Lappeenranta-Lahti University of Technology LUT

LUT School of Engineering Science

Degree Program in Computer Science

Software Engineering

Perceived Benefits of Declarative Software Deployment - An Exploratory Case Study

25 May 2020

Oskar Sonninen

Examiner Associate professor Jussi Kasurinen
LUT School of Engineering Science
Lappeenranta-Lahti University of Technology LUT

Assistant professor Antti Knutas
LUT School of Engineering Science
Lappeenranta-Lahti University of Technology LUT

Abstract

Lappeenranta-Lahti University of Technology LUT
LUT School of Engineering Science
Degree Program in Computer Science
Software Engineering

Oskar Sonninen
Perceived Benefits of Declarative Software Deployment - An Exploratory Case Study
Master's Thesis
Lappeenranta 2020
60 pages, 7 figures, 5 tables

Keywords: Declarative deployment, GitOps, Imperative deployment, Infrastructure-as-code, Infrastructure-as-data

Declarative software deployment is a deployment paradigm that promotes usage of documents to describe the desired state of a system. Deployment is defined in documents but decisions on how to execute and achieve the state is left for the tools to decide. This is a distinct difference compared to imperative tools that require the execution to be explicitly written line by line. GitOps is flavor if DevOps that promotes usage of declarative deployment so everything from code to infrastructure are versioned in a single version control repository. This thesis explores if declarative deployment and practice like GitOps that heavily depends on it are perceived beneficial in practice.

Research was conducted as an exploratory case study to gather experiences from declarative deployment. Exploratory case study allowed interviews to pursue subjects that were not know beforehand and would only be revealed during the interviews. The research was conducted in three phases: literature review, interviews and analysis of the interviews. Literature review established a basic understanding about declarative deployment and GitOps, and was later used to produce GitOps reference and process models that were used to demonstrate GitOps in the interviews.

Interviews resulted in discovery of four topics that were perceived beneficial and promoted by declarative deployment. These findings allowed teams to work in more robust development and operation environments. GitOps was seen to support these practices and worth the initial effort required to adopt. The results were supported by the literature review conducted before.

From the results a conclusion was made that declarative deployment is perceived beneficial and GitOps is seen to support its core concept by promoting usage of version control and declarative tools for development and operations despite some initial overhead requirements.

Tiivistelmä

Lappeenrannan-Lahden Teknillinen Yliopisto LUT
LUT School of Engineering Science
Tietotekniikkan koulutusohjelma
Ohjelmistotuotanto

Oskar Sonninen
Koetut Hyödyt Deklaratiivisessa Käyttöönotossa - Eksploratiivinen Tapaustutkimus
Diplomityö
Lappeenranta 2020
60 sivua, 7 kuvaajaa, 5 taulukkoa

Hakusanat: Deklaratiivinen käyttöönotto, GitOps, Imperatiivinen käyttöönotto, Infrastruktuuri koodina, Infrastruktuuri datana

Deklaratiivinen ohjelmistojen käyttöönotto on käyttöönotto paradigma, mikä korostaa kuvausdokumenttien käytön tärkeyttä järjestelmien tilan määrittelyssä. Järjestelmän tila määritellään kuvaus dokumentissa, mutta päätös miten haluttu tila käytännössä saavutetaan jää käytetyn työkalun vastuulle. Tämä on perustavanlaatuinen ero imperatiivisen käyttöönoton kanssa, missä jokainen käyttöönoton vaihe pitää kertoa eksplisiittisesti.

GitOps on DevOpsin painotus, mikä korostaa deklaratiivisia toimitapoja, jotta kaikki koodista infrastruktuuriin on versioitu ja löytyy yhdestä versionhallintatyökalusta. Työn tarkoitus on tutkia, millä tavalla deklaratiivinen käyttöönotto ja GitOpsin tapaiset toimintatavat on koettu olevan hyödyllisiä käytännön tasolla.

Tutkimus on tehty eksploratiivisena tapaustutkimuksena, missä kerättiin kokemuksia ja ajatuksia deklaratiivisesta käyttöönotosta. Eksploratiivinen tapaustutkimus mahdollisti ennalta tuntemattomien osioiden tutkimisen aiheesta, mitkä saattavat paljastua vasta haastattelun aikana. Tutkimus suoritettiin kolmessa vaiheessa: kirjallisuuskatsaus, haastattelut ja haastattelujen analyysi. Kirjallisuuskatsaus pohjusti deklaratiivista käyttöönottoa ja GitOpsia, sekä näiden konsepteja. Tätä tietoa käytettiin myöhemmin haastatteluiden pohjana, sekä tuottamaan GitOpsista prosessi ja referenssimallit, mitä käytettiin GitOpsin esittämiseen haastatteluissa. Analyysi tehtiin etsimällä haastattelujen nauhoitteista toistuvia aiheita, joista tehtiin myöhemmin johtopäätökset.

Haastattelujen perusteella löydettiin viisi aihetta, mitkä olivat koettu hyödyllisiksi ja joita deklaratiivinen käyttöönotto tuki hyvin. Kirjallisuuskatsaus tuki havaittuja tuloksia, mitkä mahdollistivat työskentelyn kestävissä kehitys- ja operointiympäristöissä. GitOps koettiin tukevan havaittuja käytänteitä ja sen tavoitteleminen oli hyödyllistä tarvittavasta alkupanos-

tuksesta huolimatta.

Tuloksista tehtiin johtopäätös, että deklaratiivinen käyttöönotto on koettu hyödylliseksi ja GitOpsin nähdään tukevan sen ydinkonsepteja korostamalla versionhallinan ja deklaratiivisten työkalujen käyttöä kehityksessä ja operoinnissa.

Acknowledgements

I would like to thank my supervisor Jussi Kasurinen for guiding me through the thesis, providing feedback and guidance on a days' notice has been invaluable. The help to close this final chapter in my studies and finally graduate, does mean a world to me.

A big thank you for all the companies and people attending the interviews. Without your insight this thesis would not have happened. A special thank goes to my own supervisor Ilkka who always kept my graduation as an important goal at work.

Some time has passed since I stepped through the main doors of my university for the first time. The time in the university have been the best of my life, with so many dear friendships made during that time. Years in Lappeenranta were the time to grow as a person and figure out what kind of person I want to be. Those are the years that I will never forget.

The last year has at times been a roller coaster of emotions. At times it felt that this day will never come but as the spring progressed the sun got brighter. A big thank you goes to all my friends for the shared years and for the years to come. Everything would be so much less without you. Especially I would like to thank Jesse, Petteri and Jan for providing input and support for this thesis.

Without a doubt in my mind the biggest thanks still go to my parents who have supported me through all these years and had faith for this day to come, even at times when I did not. For them I am eternally grateful for everything. Finally, my grandparents who have always steered my path to the right direction, thank you for the foresight.

Helsinki, 25.5.2020

Oskar Sonninen

"DevOps is solving operational challenges with development tools."

- Thesis interviewee

Contents

Abstract

Tiivistelmä

Acknowledgements

Abbreviations 8

1 Introduction 9

2 Deployment and GitOps 13

2.1 Deployment paradigms . . . 15

2.1.1 Imperative . . . 16

2.1.2 Declarative . . . 17

2.2 From DevOps to GitOps - Possible best practice? . . . 18

2.2.1 DevOps . . . 18

2.2.2 GitOps . . . 22

2.2.3 GitOps reference model and process . . . 26

3 Research problem and methodology 29

3.1 Research questions . . . 29

3.2 Research methodology . . . 31

3.2.1 Case study . . . 31

3.3 Research process . . . 33

3.3.1 Literature review . . . 33

3.3.2 Interviews 34

3.3.3 Data analysis 35

3.3.4 Questionnaire 36

4 Results **37**

4.1 Tooling 37

4.2 How declarative deployment has improved development, deployment and operations 39

4.2.1 Documentation and Visibility 39

4.2.2 Immutability and Auditing 41

4.2.3 Resource allocation 42

4.2.4 Environment creation and management 43

4.2.5 Target state and deployment repeatability 45

4.3 Perception of GitOps 47

5 Implications and Discussion **50**

5.1 Research questions 50

5.2 Limitations 53

6 Conclusion **54**

Bibliography **55**

Appendix A: Questionnaire template

Abbreviations

AWS Amazon Web Services

CD Continuous Delivery

CDe Continuous Deployment

CI Continuous Integration

CRD Custom Resource Definition

CRUD Create, Read, Update and Delete

DevOps Development and Operations

Faas Function as a Service

GCP Google Cloud Platform

SaaS Software as a Service

SDLC Software Development Life Cycle

SecOps Security and Operations

SQL Structured Query Language

VCS Version Control System

VM Virtual Machine

1 Introduction

Technological advancement in software development have taken enormous leaps during the past two decades. General availability of the Internet, ever faster hardware and the emergence of cloud computing brought seemingly infinite pool of resources for the customers. As the technology has advanced it has also enabled new possibilities on how we develop software. Development teams can nowadays work on the same project from different cities and even countries (Sutherland et al., 2007,Cusumano, 2008), many of the formerly self-hosted applications can now be bought as a service (Benlian et al., 2009) and tools for software development are better than ever before. The challenges which we cannot solve with technology alone is the software development itself. Processes, practices, and methodologies are not something we can just buy from a store shelf and start using (Kromhout, 2018). This is a problem of how we organize our work practices and as such is more a discussion of concepts and values than a technological one. This thrive for better practices and their popularization is called software engineering (Society et al., 2014).

> *"Software engineering is essentially a human activity, not just a technical matter of technology, and yet because of the emphasis placed on the technical aspects of software production, most software engineers have never considered software construction in this light"*. - Fernando Capretz, 2014

This one statement underlines that although software engineering leverages technologies that make it disciplined, it is still a practice of human origin and for this reason should be treated as one.

Software engineering as a discipline and recognised field of study began in the 1960s when developer community admitted that they had a software crisis in their hands and something needed to be done about it (Wirth, 2008). Since then software engineering has had on long journey along the evolutionary path where a common nominator has been the strife to do things more efficiently. Quite often this has meant ability to change; let it be a change in practices, tools or requirements. Defined processes started from waterfall in 80s with the idea to execute the project step by step in logical units. In the end, these units would be assembled

to the end product (Stober et al., 2010). After waterfall, during the 90s, there was a shift to more lightweight and agile methods such as Unified Process, LEAN and Scrum during the 90s (Hüttermann, 2012). One of the core ideas to solve problems with waterfall was to handle development in smaller cyclic iterations where earlier cycles end acts as an input for the next cycle (Stober et al., 2010). This kind of thinking culminated in the form of Agile Manifesto in 2001 which stated the best practices to achieve agile development model (Martin, 2019). Later in 00s when the development process was solved (in theory), people started to think next steps. This gave birth to DevOps (Development and Operations) which has continued to be the current industry best practice and goal for many organizations (Hüttermann, 2012). Core idea of DevOps is to bring different groups closer together and leverage automated processes (Hüttermann, 2012).

DevOps is often executed with suitable tooling but as Hüttermann points out this is not the core of DevOps.

> *"Some people prefer to think about tools instead of people and processes. Without respecting other people and understanding processes, these people merely introduce tools."* - Hüttermann, 2012

The two keywords are processes and automation which are the enablers of DevOps and better software engineering. Processes define how and what things should be done, and tooling is used to automate these processes (Hüttermann, 2012).

> *"Processes are more important than tools, but tools are still important, especially for automating activities along the delivery process."* - Hüttermann, 2012

Software development being automation of computational workloads in its essence has tried to leverage itself to automate parts of its own processes. It began with formalizing code management with version control, moved to test process automation, package management, integrations, software deployment and now finally it is reaching automated infrastructure deployment and management. This new practice is called infrastructure-as-code and it is part of everything-as-code movement (Vargo, 2018).

Everything-as-code could be best described as a state of mind where everything in software deployment is defined in code (Vargo, 2018). A developer does not make changes directly into a deployment but changes are propagated through source files and templates (Vargo, 2018). This should also means that not single change should end into production without a reviewed by a another person and a CI/CD pipeline (Vargo, 2018; WeaveWorks, 2017a).

Infrastructure-as-code has had on impact on how infrastructure can be deployed by describing the desired infrastructure (Artac et al., 2017). There is some discussion related to the term as some say it should not be called infrastructure-as-code as much as infrastructure-as-data (Fisher, Bret, 2019). The arguments is made because data describes what is wanted but leaves how it is achieved for an executor that is code. This is an interesting distinction on how the discussion is approached but does not necessarily change the outcome.

