Requirements Engineering for Large-Scale Big Data Applications

Thalita Vergilio, Muthu Ramachandran, and Duncan Mullier

School of Computing, Creative Technology and Engineering
Leeds Beckett University, Leeds, UK
T.Vergilio@leedsbeckett.ac.uk

Abstract: As the use of smart phones proliferates, and human interaction through social media is intensified around the globe, the amount of data available to process is greater than ever before. As consequence, the design and implementation of systems capable of handling such vast amounts of data in acceptable timescales has moved to the forefront of academic and industry-based research. This research represents a unique contribution to the field of Software Engineering for Big Data in the form of an investigation of the big data architectures of three well-known real-world companies: Facebook, Twitter and Netflix. The purpose of this investigation is to gather significant non-functional requirements for real-world big data systems, with an aim to addressing these requirements in the design of our own unique reference architecture for big data processing in the cloud: MC-BDP (Multi-Cloud Big Data Processing). MC-BDP represents an evolution of the PaaS-BDP (Platform as a Service for Big Data Processing) architectural pattern, previously developed by the authors. However, its presentation is not within the scope of this study. The scope of this comparative study is limited to the examination of academic papers, technical blogs, presentations, source code and documentation officially published by the companies under investigation. Ten non-functional requirements are identified and discussed in the context of these companies' architectures: batch data, stream data, late and out-of-order data, processing guarantees, integration and extensibility, distribution and scalability, cloud support and elasticity, fault-tolerance, flow control, and flexibility and technology agnosticism. They are followed by the conclusion and considerations for future work.

Keywords: Big Data, Requirements Engineering, Batch, Stream, Scalability, Fault Tolerance, Flow Control, Technology Agnosticism, Processing Guarantees

1. Introduction

Big data is defined as data that challenges existing technology for being too large in volume, too fast, or too varied in structure. Big data is also characterised by its complexity, with associated issues and problems that challenge current data science

processes and methods [1]. Large internet-based companies have the biggest and most complex data, which explains their leading role in the development of state-of-the-art big data technology. It has been reported that, in one minute of internet usage, 1 million people log into Facebook, 87,500 new tweets are posted, and 694,444 hours of videos are watched on Netflix [2]. Processing these vast amounts of data presents challenges not only in terms of developing the most appropriate algorithms to best inform these companies and their future decisions, but also in terms of assembling the technology needed to perform these calculations in a timely manner.

This paper contributes to the existing knowledge in the area of Software Engineering for Big Data by performing a search of the existing literature published by three major companies, and an outline of the strategies devised by them to cope with the technological challenges posed by big data in their production systems. Non-Functional requirements are important quality attributes which influence the architectural design of a system [3]. Ten non-functional requirements for big data systems are identified and discussed in the context of these real-world implementations. These requirements are used to guide the design and development of a new reference architecture for big data processing in the cloud: MC-BDP. The presentation, evaluation and discussion of MC-BDP are addressed in a different publication.

The companies targeted for this study are Facebook, Twitter and Netflix. The methodology used in this comparative study is explained in Section 2, which covers the scope of this research, the criteria used for selecting the target companies, as well as definitions for the non-functional requirements under examination. Section 3 discusses related work, and Section 4 addresses each non-functional requirement and discusses how they are implemented by the three companies in their production systems. Finally, Section 5 presents the conclusion and considerations for future work.

2. Research Methodology Using Systematic Literature Review

This section describes this research's methodology using a systematic literature review, as illustrated in Figure 1. The first step is the definition of the scope of this research. Since this research is literature-based, its scope is defined in terms of which literature sources to use. The next step is an explanation of the criteria used to select the companies under examination from a pool of potential matches. The third step consists of explaining the comparison criteria used to evaluate the architectures of the selected companies. It describes the ten non-functional requirements for large-scale big data applications which form the focus of this study. Finally, the last step is an evaluation of the different approaches taken by these companies to implement the aforementioned requirements.

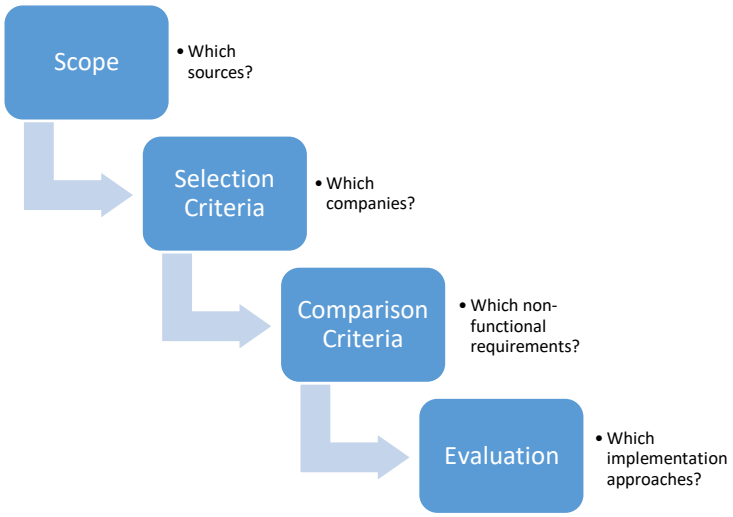

Figure 1: Summary of Research Methodology

2.1 Scope

The scope of this paper is limited to academic papers, technical documentation, and presentations or blog posts officially published by the companies under evaluation. Although the authors recognise that direct observation of the systems under evaluation by way of a set of case studies would have yielded more reliable, and perhaps more valuable results, time and resource constraints limited the scope of the present study. Aware of this limitation, this research endeavoured to use only sources officially published or endorsed by the companies under evaluation, in an effort to be true to the systems under examination. Where no information could be found with regards to specific criteria or particular aspects of the systems under examination, this is clearly stated by the authors.

Table 1 shows a summary of the materials used as source for this research, classified by type. Six academic papers were found describing the systems used by the three companies under study. Additionally, twelve technical blog articles, four presentations, and the source code or documentation of four systems were used as source for this study. Twitter had the strongest academic presence at the time of writing, with two papers available through the ACM Digital Library [4] and [5], and one paper available through the IEEE Xplore Digital Library [6]. Two relevant academic papers were found for Facebook, one available through the ACM Digital Library [7], and one available through the IEEE Xplore Digital Library [8]. Finally, only one academic paper was found at the time of writing describing the big data systems at Netflix: [9], available through the ACM Digital Library.

Table 1: Classification and summary of source materials.

Company	Academic Paper	Technical Blog	Presentation	Code/Documentation
Facebook	2	2	1	1
Twitter	3	2	1	3
Netflix	1	8	2	0

Although Netflix was underrepresented in terms of academic sources when compared to the other two companies evaluated, it had the highest number of relevant non-academic sources: eight technical blog articles, and two presentations. Both Facebook and Twitter, in comparison, had two technical blog articles and one presentation relevant to this research. Finally, in terms of source-code and documentation available for public peruse, Twitter had three entries, corresponding to the source-code for Scalding [10], Heron [11], and Storm [12]. The code for Facebook's Scribe [13] was available as open-source through the company's archive repository. At the time of writing, none of the Netflix systems assessed by this study were open-source.

This section presented the scope of the evaluation conducted in this research, which was limited to academic papers, technical blog articles, presentations and source code/documentation published by Facebook, Netflix and Twitter. The next section explains the selection criteria used to select the three target companies.

2.2 Selection Criteria

This section explains how the three companies: Facebook, Twitter and Netflix were selected as target of this study. An initial survey of big data architectures was conducted, limited to peer-reviewed academic papers. Three search engines were primarily used to perform the searches: Google Scholar, IEEE Xplore Digital Library and ACM Digital Library. The initial survey searched for terms such as "big data", "big data processing", "big data software" and "big data architecture". In the interest of thoroughness, synonyms were used to replace key terms where appropriate, e.g. "system" for "software".

The first classification which became apparent was in terms of who developed the solutions presented. The results found comprised technologies developed

1) by academia,
2) by real-world big data companies,
3) by industry experts as open-source projects, or
4) by a combination of the above.

This research focuses on category number 2.

A further classification can be drawn from the academic papers reviewed, this time in terms of how the contributions presented were evaluated. Three cases were encountered:

A) cases where there is no empirical evaluation of the proposed solution,
B) cases where the empirical evaluation of the proposed solution is purely experimental, and
C) cases where peer-reviewed published material was found describing the results of implementing the proposed solution in large-scale commercial big data settings.

In order to select suitable companies to include in this study, the focus of this research was limited to category C.

Three companies were selected within the criteria characterised above: Facebook, Twitter and Netflix. These were selected from a wider pool of qualifying companies which included Microsoft [14], [15], Google [16], [17], and Santander [18]. The rationale for choosing the three aforementioned companies is based on the quantity, quality and clarity of the information encountered, as well as availability of technical material online such as project documentation and architectural diagrams.

2.3 Definitions

This section explains how the ten non-functional requirements: batch data, stream data, late and out-of-order data, processing guarantees, integration and extensibility, distribution and scalability, cloud support and elasticity, fault-tolerance, flow control, and flexibility and technology agnosticism were identified as non-functional requirements for this study. It then provides definitions for each requirement.

The ten non-functional requirements selected for this study were based on the initial literature survey of official academic publications explained in Section 2.2. As with the previous selection, focus was given to solutions developed by real-world big

data companies. However, it is worth noting that these requirements are widely addressed in open-source, as well as purely academic solutions [19], [20], [21], [19], [22]. Likewise, they are highlighted in non-academic sources such as commercial solutions and cloud-based services [23], [24], [25].

2.3.1 Batch Data

This requirement refers to the processing of data which is finite and usually large in volume, e.g. data archived in distributed file systems or databases. An important requirement for real-world big data systems is that they must be capable of processing large amounts of finite, usually historical data. Figure 2 illustrates a typical case for batch data processing.

