Master's thesis

Master's Programme in Computer Science

Near-zero downtime deployment of uWSGI applications

Johannes Lares

December 18, 2024

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Johannes Lares

Työn nimi — Arbetets titel — Title

Near-zero downtime deployment of uWSGI applications

Ohjaajat — Handledare — Supervisors

Senior University Lecturer Matti Luukkainen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	December 18, 2024	54 pages, 4 appendix pages

Tiivistelmä — Referat — Abstract

Having near-zero downtime application updates and near-zero application downtime server maintenance is an emerging standard with web applications. Providing user experience with no disruptions leads to better user satisfaction. Container orchestrators, such as Kubernetes and Docker Swarm, provide functionality and features for application operators to achieve near-zero downtime deployments with containerized applications. Near-zero downtime deployments can lead to high availability, where the deployed application can tolerate some faults without causing application downtime.

This thesis compares the infrastructure costs of two major container orchestration tools for cluster deployments - Kubernetes and Docker Swarm - and figures out if modifications to application architecture are required for replicated deployments. The application used in the research is B2SHARE, a containerized uWSGI application for research data management. Comparisons are made against a baseline deployment of B2SHARE done with Docker Compose.

It is proved, that achieving near-zero downtime deployments with B2SHARE is possible with both container orchestration tools with relatively similar costs, Docker Swarm being a bit cheaper. The implementations did not require any modifications to the application code. It is argued, that based on features and functionality, Kubernetes is a better option.

Deployment configurations and guides, measurement scripts, and raw measurement data are published in B2SHARE: 10.23728/b2share.0ab94117e43b45baa64773625b52fa33

ACM Computing Classification System (CCS)
Computer systems organization → Dependable and fault-tolerant systems and networks → Availability
Computer systems organization → Architectures → Distributed architectures → Cloud computing

Avainsanat — Nyckelord — Keywords

software deployments, DevOps, clusters

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

In the current era of information systems and web applications, minimizing the downtime of software is one of the top priorities for big software companies to enhance user satisfaction. Downtime - the time when an application is not usable - can cause loss of revenue, loss of customers, loss of customer satisfaction etcetera. Users assume that a web application is reachable 24 hours a day, seven days a week. Thus, near-zero downtime of web applications is nowadays almost an industry standard [11, 29].

Downtime can be caused by several different methods. Planned downtime is caused by maintaining applications and underlying hardware. Unplanned downtime is caused by faults in the application or hardware [20].

Previous research exists for minimizing application downtime during application updates for containerized and non-containerized applications. The consensus is to have blue-green deployments, where a new instance of an application is brought up before the old deployment is terminated [55]. For minimizing application downtime during server maintenance, cluster deployments are the industry standard [25, 28, 39].

As a starting point, a B2SHARE - a containerized uWSGI application - deployment causes approximately 50 seconds of downtime and server maintenance causes anywhere from 10 minutes to 2 hours of downtime. Due to requirements from the service level agreement, planned downtime has to be announced one week before. Thus, applying changes, such as bug fixes, new features, and operating system updates happens approximately once a month during a service break. For streamlining application deployments and server maintenance, downtime has to be brought down to near zero to remove the requirement of having a service break.

In this thesis we'll do research if B2SHARE can be updated with near-zero application downtime, and if the application can be deployed into a cluster for near-zero downtime server updates and for high availability, and if the near-zero application downtime server maintenance is possible with the cluster implementations. The application has dependencies in the form of other containers and external servers.

The research consists of two different implementations of container orchestration tools for clusters: Docker Swarm and Kubernetes. With the implementations, we try to answer

three research questions:

- (RQ1) What architecture and infrastructure changes are required for near-zero downtime deployment of the application and aforementioned dependencies?
- (RQ2) What changes are required and is there a possibility to do operating system updates and reboots without application downtime for customers?
- (RQ3) What is the cost of the different deployment methods?

Due to the nature of B2SHARE being a publicly funded platform, the cost-effectiveness of the implementation is an important aspect. In general, a replicated deployment of an application causes costs for multiple reasons. These reasons include infrastructure, operations, and modifications to the application for replication support. In this thesis, the infrastructure costs are the only interest.