The new data driven way of describing what is wanted has given life to new practices like GitOps, which leverages the possibility to describe infrastructure-as-data and puts this description in the same bucket with code for all to be seen (WeaveWorks, 2017b). A core principle of GitOps is to use declarative deployment which describes the environment and guarantees that if the same declaration is deployed it will generate functionally same environment as the previous deployed one (Limoncelli, 2018). This is a quite similar construct with the concept of containers which can be considered as immutable and portable application packages able to be deployed time after time with same outcome (Pahl et al., 2019).

GitOps has been claimed to have several benefits: faster development, operation visibility, compliance and audition visibility, and security guarantees (WeaveWorks, 2017b; Chen, 2015). These claims are said to happen when no directs changes can be introduced to the system, all code and infrastructure changes are represented as trackable artifacts, deployments are done using a pipeline and drift from canonical state is automatically managed (WeaveWorks, 2017b). GitOps forces these practices by heavily utilizing version control, CI/CD pipelines, declarative tooling and comprehensive monitoring (WeaveWorks, 2017b). Version control enables change tracking and is a good location to force reviews for code quality. CI/CD using declarative tools enable extensive testing, repeatable deployment process and drift cor-

rection. Monitoring and logging tools combined with drift correction[1] enable a lockdown of execution environment that prevents external tampering while still being configurable and observable (WeaveWorks, 2018).

Any of the principles above are not new if compared to traditional DevOps (Hüttermann, 2012) but what GitOps emphasizes is visibility from development to production with minimal amount of manual interruptions or tampering directly into the system (WeaveWorks, 2017b). If the above principles are implemented, GitOps can in its most extreme case enable whole deployment of an application to be rolled out from a single commit with networks, VMs, functions, load balancers, access control, monitoring, software etc. in place. No manual configuration involved. This also means than in the case of unexpected errors like hardware failures the whole system can recover based on the input from monitoring the state inspection (WeaveWorks, 2018). All this requires commitment to commonly agreed workflows and possibly learning new tools from the whole team but the benefits GitOps offers can be worth it in many cases (WeaveWorks, 2017b).

In this thesis we will explore how far automation of software deployment and management can be led with declarative deployment and practices like GitOps, and what might the perceived benefits these have had for software development in practice.

[1] Drift correction is the attempt to return to a target state from an undesired changed state

2 Deployment and GitOps

Before one can understand software deployment, and why this is something worth a closer look, an understanding of software developments process is required. Software development life cycle (SDLC) defines certain step than are distinguishable from each other and serve a specific purpose in the process to deliver software (Kneuper, 2018; Jalote, 2005). There has been multiple models and methodologies during the decades of which in retrospective some have been more successful than others (Kneuper, 2018, Jalote, 2005).

All software life cycle models have common themes but there is not a single definitive model that can be said to be correct, neither there should be. Different models focus on different aspects of life cycle and delivery model, and specific steps depend on what aspects they emphasize or see as the most important parts. Waterfall defines six phases: requirement analysis, design, implementation, integration, testing and rollout (Kneuper, 2018). Rational Unified Process says four: inception, elaboration, construction and transition (Kneuper, 2018). Microsoft's security development life cycle states seven phases: training, requirements, design, implementation, verification, release and response (Kneuper, 2018). Agile methods define phases as do the older models but in addition wraps these phases to short cycled iterations that are repeated often in few week cycles (Kneuper, 2018; Stober et al., 2010). Operating in iterative cycles is used to give time to discover requirements that are not known in the beginning of the development or are discovered trough development feedback (Kneuper, 2018, Hüttermann, 2012). The difference between waterfall and agile processes iterative nature and feedback loop are shown in figure 1. As it can be seen from above the models and methodologies can have different terminology, phases and levels of iterations but all of them have the same goal; deliver quality software.

Every development methodology has a phase that includes software delivery or deployment. In agile development this phase will occur relatively often (Hüttermann, 2012) compared to older models like waterfall (Kneuper, 2018). Especially in the pre-2010 and earlier separate development and operations teams were common which presented friction when moving from development phase to operations. This hand-off is widely considered problematic as responsibilities and focus changes (Hüttermann, 2012). To patch this problem,

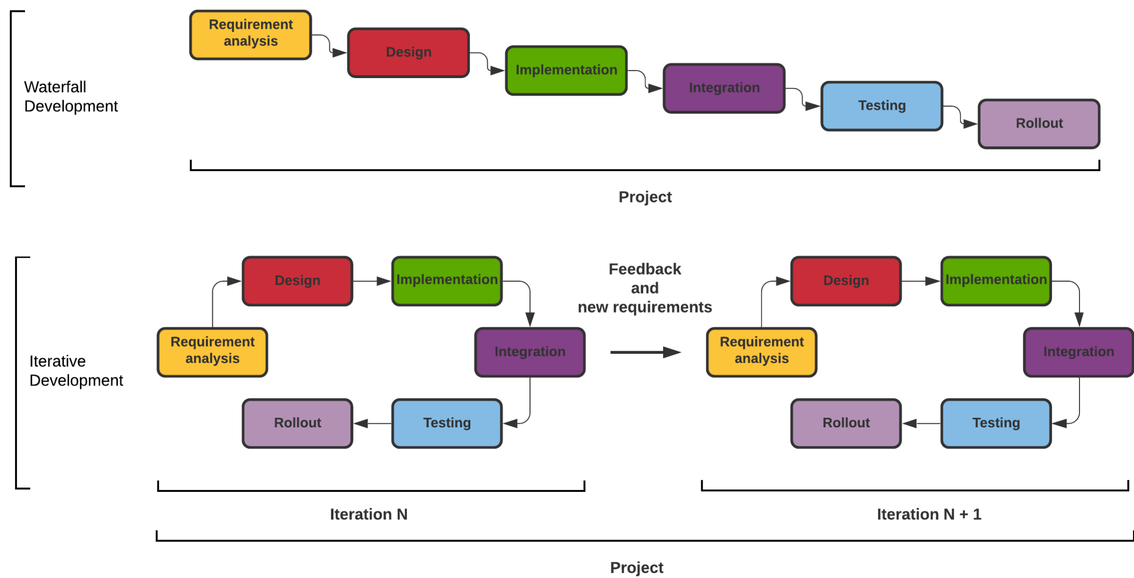

Figure 1: Difference of Iterative and Waterfall processes. (Based on Kneuper, 2018 and Hüttermann, 2012)

DevOps was introduced with a goal to bring development and operations together or at least closer (Rossberg, 2019). DevOps is a practice that promotes close collaboration between development and operations to make software delivery from developers computer to end-user one seamless process (Hüttermann, 2012).

Traditionally software deployment has been the part between development and operations teams (Hüttermann, 2012). As mentioned before this has been a complicated part in the past and DevOps was introduced as practice to bridge the gaps. Historically software deployment has been quite labour intensive work (Leppänen et al., 2015). Deploying a package has often required expert knowledge about the software, a maintenance window, figuring out related software and dependencies, updating documentation and maybe even deploying new machines to host the new versions of the service. Only after this the software package can be manually deployed e.g. by logging in to the server via ssh, copying the package to correct folder, updating related configurations and restarting the service. This kind of work is very imperative where the deployment is stated step by step, each step containing a place for error (Leppänen et al., 2015). Scripts have been used for decades to automate parts of deployment but often these were made application specific and not generally available or applicable.

2.1 Deployment paradigms

C, Python and JavaScript are imperative languages where programmer defines line by line what they want their code to do and functional languages like Haskell can be used to describe what the programmer wants to accomplish without describing the control flow (Jaroslav, 2008). Likely the most well-know declarative language we all know but not necessarily consider as one is Structured Query Language (SQL). In SQL one defines what they want from the database but take no interest on how the query executes to get this data. A programmer does not tell in a SQL query how a table filtering is done but instead states that she wants all the rows which a specific value in a specific column, how the filtering is actually executed is left completely up to the database engine to decide.

In imperative programming programmers need to implement the data processing and control flows by themselves. They need to define variables, loops, branch conditions etc. to be able to realize the purpose of the software. In declarative languages the programmer does not design operations in the same way but uses the constructs provided by the language to implement the control structures and expose them trough functions. Imperative programming gives more control over how things are implemented and declarative programming leaves some of the implementation decisions for the language (Ari et al., 2014).

In a similar way as software can be written using two different paradigms, the software can also be deployed with two diffident approaches: imperative and declarative (Attardi et al., 2018). These approached differ from each other on how they deployment is done from the users point of view. In imperative deployment every step of the deployment is done by running individual commands defined by the implementer and in declarative deployment only the desired state of is given (Attardi et al., 2018). For previous reason imperative deployment requires someone or something (e.g. a programmer or a CI/CD pipeline) to executes deployment commands one by one to each instance separately to reach the desired state of the environment. Declarative deployment describes in a document what is wanted, the document is then passed for a tool which will execute and take care of how the desired state is reached based on the model expressed by document. This kind of deployment should have at least following benefits: reproducibility when every separate deployment produces identical state (Sayers, 2017), testability as declarations can be linted and injected with tests (Wurster et al.,

2018) and verifiability as declaration file acts as documentation of the environment.

It is worth a note that imperative and declarative deployment should not be considered exclusive as their co-operation can achieve things that would be harder using only one of the paradigms without the other (Sayers, 2017). For example imperative deployment workflow can be used as a trigger that lets declarative parts to check for new configuration from the repository and reconfigure itself to a new desired state. This can be easily seen from many CI tools that invoke a declarative deployment. Many of the tools available like chef[2] can do both configuration and deployment management or at least allow both ways through templates, workflows and hooks.

2.1.1 Imperative

One fundamental implication of imperative deployment is that it does not have a model. The absence of the model also means it cannot maintain a state because there is no state to beginning with. The individual statements forming the imperative deployment are bundled together into a single "arbitrary" workflow which will get executed but which cannot be interpreted to form a model. This makes imperative or workflow deployment essentially a process of steps that gets executed in specific order, a one-way set of rules that cannot be deduced from the deployment itself. This is not necessarily a bad thing by itself as it gives a lot of control for the process but at the same time it can lead to a situations that are harder to maintain when the amount of components grows larger. (Sayers, 2017; Breitenbücher et al., 2017)

An analogy for imperative deployment can be made to a checklist. The checklist says explicitly what to do and in what order and it is up to the writer to make individual items in the list to make sense as a whole. If the goal is to keep target servers package on the newest version the checklist could be something like below and it could be run manually by a person or automated in a workflow runnable by a machine:

1. Copy software package with scp to target server.

   ```
   "scp <package> <target\>:/packages"
   ```

[2] https://www.chef.io

2. Login to target server using ssh.

   ```
   "ssh target"
   ```

3. Extract the package to /deployment folder

   ```
   "tar -xjvf /packages/package /deployment/"
   ```

4. Restart the service

   ```
   "service <service> restart"
   ```

The interesting and possible challenging aspect of this kind of checklist process is; what would happen if the same package is already deployed to the target server and the process is run again? Should it not deploy the package and restart the service if there is no changes or should it do it anyway? What if the restart takes a long time and restart will not be wanted if the package is up-to-date. Considering these situation can quickly become time consuming and error prone.

2.1.2 Declarative

Declarative deployment is tooling dependent practice as it utilises templates to describe the wanted deployment state. Templates are commonly written in a well-know formats like yaml, ini or xml. The template file is read by a tool written in another language e.g. Go that inspects the file, forms a model based on the template and then executes the model. This process can be referred with read, diff, plan and execute which in practice follows CRUD principle (Boyer et al., 2018). Read phase reads the template file and converts it to a model, the model defines the wanted state of the system, diff phase pulls the current state from the target environment and generates a difference model from the two systems, plan phase translates the diff model into single operations/statements and finally execution phase runs each of these operations (Boyer et al., 2018). It is worth mentioning that while declarative deployment is a different paradigm to deploy software it is also just an higher abstraction provided by tooling that will on its lowest level run imperative commands in the target system to fulfill desired state for the deployment. This can also mean that using these tools one gives away some of the more fine grained control.

