Antti Karkimo

Monolithic Web Application Infrastructure Upgrade Considerations

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

15 May 2023

PREFACE

Thank you for deciding to take a look at my thesis. The topic might not be groundbreaking, but it was important for the company, and I find it exciting. Especially the part where we discover that you can use the end product to rewrite the end product itself. I hope you, the reader, also find it interesting.

If you have similar kind of traditional web applications, please do really think about implementing some of the things discussed in this thesis. It would be easy to dismiss aspects such as microservices or ChatOps as fads that will come and go, and not spare them a second thought. The important part is to think every part through in relation to what you currently do, even if the decision is to not implement them. You could come up with your own better ideas for your team or company.

Espoo, 15th May 2023
Antti Karkimo

Abstract

Author:	Antti Karkimo
Title:	Monolithic Web Application Infrastructure Upgrade Considerations
Number of Pages:	40 pages
Date:	15 May 2023
Degree:	Master of Engineering
Degree Programme:	Information Technology
Professional Major:	Networking and Services
Supervisors:	Ville Jääskeläinen, Principal Lecturer

A small Finnish software development company had problems with the infrastructure and development process of their ERP web application. The problems have caused delays in publishing new versions of the application, inefficient use of employee resources and wasted computing resources. The state of the processes and techniques used by the company were analysed and compared to the current best practices. Studies on the impact of the problems and usage of the best practices were explored.

A set of recommendations was created to rectify the situation. The set of recommendations includes processes and tools for the company to implement. It also includes a plan on how to manage the transformation work and suggested implementation path to take.

Keywords: web applications, monolithic applications, microservices, DevOps, Kubernetes, autoscaling

Contents

List of Abbreviations

1 Introduction 1

1.1 Changes in the Web Infrastructure and DevOps Fields 1

1.2 System X Servers 4

1.3 Software Development Process 5

1.4 Objective of Thesis 5

2 Current State Analysis 7

2.1 Description of the Current System 7

2.2 Infrastructure Issues 8

2.2.1 Infrastructure Issues at a Detail 9

2.2.2 Impact of the Infrastructure Issues 10

2.3 DevOps issues 12

2.3.1 DevOps Issues at a Detail 12

2.3.2 Impact of the Devops Issues 13

3 Theoretical Background and Best Practices 16

3.1 A Review of Possible Solutions 16

3.1.1 Infrastructure Solutions 16

3.1.2 DevOps Solutions 20

3.1.3 Interoperability Between Solutions 24

3.2 Academic studies 25

3.2.1 Studies on Autoscaling Infrastructure Solutions 25

3.2.2 Studies on DevOps 26

4 Set of Recommendations 28

4.1 General Overview of the New System 28

4.2 Processes and Tools 29

4.2.1 Continuous Integration (CI) 30

4.2.2 Continuous Deployment (CD) 31

4.2.3 Microservices 32
4.2.4 Infrastructure as Code 32
4.2.5 Monitoring and Logging 33
4.2.6 Communication and Collaboration 33
4.3 Infrastructure Specifications 34
4.4 Implementation Path 36
5 Conclusion 39
References 41

List of Abbreviations

AKS	Azure Kubernetes Service
API	Application Programming Interface
AWS	Amazon Web Services
CGI	Common Gateway Interface
CI	Continuous Integration
DDoS	Distributed Denial of Service
DNS	Domain Name Service
DORA	DevOps Research and Assessment
EC2	Elastic Compute Cloud
EKS	Amazon Elastic Kubernetes Service
ERP	Enterprise Resource Planning
GKE	Google Kubernetes Engine
IaaS	Infrastructure as a Service
IP	Internet Protocol
JVM	Java Virtual Machine
JSON	Javascript Object Notation
SaaS	Software as a Service
SDN	Software Defined Network

1 Introduction

This study was made for a small Finnish software development company. In addition to customer projects and consulting, the company has some SaaS products of its own. This thesis concentrates on one of these software products, a web application is called System X for the sake of this thesis. System X has been in development for over ten years. During the product lifecycle, the infrastructure supporting System X has been upgraded and updated incrementally, but it has still accumulated technical debt. Some of this technical debt comes from actual software solution implementations that the system has overgrown with the number of users steadily climbing. Other parts are challenges in the DevOps processes that support System X in its day-to-day operations and also the release and installation of newer versions.

1.1 Changes in the Web Infrastructure and DevOps Fields

The foundations of the interactive web were built in the 1990's on techniques such as Common Gateway Interface (CGI) and a collection of scripts contained therein. After reaching a certain level of maturity and cohesion, the traditional web application has been run with either a language specific application server or a generic web server and extensions.

Some of the original popular application servers included Microsoft's Internet Information Services (IIS) serving Active Server Pages (ASP) and Sun Microsystems Servlet or later Apache Tomcat serving JavaServer Pages (JSP), nowadays known as Jakarta Server Pages (JSP). The most popular way to use a web server for interactive web content for a long time was the Apache web server with a plugin module of the user's choice, for example mod_perl, mod_php or mod_python and an interpreter of the same language.

The previous decade in the web application world has been one of constant evolution. Coming to the 2010's saw a rapid rise in microservice architectures as the state of the art [1].

In a typical monolithic service the single server software handles all tasks including database connections, user session management and authorization, serving static content, generating dynamic content, interpreting programs or fetching previously compiled programs and combining all of these to be delivered to the user.

In a microservice architecture any and all previously mentioned parts can be delegated to programs specific to each task. The services communicate with each other through Application Programming Interfaces (APIs), and don't necessarily have to have any relation to one another. They can be in different programming languages, geospatially separated, from different vendors and in essence completely black boxes as far as the other parts are considered. [2]

Often mentioned as closely related, but not always necessitated by eachother with a microservice architecture is the concept of DevOps. DevOps combines the processes of software development with the processes of IT operations. In the traditional sense the responsibility of the software development side starts and ends with the planning and programming of the software in question. IT operations is responsible for managing the servers, maintaining them and installing new versions of the software that the development team provides to them.

In a modern DevOps process both sides are integrated into a single seamless workflow. The idea of how the software deliverable will be deployed and what kind of infrastructure it will run on in the production environment is take into consideration from the very beginning of the software development lifecycle. New versions of the software, when deemed complete, can be deployed even completely automatically in the most extreme of cases. The term continuous delivery encapsulates the routine quite well. Continuous delivery is contrasted to

continuous integration (CI). Continuous integration is a software development methodology in which a CI server compiles and tests new versions of the software immediately when they are available in the version control repository. CI has been a staple of the modern software development toolkit and can be used by itself without necessarily having to take measures to transform into a modern DevOps workflow. A well working CI system however lends itself into a smoother transition towards a modern DevOps oriented way of developing software.

The adoption of a functional DevOps system requires some software. Such software often aims to replicate the environment of the final deployment already on the developers' workstations. In effect the developers will experience the software running on their workstations exactly as it would behave for the users in the actual production environment. The workstation and the server can have a completely different operating system in addition to any number of other parameters. Therefore, some sort of abstraction layer is often used on both computers to standardize the environment. Most notable development in this field have been the various containerization technologies. Whether on the developer's workstation or the server, the operating environment is packaged inside a container image. When the developer starts to work on the project, they will start up the container and have the environment set up exactly as it would be on the server. They can work with the source code and see the changes happen on their own version of the server, exactly as it would be on the production server.

The containers also work to isolate the server infrastructure choices from affecting how and where the software is run. The infrastructure can be upgraded, changed, or moved to a completely different physical site. As long as the resulting configuration can run the selected containerization platform, the software container should work as well as anywhere else. This means that somebody still has to be responsible for the choice of the underlying technology, but the choice can be relatively arbitrary from the software development point of view. No longer will specific versions of the software runtimes such as the PHP interpreter or the Java Virtual Machine (JVM) have to be taken into consideration, as these can and should be packaged in the container image of the software.

1.2 System X Servers

System X's backend software is written in PHP. The database is stored in a MariaDB database system. Both of these server applications are running on virtualized Linux installations. These virtualized Linux servers don't allow any kind of automatic scalability of resources. The only kind of scalability available is vertical, in that the virtual server instances can be started up with more resources if they become seriously overloaded. Resources that can be allocated would be for example processing power by the way of adding more Central Processing Unit (CPU) cores or memory. This scaling can only be evoked manually, and System X would experience a slight moment of unavailability while the scaling takes place. In practice this means that the virtualized servers must permanently have resources allocated dictated by the usage and load of the worst-case scenario.