B2SHARE as a platform for publishing research datasets relies heavily on data. Thus, deploying the application into a cluster requires a good system for managing and sharing data between instances in the cluster. For the research implementations, a multi-attach disk was used, but this thesis presents other suitable methods for data management for distributed computing as well.

The research is done following the principles of Design Science Research (DSR). DSR is a cyclic research method, which aims to solve relevant problems via building design processes and artifacts and evaluating them [19, 18]. The research process consists of four design cycles.

For measuring the application downtime during application updates and server maintenance, a simple Python script was implemented. This script sends an HTTP request to the application every 0.5 seconds and logs the response status code. The status code defines, if the application is running and usable, or if the application has downtime and is out of reach.

This thesis has the following structure: chapter 2 presents previous work and theory behind containers, downtime, clusters, and other required knowledge. After that, in chapter 3 Design Science Research is described as a research method and the implementation of the selected cluster methods is briefly explained. This chapter presents the measurement methods to validate if an implementation has reached near-zero downtime application updates and near-zero application downtime server maintenance.

In chapter 4 the measurements are analyzed and the research questions are answered. Chapter 5 compares the results to previous work and to the presented theory in chapter 2. Chapter 5 also proposes best practices for achieving near-zero downtime application updates and near-zero application downtime server maintenance based on the implementations, research, and previous work.

This thesis was contracted and made in collaboration with the main developer of B2SHARE: CSC - IT Center for Science Ltd.

2 Background

This chapter goes through the theory behind near-zero downtime updates and server maintenance. First, downtime and why near-zero downtime deployments are important are explained in the section 2.1. Section 2.2 explain what containers are and how containers are used. Section 2.3 describes different methods of deployments concerning high availability, and fault tolerance and sets some minimum requirements for zero downtime deployment systems. Section 2.4 describe what clusters are, how clusters can be used with container deployments, and what is the role of clusters in near-zero downtime deployments. Different data storage methods for distributed systems are described in the section 2.5. The architecture of B2SHARE is introduced in section 2.6.

2.1 Downtime

Downtime is the time, when a service, such as a web application, is not usable. Downtime does not only concern information systems but concerns also other domains. Critical infrastructure, such as power grids can have downtime in the form of blackouts. Downtime in information systems usually causes user dissatisfaction, monetary harm, or reputational damage [4, 11], but in some cases can lead to life-threatening consequences [29].

Downtime in information systems can be caused by many reasons, for example: the application is not running, the service is in maintenance mode (some functionality might exist), the instance, where the application runs, is not running etcetera. In some cases, downtime can be planned or scheduled and is informed to the customer or end user. In other cases, downtime can be unplanned [20, 4].

Server maintenance, application updates and deployments, and other maintenance are usually the cause of planned downtime. Other maintenance, that can cause planned downtime can be infrastructure changes and updating services that the software relies on (e.g. database) etcetera. Faults, bugs, and other disruptions that are not foreseeable by the application or environment administrators cause unplanned downtime. Faults and disruptions can be for example network outages, power outages, and firmware crashes. Human error can also lead to unplanned downtime [20].

Downtime is usually measured in percentages, which measure the amount of downtime during a specified time interval. The percentage is often in the form of up-time and the measuring interval is one year [20]. Applications can have service-level agreements or operation-level agreements, where a goal for up-time is set. If a service has a goal of 99% up-time in a year, the service can have 3,65 days of downtime, if a service has 99,999% up-time in a year, it can have 5,26 minutes of downtime etcetera.

B2SHARE has a service level agreement, which defines availability to 97%, which means 10,96 days of acceptable downtime in a year. However, between April 22nd and July 22nd 2024, the amount of known downtime was 0,002%, which was 2 minutes 11 seconds and 0,040% of undetermined status, which was 53 minutes and 10 seconds. The amount of unscheduled downtime was 0%. Exceeding the agreed maximum amount of downtime can lead to sanctions for the service provider.

