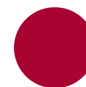
UNIVERSITY OF COPENHAGEN



From a Monolithic Big Data System to a Microservices Event-Driven Architecture

Nunes Laigner, Rodrigo; Kalinowski, Marcos; Diniz, Pedro; Barros, Leonardo; Cassino, Carlos; Lemos, Melissa; Arruda, Darlan; Lifschitz, Sérgio; Zhou, Yongluan

Published in:
Proceedings of 46th Euromicro Conference on Software Engineering and Advanced Applications

Publication date:
2020

Citation for published version (APA):
Nunes Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., Arruda, D., Lifschitz, S., & Zhou, Y. (2020). From a Monolithic Big Data System to a Microservices Event-Driven Architecture. In *Proceedings of 46th Euromicro Conference on Software Engineering and Advanced Applications* (pp. 213-220). [9226286] IEEE.

Download date: 25. sep.. 2021

From a Monolithic Big Data System to a Microservices Event-Driven Architecture

Rodrigo Laigner
Department of Computer Science (DIKU)
University of Copenhagen, Denmark
rnl@di.ku.dk

Marcos Kalinowski, Pedro Diniz
Informatics Department
PUC-Rio, Brazil
{kalinowski,pfonseca}@inf.puc-rio.br

Leonardo Barros, Carlos Cassino, Melissa Lemos
Tecgraf/PUC-Rio, Brazil
{lbarros,cassino,melissa}@tecgraf.puc-rio.br

Darlan Arruda
Department of Computer Science
Western University, Canada
darruda3@uwo.ca

Sérgio Lifschitz
Informatics Department
PUC-Rio, Brazil
sergio@inf.puc-rio.br

Yongluan Zhou
Department of Computer Science (DIKU)
University of Copenhagen, Denmark
zhou@di.ku.dk

***Abstract*—[Context] Data-intensive systems, a.k.a. big data systems (BDS), are software systems that handle a large volume of data in the presence of performance quality attributes, such as scalability and availability. Before the advent of big data management systems (e.g. Cassandra) and frameworks (e.g. Spark), organizations had to cope with large data volumes with custom-tailored solutions. In particular, a decade ago, Tecgraf/PUC-Rio developed a system to monitor truck fleet in real-time and proactively detect events from the positioning data received. Over the years, the system evolved into a complex and large obsolescent code base involving a costly maintenance process. [Goal] We report our experience on replacing a legacy BDS with a microservice-based event-driven system. [Method] We applied action research, investigating the reasons that motivate the adoption of a microservice-based event-driven architecture, intervening to define the new architecture, and documenting the challenges and lessons learned. [Results] We perceived that the resulting architecture enabled easier maintenance and fault-isolation. However, the myriad of technologies and the complex data flow were perceived as drawbacks. Based on the challenges faced, we highlight opportunities to improve the design of big data reactive systems. [Conclusions] We believe that our experience provides helpful takeaways for practitioners modernizing systems with data-intensive requirements.**

***Index Terms*—big data system, microservices, event-driven**

I. Introduction

Data has been generated at an increasingly higher pace over the last years. Social media interactions, sensors, mobile phones, and business processes are examples of sources. Surveys indicate that 2.5 quintillion bytes of data are generated each day, which will lead to approximately 79.4 zettabytes of data by 2025 [1], [2]. This context made the case for the design of big data systems (BDS), which arose to handle the collection and manipulation of large volumes of data in modern business applications.

Gorton and Klein [3] define BDS as "distributed systems that include redundant processing nodes, replicated storage, and frequently execute on a shared cloud infrastructure [...] employing a heterogeneous mix of SQL, NoSQL, and NewSQL technologies." As a result, the development of BDS often imposes challenges to software engineers, as noted by Hummel et al. [4], which cataloged a set of challenges, such as steep learning curve and complex data processing. Besides, Laigner et al. [5] found that the major challenges on developing BDS are about software architecture design.

Event-driven systems and microservices have emerged as compelling architectural paradigms to the development of data-driven software applications [6], [7]. Microservices are small scalable units where each represent a bounded business capability that are often autonomously deployed. In contrast to traditional monolithic systems, microservices do not share resources, communicating mainly via message-passing semantics [8]. In line with microservices, an event-driven architecture (EDA) is comprised by a set of high-cohesive components that asynchronously react to events to perform a specific task [9].

In this paper, we report the complete replacement process of a legacy BDS to a microservice-based event-driven architecture. The replacement comprised a 19-month long development period that took place at PUC-Rio's Tecgraf Institute, which provides technical and scientific solutions for a wide range of strategic industrial partners. One of the solutions, developed for a customer in the Oil & Gas sector back in 2008, concerns a monolithic BDS that monitors moving objects (MOs) and proactively detects events that incur in risks to the operation, such as vehicle route deviations. Over the years, the system evolved into a complex and large obsolescent code base that involves a difficult maintenance process. In this context, in 2018, with the advent of a new industrial partner interested in the outcomes of the previous project that employed the legacy BDS, Tecgraf's managers decided to take advantage of a new contract to accommodate a complete rewrite of the legacy BDS by adopting current big data technologies, such as Cassandra and Kafka. Furthermore, based on the lessons learned of the legacy BDS, Tecgraf's managers decided that the new project must adopt a microservice-based EDA.

