Towards a domain-driven distributed reference architecture for big data systems

Completed Research Full Paper

Introduction

Almost all aspects of human life are integrated with some sort of software system that is performing computational processes on data. The rapid expansion and evolution of data from a structured element that is passively stored in the database to something that is used to support proactive decision making for business competitive advantage, have dawned a new era, the era of Big Data (BD). The BD era emerged when the velocity, variety, and volume of data overwhelmed existing system capability and capacity to effectively and efficiently process and store data [*citation hidden*].

BD is the practice of crunching large sets of heterogenous data to discover patterns and insights for business competitive advantage [*citation hidden*]. Since the inception of the term, ideas have ebbed and flowed along with the rapid advancements of technology, and many strived to harness the power of BD. Nevertheless, there are many failed attempts, for example, as of 2021 only 13% of organizations succeeded in delivering on their data strategy (technology review insights in partnership with Databricks, 2021) and 20–24% successfully adopted BD (Partners 2021; White 2019). Among the challenges of adopting BD, latent complexity in data engineering, rapid technology changes, data architecture and a poor supply of skilled data engineers are highlighted. [*citation hidden*].

A BD system is motivated by an array of functional requirements and quality goals. But if this system is to be successful, it must achieve these functional requirements within acceptable performance, availability, modifiability, and cost parameters. The software architecture is the key ingredient in determining whether these goals are attainable, before colossal number of resources are committed to it. The initial design, development and deployment of a BD system does not mean success. As systems grow larger, data providers and data consumers increase, data variety expands, data velocity extends, and metadata becomes increasingly more challenging to handle. This means, only a handful of highly specialized data engineers would be able to understand the system internals, resulting in silos, burnt out and potential friction. This creates a perfect ground for immature architectural decisions that result in fragile systems that are difficult to maintain and scale. To this end, we explore the concept of generally applicable domain driven distributed software reference architecture (RA) that is technology independent. The contribution of our work is threefold: 1) design theories that critique current BD RAs, 2) design theories that creates the foundation of our RA, 3) the artifact.

Background

In this section, the current state of BD architectures, identification of the study focus, and study objectives are provided. Two important elements are discussed, the current state of BD architectures and why RA is needed.

Overview of BD architectures

In this overview, three generations of BD architecture are presented. A key issue observed in all the approaches analyzed was the existence of an architecture featuring a federated or contiguous data pipeline to channel all data without any consideration for data quality and ownership. Thus, data that logically belongs to different domains are aggregated and processed together, creating impediments to scalability and maintainability. In addition, cross-cutting concerns such as metadata management, security, privacy, data ownership and interoperability are hardly addressed.

Enterprise Data Warehouse: Enterprise Data Warehouses are typically designed based on specific requirements, which can limit the effective exploitation of BD characteristics. These architectures rely on

monolithic data warehouses, ETL processes, and data analysis and visualization software. However, as the Enterprise Data Warehouse grows and more users and providers join, maintaining and managing ETL processes becomes increasingly difficult and time-consuming. This leads to slower transaction processing and increased dependency on specialized staff, which creates silos and friction within and between teams and organizational boundaries. Moreover, scaling, maintenance, and the extension of data engineering pipelines become challenging and expensive, making it difficult to effectively handle modern BD analytics requirements.

Data Lake: The Data Lake emerged as a new BD ecosystem to address issues in data warehousing, where data is written into the lake without much transformation and retrieved by data scientists and machine learning engineers in raw format. While this approach handles data variety well, it may lack optimization features, leading to the risk of turning into a data swamp as the number of consumers and providers increase. The absence of clear data domains and ownership also leads to decreased data quality over time. Specialized data engineers managing the BD stack may become siloed, unaware of the data's semantics, value, and domain, further reducing data quality, and making maintenance and scaling a challenging task.

Cloud Based Solutions: Given the high cost of running an on-premise data engineering cluster, talent gap in the market, and complex infrastructure provisioning for increasing data processing loads, companies may opt for deploying their BD solutions on the Cloud. The current technology generation favors Lambda or Kappa architectures for stream or batch processing or frameworks like Databricks or Apache Beam that unify the two. Although these architectures may reduce cost and complexity for data architects, they still suffer from the same challenges of undefined data domains, siloed data engineers, and a monolithic pipeline architecture.

To address these issues, we explore a domain-driven distributed RA for BD systems and propose an RA that addresses some of the challenges. This RA is inspired by the concept of data mesh (Dehghani 2022) and domain-driven design (Evans and Evans 2004).

Why a Reference Architecture?

