On the Maintenance of a Scientific Application based on Microservices: an Experience Report

***Abstract*—Microservices Architecture has been adopted by several companies to replace monolithic applications and develop new ones. Several works point out that this approach supports the design of maintainable software systems. However, none of them presents a quantitative empirical study on the extent of the maintenance support in a real-world application. This work assesses how Microservices Architecture supports software maintenance through an empirical quantitative study of a scientific application built from scratch. We collected data since the beginning of this project, from January 2016 to December 2019, and analyzed 19 microservices, 34 repositories, and 15,408 commits. Then, we present the lessons learned during the project that allowed reaching the assessment results. Our findings will assist practitioners in making architectural decisions and pointing out research opportunities for academics.**

I. Introduction

Microservices Architecture has been adopted by several companies such as Netflix [1], Amazon [2], and Uber [3]. Microservices Architecture is a decentralized state-of-the-practice Service-Oriented Architecture (SOA) in which systems are composed of services that collaborate to meet goals, communicating through lightweight mechanisms (*e.g.*, as Web APIs) [4], [5]. This approach relies on small and autonomous services that work together, instead of building coarse-grained services (a.k.a. monolithic applications). It also promotes low coupling and high cohesion among microservices aiming to avoid a change in one microservice that affects another, thus minimizing maintenance efforts. Being composed of fine-grained services, a Microservices Architecture also enables to scale of only those services that need it. Furthermore, by enabling different services to have distinct deployment cycles, microservices can be available independently, thus minimizing release cycles.

Several works claim that Microservices Architecture supports the design of maintainable software systems [6], [7]. According to Bogner *et al.* [7], there are four types of software system maintenance that can be classified into two groups. When the functional requirements are stable, maintenance is *corrective* or *perfective*. When those requirements change, maintenance is *adaptive* or *extending*. Based on this definition, some works from the literature have analyzed maintenance metrics. For instance, Xie *et al.* [8] analyzed maintenance metrics of seven open-source projects, covering 653 official releases. However, to our knowledge, there is no quantitative empirical study on the maintenance of real-world applications based on Microservices Architectures. Thus, there is a lack of understanding of how Microservices Architecture endure adaptive and extending maintenance scenarios in such perspective.

This work aims at investigating the use of Microservices Architecture to support the development and maintenance of real-world software applications. Thus, we conducted an empirical quantitative study of a scientific application that has been built from scratch using Microservices Architecture. We collected data from the beginning of this project, in January 2016, until December 2019, and analyzed 19 microservices, 34 repositories, and 15408 commits. During this period, the application endured different types of adaptive and extending maintenance scenarios, such as new features, changes in requirements, and significant refactorings to enhance distinct quality attributes. We took many decisions to reach the results presented by the metrics assessment, and we highlighted the lessons learned. They are useful for practitioners, so they can avoid the decisions we have made that led to architectural issues and follow the ones that brought good results, and for researchers that want to further investigate the maintenance aspects of Microservices Architectures.

The remainder of this work presents a description of Microservices Architecture and its main principles (Secion II), the target scientific application (Section III) and its study (Section IV). Then, we discuss the lessons learned (Section V). In Section VI, we contrast our study against related work and, in Section VII, we present the conclusions.

II. Background: Microservices Architecture

Microservices are small and autonomous services that work together and should be built according to the following principles [5], [9]:

- **Model around business concepts**: Services modeled around business-bounded contexts are arguably more stable than those structured around technical ones [5]. Business concepts should be a primary driver for modeling services to build more stable interfaces and to facilitate the application of changes in business processes. Employ techniques and principles to identify and conceptualize services (such as Domain-Driven Design [10]), focusing on the most valuable organization's features.
- **Hide internal implementation details**: Information hiding is a key strategy to support architectural changeability [11]. Thus, to maximize the ability to evolve microservices independently, they should hide implementation details and databases.
- **Fine-grained interfaces**: Define units with specific responsibilities that encapsulate both processing logic and data [12].