A much better and modern approach is to scale computing resources automatically and horizontally. The process is called autoscaling. In autoscaling, the system has predefined limits on performance or resource usage, after which being met make the system start allocating more resources for itself. When done horizontally instead of vertically, it means that instead of adding resources such as memory to the existing virtual server instance, the system can add identical or similar entirely new server containers into the pool. The server containers start from a suspended but otherwise instantly ready-to-serve state. The server containers can be very small in their throughput and resources by themselves.

During times of very light and sporadic usage, like in the middle of the night, the system could be running on a single node or possibly even without any nodes at all, quickly provisioning one the instant a user tries to access the system. This kind of flexibility doesn't come for free, since there needs to be a way to divide the load and supervise the nodes that currently doesn't exist for System X. The data also needs to be distributed between the nodes in real time, currently handled by the MariaDB software but possibly becoming more complex with the addition of the autoscaling.

1.3 Software Development Process

The other major issue which has arisen from the organic growth experienced by System X and its developers is the archaic way new versions of the software are brought to the users. The process is well understood and mostly straightforward, but it includes a lot of manual steps and other difficulties. The actual development work has been kept up to date with good practices, but after the version meant for deployment has been finished and marked as such in version control, the time-consuming operations begin.

Getting a new version of System X online requires moving files to the correct places, running scripts to update certain fields and generally cannot be done without the database being locked to a read-only state. Some small bugfixes can be updated live without the user noticing anything, but in most cases the system cannot be accessed while the upgrade is taking place. In essence that means that there has to be a maintenance break, which has to be scheduled in advance and notified to the users. The maintenance breaks have to be scheduled to times of low usage in order to cause the least disruptions. Combined with the manual steps of the software upgrade itself, it means that somebody has to be at work doing those steps at very inconvenient working hours.

1.4 Objective of Thesis

The objective of this thesis is to provide a Plan of Recommended Action to be taken by the target company in order to solve the issues manifesting in System X. The solution to this will be a dedicated DevOps automation and orchestration system, together with a container software selected for the autoscaling system in the infrastructure problems section. This is the reason why these two will be investigated at the same time.

This thesis will consist of more precise presentation of the issues and what kind of problems they cause in the Current State Analysis section. After the Current State Analysis there is a review of theory related to the problems and what kind

of solutions exist to solve these problems. The impact of the problems will be analyzed, as will the impact of the solutions. The solutions will be studied from a general standpoint, and from how they apply to projects of similar scale and point in lifecycle as System X. The interoperability of the solutions will be reviewed. A set of recommendations will be formed from solutions outlined in the theory section. And finally, there is a conclusion section.

2 Current State Analysis

This section describes the current state of the System X infrastructure, software and processes relating to the maintenance and use of the system. This section has been done by examining the current system and reading documentation provided by the target company. Information has been gathered from interviews with key personnel employed by the target company and monitoring how the process works during a deployment. The software development process has also been investigated during normal development life cycle.

2.1 Description of the Current System

The system has been in development since 2012. It started out as a custom web application for a single customer. It was originally developed as a replacement for their aging system. The first version resembled more like a collection of structured spreadsheets than the Enterprise Resource Planning (ERP) the system would grow to become in the next decade.

The software has been upgraded and rewritten while development has been taking place as the needs have changed, but the underlying technology has not been wholly reworked. The growth has been mostly organic. The software is written in the PHP language. It stores its data in a MariaDB database.

System X is located on virtualized Infrastructure as a Service (IaaS) systems running Linux. The system consists of a web server hosting the frontend of the software to the end users and the Application Programming Interface (API) in Javascript Object Notation (JSON) and a database server containing the actual data.

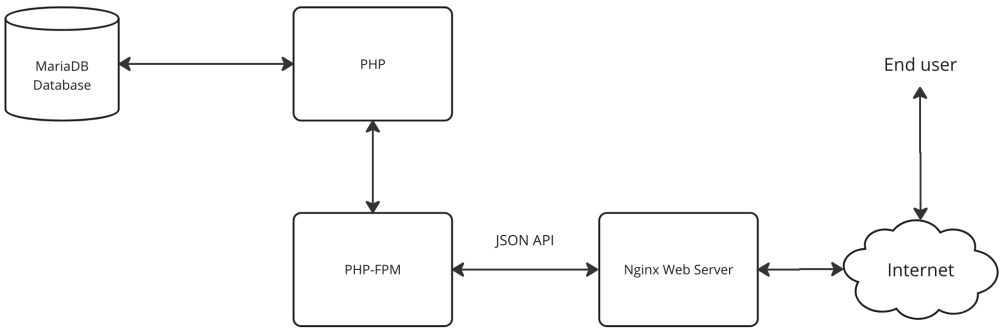

Figure 1: Current infrastructure block diagram

The demands of the upkeep of System X have outgrown the infrastructure and DevOps setup currently implemented. There is nothing inherently wrong with the current way, but there are a lot of things that could be made easier and more flexible for both the maintainers and the users of the system. The reason why these two problems are being addressed at the same time is that the motivation of the target company is to solve these problems in an integrated way that would work well as a package. The infrastructure decisions made will affect not only the ease of DevOps processes, but also what kind of possibilities exist. Many modern infrastructure paradigms such as containers affect the way the deployments or even the software development itself will be handled. If such a technology will be selected, it would negate the need to install the software to the disk of the virtual server or the developers' workstations, instead relying on a self-contained container image.

2.2 Infrastructure Issues

Infrastructure issues as understood here refer to the way the system installation is running and the underlying technologies therein. While other issues might be present, the issues discussed here are ones that the target company has noticed by themselves during the years System X has been in operation.

The infrastructure and its IaaS platform is sourced from a multinational IaaS provider, with which the target company has had a very good working relation for years. The target company wishes to continue using this provider, as much of the other IT infrastructure unrelated to System X is also handled by them. None of the infrastructure issues listed by the target company or which have been noticed during the creation of this Current State Analysis stem from the actions of the vendor, rather the chosen IaaS platform in and of itself doesn't lend itself very well to any kind of non-static setup without additional software to control it. The vendor has an API that allows for more flexible usage of the platform, but no API integrations have been implemented by the target company.

2.2.1 Infrastructure Issues at a Detail

The issue in this kind of static set up is wasteful resource usage. The IaaS system used allows for the addition of CPU or memory resources, but this process is entirely manual and necessitates the stopping and restarting of the server being modified. It would cause an interruption to the users' work, require the attention of a system admin to do, and generally be a nuisance. Therefore, the only option is to allocate these system resources to the servers calculated on the near-worst case scenario of usage. Naturally it is not possible to allocate all the resources the system could ever use, since there are unforeseen circumstances or even unwanted high-load scenarios like being under a Distributed Denial of Service (DDoS) attack.

In a DDoS attack multiple computers or other similar end-devices are used to cause massive amounts of network traffic on the target system. This has the effect of rendering the target system unable to serve any legitimate users, either by saturating the capacity of the network interface, or by forcing the target system to use all of its available CPU and/or memory resources into trying to formulate responses to the spurious requests. This kind of attack should not be mitigated by adding the resources to the system, since these kinds of malicious networks have the capabilities of countless, often hijacked systems, and trying to pay your

way into winning this kind of fight will not end well. A better way is to build reasonable defences like firewall rules beforehand, and if they fail, just let the system be unavailable until you can mitigate it in some sort of different way.

Another kind of unforeseen circumstance could be a software bug that causes massive resource usage, for example a memory leak. In a memory leak the software reserves system memory for some use, but after it is not needed anymore, it fails to free this reserved memory portion. The amount of memory left reserved can be miniscule, but since the system is always running, it can slowly fill up all the memory allocated to the server. The correct way to fix this issue is to fix the bug causing the memory leak, not to add more memory to the system.

An actual worst-case scenario worth taking into account for resource allocation could be calculated by looking at the resource usage at normal peak-hour levels and observing how many of the users are using the system at that time. This resource usage figure can then be extrapolated to the number of all possible users at the same time.

