Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Assuring the Evolvability of Legacy Systems in Devops transformation/adoption: Insights of an experience report

Autor: Álax de Carvalho Alves
Orientador: Carla Silva Rocha Aguiar

Brasília, DF
2020

Álax de Carvalho Alves

Assuring the Evolvability of Legacy Systems in Devops transformation/adoption: Insights of an experience report

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Carla Silva Rocha Aguiar

Brasília, DF

2020

Álax de Carvalho Alves

Assuring the Evolvability of Legacy Systems in Devops transformation/adoption: Insights of an experience report/ Álax de Carvalho Alves. – Brasília, DF, 2020-

43 p. : il. (algumas color.) ; 30 cm.

Orientador: Carla Silva Rocha Aguiar

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2020.

1. DevOps. 2. Refactoring. I. Carla Silva Rocha Aguiar. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Assuring the Evolvability of Legacy Systems in Devops transformation/adoption: Insights of an experience report

CDU 02:141:005.6

Álax de Carvalho Alves

Assuring the Evolvability of Legacy Systems in Devops transformation/adoption: Insights of an experience report

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 18 de dezembro de 2020 – Data da aprovação do trabalho:

Brasília, DF
2020

Resumo

DevOps has changed the software industry to enable continuous delivery. While many studies have investigated on how to introduce DevOps into a software product from the organizational perspective, less is known about the technical challenges developers and practitioners face when transforming legacy codes into DevOps, despite the undisputed importance of this topic. In this paper, throughout the context of web applications, we report the results of a study case with the adoption of four legacy open-source projects into DevOps to understand which refactoring techniques and strategies influence developers' decisions. We analyze two dependent variables: the technique used and how they are applied to the project. After every implementation, there was an overview of the process that just occurred and later a written report on how the strategies have been applied, their respective order, which strategy has been more fruitful, and such. Those reports have been the foundation of this study. The main findings of such study are that some strategies are more efficient when viewed from the evolution aspect and the sequence these techniques are employed matter.

Key-Words: Devops, Refactoring, Program Comprehension, Experience Report, Guideline, Legacy.

Resumo

DevOps mudou a indústria de software para permitir a entrega contínua. Embora muitos estudos tenham investigado como introduzir DevOps em um produto de software do ponto de vista organizacional, menos se sabe sobre os desafios técnicos que os desenvolvedores e profissionais enfrentam ao transformar códigos legados em DevOps, apesar da importância indiscutível deste tópico. Neste artigo, através do contexto de aplicações web, relatamos os resultados de um estudo de caso com a adoção de quatrp projetos legados de código aberto em DevOps para entender quais técnicas e estratégias de refatoração influenciam as decisões dos desenvolvedores. Analisamos duas variáveis dependentes: a técnica usada e como são aplicadas ao projeto. Após cada implementação, havia uma visão geral do processo acabado de ocorrer e posteriormente um relatório escrito sobre como as estratégias foram aplicadas, sua respectiva ordem, qual estratégia foi mais vantajosa e afins. Esses relatórios foram a base deste estudo. As principais conclusões desse estudo são que algumas estratégias são mais eficientes quando vistas do ponto de vista da evolução e a sequência em que essas técnicas são empregadas importa.

Key-words: Devops, Refatoração, Compreensão de Programa, Relatório de Experiência, Guia, Legado.

Lista de tabelas

Lista de abreviaturas e siglas

DevOps	Development and Operations
OSS	Open Source Software
CI	Continuous Integration
CD	Continuous Deploy/Delivery
CT	Continuous Testing
SW	Software
WIP	Work In Progress
CMS	Content Management System
RSS	Rich Site Summary
VM	Virtual Machine

Sumário

1 Introduction and Motivation

DevOps combines cultural philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity: evolving and improving products faster than organizations using traditional software development and infrastructure management processes. This speed enables organizations to serve their customers better and compete more effectively in the market (AMAZON, 2020).

As more companies adopt DevOps to improve their workflow and productivity, many challenges related to the infrastructure and the legacy software systems have arisen. DevOps is about people and processes (LEITE et al., 2019). It is a methodology that enables organizational groups to communicate with each other across silos and to coordinate their activities. Thus, it is not surprising that established cultural habits are the number one challenge to DevOps, especially barriers to cross-organizational collaboration, the critical element of successful DevOps practice (INCORPORATED, 2014).

Legacy codes are usually characterized by the following: use of outdated frameworks, no test (neither unit nor integration), no containerization, no automation tools, nonexistence of technical documentation, monolithic architecture, no continuous integration, no automation at all.

The professionals working on a legacy system have to put in extra effort to refactor it, especially legacy systems. There are no strategies in place for implementation of refactoring techniques. No plans designed to guide the developers with the same. Developers performing with their own knowledge and sometimes ended up messing with the code (KHANAM, 2018).

Outdated legacy applications usually work tied to running a production service that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows (MURPHY, 2016). While cloud applications are needed for quick application delivery, some legacy systems cannot be integrated, leaving the IT infrastructure out of sync and incapable of operating at a fast pace (VASSIT, 2016). The solutions to such legacy challenges are often time-consuming, or costly (VASSIT, 2016). All this is stated still considering the web applications spectrum.

Consistently, the core teams introduce new features to frameworks and languages. Most of the time, these changes result in performance gains and overall reliability in terms of security - but not limited to it. Embracing such performance and security gains could result in a faster app, which is very motivational to upgrade.

Upgrading a particular app is not limited to the language or the framework itself.

That is made very clear when we acknowledge that many projects have been widely taking advantage of container-based architectures tools, such as Docker, to make contributing easier. Grasping such DevOps aspects are helpful to such upgrades since it is a growing concept in Open Source Software projects.

Evolvability is a vital software aspect, and fundamental to its existence. It motivates software engineering researches, and practices (RAJLICH, 2018). 'Evolution consists of repeated software changes. Defined software change processes lead to improved productivity and quality of software evolution' (RAJLICH, 2018).

Microservices is an important architectural style that prioritizes evolvability. Evolvability is especially crucial for software with frequently changing requirements, internet-based systems for instance. Software professionals apply a set of numerous activities that we refer to as evolvability assurance (BOGNER et al., 2014). These activities are usually of analytical nature to identify issues or a constructive nature to remediate issues. That includes techniques like code review or refactoring, standardization, guidelines, conscious technical debt management, and tools, metrics, or patterns (BOGNER et al., 2014).

Thus, in the appropriate context, migrating monolithic architectures to microservices could bring in many benefits including, but not limited to, flexibility to adapt to the technological changes in order to avoid technology lock-in, and more importantly, reduced time-to-market (BALALAIE; HEYDARNOORI; JAMSHID, 2016).

Legacy apps are typically harder to adopt DevOps due to a blend of technology, process and cultural issues. The cultural issue regard to the initial resistance from teams to move into DevOps ways is due to reluctance to change, emerging from an inertia of doing things a certain way for years if not decades. Most of these systems were not built for the agile workflows that focus on incremental and iterative deliveries. Amidst challenges like too much technical debt, tightly integrated hardware components, fragile codebase, it is tough to select specialised approach like DevOps.