- **Smart endpoints**: Services should communicate with each other via lightweight protocols, like APIs [5], *e.g.*, exposed via Web APIs or asynchronous message queues. These can be managed and scaled independently of each other.
- **Decentralization**: Microservices should be highly cohesive and loosely coupled to support modifiability, *i.e.*, a change in one microservice should not affect another microservice. Furthermore, approaches that might lead to centralization of business logic, like *orchestration* and *Enterprise Service Bus* [13], should be avoided - prefer a choreography approach [5], [9].
- **Independently deployable**: Developers should be able to change a microservice and release it into production without having to deploy any other service [5]. It increases the speed of release of new features and the autonomy of the teams owning each microservice. Effective packaging methodologies (*e.g.*, Docker [14]) promotes artifacts through continuous deployment processes .
- **Culture of Automation**: Microservices might add complexity by increasing the number of independent services that communicate with each other. Thus, embracing the culture of automation is a crucial way to address this complexity. Automation can rely on (DevOps) continuous delivery processes and tools [15], such as Jenkins[1]. Employ automated approaches for configuration, performance, fault management and testing, extending agile practices, and using service monitoring.
- **Cloud-native design principles**: Employ guidelines such as distribution, elasticity, automated management, and loose coupling [16].
- **Isolate failure**: A Microservices Architecture can be more dependable than a monolithic system, but only if it is planned for that [5]. A consequence of relying on multiple microservices is that they need to be designed to tolerate the failure of services. Any service call could fail due to service provider unavailability or a network problem, and the client has to respond as gracefully as possible.
- **Highly observable**: Breaking a system up into fine-grained microservices results in benefits, but also adds complexity, *e.g.*, to monitor the behavior of multiple microservices [5]. Techniques like semantic monitoring [17], [5] should be used. It checks if the system is behaving correctly by injecting synthetic transactions to simulate real-user behavior, enabling the monitoring of multiple microservices automatically.
- **Multiple paradigms**: Combine and leverage best fit computing approaches and storage systems (e.g., relational or NoSQL databases) in a polyglot programming [18] and persistence [19] strategies.

Ensure that teams own their microservices is an essential step in making them responsible for the changes that are made [5]. Teams should also be aligned to the organization according to Conway's law [20], so they become domain experts on the business-focused service they are creating [5].

[1]https://jenkins.io

III. Seismic Interpretation Advisor (SIA) Prototype

The Seismic Analysis Prototype is an artificial intelligence advisor to speed-up the appraisal of oil and gas prospects. The tool has been developed to acquire knowledge from geophysicists and previous seismic interpretations.

The goal of the project was to investigate techniques that allow geophysicists and geologists to enhance their ability to interpret seismic data. Such interpretation enhancement is achieved using a combination of physics-based models, machine learning models for image comprehension, and data visualization techniques integrated with the knowledge extracted from seismic interpreters.

A. *Software Architecture*

Scientific software may be long-lived and, as a consequence, subject to regular maintenance activities as requirements change [21]. One of the expected benefits for choosing Microservices Architecture to design and implement the prototype was support for maintenance. The prototype has many microservices that can be considered "scientific software" given the complexity of their domain [22]. For instance, a microservice builds sophisticated statistical models based on the physical properties of seismic data. The research related to this scientific software was published or patented. Most of this scientific software was initially implemented as a set of scripts, which later became microservices. Thus, one of the benefits of Microservices Architecture was that it allows different teams of researchers to conduct their experiments and independently develop microservices. Furthermore, it allowed each team to choose the most appropriate programming language as described in Multiple paradigms (see Section II). Python was used in 79% (15/19) of the microservices, Java in 26% (5/19), Javascript in 11% (2/19), and both R and prolog in 5% (1/19). Note that some microservices were implemented using two programming languages.

Synchronous communication among microservices was implemented using REST API, while asynchronous and long-running requests were handled using RabbitMQ[2]. Services' contracts were specified using Open API (a.k.a. Swagger in its version 2.0)[3]. NoSQL Mongo database[4] for documents and AllegroGraph triplestore[5] for a knowledge base were used.

The prototype's software architecture relies on the "API Gateway" pattern to encapsulate the internal structure of the prototype and to implement authorization and request logging [23]. One of the microservices principles is that they must be highly observable (Section II), thus each microservice has an operation for checking its availability as suggested by the "Health Check API" pattern [23]. Furthermore, the ELK

[2]https://www.rabbitmq.com
[3]https://swagger.io/docs/specification/about/
[4]https://www.mongodb.com/
[5]https://allegrograph.com/

stack[6] is used for logging, monitoring, and debugging the microservices.

B. *Continuous Delivery*

One of the reasons for choosing microservices architecture was that it enables continuous development and delivery [24]. We adopted the ”Deploying a service as a container” pattern [23], because containers have a lower overhead than virtual machines[25]. Each microservice is encapsulated by one container, which is deployed on a Kubernetes cluster. Kubernetes helps to manage the containers and provide availability and scalability [26] to the prototype.

Instances of the prototype were deployed on three on-premise environments that were created on a local cluster (*e.g.*,, `dev`, `user--tests`, and `prod`). The motivations for creating three environments were both from development and business perspectives. From the development perspective, the environments allowed researchers to test different aspects of their microservices (*e.g.*, integration tests, stress tests). Once a microservice passed all the tests in one environment, it could be deployed in the next environment. Thus, these environments could be ranked from low-dependable to high-dependable environments. This strategy was particularly helpful in executing tests that might take hours, which are difficult to execute in an environment in which microservices are faulty and being frequently re-deployed. From a business perspective, the high-dependable environment was often used for demos.