Thus, we investigate the integration of microservices and EDA to support data-intensive requirements. The main contributions of this paper are: (i) an investigation on the motivation to adopt a microservice-based EDA; (ii) a 19-month experi-

ence report on replacing a legacy BDS with a microservice-based EDA; (iii) a discussion of the obtained results in form of challenges and lessons learned.

The remainder of this paper is organized as follows. Section II provides the background of this work. Next, the action research design is presented, describing the goal, research questions, and methodology. The results are presented in Section IV. Lastly, Section V presents the concluding remarks.

II. Background

A. Big data systems

Chen et al. [10] explain that traditional software development is characterized by "structured, batch-oriented, relational small data [volume]," and straightforward development lifecycle and architecture design. Besides, Gorton and Klein [3] argue that traditional business systems are "relatively well constrained in terms of data growth, analytics, and scale." On the other side, Gorton and Klein [3] synthesize BDS based on four requirements: (i) write-heavy workload; (ii) variable request loads (adding new resources and release them as necessary); (iii) computation-intensive analytics (diverse query workloads and varying latency demands); and (iv) high availability. These requirements represent a significant shift from traditional business systems.

B. Microservices

Software systems have traditionally adopted a monolithic architectural style, on which modules and/or subsystems are integrated and cooperate in a centralized manner. According to Bucchiarone et al. [8], in such architecture, "the modularization abstractions rely on the sharing of resources of the same machine [...], and the components are therefore not independently executable." However, concerns related to the complexity involved on scaling monolithic architectures [8] and aspects related to change, such as evolutionary maintenance [11], have shifted interests in industry towards the adoption of decoupled architectures. Built on SOA principles of loosed-coupled services, microservices have emerged as an "organic implementation approach to SOA[, encompassing] polyglot programming in multiple paradigms and languages, and design for failure; decentralization and automation" [12].

C. Event-driven architecture

Systems that adopt an EDA, also known as reactive systems, are a current subject of interest in the development of data-driven software systems [6]. According to Richards [9], EDA is a pattern "made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events". Richards argues that "event processors are self-contained, independent, highly decoupled architecture components that perform a specific task in the application or system" [9]. Therefore, in EDA, each single-purpose service employs programming primitives for enabling reaction and response to a set of predefined events. To the best of our knowledge, literature does not clearly differentiates these from microservices.

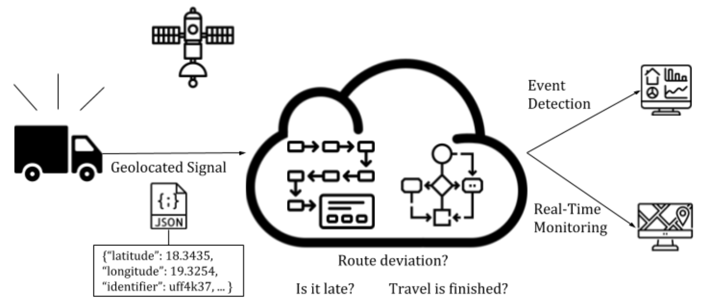


Fig. 1. Legacy big data system overview

III. Research Design

Our study design follows the Action Research (AR) [13] methodology. The study context, goal and research questions, and methodology are presented hereafter.

A. Context

This study reports on the process of replacing a legacy big data system with a microservice-based EDA. The experience described herein occurred in the context of PUC-Rio's Tecgraf Institute. Tecgraf is a mid-size non-profit research and development organization that conducts projects with industrial partners and government institutions. Our subject legacy system is a large-size BDS that had been under active development from 2008 to 2014. Figure 1 shows a high level view of the legacy system. MOs, such as vehicles, have tracking devices installed, so that positioning data (PD) are sent periodically. Every PD are sent to an information flow processing (IFP) engine [14], which analyzes them to uncover non-conformities, such as a vehicle route deviation. Then, the streams are enriched with domain data and presented to users.

Over the years, the system has undergone a natural process of corrosion, on which the large source code became difficult to maintain due to the complexity of the system. Besides, the technology stack became obsolete, outpaced by current technologies, and the monolithic structure undermined the introduction of new technologies. Thus, in the advent of a new industrial partner with a closely related problem context, Tecgraf's managers realized that the process of recruiting and training new developers that would be able to implement a new instance of the legacy BDS was not feasible. Also, with the myriad of big data technologies that emerged in the last decade, such as Cassandra and Kafka, and the mentioned drawbacks found in the legacy BDS, Tecgraf's managers decided that the best approach would be designing a new architecture from scratch, based on microservices and event-driven principles. Besides, the new architecture should embrace widely adopted open-source technologies instead of relying on in-house solutions.