While designing and developing a greenfield project, architects and developers start afresh and have the opportunity to take into consideration the requirements of DevOps (RAO, 2018). In case of legacy systems, which have evolved over a period of time without any consideration of automation, the adoption of the DevOps approach may result in large-scale refactoring or redesign. It may prove to be a significant challenge to automate the vast amount of legacy code and processes (RAO, 2018).

Considering tests and their automation, Legacy systems tend to have low code coverage due to few or no unit tests. Testing is typically done in higher environments and is manual. As more features are added to a legacy system, the manual testing effort increases drastically, eventually slowing down feature delivery. This problem is amplified when there are multiple teams working on the same code base (RAO, 2018).

Microservices is also observed as a growing concept (BALALAIE; HEYDARNOORI; JAMSHID, 2016). Although DevOps practices can also be used for monolith architecture, using them in the context of microservices enables practical implementation of DevOps (BALALAIE; HEYDARNOORI; JAMSHID, 2016).

In the context of an upgrade, microservices are strongly recommended because they bring up some desired advantages that can be very useful to certain context. Although we have a set of favorable concepts and tools at our disposal, there is still no standardized collection of steps to be followed; in other words, it has no pattern defined when upgrading software. Actually, it seems to be an empirical-like process.

This could represent a considerable challenge to teams when performing an upgrade, because since there is very-well defined tools and appropriate concepts, one could think that embracing any of those tools and concepts will result in a successful, or even a painless upgrade process.

The challenge is identifying which tool and concept are adequate to the context. As in any software process improvement initiative, the path to a successful DevOps implementation is unique to each organization. Still, it is possible to learn from challenges experienced during other process adoptions in order to plan future initiatives (SMEDS; NYBOM; PORRES, 2015).

In this work, we report the results of a study case with the adoption of four legacy open-source web projects into DevOps to understand which refactoring techniques and strategies influence developers' decisions. We map the refactoring techniques used, the sequence they were employed, the benefits perceived by the organization, and the challenges faced by developers when deploying each refactoring technique. We analyse the project repositories, the commits, the issues discussions, the communication channels. We present a set of lessons learned, with the DevOps benefits for each refactoring technique experimented, the impact of the order the techniques are employed from developers perspective and some guidelines for legacy projects aiming at adopting DevOps .

2 General Aspects

2.1 Related works

Although there have been several discussions on DevOps practices and how they benefit one's project, applying such practices to an existing project is often painful.

When it comes to refactoring legacy code with focus on DevOps, S.A.M. Rizvi and Zeba Khanam have proposed methodology (S.A.M.RIZVI; KHANAM, 2011). Their article proposes a methodology that can be employed to apply the refactoring activities on the legacy system, employing the aspect-oriented techniques. Considering the refactoring activities that are more likely to improve the software design and quality, the developers should adopt an approach that would focus on a restricted set of refactoring patterns. Thus allowing the developers to choose their desired set of strategies (S.A.M.RIZVI; KHANAM, 2011).

Gangadhar Hari Rao proposes a roadmap for implementing DevOps in a legacy software, with focus on building a CD pipeline with the supporting capabilities. With this roadmap Rao concludes that the successful adoption of the DevOps methodology for a legacy system is possible only if the teams working on legacy systems also change their processes and mindset towards Agile and CD. In each stage of the roadmap he considers the challenges and proposes actions to overcome them, always with a great focus on CD - what could involve great costs, depending on the legacy system to be considered (RAO, 2018).

Chia-Chu Chiang and Coskun Bayrak propose a refactoring strategy that consists of converting legacy systems into component-based systems. The process involves program understanding, business rules extraction, and software transformation. In their paper, they present a semi-automated program slicing technique for business rules extraction from legacy code and convert the reusable code into a component conforming to the protocols of a component interconnection model (CHIANG; BAYRAK, 2006).

Errickson-Connor (ERRICKSON-CONNOR, 2003) also proposed a strategy that consisted of steps of a software modernization process where a legacy code is transformed into new languages and new environments. She suggests that a legacy code needs to be cleaned up, such as removing program anomalies before being transformed. The next stage involves software restructuring tasks such as isolating business rules, identifying business rules, and extracting business rules as reusable services. When the code corresponding to a business rule is extracted, it is ready for transformation into components in stage three. The final stage is to manage these reusable components in a software environment.

Regarding implementing certain levels of DevOps in legacy software, the SmartSheet website ensures Virtualization and, consequently, Microservices as core practices. Working with small, reusable building blocks of code ensures that the application under development is not affected by the increase in deployments' velocity in the DevOps environment. Containers are the next evolutionary step in virtualization technology (SMARTSHEET, 2020).

Finally, Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles in their work A Survey of DevOps Concepts and Challenges outline a conceptual framework to guide engineers, managers, and academics in the exploration of DevOps tools, implications, and challenges. The conceptual framework is composed of conceptual maps, which are diagrams structured as graphs in which nodes depict concepts and arrows represent relationships among concepts(LEITE et al., 2019).

Their survey investigates the DevOps concepts and challenges from multiple perspectives: engineers, managers, and researchers. Also explores a much broader range of sources. More up-to-date concepts of DevOps and its tools are analyzed, categorized and correlates to the DevOps concepts, and discuss which roles in the organization should use which tools. It concludes by summarizing and discussing some of the main DevOps challenges (LEITE et al., 2019).

2.2 Background

2.2.1 DevOps - Practices and Strategies

DevOps is a software development and delivery process that helps in emphasizing communication along with cross-functional collaboration between product management, software development, and operations professionals. Also widely considered a collaborative and multidisciplinary organizational effort to automate continuous delivery of new software updates while guaranteeing their correctness and reliability (LEITE et al., 2019).

From the technical perspective, DevOps relies heavily on automation tools, including tools for container management, continuous integration, orchestration, monitoring, deployment, and testing (ZHU; BASS; CHAMPLIN-SCHARFF, 2016). Automated deployment pipelines and monitoring facilitate error detection. The micro-services architectural style is quickly becoming the standard for building continuously integrated and deployed systems. DevOps aims to achieve some business outcomes, such as reducing risk and cost, complying with regulations, and improving product quality and customer satisfaction (LEITE et al., 2019).

DevOps is an evolution of the agile movement, it proposes a complementary set of agile practices to enable the iterative delivery of software in short cycles effectively.

Besides automating the delivery process, DevOps initiatives have also focused on using automated runtime monitoring for improving software runtime properties, such as performance, scalability, availability, and resilience (LEITE et al., 2019).

Using containers, one could run a single container to execute a small micro-service or software process to a more extensive application (GOLDEN, 2019). Containers and micro-services enable DevOps (LEITE et al., 2019). Considering a hypothetical context, running a micro-service on bare metal is an attractive option, since multiple services on a single operating system instance can lead to conflicting library versions; one micro-service failure could affect others' behavior.

Regarding DevOps practices, Continuous Testing (CT) highlights as the most fitting concept with two of the core aspects of DevOps, continuity of the process of development and a source of uninterrupted feedback - despite being a relatively new concept in Software Engineering. The practice of Continuous Testing pivots around test automation as well as early and frequent testing. Continuous Testing is a crucial component of the software development cycle that includes continuous development, integration, and deployment.

