

# Conference Paper Title\*

\*Note: Sub-titles are not captured in Xplore and should not be used

1<sup>st</sup> Given Name Surname  
*dept. name of organization (of Aff.)*  
*name of organization (of Aff.)*  
City, Country  
email address or ORCID

2<sup>nd</sup> Given Name Surname  
*dept. name of organization (of Aff.)*  
*name of organization (of Aff.)*  
City, Country  
email address or ORCID

3<sup>rd</sup> Given Name Surname  
*dept. name of organization (of Aff.)*  
*name of organization (of Aff.)*  
City, Country  
email address or ORCID

4<sup>th</sup> Given Name Surname  
*dept. name of organization (of Aff.)*  
*name of organization (of Aff.)*  
City, Country  
email address or ORCID

5<sup>th</sup> Given Name Surname  
*dept. name of organization (of Aff.)*  
*name of organization (of Aff.)*  
City, Country  
email address or ORCID

6<sup>th</sup> Given Name Surname  
*dept. name of organization (of Aff.)*  
*name of organization (of Aff.)*  
City, Country  
email address or ORCID

**Abstract**—This document is a model and instructions for  $\text{\LaTeX}$ . This and the `IEEEtran.cls` file define the components of your paper [title, text, heads, etc.]. **\*CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

**Index Terms**—component, formatting, style, styling, insert

## I. REQUIREMENTS SPECIFICATION

Precursor to theorizing about the potential of microservices patterns for big data systems, we need to define what are the requirements of these systems. System and software requirements come in different flavours and can range from a sketch on a napkin to formal (mathematical) specifications. Therefore, we first need to identify what kind of requirements is the most suitable for the purposes of this study. To answer this question, we first explored the body of evidence to understand the current classification of software requirements.

There's been various attempts to defining and classifying software and system requirements. For instance, Sommerville ([1]) classified requirements into three levels of abstraction that are namely 1) user requirements, 2) system requirements and 3) design specifications. The author then mapped these requirements against user acceptance testing, integration testing and unit testing. While this could satisfy the requirements of this study, we opted for a more general framework provided by Laplante ([2]). In Laplante's approach, requirements are categorized into three categories of 1) functional requirements, 2) non-functional requirements, and 3) domain requirements.

Our objective is to define the high-level requirements of big data 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. Therefore, our primary focus is one functional and domain requirements and secondly on non-functional requirements.

Identify applicable funding agency here. If none, delete this.

After clarifying the type of requirements, we then explored the body of evidence to realize the general requirements of big data systems. By the result of this, we realized that the most discussed characteristics of big data systems are the popular 5Vs which are velocity, veracity, volume, Variety and Value ([3], [4], [5], [6], [7], [8] ). Many researchers such as Nadal et al. ([9]) and Klein et al. (klein2016reference) have underpinned their reference architecture on these characteristics and requirements that goes with them.

On the other hand, in an extensive effort, NIST Big Data Public Working Group embarked on a large scale study to extract requirements from variety of application domains such as Healthcare, Life Sciences, Commercial, Energy, Government, and Defense. The result of this study was the formation of general requirements under seven categories. In another effort by Volk et al. ([10]), nine use cases for big data projects are identified by collecting theories and use cases from the literature and categorizing them using a hierarchical clustering algorithm. Bashari et al. ([11]) focused on the security and privacy requirements of big data systems, Yu et al. presented the modern components of big data systems ([12]), Eridaputra et al. ([13]) created a generic model for big data requirements using goal oriented approaches, and Al-jaroodi et al. ([14]) investigated general requirements to support big data software development.

We've also studied the reference architectures developed for big data systems to understand general requirements. In one study, Ataei et al. ([15]) assessed the body of evidence and presented with a comprehensive list of big data reference architectures. This study helped us realize the spectrum of big data 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 big data reference architectures, we created a set of high-level requirements based on big data characteristics. We have then 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's a well established method in the industry and academia ([16]).

