PaaS-BDP
A Multi-Cloud Architectural Pattern for Big Data Processing on a Platform-as-a-Service Model

Thalita Vergilio and Muthu Ramachandran
School of Computing, Creative Technologies and Engineering, Leeds Beckett University, Leeds, U.K.

Keywords: Big Data, Containers, Resource Pooling, Docker Swarm, Orchestration, Multi-Cloud, PaaS.

Abstract: This paper presents a contribution to the fields of Big Data Analytics and Software Architecture, namely an emerging and unifying architectural pattern for big data processing in the cloud from a cloud consumer's perspective. PaaS-BDP (Platform-as-a-Service for Big Data) is an architectural pattern based on resource pooling and the use of a unified programming model for building big data processing pipelines capable of processing both batch and stream data. It uses container cluster technology on a PaaS service model to overcome common shortfalls of current big data solutions offered by major cloud providers such as low portability, lack of interoperability and the risk of vendor lock-in.

1 INTRODUCTION

Big data is an area of technological research which has been receiving increased attention in recent years. As the Internet of Things (IoT) expands to different spheres of human life, a large volume of structured, semi-structured and unstructured data is generated at very high velocity. To derive value from big data, businesses and organisations need to detect patterns and trends in historical data. They also need to receive, process and analyse streaming data in real-time, or close to real-time, a challenge which current technologies and traditional system architectures find difficult to meet.

Cloud computing has also been attracting growing interest lately. With different service models available such as infrastructure as-a-service (IaaS), platform as-a-service (PaaS) and software as-a-service (SaaS), it is no longer essential that companies host their IT infrastructure on-premises. Consequently, an increasing number of small and medium-sized enterprises (SME) has ventured into big data analytics utilising powerful computing resources, previously unavailable to them, without having to procure their own hardware or maintain an in-house team of highly skilled IT professionals. The popularisation of cloud computing is not without its challenges, particularly when it comes to guaranteeing the portability and interoperability of components developed, thus preventing the risk of vendor lock-in. The solution presented in this paper is an answer to these challenges.

2 MOTIVATION

The plethora of technologies currently being used for Big Data processing, and the lack of a systematic, unified approach to processing big data in the cloud is a motivation for this research. There is no single accepted solution to cater for all types of big data, so various technologies tend to be used in combination. Consequently, the learning curve for a developer working with big data is steep, and the processing logic developed within one system is generally incompatible with other systems, leading to code duplication and low maintainability.

The aim of this research is to produce a systematic and unified approach to developing portable and interoperable Big Data processing services on a multi-cloud PaaS service model. PaaS-BDP is based on a programming model applicable to both stream and batch data, thus eliminating the need for the Lambda Architecture where multiple technologies are used in combination.

Vergilio, T. and Ramachandran, M.
PaaS-BDP.
DOI: 10.5220/0006632400450052
In *Proceedings of the 3rd International Conference on Complexity, Future Information Systems and Risk (COMPLEXIS 2018)*, pages 45-52
ISBN: 978-989-758-297-4

3 RELATED WORK

This research proposes a solution to the vendor lock-in aspects of low portability and lack of interoperability affecting existing big data processing offerings in the cloud. Solutions to the vendor lock-in issue encountered in the literature can be categorised as follows:

3.1 Standardisation

Standardisation of cloud resource offerings is a way of dealing with the vendor lock-in issue. However, no universal set of standards has yet been identified which would successfully solve the issues of portability and interoperability between different cloud providers (Martino, 2014), and the standards that do exist have not been widely adopted by the industry (Guillén et al., 2013).

3.2 Cloud Federations

Another alternative solution to the vendor lock-in issue is the establishment of cloud federations (Kogias et al., 2016). In a cloud federation, providers voluntarily agree to participate and are bound by rules and regulations. This however places the focus on the cloud provider, rather than on the consumer of cloud services. As this research approaches the vendor lock-in issue from the cloud consumer's perspective, cloud federations are excluded from its scope.