The deployment pipeline is automatic, and initiated when the researcher pushes new code to GitHub Enterprise (GHE)[7]. Jenkins detects that new code has been pushed and builds a new Docker[8] image. After building the new image, some of the researchers also included automated tests before deployment. Then, a new Docker instance is deployed into Kubernetes according to the specified environment.

Two environments on client's on-premise clusters were also created (`user--tests` and `prod`). The purpose was to have environments with progressively levels of dependability. The low-dependable environment has limited access; that is, only a few selected people could access the prototype, and manually tested. The high-dependable environment can be accessed by multiple end-users. Due to the client's legal and security constraints, we were not allowed to deploy the prototype in a cloud environment.

IV. Empirical Study

This section presents the empirical study conducted in this work.

A. *Study Planning*

The goal of this study is to assess to what extent Microservices Architecture supports software maintenance. Considering this goal, we selected a set of maintenance metrics suitable for microservices and maintenance based on a literature review [7] which used references we point out for each category. The selected metrics can be classified into three categories: (i) size metrics: measure the number of services, LOC, and number of operations (a.k.a. endpoints) for each service - based on [27][8]; (ii) coupling metrics: measure the interdependency between microservices - based on [28]; (iii) bug-fix metrics: measure the relation between bugs and code - based on [29].

We collected data of the 19 microservices of SIA, and their dependencies from January 2016 until December 2019. The relation between microservices and GHE repositories varies: some microservices were stored into a repository and their dependencies into other repositories. In a few cases, two microservices were stored into a single repository. The total number of repositories analyzed was 34.

B. *Study Operation*

In order to collect some of these metrics, we implemented a GHE mining tool for extracting and analyzing data from GHE API[9]. In SIA, microservices and repositories do not always have a one-to-one mapping: a single repository can contain two microservices, and a single microservice might have used multiple repositories over time. Our GHE mining tool handles such a many-to-many relationship between services and repositories. Furthermore, it handles source code at the file level, including or excluding files based on filename patterns. For instance, `html` files can be excluded from counting the number of LOC of a python-based microservice. Data extracted from GHE is stored into an SQLite database to avoid unnecessary calls to GHE API, since there is a limited number of calls *per* day for each client, and to facilitate our analysis.

Many collected metrics are based on LOC, which was computed using the Github API of the microservices' repositories and their libraries. Note that, for computing LOC, we included only libraries that were implemented by our department's staff. If multiple microservices reuse a library, its number of LOC is added to each microservice that uses it.

We have made the GHE mining tool available[10] to support the ability to reproduce this study and also to foster studies based on mining software repositories.

Other metrics, such as coupling metrics, were manually collected by analyzing the source code of all the microservices.

C. *Size Metrics Analysis*

Figure 1 shows the number of microservices that have been created during the project. In the first year, four microservices were created as some workstreams were not mature enough. From the beginning of 2017 until the end of 2019, 13 microservices were created because it was the most intensive development and integration period, *i.e.*, the experiments conducted by some research teams achieved conclusive results requiring the development of microservices. By the end of 2019, all prototype's microservices have been created.

[6]https://www.elastic.co/what-is/elk-stack
[7]https://github.com/enterprise
[8]https://www.docker.com

[9]https://developer.github.com/v3/
[10]Link omitted due to double-blind review

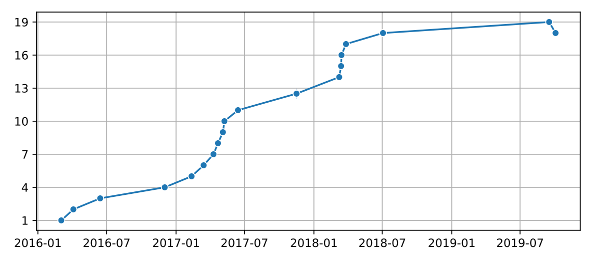

Fig. 1. Number of microservices over the time

Figure 2 shows the number of LOC (comments and blank lines are included) over time. The median number of LOC barely increased since the beginning of the prototype development: in July 2016, it was 6497, and in December 2019, it was 6856, a 6% increase. In the same period, the average number of LOC *per* microservice increased from 7511 to 14667 (95%). The blue shade represents the standard deviation, and it shows a wide range of LOC *per* microservice, which highlights that the teams had two distinct approaches for handling new functionalities: (i) Add new functionalities to existing microservices; or (ii) Create a new microservice to implement new functionality. The majority of microservices are below the average (see the difference between the median and average curves), *i.e.*, some microservices were small and simple, and others were large and complex to maintain. Schermann *et al.* [30] observed that most of the participants in the survey reported a number of LOC *per* microservices ranging from 1000 to 10000. The median of our study lies within such a range.