Software architecture is an artefact that aims to satisfy business objectives through a software solution that is adaptable, cost-efficient, maintainable, and scalable. In addition, it allows for the capture of design issues at an early stage in the development process. While this practice can be applied to any class of systems, it is particularly useful in the design and development of complex system such as BD [*citation hidden*]. Despite the known complexity of BD systems, the development, analysis, and design of an RA that incorporates best practices, techniques, and patterns and that supports the achievement of BD goals is possible (Ataei and Litchfield 2022). Therefore, engineers and architects can better absorb the complexity of BD system development and make it tractable. This can be seen in Software Product Line (SPL) development where RAs that provide generic artefacts can be instantiated and configured for use in specific system domains (Cloutier et al. 2010; Derras et al.).

Therefore, RAs are effective for addressing complex BD system development, because: 1) RAs promote adherence to best practice, patterns, and standards, 2) RAs can endow the architecture team with increased openness and interoperability, incorporating architectural patterns that provide desirable quality attributes, and 3) RAs serve as the locus of communication, bringing various stakeholders together.

Related Work

The application of RAs for solving challenges in data architecture is not a new concept and the production of BD RAs has had support from governmental agencies, for example the NIST BD RA (NBDRA) (Chang and Boyd 2018), and from industry, for example IBM (Quintero et al. 2019), Microsoft (Levin 2013), and ISO (ISO/IEC 2020). Numerous domain specific RAs have been developed, for example, in national security (Klein et al.). Some effort has been put into the adoption of microservices architecture for BD systems such as Neomycelia [*citation hidden*] and Phi (Maamouri et al.).

The RAs investigated may be instantiated but outwardly, most appear to be conceptual studies and do not provide sufficient data on cross-cutting concerns such as data quality, data integrity, data ownership, security, and privacy. Therefore, by providing a domain-driven distributed architecture for BD systems, this study extends current learnings by addressing the current limitations of BD RAs. The RA presented here focuses on the logical separation of data into domains through event-driven communication and with clearly defined boundaries [*citation hidden*].

Method

The study is comprised of two phases: the first is the identification of high-level requirements of the artefact, and second applies a well-established methodology for building the RA. These phases are elaborated in the following sub-sections.

Requirement Specification

Prior to RA modelling and design, the desired properties of the artefact are defined as requirements. System and software requirements range from a sketch on a napkin to formal (mathematical) specifications. Therefore, the kind of requirements for the purpose of this study are defined in an exploration of the body of evidence. [*citation hidden*].

Classifying software and system requirements is a frequently debated topic. Sommerville (2011) uses three levels: user, system, and design specs. This study adopts Laplante (2017) more general framework with functional, non-functional, and domain requirements. The aim is to define high-level requirements for BD systems. We analyzed current Big Data requirements and realized that Velocity, Veracity, Volume, Variety, and Value are common ways to classify them. The NIST Big Data Public Working Group conducted a large-scale study on requirements from various domains, resulting in seven general categories ((NIST) 2015). Other studies, such as Volk et al. (2020) and Bashari Rad et al. (2016), focus on use cases, security, and privacy requirements.

Our approach to determining high-level software and system requirements for BD RAs involves analyzing studies and utilizing a rigorous method for model verification. We organize our functional requirements according to ISO/IEC 29148 (ISO/IEC 2018) guidelines and system modes, categorizing them by BD characteristics such as value, variety, velocity, veracity, volume, security, and privacy (Table 1).

Category	Requirements
Volume	Vol-1) System shall support asynchronous, streaming, and batch processing to collect data from centralized, distributed, and other sources, Vol-2) System shall provide scalable storage.
Velocity	Vel-1) System shall support variable intensity of data transmission between data sources, Vel-2) System needs to stream data to data consumers in a timely manner, Vel-3) System shall provide the capability 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 may provide the capability to process data in real-time or near real-time.
Variety	Var-1) System shall support data in various formats ranging from structured to semi-structured and unstructured data, Var-2) System shall support aggregation, standardization, and normalization of data from disparate sources, Var-3) System shall support adaptations mechanisms for schema evolution, Var-4) System may provide mechanisms to automatically include new data sources
Value	Val-1) Shall provide capability to handle compute-intensive analytics and machine learning processes, Val-2) System shall provide support for batch and stream processing, Val-3) System shall support different output file formats for different purposes, Val-4) System shall provide support for streaming of results to consumers.
Security & Privacy	SaP-1) System shall provide protection and retention of privacy and security measures for sensitive data, SaP-2) System shall provide protection of data and processing nodes via

	access control methods and policy-driven authentication at multiple levels.
Veracity	Ver-1) System shall support data quality curation including classification, preprocessing, format, reduction, and transformation, Ver-2) System shall support data provenance including data life cycle management and long-term preservation.