3.3 Middleware

The introduction of a layer of abstraction to enable distribution and interoperability between different cloud providers has also been proposed as a possible solution to the cloud lock-in problem (Guillén et al., 2013), (Silva et al., 2013). One such model, called Neo-Metropolis, was proposed by H. Chen et al. (Chen et al., 2016). This model is based on a kernel, which provides the platform's basic functionality, a periphery, composed of various service providers hosted on different clouds, and an edge, representing customers who utilise services and provide requirements (Chen et al., 2016). Whilst the kernel would be fairly stable and backwards compatible, with stable releases, the periphery would be in constant development, or perpetual beta, and would be based on open-source code (Chen et al., 2016).

One criticism to this type of approach, however, is that the lock-in problem is not resolved, it is simply shifted to the enabling middleware layer (Guillén et al., 2013).

3.4 Unified Models

A model-driven approach to development, combined with a unifying framework for modelling cloud artefacts, has been suggested as a possible solution to the vendor lock-in problem. In fact, the "model once, generate everywhere" precept of MDA (Model Driven Architecture) suggests that software can be cloud platform-agnostic, provided that the necessary code generating engines are in place (Martino, 2014). In reality, however, it is difficult to find concrete examples of perfectly accurate code generation engines capable of producing all of the source code exclusively from the models (Guillén et al., 2013).

MULTICLAPP is an architectural framework that separates the application design from cloud provider-specific deployment configuration. Application modelling is done using an extended UML profile. The models are then processed by a Model Transformation Engine, responsible for inserting cloud provider-specific configuration and generating class skeletons (Guillén et al., 2013). Although this approach ensures the perpetuation of the models in case of cloud provider migration, application implementation code would still need to be re-written.

3.5 Virtualisation

The use of containers or hypervisor technology (virtual machine managers) to deploy software in the cloud is a pattern which minimises the effects of vendor lock-in, as the environment configuration and requirements are packaged together with the deployed application.

3.5.1 Virtual Machines

The use of VMs to deploy applications is generally associated with the IaaS cloud service model. Together with the code for the developed application, a VM also contains an entire operating system configured to run that code. Since containers are lighter and easier to deploy and maintain than VMs, this research advocates the use of container technology in its proposed architecture.

3.5.2 Containers

Containers are a lighter alternative to VMs (Bernstein, 2014). They have gained increased popularity recently, following the open-sourcing of the most widely-accepted technology, Docker, in March 2013 (Miell & Sayers, 2015).

The benefits of using containers become more apparent when it comes to implementing distributed architectures (Bernstein, 2014), as their small size and relative ease of deployment allow for better elasticity across different clouds. Docker is based on the Linux operating system, which is a good fit for the cloud as it is reliable, has a wide user base, and allows containers to scale up without incurring additional licensing costs (Celesti et al., 2016).

This research embraces the emerging trend towards containerisation as it recognises the benefits of using a multi-cloud environment for the deployment of distributed big data processing frameworks.

4 PROPOSED SOLUTION

The proposed solution is an architectural pattern for big data processing using frameworks and containers on a PaaS service model.

4.1 Conceptual Elements

4.1.1 Framework

The framework takes care of parallelising the data processing, scheduling work between the processing units and ensuring fault tolerance. In traditional, on-premises implementations, the framework code is generally downloaded and unpacked in each participating machine. A number of setup steps are then completed to integrate each machine into the cluster as a worker. As this process is executed within each participating machine, usually by entering commands on a terminal, it is prone to failure due to differences between environments or human error. The architectural pattern proposed in this section presents a solution to this problem.

4.1.2 Image

An image specifies how to build/get an application, its runtime environment and dependencies and execute it in a container. It is abstract, whereas a container is concrete. Many identical containers can be created from a single image, which makes them a good choice of technology for exploring the elasticity of the cloud when building distributed systems. In a similar way in which a class is used to instantiate an object in object-oriented programming, an image is used to instantiate containers in container-based implementations.

Images are stored in a registry, which can be private or public, and downloaded when needed. Registries enable version control and promote code sharing and reuse.

4.1.3 Container

Containers are lightweight runtime environments deployed to virtual or physical machines. Each machine can have several containers running in it. They share the same operating system, but are otherwise separate deployment environments.

4.1.4 Machine

A bare-metal or virtual machine can have a number of containers running on it. They can be based on-premises or in the cloud, with the latter generally exhibiting greater elasticity. AWS, for example, allows vertical scaling of their virtual machines through re-sizing, which involves selecting a more powerful configuration from the offers available (Resizing Your Instance - Amazon Elastic Compute Cloud, 2017).