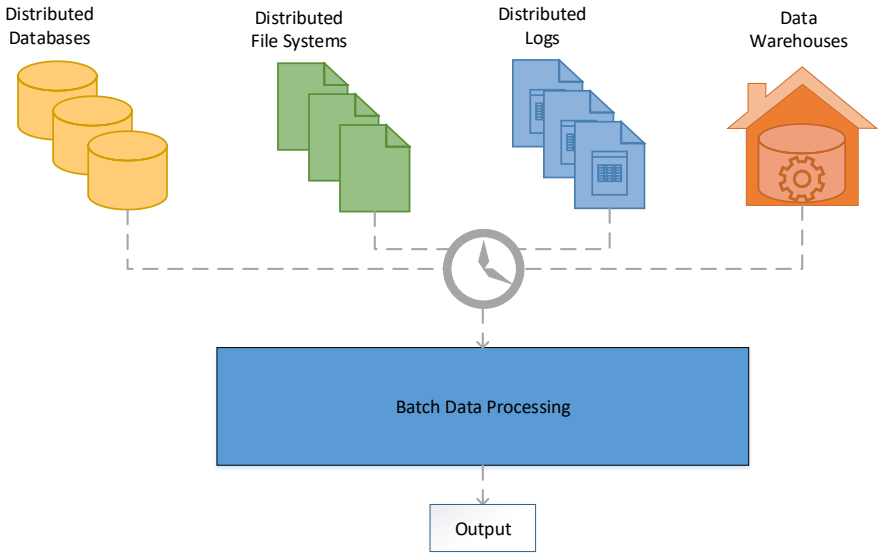

Fig. 2: Batch Data Processing.

As we can see in Figure 2, large amounts of data is first collected in static storage spaces such as, for example, distributed databases, file systems, logs or data warehouses. It is then processed in finite batches by powerful, usually distributed technology. The name batch processing comes from this approach to data processing whereby the data is collected into finite batches before it is processed.
This section described the non-functional requirement for a large-scale big data system to be capable of processing batch data, defined as data which is finite, usually historical, and large in volume. The next section describes the non-functional requirement for a large-scale big data system to be capable of processing stream data.

2.3.2 Stream Data

This requirement refers to the processing of data which is potentially infinite and usually flowing at high velocity. For example, real-world big data systems are generally required to capture and process user activity or monitoring data in real-time, or close to real-time. Figure 3 illustrates a typical case for stream data processing.

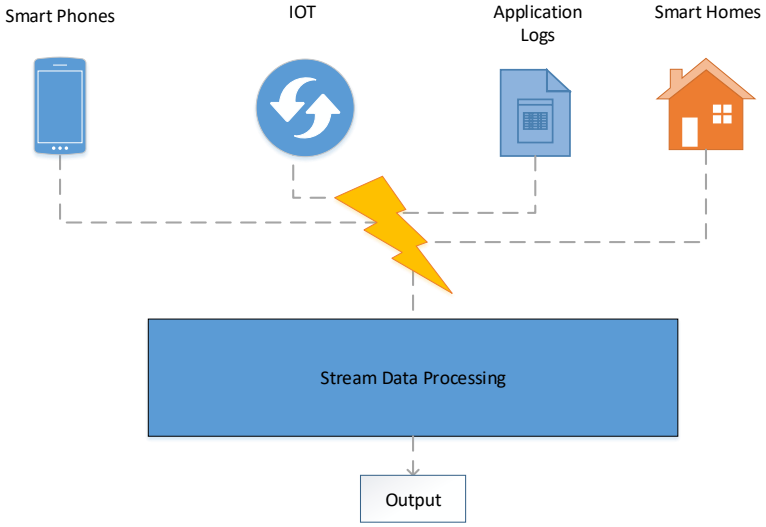

Fig. 3: Stream Data Processing.

In Figure 3, we can see that data is collected from a variety of sources such as, for example, smart homes, application logs, or the Internet of Things (IOT). It is then processed in real-time, or as close to real-time as the technology allows. This approach is called stream processing because the incoming data is very large, potentially infinite, so processing cannot wait until all the data is available before it starts. Instead, processing is ongoing. It takes place at defined intervals, and emits results at defined intervals. Differently from batch processing, completion is not a concept that is used in the context of stream processing, since the data source is potentially infinite. This, however, does not mean that accuracy is compromised, as it remains not only possible, but indeed a desirable quality of mature streaming systems, as demonstrated by [17].

The capacity to process stream data was identified as a non-functional requirement not only within the architectures of the three companies evaluated, as discussed in detail in Section 4.2, but also of other large-scale big data companies such as Microsoft's library for large-scale stream analytics, Trill, used in Azure Stream Analytics and ads reporting for the Bing search engine [26], Google's Dataflow, used for statistics calculations for abuse detection, billing, anomaly detection, and others [17], and Linkedin's Samza, currently used in production and deployed to more than 10,000 containers for anomaly detection, performance monitoring, notifications,

real-time analysis, and others [27]. The literature review conducted as part of this research therefore concluded that the capacity to process stream data is a fundamental requirement of large-scale big data architectures, and, as stream technology develops, it becomes capable of catering for a larger number of use-cases previously consigned to batch processing, as discussed comprehensively in [28].

This section described the non-functional requirement for large-scale big data systems to be capable of processing stream data, defined as data which is potentially infinite in size and usually arriving at high velocity. The next section describes the non-functional requirement for a large-scale stream big data system to be capable of processing late and out of order data.

2.3.3 Late and Out-of-Order Data

This requirement relates to stream processing and refers to the processing of data which arrives late or in a different order from that in which it was emitted. Streaming data from mobile users, for example, could be delayed if the user loses reception for a moment. In order to handle late and out of order data, a system must have been designed with this requirement in mind. Figure 4 illustrates late and out-of-order data. The records emitted at 10:01:55 and 10:03:22 are significantly delayed. The record emitted at 10:02:38 actually arrives before the one emitted at 10:01:55. Similarly, the record emitted at 10:04:05 arrives before the record emitted at 10:03:22. Late and out-of-order records such as the ones depicted are common with real-time user data which is subject to network delays.

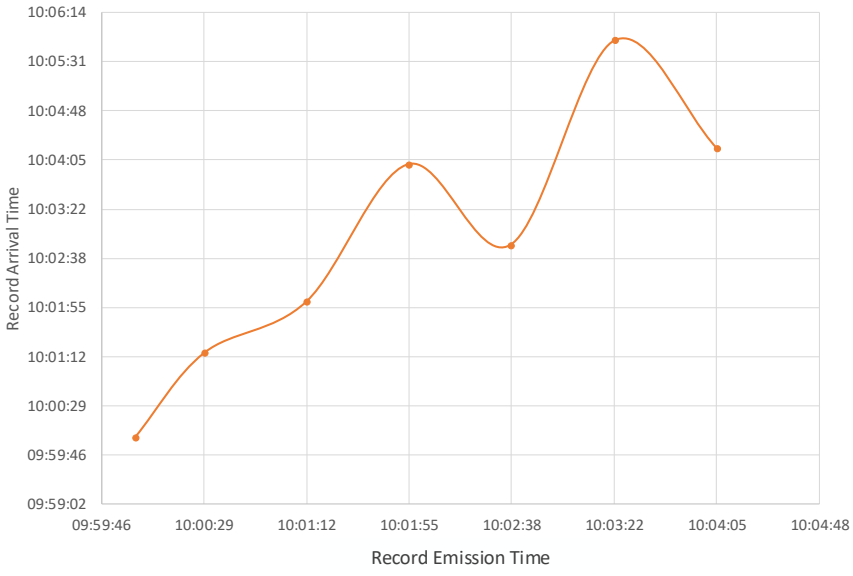

Fig. 4: Late and Out of Order Data.

Figure 5 shows a summary of strategies for dealing with late and out of order data. The windowing strategy defines how the data is grouped into windows of time to enable the processing of otherwise infinite data. At a minimum, the period (how frequently each window starts) and duration (how long each window lasts for) must be defined. Thus, a scenario where the period is longer than the duration characterizes sampling, whereas one where the duration is longer than the period characterizes tumbling or sliding windows. Where the period and duration are the same length, the window is considered a fixed window. The triggering strategy defines how often results are emitted, e.g. at the end of every window, at the end of every windows plus a defined tolerance, at fixed intervals, etc. The state strategy defines what data is persisted during stream data processing and how long for. It is useful for computing aggregates, and can be defined at key, window or application level. Finally, the watermark strategy defines how late the data is expected to be, and usually signals the application to start processing at a time all data is believed to have been received. These four strategies combined define how late and out of order data is handled by a stream application.

Fig. 5: Strategies for Dealing with Late and Out of Order Data.

As an example, a system may be configured to process a simple count of distinct words entered into a search engine. The windowing strategy is defined to use sliding windows of 10 seconds, starting every 5 seconds. The triggering strategy is configured to emit (partial) results every 5 seconds, and to accumulate more data as it arrives to emit in the next 5 seconds. The state strategy is configured to use per-window state. Finally, the watermark strategy is configured to expect all data to have arrived within 5 minutes of emission. Figure 6 summarises this sample configuration.

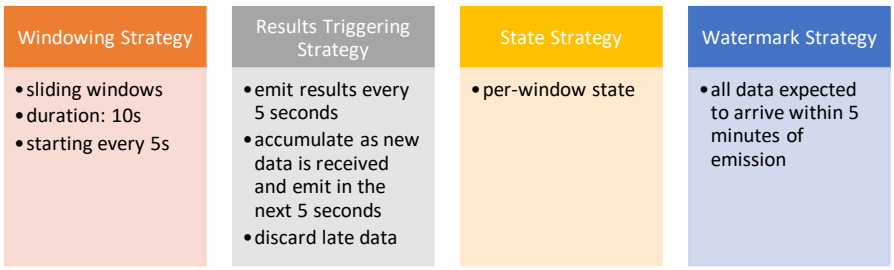

Fig. 6: Sample Configuration for Dealing with Late and Out of Order Data.

Any data that is less than 5 minutes late is incorporated into the calculations. Partial results for a window of 10 seconds are emitted after 5 seconds, with subsequent emissions (adjustments) occurring every 10 seconds, until the watermark is achieved. Any data arriving later than the watermark is discarded.

This section described the non-functional requirement for streaming systems being capable of processing data which arrives late and out of order. The next section describes the non-functional requirement of ensuring that a distributed system honours one of three processing guarantees.

2.3.4 Processing Guarantees

This requirement refers to the processing guarantees that a distributed stream system offers, i.e. exactly once, at least once and at most once. It determines whether processing tasks assigned to workers are replayed in case of system failure [29]. While exactly once processing is ideal, it comes at a cost which could translate into increased latency. This requirement is used to evaluate how different systems and different use-cases warranted different compromises in terms of latency and processing guarantees. Figure 7 illustrates the three types of processing guarantees.