Table 1: BD system requirements

Artefact Development Method

There are several approaches to the systematic development of RAs. Cloutier et al. (2010) demonstrate a high-level model for the development of RAs through the collection of contemporary architectural patterns and advancements. Bayer et al. (1999) introduce a method for the creation of RAs for product line development called PuLSE DSSA. Similarly, Nakagawa et al. (2014) present a four-step approach to the design and development of RAs. Additionally, Galster and Avgeriou (2011) propose an empirically grounded RA methodology, which is the most appropriate methodology for this study due to its widespread adoption and alignment with the study objectives.

However, the methodology required additional approaches to achieve the desired level of rigor and relevance. The lack of comprehensive guidelines for collecting empirical data in step 3 made it difficult to approach data collection, synthesis, and modeling. To address this, Nakagawa et al. (2009) provided investigation guidelines and the RAModel concept. Additionally, the methodology lacked a systematic and stronger evaluation approach for the RA. To remedy this, an instantiation of the RA was deployed in a real-world practice, and the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al.) was used to evaluate the artifact. The methodology constitutes six steps as below:

1. **Decision on type of RA**: Before developing the RA, the type must be identified using Angelov et al. (2009) classification framework, which categorizes RAs into standardization and facilitation RAs. The selected domain-driven distributed RA for this study aims to promote interoperability and facilitate BD system development. The resulting artefact is classified as a standardization RA that can be implemented in multiple organizations.
2. **Selection of Design Strategy**: Galster and Avgeriou (2011 suggest that RAs can be created in two ways: 1) using existing patterns, principles, and architectures, or 2) building from scratch, which is uncommon and typically occurs in underdeveloped areas. However, RAs are more effective when developed using established practices (Cloutier et al. 2010). The RA developed for this study is research-based and draws from existing RAs, concrete architectures, patterns, standards, and best practices.
3. **Empirical Acquisition of Data:** To improve transparency and systematization, we added a systematic literature review (SLR) to our methodology. We used the ProSA-RA's "information sources investigation" to collect data from publications, with the aim of identifying essential architectural elements. The SLR focused on finding common architectural constructs and limitations in existing BD RAs. This helped us determine the appropriate method for creating and presenting our own artifact. The details of this SLR are discussed in [citation hidden].
4. **Construction of the RA:** Using Table 1 and SLR findings, we began developing the RA, focusing on component and communication channel identification. We researched integration models, trade-offs, and best practices, and followed ISO/IEC/IEEE (2017) guidelines using Archimate as the architectural description language.
5. **Enabling RA with variability:** One of the integral elements that help with instantiation of the RA is variability. This enables RA to remain useful as a priori artefact when it comes down to organization-specific regulations, and regional policies that may constrain the architect's freedom in design decisions. For the purpose of this study, we chose to represent variability by the means of "annotation" as recommended as one of the three accepted approaches by Galster and Avgeriou (2011
6. **Evaluation of the RA:** The final phase of the methodology tests the RA's effectiveness and usability, following Galster and Avgeriou (2011 criteria of utility, correctness, and efficiency. However, most

established methods for assessing architectures are not suitable for RAs due to their level of abstraction and undefined stakeholders. Angelov et al. (2008) adapted ATAM for RAs, but for this study, we instantiated and evaluated the prototype in practice.

Domain-driven Distributed RA for BD Systems

The principles upon which the RA is built requires that it is 1) Domain-driven: to address data quality, siloed teams, data swamp issues and communication issues, affecting velocity, variety, and veracity requirements; 2) Distributed: to address the challenges of scaling monolithic data systems affecting velocity, and volume requirements; 3) Data as a service: to allow for increased discoverability of data and autonomy of various analysis and data science teams without frictions with data engineers affecting value, variety, veracity, security and privacy requirements; 4) Governance through a federated service: to prevent team-based and rather immature decisions that may not be in-line with global organizational visions, policies, standards and procedures, affecting all requirements; 5) Event driven: to address point to-point communication issues that arises in distributed systems, affecting velocity requirement. Additionally, the elements of the RA that are annotated with the phrase 'variable' can be modified, adjusted, or even omitted based on the architect's decision.