Application downtime can cause customer dissatisfaction towards the service. This can lead in some cases to loss of revenue due to loss of customers. Thus, planned downtime is usually planned for weekends, nights, or after hours, when service usage is expected to be minimum [4]. However, doing maintenance at these times costs more compared to doing it at regular work hours, due to the extra pay required for developers. For this reason, in the B2SHARE development team, downtime is planned at normal business hours. Downtime is planned for 4 hours every three weeks.

With applications that directly bring revenue to a company, the cost of downtime can be calculated. However with B2SHARE, loss of revenue is not measurable directly, because the software is free to use and the development is funded from different sources, for example EU and the Ministry of Education and Culture. However, loss of users can lead to less funding.

The second major aspect of non-zero downtime deployments and instance maintenance is the time it ties developers and managers to a service break. This time can be anywhere from 10 minutes to multiple hours and usually includes multiple people. Thus, reducing the time deployments and maintenance take from the team reduces the cost and brings more time for other relevant tasks.

2.2 Containers and virtualization

Containers are a lightweight, scalable, and portable method of virtualization, that is a fully contained package of an application, its dependencies, and usually its configuration [37, 27]. However, the configuration can be passed to the container when it's started. When a container is built from an application, a container image is created. Containers are usually cross-platform compatible or at least a container can be built for different platforms enabling easy deployment of an application to different instances [3, 54]. Built containers can be uploaded into a container registry, from which containers can be downloaded as needed. Containerized applications are nowadays almost the industry standard for deploying applications [16].

Another common method for virtualization is using virtual machines (VMs) [37]. A virtual machine is allocated hardware on a physical instance, such as a server. Thus, a virtual machine is a virtualized instance. Compared to containers, a virtual machine needs an operating system and VMs are not as lightweight, as containers are [36].

A container has a lifetime and a life cycle. During the startup, the container initializes itself and starts the application. After the startup, the container goes to a running state. After the container has finished running or an operator wants to stop the container, the container goes into a termination state, in which the container gracefully stops the processes. If an operator wants to remove the container immediately or a system error, such as out of out-of-memory error, happens the container goes to a killed state. After termination or killed state, the container is removed. If a container fails to start or the container terminates unscheduled, the container engine usually tries to restart the container multiple times [22, 37]. With B2SHARE, the container life cycle starts with approximately a 40-50 second initialization period and at the end of that period, uWSGI workers are spawned to handle connections leading the container to a running state. When a B2SHARE container is terminated, ongoing requests are finished and the uWSGI workers are killed. After that, the B2SHARE container can be removed.

Container deployments have three layers of software. For building and deploying containers it is important to know the difference between these layers. On the lowest level, a container runtime (for example runc [40]) is the software, that builds and runs containers. On top of the container runtime is a container engine (for example containerd [53]), which provides an API for container managers and orchestrators to interact with the container runtime. On the top level container managers and orchestrators (for example Docker

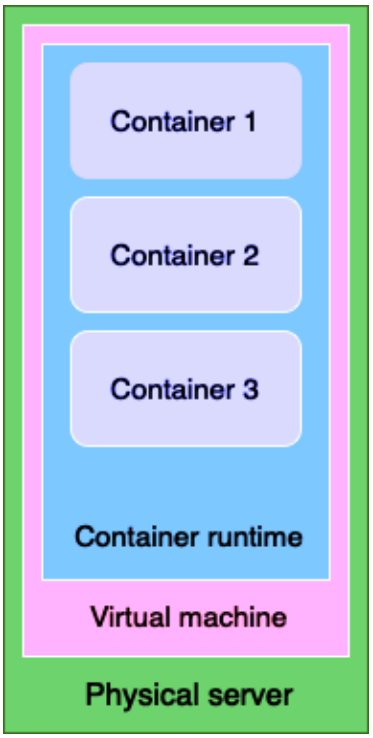

Figure 2.1: The architecture of a container deployment in a virtual machine environment. The virtual machine runs on a physical server and the VM is used to run the container runtime.

and Kubernetes [48, 22]) interact with the container engine providing interfaces for a user to manage containers [38, 53]. A container engine or a container manager is not required to run containers and a user can give commands directly to the container runtime [40]. Container runtimes, container images, and container distribution are standardized by Open Container Initiative (OCI) [34].

