Foundational DevOps Patterns

PAULO MARQUES, Faculty of Engineering, University of Porto.
FILIPE F. CORREIA, Faculty of Engineering, University of Porto. INESCTEC.

Adopting DevOps practices is nowadays a recurring task in the industry. DevOps is a set of practices intended to reduce the friction between the software development (Dev) and the IT operations (Ops), resulting in higher quality software and a shorter development lifecycle. Even though many resources are talking about DevOps practices, they are often inconsistent with each other on the best DevOps practices. Furthermore, they lack the needed detail and structure for beginners to the DevOps field to quickly understand them.

In order to tackle this issue, this paper proposes four foundational DevOps patterns: VERSION CONTROL EVERYTHING, CONTINUOUS INTEGRATION, DEPLOYMENT AUTOMATION, and MONITORING. The patterns are both detailed enough and structured to be easily reused by practitioners and flexible enough to accommodate different needs and quirks that might arise from their actual usage context. Furthermore, the patterns are tuned to the DevOps principle of Continuous Improvement by containing metrics so that practitioners can improve their pattern implementations.

1. INTRODUCTION

The concept of DevOps was introduced in 2009 [Humble and Molesky 2011] and quickly became popular, as it promises teams to deliver value faster while keeping the same quality level [Forsgren et al. 2018; Forsgren et al. 2019]. In fact, adopting DevOps has been shown to lead to less burnout and an increase in job satisfaction [DORA 2022] and is now very common in the software industry. The word DevOps comes from joining *Development* with *IT Operations*, and although DevOps has been discussed for many years now, there is still no academically accepted definition [Dyck et al. 2015]. This contrasts with the plethora of definitions that is possible can find online. One definition proposed by Leite *et al.* [Leite et al. 2019] is:

> "DevOps is a collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions while guaranteeing their correctness and reliability."

However, the lack of a consensual definition has led to an increasing diversity of software development practices being described under the general term of DevOps [Lwakatare et al. 2016]. This, together with the fact that it often requires complex changes in organizations' processes and workflows [Bucena and Kirikova 2017] makes its adoption anything but trivial.

Furthermore, a quick online search reveals several resources suggesting best practices for DevOps, but these practices are often not sufficiently explained or documented in such a way that they can be learned quickly and communicated effectively.

For these factors, organizations seeking to adopt DevOps often endure additional costs, as they feel the need to hire experts on the topic, or unnecessary costs such as paying to use the wrong tools for the wrong job due to misguidance. Therefore, facilitating an efficient DevOps adoption by providing extra guidance on the adoption process becomes of the utmost interest.

A possible way to tackle this issue is by formalising the DevOps best practices as patterns. These provide sufficient detail so that people can more easily understand and apply the practices contained in them to the context of their organisation. With this goal in mind, we decided to gather a collection of patterns which we refer to as *foundational*. By foundational, we mean a collection of patterns that are meant as a starting point for new DevOps adopters. As such, we propose four new patterns that do not, by themselves, define DevOps but which provide a foundation for a broader pattern collection regarding DevOps practices and their adoption. In other words, they allow to generate the environment that enables such practices.

For mining and writing the patterns proposed in this paper we use three different sources of information for pattern mining:

—**Literature** – By performing the literature review in Section 2 we acquired different perspectives and definitions of the various DevOps practices/patterns. As such, these works provide the foundation for our pattern elaboration.

—**Grey literature** – Many DevOps practices are shared through grey literature. They may appear in blog posts, DevOps tools' documentation, wikis made by companies and more. As such, they also provide valuable information for the writing of the patterns.

—**Past Experience** – To add some details that the two previous kinds of literature do not possess, we use our experience using and implementing DevOps practices to fill in the blanks.