Our approach follows the guidelines explained in ISO/IEC/IEEE standard 29148 ([17]) for representing functional requirements. Our requirement representation is organized in system modes, that is we explain the major components of the system and then describe the requirements. This approach is inspired by the requirement specification expressed for NASA WIRE (wide-field infrared explorer) system explained in [2]. We also taken inspiration from Software Engineering Body of Knowledge ([18]).

Taking all into consideration, we categorized our requirements based on the major characteristics of big data, that is value, variety, velocity, veracity, and volume ([19]), plus security and privacy ([11]). These requirements are described in table I.

## II. MICROSERVICE PATTERNS

As a result of this SLR, 50 microservice patterns have been found. These patterns are then classified based on their function and the problem they solve. Each classification and its reasoning is depicted in table II.

## III. 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 microservices design patterns for big data systems. The patterns gleaned, are established theories that are derived from actual problems in microservices systems in practice, thus we do not aim to re-validate them in this study. Moreover, we do not aim to validate the theories proposed in this study through an empirical 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, big data. Based on this, we map big data system requirements against a pattern and provide with reasoning on why such pattern might work for big data systems.

Big data systems and microservices architecture are both inherently distributed. While majority of current big data applications are designed underlying a monolithic data pipeline architecture, here, we propose microservices architecture for a domain-driven and decentralized big data architecture. We support our arguments by the means of modeling. We use Archimate ([20]) as recommended in ISO/IEC/IEEE 42010 ([21]).

We posit that a pattern alone would not be significantly useful to a data engineering or a data architect, and propose that

collection of a pattern in relation to current de facto standard of BD architectures is a better means of communication.

To achieve this, we've portrayed 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 big data systems.

### A. Volume

There has been two requirements associated to the Volume aspect of big data systems which are about handling various data types (Vol-1) and providing with a scalable storage (Vol-2).

For Vol-1 and Vol-2 we suggest the following patterns to be effective; 1) External Configuration Store, 2) API gateway, 3) Gateway offloading

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 in this process 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) what are the typical use cases for the data being ingested? is the big data system ingesting data reliably? what is the next data destination? How frequently should data be ingested? In what volume the data typically arrives? Does streaming data need to be transformed before reaching the destination?

Given the challenges and particularities of data types, different nodes maybe spawned to handle the volume of data as witnessed in big data reference architectures studied by Ataei et al ([15]). Another popular approach is the segregation of concerns by separating batch and streaming processing nodes. Given the requirement of horizontal scaling for big data systems, it is safe to assume that there is usually more than one node associated to the data being ingested. This can be problematic as different nodes will need to account for security, privacy and overall regulations of the context, alongside, the software engineering demand that each may have.

This means that each node needs to reimplement the same interface for the aforementioned cross-cutting concerns, which makes scalability and maintainability of the big data system a daunting task. This also introduces unnecessary repetition of codes. 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, the API gateway in this case, not only will we achieve a separation of concerns and a good level of usability, but we increase security and performance, by

Volume	Vol-1) System needs to support asynchronous, streaming, and batch processing to collect data from centralized, distributed, and cloud data sources, and sensors, instrument and other IOT devices 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 and computing clusters 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 shall support fast search from streaming and processed data with high accuracy and relevancy Vel-5) 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 graph, web, text, document, timed, spatial, multimedia, simulation, instrumental, and geo-spatial 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 be 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 such as descriptive analytics, predictive analytics, reporting and visualizations. 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. Ver-3) System needs to support data validation in two ways: automatic and human annotated. Ver-4) System should be able to handle data loss or corruption.

TABLE I: Big data system requirements

processing and filtering incoming data through a well specified ingress.

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 service tries to communicate with a service that is down. Given that, the 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 big data system can be compromised. This scenario remains the same, and can be even worst if the company decides to have an on-premise data center.

Additionally, the gateway can increase the system reliability and availability by doing a constant health check on services, and distribute traffic based on healthy nodes. There 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 data or state into the overall system. We have portrayed a very simplistic representation of this pattern in fig 1.

2) *External Configuration Store*: As discussed earlier, big data 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 to achieve their functionality.

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, also known as the 'externalized configuration 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 big data systems and provides with a scalable solution for handling configurations. This pattern is portrayed in fig 1.

