Riku Seittenranta

Modernizing Proprietary E-commerce Platform Infrastructure

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

24 November 2018

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Riku Seittenranta Modernizing Proprietary E-commerce Platform Infrastructure 39 pages + 3 appendices 24 November 2018
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Ville Jääskeläinen, Head of Master's program (Metropolia)

This study was made for a Finnish company that develops a proprietary e-commerce platform. The need for the study came from the realization, that the infrastructure design that has been used till now might not be sufficient in the future, as the number of environments and the amount of traffic they receive increases. The current infrastructure had served the company well for years, but the design eventually turned out not to be as scalable as desired and didn't provide the level of availability that was excepted from today's systems.

The goal of the thesis was to create an improved infrastructure design for the e-commerce platform. This new design was expected to improve on the shortcomings of the current infrastructure and utilize available modern technological solutions and the latest trends in infrastructure design and DevOps. The final design was built upon three widely used technologies: Kubernetes, Redis and PostgreSQL.

As an additional result of this study, a proof of concept infrastructure was created to validate the feasibility of the final infrastructure design. The technologies used in the proof of concept infrastructure were evaluated and chosen from established best practices and current trends. The proof of concept environment provided a solid foundation to continue the development work on, with improvements on all the problem areas that were identified on the current infrastructure. The proof of concept infrastructure will act as a base for future development and will be further refined to be used eventually for production environment deployments.

Keywords	infrastructure, availability, scalability, portability, software, devops, containers, docker, kubernetes, jenkins

Table of Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
DevOps	Development and Operations
DSL	Domain Specific Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
IT	Information Technology
PaaS	Platform as a Service
POC	Proof of Concept
QA	Quality Assurance
SaaS	Software as a Service
SQL	Structured Query Language
REST	Representational State Transfer
UI	User Interface
VM	Virtual Machine

1 Introduction

New software startups emerge almost daily, and all those software projects face similar questions in the beginning of their life time. Decisions need to be made on the technologies and coding conventions to be used, the architecture of the software and the initial infrastructure where the application will reside. Software startups pursue rapid growth, which often means making quick decisions and accepting to take on technical debt to be able to deliver new features in a quick pace.

While building a software project, it is important to take notice that each technological and architectural decision that is made, may limit the interoperability and portability of the application in the future. By allocating enough time to evaluate the right components and design decisions for the project, one can keep options open for more straightforward portability in the future by minimizing vendor lock-in.

When considering cloud vendors for a software project, one major thing to consider is cloud lock-in. A cloud lock-in may become costly for companies when switching from one cloud provider to another if the need should arise. A cloud vendor lock-in is a result of the lack of standardization between cloud providers.

1.1 Scope and Outcome

This study focuses on modernizing the infrastructure design of a proprietary e-commerce platform, developed by a Finnish company. The e-commerce platform is available as a Software as a Service (SaaS) offering and as an enterprise solution that may be installed into the customer's own data center or into public or private cloud environments.

Current infrastructure design of the platform consists of a collection of virtual machines with predefined roles. New environments require considerable amount of manual software installation and configuration work. The scalability of the environments are limited by the amount of resources available for the running virtual machines. Deploying new software versions may require short service breaks when the running services are restarted.

The goal of this study was to create a new infrastructure design for the platform that allows it to be easily installable to different on-premise and cloud environments with minimal additional work. The new design should also offer improvements on the scalability and availability of the platform and allow rolling software updates without service breaks. As the outcome of this study, a reproducible proof of concept (POC) environment was produced, which may later be further developed into an environment that is suitable for production use.

This study concentrates on establishing a new infrastructure design for the platform that takes notice of and improves on the shortcomings of the current one by utilising latest technologies and best practises. The new design was validated by using a proof of concept environment, to ensure that the design is feasible to implement in the near future. This study does not include the actual migration of the platform on top of the new infrastructure, nor does it make efforts to quantitatively compare the current infrastructure to the new one.

1.2 Design Requirements

The new design needs to enable straightforward deployment to environments with different characteristics. These environments include on-premise data centers, private clouds and public clouds. Because the environments may have considerable differences, the design should provide reasonable amount of flexibility to make it adaptable to these variances.

When possible, managed infrastructure services should be preferred in environments where such alternatives are available. Examples of such services include managed database services such as Amazon Relational Database Service or Microsoft Azure SQL Database, and managed in-memory data stores such as Amazon ElastiCache and Microsoft Azure Redis Cache. The use of managed services reduces the maintenance work and specialized expertise needed to keep the service running smoothly and keeping the service updated (Atchison, 2016).

The new design will be used as the basis for future production environments. Because of this, a range of requirements were set for the design. The design needs to take into account the following requirements:

- **Availability**; The platform needs to be able to serve its function without interruptions, preferably with the same availability percent that the infrastructure provider provides; software updates shouldn't affect the availability of the service.
- **Scalability**; The infrastructure needs to conform to changes in traffic and increase or reduce resources accordingly to match the incoming requests.
- **Security**; The platform handles sensitive information and has publicly available interfaces; the design needs to provide adequate security against outside attacks.
- **Deployability**; The platform will be deployed into several different environments; infrastructure deployments and changes to existing environments need to be easily repeatable with minimum additional work.
- **Manageability**; As the number of environments gets higher, the environments should stay manageable in a controlled manner with reasonable effort.

1.3 Research Design

This study was conducted in the following phases

1. Recognize the problematic sections of the current infrastructure.
2. Analyze possible solutions for each problem.
3. Identify solutions that provide resolution for the problems, utilizing latest trends and best practices of the field.

This study is divided into six chapters. In the first chapter, the research problem and the design are introduced. The second chapter delves into the core terms and concepts, that the reader should be familiar with. In the third chapter, the current architecture and infrastructure of the platform are presented, and known problems identified. The fourth chapter investigates solutions for the identified problems. The fifth chapter present the new infrastructure design with details of the proof of concept implementation as well as the reasoning behind the selected technologies and reflection on how the problems of the current infrastructure are mitigated in the new design. In the sixth chapter, the results of the study are summarized, and future plans of the design are laid out.

2 Core Concepts

This section describes the central terms and concepts used in this study. Understanding these terms is essential in comprehending the study.

2.1 Availability

Availability is a term used to describe a period of a time during which a service is operating normally. A measure of availability is the percentage of uptime in a certain time period, often during a year. Availability may be calculated with a simple equation shown in Formula (1). (Marcus & Stern, 2003)

$$A = \frac{MTBF}{MTBF+MTTR} \quad (1)$$

In the Formula (1), *A* is the percentage of availability, *MTBF* is the mean time between failures and *MTTR* is the maximum time to resolve a problem. If the MTTR approaches zero, A closes on 100 percent. (Marcus & Stern, 2003)

The time when a service is unavailable is commonly known as downtime. Definitions for when the service is considered unavailable vary between organizations. Downtime may be scheduled or unscheduled. Downtime is usually scheduled for system maintenance work, which cannot be performed without it affecting the uptime of the system. Unscheduled downtime may be caused by a variety of reasons, including but not limited to human error, hardware or software malfunction, network failure or a natural disaster. (Marcus & Stern, 2003)

2.2 Scalability

Scalability can be defined as the ability of a system to conform to increasing amount of traffic while performing adequately. Scalability may be limited by the design of the system itself, or by the hardware resources it consumes. Scalability limitations caused by the system's architectural characteristics are more likely harder, or even impossible, to overcome than limitations in the available hardware resources. (Bondi, 2000)

There are two different methods of scaling that can be used for adding additional hardware resources for a system. The decision on which to use, may vary between different parts of a system. Some parts may benefit more from one than the other, or in some cases both might be beneficial. When scaling vertically, one adds more capacity or computational power to existing hardware units, or nodes. Vertical scaling usually has modest costs and a low risk, but the benefits may be limited on how well the system uses the available resources. By scaling horizontally, additional nodes are added to increase the capacity. Horizontal scaling is usually considered more complex than vertical since it requires support from the architecture of the system. (Wilder, 2012)

2.3 Portability

Portability of a software is defined by the amount of work needed to make it work in different environments. Portability does not mean that the software will work in other environments without any modifications, but that the work needed to make it work is within reasonable limits and doesn't outweigh the cost of rewriting the software fully. (Garen, 2007)

Software interacts with the environment it is run in via interfaces that abstract the underlying details of implementation. Software can be considered to be portable into an environment, if the environment can be made to contain an identical set of interfaces from which the software is reliant on. As such, portability can be achieved by having identical interfaces available in each target environment. Portably designed software has lower porting costs and often lower overall costs during its lifetime. (Mooney, 2000)

Vendor lock-in is a situation where a customer is dependent on products or services of a specific vendor, unable to move to another vendor without substantial costs and extra work. This situation is usually the result of the lack of compatibility between different software, hardware, operating systems or file formats. (The Linux Information Project, 2006)

Vendor lock-in is sometimes also referred to as proprietary lock-in, customer lock-in or just a lock-in. In cloud computing the case of becoming dependent on a specific cloud provider is known as cloud lock-in (Opara-Martins, et al., 2016). The impact of cloud lock-in is reduced by having interoperability and portability between cloud vendors (Opara-Martins, et al., 2016).

2.4 Virtualization

Virtualization is a concept initially formed in the 1960s, as a mean to run multiple workloads simultaneously on mainframe computers. The first commercial virtualization solution was released in 2001 by VMWare. In the years that followed, other competing virtualization solutions surfaced. Nowadays, especially in the era of cloud computing, virtualization is used through-out the industry. (Portnoy, 2012)

Virtualization allows one to run a virtual instance of a system, instead of an actual one, that is abstracted from the underlying hardware. The use of virtualization allows for more efficient use of available resources since the unused physical resources may be allocated for other workloads. Virtualization enables availability, as it is possible to move workloads between physical server instances, to prevent the need to schedule downtime for maintenance operations. Virtualization is utilized in various areas of computing, such as desktop and server systems, networking, storage, software. (Portnoy, 2012)

Server virtualization is a common virtualization case, where multiple operating systems are simultaneously run within a computer system. Virtualized operating systems running on a server virtualization environment are called virtual machines (VM). Virtual machines have access to resources, that are provided to them by a hypervisor. (Portnoy, 2012)

A hypervisor is a layer which operates between the hardware and the virtual machines (Portnoy, 2012). The hypervisor is responsible for the creation and running of the virtual machines on the host machines (Portnoy, 2012). The VMs that are run on the hypervisor are called guests (Portnoy, 2012). See Figure 1, for an illustration of server virtualization.

