Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Vitalii Ivanov

Implementation of DevOps pipeline for Serverless Applications

Master's Thesis
Espoo, May 27, 2018

Supervisor:	Professor Kari Smolander, Aalto University
Advisor:	Jussi Kyröhonka M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Vitalii Ivanov		
Title: Implementation of DevOps pipeline for Serverless Applications			
Date:	May 27, 2018	**Pages:**	78
Major:	Software and Service Engineering	**Code:**	SCI3043
Supervisor:	Professor Kari Smolander		
Advisor:	Jussi Kyröhonka M.Sc. (Tech.)		

Serverless computing is a cloud computing execution model where server-side logic runs in the stateless compute containers that are event-triggered and usually fully managed by vendor hosts such as AWS Lambda. This approach is also called Function as a Service (FaaS). Applications that rely on this approach are called Serverless applications. Serverless usage promises infrastructure cost reduction and automatic scalability. One more important benefit of serverless is making the operations part of DevOps process simpler. It reduces the time on the management and maintenance of the servers and sometimes makes them even completely unnecessary. Despite this fact, applications using serverless computing model require a new look at DevOps automation practices since it is a new approach to software architecture design and software development workflow.

The goal of this thesis is to implement DevOps pipeline for a Serverless application within a single case organization and evaluate the results of implementation. This is done through design science research, where result artifact is a release pipeline designed and implemented according to the requirements for a new project in the case organization.

The result of the study is automated DevOps pipeline with implemented Continuous Integration (CI), Continuous Delivery (CD) and Monitoring practices required for the case project. The research shows that architecture of Serverless applications affects many DevOps automation practices such as test execution, deployment and monitoring of the application. It also affects the decisions about source code repositories structure, mocking libraries and Infrastructure as Code (IaC) tools.

Keywords:	DevOps, Continuous Integration, Continuous Delivery, Serverless, Design Science Research
Language:	English

Acknowledgements

I would like to express my deep gratitude to Professor Kari Smolander for his constructive suggestions and professional guidance during the planning and development of this research work. I would also like to thank Jussi Kyröhonka for his enthusiastic encouragement and valuable recommendations on this project. I am particularly grateful for the knowledge and experience given by academic staff of Aalto University. Finally, I wish to thank my family for their support throughout my study.

Espoo, May 27, 2018

Vitalii Ivanov

Abbreviations and Acronyms

AWS	Amazon Web Services
BaaS	Backend as a Service
CI	Continuous Integration
CD	Continuous Delivery
DB	Database
DSRM	Design Science Research Methodology
FaaS	Function as a Service
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
IT	Information Technology
PaaS	Platform as a Service
QA	Quality Assurance
SDK	Software Development Kit
VCS	Version Control System
VM	Virtual Machine

Contents

Chapter 1

Introduction

Over the last decade, DevOps has become important part of cultures of many successful companies. Large influence on DevOps was done by wide adoption of microservices, containers and cloud computing [4]. The next step in evolution of cloud based and microservice architecture is serverless computing - code execution model where cloud provider totally takes the responsibilities of operating system and hardware management. The purpose of serverless computing is to simplify operations part of DevOps, provide very scalable execution architecture and predictable pricing model, where platform users are charged depending on amount of consumed resources.

Despite the large number of materials investigating serverless use cases [51][2], scalability [43], cloud providers, platforms [42] and success stories of serverless computing [35], less studies have focused on DevOps practices supporting Serverless applications. Even if serverless computing is by design aimed at simplification of operation processes, it still requires the changes of DevOps practices. How exactly DevOps practices should be implemented depends on specific use case of serverless computing and concrete project. But since serverless concept is the same regardless of provider and platform, the experience from one project might be useful for many other software companies.

The aim of this thesis is to implement DevOps pipeline for Serverless application in order to ensure high quality of the software development in a single case company. The output will be DevOps automation pipeline which includes the practices of Continuous Integration, Continuous Delivery and Monitoring implemented according to the requirements of the company such as technology stack and software architecture. In order to develop this solution, this thesis will use design science research methodology. To ensure the usefulness of DevOps practices, a series of workshops and interviews with developers, QA engineers and DevOps specialists will be done. The implemented results will be demonstrated, evaluated and discussed with the development team of the project.

The scope of the thesis will be limited to one project. The features of the project will not be revealed. The architecture will be shown sufficiently to design and implement required DevOps practices without any extra details. Nowadays, the concept of DevOps includes Culture, Automation, Lean, Measurement and Sharing [14] [23]. This thesis is focused on technical aspects of DevOps and covers the following groups of practices: Continuous Integration, Continuous Delivery and Monitoring.

1.1 Problem statement

The thesis will answer the following research questions:

RQ1 *What are the requirements for the DevOps pipeline of the case project?*

RQ2 *How does Serverless architecture affect DevOps practices such as CI, CD and Monitoring of the application?*

RQ3 *How well does implemented DevOps pipeline fulfil company requirements?*

1.2 Structure of the Thesis

This paper is structured as follows. Chapter 2 reviews existing literature about DevOps and Serverless applications, as well as describes how to combine them together. Chapter 3 defines the research method and research questions that the thesis is supposed to answer. Chapter 4 describes the current state of the project, architecture and expectations for DevOps pipeline from development team. Chapter 5 provides the details about DevOps pipeline implementation. Chapter 6 presents evaluation of implemented pipeline and cost calculation. Chapter 7 answers the research questions and proposes new ideas for the future research. Chapter 8 summarizes the findings and draw conclusions from the research.

Chapter 2

Background

This chapter presents the results of literature review about impact of Serverless architectures on DevOps automation pipeline. It also introduces the key terms and concepts that will be used in the thesis. Section 2.1 explains the term of DevOps and summarizes the DevOps adoption benefits and problems. Section 2.2 examines which groups of DevOps practices are influenced by architecture of the application and provides their description. Section 2.3 introduces the term Serverless and describes what is Serverless application and architecture. Finally, section 2.4 reviews the implementation of DevOps practices for applications with Serverless architectures and demonstrates the research gap in this field. In addition, section 2.5 explains the notation that will be used to describe the CI and CD pipeline in chapter 5.

Information provided in this chapter is required to find the answers on research questions defined in chapter 3. Understanding of available DevOps practices and awareness of their adoption will help with elicitation of requirements in RQ1. Research about influence of Serverless architectures on design and implementation of DevOps pipeline will give the background for the RQ2.

2.1 Introduction to DevOps

DevOps is a relatively new term that appeared a decade ago [30]. Initially it was a word to explain the need of collaboration between development and operation teams. Till now DevOps does not have a single definition and many companies understand it differently [17]. For instance, possible definitions are "a way of collaboration in which processes are automated as much as possible", "aspect of organizational culture in which development and operations personnel work together closely" or "the principles and practices which are needed to create a scalable service infrastructure" [20].

Definition from Gartner IT Glossary says that *DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach. DevOps emphasizes people (and culture), and seeks to improve collaboration between operations and development teams. DevOps implementations utilize technology - especially automation tools that can leverage an increasingly programmable and dynamic infrastructure from a life cycle perspective* [1].

Definition from Gartner reflects five most important aspects of DevOps: Culture, Automation, Lean, Measurement and Sharing. These five words emphasize the most modern understanding of DevOps and can be abbreviated as CALMS. Short description of each of these aspects is presented below.

Culture

DevOps requires open culture with good communication and collaboration channels in the core of it. Collaboration is required not only between development and operations teams but also between all stakeholders within the company and even outside of it such as company clients. Good collaboration is the key of successful implementation of all other practices and sometimes it even requires changes in organization's structure [33]. Attributes of good culture are empathy, good working environment and support between employees [34].

Automation

Automation of routine tasks in software development makes their execution faster and reduces the risk of mistake [33]. The most repeated tasks in software lifecycle are building, testing and deployment. CI, CD practices supported by declarative description of the infrastructure called "Infrastructure as Code" (IaC) are aimed at elimination of manual actions.

Lean

Lean software development describes a set of principles aimed at waste minimization and maximization of the customer value. DevOps should help to find and address the causes of development wastes and it will allow to reduce product time to market which might be critical for the business needs. Possible practice here is Continuous Monitoring with the focus on immediate alarm in case of the problems [21]. It can help to find problematic places in the software and fix them quickly to create a value for the customer.

Measurement

Performance and usage data gathered from production environment allows to understand the product better. This data should drive decisions, improvements and changes to the system [34]. This principle is called ”putting efficiency and process into perspective” [33]. Measurement and visualization are important not only for application running but also for QA status and code quality data.

Sharing

This aspect of DevOps emphasizes knowledge sharing part of collaboration within organization. For instance, developers and operations should try to make product documentation accessible and understandable by both sides. Sharing process should be facilitated by knowledge management [19].

Adoption of DevOps

Adoption of DevOps expects cultural change in IT organization. Only then it will make a positive impact on the overall productivity. Understanding of adoption benefits as well as problems will help to select and propose the useful practices for every new company and project case avoiding possible issues.

The expected benefits of DevOps are similar in all selected research papers. They are related to faster feature release, improved quality assurance and enhanced collaboration within the team [40]. Interesting fact is that even the papers focused only on CI practice of DevOps list exactly the same benefits [44][39][48].

DevOps practices' adoption problems can be divided into two groups: technological and social [32]. Example of technological barriers is dissimilarity of development and production environments [40].

Example of social problems is ambiguity in the definition of DevOps which is called as one of the main failure reasons in DevOps adoption [40]. Elberzhager and Arif with the group of authors also emphasize the importance of common understanding of DevOps within the company [18]. Based on their studies they recommend starting with the small changes in a manageable context of a small project and define requirements for tools and processes to be utilized in the DevOps environment.

Continuous Integration, Delivery and Monitoring

Lean concept of flow introduced many continuous activities such as Continuous Planning, Continuous Security, Continuous Trust and Continuous Innovations [23]. These activities affect business and planning, development, operations and innovations processes within the company. Continuous Integration, Delivery and Monitoring are the most popular activities from the family of "continuous" and they became the cornerstones of successful DevOps implementation.

Continuous Integration can be defined as a software development practice where developers merge their code changes into the shared code base as often as possible and these changes are validated by running a code quality analysis tools, building the project and running automated tests. It allows to avoid integration problems when building and testing of the project is delayed for the last minute [24].

Continuous Delivery is the next step after CI that makes sure that a project is ready to be released at any time by request. It can be released to development, staging or production environments. One more similar term is Continuous Deployment which means that deployment of successful builds to chosen environment happens automatically after every commit to selected branches [29].

Continuous Monitoring is the set of practices to monitor applications' run-time behaviour for early detection of problems, such as performance degradation or infrastructure or business logic errors [23].

Together CI and CD practices can be called release engineering pipeline [17]. This pipeline can also be called with a more wide term - DevOps pipeline, especially if it includes additional practices such as Continuous Monitoring.

2.2 DevOps practices

From software architecture perspective DevOps has an impact on the development cycle including build, test, deployment and post-deployment monitoring phases [6]. This section presents the practices for effective work with all these phases of software development cycle. In addition, source control practices are combined into separate group because they play crucial role in code sharing and team collaboration and are also widely presented in DevOps literature. Analysis of the practices was done based on four popular books about DevOps [6][31][50][29] and one CD maturity checklist [37].

2.2.1 Source Control

VCS is used to store code history and share the code. Version Control System (VCS) allows developers to save the code changes and rollback to the previous versions in case of mistakes. It is also a collaboration tool to share the code within the team being a single storage of the code base where all changes are merged.

Branches are used for isolating work. Branching is a powerful feature of VCS. Branching strategy is a set of rules that describes when branches should be created, how they should be named and what is the purpose of the shared branches. This strategy should be chosen properly to maintain code base in a consistent way and isolate developers work to avoid blocking of development process because of unstable or unfinished code in the main branch.

Pre-tested merge commits. Commits into the shared branches should be tested before merging to avoid unstable state of the code base. This practice is connected to the Testing and QA practices that are described in section 2.2.3.

All commits are tied to tasks. Commits should be linked to the tasks to simplify project navigation and issues investigation. Developers can check the tasks descriptions linked to the commits and the opposite - all commits linked to the task are listed in issue tracker. Implementation of this practice is usually achieved through integration of VCS with an issue tracker and usage of conventional commit messages.

Version control DB schema changes. Database (DB) schema changes also should be under version control. It helps to automatically apply the new changes in the data model and recover the schema in case of the problems.

Release notes auto-generated. A lot of projects require release notes reports. Usually release notes are based on the commits to the VCS. Release notes auto-generation can help to save a lot of time on manual copying of commit messages and making the reports out of them.

Table 2.1 shows summary of source control practices in existing literature.

2.2.2 Build Process

Build process is run automatically on commit. Build process is a process of making an artifact from the source code. It should be automatically triggered to immediately notify developers about the issues. Faster feedback about build status helps to keep the project in a stable state. There are many build

	DevOps: A Software Architect's Perspective [6]	The DevOps Handbook [31]	Practical DevOps [50]	Continuous Delivery [29]	CD: Maturity checklist [37]
VCS is used to store code history and share the code	X	X	X	X	X
Branches are used for isolating work	X	X	X	X	X
Pre-tested merge commits	X		X		X
All commits are tied to the tasks			X		X
Version control DB schema changes		X			X
Release notes auto-generated					X

Table 2.1: Source Control practices

process automation solutions available, such as Jenkins[1] and CircleCI[2].

Build artifacts are managed by purpose-built tools, no manual scripts. Many common development routines are already simplified by automation tools. Artifacts managements is one of them. In case of cloud-based architectures there are many IaC solutions such as CloudFormation[3] and Terraform[4] that help to describe infrastructure in a declarative way and also maintain the artifacts such as AWS Lambda zip archives.

All artifacts have build versions with major version and commit or CI build number. To keep the history of artifacts and be able to rollback the project to the previous version it is helpful to have a versioning convention. Major, minor version and unique value such as commit or CI build number is a common example of such convention.

Dependencies are managed in a repository. All developers should use the same versions of project dependencies and, what is more important, the same dependencies should be used in production. To keep dependencies consistent among different environments it is recommended to store their versions in a repository.

[1]https://jenkins.io/
[2]https://circleci.com/
[3]https://aws.amazon.com/cloudformation/
[4]https://www.terraform.io/

Build environment based on VMs. Virtual machines (VMs) allow to isolate build environment and make it portable. These qualities help to achieve more stable and scalable building process comparing to build environments with manual configuration on physical machines.

Table 2.2 outlines presence of build process practices in existing literature.

	DevOps: A Software Architect's Perspective [6]	The DevOps Handbook [31]	Practical DevOps [50]	Continuous Delivery [29]	CD: Maturity checklist [37]
Build process is run automatically on commit	X	X	X	X	X
Build artifacts are managed by purpose-built tools, no manual scripts	X	X	X	X	X
All artifacts have build versions with major version and commit or CI build number	X	X	X	X	
Dependencies are managed in a repository	X		X	X	X
Build environment based on VMs	X		X	X	X

Table 2.2: Build process practices

2.2.3 Testing and QA

Automatic unit testing with every build. As with automatic build triggering, unit tests should be automatically triggered to give immediate feedback about the status of the project to fix it if it is required.

Code coverage and static code analysis is measured. Code coverage is a common approach to measure amount of unit tests and check which parts of the code base are not considered in the testing cases. Static code analysis helps to find the problems in the code based on community experience and coding style guides.

Peer-reviews. Peer-reviews allow to find the problems in the code that could not be found by static code analysis tools. In addition, peer-reviews increase the code awareness in the team and encourage knowledge sharing and collaboration.

Mockups & proxies used. Mockups and proxies allow to imitate behaviour of external components or systems that could be expensive, lead to much longer execution of the unit tests or even not required to test the targeted block of code.

Automated end-to-end testing. End-to-end tests purpose is to imitate the usage of the system by the end user with all dependencies and integrations. Execution of these tests should be automated to decrease a number of routine tasks making by QA engineers.

Integrated management and maintenance of the test data. Test data should be stored in VCS. It will help, for instance, to connect data with a certain version of database schema. Talking about the results of test execution, they should be easily accessible by project stakeholders and stored for statistics, such as code coverage dynamics, for chosen period of time.

Automated performance & security tests in target environment. Performance tests allow to understand how the system behaves under conditions of intensive usage. It helps to reveal hidden problems and bottlenecks of the system. Security tests are aimed at finding of software vulnerabilities that can be used by intruders to harm targeted company or its users.

Table 2.3 shows summary of testing and QA practices in existing literature.