Category	Pattern
Data Management	Database per Service, Shared Database, Event Sourcing, Command and Query Responsibility Segregation
Platform and Infrastructure	Multiple service instances per host, External configuration store, Sidecar, Static content hosting, Computer resource consolidation
Communicational	API gateway, Anti-corruption layer, Self Registration, Service Discovery, Competing consumers, Pipes and filters, Priority queue, Ambassador, Gateway aggregate, Gateway offloading, Aggregator, Backend for Frontend, API Composition, Saga transaction management, Gateway routing, Leader election
Fault Tolerance	Circuit breaker, Bulkhead pattern
Observability	Log Aggregation Pattern

TABLE II: Microservices categorization

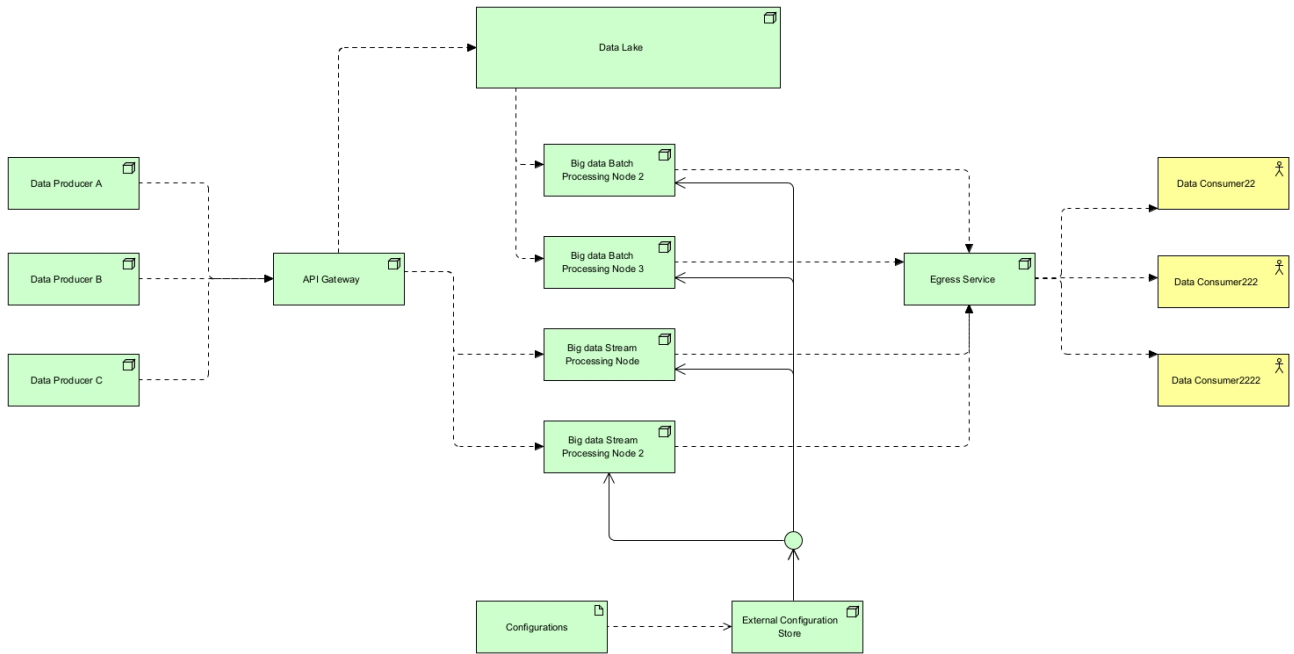


Fig. 1: Design patterns for volume requirement

### B. Velocity

Velocity is perhaps one of the most challenging aspects of the big data systems, which if not addressed well, can result in series of issues from system availability to massive losses and customer churn.

To address some of the challenges associated with the velocity aspect of big data 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.

1) *Competing Consumers*: Big data 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 big data 'low-achievers' is the 'slow processing of large amounts of data' ([22]). 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 big data systems with so many moving parts, is a challenging task, but there are microservices pattern that can be tailored to help with some of these challenges. Given the very contrived scenario of a big data 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 ([23]), 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 (could be ingress or data producers). This pattern will help alleviate challenges in regards to Vel-2, Vel-3 and Vel-5.