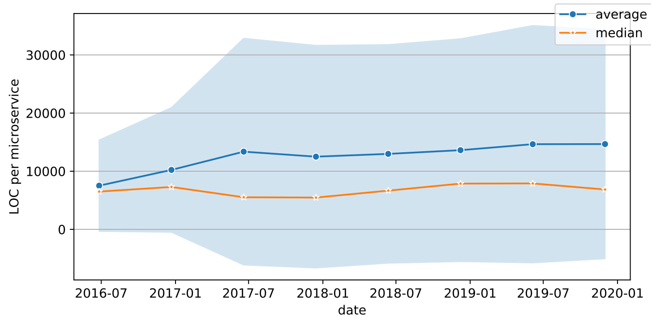

Fig. 2. Number of LOC *per* microservice

Figure 3 shows the number of operations *per* microservice, which indicates how complex a microservice is [7]. The behavior of the average number of operations *per* microservices and the standard deviation are similar to the behavior of the growth of the number of LOC *per* microservice. The average number of operations increased from 24 to 40 (67%). They are similar to Figure 2 for the same reasons: some research teams created new microservices that have few operations while other research teams implemented new functionalities by adding operations to existing microservices. The median decreased from 11 to 8 operations (-37%) as many microservices were created after 2017 were small.

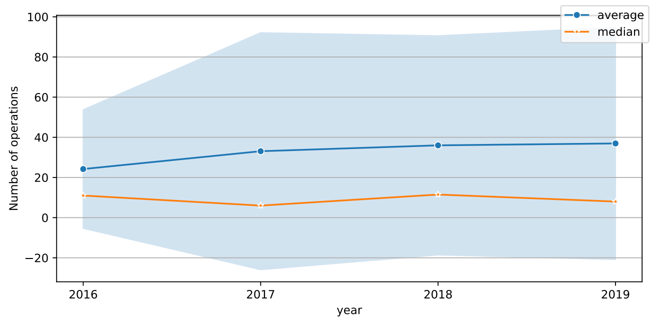

Fig. 3. Number of operations *per* microservice

Figure 4 shows the cumulative changes in 6 months, divided by the number of LOC. The median dropped from 1.36 to 0.38 (-72%), and the average dropped from 1.35 to 0.46 (-66%). Between July 2016 and January 2017, the microservices were small, and changes had a relative impact on them. In 2017, software development was intensive, and 42% of all microservices were created. It was also a period of instability, that is, a significant number of cumulative changes *per* microservice. Note that in July 2017, the standard deviation achieved its maximum value as some microservices were under intensive development while others were already stable. A slight increase in both curves in the last semester was mainly due to bug-fixing as the project was reaching the end.

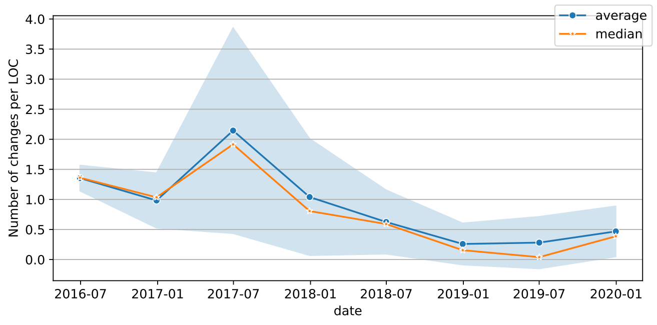

Fig. 4. Number of changes *per* LOC over time

D. Coupling Analysis

Afferent coupling of the microservice (a.k.a. Absolute Importance of the Service) is defined as the number of consumers of a given microservice. In contrast, Efferent Coupling is the number of distinct microservices that receives a request of a given microservice. In this prototype, all requests are made by either a microservice or the UI (User Interface). We omitted the UI from efferent metrics as it is not a microservice.

Figure 5 shows the efferent coupling over time, which doubled from 1.1 in May 2017 to 2.2 in November 2019, and the median increased from 1 to 2 in the same period.

90% of all microservices have been created before July 2017 (Figure 1); thus, the increase in the average efferent coupling is only partially caused by the growth of the number of microservices that were integrated with the other microservices. The other reason for the increase of coupling was that existing microservices provided new operations to other microservices. Eight microservices created at the beginning of the project provided one or more operations to a new microservice after May 2018.

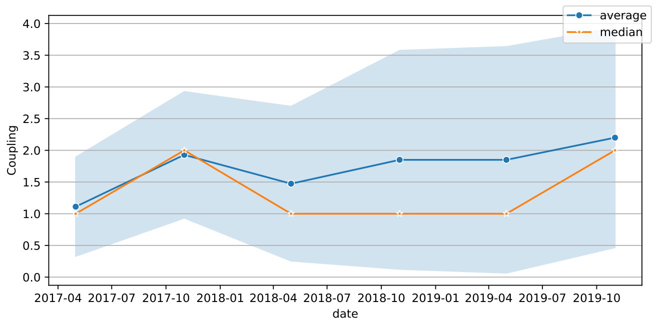

Fig. 5. Efferent coupling over time

Figure 6 shows that the average afferent coupling over time raised from 0.8 to 1.8, and the median raised from 0 to 1. Similarly to the behavior of the curves for the efferent coupling, the first growth was mainly due to the creation of new microservices. In contrast, the second growth of the average was due to the increased coupling of existing microservices. After May 2018, 5 microservices established connections with other microservices.