As alternative sources of information, we could have either mined patterns from existing tools or organized interviews with DevOps experts to check what their input was. These were mainly left out as a way to focus the scope of the work. Furthermore, we hypothesised that some of the information that we would get from both sources would already be partially present in the ones we used, mostly on grey literature. Usually the companies that provide the DevOps tools also have documentation and blog post regarding DevOps adoption. Furthermore, usually experts share their knowledge online, through blog post and others. As such, we felt that by not using these sources, we would not miss the bulk of the information, but it still might be worth while explore these sources in future research.

2. LITERATURE REVIEW

We reviewed the literature with the goal of finding what practices it describes, and which of these have already been formalized in pattern format. We resorted to Google Scholar. We also considered grey literature since it is common for practitioners and companies to write their knowledge on blogs or forums. In order to find it we used Google's search engine to find some of the results and from those we followed their references to other grey literature sources. To find works regarding patterns and practices for DevOps we used the following query:

```
devOps AND (practices OR patterns OR capabilities)
```

After having the search results, we start by selecting the most relevant publications based on title, abstract, date, and methodology (if applicable). Then we recorded the relevant ones in a *to-read* list. Finally, we read and analysed the items on the list, looked up the references, and added the relevant ones to this list, including references to grey literature. If the work under analysis was part of grey literature, related works by the same author/organisation were also considered.

We started this analysis by understanding that a relevant number of patterns have been written to document practices for developing Cloud native systems [Richardson 2018; Richardson 2021; Sousa et al. 2018a; Sousa et al. 2018b; Sousa et al. 2018; Sousa et al. 2017; Sousa et al. 2015; Sousa et al. 2016; Maia and Correia 2022; Albuquerque et al. 2022]. Despite the proximity of these works to DevOps and to what it enables, we realised that most of these works do not specifically address DevOps practices and, therefore, we do not consider them in this analysis of related works.

We found two works that tried to describe DevOps patterns. One of them is the website titled *Cloud Adoption Patterns*, which provides a collection of patterns for the Cloud gathered from papers reviewed and accepted at xPLoP conferences. They present a collection of eleven patterns for developing, testing, and managing a cloud-native architecture. Furthermore, the authors present twelve more patterns around building a DevOps pipeline and managing containers [Brown et al. 2022]. On the other hand, their focus is on DevOps patterns that support cloud adoption. As such, some DevOps patterns might not be regarded if they are not relevant for Cloud adoption. The other one is a collection of thirteen DevOps-related patterns, mined from Portuguese startup companies [Teixeira 2016]. However, we found that the context of some of these patterns lacks enough detail for an easy use.

We were able to find a variety of works regarding DevOps practices. The detail of the description for these practices ranges from a one-sentence description of the practice to somewhat detailed descriptions containing a few paragraphs as descriptions and tips on how to adopt it. However, there is an observable discrepancy in the extent of practices proposed by each author. This reflects the absence of consensus on the definition of DevOps or where the boundaries between other concepts should be set. Table I gathers all the practices that we have found, to help analyse them as a whole.

Table I. : DevOps practices and the corresponding works

Practice	Works
Deployment automation	[DORA 2022; Lwakatare et al. 2019]
Trunk-based development	[DORA 2022; Lwakatare et al. 2019]
Shift left on security	[DORA 2022]
A loosely coupled architecture	[DORA 2022]
Empowering teams to choose tools	[DORA 2022]
Continuous integration	[DORA 2022; Pedra 2021; Lwakatare et al. 2019]
Continuous testing	[DORA 2022]
Version control	[DORA 2022; Pedra 2021]
Test data management	[DORA 2022]
Comprehensive monitoring and observability	[DORA 2022; Lwakatare et al. 2019]
Proactive notifications	[DORA 2022]
Database change management	[DORA 2022]
Code maintainability	[DORA 2022]
Continuous Delivery	[DORA 2022; Pedra 2021]
Continuous Deployment	[Pedra 2021]
Automated Build	[Pedra 2021]
Cloud Infrastructure	[DORA 2022; Pedra 2021]
Automated and Continuous Feedback	[Pedra 2021]
Infrastructure as Code	[Pedra 2021; Lwakatare et al. 2019]
Change-based Code Review	[Lwakatare et al. 2019]
Agile/Lean Practices	[Lwakatare et al. 2019]

As we can observe, there are a few practices that have different names but appear to be similar or to overlap considerably, such as the case of *Comprehensive monitoring and observability* [DORA 2022] and *Continuous monitoring* [Pedra 2021] or *Cloud infrastructure* [DORA 2022] and Cloud computing [Pedra 2021]. Furthermore, some of the practices only appear in one work.

