PaaS-BDP

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

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.

2 MOTIVATION

With the popularisation of the cloud, Big Data analytics is now an accessible service to many SMEs. The top four cloud providers in terms of market share, Amazon, Microsoft, IBM and Google (Synergy Research Group, 2016), offer Big Data as a managed service on a SaaS service model. This model, however, comes with associated risks of low portability and reduced interoperability between the processing components developed.

The plethora of technologies currently being used for Big Data processing, and the lack of a systematic, unified approach to processing pipeline development, is also a motivation for this research. There is no single accepted solution to cater for all types of big data processing, so various technologies tend to be used in combination, as illustrated by the different Apache projects incorporated into what is now known as the Hadoop ecosystem. 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 pipelines 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. The risk of vendor lock-in is reduced by using containers on a PaaS service model, which increases portability, and by using container clustering technology, which provides seamless interoperability amongst different clouds.

3 SCOPE

This research focuses on big data processing, defined as the transformations which takes place after the data has been collected and before it is analysed (see Fig. 1). Storage, which can happen after collection, after processing, after analysing, or at any combination of these stages, is excluded from the scope of this project.

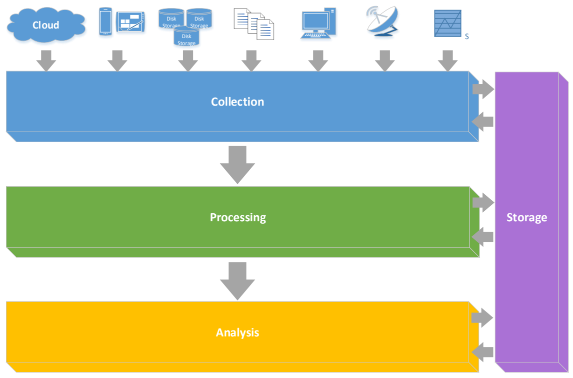

Figure 1: Big Data Architectural Layers

4 RESEARCH CONTEXT

4.1 Batch and Stream Programming

Batch processing was the first and is the most solidly established approach to big data processing. It is based on the MapReduce algorithm and was designed at Google (Dean & Ghemawat, 2008), before going open-source as the Hadoop software framework. Hadoop is a distributed system for processing large volumes of data which is easy to use and extremely powerful (Dean & Ghemawat, 2008). It abstracts the complexities of parallelisation and inter-machine communication away from the user, who only needs to specify the map and reduce functions (Dean & Ghemawat, 2008). Hadoop can handle terabytes of data (O'Malley, 2008), but one of its main criticisms is its high latency and high start-up overhead (Stewart & Singer, 2012), which renders it ineffective for real-time systems.

Stream processing architectures evolved from the need to process real-time data. While batch processing is related to the volume of big data, stream processing relates to velocity. Real-time information such as which topics are trending on Twitter needs to come from data which is constantly being updated, with minimal latency. The notion of a data stream is an abstraction used to convey the nature of the data source: continuous and potentially infinite, and the way in which it is processed: in real-time (or close to real-time), before it is persisted to storage (Zikopoulos et al., 2013, p. 47).

4.2 The Lambda Architecture

The Lambda Architecture was introduced by Marz & Warren as a description of architectures adopting a "best of both" approach to the batch/stream dichotomy. These architectures would use the stream layer for real-time data and the batch layer for historical data. The data would then be merged at query level (Marz & Warren, 2015). Examples of this type of architecture can be found in Twitter's SummingBird (Hausenblas, 2014), Yahoo's Storm-Yarn (Evans & Feng, 2013), IBM's Big Data platform (Zikopoulos et al., 2013), Lambdoop (Tejedor, 2013), AllJoynLambda (Villari et al., 2014) and Facebook's integration between its Puma, Swift and Stylus stream systems and its data stores (Chen, G. J. et al., 2016). The main criticism to this approach is having to maintain two different complex architectures and having duplicate code in the two different layers (Kreps, 2014).

4.3 Programming Language

Big data pipelines must be capable of ingesting and processing both batch and stream data in order to avoid code duplication and enhance maintainability. The Apache Beam project is a pioneering effort which currently fulfils this requirement through its released SDKs for Java and Python. That notwithstanding, other languages/libraries in the future could prove to be just as suitable, or perhaps even more suitable in some circumstances than the ones included in the Apache Beam project.