B. Goal and Research Questions

Developing a BDS poses several challenges to developers, such as steep learning curves, lack of modeling and debugging support, and data consistency [4]. Moreover, the recent trend

towards adoption of microservices architectures has shown that without a careful design, drawbacks related to redundancy and data consistency may emerge [11]. Albeit there is substantial body of work reporting on microservices decomposition [8], [11], designing a microservice-based system without decomposing an existing monolithic system is not substantially covered in the literature [11]. Besides, to the best of our knowledge, there is no work that reports the challenges and lessons learned on replacing a legacy BDS with a new microservices EDA. Thus, our goal is to report the experience of replacing a legacy BDS with a microservices-based EDA without decomposing the existing system. To achieve our goal, we derived three Research Questions (RQs), which are detailed hereafter.

RQ1. What are the reasons that motivate the adoption of a microservice-based EDA to replace a legacy big data system? This first research question intends to comprehend the reasons that motivate the adoption of architectural alternatives different from the one found in the legacy big data system, particularly regarding EDA and MS architectural style. While there are different works on the motivations for migrating to MS-based architectures, we wanted to understand the specific motivations of our context.

RQ2. What are the benefits and limitations perceived on replacing a legacy big data system with a microservice-based EDA? The second research question explores the perceived benefits and limitations (post-development) related to the adoption of a new microservice-based EDA. We aim to uncover the technical decisions that the development team has taken that derived drawbacks and positive results.

RQ3. What are the challenges faced and lessons learned while replacing a legacy big data system with a microservice-based EDA? The third research question concerns unveiling the challenges and lessons learned that were perceived throughout the development process.

C. Method

This section presents the research method employed to answer our research questions. The organization of the method follows the template of Santos and Travassos [13], which suggest one AR stage per section. Figure 2 depicts the AR process based on Davidson et al. [15]. The process starts with a diagnosis of the problem, followed by a plan to address the issue being investigated. Then, the plan is put into practice in the intervention phase. Lastly, an evaluation and analysis is carried out. Although the AR process allows for iterating through phases to achieve results incrementally, this study reports a full cycle of the methodology.

*1) **Diagnostic**:* Santos and Travassos [13] argue that this stage "consists of exploring the research field, stakeholders and their expectations". Thus, as this phase naturally maps to answering **RQ1**, we have designed an exploratory survey to collect the expectation of Tecgraf's main stakeholders involved in the project. In this survey, we aim to report on the drivers of the technical decision on moving towards a microservices EDA and obtain a view about the drawbacks found in the legacy BDS. Besides, the survey results allow to better understand the problem context and to cross-validate the findings of subsequent steps of the AR process. The target population of this study is composed by product managers, software architects, and developers of Tecgraf institute that have contributed to the decision about replacing the legacy BDS. The following questions compose our survey:

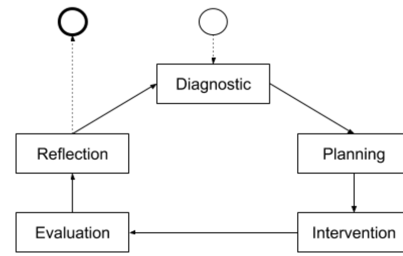


Fig. 2. Action research process

(Q1) What are the drawbacks found in the legacy BDS that motivate the substitution?
(Q2) What are the drivers for defining event-driven microservices as a target architecture?
(Q3) What characteristics of the legacy BDS are important to remain in the target system?
(Q4) What challenges would you expect to encounter in replacing a legacy BDS to a microservice-based EDA?

(Q1)-(Q3) are defined with the goal to gather information on the legacy system and to extract requirements that must remain in the new architecture. (Q4) aims at gathering the perception of the stakeholders over the challenges incurred by the replacement process. Thus, we can cross-check if the expectations on challenges are met at the end. In addition, regarding the BDS, we conduct an analysis of the documentation, inspection of the source code and historical commits to uncover technical challenges faced by developers at the time.

*2) **Planning**:* This stage concerns the definition of actions to be taken onward. A component of this phase is conducting a literature survey for the purpose of examining the research theme [13]. Thus, we report our searching process on the aforementioned themes and the sequenced set of activities to carry out during intervention step.

*3) **Intervention**:* According to Santos and Travassos [13], this phase concerns the implementation of planned actions, which are depicted "in a chronological way, describing how the activities were performed during the research." In this phase, data collection is a product of our experience playing the co-located role of software architects within Tecgraf/PUC-Rio. During this period, we have analyzed the results of several meetings, emails, interviews, and technical documents, such as use cases and user interface screen prototypes, in order to confirm our findings regarding the process of replacing a legacy BDS. The material collection was conducted between July 2018 and January 2020. Through intervention, we seek to unveil the challenges on adopting a microservice-based architecture from scratch (i.e., when no monolithic system is

refactored) with event-driven requirements. The results of this AR step enables us to partially answer **RQ2** and **RQ3**.

4) ***Evaluation and Reflection:*** This phase regards analyzing the effects of the actions taken. Although our study provides the point of view of two interveners playing a software architect role, it is worthwhile to enrich our understanding on the effects of the intervention by collecting the perceptions of other developers that also played a role in the project, i.e., contributed to the project code base. Therefore, in order to mitigate risks related to the report of outcomes of the intervention from a single point of view, we designed a survey to to gather the perception of other developers about the new architecture. This also allowed us to cross validate our findings to reduce limitations of the study. Through evaluation, we are able to complement our answer to **RQ2** and **RQ3**.

