APPLICATION OF MICROSERVICES PATTERNS TO BIG DATA SYSTEMS

Complete Research

Abstract

The panorama of data is ever evolving, and big data has emerged to become one of the most hyped terms in the industry. Today, users are the perpetual producers of data that if gleaned and crunched, will yield game-changing patterns. This has introduced an important shift about the role of data in organizations and many strived to harness to power of this new material. Howbeit, institutionalizing data is not an easy task and requires the absorption of a great deal of complexity. According to a recent survey, it is estimated that only 13% of organizations succeeded in delivering on their data strategy. Among the root challenges, big data system development and data architecture are prominent. To this end, this study aims to facilitate data architecture and big data system development by applying well-established patterns of microservices architecture to big data systems. This objective is achieved by two systematic literature reviews, and infusion of results through thematic synthesis. The result of this work is a series of theories that explicates how microservices patterns could be useful for big data systems. These theories are then validated through a semi-structured interview with 7 experts from the industry. The findings emerged from this study indicates that big data architecture can benefit from many principles and patterns of microservices architecture.

Keywords: big data, microservices, microservices patterns, big data architecture, data architecture, data engineering

1 Introduction

Today, we live in a world that produces data at an unprecedented rate. The attention toward these large volume of data has been growing rapidly and many strive to harness the advantages of this new material. While the opportunities exist with big data (BD), there are many failed attempts. According to Davenport and Bean (2021) in 2022, only 26.5% of companies successfully became data-driven. Another survey by Databricks (2021) highlighted that only 13% of organizations succeeded in delivering on their data strategy. Among the challenges of adopting BD, data architecture, organizational culture, lack of talent, and rapid technology change are bold.

Therefore, there is an increasing need for more research on reducing the complexity involved with BD projects. One area with good potential is data architecture. Data architecture allows for a flexible and

scalable BD system that can account for emerging requirements. One concept that has the potential to help with development of BD systems is the use of microservices (MS) architecture. MS architecture allows for division of complex applications into small, independent, and highly scalable parts and, therefore, increases maintainability and allows for a more flexible implementation (Richardson, 2022, p. 20). Nevertheless, design and development of MS is sophisticated, since heterogenous services have to interact with each other to achieve the overall goal of the system. One way to reduce that complexity is the use of patterns. Patterns are proven artifacts on how certain problems can be solved. Despite the potential of MS architectural patterns to solve some of complexities of BD development, to our knowledge, there is no study that properly bridges these two concepts.

To this end, this study aims to explore the application of MS patterns to BD systems, in aspiration to solve some of the complexities of BD system development. For this purpose, the result of two distinct systematic literature reviews (SLRs) are combined. The first SLR is conducted as part of this study to collect all MS patterns in the body of knowledge. The results of these SLRs are collected, captured and combined through thematic synthesis. As a result, various design theories are generated and validated through a semi-structured interview. The second SLR is done by Ataei and Litchfield (2022) to find all BD reference architectures (RAs) available in the body of knowledge and to point out architectural constructus and limitations.

The contribution of this study, is thereby threefold: 1) it generates numerous design theories and validate them through expert opinion, 2) it assembles an overview of relevant microservice patterns and, most importantly, 3) it creates a connection between the two SLRs to facilitate BD system development and architecture.

2 Related Work

To the best of our knowledge, there is no study in academia that has shared the same goal as our study. Laigner et al. (2020) applied an action research and reported on their experience on replacing a legacy BD system with a microservice based event-driven system. This study is not a systematic review and aims to create contextualized theory based on a specific experience. In another effort, Zhelev and Rozeva (2019) described why event-driven architectures could be a good alternative to monolithic architectures. This study does not follow any clear methodology, and seems to contribute only in terms of untested theory.

Staegemann et al. (2021) examined the interplay between BD and MS by conducting a bibliometric review. This study aims to provide with a general picture of the topic, and does not aim to explore MS patterns and their relationship to BD systems. While the problem of BD system development has been approached through a RA that absorbs some of the concepts from MS as seen in Neomycelia (Ataei and Litchfield, 2021) and Phi (Maamouri, Sfaxi, and Robbana, 2021), there is no study that aimed to apply MS patterns to BD systems through a systematic synthesis.

3 Methodology

To get a comprehensive overview over both domains it was decided to combine the results of two SLRs, one for each domain. Both SLRs are conducted following the guidelines presented in Kitchenham, Dyba, and Jorgensen (2004) and Page et al. (2021) on Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA). The former was used because of its clear instructions on critically appraising evidence for validity, impact and applicability in software engineering and the latter was used because it is a comprehensive methodology for increasing systematicity, transparency, and prevention of bias. To synthesize our findings, thematic synthesis proposed by Cruzes and Dyba (2011) was applied.

3.1 First Review

The first SLR, was a rigorous approach from scratch and was conducted in six major phases. These phases are described as following:

Selecting data sources and developing a search strategy: To assure the comprehensiveness of the review, both meta databases and publisher bound registers have been chosen. For this study, ACM Digital Library, AISeL, IEEE Xplore, JSTOR, Science Direct, Scopus, Springer Link, and Wiley were included into the search process. For all of these, the initial keyword search was conducted on June 19, 2022, and there was no limitation to the considered publishing date.

Since there are differences in the filters of the included search engines, it was not possible to always use the exact same search terms and settings. Nevertheless, the configurations for the search were kept as similar as possible. The exact keywords and search strategy used can be found at *Systematic literature review search terms table for the Paper Titled: Application of Microservices Patterns to Big Data Systems* (2022).