Whereas some authors say microservices facilitate effective implementation of DevOps, others say microservices require DevOps, since deployment automation minimizes the overhead to manage a significant number of microservices. However, adopting microservices comes with several challenges. First, there is heterogeneity in non-functional patterns such as "startup scripts, configuration files, administration endpoints, and logging locations". Technological heterogeneity can be a productivity barrier for newcomers in the team. Second, microservices must be deployed to production with the same set of versions used for integration tests (LEITE et al., 2019).

2.2.2 Legacy Software and its challenges

Legacy software is commonly defined as an application that is no longer updated or supported by the developer. Likewise, the software can become legacy if the developer's operation ceases or bought by another entity that decides to throw it out (CHIMA, 2016). The definition is not limited to that, Legacy software systems are also considered programs that are still well used by the community or have some potential inherent value but were developed years, days, or even hours ago (GREENOUGH; WORTH, 2003). A software becomes legacy when its dependencies are not keeping up with the latest updates. That could represent software developed a few days ago, which has a vast and active maintainer community.

According to Sommerville (SOMMERVILLE, 2015), Legacy systems can be rawly defined as old software systems that are used by an organization and usually rely on obsolete technology but are still essential to the business. This definition is totally correct, but coming to a wider definition legacy software also represents a software created one day ago. That is justified given the rapid advances and the increased reliance on software-related technologies (CASCIO; MONTEALEGRE, 2016).

Often legacy codes have been maintained and developed by hundreds of programmers. While many changes have been made to it, the supporting documentation may not be current, and the programming style does not follow current standards (GREENOUGH; WORTH, 2003). The challenges could get more prominent, as such software might offer a volatile development environment - which makes contributing at any level very hard. Those challenges could be not have a representative test coverage or an arduous setup of the work environment.

A very proper example of legacy software is the web-based social platform Noosfero (CONTRIBUTORS, 2007). Basically Noosfero is an open source framework for social networking. Considering that its first versions are dated back to 2007 it has several legacy practices and code.

Including DevOps into a large-scale legacy system day-to-day is a challenging exercise since they often predate DevOps or may have been developed without taking into account the unique characteristics of its practices.

DevOps principles, practices, and tools are changing the software industry. However, many industry practitioners, both engineers and managers, are still not aware of how their daily work can be affected by such principles, practices, and tools. As well, some legacy architectures might not be designed to run automated tests. Nonetheless, teams must be aware that cultural factors, such as managers who say “This is the way we have always done it”, can limit the adoption of continuous delivery more than technical factors. (LEITE et al., 2019).

Although companies recognize the importance of automated testing, they still struggle to implement it fully. Other factors that make automated testing complex are hardware availability for load testing and user experiment assessment. The benefits delivered by a deployment pipeline, that is continuous delivery, are many. However, engineers must be aware that setting up the infrastructure for continuous deployment can demand a considerable effort. Breaking down the system into microservices also requires building multiple pipelines (LEITE et al., 2019).

There are still many open questions about how organizations should adopt DevOps. It is stated that DevOps adoption requires top-management support. Sometimes it does not happen in the first moment, and an anti-organizational strategy can take place. Moreover, arguments to encourage DevOps adoption can differ from engineers to managers - which is the organizational hierarchy structure of most legacy code teams (LEITE et al., 2019).

2.3 Strategies to Bring DevOps into Legacy Code

Refactoring is the process of changing a software system so that it does not alter the code's external behavior yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when one refactors actually is improving the design of the code after it has been written (FOWLER; BECK, 2002).

In the context of Legacy Code involves embracing several strategies and practices. Considering the challenges involved in refactoring legacy code, several organizations are not rushing to adopt such practices properly.

It is important to mention that the techniques, or strategies, described below in this work will not be considered refactoring strategies - analyzing through the spectrum of SOLID and Clean Code set of techniques. In this context,F the following strategies are considered to be strategies simply required to a legacy software team to adopt the DevOps culture.

2.3.1 Legacy in the box

Legacy code, especially massive monoliths, is one of the most unsatisfying, high-friction experiences for developers. Although there is always much caution involved in extending and maintaining legacy monoliths, such upgrades continue to prove to be very necessary, even though it takes a lot of work and money to keep maintaining such monolith.

To help reduce the friction, developers have used virtualized machine images or container images with Docker containers to create immutable images of legacy systems and their configurations. This technique, called "legacy in the box", contain the legacy code in a box for developers to run locally and remove the need for rebuilding, re-configuring or sharing environments. In an ideal scenario, teams that own legacy systems generate the corresponding boxed legacy images through their build pipelines, and developers can then run and orchestrate these images in their allocated sandbox more reliably.

Adopting Legacy in the Box practice is not only about wrapping the legacy code in a container and ship it. It also features some other DevOps-related practices, such as adopting Continuous Integration and Continuous Deploy (CI/CD) into the project workflow - previously described.

For instance, when it comes to adopting Continuous Integration into the Legacy project, a core practice consists of all developers committing to the mainline branch daily. When a team makes changes in smaller increments and integrates them into the mainline regularly. More minor changes, shipped to production quickly, are a lot easier to debug when something breaks. Rather than living in branches for long chunks of time, changes are continuously integrated (MEYER, 2014).

Beyond version control, a continuous integration server is one of the more essential tools a development team can put to fair use. A continuous integration server is unbiased. Its tasks boil down to telling the team whether the most recent changes still pass the stages it is configured to run (MEYER, 2014).

The last step of a fully automated build is deploying to production, which requires an automated deployment process that every developer should be able to run, just like the continuous integration server. With an automated build in place, everyone can deploy to staging or production, anytime (MEYER, 2014).

2.3.2 Testing, Integrating and Deploying Continuously

As previously said, CI/CD is a method to frequently deliver apps to customers by introducing automation into the stages of app development. Such practices introduce ongoing automation and continuous monitoring throughout the software life-cycle, from integration and testing phases to delivery and deployment. Taken together, these connected practices are often referred to as a CI/CD pipeline (HAT,).

When Continuous Testing is adequately implemented, an organization can get a constant insight into the robustness of the latest software build and ensure speedy delivery of high-quality software. CI/CD is a method to frequently deliver apps to customers by introducing automation into app development stages. Continuous Integration helps teams work more efficiently because the different components of a complex system will more assuredly work together. By having each piece of code verified by an automated build, a team is allowed to develop cohesive software more rapidly. Leading to significantly reduced integration problems and quick error detection. However, once this bottleneck is overcome, CD presents several benefits that directly influence the end product. Since CD - and CI - are all about automation, it allows teams to focus on the actual product and testing. Also, make it possible to integrate teams and processes with a unified pipeline, thus standardizing the entire project.

Continuous practices are expected to provide several benefits such as: getting more and quick feedback from the software development process and customers; having frequent and reliable releases, which lead to improved customer satisfaction and product quality; through CD, the connection between development and operations teams is strengthened and manual tasks can be eliminated.