2.5 Containers

Containerization, or operating-system-level virtualization, is a method of running software in an isolated user-space environment. The programs run inside an isolated environment, the container, that can only access devices and storage that have been especially assigned to it. Since containers are virtualized in the operating system level, they are much more lightweight compared to virtual machines, where the entire host operating system is virtualized. See Figure 1 for a comparison of virtual machines and containers. (Google, 2018)

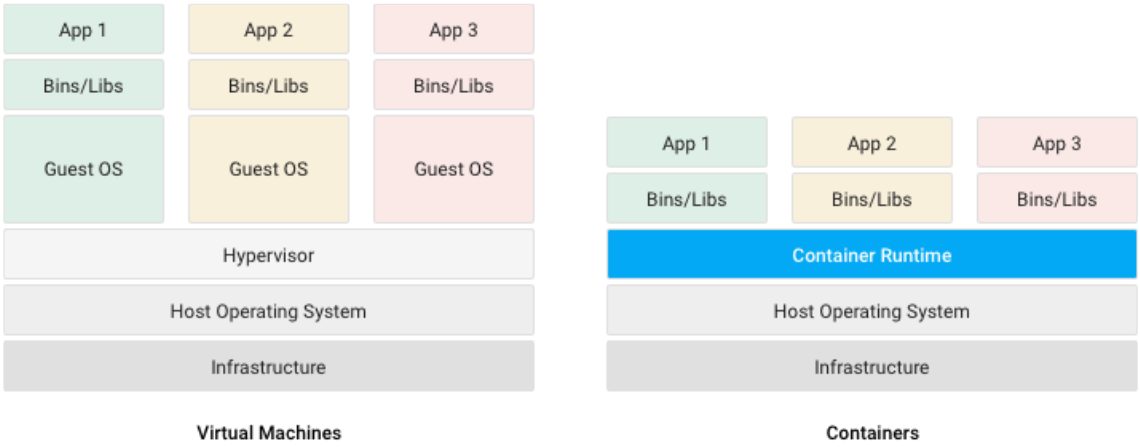

Figure 1. Comparison of virtual machines and containers. (Google, 2018)

Containers provide the means for the developers to package their applications into coherent environments with all the necessary application dependencies bundled in. Containers can easily be run on different environments provided that a compatible containerization layer is available. There are many different container formats available, of which Docker is the best known and most commonly used. (Google, 2018)

2.5.1 Docker

Docker is a container technology first released in March 2013 as open source software, by the dotCloud company. Docker was developed in the beginning as an internal project in the dotCloud company. Initially Docker leveraged Linux Containers (LXC) as its default runtime for executing containers. LXC uses Linux kernel containment features to enable the running of multiple isolated Linux systems on top of a single Linux kernel (Canonical Ltd., 2018). (Tozzi, 2017a)

Since version 0.9, released in 2014, Docker switched the default runtime to libcontainer, because of insufficient isolation layers in the LXC. Libcontainer is an alternative runtime to LXC that leverages the Linux kernel features directly. The move to libcontainer as the default container driver for Docker, provided more stability and reduced the reliance on external parts. (Hykes, 2018)

Docker quickly gained popularity and has since been widely adopted in the field of IT (Datadog, 2018). The reason for Docker's success are many. Docker made it quick and

easy for developers to create containerized applications compared to the implementations before it (Vaughan-Nichols, 2018). Docker also created a stronger ecosystem than competing technologies such as LXC and OpenVZ (Tozzi, 2017b). The timing of Docker was also crucial. Docker was introduced during a time when DevOps was gaining traction and traditional virtualization was starting to seem not flexible enough (Tozzi, 2017b). The technology for running containerized workloads already existed, Docker just made it easier to adopt.

On June 2015, Open Container Initiative (OCI) was launched as a Linux Foundation project by Docker, CoreOS and container industry leaders. Open Container Initiative is an open governance structure for establishing open standards around container formats and runtime. Docker contributed its image format and runtime implementation to serve as the basis for the OCI Image Specification and Runtime Specification. (The Linux Foundation®, 2018)

2.5.2 Container Orchestration

Managing and deploying one or a few containers isn't difficult but becomes harder with more complex applications that are formed from multiple interoperable containers. When managing hundreds or thousands of containers it is no longer feasible to manage containers one by one, but in groups instead. In such cases, it is essential to use a container orchestration system to automate the deployment, management, networking and scaling of the containers. (Eldridge, 2018)

Container orchestration systems manage the containers during their lifecycle. Orchestration systems manage the deployment and configuration of containers and their availability and scaling during their lifetime. Orchestration systems makes sure that the containers have the resources available which were allocated to them, and if necessary moves the containers to hosts where the required resources are available. Orchestration system also provides means to expose services to the internet, load balance incoming traffic between containers, and monitor the status of running containers using health checks. (Eldridge, 2018)

2.6 Continuous Integration and Continuous Deployment

Continuous Integration (CI) is a development process where changes made by the developers are integrated into the main code base as often as possible. After each checked-in change, the CI system initiates an automated build and runs automated tests against the build. The CI system then notifies the developers or quality assurance (QA) immediately if errors are detected in the newly created build. This immediate response allows the developers to react and find errors faster. (Sten Pittet, 2018)

Continuous delivery is a prolongation of continuous integration. With continuous delivery the application is always in a deployable state and can be deployed into an environment at any point of time. With continuous delivery, the automated build and test processes from the continuous integration is extended with an automated release process. Continuous deployment removes the human element from the release process. In continuous deployment, every change that passes the automatic test suite, is deployed into production. (Sten Pittet, 2018)

3 Platform Overview

This section describes the current architecture of the platform in detail. Prominent parts of the platform will be reviewed, and their roles and functions identified.

3.1 Architecture

The architecture of the platform can be examined at two different depth levels. On the top level, Figure 2, the platform can be viewed as three distinct layers:

- **Client layer**, which contains web applications that provide a user interface (UI) for the users of the platform. The Client layer uses the API layer to interact with the rest of the platform.
- **API layer**, which provides an interface to communicate with the platform. The API layer contains multiple RESTful APIs that are utilized by the Client layer and other external sources. The API layer itself contains only a minimal amount of logic as most of the requests are passed onto the Core layer. The API layer includes a caching mechanism that reduces the requests reaching the Core and its persistent storage.
- **Core layer**, which contains the main application logic, background processes and the database for persistent storage.

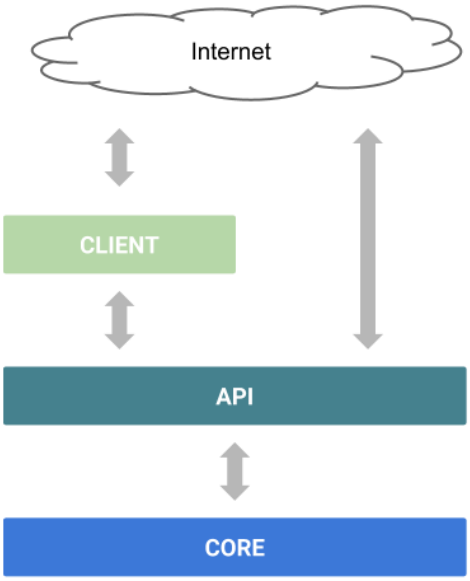

Figure 2. High level view of the architecture of the platform and its three layers.

On a deeper level, the platform comprises of the following five major components: Frontend, API, Cache, Backend and Database as shown in Figure 3.

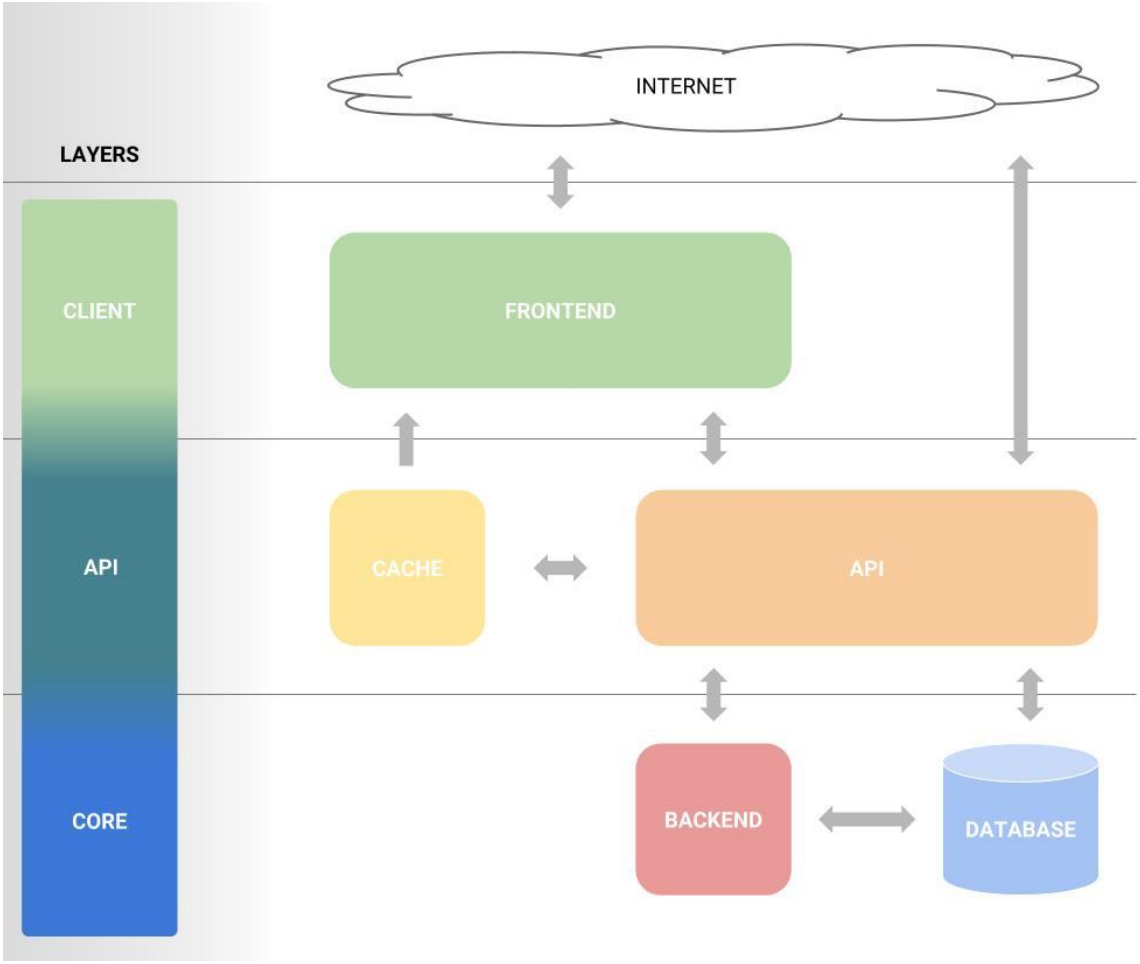

Figure 3. Platform components and their relations, in respect to the layers of the platform.

