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

Sampsa Laapotti

A Case Study Measuring Changes in Deployment between Virtual Machines and Orchestrated Containers

Master's Thesis
Espoo, April 27, 2020

Supervisor: Eero Hyvönen, Prof.
Advisor: Riku Lääkkölä, M.Sc. (Tech.)

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

ABSTRACT OF MASTER'S THESIS

Author: Sampsa Laapotti

Title:
A Case Study Measuring Changes in Deployment between Virtual Machines and Orchestrated Containers

Date: April 27, 2020 **Pages:** 62

Major: Computer Science: Web Technologies, Applications and Science **Code:** SCI3047

Supervisor: Eero Hyvönen, Prof.

Advisor: Riku Lääkkölä, M.Sc. (Tech.)

Our world runs increasingly on software. Earlier studies have found evidence that software deployment frequency is a predictive variable about the organizational performance of a company producing a software product. We investigated the runtime and deployment environment change of a startup company (Kaiku Health) and measured if a change from cloud based virtual machines to a cloud based orchestrated container system (Kubernetes) affected the deployment frequence and duration of deployment. The deployment duration or deployment frequency did not change in a statistically significant manner. Some evidence was found pointing that orchestrated containers might scale better with large batch of workloads.

Keywords: Docker, Kubernetes, Continuous Delivery, DevOps, Container Orchestration

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Computer, Communication and Information Sciences

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Sampsa Laapotti		
Työn nimi:	Tapaustutkimus virtuaalikoneiden ja orkestroitujen konttien tuotantoonviennin eroista		
Päiväys:	27. huhtikuuta 2020	**Sivumäärä:**	62
Pääaine:	Computer Science: Web Technologies, Applications and Science	**Koodi:**	SCI3047
Valvoja:	Professori Eero Hyvönen		
Ohjaaja:	Diplomi-insinööri Riku Lääkkölä		

Ohjelmiston rooli elämissämme on kasvanut. Aiemmat tutkimukset ohjelmiston tuotantoonviennistä ovat havainneet tuotantoonvientitaajuuden olevan tärkeä ennustava mittari ohjelmistoja kehittävien organisaatioiden suorituskyvystä. Tämä diplomityö tutkii tapaustutkimuksen ja määrällisen tilastoanalyysin keinoin kasvuyritys Kaiku Healthin ohjelmistotuotteen tuotantoonvientiä ja sen muutosta yrityksen vaihtaessa tuotantoonvientiprosessiaan pilvipalveluissa ajautuviin virtuaalikoneisiin perustuvasta järjestelmästä toiseeen, pilvipalvelussa ajautuvaan orkestroituihin kontteihin perustuvaan järjestelmään.

Diplomityö ei löytänyt tilastollisesti merkittäviä eroja järjestelmien välille tuotantoonvientitaajuuden tai tuontantoonviennin keston mittareilla. Työ löysi viitteitä orkestroituihin kontteihin perustuvan järjestelmän kyvystä skaalautua paremmin lukumäärällisesti suuremmilla kuormilla.

Asiasanat:	Docker, Kubernetes, Jatkuva tuotantoonvienti, DevOps, Konttien orkestrointi
Kieli:	Englanti

Acknowledgements

What you are now reading is made possible by a huge network of people to whom I own a great deal of gratitude. This acknowledgement chapter is my meager attempt to start to express that gratitude. I am indebted by the humble and prudent care I have received from my beloved mother Hillevi and Father Matti, and by the care they have received form earlier generations. My dependable siblings Sanni and Juho deserve huge thanks for their patience with me during the years. In am forever grateful for all the times any of my friends have helped me either to understand how to do something and even more crucially helped me understand why some things are worth doing. I am especially honoured to have been able to spend the time used to write this thesis in our commune with Henri, Henry and Henje. The countless walks with Sami, Juuso and Maggie have kept me sane during this thesis project.

Now that single humans have been thanked, I want to thank more abstract categories of entities. Especially all of my teachers during the multi-decade path from Alhaisten Koulu in Salo to Aalto University in Espoo deserve to be thanked. In similar fashion I am superbly thankful for all the years I got to spend with the most nimble and innovating communities inside Aalto and UTU. Of those, I feel especially thankful for those who spend their time in salt mines or in the library, be it either a digital or a physical one. I am also unbelievably privileged to have been able to enjoy the merry company of all the great people at Tietokilta during my years of study.

This thesis would not have happened without my excellent supervisor

Eero, who carefully kept me on the straight, empiric road of research and dutifully pointed my failings to me. Most of the active thinking work for the thesis was facilitated by Riku, who as a thesis advisor had to bear the brunt of reading my typos and half-formed ideas. You were both essential for this thesis, thank you!

I also want to thank Kaiku Health and all the wonderful people working at Kaiku. You are have all inspired me in so many ways and without you this thesis could not have happened. I am especially thankful for Jussi, from whom I received countless tidbits about how to not embarrass oneself with statistics.

On the area of more mundane, simple matters like programming, I want to thank all the people who have ever contributed to Free and Open Software. This thesis is literally possible because of millions of lines of coded you people wrote that in the end produced this document and enabled the statistical analysis and businesses described in this thesis.

Espoo, 27.4.2020

Sampsa Laapotti

Abbreviations and Acronyms

CI	Continuous Integration
CD	Continuous Delivery
CSP	Cloud Service Provider
PR	Pull Request
VCS	Version Control System
ROR	Ruby on Rails
VM	Virtual Machine
k8s	Kubernetes
ePRO	Electrical Patient Reported Outcome

Contents

Chapter 1

Introduction

Our world runs increasingly on software. In order to produce value, software needs to be developed and the end result needs to be deployed to a production runtime environment. Software that is not in use can't produce value for the shareholders.

This master's thesis is a case study about two different deployment and runtime paradigms in the context of a healthcare startup company (Kaiku Health). This thesis compares a legacy deployment system that deploys software to pre-configured virtual machines to a new, modern deployment system that builds Linux container images (Docker) and then orchestrates their deployment on container runtime infrastructure (Kubernetes).

The deployment systems of Kaiku Health were instrumented and the resulting data was used to answer research questions. Research questions were generated before the measurements about the deployment systems were done. This comparison was done with the help of relevant variables identified by earlier research [1].

Prior literature has established that software deployment performance is relevant metric to predict the organizational performance of a company [2], thus making software deployment performance relevant from a business perspective.

Chapter 2

Background

In this chapter the reader is introduced to DevOps, Continuous Integration, Continuous Deployment and to the two different deployment paradigms relevant to the matter at hand. Software runtimes pertinent to the study are introduced. Some time is also spent discussing why DevOps matters from the viewpoint of business objectives.

2.1 DevOps

DevOps is a loosely defined set of technical and cultural practices used in development and operation of software products [3]. There is no single definition of these practices, but the term is commonly associated with practices that bring development (Dev) and operations (Ops) teams closer together. DevOps practices often aim to increase automation and lower manual work that teams do to run routine tasks inside their software systems.

Software development is often done in teams of varying sizes and responsibilities. In a small company a single team might be responsible for the whole development flow from creating features to deploying them to production. When companies grow, there is usually increase in teams and increased specialization for each of those teams. Often some teams focus on adding features (Dev) and other teams focus on keeping the software running in

production (Ops).

This division of labour can lead to conflicts between the teams. The development team wants to add new features and hence complexity to the software, as the operations team wants to keep the software stable in production [4]. Complexity and stability are often somewhat mutually exclusive, as increased complexity can result in problems when operating the software in production. DevOps aims to breach this gap and streamline the whole process without lowering the quality or throughoutput of software development process.

In practice, DevOps achieves its goal of streamlining the deployment by automating many parts the process of software deployment. Examples of this kind of automation are automated tests run in the specific automation environment called Continuous Integration Server (CI-server) and automation of the deployment of software to different environments.

2.2 Impact of Devops Practices on Organizational Performance

Why is DevOps relevant to software companies? Prior literature about the impact of DevOps practices suggests that success in DevOps predicts the organizational performance to some degree.

Forsgren et al. have conducted prior research on the impact of DevOps practices to organizational performance [2]. They have studied the DevOps practices in use by doing surveys on the industry and running statistical analysis on the data generated by those surveys. Key finding in their work is that some DevOps practices seem to predict organizational performance. We will introduce some of those practices later in section 2.3. Visual overview of the predictive power on organizational performance of each DevOps practice can be found from Figure 2.1 and is discussed with more detail in [1].

The most relevant practice in the context of this thesis in the list DevOps practices listed in Figure 2.1 is continuous delivery (CD). Continuous

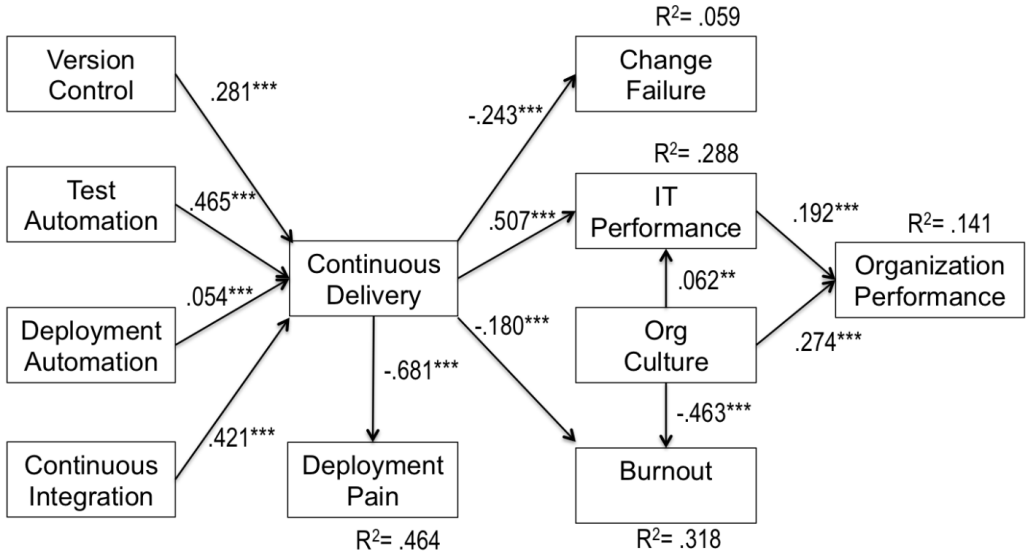

Figure 2.1: Forsgren et al. results about the impact of DevOps practices to organizational performance. Numbers marked with *** are beta-coefficients from their regression analysis. Practices and outcomes inside boxes are statistical constructs built based on the data clustering done in the study of Forsgren et al. Picture from [2].

Delivery is introduced in section 2.3.1. This thesis aims to measure how well Kaiku does CD and if the attempts at improving this capability are succeeding. This measurement is done via proxy measurement of deployment frequency.

2.3 Devops as a Set of Capabilities

How to measure how well a company is implementing DevOps?

If one asks the leadership of a company, one might get a quite rosy picture of organizations progress in implementing DevOps. The model by Forsgren et al. proposes more grounded measurable variables that one can use to assess the reality of implemented practices. Their model is based on large question-

naires with thousands of responses from hundreds of companies of different sizes. A relevant example of those questions is "How often your organization deploys code to the primary service or application?" with answer categories as ranging from "on demand (multiple deploys per day)" to "fewer than once every six months". [1]

This thesis views DevOps as set of capabilities that an organization has successfully acquired. This view has been advocated by Forsgren & Humble in their research about DevOps and its impact on organizational performance [2].

A dynamic capability is defined by Eisenhardt and Martin as "Dynamic capabilities are the antecedent organizational and strategic routines by which managers alter their resource base - acquire and shed resources, integrate them together, and recombine them - to generate new value-creating strategies" p.1107 [5]. In the context of this thesis the "dynamic" prefix is dropped from capabilities, as there is no need to contrast them with "static" capabilities.

Capabilities theory defines three types of capabilities for an organization: inside-out, outside-in and spanning.