One fundamental abstraction used to rationalise very large or infinite data inputs is the concept of windows of data. These appear in the stream processing literature as groupings based on event time, stream time, element count or, as is the case with punctuation or frame-based windows, based on some attribute of the elements (Zikopoulos et al., 2012), (Whiteneck et al., 2010). Windows of data can be fixed (e.g. every hour) or sliding (e.g. every hour, starting every minute). Akidau et al. combine these two concepts by defining fixed windows as "a special case of sliding windows where size equals period" (Akidau et al., 2015). Because sliding time windows are a recurrent pattern in stream processing (Khare et al., 2015), their inclusion in a programming language capable of processing different types of big data is to be expected. The adapter used by IBM's big data platform to allow batch data to be processed as a stream, for example, uses the concept of windows of data to perform the conversion (Zikopoulos et al., 2012, p. 128). This concept is also present in the Apache Beam solution, as well as in the Java Stream API, which uses windows based on element count to make an infinite stream finite (Java Platform, Standard Edition 8, 2014). This research uses the concept of windowing to provide a unifying programming model for both batch and stream data.

Java SE could indeed become the de facto language for big data processing, considering that amongst the significant structural enhancement released with JDK 8 was the introduction of a concept of stream to represent a processing pipeline. A Java stream has a data source, which can be a collection, a file or a function, a number of intermediate operations, and a terminal operation. Streams can be processed in parallel by calling a convenient parallel() method on an existing stream. This uses the Fork/Join algorithm to create new threads as needed and process the stream pipeline in parallel. The Fork/Join algorithm is very effective and performs better than Hadoop's MapReduce for data up to around 104.3MB, but fails with an out-of-memory error for larger files (Stewart & Singer, 2012). The JUNIPER project has overcome this limitation by introducing the concept of stored collections. Their rationale is that, if the data does not need to be loaded into memory, the Fork/Join algorithm is adequate and, indeed, at times more powerful than MapReduce for processing big data (Chan et al., 2014). The JUNIPER research tested the performance of Fork/Join vs. MapReduce on a single multicore node. More research is needed to verify how well the standard Fork/Join algorithm utilised by the Java Stream API processes distributed stored collections, and whether it could pose a threat to the hegemony of MapReduce in the big data processing domain.

4.4 Vendor Lock-in

One of the biggest drawbacks of deploying complex applications in the cloud is the risk presented by vendor lock-in (Ardagna et al., 2012), (Guillén et al., 2013), (Silva et al., 2013a), (Silva et al., 2013b), (Kogias et al., 2016), (Assis & Bittencourt, 2016) and others. Assis & Bittencourt define vendor lock-in as: "technical and monetary costs faced by the customer for migrating from one cloud provider to another when some advantage is observed (e.g. more attractive prices)." (Assis & Bittencourt, 2016, p. 55).

This definition emphasises the low portability aspect of vendor lock-in, defined as the ability to transfer an application from the cloud where it is currently deployed to a cloud from a different provider (Silva et al., 2013a). Another important aspect of vendor lock-in highlighted in the literature is its effect on interoperability (Silva et al., 2013a), (Martino, 2014), (Opara-Martins et al., 2016), (Yasrab & Gu, 2016). Interoperability occurs when an application or component deployed to a given cloud server exchanges information harmoniously with applications or components deployed to other cloud servers (Silva et al., 2013a). A distributed architecture where components are hosted by different cloud providers, for example, would rely extensively on some guarantee of interoperability between these providers. Additionally, interoperability between cloud-deployed artefacts and those hosted on-premises must also exist, as companies may choose not to deploy all their resources to the cloud for strategic or security reasons (Opara-Martins et al., 2016).

Vendor lock-in can have serious consequences for small businesses who have fewer spare resources to spend on potential redevelopment of a whole suite of applications following price rises or changes in their cloud provider's offered services (Ardagna et al., 2012). A survey on the effect of vendor lock-in on cloud adoption by UK-based SMEs and larger companies, based on 114 participants, revealed that most decision makers lack in-depth knowledge of the risks associated with vendor lock-in. This could have a significant impact on cloud computing adoption, as organisations are understandably reluctant to embark on business-critical undertakings without a clear exit strategy (Opara-Martins et al., 2016). Additionally, cloud providers generally reserve the right to control their prices, and very few guarantee an expected level of service quality through service level agreements, representing a considerable risk to potential cloud adopters (Satzger et al., 2013).