2.2.2 Impact of the Infrastructure Issues

The lack of fluidity in the server resourcing is an easily quantifiable budgetary issue in addition to being a technical one. By having to permanently provision a large enough virtual server to serve the calculated worst-case scenario, it stands to reason that most of the time the resource demand is not large enough to warrant such a large virtual server. But with the currently chosen technology, it is the only option.

Currently data for the difference between the most usage and least usage has not been collected for further use in a systematic way, only observed briefly at times of exceptionally high resource usage. The server infrastructure has been set up in such a way that the resource usage is monitored by an automated system collecting data of the CPU, memory, disk and network resource usage. All of these usage metrics have their own warning and alert levels. In addition to

these metrics, there are monitors for the general responsiveness and network accessibility status of the server. Exceeding the warning level causes the monitoring system to issue an email and smartphone app notification to the specified personnel, so appropriate action can be taken. Exceeding the alert level causes a robotic phone call to take place. Receiving such a phone call in the middle of the night or some other inconvenient time extolls unneeded stress on the recipient, which has served as a motivation to make choices that would automatically mitigate alert-worthy conditions.

As of writing this Current State Analysis in the spring of 2023, the system is performing well below the warning level of all the available metrics. Looking at the data available from the monitoring platform, the System X infrastructure has had a 99,999% availability during the past year with a single alarm caused by a 59 second outage due to network connectivity loss and no warning level criteria being met at any point. While this fact by itself sounds as if the infrastructure couldn't be running any better, it also means that with the current setup most of the system resources have been wasted with the system running at nowhere near peak capacity.

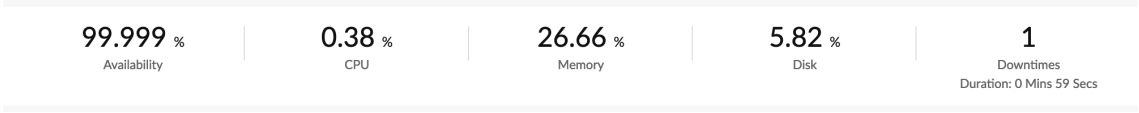

Figure 2: Monitor summary for 26th of April 2022 – 25th of April 2023

This singular network outage possibly could have been prevented by using a forward-facing content serving solution such as a reverse proxy, round-robin Domain Name Service (DNS) or other solution with multiple frontend nodes to prevent the single broken network connection from dropping the whole system off the public internet. However, this has not been seen as a major problem, considering the issue being so rare and the downtime so short. It should be taken into account when designing the new infrastructure nonetheless, since unlike the computing resource exhaustion which causes only service level degradation, the

broken network connection will render the whole system inaccessible to the users. Although the IaaS service provider is a very dependable one, the occurrences of DDoS attacks or such disruptions cannot be predicted.

2.3 DevOps issues

Devops is portmanteau of the words Development and Operations. The Development and Operations refer to Software Development and IT Operations. In this thesis, the analysis of the problem is mostly limited to the Operations side, i.e. the deployment of new versions of the System X software to the servers.

Any issues that specifically the software development of the system might have, are not taken into consideration to limit the scope of this thesis for practical reasons. The reason why they are lumped together regardless is that many of the solutions for this sort of a problem will integrate somehow into the software development side of the system as well, and hopefully help the development work also.

2.3.1 DevOps Issues at a Detail

The actual process of devops doesn’t really even come into play with the current set up of System X development. The software development and operations side of the system are almost completely disjointed from one another, even if the same people are doing both the development and the operations.

When the new version is being made ready for release, the software engineers do write the installation scripts to be ran at deployment, but other considerations are not done. While the current system does include Continuous Integration and running of automated tests, the resulting software deliverable is still not ready to run as is.

Updates to System X must be ran on the virtualized servers at the same time. The updates are scripted in advance, but many of the operations ran in these

update scripts require the data to be locked while the update is happening. In practice this means that the system must be changed into a maintenance mode, which prevents users from accessing their data. System X has so many internal and external users, both in the form of customers doing their planning work on the system and public users accessing their data or filling forms to be processed, that the usage of the system is practically nonstop. There can be times of lower usage, but there is no time when the system is sitting idle for more than a few minutes.

The upgrades also offer no easy inbuilt way of undoing the changes if some sort of bug is noticed after installing the upgrade. The whole state of the system can be reverted from a suitable backup snapshot, but there is no granularity in a change like that. Either everything in the system is reverted, or nothing is.

2.3.2 Impact of the Devops Issues

Many of these issues are items that require manual work. The problem of this is twofold. First of all, manual work takes more time from other work than automated tasks would. This problem is compounded by the fact that the upgrades to the system must be scheduled to the off-peak hours of usage, mainly weekends and/or nights. So, these tasks not only prevent doing any other work while underway, they also most likely cut into the free time of the person doing the maintenance work. This spending of work time incurs an actual measurable cost to the company. What is harder to quantify is the impact on the work of the customer organizations with them not being able to access their data or having to plan around the system outages, but it is not far-fetched to accept it as monetarily non-zero.

Second, the operations that require manual work are very unforgiving in the way of making even small mistakes during the process. While running the scripts or moving files around on the servers, absolutely nothing must go wrong or be mistyped. Most if not all of these manual work items have been written down as formal processes at one point or another, but yet again writing and updating

processes requires even more manual work. This again has a measurable and real cost attached to it.

The lack of any return path to the previous version from a newly installed one can be a major problem in case a certain kind of bug is noticed in the production environment. If the bug is a “showstopper” that cannot be left in place, it must be fixed, and the fix be immediately installed over the current version of the software. Such an occurrence is a stressful event for all those involved, and there isn’t any other particularly good way to deal with a bug like that. This means that a software developer must be present during the whole maintenance window whenever a new version is installed, and possibly for some time after while the system is being tested for a final time.

From discussions with the target company, this sort of deployment process is not particularly enjoyable one. Therefore, it is done as infrequently as reasonable. This means that often the deployments have a lot of changes rolled into one update, instead of nourishing a culture of several rapid deployments or a full-fledged Continuous Deployment process. If a bug found during installation or final testing on the production server cannot be fixed in time, the whole update will be cancelled while reverting to the previous state from a backup.

The lack of visibility to how the system will function in the production environment was also seen as a major point of concern. By programming System X installed on their own workstations the developers can verify that the end result works on their workstations. During the testing phase of the continuous integration service, they gain a better understanding on how the system will behave on a similar type of Linux server environment than the production environment. But the testing environment still does not have all the data that the users have generated during the decade of use on the system. This data of course cannot be accessed by the developers. The data can include all sorts of edge cases that simply won’t be thought of by the developers generating test data for their test cases. While not previously thought of by the developers, some sort of test data generation from

pseudonymized and/or anonymized subset of the real data could help to address the problems noticed in this part of the process.

3 Theoretical Background and Best Practices

This section builds theory on top of the problems described in the Current State Analysis part of the thesis. It includes a review of the current landscape of possible solutions to the problems presented in the Current State Analysis. This section also includes a look into data from studies that target similar problems.

3.1 A Review of Possible Solutions

The best kind of solution to the problems presented in this thesis would be one which solves all of the issues in a cohesive way. The solution most likely will not be a single product or service, but a set of processes and pieces of software that have been proven to work well with each other. We will look at these solutions first separately, and then as a combined package.

None of the problems are specific to the target company or their System X, and they are ones that can be generalized quite well. In the best-case scenario, the solutions can also be generalized, so that this thesis will provide guidance for other parties aiming to solve similar problems in their own infrastructure and DevOps processes.

3.1.1 Infrastructure Solutions

The problem of server infrastructure scalability issues is quite universal. Several solutions have been invented. They are categorized under the term autoscaling. Autoscaling can work either vertically or horizontally. In vertical autoscaling, the server instance can receive more computing resources as they are needed, such as CPU power or memory. In horizontal autoscaling, more server instances or “containers” can be started up or suspended when a differing amount of computing resource is needed. Both of these scaling paradigms require a different type of underlying set up and support from the platform they are deployed on. Not all hosting platforms can support any type of autoscaling solution.

Some of these autoscaling solutions are platform-specific and some are platform-agnostic. Platform-specific autoscaling solutions can be found from all the major vendors for example several different solutions such as Elastic Compute Cloud (EC2) for Amazon Web Services (AWS), Autoscale for Windows Azure, and Autoscaler for Google Cloud. Often the platform-specific solution is not the only one supported, so also some of the more general autoscaling software can be applied if required.