The Client layer contains the Frontend, a modern JavaScript heavy single page application that provides a user interface for interfacing with the platform. The Frontend communicates with the platform using the API and leverages the Cache component for speeding up the loading of initial content of the first page load. No application state information is persisted on the Frontend.

On the API layer, the platform has the API component, which provides the main endpoint for interfacing with the platform. The API is a lightweight PHP application that provides HTTP REST interfaces for direct communication and for the Frontend to operate against.

The API contains only essential logic and computation, forwarding most of the request to Backend and the Database.

The API layer also includes a Cache component that is a key-value storage used for storing user sessions and other often accessed resources. The Cache component receives requests from the Frontend and from the API. Currently Redis is used in implementing the Cache component.

The Core layer contains the Backend component, which is a Java EE application that is responsible for scheduled background processes and integrations to external systems. Database component, also a part of the Core layer, is the main data storage of the platform. Open source relational database system PostgreSQL is used as the database implementation. The Database receives requests from the Backend and the API.

3.2 Infrastructure

The current infrastructure of the platform is built upon clustered virtual servers that have predetermined roles assigned to them. The named roles are Application, Integration and Database (see Figure 4). The Application servers contain a HTTP server that serve the Frontend and the API to the internet. Redis cache instance is often also installed on the application servers. Backend and Database components reside on separate servers, with named roles: Integration and Database, because of the resources that they require and to allow more straightforward horizontal scaling when needed.

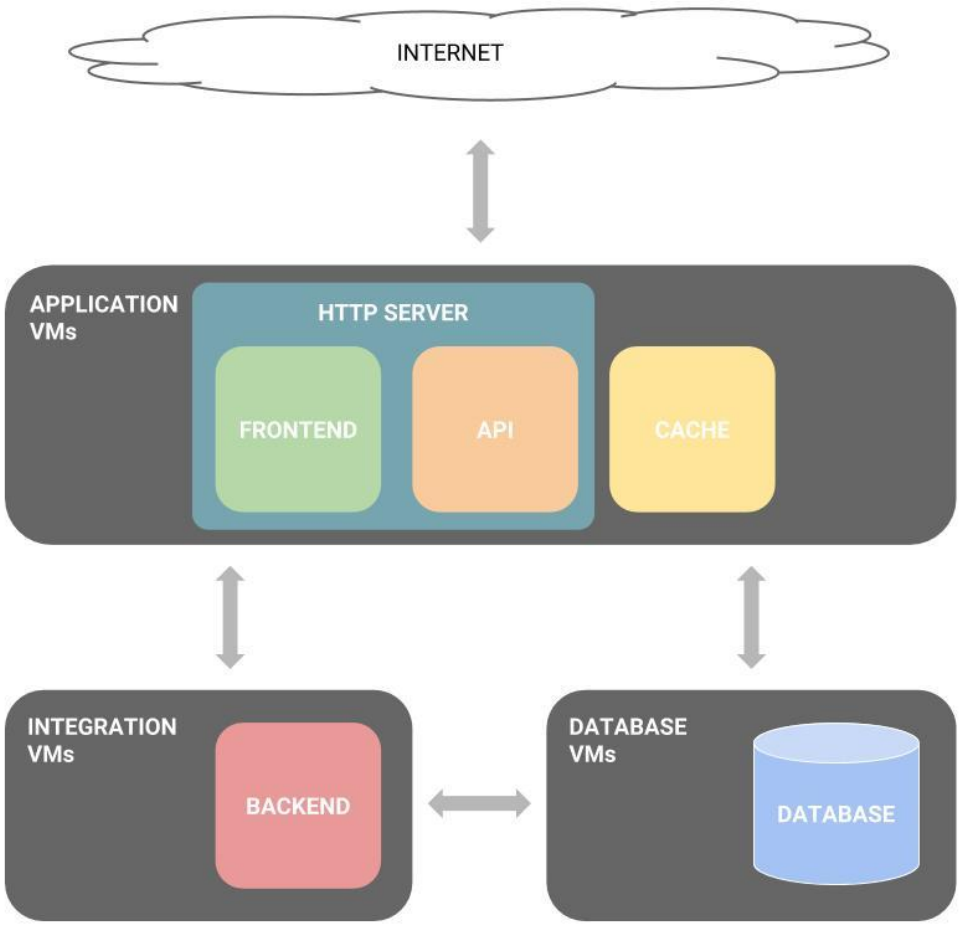

Figure 4. Platform's current infrastructure, division of the components on the virtual servers.

3.3 Identified Problems

Scalability, current infrastructure implementation doesn't offer straightforward auto-scaling. Horizontal scaling is troublesome because adding new servers needs manual work and preparations. Vertical scaling takes less work but adding resources to existing instances only helps to a certain point.

Availability, currently updates to the servers' software or to the application need a short break to the availability of the platform. While the effects of these breaks can be minimized by performing them during non-peak hours, they are still inconvenient for the customers and the developers. Developers shouldn't need to worry when updates may or may not be taken into production, and customers expect an always working service.

Deployability, being able to deploy the platform into environments with varying characteristics may need a considerable amount of manual preparations. These have sometimes been done as one-off configurations that are not quickly repeatable. The current environment setup process is partly automated but has strict limitations for the compatibility of the underlying operating system and its version.

Manageability, current environments are in large part managed using a configuration management system. Server instance configurations are quite manageable but managing the state of a whole environment is cumbersome and involves coordination with multiple outside parties.

4 Analysis of Solutions

In this chapter, the problem areas of the platform are analyzed. For each of the problem areas, a solution or an enhancement is identified. As a basis for these solutions, existing research, commonly used best practices and the latest trends of field are used.

4.1 Availability

To achieve high availability, one must consider a wide range of scenarios that can have an effect on the availability of the application. A contingency plan should be made for all, or at least the most probably ones, depending on how much money and resources one is willing to use for ensuring high availability.

Factors which can impact the availability of an application include but are not limited to the following: hardware failures, environmental and physical failures, network failures, database system failures, software bugs, insufficient resources, web and application server failures, denial-of-server attacks (Marcus & Stern, 2003). For the current infrastructure, issues are often traced back to misbehaving application servers or side effects of insufficient server resources.

In their book “Blueprints for High Availability”, published in 2003, Evan Marcus and Hal Stern present a list of 20 key principles for a high availability design, from which some can be considered directly infrastructure related solutions that are still valid today. The list contains widely known good recommendations such as K.I.S.S (Keep it simple, stupid), choose mature software and hardware, and design for growth, to name a few.

On the list is also “Remove Single Points of Failure”, which is an essential principle when developing a highly available infrastructure. One typical solution for eliminating single points of failure, is to use a load balancer before your servers to route the traffic evenly between the servers. In the case of a server going down, the traffic can be distributed to the healthy servers only, to ensure the availability of the service. However, adding a single load balancer doesn’t remove the problem of having a single point of failure, the point of failure has now just moved to another layer. One way to remedy this issue is to use systems that supports routing the traffic to multiple load balancers via IP address redirection (Marcus & Stern, 2003) .

For the current environments, application and infrastructure updates occasionally cause downtime. To prevent downtime during the update of a live application, one can use a rolling upgrade method. Rolling upgrades are performed in such way that parts of the application are available simultaneously while the incoming traffic is gradually directed from the previous version to the new version in a controlled fashion (Zukanov, 2018).

4.2 Scalability

To be scalable, the infrastructure needs to be able to adapt to changes in the traffic volumes that it receives. Sudden changes in traffic should not affect the availability of the platform, as the infrastructure should conform to these changes and provide enough resources as needed. Scaling vertically by increasing the available resources provides a low risk method for processing more traffic (Wilder, 2012). Vertical scaling is however limited by the ability of the software to use the available resource, and often involves downtime as changes in the hardware are required (Wilder, 2012).

Load balancing is a horizontal scaling solution for distributing traffic across several servers (Marcus & Stern, 2003). The number of servers performing the same function may be increased or decreased to match the load that is generated by the incoming traffic (Marcus & Stern, 2003). To achieve a sufficient level of scalability the infrastructure needs to be able to monitor the amount of traffic and load it receives and automatically adjust itself to conform to the changes by adding or removing server instances. Such automatic scaling features are usually available in public and private cloud environments, but for traditional VM deployments not so often.

4.3 Portability

The platform needs to be installable into environments with varying characteristics without major extra workload. To make the platform portable, abstraction layers need to be implemented to separate the platform from the underlying infrastructure. This way a separation of concerns is achieved as the infrastructure provides a layer, or layers, that the platform can be developed against. This allows the specifics of the infrastructure to change and vary between platform environments, but the differences won’t affect the platform.

To enable portability, it is a good practice to avoid unnecessary vendor lock-in. One way to avoid vendor lock in is to prefer the use of open industry wide standards, which cannot be controlled by a single company or organization (The Linux Information Project, 2006). Vendor lock-in cannot be avoided entirely however. Every time a piece of software is chosen to be used, some level of vendor lock-in is accepted. Since vendor lock-in is a situation where there are a limited number of options to choose from, the key to minimizing vendor lock-in is in expanding the range of options by keeping the level of interoperability and portability of the software high.

4.4 Decoupling and Containerisation

The components of the platform where originally designed not to be too tightly coupled, which allows them to be separated into decoupled services with reasonable amount of work. This section looks in to each component, and accesses their suitability for scalability, containerization and the availability of managed service implementations.

The Frontend needs a HTTP Server that serves the single page web application. Since no application state is persisted on the Frontend, this allows the service to be horizontally scaled without having to worry about synchronizing the state between the instances. Because the Frontend is a stateless service, it is an excellent candidate for containerization. Like the Frontend, the API also needs a HTTP Server to serve the HTTP REST API. Likewise, no application state is stored on the API, so the API is also easily scalable into many instances, and thus a good candidate for containerization.

The Backend includes a Java EE application server. Although the Java EE application server is capable of storing the application state, it doesn’t do so by design of the platform. Application state is instead stored in the Cache and in the Database. Since the Backend doesn’t store the application state itself, it is a suitable candidate for containerization and horizontal scaling.