5 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:

5.1 Standardisation

Standardisation of cloud resource offerings is considered a way of dealing with the vendor lock-in issue. 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).

5.2 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.

5.3 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., 2013a). One such model, called Neo-Metropolis, was proposed by H. Chen et al. (Chen, H. M. 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, H. M. 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, H. M. 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).

5.4 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 was proposed as 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.

5.5 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.

5.5.1 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.

5.5.2 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.

6 PROPOSED SOLUTION

The proposed solution is an architectural pattern for big data processing using frameworks and containers on a PaaS service model. Fig. 2 shows a simplified view of the proposed model.

There are four conceptual elements depicted in Fig. 2: framework, image, container and machine. The next sub-section discusses each of the four conceptual elements in detail.

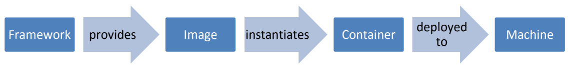

Figure 2: Simplified Architectural Pattern Diagram

6.1 Conceptual Elements

6.1.1 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 (Hadoop Cluster Setup, 2017), (Spark Standalone Mode, n.d.), (Apache Flink 1.3 Documentation: Standalone Cluster, n.d.). 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.

6.1.2 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 can be layered, which means new images can be created from a base image plus additional

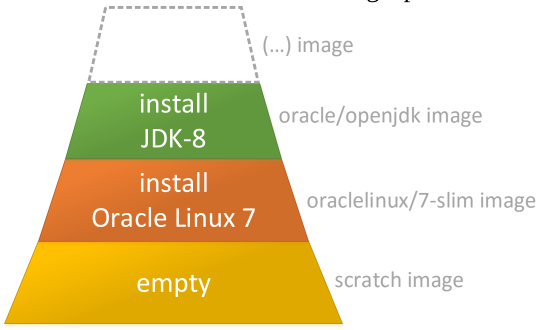

Figure 3: Image Layers

instructions (Pahl & Lee, 2015). The oracle/openjdk image published on DockerHub, for example, is built on top of the oraclelinux/7-slim image and contains additional instructions to download and install JDK-8. The oraclelinux/7-slim image, on the other hand, starts from a base image called scratch (an empty image provided by Docker) and contains instructions to download and install the Oracle Linux 7 operating system (library/oraclelinux - Docker Hub, n.d.). Fig.3 depicts this example and illustrates how images stack up.

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.

6.1.3 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.

6.1.4 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).

6.2 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.4 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.

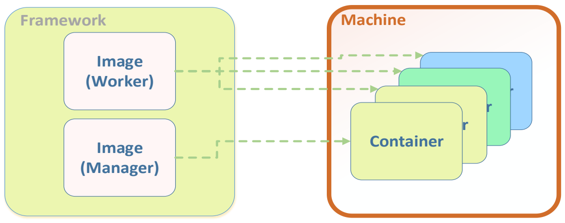

Figure 4: Container-Based Big Data Processing Deployment

Fig.5 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.

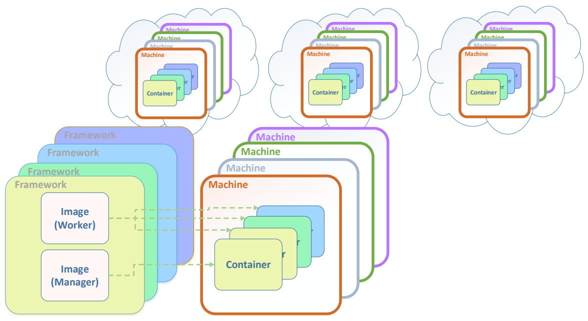

Figure 5: Container-Based Big Data Processing Deployment on a PaaS Service Model

6.3 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.

6.3.1 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.

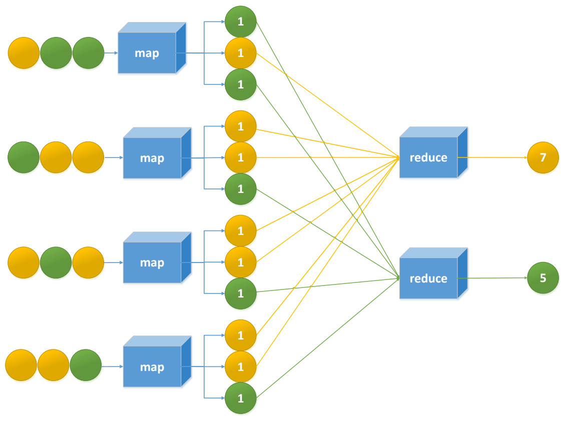

Figure 6: The Map-Reduce Algorithm

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.6 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 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. 7 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.

