NeoMycelia: A software reference architecturefor big data systems

1st Pouya Ataei
School of Engineering, Computer and Mathematical Sciences
Auckland, New Zealnd
pouya.ataei@aut.ac.nz

2nd Alan Litchfield
Service and Cloud Computing Research Lab
Auckland, New Zealand
alan.litchfield@aut.ac.nz

***Abstract*—The big data revolution began when the volume, velocity, and variety of data completely overwhelmed the systems used to store, manipulate and analyze that data. As a result, a new class of software systems emerged called big data systems. While many attempted to harness the power of these new systems, it is estimated that approximately 75% of the big data projects have failed within the last decade. One of the root causes of this is software engineering and architecture aspect of these systems. This paper aims to facilitate big data system development by introducing a software reference architecture. The work provides an event driven microservices architecture that addresses specific limitations in current big data reference architectures (RA). The artefact development has followed the principles of empirically grounded RAs. The RA has been evaluated by developing a prototype that solves a real-world problem in practice. At the end, succesful implementation of the reference architecture have been presented. The results displayed a good degree of applicability with respect to quality factors.**

***Index Terms*—Reference architecture, Architecture, Big data reference architecture, Big data architecture, Big data systems, Big data software engineering, Event driven, Microservices**

I. Introduction

The ubiquity of digital devices, the networking infrastructure of today, and the proliferation of software applications, have augmented users' capability to produce data at an unprecedented rate [1]. In a world where we have an average processing power of 1.5 GHz on smart phones, and up to 8 GHz on laptops running on a network infrastructure that will support up to 25 Mbps of transmission per second, data becomes the new oil, the atom, the dot that lays the foundation of a nexus [2].

Big data (BD) is a term that was initially coined to refer to the gradual growth and availability of data [3]. BD is an endeavor to harness patterns behind vast amounts of data for the purposes of improvement, control, and prediction. Roughly 10 years ago, the BD revolution began when the volume, velocity, and variety of data completely overwhelmed the systems used to store, manipulate and analyze that data [4], [5]. The concept of BD is a game-changing innovation [6], heralds the dawn of a new industrial revolution [7], and creates a new category of economic asset.

Nevertheless, BD is not always better data or a magic wand that enchants any business or process. Actually, it can very easily cause losses [8]. It is estimated that approximately 75% of the BD projects have failed within the last decade according to multiple sources [9], [10], [11], [4]. Among challenges of adopting BD, the most repeatedly discussed are 1) Architectural and system development challenges, 2) Organizational challenges, and 3) Rapid technology change [6], [1], [12]. The focus of this study is on architectural and system development challenges.

Today, most BD systems are developed as ad-hoc and complicated architectural solutions that do not tend to adhere to many principles of software engineering [13], [14]. In addition, as the ecosystem of BD grows and new technologies are introduced, architects will have harder time to select and orchestrate the right technology to produce the right results [14].

This can create a foundation for an immature architectural decision that results in a solution that is hard to maintain, hard to scale, and may raise high-entry barriers. Since the approach of ad-hoc design for BD systems is undesirable and leaves many engineers in the dark, novel software engineering approaches specialized for BD systems are required. To contribute towards this goal, we explore the notion of RAs and present a software reference architecture, Neomycelia.

In the case of ambiguity towards what should be developed to address what needs, RA can play an overarching role to describe the building blocks of the system and the ways in which these blocks communicate to achieve the overall goal of the system [15]. This in turn produces manageable modules that each address a different aspect of the problem, and provides stakeholders with a high-level medium to observe, reflect upon, communicate with and add into.

II. Why Reference Architectures?

BD is an interplay of analytics methods, software engineering through development and data engineering, and organizational workflows [16], [17]. Such complex systems are best approached through the lens of architecture and well-thought out design documents. Utilization of RAs for complex systems is not something new.

In fact, practitioners of complex systems, software engineers, and system designers have been frequently using reference architectures to have a collective understanding of system components, functionalities, data-flows and patterns which shape the overall qualities of system and help further adjust it to the business objectives [18], [19]. In software product

line (SPL) development, reference architectures are generic schemas that can be configured and instantiated for a particular domain of systems [20]. In software engineering, reference architecture (RA) can be defined as means to represent and transfer knowledge that bridges from the problem domain to a family of solutions [21].

RAs serve as a mechanism that embodies domain relevant qualities and concepts, breaks down the solutions, generates a terminology to facilitate effective communication, and illuminates various stakeholders and system designers [21]. This allows RAs to provide an opportunity for early identification of design issues, when making changes is still cheap. This has several side-benefits such as: 1) Ensuring cross-cutting concerns are addressed, 2) Scales the knowledge of architects and engineers across the organization, 3) Helps achieving consensus around major design choices, 4) Creates the foundation of organization memory around design decisions, 5) Acts as a blueprint and a summary artefact in the portfolio of the architects and software engineers

III. Research Methodology

To increase systematicity and allow for reproducibility of Neomycelia, we follow the guidelines presented in empirically-grounded RAs [22]. In essence, "empirically grounded" implies two major aspects: 1) "empirical foundation" which implies that Neomycelia must be grounded in proven principles (domain problems, practical concepts), and 2) "empirical validity" which implies that RA needs to be evaluated for applicability and validity. This research methodology is based on two main pillars: 1) existing RAs and 2) literature on RAs. The process follows these 6 steps:

1. Decision on type of the RA: A classification framework presented by Angelov et al ([23]) is applied. In this study, five types of RAs have been described from which our RA matches type five (facilitation architectures designed to be implemented in multiple organizations). Thus, this RA aims to facilitate the design of BD systems across multiple organizations. Examples of similar RAs are ERA [24], AHA [25], and eSRA [26]

2. Design Strategy: There are two general approaches to the development of RAs; developing RAs from scratch or from existing architectures. Our RA is developed based on existing architectures and available literature.

3. Empirical acquisition of data: Data acquisition consists of two major phases, data sources identification and capturing architecture data. It is proposed by Nakagawa et al ([27]) that good data sources for classical RA development can be people (researchers, practitioners), available literature (publications, technical reports, white papers) and systems (source codes, documentations). Howbeit, the guidelines presented by Galster and Avgeriou ([22]), provide no means or instructions on how these data should be identified and captured.