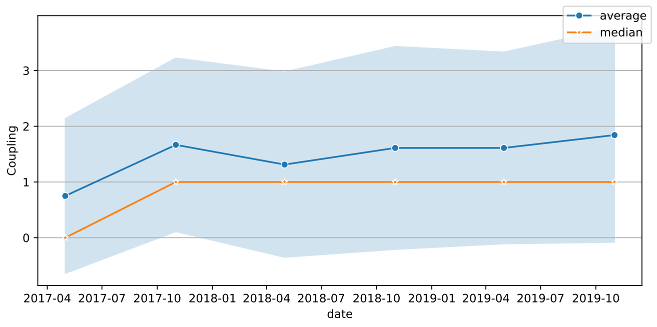

Fig. 6. Afferent coupling over time

E. Corrective Maintenance Analysis

Figure 7 shows a boxplot of repair time distribution grouped by year. This metric was computed by measuring the time from the bug report, *i.e.*, the time in which an issue labeled "bug" was created in GitHub, until the time such issue was closed. These measurements were grouped by the year in which the bug was closed. Figure 7 depicts that the upper quartile steadily grew from 2016 until 2019, while the lower quartile and the median grew from 2016 to 2017, then they barely changed in 2018 and grew again in 2019.

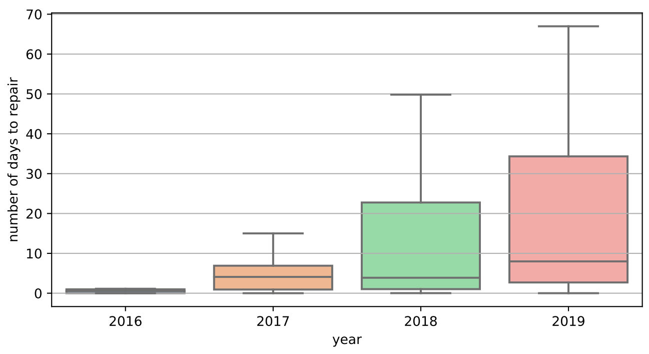

Fig. 7. Repair time distribution *per* year

One of the reasons for the growing number of days to repair bugs was a surge in the number of bug reports, which slowed down the bug-fixing rate. The number of bugs reported in 2016, 2017, 2018, and 2019 were 18, 91, 202, and 162, respectively. From the year that the number of bug reports was the lowest to its peak in 2018, it was more than ten times growth. In 2018, new end-users had their access granted to the prototype, and new datasets were explored. Some of these datasets revealed unexpected scenarios of usage of the prototype. For instance, the prototype was not handling a partially complete seismic horizon by the beginning of 2018. In this case, the requirement was modified and implemented. These unexpected scenarios caused many bugs or a change that inserted a new bug in the prototype.

Figure 8 shows the number of bug-fixes *per* KLOC, from 2016 until 2019. This metric is based on defect density metric [29], and it is computed by counting the number of bug fixes divided by the number of KLOC. Bug-fixes were counted by extracting the date when an issue that is labeled "bug" was closed. We also counted by analyzing commit messages that have terms like"bug fix" or "fixed", because not all researchers have the behavior of creating issues for each bug that they are fixing. Defect density doubled from 2016 to 2017, then increased 20% in 2018 and fell 9% in 2019.

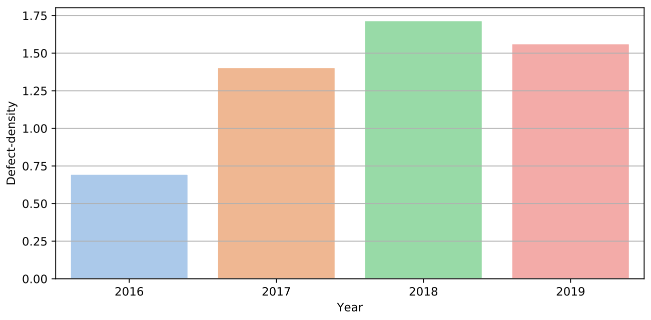

Fig. 8. Defect density *per* year

In 2017, eight microservices were created and integrated with other microservices. Thus, the number of bugs and bug-fixes increased proportionally faster than the LOC. In 2018, new end-users started using the prototype, so the number of bug-fixes increased faster than the number of LOC. In 2019, late feature implementations were responsible for the number of LOC growing faster than the number of bug-fixes, which remain stable.