2) *Circuit Breaker*: On the other hand, given the large number of nodes one can assume for any big data 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 can increase the overall availability and reliability of the system, and this achieved by providing an even-driven asynchronous fault tolerance communication mechanisms among big data 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.

### C. Log Aggregator

Given that big data 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. 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 issue, which in turn increases reliability and availability which in turn addresses the velocity requirement of big data systems. While this design pattern does not directly affect any system requirements, it indirectly affects all of them. A simplistic reference architecture of these patterns have been portrayed in fig 2

### D. Variety

Variety, being another important aspect of big data, implies the range of different data types and th challenges of handling these data. As big data system grows, newer data structures emerge, and an effective big data 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.

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 big data 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 pipeline or introduce a privacy threat. How can data engineers handle this scenario effectively?

To address this problem, and to address big data requirements Var-1, Var-3, and Var-4, 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 1.

### E. Value

Value is the nucleus of any big data 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, 3) Gateway Offloading

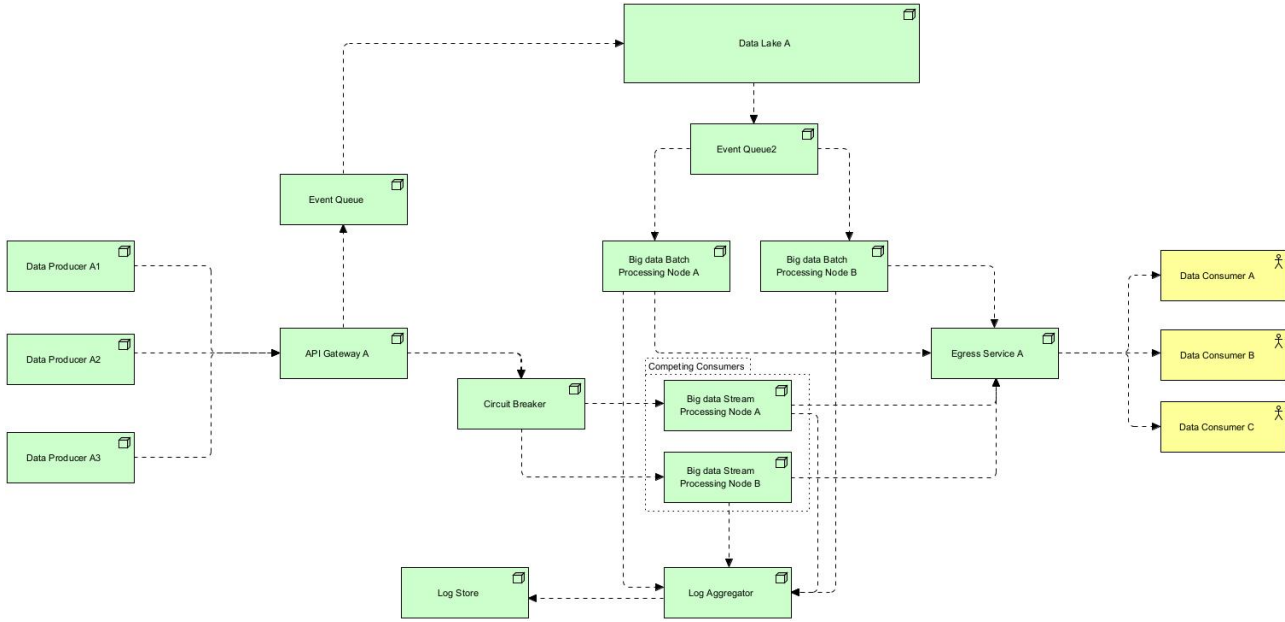


Fig. 2: Design patterns for velocity requirements

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 big data 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 (JSON to Parquet). 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 this 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, especially in big data systems that have got egress services sitting on an edge. Therefore, this pattern can potentially address the

requirement Val-1, and Val-3.

