DevOps and Microservices in Scientific System development: experience on a multi-year industry research project

Maximillien de Bayser, Vinícius Segura, Leonardo G. Azevedo
IBM Research
Rio de Janeiro, Brazil
{mbayser,vboas,lga}@br.ibm.com

Leonardo P. Tizzei
IBM Research
São Paulo, Brazil
ltizzei@br.ibm.com

Raphael Thiago, Elton Soares, Renato Cerqueira
IBM Research
Rio de Janeiro, Brazil
{raphaelt,rcerq}@br.ibm.com
eltons@ibm.com

ABSTRACT

In this work, we present the experience of a multi-year industry research project where agile methods, microservices and DevOps were applied. Our goal is to validate the hypothesis that the use of microservices would allow computational scientists to work in the more minimalistic prototype-oriented way they prefer while the software engineering team handle the integration. Hence, scientific multidisciplinary systems gain in a twofold way: (i) Subject Matter Experts (SME) use their preferable tools to develop the specific scientific part of the system; (ii) software engineers provide the high quality software code for the system delivery.

CCS CONCEPTS

• **Software and its engineering** → **Agile software development**; **Software prototyping**; **Programming teams**;

KEYWORDS

Microservices Architecture, DevOps, Modern Software Engineering, Scientific Computing, Applied Research

ACM Reference Format:
Maximillien de Bayser, Vinícius Segura, Leonardo G. Azevedo, Leonardo P. Tizzei, and Raphael Thiago, Elton Soares, Renato Cerqueira . 2022. DevOps and Microservices in Scientific System development: experience on a multi-year industry research project. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3477314.3507317

1 INTRODUCTION

Software engineering has adopted iterative approaches like Agile to mitigate the risk that software is found with real-world requirements after years of development. Several studies applying agile methodologies to scientific computing have been documented [1, 17]. Practices such as continuous integration (`CI`) foster an experimental approach to development where small incremental changes are tested under real conditions with staging, A/B testing, and extensive monitoring [15].

The dynamism of development and stability required for operation generates conflicts among areas, teams, and goals, demanding changes in people mentality and reasoning about the impacts in organizational work culture [8]. Therefore, tools and methods are required to meet the needs of scientists and engineers in scientific system development.

In this work, we explore the development of a scientific system applying recent software engineering techniques such as microservices [7] supported by `DevOps` [10] practices and `as-a-service` delivery. We support our findings on our experience of applying these trends from the start to a large applied research project involving a multidisciplinary team of more than 30 people over three years. In this project, in the domain of petroleum geology and geophysics, several microservices were developed by small teams composed of subject matter experts (SMEs) and developers. `DevOps`, `CI` and user interface were handled by our software engineering staff. We nicknamed the overall methodology "ResearchOps" [4].

It is a known property of microservices that they promote a different trade-off in the complexity of the overall system where the individual components become simple, and the integration of parts becomes more complex [14, 18]. The hypothesis that we wanted to test in this project was that this trade-off would allow us to relieve SMEs to some degree from the burden of dealing with software complexity and shifting it to the software engineering staff. The SME would use the programming language most familiar to scientists in her domain of expertise, and her work would be integrated into the application using standard interfaces. In other words, we expected that microservices would allow us to re-distribute the complexity to minimize the overall difficulty.

This paper is organized as follows. Section 2 presents the project development. Section 3 presents the quantitative and qualitative metrics that we have gathered about the prototype and the teams that implemented it. Finally, Section 4 presents our conclusions.

2 THE PROJECT DEVELOPMENT

The project object of this study is a multi-year joint research agreement between our research institute and an Oil&Gas company. On the client side, we collaborate with a team of geologists and geophysicists. On our side, we have experts in fields such as optimization, computer vision, machine learning, and visualization, supported by interns and software engineering staff. The project integrates research in several different aspects of petroleum geophysics. Each aspect has an assigned team that works quite independently.