We decided to refine this list to get a more consistent list of practices. First, we set out to find similar practices that, despite having different names, had a close-enough description that we could consider them the same practice. Afterwards, we decided to map the existing patterns to the practices to see if any of the practices appeared at least partially as part of a pattern or vice-versa. Then, we decided to exclude the practices that were only mentioned in one work and had no associated patterns. This was done so we could limit the scope of this work and focus only on the more consensual practices.

Finally, we further analysed the remaining eight practices. One of them, *Agile/Lean Practices*, was too broad for a pattern. As such, we removed it from the list. Then we have closely-related practices, such as *Continuous Integration* and *Trunk-based development* where the latter can be seen as a practice that enhances the first. While it is possible to do *Continuous Integration* using other branching strategies, *Trunk-based development* eases and allows to get the best of *Continuous Integration*, as it supports faster feedback on the changes made to the code. Then we have the relation between *Infrastructure as Code* and *Version Control* where the first can be seen as a part of the latter. To address this, we decided to combine these practices into broader practices that would become our pattern candidates. In the end, we obtained five pattern candidates represented in Table II

Table II. : Pattern candidates, together with the works where we have identified the underlying practices (*cf.* second column) and related DevOps patterns that we find in the literature (*cf.* third and fourth columns).

Practice	Works	Teixeira *et al.* [Teixeira 2016]	CAP [Brown et al. 2022]
Deployment automation	[DORA 2022; Lwakatare et al. 2019]	Deploying new instances	Automate VM Deployment
Continuous integration	[DORA 2022; Pedra 2021; Lwakatare et al. 2019]	Continuous Integration	
Version control	[DORA 2022; Pedra 2021]	Version Control Organization, Reproducible Environments	
Comprehensive monitoring and observability	[DORA 2022; Lwakatare et al. 2019]	Auditability, Alerting	Correlation ID
Cloud Infrastructure	[DORA 2022; Pedra 2021]	Cloud, Scalling	Autoscaling, One Region, Overlay Network, Three Datacenters

3. ABOUT THE PATTERNS

We describe, in this paper, four of the five pattern candidates identified in the end of the previous section (*cf.* Table II), deciding to leave out of the scope of this particular paper the CLOUD INFRASTRUCTURE pattern. The five pattern candidates and their relationships are portrayed in Figure 1. Furthermore, we decided to include the PIPELINE pattern in this figure, even though we had not derived it directly as an independent practices from the literature. However, we did find it mentioned in the literature as important for implementing *Deployment Automation* and *Continuous Integration*, and segregating it from the others allows us to make the solutions of these patterns easier to use, namely in contexts where a pipeline does not need to be present. Finally, since the PIPELINE pattern is already very well explained in other works [Humble and Farley 2010; Yoder et al. 2018], we decided it was not worth to describe it again here.
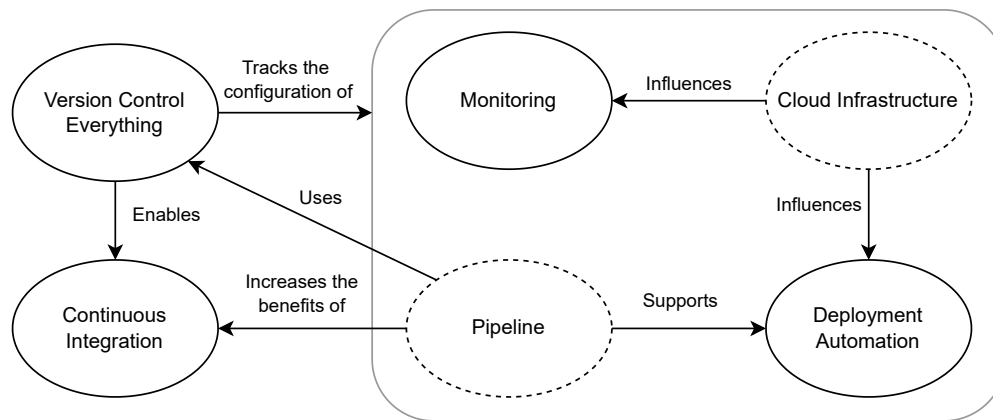
Fig. 1: Overview of the pattern candidates and their relations