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 is placed between the serving layer and data consumers, isolating different systems and translating communication. This eliminates all the complexity and coupling that could have been otherwise introduced to the serving layer. This also allows for nodes to follow the 'single responsibility' pattern ([24]). 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 3.

3) *Gateway Offloading*: We have previously discussed this pattern, but in this section we aim to relate more to the value requirements of BD. Given that the various nodes in the system may require services such as authentication, authorization, monitoring, logging, and throttling, it would become really difficult to address the value aspect of big data in a timely manner. To address these issues and to achieve Val-3 and Val-4, we recommend the gateway offloading pattern. Employing this pattern abstracts out cross-cutting services from each node

and creates a unified interface that each node can utilize. This simplifies the development of new services for handling new data formats, and free data engineers from implementing features that requires special knowledge such as security and privacy.

#### *F. Security and Privacy*

Security and privacy should be on top of mind for any big data 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)

1) *BFF*: 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 big data 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 batch or stream processing nodes. However as data consumers grow and more data gets in, maintaining the privacy rules and applying them correctly to the dataset becomes more difficult. This becomes a perfect ground for mistake, and can potentially introduce legal issues to the company.

One approach to this problem can be the BFF pattern. By creating backends (services) for frontends (data producers), we can logically segregate system's ingress 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, 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.

#### *G. Veracity*

Next to value, veracity is an integral component of any effective big data 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 ([25]), 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 big data systems.

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

#### *H. 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 resembles to the well-known pattern of 'single responsibility' ([24]). 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.

#### *I. Circuit breaker*

In an inherently distributed environment like big data, 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 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

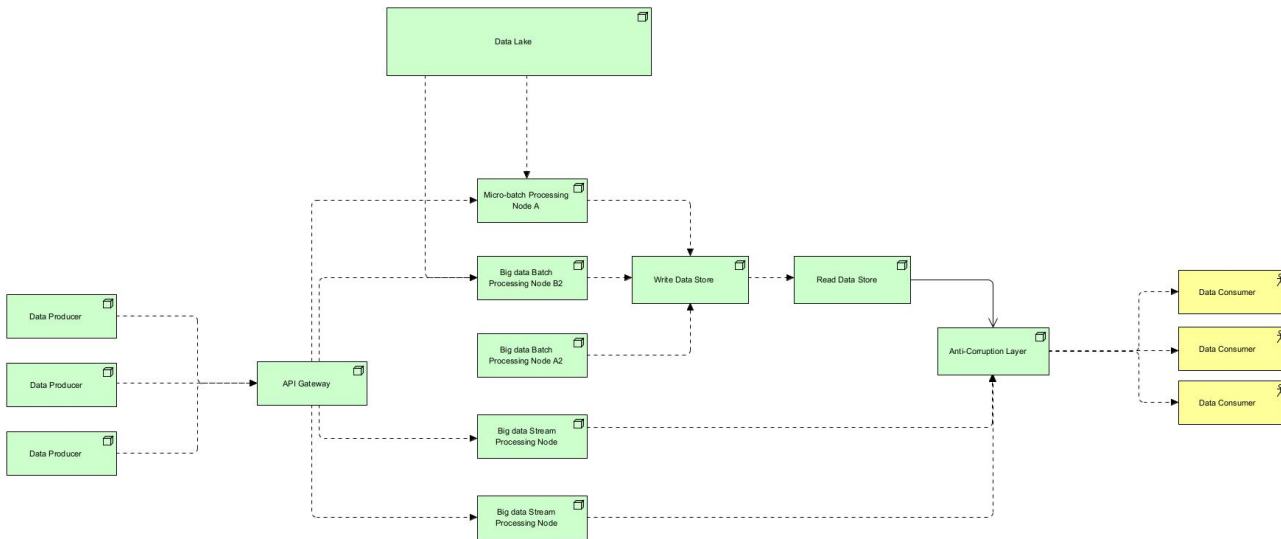


Fig. 3: Design patterns for value requirement

machine having the states close, open, and half-open. Having this proxy in place provides stability to the overall big data system, when the service of interest is recovering from an incident. This can indirectly help with Ver-4.