Therefore, to increase systematicity and transparency of this research, we conducted a systematic literature review (SLR) to capture current best evidence from the available literature. For this purpose we follow the guidelines of PRISMA presented by Shamseer et al. ([28]).

Our aim was to find all available BD reference architectures in literature in order to identify common components of big data systems. This has grounded a solid formation for development of Neomycelia.

We've selected IEEE Explore, ScienceDirect, SpringerLink, ACM library, MIS Quarterly, Elsevier, AISel as well as citation databases such as Scopus, Web of Science, Google Scholar, and Research Gate. The search keywords used are 'Big Data Reference Architectures', 'Reference Architectures in the domain of Big Data', and 'Reference Architectures and Big Data';

In the first phase of the SLR (identification), 84 literature has been pooled. Out of this pool, 57 study has been selected based on our inclusion, exclusion and quality criteria. Studied that provided with detailed analysis and practice have been included. Studied that provided with substantial case studies have been included. Papers that discussed current BD RAs, it's ecosystems and drivers have been included. Papers that are recent (in the range of 2010-2020) have been included.

On the other hand, papers that are duplicates, do not directly address the SLR aim, and are not written in English are excluded. For our quality factors, we paid extensive attention to how rich the study is in terms of its case studies and relevance to practice. The length and volume of the information provided by the studies has been considered as well. Very short and information lacking papers did not get pass through the quality assessment framework.

In the pool of selected articles, 24.5% are from SpringerLink, 16% are from ACM, 33% are from IEEE Explore, 5.2% are from ScienceDirect, and the rest are from Google Scholar. 13 conference proceedings, 30 journal articles, 12 book chapters, and a whitepaper has been selected. 33% belonged to years 2013-2015, 51% of the articles were selected from the years 2016-2020, and the rest to years 2010-2013.

We used the software Nvivo for classifying and coding the literature. We defined 3 nodes namely 'big data architecture data', 'big data reference architecture limitations', and 'big data components". Once we coded and attributed texts to our nodes, we then synthesized and inferred findings.

The result of this SLR administered 23 RAs from extant literature, 18 RAs from academia, 4 from practice, 1 through the collaboration of both domains. Majority of the RAs have been in the form of short papers, but there has been few detailed RAs as well. The exact detail and listing of the RAs are out of the scope of this study, however, there will be mentions to various RAs for comparison, inference purposes.

We found three common components among all RAs which are; BD management and storage nodes (relational, non-relational, graph, data lake, data finery, polyglot persistence), BD infrastructure nodes (latency, data transformation, in-memory data grids), and BD analytics and application nodes (real-time processing, batch processing, predictive analysis, spatial analysis). This underpinned our understanding for actual design and construction of the RA.

4. Construction of the RA: Based on the findings captured in previous step, the initial design of the RA took place.

Integral to this phase, was the underlying method to creation and design of RA. We followed ISO/IEC 42010 for architectural descriptions [29]. The standard mostly revolves around concrete architectures and because of that, our descriptions do not 100% conform to it.

For instance, the standard has bolded the identification of system stakeholders (clause 5), however RAs are highly abstract and do not have a clear group of stakeholders [30]. Another focal point in conveyance of architectural descriptions is the concept of views. In this case, ISO/IEC 42010, being the standard for concrete architectures, prescribe architecture views and viewpoints in the context of business and actual models (clause 4). This does not apply to this RA as well.

Beneken et al ([31]) classified three different kind of RAs based on views, namely functional, logical and technical. Along the lines, Vogel et al. [32] classified RAs based on their usage context, as platform-specific, industry specific, industry crosscutting, and product line RAs.

Whereas different academic efforts aimed at classifying RAs based on different criteria, arguably several distinct views can adhere to one logical view as it has been seen in the case of pattern based reference architecture for serviced based system conducted by Stricker et al. ([33]). This implies, that modules defined in technical review can potentially refine the modules of the logical view. Furthermore, Cloutier et al. ([19]) suggests that a RA should address business, technical and customer context's views.

For the purposes of this study we do not address the business view of the RA, as this software RA aims at describing a functional, technical and logical views of a BD system. Business views and viewpoints can be developed later when detailed architectures are needed (Galster and Avgeriou 2011).

After deciding on views and methods of describing architecture, and by analyzing the limitations of current RAs and by in-depth studies of the BD systems, the construction of the RA took place.

5. Enable RA with variability: To allow for easier creation of concrete architectures from this RA, variability has been enabled for some modules. Based on the guidelines of Galster and Avgeriou ([22]), there are two ways to enable variability: 1) variability models and 2) annotation of the RA. We chose the latter.

6. Evaluation of the RA: Quality of the RA is determined based on two factors: 1) utility and correctness of the RA and 2) the support for instantiation and adoption of the RA [22]. To achieve these factors, we have created a prototype of the RA and tested it in practice to solve a business problem. Nevertheless, because the RA has not been built from scratch, and has absorbed patterns and principles from existing RAs architectures, and systems in practice, the focus of the evaluation was more towards sufficient support for effective instantiation. Additionally, we evaluated the prototype using architecture tradeoff analysis (ATAM [34]). This was to increase evaluation and adherence to'utility and correctness'.

IV. A MICROSERVICES EVENT-DRIVEN REFERENCE ARCHITECTURE FOR BIG DATA SYSTEMS

In the first iteration of the research methodology discussed in previous section, in step 3, we have captured architectural data and studied common themes in current BD systems. This has laid the foundation of Neomycelia and has illuminated us on the architectural requirements of BD systems. Thus, before the actual design and creation of the software RA, we have first listed the principal characteristics of BD systems and how they should be incorporated into the final artefact.

We have then introduced some of the well-known industrial patterns into the reference architecture and justified it. In what follows, we first discuss the principal characteristics of BD, we then discuss related works, microservices, event driven approaches, and justify each decision made. From there on, we will represent the software RA and describe the building blocks.

A. Big data system characteristics

BD has five principal characteristics: Volume, variety, velocity, veracity, and variability [14], [1]. Volume refers to the amount of data passing through various pipelines. In a traditional setup, a data analyst might typically import a fragment of a data warehouse into specialized software for statistical analysis [35] but this approach is often difficult to achieve with BD deployments. Velocity is the pace at which data flows into the system and gets processed. Commonly through data streams and, to effectively handle arrival irregularities, sliding windows [36]. Variety describes the heterogeneity of data formats. Data often arrives in various formats; structured, semi-structured, or pseudo-structured [37], [5].