All of the above-mentioned platforms also support Kubernetes. Their systems are called Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE) respectively. Kubernetes as of 2022, is the most widely deployed container management system used by nearly half of all organisations that have a containerisation system [3]. Kubernetes can support either vertical or horizontal autoscaling, with horizontal scaling seemingly the one most often encountered.

For this thesis, the target company's infrastructure is currently set up on a platform that does not have its own custom autoscaling system. The most obvious possibilities are to install a custom installation of such a system and rely upon the provided API to control the server instances, or use a managed Kubernetes cluster, which the service provider does offer. If such an integration is provided by the chosen provider, it would be wise to at least strongly consider it. It will most likely work with less issues than a completely custom solution, but it is also covered under the customer service agreement in case questions arise. Such a service most likely costs something, so it is up to the organisation to calculate whether supporting a custom autoscaling system will be cheaper than buying one from the provider outright. In the case of the target company, no full-time IT operations personnel are available to deploy and support these kinds of systems.

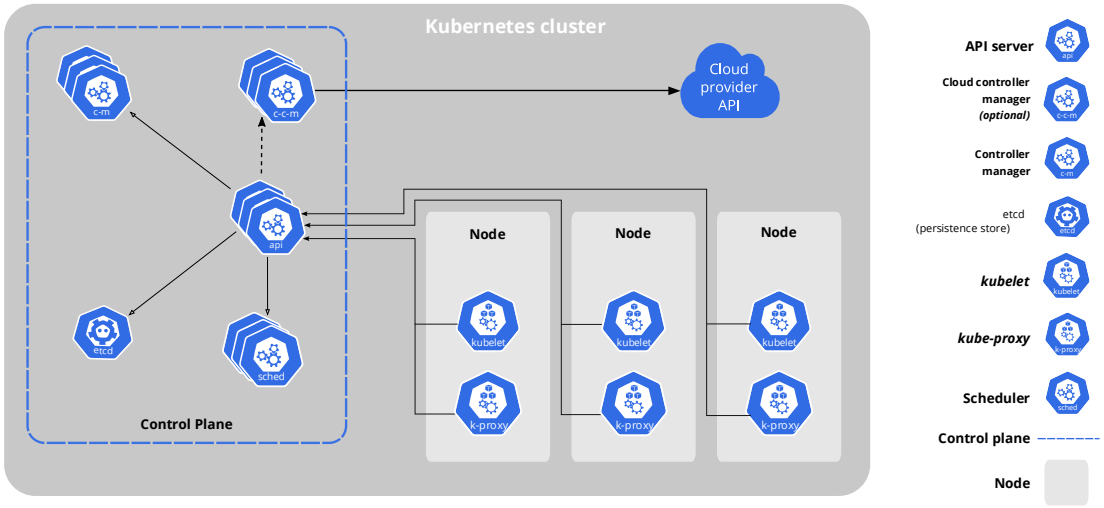

Figure 3: Components of a Kubernetes cluster [4]

A horizontal autoscaling solution works by having a central controlling system and at least one node to serve the actual request. The central controlling system keeps track of the nodes and their status and decides if more nodes should be started, or existing ones stopped based on demand. In a Kubernetes system the controlling system is located in the Control Plane. The Control Plane stores its configurations in the “etcd” persistent storage. The Controller Manager and the Scheduler are responsible for orchestrating the functionality of the worker nodes. In the previously mentioned managed clusters, the Control Plane also includes the Cloud controller manager, which is used to communicate with the service provider’s systems. The Kubernetes API server is used to communicate from the control plane with the worker nodes. The worker nodes exist outside the Control Plane, and they do not have any persistent storage for themselves.

Autoscaling solutions in general must also have some method of directing user traffic to the nodes, such as a load balancer or a reverse proxy set up. Depending on the technology, this is either working closely with the controlling system or an integral part of it. This is because the nodes and their Internet Protocol (IP) addresses are ephemeral and change with every start-up of a node, so traffic cannot be routed directly to them. Therefore, some kind of solution is required which is located at a known address accessible from the public internet. Again, using Kubernetes as an example, it provides several types of services such as

ClusterIP or LoadBalancer to facilitate connecting to nodes depending on the type of access required, internal or external.

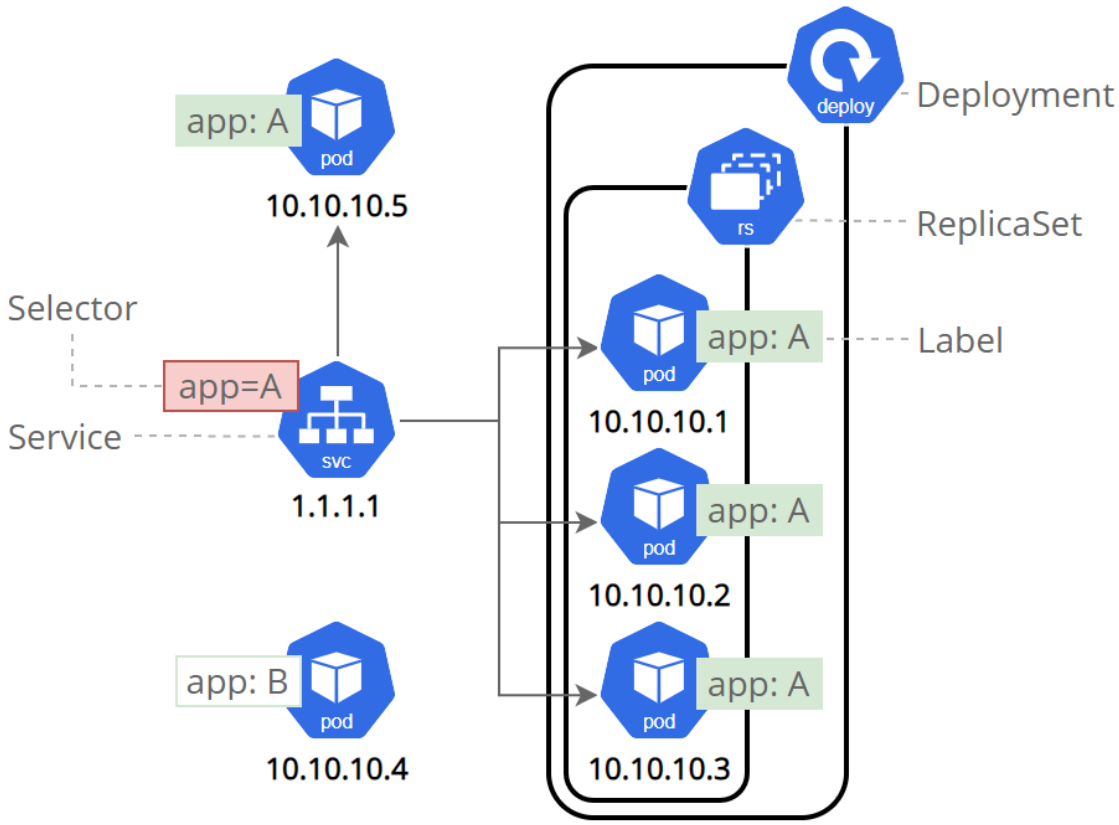

Figure 4: Exposing a Kubernetes app with a service [5]

In this picture example, the service is “App A”, and traffic destined to App A will be distributed between all the pods labelled as App A.

The statically defined server infrastructure isn’t only old-fashioned from the IT operations standpoint, but also is an obvious issue when thinking about the green transition and rising energy costs. Therefore, when comparing the impact of an autoscaling system implementation to the infrastructure, some kind of cost should also be attributed to every organisation deciding just to install more powerful servers instead of taking the time and money to invest into an autoscaling system.

3.1.2 DevOps Solutions

The nonconforming definition of DevOps makes it difficult to ascertain if an organization is “doing DevOpS” or not, at least when compared to implementing autoscaling or not. It could be argued that any organisation developing software and deploying it to be used somewhere is already doing DevOps. Also because of this, drawing a line whether or not a DevOps solution will solve an organization’s DevOps issues can be difficult. For the target company, any kind of streamlining and cost-savings would be a welcome one.

As with the autoscaling solutions, the most popular cloud service providers have their own integrated workflows with their own preferred tools. The providers also supply a large amount of documentation. The most popular cloud vendors have packaged ready-made workflows for their platforms such as Azure DevOps, AWS for DevOps and Google Cloud DevOps Research and Assessment (DORA).