To produce the RA (Figure 1), design iterations have been inspired by microservices architectural patterns (Richardson 2018). To shift from collecting data in monolithic data lakes and data warehouses to converging data through a decentralized and distributed mesh of data products communicating through standard interfaces, the RA shall be domain-driven and distributed. This addresses the limitations in current BD architectures. This RA comprises 11 principal and 9 variable components, discussed below:

Ingress and Egress service: Ingress service is responsible for controlling traffic into the system. Depending on the type of request, this service will load balance either into a batch processing controller or a stream processing controller. Ingress is an asynchronous load balancer designed to eliminate choke points, handle SSL termination, and provide with extra features such as named-based virtual hosting. This component addresses the requirements Vol-1, Vol-2, Var-1, Var-3, Var-4, Val-1, Val-3, Val-4, SaP-1 and SaP-2. Additionally, egress service is responsible for providing necessary APIs to the consumers of the system, third parties or other BD systems. This allows for the openness of the architecture and lets data scientist and business analyst easily request the data necessary for their workloads. This also promotes the idea of self-serve-data through service discovery, data catalogue and product domains. This component addresses the requirements Vel-2, Vel-4, Val-3, Val-4, SaP-1, and SaP-2.

Batch and Stream Processing Controller: Batch processing controller is responsible for handling batch processes. That is, it is responsible for receiving request for batch processing, and communicating it to the event broker. Due to the batch nature of the requests, the controller can decide to achieve this in a bulk and asynchronous manner. This component addresses the requirements Vel-1, Val-1, and Val-2. Stream processing controller achieves similar thing to the batch one, with a difference that it must handle a different nature of requests. Stream events are synchronous in nature and require high through-put. Having a specific service for stream processing requirements promote tailored customization that best suit the varying nature of stream events. In addition, this controller can provide with extended buffering mechanisms increasing reliability and response rate of the system. This component addresses the requirements Vol-1, Vel-1, Vel-2, Vel-4, Vel-5, and Val-2.

Event Broker: An important architectural construct designed to achieve inversion of control. As the system grows, more nodes and services are added, communication channels increase, and there is a need for new events to be dispatched. As each service communicates through the event backbone, each service will be required to implement its own event handling module. This can easily turn into a spaghetti of incompatible implementations by different teams and can even result in unexpected behaviors. To address this issue, an event broker is introduced to each service which has one main responsibility: communication with event backbone. This component indirectly addresses the requirements Val-1, and Ver-1.