Suitable architectural constructs must be created to address variety by, for example, normalizing or reformatting. Veracity refers to data quality, ensured by adherence to data governance protocols [38]. Data provenance, data quality assessment, data cleansing, and data liveliness are some of the architectural factors that must be taken into consideration. Variability takes into account the evolving nature of data and how data are ingested, processed, and conveyed to the next node in the pipeline [14]. Therefore, Neomycelia must provide:

- **Volume**: Capability of ingesting for massive data sets
- **Velocity**: Enable ingesting from multiple data sources at different rates (scale to demand). It shall support batch-mode and live-stream processing.
- **Variety**: Provide the means for rational processing across different data formats (structured, semi-structured, pseudo-structured)
- **Veracity**: Ensure data quality standards are maintained and includes data provenance, quality, liveliness, and cleansing
- **Variability**: Provide schema evaluation and effective interconnection

Taking the innate characteristics of BD into consideration, one should pay clear attention to quality attributes (Cloutier et al. 2010). Quality attributes help identify architectural

requirements and key drives. Additionally, quality attributes do play an important role in evaluation of the architectural prototype.

For the purposes of this software RA, the main quality attributes chosen are performance, availability and modifiability.

V. Related work

Most RAs and architectural patterns studied for this study have been designed with no attention to microservices and were mostly designed in the form of a monolith. For instance, Lambda ([39]) addresses speed, batches, and serving layers. The serving layer addresses the volume characteristics of BD by storing large amounts of heterogonous data in the master dataset.Velocity is addressed by having two different approaches to data processing, speed, and batch processing.

While the speed layer usually deals with streams of data that need to be processed in real-time, the batch layer forwards data to the serving layer, and then batch views to allow query processing. Bolster [14] augments this architecture by adding a semantic layer which supports metadata.

Meanwhile, the NIST BD RA [40] is the most comprehensive BD RA found in the SLR. The RA is delineated in terms of fabrics, providers, and services. Withall, we could not identify the notion of independent services or clearly defined contexts for the services. Along the lines, there is the pattern based approach conducted by Stricker et al. ([33]) that is heavily inspired by the works of Gamma et al ([41]).

A. From monolith to microservices

Recent approaches to software architecture tends towards increases in modularity and clearly defined boundaries [42]. The benefits derived of this shift sees organizations moving to design architectures as series of inter-communicating, yet independent services. Benefits include lower maintenance costs, increased ownership, more agile development, and better support for DevOps, versioning, and scaling [43]

Most legacy systems today run on some kind of monolithic architecture. As monolithic applications grow, complexity and overheads increase, making maintenance a daunting task. Adopting new technologies in monolithic systems is difficult and if there exist design changes, the development team may need to put an immense amount of effort to meet new requirements.

In addition, many monolithic systems are not integrated with recent cloud technologies and DevOps trends, for example introducing change or modification to an existing module is prone to side-effects and hard-to-find bugs [44], [43]. Teams are not hundred percent autonomous and holocracy, if it has been adopted, is compromised.

These factors can hinder the development of complex systems such as big data systems, and create architectural risks when it comes to tradeoff points, sensititivity points, and quality attributes such as modifiability. Therefore, Neomycelia aims to move away from monolithic designs and is founded based on event-driven microservices.

Whereas the concepts aformentioned can be beneficial to any system and not only big data systems, they can address some of the major challenges of adopting big data systems such as horizontal scaling, data quality and provenance, and throughput [45].

B. The microservices challenge

Microservices architecture offers a promising solution to issues related to monolithic systems, however the implementation of it can be challenging. Sometimes referred to as a consequence of 'monolithic microservices', one of the key challenges of a successful microservices architecture is effective communication between services [46].

The basic approach to asynchronous communication between services, is through REST API calls, where a service makes a REST call to another service. This is referred to as point-to-point communication and is useful for cases where no response is required or when an admin task needs to be executed.

But as services grow, the need for inter-communication increases and if one service is in a blocking state (for example, running a time consuming process), it will affect other services that rely on that service, and thus a ripple effect through the whole system is created. This is a form of coupling and has sometimes been reffered to as 'monolithic microservices'.

Another approach to interservice asynchronous messaging in a microservices architecture is the publisher-subscriber (pub-sub) model. In this model, a central pub-sub construct (usually a message broker) sits at the heart of the architecture and facilitates communications. This means that, instead of microservices directly calling each other, they publish a new message to the message broker on a specific topic and all the subscribers to that topic will be notified.

Any computational effort is then handled by the subscribers. Furthermore, a subscriber can be a publisher of message too [47]. In this case, the message broker is at the heart of the communications process. Whereas this approach is better than the former, it still can introduce coupling and affect quality attributes such as modifiability and performance.

Hence, the ways in which services register, publish, subscribe and communicate is still is at risk.

C. An event driven approach

The approaches to microservices communication described require some sort of coupling. Thus, interfaces need to be declared, services need to know what other services are responsible for, how messages will be processed, and services may have dependencies related to other microservices. As microservices increase in number, the maintenance of microservice coupling becomes increasingly challenging and this affect the overall quality of the architecture.

However, we assert that the concerns above are not the aim of a microservices architecture. Instead, an event driven model provides an effective means of addressing those issues. Point-to-point or pub-sub communication between microservices

means that each service should recognise related interfaces and the requirements for downstream microservices [48], [49].

The underlying mechanism of existing models is to issue a 'command', where one microservice sends commands to another microservice or topic.We provide a 'dispatch and forget' approach, in which a service dispatches an event to a central event backbone [50].

In this way, services do not need to know what happens after an event has been dispatched, they are only responsible for event dispatching through well-defined contracts. The underlying mechanism of this model is 'event'. Thus, new services can be added, removed, configured, and scaled easily and more simply.

The subtlety lies in the underlying approach, and philosophy of 'event' instead of 'command', which implies that modules react to a change of state rather than a command for action from another module. This approach solves the issue of long-running tasks that block and a service does not need to wait for another to complete its process.