In DevOps, CI/CD along with testing plays a vital role since it results in trustful services due to the use of agile development methods and concepts - also embraced by the DevOps practices. Continuous integration tools orchestrate several automated actions that, together, implement the deployment pipeline pattern. Among the stages orchestrated by the pipeline are: package generation, automated test execution for correctness verification, and deployment to both development and production environments (LEITE et al., 2019).

Continuous Delivery has been the approach to bring automation, quality, and discipline to create a reliable and repeatable process to release software into production. Pillars of DevOps : automated stages, quality, repeatable process, automated test stages, and more (SATO; WIDER; WINDHEUSER, 2019).

Continuous delivery and continuous deployment will be used as synonyms, also referred to as CD. CD usually means a developer's application changes are automatically bug tested and uploaded to a repository, where it can be later deployed to a live production environment. Another approach to defining CD is that it can refer to automatically releasing a developer's changes from the repository to production, where it is made available to customers. It addresses the problem of overloading operations teams with manual processes that slow down app delivery. CI/CD is really a process, often visualized as a pipeline, that involves adding a high degree of ongoing automation and continuous monitoring to app development (HAT,).

2.3.3 Architecture

A Software Architecture is concerned with both structure and behavior, is concerned with significant decisions only, may conform to an architectural style, is influenced by its stakeholders and its environment, and embodies decisions based on rationale. Some authors explore software design in the context of DevOps, continuous delivery, and continuous deployment. However, developers may still struggle with this in practice, since achieving the desired architecture can be infeasible in a single first DevOps (LEITE et al., 2019).

As well as defining structural elements, an architecture defines the interactions between these structural elements. And are these interactions that provide the desired system behavior (EELES, 2006).

2.3.3.1 Micro-services

When it comes to specifying among the various architectures, the micro-services architecture stands out to aid the DevOps implementation. As the size of a software systems increases, the computation algorithms and data structures no longer constitute the major design problems. When systems are constructed from many components, the overall system's organization – the software architecture – presents a new set of design problems (GARLAN; SHOW, 1993).

Micro-services is a style of architecture that emphasizes dividing the system into small and lightweight services that are purposely built to perform a very cohesive business function and is an evolution of the traditional service-oriented architecture style. This architecture is an approach to developing an application as a set of small independent services. Each of the services is running in its independent process (NAMIOT; SNEPS-SNEPPE, 2014). As the software grows, it can be a great approach to achieve scalability.

2.4 The case study

The previous section described several handy concepts that, when explored, could represent a great advantage when refactoring a legacy code. Such concepts obey a particular pattern when applied to the process of upgrading and also refactoring itself.

In order to successfully achieve this work's goal, there should be defined as a well-structured process, specifically, agile developing methodologies.

2.4.1 Open Source Software (OSS)

Open source software is software with source code that anyone can inspect, modify and enhance (OPENSOURCE.COM, 2019). Open source software can be defined as software distributed under a licensing agreement which allows the source code (computer code) to be shared, viewed, and modified by other users and organizations (SINGH; BANSAL; JHA, 2015).

Freedom with the source code allows developers to create unique solutions, which can then be built upon by other community members. This process of "crowdsourcing"allows for development shops to pull beyond their teams' talents and access a repository of information compiled by the community at large.

Open source solutions geared toward the enterprise often have thriving communities around them, bound by a shared drive to support and improve a solution that both the enterprise and the community benefit from (and believe in). The global communities united around improving these solutions introduce new concepts and capabilities faster, better, and more effectively than internal teams working on proprietary solutions. Not to mention that this brings several benefits to the end-user as well.

Furthermore, utilizing DevOps solutions in the context of an open-source community can be both time and cost-effective and also very practical to organizations in general. DevOps is a newer and less mature software practice. It requires a new tool, process, and solutions development; in other words, the developers will empirically implement the DevOps strategies according to its organizational needs. Leveraging open source solutions can expedite that process. Many of the key DevOps tools used today either are or started as open-source solutions for DevOps problems, which certainly fits an open-source software project's objectives. While DevOps and open source are two entirely separate things, though, the reality is that it's difficult to separate the two at this point. Many open source projects rely on DevOps tools and principles, and DevOps depends heavily on open source applications as both the glue that binds it all together and the engine that keeps everything moving (BRADLEY, 2016).

Open-source software development, particularly its core tenets of collaboration and transparency, has always been an integral part of DevOps. This is one of the reasons that DevOps tends to be an easier adjustment for developers, who tend to have experience with open-source software and its concepts and technologies (LYMAN, 2020).

With OSS, community members have open access to the source code and can use it in any way they see fit. Also, an open-source project can be altered and extended by any developer familiar with the source code. This grants organizations freedom and long-term viability because hundreds of developers supporting a widely adopted OSS project can be called upon long into the future.

2.4.2 Study Design

Project	Mapknitter	Noosfero	Spectral Workbench	Salicml
Number of commits	2,512 commits	16,785 commits	1,271 commits	638 commits
Contributors	75 contributors	25 contributors	18 contributors	14 contributors
Lines of source code	60.863 lines	227.024 lines	46.201 lines	85.804 lines
Date of first commit	26/04/2009	27/06/2007	27/09/2010	27/03/2018
License	GPL v3	GPL v3	GPL v3	GPL v3
Main Programming Language	Ruby	Ruby	Ruby	Python
Framework version	Rails 3.2.2	Rails 4.2.4	Rails 3.2.3	Django 2.2

Tabela 1 – Study Cases information

2.4.2.1 Methodology

Ethnography is a research method designed to describe and analyze the social life and culture of a specific social system (EDMONDS; KENNEDY, 2013). The central tenet of this approach is to understand values, beliefs, or ideas shared for a group under study from the members' point of view. For this, the ethnographer needs to become a member of the group, observing in detail what people actually do and learning their language, social norms, rules, and artifacts.

Ethnographic research is a qualitative methodology which requires the researcher to interpret the real world from the perspective of the informers in the investigation (DOBBERT, 2013). And in software engineering context, it can strengthen investigations of social and human aspects in the software development process since the significance of these aspects of software practice is already well-established.

In this work, we acquire data by using the ethnographic research method of participant observation and documentation analysis. The participant observation method makes it possible to explain and justify the meaning of the experiences through the experience of the observer and allow the informant to judge what is important rather than what he thinks is important. In addition to sensitivity, the observer needs to interpret what is happening in the community around him.

Software developers find it easier to reveal the processes present in their thoughts when communicating with other software developers, which makes this communication a valuable opportunity to observe the development process. This justifies why in this work, a method for data collection used was keep track of the various communication tools used to exchange information regarding certain project.

2.4.2.2 Contextualized Methodology

This study methodology has been fundamental in the context of this work. Through it, it has been possible to collect every needed data that has been later used to build the strategies/techniques. By observing and describing the entire process of implementing the DevOps culture in a legacy project, it was possible to obtain very relevant data that has been used to generate the DevOps strategies and their order of implementation.