We opted to follow a structure that is very similar to the patterns in the website *Cloud Adoption Patterns* [Brown et al. 2022] with only small differences. Our patterns contain the following parts:

—**Name:** An evocative name for the pattern.

—**Context:** Contains the context for the pattern providing a background for the problem.

—**Problem:** A question representing the problem that the pattern intends to solve.

—**Forces:** A list of forces that the solution must balance out.

—**Solution:** A detailed description of the solution for our pattern's problem.

—**Consequences:** The implications, advantages and trade-offs caused by using the pattern.

Similarly to the patterns by Brown et al. [Brown et al. 2022], we do not use explicit section headings to break these different parts but rely on some formatting conventions (*e.g.* the solution is always described after a *"therefore"*). Our pattern structure diverges from the ones on the website because we decided to identify the following sections explicitly and display their content in bullet points:

—**Related Patterns:** Patterns which are connected somehow to the one being described.

—**Metrics:** A set of metrics to measure the effectiveness of the pattern's solution implementation.

It is also worth noting that having a metrics section is not common in most patterns' structures. In fact, we could not find an equivalent section in the other works. However, we decided to add it since it supports a key principle of DevOps: continuous improvement. While it is often hard to measure the *consequences* of a pattern but they translate the actual benefits or liabilities of using that pattern, metrics are easy to measure but are a proxy to how well or thoroughly the pattern is implemented, or to which extent its consequences can be observed tangibly. They can be less meaningful but more actionable. A pattern's metrics should allow us to measure and plan improvements in the actual implementation of the pattern.

4.   VERSION CONTROL EVERYTHING

Nowadays, developing and operating software is a multi-disciplinary job requiring people with different skills to work on different tasks assigned to them. To fulfil these tasks, they might need to create or modify different files such as configuration files, and source code among others. As such, they work on multiple artefacts that depend on each other. One example of this is the relationship between the development team and the operations team. Even though their jobs have different goals, they need to work on the same artefacts.

How do you enable the simultaneous development and operation of multiple artefacts while keeping the consistency between them?

Usually, a version of a system is composed of multiple artefacts such as application code, dependencies, configuration files or infrastructure-related scripts. They may be worked on independently, by different members of the team that is collaboratively working on the system, but this may make it harder to maintain consistency among the different artefacts. To ensure it, we may need to add extra steps and checks to the developing process, slowing it down.

Furthermore, teams often need to reproduce an environment in the instance of disaster recovery or trying to scale the software horizontally. As such, they need a way to keep track of the multiple artefacts and system versions to guarantee reproducibility. On the other hand, guaranteeing that the system versions are reproducible does not come effortlessly. You have to have the extra effort to consistently and correctly keep track of the system versions and which particular version of the artefact is present in that system version. You may manually keep the records, which might work on smaller and simpler systems, but easily becomes unfeasible in more complex systems with many artefacts.

Another factor that comes into play is that teams often have to demonstrate the system's integrity for auditability purposes. Therefore, teams need to have the ability to pick any release of the system and quickly determine the version of every artefact used to create it. Furthermore, they also need the ability to compare two releases of the system and determine what changed. As such, they need a versioning strategy that also guarantees traceability. However, these two common requirements when done manually are labour-intensive tasks.