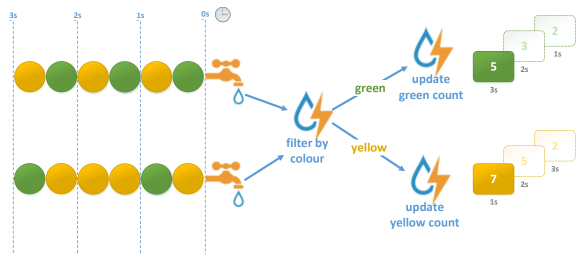

Figure 7: Spouts and Bolts Representing Stream Data Processing

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 6.3.3 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.

6.3.2 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. 8 illustrates the process of programming for specific frameworks.

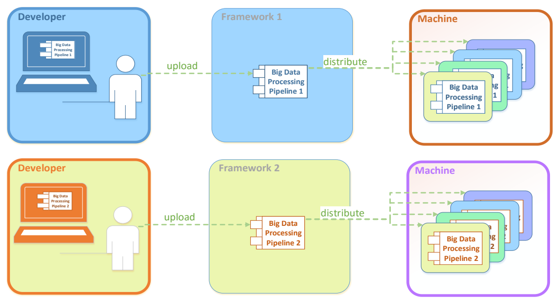

Figure 8: Framework-Specific Programming

6.3.3 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. 8. 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 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. 9 and 10 illustrate the windowing of batch data so the same programming model is applied for both batch and stream data.

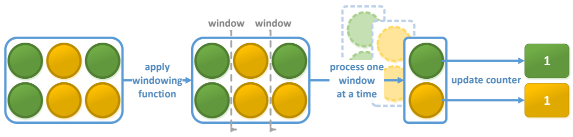

Figure 9: Batch Processing using a Stream-Based Programming Model

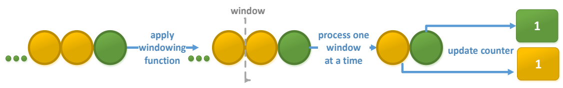

Figure 10: Stream Processing using a Stream-Based Programming Model

Because it uses the same programming model for both batch and stream, the Apache Beam SDK is compatible with a number of big data frameworks. At the time of writing, it can accept incoming data from over 20 different sources, with a number of others still in development. These include distributed batch files from Hadoop, streaming topics from Kafka, and database data from Cassandra or MongoDB (Built-in I/O Transforms, n.d.). When it comes to running the processing pipelines, however, one of five frameworks can currently be used: Flink, Spark, Dataflow, Apex and Gearpump (Apache Beam Capability Matrix, n.d.). The fact that they are all stream-based is not a coincidence, but a limitation which arises from the choice of a stream-based programming model for batch and stream big data processing in Apache Beam. Even though the Beam meta-framework accepts clearly finite batch data such as distributed Hadoop files, the way it handles batch data in the programming layer is as if it were streaming. This is both its main strength and its limitation, as it allows for a common set of abstractions compatible with most big data frameworks, but it limits the running platforms to stream-based frameworks.

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. 11 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.

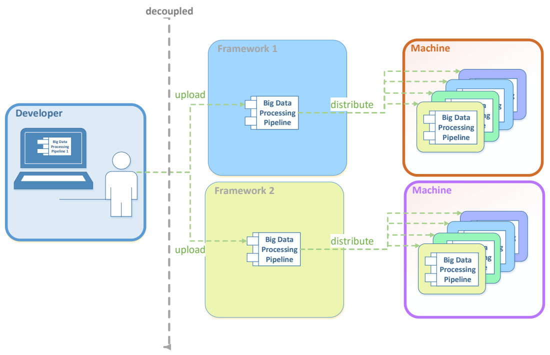

Figure 11: Framework-Agnostic Programming

6.3.4 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. 12 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.