The client SMEs do not directly participate in software development but guide the research and evaluate results. The system is delivered to them as-a-service prototype with a web-based user interface. We effectively employ `CI` to allow quick iterations.

Architecture: The project is developed as a microservice architecture that comprises independently developed services that work together to achieve a specific purpose [14].

The general architectural pattern for our microservices is to expose a RESTful [6] HTTP interface. The advantages of using HTTP as a communication protocol are language independence, human-readable, and tooling. The downsides are: text-based requires client and server spend CPU cycles converting data to text; latency due to HTTP handshake communication; bandwidth if not using compression; and, request-response only interaction pattern.

All microservices are written in `Python`, `Javascript` or `Java`. Horizontal concerns like request authorization, logging, encryption (Transport Layer Encryption[1]), load-balancing, and caching are all implemented by putting a `nginx` server as a reverse proxy in front of these microservices. Besides the advantage of uniform request logging and centralized security, this approach is one of the possible complexity trade-offs that simplify the implementation of microservices and transfer the burden to the integration team.

In several instances, individual microservices are too fine-grained for the `UI` to consume. So, we use *API gateways* pattern [14], composing services into a larger one that is called by the `UI`.

Some microservices execute numerical algorithms on large seismic data that could take days to run, which interaction cannot be handled by the `HTTP` protocol. In such a case, we return immediately a token that identifies the operation and start the operation in a background process. The token can then be used to query the task's state and obtain results when ready.

We have a data processing pipeline where pieces of data are submitted to a queue where they are picked up by services that extract and transform them and submit their results to other queues. Some services take the data off a queue and store it in a database, thus ending the pipeline.

Infrastructure: We allocate and maintain the infrastructure and develop a deployment strategy. The adoption of container technology and `docker` [13] in particular bring us many benefits. All the `Dockerfiles` are written by our software engineering team, and we write them in ways that allow the software to be configured by changing environment variables or overriding configuration files at run time. However, we have to work closely with the microservices teams to remove any hard-coded locations of files and services. For several months we used a tool called `docker-compose` to orchestrate our containers. It was convenient to start or stop the entire prototype with a single command. Another helpful feature is allocating a private virtual network on the host machine so that all containers run inside a single segregated addressing space. This virtual network is convenient when we start to run several environments simultaneously. `Kubernetes` [3, 9] is the container orchestration we chose to scale the project from a single workstation to a small server farm.

DevOps and CI: DevOps (a portmanteau of "development" and "operations") is a software development method that extends the agile philosophy to produce software products and services in a faster way and to improve operations performance and quality assurance [10]. DevOps aims at a continuous delivery pipeline where new features are automatically tested in the correct environment and then approved for production.

We embrace the *Infrastructure as Code* (`IaC`) [16], and we base it on the `git` distributed version control tool, both on the microservice and the integration level.

We have three standard branches in microservice repositories: `master`, `stable` and `prod`. All these branches contain `Dockerfiles` to build images from them. We configure a `Jenkins CI` server to watch these branches for changes. Every time a change is detected, a `Jenkins` job checks out the latest version, builds the container image, and submits it to a central registry.

At the integration level, we have a `kubernetes` configurations repository. We have a `Python` script around the `kubectl CLI` that sends the declarative `YAML` configuration files to the `kube-api-server`, creates `configmaps` from configuration files and, ensures that the services restart if the configuration changes. It also supports environment-specific configurations by leveraging the `Jinja2` templating library to bind configuration variables, but in general we tried to avoid differences between versions as much as possible.

As an advantage, this scheme tracks all changes to the global state of the prototype. It puts publishing changes decision to any environment entirely in the hands of the microservice developers. Nevertheless, this scheme is probably only applicable when all teams are part of the same organization, as is our case.

Despite the successful application of `CI` to program code, the `CI` of data and data *schemata* is still a challenge. For instance, apply changes automatically in ontologies stored as `RDF`[2]. Although the ontologies are modeled using `OWL`[3] and despite the fact that these files are also stored in a git repository following the same branch scheme, we have not been able yet to develop a way to apply ontology changes automatically.