IV. RESULTS

This section provides detailed discussions on the AR methodology employed at Tecgraf to answer the RQs.

A. Diagnostic

The diagnosis phase "enables the identification of primary causes and circumstances faced by an organization" [13]. In this section, we analyze the project context by inquiring stakeholders' expectations and understanding the legacy BDS.

1) Survey with stakeholders: We applied the survey to four Tecgraf stakeholders (SH{1-4}) that were closely involved in the arrangements of the project. Regarding the drawbacks of the legacy BDS, the respondents unanimously agreed on the complexity and obsolescence of the source code. For instance, SH1 argues that the "code was poorly structured and documented," and "the technology was obsolete." Next, SH1 asserts that a driver for microservices adoption is that "data could grow rapidly and the performance could be a bottleneck; [...] an architecture able to escalate easily was very attractive." Besides, SH2 highlights that an EDA "facilitate the processing of events from different sources," "[supporting] the communication and isolation among microservices." Next, the respondents agreed that the core requirements to remain are georeferenced tracking of MOs, definition of route and activities for MOs, and proactive event generation based on trajectory data of MOs. Finally, they agreed that the lack of technical maturity could impose challenges. For instance, definition of API interfaces, microservices decomposition, database transaction issues, and high coupling among microservices were concerns raised.

2) Legacy big data system: In this step, we describe the process carried out to further understand the legacy BDS. We have acquired access to the legacy system's wiki, a repository for sharing information about the project. We found that the core functional requirements of the legacy system were about efficient ingestion and fast retrieval of positioning data from MOs and detection of non-compliance patterns in the streams, such as speed limit overrun. In production settings, problems related to data consistency started to arise. For instance, thread blocking and data races have led to slowness in the processing of events. The process for understanding issues was complex, once traceability information was insufficient due to the large code base divided in several modules. We found that the system has started with 500 MOs sending PD every minute and has grow up to 10000 MOs sending PD every 15 seconds.

We have also acquired access to the source code repository of the legacy BDS. Due to the large code base and minimal contact with developers of the legacy system, once they were not part of the organization anymore, the process lasted several weeks. In summary, we found that batches of streams were posted on queues placed in-memory, in a scheme similar to message queue systems. As multiple threads were employed to concurrently process the streams, this solution often led to data races and thread blocking. Misuse of Java concurrency primitives was the main reason for such issues. Lastly, we found that a custom-tailored IFP engine was engineered to monitor MOs (retrieved from in-memory queues) and detect events such as route deviations. Lastly, the legacy BDS LOC count for 400K, divided into application code and libraries.

B. Planning

The foundations that guide our planning stage were elucidated in the diagnosis stage. In this stage, we sought to gather knowledge on the research themes by searching the literature. We have submitted searches on Scopus digital library regarding microservices and event-driven architecture. In summary, although studies on microservices are prevalent [8], [16], we found that the problem of defining a microservice (MS) architecture for a new system still an ongoing problem in literature, with no clear guidelines [11]. Hence, we defined a sequence of phases to be followed in a defined timetable. We start with the conception, describing the requirements elicitation process. After, the architecture and design phase are conducted. Lastly, with the architectural blueprint of the system, we were ready to start the implementation phase.

C. Intervention

In this section we provide an in-depth discussion over the process conducted to replace the legacy BDS with a microservices EDA. As suggested by Santos and Travassos [13], the intervention is described in a chronological way.

1) Conception: The project has started with a researcher (the first author), a project manager (fourth author), a senior developer, and two requirements analysts. Soon, the requirements started to change. By working closely with the industry partner, the requirements team realized that their needs were different from the ones described in the contract. As we aimed a microservice-based architecture, this context has played a role on the process of defining our services.

Although there are general guidelines on migration patterns and deducing microservices from a monolithic system [17] [18], by the time we were investigating approaches to support the process of defining our services, we have not found studies focused on how to define a microservices architecture from scratch, i.e., when no monolithic system is decomposed.

