RESEARCH

Application of Microservices Patterns to Big Data Systems

Jane E. Doe[??*] and John R.S. Smith[??,??]

[*]Correspondence:
jane.e.doe@cambridge.co.uk
[??]Department of Science, University of Cambridge, London, UK
Full list of author information is available at the end of the article

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 various sources, 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 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. Along these lines, academics and practitioners have considered means through which they can incorporate data-driven functions and explore patterns that were otherwise unknown. While the opportunities exist with big data (BD), there are many failed attempts. According to New Vantage Partners report in 2022, only 26.5% of companies successfully become data-driven [**?**]. Another survey by Databricks highlighted that only 13% of organizations succeeded in delivering on their data strategy [**?**].

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 way to absorb the body of knowledge available on data architecture, can be reference architectures (RAs). By presenting proven ways to solve common implementation challenges on an architectural level, RAs support the development of new systems by offering guidance and orientation.

Another 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, increase maintainability and allows for a more flexible implementation [**?**]. 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. Comparable to RAs, they are proven artifacts on how certain problems could be solved. In the realm of MS, there are numerous patters that can be utilized, depending on the desired properties of the developed system. Despite the potential of RAs and MS architectures to solve some of complexities of BD development, to our knowledge, there is no study that properly bridge these two concepts.

To this end, this study aims to explore the application of MS patterns to BD system, in aspiration to solve some of the complexities of BD system development. For this purposes, two distinct systematic literature review (SLR) is conducted. The first SLR is an updated SLR on BD RAs [**?**], aiming to capture the years 2020-2022, and the second SLR is on MS patterns. The results of these SLRs are then captured through thematic synthesis, and design theories are generated. The theories are then validate through a semi-structured interview.

The contribution of the publication, is thereby threefold: 1) it provides an updated synopsis of the existing BD reference architectures, 2) it assembles an overview of relevant microservice patterns and, most importantly, 3) it creates a connection between the two 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. [**?**] applied an action research and reported on their experience on replacing a legacy BDS 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 et al. [**?**] 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 [**?**] 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 Phi [**?**] and Neomycelia [**?**], there is no study that aimed to apply MS patterns to BD systems through a systematic methodology.

3 Methodology

Since the goal of this study is to map BD architectures and microservice patterns, it is consequently mandatory to get a comprehensive overview over both domains. For this purpose, it was decided to conduct two systematic literature reviews (SLR), one for each domain. Both SLRs are conducted following the guidelines presented in

Kitchenham et al. [**?**] and Page et al. [**?**] 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's a comprehensive and well-established methodology for increasing systematicity, transparency, and prevention of bias. To synthesize our findings, thematic synthesis proposed by Cruzes and Dyba was applied [**?**].

3.1 First Review

The first SLR was the less extensive one, since it is just an update to an already existing study. For this purpose, the SLR conducted by Ataei et al. [**?**], was extended up unto the current date, providing us with the necessary overview. We followed the exact same methodology and covered the years 2020-2022. Our main objective for this SLR was to highlight the fundamental building blocks and requirements of BD systems.

While the common architectural constructs was discussed in [**?**], the study was not focused on requirements. We therefore extended the data synthesis by adding a new code: software and system requirements. This code had sub-codes each being named after one characteristics of BD. This was necessary as we needed to map patterns against requirements. This is further elaborated in the results section, subsection A.

3.2 Second Review

The second SLR, was a rigorous approach from scratch and was conducted in the following steps: 1) selecting data sources 2) developing a search strategy 3) developing inclusion and exclusion criteria, 4) developing the quality framework 5) pooling literature based on the search strategy, 6) removing duplicates, 7) scanning studies titles based on inclusion and exclusion criteria, 8) removing studies based on publication types, 9) scanning studies abstract and title based on inclusion and exclusion criteria, 10) assessing studies based on the quality framework (includes three phases), 11) extracting data from the remaining papers, 12) coding the extracted data, 13) creating themes out of codes, 14) presenting the results. These steps are not direct mappings to the following sub-sections. Some sub-sections include several of these steps.