	DevOps: A Software Architect's Perspective [6]	The DevOps Handbook [31]	Practical DevOps [50]	Continuous Delivery [29]	CD: Maturity checklist [37]
Automatic unit testing with every build	X	X	X	X	X
Code coverage and static code analysis	X	X	X	X	X
Peer-reviews	X	X	X		X
Mockups & proxies used	X	X	X	X	X
Automated end-to-end testing	X	X	X	X	X
Integrated management of the test data		X	X	X	X
Automated performance & security tests	X	X	X	X	X

Table 2.3: Testing and QA practices

2.2.4 Deployment

Fully scripted deployments. Manual task execution increases the risk of mistake. Multistep deployments should be fully scripted, especially for the production environments.

Auto deploy to the test environment after tests pass. This practice allows to reduce time on manual routine tasks and immediately notify team members about the issues in deployment process or test execution.

Standard deployments across all environments. Standard deployments help to find the problems that can happen in production on an earlier stage such as testing or staging deployment. Identical environments for development and production increase reliability of the tests and deployment scripts.

Database deployments. This practice expects the usage of database migration tools that automatically apply schema changes. Database migration script should be a part of deployment script. It will help to make deployment process completely automatic.

Table 2.4 summarizes presence of deployment practices in existing literature.

	DevOps: A Software Architect's Perspective [6]	The DevOps Handbook [31]	Practical DevOps [50]	Continuous Delivery [29]	CD: Maturity checklist [37]
Fully scripted deployments	X	X	X	X	X
Auto deploy to the test environment after tests pass	X	X		X	X
Standard deployments for all environments	X			X	X
Database deployments		X	X	X	X

Table 2.4: Deployment practices

2.2.5 Monitoring

Log aggregation. Log aggregation helps to deal with the large volumes of software execution log messages and provides centralized access to them.

Finding specific events in the past. Finding information in the logs history is a common operation and it should be simplified through the use of convenient log browsing tools. These tools can have graphic UI and index the logs for faster search operations.

Large scale graphing of the trends. Visualization of the trends with the monitoring dashboard helps to identify the problems in the application before they became critical. Useful trends can be memory consumption, CPU usage and system response latency.

Active alerting according to the user-defined heuristics. Timely notification about the issues in an application is an important factor of faster troubleshooting. Alerting can be done by email or SMS messages. Possible alerting heuristics are too high CPU or memory consumption, errors in the logs or long system response latency.

Tracing. Tracing allows to follow the request through all components of the application and measure how long each part of handling process takes. It is especially useful feature in applications with microservice architecture where one request can go through many services which might lead to complicated issue investigation process without enabled tracing.

Table 2.5 shows summary of monitoring practices in existing literature.

	DevOps: A Software Architect's Perspective [6]	The DevOps Handbook [31]	Practical DevOps [50]	Continuous Delivery [29]	CD: Maturity checklist [37]
Log aggregation	X	X	X	X	
Finding specific events in the past	X	X			
Large scale graphing of the trends (such as requests per minute)	X	X	X	X	
Active alerting according to the user-defined heuristics	X	X	X		
Tracing	X				

Table 2.5: Monitoring practices

2.3 Serverless Applications

The term "serverless" in the cloud context appeared in 2014 when Amazon launched its AWS Lambda[5] service [10]. AWS Lambda allows to deploy in-

[5]https://aws.amazon.com/lambda/

dividual functions to the cloud and pay only for their execution avoiding unnecessary expenses on the idle resources. This model is called Serverless Compute or FaaS and is defined as a cloud computing execution model where logic runs in the stateless containers that are event-triggered and fully managed by third party platforms [41]. The term ”serverless” also defines Backend as a Service (BaaS) model that describes applications that rely on extensive usage of services such as databases or authentication managers provided by external vendors [10]. Serverless does not mean that there is no server-side logic or servers in general. It emphasizes that developers can leave most of operational tasks related to the server maintenance such as operating system updates, fault-tolerance, scalability and monitoring to the cloud provider [5].

Currently FaaS is considered as one of the most modern step in evolution of public clouds [49]. The first one was Infrastructure as a Service (IaaS) that provides virtualized computing resources where user should configure operating systems by himself or herself. The next one was Platform as a Service (PaaS) which is built on top of IaaS and provides the services such as web-application hosting where provider is responsible for operating system maintenance. The wide spread of container technologies, such as Docker[6] and Kubernetes[7], made popular one more cloud computing model - Container as a Service that combines flexibility of IaaS and reduced maintenance cost of PaaS solutions [10]. The main difference of FaaS with earlier cloud computing models is that user pays only for actual code execution and has out of the box application scalability.

Usage of BaaS and FaaS together allow to build powerful applications with minimum expenses on infrastructure or server maintenance [43]. Applications that use BaaS, FaaS or both of them are called Serverless Applications or applications with Serverless Architecture [41].

Serverless applications have following benefits:

- *Low cost.* Serverless provides a pay-per-use model without a need to pay for idle resources. In addition, organization saves the resources on hardware maintenance. It applies not only to the expenses on hardware but also to the reduced labor cost. Some use cases show that migration of an application to the Serverless architecture can reduce the cost by between 66% and 95% [2].

- *Enhanced scalability.* Having huge amount of calculation resources, the cloud provider can easily do horizontal scaling of the containers running

[6]https://www.docker.com/
[7]https://kubernetes.io/

the deployed functions. It allows to execute thousands of operations concurrently without additional configuration [10].

- *Decreased time to market.* Time and resources that were previously spent on server maintenance now can be aimed at feature development and delivery. The use of external BaaS services promotes prototyping and lean approach which might be especially helpful for startups [10].

Serverless applications have their weaknesses and limitations as well:

- *Latency.* FaaS solutions are based on containers. Even if container is considered as the lightweight alternative to VM it is still takes from tens to hundreds of milliseconds to run it. It explains the problem of cold start that might make this Serverless computing as inappropriate for some use cases which require real-time processing [2][10].

- *Security.* Developers have access only to the security settings provided by the cloud platform. The fact that the code runs in a shared environment with many other applications cannot make it absolutely secure [10][2].

- *Limited life-span.* AWS Lambda execution has a limit of 5 minutes which makes it unacceptable for some long-running processes like keeping open HTTP connection and receiving a stream of data for a long period of time. Some use cases, such as big data analysis, can eliminate this problem through workflow chains [2].

- *Vendor lock-in.* Code running in Serverless environment is usually highly dependent on other services such as database, logging or API mapper provided by the same platform which can lead to the vendor lock-in [2].

The common use cases of Serverless applications are websites, chatbots, triggered processing, scheduled events and big data processing. CPU-intensive long running tasks and real-time processing such as multiplayer-intensive games should be avoided [53][52]. The typical Serverless application relies on third party cloud platform and uses the following services: API mapper, database, binary storage, logging solution, notification service, infrastructure provisioning tool and Serverless computing service by itself [47][12][53].

Success of AWS Lambda service led to the raise of new FaaS platforms such as Google Cloud Functions [8], Microsoft Azure Functions [9] and IBM

[8]https://cloud.google.com/functions/

[9]https://azure.microsoft.com/en-us/services/functions/

OpenWhisk [10] [25]. The difference between these platforms is mostly in integration with other services of these cloud providers. It is also possible to deploy FaaS solution on-premises using open source platforms such as OpenWhick[11], Kubeless[12] or OpenFaaS[13]. Deployment of your own Serverless platform adds server maintenance overhead but gives better security control.

It is predicted that Serverless applications will become more and more popular due to high demand in edge computing to serve billions of mobile devices and impracticality of keeping idle servers in resource constrained environments [49].

2.4 DevOps for Serverless Applications

Literature review showed the lack of academic resources about DevOps for Serverless applications. At the same time, there are many blogposts about this topic mentioned below in this section. It confirms the observation made by Dingsøyr and Lassenius that continuous deployment and DevOps topics are industry rather than research driven [15].

The common pattern observed in existing literature is that cloud-based architectures encourage the use of DevOps through decomposition of the system into smaller and more manageable components which leads to smaller teams and simplifies decision making about each single component comparing to large monolith systems [4][13]. Serverless computing, being a next step in evolution of cloud computing as shown in the previous section, is sometimes described even as the No-Ops solution [11], but in practice, it still requires CI/CD pipeline and maintenance operations [7].

Serverless gives the following advantages to DevOps process:

- *Operability from the start.* Serverless approach combines development and operations. Even the simplest workflow with Serverless functions expects deployment to the cloud platform. And this deployment requires operational decisions such as amount of memory for calculations [26]. Serverless promotes a culture where software is developed to be ready for production right from the start [3].

- *Broad skillset but less specialists are required.* Serverless erases the difference between development and operations specialists. Their skills

[10]https://www.ibm.com/cloud/functions
[11]https://openwhisk.apache.org/
[12]https://github.com/kubeless/kubeless
[13]https://github.com/openfaas/faas

can be merged into single role of a cloud engineer. It makes the specialists more universal and ready to solve all Serverless platform related problems [26]. In general, Serverless application maintenance requires less specialists comparing to on-premises deployments because it does not need hardware infrastructure or even virtual environments support [9].

Discovered DevOps challenges for Serverless applications are:

- *Local debugging is complicated.* Serverless approach expects application execution in a cloud and usually together with other cloud services. It makes difficult to reproduce execution environment in local debugging and requires extensive usage of mocking libraries [5].

- *Limited access and monitoring.* Developers no longer have access to the servers to monitor behaviour of the applications on operating system level. They only can read the messages that were written by the applications and use cloud platform logging and tracing services. Traditional tools for monitoring and server access are not applicable anymore [5].

Serverless applications make CI and CD practices "a new normal"[26]. Resulting infrastructure can be easily modified because serverless expects storage of configurations and business logic together in the same repository. This close connection between business logic and infrastructure together with atomic nature of serverless functions makes deployments and rollbacks especially simple with a help of IaC tools.

Literature review helped to identify only few aspects of Serverless applications that directly affect DevOps process and could be called as Pros and Cons of Serverless. But at the same time, almost all materials mentioned that this new cloud computing model requires properly designed CI/CD pipeline [9] development of which is the main goal of this work.

2.5 Integration flow model

This section presents a notation for description of software integration flows. It will be used in chapter 5 to visualize implemented CI and CD pipeline. This notation was proposed by Ståhl and Bosch to address the variation points in continuous integration flows [46]. Later the authors of the model improved it based on the feedback from the integration engineers and successfully used it to describe and compare different CI implementations in another work [45]. The reason for using an established descriptive model in this study is to make representation of developed DevOps pipeline comparable to existing and future research simplifying experience exchange.

The model includes five elements to describe software integration flows. Figure 2.1 shows these elements and their possible relations. *Input nodes* (triangles) represent sources that provide input data to the model. *Trigger nodes* (circles) describe external triggering factors. *Activity nodes* (rectangles) perform actions on data input. *Input edges* (dashed arrows) show the flow of data between the elements. *Trigger edges* (solid arrows) define the conditions for triggering the activity nodes.

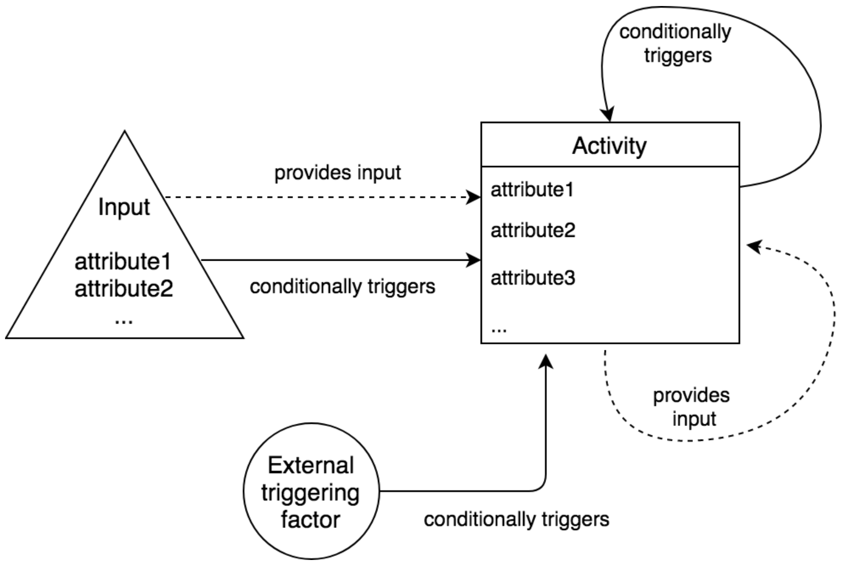

Figure 2.1: The elements of the Integration flow model

This chapter showed a context of the work and reviewed existing literature about DevOps, Serverless and their interconnection. This knowledge is required to achieve the objectives of the work and answer the research questions defined in the next chapter.

Chapter 3

Research Methods

This chapter describes research motivation and defines research questions. It also presents the design science research method that will be used to achieve the goals of the study. Furthermore, the data collection process is described.

3.1 Research motivation

While many studies on the DevOps theory and adoption have been made [33][20], there are still not enough research on DevOps specific for architectural patterns, in our case for Serverless applications. Every new project with Serverless architecture has a risk to face the issues with maintenance and code quality because there are not enough materials and best practices about Serverless development. Despite the fact that Serverless promises many benefits comparing to monolith and microservices in terms of operations, it is still unclear how much effort does it take to maintain DevOps pipeline for Serverless application. Available materials are usually focused on simple cases with only few Serverless functions and cannot be considered as the research works [7][27]. At the same time the usage of tens and hundreds of Serverless functions can lead to maintenance overhead. The reason can be atomic nature of Serverless functions that leads to many challenges. Some of the challenges are: should each function has a separate VCS repository, version and CI config file, should they share the common unit test data or not? The answers on these questions depend on every specific team and project, but the findings from a single case can be useful for the decisions about new projects.

The case organization studied in this thesis made a decision to use Serverless architecture for one of the projects. Preliminary analysis that was made within the company showed that the usage of Serverless model might be beneficial for the project. Architecture of the project is also ready. The main question is how to organize DevOps pipeline to satisfy the expectations of the company. Since it is a new project there is no data to compare the new pipeline with the previous one, so it is required to understand not only how to build the pipeline but also how to evaluate it.

This motivation and challenges allow us to define three research questions:

RQ1 *What are the requirements for the DevOps pipeline of the case project?*
The aim is to understand the company's needs and expectations about the DevOps pipeline and how product development workflow should be designed considering available team resources, knowledge and possible limitations such as a cost of cloud services usage. It allows to propose and implement specific DevOps automation practices to support company processes.

RQ2 *How does Serverless architecture affect DevOps practices such as CI, CD and Monitoring of the application?*
The purpose of this question is to identify the expectations of stakeholders regarding the usage of Serverless architecture, define the influence of Serverless approach on DevOps automation practices and understand how to emphasize the opportunities and reduce the risks introduced by Serverless approach in the new DevOps pipeline.

RQ3 *How well does implemented DevOps pipeline fulfil company requirements?*
Does the implemented pipeline match the expectations from DevOps process identified with RQ1? Does it allow to emphasize benefits provided by Serverless architecture and reduce the risks of its usage identified with RQ2? What is the cost of execution of implemented pipeline on the chosen cloud platform?

3.2 Research method

The method used in this thesis is design science research. The key idea of design science is to produce and evaluate IT artifact intended to solve identified organizational problems [28]. This method was chosen because the expected outcome of the work should be an artifact - designed and implemented DevOps pipeline. To achieve this goal multiple problems should be addressed: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation and communication. All these activities and research examples are described in the guideline by Peffers and Tuunanen [38].

The design science research methodology (DSRM) process model is shown in Figure 3.1.