Developing inclusion, exclusion criteria and the quality framework: Inspired by PRISMA checklist presented by Tricco et al. (2018), Our inclusion and exclusion criteria are formulated as following:

Inclusion Criteria: 1) Primary and secondary studies between Jan 1st 2012 and June 19th 2022, 2) The focus of the study is on MS patterns, and MS architectural constructs, 3) Scholarly publications such as conference proceedings and journal papers.

Exclusion Criteria: 1) Studies that are not written in English, 2) Informal literature surveys without any clearly defined research questions or research process, 3) Duplicate reports of the same study (a conference and journal version of the same paper). In such cases, the conference paper was removed. 4) Complete duplicates (not just updates) were also removed. 5) Short papers (less than 6 pages).

Quality Framework: Quality of the evidence collected as a result of this SLR has direct impact on the quality of the findings, making quality assessment an important undertaking. To address this, we developed a criteria made up of 7 elements. These criteria are informed by guidelines provided by Kitchenham, Dyba, and Jorgensen (2004) on empirical research in software engineering. These 7 criteria are applied in three different stages (quality gates). These 7 criteria are discussed in Table 1.

Table 1. The quality framework

Quality Gate	Criterion	Considered Aspect	Rating to pass
1	Minimum quality threshold	1) Does the study report empirical research or is it merely a 'lesson learnt' report based on expert opinion? 2) The objectives and aims of the study are clearly communicated, including the reasoning for why the study was undertaken? 3) Does the study provide with adequate information regarding the context in which the research was carried out?	5/6
2	Rigor	1) Is the research design appropriate to address the objectives of the research? 2) Is there any data collection method used and is it appropriate?	3/4
3	Credibility Relevance	1) Does the study report findings in a clear and unbiased manner? 2) Does the study provide value for practice or research?	3/4

Pooling literature and evaluation: Overall, the keyword search yielded 3064 contributions. The total number of found publications per source as well as an overview of the further search process can be seen in Figure 1.

In the initial phase, 1,196 papers have been removed due to duplication and publication type. The remaining 1868 papers were filtered by title to evaluate their relevance to the concepts of

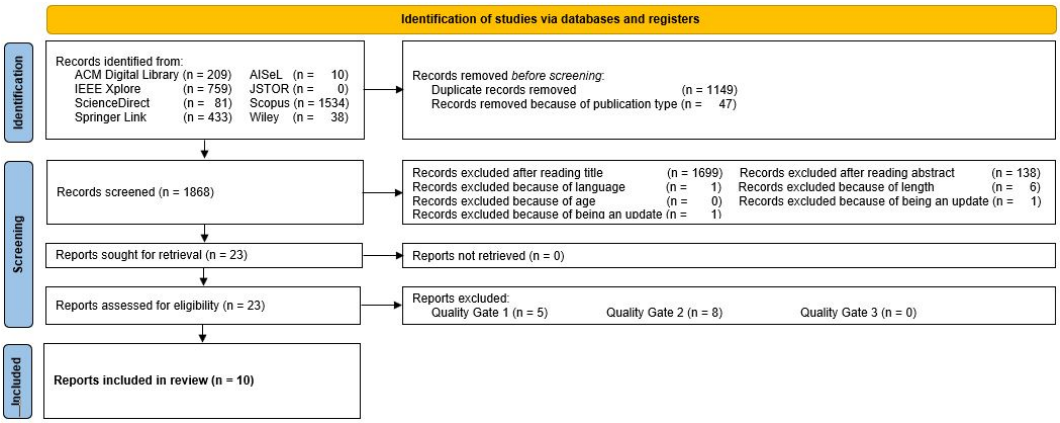

Figure 1. Overview of the search process

microservice patterns or architectural constructs related to MS. For this purpose, the first two authors separately evaluated each entry. If both agreed, the verdict was honored. In case of disagreement, they discussed the title to come to a conclusion. In this phase 1699 papers were excluded. Next, The same approach was followed for abstracts. Out of 169 remaining papers, 138 papers has been exluceded. From there on, the papers that were not written in English (despite the abstract being in English), were published before the year 2012, and had a length of less than six pages were removed. 23 papers have been selected for the quality assement against the quality framework. The agreement rate among researchers for this phase equates to 88 percent.

In the first phase, the aim was to ensure that reports fulfill at least a desired minimum level of comprehensiveness. Authors independently rated the three aspects for all 23 remaining papers, giving one point respectively, if they deemed a criterion fulfilled and no point if they considered that aspect lacking. For passing the first phase at least 5/6 points required. In this phase 5 papers have been exlcuded. In the second phase, studies were judged based on their research design and the data collection methods. Minimum rating to pass was 3/4 and 8 papers has been excluded. Finally, the remaining 10 papers went through the third phase. Here, the credibility of the reporting and the relevance of the findings were evaluated. No paper has been removed in this phase.

The overall agreement rate between researchers is 71.6 percent. All ten studies selected have been published in 2018 or later, with three of them being published in 2022, which shows the timeliness of the topic. Eight of the ten papers were found via Scopus, whereas the remaining two have been identified through IEEE Xplore. These papers are described in *Papers found for the paper: Application of microservices patterns to big data systems* (2022).

Data Synthesis: After selecting the quality papers, we embarked on the data synthesis process. For this phase we follow the guidelines of thematic synthesis discussed by Cruzes et al. Cruzes and Dyba (2011). To begin, we first extracted the following data from each paper: 1) findings, 2) research motivation, 3) author, 4) title, 5) research objectives, 6) research method, 7) year. We extracted these data through coding, using the software Nvivo. After that, we created two codes: 1) patterns, and 2) BD requirements, and coded the findings based on it. By the end of this process, various themes emerged.

