Conference Paper Title*

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

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

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

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

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

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

6th 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 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 we mean by big data systems and what are the requirements of these systems. System and software requirements come in different flavour 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 systems 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 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 seek to explore 'non-functional' requirements. Non-functional requirements are emerged from the particularities of an environment, such as a banking sector and do not correlate to our study. Therefore, the type of requirements we are looking for is functional and domain requirements.

After clarifying the type of requirements, we then explored the body of evidence to realize the general requirements of big data systems. Indeed, 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]) have underpinned their artifact development on these characteristics and requirements that emerge from them.

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 and 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]),9 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 realized 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

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

a rigorous approach to present these requirements. There are numerous approaches used for 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 for representing functional requirements. Our requirement repesentation 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 Version ([17]).

Taking all into consideration, we categorized our requirements based on the major characteristics of big data, that is value, variety, velocity, veracity, and volume ([**?**]), plus . These requirements are as followings:

II. Microservice Patterns

As a result of this SLR, 50 microservice patterns have bene found. These patterns are then classified based on their function and the problem they solve. Each classifications and it's reasoning is depicted in table **??**.

1) Database per service ✓
2) Shared database ✓
3) Event sourcing ✓
4) Multiple service instances per host ✓
5) API gateway ✓
6) Self registration ✓
7) Service discovery ✓
8) Circuit breaker ✓
9) Bulkhead pattern ✓
10) Command and query responsibility segregation ✓
11) Competing consumers ✓
12) Pipes and filters ✓
13) Strangler ✓
14) Anti-corruption layer ✓
15) External configuration store ✓
16) Priority queue ✓
17) Log Aggregation ✓
18) Ambassador ✓
19) Sidecar ✓
20) Gateway aggregate ✓
21) Gateway offloading ✓
22) Aggregator ✓
23) Backend for Frontend ✓
24) API Composition ✓
25) Saga transaction management ✓
26) Static content hosting ✓
27) Computer resource consolidation ✓
28) Leader election ✓

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.

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 the application handling various data types (Vol-1) and the application providing with a scalable storage (Vol-2).

For Vol-1 we suggest the following patterns to be effective:

1) External Configuration Store
2) API gateway
3) Gateway offloading

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 ([**?**]) 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 collection of a pattern 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. For this purpose we portray the patterns for Vol-1 in figure III-A

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. The problem is that data comes in various formats from structured to semi-structured to unstructured, and the systems needs to handle different data through different

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 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 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.

