique for MapReduce applications that minimizes interference while keeping task-data localization. The study, however, disregards network effects, assuming that data flow between co-hosted virtual machines is equally efficient as local data access. Choi et al. [39] present a mechanism for locality-aware resource management in High-Performance Computing (HPC) cloud environments, called Data-Locality Aware Workflow Scheduling (D-LAWS). Their solution condenses virtual machines and includes task parallelism through data flow into the task execution planning of a data-intensive scientific process. However, they do not take into account the role of software containers in scientific workflows.

Ahlehagh et al. [40] present a video-aware scheduling strategy that includes storing video data in a macro-base station to boost video throughput and lessens the likelihood of movies freezing. A heuristic approach to the storage allotment issue in macro-base stations is presented by Gu et al. [41]. Small base stations may deliver better data rates than macro-base stations since they are located closer to users. Finally, Vengadeswaran and Balasundaram [42] propose an approach that also considers the data locality factor, but it is limited to Hadoop. The default data placement strategy of Hadoop creates and allocates blocks randomly across the cluster. To overcome this issue, they propose an optimal data placement strategy to improve the performance of big data applications. These methods focus on optimising data placement strategies rather than the real-time migration of computing steps closer to the data.

In the following, we review existing orchestration tools selected 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 data management solutions.

- Snakemake [43] 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 (https://www.kubeflow.org, accessed on 8 November 2021) is a workflow orchestration tool for machine learning-related workflows. The only storage supported is Minio (a cloud-native, open-source storage solution implementing Amazon S3 cloud storage protocol). It offers no support for data locality.

- Makeflow [44] is a workflow orchestration tool that can 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 (https://github.com/pachyderm/pachyderm, accessed on 8 November 2021) is another machine learning workflow orchestration solution, but the only storage system it supports is the Pachyderm file system, a distributed file system built to provide supporting features to Pachyderm.
- Pegasus (https://pegasus.isi.edu, accessed on 8 November 2021) 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 using only file systems.
- Airflow (https://airflow.apache.org, accessed on 8 November 2021) is one of the most popular data workflow orchestrators that supports the execution of the workflows on a Kubernetes cluster. It also controls where instances of steps are created, but it should be set manually when the workflow is defined, making it inefficient in dynamically-changing environments. It is possible to integrate different data management solutions by extending the Airflow code with providers, limiting it to Python implementations only.
- Argo Workflows (https://argoproj.github.io/projects/argo, accessed on 8 November 2021) 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 ad-hoc changes and integration with the Argo code libraries.

All of the considered solutions leverage short-lived containers as part of the orchestration—a container is created to execute work and is destroyed as soon as the processing completes because these solutions target primarily batch processing scenarios. In terms of data locality specification, Argo offers the most expressive features as it leverages the complete functional offering of Kubernetes. However, by default, the limitation of having to specify data locality at workflow definition time (introduced with Airflow analysis) applies to Argo. 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. Although limited in terms of data locality features, Pegasus does handle data locality implicitly, without modifying 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.

5. Proposed Solution

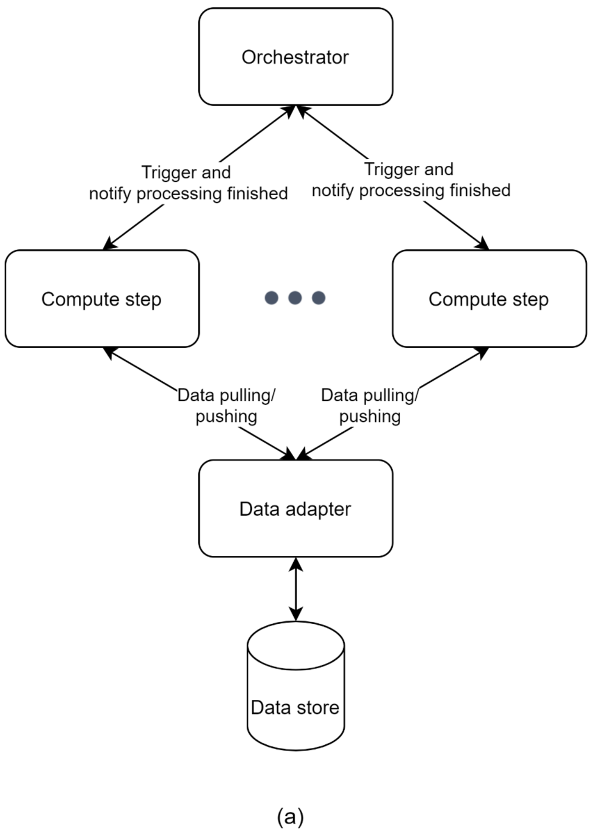
We propose an approach based on a workflow system architecture covering the run-time considerations of big data workflows that take into account the separation of concerns principle. The proposed architecture has three main layers:

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**.
2. **Data layer:** It collectively refers to all the components involved in data handling (i.e., 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, in a

sequence, they form a workflow. Additionally, the approach allows 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. Using a centralised architecture is motivated by leveraging data locality when the execution of 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 are 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 system. These represent the units of work at both the orchestration level and individual step level. Each unit is processed independently, and multiple units can be processed in parallel across different steps. Handling the execution of the workflow at the data unit level can improve the performance significantly compared to the models that execute steps synchronously (all outputs of the previous step have to be available to start the next one) [12,45].

Whenever a new unit of data is available in the system, the orchestrator is notified, and it passes on the notification to a "compute step" for processing. 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 2a depicts a high-level overview of the three layers and their interactions. The following sections present each layer in detail.



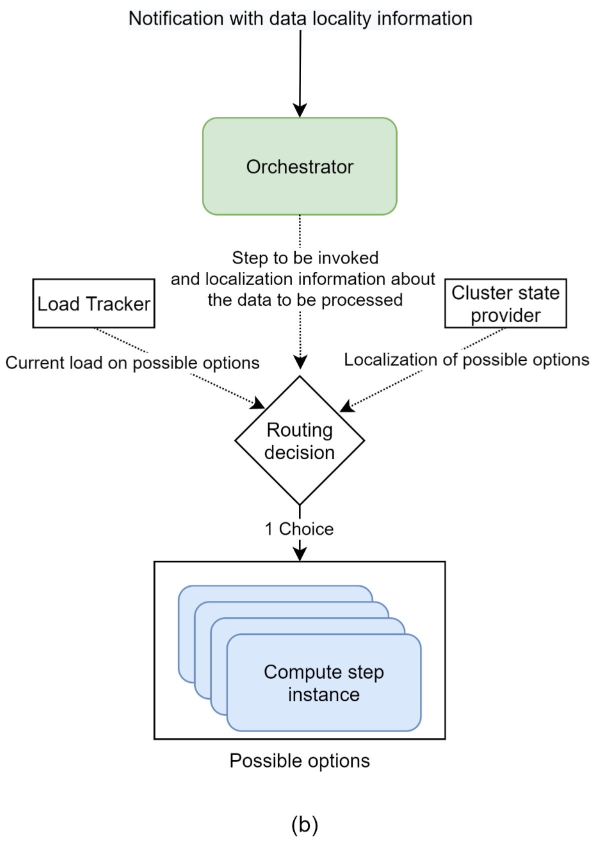

Notification with data locality information
Orchestrator
Step to be invoked and localization information about the data to be processed
Load Tracker
Cluster state provider
Current load on possible options
Localization of possible options
Routing decision
1 Choice
Compute step instance
Possible options
(b)

Figure 2. (**a**) Detailed view of a compute step, and (**b**) data locality as a multi-objective optimisation problem.