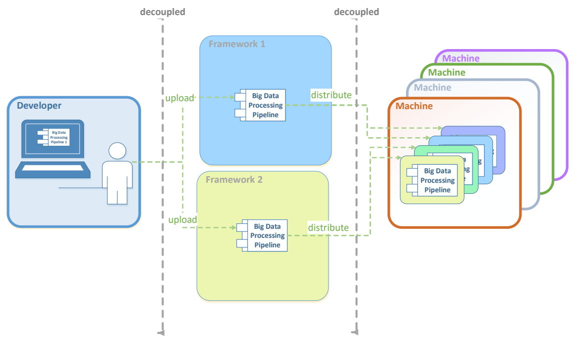
decoupled
decoupled
Developer
Big Data Processing Pipeline 1
upload
upload
Framework 1
Big Data Processing Pipeline
distribute
Framework 2
Big Data Processing Pipeline
distribute
Machine
Machine
Machine
Machine
Big Data Processing Pipeline

Figure 12: Framework-Agnostic Programming with Pooled Resources

The second decoupling line in Fig. 12 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. 11. 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.

7 EVALUATION

Given the vendor lock-in problem which pervades current Big Data solutions offered by cloud providers, the products of this research are evaluated in terms of inter-cloud portability and interoperability, as well as usability. Additionally, as existing solutions tend to combine a range of technologies to enable processing of different types of big data, leading to code duplication and low maintainability, these factors are considered in our evaluation.

7.1 Prototype

For the purposes of evaluating the proposed architecture, a prototype was constructed and implemented using the OSDC and Microsoft Azure clouds. 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.).

Only streaming data was tested in this initial experiment. A simple pipeline was constructed to calculate the PUE (power usage effectiveness) of a fictitious server room. The PUE calculation was implemented using the following formula:

$$\mathbf{PUE} = \frac{\mathbf{Total\ Facility\ Energy}}{\mathbf{IT\ Equipment\ Energy}}$$

An integrated gateway server dispatching consumption data was simulated programmatically. The simulator dispatched Total Facility Energy and IT Equipment Energy data every second to a Kafka server, which fed streaming data to the Big Data Framework.

The data was processed in parallel by Flink workers deployed as containerised services to a virtual machine in the OSDC, a virtual machine in the Azure cloud and two virtual machines running on local hardware. As Docker Swarm comes as standard with the latest Docker implementations, it was used for orchestration and for scaling the number of running containers.

The pipeline code processed the streaming data using fixed windows of 30 seconds, based on event time. It emitted results at the end of every window, and the results were posted to a different Kafka topic. For the sake of simplicity, no late data was allowed and late data was discarded.

7.2 Results

This initial experiment proved that the proposed architectural pattern is feasible and allows for a unifying model for both batch and stream processing by a pool of seamlessly integrated multi-cloud resources. It demonstrated the possibility of horizontal scaling using orchestration technology to increase the number of containers running the framework's workers. It also demonstrated the possibility of vertical scaling through the incorporation of additional machines in a provider-agnostic way.

In terms of portability and interoperability, we compared the proposed solution with existing big data architectures offered as managed services in the cloud. Table 1 summarises the result of this comparison. Solutions which are based on separate processing for stream and batch data (Google Cloud Dataproc, Microsoft Azure HDInsight, Amazon EMR, Oracle Big Data Cloud Service and IBM BigInsights) are grouped together and displayed as Lambda Architecture.

Table 1: Comparison Between the Proposed Solution and Existing Big Data Architectures in the Cloud

Evaluation Metric	Lambda Archit.	Google Dataflow	Proposed Solution
Inter-Cloud Portability	Medium	Low	High
Inter-Cloud Interoperability	Medium	Medium	High
Code Duplication	High	Low	Low
Usability	Low	Medium	Medium

Inter-cloud portability is defined as the ease with which a service hosted in one cloud can be migrated to a different cloud. Services based on the Lambda Architecture are generally portable, provided that the technologies used are open-source and available in both the source and the destination clouds. The Dataflow solution is based on an open-source programming language, but it would require a change in service model from SaaS to PaaS in order to be easily portable across clouds. Our proposed solution is highly portable, as it is based on containers and a PaaS service model from the start. Irrespective of which cloud provider is utilised, the containers are built from the same base image, and guarantee an identical execution environment for the service.

Inter-cloud interoperability is defined as the ease with which a service hosted in one cloud can operate harmoniously with services hosted on different clouds as part of the same architecture. Both the Lambda Architecture and the Dataflow solution were evaluated as medium, as they do allow for distribution, although additional setup would need to be carried out in order to provide service discovery and orchestration. The proposed solution is ranked as high, as it is based on Docker Swarm technology, which provides service discovery and orchestration as part of the Docker distribution, with no need to install, configure and manage an external tool for this purpose. Because machines can self-register and automatically become part of an orchestrated swarm, it does not matter where specific container nodes reside, they are all part of the swarm and are seamlessly interoperable.