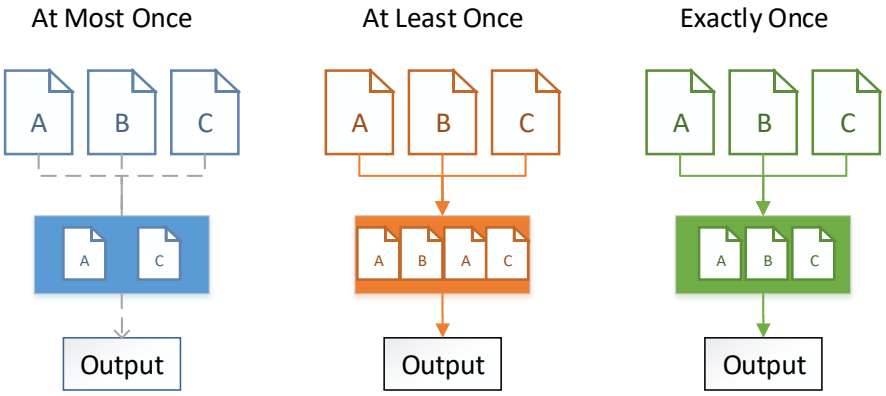

Fig. 7: Processing Guarantees

The first type of processing guarantee illustrated in Figure 7, at most once, focuses on avoiding re-processing of data, to the detriment of duplication. In the event of worker failure, the data processing task assigned to that worker will not be restarted, resulting in data loss, as illustrated in Figure 7. The second type of processing guarantee, at least once, focuses on avoiding data loss, even if it means that processing task (and results) are duplicated. Finally, the third type of processing guarantee, exactly once, is a combination of the former two: it ensures that there is no data loss, and it also ensures that there is no duplication. Although the exactly once processing guarantee is the most accurate, it is not always the most desirable, as it is more costly resource-wise to achieve when compared to the other two. In order to ensure that data is processed exactly once, an external checkpointing system is usually employed to ensure that each task can be re-played from where it left off (or as close as possible to that) in the event of node failure. These checkpoints can be expensive and involve additional disk and networking resources which may not be desirable in every particular use-case. The less processing duplication desired, i.e. the stricter the exactly once guarantee required, the higher the checkpointing frequency needed, since each worker must output the state of the processing task several times throughout its execution.

This section described the non-functional requirement for processing guarantees, defined as assurances provided by a distributed parallel system with regards to how many times incoming data will be processed regardless of possible node or transmission failures. The next section describes the non-functional requirement for integration and extensibility.

2.3.5 Integration and Extensibility

This requirement refers to how well the systems presented integrate with existing services and components. It also refers to provisions made to facilitate the extension

of the existing architecture to incorporate different components in the future. For illustration, Figure 8 is a simplified diagram representing Heron's architecture. Heron was designed by Twitter to be fully compatible with Storm, their previous big data framework for stream processing. As Figure 8 shows, the Heron API accepts both Heron and Storm topologies, thus facilitating the integration of the new system with legacy processing code defined as topologies.

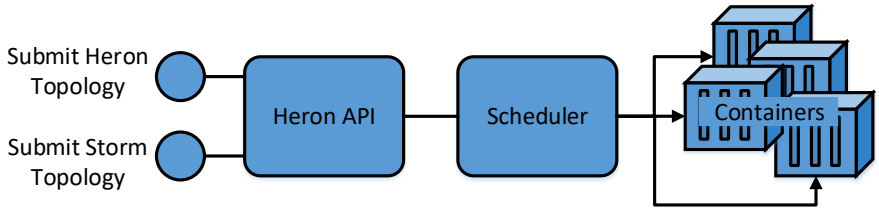

Fig. 8: Twitter Heron Simplified Architecture

This section described the non-functional requirement for integration and extensibility, defined as the capacity of a system to integrate with existing services and components, as well as the provisions made to facilitate the extension of the existing architecture to incorporate different components in the future. The next section defines the non-functional requirement of distribution and scalability.

2.3.6 Distribution and Scalability

This requirement refers to how easily the data processing can be distributed amongst different machines, located in different data centres, in a multi-clustered architecture. Dynamic scaling, which addresses the possibility of adding or removing nodes to a running system without downtime, is also part of this requirement. Figure 9 illustrates the processing of stream big data by a container-based architecture using a pipe analogy: the length of the pipe represents the time each container takes to process data, and the diameter of the pipe represents the number of containers processing the data.

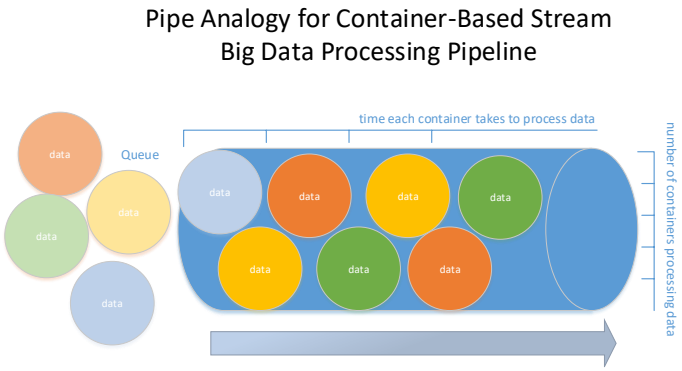

Fig. 9: Pipe Analogy for Container-Based Stream Big Data Processing Pipeline

The wider the diameter of the pipe, the more containers there are processing the data, so the pipe is shorter and the queue is reduced, since data is processed faster. An example of horizontal scaling would be to launch more data processing containers running on the same physical infrastructure (same number of nodes, of same capacity). This is illustrated in Figure 10.

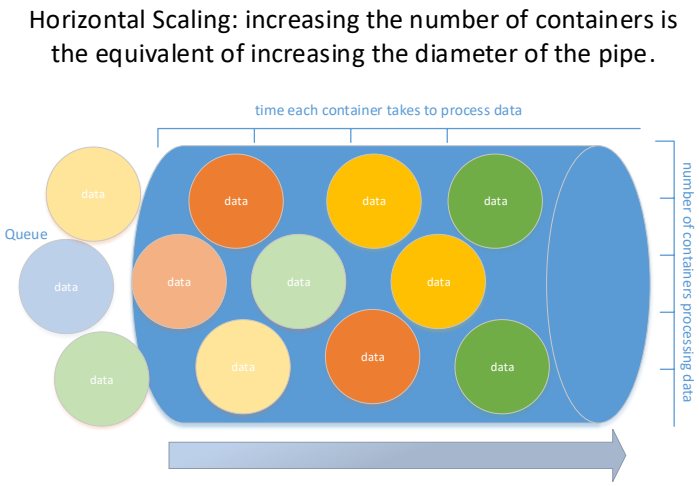

Fig. 10: Horizontal Scaling

At some point, however, horizontal scaling fails to translate into faster processing, and it is necessary to commission more nodes to provide more processing capacity at infrastructure level. This is known as vertical scaling and, using the previous analogy, it is the equivalent of adding more pipes. As consequence, the data flows faster through the pipes and the queue is reduced, as illustrated in Figure 11.

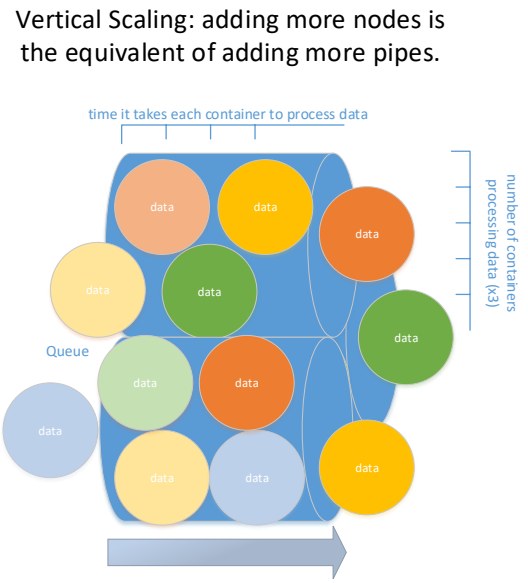

Fig. 11: Vertical Scaling

This section described the non-functional requirement of distribution and scalability, defined as how easily the data processing can be distributed amongst different machines, located in different data centres, in a multi-clustered architecture. The next section describes the non-functional requirement of cloud support and elasticity.

2.3.7 Cloud Support and Elasticity

This requirement refers to the ease with which the architecture (or part of it) can be moved into the cloud to take advantage of the many benefits associated with its economies of scale. Elasticity in particular is a cloud property which allows a system to scale up and down according to demand. Since the user only pays for resources actually used, there is less wastage and it is theoretically cheaper than running the entire infrastructure locally with enough idle capacity to cover for eventual spikes.

Since being unable to easily switch between cloud providers represents a risk to cloud consumers [30], support for a multi-cloud architecture which mitigates the risk of vendor lock-in and allows cloud consumers to transfer resources across providers is addressed as part of this requirement. Figure 12 shows MC-BDP's multi-tenant multi-cloud infrastructure enabled through the use of container technology.

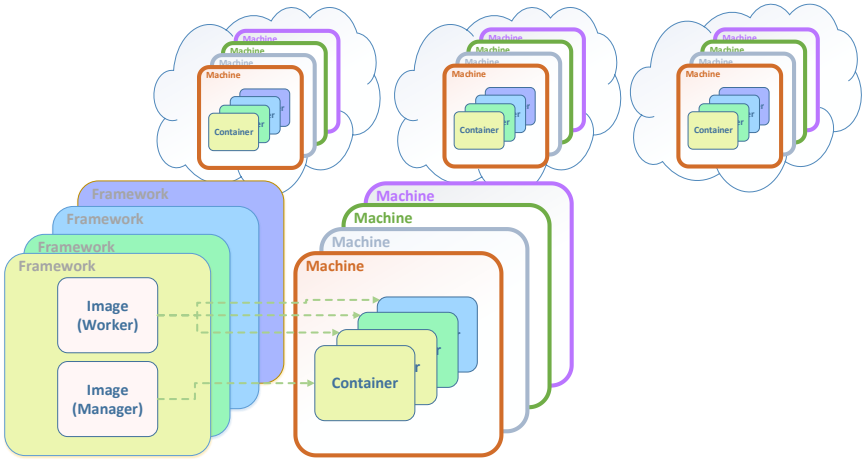

Fig. 12: Multi-Tenant Multi-Cloud Infrastructure Enabled by Container Technology

This section described the non-functional requirement of cloud support and elasticity, defined as the ease with which the architecture (or part of it) can be moved into the cloud to take advantage of the many benefits associated with its economies of scale, in particular with respect to elasticity, defined as the capacity of a cloud system to scale up and down according to demand. The next section describes the non-functional requirement of fault tolerance.