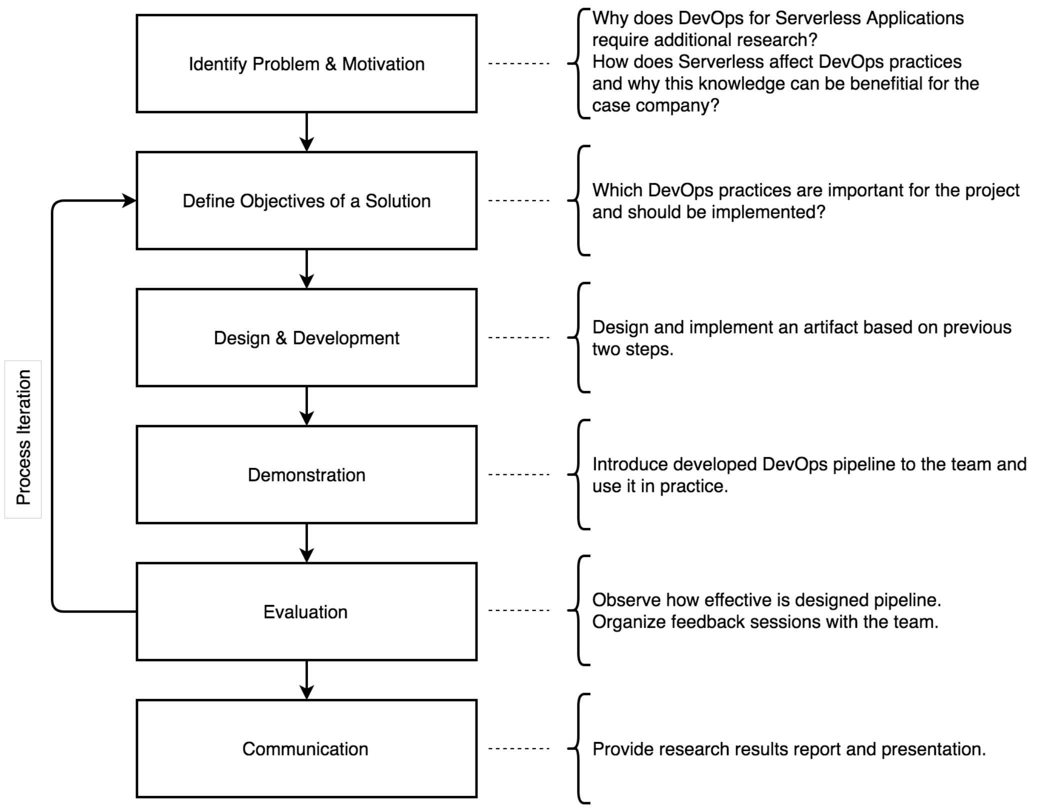
Identify Problem & Motivation
Why does DevOps for Serverless Applications require additional research?
How does Serverless affect DevOps practices and why this knowledge can be benefitial for the case company?
Define Objectives of a Solution
Which DevOps practices are important for the project and should be implemented?
Design & Development
Design and implement an artifact based on previous two steps.
Demonstration
Introduce developed DevOps pipeline to the team and use it in practice.
Evaluation
Observe how effective is designed pipeline.
Organize feedback sessions with the team.
Communication
Provide research results report and presentation.
Process Iteration

Figure 3.1: DSRM Process Model

The result of the thesis is the DevOps pipeline made by DSRM guidelines with four iterations of improvements. Every iteration was a two-week Scrum sprint. In the beginning of the first sprint and at the end of the last sprint special workshops were organized to gather the data required to answer the research questions of this work. The details of these workshops are described in Section 3.3

3.3 Data collection

Problem and motivation identification was done by the author of the thesis based on the personal experience in the case company and preliminary literature review. It was later discussed and approved by the stakeholders within the company. In order to gather requirements, define the objectives and evaluate the solution, a series of workshops was organized. First workshop to define the objectives was the longest one and its duration was four hours. The workshop was attended by DevOps engineer, team lead, two developers and QA engineer. The participants were suggested to discuss CI, CD and Monitoring practices, evaluate their importance, efforts on implementation and answer the question how does Serverless architecture affect these practices. Based on the results of the first workshop first version of the pipeline was designed and discussed with the development team. After the discussion it was modified and introduced into the project workflow. After this, the process was repeated in an iterative way: evaluation and objectives definition discussion, then design and development, then demonstration and usage. These steps were repeated four times with an interval of two weeks. The final review session was organized as a workshop that was attended by senior DevOps engineer, team lead, two developers and QA engineer. The goal of the final workshop was to get the feedback about implemented DevOps pipeline. The content of the workshop was organized in a following way: the author presented the architecture of the project, emphasised the Serverless parts of it and described all steps of DevOps pipeline execution. The participants were free to ask the questions and give the suggestions during the presentation. At the end of presentation, the participants answered the questions what the best parts and risky parts of the pipeline are and also shared their opinion in a free form.

In addition to the workshops within the company the author interviewed two developers and one QA engineer outside of the case company to get their opinion about influence of Serverless architecture on DevOps automation practices.

Designed DevOps pipeline was the first pipeline for the current project, which means there were no quantitative or qualitative data available about previous DevOps process. Data gathered from the workshops was mostly qualitative. Every new review session provided more qualitative data about current version of DevOps pipeline. For the final pipeline the author made an estimation of the cost based on the pricing model of chosen cloud platform.

Data collected in the first workshop helped to answer the RQ1 and RQ2. Interviews with the developers outside of the company and background research contributed into answering RQ2. The final workshop was the main source of data to answer the RQ3.

This chapter described research motivation, research questions and introduced DSRM together with data collection process. Next chapter will present the case project and requirements for the DevOps pipeline.

Chapter 4

Current State Analysis

This chapter introduces the case project and provides the results of DevOps pipeline requirements elicitation. Case project introduction provides its architecture description and explanations of Serverless design patterns used in it. The most important part of the chapter is description of the results of DevOps practices workshop. These results help to understand company's requirements for the DevOps pipeline.

4.1 Project architecture

The thesis does not descibe the domain field of the project. The focus is made on Serverless aspects of it. In addition, the architecture of the project is simplified to visualize it in a more concise way emphasizing the Serverless parts of it. The project was designed with the usage of Amazon Web Services including AWS Lambda. Rationale of cloud platform provider choice and architectural decisions is out of scope of this work. These decisions are accepted as an input of this research. Architecture of the project is shown using context and container diagrams from C4 model [8].

Context diagram of the project is shown in Figure 4.1.

Project described in current work is a web service for binary files processing. For authentication it uses third party authentication service.

Container diagram shows architecture of the system in more details. All containers except web client should be hosted by Amazon cloud platform and use particular AWS services. Icons inside of the blocks are official icons of Amazon Web Services which correspond to the container names. The case

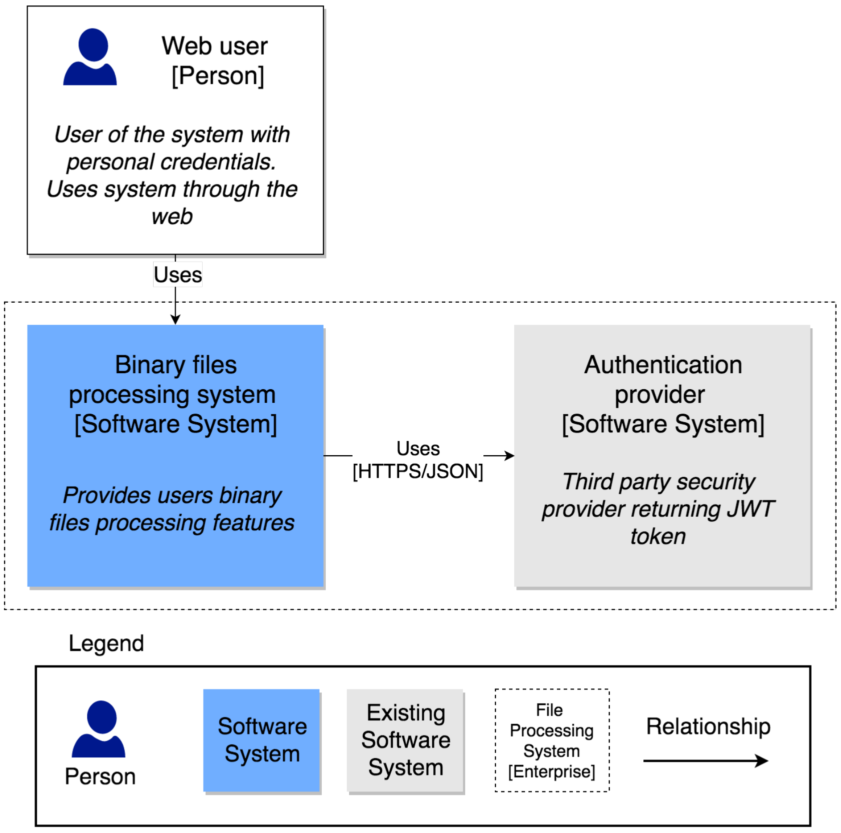
Web user
[Person]
User of the system with personal credentials. Uses system through the web
Uses
Binary files processing system
[Software System]
Provides users binary files processing features
Uses
[HTTPS/JSON]
Authentication provider
[Software System]
Third party security provider returning JWT token
Legend
Person
Software System
Existing Software System
File Processing System
[Enterprise]
Relationship

Figure 4.1: Case project context diagram

project uses following Amazon services: API Gateway[1], AWS Lambda[2], S3[3], DynamoDB[4] and SNS[5]. Web client is a web application running in a browser. Component diagram is not presented, because it would repeat container diagram due to atomic design of Lambda functions where each Lambda is responsible for one particular feature. Code diagram is not presented, because such level of details is not required to make decisions about DevOps pipeline.

Container diagram is shown in Figure 4.2

[1]https://aws.amazon.com/api-gateway/
[2]https://aws.amazon.com/lambda/
[3]https://aws.amazon.com/s3/
[4]https://aws.amazon.com/dynamodb/
[5]https://aws.amazon.com/sns/

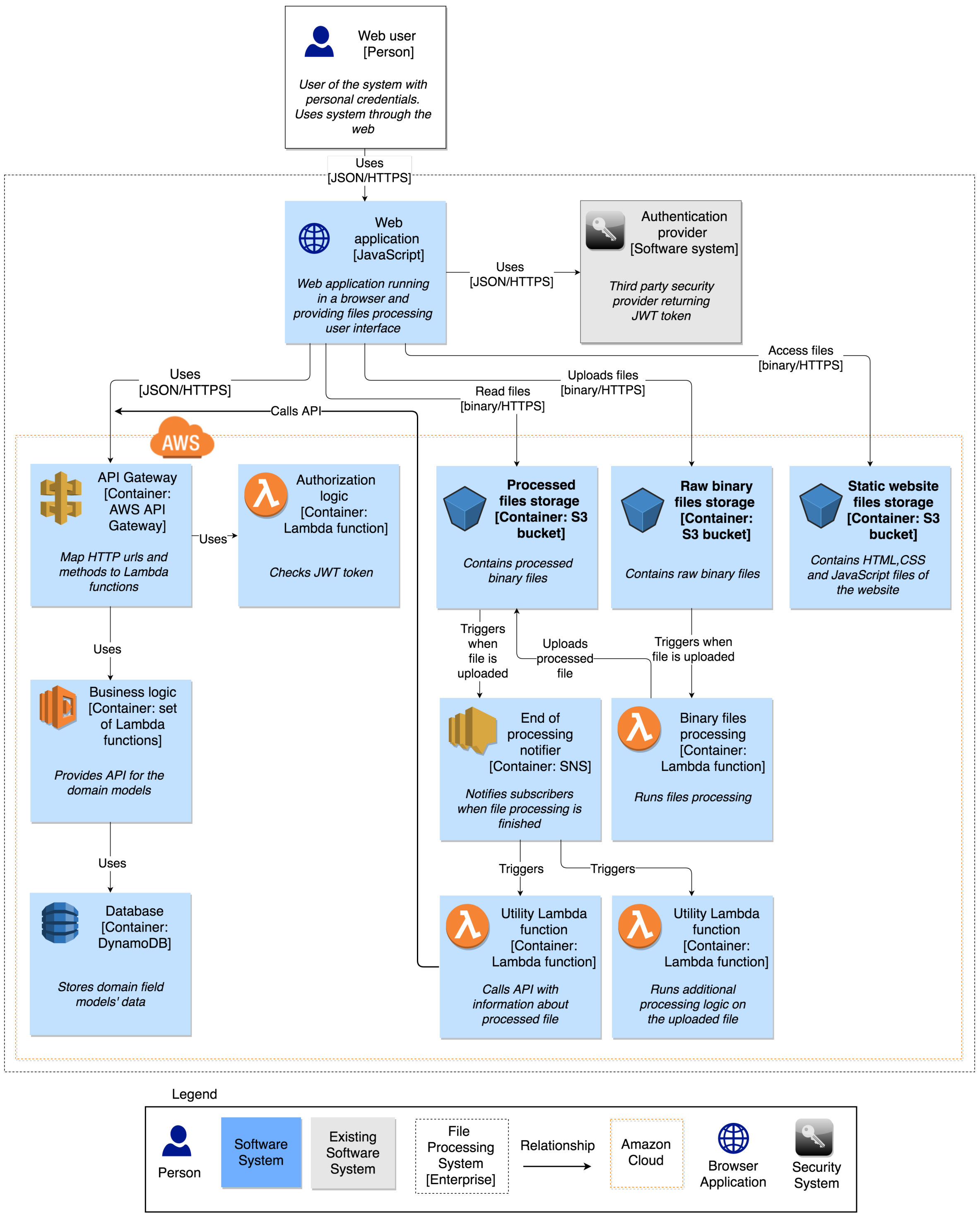
Web user
[Person]
User of the system with personal credentials. Uses system through the web
Uses
[JSON/HTTPS]
Web application
[JavaScript]
Web application running in a browser and providing files processing user interface
Uses
[JSON/HTTPS]
Authentication provider
[Software system]
Third party security provider returning JWT token
Access files
[binary/HTTPS]
Uses
[JSON/HTTPS]
Uploads files
[binary/HTTPS]
Read files
[binary/HTTPS]
Calls API
AWS
API Gateway
[Container: AWS API Gateway]
Map HTTP urls and methods to Lambda functions
Uses
Authorization logic
[Container: Lambda function]
Checks JWT token
Processed files storage
[Container: S3 bucket]
Contains processed binary files
Raw binary files storage
[Container: S3 bucket]
Contains raw binary files
Static website files storage
[Container: S3 bucket]
Contains HTML,CSS and JavaScript files of the website
Uses
Triggers when file is uploaded
Uploads processed file
Triggers when file is uploaded
Business logic
[Container: set of Lambda functions]
Provides API for the domain models
End of processing notifier
[Container: SNS]
Notifies subscribers when file processing is finished
Binary files processing
[Container: Lambda function]
Runs files processing
Uses
Triggers
Triggers
Database
[Container: DynamoDB]
Stores domain field models' data
Utility Lambda function
[Container: Lambda function]
Calls API with information about processed file
Utility Lambda function
[Container: Lambda function]
Runs additional processing logic on the uploaded file
Legend
Person
Software System
Existing Software System
File Processing System
[Enterprise]
Relationship
Amazon Cloud
Browser Application
Security System

Figure 4.2: Case project container diagram

4.2 Serverless elements of application architecture

Container diagram shows that all backend logic of the application is stored in AWS Lambda functions. Solutions used for data storage, such as DynamoDB and S3 buckets, also can be considered as Serverless because they do not require any server configuration. It means that case project can be considered as a project with clean Serverless architecture using FaaS and BaaS services. Serverless use cases and design patterns used in the project are described below.

Use case: Use AWS Lambda with AWS services as event sources
Description: Event source publishes an event that triggers Lambda function. This use case ensures loose coupling between the software components and is popular in event-driven architectures. In the case project S3 bucket triggers Lambda function when file uploading is finished. After that Lambda function can run some business logic to process uploaded file.

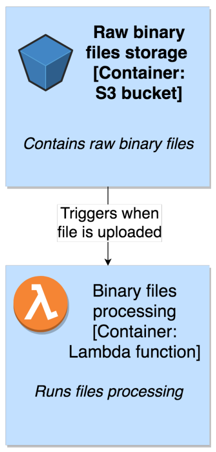
Raw binary files storage [Container: S3 bucket]
Contains raw binary files
Triggers when file is uploaded
Binary files processing [Container: Lambda function]
Runs files processing

Figure 4.3 Event source use case

Use case: On-demand Lambda function invocation over HTTPS (Amazon API Gateway)
Description: Lambda function is invoked over HTTPS. In this case API Gateway is configured to map particular HTTP request on specific Lambda function. This configuration contains HTTP method, URL path and authentication type. API Gateway together with AWS Lambda is the most typical way of design and implementation of REST API endpoints with AWS cloud infrastructure in Serverless applications.

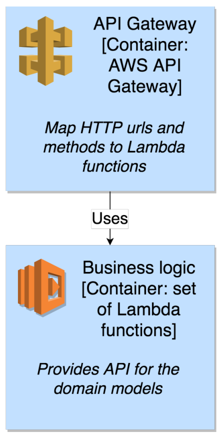
API Gateway [Container: AWS API Gateway]
Map HTTP urls and methods to Lambda functions
Uses
Business logic [Container: set of Lambda functions]
Provides API for the domain models

Figure 4.4 Lambda invocation over API Gateway

Use case: Fan-out pattern
Description: Fan-out pattern is used to distribute a message to all listeners subscribed on its publisher. In the case project SNS triggers two Lambda functions when file processing is finished. This pattern is also considered as a way to create event-driven architectures and perform multiple operations in parallel.