Therefore,

The use of a version control system for all production artefacts, including application code and its dependencies, application configurations, system configurations, scripts for automating the build and configuration of environments, and the configuration files for infrastructure specifications. In the version control system, teams must be able to query the current (and historical) state of their environments.

As a consequence, teams who employ this pattern have better disaster recovery capabilities when compared to teams who only version control some of the artefacts. Furthermore, it provides better auditability and traceability capabilities since teams must keep the history of all the artefacts on the version control system. This, also, enables better response to defects, since it provides the ability to roll back to a previous version when a critical defect or vulnerability is discovered.

As a downside, teams have to have the extra effort by defining, learning and using the version control strategy and maintaining a clear and consistent history of the artefacts. As such, they have to keep up-to-date documentation on how to use their version control system according to their strategy.

Related Patterns

—CONTINUOUS INTEGRATION: Provides the inputs for the build and testing of the application. Furthermore, it facilitates the process of fixing a broken build by providing the ability to roll back to a previous version.

—DEPLOYMENT AUTOMATION: The scripts and the configuration information of the deployment process should be tracked in a version control system to ensure it is possible to recreate any environment in the event of disaster recovery.

—CLOUD INFRASTRUCTURE: The cloud infrastructure configuration should be tracked in a version control system.

—MONITORING: The monitoring configuration should be tracked in a version control system.

—PIPELINE: The implementations of the PIPELINE pattern are strongly related to the version control system since the code that is supposed to run the pipeline is tracked in Version Control. As such, the pipeline implementation may depend on how you do Version Control. Additionally, the pipeline itself may be configured by artefacts tracked in the version control system.

Metrics

The following metrics may be collected to measure the effectiveness of the implementation of this pattern:

—The percentage of Application code in version control.

—The percentage of System configurations in version control.

—The percentage of Application configuration in version control.

—The percentage of Scripts for automating build and configuration in version control.

5.   CONTINUOUS INTEGRATION

A team of developers is working on the same system or related systems. The system under development has a main version of the codebase (known as trunk, main, or mainline) and, as developers work on the system, diverging changes might appear.

How often should developers integrate their work with the mainline to optimize for feedback?

Software systems are complex, and an apparently simple, self-contained change to a single file can have unintended side effects on the overall system. When a large number of developers work on the same system or related systems, coordinating code changes is a hard problem, as changes from different developers can be incompatible and it is difficult for team members to have visibility into what others are doing/changing. This results in problems from code conflicts to unexpected security concerns. Furthermore, the cost of merging diverging artefacts increases rapidly the more the artefacts to be merged differ from each other.

However, if you try to integrate your changes as fast as possible, you might introduce regressions if you are not careful. There is, also, the added complexity of having to break down features into smaller chunks of code instead of delivering them all at once. Here a decision has to be made if there are enough changes to integrate and get feedback, or if the changes are too small to be meaningful.

Therefore,

Developers do mainline integration as soon as they have a healthy commit they can share. As such, developers regularly integrate the code in short-lived branches or directly to the mainline/trunk. Ideally, a healthy commit equates to less than a day's work that has been tested and reviewed by a second person. To make the best of this, each integration triggers a build of the software and a set of quick tests to discover regressions. In the event that the build fails or regressions are found, developers must fix them immediately. This can be easily achieved with the usage of a CI/CD PIPELINE. The combination of these allows teams to integrate quicker, thus testing quicker resulting in a shorter feedback loop, thus making software delivery faster.

As a consequence, teams who adopt this pattern will integrate more often, which means they can test more often, resulting in quicker feedback loops. Testing often results in less likelihood of regressions occurring during development, and, thus, higher quality software. Furthermore, integrating often leads to team members having more visibility into what others are working on, reducing even the likelihood of regressions from diverging changes.

Also, integrating more often and with smaller chunks of code makes it simpler to solve the merges of diverging changes. This allows teams to allocate more time to the development of new features and, thus, making them more productive.