3.2 Second Review

The second SLR is conducted by Ataei and Litchfield (2022) on available BD RAs in academia and industry. This is comprehensive study that covers various aspects of BD RAs such as limitations, and

common architectural blocks. The findings from this SLR helped us shaped the requirements necessary for BD systems. We do not explore this SLR in this paper, and aim to only discuss the results of it.

4 Results

In this section, we present with three integral elements: 1) BD software and system requirements, 2) MS patterns, 3) the mapping between the two and theories that emerged as a result.

4.1 Big Data Software and System Requirements

While we could find BD major building blocks and system requirements by coding the findings gained from Ataei and Litchfield (2022) work, our coding process did not include categorization and representation of these requirements. To this end, we performed a lightweight literature review in the body of knowledge to realize what type of requirement is most suitable for our study, how these requirements are categorised and what's the best way to present them. The results are as following:

1. **Type of requirements:** For the purposes of this study, we opted for a well-received approach discussed by P. A. Laplante (2017). In this approach, requirements are classified into three major types of 1) functional requirements, 2) non-functional requirements, and 3) domain requirements. Our primary focus is one functional and domain requirements.
2. **Categorizing Requirements:** For categorization of requirements, we followed the well-established categorization method based on BD characteristic, that is the 5Vs. These 5Vs are velocity, veracity, volume, variety and value (Rad and Ataei, 2017). We took inspiration from various studies such as the work of Nadal et al. (2017), and the requirements categories presented in NIST BD Public Working Group (Chang and Grady, 2019). This resulted in addition of 'security and privacy' to the category of requirements. We do not aim to define these characteristics in this study. Explanation of these characteristics can be found in the works of Rada et al. (2017).
3. **Presenting Requirements:** For the purposes of this study, we opted for an informal method because it is a well established method in the industry and academia (Kassab, Neill, and P. Laplante, 2014). Our approach follows the guidelines explained in ISO/IEC/IEEE standard 29148 (2018) for representing functional requirements. These requirements are described in Table 2.

4.2 Microservice Patterns

As a result of this SLR, our data synthesis yielded 28 MS patterns. These patterns are classified based on their function and the problem they solve. While we elaborate the patterns adopted for BD requirements in detail, the aim of our study is not to explain each MS pattern. Nevertheless, the name of all patterns found and their category can be found in *Microservices patterns classified for the Paper Titled: Application of Microservices Patterns to Big Data Systems* (2022). Additionally the definition of these patterns with examples can be found in the works of Richardson (2018).

5 Application of Microservices Design Patterns to Big Data Systems

In this section, we combine our findings from both SLRs, and present new theories on application of MS design patterns for BD systems. The patterns gleaned are established theories that are derived from actual problems in MS systems in practice, thus we do not aim to re-validate them in this study. The main contribution of our work is to propose new theories in regards to application of MS patterns to BD systems. To achieve this, we map BD system requirements against a pattern and provide with reasoning on why such pattern might work for BD systems. We support our arguments by the means of modeling.

Table 2. BD system requirements

Volume	**Vol-1)** System needs to support asynchronous, streaming, and batch processing to collect data from centralized, distributed, and other sources, **Vol-2)** System needs to provide a scalable storage for massive data sets
Velocity	**Vel-1)** System needs to support slow, bursty, and high-throughput data transmission between data sources, **Vel-2)** System needs to stream data to data consumers in a timely manner, **Vel-3)** System needs to be able to ingest multiple, continuous, time varying data streams, **Vel-4)** System should be able to process data in real-time or near real-time manner
Variety	**Var-1)** System needs to support data in various formats ranging from structured to semi-structured and unstructured data, **Var-2)** System needs to support aggregation, standardization, and normalization of data from disparate sources, **Var-3)** System shall support adaptations mechanisms for schema evolution, **Var-4)** System can provide mechanisms to automatically include new data sources
Value	**Val-1)** System needs to able to handle compute-intensive analytical processing and machine learning techniques, **Val-2)** System needs to support two types of analytical processing: batch and streaming, **Val-3)** System needs to support different output file formats for different purposes, **Val-4)** System needs to support streaming results to the consumers
Security & Privacy	**SaP-1)** System needs to protect and retain privacy and security of sensitive data, **SaP-2)** System needs to have access control, and multi-level, policy-driven authentication on protected data and processing nodes.
Veracity	**Ver-1)** System needs to support data quality curation including classification, pre-processing, format, reduction, and transformation, **Ver-2)** System needs to support data provenance including data life cycle management and long-term preservation.

We use Archimate as our architectural description language. This is recommend in ISO/IEC/IEEE 42010 (Chaabane, Bouassida, and Jmaiel, 2017).
We posit that a pattern alone would not be significantly useful to a data engineering or a data architect, and propose that a collection of patterns in relation to current defacto standard of BD architectures is a better means of communication. To achieve this, we provide theories on how MS patterns can address BD requirements. Additionally, we portray selected patterns in a reference architecture.