When deploying a container, it needs a host operating system and with cloud applications, the host is usually a virtual machine or a container cloud. On top of the host operating system, there is a container runtime, that is used to run the containers. Running containers can have limited access to the host operating system resources, such as disk [37]. A container deployment architecture on a virtual machine is described in figure 2.1.

Docker is a popular container manager and Docker uses containerd container engine to interact with the runc container runtime [43, 48]. Kubernetes is another common container orchestration tool and it is used widely in clustered container deployments, containerized cloud deployments, large-scale deployments, and high availability deployments [44, 12, 27]. Kubernetes can be configured to use any container runtime that supports Kubernetes Container Runtime Interface (CRI) [22].

Deploying and updating containerized applications have some variability depending on the orchestration tool used and the configuration of the container deployment. Docker provides commands to manage single containers and sets of containers. Docker, which is a command to manage single containers, and Docker compose, which is a command to manage a set of containers, does not have support for rolling updates, and needs manual or automated commands for updating the containers on a deployment. The update usually happens in the order of stopping the old container, pulling or building a new container, and starting the new container [48]. Docker Swarm and Kubernetes support rolling updates, where the update of a container is done with one command and depending on the configuration usually follows blue-green deployment [22]. Docker Swarm is a tool provided by Docker to manage a cluster of instances running containers [26].

A blue-green deployment does not cause downtime. For the update, a new version of the container is started. This version is the green deployment. The orchestration tool monitors the green version and when the green version is ready, connections are routed to this deployment of the application. After the traffic has been routed, the old version of the deployment is terminated. The old version of the application is the blue deployment [55].

Container orchestrators provide methods for automating application updates. The application deployment configurations can be stored in version control such as git. When the deployment configuration is updated, the deployment is automatically updated by the container orchestrator by polling the git repository or by the git repository pushing the deployment to the orchestrator. The git-based deployment handling is called GitOps. GitOps and other forms of deployment automation reduce the amount of manual work needed for administering the deployments [2].

2.3 High availability

High availability means, that the application deployment can withstand some malfunction in the application environment and that the application is still usable when some containers are not running [44]. An application has reached its high availability target when the uptime requirement is met [25]. For B2SHARE this target is 97%.

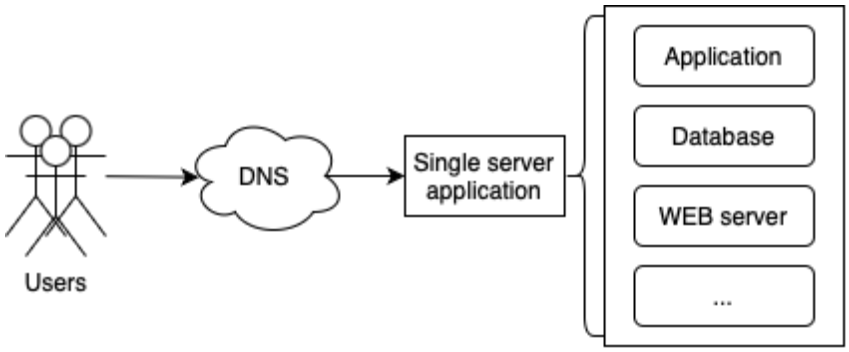

Figure 2.2: Single server deployment of an application without replication

2.3.1 Methods of high availability

There are many different methods for achieving high availability [25] and the methods are divided into levels of high availability. Each level brings more robustness for faults in the environment but also adds costs. The levels are not standardized but introduced in this thesis to be clearer for the reader.

The first level is no high availability at all. The application stack is deployed to one server and there is no replication. This deployment is vulnerable to many different faults. For example, the application is not usable, if there is a fault in the host machine if there is a fault in the application and it crashes, if the network goes down, etcetera. An example of this kind of deployment is described in figure 2.2. This deployment is the cheapest one because only one host machine is required.