5.1. Control Layer

Upon receiving a notification indicating that new data are available to be processed, the orchestrator needs to determine what type of compute step needs to invoke 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 process 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 (see Figure 2b). This list is not exhaustive and presents only a subset of potential aspects considered when making a routing decision. Optimising cost and performance while ensuring all the explicit requirements of the workflow are met makes the routing a complex, multivariate optimisation problem, with no clear best decision where trade-off in one or more areas is necessary.

To provide data locality, the orchestrator needs to know the physical location of both the data to be processed and the possible target step instances and also a model to calculate the distance between two locations. Moreover, the orchestrator needs to continue operating when presented with unexpected, partial, or missing locality data, as some data adapters may not provide granular data locality information. 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 pre-calculated or estimated asynchronously, as acquiring information about all the possible targets synchronously may incur a significant performance cost.

5.2. Data Layer

The ability to share data between the nodes hosting the compute step instances is a fundamental requirement of the proposed solution. 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 reusability of both components in multiple workflows. For example, a compute step processing images can be used to process images from multiple data sources, and the same data solution can be reused with different compute steps.

In addition to reusability, 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 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 to support locality-aware work scheduling. The data locality model employed to communicate the localisation information needs to have 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. Different data storage solutions have different capabilities of exposing information about where data are stored; thus, the data locality model should support reserved values to indicate that parts of the information are missing.

5.3. Compute Layer

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

1. A compute step is provided with 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, 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.

5.4. Extension Model

A widely used model for building upon and extending a framework is integrating a software development kit (a set of language-specific libraries) that handles the interaction between the features provided by the framework and the code extending or building upon it (user logic). The communication between the two components then happens via language-specific constructs, such as method invocations, and is backed by the memory of the hosting process. With container technologies easing the management and deployment of microservices, the framework functionality can be completely separated from the user logic by being placed in a dedicated container. This container can communicate with the user logic (presumably in another container) through an external communication medium.

This approach provides better isolation of the two components and allows for integration between different programming languages or frameworks, as the communication is agnostic to the used programming language. Even though the communication medium (e.g., network calls, files, sockets) can be accessed by both components, a crucial aspect is to ensure the communication protocol (e.g., HTTP, gRPC) is supported by both ends. 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 3). 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 it allows them to have simplified interfaces they need to adhere to dedicated only to their function (handling data or processing it), as follows:

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 must send a request to trigger the computation.
4. The output data are passed to the data adapter and the orchestrator is notified that new data have become available.

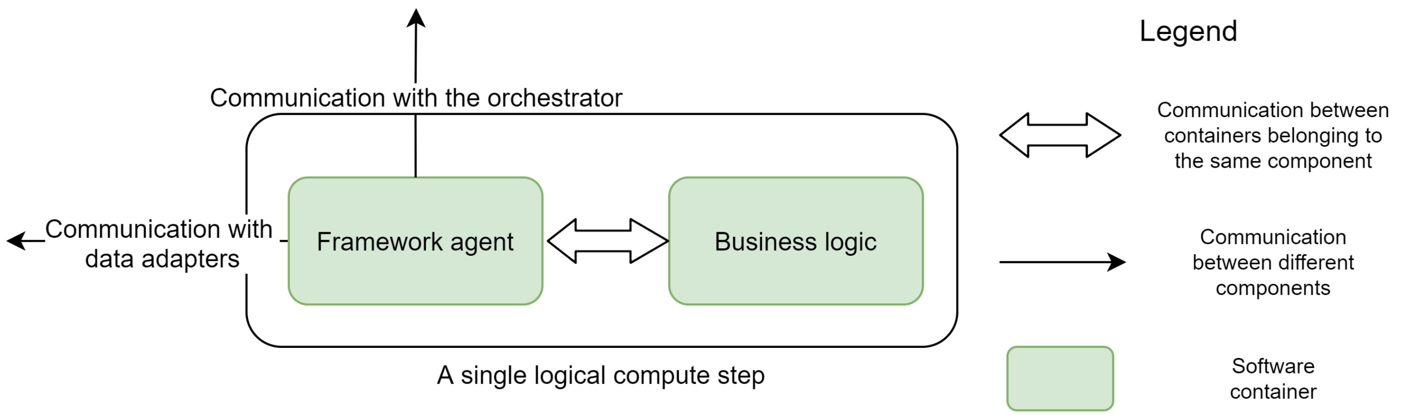

Communication with the orchestrator
Communication with data adapters
Framework agent
Business logic
A single logical compute step
Legend
Communication between containers belonging to the same component
Communication between different components
Software container

Figure 3. A detailed view of a compute step.

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 define and execute 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, allowing them to be created and to evolve independently, thus helping with the separation of concerns between the stakeholders involved in creating workflows. The proposed architecture allows the framework users to inject data and processing logic into a workflow definition, combining the two elements.

6. Implementation

We implemented our proposed solution as a proof of concept using Docker and Kubernetes [15]. The code of our 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 (https://github.com/alin-corodescu/MSc-workflows, accessed on 8 November 2021).

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 was 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. Additionally, it is tailored to work well in resource-constrained environments, such as edge environments.

The following is a brief introduction to the main Kubernetes abstractions and features referenced in the remainder article:

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 units that 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 with 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.

6.1. Fundamentals

The current implementation opts for a point-to-point communication model (cf. control flow communication medium). It allows for more straightforward and explicit communication between two parties, making it more suitable for the deliberate routing decisions that consider data locality. The gRPC framework is chosen to support the communication between different components. The containers used to perform the steps, inspired from the microservice-oriented architecture, 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. A clear distinction between different aspects of the communication between components is made:

1. The components communicating with each other adhere to a particular contract or interface.
2. The actual execution is handled by the logic implementing the contract/interface.
3. The communicating parties have to employ compatible serialisation/deserialisation protocols to exchange messages.
4. The transport aspect encompasses both fundamental mediums (memory, disk, network) and abstractions on top of those (such as the HTTP protocol).

By leveraging the long-lived nature of the containers, the proposed implementation also attempts to reuse established connections to remote machines (connection pooling). The TCP protocol requires three network round-trips to perform the three-way handshake. If the protocol on top of TCP uses SSL to secure the connections (e.g., HTTPS), more network round-trips are required to negotiate and establish a secure connection. If the connection pooling is not enabled, the time to establish connections increases with the number of connections. For example, if many small data units flow through the system, the time for establishing connections can represent a significant percentage of the total execution time of the workflow. With geographically distributed resources, the performance cost of network round-trips becomes more significant with the increase in the physical distance between hosts, thus making connection pooling even more critical.

6.2. Orchestrator

A single instance of the orchestrator (Figure 4) is deployed, and it is for all the components in the cluster through a Kubernetes service. The single-replica strategy is needed because the central orchestrator is a stateful service that keeps the state 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 names of steps to find pods hosting the specified processing logic through Kubernetes labels. In addition to the name, data source and data sink identifiers are also part of step specification. The data source and sink identify the data storage solutions where the input and output data should be retrieved and stored, respectively. A single workflow can utilise multiple storage solutions, such as cloud, edge, and shared storage (cf. data communication medium).