5.1 Volume

To address the volume requirements of BD , and in specific for Vol-1 and Vol-2 we suggest the following patterns to be effective; 1) Gateway offloading, 2) API gateway, 3) External Configuration Store

1.Gateway Offloading and API Gateway: In a typical flow of data engineering, data goes from ingestion, to storage, to transformation and finally to serving. However there are various challenges to maintain this. One challenge is the realization of various data sources as described in Vol-1. Data comes in various formats and system needs to handle different data through different interfaces.
Given the challenges and particularities of data types, different nodes may be spawned to handle the volume of data. This can be problematic as different nodes will need to account for cross-cutting concerns such as certificate management, throttling, logging, monitoring and authentication. If each node reimplement the same interface for the cross-cutting concerns, scalability and maintainability can be a daunting task. This also introduces unnecessary repetition of codes and can result in incompatible interfaces. To solve this problem, we explore the concept of gateway offloading. Offloading cross-cutting concerns to a single architectural construct, achieves 'separation of concerns'. This simplifies development and removes the need for re-implementing cross-cutting concerns. In addition, it introduces consistency in terms of logging

and monitoring and allow teams to work on things that they are specialized in.
Moreover, if data producers directly communicate with the processing nodes, they will have to update the endpoint address every now and on. This issue is exacerbated when the data producer tries to communicate to a service that is down. Given that lifecycle of a service in a typical distributed cloud environment is not deterministic and many container orchestration systems constantly recycle services to proactively address this issue, reliability and maintainability of the BD system can be compromised. This also increases round-trips, and introduces security issues. To address this, we introduce the API gateway pattern. API gateway can be the single point of entry to the system, providing with request aggregation, gateway routing and proxying. This is an effective pattern in tackling cross-cutting concerns specially in distributed system with many nodes. These pattern are portrayed in Figure 2.

2. External Configuration Store: As discussed earlier, BD systems are made up of various services in order to achieve horizontal scalability. These services will have to communicate with each other in order to achieve the goal of the system. Thus, each one of them will require a set of runtime environmental configurations. These configurations could be database network locations, feature flags, and third party credentials. Moreover, different stages of the data engineering may have different environments for different purposes. For instance, privacy engineers may require a completely different environment to achieve their requirements.
Therefore, the challenge is the management of these configurations as the system scales, and enabling services to run in different environments without modification. To address this problem, we propose the external configuration store pattern. By externalizing nodes configuration to another service, each node can request its configuration from an external store on boot up. This pattern solves the challenges of handling large number of nodes in BD systems and provide with a scalable solution for handling configurations. This pattern is portrayed in Figure 2.

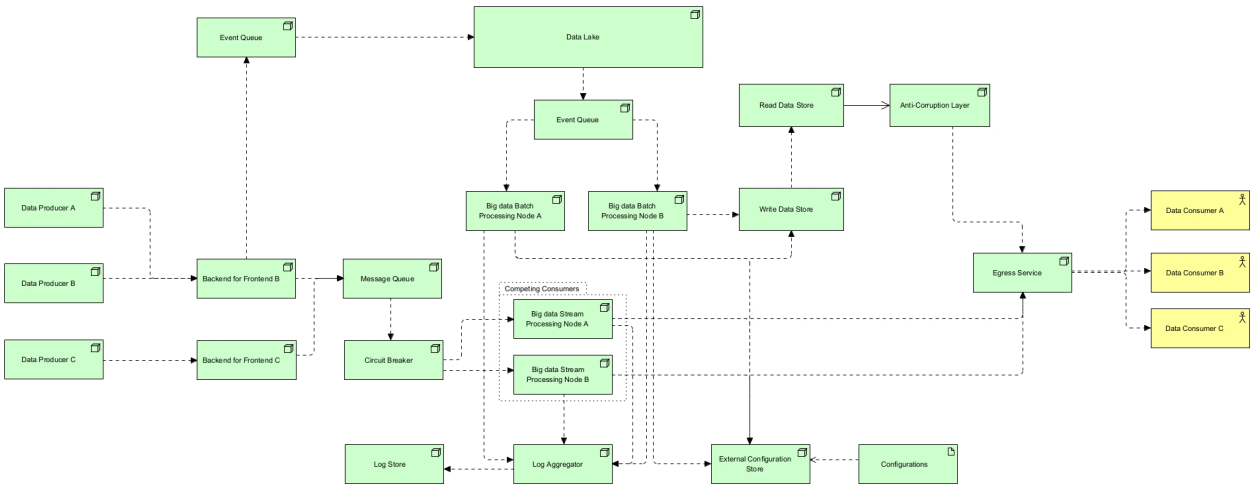

Figure 2. Big data reference architecture with microservices patterns

5.2 Velocity

To address some of the challenges associated with the velocity aspect of BD systems, we recommend the following patterns for the requirements Vel-1, Vel-2, Vel-3, and Vel-4; 1) Competing Consumers, 2) Circuit Breaker and 3) Log Aggregation.

1. Competing Consumers: According to a recent MIT report in collaboration with Databricks, one of the main challenges of BD 'low-achievers' is the 'slow processing of large amounts of data' (Databricks, 2021). If the business desires to go data driven, it should be able to have an acceptable time-to-insight.

Data needs to be ingested quickly, stored in a timely manner, batch, or stream processed, and served to the consumers. So what happens if one node goes down or becomes unavailable? The system has to wait for the node to become available, and any workload for stream processing comes to halt, failing to achieve requirements Vel-2, Vel-3 and Vel-4. This issue is exacerbated if the system is designed and architected underlying monolithic pipeline architecture with point-to-point communications. One way to solve some of these issues, is to introduce a non-blocking asynchronous event driven communication style to increase fault tolerance and availability. In this communication style, there's a producer that produces the event, the message queueing system that transfers the message and a consumer that consumes the message. This communication is logically separated in topics.
While this communication may improve the availability and idle time, it can fail due to sudden increase in the number of requests. This is because the consumer that receives the message may not be unhealthy and unavailable.To address this challenge, we can adopt the competing consumer pattern. Adopting this pattern means instead of one consumer listening on the topic, there will be a few. This means if one node is down, other nodes can listen to the incoming event. In this paradigm, message queue distributes the traffic to healthy consumers and avoid the unhealthy ones. This improves scalability, reliability, and resiliency. This pattern will help alleviate challenges in regards to Vel-2, Vel-3 and Vel-4. This pattern is portrated in Fig 2.