Logging, Monitoring & Debugging: Seven accepted techniques can help in reducing errors in distributed systems [2]: testing, model checking, theorem proving, record and replay, tracing, log analysis, and visualization. Apart from automated testing that some of our microservices implement in their `CI` pipelines, we employ record and replay, some tracing, log analysis, and visualization.

The foundation of our strategy is to record application logs and performance metrics on all servers in a continuous way. We accumulate these logs and metrics in a database, `elasticsearch`, and make them available for searching in the form of graphs in a web-based user interface.

3 MIXED-METHOD STUDY

This study aims to investigate the effects of DevOps practices and tools on the development of a large scientific information system. We conducted a mixed-method study composed of a questionnaire, static source code repository analysis, and some usage metrics.

Qualitative survey: We applied a self-administered questionnaire, which is a case-control study [12] since we asked "participants about their previous circumstances to help explain a current

[1]https://tools.ietf.org/html/rfc8446

[2]https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[3]https://www.w3.org/TR/owl2-overview/

phenomenon". In this particular case, we asked them about their background, their knowledge of software engineering practices and tools, focusing on DevOps and microservices. The response rate was 18 out of 29 members (~62%). The questionnaire was not sent to the SMEs of the client company.

From the 18 respondents: 15 declared having a computer science background; 3 had formal training in design. In the computer scientists group: 2 declared software engineering specialist; 12 rated their software engineering expertise as high or above average. The other is specialized in a diverse set of areas like human-computer interaction, databases, computer graphics, multimedia, and computer vision. Only six respondents reported having worked with more than 50K LoCs. Eight have experience with agile methodologies.

On a scale of 1 to 5, 15 people chose 3 or above to rate their involvement with development. They were the same 15 people with a computer science background. We call them as *developers*. Thirteen developers had worked with agile methodologies and know git. Concerning the number of developers with experience in subjects: cloud computing (4); client-server programming (8); computer network (4); docker or virtual machine (7).

Most developers would not split up microservices into smaller ones or otherwise rearrange the microservices. Four declared to have difficulty remembering which service was called by others. Four thought that some APIs had poor encapsulation. Nine declared that internal change in the code of called services could break their own. Six declared being able to draw an accurate diagram of the service logical architecture.

Seven developers built a docker image of their service during the project, showing that the CI automation was considered adequate. Three declared to be able to draw an accurate infrastructure diagram. Five reported having made significant changes to the IaC code.

Eight developers had automated testing, reflecting the difficulty in testing scientific code as reported in the literature.

Seven developers declared that their code relied heavily on calls to other services. Four considered the dependency to be strongly incidental. Nine declared they would be able to setup everything needed to run their code. Three reported relying heavily on asking others to make progress in their work. 2 (from the UI team) declared to be blocked in their work by malfunctioning of services daily.

Eleven had participated in debugging activity and valued application logs for this activity. Nine considered debugging harder with microservices than monolithic code, and five felt it much harder. Seven used logging libraries or have a consistent methodology for print statements. Nine debuggers used the log searching tool, and of those, 2 had trouble using them or considered them insufficient.

Source code metrics: The source code metrics provide an overview of the project implementation. Table 1 shows the size in KLOC (thousands of lines of code) and implementation languages for each service. MS-* are RESTFul microservices and P-* are services based on the Kafka message queue. Services were implemented using different programming languages, but Python is the most commonly used due to developers' know-how.

Microservices are language agnostic because services communicate with each other via lightweight mechanisms [14]. MS-5, which is heavily based on statistics, has its core implemented in R and the P-7 service was implemented using PROLOG and LISP, which are common in the artificial intelligence domain. The last line accounts for all IaC (Infrastructure as a Code) implementation, including kubernetes configuration, Dockerfiles and service configuration files. Notably it represents a mere 3% of the overall LoCs, although the CI configuration in the Jenkins is not included.