VI. Neomycelia

In this section, Neomycelia is described. Strong points for Neomycelia are its attention to metadata and data caching. With the exception of Bolster [14], RAs scarcely include metadata. Metadata addresses a wide range of needs like security, privacy, scalability, and efficiency. In the context of a distributed system, metadata bridges data stored in different functions such as in the Cloud versus that on premise [51].

Furthermore, there was no evidence of an RA that provides for data caching. Avoiding issues related to monolithic systems, and by adhering to principles of event-driven communications, the Neomycelia RA (1) provides improved modularity, extendibility, modifiability, scalability, and maintainability. This is in line with quality attributes that has been set for evaluation purposes.

A. Components

Neomycelia is comprised of 14 essential architectural components. Our aim is to provide a high level overview of these services, and detailed explanation of each service is outside the scope of this study. The terms node, and service are used interchangeably.

Gateway: This is the main entry point and determines how messages are handled. Gateway acts as a reverse proxy that accepts all incoming requests from Application Programming Interfaces (API) and appoints them to appropriate services. This mechanism provides the performance and modifiability aspects of the RA. In addition, gateway can bring other benefits such as increased security, logging, and load balancing.

Stream processing controller: This service forwards requests to the stream processing service mesh. Low impact processes like basic sanitization, authentication, and potential validation take place here. Computationally expensive events are not processed in this node. The stream processing controller provides stream provenance and may decide which one-pass algorithm should run over what event stream. This services serves as the initial invocation.

Batch processing controller: The function of this service is to forward requests to the batch processing service mesh. Low impact pre-processing such as preliminary normalization can take please in this node. It is necessary to avoid computationally expensive processes in this service. This service serves as the initial invocation.

Stream processing service mesh: Based on the requirements for specific stream processing approaches, this service is comprised of an arbitrary number of nodes for stream processing. A cache manager keeps track of values processes in the event queue and keeps an event archive with a list of associated computed values. Each service has its own local database. Most stream processing algorithms are operating on in-memory stateful data structures, such as HyperLogLog, to compute distinct values and provide summary output. Different services can adopt unique processing and windowing approaches such as micro-batch processing and tuple-at-a-time [52].

To avoid obfuscation of the event backbone, each service in the service mesh will communicate its results through an internal message broker (an event queue) that is responsible for communication to the external entities. Linking service results to a message broker that communicate with the event backbone provides parallelism, encapsulates stream processing to its own service mesh, and provides abstraction.

Any service may have a sidecar, or service proxy, attached to it. These proxies act as an intermediary to encapsulate networking and infrastructure needs of the service. Sidecars abstract out all the platform requirements necessary for the service to run. This allows software engineers to focus on what matters, development, and avoid being distracted by infrastructure issues.

Finally, every service mesh will have a control tower which communicates with sidecars and acts as a centralized unit that registers, updates, and passes changes to the services through their proxy. For example, if all services in this service mesh need to update their SSL certificate monthly, they can do it through sidecar communication to the control tower.

Batch processing service mesh: Similar to the stream processing service mesh, except that this does not require a message broker (event queue). Batch processes do not need to happen in real-time, so there is no advantage in providing parallelism. Every service can dispatch directly to the event backbone.

Service discovery: As the number of services increases and deployment and configuration demands become more frequent, identification and invocation of appropriate services become daunting. Service discovery addresses this problem by being a central register for all services. This means, services can now utilize a central node to register their functionalities.

Monitoring: A challenge for a microservices architecture is the ability to track, monitor, and log services behaviors. An event is dispatched from a service, which invokes another service, which in turn results in another dispatch. It's a difficult

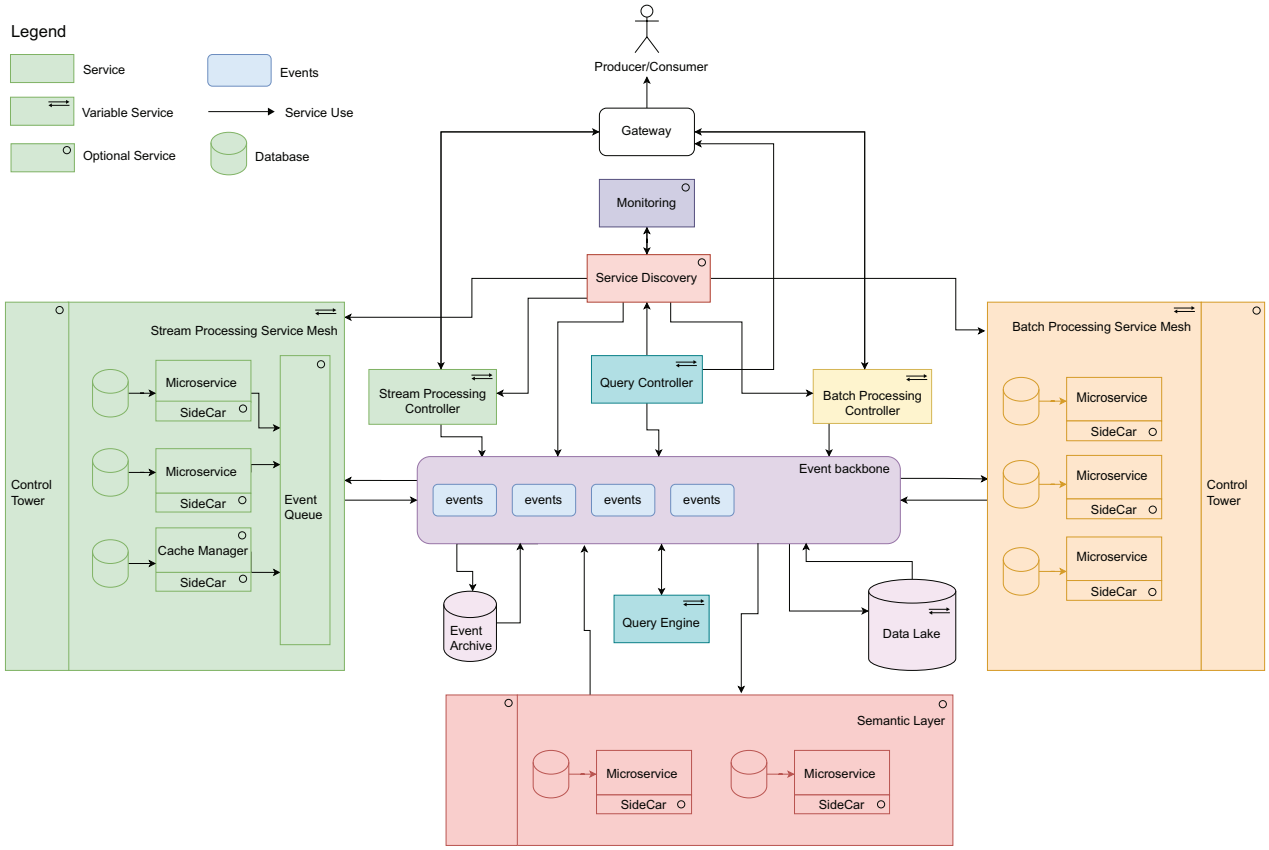

Fig. 1. Neomycelia Software Reference Architecture.