2. Circuit Breaker: On the other hand, given the large number of nodes one can assume for any BD system, one can employ the circuit breaker pattern to signal the service unavailability. Circuit breakers can protect the overall integrity of data and processes by tripping and closing the incoming request to the unhealthy service. This communicates effectively to the rest of the system that the node is unavailable, allowing engineers to handle such incidents gracefully. This pattern, mixed with competing consumers pattern can increase the overall availability and reliability of the system, and this is achieved by providing an even-driven asynchronous fault tolerance communication mechanisms among BD services. This allows system to be able to be resilient and responsive to bursty, high-throughput data as well as small, batch oriented data, addressing requirements Vel-1, Vel-3, and Vel-4.

3. Log Aggregator: Given that BD systems are comprising of many services, log aggregation can be implemented to shed lights on these services and their audit trail. Traditional single node logging does not work very well in distributed environments, as engineers are required to understand the whole flow of data from one end to another. To address this issue, log aggregation can be implemented, which usually comes with a unified interface that services communicate to and log their processes from. This interface then, does the necessary processes on the logs, and finally store the logs.
In addition, reliability engineers can configure alerts to be triggered underlying certain metrics. This increases teams agility to proactively resolve issues, which in turn increases reliability and availability which in turn addresses the velocity requirement of BD systems. While this design pattern does not directly affect any system requirements, it indirectly affects all of them. A simplistic presentation of this pattern is portrayed in Figure 2.

5.3 Variety

To address some of the challenges of this endeavour, we recommend the following patterns to address requirements Var-1, Var-3, Var-4; 1) API Gateway, 2) Gateway Offloading.

1. API Gateway and Gateway Offloading: We have previously discussed the benefits of API Gateway and Gateway Offloading, however in this section we aim to relate it more to BD system requirements Var-1, Var-3, and Var-4. Data engineers need to keep an open line of communication to data producers on changes that could break the data pipelines and analytics. Suppose that developer A changes a field in a schema of an object that may break a data pipeline or introduce a privacy threat. How can data engineers handle this scenario effectively?

To this end, API Gateway and Gateway Offloading can be used. These patterns can be used to offload some of the light-weight processes that may be associated to the data structure or the type of data. Moreover, as the data engineering loads increases, there will be more servers spawned for batch and stream processing. Gateways help with handling the traffic to these new servers and provide with zero down-time transitions. Additionally, as new data types are added, there will be specific nodes to process them. Gateways can help scaling for new data types. High-level overview of API Gateway pattern is portrayed in Figure 2.

5.4 Value

To address some of these challenges, we propose the following patterns to address the requirements Val-1, Val-3, and Val-4; 1) Command and Query Responsibility Segregation (CQRS), 2) Anti-Corruption Layer.

1. CQRS: Different consumers such as business analysts and machine learning engineers have very different demands, and would therefore create different workloads for the BD systems. As the consumers grow, the application has to handle more object mappings and mutations to meet the consumers demands. This may result in complex validation logics, transformations, and serialization that can be write-heavy on the data storage. As a result, the data serving layer can end up with an overly complex logic that does too much.
Read and write representation of the data are often different and miss-matching and require a specific approach and modeling. This is something a data engineer should consider from the initial data modeling, to data storage, retrieval and serialization. And while the system may be more tolerant on the write side, it may have a requirement to provide reads in a timely manner. For instance a snowflake schema maybe expensive for writes, but cheap for reads.
To address some of these challenges, we suggest CQRS pattern. CQRS separates the read from writes, using commands to update the data, and query to read data. This implies that the read and write databases can be physically segregated and consistency can be achieved through an event. To keep databases in sync, the write database can publish an event whenever an update occurs, and the read database can listen to it, retrieve the data, optimize for read and persist. This allows for elastic scaling of the read nodes, and increased query performance. This also allows for a read optimized data modeling tailored specifically for data consumers. Therefore, this pattern can potentially address the requirement Val-1, and Val-3. This pattern is portrated in Figure 2.

2. Anti-Corruption Layer: Given that the number of consumers and producers can grow and data can be created and requested in different formats with different characteristics, the ingestion and serving layer may be coupled to these foreign domains and try to account for an abstraction that aims to encapsulate all the logic in regards to all the external nodes. As the system grows, this abstraction layer becomes harder to maintain, and its maintainability becomes more difficult.
One approach to solve this issue is anti-corruption layer. Anti-corruption layer is a node that can be placed between the serving layer and data consumers or data producer, isolating different systems and translating between domains. This eliminates all the complexity and coupling that could have been otherwise introduced to the ingestion layer or the serving layer. This also allows for nodes to follow the 'single responsibility' pattern. This pattern can help with requirements Val-3 and Val-4. We have portrayed this pattern in Figure 2.