## REFERENCES

- [1] I. Sommerville, *Software Engineering*, 9/E. Pearson Education India, 2011.
- [2] P. A. Laplante, *Requirements engineering for software and systems*. Auerbach Publications, 2017.
- [3] Y. Demchenko, C. De Laat, and P. Membrey, "Defining architecture components of the big data ecosystem," in *2014 International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2014, Conference Proceedings, pp. 104–112.
- [4] J. Bughin, "Big data, big bang?" *Journal of Big Data*, vol. 3, no. 1, p. 2, 2016.
- [5] M. Bahrami and M. Singhal, *The role of cloud computing architecture in big data*. Springer, 2015, pp. 275–295.
- [6] B. B. Rad and P. Ataei, "The big data ecosystem and its environs," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 38, 2017.
- [7] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.
- [8] H.-M. Chen, R. Kazman, and S. Haziye, "Agile big data analytics development: An architecture-centric approach," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2016, Conference Proceedings, pp. 5378–5387.
- [9] S. Nadal, V. Herrero, O. Romero, A. Abelló, X. Franch, S. Vansummen, and D. Valerio, "A software reference architecture for semantic-aware big data systems," *Information and software technology*, vol. 90, pp. 75–92, 2017.
- [10] M. Volk, D. Staegemann, I. Trifonova, S. Bosse, and K. Turowski, "Identifying similarities of big data projects—a use case driven approach," *IEEE Access*, vol. 8, pp. 186 599–186 619, 2020.
- [11] B. Bashari Rad, N. Akbarzadeh, P. Ataei, and Y. Khakbiz, "Security and privacy challenges in big data era," *International Journal of Control Theory and Applications*, vol. 9, no. 43, pp. 437–448, 2016.
- [12] J.-H. Yu and Z.-M. Zhou, "Components and development in big data system: A survey," *Journal of Electronic Science and Technology*, vol. 17, no. 1, pp. 51–72, 2019.
- [13] H. Eridaputra, B. Hendradjaya, and W. D. Sunindyo, "Modeling the requirements for big data application using goal oriented approach," in *2014 international conference on data and software engineering (ICODSE)*. IEEE, 2014, pp. 1–6.
- [14] J. Al-Jaroodi and N. Mohamed, "Characteristics and requirements of big data analytics applications," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 2016, pp. 426–432.
- [15] P. Ataei and A. T. Litchfield, "Big data reference architectures, a systematic literature review," 2020.
- [16] M. Kassab, C. Neill, and P. Laplante, "State of practice in requirements engineering: contemporary data," *Innovations in Systems and Software Engineering*, vol. 10, no. 4, pp. 235–241, 2014.
- [17] I. 29148:2018, "Iso/iec 29148:2018," 2018. [Online]. Available: <https://www.iso.org/standard/72089.html>
- [18] A. Abran, J. W. Moore, P. Bourque, R. Dupuis, and L. Tripp, "Software engineering body of knowledge," *IEEE Computer Society, Angela Burgess*, p. 25, 2004.
- [19] B. B. Rada, P. Ataei, Y. Khakbiz, and N. Akbarzadehd, "The hype of emerging technologies: Big data as a service," 2017.
- [20] M. Lankhorst, "A language for enterprise modelling," in *Enterprise Architecture at Work*. Springer, 2013, pp. 75–114.
- [21] M. Chaabane, I. Bouassida, and M. Jmaiel, "System of systems software architecture description using the iso/iec/ieee 42010 standard," in *Proceedings of the Symposium on Applied Computing*, Conference Proceedings, pp. 1793–1798.
- [22] M. technology review insights in partnership with Databricks, "Building a high-performance data organization," 2021. [Online]. Available: <https://databricks.com/p/whitepaper/mit-technology-review-insights-report>
- [23] P. Ataei and A. Litchfield, "Neomycelia: A software reference architecture for big data systems," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 452–462.
- [24] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides *et al.*, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.
- [25] E. Eryurek, U. Gilad, V. Lakshmanan, A. Kibunguchy-Grant, and J. Ashdown, *Data Governance: The Definitive Guide*. O'Reilly Media, Inc., 2021.