task to identify errors in this chain with loosely coupled services but the monitoring service solves this problem by tracking usage and behavioral data from services. Data collection is provided by the service discovery unit. The architect and system designer decide on the granularity of data collected from each service and log aggregation mechanisms.

Event backbone: This is the heart of Neomycelia, facilitating communication between the parts of the system. Every service in the system communicates through event backbone as choreographed events, analogous to a dance troupe where everyone responds to the rhythm of the music and moves according to their specific role. Each service (dancer) listens to the event backbone (the music) and takes action only as required. Thus, a service is responsible only for dispatching an event in a 'dispatch and forget' model. Services listening on a specific topic execute the desired process and may dispatch another event to the event backbone.

The event backbone is the sole controller of communication between services, through events. This allows development teams to focus on various services and develop independently. This also allows for simple plug-in and plug-out of a service, reducing concerns about side-effects. Moreover, a failure in a microservice is less likely to affect the whole system because the loss of availability is periodically checked and prevented.

Event archive: Communications between services can sometimes be faulty. Services can timeout, there could be issues in the message contract, the existence of bugs, or type problems. One way to achieve an effective failure handling and recovery approach is through an event archive. This means, if an event fails, it can be retried and recovered because it was registered in an event archive. Furthermore, in the case of an event backbone service going down during an event transmission, the event backbone can recover itself by reading the events it had to handle from the event archive. This can be series of events or a single event.

Data Lake: This contains structured, pseudo-structured, unstructured, and semi-structured data. In a generic setup, data is usually stored in the data lake before it is stream or batch processed. This may not provide the best-case scenario for all situations, so it is up to the architect to decide if that is the most appropriate data flow in the system.

Semantic Layer: This service mesh is the central hub for all metadata processing, containing the MetaData Management (MDM) system. The MDM is responsible for providing services with information to define and model raw data. This is where a domain vocabulary is defined and controlled by data stewards. The semantic layer provides the opportunity to store metadata, preparation rules, and data evolution and reduces the need for the data analyst or scientist to repeat work.

Query Controller: This service determines the type of query received (stream, batch, or other) that come from the gateway and dispatches the event to the query engine. The

controller performs straight forward pre-processing or sanitization as required but heavy computation should be avoided.

Query Engine: This service processes query requests. Once values are created, an event is dispatched the subsequent service to return results.

B. Variability

Neomycelia components are classified as required, variable, and optional. This RA is governed by a strict decoupling rule, such that the removal of any service shall not affect other services.

Gateway, event backbone service, microservices and event archive are required if the architect decides to adhere to the decoupling rule. However, an architect can decide to utilize another method of communication like orchestrated events or point-to-point protocol.The required service gateway provides load balancing, security, lower cumulative latencies, and code simplification. The required service event backbone provides communication between services. The required service event archive is a retry/fail mechanism for the event backbone.

Variable services provided that a context is maintainable, in which the data lake, stream processing service mesh, batch processing service mesh, controller services (batch, stream, query) and query engine are variable and can be substituted to meet contextual demands.

Optional services include sidecars, control towers, event queue, cache manager services, service discovery service, monitoring service, semantic layer service mesh, and policy manager service mesh. A semantic service mesh and policy manager service mesh provides a solution for privacy and metadata issues in BD systems. An architect may implement a semantic service mesh but must determine if the company cope with the complexity.

VII. Evaluation

RAs lack clearly defined stakeholders, exist at a high level of abstraction, and are highly adaptable [30].RA evaluation is a challenge [24], [53], [54], [55] because they are not the same as a typical architecture. Based on the guidelines provided by Galster and Avgeriou ([22]), utility, correctness and instantiation support determine the quality of the RA, and therefore an effective evaluation is critical. To achieve that, we have first deduced a prototype architecture and then followed the guideline of architecture tradeoff analysis method (ATAM [34]) to evaluate the prototype in practice.

This is achieved by evaluating the correctness and utility of the RA, and how efficiently adaptation can be instantiated [22]. Quality is assessed by how the RA can be transformed into an effective organization-specific concrete architecture and through qualitative and quantitative methods discussed in ATAM. This evaluation follows two phases. Phase 1 includes gathering basic information about the company, practice, and interaction with key stakeholders.

Phase 2 includes analysis and evaluation of the proposed architecture, understanding quality attributes, tradeoff points, sensitive points and risks. We do not describe every step of the ATAM for brevity purposes. Additionally, some of the details have been changed for academic purposes and to protect the intellectual property of the practice. Nevertheless, the evaluation results are not materially affected.

A. Phase 1

Evaluation is undertaken in an Auckland-based company and being applied to new and existing workflows. The company provides practice management software to veterinary professionals via a Software as a Service (SaaS). The company serves over 15,000 clients from New Zealand, Australia, USA, UK, and Canada with some large equine hospitals and veterinary clinics. The company aims to embark on big data and machine learning endeavour, and seize abundance of business opportunities that exist.

During the initial meeting, we first presented the ATAM and Neomycelia, and the business stakeholders presented the business case. We have then scoped down the business case to one workflow that NeoMycelia can account for. We learnt that countries adhere to local regulatory frameworks for drug prescriptions for animals, which means local jurisdictions control access to veterinary drugs as Restricted Veterinary Medicines (RVM). Under this protocol, all patients should be background-checked before a drug is dispensed. Therefore, real-time stream processing for background-checks and batch processing for billing triggers and reports generation is needed. In the future, the system may provide breed detection via image processing of animal photos.