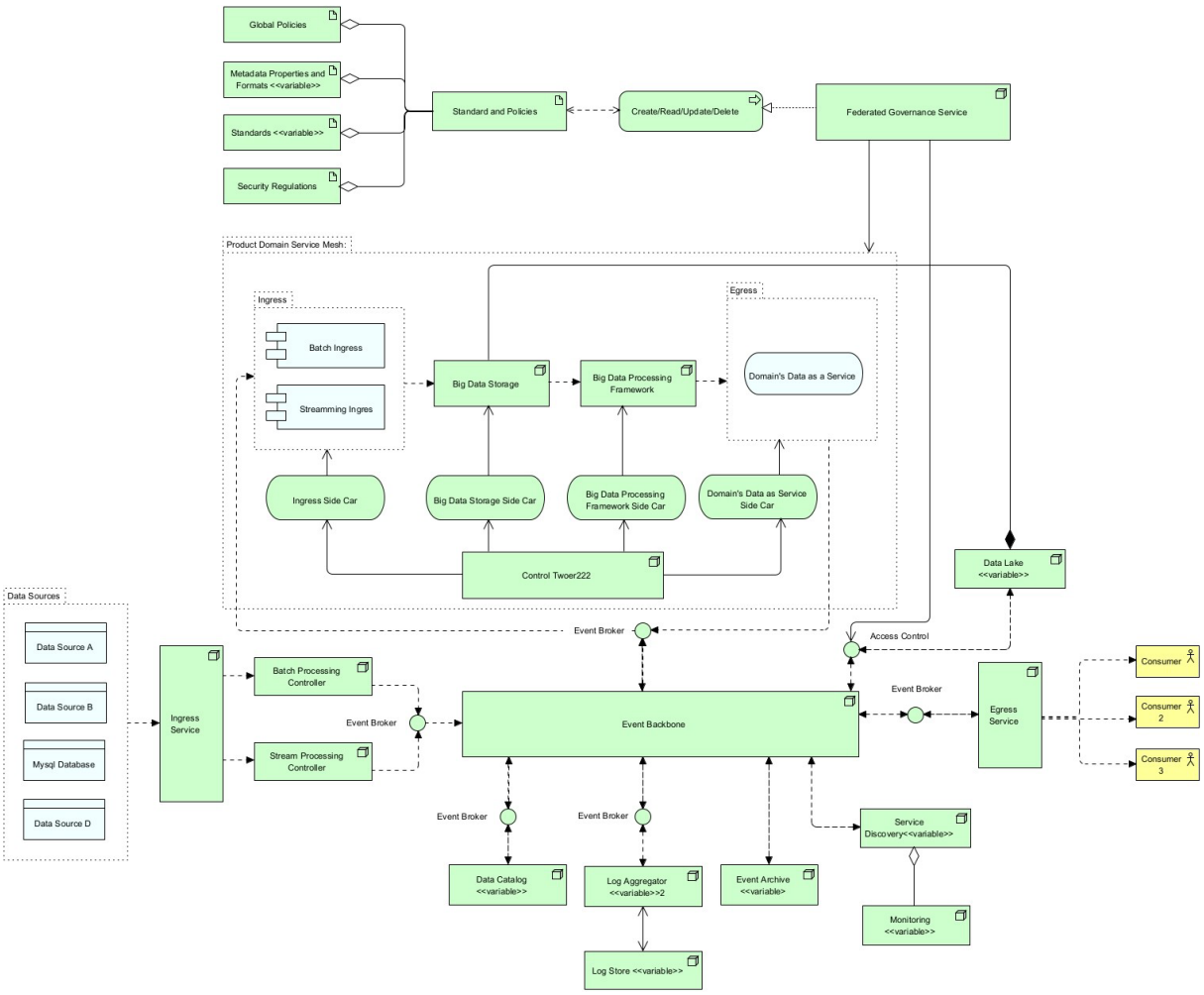

Figure 1: domain-driven Distributed RA

Event Backbone: Event backbone is the heartbeat of the system, facilitating communication between all services. Whereas this service is displayed as one technology service in the Archimate diagram, we recommend the event backbone to be designed underlying distributed paradigms itself. This is to ensure scalability as the number of topics and events grows. Event backbone and its relationship to other nodes is analogous to a dance troupe in which dancers move to the rhythm relative to their position. In this case, the event backbone is the music and services are the dancers. Thus, services are only responsible for dispatching events in a 'dispatch and forget' model, subscribing to topics they are interested. This component addresses the requirements Vel-1, Vel-2, Vel-3, Vel-4, Vel-5, Val-1, Val-2, Ver-1, Ver-2, and Ver-3.

Product Domain Service Mesh: Driven by the idea of domain-driven design, every product has it is own bounded context and ubiquitous language and is technically governed by a service mesh. Product domain is one of the highlights of our RA, introducing the concept of 'data product' as the architectural quantum. Product domain service mesh is the node in the architecture that encapsulates 6 structural components: ingress, BD storage, BD processing framework, domain's data service, control tower and the side car. These components provide the necessary means for the domain to achieve its ends regarding BD processing with high cohesiveness, low coupling, and clear interfaces. Product domain service mesh is the aggregation of all components necessary: data, code, and infrastructure as the elements of the bounded context. This component indirectly addresses Vol-1, Vel-3, Vel-4, Vel-5, Var-1, Var-2, Var-3, Val-1, Val-2, Val-3, Val-4, Sap-1, SaP-2, Ver-1, Ver-2, and Ver-3.

Federated Governance Service: Given the distributed nature of the architecture and sheer number of moving parts with varying life cycles; there is a need for some global contextual standards and policies that are designed to streamline processes and avoid losses. This is not to limit the autonomy of teams, but to inject them with best practices and organizational policies that tend to reflect the capability framework, regional limitations, and legal matters that can cause severe damage to the business. This component can indirectly affect all requirements.

Data Catalog: As data products increase in the system, more data become available, interoperability increases, and thus services have to know who provides what data. Data catalog is responsible for keeping a catalog of all data available among services with relative paths to fetch those data. This component addresses the requirements Vel-4, Var-1, Var-3, and Var-4.

Log Aggregator and Log Store: Operating underlying a distributed paradigm, requires a shift in a way that logging occurs. This means systems cannot rely only on applications reporting logs in a single environment, but there's a need for a distributed tracing that shows a lifecycle of a process and how it went through different services. Therefore, this RA benefits from the popular log aggregator pattern initially released by the microservices community. This allows for a graceful scaling of system's logging strategy. This component indirectly addresses the requirements Vol-1, Vel-1, Val-1, and Ver-1.