The second level of high availability can tolerate faults in the application, but not in the host machine or network. In this kind of deployment, the application is replicated in the host machine, for example, multiple containers of the same application are replicated, thus, if one crashes, the other ones are still running. This deployment method is described in figure 2.3. This method has similar infrastructure costs compared to the first level, but deployment costs can be higher because the application has to be able to be replicated making the application more complex.

The third level of high availability has multiple servers or a cluster running the application and data is somehow shared or synchronized between these servers. This kind of deployment can tolerate faults in $n - 1$ servers, where n is the amount of servers running the application stack. To route connections to the servers, a load balancer is used. This method is described in figure 2.4. This method of deployment can not tolerate faults with the load balancer, except if another server can take over as a load balancer. This method

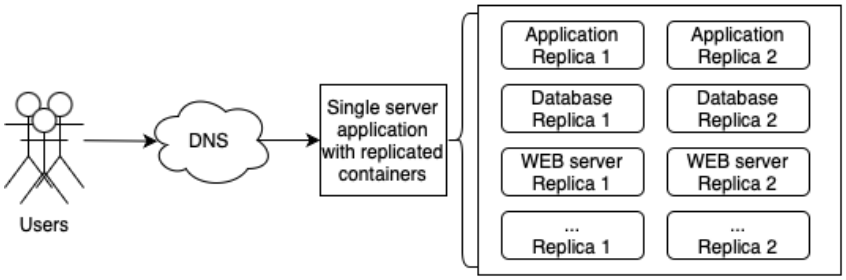

Figure 2.3: Single server deployment of an application with replication

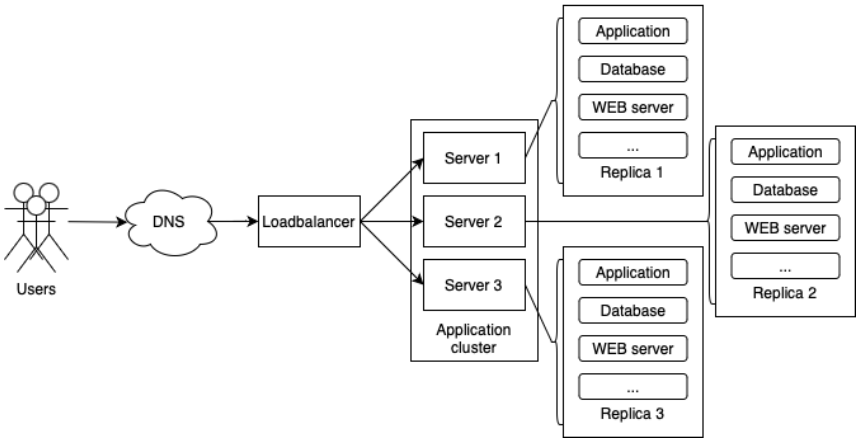

Figure 2.4: Application deployed to multiple servers. To route connections to servers, a load balancer is used. DNS can be used as a load balancer.

is more expensive than levels one and two because more infrastructure is required [25].

Load balancing can be done with the Domain Name System (DNS) removing the requirement for a separate load balancing system [42]. A common way for doing load balancing with DNS is round-robin load balancing, where IP addresses are returned from the DNS in rotating order [8]. DNS load balancing can usually handle scenarios, where one or multiple target servers are down, but it's up to the user's browser to go through the list of returned IP addresses from DNS until a working IP address is found.

The fourth level of high availability is to run servers in multiple different data centers in different locations. This method ensures, that a fault in one data center does not affect the usability of the application. This method is complex to implement and the most expensive implementation of high availability.

One of the major questions when replicating applications is how the data is managed. If multiple servers are running the same application, the data has to be shared or synchronized between the servers almost instantly so as not to affect the usability of the

application. One possibility is to use a separate server for storing the data, but then the application can not tolerate faults with that server. Another possibility is to use an NFS (network file system) to store data on a file system connected to a network, and the file system is mounted to all the application machines. Managing data when the application is replicated across multiple physical locations brings more challenges. Distributed data management is described better in section 2.5.