Registering and retrieving workflows is done through a service, **workflow definition service**, using **workflow definition store** and exposed by the central orchestrator. The orchestrator exposes another service, **orchestration 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 [46]. The orchestrator is responsible for notifying the pods hosting the logic necessary to process the new unit of data according to the workflow definition. The communication between the orchestrator and pods hosting steps is asynchronous with respect to the actual execution of the step. The step pod does not wait for the step execution to finish before returning to the orchestrator. The **work tracker** component stores a mapping between a request and the step executing the request. The orchestrator reads this mapping to determine what is the next step to invoke. In parallel, the work tracker also keeps a counter of active computation requests for each pod. When a request arrives, it is put in a queue. This allows a response to be sent back to the caller without waiting for the finishing of the processing of the data, which can potentially be a long-running operation. The **executor** component retrieves requests from the queue and starts a background thread to process the request, allowing it to process multiple requests in parallel quickly. The **cluster state provider** component, using Kubernetes API Server, provides a list of all the pods in the cluster that can perform the desired processing logic. Finally, the agent extends the central orchestrator and handles the orchestration at a local and individual step level. It handles the communication with other components on behalf of the compute step.

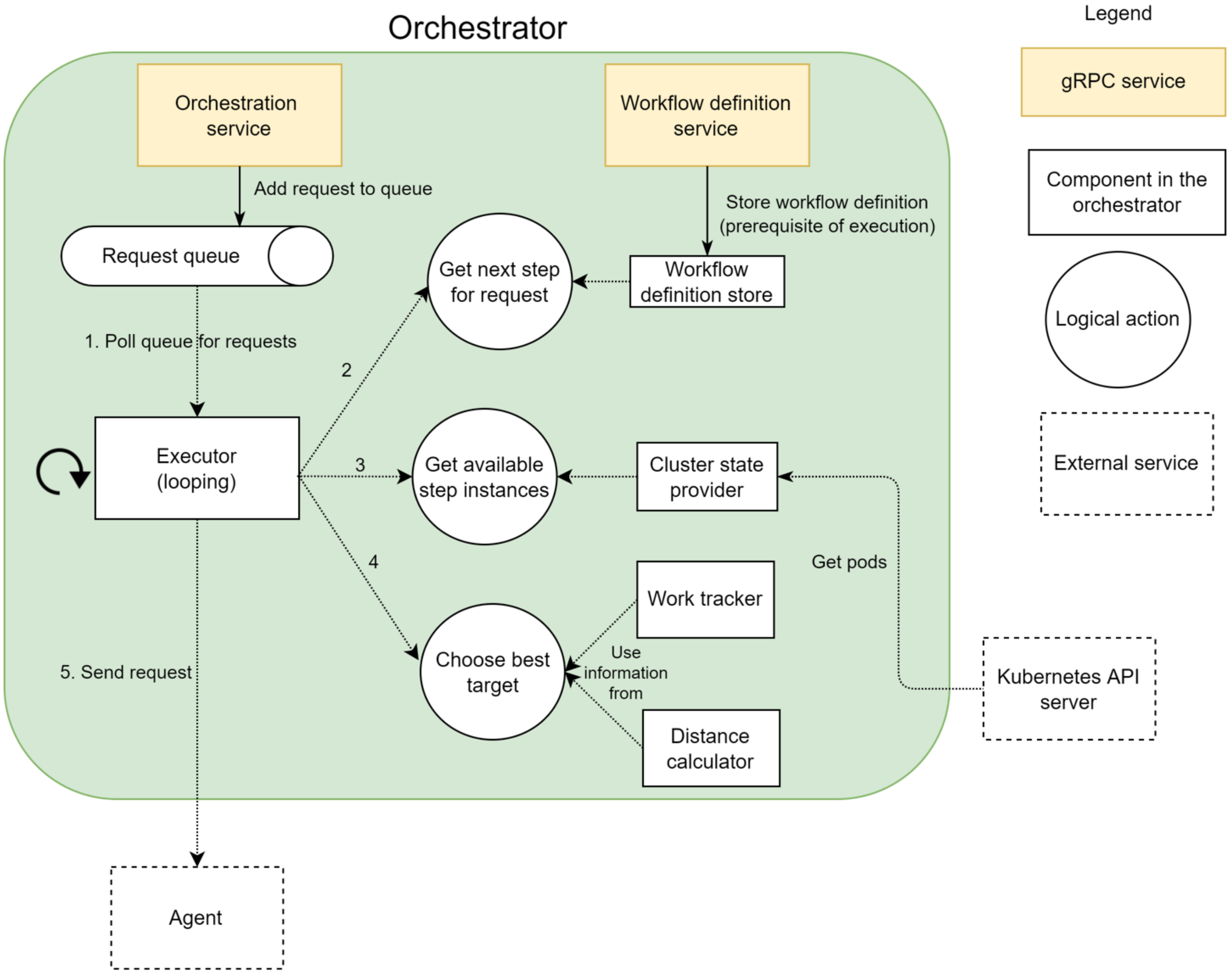


Figure 4. An overview of the orchestrator component.

6.3. Request Routing

Request routing discussed earlier is a multi-objective problem. For the proposed implementation, request routing uses the distance and current load as inputs. Data localisation captures information about the host (either storing the data or running a pod) and the zone. The zone is a general term meant to capture groups of hosts near one another. When routing a request through the orchestrator, a unit of data with a specified data localisation must be paired with a pod that can process it, with a potentially different data localisation.

One input to the routing decision algorithm is the distance between the localisation of the data to be processed and the localisation of each pod. A large number indicates the data transfer to the pod is likely to take longer. The distance between the same hosts is considered to be 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 the distance between two different zones. The orchestrator uses the **distance calculator** to calculate the distance (see Figure 4) and the work tracker component to get 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 zone level and falls back to a different zone 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. Given seven requests to route, the order in which each strategy leverages the available resources is presented in Figure 5. Both algorithms exhibit the valuable characteristic of leveraging the full set of resources available in the continuum in a prioritised order to reduce the cost and latency introduced by data transfer while at the same time avoiding overloading a particular set of pods.