Inside-out capabilities are capabilites that act primarily inside the organization. One example of this would be Continuous Delivery pipelines, that aim to make the development of software more streamlined and efficient [2]. This thesis is mostly about measuring one of those capabilities, the capability to deploy software rapidly to production.

Other capability types (outside-in and spanning) will not be discussed further, as those are not really relevant to the matter at hand. Further information about capability theory can be found from [5].

2.3.1 Continuous Delivery (CD)

Continuous delivery is a software deployment practice that aims to keep the software constantly in releasable condition and release it often [6]. This can be contrasted with releasing software on some time-based schedule, for

example three times a year. There is some evidence that releasing often correlates with good organizational performance [2].

2.3.2 Continuous Integration (CI)

Continuous integration is a software development practice where changes made by each developer are integrated multiple times each day [7]. In practice this means running automated tests and checking mergeability to master software development branch automatically. The frequency of this integration depends on how often new commits are made and usually does not require manual labour from the developers, except in the case when integration problems surface during automated tests. But these problems are usually solved quickly, as no new work can be built on the problematic parts of the software, as it is not merged to master before integration passes.

A relevant comparison to this continuous integration practice would be the older model of developing software using the "waterfall" methodology, originally introduced by Royce [8]. In waterfall projects, design, coding and testing are done on different phases of the project. This can lead to substantial delays between coding, testing and integrating different parts of software together. New code can be built on top of old, non-integrated code, and thus fixing the problems found in integration phase can require substantial, recursive changes to the codebase.

Continuous integration is usually implemented by using a self-administered CI-server or an external CI-service. This server pulls the changes made by a developer, builds them and runs various tests to check for regressions. Once these tests have passed, the CI flags the build as successful.

2.4 Production Runtimes for Software

Software need to be run on some hardware and requires some amount of external dependencies in order to be useful. A software runtime environment

provides these dependencies. These requirements and dependecies are different for different software products. For example, embedded software and web software require different dependencies and runtimes.

This thesis focuses on single web-based software and two different runtimes used for it. These are introduced in 3.2.

2.4.1 Cloud Based Virtual Machines

One common way to provide a runtime environment for web-based software is to provision a virtual machine from a cloud provider and install an operating system and dependencies of the software in question on it. This can be contrasted to buying the hardware and operating it in-house. There are multiple benefits to subcontracting the physical operations and hardware ownership to specialized companies, especially for smaller companies. Further information and a modeling solution to costs estimation has been published by Martens et al. [9].

2.4.2 Linux Containers

Encapsulating different parts of the operating system and software running on it is a common practice in software engineering. This allows running multiple programs fluently on the same hardware and makes managing the environment easier.

One way to encapsulate the runtime environment of a process is building a container around it. The Linux Kernel provides mechanisms (control groups, namespaces) that allow us to partition the userland for container usage.

Linux containers are often mistaken as virtual machines. This is not true, as all containers running on a single host share the same kernel. In contrast, virtual machines are usually running their own kernel. This is illustrated in Figure 2.2 showing comparison of solution stacks in Linux containers and VMs.

This thesis focuses on Docker-containers, a special case of Linux contain-

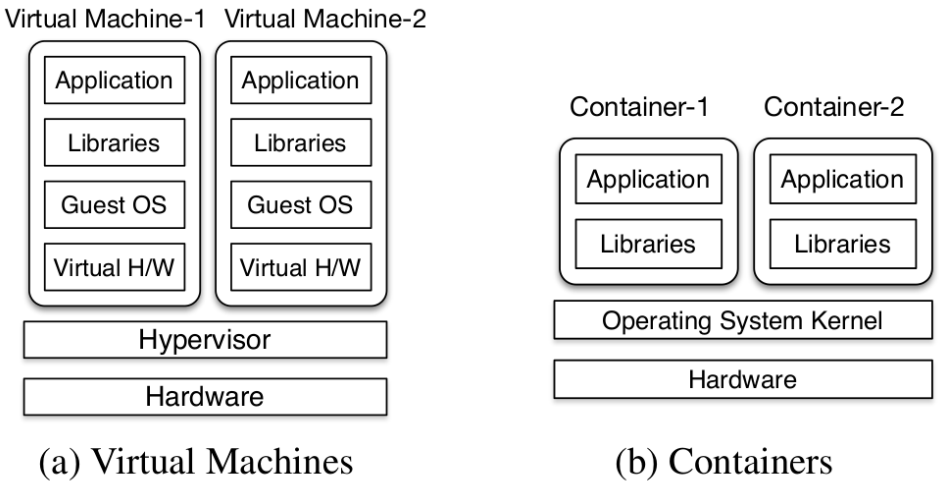

Figure 2.2: Illustration about differences of solution stacks in containers and virtual machines. Containers and VMs share different components of their stacks between other instances of their kind. This results in different abstractions for resources and different security implications of resource sharing. Illustration from Sharma et al. [10].

ers.

Compared to normal Linux processes, containers have some overheads in terms of performance, but the overheads are not as high as with virtual machines [10].

From a security perspective, containers are not as isolated from the host as virtual machines are, as containers share the underlying kernel and hence have less security barries in place.

2.4.3 Container Orchestration

Container orchestration is quite a new and upcoming concept in software engineering. A good introduction to the concept can be found from Emiliano Casalicchio's survey [11].

Container orchestration is the set of operations that infrastructure providers and application owners undertake to define how to select, to deploy, to monitor, and to dynamically control the configuration of multi-container applications in the cloud [12]. The point of orchestration is to make sure that the container running the software has all the neccessary resources and dependecies at hand.

There are multiple different software products implementing container orchestration with different divisions of responsibility. A quick overview of different orchestration systems can be found from Gogouvis et. al [13].

2.4.4 Kubernetes

Kubernetes is a Container Orchestration system that is build from the ground-up to solve the problems of running containers en masse.

The most important feature of Kubernetes is its declarative nature. This means that users define the resources and state wanted, and Kubernetes tries to achieve that state and create the resources it requires. This enables many useful and interesting properties, like self-healing and somewhat painless scaling.

The main configuration unit of Kubernetes is a manifest file. Manifests declare resources inside the cluster and describe the wanted state for these resources.

Under the hood Kubernetes is a stack of software run on compute substrate. Each copy of this software running on a machine is called a node and these nodes together form a cluster. Cluster state is kept in the master nodes distributed key-value-store, etcd [14], and applied to nodes via kubelet agent running on the nodes. Master nodes with their services and kubelets actuating the state changes on nodes create the Kubernetes control plane [15]. This control plane is responsible for keeping the cluster state (workloads and their configuration etc.) in accordance of state described in the master node's etcd. Container workloads (eg. production software and its components) are run on worker nodes and are orchestrated by the control plane. These components of Kubernetes and their location in the node types are illustrated visually in Figure 2.3.

Further analysis of Kubernetes compotents and their functions is explained in the Kubernetes documentation [16].

Kubernetes is based on the experience of Google engineers about how to orchestrate containers at the scale of Google during the last decade [17].

Multiple cloud providers provide managed Kubernetes clusters [18]. This provides their customers a cluster without having to take care of the configuration of underlying nodes and updating Kubernetes versions. This is useful, especially to smaller companies, because running self-managed Kubernetes in a secure and reliable way requires intricate knowledge about Kubernetes, which seems to be in short supply at the moment.

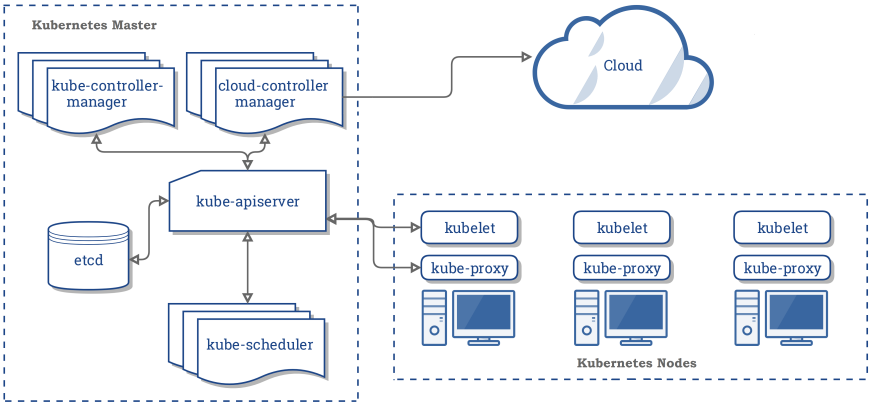

Figure 2.3: Components of Kubernetes. Illustration from Kubernetes documentation [16]

Chapter 3

Case study

In this chapter Kaiku Health is introduced as a company, the development flow in use at Kaiku Health is examined and the two deployment systems in use at Kaiku Health are analyzed.

3.1 Kaiku Health

Kaiku Health is a Finnish healthcare software company based in Helsinki and founded in 2012. Kaiku employs about 30-40 people and focuses on bringing digital health interventions to all cancer patients. The main product of Kaiku Health is Kaiku, a digital therapeutics platform used in communication between patients and healthcare personnel. Kaiku is also used to prompt the patient for their current symptoms and by their healthcare personnel to guide the patient based on that symptom data. A picture about how Kaiku looks like to patients can be seen in Figure 3.1. This view shows the basic patient landing page of the application. From this view the patient can access the most important features of Kaiku (Conversations and Questionnaires) easily. Kaiku also interacts with users by sending email notifications about messages and questionnaires that should be filled.

There is some evidence in literature that this kind of activity (Electrical Patient Reported Outcomes, ePRO) increases patient survival rate and

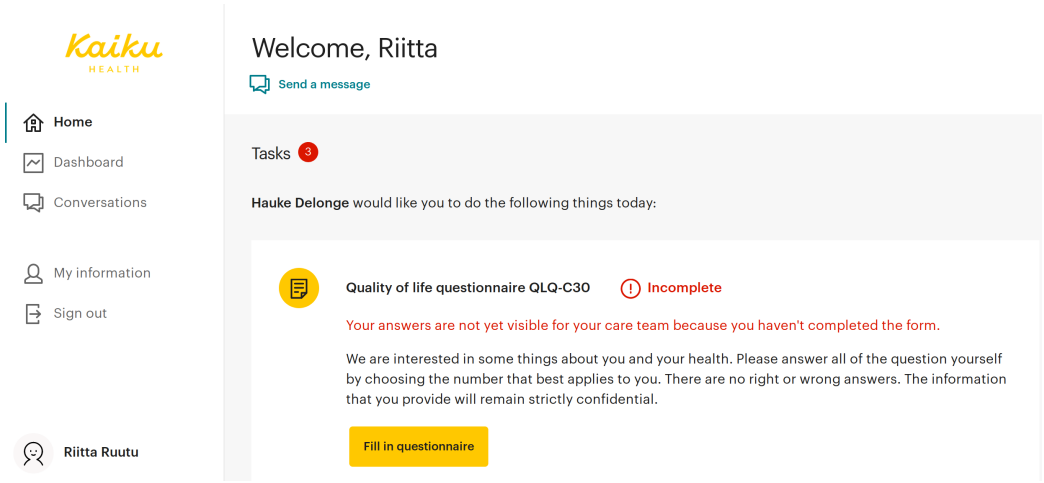

Figure 3.1: A picture of a patient-view in Kaiku. Using this view the patient is able to contact her care team ("Conversations") and answer questionnaires ("Fill questionnaire").

decreases Emergency Room visits among patients [19].

Nurses and doctors have similar views tailored for their use cases. These views can be configured based on customer requirements.

Kaiku currently is hosted on virtual machines that are provisioned from multiple cloud service providers. Kaiku operates about 20-40 virtual machines that host the software solution around the world. These servers are operated in a safe manner and fulfill the special needs of each customer and their respective data protection rules and regulations of their country.

3.2 Kaiku's Current Deployment and Runtime System

This subchapter familiarizes the reader with the Kaiku development and deployment systems. Kaiku customer tenancy model is introduced. A quick overview is presented of how code changes made by a developer proceed through the CI/CD pipeline.