Both the Backend and the Database components contain stateful application data. Horizontal scaling of stateful services is much more complex tasks than for stateless services. With stateful services one needs to make sure that the state is synchronized between the instances. Thankfully both the Backend and Database are using implementations that have support for scaling the service across multiple instances. This however makes the Backend and Database not ideal for containerization.

Although the Cache and Database implementations, Redis and PostgreSQL, have a support for scaling across multiple instances, the maintenance of these implementations require additional personnel resources and expertise. Because of this extra complexity, it would be more cost effective in many cases to use managed services for Redis and PostgreSQL when possible.

4.4.1 Managed Services

The platform needs to be able to use managed service alternatives in environments where such alternatives are available. Managed services move the burden of maintaining and updating the software to the provider of the service. By not having to invest resources on the maintenance of the service, one can focus more on the actual product. Components of the platform that have known and well proven managed alternatives are the Database and Cache, with PostgreSQL and Redis implementations.

Google has Cloud SQL service which provides a fully managed PostgreSQL database (Google, 2018a). AWS provides a similar service called Amazon RDS with support for PostgreSQL as well (Amazon Web Services, 2018b). Microsoft Azure also has a fully managed PostgreSQL database, Azure Database for PostgreSQL (Microsoft, 2018a). Each provider's database service offers comparable promises of high performance, scalability and high availability.

Similarly, to managed databases, all three major cloud providers also have fully managed Redis services available. Google Cloud has a Cloud Memorystore offering for Redis (Google, 2018b). Amazon Elasticache has support for a Redis engine (Amazon Web Services, 2018a). Microsoft Azure also has Azure Redis Cache service (Microsoft, 2018b). All providers have equivalent promises of scalability, high availability and high performance for their offerings.

Since well proven managed services exists for the stateful components of the platform, these should be preferred in cloud deployments where such services are available. By using managed services instead of managing the services by ourselves, resources are freed for other tasks.

4.4.2 Container Orchestration

To make the containerized services work together, a container orchestration platform is needed to deploy, scale and manage the containers. The orchestration platform needs to be available as a self-hostable solution, and preferably as a managed service as well. Two of the most popular orchestration solutions for Docker containers are Docker Swarm and Kubernetes.

Docker Swarm is a container orchestration system has been natively built into Docker since version 1.12.0 released in July 2016 (Docker Inc., 2018b). It provides simple and powerful solution for clustering your Docker containers, without the need of additional orchestration software. Docker Swarm provides a stable and well proven environment for managing containers. Docker Swarm is operable using the same command line interface (CLI) that is used to manage Docker. (Docker Inc., 2018a)

Since Docker Swarm requires no additional software components, it is simple to get started with. Docker Swarm is not offered as a managed service on any of the major cloud vendors, which means that the Docker Swarm cluster needs to be setup and maintained by yourself in every environment.

Kubernetes is an open source platform for managing container orchestration. Kubernetes was created by Google, based on what they learned with their internal cluster management system called Borg. Kubernetes was first released as an open source project in 2014 (Beda, 2014). Kubernetes is a fully featured container management platform that makes it straightforward to deploy and manage applications. Kubernetes provides features such as health monitoring with automated restarts and automated scaling. (Google, 2018)

Kubernetes can be run locally, in cloud environments and even on bare metal servers. There are several solutions for installing and managing your own Kubernetes cluster, ranging from turnkey solutions to complex customized setups. Fully managed Kubernetes cluster services are also available by all the major cloud vendors: AWS, Google Cloud and Microsoft Azure. (The Kubernetes Authors, 2018a)

Kubernetes has also been embraced by Docker Inc., the company behind Docker, in their enterprise offering: Docker Enterprise Edition (Docker EE). Starting from version

2.0, Docker Enterprise Edition supports Kubernetes as an alternative orchestration layer to Docker Swarm (Saraswat, 2018).

4.5 Deployability and Manageability

The provisioning of new environments should be effortless and as automated as possible. Each environment needs to have implementations in place for the basic infrastructure abstraction layers. The implementation details may vary between environments, but the required layers should use well known and widely supported interfaces, or preferably established standards. When the environment has implementations for each of the layers, deploying and managing becomes straightforward for the developers, because they can focus on the application and not the implementation details of the infrastructure.

Various instances of the platform should be manageable by the developers with minimum effort. Infrastructure and configuration changes and new deployments should be version controlled and easily performable by the developers of the platform. Every change to the infrastructure and configuration should be performed in an orderly fashion using suitable tools which tracks the changes that were made. Changes should never be done by hand in an ad-hoc manner, except under extreme circumstances.

4.5.1 Infrastructure as Code

Infrastructure as code (IaC) applies the same ideas used to enhance the quality of software engineering to controlling your IT infrastructure. If the infrastructure can be handled the same way as code, it can be version controlled and tested, which results in better quality of the infrastructure. Infrastructure as code brings the infrastructure closer to the developers, enabling a DevOps approach for delivering a greater overall quality for the customers. (Wittig & Wittig, 2016)

One popular software for defining infrastructure is Terraform, which is cloud-agnostic tool for building, changing and versioning your infrastructure. Terraform supports using multiple service providers in combination. Terraform supports a wide range of providers, with integrations available for all major IaaS, PaaS and SaaS services. Because Terraform supports using different providers at the same time, it can be used to define the overall infrastructure even when using services from many different providers. Terraform is an

open source project develop by HashiCorp, it was initially released in 2014 (HashiCorp, 2018a). (HashiCorp, 2018b)

4.5.2 Configuration Management

During the lifetime of a software project, the software goes through many changes that affect the software in various ways. This results in multiple versions of the software existing at the same time, which increases confusion surrounding the software project. If this confusion goes unmanaged it will hinder the productivity of the team developing the software. (Babich, 1986)

In his book “Software Configuration Management: Coordination for Team Productivity”, Wayne A. Babich (Babich, 1986) describes configuration management in the following manner:

> Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

Software Configuration Management (SCM) is an activity that should be used through each step of the process of developing a software. For a well-run software process, it is essential that changes are performed in a controlled manner at every level of the process. By planning the changes thoroughly, and ensuring the changes are properly implemented, one can reduce the amount of quality issues during the lifetime of a project. (Pressman, 2010)

While Babich’s definition focuses on Software Configuration Management, the same definition can be applied to other project areas, such as documentation and other operating systems. Since the goal of configuration management is to maximize productivity while minimizing the mistakes, it is an obvious target for automation. As such, a wide range of automated solutions exists for common use cases, to reduce the manual work of the configuration management process.

There are many free and open source tools available to fill the role of a Configuration Management system. Many of the tools have an equivalent range of capabilities, which means that often the choice of the tool is dictated by compatibility with other technologies already in use, but also by personal preference and existing know-how. Although many

of the tools can be used to accomplish similar end results, underneath they might operate quite differently.

4.5.3 Release Management and Versioning

Release management is a process that aims to delivery software to its users with quality. The process of release management often contains a set of prespecified steps that are performed in an orderly manner with the goal of delivering a high-quality release within schedule. These steps might include technical and management tasks such as building, testing, packaging and deploying of the software, and collaboration. The release management process should also include a plan for backing-out in case issues are discovered during the process. (Cleveland & Ellis, 2014)

To keep track of the releases, a versioning scheme should be used so each released state of the software maybe referred to unambiguously. A clear versioning scheme and accompanying well thought release management process aids in producing high quality software releases.

Releasing new software updates of the platform and pushing those releases to the environments should be as automated as possible to minimize manual work efforts and human errors during the process. To reduce the efforts needed for managing software builds and releases, a continuous integration (CI) and continuous deployment (CD) processes should be designed and implemented. The CI/CD process needs to be able to periodically produce build artefacts of the latest code changes and deploy these artefacts to specific development, staging or production environments automatically or after a manual approval step. A wide range of proprietary and open source tools are available for building such automated CI/CD processes.

5 Final Design and Proof of Concept

This chapter describes the design for the new overall infrastructure, including the newly designed proof of concept platform environment and its supporting infrastructure. First, the design for the platform infrastructure is presented in detail with the technologies that were chosen to be used in the new design. In addition to the infrastructure of the platform, we'll have a look at the surrounding overall infrastructure, relevant processes and how the platform environments can be managed.

5.1 Platform Design

The new platform design is built on top of three core building blocks (Figure 5): containers running on Kubernetes, a key-value store as the cache and a SQL database for persistent storage.

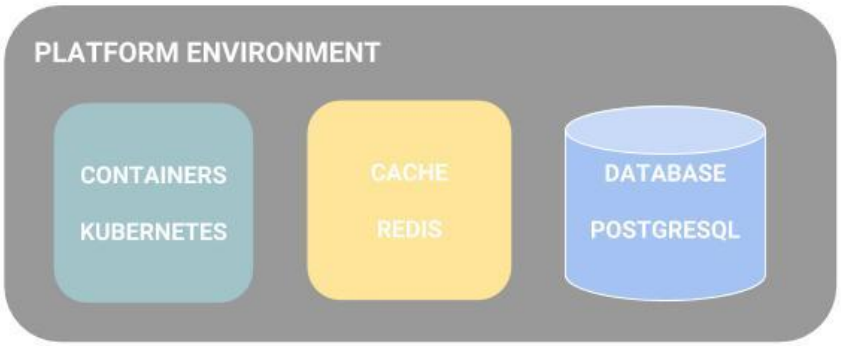

Figure 5. Platform environment and its building blocks.

These three building blocks provide an abstraction layer for the platform to run on. As long as these services can be provided by an environment, the platform should be deployable on it without major porting efforts.

Kubernetes was chosen as the container orchestration platform of choice. One key factor for the selection of Kubernetes, is the fact that its currently the only prominent container orchestration platform that is offered as a managed solution on all major public cloud

providers. With Kubernetes, one also has an enterprise quality orchestration layer with an active community to build one's platform on.

For the cache and database blocks, the current implementations with Redis and PostgreSQL are used. The selection of Redis and PostgreSQL technologies was largely dictated by previous usage and requirements of the platform, but at the concept level of the design, these are interchangeable to other compatible technologies. PostgreSQL can be considered a hard dependency of the platform that cannot easily be changed to another SQL database. Redis on the other hand is not a hard dependency, any persistent key-value store system should be possible to use with a minimal integration work. The platform environments may be deployed into various cloud or non-cloud environments, provided that the environment can provide compatible implementations for Kubernetes, Redis and PostgreSQL. Kubernetes, Redis and PostgreSQL are all freely available open source projects, with active development and communities behind them.