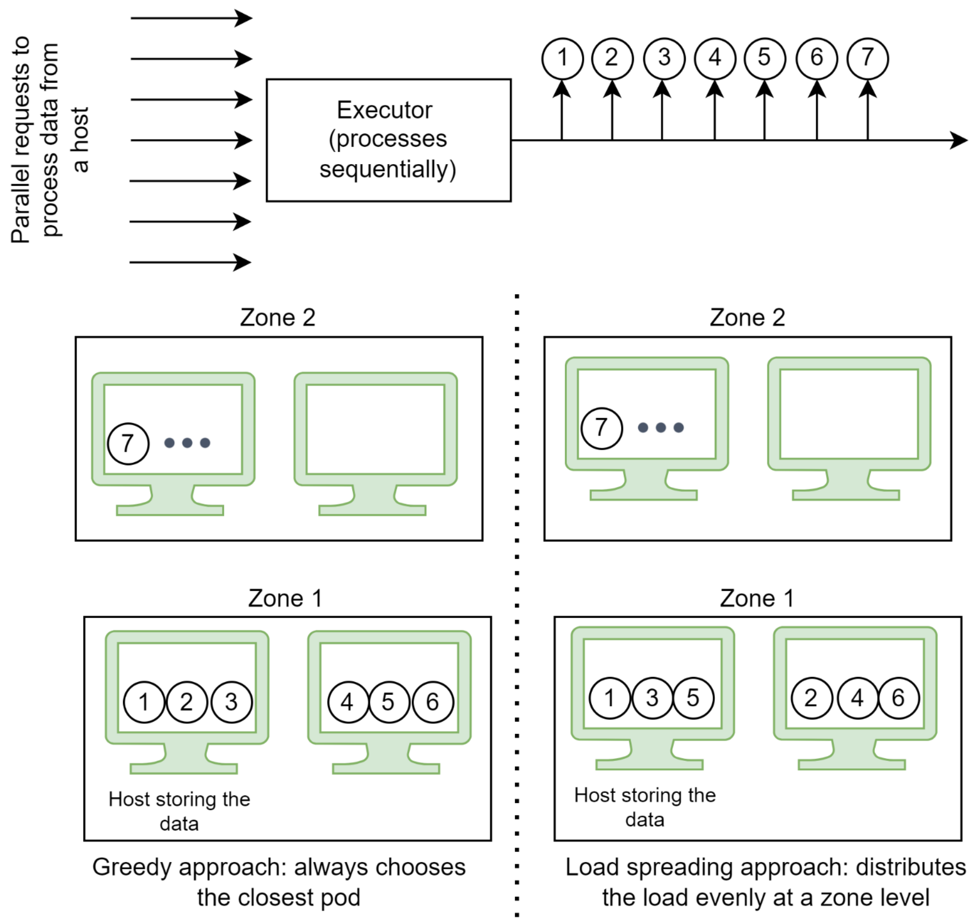

Parallel requests to process data from a host
Executor (processes sequentially)
1
2
3
4
5
6
7
Zone 2
7
Zone 2
7
Zone 1
1
2
3
4
5
6
Host storing the data
Zone 1
1
3
5
2
4
6
Host storing the data
Greedy approach: always chooses the closest pod
Load spreading approach: distributes the load evenly at a zone level

Figure 5. Routing priority for the greedy and load spreading algorithms.

6.4. 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 [47]. The agent and the compute step are separated in different containers, and they can communicate efficiently via local network calls within the host. 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, 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 are 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 request, back to the orchestrator to notify that new data are available.

The agent keeps track of the number of concurrent requests being executed, allowing only a configurable number of requests to be executed in parallel.

6.5. Data Adapter

In the current implementation, 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. Different storage solutions can be integrated, and the associated adapters can be added to extend the solution's capabilities.

The distributed storage system leverages the local storage of every node in the cluster to store the data 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. The data master is responsible for keeping track of the location of files stored in the proposed implementation for the distributed storage. The data master component could potentially be extended to offer more functionalities (e.g., data lineage, sharing the same data across multiple workflows, and others). 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.

The proposed solution for the distributed storage system only offers basic functionalities and lacks most of the features other alternative solutions provide (for example, data replication and redundancy, security, and resilience to failures). However, in some cases, it is possible that results from the processing steps only need to serve as input for the next step and do not have strict requirements for how the data should be stored. For such cases, the presented solution can be used as the data transfer medium, thus benefiting from an efficient means to fully leverage the data locality functionality. Within a single workflow, it is possible to use multiple storage solutions.

6.6. Example Flow

An example request flow is depicted in Figure 6. When a request is placed in the queue, the orchestrator is notified about available new data (1). The executor then retrieves the request (2) and starts a background thread to process the request. The first step in the processing is to determine which step should be invoked for the current notification (3), based on the request and the workflow definition. Once the step is determined, the cluster state provider provides all the available pods to perform the required step (4). Based on the current load and distance to the data to be processed, one of the available options is chosen (5), and the request is sent to the appropriate agent (6).

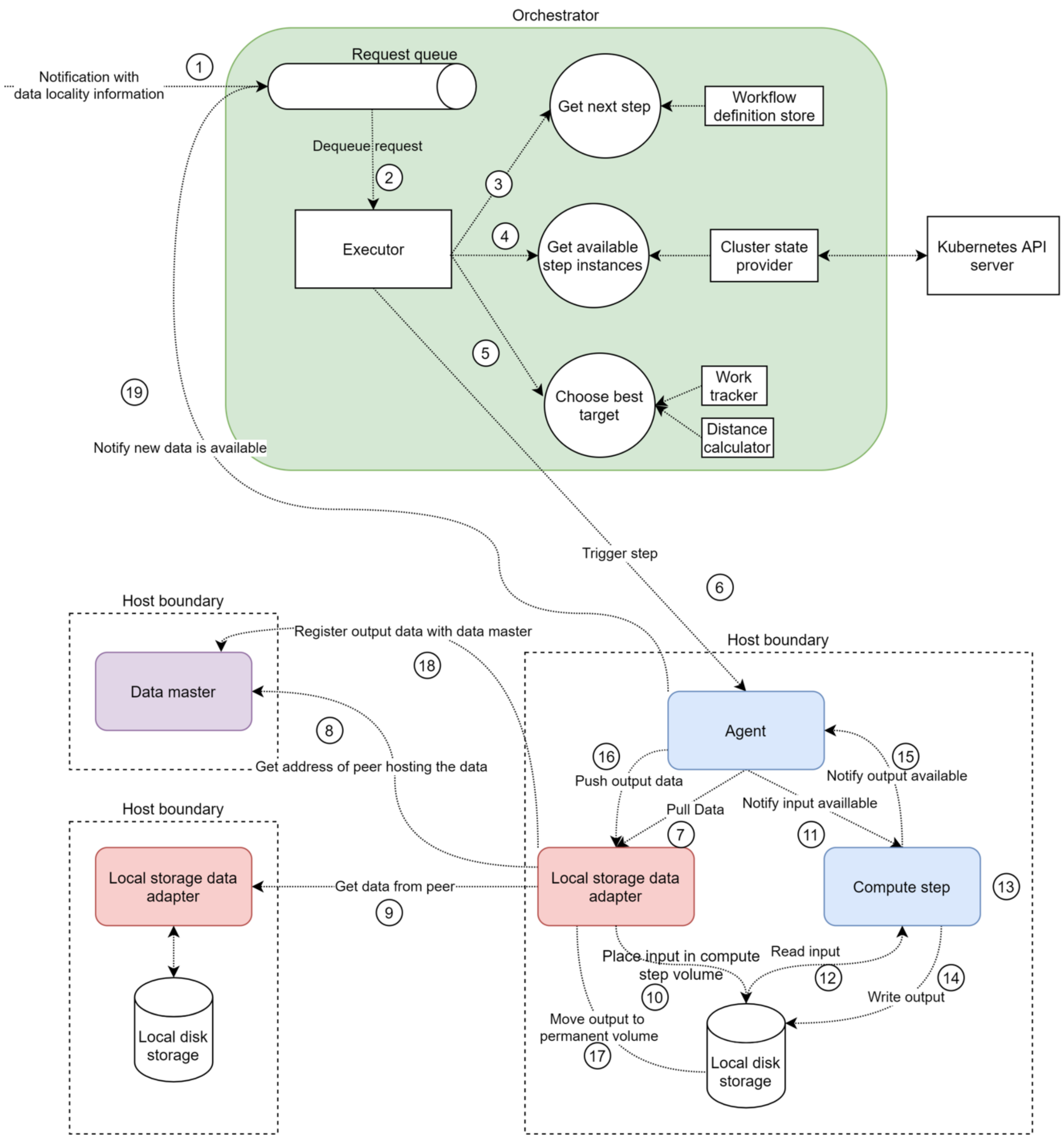


Figure 6. An example request flow in detail.