One of Lehman's evolution laws, the "Declining Quality" law, states that adding new functionality to a system inevitably introduces new bugs [31]. The prototype seems to be following this law, because as the implementation of new features gradually increased, defect density metrics gradually increased despite a minor fall in 2019. Since it is complex to measure how many new functionalities were introduced, evidence of the growth of functionalities is the average number of LOC (Figure 2) and the average number of operations (Figure 3), which explains why defect density gradually increased.

F. Threats to Validity

The set of metrics collected in this study might not be a comprehensive representation of microservice maintenance. To minimize this threat, we ground the work in literature. Size and coupling metrics are considered applicable to microservices as maintenance metrics [7]. Corrective maintenance metrics (repair time [32] and defect density [29]) were extensively used in empirical studies as a maintenance measure, but not necessarily the maintenance of microservices.

Some metrics were manually collected (*e.g.*, number of endpoints), which is an error-prone task. Two of the authors collected the data, or an author executed it twice to minimize that threat. Furthermore, some bugs might not have been linked to the reports, the so-called bugs incognito [32], [29]. We reduced this threat by implementing a tool that detects a bug-fix message in the description.

LOC is an important measure that was used for many analyses in this work (*e.g.*, changes *per* LOC, bugs *per* LOC). Then, to minimize the threat of collecting invalid data, we validated our GHE mining tool against CLOC[11], a tool that counts LOC in many programming languages. We compared 37% of all commits (5641 out of 15408) selected by our GHE mining tool. For each one of these commits, we counted the number of LOC using CLOC and subtract it from the number of LOC computed using our GHE mining tool. The 1st percentile is -128 and the 99th percentile is 60, that is, 98% of all subtractions have an absolute value that is less than or equal to 128 LOC. We measured the percentage difference between the number of LOC computed by each tool and the maximum value is 6%, which evidence that 128 is relatively small compared to the size of the microservices.

V. Lessons Learned

A. Independent Development and Deployment

Independent deployment is considered a characteristic of Microservices Architecture that supports the independent development of microservices [5]. Some of the teams involved in the project have a specific research agenda that had to be completed before starting developing microservices (Section III). Each of these teams conducted the experiments at different speeds due to many reasons, such as the nature of the experiments (more time consuming), the number of team members, etc. However, this lack of synchronization did not hamper the development of the prototype because each team could develop the microservices independently from the others. Eighteen months separate the first microservice, which was created in December 2015, and the last microservice, which was implemented in July 2018 (Figure 1). The "Deploying a service as a container" pattern [23] (Section III) supports polyglot implementation, which was observed in this project. One of the microservices of this project implements a statistical model that correlates seismic traces and well properties. It was initially implemented using R, and later re-implemented using Python. Another microservice parses a textual knowledge base to extract information on the oil and gas domain using prolog. Other implements computer vision and machine learning techniques in Python to analyze seismic facies. These examples demonstrate that teams have chosen the technology they considered the best to address the problem.

Lessons learned: (i) ***Independent deploy*** allowed the teams to conduct their experiments, and develop microservices at their own pace without hampering the development of other microservices; (ii) ***Multiple paradigms*** allowed the teams to use the best technology for each problem, which is very diverse in the research area.

B. Shared database trade-offs

Despite creating three different environments for the prototype, some of its microservices have access to the same database instance (Section III-B). Database sharing among multiple microservices has been reported in the literature [33], but it is often considered a bad smell [34]. The reasons for this architecture choice was due to: (i) a single dataset is composed by a seismic, horizons, and wells, which can add up to 1Tb; and, (ii) dozens of datasets are stored into the local environment and replicated in the production environment. If these data were also replicated into multiple databases, one for each microservice, then it would overload our computing resources available for developing the prototype. Another alternative would be to create a microservice for handling seismic data as described by the database wrapping pattern [35]. However, in this way, the microservice would not be modeled around a "business capability" according to its principle (Section II).

Another issue related to database sharing among multiple microservices is that having a single database *per* service supports the independence of the microservice and its development team. As we adopted a NoSQL database, evolving database schema was not particularly difficult. Furthermore, we also partially chose a solution proposed by Taibi and Lenarduzzi [34], which is to "use a shared database with a set of private tables for each service that can be accessed by only that service". Some collections are accessed by a single microservice, and when another microservice needs data from

[11]https://github.com/AlDanial/cloc

these collections, it calls the microservice that provide access to those collections.

Lessons learned: (i) ***Database sharing*** is a valid approach for Microservices Architectures that work with large datasets manipulated by multiple services; (ii) ***Explicit assignment*** of which parts of the database schema are accessed by each service helps reduce the coupling generated by using a shared database; (iii) ***NoSQL database*** technologies help reduce the burden of evolving the schema of a shared database.