Finally the two different deployment systems in use at Kaiku are introduced. These deployment systems and measuring their performance is the main subject of this thesis.

3.2.1 Kaiku Tenancy Model

Each customer organization usually has one Kaiku instance("site"), which has it's own configuration, database and ruby-processes. When a new customer organization start using Kaiku, this configuration is generated and a new instance is deployed for the customer organization.

3.2.2 Development Flow

Kaiku Health uses a trunk-based development flow. Developers work on their own local copy of the software and push their changes to a remote repository. CI-server then pulls a copy of these changes and runs integration tests and other automated checks against that copy. Development branches are short-lived and merge conflicts are rare.

Once the developer is satisfied with the changes, she creates a Pull Request (PR) about the changes and asks another developer to be the reviewer of this PR. The PR can't be merged before all the required automated tests have passed and at least one other developer has approved the changes. This whole development flow is illustrated in Figure 3.2.

3.2.3 Current Runtime System

Currently Kaiku is run as a Ruby on Rails (ROR) [20] application running on nginx [21] + Phusion Passenger application server [22]. The ROR application uses a Postgresql-database [23] to store state. This whole stack is running on a VM provided by a cloud service provider. Kaiku also uses some external API dependencies for object storage and error reporting, but those are omitted from analysis for brevity. The whole stack is illustrated in Figure 3.3.

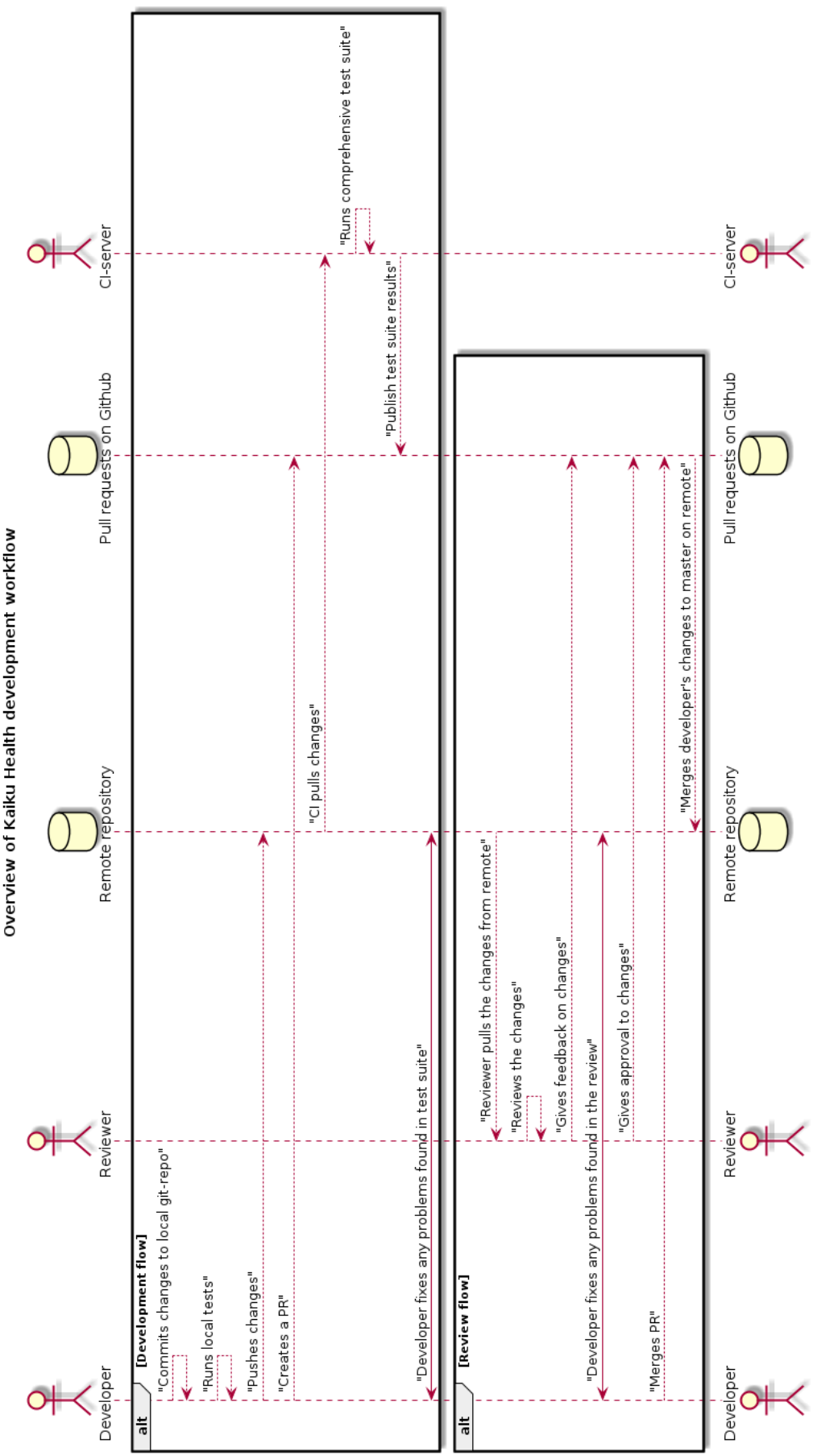

Figure 3.2: A simplified sequence diagram of Kaiku's development flow showing how changes in the codebase progress through the CI-pipeline and end up merged in the master software development branch

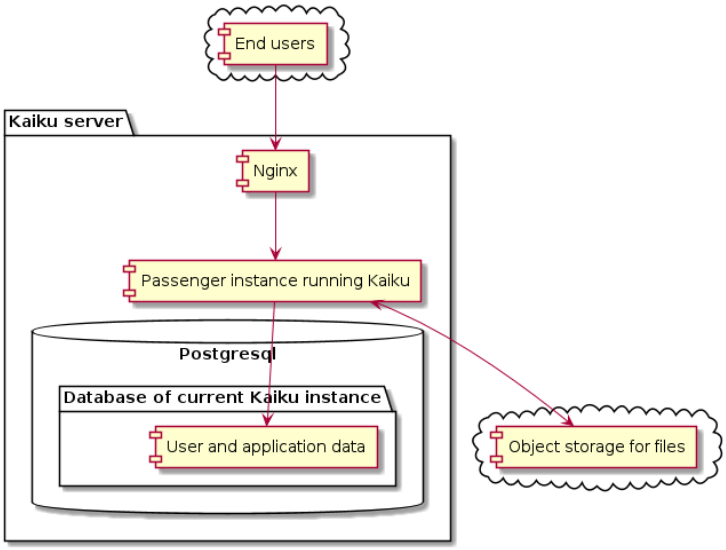

Figure 3.3: A diagram of the old runtime system of Kaiku

3.2.4 Current Deployment System

The deployment process is handled by a developer pair who is responsible for internal tech support at the moment. The tech support role is on rotation and changes after two weeks. Kaiku has an internal agreement that deploys should happen twice a week.

Deployment begins by tech support announcing her plans to deploy on the chatroom of developers. This is done in order to make sure that any features that might be promised to the customers will be merged before deployment begins. Once the announcement is done and the developer in tech support is ready, she will tag a release in Git and start the deploy script that imperatively deploys all sites.

Each customer site is deployed in a linear fashion and by the end of the deployment, all Kaiku production instances will be running the new version. If any errors occur during the deployment of a new version, a developer assigned to tech support will investigate and correct those errors. This deployment flow is visualized with a sequence diagram in Figure 3.4.

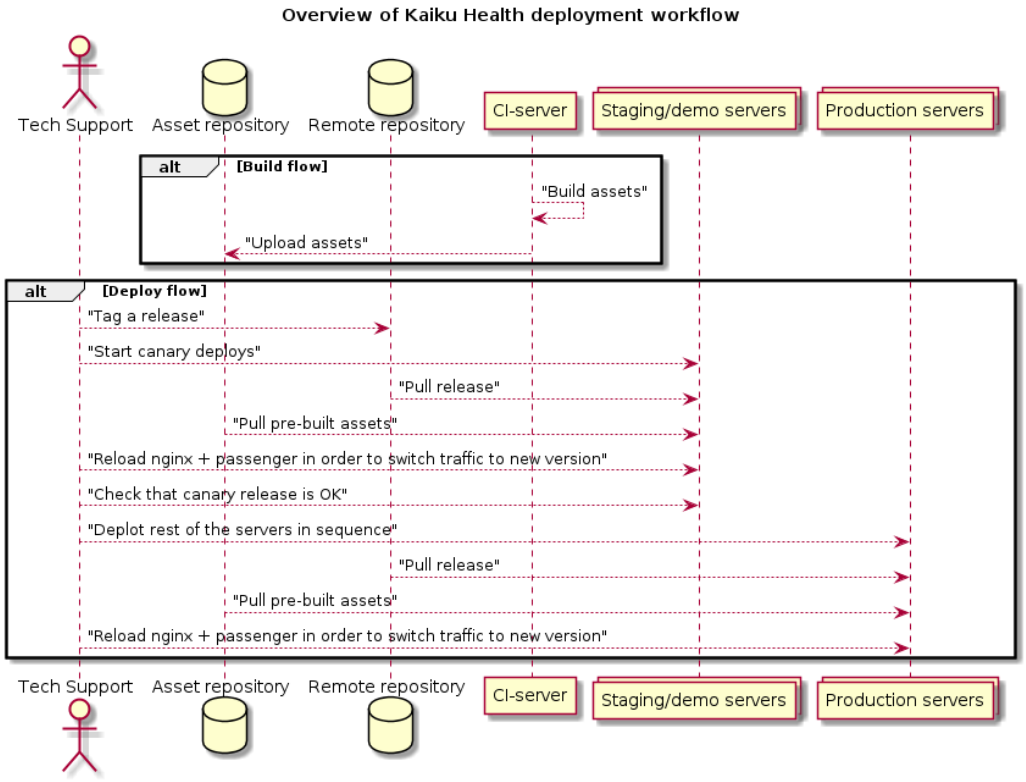

Figure 3.4: A simplified sequence diagram of Kaiku's deployment flow on the old deployment system. Developers assigned to tech support run scripts that actuate the process and monitor it's outcome.

3.2.5 New Runtime System

The new runtime system is a set of Kubernetes clusters provided by a cloud service provider. Kaiku application is packaged in to a Docker container and run as workload on a Kubernetes cluster. Traffic to the Kaiku instance is routed via Google cloud load balancer, which forwards it to the cluster ingress. Inside the cluster routing is handled by Kubernetes using service-resources. The relational database service is provided by the cloud provider. This stack is illustrated with a diagram in figure 3.5.

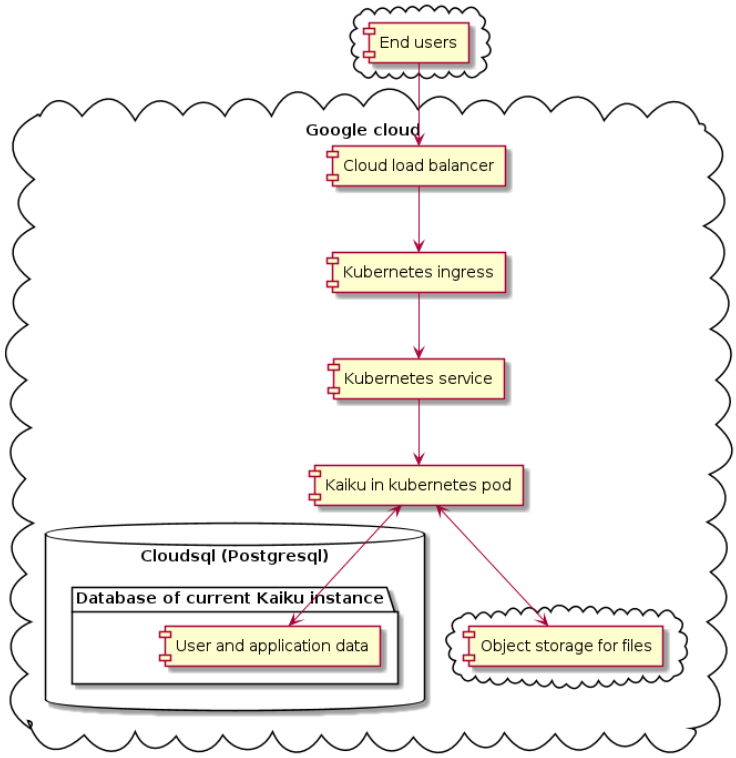

Figure 3.5: A diagram of the new runtime system of Kaiku on Kubernetes in Google Cloud.