On the other hand, teams are obliged to maintain not only the code but also a test suite resulting in added complexity in the development process.

Furthermore, there is additional complexity in the development process by having to break down features into smaller incremental steps.

Related Patterns

—VERSION CONTROL EVERYTHING: The pattern presented here relies on the existence of a version control system as well as the ability to quickly fix broken builds by rolling back to a previous version.

—PIPELINE: Even though you technically do not need a pipeline to adopt CONTINUOUS INTEGRATION, pipelines potentiate the benefits of continuous integration, as they enable richer feedback about the changes you integrate in the code.

Metrics

The following metrics may be collected to measure the effectiveness of the implementation of this pattern:

—The percentage of code commits that result in a software build.

—The percentage of commits that are automatically tested.

—The percentage of successfully executed automated builds every day.

—The percentage of successfully executed automated tests every day.

—The percentage of builds available for testing.

—The percentage of acceptance tests' results that are available within a day.

—The time it takes between the build breaking and having it fixed.

6. DEPLOYMENT AUTOMATION

The development team has decided to deploy the software in a new environment, such as testing or production. As such they reach out to the operations team as this is a combined effort of the two. There is already infrastructure to support the environment.

How to deploy software in a reproducible, consistent and timely way?

Depending on the target environment and the type of software, the deployment process' complexity might range from simple like executing only a program to a very complex process containing various steps.

Manually deploying a system is an easier task when compared to automating its deployment. Depending on the system, it might be really time-consuming to automate its deployment when compared to manual deployment. Adding to this fact, deployments might be rare events for certain organizations. However, if the organization is trying to accelerate the speed with which they deliver features, they should expect to deploy more often, and, as such, deployments become more frequent.

Furthermore, when manually deploying, there is a higher risk of mistakes occurring during the deployment process, such as misconfigurations. This may lead to failed deploys when the new version of the software being deployed did not cause the issues. Therefore, debugging effort must be put into the failed deployment, which presents an obstacle to faster feedback. As such, the deployment process should be reliable. On the flip side, if your system is simple enough, the chance that manually deploying introduces problems might be low.

Finally, depending on the deployment complexity, different people deploying the software might do things slightly differently, obtaining different results and making deployments not reproducible. As a consequence, if the need comes to redeploy in the event of a disaster recovery scenario or even for manual testing and validation of the software, recreating the state of the environment might be extremely hard. However, if the system is simple enough, recreating the state might also be trivial.

Therefore,

The deployment process should be automated as much as possible.

The first step is to document the typical deployment process in a tool that both the development and operations have access to. From there, work on simplifying and automating the deployment process, implementing idempotence and order independence wherever possible. If available, the capabilities of the infrastructure platform should be leveraged. The goal is to make deployments as straightforward as possible.

As a consequence, having an automated process to deploy is usually faster than a manual one. As such, teams do not have to dedicate time to the deployment part, they can allocate it to other more valuable tasks, thus, increasing the productivity of the teams. Furthermore, teams no longer have to wait for someone to start manually deploying to find issues in the software changes related to the deployment process, shortening the feedback loop and allowing teams to fix these defects more quickly. On the other hand, automating the deployment process is not a trivial task. Depending on the complexity of the software, automating every step of the process might prove complex and time-consuming.

Related Patterns

—VERSION CONTROL EVERYTHING: The scripts and the configuration information of the deployment process should be tracked in a version control system to ensure it is possible to recreate any environment in the event of disaster recovery.

—CLOUD INFRASTRUCTURE: Cloud infrastructure often simplifies the deployment process by having built-in tools for that purpose.

—PIPELINE: It is possible to have DEPLOYMENT AUTOMATION without PIPELINE but the latter supports and makes it easier to do the former.

Metrics

The following metrics may be collected to measure the effectiveness of the implementation of this pattern:

—Number of manual steps in the deployment process.

—Time spent on deployment pipeline's delays.

7. MONITORING

A group of developers is planning the deployment or has already deployed the software system they are working on into the production environment.