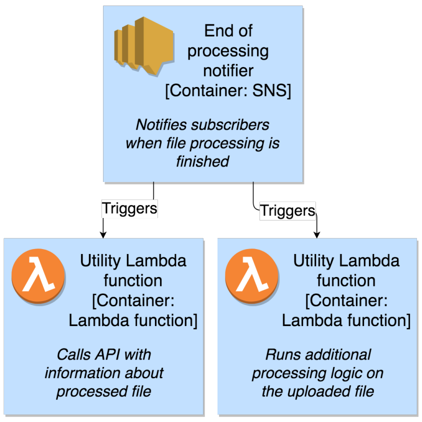
End of processing notifier [Container: SNS]
Notifies subscribers when file processing is finished
Triggers
Triggers
Utility Lambda function [Container: Lambda function]
Calls API with information about processed file
Utility Lambda function [Container: Lambda function]
Runs additional processing logic on the uploaded file

Figure 4.5 Fan-out pattern

Use case: Custom Authorizer
Description: Custom Authorizer is a design pattern where API Gateway endpoints call Lambda function to authorize the requests. In the case project Lambda function checks that JWT token passed in HTTP header is valid.

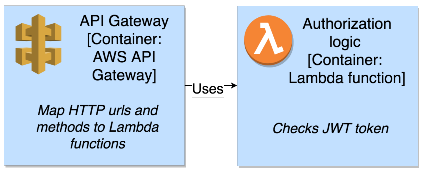
API Gateway [Container: AWS API Gateway]
Map HTTP urls and methods to Lambda functions
Uses
Authorization logic [Container: Lambda function]
Checks JWT token

Figure 4.6 Custom Authorizer

4.3 Expectations for Serverless Applications

The author of the work interviewed two developers and one manager about their expectations for Serverless architecture of the project. It was done not to evaluate the architecture but to better understand what a motivation to use Serverless is and how it can be considered in design and implementation of DevOps pipeline. The results of interviews are provided below. They showed that expectations and concerns mostly correspond the benefits and potential drawbacks of Serverless architectures explored in section 2.4.

Benefits

- Scalability of the project without additional configuration. The cloud platform takes care of execution of the code and scales it automatically.

- Faster development because there is no need to spend time on server configuration and updates. Cloud provider takes care of it.
- Pay-per-use policy of FaaS allows to save money on idle services.

Risks

- Possible vendor lock-in. But this is a justified risk because of all positive aspects of Serverless. To eliminate it the developers are going to use more interfaces and abstractions in their code which is in general considered as a good practice of coding.
- Serverless and cloud architectures usually expect Infrastructure as Code approach which might require DevOps skills from the developers. Not all developers might be ready to take additional responsibilities. From the other side, nowadays it is a normal practice that the developers expend their skills to DevOps, especially if it matches the culture of the company.
- Debugging can be more complicated because the only real environment of application execution is cloud platform environment. But the good code coverage by unit tests and extensive usage of mocking libraries can help to eliminate this problem.
- Lack of experience with Serverless might lead to some hidden problems. For instance, how to organize the VCS repositories and structure the code.

Even if interviewees were mostly speaking about the risks of Serverless applications, they were still very positive about it. The reason was that the benefits of the usage are worth of taking the risks. And for every risk the developers provided some ideas of how to eliminate it.

Some of the listed benefits and risks affect DevOps decisions. For instance, IaC might require organizing of the code in a certain way - i.e., store the infrastructure code together with business logic code. Serverless functions debugging limitations increase the necessity of unit tests and their automatic execution after every commit. Fear of vendor lock-in is raised because of extensive usage of external services provided by cloud platform. All integrations should be tested with the help of mocking libraries at the unit tests stage and also at the stage of end-to-end tests in the production-like environment.

4.4 Elicitation of requirements for DevOps pipeline

As was described in section 3, to elicit requirements for DevOps pipeline the author organized workshop within the case company where it was proposed to discuss CI, CD and Monitoring practices for the project and gather the opinions of participants how does Serverless architecture affect these practices.

All practices were grouped into five categories: Source Control, Build Process, Deployment, Testing & QA and Monitoring. For each practice the participants discussed Implementation ideas, answered the question how does Serverless architecture affect practice implementation and estimated how important the practice is.

In addition to the workshop, the author interviewed two developers and one QA engineer outside of the case company and asked their opinion how does Serverless architecture affect DevOps practices. In the beginning of the workshop and interviews author made sure that by Serverless participants understand extensive usage of FaaS services and cloud databases provided by third-party cloud platforms. Following sections present the results of the workshop and provide summary table with practices sorted by importance for the case project.

The only one technical requirement for DevOps pipeline was the usage of GitLab Community Edition platform because it was already in use in the company and preliminary analysis showed that it can be used for Serverless applications as well. Implementation of the case project is done with Node.js framework which affects the choice of unit test and mocking libraries.

4.4.1 Source Control

VCS is used to store code history and share the code

Serverless influence: Serverless architecture affects the decision of how many code repositories to create. In case of monolithic architecture, the source code is tightly coupled and is usually stored in a single repository. Serverless architecture expects presence of multiple deployable units or functions that can be completely isolated from each other. Source code of these deployable units, for instance, of AWS Lambda functions, can be stored in multiple repositories with one repository per function approach or in one single repository for all functions. One more approach is to create several repositories, each of which will store multiple functions based on their purpose or domain field.

Implementation ideas: VCS should be Git since it has very efficient branching model and team already has an experience with Git and GitLab. A number of created repositories should be defined based on the good practices of Serverless applications code maintenance.

Branches are used for isolating work

Serverless influence: Branching model does not depend on the architecture of the application. It can be the same for applications with monolithic, microservice or serverless architecture. The team can decide to use only master branch or extensively use feature branches depending on the team work practices.

Implementation ideas: Every feature and bugfix should be done in a separate branch. In addition, personal forks can be used to hide personal branches. Merges should be performed often to avoid merge conflicts.

Pre-tested merge commits

Serverless influence: As in the case of branching model, practice with pre-tested merge commits is not affected by Serverless architecture.

Implementation ideas: GitHub open source development through merge requests can be used as a reference. Every merge request can be merged only after the unit tests were successfully executed. GitLab supports this feature.

All commits are tied to the tasks

Serverless influence: Fulfillment of this practice is ensured only by Git commit message conventions and integration with issue tracker.

Implementation ideas: Issue management system and VCS should be connected to bind tickets and commits. Jira and GitLab can have such integration. Team should decide about Git messages conventions to connect the commits to the tickets.

Version control DB schema changes

Serverless influence: Serverless approach does not affect this practice. Schema changes should be stored as IaC and be launched before deployment of the new version automatically.

Implementation ideas: DynamoDB does not require version control of schema changes since it is a flexible NoSQL DB.

Release notes auto-generated

Serverless influence: Release notes content depends on how Serverless functions are grouped. In case if they are stored in separate repositories, then release notes can be unique for each function. If they are grouped, then release notes describe a set of functions or even all functions.

Implementation ideas: Each release version should be tagged with a Git tag. Tag creation can run a script to gather commit history and put it into release notes.

4.4.2 Build Process

Build process is run automatically on commit

Serverless influence: Build process triggering does not depend on the software architecture. It depends only on the build automation system.

Implementation ideas: GitLab Runners can be triggered right after the commit to run the build process.

Build artifacts are managed by purpose-built tools, no manual scripts

Serverless influence: In case of Lambda functions artifact is a zip archive. Purpose-build tools such as Terraform and CloudFormation should support management of the artifacts. Serverless affects what kind of purpose-built tools to use and how to store and deploy the artifacts.

Implementation ideas: Build artifacts can be managed by IaC tool such as CloudFormation or Terraform and stored in AWS S3 buckets.

All artifacts have build versions with major version and commit or CI build number

Serverless influence: Instead of API versioning Lambda functions usually have their own version number and alias. It affects the versioning policies and the way how Lambda functions are called.

Implementation ideas: Every service and Lambda function zip archive should have a major and minor version. In addition, they should have a CI build number suffix.

Dependencies are managed in a repository

Serverless influence: Dependency management of Serverless functions has no differences with non-serverless project. The tools depend on the using language.

Implementation ideas: All project dependencies and their versions should be defined and stored in VCS.

Build environment based on VMs

Serverless influence: Serverless functions can be built inside of the VMs or Docker containers as all other project types. Serverless does not affect this practice.

Implementation ideas: GitLab Docker Runners can be used to provide build containers that can be easily updated and reused.

4.4.3 Testing and QA

Automatic unit testing with every build

Serverless influence: Triggering of unit tests does not depend on project type or architecture. It depends on the CI system.

Implementation ideas: Test stage of GitLab CI should be run after each commit.

Code coverage and static code analysis is measured

Serverless influence: Code coverage and static code analysis tools are not affected by Serverless architecture, but atomic nature of Serverless raises a question: should code coverage and static code analysis be measured for every Lambda function separately or for the set of Lambda functions of the same domain group.

Implementation ideas: Code coverage and static code analysis tools should be chosen depending on the project programming language.

Peer-reviews

Serverless influence: Serverless architecture does not affect peer-review practices, but number of review requests may be increased because of fragmented codebase in case if every function is stored in separate repository.

Implementation ideas: Peer-reviews can be implemented using GitLab merge request feature.

Mockups & proxies used

Serverless influence: Mockup and proxy libraries are used more extensively. The code should be designed to support mockups, e.g., through the use of interfaces.

Implementation ideas: Mockup and proxies libraries should be chosen depending on the project programming language and used cloud services.

Automated end-to-end testing

Serverless influence: Since many of the functions are event-driven and are called asynchronously, writing end-to-end tests might be more cumbersome comparing to non-serverless computing. On the other hand, triggering end-to-end tests does not depend on the architecture and depends only on CI automation system. In addition, end-to-end tests execution price should be calculated to avoid extra cost. High price of execution and as well as long running time might be the reason to run the tests nightly.

Implementation ideas: End-to-end tests should be triggered automatically or manually from the CI system. The decision should be based on test execution price and duration.

Integrated management and maintenance of the test data

Serverless influence: Serverless architecture affects the decision of how to organize the test data. Atomic Lambda functions create a risk of duplication of the test data. For instance, if multiple Lambda functions work with the same database, then database initial test data has to be duplicated in all functions. Decision to store Lambda functions of the same group in one VCS repository can help to share test data among several Lambda functions. At the same time, test results management and maintenance are not affected by Serverless architecture and depends on the chosen CI solution and testing framework.

Implementation ideas: Test data should be stored within the project repositories. Lambda functions code should be organized in a way to avoid test data duplication. Test reports can be stored in GitLab CI as the artifacts.

Automated performance & security tests in target environment

Serverless influence: The nature of Serverless computing make the production-like deployment possible only on infrastructure of the cloud provider.

Implementation ideas: Performance and security testing should be done in production-like environment using separate AWS account.

4.4.4 Deployment

Fully scripted deployments

Serverless influence: Serverless architecture influences a choice of cloud provider and deployment tool. Deployment tool should be scripted in a certain way.

Implementation ideas: Deployment of Lambda functions can be done with CloudFormation, Terraform or Serverless Framework tool. Any of these tools can be run by GitLab CI.

Auto deploy to the test environment after tests pass

Serverless influence: Script triggering depends on CI pipeline tool and does not depend on the architecture of application. At the same time, if the team decides to keep all Lambda functions in separate repositories with their own deployment scripts, it can make the deployment of the project more cumbersome.

Implementation ideas: Deployment stage of GitLab CI pipeline should be triggered only after the test stage was passed.

Standard deployments for all environments

Serverless influence: Serverless environment depends on cloud provider. How exactly functions will be deployed depends on the Serverless architecture. On-premises Serverless solutions such as Kubeless and OpenFaaS can help to avoid testing environment and vendor lock-in problems but they would also require resources on server maintenance nullifying time-saving benefits of third party provider usage.

Implementation ideas: Infrastructure provisioning should be done with a help of IaC tools: CloudFormation or Terraform. The project should be deployed directly to the cloud but to different stages or even accounts.

Database deployments

Serverless influence: Serverless databases make database admin tasks unnecessary. Only schema changes or query changes are required. In case of NoSQL databases even schema changes are not required.

Implementation ideas: DynamoDB database will be deployed to the cloud as all other services. It also does not require database schema migrations, because it is NoSQL DB.

4.4.5 Monitoring

Serverless influence: Serverless logging capabilities are tightly coupled with the logging tools provided by the host platform. They even can be limited by them.

Log aggregation

Implementation ideas: AWS Lambda is automatically integrated with Amazon CloudWatch. All logs per one function/version are aggregated into CloudWatch group.

Finding specific events in the past

Implementation ideas: Use CloudWatch search feature that allows to search Log Groups, Log Streams and specific events within the Log Streams.

Large scale graphing of the trends (such as requests per minute)

Implementation ideas: Use CloudWatch Metrics that allow to customize dashboard with different metrics such as latency, calls counter, amount of errors, etc. For better visualization the tools such as Grafana can be used. It can be integrated with CloudWatch and provide better user experience comparing to the native Metrics tool.

Active alerting according to the user-defined heuristics

Implementation ideas: Use CloudWatch Alarms tool that allows to create alarms based on Metrics values. The alarm performs one or multiple actions based on the value of the chosen metric. The action can be a Lambda function call or a notification sent to Amazon SNS topic leading to email sending.

Tracing

Implementation ideas: Use Amazon X-Ray tool that allows to trace event sources that invoked Lambda functions or trace downstream calls that function made.

4.5 Results

The results of the workshops and interviews are summarized in the Table 4.1. The column "Importance" answers the question "How important is implementation of the practice for the case project". The result value is the average of the grades given by the participants of the workshop where 0 means "does not affect at all" and 10 means "strongly affects". The column "Affected by Serverless" answers the question "Is implementation of this DevOps practice affected by Serverless architecture". In case if at least one person said "yes", it was marked as "Yes".

The results of the workshop and interviews showed that usage of Serverless architecture affects more than half of suggested DevOps automation practices:

Practice	**Category**	**Importance**	**Affected by Serverless**
VCS is used to store code history and share the code	Source Control	10	Yes
Log aggregation	Monitoring	10	Yes
Build process is run automatically on commit	Build Process	9	No
Build artifacts are managed by purpose-built tools, no manual scripts	Build Process	9	Yes
Dependencies are managed in a repository	Build Process	9	No
Automatic unit testing with every build	Testing & QA	9	No
Automated end-to-end testing	Testing & QA	9	Yes
Fully scripted deployments	Deployment	9	Yes
Standard deployments across all environments	Deployment	9	Yes
Finding specific events in the past	Monitoring	9	Yes
Active alerting according to the user-defined heuristics	Monitoring	9	Yes
Branches are used for isolating work	Source Control	8	No
Peer-reviews	Testing & QA	8	No
Mockups & proxies used	Testing & QA	8	Yes
Large scale graphing of the trends (such as requests per minute)	Monitoring	8	Yes
Tracing	Monitoring	8	Yes
Pre-tested merge commits	Source Control	7	No
All artifacts have build versions with major version and commit or CI build number	Build Process	7	Yes
Build environment based on VMs	Build Process	7	No
Code coverage and static code analysis is measured	Testing & QA	7	Yes
Integrated management and maintenance of the test data	Testing & QA	7	Yes
Automated performance & security tests in target environment	Testing & QA	7	Yes
Auto deploy to the test environment after tests pass	Deployment	7	Yes
All commits are tied to tasks	Source Control	6	No
Database deployments	Deployment	6	Yes
Release notes auto-generated	Source Control	3	Yes
Version control DB schema changes	Source Control	0	No

Table 4.1: Importance of DevOps practices for the case company

- 2 out of 6 source control practices
- 2 out of 5 build process practices
- 5 out of 7 testing and QA practices
- 4 out of 4 deployment practices
- 5 out of 5 monitoring practices

  18 out of 27 practices in total are affected by Serverless Architecture

Such a significant impact of Serverless architecture on DevOps practices proves the relevance of current research work.

The participants of the workshop proposed to implement the practices that have a grade of Importance not less than 5. This criterion left only two practices out of scope of the developing DevOps pipeline. It proves that company requires reliable DevOps pipeline which would include almost all discussed practices and would guarantee high quality of the code through build process, QA and deployment automation, source control and monitoring.

The input condition for the DevOps pipeline was the usage of GitLab Community Edition. The AWS-based architecture of the case project also applied some limitations on the future DevOps pipeline such as usage of certain IaC tools and AWS cloud platform by itself. Node.js as the project implementation technology affects the choice of unit testing frameworks and mocking libraries. Jira was chosen as a project issue tracker that affects only one practice: "All commits are tied to tasks". No limitations for the cost of the pipeline execution were announced. The research plan expects two months of work on design and implementation of DevOps pipeline.