5.5 Security and Privacy

Security and privacy should be on top of mind for any BD system development, as these two aspects play an important role in the overall data strategy and architecture of the company. To this end, we propose the following pattern to address requirements SaP-1 and SaP-2; 1) Backend for Frontend (BFF)

1. Backend for Frontend: In terms of privacy, given the increasing load of data producers, and how they should be directed to the right processing node, how does one comply with regional policies such as

GDPR? how do we ensure, for example, that data is anonymized and identifiable properties are omitted? One approach is to do this right in the API gateway. However as data consumers grow and more data gets in, maintaining the privacy rules in the API gateway becomes more difficult. This can result in a bloated API gateway with many responsibilities, that can be a potential bottleneck to the system.
On approach to this problem can be the BFF pattern. By creating backends (services) for frontends (data producers), we can logically segregate API gateways for data that requires different level of privacy and security. Implementing this pattern means that instead of trying to account for all privacy related concerns in one node (API gateway), we separate the concerns to a number of services (backends) that are each responsible for a specific class of requirements. In addition, this pattern introduces a great opportunity for data mutation, schema validation, and potentially interface change (receive REST, and return GraphQL). On the other hand, from the security point of view, BFF can be implemented to achieve token isolation, cookie termination, and a security gate before requests can reach to upstream servers. Other security procedures such as sanitization, data masking, tokenization, and obfuscation can be done in this layer as well. This addresses the requirements SaP-1 and SaP-2. This pattern is portrayed in Figure 2.

5.6 Veracity

For veracity, we propose the following patterns for addressing requirements Ver-1, and Ver-4; 1) Pipes and Filters, 2) Circuit breaker

1. Pipes and Filters: Suppose that there is a data processing node that is responsible for performing variety of data transformation and other processes with different level of complexities. As requirements emerge, newer approaches of processing may be required, and soon this node will turn into a big monolithic unit that aims to achieve too much. Furthermore, this node is likely to reduce the opportunities of optimization, refactoring, testing and reusing. This is not elastic and can produce unwanted idle times.
One approach to this problem could be the pipes and filters pattern. By implementing pipes and filters, processing required for each stream can be separated into its own node (filter) that performs a single task. Following this approach allows for standardization of the format of the data and processing required for each step. This can help avoiding code duplication, and results in easier removal, replacement, augmentation and customization of data processing pipelines, addressing the requirements Ver-1 and Ver-4.

2.Circuit breaker: Circuit breakers can help with veracity by preventing data from getting into an unhealthy node and therefore getting corrupted. In addition, circuit breakers combined with 'pipes and filters' pattern can provide a great level of fault tolerance, by preventing data to be piped into the next filter, if the filter is not healthy. This indirectly increases the quality of data, and improves reliability. Having circuit breakers as proxies in place provides stability to the overall BD system, when the service of interest is recovering from an incident. This can indirectly help with Ver-4. This pattern is depicted in Figure 2.

6 Validation

After the generation of the design theories, we sought for a suitable model of validation. This involved a thorough research in some of the well-established methods for validation such as single-case mechanism experiment and technical action research (Wieringa, 2014). For the purposes of this study we chose semi-structured interviews (SSIs), following the guidelines of Kallio et al. (2016).

6.1 Interview design and Sampling strategy

Our SSI methodology is made up of four phases: 1) identifying the rationale for using semi-structured interviews, 2) formulating the preliminary semi-structured interview guide, 3) pilot testing the interview

guide, 4) presenting the results of the interview. SSI are suitable for our study, because our conceptual framework is made up of architectural constructs that can benefit from in-depth probing and analysis. Our questions are categorized into main themes and follow-up questions, with main themes being progressive and logical. We pilot tested our interview guide using internal testing, which involved an evaluation of the preliminary interview guide with the members of the research team. Our interview guide is available at *Interview guide for the paper: Application of microservices patterns to big data systems* 2022.
For sampling strategy, we used purposive sampling (Baltes and Ralph, 2022) to select experts. We chose purposive sampling because it allowed us to collect rich information by expert sampling. In addition, this approach enabled us to ensure representativeness and removed the need for a sampling frame. We reached out to experts in BD system development and architecture through different mediums such as Linkedin and ResearchGate. We interviewed 7 experts from various industries over a period of 3 months. An overview of these experts are as following: **i1**) Lead development architect with 18 years of experience working in health care industry, **i2**) Software architect with 8 years of experience working in human resource, **i3**) Associate professor with 15 years of of exprience working in consulting, **i4**) Senior Data Engineer with 20 years of experience working in software industry, **i5**) Big Data Architect with 8 years of experience working in insurance industry, **i6**) Technical Solution Manager with 40 years of experience working in consulting, **i7**) Solution Architect with 9 years of experience in BD systems.

6.2 Data Synthesis

All the interviews have been done through the software Zoom. We saved all of the recordings, and then downloaded the automatically generated transcripts. Transcripts for each interview have been added to Nvivo and then codes were created. We created a code for each BD characteristics discussed in Section 4.1.

6.2.1 Volume

For volume, we went through the theories elaborated in Section 5.1. All of the interviewees took the idea of API gateway and gateway offloading naturally, while we had to explore the 'external configuration store' a bit deeper. We used the idea of Kubernetes ingress to help with elaboration of API gateway. For the externalized configuration pattern, we discussed how environment variables may vary, how development and trial environments may not need as much resources as the production.
One interviewee mentioned that this can even be utilized for special privacy requirements. In addition, there's been discussion in regards to regional privacy and security requirements and how configurations can help derive them. One of the interviewees from insurance and finance sector mentioned that scaling the gateway and corresponding nodes may not be as easy as it seems. He mentioned that during normal days there are hardly any claims, and while there's a special event, the storm comes. Another interviewee from the financial sector has affirmed us that gateway offloading and API gateways are pretty common patterns, and he has witnessed it in several major banks