5.1.1 Docker

The components of the platform were containerized, using Docker as the choice of container technology. Docker is an obvious choice because it is the most commonly used container technology today. Because of the widely spread use of Docker and its large userbase, choosing some other container technology at this point would have needed a compelling technical reason or other external requirement.

To containerize the components of the platform, a Dockerfile was created for each component: Frontend, Backend, API, Cache, Database. See Figure 6, which illustrates the file structure with Dockerfiles for each of the components. Docker uses text files called Dockerfiles to define how a Docker image is structured and built, see Figure 7 for an example of a simple Dockerfile. A Dockerfile usually contains a series of scripts that copy and setup everything in to its place. A Docker image may consist of many layers, sometimes referred to as intermediary images (Docker Inc., 2018c). A Docker image may be built upon an existing image that serves as a base for the new image (Docker Inc., 2018c). Docker image layers are especially useful when building multiple varying images on top of a similar base structure and dependencies.

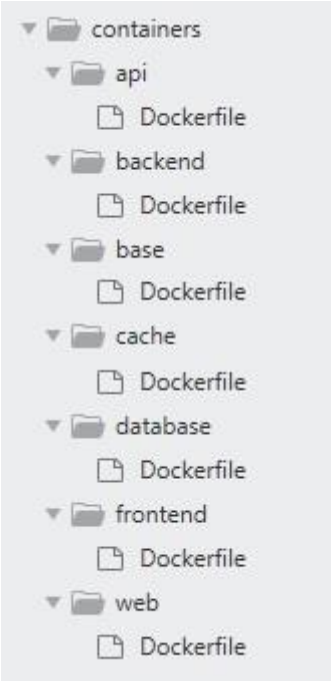

Figure 6. Directory structure showing the Dockerfiles for each component.

Although the Cache and Database components are most likely to be implemented without containers in most environments, containerized versions of them were created for convenience for this proof of concept. A base Docker image was created to build the component Docker images from. Using a common base image allows us to easily make changes that are reflected on the images of each component. Generally, it is also considered to be more secure to rely on base images that have been produced by well-known entities, and which you have validated yourself, than to use third party images from untrusted providers.

```
FROM centos:centos6

RUN yum update -y
RUN yum install epel-release -y
```

Figure 7. Dockerfile example for the base image.

The API and Frontend will be serving web services using a similar HTTP server setup. To reduce duplication in the Dockerfiles, a Docker intermediary image was created that contains a working HTTP server setup without configuration. This image, named Web, is based on the base image and used by the API and Frontend images. The API and Frontend images both contain their own configuration files that are used with the HTTP server setup from the Web image.

5.1.2 Kubernetes Application

Since Kubernetes was selected as the container orchestration platform, a Kubernetes application is needed to be created, to be able to deploy the platform into a Kubernetes cluster. The Kubernetes application defines how the containerized components of the platform work together to provide the a fully working software.

A Kubernetes application may consist of several YAML formatted files, which contain declarative definitions of Kubernetes objects. These Kubernetes objects define a desired state of the cluster that the Kubernetes system will try to ensure is in effect. (The Kubernetes Authors, 2018b)

A Pod is the smallest building block in Kubernetes that can be created and deployed. A Pod depicts a single unit of deployment, which may run single or multiple containers. A Pod should contain only a single instance of an application, which may then be scaled horizontally by creating multiple instances of the Pod. In Kubernetes this is referred to as replication. (The Kubernetes Authors, 2018c)

To make management of multiple instances of Pods easier, they are often managed through a Controller, which takes care of replicating the Pods, while providing additional functions as well, such as self-healing capabilities. In Kubernetes there are several different types of Controllers for various purposes. A ReplicaSet controller makes sure that a specified number of instances of a pod exists always (The Kubernetes Authors, 2018d). A Deployment controller can be used manage the state of Pods and ReplicaSets. (The Kubernetes Authors, 2018c)

For each of the components, a Kubernetes Deployment and a Service was created. A Kubernetes Deployment describes the desired state for Pods and ReplicaSets that the

Deployment controls (The Kubernetes Authors, 2018e). See Figure 8 for a YAML formatted snippet of the Deployment of the Frontend component. By changing the values of the Deployment, one can for example add or reduce running Pod instances or change a container image version running in the Pods. In addition to a Deployment, a Kubernetes Service is needed to expose the running Pods of each Deployment to other layers of the platform (The Kubernetes Authors, 2018f).

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend-deployment
  labels:
    app: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: ecom-project-frontend:latest
        name: frontend
        ports:
        - containerPort: 80
```

Figure 8. YAML formatted definition of a Deployment for the Frontend component.

5.1.3 Kubernetes Cluster

Major cloud providers such as AWS, Google and Azure have managed services of Kubernetes available, which integrate directly to their other offerings. Since Kubernetes is a complex platform, managed cluster should be preferred to avoid the work needed to maintain the Kubernetes cluster itself. In environments where a managed Kubernetes cluster isn't a viable option, there exists various projects that try to make it easier to create and manage a self-hosted Kubernetes cluster.

For the proof of concept environment, Rancher Kubernetes Engine (RKE) was used to install and manage the Kubernetes cluster. RKE is an open source Kubernetes installer

that supports provisioning of the cluster to bare-metal and virtual servers (Rancher, 2018). RKE is a certified Kubernetes distribution developed by Rancher Labs Inc (Cloud Native Computing Foundation, 2018). Using RKE, it is straightforward to deploy and manage a Kubernetes cluster on-premise or in a cloud server environment.

To be deploy the RKE, one first needs to have one or more nodes, servers, prepared with a compatible Docker version and correct ports opened. The RKE binary file needs to be downloaded to issue commands for the cluster. To deploy the RKE cluster, one needs to pass a cluster configuration file for the RKE binary. This cluster configuration file, Figure 9, at minimum contains information on which nodes to provision the cluster to. In addition to the node definitions, the config file may be used to provision Kubernetes cluster services with specific versions. (Rancher Labs Inc., 2018)