Event Archive: One of the main challenges of this architecture is its reliance on event backbone. Whereas event backbone itself is recommended to be distributed and fault tolerant, event archive further solidifies the service recovery from unexpected events. This implies that, if the event backbone went out of service, the history of events can be stored and retrieved from the event archive to bring various services to the current state of operation. This component indirectly addresses the requirements Vol-1, Vel-1, Val-1, and Ver-1.

Data Lake: Whereas product domains are demarcated, and boundaries are well-defined, we do not find it necessary for each domain to maintain it is own data lake. This is under the assumption that a lot of data are now processed at the time of storage, and is required whenever there is a analytical business case for it. Whereas there isn't a data lake per domain, different domains can have a quota in the data lake that is owned and handled by access control. This component addresses the requirements Vol-2, Vel-1, Var-1, Var-3, Var-4, and Val-3.

Service Discovery: In a distributed environment, services need to find each other to communicate their means. Service discovery solves this issue with primary responsibility of identifying services and answering queries about services. This is achieved by services registering themselves to service discovery on boot up. This component indirectly addresses the requirements Vel-2, Vel-4, Var-2, Var-4, Val-3, Val-4, and SaP-2.

Monitoring: To take proactive measures for the overall health of the system and its considerable moving parts, one needs to actively monitor the state of the individual nodes and the overall flow of things. Services emit large amounts of multi-dimensional telemetry data that can be read and analyzed for the supporting actions. Monitoring services help with storing these data to fuel proactive actions. This component indirectly addresses all requirements.

Evaluation

The aim of evaluating the RA is to test its practical application and usefulness in solving real-world problems. We chose to use ATAM for this evaluation because it is a widely recognized method that provides rigor and aligns with our conceptual constructs. ATAM helped identify key tradeoffs, risks, and sensitivity points that improved our confidence in the RA. We tailored the ATAM evaluation to our study by creating a prototype of the architecture and testing it in practice. We used ISO/IEC 25000 (ISO/IEC 2014) to select the technology for the instantiation of the RA.

The RA components used in the instance include Node JS for all APIs, Nginx for ingress, AWS Lambdas for stream and batch processing controllers, Kafka for event backbone, Kafka event brokers as the event broker, AWS application load balancer as the egress load balancer, Istio as the control tower, Envoy as the side car, Kubernetes as the container orchestrator, AWS S3 as the BD store and event

archive, and Data Bricks for stream and batch processing. Although logging, monitoring, service discovery, federated governance service, and data catalog were omitted, it did not impact the evaluation negatively. We do not describe the ATAM steps in detail but explain the evaluation process instead. To ensure security and intellectual property of the practice, some evaluation detail is omitted but that does not affect the integrity of the evaluation.

Phase 1

Evaluation was conducted in a veterinary practice management software company that provides SaaS services globally. Key stakeholders such as lead architects were consulted in the ATAM process. Two lead development architects, the head of product, a quality assurance engineer, and three developers were involved. The initial meeting explained the purpose of ATAM, and in step 2, stakeholders discussed business challenges and goals. In step 3, the prototype was presented with assumptions and variability points. Then architectural styles to achieve quality attributes were agreed upon. For availability, Kafka's partitions, Nginx worker connections, Data Lake and Istio were identified. For performance, Nginx asynchronous processing, Kafka topics and consumers, AWS application load balancer, and Kubernetes deployments were identified. For modifiability, the concept of domain-driven design, side cars, and event brokers were discussed. The approaches were then analyzed for tradeoffs, sensitivity points, and potential risks.

To generate a utility tree, consensus on the most important quality attributes for the evaluation was required. Assumptions were presented and after discussion, considering concerns over privacy, agreement was reached that availability, performance, and maintainability were chosen. The utility tree was then created with the requirements: 1) performance: the system should be able to process real time streams under 1200 ms, queries from a data scientist should not take more than 2 hours 2) availability: the load balancer and data bricks cluster shall have 99.999% availability, and 3) modifiability: the new product domain should deliver within a month and with less than 5 persons. Due to resource constraints, we skipped a preliminary analysis of architectural approaches in phase 1 and only conducted it after the scenarios had been prioritized. This did not negatively affect the evaluation process.

Phase 2

Scenarios are the quanta of ATAM and help capture architectural stimuli. Thus, for this step, stakeholders were asked to prioritize three classes of scenario, growth, use-case, and exploratory scenarios. From this, 20 scenarios were pooled that stakeholders voted on. The voting process yielded 5 scenarios, described as two user journeys: 1) The pet owner brings the pet to the veterinary hospital, the pet is diagnosed with cancer wherein the pet's environmental factors should be studied for potential clues for the root cause of cancer; 2) a pet owner brings the pet to the veterinary hospital, the cat's symptoms should be processed for early detection of Lyme disease.