6.2.2 Velocity

For velocity, we first started by exploring an event-driven data engineering architecture, and then justified the idea of competing consumers. We then explored how competing consumers can fail, and how circuit breaker pattern can help. Finally we explored the idea of logging and how tail logging and distributed tracing can be achieved through it. An interviewee challenged the idea of competing consumer and stated that a leader election may be a better choice for a distributed setup as such. The interviewee also mentioned that circuit breakers could be implemented on the competing consumers themselves, but he could see the value of separating it to its own service.
In another interview, interviewee asked about the amalgamation technique for the logs, and discussed how dimensionality of the logs can be challenging. One candidate brough to light the challenges of time-sychroonization in log aggregator pattern. The candidate, who had a background in financial sector,

discussed how handling large amount of log data can introduce a challenge of its own. He continued to describe how critical these logs can be during sensitive stream processing tasks, and how data can easily get into petabytes in the banks. From his experience, handling logs alone would introduce a significant challenge as system grows.

6.2.3 Variety

For variety, we discussed common data types that need support, and how systems may use parquet, JSON, or how unstructured data can introduce challenges. An interviewee suggested the 'API composition' pattern and suggested that we may have various services that handle different data types, but the composition of these data may be necessary. The interviewee suggested that 'API composition' can occur at the egress level.

One interviewee provided details on how painful it has been for his team to onboard new data producers and how that dramatically slowed the project deadline. The interviewee added that data received from data producers hardly have the standards necessary, as these data are generated by third-party software that they have no control over. Another interviewee from insurance sector discussed how the rate of change is very low and most things are standard in insurance and finance sectors. For instance, he stated that if Avro is being used as the data format, the industry will be using the same format for the next 5 years.

6.2.4 Value

For value, we discussed CQRS and anti-corruption layer. We first began by exploring the challenges of having to optimize for read and write loads. We discussed how it could be essential for the business to provide read queries in a timely manner, and how trying to model for both read and write queries may not be efficient. An interviewee discussed how this pattern can be helpful in companies that have adopted domain-driven design, and how each bounded context can decide how the data should be modeled. Our interviewees shared the same idea that CQRS should only be applied when the needs arise and not proactively. This is due to the fact that implementing and getting CQRS right comes with complexity, and can dramatically increase cost.

Furthermore, we explained the anti-corruption layer. We discussed how the consumer needs can emerge, and how coupling it all to the read service can affect the system modifiability negatively. An interviewee raised the concern that defining the scope of anti-corruption layer may be a challenge. In his experience, data scientists need 'all the data' available in the company, and that's been a challenge for his team in the past. He continued discussing that this pattern can be useful not only from decoupling perspective but from a security and governance point of view.

6.2.5 Security and Privacy

For security and privacy, we started the discussion by exploring how different companies and regions may have different requirements, and how consuming data from data producers might be affected. We then discussed how having a single gateway to encompass all that logic can be daunting to scale. We then introduced the BFF pattern and elaborated that how each class of data consumers can be associated to a specific BFF. This pattern was well-received. An interviewee pointed out a potential of the access token pattern to be applied to the BFF. The interviewee elaborated that how having BFFs can help with cloud design and potential usage of private networks to increase security.

In another interview, the participant elaborated on how challenging it would be to have several ingresses into the system, and how BFF pattern may provide some stress on security and platform teams. While he admitted that performance and maintainability may increase, he found challenges of controlling what comes into the system significant. The interviewee added that going BFF requires substantial resources and may not be ideal for every company.

6.2.6 Veracity

For veracity, we discussed the transformation process required for any data engineering pipeline. We made an analogy to Linux philosophies and how different commands pipe into each other. This pattern was perceived to be one of the defacto patterns that many data engineering pipelines use. In addition, we discussed circuit breakers again, but this time for capturing transformation nodes that are unavailable.
An interviewee discussed how pipe and filters have been the key for them in production, and how it helped them scale and avoid data corruption. He added that without adopting such pattern, if something broke in a large transformation, you would never know what went wrong, and you might be forced to rerun a process that takes 5 hours to complete. In another interview, the participant discussed how circuit breaker should be tied to the end of the data processing and not only to the beginning of it. He elaborated that the server might be healthy when the transformation starts, but that might not be the case when it is about to end, therefore corrupting the data. He added that this can introduce unnecessary reprocessing.

7 Discussion

The result of our interviews provided us with more confident in our theories. In addition, this study shed lights on two major findings; 1) the progress in the data engineering space seems to be uneven in comparison to software engineering, 2) MS pattern provide with a great potential for resolving some of the BD system development challenges. While there has been adoption of a few practices from software engineering into data engineering like DataOps, we posit that data engineering space can benefit more from well-established practices of software engineering.
Majority of the studies that we have analyzed to understand BD systems, seems to revolve around crunching and transforming data without much attention to data lifecycle management. This is bold when it comes to addressing major cross-cutting concerns of successful data engineering practice such as security, data quality, DataOps, data architecture, data interoperability, and data versioning. Based on this, we think that data architecture remains a significant challenge and requires more attention from both academia and industry.

8 Conclusion

With all the undeniable benefits of BD, the success rate of BD projects is still rare. One of the core challenges of adopting BD lies in data architecture and data engineering. While software engineers have adopted many well-established methods and technologies, data engineering and BD architectures don't seem to benefit a lot from these advancements.
The aim of this study was to explore the relationship and application of MS architecture to BD systems through two distinct SLR. The results derived from these SLRs presented us with interesting data on the potential of MS patterns for BD systems. Given the distributed nature of BD systems, MS architectures seems to be a natural fit to solve myriad of problems that comes with decentralization. Therefore, further emperical research in this area is required.