4.2 Resource Sharing

The new architectural pattern proposed in this research decouples the physical deployment environment, i.e. machines, from the artifacts that are deployed to them and ultimately the frameworks that own the artifacts. Instead of having dedicated machines for Hadoop, Spark, etc, these frameworks share a pool of resources and take or drop them as needed. Increased utilisation and improved access to data sharing have been highlighted in the literature as advantages associated with pooling resources between big data frameworks (Hindman et al., 2011). In fact, these factors are particularly relevant in the context of cloud-based architectures, where costs are transparent and changes are immediately visible. If we take, for example, a multi-cloud setup where resources are fluid and vendor lock-in is negligible, it is possible to scale up using whichever provider is most suitable at the time, or even replace providers without detrimentally affecting the system.

The existence of mixed big data packages, such as the Hadoop Ecosystem, suggests that there is no de-facto big data technology to cater for all different needs and scenarios. Instead, organisations tend to utilise more than one framework concurrently. This is another strong argument for choosing an architecture which allows resources to be pooled and shared between frameworks.

Fig.1 illustrates how the proposed architectural pattern decouples frameworks from machines by introducing a new abstraction: containers. From a machine's perspective, it runs containers. A machine is unaware of which frameworks, if any, are associated with the containers running on it. Specific environment configuration is defined at container level, leaving the machine itself generic and agnostic. The framework, on the other hand, knows nothing about the specific machines on which their workers and managers run. It does know which containerised workers and managers are part of the cluster at a given time, and their corresponding states, but it has no knowledge of machines and their configurations.

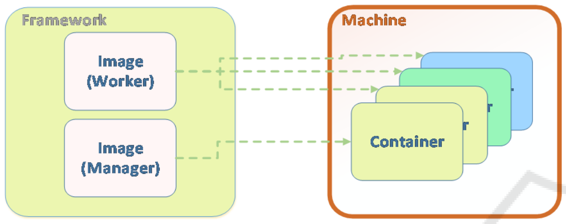

Figure 1: Container-Based Big Data Processing Deployment.

Fig.2 illustrates how the proposed architecture scales up. Different big data frameworks are maintained concurrently, as are different sets of machines hosted in different locations. More machines can be added to the cluster to scale the system vertically. Likewise, more containers can be created from a worker image and deployed to the cluster if a particular job executed by a framework needs to be scaled horizontally.

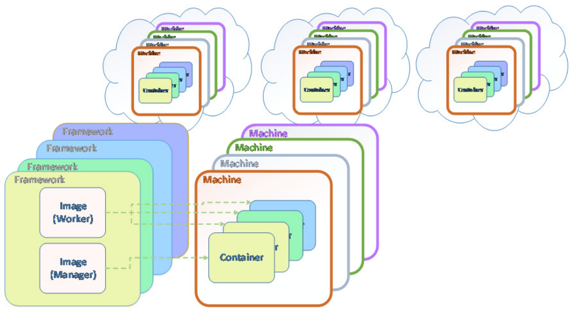

Figure 2: Container-Based Big Data Processing Deployment on a PaaS Service Model.

4.3 Programming the Big Data Processing Pipeline

This section decouples the big data processing pipeline code from the framework under which it ultimately runs. When the concept of a processing pipeline is abstracted as a series of operations, defined by business needs, performed on units of data, we find no reason to believe it could not be written in a framework-agnostic way. This avoids duplication when different frameworks are used and enables the portability of the developed artifact between frameworks.

4.3.1 Different Programming Models

Many big data frameworks claim that any programming language can be used to define their processing pipelines, e.g. (Apache Storm - Project Information, n.d.), (MapReduce Tutorial, 2013). This section shall demonstrate that the main issue affecting the portability and interoperability of artifacts produced for a given framework is not to do with the programming language, but with the abstractions and programming model to which a developer must adhere. These tend to be specific to each framework and not easily translatable between them. A simple visual example of a classifier and counter implementation is used to illustrate this point.