We have used this information to create a prototype from Neomycelina, quality attributes utility tree, to analyze our architectural approach

B. Phase 2

In phase 2, We have first created an archetype of Neomycelia. We utilised ISO/IEC 25000 SQuaRE standard (Software Product Quality Requirements and Evaluation) ([56]) as a reference quality model for technology selection. This model is described in terms of characteristics and sub-characteristics. The explanation of these characteristics is out of the scope of this paper.

This archetype incorporates Kafka as the event backbone, AWS Lambda for the stream processing and batch processing controllers, Amazon load balancer as the gateway (application load balancer), Envoy for the sidecar services, Kubernetes as cluster manager, Docker for container technology, Resilience4J for a fault-tolerant communication module, Prometheus for the monitoring tool, Zookeeper for the service discovery tool, Istio as the service mesh control tower, Amazon RDS for services private database, Amazon S3 for the event archive, Amazon MQ as the message broker inside the mesh, an Amazon data lake, and Amazon EC2 for services. To account for better availability we have additionally setup autoscaling groups targeting two different target groups in two different zones.

We have then presented the architecture to the stakeholders, identified architectural approaches, elicited quality attribute trees, and prioritized scenarios.

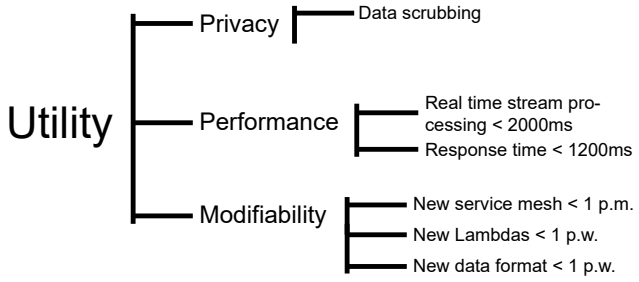

Fig. 2. Utility Tree

1) Identifying architectural approaches: To identify architectural approaches, we elicited information from the archetype with respect to modifiability, availability, and performance.

- for performance, application load balancer, lambdas and service meshes have been described
- for availability, we discussed scaling groups, and event archive
- for modifiability, we discussed the 'zero coupling' philosophy of Neomycelia and decentralized nature of it. We discussed th event driven nature of it as well.

We probed each quality attribute for tradeoffs, sensitivity points, and risks.

2) Scenario Prioritisation: Scenarios allow for the capturing of stimuli to which the architecture has to react. These are captured to identify system's ability to meet functional and non-functional requirements. In this step, we asked stakeholders to come up with 3 different kind of scenarios namely use-case scenarios (typical use of the system), growth scenario (anticipated changes), and exploratory scenarios (extreme changes). We have then gave each stakeholder time to vote for each scenario. The result of this created over 30 scenarios, out of which 5 has been selected.

We describe those 5 scenarios as two journeys;

- User needs to get restricted medication for his/her pet, but he/she has to be background checked before the medication can be dispensed.
- A large database of animals have been given to the system to predict certain disease.

3) Ulitity tree elicitation: In this step, we learnt from the stakeholders that privacy, modifiability and performance are the most important quality attributes, and we adjusted to it. We have also learnt that whereas availability is of concern, it still fall in a lower place than privacy. Based upon this, utility tree has been generated (2).

4) Analyze architectural approaches: At this stage, we mapped our architectural approaches against a simulation scenario that matches businesses' case study. This approach was used to provide for heuristic qualitative analysis, and to understand risks, tradeoffs and sensitivity points.

A simulation scenario with a request that emulates the system in production has been formulated. We created relevant topic in the event backbone, and sent a mock API request with a mock data to test the system's response. Our mock data has been duplicated from real data.

We engineered API gateway to pass the request to the representative Lambda. The Lambda performed straightforward preprocessing (adding properties or mutating JSON objects) on the data and dispatched an event to Kafka.

We raised several screening questions to better probe our approach and the business case, some of these questions were:

- How do we recover from a failed event?
- What happens if event backbone goes out of service?
- How events are process? in what order?
- If we need to remove sensitive data, how do we achieve that?
- How easily can be extend and modify our services?

It is important to note that the Neomycelia itself as delinated in the figure 1 does not entail all the details to understand the system. Thus, scenarios, prototypes, and quality attribute based analysis and screening questions can help illuminate the architectural view, tradeoff and sensitivity points. [34]

Sensitivty points for quality attributes are points that if changed, can drastically affect the behavior of the system. Therefore, one should pay close attention to these points as the pose the highest risk, and can affect the system's evolvability.

Tradeoff points are points at which one service can affect various quality attributes, thus the architect should decide if the tradeoff is worth the implementation.

There are various approaches for building models of quality attrbiutes, such as rate monotonic analysis for real-time systems ([57]), Markov models to understand reliability ([58]), and SAAM for modifiability ([59]). Whereas some of these models are quite simple, they illuminate hidden risks that may not be very obvious from the beginning. For the purposes of this evaluation we do aim for a thorough and detailed analysis.

Hence, based on stakeholders feedback and utility tree, we can state that system quality Q_S, is a function f of the quality attribute performance Q_P, modifiability Q_M and privacy Q_{Pr}:

$$Q_S = f(Q_M, Q_A, Q_P) \tag{1}$$

- **Privacy**: We learnt that our prototype does not account for 'privacy' quality attribute. Thus we added a new service mesh called 'policy manager'. This service mesh is responsible for applying or checking policies about input data. The policy manager aligns policies with various contexts to determine how data should be retrieved or processed. This is achieved without significant coupling to the system. The main responsibility of this service mesh is to check if the input data is in line with business predefined policies. It then flags and dispatches subsequent event depending on the nature of the request. This has addressed stakeholders concern for geopolitical laws pertaining to data privacy. This has also addressed

an architectural risk as we have not forseen the need for such service.