After identifying architectural approaches and prioritizing scenarios, we ran the scenarios against our prototype. This provides opportunity for heuristic qualitative analysis and identification of sensitivity points and tradeoffs. The process was initiated by creating a custom script to extract data from the company's mySQL database and send it through the ingress process. This was done through Kafkaconnect and Debezium. The topics for Kafka were created, then Nginx was configured to pass the requests to responsible lambdas for batch and stream processing. We then followed with event producers, Istio, Envoy, Kubernets, Data Bricks and the rest of the system. How architectural decisions contribute to the realization of each scenario was explained.

Results

While running the scenario simulations against the artifact, the architectural approaches were constantly assessed. Many implementation issues arose, and sensitivity points noted. The true cost of the system, its tradeoffs, and potential challenges were realized. Based on these and stakeholder feedback, it was found that system quality, Q_S, is a function, f, of the quality attributes of performance, Q_P, availability, Q_A, and modifiability, Q_M, expressed as the equation $Q_S = f(Q_P, Q_A, Q_M)$.

For performance, cloud stress testing agent StressStimulus was applied. The stress test was run against the system and revealed the cold start time (100-1000ms) of AWS Lambdas impact performance. However, for an accurate evaluation, we opted not to use micro-batch while using Data Bricks for stream processing. Also, to test the worst-case scenario, the fair scheduling pool was not configured.

Performance tests of the prototype included periodic data dispatch, large volume of data, and many concurrent requests. It became evident that latency, input/output, and object mutations negatively impacted performance. The event driven design produced better outcomes when handling simulations. Therefore the system is sensitive to latency (l), side effects (s), and concurrency (c), expressed as $Q_P = h(l,s,c)$.

Next, the prototype was tested for availability. Since the system is distributed, a poorly handled failure in one service can create a ripple effect through other services. Through the implementation of circuit breakers in event brokers, the prototype can recover from this situation. To achieve a stable state, the prototype archives events in the event backbone before any failure. Moreover, health checks and alarms on pods in the Kubernetes cluster provide monitoring.

Kubernetes ensures a minimum quantity of services are available through deployments and replica sets. The evaluation demonstrates that the architecture complies with 12 factor methodology and is deemed cloud native, which positively affected the availability score. Therefore, system availability is affected by the time it takes for circuit breakers to trip and become available again (μ_C), failure of the event backbone (λ_E), and the time it takes for the services to recover (μ_S), with g being the fraction of time that the system is operating. Thus, system availability is expressed as $Q_A = g(\mu_C, \lambda_E, \mu_S)$.

Finally, the prototype was tested for modifiability. The distributed and domain-driven nature of the architecture made it easy to achieve the desired modifiability objectives. Adding a new data domain only required an extension of the HCL module, written in Terraform for the EKS cluster, and modification of the Docker images. Brokers were also streamlined, so a new broker can be stood up within minutes. The certification lifecycle is handled by Istio, Local Cert Manager and Let's Encrypt. Therefore, system modifiability is sensitive to the provisioning, maintenance, and configuration of Kafka (K_a), Kubernetes (K_u), and Databricks (D), and the skill set (s) required to achieve these. Thus, system modifiability is expressed as $Q_M = s(K_a,K_u,D)$.

After analysis, two tradeoff points are identified: 1) Event brokers and the event backbone, and 2) service mesh. The event backbone was of concern and how it might turn into a bloated architectural component like Enterprise Service Bus (ESB) in Service Oriented Architectures (SOAs). However, given the nature of the distributed system, the event archive, and event brokers, that is unlikely to be an issue because the event backbone is responsible for one function only, to provide communication between services. In addition, in the case of service outage, the order of events can be retrieved from the event archive (dead letter queue), providing a stable state for the system. Event brokers facilitate modifiability by providing native event handling mechanisms but at a cost of one more layer and the potential latency that comes with it. Therefore, while providing positive impact on performance and maintainability, the event backbone can have negative impact on availability and reliability. Furthermore, event brokers provide positive impact on modifiability and availability without negatively affecting performance.

The service mesh was a subject of concern. Getting service mesh operational required significant effort from developers. However, modifiability is positively affected longitudinally, and the service mesh promotes the concept of clear interfaces, separation of concerns, and a well-defined bounded context. Therefore, the service mesh positively impacts modifiability but may negatively impact performance due to network communications between services and proxying.