Once the agent receives the notification, it sends a request to the correct data adapter, based on the received data (7). If the file is not available locally, the local storage adapter will send a request to the data master (8) to determine where the file that needs to be processed is stored and will then send a request to download the data from the respective peer (9). After (8–9), or if the file is available locally, the local storage data adapter places the data in the volume the compute step can access (10) and responds to request (7). The agent then notifies the compute step that data are available (11), and the compute step reads the data (12), performs the processing logic (13), writes the output to the corresponding volume (14), and responds to the agent with the path to the output data (15). The agent sends a request to the local storage adapter to register the newly available data (16). The local storage adapter moves the new data into the permanent storage folder (17) and notifies the data master about the new data (18). The agent notifies the orchestrator that new data are available (19), and the process is repeated for all units of data flowing through the system.

7. Evaluation

Through a series of experiments, we compare: (i) our approach with Argo Workflows in order to analyse the run-time performances, and (ii) different configurations for the proposed solution with one another, for analysing individual aspects in isolation. 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 Standard D2s v3 (two vCPUs, 8GB memory) virtual machines, provisioned in three different Azure regions (EastUS, WestEurope, and NorthEurope) to mimic the geographical distribution of resources in a real cloud and edge topology. One virtual machine serves as the Kubernetes master node (and did 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 (https://k3s.io, accessed on 8 November 2021) distribution of Kubernetes is manually installed on each virtual machine. All machines are part of the same cluster.

We use an example workflow to evaluate the architecture and implementation of the systems, which contains 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 in specific cases is more restrictive in terms of size and data characteristics. For the first three steps, every machine in the cluster runs one instance of each step type. The final step is run only on 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 the experiments. The load generator is responsible for injecting data into the cluster and notifying the orchestrator when data are available. The load generator creates files containing random bytes and triggers 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 are retrieved using the Jaeger API and is exported to CSV files, which are later analysed through Python/Jupyter notebooks.

The choice of the testing environment is motivated by the main focus of the paper that is the impact of data locality on a geographically distributed computing setting. The proposed cloud setup is capable of creating a geographically distributed system by provisioning virtual machines in different regions and integrating them into a single system. There are key differences from using a real IoT/Edge/Cloud setup, such as hardware heterogeneity (different CPU capabilities and even architectures, I/O throughput, network characteristics). However, to better isolate the benefit of data locality from other factors, the cloud setup is sufficient, as it reduces the entropy introduced by a real world setup. Changing the experimental setup either at hardware level or any upper level may result in different results, to be demonstrated and discussed further in the following subsections; however, we consider that the conditions simulated in the test setup are sufficiently generic to be applicable to a wide-range of real world scenarios. We address the primary parameters (data size and contention rates) that could influence the benefits introduced by the data locality and leave out the parameters that could play a secondary role.

7.1. Comparison with Argo Workflows

The example workflow presented earlier is run with four different file sizes/counts, both on Argo Workflows and the proposed solution, for comparing run-time performance. 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 were 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 s. 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.

Figure 7 presents the average running time of a workflow, given different data sizes and numbers of files 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, 1 MB" indicates that three files of 1 MB 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 several 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 can observe the following:

1. In the case of processing small data chunks that take little time to process, the proposed solution outperformed 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 reduces.
3. Data from the second case (three files, 10 MB) show that using data locality does not affect the case of Argo workflows. Furthermore, in the fourth case (three files, 100 MB), using data locality in the workflow has a detrimental effect on performance.
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 10 MB and 100 MB, 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 proves faster to execute the example workflow. However, the benefits vary, depending on the data size, from 500% (for the three files, 1 MB) to roughly 20% (for the two files, 100 MB case).

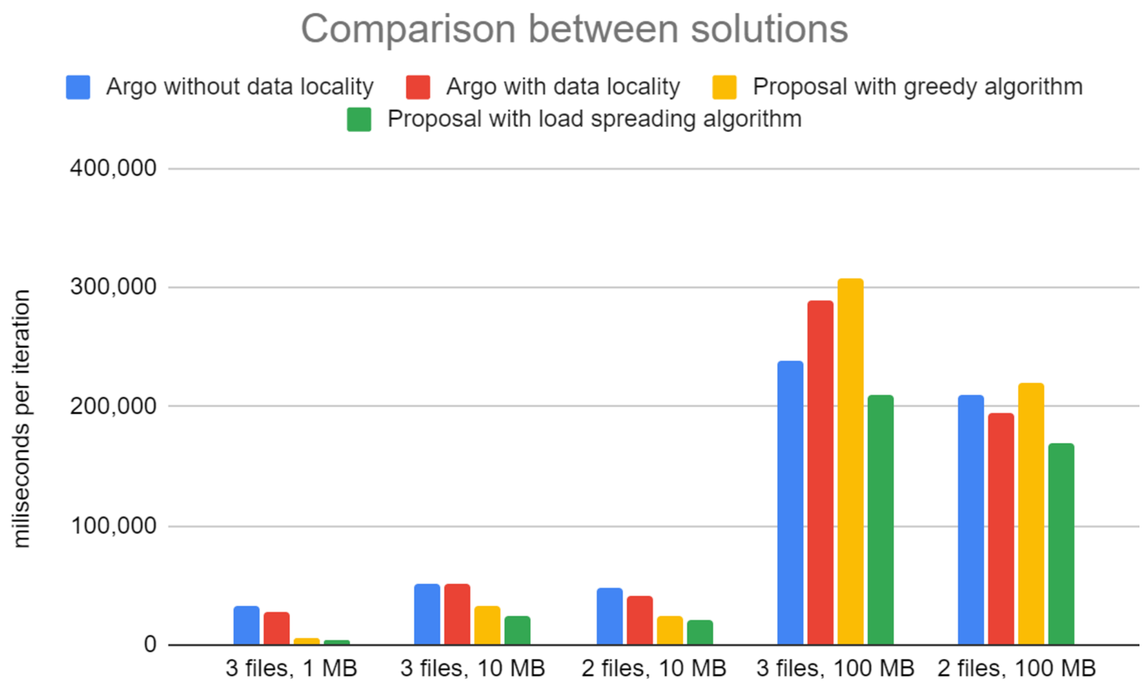


Figure 7. Performance comparison between the proposed solution and Argo Workflows.

7.2. Evaluation under Different Configurations

A series of experiments are run with different configurations of the proposed solution to better understand the effect and behaviour of particular aspects in isolation. The time spent during the execution of the workflows is split into three categories:

(i) **Data** is referring to the time spent handling the necessary data movement (downloading the input data for a step, uploading the outputs, and moving data between directories on the node file system).
(ii) **Compute** is referring to the time spent executing the logic of the processing step.
(iii) **Control** is referring to the time spent in the orchestration component and calculated as a difference between the total execution time of a step measured by the orchestrator and the time spent on the other two categories.

Optimising individual areas does not always translate into significant improvements in end-to-end performance. In some cases, optimisations in one area may degrade performance for the other areas. In the example workflow, most of the time is spent on the compute category, especially on larger data sizes. A different step implementation, which only reads the input and writes it to the output directly (echo functionality), is used for the following experiments. Under the same conditions, the two implementations cause a significant difference in the distribution of time. Figure 8 presents the distribution of where time is spent on average for a single-step processing 10 MB of data and showcases the significant difference between the two-step implementations.