Compared to the earlier example (p.16) about imperative deployment and checklist, declarative deployment would look quite different. Our deployment command could consist of a deployment file and a command which executes the tool to deploy. The deployment file would contain information like what package to deploy and where. Running the tool could look like `"deploy_tool deploy <deploy_file>"` and the tools job is to decide how to do the deployment whether new components need to be created, services restarted, network updated etc.

2.2 From DevOps to GitOps - Possible best practice?

In this section we go trough DevOps and its core concepts. We will explore what the literature says about the benefits of DevOps and how adopted its practices really are. We will also introduce the concept of GitOps, a relatively new flavor of DevOps that emphasizes usage of version control for development but also for operations. We will go through how GitOps through heavy utilization of declarative deployment tools enables operations straight from the source control and how this operation is done using pull and push methods.

2.2.1 DevOps

Before the DevOps practices there was often a distinct separation of people who developed the software and people who operated it. There was no single pipeline where to put software in and it would appear on the other end ready to be used. The separation of development and operations brought friction into the whole software development life cycle as the two side could blunder themselves into a blame game as they did not try to solve the problems together. One simple way to present the conflict between two ends is to say development has a need for and operations has a fear for it. To counter this problem DevOps introduced in DevOpsDays conference in 2009. Naturally the concepts and ideas of DevOps started long time before the term formalization of the methods can be given for the conference. (Hüttermann, 2012)

Definition of DevOps are many but for the rest of the thesis we will use a recent metastudy about DevOps which synthesises several definitions as follow:

> *"DevOps is a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices."* - Jabbari et al., 2016

Above definition does not mention anything about tooling or products because DevOps is a practise of aligning processes and workflows. DevOps practices are often paired with brilliant tooling but tools are secondary to practices. Practices are described in processes and processes succumb to organization culture and because culture cannot be bought we cannot buy DevOps (Hüttermann, 2012). Development culture can only be built from within the organization as it forms the basis for everything else in the development (Hüttermann, 2012). Importance of organization culture is brilliantly summarized in the following quote.

> *"Culture eats process for breakfast"*. - Bjork, 2017

This remark underlines that no matter how well defined processes or structures are present, the fundamental ways of working will always win in the end.

DevOps is often done with agile frameworks like Scrum or Lean which guides the whole development life cycle (Stober et al., 2010). DevOps does not state a need to use a specific agile framework (or a framework at all) but Scrum has become a popular framework to guide it (Sharma et al., 2016). Many other methods like Extreme Programming, Agile Unified Process and Lean exist but are not as popular (Sharma et al., 2016) which can be caused due to the fact that they embrace different aspects of culture. Scrum is leaning towards development part of the processes and Lean is considered to focus more on process optimization as whole (Poppendieck et al., 2012).

DevOps promotes continuous integration (CI) and continuous delivery (CD) using tools that automate processes like building, testing and delivery. They also enforce that the software is always build the same way in a standard environment which catches most of the *"It worked on my machine"* problems before reaching production. Usage of CI/CD should firstly be considered as practices and secondarily as a set of tools to automate the process (Hüttermann, 2012). Last decade has spawned dozen of these tools. Some are provided as traditional

software that can be deployed to on-premise systems (Jenkins) and some are provided as a service (GitLab). Bamboo [3] has only on-premise installation options, and GitHub [4] and GitLab [5] offers their product both as a service and on-premise installations. Technically both continuous integration and continuous delivery are separate practices and continuous delivery could be done without CI, though in almost every case CD becomes useful only after CI pipeline is working. For this reason, CD should be considered as an extension to CI rather than a completely separate practice (Pittet).

Depending on the source continuous delivery can be divided into two parts; continuous delivery and continuous deployment (CDe) (Shahin et al., 2017). If this separation is used then continuous delivery is the ability to deliver software on will but it is not deployed to production automatically. Continuous deployment is a practice that allows deployment to production when new deliverables are available aka they have passed integration and delivery parts (See figure 2.). This can be confusing as quite often continuous delivery and continuous deployment are used interchangeably and tooling able to do continuous delivery can also do the deployment part (GitLab; Jenkins). For example articles in LinkedIn [6] and Atlassian [7] have different opinions about what are the exact differences.

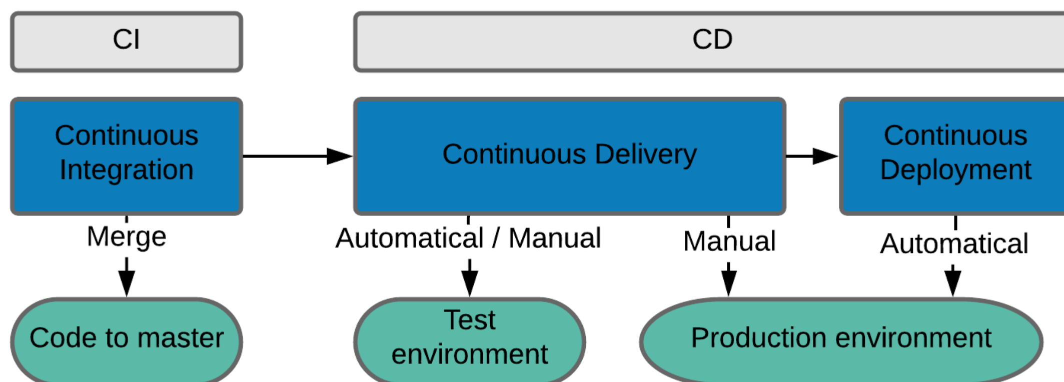

Figure 2: Relation of CI, CD and CDe pipeline (Based on Shahin et al., 2017)

DevOps has gotten huge attentions since its birth. Google Trends show that DevOps as a

[3] https://www.atlassian.com/software/bamboo
[4] https://github.com
[5] https://about.gitlab.com/
[6] https://www.linkedin.com/pulse/here-why-continuous-delivery-deployment-different-uday-kumar
[7] https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

search trend was bit more quiet for a few years after the terms introduction in 2009 but since 2014 the interest for DevOps has steadily risen as seen in figure 3. Search to ACM and IEEE confirms this: ACM provides 261 hits for DevOps since 2011 and as many as of them 253 are published after 2014 and IEEE gives 347 since 2014 of 355 in total. 2011 was also the first year when both databases gave DevOps any results. Surveys done about DevOps confirm that it has gotten more traction behind it year after year and more companies are adopting its practices into their workflows (Forsgren, 2019).

Adoption of DevOps can clearly be seen from the Accelerate State of DevOps survey that has been running for six years and is probably one of the best and most comprehensive surveys related to adoption of DevOps worldwide (Forsgren, 2019). The survey responses are used to give a data backed report about DevOps adoption and what benefits it can yield if wielded properly. It is also worth to notice that traditionally slow moving industry sectors like banks have adopted DevOps practices successfully by adopting agile practices, and automating testing and deployment (Ravichandran et al., 2016; Forsgren, 2019). Studies researching delivered competitiveness based on software development maturity also confirm findings that many DevOps practices (not limited to) give real life benefits for the whole organization (Lesser et al., 2016).

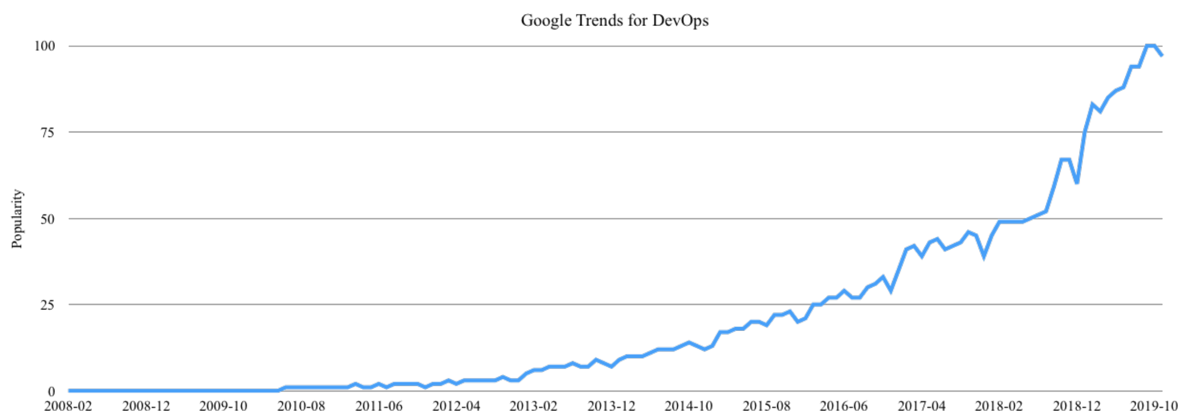

Figure 3: DevOps popularity in searches

DevOps has spawned a variety of different ”flavors” and industry has not yet agreed on how to refer to all of these flavors but a likely competent for the term is xOps where the x basically stand for flavor (Leslie, 2020; Glauser, 2020). Terms like CloudOps, SecOps, DataOps and GitOps are already present and the list is most likely to expand in the 20’s.

2.2.2 GitOps

The term GitOps was tinkered by a company called Weaveworks in August 2017 and the term derives from the well established DevOps practices (WeaveWorks, 2017b; Limoncelli, 2018; S. Aravena, 2018). Very little research exist at the current moment which can be a direct impact from the fact that the idea is relatively new. Google Scholar results 20 hits since 2015 of which only 2 are focused to GitOps and published in a journal. There is however a steady increase in industry interest as multitude of blogs and articles signal of a movement behind GitOps. Same observation can be made from Google search trends in figure 4.

The components of GitOps have existed for a long time in the same way as practices of DevOps preceded the term. It can be argued that DevOps already includes all the required practices to do GitOps which is probably true but practices emphasize different aspects of the process. WeaveWorks puts this nicely in their original GitOps blog post.

> *"This post talks about GitOps, which is 90% best practices and 10% cool new stuff that we needed to build."* - WeaveWorks, 2017b

GitOps can easily be associated to be limited to Kubernetes but this is not the case, Kubernetes is only a good implementation of tooling that can be used with GitOps (Fisher, Bret, 2019).

GitOps has several focus areas that justifies its existence besides DevOps and other xOps introduced during the last decade. Where DevOps aimed to break walls between development and operations, GitOps was to establish one place to manage the project for both development and operations. They do this by shifting the focus left from CI/CD (which is the heart of DevOps) in the pipeline and state that everything should be in version control like git. GitOps argues; **the source control would present the one and only source of truth about your system. All changes and management goes through it.** As mentioned earlier this is not blatantly new idea but after infrastructure-as-code became possible, the industry had tools to really put everything needed to deploy an application to production without manual involvement after the source was committed into a repository.(WeaveWorks, 2017b, 2018)

Thinking of source control as the single source of truth is important for GitOps (WeaveWorks, 2017b). This allows GitOps to manage and observe everything from one central lo-

cation which can be versioned and monitored (WeaveWorks, 2018). Trough a commit review process this single source of truth acts as a broker that can tell everyone what their project contains. In the same way to make a change the change need to be made only to a single location to be populated everywhere and made available for everyone.

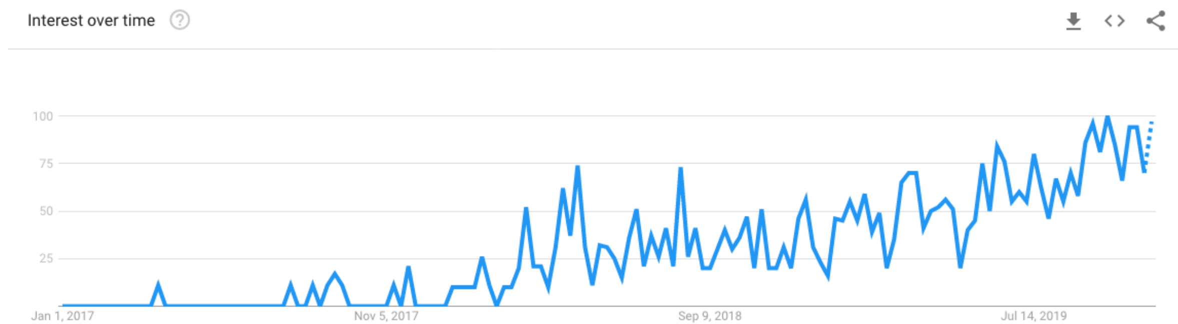

Figure 4: Search trends of GitOps

An interesting note from WeaveWorks about infrastructure-as-code especially with declarative tools is that we should not speak so much of as-code as we should talk about as-data. As-data would describe the case much better because declarative descriptions do not really hold any statements as code does. Code is essentially an algorithm that tells how data is processed but the structured documents used with declarative deployment only describe what kind of infrastructure one wants. To handle only data in the documents also give some benefits over code, linting data is more simple than code which needs to be more context aware. Data is also easier to test as data does not have a state is the same way as executed code does. (Fisher, Bret, 2019)

Putting everything in source control has several implications that make GitOps profoundly interesting. It strongly encourages using pull requests (PR) which can be considered as code review process. Once a developer has finished a feature he files a pull request from his own branch to start review process to merge changes to master. At this stage there is a possibility for discussion about the related changes, accept or reject the commit. Pull request can also be updated if needed based on the discussion. Important best practice is that another developer or project owner is the one making final decision if the request is to be merged. This makes version control system (VCS) the single source of truth and all the operations on code and environments goes through here. (WeaveWorks, 2018)

Another implication from version controlling is that every modification is discoverable which should make change tracking much easier (WeaveWorks, 2018). No changes are dependent on someone's memory to remember if something was done or if it was documented. This traceability can be used in the project management to link every change in the system directly to e.g. SCRUM backlog items and support tickets. This makes tracking the whole projects and its features easier as it gives a single location to look if a feature has been shipped, is still in a quality assurance environment or possibly in a non-merged feature branch. All the changes can be searched directly from VCS logs which can linked project management tool to tell why the change was done.