2.3.8 Fault Tolerance

This requirement refers to provisions made at design time so the system can continue to operate should one or more nodes fail. Ideally, production systems should recover gracefully, with minimal effect (if at all) on the user experience. Fault tolerance can be observed at different levels, as illustrated in Figure 13. Using MC-BDP's container-based architecture as an example, a container could become unresponsive, as shown in A, and the container orchestrator would be expected to provide fault tolerance (i.e. re-launch the task in another container). Similarly, a node where several containers are running could become unresponsive, as in B, requiring fault tolerance at both container and node levels (i.e. launch an equivalent node and re-launch the containers that were running on the lost node). A more serious scenario is depicted in C, where a cloud provider's entire region fails. This requires a number of nodes and other resources to be re-created, and all the containers running in those nodes to be re-launched. Finally D illustrates a multi-region failure for a single provider where all resources previously running on that cloud need to be re-launched. Although uncommon, multiple availability zone failures do sometimes occur [31], as do multiple region outages, as exemplified by a DNS disruption that affected Azure customers in all regions in 2016 [32].

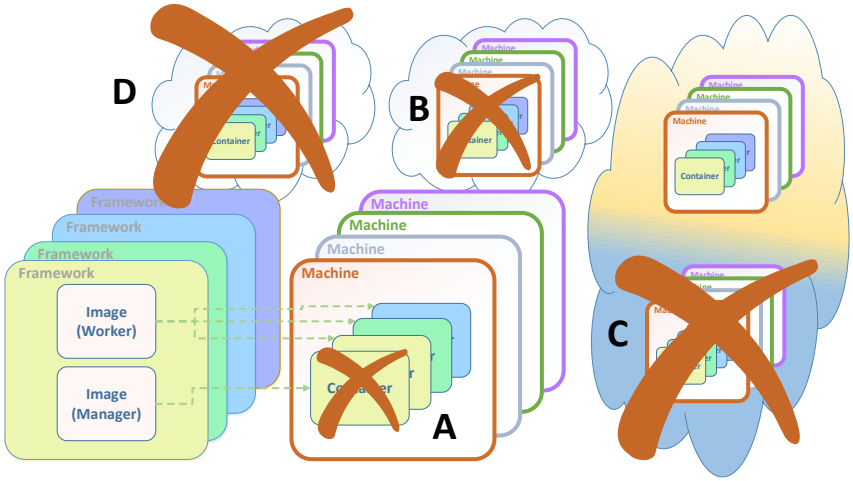

Fig. 13: Fault Tolerance at Different Levels

2.3.9 Flow Control

This requirement refers to scenarios where the data source emits records faster than the system can consume. Strategies for dealing with backpressure, e.g. dropping records, sampling, combining, applying source backpressure, etc. are generally required from real-world big data systems. Figure 14 illustrates flow control defined by an abstract function $f(x)$, which results in fewer records flowing down the stream.

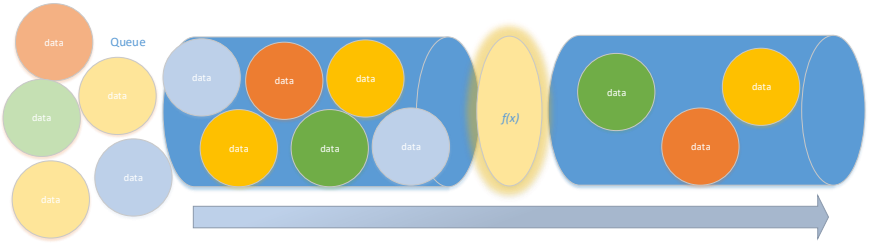

Fig. 14: Flow Control

2.3.10 Flexibility and Technology Agnosticism

This requirement refers to the extent to which the architecture provides the implementer with the option to use different technology in place of existing components. A modular architecture, for example, allows separate components to be replaced or

upgraded with no detrimental effect to the functioning of the system as a whole. Figure 15 shows the MC-BDP reference architecture developed by the authors, together with a sample prototype implementation. MC-BDP is an example of a highly flexible reference architecture designed with technology agnosticism in mind. Each module depicted in the concrete prototype implementation can be replaced with a technologically equivalent component.

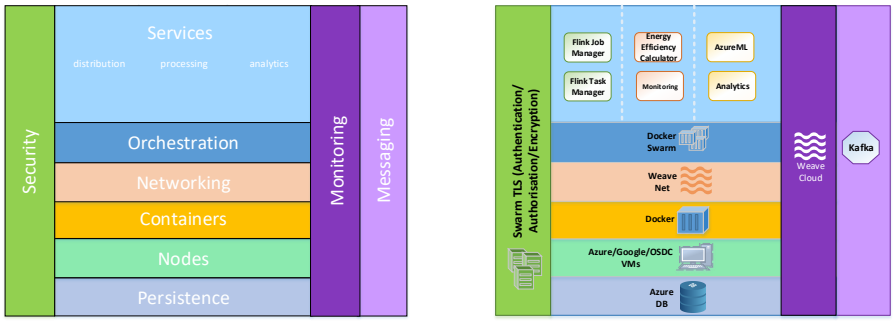

Fig. 15: MC-BDP Reference Architecture and Prototype Implementation

This section presented the methodology used in this research, and included a subsection where the ten non-functional requirements referred to throughout this chapter were defined. The next section discusses related work, and is followed by Section 4, which looks at how the three target companies, Facebook, Twitter and Netflix, addressed the aforementioned requirements.

3. Related Work

This section is divided in three parts: Section 3.1 discusses academic developments in the area of requirements engineering for large-scale big data systems, Section 3.2 examines the literature around big data architectures, and finally Section 3.3 reviews the gap in the literature which the current research endeavours to bridge.

3.1. Requirements Engineering for Big Data

Although requirements engineering is an established area of academic research, requirements engineering for big data is an incipient field and has tended to concentrate on infrastructure requirements, to the detriment of other aspects [33]. This section discusses work which attempted to bring such aspects to the forefront of requirements engineering research. Section 3.1.1. addresses work which attempted

to incorporate the 3 Vs of big data (volume, velocity and variety) into traditional requirements engineering. Section 3.1.2. looks at proposals to devise a requirements engineering context model for the domain of big data.

3.1.1. Incorporating the 3 Vs of Big Data into Requirements Engineering

This section addresses related work which integrates key aspects of big data into the field of requirements engineering. Big data is traditionally defined as displaying the three characteristics of volume, velocity and variety. Being able to integrate these characteristics into a systematic process for the specification of requirements has been highlighted as one of the research challenges in the field [34]. Moreover, these characteristics must be represented in requirements notation in order to ensure they are adequately captured [33]. Noorwali et al. [35] proposed an approach to integrate big data characteristics into quality requirements. However, at the time of writing, their approach has not yet been evaluated empirically. The current research uses known large-scale industry implementations to identify ten non-functional requirements for the specific domain of big-data processing.

3.2.2. Devising a Context Model for Big Data Requirements Engineering

This section discusses related work which propose a new context model for big data in the field of requirements engineering. Eridaputra et al. used the goal-oriented requirements engineering (GORE) method to model requirements and propose a new requirement model based on the characteristics of big data and its challenges. Their research was empirically evaluated through a case study set at a government agency where 26 functional and 10 nonfunctional requirements were obtained from the model, and further validated by stakeholders as accurate [36]. Al-Najran & Dahanayake developed a new requirements specification framework for the domain of data collection which incorporates requirements engineering for big data [37]. This framework was empirically evaluated through quantitative experiments to measure the relevance of Twitter feeds [38].

Madhavji et al. introduced a new context model of big data software engineering where not only computer science and software engineering research were taken into account, but also big data software engineering practice, corporate decision making, and business and client scenarios [33]. Arruda and Madhavji subsequently identified a lack of known artefact models to support requirements engineering process design and project understanding, and proposed the creation of a requirements engineering artefact model for big data end-user applications (BD-REAM) [39]. Based on this initial study, Arruda later developed a context model for big data software engineering and introduced a requirements engineering artefact model containing artefacts such as development practice, corporate decision-making, and research, as well as the relationships and cardinalities between them. At the time of

writing, this research was still in early stages of development, and had not been empirically evaluated [34].

This research is similar to Madhavji et al.'s, in that it looks beyond theoretical contributions in the computer science and software engineering fields in an effort to incorporate recent developments from the industry into its review of non-functional requirements for the domain of large-scale big data applications. Since big data is a very active area of development not only in academia, but also (and perhaps even more) commercially, a thorough specification of requirements for big data ought to include both spheres. Furthermore, the current study looks at real-world implementations of non-functional requirements by some of the largest big data corporations, and discusses the particular use-cases that led to some of their key design decisions.

This section discussed related work in the area of requirements engineering for big data and briefly presented a number of functional services proposed by this research for big data requirements engineering in a multi-cloud environment. The next section addresses the literature related to big data architectures.

3.2. Big Data Architectures

The literature addressing the challenges presented by big data is vast, and the majority of new solutions, architectures and frameworks proposed acknowledge and aim to fulfil one or more of the non-functional requirements identified in Section 2. This section therefore discusses related work by introducing a classification based on the type of contribution proposed:

3.2.1. Evaluation or unique application of widely adopted existing technologies for big data processing.

This class consists of research which leveraged existing, widely adopted technologies for big data processing, and either applied it to a original case or evaluated it in a unique way.

Examples of papers which present an evaluation of existing technology are Spark's evaluation by Shoro and Soomro using a Twitter-based case study [40] and Kiran et al.'s implementation of the Lambda Architecture using Amazon cloud resources [41].