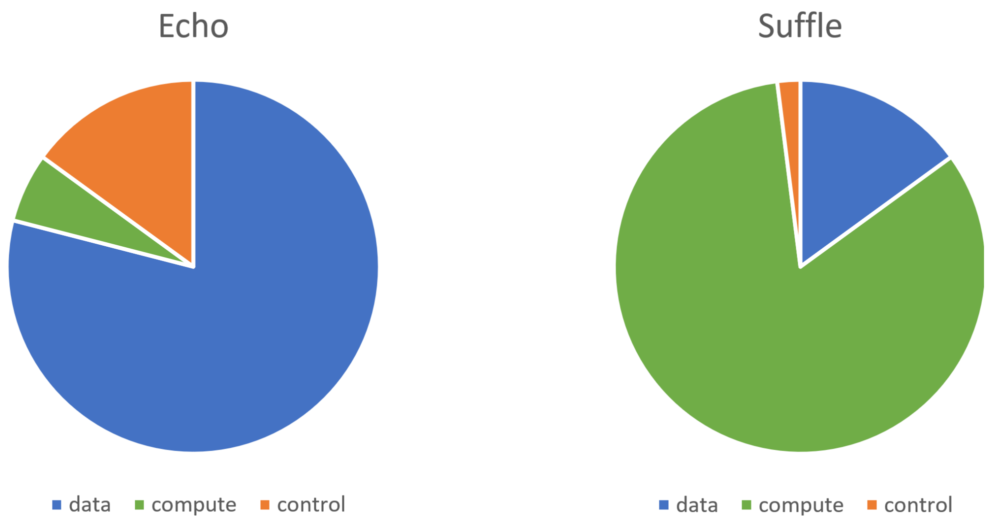


Figure 8. Time distribution for same workflow, but different steps: echo (**left**) and byte shuffle (**right**).

Regarding data handling, we analyse locality-aware routing, hard linking, and load spreading across the Computing Continuum. To study the effect of the locality-aware scheduling in isolation, a configuration allowing the orchestrator to skip the locality-aware routing and rely instead on reaching individual step instances through a Kubernetes service is used. By default, Kubernetes services use a round-robin routing algorithm, thus spreading the requests between all available instances, regardless of their physical localisation. Figure 9 shows a significant difference in average time spent for transferring data between machines, with data locality significantly reducing the time spent. The X-axis represents three different experiments. We pass files of 1 MB, 10 MB, 100 MB through the workflow, respectively. These sizes are reasonable when taking into account the scale of IoT/Edge setups. For example, given 1000 devices generating 100 MB per minute each, it suddenly becomes 100 Gb per minute in the entire system. The Y-axis represents the average time spent on moving data (in milliseconds). We observe that the more routing decisions in data are transferred over more considerable distances (e.g., from WestEurope to EastUS), the higher the step's overall execution time. This implies that data locality is more beneficial for systems spread over more expansive areas.

An experiment comparing the performance of hard linking against the copying data indicated that hard linking executes in constant time. In contrast, time spent copying data increases with the size of the data. However, the performance benefit is negligible at the end, especially under the assumption that the size of the data directly influences the execution time of the processing step. For load spreading across a Computing Continuum, tuning the example workflow with configurations where data is produced in a single zone at increasing rates showed that the proposed solution spreads the load across the available resources. Prioritising step instances, in the order of host storing the data, host in the same zone as the host storing the data, host in the next closest zone, and host in the EastUS zone, results in bandwidth utilisation savings, especially for cases when the data locality information captures the exact host storing the data. The greedy solution is more likely to perform better when it comes to bandwidth savings, as it weighs data locality higher than the load spreading. However, it may perform poorly in terms of execution time for resource-intensive cases, as presented in experiments with Argo workflows.

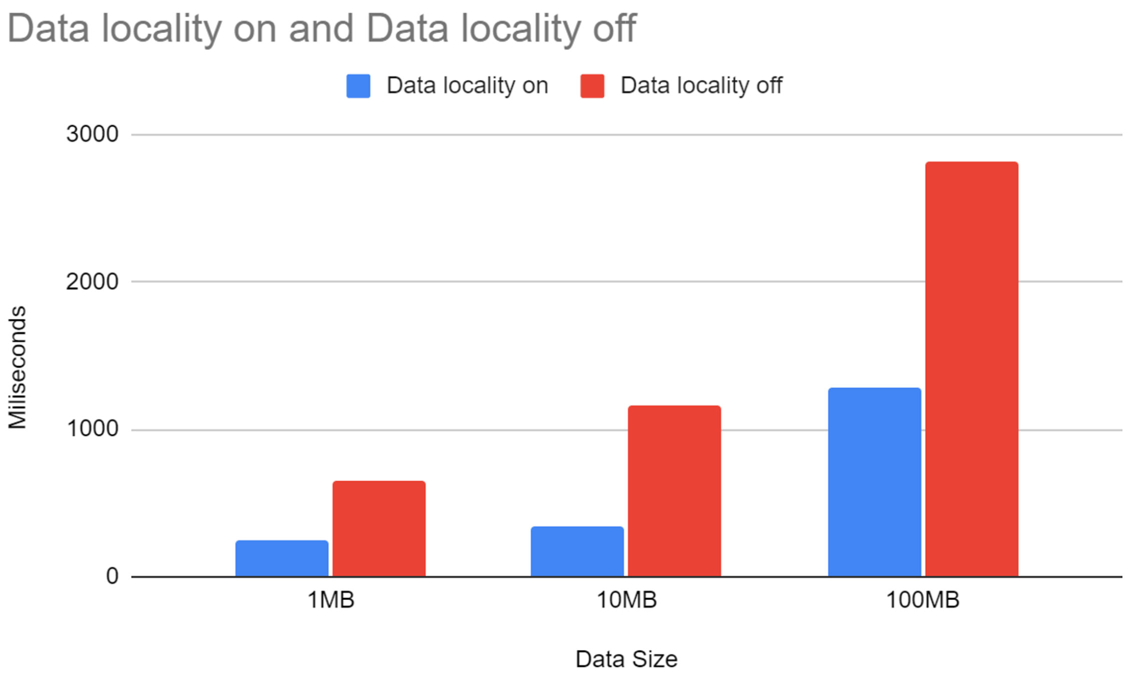


Figure 9. Average time spent on transferring data with and without data locality.

Regarding the control time, we focus on long-lived containers and connection re-use. The significant difference between Argo and the proposed solution in the cases where small and frequent data units need to be processed is due to the overhead introduced by Argo using ephemeral containers. Apart from the cost of instantiating the container, observations made during the experimentation indicate that a significant portion of the time is spent starting up the application after the container is created. This observation is made because the first run of a workflow after deploying the Kubernetes cluster was significantly slower than any subsequent run. For the example workflow, with three files of 1 MB each as input, the first run took roughly 18 s while the subsequent runs took around 5 s.

We compare two configurations of the agent to measure the effect of connection re-use in the agent component: (i) in the first configuration, the connection re-use is not enabled; thus, a new connection for each message to be transmitted should be created, and (ii) in the second configuration, a single connection is re-used for all the messages. The observations indicate that while there is a performance gain in isolation (on average, 30 milliseconds per message for the first case and 10 milliseconds per message on the second case), the impact is not noticeable on the end to end latencies for the considered data sizes. However, more frequent events and lower data sizes could potentially make connection pooling a relevant optimisation.

The observations made throughout the experiments indicate that there are several variables on which the benefit of the proposed approach depends:

1. The nature of the processing steps: The resource contention caused by multiple steps running on the same machine can influence the routing algorithm's optimal configuration controlling the balance between load spreading and data locality.
2. Distance and connection speed between the resources: A topology spreading over a wider area benefits more for the data locality-aware routing.
3. Frequency and processing duration of events: The biggest improvement is obtained by leveraging long-lived containers for frequent and small events.