Chapter 5

Implementation

This chapter describes the implementation of the DevOps pipeline for the case project. It also justifies a choice of IaC tool. Pipeline implementation is based on requirements presented in chapter 4. Implementation was done in four iterations. Every iteration was a two-week Scrum sprint. This chapter describes only the final solution. The main changes that were proposed during the sprint review sessions are provided at the end of the chapter. Evaluation of the implemented pipeline is described in chapter 6.

5.1 Architecture

One of the requirements for the DevOps pipeline was usage of GitLab[1] solution. Requirements elicitation workshop and preliminary analysis of the tool showed that it can be successfully used to implement selected DevOps practices.

DevOps pipeline components are shown in figure 5.1. When the engineers push the code to the GitLab Server it triggers GitLab Runner[2]. GitLab Runner executes CI and CD pipelines which consist of jobs. Job is a GitLab term that describes an activity in CI/CD pipeline. The jobs are run using Docker[3] containers with required build environment including following tools: npm[4], Node.js[5] and Serverless Framework[6]. The images of Docker containers are stored in GitLab Container Registry[7] which is also a part of GitLab suite.

[1]https://about.gitlab.com/
[2]https://docs.gitlab.com/runner/
[3]https://www.docker.com/
[4]https://www.npmjs.com/
[5]https://nodejs.org
[6]https://serverless.com/
[7]https://docs.gitlab.com/ee/user/project/container_registry.html

Figure 5.1 shows that GitLab suite including Runner instances is deployed and works on company's premises. But the case project, having cloud-oriented architecture, is deployed on the AWS cloud infrastructure. Different environments expect different AWS accounts to avoid mix of the AWS instances such as AWS Lambdas.

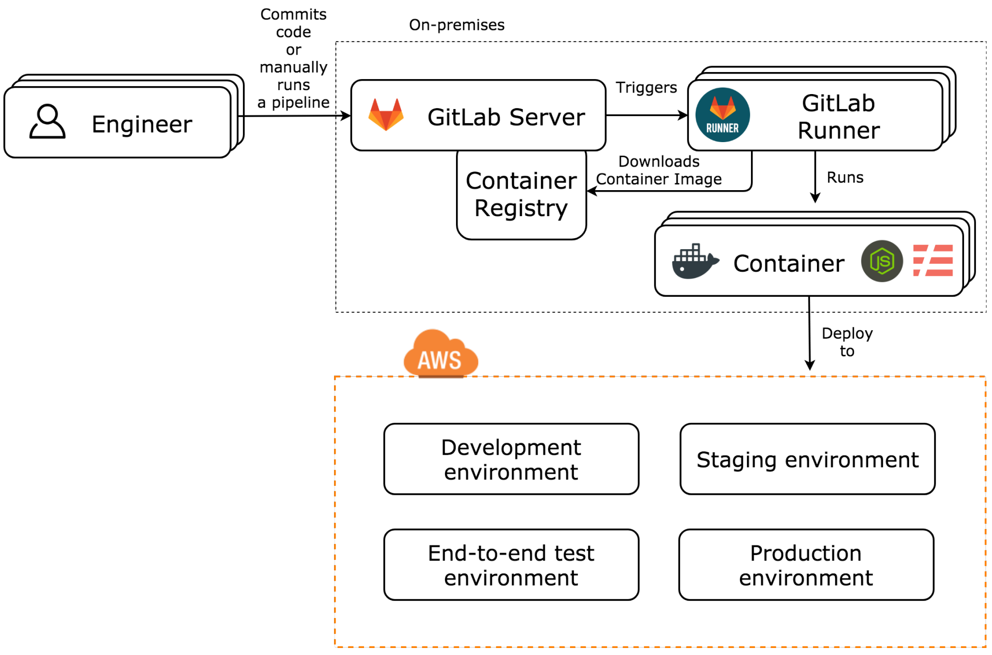

Figure 5.1: DevOps pipeline components

5.2 Infrastructure as Code solution

The most popular IaC tools for Serverless applications supporting AWS platform are Serverless Framework[8], Apex[9], SAM[10] and Terraform[11]. Their comparison is shown in the table 5.1.

Based on comparison the choice was done in favor of Serverless Framework. This tool has solid support by community and highest amount of materials and tutorials available. It also supports plugin extensions to add additional features to the built-in functionality and description of non-serverless

[8]https://serverless.com/
[9]http://apex.run/
[10]https://github.com/awslabs/serverless-application-model
[11]https://www.terraform.io/

	Serverless Framework	Apex	SAM	Terraform
GitHub stars	22,819	6,873	2,565	11,687
Contributors	394	97	33	1,207
Non-serverless resources support	Through CloudFormation	Through Terraform	Through CloudFormation (SAM is extension of CloudFormation)	Terraform is universal tool for all kinds of resources
Plugin extensions	Yes	No	No	Only for cloud providers
Other cloud providers support	Yes, through plugins	No	No	Yes, through plugins

Table 5.1: Infrastructure as Code tools comparison

resources using native AWS tool which is called AWS CloudFormation[12].

5.3 Source Control

The case project was split into three subprojects and each of them is stored in separate VCS repository. The separation was done based on the purpose of the code blocks following the best practices provided by Munns [36]. One of the practices says "unless independent Lambda functions share event sources, split them into their own code repositories". The separation into different repositories is shown in the figure 5.2.

"Api-service" groups Lambda functions that receive API calls from API Gateway to process the same domain model. "Files-processing-service" groups functions to handle event that is triggered after uploading a file to the S3 bucket with raw binary files. And "static-website" stores the files providing user interface to the application through the web application. Each of these projects has its own Serverless Framework definition file which is called "serverless.yml". It was done for independent maintenance and deployment of the subprojects.

GitLab requires the use of Git[13] as a version control system. Git branch-

[12]https://aws.amazon.com/cloudformation/
[13]https://git-scm.com/

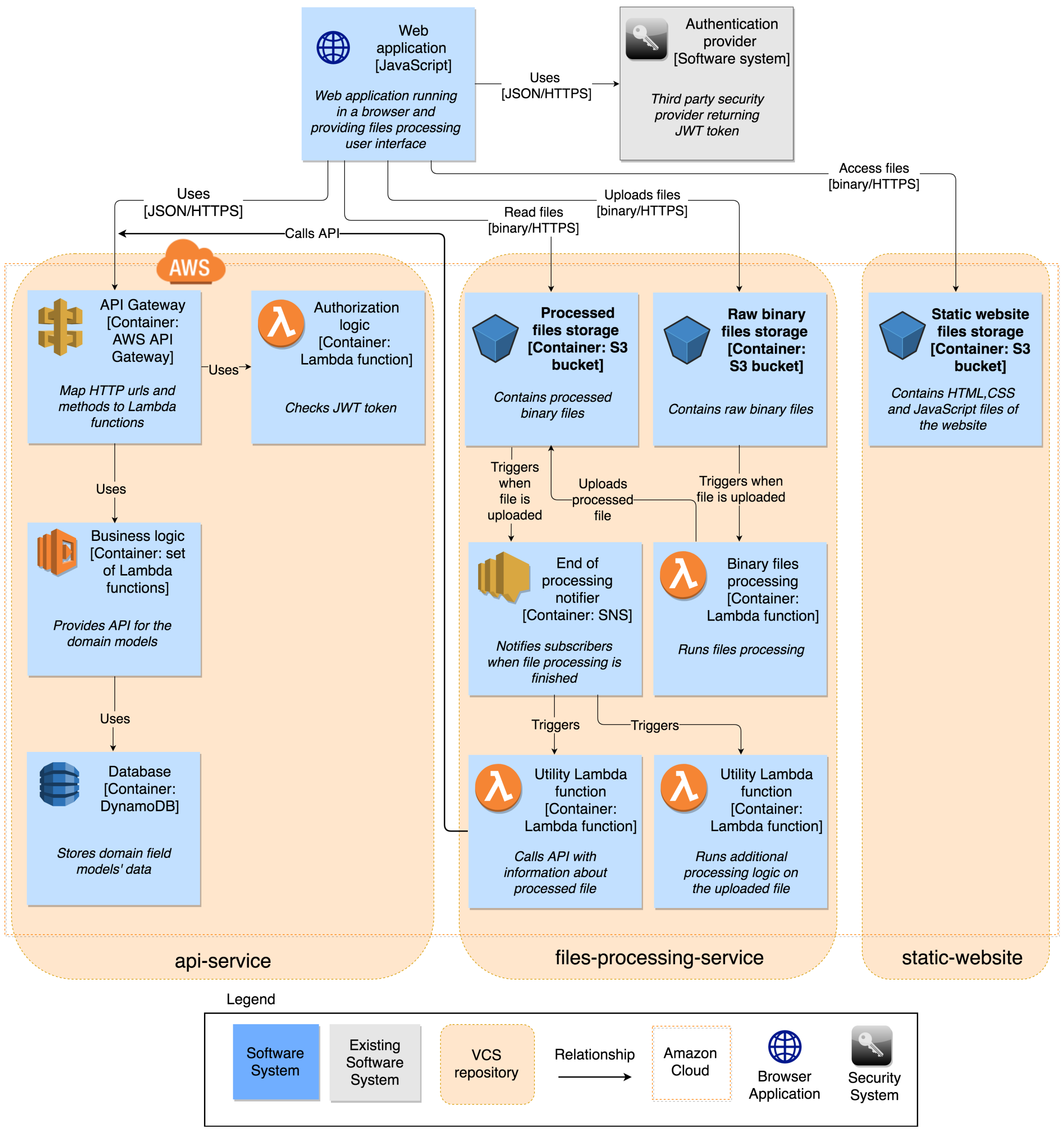
Web application [JavaScript]
Web application running in a browser and providing files processing user interface
Uses [JSON/HTTPS]
Authentication provider [Software system]
Third party security provider returning JWT token
Access files [binary/HTTPS]
Uploads files [binary/HTTPS]
Read files [binary/HTTPS]
Uses [JSON/HTTPS]
Calls API
AWS
API Gateway [Container: AWS API Gateway]
Map HTTP urls and methods to Lambda functions
Uses
Authorization logic [Container: Lambda function]
Checks JWT token
Uses
Business logic [Container: set of Lambda functions]
Provides API for the domain models
Uses
Database [Container: DynamoDB]
Stores domain field models' data
api-service
Processed files storage [Container: S3 bucket]
Contains processed binary files
Raw binary files storage [Container: S3 bucket]
Contains raw binary files
Triggers when file is uploaded
Uploads processed file
Triggers when file is uploaded
End of processing notifier [Container: SNS]
Notifies subscribers when file processing is finished
Binary files processing [Container: Lambda function]
Runs files processing
Triggers
Triggers
Utility Lambda function [Container: Lambda function]
Calls API with information about processed file
Utility Lambda function [Container: Lambda function]
Runs additional processing logic on the uploaded file
files-processing-service
Static website files storage [Container: S3 bucket]
Contains HTML,CSS and JavaScript files of the website
static-website
Legend
Software System
Existing Software System
VCS repository
Relationship
Amazon Cloud
Browser Application
Security System

Figure 5.2: Container diagram separated by services

ing model described by Driessen [16] was chosen as a development workflow. Figure 5.3 shows the illustration of the branching strategy from this article.

This model has several branches with clear purpose. Develop branch contains the latest development changes merged from the feature branches. Once all features are implemented it can be branched off into release. Release has the version of the project that is ready for internal company testing. Only bugfixes can be applied to release branch and once it is tested it can be merged into master branch. Master branch contains only production ready code. Master is a protected branch and only team members with GitLab Master role should have access to merge the changes into it. All fixes to release and master branches should be merged back into develop branch.

All commits to develop, release and master branches should be done through merge requests. Merge request feature is supported by GitLab. Code review should be done after submitting merge request and before merging. GitLab provides user interface for code reviewing of the merge requests. Code reviewing process should respect company ethic and be based on the best practices of developers' community [22].

5.4 CI and CD pipeline

Figure 5.4 shows CI and CD pipeline of the case project using Ståhl & Bosch notation.

Git workflow model suggests that developer checks out the code and creates a feature branch. The input to the pipeline is a commit to the feature branch. After local development and testing the changes from the branch are committed into the repository. It triggers the build process in GitLab which runs the code analysis and unit tests. Code analysis can be done with ESLint[14] tool. Unit tests can be written with the help of libraries such as Mocha[15], Chai[16] and aws-sdk-mock[17]. The choice of the libraries depends on the developers' preferences and can be changed. Unit tests are run only if code analysis step did not return any errors. After feature branch build has been succeeded a developer can create a merge request to the develop branch. Merge request should go through the code reviewing process. If review process reveals the issues in the code, then they should be fixed and committed to the feature branch again. Build and merge request processes should be repeated. If the code is approved for merging, then the merge re-

[14]https://eslint.org/
[15]https://mochajs.org/
[16]http://www.chaijs.com/
[17]https://github.com/dwyl/aws-sdk-mock

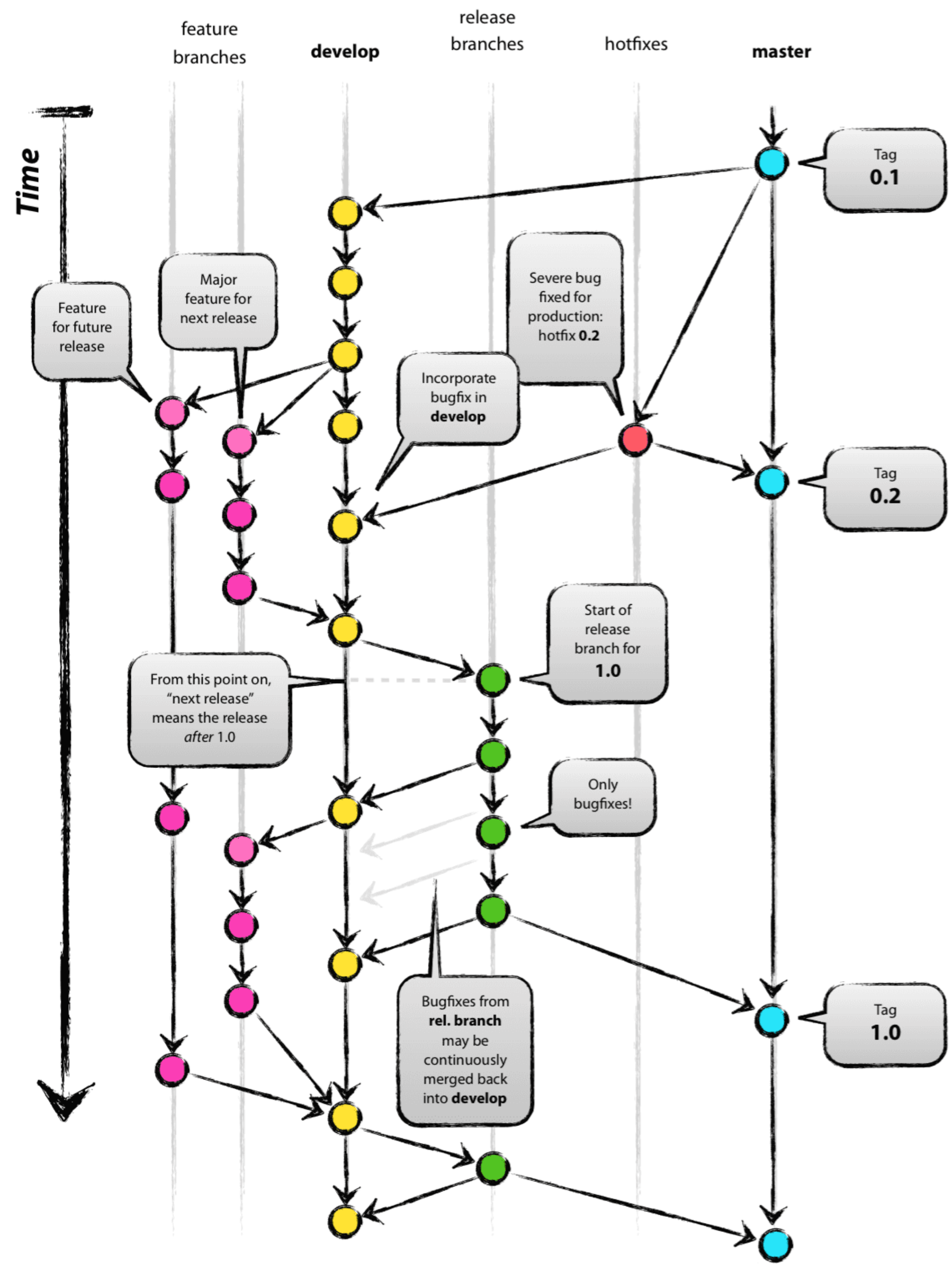
feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Figure 5.3: Git Branching Model