3.2.1 Selecting data sources

To assure the comprehensiveness of the review, a broad set of scientific search engines and databases was queried. To increase the likelihood of finding all relevant contributions, it was decided to not discriminate between meta databases and publisher bound registers. Thus, both types were utilized. To achieve this, 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.

3.2.2 Developing a search strategy

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 [**?**]. These search terms are chosen because *patterns* are exactly what was sought for, *architectures* can contain such patterns, and *design* is often used as a synonym for architecture. Further, patterns can be seen as *building blocks*, therefore, the building blocks was also included.

3.2.3 Developing inclusion and exclusion criteria

Inspired by PRISMA checklist [**?**], 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).

3.2.4 Developing the 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 [**?**] on empirical research in software engineering. These 7 criteria are discussed in table 1.

3.2.5 Pooling literature based on the search strategy

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.

[width=8cm]../Media/PRISMA-Flowchart.JPG

Figure 1: Overview of the search process

3.2.6 Evaluating papers based on inclusion and exclusion criteria

The remaining 1868 papers were filtered by title to evaluate their relevance to the concepts of microservice patterns or architectural constructs related to MS. For this purpose, the first two authors separately evaluated each entry. If both agreed, this verdict was honored. In case of disagreement, they discussed the title to come to a conclusion. In this phase, the first author initially included 113 papers and the second author 146. Of those, 41 were present in both sets and 1650 were excluded

by both. This equates to an agreement rate of 90,5 percent (1691 of 1868 records) between the authors. After discussing the contributions with divergent evaluations, in total, 1699 of the 1868 papers were excluded, leaving 169 items for the next round.

The same approach was followed for abstracts. As a result, the first author evaluated 40 papers positively, and the second one 28. Both agreed on the inclusion of 22 papers and the exclusion of 123. This equates to an agreement rate of 85,8 percent (145 of 169 records) between the authors. In total of the 169 papers, 138 were removed and 31 were included in the next phase.

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.

3.2.7 Evaluating papers based on the quality framework

After having filtered out the pooled studies based on inclusion and exclusion criteria, we initiated a deeper probing, by running the remaining studies against the quality framework. The filtering based on the quality criteria was divided into three differently focused phases, with each of them requiring the passing of a quality gate as portrayed in Table 1. In the first phase, the aim was to ensure that reports fulfill at least a desired minimum level of comprehensiveness. For this purpose, studies were evaluated for their content to see if they are actual research or just a mere report on some lessons or expert opinions. In addition, we checked if objectives, justification, aim and context of the studies are clearly communicated.

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. Consequently, for each aspect, zero to two points were achievable and for all aspects, six points were available per paper. For inclusion into the second phase, at least five out of six points were demanded to assure a sufficient base quality. This corresponds to having at least 75 percent of the points.

In total, the authors agreed on 51 of 69 evaluations, resulting in an agreement rate of 73,9 percent. The second phase was focused on rigor. In this phase, studies were judged based on their research design and the data collection methods. The general procedure with the first two authors independently evaluating the reports remained the same. For inclusion in the next phase, again, 75 percent of the obtainable point were needed (this time three out of four). In total, the authors agreed on 23 of 36 evaluations, resulting in an agreement rate of 63,9 percent. While this value is rather low, this is likely caused by the narrow margins for some decisions.

Once more, the papers with the highest score (this time two) were discussed before inclusion, to further counteract possible fuzziness in the individual evaluations. The remaining 10 papers went through the third and final phase. Here, the credibility of the reporting and the relevance of the findings were evaluated. The procedure was the same as the previous phases. However, this time, all of the remaining papers passed. In this last phase, the authors agreed on 14 of 20 evaluations, resulting in an agreement rate of exactly 70 percent.

All ten publications 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

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	3.1 Credibility 3.2 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

were found via Scopus, whereas the remaining two have been identified through IEEE Xplore.

3.2.8 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. [**?**]. 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) quality attributes, and coded the findings based on it. By the end of this process, various themes emerged.