3.2.6 New Deployment System

The new deployment system is based on Linux containers, specifically on Docker. These containers are run and managed in a Kubernetes cluster managed by Google. Kubernetes was introduced earlier in 2.4.4.

The biggest philosophical change between Kubernetes and the old system is that Kubernetes is declarative in nature, whereas the old system is imperative. In practice this means that the old system runs commands against the VM's based on deployment automation scripts, and the new system applies manifests to the cluster and the cluster configures itself based on those manifests.

In Kaiku's case this means that these manifests are generated based on configuration and code in the product Git repository. This generation is done with Ruby scripts built in-house.

The new deployment is started manually by tech support tagging a release. Once the tagging is done, tech support runs the manifest generation scripts and applies them to all clusters that are part of the deployment. Clusters then enact the changes defined in the manifests and tech support can monitor the changes with a logging script. The sequence of actions and their targets in the new deployment system are visualized with a sequence diagram in Figure 3.6.

The new system deploys each customer site parallel to the other sites if the underlying Kubernetes cluster has enough spare capacity to create all required resources.

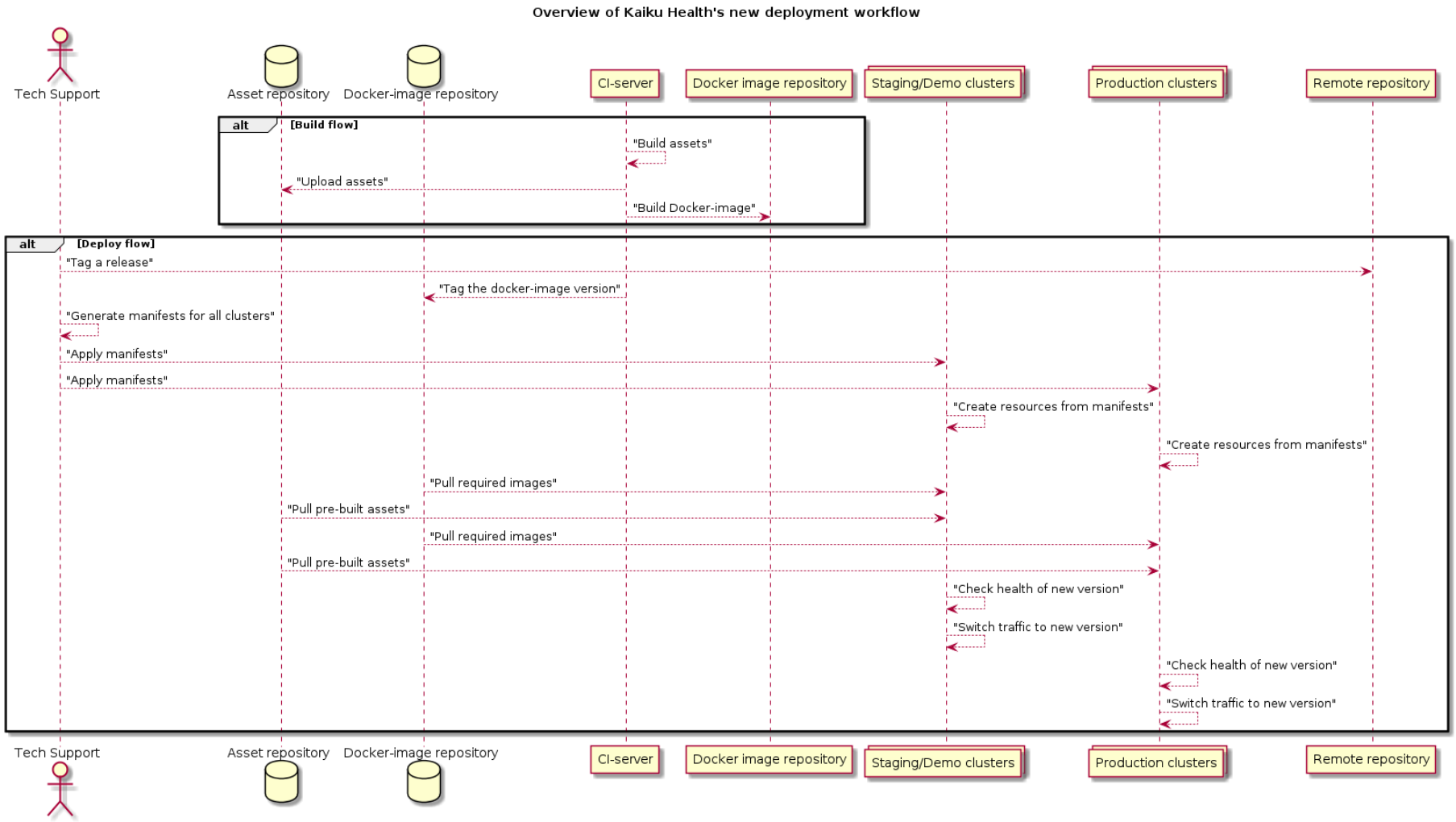

Figure 3.6: A simplified sequence diagram of Kaiku's deployment flow on the new deployment system. This diagram shows how a new version of Kaiku is deployed to all Kubernetes environments by developer doing tech support.

Chapter 4

Methodology

This chapter introduces the methodology used for measuring the deployment systems. Deployment systems and their instrumentation are explained to the reader. Research questions are introduced, methods for answering each of those questions are chosen and null hypothesis and alternative hypothesis are formulated for each question.

This thesis is a case study with measurements done on a live production system. The thesis is written from a pragmatist-positivistic viewpoint, more specifically meaning that efforts will be made to control the test setup in order to make it robust, but it is acknowledged that this is not a fully controlled study. The positivistic-pragmatic viewpoint was chosen based on information presented by Runeson et al. [24]

The specific challenge with measuring the deployment at Kaiku Health is that completely separating the deployment systems is unfeasible as the whole version deployment is done on a live production system that is in use.

There is some potential for confounding variables to influence the results of the study. Examples in this case would be time of day of the deployment and the developer doing deployment. Both could affect the deployment duration, time of day via difference of resource utilization of compute resources used at deploy and developer doing the deployment via the different local configuration on her machine and it's internet connection quality.

No control for confounding variables will be done, as time resources to do it were not available. Some of the data for controlling confounding variables was collected and is included in the anonymized dataset. Analysis of possibility of controlling confounding variables is based on Pourhoseingholi et al. review of controlling methods [25].

4.1 Research setup

Both deployment systems will be instrumented to push events to a logging service. At the end of the measurement period those events will be analyzed, relevant deployment data will be visualized and relevant statistics will be calculated from the data.

4.1.1 Systems Under Study

The whole deployment of Kaiku is a deployment of a new version of Kaiku to all Kaiku sites. In this thesis, this deployment system is named as **System I**. A single site deployment consists of deploying a new version to a single customer site and is considered as a different deployment system with name **System II**. One **System I** deployment consists of multiple **System II** deployments. This is illustrated with a graph of datasets, systems and their relationships in Figure 4.1.

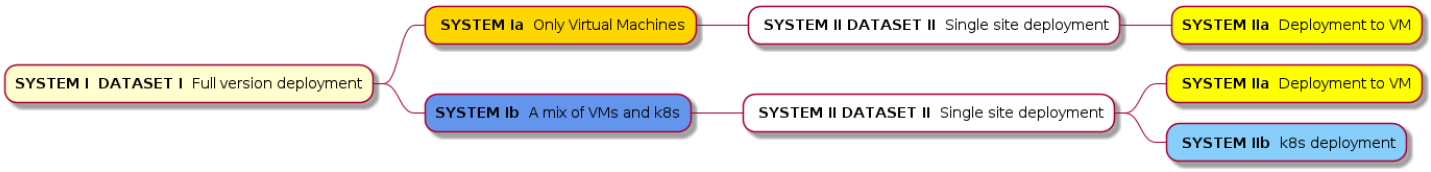

Figure 4.1: Systems under study and their relation to datasets visualized as a graph. Kubernetes is abbreviated as "k8s" in order to keep the graph readable.

Measurement of the case where the new deployment system is the only

Dataset	Measurement target	Systems under study
Dataset I	Whole version deploy to all sites	**System Ia**: VM-only whole version deploy **System Ib**: VMs + Kubernetes mixed whole version deploy
Dataset II	Version deploy to single site	**System IIa**: Site deploy on VM **System IIb**: Site deploy on k8s

Table 4.1: A table about systems under study and their relation to datasets

system in use is impossible at the moment, as the migration of most of the sites is still in progress. But data gathering about the complete deployment with the old system (before any sites were migrated, **System Ia**) and about the hybrid case, when both system are in use at the same time (**System Ib**) is possible and is the way that this thesis approaches the problem.

The relative amount of Kubernetes and VM sites during the measurements is visualized in Figure 4.2. When calculating full deployment metrics, all the deployments with only VMs are counted as part of the **System Ia** and all the full deployments with at least one Kubernetes site as part of **System Ib**. This will not give us a complete picture of the performance of the only Kubernetes-only system, but it should allow us see how the charasteristics of the system change when more sites are migrated to the new system.

When counting **System II** deployments any site deployed on a virtual machine will be counted as **System IIa** and any site deployed on Kubernetes as **System IIb**.

4.1.2 Instrumentation of The Deployment System

The deployment system of Kaiku has been instrumented by adding event-sending logic to the scripts that handle deployment on **System I** and **System II**. These events are sent when any site deployment starts (**System**

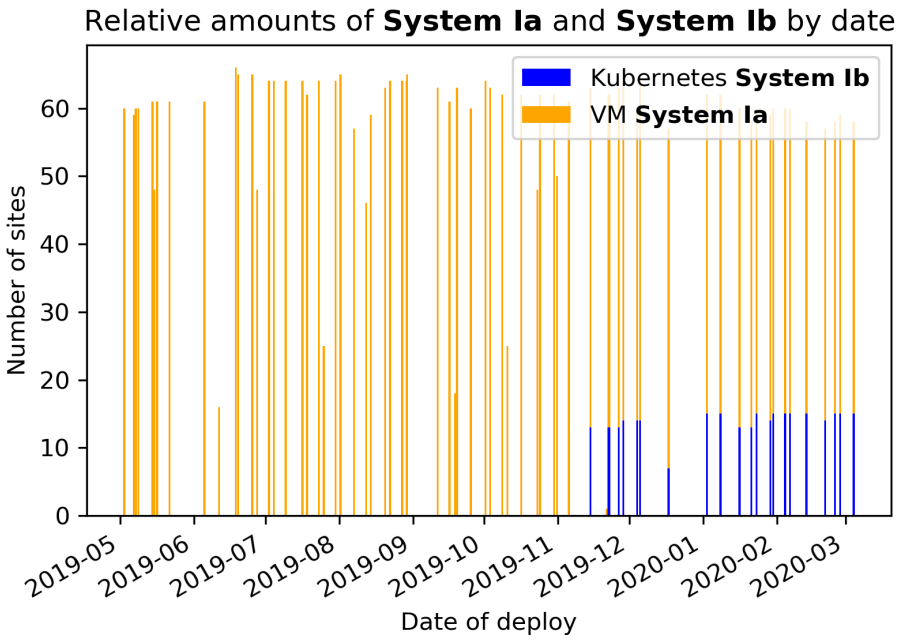

Figure 4.2: A stacked bar graph of the relative amount of **System Ia** and **System Ib** based sites in a single full deployment by date

II) or finishes and when the whole version deployment process starts or finishes (**System I**). The events contain site name, event name and deployment system type. Events are sent to Kaiku's centralized logging platform (Elasticsearch [26]), which adds timestamp and other relevant metadata to the event in question.