Table 1: Service's size and language

Service	KLOC	Language
UI	52.3	Typescript, CSS, HTML
MS-1	39.3	python
MS-2	40.1	python
MS-3	16.4	python
MS-4	11.9	python
MS-5	6.1	python, R
MS-6	0.8	python
MS-7	3.2	python, Java
MS-8	8.8	Javascript
P-1	0.8	python
P-2	0.5	Java
P-3	0.6	Java
P-4	2.5	python, Java
P-5	0.3	python
P-6	3.5	python
P-7	5.6	python, lisp, prolog
IaC	7.1	yaml, other config lang.

The size of the service can vary two orders of magnitude, from a few hundred LOC to dozens of thousands LOC, even when they are implemented using the same language. Some services are inherently more complex than others due to the many features provided.

To assess to what extent CI tools have influenced software engineers, we measured the impact of introducing Jenkins to their daily work routine. We analyzed the Git logs of services implementation that started before Jenkins was made available. Jenkins availability started on the 1st of June 2016 and the Git logs that were analyzed since the first commit on 29th December 2015 until 5th of September 2018. Based on these Git logs, we selected the commits of 10 software engineers, which were the only that had worked on this project before and after Jenkins' availability. We counted the total number of commits per week of these selected software engineers. Then, we divided these total commits per week by the number of software engineers who have committed at least once in the week to select only those actively working on the project and remove those on vacation and those who left the project.

Figure 1 shows a boxplot of the average number of commits per week of the selected software engineers. The average number of commits increased from a 5.4 to 11.3 median after Jenkins, which introduction facilitated the deployment of the services on multiple environments. Then, when software engineers finish implementing new features or bug fixes, they can quickly deploy on a development environment to perform integration testing.

Three services implementation started before Jenkins availability, namely MS-1, MS-2, and MS-3. Figure 2 shows that the development of two of them was more intense, in terms of lines of code (LOC), before Jenkins was available than after it. Thus, it was not the development intensity that influenced the number of commits.

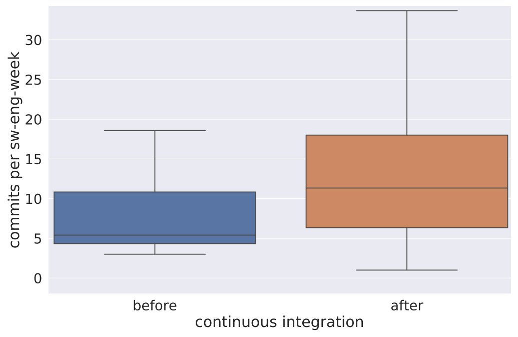

Figure 1: Average commits per software engineer-week

These results are similar to the perceived benefits of DevOps tools observed in other studies, such as Itkonen *et al.* [11].

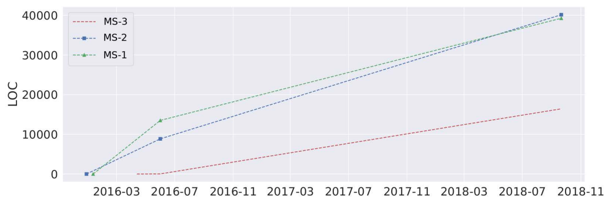

Figure 2: LOC evolution per service before and after the availability of CI tool on June 2016

The prototype had a total of 16 SME users who would test the prototype. Every weekday, on average, 3 of them accessed the prototype. For three months, we measured a total of over 3 million HTTP requests, of which 72% were completed in under 10ms (as measured by the nginx reverse proxy, not including user network latency). In the same period, we recorded over 5000 background task execution, the longest of which took almost seven days, although only 10 took more than 24 hours.

Limitations of study: The authors were also developers of the project and also answered the questionnaire. Since the authors account for about 20% of the respondents, we thought excluding their perspective would impact the validity even more by reducing the sample size and excluding a large part of the people with a software engineering background.

4 CONCLUSION

In scientific information systems development, scientists (SME) and software engineers should work close together to create the multidisciplinary scientific code having high quality of delivery. We presented our practical experience towards bridging the gap in scientific information systems development concerning the use of modern software engineering in scientific computing.