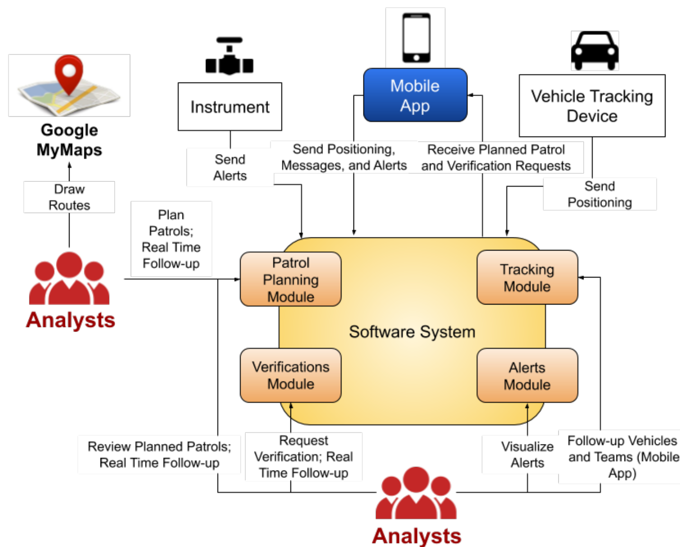


Fig. 3. Business capabilities

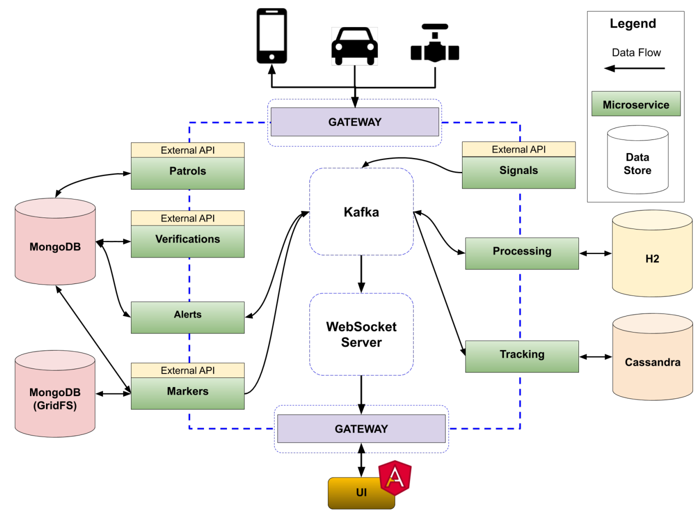


Fig. 4. The resulted microservices event-driven architecture

Literature [17] [18] often cites Domain-Driven Design (DDD) [19] as a compelling technique to identify subdomains of the business, where industry practitioners advocate that each subdomain maps to a bounded context (a deployable unit). However, Evans [19] advocates first for a discovery process of the application domain, where its "understanding comes from diving in, implementing an initial design based on a probably naive model, and then transforming it again and again."

Hence, we followed the advice of starting with a naive model based on the business capabilities (BCs) [20] identified so far (1-month period). A BC, also referred as a bounded context [16], "is something that a business does in order to generate value," often representing a delimited domain business model. We documented the conducted requirements gathering meetings and identified four major BCs of the domain, as shown in Figure 3. Our reasoning for defining the BCs is explained as follows.

Analysts plan a patrol (trip) composed by a route to be followed and a set of inspections (stop points) to be performed along the work journey. The structure of patrol and verification trips are no different, however, a verification corresponds to an unforeseen inspection triggered by the reception of a denounce, while a patrol is previously defined and scheduled in advance. Also, distinct analyst teams handle patrols' planning and verifications. Therefore, we defined both as distinct BCs.

The *Alerts* BC is responsible for the ingestion, processing, and exposition of alerts coming from any source. For instance, instruments installed in oil pipelines periodically send alerts concerning suspicious activities, such as manual excavations. Besides, through the mobile app, in-field operators can communicate messages and alerts to analysts in real-time.

The *Tracking* BC is responsible to ingest and process real-time PD of patrols and verifications. An in-field team is assigned to a patrol or a verification and tracking is automatically enabled by the the mobile app they carry in operation. In addition, vehicles assigned to a team also send tracking data. Lastly, the *Tracking* BC is responsible for storing and serving all trajectory data history of mobile devices and vehicles.

2) Architecture and design: This section discusses how the requirements elicited were translated to an architectural design, which is exhibited in Figure 4.

Defining a target stack. The intervention process started at a slow pace due to frequent requirements changes. This context has made the case for focusing on the analysis of suitable technologies for the target architecture. Tecgraf has long-lasting expertise in developing distributed systems with Java. Besides, as the developers were proficient in the language, Java was a natural choice to be the main back-end programming language. As the project comprised the development of a web application with a distributed architectural style, in order to meet stakeholders expectations over embracing open-source technology, instead of writing custom-tailored infrastructure solutions (e.g., logging, data access, and dependence management) from scratch, a reasonable choice was relying on a well-adopted framework to support our development. Thus, we listed a set of capabilities that the framework must deliver: support for the development of REST APIs [21]; support for dependency injection [22]; support for hot reload; embedded support for database access; integrated support for message queuing systems; and support for reactive programming model. Although there are a number of feasible web framework options for Java platform, we decided to select Spring due to its rich ecosystem, composed by multiple integrations built by a supportive community. Besides, support for Spring in Question&Answers communities and extensive online documentation played a role in the decision.

As scalability is a major driver of the target architecture, we opted to adopt the database per service pattern [23]. Besides, as the requirements were being progressively elicited, we aimed to get rid of schema changes for each new version. Then, we listed three important features for a default persistence technol-

ogy for our services: (i) flexible-schema model, (ii) geospatial query support, (iii) high industrial adoption. Then, MongoDB was selected due to its support to geospatial indexes, replication, load balancing, file storage, and the representation of complex associations within a single record.