4.1.3 Data Analysis Tools

The deployment events data was analyzed with Python [27]. For statistical data analysis, a python library called SciPy [28] was used. A Jupyter Notebook [29] was created for quick iteration and easier interoperatibility. The Jupyter Notebook and anonymized data will be published for other researchers to tinker with. Statistical framework for this thesis is the frequentist statistical framework because author is familiar with it. The commonly accepted threshold of 0.05 for statistical signicance ($\alpha = 0.05$) was used.

4.2 Research Questions

RQ1:	Is there a statistically significant difference in the whole version deployment duration between the systems?
RQ2:	Is there a statistically significant difference in the site-normalized whole version deployment duration between the systems?
RQ3:	Is there a statistically significant difference in the single site deployment duration between the systems?
RQ4:	Is there a statistically significant difference in how the site count affects the whole system deployment time between systems?
RQ5:	Is there a statistically significant difference in the frequency of whole system deployments between systems?

RQ1 can be investigated with descriptive statistics and statistical tests based on the data acquired from deployment instrumentation. More specifically normality of full deployment events from both systems will be checked, then descriptive statistics will be calculated from the data and then statistical tests to check if those differences are statistically significant will be done. Either Student's T-test or Welch's T-test will be used. The choice between the tests will be based on the differences in variances of the populations in question [30]. Both statistical tests require the data to be normally distributed, which means that we have to check the distribution of the data before accepting the test in to use. Distribution checking will be done by visually checking the histogram.

RQ2 can be investigated in the same manner as RQ1, but the data will be normalized with the site count before the statistical analysis will be done.

RQ3 can be investigated by using the data from single site deployment events. Descriptive statistics can be calculated from the data and statistical test can be done to see if the differences are statistically significant.

RQ4 can be investigated by doing linear regression on deployment data with regards of site count of both systems and then comparing the regression parameters. This gives us estimates about the relative performance of the systems as a function of the site count.

RQ5 might end dissolving, as the product being deployed is the same in both deployment methods and thus it's release cycle is the same. The deployment process in use at Kaiku generally results in about two full deploys in a week, and both systems are deployed in one full deploy. RQ5 will be included still, because it is possible that our deployments are not always in sync. RQ5 can be investigated by calculating the time delta between deployments in both systems and then analyzing these time deltas with descriptive statistics and statistical tests.

4.3 Hypotheses

Our null hypothesis H_0 for all research questions is that there is no statistically significant difference between the systems under comparison.

RQ1. *Is there a statistically significant difference in the whole version deployment duration between the systems?*

The alternative hypothesis H_1 is that deploying Kaiku's product on Kubernetes (**System Ia**) should be faster, as each single site deployment is running in parallel, if the cluster has enough capacity available. The old system (**System Ib**) runs deployment in a serial fashion and hence should be slower.

This question will be answered using **Dataset I**.

RQ2. *Is there a statistically significant difference in the site-normalized whole version deployment duration between the systems?*

The alternative hypothesis H_1 for the site-normalized deployment duration is that the Kubernetes based system (**System Ia**) should be faster per site, because of the parallelization of the deployment.

This question will be answered using **Dataset I**.

RQ3. *Is there a statistically significant difference in the single site deployment duration between the systems?*

The alternative hypothesis H_1 for single site deployment is that the VM based system (**System IIa**) should be faster, as it has most of the required state and dependencies already in place. It is possible, that the Kubernetes based system (**System IIb**) is faster, if image caching can be used effectively.

This question will be answered using **Dataset II**.

RQ4. *Is there a statistically significant difference in how the site count affects the whole system deployment time between systems?*

The alternative hypothesis H_1 for scaling of the systems is that the Kubernetes-based system (**System Ib**) should scale better. Scaling in this case is viewed from the viewpoint of adding more customer sites to the deployment. Kubernetes deploys sites in parallel, so the deployment time should increase sub-linearily. The old system (**System Ia**) is deploying sites one-at-a-time, which will probably mean linear scaling in relation to number of sites being deployed.

This question will be answered using **Dataset I**.

RQ5. *Is there a statistically significant difference in the frequency of whole system deployments between systems?*

The alternative hypothesis H_1 for deployment frequency is that it would be different. This is improbable as the deployment systems are still tied together.

This question will be answered using **Dataset I**.

4.4 Outlier Removal

In order to use the toolbox of statistical analysis and tests, the data under analysis has to be well formed and follow certain guidelines. In this thesis, following rules will be used to filter outliers and malformed data.

Before any outliers are removed, all the malformed data is removed. Malformation in this thesis is a case when one can't establish the start and end events for a single site deployment (**Dataset II**) or whole version deployment (**Dataset I**). All deployments with internal inconsistencies, like different site-counts by different counting methods, will also be removed as malformed data.

System I has some double deployments, as some sites are used as "canary"-deployments. These canary deployments allow the tech support to manually verify that the current version that is being deployed is not obviously broken before deploying it to all customer sites. These deployments

are double-counted, as those sites are deployed twice and hence these double deployments are not counted as outliers.

Outlier removal for **Dataset I** will be done by filtering any full deploys that last longer than 150 minutes, are shorter than 40 minutes or contain less than 40 sites or more than 70 sites. Deployments that don't fill these assumptions are probably not real deployments, but someone using the deployment tool to run partial deployments for a special single site deployments, or using the tool to set site settings for all sites.

Outlier removal for **Dataset II** will be done by filtering any single site deploy that differs more than 3 standard deviations from the mean. These deployments are some kinds of anomalies that don't reflect the normal performance of the system.

Chapter 5

Results

This chapter introduces the results of the measurements made on **System I** and **System II**. After a quick glance on the results per system, the system are compared using measured variables and statistical analysis and tests comparing the systems are reviewed. In the end the evaluation of the hypotheses for the research questions is done and those hypotheses are either accepted or rejected based on the evidence from the instrumentation data.

All of the plots calculated for the data-analysis are color-coded in order to clarify which system the plot or datapoint is representing. All datapoints from the old system (**System IIa**, **System Ia**) are colored orange and datapoints from the new system (**System IIb**) are blue. Plots containing mixed data (**System Ib**) are colored brown.

5.1 Results of Outlier Removal

The procedure and basis for outlier removal is explained in detail in section 4.4.

Results of outlier removal for **System I** can be seen in Figure 5.1. Results of outlier removal for **System II** can be seen in Figure 5.2. Most of the data from both system fits inside our criteria and looks reasonably normally distributed, which is a pre-requisite for the statistical methods we use.

Normality requirement for our statistical tests was introduced in 4.2

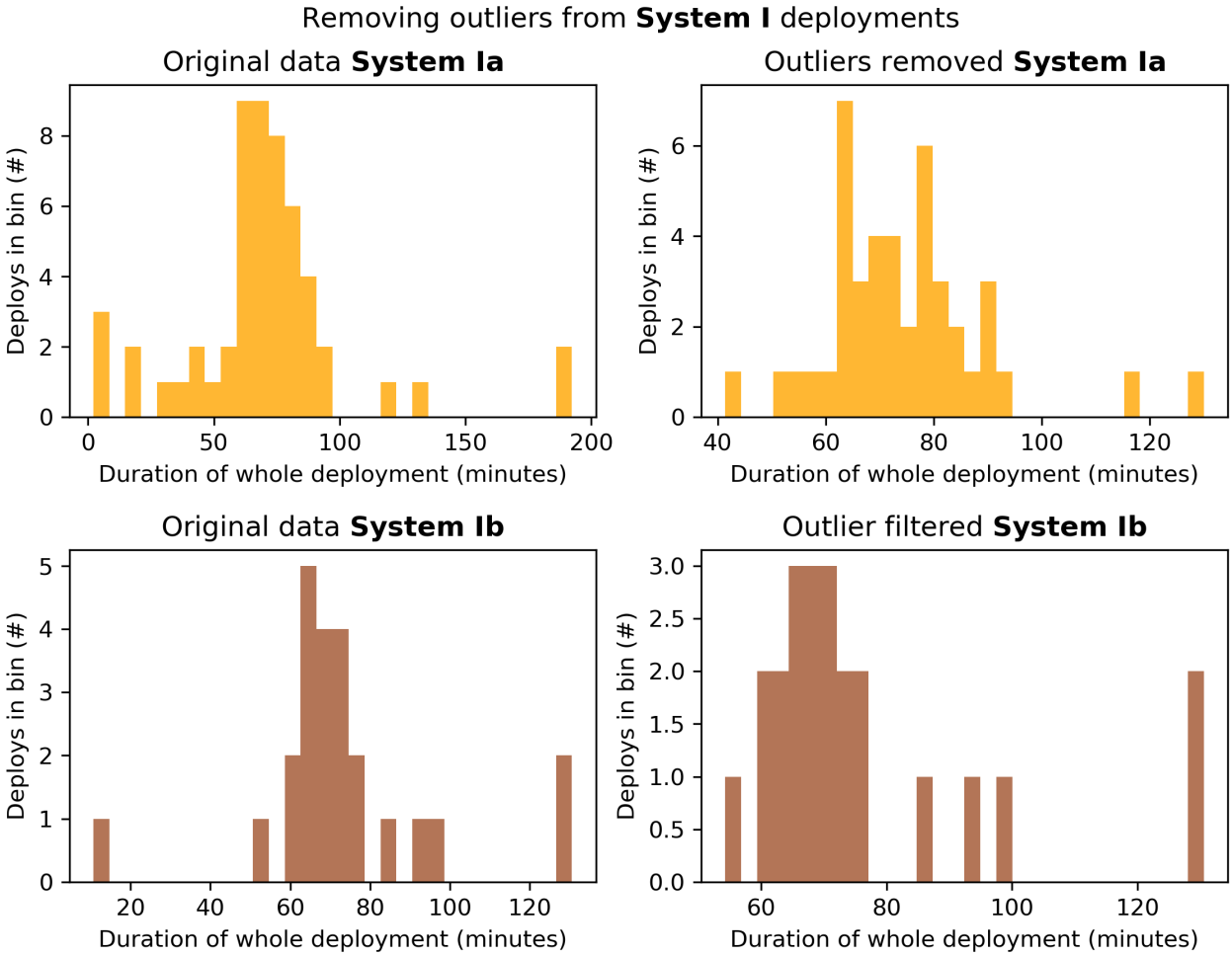

Figure 5.1: A collection of plots showing outlier removal and it's effect on the histogram of whole system deployment in **System I**

5.2 Whole Version Deployment Performance, System I

Whole version deployment of Kaiku takes about 70 minutes, as one can see from Figure 5.3.