4 Results

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

4.1 Requirements specification

The results of our data synthesis emerged a few themes in regards to BD requirements. While we could find BD major building blocks and system requirements from the result of our SLR, our SLR did not include categorization and representation of these requirements. We also did not know what type of requirements is the most suitable to the goal of this study. To this end, we performed a lightweight

literature review in the body of knowledge to realize three major elements: 1) the type of requirements that we need, 2) an approach to categorizing the requirements, 3) presentation of these requirements.

4.1.1 Type of requirements

System and software requirements come in different flavours and can range from a formal (mathematical) specifications to sketch on a napkin. There's been various attempts to defining and classifying software and system requirements. For the purposes of this study, we opted for a well-received approach discussed by Laplante [**?**]. In this approach, requirements are classified into three major types of 1) functional requirements, 2) non-functional requirements, and 3) domain requirements.

Our objective is to define the high-level requirements of BD systems, thus we do not fully explore 'non-functional' requirements. Majority of non-functional requirements are emerged from the particularities of an environment, such as a banking sector or healthcare, and do not correlate to our study. Our primary focus is one functional and domain requirements.

4.1.2 Categorizing requirements

After having filtered out the right type of requirement, we then sought for a rigorous and relevant method to categorize the requirements. For this purpose, we followed the well-established categorization method based on BD characteristics, that is the 5Vs. These 5Vs are velocity, veracity, volume, Variety and Value [**?**], [**?**]. We took inspiration from various studies such as Nadal et al. [**?**], and the requirements categories presented in NIST BD Public Working Group [**?**].

We've also studied the reference architectures developed for BD systems to understand general requirements. In one study, Ataei et al. [**?**] assessed the body of evidence and presented with a comprehensive list of BD reference architectures. This study helped us realize the spectrum of BD reference architectures, how they are designed and the general set of requirements. By analyzing these studies and by evaluating the design and requirement engineering required for BD reference architectures, we adjusted our initial categories of requirements and added security and privacy to it.

4.1.3 Present requirements

After knowing the type and category of requirements, We looked for a rigorous approach to present these requirements. There are numerous approaches used for software and system requirement representation including informal, semiformal and formal methods. For the purposes of this study, we opted for an informal method because it is a well established method in the industry and academia [**?**]. Our approach follows the guidelines explained in ISO/IEC/IEEE standard 29148 [**?**] for representing functional requirements. We have also taken inspiration from Software Engineering Body of Knowledge [**?**]. However, our requirement representation is organized in term of BD characteristics. These requirements are described in Table 2.

Table 2: BD system requirements

Volume	**Vol-1)** System needs to support asynchronous, streaming, and batch processing to collect data from centralized, distributed, a 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 ne to stream data to data consumers in a timely manner, **Vel-3)** System needs to be able to ingest multiple, continuous, time vary data streams, **Vel-4)** System shall support fast search from streaming and processed data with high accuracy and relevancy, **Vel** 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** System needs to support aggregation, standardization, and normalization of data from disparate sources, **Var-3)** System s support adaptations mechanisms for schema evolution, **Var-4)** System can provide mechanisms to automatically include new d sources
Value	**Val-1)** System needs to able to handle compute-intensive analytical processing and machine learning techniques, **Val-2)** Syst needs to support two types of analytical processing: batch and streaming, **Val-3)** System needs to support different output 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 cont 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 transfor tion, **Ver-2)** System needs to support data provenance including data life cycle management and long-term preservation.

4.2 Microservice Patterns

As a result of this SLR, our data synthesis yielded 50 MS patterns. These patterns are classified based on their function and the problem they solve. Our categories are inspired by the works of Richardson [**?**]. While we elaborate the patterns adopted for BD requirements in detail, the aim of our study is not to explain each MS pattern. These patterns can be found in [**?**].

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 and try to apply some of the well-known software engineering patterns to the realm of data engineering and in specific, BD. Based on 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. We use Archimate [**?**] as recommend in ISO/IEC/IEEE 42010 [**?**].