Imagine a scenario where there are various green and yellow circles, and a business need to count how many circles there are of each colour. This section compares two different approaches to implementing a solution: one using MapReduce, a popular algorithm for distributed parallel processing of batch files, and another using a topology of spouts and bolts, abstractions provided by the Storm framework. Fig.3 presents a MapReduce-based solution. A mapping function is first applied to each element of each data set. It accepts coloured circles and outputs numbered coloured circles which, in code, would be represented as key/value pairs where the key is the colour and the value is the number. The reduce function then takes a set of values (all the circles where the colour is yellow, or all the circles where the

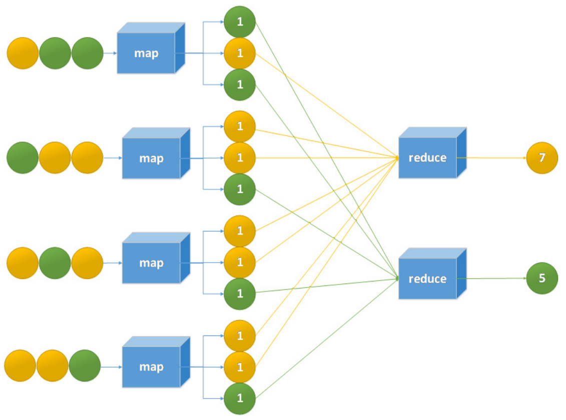

Figure 3: The Map-Reduce Algorithm.

colour is green) and performs a reduction operation, i.e. transforms them into a smaller set of values. In this case, it outputs one element, a circle where the number is the sum of all the other numbers in the set. The result could also be represented as a key/value pair where the key is the colour or the circle and the value is the number it displays.

Storm, a framework mainly designed for stream processing, uses a different type of abstraction from those used by MapReduce, namely spouts and bolts. Spouts represent sources of streaming data, whilst bolts represent transformations applied to them. Because streaming data is infinite, a similar exercise of counting circles by colour only makes sense if time is taken into consideration, not only in the visual representation, but also in the code implementation of a possible solution. Fig. 4 represents a stream-based approach to the circle count exercise. Using spouts and bolts, it processes each element it sees in real-time and updates a counter. Because the source of data is infinite, the processing of the data is also infinite, and the results are never final.

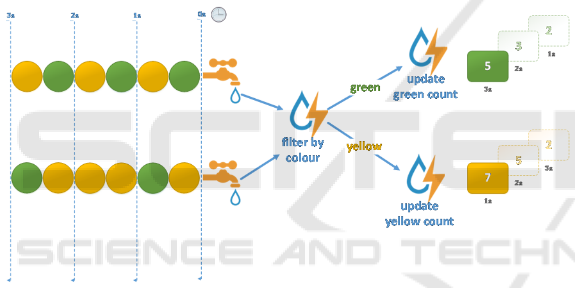

Figure 4: Spouts and Bolts Representing Stream Data Processing.

Having examined two very simple examples where the same problem is solved using different frameworks and different approaches to big data processing, it becomes apparent that the lack of portability or interoperability between artifacts produced for different frameworks is a complex issue that goes beyond the simple translation of one library into another. The abstractions upon which these frameworks are built are fundamentally diverse, one of the reasons why they generally provide their own libraries.

In this section, a simple example of a counter for different coloured circles was used to obtain an insight into how big data frameworks use different abstractions and a different programming model to implement solutions to the same problem. In particular, batch and stream architectures appear to differ fundamentally due to the limited or unlimited nature of the data source. The Lambda Architecture, developed as an answer to this predicament, never did circumvent the inconvenience of developers having to maintain different pieces of code, containing the same business logic, in different places, only because the incoming data is, in some cases, limited and, in others, unlimited. Section 4.3.3 looks at how this dichotomy has been broken and explores the benefits associated with using a unified programming model with different big data frameworks. Before that, however, the following section takes a closer look at the traditional scenario of software development using different frameworks and different programming models.

4.3.2 Framework-Specific Programming

Framework-specific programming is hereby defined as writing software code which is intended to be executed from within a given framework. In a scenario where multiple frameworks are used, artifacts produced for each framework exist independently and are maintained independently throughout their existences. If a business case arises to duplicate the functionality developed within one framework onto a different framework, it is usually the case that new code will need to be written, as the conceptual model and abstractions used in the implementation will be incompatible. Fig. 5 illustrates the process of programming for specific frameworks.