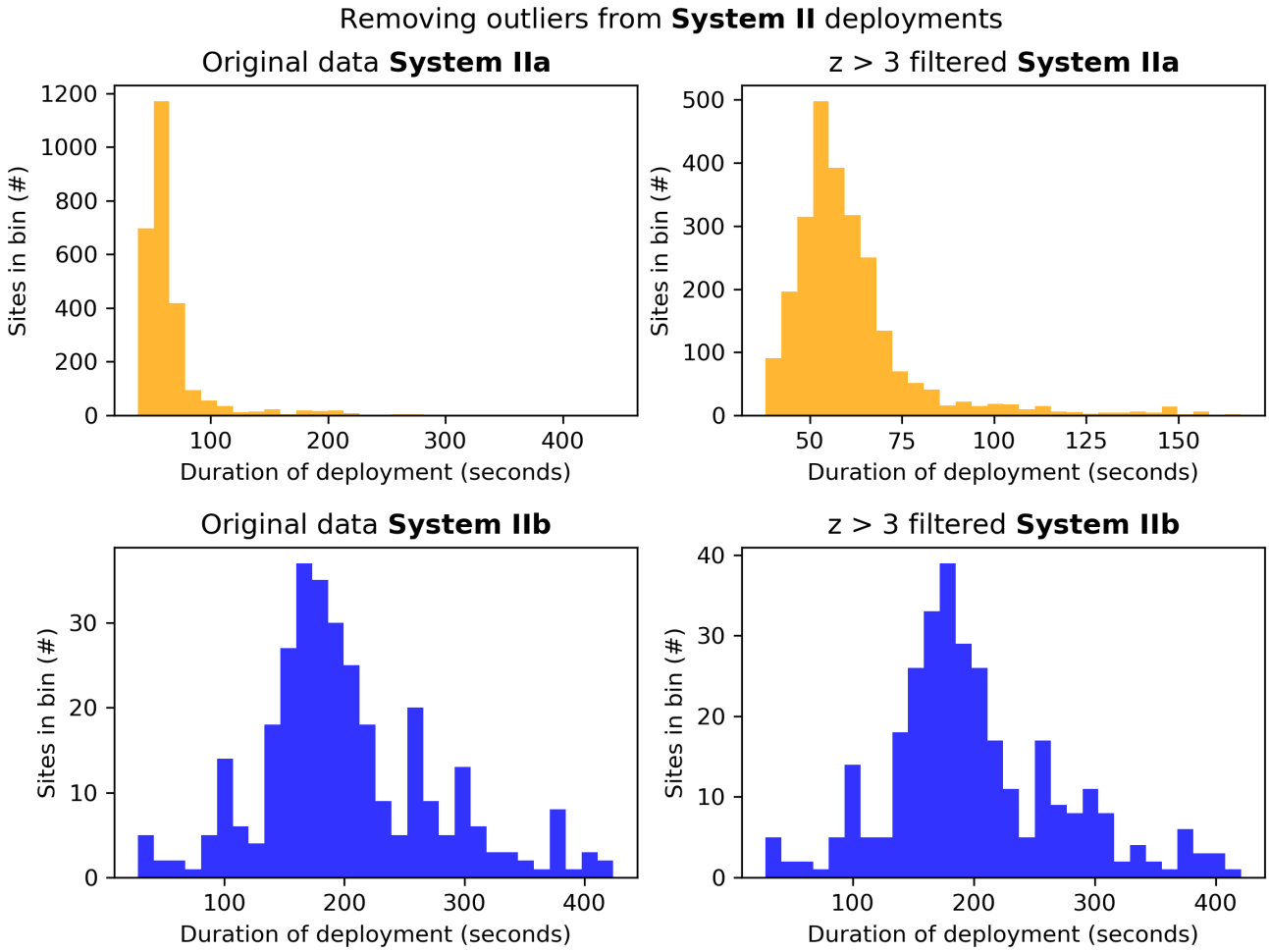

Figure 5.2: A collection of plots showing outlier removal and it's effect on the histogram of single site deployment in **System II**. All data with deviance from the mean (z-score) greater that 3 standard distributions was discarded as outlier.

A more detailed view with the amount of sites, deployment system in use and deployment duration can be seen in Figure 5.4. From this figure one can see that both **System I** deploy systems seem to have quite similar means, but variance seems bigger in **System Ia**.

The site-normalized deployment duration is relevant from the business viewpoint of a company in midst of growth, as growing means adding more

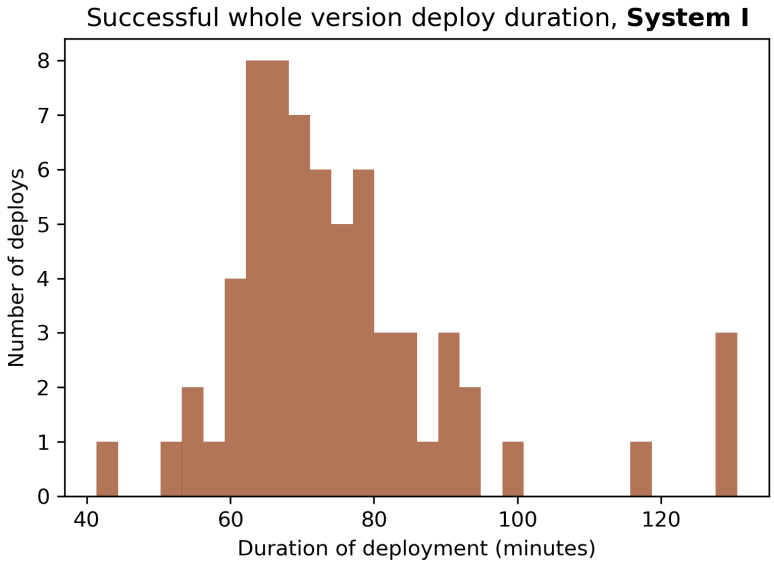

Figure 5.3: Histogram of all **System I** whole version deployment durations

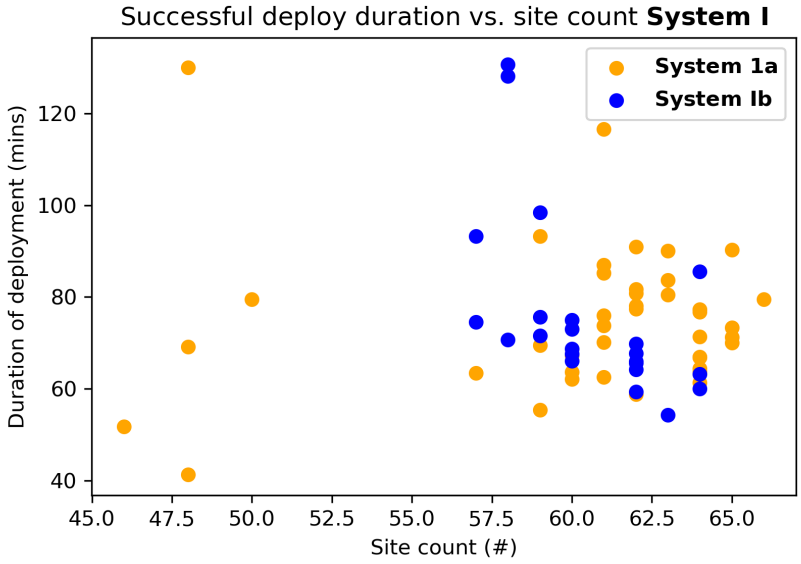

Figure 5.4: Scatterplot of whole version deployment, **System I**

customers and hence more sites to Kaiku. The site normalized view to deployment systems can be found from Figure 5.5. Scaling of Kaiku with new

sites seems about linear, as the site-normalized deployment time does not seem to increase with the site count.

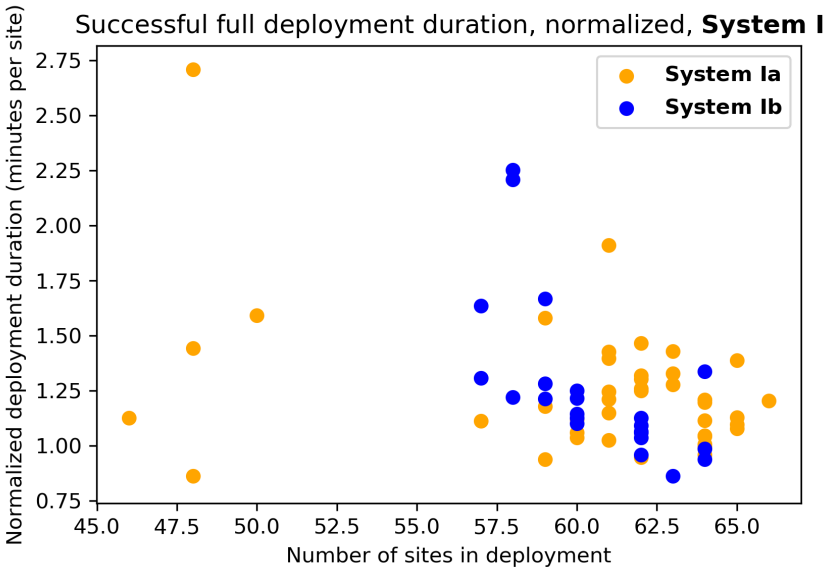

Figure 5.5: Site-normalized deployment duration in the two systems

5.3 Single Site Deployment Performance, System II

Histograms of single site deployment performance can be found from figures 5.6 and 5.7. Data in both histograms looks reasonably normally distributed, but **System IIa** data has a bit longer tail, which could point towards the data being beta-distributed. This should not cause problems with data-analysis, as most samples are nicely clustered.

A deployment of Kaiku to a single site takes about 50 seconds in **System IIa** and about 180 seconds in **System IIb**.

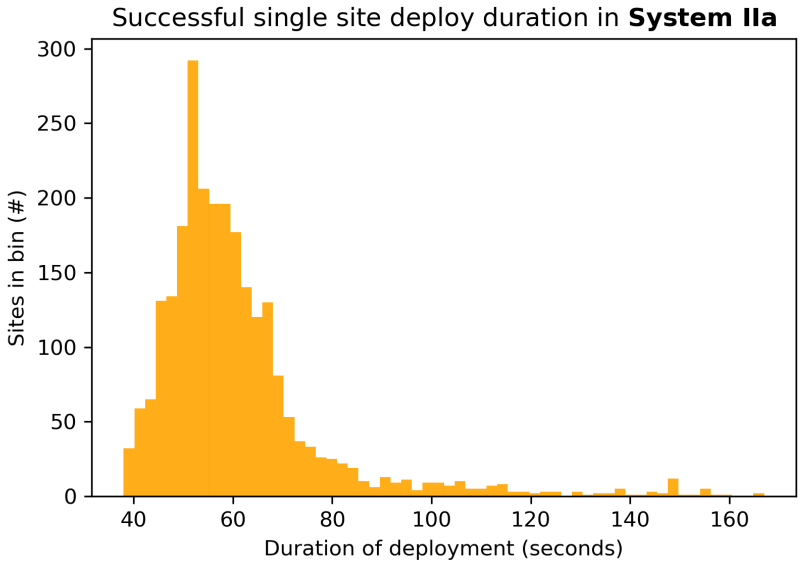

Figure 5.6: Histogram of all **System IIa** single site deploy durations, outliers removed

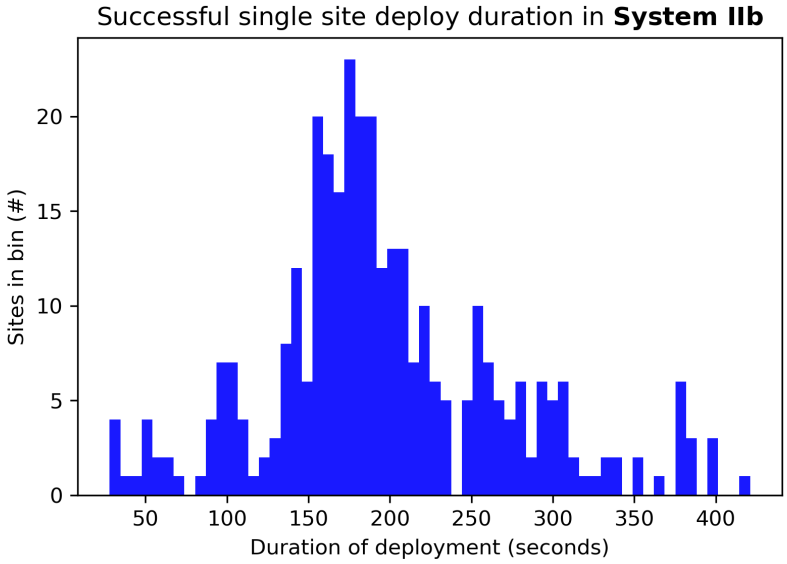

Figure 5.7: Histogram of all **System IIb** single site deploy durations, outliers removed