```
nodes:
    - address: 10.0.0.10
      user: docman
      role:
        - controlplane
        - etcd
        - worker
```

Figure 9. A minimal Rancher Kubernetes Engine config file which was used to create the Kubernetes cluster used in the proof of concept environment.

5.2 Supporting Infrastructure

Since the platform environments contain no build or deployment logic, supporting infrastructure is needed to handle the building and packaging of the platform. These supporting features are centralized into a main environment, which also contains source code management and platform release systems. The main environment acts as a single point of truth and a control point for the platform environments.

Combined, the multiple platform environments and the main environment, form an overall infrastructure as depicted in Figure 10. The platform environment infrastructures and the main environment are controlled from developer workstations, using a combination of infrastructure as code and configuration management tools, such as Terraform and Ansible.

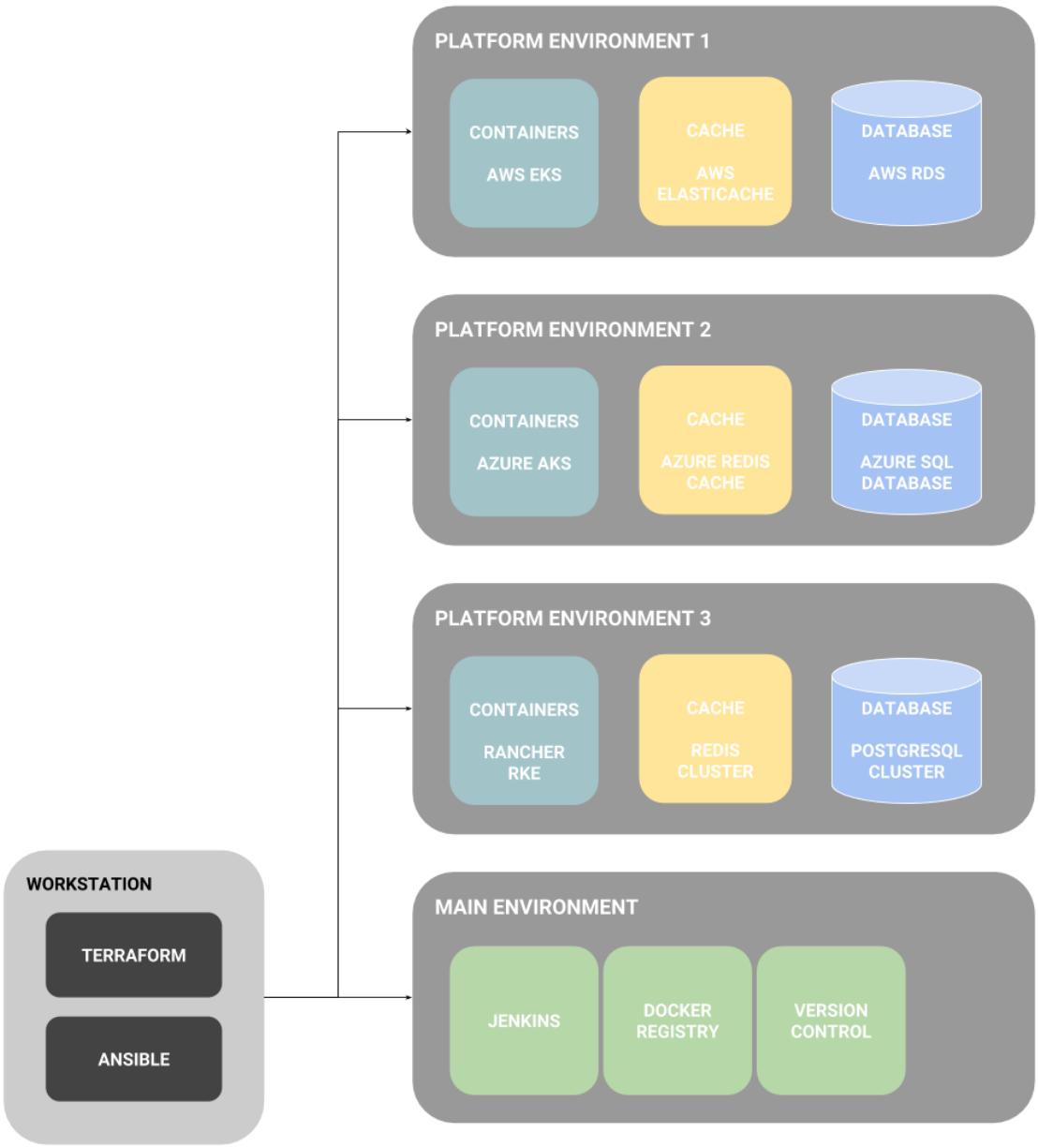

Figure 10. Overall infrastructure with platform environments and the main environment.

5.2.1 Infrastructure Management

To create and manage the infrastructure of the main and platform environments, Terraform was initially selected because of its support for a declarative approach and its extensive support for various providers. With Terraform the Kubernetes, Redis and PostgreSQL clusters may be created and administered using a single centralized configuration, that should be stored in version control.

However, since the proof of concept environment was needed to have running on a local workstation inside virtual machines, it turned out that Terraform didn't provide suitable provider support out of the box. Because of this, Terraform was eventually not used for the proof of concept environment, but it still remains the preferred method of managing infrastructure going beyond this proof of concept environment.

In the end, the proof of concept environment was built onto a single virtual machine using Vagrant for provisioning and managing its lifetime. Vagrant is a tool for building and managing reproducible and portable virtualized development environments (HashiCorp, 2018c). The software running inside the virtual machines can be installed and configured using numerous different methods, such as shell scripts or a configuration management software like Ansible or Chef (HashiCorp, 2018c).

5.2.2 Configuration Management

In environments, such as the proof of concept, where managed services are not available for Kubernetes, Redis or PostgreSQL, additional tooling is required to install and configure the self-hosted clusters onto virtual machines. These virtual machines should be created and managed using Terraform, when possible. For installing the software onto the machines, any configuration management software or shell scripting language could be used. All configuration changes made to the servers, should be done using the configuration management software to reduce unaccounted differences inside the environment. In the proof of concept environment, Ansible was used to manage the configuration of the environments.

Ansible is an automation tool that can be used for configuration management, in addition to application deployment, orchestration and provisioning (Red Hat, Inc., 2018). Ansible is agent-less, so no agent software is needed to be present on the hosts it manages (Red Hat, Inc., 2018). Ansible was originally created by Ansible Inc, but it was acquired by Red Hat in 2015 (Red Hat, Inc., 2015). Ansible can be used to perform ad-hoc task execution across multiple servers, but the real value of Ansible comes from its playbooks (Red Hat, Inc., 2018).

Ansible Playbooks are a collection of YAML defined ‘plays’ that map tasks to specific hosts. Playbooks are usually kept in version control and used to keep configuration on remote systems in sync with the spec. (Red Hat, Inc., 2018)

Ansible was chosen because of the low implementation barrier, that is the result of its agent-less nature. Ansible can handle installing and configuring software with ease and is suitable for additional automation in the future. Because of the large user base of Ansible, there also exists a lot of configuration examples that are straightforward to utilize in one’s configurations. Ansible was also familiar to the company because it was already in use to manage configuration of some of the existing environments. For the proof of concept environment, a collection of Ansible Playbooks was created for installing and configuring the needed software on of the environment.

5.2.3 Packaging the Platform

Kubernetes provides a robust platform for managing and deploying our containers, but for managing our application in Kubernetes and its versions, a lot of repeated commands are needed to run. One can of course handle the deployment and rollbacks of the application directly with the interfaces provided by Kubernetes, but fortunately there exists other solutions as well.

Helm is one such solution. Helm describes itself as a package manager for Kubernetes. Helm charts are used to define an application. Helm provides commands for easily installing and upgrading a Kubernetes application. Helm Charts can be shared, and Helm also provides a community chart repository where one can find ready-made Helm Charts for many commonly used software that one can install with a single command. (Cloud Native Computing Foundation, 2018)

To make it easier to manage the platform, its deployment, versions and their rollbacks, the platform was packaged using Helm. A Helm chart was created for each of the components. In addition to these, a Helm chart was created for the whole platform. This Helm Chart combines the other Helm charts as its subcharts. By separating each component into its own chart, they are straightforward to swap into different implementations in the future.

5.2.4 Continuous Integration and Deployment

Jenkins is used for automating continuous integration and continuous deployment processes. Jenkins is a cross-platform open source automation server. Jenkins has a wide range of plugins available, which enable one to use it for building, deploying or automating projects (Jenkins, 2018a). Jenkins started as a fork of a project called Hudson in 2011, because of disputes regarding Oracle's control of the project (Proffitt, 2011).

Jenkins was chosen because it was already in use in the existing environments and the developers are familiar with it. Jenkins is also one of the most used open source automation servers for integration and deployment, it is actively developed and has a large collection of plugins for extending its functionality or integrating to external systems.

Jenkins uses Jobs as building blocks for enabling automation. A Job can be started by various methods, such as version control triggers, predefined schedules or manually. The amount of Jobs may easily rise to tens or hundreds of jobs, so to avoid having to manually input each Job from the user interface and to easily replicate and manage the jobs, a Job DSL plugin was used. The Job DSL plugin allows one to describe Jobs in a Domain Specific Language (DSL) using a Groovy-based language (Jenkins, 2018b). See Appendix 1 for the DSL definition of the proof of concept.

Jenkins supports many different types of Jobs. Pipeline Jobs provide a solid foundation for creating and integrating continuous delivery pipelines. Pipeline Jobs allows you to define your delivery pipelines using the Pipeline domain specific language, the Pipeline DSL uses Groovy's syntax (Jenkins, 2018c). Figure 11 shows how a Jenkins Pipeline Job looks in the UI. For tasks that need more flexibility, Jenkins provides Freestyle Jobs which offer more flexibility with minimal limitations. Additional Job types may also be added via plugins.

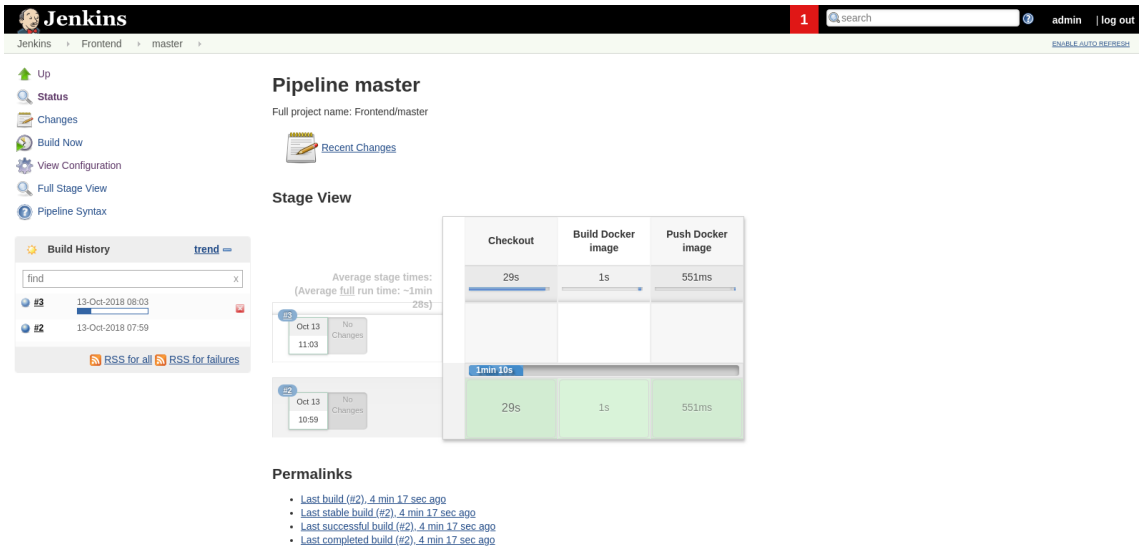

Figure 11. Jenkins UI view of the Frontend Pipeline with one finished job and one running.

A Pipeline can be defined using a Jenkinsfile, that describes a declarative Pipeline with the steps from getting the source code from the version control system to deploying the platform to an environment. Jenkinsfile is a text file containing the exact steps to be done in each stage of the Pipeline, Figure 12. The Pipeline may also be defined using the Jenkins web user interface, but the recommended best practice is to use a Jenkinsfile to define the Pipeline and store the Jenkinsfile in version control. (Jenkins, 2018d)

For each component, a Jenkinsfile was created with separate stages for fetching the latest source code from the version control, building the code into a Docker image, and the pushing the latest image into a private Docker image registry. The Jenkinsfiles were kept simple and identical between different components, Appendix 2. The actual building logic varies between projects, so a unified interface was needed to mask the differences. By keeping the component specific building logic out of the Jenkinsfile, we can also use the same build and deploy actions locally if necessary. To define these actions, a tool called Make was used.

```
pipeline {
    agent any

    stages {

        stage('Checkout') {
            sh "make checkout"
        }

        stage('Build Docker image') {
            sh "make build"
        }

        stage('Push Docker image') {
            sh "make push"
        }

    }
}
```

Figure 12. Simplified Jenkinsfile with a declarative Pipeline for building and pushing a Docker container image.

Make is a tool for building source code to executable programs and libraries. Make can be used to abstract the details of building and installing a package. Make utilizes a file called a makefile, to read in declarative instructions. A makefile is a text file, which contains a set of directives for generating a target or a goal. Make is a language agnostic tool that can be used for various different activities in addition to just build packages. (Free Software Foundation, Inc., 2016)

Like many programs that follow the Unix philosophy, Make also does one thing and does it well. Because of this, Make is a versatile tool that that is still useful 40 years after its initial release. First released in 1976 by Stuart Feldman (Feldman, 1979), Make is still widely used in Unix-like operating systems particularly. Makefiles were created for each component with targets for building and pushing the Docker image, Figure 13.