Defining services corresponding to business capabilities. *Patrol Planning* and *Verifications*, as depicted by Figure 3, comprehend distinct BCs. This comprehension led to designing both as distinct microservices, as shown in Figure 4. In-field teams are either assigned to a planned patrol or a verification, and, through the mobile app, they are able to retrieve data from the respective service in order to support daily operation. As a verification is spawned due to a denounce (i.e., it is not planned daily), the mobile app is programmed to proactively request the existence of an assigned verification in a time interval. In case the team is assigned to a verification, the patrol is paused until the end of the verification operation.

Next, we chose to define the *Tracking* BC as a microservice due to two reasons: (i) guarantee PD durability and (ii) enable retrieval of historical PD. As scalability of PD ingestion and retrieval is a central concern in the architecture, we listed essential quality attributes a data store must deliver in this case: (i) write-heavy workload support, (ii) availability, (iii) scalability, and (iv) consistency. Thus, we surveyed DBMS to compare additional quality attributes. The work of Lourenço et al. [24] gave us a starting point, as shown in Table II. Although not considered a database, but rather a pattern, we found worthwhile to also analyze CQRS [25] as a candidate solution. Given the superior write-performance, we selected Cassandra as our solution to intensive PD ingestion. From the point of view of event sourcing pattern [26], we modeled PD as an event, thus the state of a MO is represented as a sequence of state-changing events (i.e, a historical track).

Following the advice of Balalaie et al. [17], which recommend "to start with a low number of services [...] and incrementally add more services" as the team understands requirements better, we realized that letting *Tracking* MS also deal with the reception of data from external sources (e.g., vehicle and mobile app) could compromise its performance on serving tracking historical data. Thus, in order to provide a singular interface for reception of PD, we defined a MS (*Signals*) that is responsible for abstracting the receiving of PD through a RESTful API by guaranteeing the conformance of the API defined and communicating it to interested services. This choice proved to be right due to scalability requirements entailed by the application, on which we could increase the number of *Signals* instances to cope with growing number of MOs sending PD without affecting serving historical data.

Lastly, we defined a specific MS (*Alerts*) responsible for reception, processing, and serving alerts. In other words, alerts are events communicated to the system that should be stored consistently and informed to interested services. Thus, we applied the transactional outbox pattern [27], which advocates that a service that publishes a domain event [28] (in our case, an alert) must atomically update the database and publish an event. In the case of a growing number of different classes of events and interested services, it is important to have a unit of scalability which is represented by this MS.

TABLE I
DOMAIN EVENTS IDENTIFIED

Actor	Command	Event
Mobile app	Start patrol	Patrol started
Analyst	Assign verification	Verification assigned
Mobile app	Start verification	Patrol paused
Mobile app	Finish verification	Patrol resumed
MO	Update positioning	MO state changed
Processing	Register route deviation	Route deviation detected

Defining domain events. Domain events [28] are often employed in EDA to communicate services about a change in the state of a domain object. A domain event, when received by a service, may trigger actions to be performed. A domain event is often published through a communication channel (a.k.a. topic) that is subscribed by interested parties, allowing a low coupled and non blocking message passing [29]. This approach is particularly important in data-intensive systems to avoid polling mechanisms, which may become a bottleneck with time. In our case, domain events were elicited along brainstorming reunions to evolve the domain knowledge, and, due to space constraints, we summarize the main ones in Table I. Based on the work of Brandolini [30], the columns are explained as follows: an Actor is the source object of an action; a Command represent an action triggered by an actor; and an Event is the consequence of an action.

As mentioned earlier, a PD received by the system is a domain event that represents that the state of a MO (mobile app or vehicle) is updated. When a PD is received by *Signals*, it checks the conformance of the PD object and publishes it to a Kafka topic called *signals*. Although we do not adopt transactional outbox pattern [27] in *Signals*, we allow for faster communication of the new state to interested services (since we do not wait for a synchronous database operation) by relying on Kafka consistency guarantees, configuring a suitable trade-off when it comes to a (quasi) real-time system. On the other side, every alert processed by the MS *Alerts*, after stored in the database, is published in a topic called *alerts*. After analysis (by an analyst), an alert may result in the assignment of a verification to an in-field team. This assignment event is then delivered to the given in-field team through the mobile app.

Serving domain events to end-users. A user interface application (UI-APP) was developed in Angular [31] to enable analysts to visualize domain events (e.g., alerts) in real-time. We chose not to include programming polling mechanisms in our UI-APP because we envisioned that real-time domain events retrieval from our microservices could experience high latency as the number of users and amount of data evolve. A technology that could intermediate domain events coming from Kafka topics to web browser clients was necessary. Thus, we found that *WebSocket* [32] make a suitable protocol to handle real-time event-driven communication in the front-end layer by avoiding polling the server for data.