There were four study cases conducted as shown in Table 1, each case had its our peculiarities which has allowed us to apply a different approach at every study. Each case study consisted in contributing to an open-source software community in terms of applying certain strategies to get the community to embrace the DevOps culture and practices.

The first case was Noosfero, an open-source framework for social networking that has around fourteen years since its first commit, nearly two hundred and fifty thousand lines of code, twenty thousand commits and twenty-five contributors - the most legacy of all cases. The second case analyzed was Mapknitter, which is a project that is part of a huge ecosystem of services provided by the PublicLab community, it allows geographical data exporting and uploading, with around seventy-five contributors, sixty thousand lines of code and eleven years old. The third study case conducted was Spectral Workbench, which is also part of the PublicLab ecosystem, a web based application to collect, archive, share, and analyze spectral data. It has eighteen contributors, twelve hundred commits and around forty-six thousand of lines.

In order to get a possible different point of view, the fourth case was conducted mainly by Victor Moura. It consisted in the project Salicml, that has around eighty-five thousand lines of code, fourteen contributors and over six hundred commits. Salicml is a web application that processed business indicators from cultural projects and presented them in a web dashboard.

In each project, the study has lasted 5-6 months, including the one conducted by Victor Moura. The only exception to this has been the Noosfero case, in which the it has lasted around a year - as it is the bigger project in number of lines.

As more studies have been conducted the pattern of strategies to be applied were becoming more and more clear. During every case, it was noticed that before starting any framework upgrade, it would be indispensable to cover the project of tests. However, to test it properly, it would also be fascinating to know which parts of the code I would be testing and how much of the project I would be testing, that is, in percentage. Also, a smart idea would be automating the entire test process since it increases the number of times exponentially one has to trigger the command to run the tests.

All of this empirical work done in various legacy projects leads us to conclude that before adopting Continuous Integration, one should adopt Continuous Testing before it. The interesting part is that every strategy has been obtained through this, making the Case Study methodology very important for this work accomplishment.

It is also worth mentioning that after the first case study conducted - Noosfero - there was already a solid set of practices to-become-strategies and their most adequate usage order. As there were more study cases, the strategies became more and more evident and their order.

After the completion of every study case every information source was analyzed, as commits, issue reports, pull requests, informal communication tools and such. By analyzing that kind of resource it was possible building the set of practices, called Strategies in this paper. These resources also made possible gathering the posthumous lessons learned from the cases experiences, that would later become the foundation for this study.

2.5 Results

In Table 2, it is objectively pointed out which strategy and its order of usage to every case during the study. The following sections details more about each case.

Project	**Repository Link**	**Description**	**Technique Applied (in order of usage)**
Noosfero	https://gitlab.com/noosfero/noosfero/	An open-source framework for social networking with blogs, e-Portfolios, CMS, RSS, thematic discussion, events scheduling, and more.	Continuous Integration, Legacy in the Box.
Mapknitter	https://github.com/publiclab/mapknitter/	A free and open-source software created run by Public Lab. It lets people upload their own aerial images in a web interface over some existing map data, share it, and export for print.	Legacy in the Box, Continuous Integration, Microservices architecture, Continuous Deploy.
Spectral Workbench	https://github.com/publiclab/spectral-workbench	A web based application to collect, archive, share, and analyze spectral data, for Public Lab DIY spectrometers and other spectrometers.	Continuous Integration, Legacy in the Box, Continuous Testing, Continuous Deploy, Microservices architecture.
Salicml	https://github.com/lappis-unb/salic-ml/	A web application that processed business indicators from cultural projects and presented them in a web dashboard to optimize the analysis of each project accountability by the technical team from the Brazilian Ministry of Culture.	Continuous Integration, Continuous Testing, Continuous Deploy

Tabela 2 – Strategies per study case

2.5.1 Noosfero

Noosfero		
	Before DevOps	**After DevOps**
Docker/Docker Compose	Misconfigured. Services were properly split but with several misconfigurations. Not used in production.	Working properly for development and production environments.
Framework version	Rails 4.2.4 with several deprecated dependencies and vendors. A lot of monkey-patches.	Updated to Rails 5.1.6 with latest features.
Continuous Integration	GitLabCI builds took too long to finish and had important pipelines missing.	Implemented caching to speed things up and added missing builds to the pipeline executor.
Continuous Deploy	None.	None.
Coding stylesheet	None. Every developer had its own technique.	Configured a stylesheet and integrated it with the CI pipelines, and fixed all of the linting errors.

Tabela 3 – Noosfero Comparative: Before DevOps and After DevOps.

Noosfero is a vast system, with over 70 database tables. Since there was a stable Continuous Integration tool set up and microservices have been widely made use of, there were only a couple of DevOps related improvements to do.

During the Noosfero study, which has been done first, there was a limited implementation for containers and continuous integration. Since it was the first upgrade of this kind that it has been worked on, a few errors have resulted in valuable learnings. When the Rails framework upgrade started, it was noticed that some steps should have been taken before, which would make the upgrade less painful.

In the middle of the refactoring, the Continuous Integration pipeline could have been improved by adding other testing stages, which could have identified some issues that appeared later. For instance, by previously adding a stage that tested out the Docker image building, we could assure every time that we included a change, this part of the project could remain stable, we should have improved the pipeline before starting to fix the broken tests, what would've had provided a better visibility of next steps. Also by including a code quality and stylesheet compliance stage we could also assure that our code refactoring was changing the code for the better, by making it more maintainable for example.

The project was also not properly wrapped in a container image, which should have been done before the upgrade started. By wrapping up the monolith through the concept of Legacy in the box, there was a homogeneous environment for every developer to work with. That has provided a consistent environment for the Noosfero application. In a different approach, Docker containers ensure consistency across multiple development and release cycles, thus standardizing the Noosfero environment.

Realistically, containerizing Noosfero before upgrading the Rails framework has been of great advantage; that meant parity, meaning that the Noosfero images ran the same no matter which server or whose laptop they were running on. The Noosfero study case only involved me as developer for this task, even though the maintainers allowed me to freely experiment the strategies, as in Mapknitter, due to the project complexity and few resources, by the end of the study it was possible to apply only the Legacy in the Box and Continuous Integration strategies.

It was also acknowledged that the Continuous Integration tool could be better used in terms of performance, so all of the testing and integration pipelines have been split to run in parallel since there was no inter-dependency between the suites. A style-sheet guide has also been added to this pipeline using Rubocop in order to enforce and obtain a more standardized code pattern.

2.5.2 Mapknitter

In Mapknitter the maintainers aimed to achieve a more compliant and stable project, with that, more newcomers are attracted to contribute with the open-source code.

Mapknitter		
	Before DevOps	**After DevOps**
Docker/Docker Compose	Misconfigured. Database and services all wrapped in a container. Only worked in production.	Working properly for development and production environments.
Framework version	Deprecated Rails 3.2.2 with several deprecated dependencies.	Updated to Rails 5.2.3 with latest features.
Continuous Integration	Misconfigured TravisCI, worked poorly.	Improved to cached pipelines with reduced timeouts with more stages running in parallel.
Continuous Deploy	Misconfigured JenkinsCI, didn't work.	Improved build and startup steps arrangement in order to have it working the best way it could. Every repository push would trigger a build that could be followed live.
Coding stylesheet	None. Despite the other Org repositories had it configured.	Configured a stylesheet and integrated it with the CI pipelines, and fixed all of the linting errors. Thus, making the project following the org's coding patterns.