C. Service Cutting

The standard deviation of most of the size metrics (Section IV-C) was significant, which indicates two distinct patterns for handling changes in requirements: (i) keeping adding new functionalities to existing microservices; (ii) creating new microservices to implement new functionalities. On the one hand, adding functionalities to existing microservices speeds up development as it allows developers to reuse internal code (*e.g.*, code for accessing the database) and eliminates the need for creating new infrastructure code, such as Docker images, Kubernetes configuration files, and Jenkins setup, for each new microservice. Furthermore, teams can focus on the development of new features and figure out the boundaries of microservices later on. The team of one of the largest microservices decided to split it into two after two years and a half of the creation of the microservice. On the other hand, this pattern of adding functionalities to existing microservices led to large microservices that are difficult to maintain, which is known as the mega-service anti-pattern [34]. At the end of 2019, the three largest microservices were responsible for around 58% of the total number of LOC, 40% of all operations, and their median afferent coupling was 5, while the median afferent coupling of the remainder 16 microservices was 2. Creating new microservices to implement new functionalities makes them more cohesive and focused as it is recommended by the single responsibility principle [36].

Lessons learned: (i) ***Microservices can become mega-services*** if the addition of new features is not carefully planned and the maximum scope of features supported by each service is not clearly defined from the beginning of the project; (ii) ***Mega-services can be decomposed*** into microservices when necessary if their internal design provides clear isolation between business logic and external communication.

VI. Related Work

This section presents related work and compares them against this paper in the end.

Hasselbring and Steinacker [37] reported a refactoring of the implementation of an e-commerce software from scratch to a Microservices Architecture. The drivers were non-functional requirements, namely scalability, performance, and fault tolerance. The refactored software was decomposed in a way that microservices share neither code nor database. They also employed Conway's Law [20] by separating teams according to the Microservices Architecture. The impact of the microservices adoption was analyzed by the number of deployments *per* week and the number of incidents *per* week, from 2014 until 2017. While the number of deployment increased from 40 to more than 500 *per* week, the number of incidents remained at the same level. The authors concluded that microservices supported scalability, agility, and reliability.

Luz *et al.* [38] reported the adoption of microservices on three public institutions. The authors conducted a self-observation study, where the observer participated in the study, and a survey with the study participants. They concluded that microservices bring three main benefits: (i) innovation because they allow polyglot programming and new development models (*e.g.*, hackathons); (ii) time-to-market as microservices allow DevOps practices; (iii) motivation of development teams, because it promotes team independence to choose the technology (*e.g.*, programming language, framework) for a particular microservice.

Sampaio *et al.* [39] proposed a model for microservices and their evolution. This model is divided into three layers: (i) the architecture layer represents the topology of a microservices-based application; (ii) the instance layer represents the replicas, upgrades, and message flow; (iii) the infrastructure layer represents deployment-related characteristics, such as environment and host. They also implemented a prototype to assess the feasibility of the proposed model.

Balalaie *et al.* [40] reported a migration from an existing application to a Microservices Architecture to support reuse, data decentralization, scalability, and automated deployment. The authors listed a set of migration steps that were followed, and they highlighted the importance of continuous delivery in the process. A lesson learned in this migration was the difficulty of deploying in the development environment because of microservices dependencies. Despite the benefits of microservices architecture, they say that the adoption of this architectural style introduced many complexities that would be easier to address using other architectural styles.

Azevedo *et al.* [41] presented a system architecture developed using microservices and polyglot persistence technologies to handle the requirements of geological data in Oil & Gas domain. They present the benefits and drawbacks of the proposed solution and the lessons learned during its development encompassing development process, database design, database partition and integration, deployment, microservices contracts, implementation, testing, and debugging.

None of these works collected maintenance metrics for service-based systems of real-world scientific software, and they did not analyze how microservices deal with adaptive and extending maintenance scenarios.

VII. Conclusion

This work investigated to what extent Microservices Architecture support maintenance through an empirical study of a real-world scientific project. The results have shown evidence that Microservices Architecture supports maintenance because maintenance metrics analyzed in this study were not significantly changed after almost four years of adaptive and extending maintenance scenarios, particularly when we analyze the median of these metrics that is less influenced by outliers. However, further empirical studies on the maintenance of microservices are necessary to generalize this conclusion.

The presented lessons learned discussed trade-offs regarding principles of Microservices Architecture that are useful for practitioners and software engineering researchers that want to explore this subject from the maintenance perspective.

Another contribution of this paper is a GHE mining tool that has been made available and can foster the replication of this study or new investigations involving microservices and software repository mining.

As future work, we plan to conduct empirical studies on the maintenance of Microservices Architectures that involve more than one target application to analyze the results of maintenance metrics of applications from other domains. Furthermore, we are also considering including network communication among microservices and the number of deployments as metrics to support our analysis.