- **Modifiability**: for modifiability analysis, we followed SAAM ([59]) and expanded on it to justify the utility tree generated. The decoupled and distributed nature of Neomycelia allowed us to easily account for utility tree generated. The fact that we could 'plug-in' and 'plug-out' services, service meshes and lambdas have proven to account for great level of modifiability. This has also helped in addressing other quality attributes such as privacy. We added a privacy service mesh and measured the time and effort it required. In our simple scenario, we have created a manifest file out of existing ones for our Kubernetes cluser, and deployed a new one, but this time with different labels pertaining the privacy manager service mesh. Depending on the complexity of the system, this may way, but our result prove that we can account for 1 p.m. that has been discussed in the utility tree.
- **Performance**: we implemented a simple performance test by utlising a clous stress testing agent. The agent allowed us to simply execute an on-demand stress on the system. It became clear to us that Lambdas time to boot up (cold start) can affect our system's response time. We have captured an average of 250ms up to 500ms coldstart time for each Lambda to spin up and handle the request. Whereas replace Lambdas with an EC2 or a Fargate on AWS could solve the issue, we did not find that necessary, as even with the coldstart time, the system still could account for the response time required. Furthermore, we have realised that one solution to Lambda coldstart problems could be predictable start-up times with privisioned concurrency. However this is outside the scope of this study. We have also measured throughput between services. We encapsulated our service meshes in private networks, thus the communication were fast and our response time were under 1200ms.

5) Tradeoff points: As a result of these analysis, we identified two tradeoff points. One of which is the controllers and more specifically the Lambdas. We can refine this tradeoff point as:

$$Q_{\mathrm{P}} = g(\lambda_{\mathrm{C}}, \mu_{\mathrm{C}}) \tag{2}$$

That is, system performance is affected by the failure rate of the controllers to address and dispatch requests in a timely manner, implying λ_{C} is being the failure rate of the controller, and μ_{C} being the time it takes for the controller to spin up and address demand. Whereas there are solutions to this issue and some cloud providers have addressed this problem (Cloudflare workers), we have not taken that into consideration and our evaluation is based on AWS services.

Therefore, we determine this as a risk. This is due to the fact that practitioners may need to do a background check up on the individual before dispensing a medicine, and this should be stream processed in a timely manner. On the other hand, substituting a Lambda for an EC2, would decrease the modifiability of the system, as EC2 is a complete virtual machine that needs to be maintained. This also means, the concept of 'controller' loses it's context, as we have perceived 'controller' as a very small traffic forwarding piece of our architecture (like a traffic light).

Another aspect that challenged Neomycelia was the criticality of the 'event backbone'. Being at the core of the architecture, we've seen worries around how it would recover in the case of failure and how does it scale to account for all the communications necessary. Kafka recovery is achieved through event archive that keeps track of all the events in a separate storage. This allows for recovery from abrupt failures. We have also realized that kafka can scale and effectively address the demand during our stress test.

Therefore, we deduced that Kafka can account for the performance quality attribute. However, different stream processing platforms come with various schema registries that are sometimes need modification and do not adhere to the same interface. This means modifiability is affected, and it would be very difficult for a business to move to a new stream processing technology if desired. We have realized that the distributed nature of Kafka is really in line with our performance and modifiability goals. But this also limits the number of choices an architect has, and therefore modifiability can be compromised.

6) Limitations: Whereas many of the developers and the architect itself like the choreographed nature of our event-driven system and it's decoupled nature, Neomycelia has been perceived complicated to implement. practitioners perceived the prototype as requiring a lot of expertise in platform, specifically in Kubernetes cluster management and deployment. Furthermore, some stakeholder found internal message queue of the service mesh unnecessary in the initial phase.

VIII. Conclusion

As data-oriented technologies emerge, long-established ideologies toward the nature of system design and architecture are revolutionized. Companies are required to be prepared to tackle the apparent difficulty of this field and absorb the complexity. RAs provide a starting point to manage the complexity of BD. In the case of ambiguity towards what should be developed against need, architectures play an overarching role in describing the building blocks of the system and the ways in which blocks communicate to achieve the goals of the system. Future work on Neomycelia and other RAs address security concerns and creating more complex and distributed metadata management technologies.

References

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

[2] M. Shafi, A. F. Molisch, P. J. Smith, T. Haustein, P. Zhu, P. De Silva, F. Tufvesson, A. Benjebbour, and G. Wunder, "5g: A tutorial overview of standards, trials, challenges, deployment, and practice," *IEEE journal on selected areas in communications*, vol. 35, no. 6, pp. 1201–1221, 2017.

[3] M. Lycett, "'datafication': Making sense of (big) data in a complex world," 2013.

[4] N. Heudecker and L. Kart, "Survey analysis: Big data investment grows but deployments remain scarce in 2014," *Stamford: Gartner*, 2014.

[5] 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.

[6] H.-M. Chen, R. Kazman, J. Garbajosa, and E. Gonzalez, "Big data value engineering for business model innovation," 2017.

[7] M. Huberty, "Awaiting the second big data revolution: from digital noise to value creation," *Journal of Industry, Competition and Trade*, vol. 15, no. 1, pp. 35–47, 2015.

[8] J. Ranjan, "The 10 vs of big data framework in the context of 5 industry verticals," *Productivity*, vol. 59, no. 4, 2019.

[9] N. Partners, "Big data and ai executive survey 2019," *Data and Innovation*, 2019.

[10] M. Analytics, "The age of analytics: competing in a data-driven world," Technical report, San Francisco: McKinsey & Company, Tech. Rep., 2016.

[11] H. Nash, "Cio survey 2015," *Association with KPMG*, 2015.

[12] N. Singh, K.-H. Lai, M. Vejvar, and T. Cheng, "Big data technology: Challenges, prospects and realities," *IEEE Engineering Management Review*, 2019.

[13] I. Gorton and J. Klein, "Distribution, data, deployment," *STC 2015*, p. 78, 2015.

[14] 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.

[15] O. Sievi-Korte, I. Richardson, and S. Beecham, "Software architecture design in global software development: An empirical study," *Journal of Systems and Software*, vol. 158, p. 110400, 2019.

[16] F. Mannering, C. R. Bhat, V. Shankar, and M. Abdel-Aty, "Big data, traditional data and the tradeoffs between prediction and causality in highway-safety analysis," *Analytic methods in accident research*, vol. 25, p. 100113, 2020.

[17] S. A. M. Selamat, S. Prakoonwit, R. Sahandi, and W. Khan, *Big Data and IoT Opportunities for Small and Medium-Sized Enterprises (SMEs)*. IGI Global, 2019, pp. 77–88.