Tabela 4 – Mapknitter Comparative: Before DevOps and After DevOps.

The Mapknitter case study was a very challenging project. It includes various sub-components; among them, there is the core application written in Rails and a Javascript interface. At first, Docker has provided several benefits to the Mapknitter project itself but mostly for the Rails framework upgrade. The time required to build the container was very low, and in short time we had a working developing environment. During the containerization process, we could notice that the Travis CI tool had been using the production environment. So it was necessary to split the development, test and production environments, which has been done. With a few more improvements, Travis had set parallel jobs - what caused the builds to run twice as fast.

So by first adopting the concept of Legacy in the box, leads the update to take a further step and adopt the microservice architecture. That has been achieved at first, by splitting the MySQL database and the Mapknitter web app. Later we got also to containerize the ForeGo service, thus having three independent services running alongside.

Later on the project, we also got to setup Rubocop linter and stylesheet, which following the same standards used in Plots2 project - other project part of the PublicLab community ecosystem. By doing this, now there was a more cohesive and uniform set of projects in the organization. Also this linting tool has been integrated with the continuous integration tool to keep track of the syntax changes.

This refactoring involved two developers, me and another member of the community, the maintainers let us work very freely through the process, what has given us the chance to explore and try several ways of applying the strategies. And by the end of the Mapknitter study it was possible to apply a wide set of strategies, which were, in order: Legacy in the Box, Continuous Integration, Microservices architecture and Continuous Deploy.

2.5.3 Spectral Workbench

Spectral Workbench		
	Before DevOps	**After DevOps**
Docker/Docker Compose	None	Working properly for development and production environments.
Framework version	Rails 3.2.3 with several deprecated dependencies and vendors.	Updated to Rails 5.2.4 with latest features.
Continuous Integration	Misconfigured TravisCI, worked poorly.	Improved to cached pipelines with reduced timeouts with more stages running in parallel.
Continuous Deploy	None.	Configured JenkinsCI pipelines. Build and startup steps arranged in order to have it working the best way it could. Every repository push would trigger a build that could be followed live.
Coding stylesheet	None.	Configured a stylesheet and integrated it with the CI pipelines, and fixed all of the linting errors.

Tabela 5 – Spectral Workbench Comparative: Before DevOps and After DevOps.

With the previous experience acquired from the other study cases, there was already an implicit order of strategies to be applied. First the docker workflow of the project was rewritten, since it was an "old"repository - with legacy code and practices, it required some restructuring and refactoring on the configuration files. For instance, the MySQL instance was not dockerized and there was no automation that aided a developer to easily start coding.

The Continuous Integration tool needed to be configured to execute local builds, so that we could obtain a testing environment that simulated faithfully both the development and production environment. This same CI tool previously was configured to run all tests at once - what caused the builds to take valuable coding time. So I had to split the test running by groups, in a way that each test suite was executed separately, thus taking advantage of the parallelism provided by the tool.

When it comes to testing, a main request of one of the maintainers was the configuration and inclusion of system tests and increase of the test coverage. Both of the requirements have been accomplished.

After the Rails framework upgrade was complete, it was required a staging environment so that we could test out the changes that were made on the cloud, a staging environment. So along with the help of PublicLab's sysadmin this was set, in an automated manner. And with Rubocop we got to standardize the coding style among the several contributors; the Rubocop settings used were the same as the ones used in Plots2, Mapknitter and Spectral Workbench.

The Spectral Workbench study case only involved me as developer, the maintainers let me work very freely through the process, what gave me the chance to explore and try new strategies, besides the ones I had used in previous study cases, via Continuous Testing. And by the end of the study it was possible to apply the greater set of strategies of all study cases: Continuous Integration, Legacy in the Box, Continuous Testing, Continuous Deploy, Microservices architecture.

2.5.4 Salicml

Salicml		
Docker/Docker Compose	None.	Working properly in development and production environments. Included a private database proxy to abstract VPN connections to developers. Multiple configurations to reflect every existing environment
Framework version	Django 2.2	Django 2.2
Continuous Integration	None.	Configured Gitlab CI tool to check on docker builds and automated test running. Integrated CI tool with docker containers management.
Continuous Deploy	None.	Configured properly. Totally automated by using Rancher and Watchtower tools.
Coding stylesheet	None.	None.

Tabela 6 – Salicml Comparative: Before DevOps and After DevOps.

Salic is an open source Brazilian governmental project, written in PHP, monolithic, with no tests or technical documentation. The maintainers wanted to include a machine learning module in the project. Since most machine learning libraries are written in python, we containerize the legacy source code, and we build this new module as a microservice, already employing all DevOps good practices and automations.

In the Salicml study I have not worked directly in this project, so that I could obtain a third-party point of view regarding the DevOps strategies to apply and their respective order of application. This different approach was very useful, as it helped reasserting certain practices applied in the previous studies, and thus it was possible forming them into strategies.

First it was included a docker development workflow for both development and production environments. A private database proxy to abstract VPN connections to developers was also included.

Also, the main application image was built from another custom image. In practice, whenever a change was inserted into the codebase and it didn't affect the application's dependencies, the requirements docker image didn't have to be rebuilt, thus optimizing the pipeline resources usage.

This refactoring involved Victor Moura, the maintainers allowed him to work very freely through the process, what has given him the change to use a wide set of tools. And by the end of the Salicml study it was possible to apply a great range of strategies, which were, in order: Legacy in the Box, Continuous Integration, Microservices architecture and Continuous Deploy. The main focus of this study in question was containerizing the legacy software and assure evolvalibility of it through the implementation of the Microservices architecture.

By the end of the Salicml study it was possible to apply a great range of strategies, which were, in order: Legacy in the Box, Continuous Integration, Microservices architecture and Continuous Deploy. The main focus of this study in question was containerizing the legacy software and assure evolvalibility of it through the implementation of the Microservices architecture.

2.6 Discussion

One of the most important things that could be extracted from those refactorings is that the order of the strategies to apply matters a lot. For instance, if you choose to implement Continuous Deploy in your legacy software before having Continuous Integration set up, you could be taking a lot of risks by pushing certain amount of untested code to the cloud, or even be wasting a lot of precious time by manually testing it first and then deploying.

Considering another hypothetical case, one could choose to split the various components of the legacy software in several services - thus taking advantage of the Microservices architecture strategy - but it does that before implementing the Legacy in the Box technique. It may be very complicated to keep this architecture change flowing in a good pace without taking advantage of the various benefits that a legacy in box tool, such as Docker, could bring. Actually, making the refactoring way easier. In fact, it means that choosing the right order of strategies to be applied could prevent one from taking several extra hours, even days, of massive manual labor.