Code Duplication and Maintainability are related metrics, so they are evaluated together. Maintainability is defined as the ease with which the big data processing code can be changed. Code duplication leads to low maintainability, as any change in the processing logic needs to be implemented twice, increasing the amount of development work involved, and the risk of bugs being introduced. Lambda Architecture-based services use different technologies to process batch and stream data, so the logic within the processing pipelines is duplicated, leading to low maintainability. Both Google Dataflow and the solution proposed in this research are based on a unified data processing pipeline for batch and stream data, so code duplication is low and maintainability is improved.

Usability is defined here as the ease with which a developer can approach, learn and work with the technologies involved in a given solution. As the Lambda Architecture involves a number of different technologies such as Hadoop, Spark and Hive, the learning curve is high. The Google Dataflow solution is based on well documented open-source SDKs which are easily accessible to software developers. It does not however offer a systematic approach to software design and development, leading to code disparity and reducing the potential for collaboration and reuse. The proposed solution is also a medium at this point, but our aim is to develop a domain-specific modelling language to facilitate the design and model of container-based big data framework deployments using a notation that is easy to use and instantly recognisable, not only by developers, but also by architects and business users.

8 CONCLUSIONS AND FUTURE WORK

This paper presented a contribution to the field of Big Data Analytics and 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 demonstrated 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.

The issues of code duplication and low maintainability, which are known to affect the Lambda Architecture, are also addressed by our solution. By adopting a unifying programming model for processing batch and stream data, we have demonstrated how these metrics are improved.

A limitation of this initial experiment is the fact that the evaluation of other big data architectures was based on documentation rather than reproduction of the experiment. We aim to address this as our research progresses.

We envision further development of the initial prototype 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).

We are also working on developing a modelling notation and a model-driven engineering approach to designing, modelling and developing big data solutions using the architectural pattern proposed in this paper.

9 ACKNOWLEDGEMENTS

This work (blinded for reviewing purposes).

9 REFERENCES

Akidau, Bradshaw, Chambers, Chernyak, Fernández-Moctezuma, Lax, McVeety, Mills, Perry, Schmidt & Whittle (2015) The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 8, pp. 1792–1803.

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

Apache Flink 1.3 Documentation: Standalone Cluster (n.d.) [Online]. Available from: < https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/cluster_setup.html> [Accessed 14 July 2017].

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

Ardagna, Di Nitto, Casale, Petcu, Mohagheghi, Mosser, Matthews, Gericke, Ballagny, D'Andria, Nechifor & Sheridan (2012) MODAClouds: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds. In: *Proceedings of the 4th International Workshop on Modeling in Software Engineering, 2012*. Piscataway, NJ, USA: IEEE Press, pp. 50–56.

Assis & Bittencourt (2016) A Survey on Cloud Federation Architectures: Identifying Functional and Non-Functional Properties. *Journal of Network and Computer Applications,* 72 September, pp. 51–71.

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

Built-in I/O Transforms (n.d.) [Online]. Available from: <https://beam.apache.org/documentation/io/built-in/> [Accessed 9 August 2017].

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

Chan, Gray, Wellings & Audsley (2014) Exploiting Multicore Architectures in Big Data Applications: The JUNIPER Approach. *Programmability Issues for Heterogeneous Multicores (MULTIPROG)*.

Chen, G. J., Yilmaz, Wiener, Iyer, Jaiswal, Lei, Simha, Wang, Wilfong & Williamson (2016) Realtime Data Processing at Facebook. ACM Press, pp. 1087–1098.

Chen, H. M., Kazman, Haziyev, Kropov & Chtchourov (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.

Dean & Ghemawat (2008) MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51 (1) January, p. 107.

Evans & Feng (2013) *Storm-YARN Released as Open Source | YDN Blog - Yahoo* [Online]. Available from: <https://developer.yahoo.com/blogs/ydn/storm-yarn-released-open-source-143745133.html> [Accessed 28 October 2016].

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

Hadoop Cluster Setup (2017) *Apache Hadoop 3.0.0-alpha4 – Hadoop Cluster Setup* [Online]. Available from: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html> [Accessed 14 July 2017].