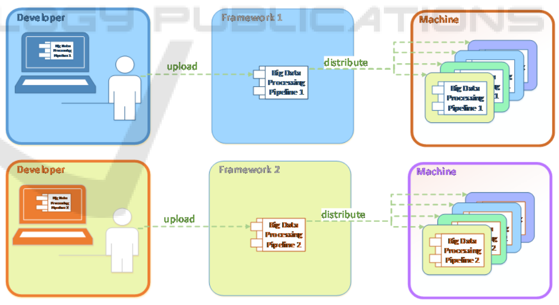

Figure 5: Framework-Specific Programming.

4.3.3 Framework-Agnostic Programming

As seen in the previous section, in framework-specific programming, developers use specialised libraries provided by each framework. The processing code written is therefore only compatible with a particular framework, as are the artifacts produced. The necessary conditions to enable framework-agnostic programming can be understood by once again referring to the process in Fig. 6. If, instead of using a framework-provided library, developers used a common library compatible with the main big data frameworks, they would be able to write processing code in a framework-agnostic way, and to produce artifacts compatible with multiple frameworks. The Apache Beam SDK (Apache Beam, 2017) comes very close to being such a library. The issue of fundamental differences between programming models for batch and stream is resolved by treating all data as if it were streaming. Streaming data, due to its unbounded nature, is divided into bounded subsets called windows, which are finite and can be processed one at a time. The same approach is used for the processing of batch data: even though the data is finite, it could be so large that, for processing purposes, it makes sense to treat it as infinite. Large sources of batch data would therefore be divided into windows and processed one subset at a time, as if they were streaming. Fig. 6 and 7 illustrate the windowing of batch data so the same programming model is applied for both batch and stream data.

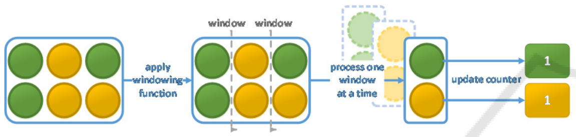

Figure 6: Batch Processing using a Stream-Based Programming Model.

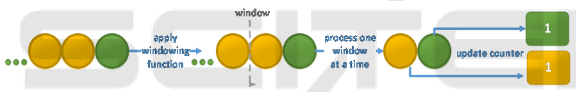

Figure 7: Stream Processing using a Stream-Based Programming Model.

The benefits of decoupling the programming model from specific big data frameworks include less code duplication, increased reusability and maintainability of artifacts produced, and a shallower learning curve for developers wanting to work with big data. Fig. 8 illustrates the framework-agnostic programming model. Note how the developer only produces one artifact, which is then uploaded to different frameworks, to be processed using their respective resources.

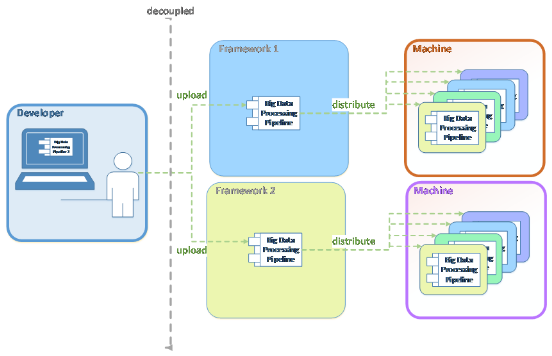

Figure 8: Framework-Agnostic Programming.

4.3.4 Framework-Agnostic Programming with Pooled Resources

This section aims to amalgamate the framework agnostic programming model described in the previous section with the container-based architectural pattern proposed earlier. It demonstrates how decoupling artifacts from frameworks and from the machines that run them leads to less duplication, higher maintainability of code, as well as easier, simpler and more effective management of machine clusters.

Fig. 9 illustrates the framework-agnostic model with pooled resources. Big data processing pipelines are developed once per business case, instead of one per framework. Once the artifact is ready to be released into production, it can be uploaded to and executed by any compatible big data framework. Because the programming model used is stream-based, the big data framework must be able to execute stream pipelines. This is one of the limitations of the model. However, should users of major batch processing frameworks such as Hadoop wish to adopt the proposed unified model, they could do so with minimal impact and without the need for migration by adopting a parallel transition strategy over a long period of time (Okrent & Vokurka, 2004). Since HDFS files can be used as data sources in the proposed model, new processing pipelines can be developed using the unified model and run by a stream engine without affecting the existing code developed to run in Hadoop.