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've portray patterns selected for each requirement in a reference architecture. We then justify the components and describe how patterns could address the requirement. These descriptions are presented as sub section, each describing one characteristic of BD systems.

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

5.1.1 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 achieve this process. One challenge is the realization of various data sources as described in Vol-1. Data comes in various formats from structured to semi-structured to unstructured, and system needs to handle different data through different interfaces. There is also streaming data that needs to be handled separately with different architectural constructs and data types. So some of the key engineering consideration for the ingestion process is that; 1) is the BD system ingesting data reliably? How frequently should data be ingested? In what volume the data typically arrives?

Given the challenges and particularities of data types, different nodes maybe spawned to handle the volume of data as witnessed in BD reference architectures studied by Ataei et al. [**?**]. Another popular approach is the segregation of concerns by separating batch and streaming processing nodes. Given the requirement of horizontal scaling for BD systems, it is safe to assume that there is usually more then one node associated to ingesting data. This can be problematic as different nodes will need to account for security, privacy and regional policies, in addition to the encapsulated data processing functionality.

This means that each node needs to reimplement the same interface for the aforementioned cross-cutting concerns, which makes scalability and maintainability a daunting task. This also introduces unnecessary repetition of codes and can result in non-conforming interfaces. To solve this problem, we explore the concept of gateway offloading and API gateway patterns. By offloading cross-cutting concerns that are shared across nodes to a single architectural construct, not only we will achieve a separation of concerns and a good level of usability, but we increase security and performance, by processing and filtering incoming data through a well specified ingress controller.

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.

Additionally, the gateway can increase the system reliability and availability by doing a constant health check on services, and distribute traffic based on liveliness probes. The is also an array of other benefits such as having a weighted distribution, and creating a special cache mechanism through specific HTTP headers. This also means that if the gateway is down, service nodes won't introduce bad state into the overall system. We have portrayed a very simplistic representation of this pattern in Fig 2.

5.1.2 External Configuration Store

As discussed earlier, BD systems are made up of various nodes in order to achieve horizontal scalability. While these systems are logically separated to their own service, they 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 configuration.

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 all nodes configuration to another service, each node can request its configuration from an external store on boot up. This can be achieved in Docker files through the CMD command, or could be written in Terraform codes for a Kubernetes pod. 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 Fig 2.

[width=17cm]../Media/All together.jpg

Figure 2: Big data reference architecture with microservices patterns

5.2 Velocity

Velocity is perhaps one of the most challenging aspects of the BD systems, which if not addressed well, can result in series of issues from system availability to massive loses and customer churn. 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-5; 1) Competing Consumers, 2) Circuit Breaker and 3) Log Aggregation.

5.2.1 Competing Consumers

BD doesn't imply only 'big' or a lot of data, it also implies the rate at which data can be ingested, stored and analyzed to produce insights. 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' [**?**]. If the business desires to go data driven, it should be able to have an acceptable time-to-insight, as critical business decisions cannot wait for data engineering pipelines.

Achieving this in such a distribute setup as BD systems with so many moving parts, is a challenging task, but there are MS pattern that can be tailored to help with some of these challenges. Given the very contrived scenario of a BD system described in the previous section, at the very core, data needs to be ingested quickly, stored in a timely manner, micro-batch, batch, or stream processed, and lately

served to the consumers. So what happens if one node goes down or becomes unavailable? in a traditional Hadoop setup, if Mesos is utilized as the scheduler, the node will be restarted and will go through a lifecycle again.

This means during this period of time, the node is unavailable, and any workload for stream processing has to wait, failing to achieve requirements Vel-2, Vel-3 and Vel-5. 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 an event driven communication as portrayed in the works of Ataei et al. [**?**], and try to increase fault tolerance and availability through competing consumers, circuit breaker, and log aggregation.

Underlying the event-driven approach, we can assume that nodes are sending each other events as a means of communication. This implies that node A can send an event to node B in a 'dispatch and forget' fashion on a certain topic. However this pattern introduces the same problem as the point-to-point REST communication style; if node B is down, then this will have a ripple effect on the whole system. To address this challenge, we can adopt the competing consumer pattern. Adopting this pattern means instead of one node listening on the topic, there will be a few nodes.