Because the direct access to the environment is prevented the importance of monitoring and logging capabilities that can either export the data or expose in read only mode are definitive requirements for GitOps. In a case there is a problem one cannot access to the deployment itself check the state of the deployment; state can only be observed through logs and monitoring. (WeaveWorks, 2019)

Above descriptions of GitOps characteristics points out what makes GitOps fundamentally observable. Observability comes from the version control and PR process which together with a deployment pipeline act as gatekeepers ensuring that no changes can be introduced without being approved and tracked to version control. This implies that there should not be a separate need to verify which software version is running, what the software consists of and who have access to the environment. By being observable GitOps can also support other xOps flavors like SecOps which focuses implementing security features to DevOps processes (Ur Rahman et al., 2016).

As with everything else GitOps does have its limitations on how far down the infrastructure stack can go (Fisher, Bret, 2019). There will always be a limit how far down one can go with declarative tools as there needs to be something to run the commands. Limits can also be imposed by how much external information the task needs or rarely it occurs. Things like creating a billing accounts requiring a credit card is something that occurs so rarely and is enormously hard to implement that it is not feasible to do declaratively.

Pull and push - Two ways of GitOps - Separation of Concerns

There are two main ways of doing GitOps from process perspective: push and pull. The push and pull refers to how changes should be transferred from version control for the state managing software. Using push process there is security concerns to be evaluated which can be mitigated using the pull process that introduces a separation of concerns on which parts of the pipeline do code integration and management, and which deploys the code. (WeaveWorks, 2017a)

In a push deployment a CI/CD pipeline builds and tests the software but it also deploys the software into an runtime environment (WeaveWorks, 2017a, 2018). This is done by telling the pipeline what to deploy, where to deploy and give the pipeline access to the said runtime environment or command the tool doing the deployment. This has been the way how many pipelines and tools have operated when CI/CD pipelines is used as a trigger and orchestrator of the whole deployment process. This also means that without constantly polling the target runtime environment there is no knowledge about the current state of the deployment.

In pull process there is no external trigger which would tell the runtime environment what to do. Pull process listens to changes in external version control or other artifact repository and only when a change is seen it will pull these changes and execute them as seen fit. This separates the management and building of artifacts from the runtime. Also by being part of the runtime environment the pull method has a much better ability to detect changes in the deployment. This enables pull process to better suited for state management than push process. (WeaveWorks, 2017a, 2018)

In a push process the CI/CD in addition to building the software and deployment artifacts, the pipeline pushes the changes forward. This introduces at least some security concerns as the CI/CD tool needs access to the runtime environment (WeaveWorks, 2017a). By allowing pushes to happen, one introduces a remote control that can change the runtime on a whim (Fisher, Bret, 2019). This is might be sometimes necessary and is not inherently wrong but there are ways to prevent this.

Using pull process to deploy the software introduces a separation of concerns. Separating CD away from the CI is definitive difference from traditional DevOps but can provide concrete

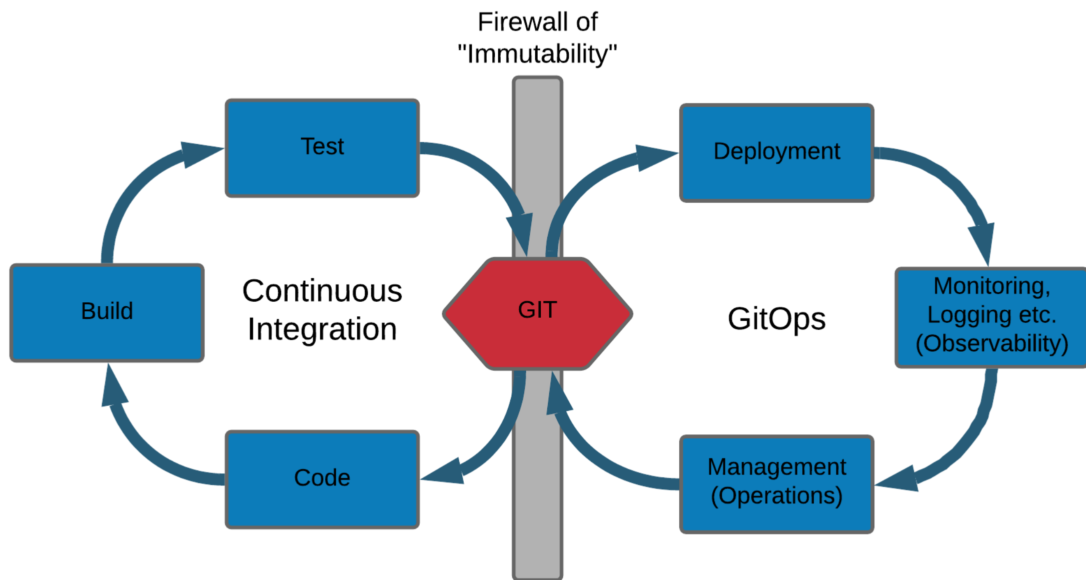

Figure 5: Separation of concerns in GitOps (WeaveWorks, 2018)

benefits (WeaveWorks, 2017a). By limiting access to the runtime environment one limits attack surface on the environment. Also by separating CI and CD there comes a definitive line between packaging and running the software - a separation of concerns. Each side can be operated separately separated by a ”Firewall of Immutability” as the two part have no dependencies on each other. The whole separation of concerns is outlined well in a post written by WeaveWorks (WeaveWorks, 2018) and presented in figure 5.

Originally bundling CD/CDe together with the CI made sense but as tooling has gotten better during the years, today there is less and less explicit reasons to do so. Doing push deployments is not inherently insecure if the CI/CD pipeline can be reinforce and access to it limited but separating runtime from software packaging limits attack vectors into the system by disabling remote control capabilities of the CI.

2.2.3 GitOps reference model and process

GitOps reference model and process were drawn to visualize the core concepts of GitOps for the interviews. The reference model and process were used to present how GitOps is organized, how it compares to DevOps and how interviewee’s current deployments align to or

differ from GitOps. Reference process enables a quick demonstration on how GitOps relates to DevOps and what parts of the process is emphasizes. Neither the model or the process is purposed to be exhaustively accurate in every situation but to give an idea of what the process and model can look in a generalized situation.

The reference process presented in figure 6. is partly derived from figure 5. that demonstrates the core concept of GitOps where separation of concerns between development and operations are done using version control like git. To compare GitOps and DevOps their processes are mapped to process timeline with the most important phases. The first note to take is how GitOps spans exactly the same parts as DevOps though there is a clear difference how the process is weighted. GitOps does not make clear distinction between development and operations as both parts are managed from a single repository and the operations part should not require extensive manual operation. The second more important note is that where in a traditional DevOps process the CI/CD pipeline holds the most likely truth of the application, in GitOps the truth has shifted left. DevOps truth is presented somewhere along the CI/CD pipelines artifacts and deployment pipeline (depending on the implementation), GitOps truth is right after code push. This shift to left leaves only minimum amount of places where deployment errors can happen.

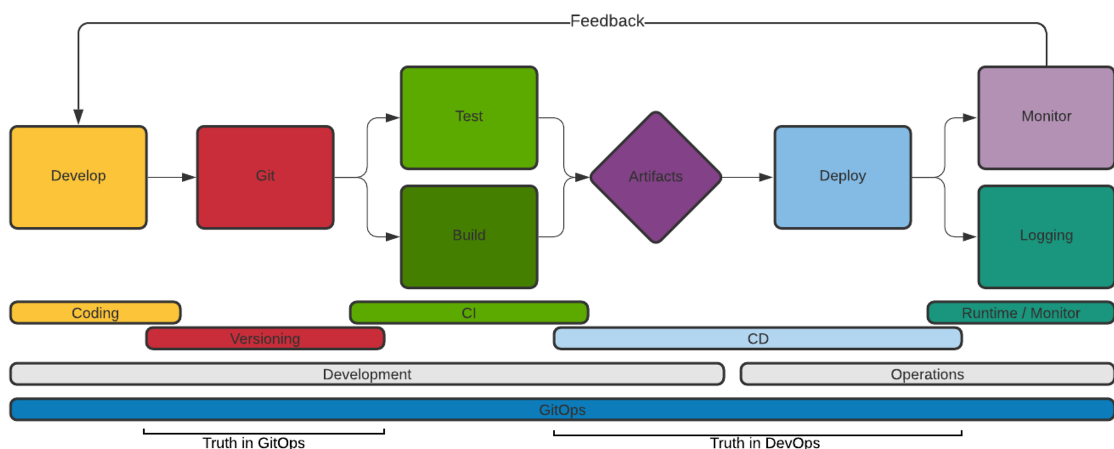

Figure 6: GitOps reference process

Figure 7. presents a reference model to indicate how different user interact with the system and how the systems works internally. The model is based on the descriptions of GitOps by WeaveWorks, 2018, Limoncelli, 2018 and WeaveWorks, 2017a. Both the model and process were reflected to aligned with concepts of DevOps presented in Hüttermann, 2012 and

Forsgren, 2019. The take from the reference model in fig 7 is that on the left side is development where changes can happen and on the right there is an area for operations where no direct modifications should be made. In this, the separation of concerns of GitOps can easily be seen. The whole runtime environment is out of reach from abrupt changes. It is technically possible to include CI inside the operations area but we decided to leave it outside to underline how using pull process in GitOps can allow CI and CD to be completely separate processes.

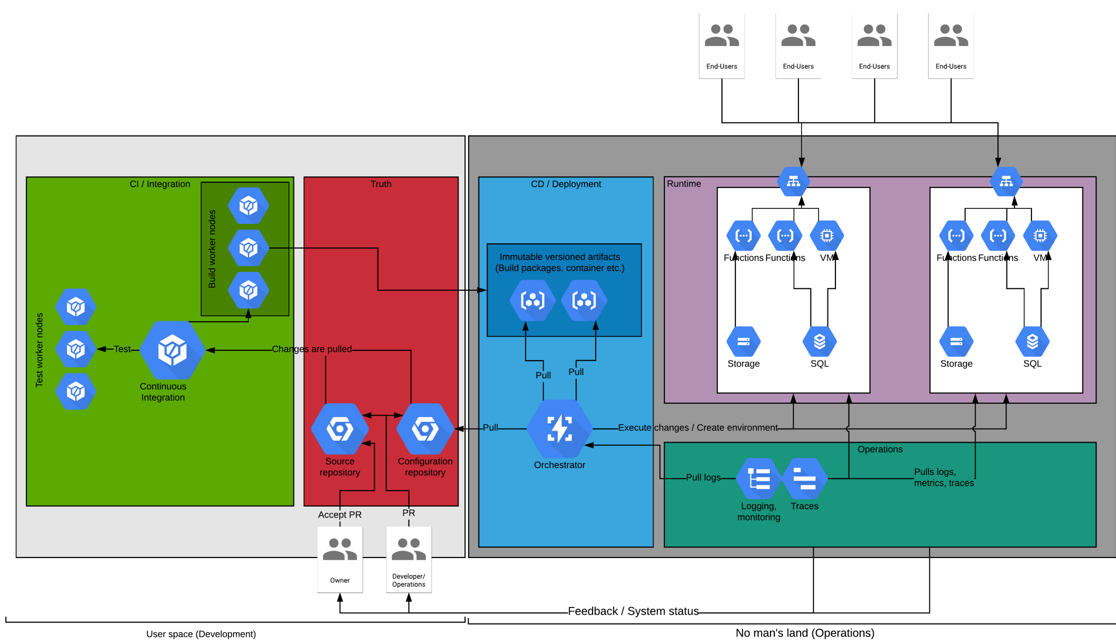

Figure 7: GitOps reference model

3 Research problem and methodology

As observed in the literature above, there exists a large gap between theorized and perceived benefits of declarative deployment. Published research about declarative deployment exist but there is not much research on how the theorized benefits have turned into practice and what kind of experiences people have about it. GitOps that relies heavily on declarative deployment has practically no formal research conducted upon it. Most likely this is because the term is only couple of years old as seen in figure 4. and industry is possibly only in the process of adopting it. Another possible reason for absent research and adoption might be that organizations are utilizing GitOps practices they have not felt a need to differ those from DevOps. In practice they would not make a difference between their current DevOps and CI/CD workflow, and GitOps. This could also be an indication of GitOps is considered as an extension to infrastructure-as-code (Limoncelli, 2018).