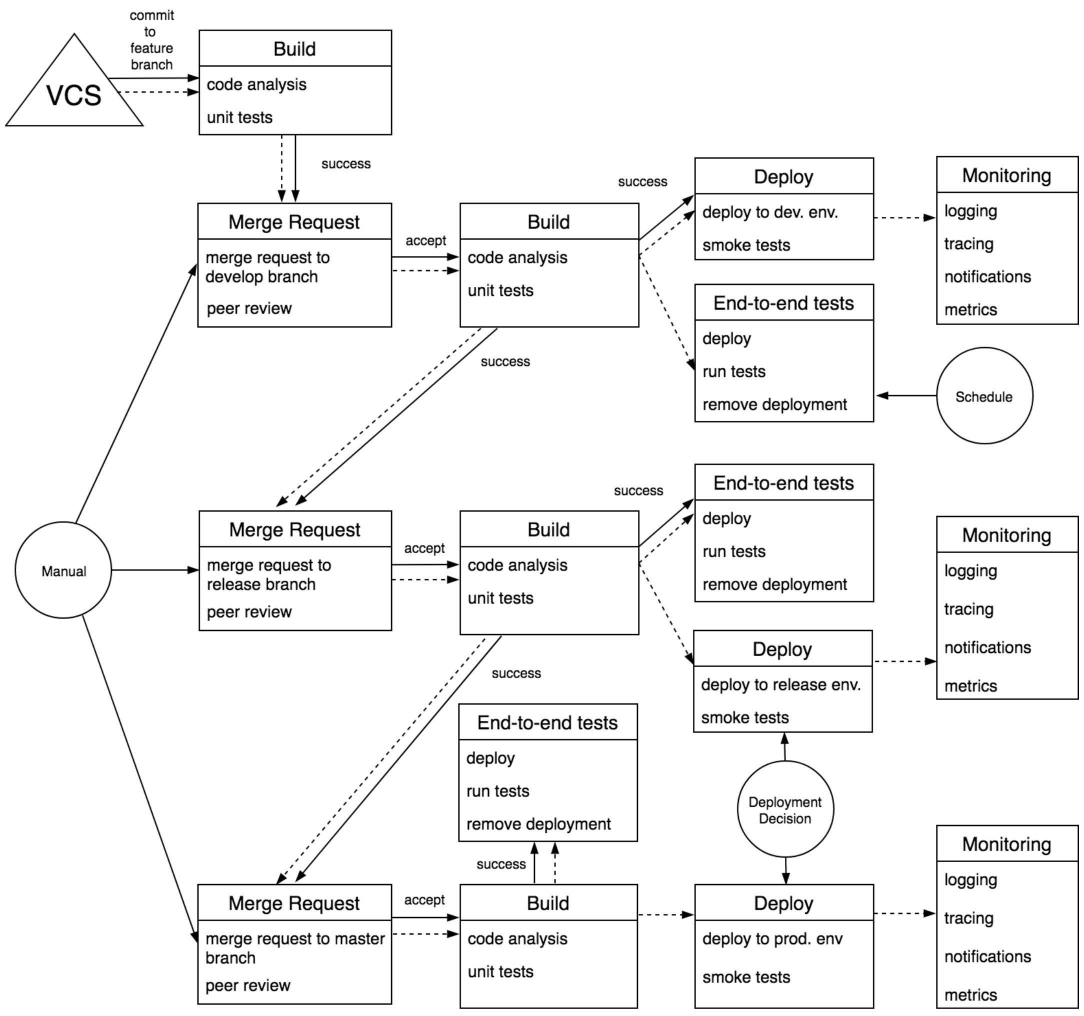

Figure 5.4: CI and CD pipeline

quest is accepted. Merge request triggers the build in develop branch. After successful build the code is deployed to the cloud platform using development account. End-to-end tests of development pipeline are run nightly to minimize the cost of pipeline execution.

The changes in develop branch can be committed to the release branch. It again runs the build but this time it triggers the end-to-end tests automatically. The reason is that commits into release branch are done less often than to the develop branch. The assumption is that develop branch will receive five merge requests per day and release branch only one merge request per two weeks.

Committing into master branch is also done through the merge request and is followed by the build process. Deploy of the release and production versions is done after deployment decision which is based on release plan made by product management. Deployed release version is available only within the company for the internal testing purposes.

Release and production deployments are triggered from a script located in additional, protected repository. In this case "protected" means that only the users with "Admin" rights have access to that repository. Deployment script triggers the pipelines of the "api-service", "files-processing-service" or "static-website" projects. The reason to create additional repository with the script to trigger pipelines of these projects is to store release and production AWS access keys in secure place and pass them through GitLab API[18]. All deployments are done with a help of Serverless Framework using *'sls deploy'* command. After deployment is done, smoke tests are executed to check that application was successfully deployed and its most important functions work. Serverless Framework configuration file also describes the monitoring infrastructure and activities that should be applied to the running application. Data received from deployed service is an input to the monitoring tools that also run in the cloud.

This pipeline is repeated for each repository of the case project: "api-service", "files-processing-service" and "static-website". It can be scaled in case if there will be more repositories with new services. Approach with atomic deployments of the services assumes that every service can follow its own release roadmap and be deployed independently.

Atomic deployment of the services requires multiple steps to deploy the whole system for end-to-end tests. Figure 5.4 shows End-to-end tests activity as a single block. Figure 5.5 shows detailed end-to-end tests preparation and execution pipeline using Ståhl & Bosch notation.

End-to-end tests pipeline can be started by schedule or automatically depending on the branch where the changes were committed. Then it deploys all three services of the project, runs the tests and undeploys the services. Deployment of the system from scratch is required to clean the data of previous tests execution such as database data and caches. Undeployment of the services right after the test execution is required to avoid spending the money on the idle services, such as databases.

End-to-end tests for Serverless applications require deployment of the services in the cloud to achieve production-like environment. This approach requires cloud infrastructure description which can be achieved by usage of specific IaC tools such as Serverless Framework and AWS CloudFormation.

End-to-end tests are stored in a separate repository. This repository

[18]https://docs.gitlab.com/ee/ci/triggers/README.html

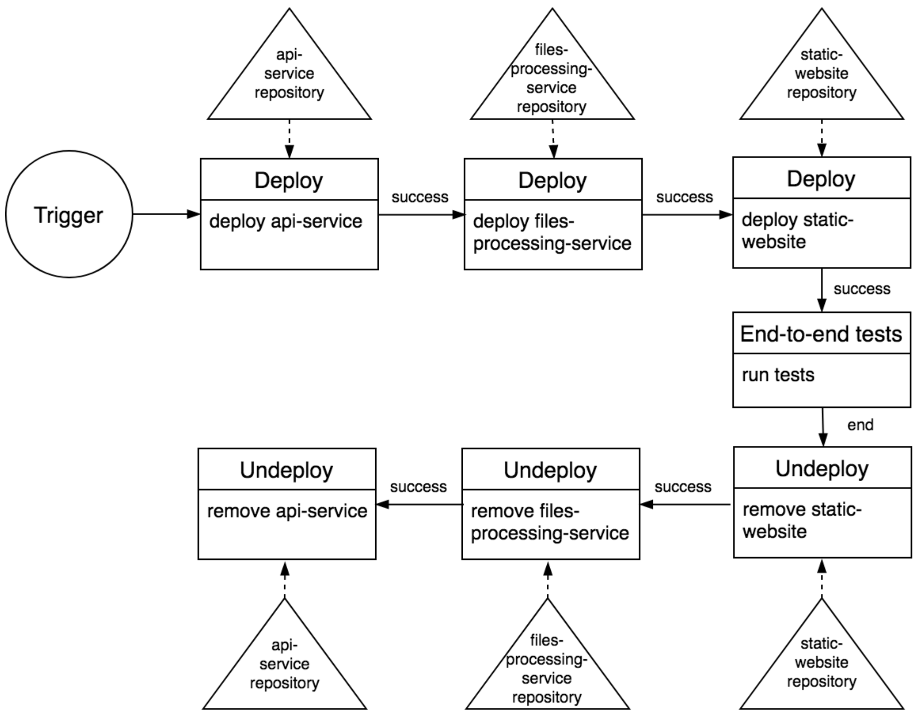

Figure 5.5: End-to-end tests execution

contains a script that triggers pipelines of other projects through GitLab API. Failure of deployment or undeployment of one of the projects might require manual relaunch of the undeploy commands to clean the infrastructure. This recovery step is not automated, because it can be considered as exceptional and requires additional investigations of why it failed.

5.5 Monitoring of the solution

Figure 5.6 shows the monitoring infrastructure of the application. It is implemented using the tools provided by AWS: CloudWatch[19] and X-Ray[20]. AWS Lambda functions write the logs to the CloudWatch Logs, where they are aggregated by function names.

CloudWatch provides Metrics feature. Metrics that were chosen for pipeline implementation are AWS Lambda functions duration, invocations, errors and throttles, S3 buckets number of objects and size, DynamoDB provisioned write and read capacity units. They can be easily changed in the future

[19]https://aws.amazon.com/cloudwatch/
[20]https://aws.amazon.com/xray/

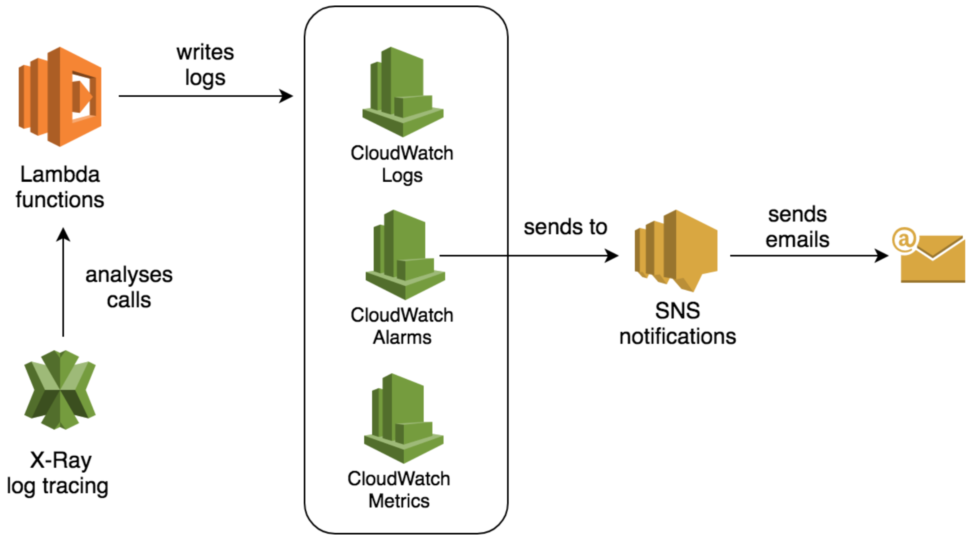

Figure 5.6: Monitoring diagram

depending on the criteria that the team finds important.

CloudWatch Alarms feature was configured to publish the messages to the SNS service. SNS service in its turn sends the alarm emails to a list of recipients. CloudWatch view graphs and statistics are used to visualise and monitor Amazon Web Services in one location. X-Ray is used to show detailed statistics about execution of the requests. It was turned on using 'Enable active tracing' option for all Lambda functions. This feature is turned on only in testing environment to find the performance bottlenecks at the early stage of development. X-Ray feature can be expensive for wide production usage and should be turned on only for the requests that have a risk of low performance.

5.6 Implementation problems

Challenges of implemented pipeline are described in chapter 6. This section provides information about the problems that were faced during its development.

Secure way of storing AWS keys

First problem was related to the storage of credentials for release and production AWS accounts. GitLab CI can use the environment variables provided in the pipeline settings of the repository. AWS user access keys can be passed to the GitLab Runner Docker container to deploy application to the cloud platform. But if they are stored in the same GitLab project where the source code and CI file are located, then they will be available for all users who have access to that repository. If company policy requires limitation of access to the production keys, then this data should be stored in another GitLab project with access allowed only to the selected users. To store release and production AWS keys privately, additional GitLab repository was created. This project triggers pipeline of the targeted repository and passes the keys through the GitLab API. Maintenance of additional project increases the complicity of the pipeline.

End-to-end test deployment

Second problem is complicated deployment of the end-to-end tests. For the testing of the whole system it is required to deploy all three subprojects of the system.

Every subproject has IaC description in its repository. End-to-end testing requires execution of these IaC files with Serverless Framework. A script to trigger this execution is stored in additional repository and it also requires resources of the team for the maintenance. Decision to store all source codes in single repository would be significant simplification for the DevOps pipeline because it would make single Serverless Framework configuration file enough. The pipeline was intentionally implemented in microservice manner to support complicated case with atomic deployments of different services and it can be easily simplified through moving all source code into one repository.

Admin GitLab users

Third problem is related to the separation of the roles in GitLab. Who can have access to the protected branches and protected repositories with the release and production keys? These decisions should be made on the management level and they depend on the culture within the company. Support of different roles of the users requires additional resources on the pipeline maintenance. Security of the project can also differ from project to project. Implemented pipeline did not have any requirements about storage of the secret keys. That is why it was decided to implement the pipeline with protected branches and private keys to show how to implement the complicated case. In case if it is required to give all developers access to the secret keys, the pipeline will be easily updated to support it.

The implementation of the pipeline was done within the planned time frame of two months. The main change that was done based on the feedback from sprint review sessions was to change one repository per Lambda function approach to repository per one service approach. After that, the project was divided into three services according to the business logic purposes: "api-service", "files-processing-service" and "static-website". Another change was related to storage of the AWS access keys. It was asked to propose an approach of how to keep them securely, giving access only to the users with special permissions.

The diagrams in the figures 5.4 and 5.5 do not explicitly show which parts of pipeline are affected by Serverless approach, but the impact of Serverless architecture was especially significant on repositories structure, unit tests and IaC tools selection, deployment and monitoring solutions. Source code was organized into Git repositories to combine multiple Lambda functions according to their purpose. Unit tests uses special libraries, such as aws-sdk-mock to mock AWS SDK calls. Serverless Framework is used to maintain Lambda functions and other cloud resources and deploy the applications to the cloud for end-to-end testing and execution in multiple stages. Monitoring pipeline is totally based on external tools that are provided by Amazon platform and are integrated with the Lambda functions.

The results of implemented pipeline evaluation made in final workshop are described in chapter 6.

Chapter 6

Evaluation

This chapters provides evaluation of implemented pipeline. The first part of it describes how implemented pipeline covers selected DevOps practices. It provides implementation details for every practice chosen in Chapter 4. The second part of this chapter provides cost calculations for execution of the DevOps pipeline. The calculation was made based on pricing model of AWS services considering some assumptions of how intensively the pipeline will be used. The third part describes the results of the final workshop where the participants discussed the implemented pipeline. This evaluation helped to get the feedback not only from the people who already worked with it and commented on it during the review sessions but also from the people who saw it first time. This part gives evaluation of the implemented practices in general, without focusing on the parts of it affected by Serverless architecture. The next chapter will summarize the findings related to the Serverless aspects of the project.

6.1 Implemented DevOps practices

Source Control

VCS is used to store code history and share the code. Git was chosen as a VCS. The GitLab Community Edition was used as a tool to store and manage Git repositories as well as organize the whole DevOps automation process. All Lambda functions are divided into groups of projects based on input source and their purpose such as API, files processing and website files.

Branches are used for isolating work. Git branching model described by Driessen [16] was chosen as a Git workflow model. This model suggests usage of branches to isolate work as well as keep master branch for production ready changes only. This model allows to separate feature branches and several types of stable releases: develop, release candidate and production.

Pre-tested merge commits. All feature branches are created in the personal developers forks and merged to the main repository through the merge requests. GitLab automatically runs CI pipeline for the merge requests including the tests.

All commits are tied to tasks. GitLab was integrated to Jira to bind the commits to the tasks. It is enough to mention task id in merge request message to make a link between them. It allows to see the list of accepted merge requests for the chosen task.

Build Process

Build process is run automatically on commit. GitLab checks every commit on the presence of *.yml file in the repository. If the file exists, then GitLab runs the CI pipeline including the project build.

Build artifacts are managed by purpose-built tools, no manual scripts. Serverless Framework is used to build and maintain the build artifacts. It uses CloudFormation templates and uploads the artifacts to the utility Amazon S3 buckets.

All artifacts have build versions with major version and commit or CI build number. Serverless Framework marks the artifacts with unique id and timestamp. It also adds the version number to the Lambda functions to make it possible to call old version of it.