[18] J. Kohler and T. Specht, *Towards a Secure, Distributed, and Reliable Cloud-Based Reference Architecture for Big Data in Smart Cities*. IGI Global, 2019, pp. 38–70.

[19] R. Cloutier, G. Muller, D. Verma, R. Nilchiani, E. Hole, and M. Bone, "The concept of reference architectures," *Systems Engineering*, vol. 13, no. 1, pp. 14–27, 2010.

[20] M. Derras, L. Deruelle, J.-M. Douin, N. Levy, F. Losavio, Y. Pollet, and V. Reiner, "Reference architecture design: A practical approach," in *ICSOFT*, Conference Proceedings, pp. 633–640.

[21] J. Klein, R. Buglak, D. Blockow, T. Wuttke, and B. Cooper, "A reference architecture for big data systems in the national security domain," in *2016 IEEE/ACM 2nd International Workshop on Big Data Software Engineering (BIGDSE)*. IEEE, Conference Proceedings, pp. 51–57.

[22] M. GALSTER and P. AVGERIOU, "Empirically-grounded reference architectures," *Joint ACM*.

[23] S. Angelov, P. Grefen, and D. Greefhorst, "A classification of software reference architectures: Analyzing their success and effectiveness," in *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE, 2009, pp. 141–150.

[24] S. Angelov and P. Grefen, "An e-contracting reference architecture," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1816–1844, 2008.

[25] H. Wu, *A reference architecture for Adaptive Hypermedia Applications*. Citeseer, 2002.

[26] A. Norta, P. Grefen, and N. C. Narendra, "A reference architecture for managing dynamic inter-organizational business processes," *Data & Knowledge Engineering*, vol. 91, pp. 52–89, 2014.

[27] E. Y. Nakagawa, R. M. Martins, K. R. Felizardo, and J. C. Maldonado, "Towards a process to design aspect-oriented reference architectures," in *XXXV Latin American Informatics Conference (CLEI) 2009*, 2009, Conference Proceedings.

[28] L. Shamseer, D. Moher, M. Clarke, D. Ghersi, A. Liberati, M. Petticrew, P. Shekelle, and L. A. Stewart, "Preferred reporting items for systematic review and meta-analysis protocols (prisma-p) 2015: elaboration and explanation," *Bmj*, vol. 349, 2015.

[29] 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.

[30] P. Ataei and A. T. Litchfield, "Big data reference architectures, a systematic literature review," 2020.

[31] G. Beneken, "Referenzarchitekturen," 2018.

[32] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, and U. Zdun, "Software-architektur: Grundlagen–konzepte," *Praxis*, vol. 2, 2009.

[33] V. Stricker, K. Lauenroth, P. Corte, F. Gittler, S. De Panfilis, and K. Pohl, "Creating a reference architecture for service-based systems-a pattern-based approach," in *Future Internet Assembly*, Conference Proceedings, pp. 149–160.

[34] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193)*. IEEE, Conference Proceedings, pp. 68–78.

[35] C. Ordonez, "Statistical model computation with udfs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 12, pp. 1752–1765, 2010.

[36] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, Conference Proceedings, pp. 1–16.

[37] I. G. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino, "Data wrangling: The challenging yourney from the wild to the lake," in *CIDR*, Conference Proceedings.

[38] D. Plotkin, *Data stewardship: An actionable guide to effective data management and data governance*. Academic Press, 2020.

[39] M. Villari, A. Celesti, M. Fazio, and A. Puliafito, "Alljoyn lambda: An architecture for the management of smart environments in iot," in *2014 International Conference on Smart Computing Workshops*. IEEE, Conference Proceedings, pp. 9–14.

[40] W. L. Chang and D. Boyd, "Nist big data interoperability framework: Volume 6, big data reference architecture," Report, 2018.

[41] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[42] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[43] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer, Conference Proceedings, pp. 128–141.

[44] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[45] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, "Critical analysis of big data challenges and analytical methods," *Journal of Business Research*, vol. 70, pp. 263–286, 2017.

[46] R. M. Munaf, J. Ahmed, F. Khakwani, and T. Rana, "Microservices architecture: Challenges and proposed conceptual design," in *2019 International Conference on Communication Technologies (ComTech)*. IEEE, Conference Proceedings, pp. 82–87.

[47] K. Indrasiri and P. Siriwardena, "Microservices for the enterprise," *Apress, Berkeley*, 2018.

[48] P. Gupta, T. P. Mokal, D. Shah, and K. Satyanarayana, "Event-driven soa-based iot architecture," in *International Conference on Intelligent Computing and Applications*. Springer, Conference Proceedings, pp. 247–258.

[49] J. M. Molina, J. F. García, and C. K. Jiménez, "Archer: An event-driven architecture for cyber-physical systems," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, Conference Proceedings, pp. 335–340.

[50] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, no. 12, pp. 10–1571, 2006.

[51] R. K. Eichler, "Metadata management in the data lake architecture," Thesis, 2019.

[52] R. Sahal, J. G. Breslin, and M. I. Ali, "Big data and stream processing platforms for industry 4.0 requirements mapping for a predictive maintenance use case," *Journal of manufacturing systems*, vol. 54, pp. 138–151, 2020.

[53] P. Avgeriou, "Describing, instantiating and evaluating a reference architecture: A case study," *Enterprise Architecture Journal*, vol. 342, pp. 1–24, 2003.

[54] E. Cioroaica, S. Chren, B. Buhnova, T. Kuhn, and D. Dimitrov, "Towards creation of a reference architecture for trust-based digital ecosystems," in *Proceedings of the 13th European Conference on Software Architecture-Volume 2*, Conference Proceedings, pp. 273–276.

[55] M. Maier, A. Serebrenik, and I. Vanderfeesten, "Towards a big data reference architecture," *University of Eindhoven*, 2013.

[56] I. Iso, "Iec25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models," *International Organization for Standardization*, vol. 34, p. 2910, 2011.

[57] M. H. Klein, T. Ralya, B. Pollak, R. Obebza, and M. Harbour, "Guide to rate monotonic analysis for real-time systems," 1993.

[58] A. Iannino, "Software reliability theory," *Encyclopedia of Software Engineering*, pp. 1237–1253, 1994.

[59] R. Kazman, L. Bass, G. Abowd, and M. Webb, "Saam: A method for analyzing the properties of software architectures," in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 81–90.