Albeit formal research is currently missing, there is a few good blogs which claims GitOps can give real and comprehensive benefits in speed, recoverability, security and transparency for software development and delivery (WeaveWorks, 2017b). Probably the best and most comprehensive generalization of GitOps practices in academical journals is *"GitOps: A Path to More Self-service IT"* (Limoncelli, 2018) which gives a good general introduction about what is GitOps and what GitOps benefits can be.

Rest of the section will progress as follow: we will go trough the each of the research and sub-research questions in subsection 3.1, research methodology presents case study as the basis of research in subsection 3.2 and finally research process is discussed in subsection 3.3 consisting of a literature review, interviews and analysis.

3.1 Research questions

The main goal is to research if declarative deployment is perceived beneficial and could practices like GitOps help with declarative deployments and development. Research questions are divided to one main research question and four sub-question. The aim of sub-questions is to give insight on the subject and give ground to answer the main question. Sub-questions

also divided the subject to more manageable pieces and enable the interviews questions to target more independent areas of main question.

- Main research question: *Is declarative deployment perceived beneficial?*
  - Sub-research question 1: *What reasons people have to pursue automated deployment and operations?*
  - Sub-research question 2: *What kind of tooling is used to enable declarative deployments?*
  - Sub-research question 3: *Are some aspects of declarative deployment proven to be more important than others?*
  - Sub-research question 4: *Can GitOps be seen helpful to improve current processes?*

The main research question seeks to answer if declarative deployment is perceived beneficial and what are possible experiences giving that opinion. The goals is not to proof if declarative deployment is the way to but gather possible reason why it might be.

First sub-question aims to give more general knowledge why automation of deployment and operations are pursued. What are the underlying problems that are tried to be solved? This should us help to understands the general goals what declarative deployment should solve.

Second sub-question focuses on used tooling. Processes and culture are more important in the long run as mentioned earlier but tooling is still important and can have an affect if the solving of a problem is a positive or negative experience.

Third sub-question explores if some aspect related to declarative deployments are experienced to have greater impact than others. This can suggest low-hanging fruits that can be used as a starting point or motivation to pursue more automated process. Depending on the what the experiences are GitOps might be able to solve some of these.

Fourth sub-question purpose is to explore if GitOps has been useful or if it can be seen to solve current problems. The question gives more insight on what exactly might be beneficial

in GitOps or if there is more research to be conducted in the future.

3.2 Research methodology

The research was decided to be conducted as an case study as experiences wanted to be gathered without a need of an existing theory or claim to be proved. Case study allows interview of people with open-ended question which are useful when pre-existing knowledge does is limited as is the case with GitOps.

3.2.1 Case study

An exploratory case study was chosen as the research methodology because it provides a way to research contemporary phenomena in its natural context (Runeson et al., 2008). Case study has originated from social sciences researching organizations, groups and individuals interacting with each other (Seaman, 1999). These areas of interest are all present in software engineering and so case studies have found a solid foothold in software engineering (Runeson et al., 2008).

Data collection methods (interviews, questionnaires, observations and archives)

Case studies can be performed in several different ways: interviews, questionnaires, observations and archives. Of these four the most common ones are interviews and questionnaires (Eisenhardt, 1989). Mentioned data collection methods can and usually should be combined to give both quantitative and qualitative data (Eisenhardt, 1989; Given, 2008). For example, formal questionnaires can be presented as questions with numerical answer that are easy to quantify. However, this requires more pre-existing knowledge of the subject to know how to form specific questions (Given, 2008). Interviews with more general subjects and possible open-ended questions can be used to gather qualitative data from interviews when knowledge is limited and the aim is to map subjects related to the researched field (Eisenhardt, 1989; Given, 2008).

Quantitative research as in the case of questionnaires requires generally more data behind it to draw meaningful conclusion and suggestions. It would also require research questions to be presented in a way that can be quantified which does not sit well with the focus on gathering experiences, something what cannot be predicted in advance. Qualitative research is well suited when no specific quantifiable questions can be formed, or the interception of data requires more semantic interpretation.

Because existing research about GitOps and practical experiences of declarative deployment are limited there existing no theory to be proved or examined. For this reason, we will continue with qualitative tools to gather current experiences and subjects for future research.

Interview types (Unstructured, semi-structured and structured)

Generally accepted way to categorise interviews is to divide them based on how structured they are. Three categories can be distinguished: unstructured, semi-structured and structured (Robson, 2011). Unstructured interview has a limited amount of questions, they are open-ended, and the discussion is only loosely steered and allowed to progress to areas not known in advance. Unstructured interviews emphasize open questions which do not try to impose any tone for the discussion and try to discover experienced aspects of the topic (Given, 2008).

Fully structured interviews have strictly predefined set questions without real option for follow-questions. Because the questions are specified before the interview, it is the interviewer's job is to keep the discussion within the limits of the questions (Runeson et al., 2008). Structured interviews can be very close to direct questionnaires as it gives very little movement space during the interview. Structured interviews require deep and specific knowledge from the interviewer about the problem in advance, tries to answer more specific questions and find relations between already established constructs (Runeson et al., 2008). As a downside it can undermine interviewees deeper knowledge about the subject as all the questions are predefined and interviewee might interpret the questions incorrectly (Lincoln et al., 1985). In the extremities structured interview can be a spoken questionnaire with fully quantifiable answer like yes and no (Lincoln et al., 1985).

Semi-structured interviews have been popular in case studies as it combines open-ended

and closed questions leaving possibility to find unexpected information (Seaman, 1999). We used a semi-structured interview was chosen for data collections as it gives more opportunities to discover questions and reasons which are not so clear ahead of interview. In the planned case study, there is no knowledge of what parts of declarative deployment the case projects practice or what is their perceived benefits so open-ended question are needed.

3.3 Research process

Research was conducted in three phases. First a literature review was done to establish understanding of GitOps, DevOps and declarative deployment processes which was then used to generate a reference model of GitOps to be used in the interviews and establish understanding of GitOps role in development and deployment. Second part was to collect data using interviews to find experiences related to deployments and GitOps, and the third part was to analyse the data and reporting findings.

3.3.1 Literature review

Literature review was done in December 2019. Two journals databases ACM and IEEE, Google Scholar and Springer Link were the main sources for literature review. In addition to selected sources we selected few books few books and articles to complementary source material and understand the current state of DevOps.

The review was done using two different search words: GitOps and declarative deployment. Both search terms were searched separately from each source, result needed to be published after 2010. In ACM and IEEE the search terms needed to be present in the abstract. Most relevant results based on the title and recommended relevance were used to create a shortlist. Shortlisted results were filtered based on their abstract and finally the last list was used as basis for starting point of literature. To compensate Google scholar limited metadata filtering the exact search terms were required. As seen in table 1. GitOps was hardly present in any search results. For this reason, blog post and professional journals/podcast where used to gather additional knowledge about GitOps, its promises and possible experiences.

Term	Source	Hits	Shortlist	Reviewed
GitOps	ACM	5	2	2
	IEEE	0	0	0
	Google Scholar	20	2	2
	Springer Link	1	0	0
Declarative Deployment	ACM	88	11	6
	IEEE	77	15	10
	Google Scholar	86	20	10
	Springer Link	21	4	4

Table 1: Search word hits

3.3.2 Interviews

The data was gathered from three different companies spanning fours different projects. Data collection was done using semi-structured interviews as in-person meetings which helped with the follow-up questions. Interviews were supported with a GitOps reference model and process derived from literature review to establish common understanding of GitOps and its relation to more commonly know DevOps processes between the interviewees and interviewer. This was done on the presumption that at least not all of the interviewees had previous knowledge about GitOps.

All the participating interviewees were contacted beforehand to discuss if their projects deployment process would fit the case study. The main criteria for a project to be selected for the case study was it to have at least some parts of their deployment process to be automated either imperatively or declaratively. Projects with imperative or manually deployed parts were not ruled out because discovering reasons behind manual parts was considered as valuable information.

The companies selected for the interviews were of following types: one was a small company (Tilastokeskus, 2019) with less than ten employees but serving software for few thousand users, second one had slightly bigger development team serving several thousands of employees and third one had thousands of employees with internal software development and operations teams spanning dozens of employees. More detailed breakdown of the companies can be found in table 2. Different kind of companies were selected to broaden the variety of companies to avoid bias towards one type and size of company.

Company	Industry	Project	Company employees	Development team
A	Health care	Website (P1) SaaS (P2)	~18 000	<100
B	Staff leasing	Internal (P3)	~2000	~10
C	Shift planning	SaaS (P4)	~10	~5

Table 2: Company characteristics

In total seven person were interviewed in five different interviews listed in table 3. One interview per project was preferred but due to time constraints the project 1 interviews were done individually. The data was gathered with in-person interviews during a timespan of nee week. The interviews were recorded so conversations could be reviewed later during the analysis. Results from the previous interviews were used to improve questions for the next interview but the original questionnaire topics (see 6) were not altered. Same principle was followed with the reference model. From all the projects a preference of two people were asked to attend the interview to give several views per interview but this was not kept as mandatory.

#	Company	Project	Role
I1	A	P1	DevOps Specialist
I2	A	P1	Lead Developer
I3	A	P2	Cloud Architect
I4	A	P2	Software Developer
I5	B	P3	Lead Software Developer
I6	B	P3	Senior Software Developer
I7	C	P4	Software Architect

Table 3: The roles of interviewees

3.3.3 Data analysis

As hours worth of interviews is not possible to present directly in a feasible for the reader. For this reason, the discussions and conclusions drawn from the interviews should be presented with clear connection between what was said in the interviews and what conclusion are drawn from these (Seaman, 1999). By doing this it is possible to keep external validity between data and conclusions. Analysis of the interviews were done by recording the interviews with the permission from each of the interviewees and the material was later analysed by listening the material thorough several times while making notes from the discussions with the aim of

finding common topics between interviews. These topics were then mapped to the research questions and finally conclusions were drawn from these.

3.3.4 Questionnaire

The questionnaire was a semi-structured one with open-ended question to draw unforeseen information about the topic. Exact questions were not presented beforehand in a written form and they were not shown at any point to the interviewees to not steer their thinking. All the topics were intended to be used in each of the interview but individual questions could be skipped to give more time to follow-up questions during individual interview so the questionnaire should be mostly considered as template for the interviewer (See Appendix A: Questionnaire template). All interviews were conducted in Finnish and the length of the interview was aimed to be less than one and a half hours.

4 Results

Results will be presented in three parts: tooling, topics and perception of GitOps. The topics were synthesized and grouped from the interviews when a common theme was experienced by multiple interviewees that improved software deployment and made a case about possible benefits of declarative deployment.

4.1 Tooling

As mentioned in the research methodology (chapter 3.2), the selected projects were verified to use automated deployment pipelines before the interviews were conducted. Between the projects there was variation on the extent of how far the automation was implemented and what kind of tools were used. Some projects utilized highly automated deployments pipelines with only few manual triggers to give more control over e.g. testing environments to prevent collision between tests executed manually.