Dependencies are managed in a repository. Information about dependencies is stored in the repositories together with the source code. For the JavaScript projects it is achieved by the npm and Yarn tools. Dependencies are downloaded and put into the artifact at the build stage of the CI process.

Build environment based on VMs. Build process happens in Docker containers running within Docker Runners. The image with prepared environment including installed Serverless Framework and Node.js is stored in GitLab Container Registry - private registry designed to store Docker images.

Testing & QA

Automatic unit testing with every build. As in the case with automatic build process, unit tests are run automatically with every commit to the repository when the file *.yml with a test stage is presented.

Code coverage and static code analysis is measured. The designed pipeline runs static code analysis before the unit tests and code coverage analysis. It allows to fail the pipeline in case if code violates some static code analysis rules. If this step is passed, then CI tool goes to the next step of unit tests with code coverage which usually takes more time to execute. Code coverage statistics is visible in the pipeline logs.

Peer-reviews. Designed pipeline uses peer-reviews at the moment of merge requests using GitLab code review features. Developer who created the merge request assigns a person responsible for code review. After the code review is done the code is either should be updated or merged into the main repository.

Mockups & proxies used. To emulate the behaviour of AWS services in the unit tests some libraries such as mock-aws-s3[1] and aws-sdk-mock[2] were used.

Automated end-to-end testing. End-to-end tests are run by manual launch of the "end-to-end-test" project pipeline in GitLab. Execution of this project could be easily automated but it was not done purposely to avoid extra cost in case of small changes in the repository.

Integrated management and maintenance of the test data. Test data is stored together with the unit tests and not duplicated because of the decision to store Lambda functions of the same domain field together. It allows to avoid not only test data duplications but also duplication of some Lambda functions utility code. Test results are stored in GitLab CI logs and can be found by pipeline execution date or commit number.

Automated performance & security tests in target environment. Performance and security tests were not implemented for the case project, because they require certain QA skills. But the approach of their execution is the same as for automated end-to-end tests. The pipeline requires that these tests are stored in a separate Git repository and are targeted at a project deployed in a testing AWS environment.

Deployment

Fully scripted deployments. Serverless Framework was chosen as a IaC tool. It allows to describe the Serverless functions configuration in a concise way. Additional resources such as DynamoDB and S3 databases are described with the CloudFormation templates. CI and CD pipelines are scripted as a combination of Serverless Framework, GitLab CI and Docker tools.

Auto deploy to the test environment after tests pass. This practice is achieved with the help of GitLab CI where the stages can be configured in a way that deployment stage will go only after successful execution of the unit tests stage.

[1]https://github.com/MathieuLoutre/mock-aws-s3
[2]https://github.com/dwyl/aws-sdk-mock

Standard deployments across all environments. All deployments are done on the Amazon cloud platform. The only difference between the deployments is access keys of the user accounts. Production credentials are available only for the administrators of the project. The pipeline includes four environments: "develop" with the latest changes for development purposes, "release" with the changes ready to release for the purpose of internal testing, "test" for the integration and performance tests and "production" with the stable and properly tested version of the project.

Database deployments. Cloud platform already hosts the database management systems. The IaC scripts just create the DynamoDB tables with certain parameters.

Monitoring

Log aggregation. AWS CloudWatch is used as a log aggregation tool. It is built-in AWS tool that allows to group and search the logs. The logs are aggregated into the groups by AWS service type and function name. Each group has a set of log streams. One log stream per Lambda execution. CloudWatch allows to choose logs retention policy which was chosen as one month for the case project.

Finding specific events in the past. CloudWatch Filter and Search features allow to find the specific events in the logs. Current pipeline does not introduce any additional logging and log analysis tools except native AWS tools.

Large scale graphing of the trends. Implemented pipeline uses CloudWatch Metrics for graphing of the trends. The metrics added to the dashboard are API Gateway calls count and latency, amount of 5xx and 4xx errors, Lambda functions duration, errors and invocations. Other metrics can be easily added in the future if required.

Active alerting according to the user-defined heuristics. CloudWatch Alarms tool is used for active alerting. It sends email to specified address in case of at least one Lambda execution error or API Gateway error. CloudWatch Alarms tool allows to create alarms based on any available CloudWatch metrics.

Tracing. AWS X-Ray is used to trace the requests. X-Ray feature was applied to all Lambda functions to build detailed statistics of execution. The influence of X-Ray on the cost of execution in production should be estimated very carefully. Based on this estimation the team should make decision does it make sense to turn it on in production or not.

6.2 Cost calculation

Cost of execution of implemented DevOps pipeline totally depends on how intensive are the end-to-end tests and how often the project is deployed to the cloud platform. All other proposed practices do not lead to the explicit expenses. All builds and unit tests are executed in GitLab runners on the company's premises. Expenses on the maintenance of the on-premises infrastructure are out of scope of this work. The tables 6.1 and 6.2 show the pricing of used AWS services and calculated cost per 1, 50, 250 and 1000 test set executions per service. In case if the price of AWS[3] was different for different regions, EU Ireland pricing model was chosen as a reference. 50 is expected amount of end-to-end test executions per month in case when the tests are triggered by schedule once per day for development build and several times per month for release and production builds. The number is rounded to 50 to consider expenses on smoke tests and API calls of deployed development version. 250 test executions present the case when the end-to-end tests will be run after every merge into stable branch and the frequency of such merges is seven times per day. This value corresponds to expected performance of a small team. Calculations for 1000 end-to-end test runs are provided to show how the pricing model scales for more intensive test execution.

Service	Price	Per 1 end-to-end test set execution	1 execution, $	50 executions, $	250 executions, $	1000 executions, $
CloudWatch Logs	$0.57 per GB	5 MB	0.00285	0.1425	0.7125	2.85
X-Ray	$0.000005 per trace	5000 traces	0.025	1.25	6.25	25

Table 6.1: AWS Monitoring Services cost evaluation

In addition to CloudWatch Logs and X-Ray services, the implemented monitoring solution uses CloudWatch Metrics and CloudWatch Alarms. The price of these services does not depend on amount of test executions. It depends only on amount of configured metrics and alarms. The cost of these services is $0.30 per metric per month and $0.10 per alarm per month. With 100 metrics and 100 alarms it gives the price of $40 per month.

Estimated cost of the services does not include expenses on data storage, because all data is deleted from the databases immediately after the test

[3]https://aws.amazon.com/pricing/services/

Service	Price	Per 1 end-to-end test set execution	1 execution, $	50 executions, $	250 executions, $	1000 executions, $
API Gateway	$3.50 per 1M calls	5000 calls	0.0175	0.875	4.375	17.5
Lambda requests	$0.20 per 1M requests	50000 requests	0.01	0.5	2.5	10
Lambda computing	$0.000000417 per 300ms with 256MB of memory	5000 calls	0.00208	0.104	0.52	2.08
DynamoDB read	$0.000147 per RCU per hour	250 RCU	0.03675	1.8375	9.1875	36.75
DynamoDB write	$0.000735 per WCU per hour	150 WCU	0.11025	5.5125	27.5625	110.25
S3 read	$0.0004 per 1000 requests	1000	0.0004	0.02	0.1	0.4
S3 write	$0.005 per 1000 requests	1000	0.005	0.25	1.25	5

Table 6.2: AWS services cost evaluation

execution. It makes the cost of data storage close to zero. In addition, AWS provides free tier[4] that can cover the expenses on the small projects or almost set to zero the expenses on around 20 end-to-end test executions in developed pipeline.

The estimated price of one end-to-end test set execution is $0.209835 which gives following numbers:

- $10.49175 per 50 tests executions
- $52.45875 per 250 tests executions
- $209.835 per 1000 tests executions.

[4]https://aws.amazon.com/free/

Together with expenses on metrics and alarms it gives the maximum pipeline execution price of \$249.835. On practice, expected amount of tests execution is 50 times per months which gives the price of \$50.49175 with turned on alarms and metrics and \$10.49175 without them. These numbers cannot be considered as significant and should not prevent the system from frequent end-to-end testing of the case project. Database expenses are the main part of the overall cost. Maximum cost of FaaS services running all business logic of the application is only \$12.08 even for 1000 tests executions. At the same time, it is difficult to estimate the price of the same pipeline on non-serverless architecture because it would require significant changes of the application and its execution infrastructure.

6.3 Pipeline evaluation

This section describes the results of the final review workshop. The ideas and comments are grouped according to the DevOps practices. The arrangement of this workshop is described in the section 3.3.

Source Control

"This is a very famous branching model", "This approach requires a lot of discipline"

The first concern was that proposed branching model requires more discipline comparing to simplified models with one stable master branch and many feature branches. Lack of discipline can lead to the problems with merges and versioning of the releases. One example of a possible mistake is a merge request made directly to the master branch from unstable feature branch. Even if the master branch is protected and only users with the role of "Master" can push the changes there, they still can by mistake merge the changes from unstable branch to the master branch. Complicated branching model might require more resources and Git knowledge. But it cannot be considered as a drawback, because Git usage already became a part of daily work process and spreading of the knowledge about best practices of its usage is welcomed by the team. The proposal to protect master branch from commits of developers without "Master" role should be discussed additionally. The team can switch to more flexible model where all developers have access to all branches. It requires more responsibilities but reduces the risks in case if "Admin" team members are not available.

"Why was the project split into multiple repositories? It seems like it adds a lot of complications"

The second concern was about the decision related to Serverless architecture of the project - to divide the project into multiple services and store them in separate repositories. The team raised a question - should the project be stored in a single repository. For the case project it definitely might be beneficial because maintenance of multiple repositories for such a small project might be a significant overhead. But with the growth of the project it can be divided into multiple services if separation of concerns will be required. From that perspective, implemented DevOps pipeline covers more complicated case with multiple services and can be easily adapted for single service scenario.

In general, the proposed Source Control practices were considered as good practices that should be tried within the team.

Build Process

"Who will write and support these CI configs?", "It is time to learn Docker"

The team emphasised that GitLab provides powerful tools for building process such as Docker image registry and pipeline execution. Important feedback was that now developers will be involved into DevOps process which might require additional skills such as IaC tools, Docker and GitLab CI configs. It was not considered as a bad approach, because it does not contradict to the company's culture where developers should have wide knowledge about software engineering including DevOps practices.

Testing & QA

"Now the price of tests execution is very low, but if it becomes expensive, it will reduce the motivation to run them often"

The workshop participants noted that the price of end-to-end tests execution is very low even for intensive testing. But at the same time the team still should keep the pricing model in mind and check it from time to time to avoid unexpected payments, especially in case of significant application and tests changes.

In case if test execution becomes expensive, the motivation to run the tests will decrease. Compromise might be to run intensive end-to-end tests and performance tests only when release candidate is ready. In other cases, run only unit tests and smoke tests. But with the current size of the project and AWS pricing model the probability of high cost is very low.

"The role of unit tests is now even more important"

The participants noted the importance of unit tests for Lambda functions testing. It is especially important to mock interaction with other AWS services such as DynamoDB and S3. One more suggestion was to provide an option to open the code coverage and unit test reports without opening the logs of the application. They could be stored as the artifacts of the GitLab pipeline execution.

"Too many code reviews required..."

The team also noted that extensive branching model may increase the amount of peer-reviews which might distract the developers from their current tasks.

Generally, implemented pipeline covers all main testing types but end-to-end and performance testing of the cloud application will require additional research from QA point of view.

Deployment

"Is it really the best way to store secret environment variables in GitLab?"

The main concern about deployment practices was related to the storage of production keys. GitLab Community Edition does not have enough flexibility to store secret environment variables in a convenient way. If they are stored in the repository of the project, then all developers will have access to them. If the company's policy requires only administrator access to the keys, then it is an unacceptable option. The proposed model with separate project that would keep the production keys and trigger the pipeline of targeted project was found as good enough considering available GitLab features.

The participants also asked a question about the guidelines and management of AWS accounts. Do the developers need separate AWS accounts to have their own deployment environments for personal testing or is it enough to have one development account with many users as it was suggested by the designed pipeline? It was discussed that for the first time it is enough to share the same AWS account among all developers for development environment, but have separate accounts for testing, release candidates and production deployments.

"Is Serverless Framework production ready? Who else does use it?"

The script with consequential deployment of different project services was called the most complicated and risky part of the whole deployment process. The error in the script might lead to some manual recovery. The guidelines how to do recovery in case of unexpected errors in Serverless Framework or CloudFormation should be described in more details.

Generally, deployment environments purposes are clear enough and cover the main use cases of development environment, internal company's environment for release candidate testing and production environment.

Monitoring

"Cloud providers want to hook their users... Starting using Lambda functions we will have to start using many other Amazon services"

Native AWS CloudWatch tools were found as not very user friendly. This opinion was supported by the fact that some companies use external tools for log aggregation and analysis in addition to CloudWatch. The CloudWatch concept of log streams was also noted as not obvious. AWS X-Ray tracing feature seems to be very powerful, but it is not clear at what stage it should be used: at the stage of development, testing or production? The usage of native AWS logging tools seems to be an additional risk of vendor lock-in. As a result of discussion, proposed monitoring methods were considered as a good starting point of DevOps process for Serverless application.

Summing up the results of the workshop the participants were positive about implemented DevOps pipeline. They noted that it has a focus on a high quality of the code and result product, includes extensive testing and different stages of deployment. The price of pipeline execution was noted as low. At the same time, maintenance of the pipeline requires from the developers DevOps skills and knowledge of the new tools such as Serverless Framework and GitLab CI. The highest risks are related to introduction of Serverless Framework for production deployments since it is a new tool and team has not enough experience to rely on it for critical operations. But good community support and production use cases from other companies allow to make a choice in favor of Serverless Framework tool.

This chapter presented the evaluation of the work through the description of implemented practices, estimated pipeline execution cost calculation and feedback from the stakeholders. The next chapter will be focused on the evaluation of the work against design science research guidelines and discussion of results through answering the research questions.

Chapter 7

Discussion

In the previous chapter the implemented DevOps pipeline was evaluated according to identified company's requirements. In this chapter the results of the work are evaluated against the guidelines of design science research methodology described in section 3.2. It also gives the answers to the research questions with the comparison to existing research works.

7.1 Answers to research questions

RQ1 What are the requirements for the DevOps pipeline of the case project?

The aim of this RQ was to gather the requirements for the DevOps pipeline that could be implemented for the case project. The focus had to be on the Automation practices of DevOps considering Serverless architecture of the case project. The requirements elicitation was done in a form of discussion of selected DevOps practices. The practices were ordered by priorities based on the workshop within the team. It was done to choose the most important of them for implementation. The results of practices selection for DevOps pipeline are presented in the table 4.1. The key findings of requirements elicitation are:

- Case project development has some requirements about the development process and tools. Issue tracker should be Jira, CI tool suite and repository management tool - GitLab. The size of the team is up to seven developers and QA engineers. Development process is Scrum with two-week sprints. Architecture of the case project is already designed and it relies on Serverless approach. The case project uses Amazon Web Services such as API Gateway, Lambda, DynamoDB, S3 and

SNS. There is no initial cost limitation for the pipeline execution.

- The case project requires high quality of the code. The team found important 25 out of 27 suggested DevOps automation practices for the case project. These practices are related to source control, build process, deployment, testing & QA, deployment and monitoring. The most critical DevOps automation practices are usage of VCS and log aggregation. The team found "release notes auto-generation" as the less important practice among suggested. "Version control DB schema changes" was noted as an important practice for relational databases but in context of the case project and its schemaless database this practice was marked as the least important. Rationale of practices selection for proposal was done based on the literature review and is presented in Section 2.2.

- The time limit for DevOps pipeline implementation was two months or four sprints. The implemented DevOps pipeline was considered as a proof-of-concept and the decision which parts of it to use in practice has to be done after the final review workshop.

DevOps pipeline requirements correspond to findings in existing studies about implementation and adoption of DevOps automation practices [39][48]. Serverless architecture of the project did not reveal any unusual expectations from CI, CD and Monitoring practices even if it was clear that it affects their implementation.

RQ2 How does Serverless architecture affect DevOps practices such as CI, CD and Monitoring of the application?

The aim of the second RQ was to identify how do elements of Serverless architecture affect chosen DevOps practices and understand what the expectations of developers for the Serverless applications are. The answer to this question was found through the workshop within the team working on the case project and multiple interviews with the developers outside of the case company.

The results of the workshop are presented in table 4.1 and they show that Serverless architecture significantly affects DevOps pipeline practices. 18 out of 27 practices were marked as affected. There was no group of DevOps automation practices that would not be influenced by Serverless. The most significant influence Serverless makes on the monitoring practices. It is explained by close connection of Serverless services with the cloud platform and the fact that monitoring services are also provided by cloud platforms.