Examples of unique applications of widely adopted existing technology for big data processing are Sun et al.'s use of Hadoop, Spark and MySQL to process big data related to spacecraft testing [42] and Naik's use of Docker Swarm to create a distributed containerised architecture for data processing using Hadoop and Pachyderm [41].

3.2.2. New technologies for the processing of big data.

This class consists of research which proposed entirely new technologies for the processing of big data.

Examples of contributions within this category are Borealis, a distributed stream processing engine developed by a consortium involving Brandeis University, Brown University, and the MIT [19], Millwheel, Google's distributed stream processing system based on the concept of Low Watermarks which was later open-sourced as Apache Beam [16], and Storm, one of the most popular open-source stream processing frameworks, originally developed by Twitter [4].

3.2.3 Original architectural proposals where existing technologies are used or recommended.

This class consists of research which proposed an entirely new architecture based on existing technologies.
AllJoynLambda is an example of an architecture which makes use of MongoDB and Storm as part of an architecture for smart environments in IoT [44]. Basanta-Val et al. propose a new time-critical architecture which utilises Spark and Storm for data processing [45]. An ETL-based approach to big data processing is proposed by Guerreiro et al. This proposal utilises Spark, SparkSQL and MongoDB technologies [46]. Finally, an example of a proposed architecture where the choice of big data technology is left open to the implementer is [47].

This section discussed related work by introducing a classification for research on big data architectures. The next section identifies the gap in the literature addressed by this research.

3.3. Gap in the Literature

Significant advances have been made in the fields of big data recently, and it continues to develop further as devices and applications produce more and more data. However, although new frameworks and technologies are developed and launched into the market at a very fast pace, matching them with systems requirements continues to present a challenge [34]. Arruda highlights the need for addressing big data-specific characteristics in the definition, analysis and specification of both functional and non-functional requirements [34]. This research aims to bridge this gap by abstracting from functional and concentrating on non-functional requirements for the domain of big data which are common to well-known large-scale big data implementations.

Eridaputra et al. call for new methods to model requirements for big data applications [36], and Madhavji et al. identify the need to develop new techniques to assess the impact of architectural design decisions on functional and non-functional requirements [33]. This research is a step in the recommended direction, as it looks at implementations of non-functional requirements for big data by real-world companies and discusses the design decisions that resulted in specific implementations.

Madhavji et al. also drew attention to the lack of academic work on reference architectures and patterns for big data applications, and how these reference architectures can be translated into existing technologies, frameworks, and tools to yield concrete deployments [33]. The current research bridges this gap by proposing a new reference architecture for the domain of stream data processing, and by providing a prototype implementation based on open-source technology.

This section discussed the gap in the literature which motivated the authors to undertake the current research. The next section examines the literature published by the three companies under study to understand how the non-functional requirements defined in Section 2 are addressed in their real-world implementations.

4. Requirements Engineering for Big Data

Systematic approaches to identifying functional services and non-functional requirements are necessary to design and build big data systems. The traditional requirements engineering approaches are unsatisfactory when it comes to identifying requirements for service-oriented systems [48], [35], [49], [34]. This research bridges this gap by identifying five functional services for big data requirements engineering in Section 4.1, and by providing a comparison of three large companies' approaches to implementing non-functional requirements for big data in Section 4.2.

4.1 Identification of Functional Services for Big Data

The non-functional requirements identified in this study, together with a gap analysis exercise based on the literature survey, were used to identify a number of functional services for big data requirements engineering in a multi-cloud environment. These functional services are illustrated in Figure 16.

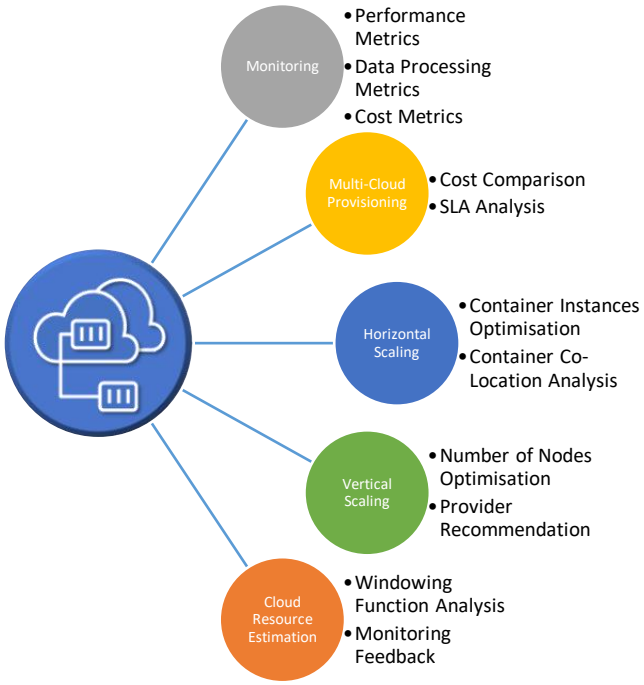

Fig. 16: Identified Functional Services for Big Data Requirements Engineering in a Multi-Cloud Environment

A monitoring service provides performance metrics (CPU, memory, and network usage), as well as data processing metrics (number of records ingested, number of records processed, percentage of data loss). It also provides cost expended in terms of resources utilized from each provider, and cost estimations based on past usage. A multi-cloud provisioning service compares offerings from different providers in terms of cost and other preconfigured SLAs. A horizontal scaling service provides optimisations for the number of container instances running in the cluster, as well as container co-location analysis and recommendations. A vertical scaling service provides optimisations for the number of nodes in the cluster and offers provider comparison and recommendations based on weighted desired qualities. Finally, the cloud resource estimation service adjusts the initial estimations entered by designers of stream processing systems before a given configuration is run, based on the windowing function selected. It also communicates with the monitoring service to adjust estimations for running systems. Figure 17 summarises the resource estimation process provided by MC-Compose, a cloud resource estimation service developed as part of this research.

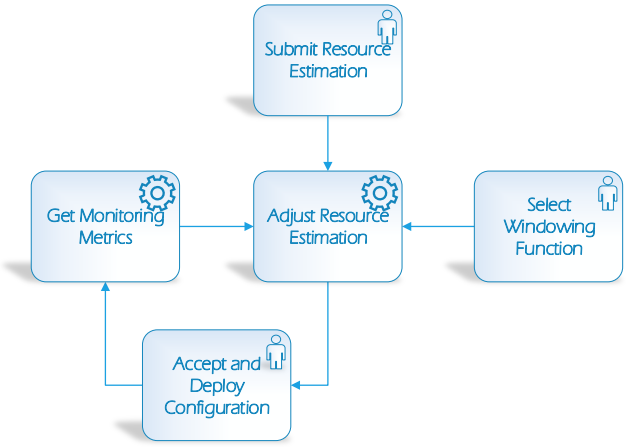

Fig. 17: MC-Compose Resource Estimation Process Summary

The resource estimation process summarised in Figure 17 starts with a resource estimate for CPU, memory and network consumption submitted by a user. This is based on the assumption that the system is unknown, or has not yet been deployed to production. The windowing function, used to render the potentially infinite stream of data finite for processing, is then selected. It consists of a period, which represents how frequently a processing window starts, and a duration, which represents the duration of each processing window. Strategies such as sampling can be implemented by selecting a period higher than the duration, whereas sliding windows can be implemented by selecting a higher duration than period. MC-Compose takes into account the windowing function selected, and adjusts the resource estimation entered by the user, who is prompted to accept the adjusted requirements, or manually override them. Once the system is deployed, the monitoring service depicted in Figure 16 sends metrics to MC-Compose, which are used to further refine the resource estimation calculations and thus improve the recommendations made to the user.

This section summarised five functional services for big data in a multi-cloud environment, identified as part of this research. The next section discusses the approaches of the three companies selected by this study to implementing the ten non-functional requirements identified in Section 2.3.

4.2 Non-Functional Requirements

This section presents ten non-functional requirements discussed in the literature published by the three companies selected in Section 2.2. It then examines how they implemented these requirements and compares the different solutions.

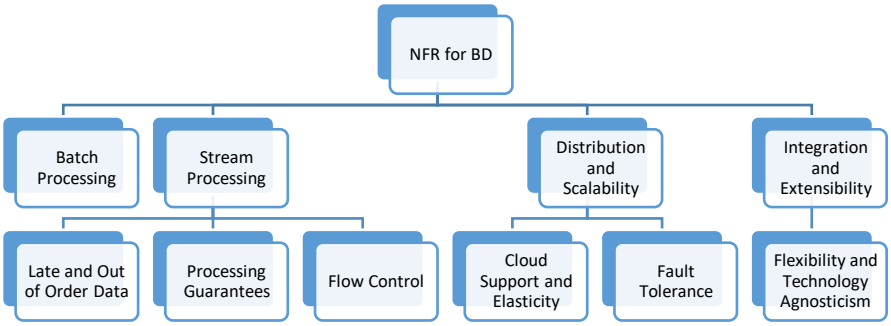

Fig. 18: Non-Functional Requirements for Large-Scale Big Data Systems

Figure 18 shows a summarised view of the ten non-functional requirements discussed in this section, organized hierarchically. At the top level, there are three main requirements: the capacity to process batch data, the capacity to process stream data, the capacity to distribute processing tasks across several machines and to scale up that number, and the capacity to seamlessly integrate with existing and future technology. Three other requirements are related to stream processing: the capacity to handle late and out of order data, the capacity to offer one of three processing guarantees, and the capacity to offer strategies for flow control. Two requirements are related to distribution and scalability: the capacity to offer cloud support and elasticity, so distribution can expand to encompass the ream of the cloud, and the capacity to offer fault tolerance, usually by taking snapshots of the state of data processing and re-launching the failed task on a healthy node. Finally, the capacity of a system to have its components substituted for equivalent technology is related to the requirement for integration and extensibility.

The remainder of this section explores how the three companies selected for this study implemented these ten requirements and compares their different solutions.

4.1 Batch Data

This requirement refers to the capability to process data which is finite and usually large in volume.

In terms of size, both Facebook and Twitter estimate that the finite data they hold on disk reaches hundreds of petabytes, with a daily processing volume of tens of petabytes [50]. Netflix's big data is one order of magnitude smaller, with tens of petabytes in store and daily reads of approximately 3 petabytes [51].