Deployment is not tooling dependent but tooling can still guide or limit how the deployment process will happen.

> *"When selecting tools for the project, AWS Lambda just had a breakthrough and we wanted to focus on coding. This was likely one of the main reasons why we selected serverless"*. - P3

They later elaborated that they did not choose CloudFormation for deployment because it was declarative but because it was the preferred way to configure AWS Lambdas. CloudFormation is a verbose tool because it is declarative but the selection happened for another reasons.

The projects utilized three different declarative tools listed in table 4. Terraform[8] is a proven declarative tool to deploy infrastructure-as-code. It has binding to all the major cloud vendors but as it is not run within the runtime environment there are better tools to detect drift changes. AWS CloudFormation[9] is similar to Terraform but is AWS specific and can

[8] https://www.terraform.io/

[9] https://aws.amazon.com/cloudformation/

manage the runtime environment better as it has good capabilities to recreate AWS specific components like EBS and lambda functions. Kubernetes[10] is currently the most well-know container orchestration platform. Kubernetes is designed to be provider neutral in a similar way as Terraform is but Kubernetes is more focused on the execution side of deployment than pure infrastructure. Kubernetes uses a construct of containers that wrap the application and related libraries to a single runnable image. Kubernetes has an extension system called Custom Resource Definition (CRD) to extend its capabilities to custom environments which can allow provisioning supporting infrastructure like load balancers, firewalls etc.

Projects P2 and P4 utilized Terraform to manage their infrastructure in AWS, project P2 and P3 had AWS CloudFormation to deploy the application and project P1 utilizes Kubernetes to manage everything in the environment except cluster creation. Project P2 used Kubernetes to run the containers in collaboration with CloudFormation and Terraform which were used for the infrastructure creation and management.

Project	Terraform	Kubernetes	AWS CloudFormation
P1	-	X	-
P2	X	X	X
P3	-	-	X
P4	X	-	-

Table 4: Tooling used for declarative deployment

As shown in table 5. more differences are found when asked what kind of platform the software was running on, was there a CI/CD pipeline, automated tests and how the deployment was triggered. Project P3 had the most comprehensive deployment pipeline including test automation suite with good test coverage, CI/CD pipeline and CDe enabled from master branch. Every project used GitLab as their continuous integration pipeline and Gitlab runners were used to build and test the software.

Projects P1 and P2 used containers to deploy the application which required their pipelines to build and store container images. Project P1 used separate container building and storage services which listened for commits in their repository. Applications that were wrapped to containers were built outside the execution environment. P1 had future plans to migrate more of the build and testing processes from dedicated CI tool to leverage the capabilities of the

[10]https://kubernetes.io/

existing Kubernetes cluster. Project P2 used self-managed GitLab to build, test, store and push the containers into the execution environments.

Project	Container	Serverless	Virtual Machine	Pull	CI Push	Manual Push
P1	X	-	-	X	-	-
P2	X	-	-	-	X	-
P3	-	X	-	-	X	-
P4	-	-	X	-	-	X

Table 5: Deployment platform

4.2 How declarative deployment has improved development, deployment and operations

Below sections break down how the identified topics have been perceived beneficial and foster declarative deployment. Identified topics were *documentation and visibility*, *immutability and auditioning*, *resource allocation*, *environment creation and management*. Environment creation and management was split into three sub-topics: *development environment*, *zero-downtime deployment* and *target state and deployment repeatability*. Some of the topics do overlap with each other on some parts but presenting them as individual results gives more granular understanding of the benefits. For example *immutability* and *deployment repeatability* have overlapping components but immutability promotes others aspects of deployment including but not limited to repeatability.

4.2.1 Documentation and Visibility

Each project mentioned documentation as a great benefit of declarative deployments. In this context the documentation is not referred as user manuals, requirements or expectations related to what the system should do, but documentation of what the system actually is. This became more apparent when the whole pipeline was declarative but imperative tools were also considered to provide useful documentation when used with templates. Importance of being able to derive system state from the deployment files was compared to be as important as database migration files.

> *"To me it is almost as important to know what our system should be as to have our database migration scripts"* - I7

One interviewee mentioned that if documentation is not strictly tied into the codebase keeping changes and documentation in-sync might get really hard.

> *"The version control is the one thing that can stay up-to-date. Managements software like Jira and Confluence are hard to keep current and, if wanted, the release notes can be generated from the definitions. We have reinstalled whole systems to make sure the documentation matches the system"*. - P2

Last part of the previous statement refers to deployments made by reading documentation and executing the steps in order. A manual that describes how to achieve the desired state of the system *at one point in time*. Documentation does not hold any guarantees of being up-to-date with the currently deployed system configuration. If a change was made to the system which was never updated into the documentation, it would generate extra work just to identify which end had the misinformation. Is the documentation wrong or is the system state wrong?

Describing the environment declaratively had improved development process by improving sharing of knowledge about the system. At least two projects explicitly mentioned that by using documents to describe the deployment and pipeline they minimize the responsibility of a single developer.

> *"I think it is not viable (for our business) that the environment would depend on one persons knowledge or even that the description would be in a natural language"*. - I7

Another developer from a different project said it is much easier to pinpoint bug reports to recent changes.

> *"It is easy to look that a breaking change was introduced at the same time as an API change was made in the infrastructure declaration."* - P3

4.2.2 Immutability and Auditing

Immutability was closely associated with how the software is delivered to execution environment in contrast to how the environment itself is established. Projects P1 and P2 promoted usage of immutable deployments artifacts in the form of containers. This gave them the benefit of artifacts that can be build and tested once, and used after in different environments without a possibility of intermediate changes or manipulation. This was a difference to P3 and P4 of which both agreed that their deployment process do technically allow changes to software packages between environment builds. This was considered rare (due to package locks) compared to times the software has been deployed but it has happened.

> *"Our client build has been broken because NPM provided us with different version of React than what we run in development."* - P4

Immutability's contribution to deployment times and feedback loop were observed from multiple projects. Immutability enabled easier artifact tracking and decreased the total time required for deployments as source code was not build every time it was deployed into a new environment.

> *"Deployment has become at least twice as fast maybe even little more and we got better tracing as a bonus."* - I1
>
> *"It is much easier to keep one container that will be promoted, tagged, tested and moved forward. It is also much faster to move from environment to another"* - I2

This seems to be a change since their previous deployment strategies before using containers. Containers concept of "build once, run anywhere" has cut down need to build the software between environments as was pointed in the case of projects P1 and P2.

Couple of the interviewees mentioned that historically there has been problems with verifying which version of the software they were currently running. To be able to check and know what version is running, was considered to be very important as it is the basis for everything else.

> *"I think that to know we are able to verify what exact version we run is the most important though (automated) tests are right next to it"*. - P1

Similar thing was said by another developer who had seen problems in the past. In this case they were forced to use an anti-pattern where Enterprise JavaBeans (EJB) had to implement a version API so they could later verify the version of the deployed EJBs.

Immutability of the software contributed to auditioning but also declarative deployment itself was considered to make auditing easier. One of the projects was already been audited and their documents describing the environment had clearly helped them.

> *"We have been audited with good results. Overall because we have clear repositories, we are quite easy to audit."* - P3

P2 felt that declarative deployment and container will help them in the future when time for possible external audit comes: *"We do not want that others need to trust only our own words, we want more concrete proof"*. P1 and P3 agreed that in the case of need for auditing it would be quite easy but did not see it required in the foreseeable future.

4.2.3 Resource allocation

Resource usage was seen in two ways: direct monetary costs like infrastructure and services, and human resources. All the projects were hosted in a third party cloud environment which introduced direct cost. Depending on the deployment model, the fineness of billing units varied from single instances to purely runtime based costs.

P3 said to achieve really good cost scaling with AWS lambdas when running the software costed more during the day when there were many more people were using it and much less during the nights. Project P2 saw usage of managed services like CloudFormation as a time saving measure to let developers focus on producing software.

> *"We do not need to worry much about Linux virtual machines and their administration. It is not really what we are supposed to do."* - I1

In this case managed service was configured using Terraform declarations and resources were

seen as time needed to maintain environments which would be away from development. P4 mentioned to use Terraform to define auto scaling groups to deprovision their non-essential infrastructure for the night. This has achieved them concrete cost savings in infrastructure hosting without constant manual effort or any effect on their development or service levels.

4.2.4 Environment creation and management

Each of the projects brought up how the use of declarative tools has enabled them to create either identical or at least functionally identical environments of the production. The perspective varied slightly depending on the role of the interviewee but the views did not contradict with each other. Some focused more on the ability to create identical shared environments running different version of the code for testing purposes. Others were more focused on deployment practices and upkeeping the environments state. For this reason, this topic is split into three areas to give each aspect a voice.

Development environment

Many developers brought up on several occasions that with declarative tool they are able to replicate the whole production environment on their own personal development machines which might have been possible in the past but was very hard to implement and maintain. The ability to replicate production on a development machine has been experienced to make their initial code pushes more robust as the code is better tested before committing to repository.

> *"With docker-compose developers can develop and test the code on such an identical environment to production, that there is rarely a need for to use separate test environments".* - I5

Interviewee I4 emphasized that it is much more comfortable to use local environments for development. This way a single breaking change does not prevent the work of other developers, which would be the case with shared development server. This offloaded unnecessary stress experienced earlier when using shared environments with other developers.

After adopting declarative deployment, on-boarding of new employees has become less

complicated as the software required for development, building and configuration is handled by the tooling.

> *"I basically gave the new employee repository address and couple of command and he could start developing."* - P1

P3 gave an example on how automated deployments allowed their new developers to commit code even into production without a need to know the details of the deployment, enabling them to be more productive sooner.

Zero-downtime deployment

Two projects explicitly mentioned benefits of zero-downtime deployments that are now possible after introducing declarative deployments. Especially P3 had experienced this very satisfying as they do not need to schedule maintenance windows anymore.

> *"We do not need to have maintenance windows anymore or to communicate maintenance schedules to customers."* - P3

If the deployment would not be completely automated and robust they might even lose this benefit as manual deployment could introduce downtime and in turn prevent integration of code to production.

> *"With downtime we would not be able to release whenever wanted."* - P3

For them the execution environment is recreated on every deployment, defined by the declarative build configuration. When the brand new environment is ready, the traffic will be automatically forwarded to the new environment. They also mentioned that they do not experience problems with non-matching versions within the application as the environment is fully recreated on each deployment.

P1 said that they prefer to do the deployments in the evenings but when needed they have had no problems doing zero-downtime deployments during the day.

> *"Evenings are preferred mostly because there is a temporary effect on*

> *the performance and because of high volume of users we do not want to take an unnecessary risk."* - P1

The fact that deployments are now faster and more robust allowed them to deploy during lower traffic hours. This was experienced especially convenient as earlier they needed to wait until the most quiet hours of the night.

4.2.5 Target state and deployment repeatability

One perceived benefit of declarative deployment was its ability to manage the state which also guaranteed deployments to be highly repeatable once configured. This was experienced to release resources away from operations, though this was not always considered to mean no need for operational teams at all. For example projects P2 and P3 had varying experiences. P3 said that if they would not have an automated and declarative deployment, they would possibly need a extra person to do and manage the deployments. P2 noted that even strictly operations focused teams might not be needed in the same way as in the past, there is still demand for people who understand the underlying technology in a great detail as external upkeep does not help designing complex environments.

Depending on the exact tools and environments used, the deployments were able to self recover from error and/or scale accordingly to a certain degree. Projects P1, P2 and P3 had constant self recovery with Kubernetes and AWS serverless that could keep the original state. Others using tools like Terraform[11] did not have automatic drift correction after initial deployment. Having self healing environments kept the system more robust and constant manual monitoring is not needed.

> *"I sleep my night more calmly when I know the production matches the deployment configuration."* - I7

When the environment can manage possible crashes, scheduled cloud provider maintenance, scaling and self manage the deployments, it has released the development team to focus on development and not maintaining the environment.

[11] Terraform is able to do limited drift correction with extra tooling

> *"These days, there is only rarely a need to inspect or debug the deployment, all our resources are in development."*. - P3

Closely related to self recovering nature of declarative tools and why these tools might become more and more important in the future was received from I2. He mentioned that the way how software and infrastructure was run and managed today is completely different from the past.