REFERENCES

[1] C. Watson, S. Emmons, and B. Gregg, "A microscope on microservices," 2015, available from: URL: https://netflixtechblog.com/a-microscope-on-microservices-923b906103f4. Last accessed: May/2020.

[2] S. M. Fulton III, "What led Amazon to its own microservices architecture," 2015, available from: URL: https://thenewstack.io/led-amazon-microservices-architecture/. Last accessed: May/2020.

[3] S. J. Fowler, *Production-Ready Microservices. Building Standardized Systems Across an Engineering Organization*. O'Reilly, 2016.

[4] M. Fowler and J. Lewis, "Microservices," 2014, available from: URL:https://martinfowler.com/articles/microservices.html. Last accessed: May/2020.

[5] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2015.

[6] N. Ford, R. Parsons, and P. Kua, *Building Evolutionary Architectures*. O'Reilly Media, Inc., 2017.

[7] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically Measuring the Maintainability of Service- and Microservice-based Systems: A Literature Review," in *International Workshop on Software Measurement and International Conference on Software Process and Product Measurement*. New York, NY, USA: ACM, 2017, pp. 107–115.

[8] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *International Conference on Software Maintenance*, 2009, pp. 51–60.

[9] J. Lewis, "Microservices: a definition of this new architectural term," 2014, available from: URL: http://martinfowler.com/articles/microservices.html. Last accessed: May/2020.

[10] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[11] R. Kazman, M. Klein, and P. Clements, *Evaluating Software Architectures — Methods and Case Studies*. Addison-Wesley, 2001.

[12] A. Mehta and G. T. Heineman, "Evolving legacy system features into fine-grained components," in *Managing Corporate Information Systems Evolution and Maintenance*. IGI Global, 2005, pp. 108–137.

[13] E. Hewitt, *Java SOA Cookbook: SOA Implementation Recipes, Tips, and Techniques*. " O'Reilly Media, Inc.", 2009.

[14] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[15] M. Hüttermann, *DevOps for developers*. Apress, 2012.

[16] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Science & Business Media, 2014.

[17] R. Vaculin and K. Sycara, "Specifying and monitoring composite events for semantic web services," in *European Conference on Web Services*, 2007, pp. 87–96.

[18] D. Wampler and T. Clark, "Multiparadigm programming: guest editors' introduction." *IEEE Software*, vol. 27, no. 5, pp. 2–7, 2010.

[19] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013.

[20] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[21] T. Storer, "Bridging the chasm: A survey of software engineering practice in scientific programming," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 1–32, 2017.

[22] J. Segal, "Scientists and software engineers: A tale of two cultures," in *Proceedings of the Psychology of Programming Interest Group*, 2008.

[23] C. Richardson, *Microservices Patterns*. Manning publications, 2018.

[24] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[25] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *International Symposium on Performance Analysis of Systems and Software*, 2015, pp. 171–172.

[26] Hightower, K. and Burns, B. and Beda, J., *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, 2017.

[27] Dmytro Rud and Andreas Schmietendorf and Reiner R. Dumke, "Product Metrics for Service-Oriented Infrastructures," in *International Workshop on Software Measurement*, 2006.

[28] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling metrics for predicting maintainability in service-oriented designs," in *Australian Software Engineering Conference*, 2007, pp. 329–340.

[29] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, Jul. 2002.

[30] G. Schermann, J. Cito, and P. Leitner, "All the services large and micro: Revisiting industrial practice in services computing," in *International Workshop On Engineering Service-oriented Applications*. Berlin, Heidelberg: Springer-Verlag, 2015, p. 3647.

[31] M. M. Lehman, D. E. Perry, and J. F. Ramil, "On evidence supporting the FEAST hypothesis and the laws of software evolution," in *Proceedings Fifth International Software Metrics Symposium*, 1998, pp. 84–88.

[32] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links," in *SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM Press, 2010, pp. 97–106.

[33] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," in *International Conference on Cloud Computing and Services Science*, 2018.

[34] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[35] S. Newman, *Monolith to Microservices*. O'Reilly Media, Inc., 2019.

[36] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003.

[37] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *International Conference on Software Architecture Workshops*, 2017, pp. 243–246.

[38] 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 *Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2018, pp. 32–41.

[39] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, "Supporting Microservice Evolution," in *International Conference on Software Maintenance and Evolution*, 2017, pp. 539–543.

[40] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report," in *Advances in Service-Oriented and Cloud Computing*, A. Celesti and P. Leitner, Eds. Cham: Springer International Publishing, 2016, pp. 201–215.

[41] L. G. Azevedo, R. d. S. Ferreira, V. T. d. Silva, M. de Bayser, E. F. d. S. Soares, and R. M. Thiago, "Geological data access on a polyglot database using a service architecture," in *Brazilian Symposium on Software Components, Architectures, and Reuse*, 2019, pp. 103–112.