Conclusion

BD engineering is a sophisticated process and while there are many good practices in software engineering, the data engineering domain does not appear to obtain all those benefits. Consequently, several challenges in the development of BD systems exist where projects fail to identify the potential of data-driven decision making. We have aimed to facilitate this process by proposing a BD RA. Nevertheless, more research is required in data processing, reactive event driven data processing

systems, data engineering and BD architectures and with these, security, privacy, and metadata management for BD architectures.

References

(NIST), N. I. o. S. a. T. 2015. "Nist Big Data Interoperability Framework: Volume 5, Architectures White Paper Survey."

Angelov, S., Grefen, P., and Greefhorst, D. "A Classification of Software Reference Architectures: Analyzing Their Success and Effectiveness," *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pp. 141–150-141–150.

Angelov, S., Trienekens, J. J. M., and Grefen, P. "Towards a Method for the Evaluation of Reference Architectures: Experiences from a Case," *European Conference on Software Architecture*: Springer, pp. 225–240-225–240.

Ataei, P., and Litchfield, A. 2022. "The State of Big Data Reference Architectures: A Systematic Literature Review," *IEEE Access*).

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M. "Pulse: A Methodology to Develop Software Product Lines," *Proceedings of the 1999 symposium on Software reusability*, pp. 122–131-122–131.

Chang, W. L., and Boyd, D. 2018. "Nist Big Data Interoperability Framework: Volume 6, Big Data Reference Architecture."

Cloutier, R., Muller, G., Verma, D., Nilchiani, R., Hole, E., and Bone, M. 2010. "The Concept of Reference Architectures," *Systems Engineering* (13:1), pp. 14-27.

Dehghani, Z. 2022. *Data Mesh: Delivering Data-Driven Value at Scale*. O'Reilly Media.

Derras, M., Deruelle, L., Douin, J.-M., Levy, N., Losavio, F., Pollet, Y., and Reiner, V. "Reference Architecture Design: A Practical Approach," *ICSOFT*, pp. 633-640.

Evans, E., and Evans, E. J. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

Galster, M., and Avgeriou, P. 2011 "Empirically-Grounded Reference Architectures," *Joint ACM*).

ISO/IEC. 2014. "Iso/Iec 25000:2005. Software Engineering — Software Product Quality Requirements and Evaluation (Square) — Guide to Square."

ISO/IEC. 2018. "Iso/Iec 29148:2018. Systems and Software Engineering — Life Cycle Processes — Requirements Engineering."

ISO/IEC. 2020. "Iso/Iec Tr 20547-1:2020. Information Technology — Big Data Reference Architecture — Part 1: Framework and Application Process."

ISO/IEC/IEEE. 2017. "Iso/Iec/Ieee 42010:2011. Systems and Software Engineering — Architecture Description."

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J. "The Architecture Tradeoff Analysis Method," *Proceedings. fourth ieee international conference on engineering of complex computer systems (cat. no. 98ex193)*: IEEE, pp. 68–78-68–78.

Klein, J., Buglak, R., Blockow, D., Wuttke, T., and Cooper, B. "A Reference Architecture for Big Data Systems in the National Security Domain," *2016 IEEE/ACM 2nd International Workshop on Big Data Software Engineering (BIGDSE)*: IEEE, pp. 51–57-51–57.

Laplante, P. A. 2017. *Requirements Engineering for Software and Systems*. Auerbach Publications.

Levin, B. O. 2013. "Big Data Ecosystem Reference Architecture," *Microsoft Corporation*).

Maamouri, A., Sfaxi, L., and Robbana, R. "Phi: A Generic Microservices-Based Big Data Architecture," *European, Mediterranean, and Middle Eastern Conference on Information Systems*, pp. 3–16-13–16.

Nakagawa, E. Y., Martins, R. M., Felizardo, K. R., and Maldonado, J. C. "Towards a Process to Design Aspect-Oriented Reference Architectures," *XXXV Latin American Informatics Conference (CLEI) 2009*.

Nakagawa, E. Y., Oquendo, F., and Becker, M. "Ramodel: A Reference Model for Reference Architectures," *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 297–301-297–301.

Partners, N. 2021. "Big Data and Ai Executive Survey 2021." NewVantage Partners.

Quintero, D., Lee, F. N., and others. 2019. *Ibm Reference Architecture for High Performance Data and Ai in Healthcare and Life Sciences*. IBM Redbooks.

Richardson, C. 2018. *Microservices Patterns: With Examples in Java*. Simon and Schuster.

White, A. 2019. "Our Top Data and Analytics Predicts for 2019." Gartner.