How do you allow teams to understand and actively diagnose the health of their systems?

When working with complex systems that are already in production, there is the risk that changes or added functionality may introduce unexpected side effects that may have escaped the tests. However, changes are usually welcomed since they usually provide extra value to the clients.

In the instance of an outage or service degradation teams need to find what is causing the issues and how to fix them promptly. In other words, teams need to minimize the time to restore (TTR). However, teams want to avoid the need to restore a system altogether. As such, teams must detect possible service degradation and outages before they occur. However, these might prove a time-consuming task, requiring the allocation of multiple resources.

On the other hand, enabling the previously mentioned capabilities increases the development and operational complexity of the system. It is easier to design and develop a system without having observability in mind. However, it is harder to achieve the above capabilities if the system was not designed having observability in mind. Reworking the system to make it more observable might prove difficult and costly. Therefore,

Implement a comprehensive monitoring system that allows teams to monitor predefined metrics on the overall health of the system and the system state as experienced by the users. Furthermore, it should provide the tooling that enables teams to actively debug the system and make it observable. When the system detects an issue or predicts that one is about to arise, it should preemptively alert the teams of the issue.

You should monitor things like the state of the networks, the state of the infrastructure, the performance of systems, and key business and system metrics, among others.

As a consequence, it is possible to understand how the system is doing in production, getting faster feedback from the production environment, when compared to an approach where you would wait got problems to be reported by the end user. This allows the team to address problems before they impact the users resulting in a better quality service. Furthermore, with monitoring, it is possible to detect problems that would result in system downtime if not timely addressed.

Related Patterns

—VERSION CONTROL EVERYTHING Relies on the version control system to track the monitoring configuration.

—CLOUD INFRASTRUCTURE The monitoring strategy implemented is dictated by the use or not of cloud infrastructure

—The patterns by Albuquerque et al. [Albuquerque et al. 2022] provide further information on how to adopt a monitoring strategy.

Metrics

The following metrics may be collected to measure the effectiveness of the implementation of this pattern over some time:

—Number of alerts resulted in no action, or were marked as "Working as Intended"? (False positives).

—Number of system failures happened with no alerting, or alerting later than expected? (False Negative).

—Mean time to detect.

—Mean time to repair.

8. CONCLUSIONS

This paper proposes four new patterns for DevOps: VERSION CONTROL EVERYTHING, CONTINUOUNS INTEGRATION, DEPLOYMENT AUTOMATION and MONITORING. These patterns provided a guided method to facilitate the DevOps adoption while keeping some of the DevOps core principles, such as Continuous Improvement.

The proposed patterns were mined from existing literature, complemented grey literature, and with the authors' experience. Since the patterns are newly written, their description may not be complete or miss crucial details. In future work, we may further evaluate their accuracy, corroborating or refuting their existence, following some of our previous research [Sousa et al. 2022; Vale et al. 2022]. Finally, we do not believe that we suggest all the DevOps patterns and as such future work may consist of mining the remaining patterns for a complete pattern catalogue. Contrasting to the mining process we used here, future mining should look for patterns in the tooling used in the industry.

9. ACKNOWLEDGMENTS

REFERENCES

Carlos Albuquerque, Kadu Barral, Filipe Correia, and Kyle Brown. 2022. Proactive monitoring design patterns for cloud applications. In *Proceedings of the 27th European Conference on Pattern Languages of Programs (EuroPLoP '22)*. Association for Computing Machinery, New York, NY, USA.

K. Brown, B. Woolf, C. D. Groot, C. Hay, and J. Yoder. 2022. Patterns for Developers and Architects building for the cloud. (2022). `https://kgb1001001.github.io/cloudadoptionpatterns/` Accessed: 2022-01-03.

Ineta Bucena and Marite Kirikova. 2017. Simplifying the DevOps Adoption Process. (2017).

DORA. 2022. DORA research program. Available at: `https://www.devops-research.com/research.html`. (2022). `https://www.devops-research.com/research.html` Accessed: 2022-03-06.