With regards to how this requirement is addressed, Facebook uses a combination of three independent, but communicating systems to manage its stored data: an Operational Data Store (ODS), Scuba, Hive and Laser [7].

Twitter's batch data is stored in Hadoop clusters and traditional databases, and is processed using Scalding and Presto [50]. Scalding is a Scala library developed in-house to facilitate the specification of map-reduce jobs [10]. Presto, on the other hand, was originally developed by Facebook. It was open-sourced in 2013 [52], and has since been adopted not only by Twitter, but also by Netflix [53].

Differently from the previous two companies, Netflix's Hadoop installation is cloud-based, and it uses an in-house developed system called Genie to manage query jobs submitted via Hadoop, Hive or Pig. Data is also persisted in Amazon S3 databases [54].

4.2 Stream Data

This requirement refers to the capability to process data which is potentially infinite and usually flowing at high velocity.

Stream processing at Facebook is done by a suite of in-house developed applications: Puma, Swift and Stylus. Puma is a stream processing application with a SQL-like query language optimised for compiled queries. Swift is a much simpler application, used for checkpointing. Finally, Stylus is a stream processing framework which combines stateful or stateless units of processing into more complex DAGs [7].

Storm, one of the most popular stream processing frameworks in use today, was developed by Twitter [4]. Less than five years after the initial release of Storm, however, Twitter announced that it had replaced it with a better performing system, Heron, and that Storm had been officially decommissioned [6]. Heron uses Mesos, an open-source cluster management tool designed for large clusters. It also uses Aurora, a Mesos framework developed by Twitter to schedule jobs on a distributed cluster.

Netflix also uses Mesos to manage its large cluster of cloud resources. Scheduling is done by a custom library called Fenzo, whereas stream processing is done by Mantis, which is also custom-developed.

4.3 Late and Out of Order Data

This requirement relates to stream processing and refers to the capability to process data which arrives late or in a different order from that in which it was emitted. All three streaming architectures utilise the concept of windows of data to transform infinite streaming data into finite windows that can be processed individually [5], [7], [55].

For handling late and out of order data, Facebook's Stylus utilises low watermarks. No mention was found in Twitter Heron's academic paper of whether it provides a mechanism for dealing with late or out of order data. However, looking at the source code for the Heron API, the BaseWindowedBolt class, merged into the master project in 2017, has a method called withLag(), which allows the developer to specify the maximum amount of time by which a record can be out of order [56].

No mention was found in documentation published by Netflix of Mantis's strategy for dealing with late and out of order data. Because the source code for Mantis is proprietary, further investigation was limited.

4.4 Processing Guarantees

This requirement refers to a stream system's capability to offer processing guarantees, i.e. exactly once, at least once and at most once. Exactly once semantics involves some level of checkpointing to persist state. There is therefore an inherent latency cost associated with it, which is why not all use-cases are implemented this way.

An example of a use-case where exactly once semantics is not a requirement is Facebook's Scuba system. Since the data is intended to be sampled, completeness of the data is not a requirement. Duplication, however, is not acceptable. In this case, at most once is a more fitting processing guarantee than exactly once [7], since it is in line with sampling and does not allow duplicate records to occur. Facebook also has use cases where exactly once processing guarantees are required. These are catered for by Stylus, a real-time system designed with optimisations to provide at least once processing semantics through the use of checkpointing. [7].

At Twitter, both Storm and its successor, Heron, offered at least once and at most once guarantees. Identified as a shortcoming by Kulkarni et al. [5], the lack of ex-

actly once semantics in Heron was subsequently addressed and implemented as "effectively once semantics". Effectively once semantics means that data may be processed more than once (the processing would undergo a rewind in case of failure), but it is only delivered once [57].

Netflix uses Kafka as its stream platform and messaging system [58], which means it provides inherent support for exactly once processing through idempotency and atomic transactions [59]. Additionally, at least once and at most once processing guarantees are also supported by Kafka [60].

4.5 Integration and Extensibility

This requirement refers to the capability to integrate with existing services and components. It also refers to provisions made to facilitate the extension of the existing architecture to incorporate different components in the future.

Although Facebook's real-time architecture is composed of many systems, they are integrated thanks to Scribe. Scribe works as a messaging system: all of Facebook's streaming systems write to Scribe, and they also read from Scribe. This allows for the creation of complex pipelines to cater for a multitude of use-cases [7]. In terms of extensibility, any service developed to use Scribe as data source and data output could integrate seamlessly with Facebook's architecture.

As part of a process to make Heron open-source, Twitter introduced a number of improvements to make it more flexible and adaptable to different infrastructures and use-cases. By adopting a general-purpose modular architecture, Heron achieved significant decoupling between its internal components, and increased its potential for adoption and extension by other companies [6].

Netflix's high level architecture is somewhat rigid in that there is no alternative to using Mesos as an orchestration and cluster management tool, or AWS as a cloud provider [61]. Additionally, Titus must run as a single framework on top of Mesos. This limitation however was introduced by design. With Titus running as a single framework on Mesos, it can allocate tasks more efficiently, and has full visibility of resources across the entire cluster [9]. Because Titus is a proprietary system designed by Netflix and optimised to fulfill its own use cases, it was initially tightly coupled to Netflix's infrastructure. It has however evolved into a more generic product since being open-sourced in April 2018 [62].

4.6 Distribution and Scalability

This requirement refers to the capability to distribute data processing amongst different machines, located in different data centres, in a multi-clustered architecture.

Dynamic scaling, which addresses the possibility of adding or removing nodes to a running system without any downtime, is also addressed as part of this requirement.

Scalability was one of the driving factors behind the development of Scribe as a messaging system at Facebook. Similarly to Kafka, Scribe can be scaled up by increasing the number of buckets (brokers) running, thus increasing the level of parallelism [7]. There is no mechanism in place for dynamic scaling of Puma and Stylus systems [7].

At Twitter, Heron was developed as a more efficient and scalable alternative to Storm. Heron, uses an in-house developed proprietary framework called Dhalion to help determine whether the cluster needs to be scaled up or down [63].

As Netflix's architecture is cloud-based, it is inherently elastic and scalable. Fenzo is responsible for dynamically scaling resources by adding or removing EC2 nodes to the Mesos infrastructure as needed [55].

4.7 Cloud Support and Elasticity

This requirement refers to the capability to move the architecture (or part of it) into the cloud to take advantage of the many benefits associated with its economies of scale.

Based on the material examined, Neflix's architecture is the only which is predominantly cloud-based. Having started with services running on AWS virtual machines, they are now undergoing a shift towards a container-based approach, with a few services now running in containers on AWS infrastructure [9]. Twitter has also undergone a shift towards a containerised architecture, albeit not cloud-based, with the development and implementation of Heron. As containers become more widespread, the risk of vendor lock-in is lowered, since containers enable the decoupling of the processing framework from the infrastructure they run in. Future migration to a safer multi-cloud setup is not only possible, but desirable [64].

4.8 Fault Tolerance

This requirement refers to the capability of a system to continue to operate should one or more nodes fail. Ideally, the system should recover gracefully, with minimal repercussions for the user experience.

Fault-tolerance is a requirement of Facebook's real-time systems, currently implemented through node independence and by using a persistent messaging system for all inter-system communication. Scribe, Facebook's messaging system, persists

data to disk, and is backed by Swift, a stream platform designed to provide checkpointing. [7].

At Twitter, fault tolerance is addressed at different levels. At architectural level, a modular distributed architecture provides better fault tolerance than a monolithic design. At container level, resource provisioning and job scheduling are decoupled, with the scheduler being responsible for monitoring the status of running containers and for trying to restart any failed ones, along with the processes they were running. At JVM level, Heron limits task processing to one per JVM. This way, should failure occur, it is much easier to isolate the failed task and the JVM where it was running [6]. At topology level, the management of running topologies is decentralised, with one Topology Master per topology, which means failure of one topology does not affect others [5].

As Netflix's production systems are cloud-based, fault tolerance is addressed from the perspective of a cloud consumer. The Active-Active project was launched by Netflix with the aim of achieving fault tolerance through isolation and redundancy by deploying services to the US across two AWS regions: US-East-1 and US-West-2 [65]. This project was later expanded to incorporate the EU-West-1 region, as European locations were still subjected to single points of failure [66]. With this latest development, traffic could be routed between any of the three regions across the globe, increasing the resilience of Netflix's architecture.

4.9 Flow Control

This requirement refers to the capability to handle scenarios where the data source emits records faster than the system can consume.

All real-time systems at Facebook read and write to Scribe. As described by Chen et al., this central use of a persistent messaging system makes Facebook's real-time architecture resilient to backpressure. Since nodes are independent, if one node slows down, the job is simply allocated to a different node, instead of the slowing down the whole pipeline [7]. The exact strategy used by Scribe to implement flow control is not made explicit in the paper.

Heron was designed with a flow control mechanism as an improvement over Storm, where producers dropped data if consumers were too busy to receive it. When Heron is in backpressure mode, the Stream Manager limits incoming data through the furthest upstream component (the spout) in order to slow down the flow of data throughout the topology. The data processing speed is thus reduced to the speed of the slowest component. Once backpressure is relieved and Heron exits backpressure mode, the spout is set back to emit records at its normal rate [5].

At Netflix, Mantis jobs are written using ReactiveX, a collection of powerful open-source reactive libraries for the JVM [67]. RxJava, one of the libraries in ReactiveX originally developed by Netflix, offers a variety of strategies for dealing with back-pressure such as, for example, the concept of a cold observable which only starts emitting data if it is being observed, at a rate controlled by the observer. For hot observables which emit data regardless of whether or not they are being observed, RxJava provides the options to buffer, sample, debounce or window the incoming data [68].

4.10 Flexibility and Technology Agnosticism

This criterion refers to the capability of an architecture to use interchangeable technology in place of existing components.

Out of the three architectures investigated, Facebook's setup is the least flexible and the least technologically agnostic. With the exception of Hive and its ODS, built on HBase [69], Facebook's data systems were developed in-house to cater for very specific use-cases. This is perhaps the reason why, at the time of writing, only Scribe has been made open-source [70]. It is worth noting, however, that the Scribe project was not developed further, and the source-code has been archived [13].