TABLE II
TECHNOLOGIES SURVEYED FOR THE TRACKING MICROSERVICE

Technology	Write-Performance	Availability	Scalability	Maintainability	Read-Performance	Consistency
CQRS	Average	Bad	Bad	Bad	Great	Good
CouchDB	Below average	Great	Below average	Good	Average	Good
MongoDB	Below average	Below average	Below average	Average	Great	Great
Cassandra	Great	Great	Great	Average	Below average	Great

Information flow processing. Cugola and Margara [14] refer to systems that "require processing continuously flowing data from geographically distributed sources [..] to obtain timely responses to complex queries" as information flow processing (IFP) applications. At first we investigated Flink [33] as our IFP engine due to its ability to compute operations over stream data, like our real-time PD. However, as asserted by Orleans documentation [34], these systems present a "unified data-flow graph of operations that are applied in the same way to all stream items," thus hindering applying filtering or aggregation operations over different data items in the same computation. For example, as part of the process of checking real-time trajectory data of MOs against their respective planned route, we found limited support to integrate an external call to *Patrols* MS API in order to retrieve the planned route. Thus, we built a tailored solution (*Processing* in Figure 4) that takes advantage of reactive primitives of Spring and in-memory data processing. Thereby, based on a 5-minute time-window, *Processing* retrieves PD associated with each in-field team (from *signals* topic) and triggers the route deviation detection computation. If a deviation is detected, a route deviation is registered and the respective event is triggered. The *Alerts* MS acknowledges the event as a new alert and publishes it in the *alerts* topic. This separation of concerns allow us to scale separated parts of the system independently.

3) Implementation: In total, an overall effort of more than 7500 hours were employed by the development team, resulting in a system of around 30,000 lines of Java code, 14,000 lines of TypeScript code, and 29 data tables and documents. In average, each MS has 2,500 lines of code. Due to the large number of microservices and supportive technologies, and the lack of knowledge on state-of-the-art DevOps tools, a great effort was put into deployment. For instance, Docker containers were used to package our services. Besides, several fixes that were not expected earlier were implemented only to adapt to Docker deployment. This context makes the case for introducing DevOps earlier in the development process.

D. Evaluation and Reflection

This section reports the results of survey conducted with developers and discusses challenges and lessons learned. Although the lessons learned are related to the specific action research project context described in this paper, we believe most of them are generalizable to other industrial settings.

1) Survey with developers: As mentioned in Section III-C4, we designed a survey to collect the point of view of three developers that collaborated in the development process. First, we have defined a set of challenges collected along the intervention. Then, we questioned the developers on their agreement and also inquired them about additional challenges. Their perceptions are summarized along the next section. Due to space constraints, survey details are found online [35].

2) Challenges and lessons learned:

Defining microservices. Although *Patrols* and *Verifications* represent different domain concepts, which lead to different domain events, designing them as distinct microservices caused the problem of duplicate concepts [19], on which duplicate efforts on each requirement change (as a result of new knowledge acquired) was observed along the development life cycle. As a lesson learned, we suggest following the advice of Fowler [36], which advocates for the *Monolithic-First* approach, on which a project "shouldn't start [...] with microservices, even if you're sure your application will be big enough to make it worthwhile." Waiting for requirements to mature is essential to define microservices properly. However, emerging research discusses model-driven development of microservice-based systems, which may help mitigate some of the impedance on microservices design [37].

Data modeling. The fast-paced development process altogether with the adoption of novel technologies imposed a challenge on getting data modeling right. The distributed architecture forced us to adopt schema-less and denormalized data models, encapsulated through APIs, rather than normalized data models and data consistency guarantees usually found in monolithic systems, in line with Gorton and Klein's discourse over BDS [3]. Furthermore, even though designing services communication based on domain events augments the expressiveness of the domain, from the point of view of developers, the myriad of services and technologies led to difficulties in troubleshooting problems. The complex data flow entailed by the application often led to misunderstandings and slowed the process of identifying the root cause of errors.

Selecting an IFP engine. Attempts to translate our IFP requirements to Flink were unsuccessful (see Section IV-C2). A second problem was relying on a short time window for implementing our IFP use cases. As the number of technologies employed were already large and the team had no previous experience with IFP engines, we realized that the learning curve could compromise subsequent sprints. As a lesson learned, we highlight that the selection of an IFP solution is an architectural decision. It means that the chosen IFP engine should be adherent to the architecture, and not the opposite. Furthermore, we consider Orleans streams [34] a promising candidate for expressing computations that span different items due to its flexible processing engine.

Embracing failure. Some MS-oriented frameworks (e.g., Spring) fail to present extensive support for failure handling in workflows spanning multiple microservices. For instance, in the absence of distributed transactions, the developer should hard-code logic related to recovering from failures in such workflows. Furthermore, given the low granularity nature of MS instances and the difficulty on reasoning over each MS' local state globally, we advocate for a programming model that specifies fault-tolerance properties that we can reason about on requests spanning multiple microservices.

V. Concluding Remarks

This study reports an industrial experience regarding the replacement of a legacy monolithic BDS to an event-driven microservice-based architecture. Microservices promise to automatically react to failure and changing workloads, provide independent deployment, and support for polyglot technologies [7] [12]. EDA commit to enable a reactive programming model among high-cohesive components that proactively react to incoming events by performing a computation or triggering it in another component [9].

However, the joint use of microservices and EDA has not been previously discussed in the context of BDS. Moreover, we present how microservices can be defined without refactoring a legacy monolithic system. From requirements elicitation, through architecture design, and implementation, we provided an example on how a system with data-intensive requirements can benefit from microservices and event-driven principles.