Tools are only one part of the whole DevOps package. Adopting the DevOps as a holistic way to develop software in the entire organization instead of just a single action in the larger workflow requires a large number of processes to be defined or redefined.

As an example of what is included in a DevOps workflow, the following are DevOps best practices as listed by Amazon AWS in their DevOps documentation [6]. Amazon is a large software development company for their own needs, but also is a well-known provider of solutions and tools for the development of modern web applications.

- Continuous Integration
- Continuous Delivery
- Microservices
- Infrastructure as Code
- Monitoring and Logging
- Communication and Collaboration

Many if not all of the items on this list cross over slightly to the infrastructure side of this thesis. This is the reason why the Set of Recommendations is being created to address both the infrastructure and DevOps issues faced by the target company. These items in the best practices list all include both processes and software to facilitate those processes.

Continuous Integration

Continuous integration is the practice of making small incremental changes to the code of the software project. A continuous integration platform detects these changes in the version control system and runs the compilation, integration tests, unit tests and other quality assurance tools required by the project specification. Examples of tools are Jenkins, cloud provider specific ones such as Azure DevOps Server or AWS CodePipeline, or version control provider specific ones such as GitHub Actions or GitLab CI.

Continuous Delivery

Continuous delivery extends the continuous integration workflow further. In continuous delivery not only are the software deliverables prepared and checked for release, but they can also be automatically deployed into the production environment. Most of the continuous integration tools have implemented provisions or extensions to facilitate seamless integration of both methods into a CI/CD pipeline. This is why the most widely used tools for CD are ones also used for CI.

Microservices

Microservices is an architecture where the software application is made up of small individual components, which talk to each other. This is contrasted to the traditional monolithic architecture, where the software is a single undividable block. The inclusion of microservices as a part of DevOps, while obviously an architectural decision instead of a software development process, can be argued with the ease of deployments amongst other things [7].

However, whether microservices should be adopted by a project at all should be considered first as a separate entity. Microservices pioneers Martin Fowler and Sam Newman advice that first one should make sure to have a “really good reason” to go with microservices [8]. Also, there are other ways to look at the architecture that lie between a strict monolith and a strict microservices architecture, such as modularizing an already existing monolith. Modularization is a requirement in order to transition from a monolith to microservices, and it can be enough on its own [9].

Infrastructure as Code

Infrastructure as a code is a way to define the infrastructure setup required by the software deployment in a structured machine-readable format. This allows the deployment to happen automatically even from the very first time the servers are created. The developers have to programmatically define what kind of environment is required, for example where the files should be placed, what kind of other software should be installed, or which network ports to open.

With the infrastructure setup being in code, it can reside in the same version control repository as the application’s source code is. That way the changes to the infrastructure can be versioned, tagged to releases and so on in the same

manner as software changes are done. It is in a way a core principle of DevOps that allows the development and operations side to be in sync-lock relative to each other. A popular tool used for this by several vendors and in customized forms is Ansible, which uses the YAML markup language to define the environment.

Monitoring and Logging

Monitoring and logging are of course important in any kind of application, not just ones adhering to the DevOps mentality and workflow. Monitoring is what happens real-time to alert relevant personnel in case of an abnormality. Logging is running in the background, and the output is generally used after a problem situation has arisen in order to facilitate the troubleshooting.

Monitoring and logging require extra attention when implementing in a fully DevOps environment, as the microservices architecture and software defined infrastructure can result in volatility of the servers. Therefore, traditional system logs and text files might not exist anymore when they would be needed. Tools such as Grafana and Kibana can be used to collate the information streaming in from multiple sources. [10]

Communication and Collaboration

Communication and collaboration in general are very integral to software development or any large project of several people. However, specific attention is given to mitigating the organizational “culture shock” of adopting DevOps. As argued by Todd Waits and Aaron Volkmann, “DevOps is not something you purchase or have implemented by a DevOps Engineer.” Instead, at its core it “is a culture of communication and collaboration” [11]. Still, tools and techniques

aimed at describing the processes inherent have been defined, such as ChatOps to enhance the use of chat programs and bots in DevOps workflows.

3.1.3 Interoperability Between Solutions

The wide array of tools and processes that can be leveraged to fulfil these infrastructure and DevOps facets must be distilled to come together in a meaningful way. Care must be taken when selecting among the suitable options to form a coherent package, or at least to not cause compatibility issues.

The choice of infrastructure solutions will affect nearly all of the DevOps best practices outlined above except for Communication and Collaboration.

The ease at which Continuous Integration and Continuous Deployment can be implemented will depend greatly on not only how convenient deployments are, but there also other new possibilities brought with a new type of infrastructure set-up. Examples of such are rolling updates to negate downtime, or rollback features to undo the update if problems were to be encountered.

Implementation of Microservices will be made possible through infrastructure choice, as not all platforms will support microservices as readily as others. If microservices are brought on as a completely new paradigm to the team or company, it might also be wise to not totally depend upon them either. The adoption of microservices as a new program architecture might require extensive rewriting of the application code and could stall the progression of the DevOps process integration to the project.

As the name suggests, Infrastructure as a Code is very much a major infrastructure consideration. If this part of the DevOps best practices is to be implemented, the infrastructure must be one to have support for being defined programmatically. The technical possibilities of Infrastructure as a Code will also be leveraged in Continuous Deployment.

Monitoring and Logging is a concept which at its core should be possible with any type of infrastructure or server. However, it will require additional programs around it when dealing with automatically started and stopped containers or nodes that might not have any permanent storage or whose quantity or existence might not be even known to any one person. The infrastructure solution must be able to also handle automatic connections to the monitoring and logging system selected.

3.2 Academic studies

The scale of the impact of the problems on the target company has been measured or estimated to an extent by the company itself. The estimations are presented in the Current State Analysis section. While this paints a picture of the situation at the target company and provides enough incentive to start work on improvements, it does not make a wide enough dataset for statistical analysis.

3.2.1 Studies on Autoscaling Infrastructure Solutions

As the leading solution for infrastructure issues stated in this thesis, Kubernetes was chosen as the study example. VMware, a virtualization software company, conducts a yearly study titled The State of Kubernetes. In the survey for the year 2023, 98% of the respondents reported experiencing operational benefits from Kubernetes. The top benefit was improved resource utilization, given by 50% of the respondents.

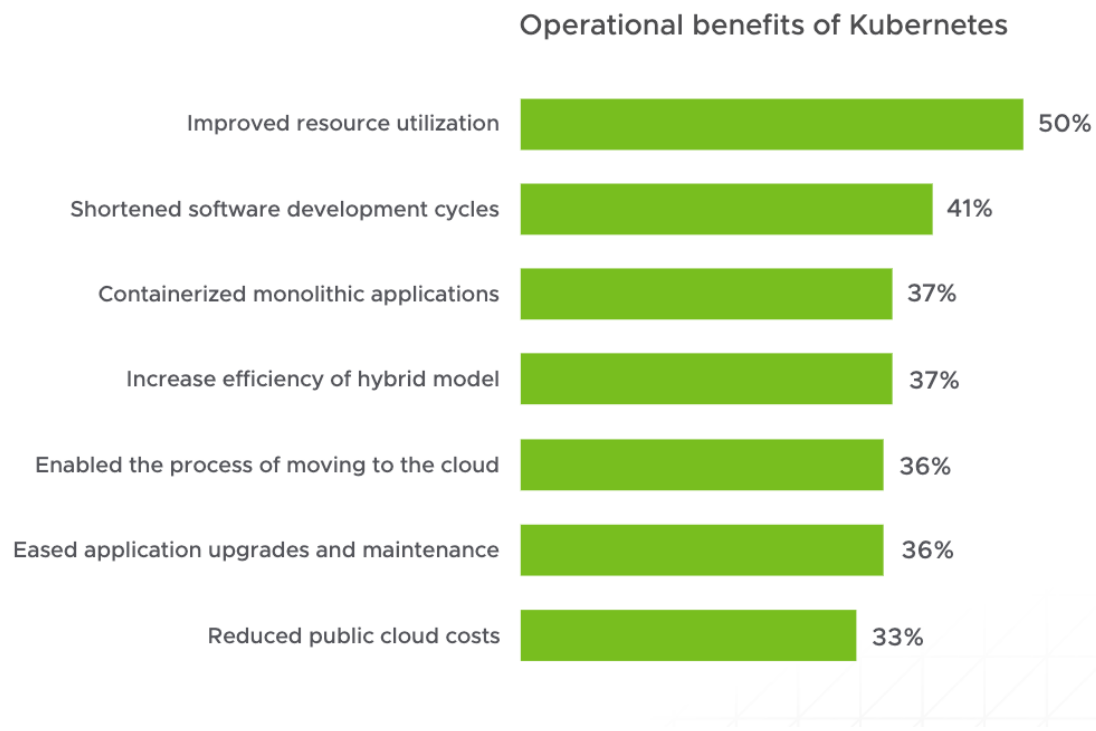

Figure 5: Operational benefits of Kubernetes from the State of Kubernetes 2023 study [12]