8. Conclusions

This article 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 and frequent data units by leveraging long-lived containers re-used to process multiple units. Furthermore, it extends the ideas behind isolating processing steps in separate containers to address 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, and lightweight). The communication between components is enabled by an efficient and contract-based remote procedure call framework. Finally, a set of experiments are executed to compare the proposed solution against Argo workflows and to study individual aspects in isolation.

Overall, the proposed solution improved the performance and reduced bandwidth usage for container-centric big data workflows while maintaining a good separation of concerns and reducing complexity for the framework's users. The optimisations proved the solution better suited for environments where cloud and edge resources are leveraged together. Separating the interaction with data in a dedicated component, thus advancing toward a better separation of concerns, also facilitated the implementation of data locality as a built-in feature of the framework. This further proves the usefulness of creating small and isolated components in a distributed system. The proposed solution is far from feature-parity with the other existing solutions. However, it highlights potential areas of improvement that do not deviate significantly from the fundamentals of the existing architectures, making it possible to integrate the presented ideas and findings into them. While limited in scope, the conducted experiments are representative of the challenges the solution attempts to address. The experiments also highlighted that the benefit provided by the solution could vary significantly based on different factors. The instrumentation provided with the software solution can facilitate the same kind of performance analysis under the conditions relevant for each use case.

Future work includes extending the solution in multiple directions: (i) Simple workflow definitions were supported in the current approach to investigate potential performance improvements, but most real use cases require complex constructs. In this context, 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. (ii) 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 such components. (iii) A central piece of the proposed solution is the routing algorithm that considers load and data locality. Further improving the heuristic and adding more dimensions (such as matching node capabilities) can lead to better results. (iv) The solution could be extended to support stream processing; in addition, avoiding using a disk to transfer the data to be processed is also desirable in such cases. (v) Finally, there are two possible scaling bottlenecks in the proposal. First, the platform could benefit from creating and deleting processing steps dynamically to meet the current demands. Second, the orchestrator and the data master components are single instance applications, becoming a bottleneck under a high load and needing horizontal scaling.

Author Contributions: Funding acquisition, D.R., N.N., M.M.; supervision, D.R., N.N., A.S, M.M., A.H.P.; conceptualisation, A.-A.C., N.N., A.Q.K., A.S., M.M., D.R.; Methodology, all; Software, A.-A.C.; investigation, A.-A.C., N.N., A.Q.K.; Writing—original draft, all; Writing—review & editing, all. All authors have read and agreed to the published version of the manuscript.

Funding: The work in this paper was partly funded by the EC H2020 project "DataCloud " (grant number 101016835) and the NFR project "BigDataMine" (grant number 309691).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DAG	Directed acyclic graph
HDFS	Hadoop file system
IoT	Internet of things
RDMA	Remote direct memory access

References

1. Ashabi, A.; Sahibuddin, S.B.; Haghighi, M.S. Big Data: Current Challenges and Future Scope. In Proceedings of the IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE 2020), Penang, Malaysia, 18–19 April 2020; pp. 131–134. [CrossRef]
2. Ranjan, R.; Garg, S.; Khoskbar, A.; Solaiman, E.; James, P.; Georgakopoulos, D. Orchestrating BigData Analysis Workflows. *IEEE Cloud Comput.* **2017**, *4*, 20–28. [CrossRef]
3. Barika, M.; Garg, S.; Zomaya, A.Y.; Wang, L.; Moorsel, A.V.; Ranjan, R. Orchestrating Big Data Analysis Workflows in the Cloud: Research Challenges, Survey, and Future Directions. *ACM Comput. Surv.* **2019**, *52*, 95:1–95:41. [CrossRef]
4. Zhou, B.; Svetashova, Y.; Pychynski, T.; Baimuratov, I.; Soylu, A.; Kharlamov, E. SemFE: Facilitating ML Pipeline Development with Semantics. In Proceedings of the 29th ACM International Conference on Information & Knowledge Management (CIKM 2020), Online, 19–23 October 2020; pp. 3489–3492. [CrossRef]
5. Baker, T.; Ugljanin, E.; Faci, N.; Sellami, M.; Maamar, Z.; Kajan, E. Everything as a resource: Foundations and illustration through Internet-of-things. *Comput. Ind.* **2018**, *94*, 62–74. [CrossRef]
6. Maamar, Z.; Cheikhrouhou, S.; Asim, M.; Qamar, A.; Baker, T.; Ugljanin, E. Towards a Resource-aware Thing Composition Approach. In Proceedings of the 17th International Conference on High Performance Computing & Simulation (HPCS 2019), Dublin, Ireland, 15–19 July 2019; pp. 803–809. [CrossRef]
7. Kimovski, D.; Matha, R.; Hammer, J.; Mehran, N.; Hellwagner, H.; Prodan, R. Cloud, Fog or Edge: Where to Compute? *IEEE Internet Comput.* **2021**, *25*, 30–36. [CrossRef]
8. Khan, W.Z.; Ahmed, E.; Hakak, S.; Yaqoob, I.; Ahmed, A. Edge computing: A survey. *Future Gener. Comput. Syst.* **2019**, *97*, 219–235. [CrossRef]
9. Corodescu, A.A.; Nikolov, N.; Khan, A.Q.; Soylu, A.; Matskin, M.; Payberah, A.H.; Roman, D. Locality-Aware Workflow Orchestration for Big Data. In Proceedings of the 13th International Conference on Management of Digital EcoSystems (MEDES'21), Hammamet, Tunisia, 1–3 November 2021; pp. 62–70. [CrossRef]
10. Roman, D.; Alexiev, V.; Paniagua, J.; Elvesæter, B.; von Zernichow, B.M.; Soylu, A.; Simeonov, B.; Taggart, C. The euBusinessGraph ontology: A&nbsp;lightweight ontology for harmonizing basic company information. *Semant. Web* **2021**, 1–28. in press. [CrossRef]
11. Soylu, A.; Corcho, O.; Elvesæter, B.; Badenes-Olmedo, C.; Blount, T.; Yedro Martínez, F.; Kovacic, M.; Posinkovic, M.; Makgill, I.; Taggart, C.; et al. TheyBuyForYou platform and knowledge graph: Expanding horizons in public procurement with open linked data. *Semant. Web* **2021**, 1–27. in press. [CrossRef]
12. Nikolov, N.; Dessalk, Y.D.; Khan, A.Q.; Soylu, A.; Matskin, M.; Payberah, A.H.; Roman, D. Conceptualization and scalable execution of big data workflows using domain-specific languages and software containers. *Internet Things* **2021**, 100440, in press. [CrossRef]
13. Balouek-Thomert, D.; Renart, E.G.; Zamani, A.R.; Simonet, A.; Parashar, M. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *Int. J. High Perform. Comput. Appl.* **2019**, *33*, 1159–1174. [CrossRef]
14. Hao, Z.; Novak, E.; Yi, S.; Li, Q. Challenges and Software Architecture for Fog Computing. *IEEE Internet Comput.* **2017**, *21*, 44–53. [CrossRef]
15. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84. [CrossRef]
16. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and Linux containers. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2015), Philadelphia, PA, USA, 29–31 March 2015; pp. 171–172. [CrossRef]
17. Pahl, C.; Brogi, A.; Soldani, J.; Jamshidi, P. Cloud Container Technologies: A State-of-the-Art Review. *IEEE Trans. Cloud Comput.* **2017**, *7*, 677–692. [CrossRef]
18. Kratzke, N.; Quint, P.C. Understanding cloud-native applications after 10 years of cloud computing—A systematic mapping study. *J. Syst. Softw.* **2017**, *126*, 1–16. [CrossRef]
19. Celesti, A.; Mulfari, D.; Fazio, M.; Villari, M.; Puliafito, A. Exploring Container Virtualization in IoT Clouds. In Proceedings of the IEEE International Conference on Smart Computing (SMARTCOMP 2016), St. Louis, MO, USA, 18–20 May 2016. [CrossRef]
20. Bellavista, P.; Zanni, A. Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi. In Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN 2017), Hyderabad, India, 5–7 January 2017; pp. 1–10. [CrossRef]