```
REPOSITORY      := platform-master.local:5000
NAME            := ecom-project-frontend
GIT_COMMIT      := $(shell git rev-parse HEAD)
VERSION         := $(GIT_COMMIT)
BRANCH          := $(shell git rev-parse --abbrev-ref HEAD)
PROJECT_DIR     := $(shell pwd)
PROJECT_NAME    := $(shell basename "${PROJECT_DIR}")
IMAGE           := ${REPOSITORY}/${NAME}:${BRANCH}-${VERSION}
IMAGE_LATEST    := ${REPOSITORY}/${NAME}:${BRANCH}-latest

checkout:
    git checkout ${BRANCH}
    git pull

build:
    docker build -t ${IMAGE} ${PROJECT_DIR}
    docker tag ${IMAGE} ${IMAGE_LATEST}

push:
    docker push ${IMAGE}
```

Figure 13. Makefile for the Frontend component with targets for checking out latest changes, building and pushing a Docker image.

When new images are pushed to the Docker repository, the Helm chart references are updated to reference the newly created Docker images. Using the latest Helm chart, the Development environment is automatically updated with the latest Docker images. After a successful deployment to the Development cluster, a set of integration and acceptance tests are run, and the results presented in the Jenkins user interface. Provided that the tests pass on the Development environment, a Job for deploying to the Staging environment is started automatically. Similarly, to the Development Deploy Job, the Helm chart is deployed, and the tests run. Automatic tests are performed after each non-production Helm chart deployment, to ensure consistent platform behavior. At any given time, the latest version of the Helm chart may be deployed into a Production cluster by manually running the Production Deploy Job. Figure 14 illustrates the various Jenkins jobs and their relations to each other and to the relevant outside objects such as Docker registry and Kubernetes clusters.

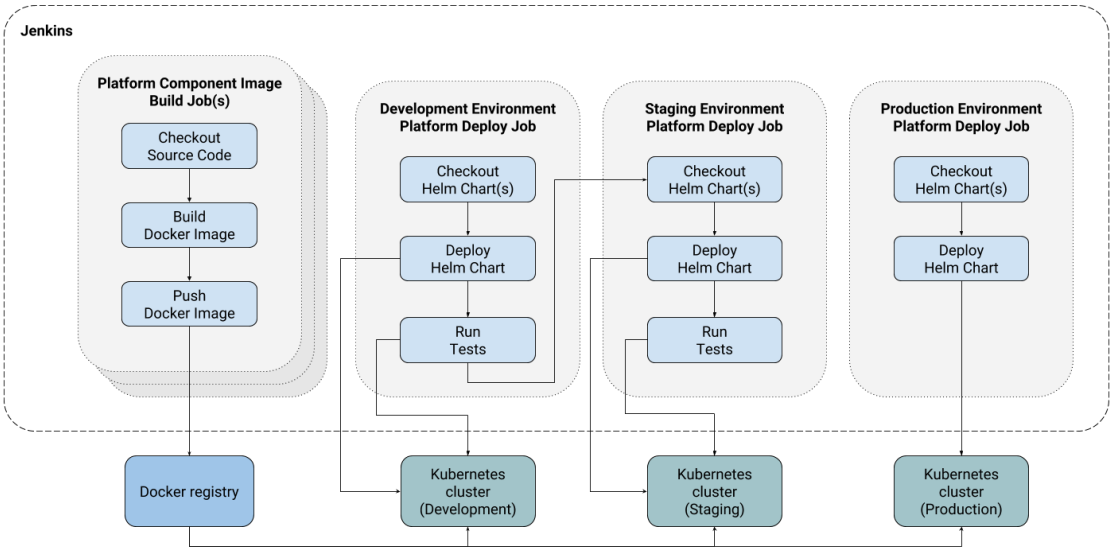

Figure 14. Jenkins Jobs and their relations to Docker registry and Kubernetes clusters.

5.3 Evaluation

This section looks in to the requirements that were set for the new design at the beginning of the study and evaluates their fulfilment in the final infrastructure design. The new design of the infrastructure provides improvements on the availability and scalability of the platform, while also making it easier to deploy to various environments and remain manageable when the number of environments rise.

The three layers of abstraction: containers in Kubernetes, a key-value store and a SQL database; provide the foundation for the new infrastructure. By building upon these commonly known and widely used technologies, the platform can be easily deployed to majority of environments as long as implementations of these layers are present. Since only these three layers are needed to be present on the environment, the complexity and outside requirements of the infrastructure stays low, which makes the management of the environments simpler.

Provided that the three abstraction layer implementations are available, and the implementations provide sufficient automatic scaling and high availability features, a reasonable level of availability and scalability can be expected. As long as the platform itself is able to operate reliably at the needed scale, the level of availability and scalability is

mostly dictated by the implementations of the abstraction layers that are in use. By utilizing managed service implementations from major cloud providers, one can assure a reasonable amount of reliability for many use cases.

The new infrastructure design is entrusted to provide reasonable amount of security to be used at scale in production and to handle sensitive user information without issues. The design relies heavily on open source technologies such as Docker, Kubernetes, Redis and PostgreSQL, that are widely used by large enterprise companies and have active communities. The practical security of the environments will be determined largely by their final implementations and cannot be evaluated within the scope of this study.

6 Conclusion

The goal of this study was to create a new overall infrastructure design for a proprietary e-commerce platform. The initial requirement for the new design was to improve the known shortcomings of the previous infrastructure. The previous infrastructure was simple in its design, lacking automatic scaling capabilities and redundancy layers that are commonly expected nowadays. Deploying the platform to new environments with varying characteristics was troublesome and time consuming. The previous infrastructure had served the platform well enough, but was starting to show its age in the quickly evolving field of IT.

To produce the new improved design, various solutions where evaluated from established best practices to the current trends in the field of IT. As a result of this evaluation, a set of technologies and practices were selected and refined into a proof of concept environment. The final design relies on three distinct building blocks: containers, a key-value store and a SQL database. Containers are orchestrated with Kubernetes, which can be considered an industry standard for enterprise container orchestration. In addition to the stateless containers running on Kubernetes, Redis was used as the key-value store for caching temporary data, and PostgreSQL as the persistent SQL database. The proof of concept environment was built using Ansible, with playbooks written for provisioning a Kubernetes cluster and other supporting software such as Jenkins and Docker registry. Moreover, Jenkins Jobs were created for building the Docker images and deploying the platform to Kubernetes. The proof of concept environment will serve as a base for future work to fully migrate the platform on top of the new infrastructure.

The new infrastructure design, which was produced as the result of this study, provides significant benefits compared to the current implementation when it is fully implemented. The new design provides an easy way to automate the scaling of the platform horizontally during fluctuations in the traffic. The availability of the platform is also increased by performing upgrades in a rolling manner and using liveness probes to make sure the running containers are functioning as intended. With the new design, the platform may be deployed to various environments with small effort, as long as the implementations for the three building blocks are in place. The use of managed services will be advisable in environment deployments for achieving a high level of availability and scalability with minimal maintenance effort.

As the next steps to advance the introduction of the new design, the platform will be fully implemented on top of the new containerized solution. At first, it will be used to deploy feature branches into disposable self-contained environments. After an adjustment period, during which the organization will get comfortable with the new infrastructure design, the new design can be introduced to new environment deployments, and possibly migrating existing ones on top of it as well after the design is considered mature enough.

The new design of the infrastructure could be further improved by introducing a centralized logging system, such as Fluentd (Fluentd Project, 2018), and a monitoring system such as Prometheus (Prometheus Authors, 2018). Complementing systems such as these are required to be in place before the design can be considered to be mature enough for a serious production use.

References

Amazon Web Services, 2018a. *Amazon ElastiCache- In-memory data store and cache.* [Online]
Available at: https://aws.amazon.com/elasticache/
[Accessed 19 July 2018].

Amazon Web Services, 2018b. *Amazon Relational Database Service (RDS) – AWS.* [Online]
Available at: https://aws.amazon.com/rds/
[Accessed 19 July 2018].

Atchison, L., 2016. *Architecting for Scale.* First Edition ed. s.l.:O'Reilly Media Inc.

Babich, W. A., 1986. Coordination for Team Productivity. In: *Software Configuration Managemen.* s.l.:Addison-Wesley, pp. 4-8.

Beda, J., 2014. *GitHub kubernetes/kubernetes.* [Online]
Available at:
https://github.com/kubernetes/kubernetes/commit/2c4b3a562ce34cddc3f8218a2c4d11c7310e6d56
[Accessed 2018].

Bondi, A. B., 2000. *Characteristics of Scalability and Their Impact on.* Ottawa, Ontario, Canada, WOSP2000: Second International Workshop on Software and Performance.

Canonical Ltd., 2018. *Linux Containers - LXC - Introduction.* [Online]
Available at: https://linuxcontainers.org/lxc/introduction/
[Accessed 2 September 2018].

Cleveland, S. & Ellis, T. J., 2014. Orchestrating End-User Perspectives in the Software Release Process: An Integrated Release Management Framework. *Advances in Human-Computer Interaction*.

Cloud Native Computing Foundation, 2018. *CNCF Cloud Native Interactive Landscape.* [Online]
Available at: https://landscape.cncf.io/grouping=landscape&landscape=certified-kubernetes-distribution&selected=rancher
[Accessed 15 October 2018].

Cloud Native Computing Foundation, 2018. *Helm - The Kubernetes Package Manager.* [Online]
Available at: https://helm.sh/
[Accessed 27 October 2018].

Datadog, 2018. *8 surprising facts about real Docker adoption.* [Online]
Available at: https://www.datadoghq.com/docker-adoption/
[Accessed 17 November 2018].

Docker Inc., 2018a. *Swarm mode overview.* [Online]
Available at: https://docs.docker.com/engine/swarm/
[Accessed 2 September 2018].

Docker Inc., 2018b. *Docker Engine release notes.* [Online]
Available at: https://docs.docker.com/release-notes/docker-engine/
[Accessed 2 September 2018].

Docker Inc., 2018c. *About storage drivers.* [Online]
Available at: https://docs.docker.com/storage/storagedriver
[Accessed 21 November 2018].

Eldridge, I., 2018. *What is container orchestration?.* [Online]
Available at: https://blog.newrelic.com/engineering/container-orchestration-explained/
[Accessed 2 September 2018].

Feldman, S., 1979. *ident.c.* [Online]
Available at: https://minnie.tuhs.org//cgi-bin/utree.pl?file=V7/usr/src/cmd/make/ident.c
[Accessed 28 October 2018].

Fluentd Project, 2018. *Fluentd | Open Source Data Collector | Unified Logging Layer.* [Online]
Available at: https://www.fluentd.org/
[Accessed 10 November 2018].

Free Software Foundation, Inc., 2016. *Make - GNU Project - Free Software Foundation.* [Online]
Available at: https://www.gnu.org/software/make/
[Accessed 28 October 2018].

Garen, K., 2007. Software Portability: Weighing Options, Making Choices. *The CPA Journal,* 77(11), pp. 10-12.

Google, 2018a. *Cloud SQL - MySQL & PostgreSQL Relational Database Service | Google Cloud.* [Online]
Available at: https://cloud.google.com/sql/
[Accessed 19 July 2018].

Google, 2018b. *Fully-managed in-memory data store service | Google Cloud.* [Online]
Available at: https://cloud.google.com/memorystore/
[Accessed 19 July 2018].

Google, 2018. *What are Containers and their benefits.* [Online]
Available at: https://cloud.google.com/containers/
[Accessed 19 August 2018].

HashiCorp, 2018a. *Releases - hashicorp/terraform GitHub.* [Online]
Available at: https://github.com/hashicorp/terraform/releases
[Accessed 8 September 2018].

HashiCorp, 2018b. *Introduction to Terraform.* [Online]
Available at: https://www.terraform.io/intro/index.html
[Accessed 8 September 2018].

HashiCorp, 2018c. *Introduction to Vagrant.* [Online]
Available at: https://www.vagrantup.com/intro/index.html
[Accessed 28 October 2018].

Hykes, S., 2018. *Docker 0.9: introducing execution drivers and libcontainer.* [Online]
Available at: https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/
[Accessed 2 September 2018].

Jenkins, 2018a. *Jenkins User Documentation.* [Online]
Available at: https://jenkins.io/doc/
[Accessed 13 October 2018].

Jenkins, 2018b. *GitHub - jenkinsci/job-dsl-plugin: A Groovy DSL for Jenkins Jobs - Sweeeeet!.* [Online]
Available at: https://github.com/jenkinsci/job-dsl-plugin
[Accessed 13 October 2018].

Jenkins, 2018c. *Pipeline.* [Online]
Available at: https://jenkins.io/doc/book/pipeline/
[Accessed 13 October 2018].

Jenkins, 2018d. *Using a Jenkinsfile.* [Online]
Available at: https://jenkins.io/doc/book/pipeline/jenkinsfile/
[Accessed 19 August 2018].

Marcus, E. & Stern, H., 2003. *Blueprints for High Availability.* 2nd ed. Indianapolis, Indiana: Wiley Publishing, Inc.

Microsoft, 2018a. *Azure Database for PostgreSQL—Fully Managed Service | Microsoft Azure.* [Online]
Available at: https://azure.microsoft.com/en-us/services/postgresql/
[Accessed 19 July 2018].

Microsoft, 2018b. *Azure Redis Cache | Microsoft Azure.* [Online]
Available at: https://azure.microsoft.com/en-us/services/cache/
[Accessed 19 July 2018].

Mooney, J. D., 2000. *Bringing Portability to the Software Process.* s.l., s.n.

Opara-Martins, J., Sahandi, R. & Tian, F., 2016. Critical analysis of vendor lock-in and its. *Journal of Cloud Computing.*

Portnoy, M., 2012. *Virtualization Essentials.* Indianapolis, Indiana: John Wiley & Sons, Inc..

Pressman, R. S., 2010. In: *Software Engineering: A Practioner's Approach, Seventh Edition.* New York: McGraw-Hill, pp. 584-586.

Proffitt, B., 2011. *Hudson devs vote for name change; Oracle declares fork | ITworld.* [Online]
Available at: https://www.itworld.com/article/2746627/open-source-tools/hudson-devs-vote-for-name-change--oracle-declares-fork.html
[Accessed 13 October 2018].

Prometheus Authors, 2018. *Prometheus - Monitoring system & time series database.* [Online]
Available at: https://prometheus.io/
[Accessed 10 November 2018].

Rancher Labs Inc., 2018. *Installation | Rancher Labs.* [Online]
Available at: https://rancher.com/docs/rke/v0.1.x/en/installation/
[Accessed 15 October 2018].

Rancher, 2018. *GitHub - rancher/rke: Rancher Kubernetes Engine, an extremely simple, lightning fast Kubernetes installer that works everywhere.* [Online]
Available at: https://github.com/rancher/rke
[Accessed 15 October 2018].

Red Hat, Inc., 2015. *Red Hat to Acquire IT Automation and DevOps Leader Ansible.* [Online]
Available at: https://www.redhat.com/en/about/press-releases/red-hat-acquire-it-automation-and-devops-leader-ansible
[Accessed 28 October 2018].

Red Hat, Inc., 2018. *Ansible Documentation.* [Online]
Available at: https://docs.ansible.com/ansible/latest/index.html
[Accessed 28 October 2018].

Saraswat, V., 2018. *Announcing Docker Enterprise Edition 2.0 - Docker Blog.* [Online]
Available at: https://blog.docker.com/2018/04/announcing-docker-enterprise-edition-2-0/
[Accessed 18 November 2018].

Sten Pittet, A., 2018. *Continuous integration vs. continuous delivery vs. continuous deployment.* [Online]
Available at: https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd
[Accessed 26 Autumn 2018].

The Kubernetes Authors, 2018a. *Picking the Right Solution.* [Online]
Available at: https://kubernetes.io/docs/setup/pick-right-solution/
[Accessed 18 November 2018].

The Kubernetes Authors, 2018b. *Understanding Kubernetes Objects.* [Online]
Available at: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/
[Accessed 20 October 2018].

The Kubernetes Authors, 2018c. *Pod Overview.* [Online]
Available at: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/
[Accessed 20 October 2018].

The Kubernetes Authors, 2018d. *ReplicaSet.* [Online]
Available at: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/
[Accessed 20 October 2018].

The Kubernetes Authors, 2018e. *Deployments.* [Online]
Available at: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[Accessed 20 October 2018].

The Kubernetes Authors, 2018f. *Services.* [Online]
Available at: https://kubernetes.io/docs/concepts/services-networking/service/
[Accessed 20 October 2018].

The Linux Foundation®, 2018. *About - Open Containers Initiative.* [Online]
Available at: https://www.opencontainers.org/about
[Accessed 26 August 2018].

The Linux Information Project, 2006. *The Linux Information Project.* [Online]
Available at: http://www.linfo.org/vendor_lockin.html

Tozzi, C., 2017a. *Docker at 4: Milestones in Docker History.* [Online]
Available at: https://containerjournal.com/2017/03/23/docker-4-milestones-docker-history/
[Accessed 2 September 2018].

Tozzi, C., 2017b. *Understanding Why Docker is So Popular.* [Online]
Available at: https://containerjournal.com/2017/05/09/understanding-why-docker-popular/
[Accessed 4 November 2018].

Vaughan-Nichols, S. J., 2018. *What is Docker and why is it so darn popular?.* [Online]
Available at: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/
[Accessed 17 November 2018].

Wilder, B., 2012. *Cloud Architecture Patterns.* 1st ed. s.l.:O'Reilly Media, Inc..

Wittig, A. & Wittig, M., 2016. *Amazon Web Services in Action.* s.l.:Manning Publications Co..

Zukanov, V., 2018. *Rolling Deployment: What This Is and How it De-Risks Software Deploys - Rollout Blog.* [Online]
Available at: https://rollout.io/blog/rolling-deployment/
[Accessed 13 October 2018].

Jenkins Job DSL definition (Jenkinsfile)