Of the 752 respondents to the survey, 90% agreed that cloud native technology, including Kubernetes, is transforming the way their business operates. 91% agreed that Kubernetes has benefited their entire organization, not just IT. The most reported business benefits were related to efficiency and productivity. [12]

3.2.2 Studies on DevOps

Analysing reliable and comparable level of studies on each of the specific issues step by step can be difficult. However, separate studies on the field have been conducted. A survey done by CloudBees with Hurwitz and Associates in 2017 on 100 enterprises which have implemented continuous delivery revealed that on average the enterprises saved 66 hours of developer work time per year [13]. 81% of the respondents felt that the adoption of continuous delivery improved their ability to bring value to customers and deliver on objectives [14]. None of the respondents reported a decline.

In 2019 a qualitative study by J. Bogner, J. Fritzsch, S. Wagner and A. Zimmerman was conducted on 14 service-based systems implementing microservices. The impact of microservices was mainly rated as positive. Most improvement was reported for maintainability of the software, but issues with security were noticed. Several of the systems targeted by the study were noticed to not implement or only partially implement some of the microservices characteristics such as decentralization. [15]

The partial adoption of DevOps or other agile processes might not be a failure in the implementation of the processes. In their 2016 paper “Exploring the impact of situational context – A case study of a software development process for a microservices architecture” Rory V. O’Connor, Peter Elger and Paul M. Clarke make the point that the process should adapt to the company and its ways of working instead of the other way around. Instead of abandoning the part of the process that does not fit the working culture, they do a case study on process discovery for a small software development company and describe ways to apply a situational factors reference framework. [16]

4 Set of Recommendations

In this section a specific set of recommendations is presented for the target company. This set of recommendations includes a cohesive set of software and processes that will benefit the infrastructure and DevOps ecosystem of System X. Once implemented, the improvements to the ease of software development will be noticeable by both the software developers and the end users of the system. The quality of the end product and the pace at which the software updates can be delivered should both be improved. Also, the maintenance and operations of the server infrastructure should be more predictable in their workload and more reliable in general.

4.1 General Overview of the New System

The new system proposed consists of a process meant to streamline the lifecycle of any change in the System X software from the technical side of the development process all the way to how the product is served to the end customer. The process here is defined specifically as the technical solutions meant to take place of existing ones in the current workflow, not all the intricacies of the software development business such as project management or change tracking.

The set of recommendations also includes a recommendation for software both on the servers and the developers' workstations, and what kind of changes should be brought to the infrastructure supporting the development and IT operations of System X.

The software selected will be one that supports the process, and the process will be one that supports the software in a complete way in order to have the best possible chance of success for the adoption of the process, and satisfaction for the target company.

4.2 Processes and Tools

The current software development process of the target company is somewhat agile at its basics. The features and bugs of System X are tracked in a Jira Cloud workspace, and the general development process is Scrum. Therefore, leveraging this as a foundation the DevOps process will be easier to implement compared to a traditional nonagile model such as waterfall.

The new infrastructure will be based upon the Kubernetes ecosystem. The cloud service provider used by the target company natively supports a managed Kubernetes environment, making the implementation of such a system considerably much easier. The managed Kubernetes cluster will be used for System X.

Kubernetes is also the market leader for containerized solutions, making finding support and future developers knowledgeable in it more likely.

Kubernetes will also facilitate DevOps practices such as Infrastructure as Code out of the box. There are synergy advantages when combining new DevOps practices and the currently implemented partially compatible solutions into a unified system.

The six-part DevOps best practices list will be partially implemented as a beginning. Parts of the list will not be implemented fully due to developer resource limitations. In the future, when additional resources are available, the adoption of further, now-skipped sections can be continued.

DevOps practice	Implementation status
Continuous Integration	Already implemented
Continuous Deployment	Will be implemented
Microservices	Will be minimally implemented
Infrastructure as Code	Will be implemented
Monitoring and Logging	Already implemented – will need reimplementation
Communication and Collaboration	Already implemented

Some of the best practices of DevOps are already implemented by the target company, but they must be reimplemented or subjected to other modifications. That way they will better serve the DevOps paradigm as a whole instead of independent parts.

4.2.1 Continuous Integration (CI)

The target company currently employs a Jenkins server as the basis for their CI system. This Jenkins server is notified whenever there is a change to the version control system, Git. The Jenkins server executes end-to-end tests with Selenium and unit tests with PHPUnit. This practice can be continued as is in the new DevOps system, as the developers of the company are already familiar with it. The CI system will however require changes in the implementation phase, to further enhance its capabilities and better integrate it with other systems.

The target company has previously hosted their Git version control as a self-hosted installation. They have since migrated to using GitHub for their version control needs. The current Jenkins set up is not fully integrated to the GitHub system. The integration should be expanded in the future to ensure better

collaboration, faster response time to pull requests and overall robustness of the software supporting the development process.

4.2.2 Continuous Deployment (CD)

Continuous Deployment is currently implemented in a rudimental way. The Jenkins CI server deploys the frontend and the backend of the software onto a testing server in order to run the tests required. It does not however deploy the database required by System X; this database is already installed on a different server. Database changes must be brought to the database server and Jenkins currently has no way to deploy them. The database system will require modifications in order to also automate this step. The Jenkins installation will be changed so that instead of installing dependencies and moving files, it will package the software container and run the container for the testing purposes.

The usage of Jenkins will be continued but expanded for CD as well. The production version of System X will also be brought to the continuous deployment ecosystem. It will benefit from the containerization of the software environment, as with the successful implementation of containerization each consecutive version of System X can be an iteration of the same container set-up.

The Jenkins configuration will be changed in such a way that it will be watching for a Git commit containing a tag starting with the word “release”. If this tag is encountered, the release pipeline will start and the branch containing this commit will be deployed to the production servers.

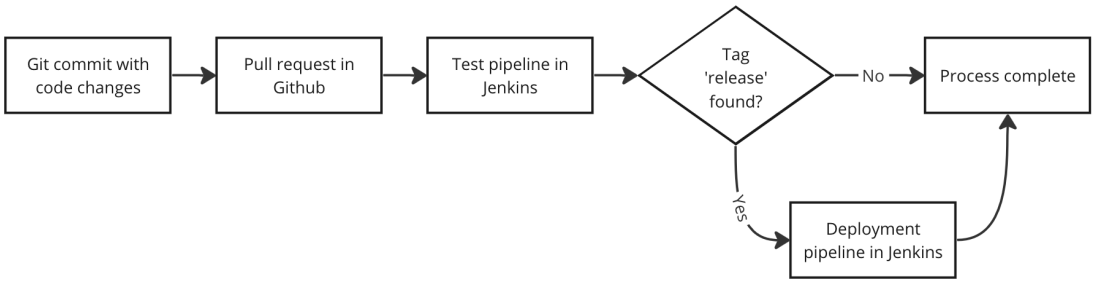

Figure 6: Jenkins pipeline with added Deployment section

The pipeline will work as it currently works, but with the added section of the Deployment pipeline. If the pipeline fails in the test section, it will not continue further to production deployment and results will be reported in the pull request.

4.2.3 Microservices

Currently the System X infrastructure is a monolith. It will be partially split into a more microservices-esque direction, but only as far as is required for the containerization and other architectural decisions.

Rewriting the current base platform of the software into a fully microservices architecture would require too large of an investment compared to what can be allocated for this project at this time. While microservices are a part of the DevOps best practices list, full rewriting now is not recommended to the target company to ensure the other changes will be successfully accomplished. However, the changes which are done to the architecture will be done in such a way that will leave open the possibilities to further expand the microservices style evolution in the future.