21. Ismail, B.I.; Goortani, E.M.; Karim, M.B.A.; Tat, W.M.; Setapa, S.; Luke, J.Y.; Hoe, O.H. Evaluation of Docker as Edge computing platform. In Proceedings of the IEEE Conference on Open Systems (ICOS 2015), Melaka, Malaysia, 24–26 August 2015; pp. 130–135. [CrossRef]
22. Bhimani, J.; Yang, Z.; Leeser, M.; Mi, N. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. In Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC 2017), Waltham, MA, USA, 12–14 September 2017; pp. 1–7. [CrossRef]
23. Sill, A. The Design and Architecture of Microservices. *IEEE Cloud Comput.* **2016**, *3*, 76–80. [CrossRef]
24. Linthicum, D.S. Practical Use of Microservices in Moving Workloads to the Cloud. *IEEE Cloud Comput.* **2016**, *3*, 6–9. [CrossRef]
25. Wang, J.; Han, D.; Yin, J.; Zhou, X.; Jiang, C. ODDS: Optimizing Data-Locality Access for Scientific Data Analysis. *IEEE Trans. Cloud Comput.* **2020**, *8*, 220–231. [CrossRef]
26. Youmin, C.; Youyou, L.; Shengmei, L.; Jiwu, S. Survey on RDMA-Based Distributed Storage Systems. *J. Comput. Res. Dev.* **2019**, *56*, 227. [CrossRef]
27. Elshater, Y.; Martin, P.; Rope, D.; McRoberts, M.; Statchuk, C. A Study of Data Locality in YARN. In Proceedings of the IEEE International Conference on Big Data (Big Data 2015), New York, NY, USA, 27 June–2 July 2015; pp. 174–181. [CrossRef]
28. Renner, T.; Thamsen, L.; Kao, O. CoLoc: Distributed data and container colocation for data-intensive applications. In Proceedings of the IEEE International Conference on Big Data (Big Data 2016), Washington, DC, USA, 5–8 December 2016; pp. 3008–3015. [CrossRef]
29. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster Computing with Working Sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud 2010) USENIX, Boston, MA, USA, 22–25 June 2010; pp. 1–7.
30. Naik, N.S.; Negi, A.; BR, T.B.; Anitha, R. A data locality based scheduler to enhance MapReduce performance in heterogeneous environments. *Future Gener. Comput. Syst.* **2019**, *90*, 423–434. [CrossRef]
31. Zhao, D.; Mohamed, M.; Ludwig, H. Locality-Aware Scheduling for Containers in Cloud Computing. *IEEE Trans. Cloud Comput.* **2020**, *8*, 635–646. [CrossRef]
32. Bourhim, E.H.; Elbiaze, H.; Dieye, M. Inter-container Communication Aware Container Placement in Fog Computing. In Proceedings of the 15th International Conference on Network and Service Management (CNSM 2019), Halifax, NS, Canada, 21–25 October 2019; pp. 1–6. [CrossRef]
33. Abranches, M.; Goodarzy, S.; Nazari, M.; Mishra, S.; Keller, E. Shimmy: Shared Memory Channels for High Performance Inter-Container Communication. In Proceedings of the Workshop on Hot Topics in Edge Computing (HotEdge 2019) USENIX, Renton, WA, USA, 9 July 2019; pp. 1–7.
34. Zheng, C.; Thain, D. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC 2015), Portland, OR, USA, 15 June 2015; pp. 31–38. [CrossRef]
35. Hayot-Sasson, V.; Brown, S.T.; Glatard, T. Performance Evaluation of Big Data Processing Strategies for Neuroimaging. In Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2019), Larnaca, Cyprus, 14–17 May 2019; pp. 449–458. [CrossRef]
36. Hsu, C.H.; Slagter, K.D.; Chung, Y.C. Locality and loading aware virtual machine mapping techniques for optimizing communications in MapReduce applications. *Future Gener. Comput. Syst.* **2015**, *53*, 43–54. [CrossRef]
37. Ernstsson, A.; Kessler, C. Extending smart containers for data locality-aware skeleton programming. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e5003. [CrossRef]
38. Bu, X.; Rao, J.; Xu, C.Z. Interference and locality-aware task scheduling for MapReduce applications in virtual clusters. In Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC 2013), New York, NY, USA, 17–21 June 2013; pp. 227–238. [CrossRef]
39. Choi, J.; Adufu, T.; Kim, Y. Data-locality aware scientific workflow scheduling methods in HPC cloud environments. *Int. J. Parallel Program.* **2017**, *45*, 1128–1141. [CrossRef]
40. Ahlehagh, H.; Dey, S. Video-aware scheduling and caching in the radio access network. *IEEE/ACM Trans. Netw.* **2014**, *22*, 1444–1462. [CrossRef]
41. Gu, J.; Wang, W.; Huang, A.; Shan, H. Proactive storage at caching-enable base stations in cellular networks. In Proceedings of the 24th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC 2013), London, UK, 8–11 September 2013; pp. 1543–1547. [CrossRef]
42. Vengadeswaran, S.; Balasundaram, S. An optimal data placement strategy for improving system performance of massive data applications using graph clustering. *Int. J. Ambient Comput. Intell. (IJACI)* **2018**, *9*, 15–30. [CrossRef]
43. Mölder, F.; Jablonski, K.P.; Letcher, B.; Hall, M.B.; Tomkins-Tinch, C.H.; Sochat, V.; Forster, J.; Lee, S.; Twardziok, S.O.; Kanitz, A.; et al. Sustainable data analysis with Snakemake. *F1000Research* **2021**, *10*, 33. [CrossRef]
44. Albrecht, M.; Donnelly, P.; Bui, P.; Thain, D. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET 2012), Scottsdale, AZ, USA, 20 May 2012; pp. 1–13. [CrossRef]

45. Dessalk, Y.D.; Nikolov, N.; Matskin, M.; Soylu, A.; Roman, D. Scalable Execution of Big Data Workflows using Software Containers. In Proceedings of the 12th International Conference on Management of Digital EcoSystems (MEDES 2020), Online, 2–4 November 2020; pp. 76–83. [CrossRef]
46. Mitchell, R.; Pottier, L.; Jacobs, S.; Silva, R.F.d.; Rynge, M.; Vahi, K.; Deelman, E. Exploration of Workflow Management Systems Emerging Features from Users Perspectives. In Proceedings of the IEEE International Conference on Big Data (Big Data 2019), Los Angeles, CA, USA, 9–12 December 2019; pp. 4537–4544. [CrossRef]
47. Martin, P. Multi-container Pod Design Patterns. In *Kubernetes: Preparing for the CKA and CKAD Certifications*; Apress: Berkeley, CA, USA, 2021; pp. 169–173. [CrossRef]