Heron's modular architecture is flexible by design, and the technologies chosen for Twitter's particular implementation, Aurora and Mesos, are not compulsory for other implementations. Heron's flexibility is evidenced by its adoption by large-scale companies such as Microsoft [71], and its technology agnosticism is evidenced by its successful implementation on a Kubernetes (instead of Mesos) cluster [72].

At programming level, Netflix is an active participant of the Reactive Streams initiative, which aims to standardise reactive libraries with an aim to rendering them interoperable. Considering that JDK 9, released in September 2017, is also compatible with Reactive Streams, there is potential for Mantis's jobs to be defined in standard Java in the future.

At cloud infrastructure level, the use of containers as a deployment abstraction reduces the tight coupling between Netflix's artifacts and specific virtual machine offerings provided by AWS. This is defined by Leung et al. [9] as a shift to a more application-centric deployment. It is worth noting, however, that, at the time of writing, Netflix officially relies on a single cloud provider: AWS, despite there being indication that they would have started to evaluate Google Cloud in an effort towards achieving a multi-cloud strategy [73].

At architecture level, because Titus was only recently open-sourced, this study did not evaluate whether essential parts of its architecture such as the Mantis, Fenzo or

the Mesos cluster could be replaced with an equivalent. It is expected, however, that its transition to open-source could attract important contributions from the community and enhance its flexibility and technology agnosticism.1

4.11 Summary and Applications

This section provides a summary of the implementation approaches of the ten non-functional requirements by the three companies selected. Additionally, it introduces a direct applications of the current study: the design and development of MC-BDP, a new reference architecture for large-scale stream big data processing.

As continuation of this research, the non-functional requirements discussed in this study were used to guide the design and implementation of a new reference architecture for big data processing in the cloud: MC-BDP. MC-BDP is an evolution of the PaaS-BDP architectural pattern originally proposed by the authors. While PaaS-BDP introduced a framework-agnostic programming model and enabled different frameworks to share a pool of location and provider-independent resources [64], MC-BDP expands this model by explicitly prescribing a multi-tenant environment where nodes are deployed to multiple clouds. Figure 19 shows a summary of how Facebook, Twitter and Netflix implemented the ten non-functional requirements discussed in this research. The last column shows MC-BDP, the proposed reference architecture.

Criteria / Architecture	Facebook	Twitter	Netflix	MC-BDP
Batch Data	ODS/Scuba/Hive/Laser	Scalding/Presto	Genie/Hadoop/Hive/Pig	flexible (integration framework)
Stream Data	Puma/Swift/Stylus	Heron	Mantis/Fenzo	flexible (integration framework)
Late and Out-of-Order Data	yes (low watermarks)	yes (record lag)	no information found	yes (low watermarks)
Processing Guarantees	exactly once*	effectively once	exactly once	exactly once
Integration and Extensibility	with Facebook technology	with a range of technologies (modular architecture)	with Netflix technology	with a range of technologies (modular architecture)
Distribution and Scalability	multiple data centres lack of dynamic scaling	multiple data centres Dhalion provides dyn. scaling	multiple AWS regions Fenzo provides dyn. scaling	multiple clouds framework provides dyn. scaling
Cloud Support and Elasticity	no	no	yes	yes
Multi-Cloud Support	no	no	no	yes
Fault Tolerance	architectural and node levels	architectural, container, JVM and topology levels	architectural level (cloud consumer perspective)	architectural, container and framework levels
Flow Control	via persistent messaging system (Scribe)	via Heron's backpressure mode	programmatically via ReactiveX	Kafka + big data framework
Flexibility and Technology Agnosticism	low (proprietary non-generic components)	high (modular architecture)	high at programming level (Reactive Streams)	high (modular architecture)

* not by all systems

Fig. 19: Summary of Non-Functional Requirements for Big Data and Implementations

MC-BDP was subsequently evaluated via a simulated energy efficiency case study where a prototype was developed using open-source technology to calculate the Power Usage Effectiveness (PUE) of a data centre at Leeds Beckett University. The components of this prototype implementation were deployed to the OSDC, Azure and Google clouds. Based on the non-functional requirements discussed in the current study, three hypotheses were formulated and verified empirically:

H1. MC-BDP is scalable across clouds.
H2. MC-BDP is fault-tolerant across clouds.
H3. MC-BDP's provision for technology agnosticism does not incur a significant increase in processing overhead.

This section examined how the three companies selected for this study: Facebook, Twitter and Netflix implemented the ten non-functional requirements defined in Section 2.2. Additionally, it introduced two instances where the present study was applied to inform the design and development of further contributions: MC-BDP and MC-Compose. A full presentation and discussion of MC-BDP and MC-Compose, however, lies outside the scope of this chapter, and will be the subject of a future publication. The next section presents the conclusion to this work and suggestions for future work.

5. Conclusion and Future Work

This study presented the results of a literature search for non-functional requirements relevant to real-world big-data implementations. Three companies were selected for this comparative study: Facebook, Twitter and Netflix. Their specific implementations of the non-functional requirements selected were compared and discussed in detail, and are summarised in this section.

Facebook and Twitter process the largest volume of data, with Twitter having the lowest requirement for latency. Differently from Facebook, these two architectures were also explicitly designed to handle late and out of order data. In terms of processing guarantees, all three architectures support exactly-once semantics.

Although the existing systems at Facebook and Netflix are integrated, they were not designed as a unified modular framework. Heron, on the other hand, was developed by Twitter as an improvement over Storm, which suffered from bottlenecks and single points of failure. Heron's modular architecture makes it more flexible and technologically agnostic, as well as a stronger candidate for adoption by other companies when compared to systems developed by the other two companies.

Differently from Facebook and Twitter, which provide mechanisms for scalability and fault tolerance in their infrastructures, Netflix approaches this concept from a cloud consumer's perspective, since its architecture is cloud-based. Netflix's deployments are distributed over multiple regions, although support for multi-cloud is still lacking.

All three architectures provide mechanisms for flow control. Facebook and Twitter control backpressure from an infrastructure level, whereas Netflix provides methods and constructs to achieve this programmatically.

The authors recognise that more thorough results could have been obtained should our approach have included direct observation of the systems under evaluation by way of a set of case studies. However, due to time and resource constraints, the scope of the present study was limited to published sources.
Future work shall involve a prototype implementation of the MC-BDP reference architecture and its subsequent evaluation in terms of a minimum of three of the non-functional requirements for large-scale big data applications identified in this study. Additionally, this research aims to develop one or more of the functional services for big data requirements engineering in a multi-cloud environment described in the previous section. An example of such service is MC-Compose, a cloud resource estimation service for stream big data systems which adjusts user-entered estimations based on the windowing function selected and on monitoring feedback.

Acknowledgments
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 **Google Cloud** and **Microsoft Azure** for Research awards.

Container and cloud native technologies were provided by **Weaveworks**.

6. References

[1] L. Cao, 'Data science: challenges and directions', *Commun. ACM*, vol. 60, no. 8, pp. 59–68, Jul. 2017.
[2] J. Desjardins, 'What Happens in an Internet Minute in 2019?', *Visual Capitalist*, 13-Mar-2019. [Online]. Available: https://www.visualcapitalist.com/what-happens-in-an-internet-minute-in-2019/. [Accessed: 22-Mar-2019].
[3] L. Chung and J. C. Prado Leite, 'Conceptual Modeling: Foundations and Applications', A. T. Borgida, V. K. Chaudhri, P. Giorgini, and E. S. Yu, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 363–379.
[4] A. Toshniwal *et al.*, 'Storm@Twitter', in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2014, pp. 147–156.

[5] S. Kulkarni *et al.*, ‘Twitter Heron: Stream Processing at Scale’, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2015, pp. 239–250.

[6] M. Fu *et al.*, ‘Twitter Heron: Towards Extensible Streaming Engines’, in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 1165–1172.

[7] G. J. Chen *et al.*, ‘Realtime Data Processing at Facebook’, in *Proceedings of the 2016 International Conference on Management of Data*, New York, NY, USA, 2016, pp. 1087–1098.

[8] N. Bronson, T. Lento, and J. L. Wiener, ‘Open data challenges at Facebook’, in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 1516–1519.

[9] A. Leung, A. Spyker, and T. Bozarth, ‘Titus: Introducing Containers to the Netflix Cloud’, *Queue*, vol. 15, no. 5, pp. 30:53–30:77, Oct. 2017.

[10] *scalding: A Scala API for Cascading*. Twitter, Inc., 2018.

[11] ‘Heron Documentation - Heron’s Architecture’, *Heron Documentation*, 2019. [Online]. Available: https://apache.github.io/incubator-heron/docs/concepts/architecture/. [Accessed: 02-Jun-2019].

[12] P. T. Goetz, J. Lim, K. Patil, and P. Brahmbhatt, *Apache Storm*. The Apache Software Foundation, 2019.

[13] *Scribe*. Facebook Archive, 2014.

[14] S. Eliot, ‘Microsoft Cosmos: Petabytes perfectly processed perfunctorily’, 11-May-2010. [Online]. Available: https://blogs.msdn.microsoft.com/seliot/2010/11/05/microsoft-cosmos-petabytes-perfectly-processed-perfunctorily/. [Accessed: 24-Jan-2018].

[15] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, ‘Orleans: Distributed Virtual Actors for Programmability and Scalability’, Mar. 2014.

[16] T. Akidau *et al.*, ‘MillWheel: fault-tolerant stream processing at internet scale’, *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013.

[17] T. Akidau *et al.*, ‘The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing’, *Proc. VLDB Endow.*, vol. 8, pp. 1792–1803, 2015.