The least impact Serverless makes on the source control practices. But at the same time, it influences the important decision of how to organize the code into the VCS repositories. Serverless functions can be placed each one into separate repository or several functions into one service according to their purpose or domain field. This decision can influence all other decisions in DevOps pipeline.

Interviews with the stakeholders helped to understand why the team chose Serverless architecture and what the concerns about it are. The best parts of Serverless architecture were identified as out of the box scalability, cheaper cost because of no payments for idle services and faster development because of lack of server configurations. The concerns were related to lack of experience and skills required for Serverless applications development which might lead to the problems during the implementation, more complicated debugging because production environment cannot be reproduced on the local premises and, finally, the possible vendor lock-in. Vendor lock-in was called a justified risk because vendor provides many tools that will be used in the project to utilise all power of Serverless features even if they are not compatible with other platforms. These benefits and risks of Serverless applications are not directly related to the DevOps pipeline implementation except the concern about lack of skills and knowledge about this new architecture type that might be required for CI, CD and Monitoring practices as well. The results of the interviews correspond to the results of existing research [5].

The implementation of the pipeline showed that Serverless architecture strongly affects the choice of the tools, for instance, IaC solutions and mocking libraries. It is also explained by close connection of Serverless applications with the cloud infrastructures that usually require deployment of the application to the cloud platform and heavy usage of the cloud platform services, in case of developing project - Amazon Web Services. The existence of Serverless-oriented tools such as Serverless Framework shows that developers and DevOps engineers realised high demand of specialized tools to maintain Serverless applications.

Lack of research about interconnection of Serverless architecture and DevOps practices makes it difficult to compare the results of current work with previous studies. But all achieved results match the observations made by engineers in software development industry. DevOps pipeline for Serverless applications makes local deployment environment unnecessary but requires from developers the knowledge about operations [26]. At the same time the need for extensive automated testing and CI/CD does not disappear and the risk of vendor lock-in is getting higher [9].

RQ3 How well does implemented DevOps pipeline fulfil company requirements?

The implemented DevOps pipeline fulfils all requirements identified in RQ1. It is based on the technologies and tools listed in the requirements such as Amazon Web Services and GitLab. The pipeline includes all practices selected in the workshop about DevOps automation of the case project and ensures high quality of the code through unit tests, code quality check tools and end-to-end testing [44]. The implementation of each practice is described in Section 6.1. In addition, the implemented pipeline deploys the project into multiple different environments: development, staging, end-to-end tests and production. It allows to separate stable production version of the project from release candidate and version which is under active development.

The research showed that stakeholders positive expectations for Serverless applications are mostly related to scalable execution model without server configuration. Some of the concerns about Serverless applications are directly related to the DevOps such as usage of IaC and lack of experience with Serverless approach. These problems are solved by the usage of Serverless Framework - the tool that is in the core of implemented pipeline. Even if it is a new tool for the team, it allows to reduce the risk of unexpected problems and lack of knowledge about Serverless applications development. The reason is that Serverless Framework use cases are based on the best practices of Serverless applications development and deployment made by software engineers' community.

There were no initial requirements about the cost of pipeline execution, but estimated price was called as very low. Expected expenses on the end-to-end testing are just \$50.49175 with turned on alarms and metrics and \$10.49175 without them. Even with 20 times more intensive testing the price will be \$249.835. The cost of only Lambda functions execution is only \$12.08 for 1000 tests executions. It confirms affordable pricing model of Serverless computing [2]. The price of production and release execution was not calculated because it is not related to DevOps pipeline and mostly depends on popularity of the project and how intensively it will be used.

7.2 Design Science Research

This section examines the results of research work according to the evaluation guidelines provided by design science research described by Hevner and Chatterjee [28].

The objective of design science research in information systems is to de-

velop technology-based solutions to important and relevant business problems. Implemented DevOps pipeline is a technology-based solution aimed at solving business problems identified as the required for the case project.

The designed artifact must be effectively represented, enabling implementation and application in an appropriate environment. The implemented artifact is a DevOps pipeline including CI, CD and Monitoring practices. It is represented in Chapter 5 and can be applied for the projects with similar requirements.

The quality and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. Produced artifact was studied for the compatibility with business case environment and for the fit into architecture of the case project. The studies were done in the form of review workshop and are presented in the Section 6.3

Effective design science research must provide clear contributions in the areas of the design artifact, design construction knowledge, and/or design evaluation knowledge. The results of the research contribute to the area of design and implementation of the DevOps pipelines because they cover the area that is not well represented in the literature - DevOps pipelines for Serverless applications.

Threats to validity

The designed pipeline presented in this thesis is valid only in the context of the considered use case. Implemented DevOps practices might not be applicable to every Serverless application. The factors that might affect the DevOps decisions include size of the project and its architecture, size of the team and its experience, culture within the company and requirements about the tools and processes used in the team. At the same time, interviews with the software engineers outside of the case company helped to build unbiased picture about influence of Serverless architecture to DevOps practices. It allows to use observations about connections of Serverless and DevOps as an input for designing CI, CD and Monitoring for other projects, but not to take implemented pipeline as it is.

7.3 Future research

Literature review showed that Serverless computing is not covered enough by existing research works. DevOps for Serverless applications is only one possible research direction for this new technology trend. This thesis showed high demand in knowledge and practices related to Serverless applications. Possible topics for the future research are:

- *Testing of the Serverless Applications.* Current research showed that end-to-end, performance and security testing require rethinking for Serverless applications because of proprietary execution infrastructure and pricing model that applies some limitations on the frequency of test execution.

- *Security of the Serverless Applications.* Third party Serverless execution infrastructure raises the question of how secure is the Lambda functions environment and what risks does it apply to the applications.

- *How does Serverless approach affect DevOps culture.* This thesis has a focus on Automation part of DevOps, but Serverless approach is marketed as a game-changing approach that might affect other aspects of DevOps such as Culture and Lean because it allows to rapidly develop and release the software. Combination of Serverless and DevOps culture requires additional research.

This chapter evaluated the work according to the design science research guidelines. It also gave the answers to the research questions. The next chapter draws the conclusions of the study.

Chapter 8

Conclusions

The aim of this thesis was to implement DevOps pipeline for Serverless application in order to automate building, quality assurance, deployment and monitoring processes and ensure high quality of the case project development. The goal was achieved using design science research methodology through series of workshops and technical implementation of the required pipeline. Implementation was done in four iterations of Scrum process started with requirements elicitation workshop and finished with the final result evaluation workshop. In addition, three interviews with the engineers outside of the case company were conducted. The topic of the interviews was connection between Serverless architecture and DevOps practices. This research process together with literature review allowed to answer three research questions related to requirements for the DevOps pipeline, influence of Serverless architecture on CI, CD and Monitoring practices and success of implemented pipeline.

The artifact of this work, DevOps pipeline for the case project, implements chosen by the development team practices of source control, building, deployment, testing & QA and monitoring of the application. The result pipeline allows to automatically run unit tests, do code quality check, run end-to-end tests and deploy project to multiple environments in Amazon cloud platform as it was required by architecture of application. GitLab Community Edition was used as a core of CI/CD solution. The final evaluation workshop showed that all elicited requirements for the pipeline were fulfilled.

The research did not reveal any DevOps automation or monitoring practices that would be rejected because of Serverless elements of the case application. At the same time, 18 out of 27 practices were noted as affected by Serverless. The qualitative data showed that some of the practices are strongly affected by this new cloud computing execution model, for instance, standard deployment to all environments, use of mockups in unit testing and all monitoring practices. The existence of these challenges is proved by high popularity of serverless-oriented tools that are aimed to simplify debugging, deployment to multiple environments and declarative definition of the infrastructure for the Serverless applications.

The results of the study do not contradict existing research about Serverless computing, such as expected benefits and concerns about Serverless applications. As in many other sources the developers called out of the box scalability, cheap pricing model and simpler environment configuration as the benefits of Serverless computing. The concerns were related to vendor lock-in and complicated debugging. Interviewees also agreed that Serverless architecture requires from engineers' knowledge of both development and operations practices and might lead to popularity of a new role of Cloud Engineer that would combine these skills.

The results about impact of Serverless computing on DevOps practices contributes to the research in this field. Despite the large amount of non-academic materials, this topic is not well presented in research papers. At the same time, obtained results cannot be generalized because they were gathered for single case project with specific requirements. But elements common for all Serverless architectures allow to consider these results as useful for other project cases.

Future research in this field might include deeper studies about impact of Serverless architectures on testing & QA practices, such as performance and security testing. Security of Serverless applications can be also an additional research direction since vendor host environment applies some privacy control limitations on execution of the system.

This work was focused on technical practices of DevOps such as CI, CD and Monitoring. But as literature review showed, the term DevOps describes the overall culture in the company. Relation between Serverless-oriented development, that promises faster software release lifecycle with less resources, and Culture within the company might be one more important topic of research because in case of bad culture and lack of knowledge sharing activities even the most effective DevOps automation pipeline will not help a company to succeed.

Bibliography

[1] Gartner IT Glossary. DevOps. `https://www.gartner.com/it-glossary/devops`. Accessed 6.5.2018.

[2] Adzic, G., and Chatley, R. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 884–889.

[3] Ajluni, C. Plotting a new devops path with serverless computing, 2017. `https://www.stratoscale.com/blog/devops/plotting-new-devops-path-serverless-computing/`. Accessed 17.4.2018.

[4] Balalaie, A., Heydarnoori, A., and Jamshidi, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software 33*, 3 (2016), 42–52.

[5] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[6] Bass, L., Weber, I., and Zhu, L. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[7] Becker, T. A production-grade ci/cd pipeline for serverless applications, 2018. https://medium.com/@tarekbecker/a-production-grade-ci-cd-pipeline-for-serverless-applications-888668bcfe04. Accessed 6.5.2018.

[8] Brown, S. *Software Architecture for Developers - Volume 2. Visualise, document and explore your software architecture*. Leanpub, 2018.

[9] Buckholz, G. The pros and cons of a serverless devops solution, 2018. `https://www.agileconnection.com/article/pros-and-cons-serverless-devops-solution`. Accessed 17.4.2018.

[10] Chapin, J., and Roberts, M. *What is Serverless.* O'Reilly Media, Inc., 2017.

[11] Claburn, T. From devops to no-ops: El reg chats serverless computing with nyt's cto, 2017. `https://www.theregister.co.uk/2017/11/30/from_devops_to_noops_nyt_cto/`. Accessed 17.4.2018.

[12] Crane, M., and Lin, J. An exploration of serverless architectures for information retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval* (2017), ACM, pp. 241–244.

[13] Cukier, D. Devops patterns to scale web applications using cloud services. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity* (2013), ACM, pp. 143–152.

[14] Debois, P. Devops: A software revolution in the making. *Journal of Information Technology Management 24*, 8 (2011), 3–39.

[15] Dingsøyr, T., and Lassenius, C. Emerging themes in agile software development: Introduction to the special section on continuous value delivery. *Information and Software Technology 77* (2016), 56–60.

[16] Driessen, V. A successful git branching model, 2010. `http://nvie.com/posts/a-successful-git-branching-model/`. Accessed 17.4.2018.

[17] Dyck, A., Penners, R., and Lichter, H. Towards definitions for release engineering and devops. In *Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop on* (2015), IEEE, pp. 3–3.

[18] Elberzhager, F., Arif, T., Naab, M., Süss, I., and Koban, S. From agile development to devops: going towards faster releases at high quality - experiences from an industrial context. In *International Conference on Software Quality* (2017), Springer, pp. 33–44.

[19] Erich, F., Amrit, C., and Daneva, M. Report: Devops literature review. *University of Twente, Tech. Rep* (2014).

[20] Erich, F., Amrit, C., and Daneva, M. A qualitative study of devops usage in practice. *Journal of Software: Evolution and Process 29*, 6 (2017).

[21] Farid, A. B., Helmy, Y. M., and Bahloul, M. M. Enhancing lean software development by using devops practices. *International Journal OF Advanced Computer Science and Applications 8*, 7 (2017), 267–277.

[22] Fink, R. Code review best practices, 2018. `https://medium.com/@palantir/code-review-best-practices-19e02780015f`. Accessed 17.4.2018.

[23] Fitzgerald, B., and Stol, K.-J. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering* (2014), ACM, pp. 1–9.

[24] Fowler, M. Continuous integration, 2006. `https://martinfowler.com/articles/continuousIntegration.html`. Accessed 6.5.2018.

[25] Fox, G. C., Ishakian, V., Muthusamy, V., and Slominski, A. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv:1708.08028* (2017).

[26] Gancarz, R. Serverless takes devops to the next level, 2017. `https://www.infoq.com/articles/serverless-takes-devops-next-level`. Accessed 17.4.2018.

[27] Ganguly, R. Automating ci/cd workflow for serverless apps with circleci, 2017. `https://serverless.com/blog/ci-cd-workflow-serverless-apps-with-circleci/`. Accessed 6.5.2018.

[28] Hevner, A., and Chatterjee, S. Design science research in information systems. In *Design research in information systems*. Springer, 2010, pp. 9–22.

[29] Humble, J., and Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[30] Kim, G. Top 11 things you need to know about devops, 2013. `https://www.thinkhdi.com/~/media/HDICorp/Files/White-Papers/whtppr-1112-devops-kim.pdf`. Accessed 17.4.2018.

[31] Kim, G., Debois, P., Willis, J., and Humble, J. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* IT Revolution, 2016.

[32] Laukkanen, E. *Adoption problems of modern release engineering practices.* PhD thesis, Department of Computer Science, Aalto University School of Science, Espoo, Finland, 2017.

[33] Lwakatare, L. E., Kuvaja, P., and Oivo, M. Dimensions of devops. In *International Conference on Agile Software Development* (2015), Springer, pp. 212–217.

[34] Lwakatare, L. E., Kuvaja, P., and Oivo, M. An exploratory study of devops extending the dimensions of devops with practices. *ICSEA 2016* (2016), 104.

[35] Montiel, I. How we migrated our startup to serverless, 2017. `https://read.acloud.guru/our-serverless-journey-part-2-908d76d03716`. Accessed 6.5.2018.

[36] Munns, C. Serverless architecture patterns and best practices, 2017. `https://www.youtube.com/watch?v=_mB1JVlhScs`. Accessed 6.5.2018.

[37] Noller, A. Continuous delivery: Maturity checklist, 2014. `https://dzone.com/articles/continuous-delivery-maturity`. Accessed 17.4.2018.

[38] Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. A design science research methodology for information systems research. *Journal of management information systems 24*, 3 (2007), 45–77.

[39] Rejström, K. Implementing continuous integration in a small company: A case study. Master's thesis, Aalto University, School of Science, Espoo, Finland, 2016.

[40] Riungu-Kalliosaari, L., Mäkinen, S., Lwakatare, L. E., Tiihonen, J., and Männistö, T. Devops adoption benefits and challenges in practice: a case study. In *International Conference on Product-Focused Software Process Improvement* (2016), Springer, pp. 590–597.

[41] Roberts, M. Serverless architectures, 2016. `https://martinfowler.com/articles/serverless.html`. Accessed 17.4.2018.

[42] Rodgers, J. Serverless hosting comparison, 2017. `https://headmelted.com/serverless-showdown-4a771ca561d2`. Accessed 6.5.2018.

[43] Sbarski, P., and Kroonenburg, S. *Serverless Architectures on AWS: With examples using AWS Lambda.* Manning Publications Company, 2017.

[44] Ståhl, D., and Bosch, J. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th IASTED International Conference on Software Engineering, Innsbruck, Austria, 2013* (2013), pp. 736–743.

[45] Ståhl, D., and Bosch, J. Automated software integration flows in industry: A multiple-case study. In *Companion Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 54–63.

[46] Ståhl, D., and Bosch, J. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software 87* (2014), 48–59.

[47] Tran, T. H. Developing web services with serverless architecture. Master's thesis, Lappeenranta University of Technology, School of Business and Management, 2017.

[48] Udd, R. Adopting continuous delivery: A case study. Master's thesis, Aalto University, School of Science, Espoo, Finland, 2016.

[49] Varghese, B., and Buyya, R. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems 79* (2018), 849–861.

[50] Verona, J. *Practical DevOps.* Packt Publishing, 2016.

[51] Wells, D. Serverless technology use cases, 2017. `https://www.youtube.com/watch?v=9vBN1Jx_h_o`. Accessed 6.5.2018.

[52] Zambrano, B. *Serverless Design Patterns and Best Practices.* Packt Publishing, 2018.

[53] Zanon, D. *Building Serverless Web Applications.* Packt Publishing, 2017.