```
def projects = [
    [
        name: 'base',
        title: 'Base',
        git: [
            url: "http://git.example.com/base.git",
            branches: ['master']
        ]
    ],
    [
        name: 'web',
        title: 'Web',
        git: [
            url: "http://git.example.com/web.git",
            branches: ['master']
        ]
    ],
    [
        name: 'api',
        title: 'API',
        git: [
            url: "http://git.example.com/api.git",
            branches: ['master']
        ]
    ],
    [
        name: 'frontend',
        title: 'Frontend',
        git: [
            url: "http://git.example.com/frontend.git",
            branches: ['customer1', 'master']
        ]
    ],
    [
        name: 'cache',
        title: 'Cache',
        git: [
            url: "http://git.example.com/cache.git",
            branches: ['master']
        ]
    ],
    [
        name: 'database',
        title: 'Database',
        git: [
            url: "http://git.example.com/database.git",
            branches: ['master']
        ]
    ],
    [
        name: 'backend',
        title: 'Backend',
        git: [
            url: "http://git.example.com/backend.git",
            branches: ['master']
        ]
    ]
]

//Docker build jobs
for (project in projects){
    multibranchPipelineJob("${project.title}") {
        branchSources {
            git {
                remote("${project.git.url}")
                credentialsId('66b88263-e4bf-48e8-ad5b-359f19d41998')
            }
        }
        orphanedItemStrategy {
            discardOldItems {
                numToKeep(20)
            }
        }
    }
}
```

```
//Platform deploy job helm
job('Deploy Platform') {
    multiscm {
        git {
            remote {
                url("http://git.example.com/cicd.git")
                credentials('66b88263-e4bf-48e8-ad5b-359f19d41998')
            }
            extensions {
                relativeTargetDirectory('cicd')
            }
        }
        git {
            remote {
                url("http://git.example.com/frontend.git")
                credentials('66b88263-e4bf-48e8-ad5b-359f19d41998')
            }
            extensions {
                relativeTargetDirectory('frontend')
            }
        }
        git {
            remote {
                url("http://git.example.com/api.git")
                credentials('66b88263-e4bf-48e8-ad5b-359f19d41998')
            }
            extensions {
                relativeTargetDirectory('api')
            }
        }
        git {
            remote {
                url("http://git.example.com/backend.git")
                credentials('66b88263-e4bf-48e8-ad5b-359f19d41998')
            }
            extensions {
                relativeTargetDirectory('backend')
            }
        }
        git {
            remote {
                url("http://git.example.com/database.git")
                credentials('66b88263-e4bf-48e8-ad5b-359f19d41998')
            }
            extensions {
                relativeTargetDirectory('database')
            }
        }
        git {
            remote {
                url("http://git.example.com/cache.git")
                credentials('66b88263-e4bf-48e8-ad5b-359f19d41998')
            }
            extensions {
                relativeTargetDirectory('cache')
            }
        }
    }

    steps {
        shell('cd cicd && make deploy_latest')
    }
}
```

Jenkins Job definition (Jenkinsfile)

```
def label = "mypod-${UUID.randomUUID().toString()}"
podTemplate(
        label: label,
        containers: [
            containerTemplate(
                name: 'jnlp',
                image: 'adriagalin/jenkins-jnlp-slave',
                args: '${computer.jnlpmac} ${computer.name}'
            )
        ],
        volumes: [
            hostPathVolume(
                mountPath: '/var/run/docker.sock',
                hostPath: '/var/run/docker.sock'
            ),
            hostPathVolume(
                mountPath: '/code',
                hostPath: '/tmp/data01'
            ),
        ]
    ) {
    node(label) {

        currentBuild.result = "SUCCESS"

        stage('Checkout') {
            checkout([
                 $class: 'GitSCM',
                 branches: scm.branches,
                 extensions: scm.extensions + [[$class: 'LocalBranch', localBranch: "**"]],
                 userRemoteConfigs: scm.userRemoteConfigs
            ])
        }

        stage('Build Docker image') {
            sh "cd ${WORKSPACE}; make build"
        }

        stage('Push Docker image') {
            sh "cd ${WORKSPACE}; make push"
        }

    }
}
```