[18] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs, ‘Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander’, in *2015 IEEE International Congress on Big Data*, 2015, pp. 592–599.
[19] D. J. Abadi *et al.*, ‘The Design of the Borealis Stream Processing Engine.’, in *CIDR*, 2005, vol. 5, pp. 277–289.
[20] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, ‘Stormy: an elastic and highly available streaming service in the cloud’, 2012, p. 55.
[21] A. Alexandrov *et al.*, ‘The Stratosphere Platform for Big Data Analytics’, *VLDB J.*, vol. 23, no. 6, pp. 939–964, Dec. 2014.
[22] J. Y. Zhu, J. Xu, and V. O. K. Li, ‘A Four-Layer Architecture for Online and Historical Big Data Analytics’, 2016, pp. 634–639.
[23] ‘Amazon EMR - Amazon Web Services’, *Amazon EMR*, 2019. [Online]. Available: https://aws.amazon.com/emr/. [Accessed: 15-Mar-2019].
[24] ‘Azure HDInsight - Hadoop, Spark, & Kafka Service | Microsoft Azure’, *HDInsight*, 2019. [Online]. Available: https://azure.microsoft.com/en-gb/services/hdinsight/. [Accessed: 15-Mar-2019].
[25] ‘Big Data Analytics Infrastructure Solutions | IBM’, *IBM big data analytics solutions*, 2019. [Online]. Available: https://www.ibm.com/it-infrastructure/solutions/big-data. [Accessed: 15-Mar-2019].
[26] B. Chandramouli, J. Goldstein, M. Barnett, and J. F. Terwilliger, ‘Trill: Engineering a Library for Diverse Analytics’, *IEEE Data Eng Bull*, vol. 38, pp. 51–60, 2015.
[27] S. A. Noghabi *et al.*, ‘Samza: Stateful Scalable Stream Processing at LinkedIn’, *Proc VLDB Endow*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017.
[28] T. Akidau, S. Chernyak, and R. Lax, *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*, 1 edition. Beijing Boston Farnham Sebastopol Tokyo: O’Reilly Media, 2018.
[29] S. M. A. Akber, C. Lin, H. Chen, F. Zhang, and H. Jin, ‘Exploring the impact of processing guarantees on performance of

stream data processing', in *2017 IEEE 17th International Conference on Communication Technology (ICCT)*, 2017, pp. 1286–1290.

[30] B. Satzger, W. Hummer, C. Inzinger, P. Leitner, and S. Dustdar, 'Winds of Change: From Vendor Lock-In to the Meta Cloud', *IEEE Internet Comput.*, vol. 17, no. 1, pp. 69–73, Jan. 2013.

[31] J. Brodkin, 'Amazon EC2 outage calls "availability zones" into question', *Network World*, 21-Apr-2011. [Online]. Available: https://www.networkworld.com/article/2202805/cloud-computing/amazon-ec2-outage-calls--availability-zones--into-question.html. [Accessed: 22-Feb-2019].

[32] A. Dayaratna, 'Microsoft Azure Recovers From Multi-Region Azure DNS Service Disruption', *Cloud Computing Today*, 15-Sep-2016. [Online]. Available: https://cloud-computing-today.com/2016/09/15/microsoft-azure-recovers-from-multi-region-azure-dns-service-disruption/. [Accessed: 22-Feb-2019].

[33] N. H. Madhavji, A. Miranskyy, and K. Kontogiannis, 'Big Picture of Big Data Software Engineering: With Example Research Challenges', in *2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering*, 2015, pp. 11–14.

[34] D. Arruda, 'Requirements Engineering in the Context of Big Data Applications', *ACM SIGSOFT Softw. Eng. Notes*, vol. 43, no. 1, pp. 1–6, Mar. 2018.

[35] I. Noorwali, D. Arruda, and N. H. Madhavji, 'Understanding Quality Requirements in the Context of Big Data Systems', in *2016 IEEE/ACM 2nd International Workshop on Big Data Software Engineering (BIGDSE)*, 2016, pp. 76–79.

[36] H. Eridaputra, B. Hendradjaya, and W. D. Sunindyo, 'Modeling the requirements for big data application using goal oriented approach', in *2014 International Conference on Data and Software Engineering (ICODSE)*, 2014, pp. 1–6.

[37] N. Al-Najran and A. Dahanayake, 'A Requirements Specification Framework for Big Data Collection and Capture', in *New Trends in Databases and Information Systems*, 2015, pp. 12–19.

[38] N. Al-Najran, 'A Requirements Specification Framework for Big Data Collection and Capture', Masters of Science in Software Engineering, Prince Sultan University, Riyadh, 2015.

[39] D. Arruda and N. H. Madhavji, ‘Towards a requirements engineering artefact model in the context of big data software development projects: Research in progress’, in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 2314–2319.

[40] A. G. Shoro and T. R. Soomro, ‘Big Data Analysis: Apache Spark Perspective’, *Glob. J. Comput. Sci. Technol.*, vol. 15, no. 1, Feb. 2015.

[41] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, ‘Lambda architecture for cost-effective batch and speed big data processing’, 2015, pp. 2785–2792.

[42] B. Sun, L. Zhang, and Y. Chen, ‘Design of big data processing system for spacecraft testing experiment’, in *2017 7th IEEE International Symposium on Microwave, Antenna, Propagation, and EMC Technologies (MAPE)*, 2017, pp. 164–167.

[43] N. Naik, ‘Docker container-based big data processing system in multiple clouds for everyone’, in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7.

[44] M. Villari, A. Celesti, M. Fazio, and A. Puliafito, ‘AllJoyn Lambda: An architecture for the management of smart environments in IoT’, 2014, pp. 9–14.

[45] P. Basanta-Val, N. C. Audsley, A. Wellings, I. Gray, and N. Fernandez-Garcia, ‘Architecting Time-Critical Big-Data Systems’, *IEEE Trans. Big Data*, vol. PP, no. 99, pp. 1–1, 2016.

[46] G. Guerreiro, P. Figueiras, R. Silva, R. Costa, and R. Jardim-Goncalves, ‘An architecture for big data processing on intelligent transportation systems. An application scenario on highway traffic flows’, 2016, pp. 65–72.

[47] C. Costa and M. Y. Santos, ‘BASIS: A big data architecture for smart cities’, 2016, pp. 1247–1256.

[48] M. Ramachandran, ‘Business Requirements Engineering for Developing Cloud Computing Services’, in *Software Engineering Frameworks for the Cloud Computing Paradigm*, Z. Mahmood and S. Saeed, Eds. London: Springer London, 2013, pp. 123–143.

[49] M. Ramachandran and Z. Mahmood, Eds., *Requirements Engineering for Service and Cloud Computing*. Springer International Publishing, 2017.

[50] S. Krishnan, ‘Discovery and Consumption of Analytics Data at Twitter’, 29-Jun-2016.
[51] T. Gianos and D. Weeks, ‘Petabytes Scale Analytics Infrastructure @Netflix’, presented at the QCon, San Francisco, 11-Aug-2016.
[52] J. Pearce, ‘2013: A Year of Open Source at Facebook’, *Facebook Code*, 20-Dec-2013. [Online]. Available: https://code.facebook.com/posts/604847252884576/2013-a-year-of-open-source-at-facebook/. [Accessed: 12-Feb-2018].
[53] E. Tse, Z. Luo, and N. Yigitbasi, ‘Using Presto in our Big Data Platform on AWS’, *The Netflix Tech Blog*, 10-Jul-2014. .
[54] S. Krishnan and E. Tse, ‘Hadoop Platform as a Service in the Cloud’, *The Netflix Tech Blog*, 10-Jan-2013. .
[55] B. Schmaus, C. Carey, N. Joshi, N. Mahilani, and S. Podila, ‘Stream-processing with Mantis’, *Netflix TechBlog*, 14-Mar-2016.
[56] B. Peng, *[ISSUE-1124] - Windows Bolt support #2241*. Twitter, Inc., 2017.
[57] ‘Heron Documentation - Heron Delivery Semantics’. 2019.
[58] S. Wu *et al.*, ‘The Netflix Tech Blog: Evolution of the Netflix Data Pipeline’, 15-Feb-2016. [Online]. Available: http://techblog.netflix.com/2016/02/evolution-of-netflix-data-pipeline.html. [Accessed: 30-Oct-2016].
[59] A. Woodie, ‘A Peek Inside Kafka’s New “Exactly Once” Feature’, *Datanami*, 07-Mar-2017.
[60] P. Dobbelaere and K. S. Esmaili, ‘Kafka Versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper’, in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, New York, NY, USA, 2017, pp. 227–238.
[61] ‘Titus’, *Titus Documentation*, 2018. [Online]. Available: https://netflix.github.io/titus/. [Accessed: 18-Mar-2019].
[62] A. Joshi *et al.*, ‘Titus, the Netflix container management platform, is now open source’, *Medium*, 18-Apr-2018.
[63] B. Graham, ‘From Rivulets to Rivers: Elastic Stream Processing in Heron’, 16-Mar-2017.
[64] T. Vergilio and M. Ramachandran, ‘PaaS-BDP - A Multi-Cloud Architectural Pattern for Big Data Processing on a Platform-as-

a-Service Model', in *Proceedings of the 3nd International Conference on Complexity, Future Information Systems and Risk*, Madeira, 2018.

[65] R. Meshenberg, N. Gopalani, and L. Kosewski, 'Active-Active for Multi-Regional Resiliency', *Netflix TechBlog*, 02-Dec-2013. .

[66] P. Stout, 'Global Cloud—Active-Active and Beyond', *Netflix TechBlog*, 30-Mar-2016.

[67] B. Christiansen and J. Husain, 'Reactive Programming in the Netflix API with RxJava', *Netflix TechBlog,* 04-Dec-2013. .

[68] D. Gross and D. Karnok, 'Backpressure', *ReactiveX/RxJava Wiki,* 27-Jun-2016. [Online]. Available: https://github.com/ReactiveX/RxJava/wiki/Backpressure. [Accessed: 15-Feb-2018].

[69] L. Tang, 'Facebook's Large Scale Monitoring System Built on HBase', presented at the Strata Conference + Hadoop World, New York, NY, USA, 24-Oct-2012.

[70] R. Johnson, 'Facebook's Scribe technology now open source', *Facebook Code,* 24-Oct-2008.

[71] K. Ramasamy, 'Open Sourcing Twitter Heron', *Twitter Engineering Blog,* 25-May-2016.

[72] C. Kellogg, 'The Heron Stream Processing Engine on Google Kubernetes Engine', *Streamlio,* 28-Nov-2017.

[73] K. McLaughlin, 'Netflix, Long an AWS Customer, Tests Waters on Google Cloud', *The Information,* 17-Apr-2018. [Online]. Available: https://www.theinformation.com/articles/netflix-long-an-aws-customer-tests-waters-on-google-cloud. [Accessed: 18-Mar-2019].