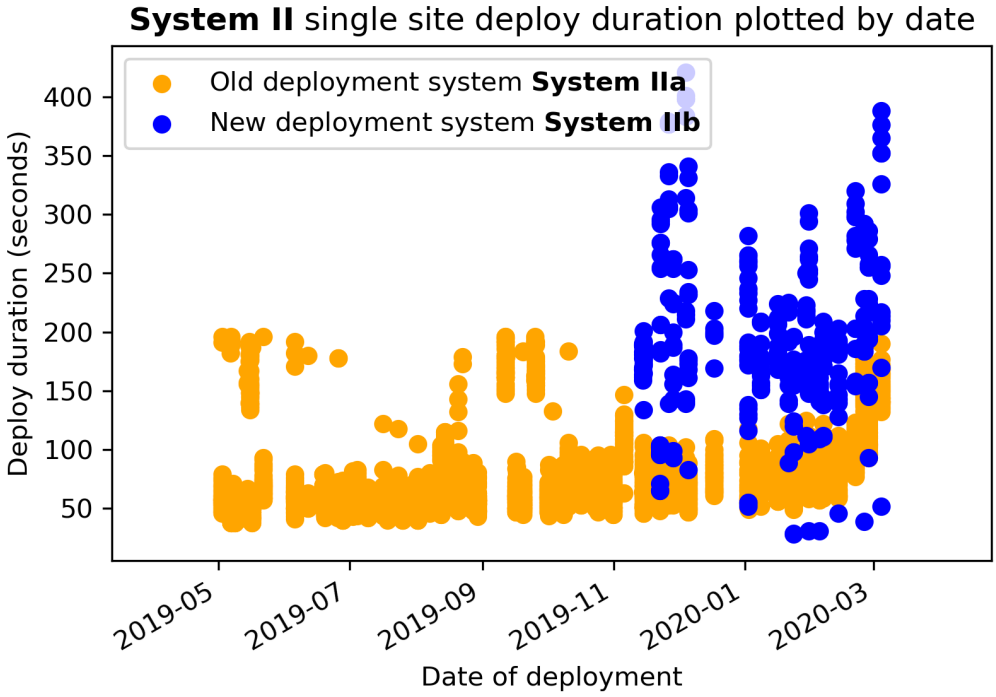

Figure 5.8: Scatter plot of single site deployment duration in **System II**, outlier removed

5.4 Statistical Analysis

Methodology for the statistical analysis was defined in 4.1.3.

Outliers were removed from data before any analysis towards answering the research question was done. Results of outlier removal can be found from 5.1.

5.4.1 RQ1 Is There a Difference in The Whole Deployment Duration Between The Systems?

The null hypothesis H_0 is that the two deploy systems have no statistical difference in their deployment duration or single site deploy deploy duration. The alternative hypothesis H_1 is that the systems differ in their deployment

Item		
System	Variable	Value
Whole deploy duration in **System Ia**	Number of samples	43.00
	Mean	74.74
	Variance	240.09
Whole deploy duration in **System Ib**	Number of samples	23.00
	Mean	76.04
	Variance	385.56
Welch t-test, single site deploy **System Ia** vs. **System Ib**	Test statistic	-0.28
	p-value	0.78

Table 5.1: Descriptive statistics and Welch t-test results about RQ1

duration. α was chosen to be 0.05 as described in section 4.1.3. Welch's t-test was chosen as a statistical test because the variances between systems differed. If systems differ statistically significantly the faster system is deduced based on mean value.

Each deployment sample is independent from other samples and an assumption that the samples are identically distributed on normal distribution is made based on the visual inspection of the data.

Mean deployment duration on **System Ib** is within two minutes of the mean duration on **System Ia**. Variances are quite different between systems.

From the statistical analysis presented in Table 5.1 it is concluded that the alternative hypothesis H_1 that the new system is faster can't be accepted, as the p-value of 0.78 is over 0.05.

Item		
System	Variable	Value
Whole deployment normalized by site count **System Ia**	Number of samples	43.00
	Mean (minutes)	1.24
	Variance	0.09
Whole deployment normalized by site count **System Ib**	Number of samples	23.00
	Mean (minutes)	1.26
	Variance	0.13
Welch t-test, whole deploy normalized **System Ia** vs. **System Ib**	Test statistic	-0.26
	p-value	0.79

Table 5.2: Descriptive statistics and Welch t-test results about RQ2

5.4.2 RQ2 Is There a Difference in The Site-normalized Whole Deployment Duration Between The Systems?

To answer this question, the deployment duration has to be normalized with regards to the site count and thus the deployment speed made more comparable across time. This is advisable as the site count changes between deployments and site-count is probably the biggest parameter determining the deployment duration.

As one can see from the Table 5.2, both systems have means that are really close to each other. No significant difference can be made between them as the p-value of 0.79 is over 0.05 choosen in methodology. Thus the null hypothesis H_0 holds for RQ2.

Item		
System	Variable	Value
Single site deploy duration in the **System IIa**	Number of samples	2525.00
	Mean (seconds)	60.94
	Variance	309.06
Single site deploy duration in the **System IIb**	Number of samples	318.00
	Mean (seconds)	200.32
	Variance	5348.25
Welch t-test, single site deploy **System IIa** vs. **System IIb**	Test statistic	- 32.31
	p-value	0.00

Table 5.3: Descriptive statistics and Welch t-test results about RQ3

5.4.3 RQ3 Is There A Difference In The Single Site Deployment Duration Between The Systems?

Results of the statistical analysis are presented in the Table 5.3.

Single site deployment ended up being a lot slower in the new system (**System IIb**). Means differ by of about a hundred seconds to the advantage of the old system (**System IIa**). This is a somewhat counter-intuitive situation, as the whole system deployment duration not statistically different, as was found in the analysis of RQ2. This is probably because of the parallel nature of deployments in the **System Ib**. The finding is statistically significant with p-value clearly under the p-value choosen in methodology. Alternative hypothesis H_1 is accepted.

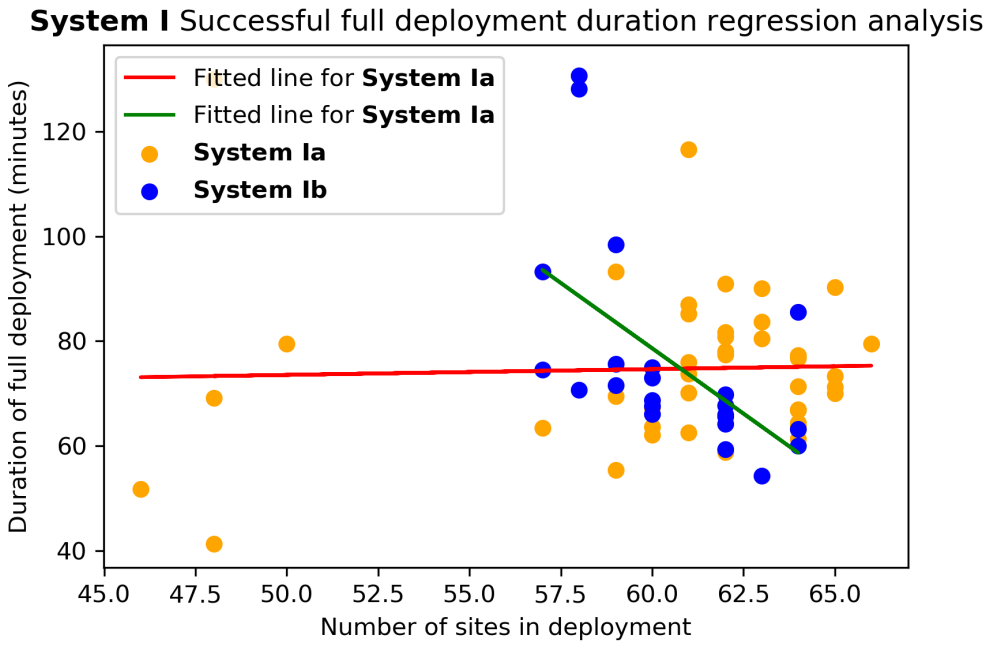

Figure 5.9: Linear regression comparison of the two deployment systems

5.4.4 RQ4 How Does the Scale Effect the Whole System Deployment Time in Both Systems?

This research question was investigated using linear regression implemented using numpy stats library [31]. Results of the Least squares regression are presented graphically in Figure 5.9 and numerically in 5.4.

Our linear regression models are quite ill-fitting to the data, with high standard error values. R-value of **System Ia** model is quite low, and thus the fit between model and data is not good. The model for **System IIb** fares a bit better.

P-value for regression model of **System Ia** 0.82 is over the 0.05 choosen in the methodology, hence the model us not statistically significant. Regression model for **System Ib** is statistically significant with p-value of 0.01.

As only have one statistically significant model was found, a rigorous comparison between the models can't be done. Alternative hypothesis H_1 can't

System	Variable	Value
Linear regression with least squares, **System Ia**	Slope	0.11
	Intercept	68.05
	r-value	0.04
	p-value	0.82
	std error	0.48
Linear regression with least squares, **System Ib**	Slope	-4.98
	Intercept	377.53
	r-value	-0.56
	p-value	0.01
	std error	1.60

Table 5.4: Numerical results of linear regression analysis for RQ4

be accepted. But statistically significant model for **System Ib** is promising for the prospect of improving the deployment speed with Kubernetes.

5.4.5 RQ5 Is There Difference in the Frequency of Whole System Deployments Between Systems?

Date differences between successful deployments in both systems were calculated. Histograms of the deployment date deltas can be seen in Figure 5.10. The number of samples for the new system is quite low and thus it is quite hard to see if the time deltas are normally distributed.

Results of the statistical analysis are presented in Table 5.5. Based on the analysis, it looks like our null hypothesis holds for RQ5, as the p-value of our Welch's T-test is 0.58, which is over 0.05.

Generally data for RQ5 seems quite non-normal and hence this analysis is not on firm standing.

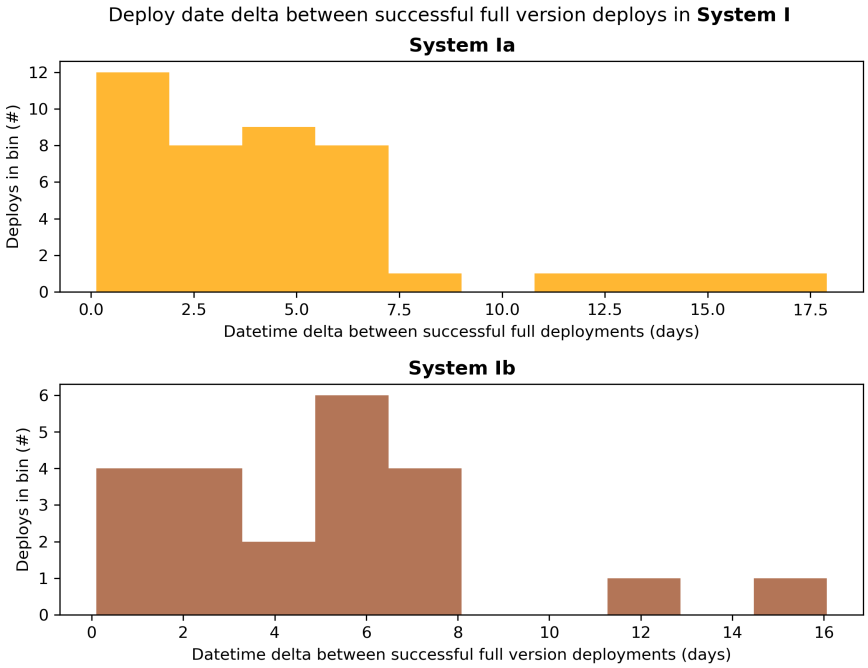

Figure 5.10: A histogram of datetime deltas between successful deployments in both systems

System	Variable	Value
Time delta between successful deployments, **System Ia**	Number of samples	42.00
	Mean (day)	4.45
	Variance	16.45
Time delta between successful deployments, **System Ib**	Number of samples	22.00
	Mean (days)	5.03
	Variance	15.01
Welch t-test, delta between successful deployments **System Ia** vs. **System Ib**	Test statistic	-0.56
	p-value	0.58

Table 5.5: Descriptive statistics and Welch t-test results about RQ5

5.5 Answers to Research Questions

RQ	Description	System	H_1 Accepted?
RQ1:	Is there a statistically significant difference in the whole version deployment duration between the systems?	**System I**	No
RQ2:	Is there a statistically significant difference in the site-normalized whole version deployment duration between the systems?	**System I**	No
RQ3:	Is there a statistically significant difference in the single site deployment duration between the systems?	**System II**	Yes
RQ4:	Is there a statistically significant difference in how the site count affects the whole system deployment time between systems?	**System I**	No
RQ5:	Is there a statistically significant difference in the frequency of whole system deployments between systems?	**System I**	No

Generally my research failed to get statistically significant differences between the two deployment systems under comparison. Only research question with a statistically significant answer was RQ3. Results for RQ3 are clear, but from a business viewpoint it is not really interesting, as single site deployment is not taking a lot of deployer time.