2.3.2 High availability for achieving zero downtime

For achieving zero downtime application deployments, at least level two of high availability is required. To achieve a zero downtime deployment, at least one replica of the application stack has to be running all the time [55]. Thus, during deployment, a new stack is brought up, connections are redirected there and the old stack is removed or only the application is updated with this process, and the dependencies are left running. If the application is replicated and an update includes backward incompatible changes, for example, database schema changes, all the replicas of the application should be updated at the same time to prevent issues where the not updated application tries to do operations, that do not work anymore.

To achieve zero downtime server updates and reboots, at least level three of high availability is required to ensure, that one server running the application can be offline during the maintenance [39]. This however creates an issue during the updates of the load balancer. To ensure, that the application is usable when the load balancer is updated there has to be at least one other server, that can function as a load balancer, which takes over during the maintenance of the primary load balancer. This issue can be avoided by using DNS load balancing [5], but it is hard to predict how the DNS load balancing routes traffic for different users.

The most used way to achieve zero downtime deployments is to use level three of high availability with two or more servers running the application [25]. For achieving zero downtime, it is important, that the environment can handle automatically fault and does not require manual operations to bring the application back up to a usable state, when a fault occurs or when a new version of the application is deployed. Usually, a tool, such as Kubernetes or Docker Swarm is used to manage the cluster and replication of the containers [44]. These container orchestration tools have automatic systems to replace containers, that are terminated due to a fault [48, 22].

The state of high availability of an application depends on the weakest link of the whole application and its dependencies and the hardware it's running on. Thus, creating a cluster of the application and dismissing a service that the application is dependent on, such as a database, creates a situation, where the state of high availability is lowered due to shortcomings of that service. To achieve levels three and four of high availability, all the application-dependent services and the application itself have to be replicated. As mentioned before, a weak link can be the underlying hardware. If all the instances are in the same data center and a fault occurs, the application goes down.

2.4 Clusters

A cluster is a set of multiple servers or virtual machines that can be used for distributed computing, data storage, and achieving high availability. As mentioned in the section 2.3, to achieve a fault-tolerant deployment that can handle downtime of an instance, a cluster of instances is needed.

Containers can be run in a cluster and a container orchestration tool can be used to deploy and manage the containers in a cluster and to manage the cluster to some level. Running containers in a cluster can have many benefits. With application deployments with many containers, the workload is shared between the instances in the cluster and the containers can be replicated to tolerate faults. If an application faces a lot of usage, the application can be scaled in a cluster, which means that the application container is replicated to other instances in the cluster to balance the workload. Some container orchestration tools can do scaling automatically. Scaling application to multiple instances is called horizontal scaling [22].

Running applications in a cluster adds complexity to the deployment. The application deployment has to be able to be distributed and if the goal is to have high availability, the application has to be able to be replicated. For applications, that rely on storage, the data management has to be planned well and implemented so that all replicas of the application have access to the data.

Clusters can be used for deploying one application and its dependencies or a cluster can be used to deploy multiple applications and service providers can sell part of the cluster as a deployment platform. If the cluster is used to deploy multiple container applications, the cluster can be called a container cloud [30]. Kubernetes is becoming the industry standard for container clusters [47, 53], but others exist, for example, Docker Swarm and HashiCorp

Nomad [48, 31]. With container clusters, the master nodes or control plane nodes have the important part of spreading the workload between worker nodes and monitoring the running containers for faults [22, 48].

When a container is deployed into a cluster, in most cases a specific target instance is not specified. The master node deploys the container to the most suitable instance. This ensures, that the target node has enough computational capacity to run the container. However, it is up to the container orchestrator to select the best target instance and to handle replication of the deployed containers [22, 48].

2.5 Storage for distributed systems

Storage for distributed systems or distributed file systems, such as Network File System (NFS) [41], Storage Area Network (SAN) [52] and GlusterFS [15] is a crucial part of any distributed system with data-related applications [6].

A distributed file system is a system, from which data can be accessed at least almost simultaneously as well as data can be written almost simultaneously [24]. There should not happen read errors or data corruption with these file systems and the performance of the file system should not decrease significantly with the amount of tenants to the system [7].