As the first strategy one should take to embrace DevOps in a legacy project is having a Continuous Integration pipeline set up. With that - along with a minimum test coverage - one can assure that the small pieces of code are still working, thus guaranteeing a more trustful code base. It is also noticeable that implementing CI strategy first will absorb the time a developer would take to run tests every time future integrations happened.

After having the work environment CI-friendly, the next step one should take is wrapping the legacy app in a container. Every configuration, third-party packages, and abstraction get to be explicitly defined in a container image, also being able to run anywhere basically.

Continuous Deploy and Testing are desired strategies, especially when it comes to testing, but needing to deploy and test are not a bottleneck - until a certain point, of course, and this affirmation also depends on the size, developers, and business rules that this legacy software goes by. If one has the chance and time to keep continuously testing the legacy code and implement an integration to ship at every successful CI tool build, then those are convenient strategies to adopt.

Moreover, implementing microservices is what one would call an utterly optional strategy because it takes a lot of time and effort to do it correctly. If it is not done right, you will only obtain a distributed monolith, with every said "micro"service executing heavy operations. Furthermore, once one has adopted the previous strategies, it is considerably less painful to implement it.

Continuous Testing is by far the broadest strategy, meaning that almost every legacy should adopt it when embracing the DevOps practices; it holds great significance for organizations using DevOps for the regular deployment of software into production. Continuous Testing in DevOps essentially interweaves testing efforts into all stages of designing, developing, and deploying the software. When it is adequately implemented, an organization can get a constant insight into the robustness of the latest software build and ensure speedy delivery of high-quality software.

When it comes to Continuous Integration, we can not say it is as "mandatory"as Continuous Testing. However, it certainly is beneficial, and, indeed, it will save a lot of the developers time. Of course, it could be painful at first for the team, and adapting a legacy software to such practices could be considerably expensive. Implementing a trustful CI pipeline could involve completely change a software development culture, adapt the organization and workflow, automate the testing bulk, and even provide certain infrastructure. Nevertheless, in the long term, the benefits are countless.

By having each piece of code verified by an automated build, a team is allowed to develop cohesive software more rapidly. Leading to significantly reduced integration problems and quick error detection. The main goal of Continuous Integration is to provide rapid feedback so that if a bug is introduced into the codebase, it can be identified and corrected as soon as possible.

As in Continuous Integration, Continuous Deploy, when done right, is full of benefits, but implementing a trustful pipeline may be irksome as in CI. The technical parts are more comfortable than the organizational and cultural parts when it comes to legacy software. However, once this bottleneck is overcome, CD presents several benefits that directly influence the end product. Certainly, Continuous Deploy, when done right, is very fruitful - primarily when used along with Continuous Integration. Since failures are detected faster and fixed faster, it leads to higher release rates, making it possible to evaluate new code faster - and in smaller portions - thus allowing the developers to focus on the product features themselves.

Containerization, or commonly legacy in the box, is by far the strategy that presents one of the most significant benefits of all strategies. It is the fastest and straightforward strategy to implement. It does not require special technical knowledge and gives support to the other strategies. It is a common misconception that using containers only makes sense if the app to be hosted is composed of microservices, but monolithic deployments can benefit from containers. Using a container provider for the legacy code immediately makes it easy to move the app from one host to another just by migrating the previously generated container image. Every developer is using the same container image - this means consistency. Several other benefits intrinsically appear with the mentioned aspects, such as scalability, bare-metal access to the hardware, easy distributing, and much.

Two significant benefits are perceived by using containers as part of the Legacy in the Box strategy. First, resource utilization is much more efficient. Second, containers are cheap in man-hours to maintain and represent only a few costs a machine's resources. Container technology supports streamlined build, test, and deployment from the same container images; it enables Continuous Integration and Deploy. By using a container provider for the legacy code, one can immediately make it easy to move the app from one host to another just by migrating the previously generated container image.

Packaging the legacy code as a container, distributing it through an image repository is very facilitated. Anyone with access can pull the container image and run it. Every developer is using the same container image - this means consistency.

In legacy apps context, the meaning of the word containerization needs to be augmented to include all that is necessary to make an existing app ready to adopt Legacy in the Box concept. That is, to an extent that is well balanced with technical feasibility and expected business benefits. Choosing the right legacy containerization technique within this spectrum is a matter of striking the right balance between investment, business outcome, cost-effectiveness gain, technical feasibility, and risk appetite (HAASJES, 2020).

While this practice delivers some benefits, it does not offer the full benefits of modular, container-based application architecture. Using containerization to the fullest involves refactoring the existing applications to adapt to the containers thoroughly. That could quickly scale out, thus providing better support for microservices architecture. Container technology supports streamlined build, test, and deployment from the same container images; it means better support for Continuous Integration and Deploy.

At last, the Microservices strategy provides many advantages, but to the right contexts. One of the most significant advantages of a microservice over a monolithic architecture is that a microservice architecture allows different components to scale at different rates. The flexibility of microservices lets a system expand fast without requiring a significant increase in resources.

Also talking about the benefits of it, we have that, for instance, we know that every single microservice work independently and thus can be written with different technologies, and since all services are independent, developers are allowed to add, replace, and remove different services without influencing the already existent services.

Nevertheless, sometimes, using different languages, libraries, frameworks, and data storage technologies can be intimidating and paralyzing for organizations at first. They could become a "Frankenstein"of services that a long term. Plus, not every team can handle the autonomy, and independence microservices offer. Like any architectural approach, Microservices are hard to design correctly, and one should plan a lot before adopting this strategy.

Finally, one should consider these techniques or strategies just the plain basics the would start permitting the evolution and maintainability of a legacy code with DevOps. That means continuously and safely update the legacy project's dependencies, keep refactoring the code so that its quality enhaces - by following the SOLID and Clean Code premisses, this refactoring should also get done in a way that leads to the componentization of the various parts of the code and when adding new features, assure that these are matching the current language standards.

It is also important to mention that throughout this entire automation process the very own team of the legacy project qualifies in DevOps technologies, thus adapting the development (Dev) and operations (Ops) processes and premisses according to the DevOps culture.

2.7 Conclusion

There were four study cases conducted as shown in 2, each case had its our peculiarities which has allowed us to apply a different approach at every study. At every study it was possible to apply Continuous Integration. The Legacy in the Box technique was not only applied in Salicml, since the project already presented a stable container environment. When it comes to Continuous Deploy it was not just possible to apply in the Noosfero project and using Microservices was only achievable on Mapknitter and Spectral Workbench. By far the techniques that immediately presented benefits were Continuous Integration and Legacy in the Box, as of their easiness of execution.

Every study case was a different sequence of strategies implementation, by registering every bottleneck and benefits perceived and based on the experiences acquired through the results achieved in this work, we found the following implementation sequence reduce the technical complexities of adopting DevOps:

1. Continuous Integration
2. Legacy in the Box
3. Continuous Testing
4. Continuous Deploy
5. Microservices architecture

The present work presented an experience report of DevOps adoption in four open source web projects.

We present a realistic analysis of the DevOps strategies that might help several teams aiming to modernize their legacy systems. It could give them guidance to consider the upcoming steps to take and provide an overview of the importance of certain things.