Hausenblas (2014) *Twitter Open-Sources Its MapReduce Streaming Framework Summingbird* [Online]. Available from: <https://www.infoq.com/news/2014/01/twitter-summingbird> [Accessed 28 October 2016].

Hindman, Konwinski, Zaharia, Ghodsi, Joseph, Katz, Shenker & Stoica (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.

Java Platform, Standard Edition 8 (2014) [Online]. Available from: <https://docs.oracle.com/javase/8/docs/api/> [Accessed 30 October 2016].

Khare, An, Gokhale, Tambe & Meena (2015) Reactive Stream Processing for Data-Centric Publish/Subscribe. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, 2015*. New York, NY, USA: ACM, pp. 234–245.

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

Kreps (2014) *Questioning the Lambda Architecture - O'Reilly Media* [Online]. Available from: <https://www.oreilly.com/ideas/questioning-the-lambda-architecture> [Accessed 28 October 2016].

library/oraclelinux - Docker Hub (n.d.) [Online]. Available from: <https://hub.docker.com/_/oraclelinux/> [Accessed 23 August 2017].

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

Marz & Warren (2015) *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1st ed. Manning Publications.

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

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

O'Malley (2008) *Apache Hadoop Wins Terabyte Sort Benchmark | Hadoopnew - Yahoo* [Online]. Available from: <https://developer.yahoo.com/blogs/hadoop/apache-hadoop-wins-terabyte-sort-benchmark-408.html> [Accessed 30 October 2016].

Opara-Martins, Sahandi & Tian (2016) Critical Analysis of Vendor Lock-in and Its Impact on Cloud Computing Migration: A Business Perspective. *Journal of Cloud Computing*, 5 (1) December, p. 4.

oracle/openjdk - Docker Hub (n.d.) - *Docker Hub* [Online]. Available from: <https://hub.docker.com/r/oracle/openjdk/> [Accessed 23 August 2017].

Pahl & Lee (2015) Containers and Clusters for Edge Cloud Architectures – A Technology Review. In: *2015 3rd International Conference on Future Internet of Things and Cloud, August 2015*. pp. 379–386.

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].

Satzger, Hummer, Inzinger, Leitner & Dustdar (2013) Winds of Change: From Vendor Lock-In to the Meta Cloud. *IEEE Internet Computing,* 17 (1) January, pp. 69–73.

Silva, Rose & Calinescu (2013a) 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.

Silva, Rose & Calinescu (2013b) Towards a Model-Driven Solution to the Vendor Lock-In Problem in Cloud Computing. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science, December 2013*. vol. 1. pp. 711–716.

Spark Standalone Mode (n.d.) *Spark 2.2.0 Documentation* [Online]. Available from: <https://spark.apache.org/docs/latest/spark-standalone.html> [Accessed 14 July 2017].

Stewart & Singer (2012) Comparing Fork/Join and MapReduce. In: *Department of Computer Science, Heriot-Watt University, 2012*. Citeseer.

Synergy Research Group (2016) *AWS Remains Dominant Despite Microsoft and Google Growth Surges* [Online]. Available from: <https://www.srgresearch.com/articles/aws-remains-dominant-despite-microsoft-and-google-growth-surges> [Accessed 2 March 2017].

Tejedor (2013) Lambdoop, a Framework for Easy Development of Big Data Applications [Online]. Presented at: *December 3, 2013*. Available from: <http://www.slideshare.net/Datadopter/lambdoop-a-framework-for-easy-development-of-big-data-applications> [Accessed 28 October 2016].

Villari, Celesti, Fazio & Puliafito (2014) AllJoyn Lambda: An Architecture for the Management of Smart Environments in IoT. IEEE, pp. 9–14.

Whiteneck, Tufte, Bhat, Maier & Fernández-Moctezuma (2010) Framing the Question: Detecting and Filling Spatial-Temporal Windows. In: *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming, 2010*. New York, NY, USA: ACM, pp. 19–22.

Yasrab & Gu (2016) Multi-Cloud PaaS Architecture (MCPA): A Solution to Cloud Lock-In. In: *2016 3rd International Conference on Information Science and Control Engineering (ICISCE), July 2016*. pp. 473–477.

Zikopoulos, deRoos, Parasuraman, Deutsch, Giles & Corrigan (2013) *Harness the Power of Big Data The IBM Big Data Platform*. McGraw-Hill Education.

Zikopoulos, Eaton, Deroos, Deutsch & Lapis (2012) *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. New York: McGraw-Hill Osborne.