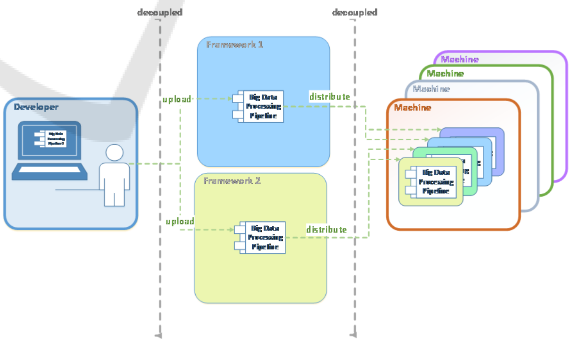

Figure 9: Framework-Agnostic Programming with Pooled Resources.

The second decoupling line in Fig. 10 shows how resources can be pooled and shared by different frameworks. Frameworks have a number of runners or workers responsible for the parallel execution of data processing pipelines. These workers are typically deployed to clusters of machines on a one cluster per framework basis, as shown in Fig. 8. This represents a potential waste of resources, magnified in a cloud scenario where machines are charged on a per-minute base. The chance of charges being incurred for machines which are idle is higher, since a machine commissioned for a given framework's cluster cannot be immediately utilised by a different framework. The proposed model solves this issue by allowing different frameworks to share the same cluster. Runners belonging to different frameworks are deployed to machines as containers. Because machines only execute containers and know nothing about which framework, if any, the containers belong to, they become framework-independent and can be shared between several of them.

This section presented the full proposed model for big data processing in the cloud. It discussed the advantages of utilising a unified programming model and a container-based architecture with pooled resources for cases where different big data frameworks are used simultaneously.

5 EVALUATION

The current section presents preliminary results for an initial experiment which consisted of setting up and configuring a multi-cloud containerised environment on a PaaS service model. A total of 12 virtual machines were commissioned from Microsoft Azure, Google Cloud and the Open Science Data Cloud (OSDC), as illustrated in Table 1. Although every attempt was made to commission identical machines to operate as workers, lack of standardisation amongst cloud providers led to our nodes being slightly, although not significantly different.

Docker was used for containerisation, and Docker Swarm for orchestration. The nodes were networked across clouds using the Weave Net Docker plugin (Weave Net, 2017). The Apache Beam SDK was used to program the big data processing pipeline, since it provides a unifying programming model for both batch and stream data. Apache Flink was selected as a runner since, at the time of writing, it provided the widest range of capabilities from the open-source technologies supported by Apache Beam (Apache Beam Capability Matrix, n.d.).

At this initial stage, we successfully verified that the proposed architecture is feasible and that the job of processing incoming streaming data is seamlessly parallelisable across different clouds. We also verified that the proposed architecture is horizontally and vertically scalable by increasing the number of containers running data processing jobs, and by adding nodes dynamically to the pool of resources.

Table 1: Multi-Cloud Virtual Machine Specification.

Cloud	Resource Commissioned	RAM	CPU	Disk	Purpose
Azure	Standard DS2 v2 Promo	7GB	2vCPU	30GB	Orchestration
Azure	Standard DS2 v2 Promo	7GB	2vCPU	30GB	Work Parallelisation
Azure	Standard DS2 v2 Promo	7GB	2vCPU	30GB	Worker
Azure	Standard DS2 v2 Promo	7GB	2vCPU	30GB	Worker
Azure	Standard DS2 v2 Promo	7GB	2vCPU	30GB	Worker
Google	n1-standard-2	7.5GB	2vCPU	10GB	Worker
Google	n1-standard-2	7.5GB	2vCPU	10GB	Worker
OSDC	m3.medium	6GB	2vCPU	10GB	Worker
OSDC	m3.medium	6GB	2vCPU	10GB	Worker
OSDC	m3.medium	6GB	2vCPU	10GB	Worker
OSDC	m3.medium	6GB	2vCPU	10GB	Worker
OSDC	ram8.disk10.eph64.core4	8GB	4vCPU	10GB	Messaging

We are currently working on a Case Study which involves developing a real-time Energy Efficiency Analysis service using the multi-cloud big data architecture proposed. This Case Study shall provide us with a solid evaluation of our contribution in a real-world scenario.