Based on real-world experience, we set out the strategies, benefits, and countermeasures for each team with a specific condition or need. This work consisted of obtaining abstract information from previous experiences when upgrading legacy software. We have extracted data based on two of those experiences and mashed into the strategies that have been portrait in previous sections.

Of course, the DevOps culture of practices presents several other practices and mindsets to make beneficial strategies. However, the strategies presented here were considered more relevant, and the ones that present the most impact in outdated legacy software.

Referências

AMAZON. *What is DevOps*. 2020. <https://aws.amazon.com/devops/what-is-devops/?nc1=h_ls>, accessed on October 2020. Citado na página 15.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHID, P. Microservices architecture enables devops: An experience report on migration to a cloud-native architecture. International Journal of Open Information Technologies, 2016. Citado 2 vezes nas páginas 16 e 17.

BOGNER, J. et al. Assuring the evolvability of microservices: Insights into industry practices and challenges. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 2014. Citado na página 16.

BRADLEY, T. *The Symbiotic Relationship of DevOps and Open Source*. 2016. <https://techspective.net/2016/06/01/symbiotic-relationship-devops-open-source/>, accessed on December 2020. Citado na página 27.

CASCIO, W.; MONTEALEGRE, R. How technology is changing work and organizations. The Business School, University of Colorado, Denver - Colorado 80217, 2016. Citado na página 22.

CHIANG, C.-C.; BAYRAK, C. Legacy software modernization. 2006 IEEE Conference on Systems, Man, and Cybernetics - Taipei, Taiwan, 2006. Citado na página 19.

CHIMA, R. *Legacy Software: How To Tell If Your Software Needs Replacing*. 2016. <https://www.bbconsult.co.uk/blog/legacy-software>, accessed on December 2019. Citado na página 21.

CONTRIBUTORS, N. *Noosfero*. [S.l.]: GitHub, 2007. <https://gitlab.com/noosfero/noosfero>. Citado na página 22.

DOBBERT, M. L. *Ethnographic research: Theory and application for modern schools and societies (Praeger studies in ethnographic perspectives on American education)*. 1st. ed. Nova Southeastern University, USA: Praeger (January 1, 1982), 2013. ISBN 0030614732, 978-0030614736. Citado na página 28.

EDMONDS, W. A.; KENNEDY, T. D. *An applied guide to research designs : quantitative, qualitative, and mixed methods*. 2nd. ed. Nova Southeastern University, USA: SAGE Publications, Inc; Second edition (May 30, 2016), 2013. ISBN 1483317277, 978-1483317274. Citado na página 27.

EELES, P. What is a software architecture? USA, 2006. Citado na página 25.

ERRICKSON-CONNOR, B. Truth or consequences. Z/Journal, 2003. Citado na página 19.

FOWLER, M.; BECK, K. What is refactoring? In: *Refactoring: Improving the Design of Existing Code*. [S.l.: s.n.], 2002. v. 1, p. 9. Citado na página 23.

GARLAN, D.; SHOW, M. An interoduction to software architecture. World Scientific Publishing Co Pte Ltd, 1993. Citado na página 26.

GOLDEN, B. *3 reasons why you should always run microservices apps in containers*. 2019. <https://techbeacon.com/app-dev-testing/3-reasons-why-you-should-always-run-microservices-apps-containers>, accessed on December 2019. Citado na página 21.

GREENOUGH, D. C.; WORTH, D. D. The transformation of legacy software: Some tools and a process. EngiUniversity of New South Wales, Sydney, Australianeering and Physical Sciences Research Council, 2003. Citado 2 vezes nas páginas 21 e 22.

HAASJES, G.-W. *Containerization of legacy applications*. 2020. <https://developer.ibm.com/technologies/containers/articles/containerization-of-legacy-applications/>, accessed on December 2020. Citado na página 38.

HAT, R. *What is CI/CD?* <https://www.redhat.com/en/topics/devops/what-is-ci-cd>, accessed on December 2019. Citado 2 vezes nas páginas 24 e 25.

INCORPORATED, S. T. Why devops matters: Practical insights on managing complex continuous change. Saugatuck Technology Inc., 2014. Citado na página 15.

KHANAM, Z. Analyzing refactoring trends and practices in the software industry. Saudi Electronic University - Dammam, KSA, 2018. Citado na página 15.

LEITE, L. et al. A survey of devops concepts and challenges. ACM Computing Surveys, 2019. Citado 6 vezes nas páginas 15, 20, 21, 22, 23 e 25.

LYMAN, J. *Open source leads to DevOps success*. 2020. <https://techbeacon.com/devops/open-source-leads-devops-success>, accessed on December 2020. Citado na página 27.

MEYER, M. Continuous integration and its tools. IEEE SOFTWARE, 2014. Citado na página 24.

MURPHY, N. *Site Reliability Engineering book*. 1st. ed. Google Ireland: O'Reilly Media; 1st edition (April 26, 2016), 2016. ISBN 149192912X, 978-1491929124. Citado na página 15.

NAMIOT, D.; SNEPS-SNEPPE, M. On micro-services architecture. International Journal of Open Information Technologies, 2014. Citado na página 26.

OPENSOURCE.COM. *What is open source software? | Opensource.com*. 2019. <https://opensource.com/resources/what-open-source>, accessed on December 2019. Citado na página 26.

RAJLICH, V. Five recommendations for software evolvability. Department of Computer Science, Wayne State University - Detroit, MI, U.S.A., 2018. Citado na página 16.

RAO, G. H. Devops for legacy systems – the demand of the changing applications landscape. Infosys Limited, Bengaluru, India, 2018. Citado 2 vezes nas páginas 16 e 19.

S.A.M.RIZVI; KHANAM, Z. A methodology for refactoring legacy code. Department of Computer Science; Jamia Millia Islamia - New Delhi, India, 2011. Citado na página 19.

SATO, D.; WIDER, A.; WINDHEUSER, C. Continuous delivery for machine learning. USA, 2019. Citado na página 25.

SINGH, A.; BANSAL, R.; JHA, N. Open source software vs proprietary software. Guru Kashi University - Talwandi Saboo, Bathinda, 2015. Citado na página 26.

SMARTSHEET. *The Way of DevOps: A Primer on DevOps Principles and Practices*. 2020. <https://www.smartsheet.com/devops>, accessed on November 2019. Citado na página 20.

SMEDS, J.; NYBOM, K.; PORRES, I. Devops: A definition and perceived adoption impediments. Lecture Notes in Business Information Processing vol 212 Springer Cham, 2015. Citado na página 17.

SOMMERVILLE, I. *Software Engineering, Tenth Edition*. 10th. ed. USA: Pearson; 10th edition (March 24, 2015), 2015. ISBN 0133943038. Citado na página 21.

VASSIT. *6 Key Challenges of DevOps Implementation*. 2016. <http://blog.vassit.co.uk/6-key-challenges-of-implementing-a-devops-strategy>, accessed on January 2020. Citado na página 15.

ZHU, L.; BASS, L.; CHAMPLIN-SCHARFF, G. Devops and its practices. IEEE SOFTWARE, 2016. Citado na página 20.