Andrej Dyck, Ralf Penners, and Horst Lichter. 2015. Towards Definitions for Release Engineering and DevOps. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. 3–3. DOI:`http://dx.doi.org/10.1109/RELENG.2015.10`

Nicole Forsgren, Jez Humble, and Gene Kim. 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution.

Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. 2019. *2019 Accelerate State of DevOps Report*. Technical Report. `http://cloud.google.com/devops/state-of-devops/` Accessed: 2022-02-10.

Jez Humble and David Farley. 2010. *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley Educational, Boston, MA.

Jez Humble and Joanne Molesky. 2011. Why Enterprises Must Adopt Devops to Enable Continuous Delivery. 24, 8 (2011), 7.

Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. A Survey of DevOps Concepts and Challenges. *ACM Comput. Surv.* 52, 6, Article 127 (nov 2019), 35 pages. DOI:`http://dx.doi.org/10.1145/3359981`

Lucy Ellen Lwakatare, Terhi Kilamo, Teemu Karvonen, Tanja Sauvola, Ville Heikkilä, Juha Itkonen, Pasi Kuvaja, Tommi Mikkonen, Markku Oivo, and Casper Lassenius. 2019. DevOps in practice: A multiple case study of five companies. *Information and Software Technology* 114 (2019), 217–230. DOI:`http://dx.doi.org/10.1016/j.infsof.2019.06.010`

Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. 2016. An exploratory study of devops extending the dimensions of devops with practices. *ICSEA* 104 (2016), 91–99.

Tiago Maia and Filipe Correia. 2022. Service Mesh Patterns. In *Proceedings of the 27th European Conference on Pattern Languages of Programs (EuroPLoP '22)*. Association for Computing Machinery, New York, NY, USA.

Mauro Lourenço Pedra. 2021. DevOps Adoption: Eight Emergent Perspectives. (2021), 19.

Chris Richardson. 2018. *Microservices patterns: with examples in Java*. Manning Publications Co., Shelter Island, NY.

Chris Richardson. 2021. A pattern language for microservices. (2021). `http://microservices.io/patterns/`

Tiago Boldt Sousa, Ademar Aguiar, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. 2016. Engineering software for the cloud: patterns and sequences. In *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*. 1–8.

Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. 2015. Patterns for Software Orchestration on the Cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs (PLoP '15)*. The Hillside Group, USA, Article 17, 12 pages.

Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. 2018. Overview of a Pattern Language for Engineering Software for the Cloud. In *Proceedings of the 25th Conference on Pattern Languages of Programs (PLoP '18)*. The Hillside Group, USA, Article 6, 9 pages.

Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. 2022. A Survey on the Adoption of Patterns for Engineering Software for the Cloud. *IEEE Transactions on Software Engineering* 48, 6 (2022), 2128–2140. DOI:`http://dx.doi.org/10.1109/TSE.2021.3052177`

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2017. Engineering Software for the Cloud: Messaging Systems and Logging. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. DOI:`http://dx.doi.org/10.1145/3147704.3147720`

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018a. Engineering Software for the Cloud: Automated Recovery and Scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. DOI:`http://dx.doi.org/10.1145/3282308.3282315`

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018b. Engineering Software for the Cloud: External Monitoring and Failure Injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. DOI:`http://dx.doi.org/10.1145/3282308.3282316`

Carlos Manuel da Costa Martins Teixeira. 2016. Towards DevOps: Practices and Patterns from the Portuguese Startup Scene. (July 2016). `https://repositorio-aberto.up.pt/handle/10216/85711` Accepted: 2019-02-06T23:06:38Z.

Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. 2022. Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 69–79. DOI:`http://dx.doi.org/10.1109/ICSA53651.2022.00015`

J. W. Yoder, H. Washizaki, and Aguiar A. 2018. Delivering Fast with Confidence DevOps. *HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 7 (October)* 7, 7 (2018).