The System X software as it currently exists already shows promising signs of diverging services into a modular monolith architecture due to the use of internal APIs by the development team. While this direction will most likely continue naturally, a successive plan should be made to further divide the system and services to ensure rapid and agile development.

4.2.4 Infrastructure as Code

Infrastructure as Code will be implemented as part of the Kubernetes architecture. Kubernetes is configured by way of configuration files written in the YAML markup language. The Kubernetes configuration files will be included in the Git version control system of System X. This way the infrastructure configuration will follow the development of the software code at the same pace.

Changes can be tracked, and collaborative efforts can be extended to the infrastructure as well as the software code.

4.2.5 Monitoring and Logging

Currently the target company uses ManageEngine Site24x7 as their monitoring and performance logging software. This software includes support for Kubernetes clusters. As the tool is currently used by the company and the company has indicated that they would prefer to continue using it, further comparison against other competing solutions will not be done.

Application and server logs are currently saved to the file system of every server. As the infrastructure is changed to a Kubernetes cluster, this will no longer be possible or meaningful due to the lack of persistent storage. The ManageEngine Site24x7 software includes support for collecting logs from Kubernetes pods. Therefore, logging will be implemented into the same system that already handles the monitoring of the servers of the target company.

4.2.6 Communication and Collaboration

The target company already employs several integrated unified methods for internal System X communication. The chat software Slack is used for day-to-day communications, informal and formal. Feature and bug tickets are handled by Atlassian Jira Cloud, and documentation is gathered into Atlassian Confluence, a wiki style platform. Jira Cloud and Confluence enjoy a high level of integration, being from the same software vendor. Linking between the two platforms and embedding Jira tickets into Confluence documents, opening Jira boards etc. are supported by default. Slack also has optional integrations to both Atlassian products, allowing tickets and other documents of note being previewed directly in the Slack conversation.

The Slack software integration will be extended with a plugin for the version control system and CI system. A special channel will be made, where automatic

messages are generated from new pull requests opened, existing ones merged and the status of the CI pipeline and tests. This way the development team will get notified when something requiring their attention is happening.

Depending on the capabilities of the monitoring system, integrations between it and the Slack software is also recommended for the same reason. It will collate all the necessary updates on the state of the software in development and in production under one application for synergy benefits.

In the adoption of the new DevOps system, care must be taken when presenting all the new processes and tools to the developers to prevent “culture shock”. The new tools selected for the development team must be supported by enough documentation and training that they will be easy to learn.

4.3 Infrastructure Specifications

The infrastructure will be split into a modular system based upon the Kubernetes architecture. The managed Kubernetes cluster from the cloud provider supports a high availability cluster with three control nodes and up to 200 worker nodes. This high availability cluster will be implemented for System X. Control nodes in the control plane cannot be added or removed in a similar flexible way as the worker nodes can. The nodes in the control node group will be deployed utilizing anti-affinity, which tries to place all the control nodes into separate physical servers to achieve better fault tolerance.

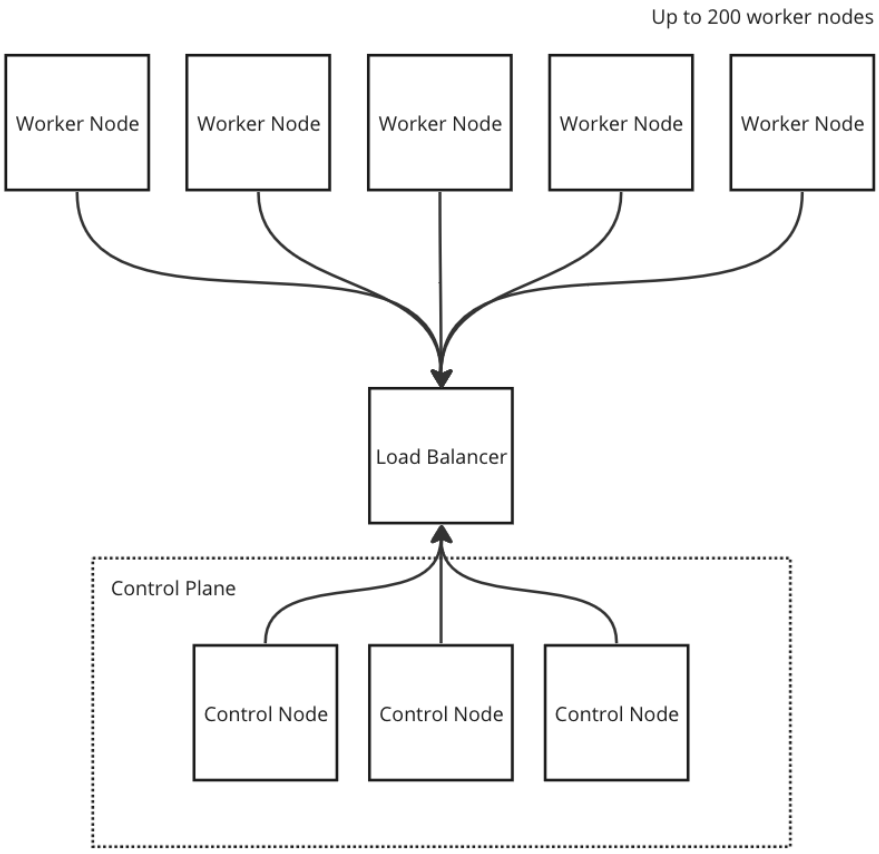

Figure 7: High availability Kubernetes cluster

The worker nodes will communicate with the control plane via a load balancer. All of the worker nodes' communication as well as the control plane communication will happen in a private Software Defined Network (SDN) inside the service provider's datacenters. The cluster will be set up in the Finnish data centers of the service provider. Most of the users of System X are in Finland. The Finnish data centers are slightly more expensive than their counterparts in central Europe, but in case stricter data protection measures are needed, the data will not be exported outside Finland. All of the nodes located geospatially in a close proximity will also reduce latency in intra-node communications.

For outside communications, a managed load balancer from the service provider will be implemented to expose the necessary services for end users to access System X through. This managed load balancer will integrate to the managed Kubernetes cluster with minimal effort.

The database of System X will be moved into a managed MariaDB database solution from the service provider. The managed MariaDB set-up will automatically create a high-availability cluster for the data, and it can be accessed via a private SDN. This way the access can be limited to the System X Kubernetes nodes, even though the databases can be physically in a different data center. This will help to enforce data security, since the database cannot be accessed from the public internet at all.

4.4 Implementation Path

The implementation of the new system can start progressing from multiple angles at the same time. Whether this is useful from a resource allocation point of view shall be defined by the target company. Many of the processes and tools in this Set of Recommendations depend upon each other if not completely then at least partially to achieve full effect. Therefore, some guidelines will be outlined here in order to facilitate reasonable progression.

Each of these implementation action points presented here are not atomic in nature, that is, they can be divided and subdivided into smaller portions. They should be approached as if they all were software development projects. It is recommended that the already existing Jira Cloud should be leveraged, and the action points be made into projects each with their own issues and backlogs. If this is not acceptable by the target company, secondarily it is recommended that the action points be made into Jira Epics in the current System X project. Either of these will achieve transparency in the process, but separate projects will allow better visibility into time estimations and velocity tracking of the implementation as a whole. Separating them as projects will however also require splitting the workload of developers into these projects from System X, whether by day of week or some other method. It will show as slowing the velocity of System X and forcing to take into account less days of effort per sprint per developer available.