From the viewpoint of enacting Kaiku's transition from VM's **System Ia** to Kubernetes **System Ib** as deployment systems, RQ4 is interesting, especially as Kaiku plans to grow in the future and this means deploying Kaiku software to more customer sites. Even if RQ4 failed to achieve statistical

significance, parts of the regression model used in it point towards better per-site scaling for **System Ib**.

During the research both systems under study were in use and under changes. This lowers the significance one should assign to the results.

5.6 Critique

The deployment systems are tied together, so full deployment is hard to measure. The system under measurement is under constant change, as it is being maintained and improved by people working under Kaiku. P-values of many most research questions were under 0.05 during most of the data gathering interval, but crucially not when the final data was exported.

Better planning of instrumentation would have enabled the separation the systems better, as the stages of the deploy process could have been measured and compared. A quick search for a unified software system deployment measurement methodology was conducted, but it did not find a validated and clear methodology to base the measurements methodology on. This lowers the reproducibility and relevance of the results from a general research viewpoint.

Analysis of RQ5 raised some questions about the normality of data. Investigating it with data-analysis tools better suited for beta-distributions would have been prudent.

Chapter 6

Discussion

In this chapter the results of the thesis, improvement to implementing a similar study and possible new paths for research around the thesis are discussed.

6.1 Overview

The thesis failed to generate statistically significant results for four out of five research questions. Overview of the results can be found in section 5.5. The only research question with statistically significant result is not really applicable from a business viewpoint. Statistical significance will improve with more data gathering, but sadly this can't be done in the timeframe of this thesis.

During the data gathering there were time intervals when research questions had statistically significant answers, but changing the data-gathering interval in situ without methodological basis would have been inprudent and would have lead the thesis to the methodological problems ("multiple comparisons") outlined by Gelman and Loken in [32].

From the viewpoint of Kaiku Health, this thesis leaves a complete data-analysis and instrumentation pipeline in place, which can be used to verify that the transition from legacy infrastructure to orchestrated containers really improves the software deployment performance of Kaiku. Running the

data-analysis with new data takes only few minutes and can thus be easily used.

From the viewpoint of the author, doing this thesis taught one a lot more about study design and the myriad problems that face ardent researcher struggling to wrangle knowledge from world with statistical analysis. The author is happy to have learned a lot more about LaTeX, Jupyter Notebooks and especially about SciPy in action.

From the viewpoint of DevOps, this thesis does not provide a lot of value. The background provided by Forsgren et al. in their research helped the author a lot, but sadly the statistically insignificant findings do not provide any further information to DevOps community. Revisiting the thesis data-analysis once the Kaiku's infrastructure transformation is ready could possibly provide some evidence that container orchestration can be used to improve software deployment performance, but even that speculative evidence would have to be discounted quite heavily, as it would originate from a single case study.

6.2 Possible Improvements

The instrumentation of both deployment systems could have been better. This would have given us better visibility inside the deployment systems. On some level, the build system of Kaiku is also partly responsible for the deploy, as it builds the image (for **System Ib**) and static assets for both systems. The choice of limiting asset and image build outside the scope was done because of resource reasons.

6.3 Further Avenues for Research

Measurement of software deployment does not seem to be a well-established area of research with clear, common methodologies, that would make the comparison of measurements between systems accurate. There seems to be

surprisingly little academic research on the topic. Hence a common methodology and agreements on what to measure and how would probably benefit the field. Especially commonly agreed thresholds for start and end of deployment would be beneficial, as those would allow more accurate comparisons of different system.

6.4 The Author's Relation to Kaiku

The author is employed by Kaiku Health and hence there is a possibility of conflict of interest. Views put forth in this thesis are my own and not those of Kaiku Health.

Bibliography

[1] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. en. IT Revolution, Mar. 2018. ISBN: 978-1-942788-35-5.

[2] Nicole Forsgren and Jez Humble. *The Role of Continuous Delivery in IT and Organizational Performance*. en. SSRN Scholarly Paper ID 2681909. Rochester, NY: Social Science Research Network, Oct. 2015. URL: https://papers.ssrn.com/abstract=2681909 (visited on 06/05/2019).

[3] Floris Erich, Chintan Amrit, and Maya Daneva. *Report: DevOps Literature Review*. Oct. 2014. DOI: 10.13140/2.1.5125.1201.

[4] Andreas Brunnert et al. "Performance-oriented DevOps: A Research Agenda". In: *arXiv:1508.04752 [cs]* (Aug. 2015). arXiv: 1508.04752. URL: http://arxiv.org/abs/1508.04752 (visited on 06/05/2019).

[5] Kathleen M. Eisenhardt and Jeffrey A. Martin. "Dynamic capabilities: what are they?" en. In: *Strategic Management Journal* 21.10-11 (2000), pp. 1105–1121. ISSN: 1097-0266. DOI: 10.1002/1097-0266(200010/11)21:10/11<1105::AID-SMJ133>3.0.CO;2-E. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/1097-0266%28200010/11%2921%3A10/11%3C1105%3A%3AAID-SMJ133%3E3.0.CO%3B2-E (visited on 07/18/2019).

[6] Lianping Chen. "Continuous Delivery: Huge Benefits, but Challenges Too". In: *IEEE Software* 32.2 (Mar. 2015), pp. 50–54. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2015.27.

[7] Martin Fowler. "Continuous Integration". en. In: (), p. 14. URL: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf.

[8] W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques". In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. event-place: Monterey, California, USA. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 328–338. ISBN: 978-0-89791-216-7. URL: http://dl.acm.org/citation.cfm?id=41765.41801 (visited on 11/17/2019).

[9] Benedikt Martens, Marc Walterbusch, and Frank Teuteberg. "Costing of Cloud Computing Services: A Total Cost of Ownership Approach". In: *2012 45th Hawaii International Conference on System Sciences*. ISSN: 1530-1605. Jan. 2012, pp. 1563–1572. DOI: 10.1109/HICSS.2012.186.

[10] Prateek Sharma et al. "Containers and Virtual Machines at Scale: A Comparative Study". en. In: *Proceedings of the 17th International Middleware Conference on - Middleware '16*. Trento, Italy: ACM Press, 2016, pp. 1–13. ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988337. URL: http://dl.acm.org/citation.cfm?doid=2988336.2988337 (visited on 09/05/2018).

[11] Emiliano Casalicchio. "Container Orchestration: A Survey". en. In: *Systems Modeling: Methodologies and Tools*. Ed. by Antonio Puliafito and Kishor S. Trivedi. EAI/Springer Innovations in Communication and Computing. Cham: Springer International Publishing, 2019, pp. 221–235. ISBN: 978-3-319-92378-9. DOI: 10.1007/978-3-319-92378-9_14. URL: https://doi.org/10.1007/978-3-319-92378-9_14 (visited on 02/11/2020).

[12] Emiliano Casalicchio. "Autonomic Orchestration of Containers: Problem Definition and Research Challenges". In: Jan. 2017. DOI: 10.4108/eai.25-10-2016.2266649.

[13] Spyridon V. Gogouvitis et al. "Seamless computing in industrial systems using container orchestration". en. In: *Future Generation Computer Systems* (July 2018). ISSN: 0167-739X. DOI: 10.1016/j.future.2018.07.033. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17330236 (visited on 11/17/2019).

[14] *etcd*. URL: https://etcd.io/ (visited on 04/22/2020).

[15] *Concepts (Kubernetes documentation)*. en. URL: https://kubernetes.io/docs/concepts/ (visited on 02/11/2020).

[16] *Kubernetes Components*. en. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 12/02/2019).

[17] Brendan Burns et al. *Borg, Omega, and Kubernetes*. en. URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/44843.pdf.

[18] Gigi Sayfan. *Mastering Kubernetes*. en. Google-Books-ID: dnc5DwAAQBAJ. Packt Publishing Ltd, May 2017. ISBN: 978-1-78646-985-4.

[19] Ethan Basch et al. "Symptom Monitoring With Patient-Reported Outcomes During Routine Cancer Treatment: A Randomized Controlled Trial". In: *Journal of Clinical Oncology* 34.6 (Feb. 2016), pp. 557–565. ISSN: 0732-183X. DOI: 10.1200/JCO.2015.63.0830. URL: http://ascopubs.org/doi/10.1200/JCO.2015.63.0830 (visited on 08/14/2017).

[20] *Ruby on Rails*. URL: https://rubyonrails.org/ (visited on 04/23/2020).

[21] *Nginx*. en. Page Version ID: 951649505. Apr. 2020. URL: https://en.wikipedia.org/w/index.php?title=Nginx&oldid=951649505 (visited on 04/23/2020).

[22] *Passenger - Enterprise grade web app server for Ruby, Node.js, Python*. en. URL: https://www.phusionpassenger.com/?ref=og (visited on 04/23/2020).

[23] *PostgreSQL: About*. URL: https://www.postgresql.org/about/ (visited on 04/23/2020).

[24] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (Dec. 2008), p. 131. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8. URL: https://doi.org/10.1007/s10664-008-9102-8.

[25] Mohamad Amin Pourhoseingholi, Ahmad Reza Baghestani, and Mohsen Vahedi. "How to control confounding effects by statistical analysis". In: *Gastroenterology and Hepatology From Bed to Bench* 5.2 (2012), pp. 79–83. ISSN: 2008-2258. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4017459/ (visited on 04/23/2020).

[26] *Elasticsearch*. en. Page Version ID: 951604210. Apr. 2020. URL: https://en.wikipedia.org/w/index.php?title=Elasticsearch&oldid=951604210 (visited on 04/23/2020).

[27] *Welcome to Python.org*. en. URL: https://www.python.org/ (visited on 12/02/2019).

[28] *SciPy library — SciPy.org*. URL: https://www.scipy.org/scipylib/index.html (visited on 12/02/2019).

[29] *Project Jupyter*. URL: https://www.jupyter.org (visited on 12/02/2019).

[30] Marie Delacre, Daniël Lakens, and Christophe Leys. "Why Psychologists Should by Default Use Welch's t-test Instead of Student's t-test". eng. In: *International Review of Social Psychology* 30.1 (Apr. 2017). Number: 1 Publisher: Ubiquity Press, pp. 92–101. ISSN: 2397-8570. DOI:

10.5334/irsp.82. URL: http://www.rips-irsp.com/articles/10.5334/irsp.82/ (visited on 03/03/2020).

[31] *scipy.stats.linregress — SciPy v1.3.3 Reference Guide*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html (visited on 11/28/2019).

[32] Andrew Gelman and Eric Loken. "The garden of forking paths: Why multiple comparisons can be a problem, even when there is no "fishing expedition" or "p-hacking" and the research hypothesis was posited ahead of time". In: *Department of Statistics, Columbia University* (2013).

Appendix A

Data

The Data and the data-analysis Jupyter Notebook will be made available after the thesis has been approved. The data will be anonymized. The URL for the data and analysis will be https://thesis.sampsa.laapotti.net/.