References

29148, I. (2018). *ISO/IEC 29148:2018. Systems and software engineering — Life cycle processes — Requirements engineering*. Ed. by I. 29148. URL: https://www.iso.org/standard/72089.html.

Ataei, P. and A. Litchfield (2021). "NeoMycelia: A software reference architecturefor big data systems." In: *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, pp. 452–462.

— (2022). "The state of big data reference architectures: a systematic literature review." *IEEE Access* 10.

Baltes, S. and P. Ralph (2022). "Sampling in software engineering research: A critical review and guidelines." *Empirical Software Engineering* 27 (4), 1–31.

Chaabane, M., I. Bouassida, and M. Jmaiel (2017). "System of systems software architecture description using the ISO/IEC/IEEE 42010 standard." In: *Proceedings of the Symposium on Applied Computing*, pp. 1793–1798.

Chang, W. L. and N. Grady (2019). *NIST Big Data Interoperability Framework: Volume 1, Definitions*. Ed. by W. L. Chang and N. Grady. URL: https://doi.org/10.6028/NIST.SP.1500-1r2.

Cruzes, D. S. and T. Dyba (2011). "Recommended Steps for Thematic Synthesis in Software Engineering." In: *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, pp. 275–284. ISBN: 978-1-4577-2203-5. DOI: 10.1109/ESEM.2011.36.

Databricks, M. technology review insights in partnership with (2021). *Building a high-performance data organization*. Tech. rep. URL: https://databricks.com/p/whitepaper/mit-technology-review-insights-report.

Davenport, T. H. and R. Bean (2021). *Big Data and AI Executive Survey 2021*. Technical Report. NewVantage Partners. URL: https://www.newvantage.com/thoughtleadership.

Interview guide for the paper: Application of microservices patterns to big data systems (2022). URL: https://anonymous.4open.science/r/SSI-Repo-F90E/SSI.pdf.

Kallio, H., A.-M. Pietilä, M. Johnson, and M. Kangasniemi (2016). "Systematic methodological review: developing a framework for a qualitative semi-structured interview guide." *Journal of advanced nursing* 72 (12), 2954–2965.

Kassab, M., C. Neill, and P. Laplante (2014). "State of practice in requirements engineering: contemporary data." *Innovations in Systems and Software Engineering* 10 (4), 235–241.

Kitchenham, B. A., T. Dyba, and M. Jorgensen (2004). "Evidence-based software engineering." In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE Comput. Soc, pp. 273–281. ISBN: 0-7695-2163-0. DOI: 10.1109/ICSE.2004.1317449.

Laigner, R., M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda, S. Lifschitz, and Y. Zhou (2020). "From a monolithic big data system to a microservices event-driven architecture." In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, pp. 213–220.

Laplante, P. A. (2017). *Requirements engineering for software and systems*. Auerbach Publications.

Maamouri, A., L. Sfaxi, and R. Robbana (2021). "Phi: A Generic Microservices-Based Big Data Architecture." In: *European, Mediterranean, and Middle Eastern Conference on Information Systems*. Springer, pp. 3–16.

Microservices patterns classified for the Paper Titled: Application of Microservices Patterns to Big Data Systems (2022). URL: https://anonymous.4open.science/r/MS-Patterns-75D0/Ms-Patterns.pdf.

Nadal, S., V. Herrero, O. Romero, A. Abelló, X. Franch, S. Vansummeren, and D. Valerio (2017). "A software reference architecture for semantic-aware Big Data systems." *Information and software technology* 90, 75–92.

Page, M. J., J. E. McKenzie, P. M. Bossuyt, I. Boutron, T. C. Hoffmann, C. D. Mulrow, L. Shamseer, J. M. Tetzlaff, E. A. Akl, S. E. Brennan, et al. (2021). "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews." *Systematic reviews* 10 (1), 1–11.

Papers found for the paper: Application of microservices patterns to big data systems (2022). URL: https://anonymous.4open.science/r/found-papers-for-MS-and-BD-83E5/foundPapers.pdf.

Rad, B. B. and P. Ataei (2017). "The big data ecosystem and its environs." *International Journal of Computer Science and Network Security (IJCSNS)* 17 (3), 38.

Rada, B. B., P. Ataeib, Y. Khakbizc, and N. Akbarzadehd (2017). "The hype of emerging technologies: Big data as a service."

Richardson, C. (2018). *Microservices patterns: with examples in Java*. Simon and Schuster.

— (2022). *A pattern language for microservices*. URL: https://microservices.io/patterns/index.html.

Staegemann, D., M. Volk, A. Shakir, E. Lautenschläger, and K. Turowski (2021). "Examining the Interplay Between Big Data and Microservices–A Bibliometric Review." *Complex Systems Informatics and Modeling Quarterly* 27 (27), 87–118.

Systematic literature review search terms table for the Paper Titled: Application of Microservices Patterns to Big Data Systems (2022). URL: https://anonymous.4open.science/r/SLR-Search-Terms-3147/.

Tricco, A. C., E. Lillie, W. Zarin, K. K. O'Brien, H. Colquhoun, D. Levac, D. Moher, M. D. Peters, T. Horsley, L. Weeks, et al. (2018). "PRISMA extension for scoping reviews (PRISMA-ScR): checklist and explanation." *Annals of internal medicine* 169 (7), 467–473.

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer.

Zhelev, S. and A. Rozeva (2019). "Using microservices and event driven architecture for big data stream processing." In: *AIP Conference Proceedings*. Vol. 2172. 1. AIP Publishing LLC, p. 090010.