6 CONCLUSIONS AND FUTURE WORK

This paper presented a contribution to the fields of Big Data Analytics and Cloud Software Architecture of an emerging and unifying architectural pattern for big data processing in the cloud. This pattern is based on the use of big data frameworks, containers and container orchestration technology for the deployment of big data processing pipelines capable of processing both batch and stream data. We discussed how the issues of low portability and lack of interoperability, identified as common shortcomings of current cloud-based solutions, are overcome by our proposed solution. We expect to complete our Case Study evaluation of the proposed architecture in the coming months.

Furthermore, we envision additional development of the initial proposal to collect metrics related to processing time from a data flow perspective. The aim is to develop a monitoring service to inform cloud consumers of delays in the processing of windows of data, thus highlighting the need to increase processing capacity by scaling the system vertically (i.e. adding more virtual machines to the pool). Correspondingly, the monitoring service would gather information on whether data is waiting too long to be processed, thus suggesting the need to scale the system horizontally (i.e. increase the number of containers running the framework's workers). This monitoring service from a unique data flow perspective is also a contribution to the field, and will be used to gather performance metrics to further evaluate the architectural pattern proposed in this paper.

ACKNOWLEDGEMENTS

This work made use of the Open Science Data Cloud (OSDC) which is an Open Commons Consortium (OCC)-sponsored project.

Cloud computing resources were provided by a Microsoft Azure for Research award.

REFERENCES

Apache Beam (2017) *Apache Beam* [Online]. Available from: <https://beam.apache.org/> [Accessed 28 February 2017].

Apache Beam Capability Matrix (n.d.) *Apache Beam Capability Matrix* [Online]. Available from: <https://beam.apache.org/documentation/runners/capability-matrix/> [Accessed 9 August 2017].

Apache Storm - Project Information (n.d.) *Project Information* [Online]. Available from: <http://storm.apache.org/about/multi-language.html> [Accessed 7 August 2017].

Bernstein, D. (2014) Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1 (3) September, pp. 81–84.

Celesti, A., Mulfari, D., Fazio, M., Villari, M. & Puliafito, A. (2016) Exploring Container Virtualization in IoT Clouds. In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP), May 2016*. pp. 1–6.

Chen, H. M., Kazman, R., Haziyev, S., Kropov, V. & Chtchourov, D. (2016) Big Data as a Service: A Neo-Metropolis Model Approach for Innovation. In: *2016 49th Hawaii International Conference on System Sciences (HICSS), January 2016*. pp. 5458–5467.

Guillén, J., Miranda, J., Murillo, J. M. & Canal, C. (2013) A UML Profile for Modeling Multicloud Applications. In: Lau, K.-K., Lamersdorf, W. & Pimentel, E. ed., *Service-Oriented and Cloud Computing*, *September 11, 2013*. Springer Berlin Heidelberg, pp. 180–187.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S. & Stoica, I. (2011) Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, *2011*. Berkeley, CA, USA: USENIX Association, pp. 295–308.

Kogias, D. G., Xevgenis, M. G. & Patrikakis, C. Z. (2016) Cloud Federation and the Evolution of Cloud Computing. *Computer*, 49 (11) November, pp. 96–99.

MapReduce Tutorial (2013) *MapReduce Tutorial* [Online]. MapReduce Tutorial. Available from: <https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html> [Accessed 7 August 2017].

Martino, B. D. (2014) Applications Portability and Services Interoperability among Multiple Clouds. *IEEE Cloud Computing*, 1 (1) May, pp. 74–77.

Miell, I. & Sayers, A. H. (2015) *Docker in Practice*. Shelter Island, NY: Manning Publications.

Okrent, M. D. & Vokurka, R. J. (2004) Process Mapping in Successful ERP Implementations. *Industrial Management & Data Systems*, 104 (8) October, pp. 637–643.

Resizing Your Instance - Amazon Elastic Compute Cloud (2017) [Online]. Available from: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-resize.html> [Accessed 24 July 2017].

Silva, G. C., Rose, L. M. & Calinescu, R. (2013) A Systematic Review of Cloud Lock-In Solutions. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science, December 2013*. vol. 2. pp. 363–368.

Weave Net (2017) [Online]. Available from: <https://store.docker.com/plugins/weave-net-plugin> [Accessed 20 December 2017].