> *"The benefit of using containers compared to earlier is that current software is more stateless. In the old world one made clusters that could not break down, today one buys so much cheap hardware that breaks do not matter. It is completely different philosophy of doing things."*. - I2

This remark sits quite well with observations on how new environments can be created on will. When discussing how environments were managed currently, the I7 said that recreating environments is now simple. They had actually just days ago recreated one environment with only hours worth of effort. The ability to easily recreate environments as declarations was considered borderline mandatory. Following answer on the need of declarations underlines the need.

> *"The question is absurd, could it be in any other way?"* - I7

Further discussion revealed that even if on-demand recreation is not always needed, it is experienced as an insurance for the business keeps developers mind at ease.

When changing of the system state was considered easy, it promoted change and gave more trust on the whole system. Making an error was no more considered a problem as all kinds of mistakes were easy to revert by oneself.

> *"If you make a mistake with a VM and you e.g. accidentally delete a file governing access to the host you really need to call someone with more technical knowledge. Now I do not have to."* - P1

Same kind of comment was made from I7 when he said that even after a disaster there is always something to come back to. Even if they would lose whole production it could be

fully deployed in a matter of hours than days.

Sometime non-declarative deployment was pondered to be close to technical dept. When there is no clear location where to put all the configurations, managing these would get harder and harder in the future. Maybe even to a point that it would become infeasible.

> *"The amount of things to remember could pile up to a point where we would not even know how to pay it back. Even if we would have time to do it."* - I7

4.3 Perception of GitOps

Most of the interviewees had heard of GitOps as a terms but only couple said to know the more precise meaning behind it. GitOps was in every case explained and demonstrated with the figures 6. and 7 to establish a common understanding. At this point each of the interviewees were asked, if they had any thoughts or questions related to the concept. Each of the projects could see clear similarities with their project to GitOps reference model and could map their deployment process to the GitOps reference model and process pictures.

Each interviewee saw that aiming to GitOps is something worth doing. The degree how fundamentally all aspects should be executed varied but core concepts were generally agreed to be solid. One answer when directly asked would they think GitOps to be beneficial.

> *"Definitely, we are apparently already doing it"*. - P3

Another one agreed with GitOps as he had always experienced this to be the goal in his mind but did not have a name for it. Generally the shift left mentality of GitOps moving the "truth" of the system into VCS was generally agreed to be a good thing. Every interviewee saw putting everything to version control as a fundamentally good thing and none raised any real concerns why this could or should not be done. However the degree of how important others aspects of GitOps remained to be debated. Things like separation of concerns, completely blocking access to execution environment and positioning of CI in the model were not always seen mandatory or even ideal.

Separation of concerns was not seen as important in practice as in the literature. Many felt that the execution environment should enable changes and access from outside the environment due to practical reasons. Testing new configurations and debugging the environment were considered important, even if the need for them would only be occasional. P3 said that they have accesses the production environment to debug problems but this was rarely done and normally external logging and monitoring was enough. A comment made by P2 was that sometimes it is just much easier to test a change by hand and only when confirmed working, commit it to version control. In this case the change would eventually be overwritten if not committed to version control by then. Overall, practicality and principle were seen something requiring balancing in between.

When talking about the push and pull models of GitOps there were clear differences in opinions. More often than not the deployment was done using CI pipeline or similar push methods and then the environment would be build from a declarative description. When asked if this was considered to be conflicting with GitOps the answer was that the CI could be considered (even being logically in another environment) to be part of the CD side of GitOps as normally no-one logs in into the CI/CD machine nor production environment. Only P1 was using GitOps recommended pull method where the execution environment pulls the changes from repository and the separation of concerns is fully realized.

One remark of GitOps was how operating from the VCS was good and something worth actively pursue to but fundamentally it is still a human process that requires some curating. Curating is needed to prevent pollution of the VCS to a point where no-one know what the change is actually for. A single change is easy to be tracked to a point in time but after months or years it gets harder and harder to understand what was the reason for the change. Another remark made about deployment models was that people do not select declarations over imperative because either ideology is better but the tools that can realize their vision to automate processes.

> *"We do not speak with these terms. We do not consider it to be declarative or imperative runtime but this is how it is done and how it works."* - I2

In practice the benefits of declarative deployments were seen and tools were partly selected

for these reasons but there was no definitive distinctions made for what made it possible.

The major hindrance on moving to full GitOps seemed to be knowledge about available declarative and supporting tooling and how to use them. People seemed to be often more familiar with VMs and moving all operations to an ephemeral environment which can be expected to be deleted and spawn somewhere else is experienced slightly disruptive. This was mainly focused on monitoring and logging as every project mentioned these two fields to be something to improve in the future, or they were maybe one of the last step in the way of fully realized GitOps. One interviewee described his thoughts about GitOps.

> *"There really is no problem here. This looks pretty good as long as someone solves monitoring in our project."* - P1

5 Implications and Discussion

In this section we will go trough each research question. First the four sub-questions and based on them we will answer the main research question. Later several limitations of the work are discussed.

5.1 Research questions

What reasons people have to pursue automated deployment and operations?

The most obvious aim of automation is resource utilization which was also the case here. The resources can be purely monetary as direct hardware cost saving but also human labour. Two main findings on why to use highly automated imperative or purely declarative tools were: focus all resources into development and ”buy” operations from outside while maintaining ability to deploy and manage the environment within own organization. Shifting operations to development was summarized by I7: *“DevOps is solving operational challenges with development tools”*. This was exactly the case with two companies as they were clearly trying to avoid committing resources to operations and focus strictly on development. Third company had a need for more dedicated personnel for operations partly due to its size but also the for the knowledge required to design and develop more complex environments. The commitment to design and develop the operations aimed to minimize effort required for pure maintenance in the future.

What kind of tooling is used to enable declarative deployments?

The tooling used for declarative deployment varied quite a lot. Just within three companies there were almost as many runtimes and deployments tools used as there were studied projects, see tables 4. and 5. This suggest that there is no single definitive tooling or way to do declarative deployments and usually declarative tools are combined with imperative ones. Due to sample size no conclusions could be made but also literature suggests that tooling is still developing on several different branches. Only the coming years will tell if there will be clear winners and consolidation towards more homogeneous tooling.

Are some aspects of declarative deployment proven to be more important than others?

According to interviews there are many experienced benefits which are mostly only available when using declarative tools. Declarative tools compared to any other tools have the benefit that they can understand the desired state of the systems. By understanding the state, the system is able to recover and transform itself to a new state when asked to. Looking solely the interviews it is hard to say if a single aspect of declarative deployment is more beneficial than another one, but at least two were mentioned many times by all of the projects: ability to replicate production to development machines and to have up-to-date documentation that the declarative tools provide via deployment configuration. These seemed also to be the first things to manifest when declarative deployment was pursued.

Developers have greatly benefited from the ability of moving away from the shared environments and keeping the same state between all local development environments. This has given them more freedom by allowing them to temporarily break the code and test it before committing anything to a shared version control. The benefits have been: improved code quality, lower stress and saved time. All of these improvements were observed in the interviews and literature (Forsgren, 2019; Limoncelli, 2018; Hüttermann, 2012).

Can GitOps be seen helpful to improve current processes?

According to literature and experiences from the interviews, GitOps should be thought as a superset or a flavor of DevOps. GitOps does not present ground breaking changes to development processes after DevOps but it gives a novel aspect on how to do operations. DevOps or GitOps is not dependent on specific tooling but both relies heavily on it to automate code integration and pushing to environments through CI/CD pipelines. The difference comes as DevOps does not exactly promote automated operations or utilize the power of version control to describe the whole system holistically.

GitOps was all in all viewed in a favourable manner. A noteworthy result was that most people already seemed to be aiming for GitOps like practices but did not have a name to describe it or did not feel a necessity to distinguish it from DevOps. Most interviewees said that they are already practicing or are approaching the operational model of GitOps. Differences between pull and push operations were not seen that interesting. This could be due to the

fact that the CI pipeline and people having access to environments were considered worth the trust. There was also suggestions that the environment access can be controlled with policies rather than strictly with technology. This gave development team more lenience to debug problems if needed.

The idea of keeping the version control as the source of truth was considered the most important aspect of GitOps by far. Implications of utilising VCS spanned from documentation and auditing to generating local development environments. For developers the ability to create identical environments matching production, leveraging declarative deployment with version control was experienced extremely useful in many ways. It allowed developers to have more trust on the code, if it run well locally then it would also run well in production. They could also easily test the code with same package version and avoid shared environments, where one breaking change could limit and affect others ability to proceed with their work. This was probably also the low-hanging fruit of declarative deployments as it had such a huge impact on developer experience.

It is worth mentioning that even if the production would not be deployed declaratively the development environment can still be created with declarative tools to match the production. Though in this case one of the big benefits to match development and production states in an exact manner is lost. As the literature suggested the overall experience of GitOps was that it will release development resources from daily operation. Conducted interviews suggested similar observations. This does not mean that professionals doing operations would not be needed but their focus moved from daily maintenance routines to design and providing insight to underlying technologies.

Is declarative deployment perceived beneficial?

Interviews gave a clear sign that declarative deployment is perceived beneficial. Declarative deployment has its initial cost as it often requires a paradigm shift and learning a bunch of new tools but overall no major unsolved drawbacks were discovered in the interviews. This observation is backed by the current literature albeit best practices and tools are still evolving (Forsgren, 2019; WeaveWorks, 2017a). Naturally there is limitations on how far declarative deployment can be pushed as pointed by WeaveWorks (Fisher, Bret, 2019). The

extensive requirement of relatively new tooling is experienced as a hindrance when adopting declarative workflows but is most likely to be solved when tooling and ecosystem matures. The ephemeral nature of containers and FaaS have at times been experienced as a problem, especially with things like logs as their value increases after anomalies and crashes. With ephemeral containers and functions there is a risk of losing these without specialized tools. This is a stark contrast to traditional VMs that have been quite easy to access in any point of time without any specialized tools. Despite above, practices like GitOps that leans heavily on declarative deployment are experienced as something worth pursuing and are felt to improve deployment process to be more transparent and require less maintenance.

5.2 Limitations

Limitations should be reflected mostly through reliability (Runeson et al., 2012; Flyvbjerk, 2006; Byrne, 2017) and external validity (Runeson et al., 2012). Threat for the external validity is sample size, for this reason the conclusions should not be to applied to all situations or taken as generalized facts. This does not mean that the results and conclusions could not imply real possibilities of the studied benefits.

Reliability is an area where results should be treated carefully. Due to resource limitations the material was not examined with dedicated tools focusing on qualitative analysis. The data was interpreted by only one person and because of that researcher bias is not something that can be ignored (Byrne, 2017). Researcher bias was tried to be avoided by using quotations from interviewees to minimize own interpretation. Usage of a dedicated tool and interpreting the data by several persons would have yielded more reliable and maybe different results.

Internal validity (Runeson et al., 2012) could be discussed but its weight is diminishing compared to reliability and external validity because the aim of the thesis was not to validate any predefined theory or establish a new one. The aim was to collect experiences and gather material to see if further study of GitOps and declarative deployment would be interesting.

For above reasons even if the results can be presented without bias the conclusions made from them should not be generalized as facts but as possible outcomes if practices like declarative deployment and GitOps are implemented.

6 Conclusion

The research had two main focuses, it examined experienced benefits of declarative software deployment and possible benefits of GitOps to current and future practices. The research was conducted in three phases: literature review, interviews, and analysis of interviews. Literature review was conducted to establish an understanding about declarative deployment and GitOps. Literature review was also used to generate a reference model on how GitOps operates and how it is related to more well-known DevOps. This model was used in the interviews to study experiences related to declarative deployment and possible benefits GitOps might yield. Interviews were studied with four separate projects to limit bias towards one specific workflow and give more ground for varying experiences. After interviews were conducted the material was analysed by identifying common topics and tools that have been experienced positive for declarative deployment. Finally, thoughts about GitOps were analysed and summarized.

The main findings of the interviews supported by the literature were: declarative deployment enables shared and personal environments identical to production, less required resources for everyday operations, more up-to-date documentation, more robust and automated deployments, and immutable artifacts that are easy to version and audit. Aforementioned findings were perceived to enable development teams to operate in more robust and intelligible development and operation environments. GitOps was seen as worth pursuing to as its core concept of keeping everything in version control allowed development and operations to be tracked and managed from a single source giving visibility to the whole project.