This can change the nature of the communication to asynchronous mode, and allow for a better fault tolerance, because if one node is down, the other nodes can listen to the event and handle it. In other terms, because now there are a few consumers listening on the events being dispatched on a certain topic, there's a competition of consumers, therefore the name 'competing consumers'. For instance, three stream processing consumer nodes can be spawned to listen on data streaming events being dispatched from the the up-stream. This pattern will help alleviate challenges in regards to Vel-2, Vel-3 and Vel-5.

5.2.2 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 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 con 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-4, and Vel-5.

5.2.3 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 communicates 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 Fig 2.

5.3 Variety

Variety, being another important aspect of BD, implies the range of different data types and the challenges of handling these data. As BD system grows, newer data structures emerge, and an effective BD system must be elastic enough to handle various data types. 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.

5.3.1 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 address this problem, API Gateway and Gateway Offloading can be used. API Gateway and Gateway Offloading could be good patterns to offload some of the light-weight processes that maybe associated to the data structure or the type of data. For instance, a light weight metadata check or data scrubbing can be achieved in the gateway. However, Gateways themselves should not be taking a lot of responsibility and become a bottleneck to the system. Therefore, as nodes increase and requirements emerge, one might chose to opt for 'Backend for Frontend' pattern. We do not do any modeling for this section, as the high-level overview of API Gateway pattern is portrayed in Fig 2.

5.4 Value

Value is the nucleus of any BD endeavour. In fact all components of the system pursue the goal of realizing a value, that is the insight derived from the data. Howbeit, realizing these insights require absorption of great deal of complexity. 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.

5.4.1 Command and Query Responsibility Segregation

Suppose that there are various application that would like to query data in different ways and with different frequencies (Val-3, Val-4). 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 serving layer can end up with an overly complex logic that does too much.

Read and write workloads are really different, and this is something a data engineer should consider from the initial data modeling, to data storage, retrieval and potential 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 (checking a fraudulent credit card). Read and write representation of the data are often different and miss-matching and require a specific approach and modeling. 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 and update its values. 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.

5.4.2 Anti-Corruption Layer

Another pattern that comes useful when handling large number of data consumers is the 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. Anti-corruption layer can define strong interfaces and quickly serve new demands without affecting much of the serving node's abstraction. In another terms, it avoids corruption that may happen among systems, by separating them. This pattern can help with requirements Val-3 and Val-4. We have portrayed this pattern and CQRS in Fig 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. At the intersection of data evolution, regional policies, and company policies, there's a great deal of complexity. To this end, we propose the following pattern to address requirements SaP-1 and SaP-2; 1) Backend for Frontend (BFF)

5.5.1 Backend for Frontend

API gateway has been discussed in several sections in this study, however, in this section we are interested to see how it can improve security and privacy of BD systems. 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 and applying them correctly to the dataset in the API gateway becomes more difficult. In addition, 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. This logical separation can include other factors such as QoS, key accounts, and even the nature of the API (GraphQL or RPC). 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 nodes that are each responsible for a specific requirement. This means, instead of creating a coupled, loosely abstracted implementation of privacy mechanisms, the system can benefit from hiding sensitive or unnecessary data in a logically separated node. This is also a great opportunity for data mutation, schema validation, and potentially protocol change (receive REST, and return GraphQL).

On the other hand, from the security point of view, and in specific in relation to authorization and authentication, this pattern provides with a considerable advantage. 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. As these BFF servers are logically isolated for specific requirements, maintainability and scalability is increased. This addresses the requirements SaP-1 and SaP-2.