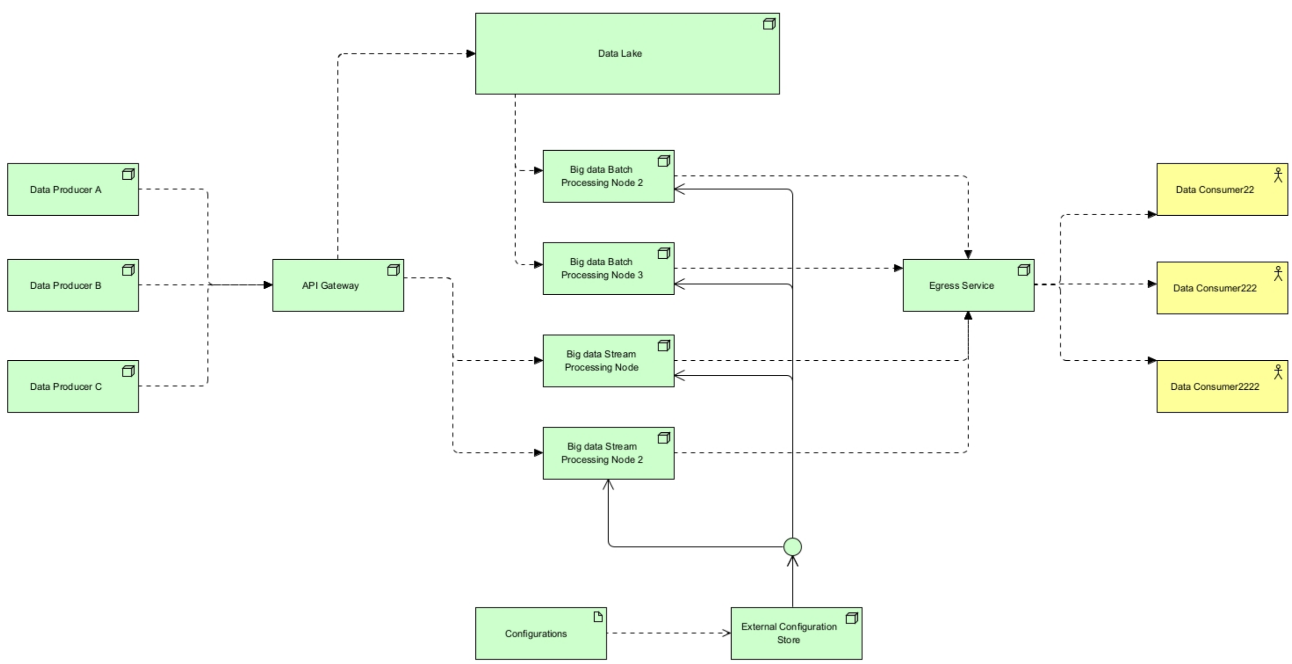

Fig. 1. Design patterns for volume requirement

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

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

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 then 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 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.

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. 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 data or state into the overall system. We have portrayed a very simplistic representation of this pattern in fig III-A.

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.

Thus, the challenge is the management of these configurations as the system scale, 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 is portrayed in fig III-A.

B. Velocity

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

Requirement	Patterns	Reasoning
Vol-1	1) Database per Service 2) Event Sourcing 3) Command and Query Responsibility Segregation 4) External Configuration Store 5) API gateway 6) Anti-Corruption Layer 7) Service Discovery 8) Self Registration 9) Priority Queue 10) Gateway Offloading 11) Gateway Aggregate 12) Leader Election 13) Log Aggregation Pattern	Reasoning
Vol-2	1) Database per Service 2) Command and Query Responsibility Segregation	Reasoning
Vel-1	1) API Gateway 2) Service Discovery 3) Pipes and Filters 4) Leader Election 5) Circuit Breaker 6) Log Aggregation	Reasoning
Vel-2	1) Command and Query Responsibility Segregation 2) API gateway 3) Competing consumers 4) Gateway aggregate 5) Gateway Offloading 6) Leader Election	Reasoning
Vel-3	1) Command and Query Responsibility Segregation 2) API gateway 3) Competing consumers 4) Gateway aggregate 5) Gateway Offloading 6) Leader Election	Reasoning
Vel-3	1) API composition 2) API gateway 4) Gateway aggregate 5) Gateway Offloading	Reasoning
Vel-4	1) API composition 2) API gateway 4) Gateway aggregate 5) Gateway Offloading 6) Event Sourcing 7) Command and Query Responsibility Segregation	Reasoning
Vel-5	1) Leader Election 2) Log Aggregation Pattern	Reasoning
Var-1	None	Reasoning
Var-2	1) Database per Service 2) API Gateway	Reasoning
Var-3	None	Reasoning
Var-4	1) API Gateway 2) Gateway Offloading 3) Gateway Aggregate	Reasoning

Val-1	1) Event Sourcing 2) Command and Query Responsibility Segregation 3) Priority Queue 4) Leader Election 5) Bulkhead Pattern	Reasoning
Val-2	1) Event Sourcing 2) Command and Query Responsibility Segregation 3) API gateway, Gateway aggregate 4) Gateway offloading 5) Priority queue	Reasoning
Val-3	1) API gateway 2) Anti-corruption layer 3) Service Discovery 4) Gateway aggregate 5) Backend for Frontend	Reasoning
Val-4	1) Event Sourcing 2) Command and Query Responsibility Segregation 3) Backend for Frontend	Reasoning
SaP-1	1) External Configuration Store 2) API Gateway 3) Gateway Aggregate 4) Backend for Frontend	Reasoning
SaP-2	1) External Configuration Store 2) API Gateway 3) Gateway Aggregate 4) Backend for Frontend	Reasoning
Ver-1	1) Pipes and filters	Reasoning
Ver-2	None	Reasoning
Ver-3	None	Reasoning
Ver-4	1) Circuit Breaker	Reasoning

To address some of the challenges associated with the velocity aspect of big data systems, we recommend the following patterns:

1) API Gateway
2) Service Discovery
3) Circuit Breaker
4) Log Aggregation
5) Command and Query Responsibility Segregation
6) Competing consumers
7) Gateway Offloading

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'. If the business desires to go data driven, it should be able to have time-to-insight within an acceptable range, as the decisions have to be made at the end of the day.

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. This issue is exacerbated if the system is designed and architected underlying monolithic pipeline architecture with point-to-point communication. One way to solve some of these issues is to introduce an event driven communication as portrayed in the works of Ataei et al ([18]), and try to increase fault tolerance and availability through competing consumers, circuit breaker, and log aggregation.

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. Haziyev, "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. Vansummeren, 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] 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.

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