Thesis concludes that declarative deployment is perceived beneficial in comparison to more imperative practices and GitOps is seen to support declaration focused workflows by leveraging fundamental abilities of version control and declarative tooling.

Bibliography

Ari, N., and N. Mamatnazarova. 2014. “Programming languages”. In *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*, 1–8.

Artac, M., et al. 2017. “DevOps: Introducing Infrastructure-as-Code”. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 497–498.

Attardi, G., et al. 2018. “Declarative Modeling for Deploying a Container Platform”. In *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 386–389. doi:10.1109/WAINA.2018.00116.

Benlian, Alexander, Thomas Hess, and Peter Buxmann. 2009. “Drivers of SaaS-Adoption – An Empirical Study of Different Application Types”. *Business & Information Systems Engineering* 1, no. 5 (): 357. ISSN: 1867-0202. doi:10.1007/s12599-009-0068-x. https://doi.org/10.1007/s12599-009-0068-x.

Bjork, A. 2017. “Agile at Microsoft”. https://www.youtube.com/watch?v=-LvCJpnNljU&feature=youtu.be. (accessed: 18.11.2019).

Boyer, F., et al. 2018. “Poster: A Declarative Approach for Updating Distributed Microservices”. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 392–393.

Breitenbücher, Uwe, et al. 2017. “Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?” *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*: 18–27.

Byrne, B. 2017. *What is researcher bias?* Sage Publications.

Chen, L. 2015. “Continuous Delivery: Huge Benefits, but Challenges Too”. *IEEE Software* 32, no. 2 (): 50–54. doi:10.1109/MS.2015.27.

Cusumano, Michael A. 2008. “Managing Software Development in Globally Distributed Teams”. *Commun. ACM* (New York, NY, USA) 51, no. 2 (): 15–17. ISSN: 0001-0782. doi:10.1145/1314215.1314218. http://doi.acm.org/10.1145/1314215.1314218.

Eisenhardt, Kathleen M. 1989. “Building theories from case study research”. *Academy of management review* 14 (4): 532–550.

Fernando Capretz, L. 2014. "Bringing the Human Factor to Software Engineering". *IEEE Software* 31, no. 2 (): 104–104. doi:10.1109/MS.2014.30.

Fisher, Bret, F. Brice. 2019. "Chat with WeaveWorks about GitOps". https://podcast.bretfisher.com. (accessed: 30.12.2019).

Flyvbjerk, Bent. 2006. *"Five misunderstandings about case-study research" Qualitative inquiry*. Sage Publications.

Forsgren, N. 2019. "2019 Accelerate State of DevOps". https://cloud.google.com/blog/products/devops-sre/the-2019-accelerate-state-of-devops-elite-performance-productivity-and-scaling.

GitLab. "GitLab Continuous Integration (CI) & Continuous Delivery (CD)". https://about.gitlab.com/product/continuous-integration/. (accessed: 17.11.2019).

Given, Lisa. 2008. "The SAGE Encyclopedia of Qualitative Research Methods". doi:10.4135/9781412963909. https://methods.sagepub.com/reference/sage-encyc-qualitative-research-methods. (accessed: 16.11.2019).

Glauser, R. 2020. "What is xOps?" https://www.saltstack.com/blog/what-is-xops/. (accessed: 8.5.2020).

Hüttermann, Michael. 2012. "Beginning DevOps for Developers". In *DevOps for Developers*, 3–13. Berkeley, CA: Apress. ISBN: 978-1-4302-4570-4. doi:10.1007/978-1-4302-4570-4_1. https://doi.org/10.1007/978-1-4302-4570-4_1.

Jabbari, Ramtin, et al. 2016. "What is DevOps?: A Systematic Mapping Study on Definitions and Practices". In *Proceedings of the Scientific Workshop Proceedings of XP2016*, 12:1–12:11. XP '16 Workshops. Edinburgh, Scotland, UK: ACM. ISBN: 978-1-4503-4134-9. doi:10.1145/2962695.2962707. http://doi.acm.org/10.1145/2962695.2962707.

Jalote, Pankaj. 2005. "Software Processes". In *An Integrated Approach to Software Engineering*, 25–77. Boston, MA: Springer US. ISBN: 978-0-387-28132-2. doi:10.1007/0-387-28132-0_2. https://doi.org/10.1007/0-387-28132-0_2.

Jaroslav, Tulach. 2008. "Declarative Programming". In *Practical API Design: Confessions of a Java Framework Architect*, 225–235. Berkeley, CA: Apress. ISBN: 978-1-4302-0974-4. doi:10.1007/978-1-4302-0974-4_12. https://doi.org/10.1007/978-1-4302-0974-4_12.

Jenkins. "Installing Jenkins". https://jenkins.io/doc/book/installing/. (accessed: 17.11.2019).

Kneuper, Ralf. 2018. "Software Processes in the Software Product Life Cycle". In *Software Processes and Life Cycle Models: An Introduction to Modelling, Using and Managing Agile, Plan-Driven and Hybrid Processes*, 69–157. Cham: Springer International Publishing. ISBN: 978-3-319-98845-0. doi:10.1007/978-3-319-98845-0_3. https://doi.org/10.1007/978-3-319-98845-0_3.

Kromhout, Bridget. 2018. "Containers Will Not Fix Your Broken Culture (and Other Hard Truths)". *Commun. ACM* (New York, NY, USA) 61, no. 4 (): 40–43. ISSN: 0001-0782. doi:10.1145/3162086. http://doi.acm.org/10.1145/3162086.

Leppänen, M., et al. 2015. "The highways and country roads to continuous deployment". *IEEE Software* 32 (2): 64–72.

Leslie, H. 2020. "DataOps and Beyond: How DevOps Methodology Transformed Our Approach to Data Science". https://devops.com/dataops-and-beyond-how-devops-methodology-transformed-our-approach-to-data-science/. (accessed: 8.5.2020).

Lesser, Eric, and Linda Ban. 2016. "How leading companies practice software development and delivery to achieve a competitive edge". *Strategy & Leadership* 44 (1): 41–47.

Limoncelli, Thomas A. 2018. "GitOps: A Path to More Self-service IT". *Commun. ACM* (New York, NY, USA) 61, no. 9 (): 38–42. ISSN: 0001-0782. doi:10.1145/3233241. http://doi.acm.org/10.1145/3233241.

Lincoln, Yvonna S, and Egon G Guba. 1985. "Naturalistic inquiry."

Martin, R. et al. 2019. "Manifesto for Agile Software Development". http://agilemanifesto.org/. (accessed: 12.11.2019).

Pahl, C., et al. 2019. “Cloud Container Technologies: A State-of-the-Art Review”. *IEEE Transactions on Cloud Computing* 7, no. 3 (): 677–692. doi:10.1109/TCC.2017.2702586.

Pittet, S. “Continuous integration vs. continuous delivery vs. continuous deployment”. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. (accessed: 2.5.2020).

Poppendieck, M., and M. A. Cusumano. 2012. “Lean Software Development: A Tutorial”. *IEEE Software* 29, no. 5 (): 26–32. ISSN: 1937-4194. doi:10.1109/MS.2012.107.

Ravichandran, Aruna, Kieran Taylor, and Peter Waterhouse. 2016. “DevOps in the Ascendency”. In *DevOps for Digital Leaders: Reignite Business with a Modern DevOps-Enabled Software Factory*, 3–14. Berkeley, CA: Apress. ISBN: 978-1-4842-1842-6. doi:10.1007/978-1-4842-1842-6_1. https://doi.org/10.1007/978-1-4842-1842-6_1.

Robson, Colin. 2011. *Real world research.* Vol. 3. Wiley Chichester.

Rossberg, Joachim. 2019. “Introduction to Application Life Cycle Management”. In *Agile Project Management with Azure DevOps: Concepts, Templates, and Metrics*, 1–36. Berkeley, CA: Apress. ISBN: 978-1-4842-4483-8. doi:10.1007/978-1-4842-4483-8_1. https://doi.org/10.1007/978-1-4842-4483-8_1.

Runeson, P., M. Rainer, and B. Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples*. Vol. 1. Wiley Publishing.

Runeson, Per, and Martin Höst. 2008. “Guidelines for conducting and reporting case study research in software engineering”. *Empirical Software Engineering* 14, no. 2 (): 131. ISSN: 1573-7616. doi:10.1007/s10664-008-9102-8. https://doi.org/10.1007/s10664-008-9102-8.

S. Aravena, J. Khan. 2018. “Kubernetes and The GitOps Face-Off”. https://www.youtube.com/watch?v=9qGqx_jdxQg. (accessed: 21.11.2019).

Sayers, D. 2017. “The Perfect Combination: Imperative Orchestration, Declarative Automation”. https://devops.com/perfect-combination-imperative-orchestration-declarative-automation/. (accessed: 18.11.2019).

Seaman, C. B. 1999. “Qualitative methods in empirical studies of software engineering”. *IEEE Transactions on Software Engineering* 25, no. 4 (): 557–572. doi:10.1109/32.799955.

Shahin, Mojtaba, Muhammad Ali Babar, and Liming Zhu. 2017. “Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices”. *IEEE Access* 5:3909–3943.

Sharma, S., and N. Hasteer. 2016. “A comprehensive study on state of Scrum development”. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 867–872. doi:10.1109/CCAA.2016.7813837.

Society, IEEE Computer, Pierre Bourque, and Richard E. Fairley. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press. ISBN: 9780769551661.

Stober, Thomas, and Uwe Hansmann. 2010. “Traditional Software Development”. In *Agile Software Development: Best Practices for Large Software Development Projects*, 15–33. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-70832-2. doi:10.1007/978-3-540-70832-2_2. https://doi.org/10.1007/978-3-540-70832-2_2.

Sutherland, J., et al. 2007. “Distributed Scrum: Agile Project Management with Outsourced Development Teams”. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS’07)*, 274a–274a. doi:10.1109/HICSS.2007.180.

Tilastokeskus. 2019. “Pienet ja keskisuuret yritykset”. https://www.stat.fi/meta/kas/pienet_ja_keski.html.

Ur Rahman, Akond Ashfaque, and Laurie Williams. 2016. “Security Practices in DevOps”. In *Proceedings of the Symposium and Bootcamp on the Science of Security*, 109–111. HotSos ’16. Pittsburgh, Pennsylvania: Association for Computing Machinery. doi:10.1145/2898375.2898383. https://doi.org/10.1145/2898375.2898383.

Vargo, Google Cloud, Seth : Developer Advocate. 2018. “Everything as Code: The future of ops tools”. Google Developer Advocates. https://www.hashicorp.com/resources/everything-as-code-the-future-of-ops-tools. (accessed: 17.11.2019).

WeaveWorks. 2017a. “Guide To GitOps”. https://www.weave.works/technologies/gitops/. (accessed: 17.11.2019).

—— . 2018. “What Is GitOps Really?” https://www.weave.works/blog/what-is-gitops-really. (accessed: 21.11.2019).

—— . 2019. “Automating the Operation of Stateful Apps in Kubernetes with GitOps”. https://www.weave.works/blog/gitops-and-automating-the-operation-of-stateful-apps-in-kubernetes. (accessed: 2.5.2020).

—— . 2017b. “Manifesto for Agile Software Development”. https://www.weave.works/blog/gitops-operations-by-pull-request. (accessed: 12.11.2019).

Wirth, N. 2008. “A Brief History of Software Engineering”. *IEEE Annals of the History of Computing* 30, no. 3 (): 32–39. doi:10.1109/MAHC.2008.33.

Wurster, Michael, et al. 2018. “Modeling and Automated Execution of Application Deployment Tests”. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, 171–180. IEEE.

Appendix A: Questionnaire template

- What is your role and background?
- Can you describe your software and architecture? (Background and general knowledge about project)
- Can you describe your deployment process?
- Did you make an acknowledged decision to use this kind of deployment
  - What was the motivation?
  - What did you consider this to be beneficial?
  - Were did you expect challenges?
- What is your perception of your current deployment status?
  - What has been successful?
  - What has given problems?
  - What would you do now differently?
- Is there deployment practices that you consider to be more important than others?
  - Has something proven to be more important than originally thought?
- Have you planned to improve your deployment process in the future
  - Why these in particular?
- Could you consider GitOps if not already doing it?
  - Gutfeels / perception?
  - How your current deployment differs from the model?
  - Can you ever see yourself in this model?
  - What it would require?
  - Does this kind of working make any sense? Do you think there would be any benefits?