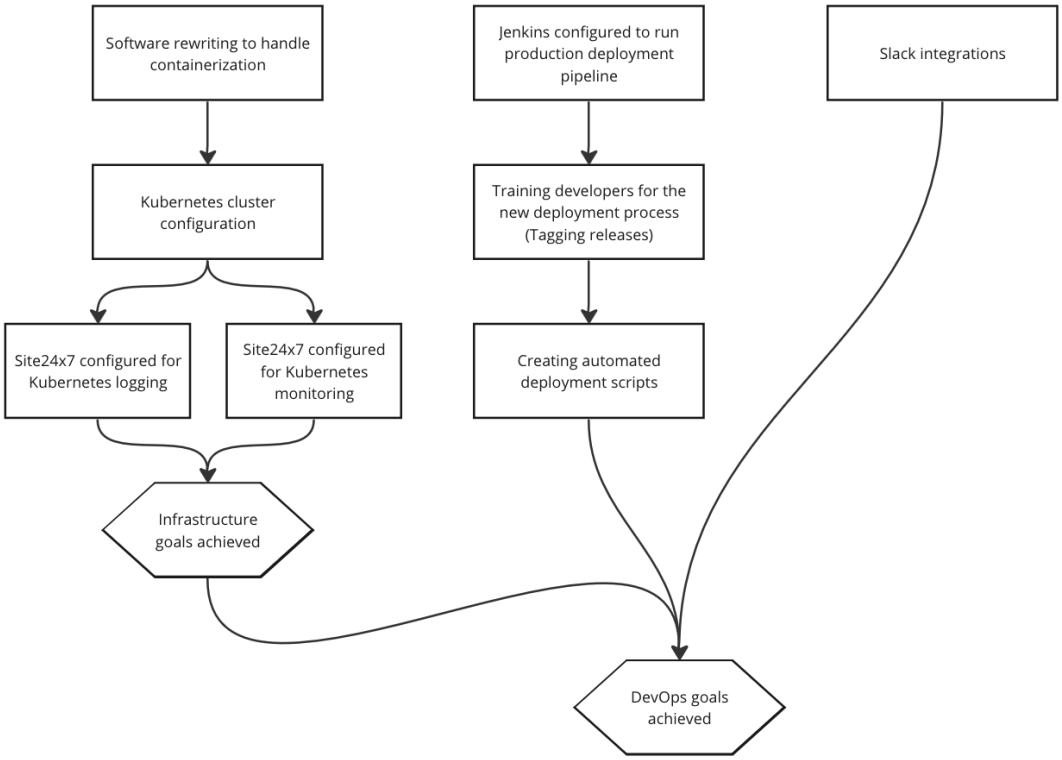

Figure 8: Implementation path

The implementation path includes multiple entry points from which the improvement process can be started. The improvements can be implemented one branch at a time, multiple branches at once or some mixture of both depending on available resources and other considerations. However, starting from the leftmost branch in this diagram is recommended. Creating a proof of concept of the new Kubernetes clustering system will lay groundwork on which to develop the CI/CD changes in the middle branch to the final specification in the first iteration. If the middle branch is developed before the infrastructure definition is completed, the changes will need to be partially rewritten. However, completing the CI/CD changes of the middle branch will increase productivity even with the current infrastructure.

The amount of initial work required for each of the action points should be minimized. At this stage a lean style will amount to the largest return on investment, because even very basic implementations will result in better

development pace in the future. After the infrastructure is defined as a code and the automatic deployment is in place, the system can be effectively bootstrapped and then iteratively upgraded using the system itself. Further improvements can be included in the normal development sprints as single tasks during regular software development. After this point the development and operations will be fully intertwined as a DevOps process.

5 Conclusion

After finishing the research and the Set of Recommendations, the solution to solve the target company's infrastructure and development process issues at the same time seems even more logical than it felt before this thesis. They seemed like two separate problems that had some relation to each other, but in actuality it would be harder to solve them separately than to plan to solve both. During the creation of the Set of Recommendations it became clear that these solutions do not only have synergy advantages to solving the problems, but they will transform the whole way System X will be developed and maintained in the future. The way code-defined infrastructure works in relation to the iterative software development process means that the system can almost rewrite itself. Code changes can have drastic changes on the actual infrastructure. The infrastructure can be dreamed to life, so to say. By writing the definitions on how the infrastructure should look like in a structured way, in the YAML markup files, the system itself will make them a reality. Coming originally from a traditional "a server is a physical computer" background, it feels groundbreaking, and it is no wonder so many companies have already adopted these practices and Kubernetes.

Hopefully this enthusiasm towards the new way of working will be also permanently instilled in the software development team of the target company. Even though the Set of Recommendations include a path to implementing all these ideas, it is greatly simplified from what the implementation project will necessitate. The actual project planning of the implementation could almost be a thesis in of itself. Because of its length the project planning will commence outside of this thesis, continuing from where the Set of Recommendations ends here.

It will be a great undertaking before all of the action points are fully realized. Even after they are complete, the DevOps thinking has to be an integral part of the workplace culture, or else the process will fall apart quickly. This thesis only takes into consideration the technical issues present. If the development team does not uphold the DevOps ideals, people problems cannot be solved with technical solutions.

The problems addressed by this thesis have been noticed by other companies as well, as there is a great amount of literature on solving such issues. Every software project is different, so the implementation starting point and path is different for each company. With developers changing companies or moving from project to another, the DevOps way of working can either be embraced or extinguished. If the developer feels that the processes complement their style of working, they will most likely also try to implement these in their future projects, thereby bringing with them a cultural change. The idea of a dedicated server will most likely never completely die, but one day it probably won't be the default choice for a new project.

References

1 Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, et al. Microservices: yesterday, today, and tomorrow. In: Mazzara M, Meyer B, editors. Present and Ulterior Software Engineering [Internet]. Cham: Springer International Publishing; 2017 [cited 2023 May 14]. p. 195–216. Available from: https://doi.org/10.1007/978-3-319-67425-4_12

2 Richardson C. Microservices patterns: with examples in java. First Edition. Shelter Island, New York: Manning; 2018. Chapter 3, Inter-Process Communication in a Microservice Architecture; p. 65-109.

3 Datadog. 9 insights on real world container use [Internet]. 9 insights on real world container use. 2022 [cited 2023 May 14]. Available from: https://www.datadoghq.com/container-report/

4 Kubernetes components [Internet]. Kubernetes. [cited 2023 May 14]. Available from: https://kubernetes.io/docs/concepts/overview/components/

5 Using a service to expose your app [Internet]. Kubernetes. [cited 2023 May 14]. Available from: https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/

6 What is devops? - devops models explained - amazon web services(Aws) [Internet]. Amazon Web Services, Inc. [cited 2023 May 14]. Available from: https://aws.amazon.com/devops/what-is-devops/

7 Wickramasinghe S. The role of microservices in devops [Internet]. BMC Blogs. [cited 2023 May 14]. Available from: https://www.bmc.com/blogs/devops-microservices/

8 When to use microservices: sam newman and martin fowler share their knowledge – dreamfactory software- blog [Internet]. [cited 2023 May 14]. Available from: https://blog.dreamfactory.com/when-to-use-microservices-sam-newman-and-martin-fowler-share-their-knowledge/

9 Ackerson TF Dan. When microservices are a bad idea [Internet]. Semaphore. 2022 [cited 2023 May 14]. Available from: https://semaphoreci.com/blog/bad-microservices

10 Why monitoring and logging are important in devops by datacademy. Ai - issuu [Internet]. 2023 [cited 2023 May 14]. Available from: https://issuu.com/datacademy.ai/docs/why_monitoring_and_logging_are_important_in_devops

11 Culture shock: unlocking devops with collaboration and communication [Internet]. [cited 2023 May 14]. Available from: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=442589

12 State of Kubernetes 2023 [Internet]. [cited 2023 May 14]. Available from: https://tanzu.vmware.com/content/ebooks/stateofkubernetes-2023

13 Driving digital transformation with devops and continuous delivery [Internet]. CloudBees. [cited 2023 May 14]. Available from: https://www.cloudbees.com/whitepapers/driving-digital-transformation-with-devops-and-continuous-delivery

14 Devops survey results: why enterprises are embracing continuous delivery [Internet]. CloudBees. 2017 [cited 2023 May 14]. Available from: https://www.cloudbees.com/blog/devops-survey-results-why-enterprises-are-embracing-continuous-delivery

15 Bogner J, Fritzsch J, Wagner S, Zimmermann A. Microservices in industry: insights into technologies, characteristics, and software quality. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C) [Internet]. Hamburg, Germany: IEEE; 2019 [cited 2023 May 14]. p. 187–95. Available from: https://ieeexplore.ieee.org/document/8712375/

16 O'Connor RV, Elger P, Clarke PM. Exploring the impact of situational context: a case study of a software development process for a microservices architecture. In: Proceedings of the International Conference on Software and Systems Process [Internet]. New York, NY, USA: Association for Computing Machinery; 2016 [cited 2023 May 14]. p. 6–10. (ICSSP '16). Available from: https://doi.org/10.1145/2904354.2904368