The main takeaways from our experience are as follows. Defining microservices too early in the development process may yield into a wrong definition. Besides, in a fast-paced development scenario, waiting for requirements to mature is essential in getting microservices right. On one hand, microservices support for easier maintenance and fault-isolation were perceived as benefits to the architecture. However, the complex data flow entailed by the number of microservices, as well the myriad of technologies were perceived as drawbacks.

References

[1] DOMO. Data never sleeps 5.0. [Online]. Available: http://bit.ly/2QPdcby

[2] IDC. The growth in connected iot devices is expected to generate 79.4zb of data in 2025. [Online]. Available: http://bit.ly/2QpEnuw

[3] I. Gorton and J. Klein, "Distribution, data, deployment: Software architecture convergence in big data systems," *IEEE Software*, vol. 32, no. 3, pp. 78–85, 2014.

[4] O. Hummel, H. Eichelberger, A. Giloj, D. Werle, and K. Schmid, "A collection of software engineering challenges for big data system development," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 362–369.

[5] R. Laigner, M. Kalinowski, S. Lifschitz, R. S. Monteiro, and D. de Oliveira, "A systematic mapping of software engineering approaches to develop big data systems," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 446–453.

[6] J. Boner, D. Farley, R. Kuhn, and M. Thompson. The reactive manifesto. [Online]. Available: https://www.reactivemanifesto.org

[7] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *IEEE ICSA Workshops*, 2017, pp. 243–246.

[8] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, May 2018.

[9] M. Richards, *Software Architecture Patterns*, 1st ed. O'Reilly, 2015.

[10] H.-M. Chen, R. Kazman, and S. Haziyev, "Agile big data analytics development: An architecture-centric approach," in *Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS)*, ser. HICSS '16. Washington, DC, USA: IEEE Computer Society, 2016, pp. 5378–5387.

[11] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio, "An experience report on the adoption of microservices in three brazilian government institutions," in *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, ser. SBES '18. New York, NY, USA: ACM, 2018, pp. 32–41.

[12] O. Zimmermann, "Microservices tenets," *Comput. Sci.*, vol. 32, no. 3-4, pp. 301–310, Jul. 2017.

[13] P. S. M. dos Santos and G. H. Travassos, "Action research can swing the balance in experimental software engineering," *CoRR*, vol. abs/1306.2414, 2013. [Online]. Available: http://arxiv.org/abs/1306.2414

[14] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

[15] M. M. G. K. N. Davison, R. M., "Principles of canonical action research," *Information Systems Journal*, vol. 14, no. 1, p. 65–86, 2004.

[16] N. Dragoni, S. Dustdar, S. Larsen, and M. Mazzara, "Microservices: Migration of a mission critical system," *IEEE Transactions on Services Computing*, vol. PP, 04 2017.

[17] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, vol. 48, no. 11, pp. 2019–2042, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2608

[18] F. Rademacher, J. Sorgalla, and S. Sachweh, "Challenges of domain-driven microservice design: A model-driven perspective," *IEEE Software*, vol. 35, no. 3, pp. 36–43, May 2018.

[19] Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[20] C. Richardson. Pattern: Decompose by business capability. [Online]. Available: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

[21] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[22] R. Laigner, M. Kalinowski, L. Carvalho, D. S. Mendonça, and A. Garcia, "Towards a catalog of java dependency injection anti-patterns," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27*, 2019, pp. 104–113.

[23] C. Richardson. Pattern: Database per service. [Online]. Available: https://microservices.io/patterns/data/database-per-service.html

[24] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, and J. Bernardino, "Choosing the right nosql database for the job: a quality attribute evaluation," *Journal of Big Data*, vol. 2, no. 1, p. 18, Aug 2015.

[25] M. Fowler. Cqrs. [Online]. Available: https://martinfowler.com/bliki/CQRS.html

[26] C. Richardson. Pattern: Event sourcing. [Online]. Available: https://microservices.io/patterns/data/event-sourcing.html

[27] ——. Pattern: Transactional outbox. [Online]. Available: https://microservices.io/patterns/data/transactional-outbox.html

[28] ——. Pattern: Domain event. [Online]. Available: https://microservices.io/patterns/data/domain-event.html

[29] K. Birman, *Reliable Distributed Systems*. Springer, 2005.

[30] A. Brandolini. Introducing event storming. [Online]. Available: http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html

[31] Angular. Google. [Online]. Available: https://angular.io

[32] Using websocket to build an interactive web application. Spring by Pivotal. [Online]. Available: https://spring.io/guides/gs/messaging-stomp-websocket

[33] Apache flink. The Apache Software Foundation. [Online]. Available: https://flink.apache.org

[34] Why orleans streams? Microsoft. [Online]. Available: https://dotnet.github.io/orleans/Documentation/streaming/streams_why.html

[35] Survey details. [Online]. Available: https://zenodo.org/record/3606316

[36] M. Fowler. Monolithfirst. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html

[37] F. Rademacher, J. Sorgalla, P. Wizenty, S. Sachweh, and A. Zündorf, *Graphical and Textual Model-Driven Microservice Development*. Springer International Publishing, 2020, pp. 147–179.

View publication stats