We aimed to validate the hypothesis that the use of microservices and DevOps would allow computational scientists to work in the more minimalistic prototype-oriented way they prefer. At the same time, the software engineering team would handle the integration.

We demonstrated that the adoption of CI and as-a-service delivery has positively contributed to this success, based on previous industry projects where we did not apply these techniques[4]. In these other projects, the difficulty of delivering new developments reduced the rate at which client SMEs could test and validate the software, thereby reducing the speed of iterations.

The CI pipeline also made the overall process easier since people were relieved of deploying their software manually. In some previous projects, different teams would manually install their software on a shared machine for integration. There was no tracking of what was installed and the steps to install it, making any update a risky activity because one could inadvertently break things and not know how to revert to a "sane" state. In contrast, now that almost anything is tracked and reversible, the barrier to test new commits is much lower. The survey that we conducted also showed that only a tiny part of the overall team needs to have expert knowledge of CI and DevOps for these methodologies to work.

REFERENCES

[1] Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. 2008. Scientific Software Development at a Research Facility. *IEEE Software* 25, 4 (2008), 44–51.

[2] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging Distributed Systems. *Commun. ACM* 59, 8 (2016), 32–37.

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1, Article 10 (Jan. 2016), 24 pages.

[4] Maximilien de Bayser, Leonardo Azevedo, and Renato Cerqueira. 2015. ResearchOps: The Case for DevOps in Scientific Applications. In *Proceedings of the 10th International Workshop on Business-driven IT Management (BDIM 2015)*. IFIP/IEEE IM 2015, IEEE, Otawa, Canada, 1398–1404.

[5] Maximillien de Bayser, Vinícius Segura, Leonardo G. Azevedo, Leonardo P. Tizzei, Raphael Thiago, Elton Soares, and Renato Cerqueira. 2021. DevOps and Microservices in Scientific System development: experience on a multi-year industry research project. In *arXiv.org*. Cornell University, arxiv.org, 1–14.

[6] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures.* Ph.D. Dissertation. Univ. of California.

[7] Martin Fowler and James Lewis. 2014. Microservices. https://martinfowler.com/articles/microservices.html. Accessed on: Feb-2021.

[8] Paulo J. A. Gimenez and Gleison Santos. 2020. DevOps Maturity Diagnosis–A Case Study in Two Public Organizations. In *XVI Brazilian Symposium on Information Systems (SBSI 2020)*. SBC, São Bernardo do Campo, São Paulo, Brazil, 1–8.

[9] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. *Kubernetes: Up and Running Dive into the Future of Infrastructure* (1st ed.). O'Reilly Media, Inc., Sebastopol, CA.

[10] Michael Hüttermann. 2012. *DevOps for Developers*. Vol. 1. Springer, New Your, NY.

[11] Juha Itkonen, Raoul Udd, Casper Lassenius, and Timo Lehtonen. 2016. Perceived Benefits of Adopting Continuous Delivery Practices. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '16)*. ACM, New York, NY, USA, 42:1–42:6.

[12] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of Survey Research Part 2: Designing a Survey. *SIGSOFT Softw. Eng. Notes* 27, 1 (2002), 18–20.

[13] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[14] Sam Newman. 2015. *Building microservices*. O'Reilly Media, Inc., Sebastopol, CA.

[15] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous Deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 21–30.

[16] D. Spinellis. 2012. Don't Install Software by Hand. *IEEE Software* 29, 4 (2012), 86–87.

[17] William A. Wood and William L. Kleb. 2003. Exploring XP for Scientific Research. *IEEE Softw.* 20, 3 (2003), 30–36.

[18] Olaf Zimmermann. 2017. Microservices tenets. *Computer Science-Research and Development* 32, 3 (2017), 301–310.

[4]De Bayser *et al.* present more details about the project, the qualitative survey, the questionnaire, static source code repository analysis, and usage metrics [5]