5.6 Veracity

Next to value, veracity is an integral component of any effective BD system. Veracity in general is about how truthful and reliable data is, and how signals can be separated from the noises. Data should conform with the expectations from the business, thus data quality should be engineered across the data lifecycle. According to Eryurek et al. [**?**], data quality can be defined by three main characteristics 1) accuracy, 2) completeness, and 3) timeliness. Each of these characteristics posit a certain level of challenge to architecture and engineering of BD systems. To this, we propose the following patterns for addressing requirements Ver-1, and Ver-4; 1) Pipes and Filters, 2) Circuit breaker

5.6.1 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. In addition, as the business requirements emerge, the nature of some of these tasks may be different. Some processes may require a different metadata strategy that requires more computing resources, while others might not require such expensive resources. 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. This is a well-established approach in unix-like operating systems. 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.

5.6.2 Circuit breaker

In an inherently distributed environment like BD, calls to different services may fail due to various issues such as timeouts, transient faults or service being unavailable. While these faults may be transient, this can have a ripple effect on other services in the system, causing a cascading failure across several nodes. This affects system availability and reliability and can cause major losses to the business.

One solution to this problem can be the circuit breaker pattern. Circuit breaker is a pattern that prevents an application from from repeatedly trying to access a service that is not available. This improves the fault tolerance among services, and signals the service unavailability. The requesting application can decide accordingly on how to handle the situation. In other terms, circuit breakers are like proxies for operations that might fail. This proxy is usually implemented as a state machine having the states close, open, and half-open. Having this proxy 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.

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, technical action research and focus groups [**?**]. For the purposes of this study we chose semi-structured interviews (SSIs), following the guidelines of Kallio et al. [**?**].

6.1 Methodology

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. As we

examine an uncharted territory with a lot of potential, we posit that these interviews can post useful leads which we can pursue to further improve the theories of this study. We've formulated our SSI guide based on our research objective to achieve the richest possible data.

Our questions are categorized into main themes and follow-up questions, with main themes being progressing and logical, as recommended by Kallio et al. [**?**]. 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. We aimed to assume the role of the interviewee and gain insight into the limitations of our guide.

6.2 Results

From the results of these interviews, we gathered a lot of insights and probed deeper some of our architectural decisions. Every interview involved in deep analysis of the design patterns with one question from the interviewee trying to understand the problem space and solution better. Our interviewees had at least 8 years of experience and held titles such as 'lead development architect' and 'solution architect'. While some of our interviewees had more experience with BD and AI, some others were well-versed in MS architecture. We first asked interviewees about their depth of understanding with MS and BD, and then asked them if patterns discussed for each characteristic makes sense. We asked every interviewee if they can think of a pattern that we failed to consider. Our interview guide is available at [**?**].

We realized that API gateway and gateway offloading pattern is easily accepted as an effective pattern, while CQRS needed more reasoning and explanation. One interviewee had a concern about the write and read services being the single point of failure in the CQRS pattern and how those services can scale. Another interviewee wanted to know if we looked into event sourcing and if that could be applied. We've also received comments on the usage of priority queue for sensitive stream processing requirements. The most experienced interviewee (14 years) suggested us to further break down our processing requirements into domains and then utilize gateway aggregate patterns to do 'data as a service'. This idea was driven by data mesh and data fabrics. All of our interviewees found the study interesting, and were eager to know more after the interview.

Another feedback was the idea of having an egress that encapsulate the anti-corruption layer and adds some more into it as well. The pattern 'backend for frontend' was well received, and our event driven thinking seemed to be very well accepted by the interviewees. By the result of this interview we realized that we have missed an architectural construct while discussing velocity requirements, which was the message queue. These interviews increased our confidence in our results and reasoning and have shed some lights on possible new patterns that we could employ.

7 Discussion

The result of this study have provided us with 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 engineer like DataOps, we posit that data engineering space can benefit from some of the well-established practices of software engineering.

Majority of the studies that we've 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, data versioning and testing. In fact, while we found a lot of mature approaches in MS and event driven architectures, we could not find many well-established patterns in the data engineering space. 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. Even though we created many design theories, modeled patterns against systems, and validated our theories, we believe that our results could be further validated by an empirical study. We therefore posit that there is a need for more attention in the area of MS and event-driven architectures in relation to BD systems from both academia and industry.