Filesystems for distributed computing can be roughly divided into two categories: Filesystems that are distributed and can be used as storage for distributed computing (category one), and filesystems that function like protocols and do not provide data distribution by themselves (category two). If data in a category two system has to be distributed, the distribution happens with other methods, such as RAID.

A multi-attach storage device is an attachable storage volume that can be attached to multiple systems at once also known as shared storage distributed file system [32]. For a multi-attach volume to work correctly, a clustered file system is required to prevent simultaneous read and write operations to a file. These file systems coordinate the file operations between the nodes to which the volume is attached. The coordination can be for example fencing, where the file system management daemon blocks other nodes for doing file operations with a file, when the blocking node is doing file operations with that file [46]. Many cluster file systems exist, for example, Global File System (GFS2) [14], Oracle Cluster File System (OCFS) [33], and VMWare Virtual Machine File System

(VMFS) [51]. A multi-attach storage device is like a category one filesystem, but it does not distribute data by itself.

A Network File System is a protocol for doing file system operations on a remote storage device. The protocol is stateless; a request contains all the information required for the storage service to do the file operations [41, 17]. NFS does not provide safeguards for file operations, but with one storage server with multiple tenants, the storage server's operating system usually handles the locking and unlocking of the files. Thus, NFS itself is not a distributed file system, but the protocol can be used for distributed computing systems or replicated systems for data management. NFS is a category two filesystem protocol.

A Storage Area Network is a network environment, where the storage connections are handled outside of the local area network (LAN). SAN can be used to connect multiple storage devices to multiple tenants and the storage devices can be mounted to behave as local disks. SAN itself does not do any safeguarding of the data, because SAN is by definition a hardware implementation, but when speaking of SAN devices and volumes in distributed file systems, the system also includes SAN management software, that is used for managing simultaneous file operations, replication, access control etcetera. Separating storage access from other network traffic is one of the major benefits of SAN [52, 7]. SAN is a category two system.

GlusterFS is a storage system where multiple storage servers are connected to a single trusted storage pool. Each connected storage server can have one or multiple storage devices attached. This system creates a Gluster volume, which can be attached to multiple tenants. To coordinate simultaneous operations to file, a management daemon is used to prevent data corruption and other errors. Thus, GlusterFS does not require an additional cluster file system to manage the file operations [15]. Longhorn is a similar distributed filesystem for distributed computing designed especially for usage in Kubernetes clusters [23]. GlusterFS and Longhorn are category one filesystems.

The presented distributed file systems are not the only distributed file systems existing. Many other distributed file systems exist, for example, Server Message Block (SMB) (category two) [45] and Simple Storage Service (S3) (category two) [1]. Many of these file systems are only protocols for transferring data from storage devices to tenants and these protocols do not do any safeguarding of the data [17]. Thus, the functionality provided by the operating system or a management daemon is required for handling concurrent file operations [46].

2.6 B2SHARE

B2SHARE is a repository for publishing research data. B2SHARE is a web application with a Python backend and a React frontend. The Python backend is based on the Invenio framework that is built with Flask. The backend is served with uWSGI. uWSGI is an application server for enabling applications to function as web servers [49].

The B2SHARE application runs in containers and the container stack is deployed with Docker Compose. The container stack is described in section 2.2. The container stack is very diverse regarding resource usage. For example, Elasticsearch and RabbitMQ require a lot of memory, running batch operations with Celery takes a lot of CPU processing power and the B2SHARE application requires a lot of disk space for the published datasets and disk i/o operations for writing and fetching the datasets.

The B2SHARE architecture consists currently of multiple containers that run on the same container stack. The application has other dependencies, that run on other servers. A list of B2SHARE production deployment containers with descriptions is listed in table 2.1 and the application architecture is described in figure 2.5.

Remote B2SHARE dependencies are a database server, a proxy server, and a vault. Each of these dependencies has requirements, when they have to be running. The database server is required when the application is running, while the Vault and proxy are required only when the application is starting. Vault is used to store secrets in a key-value format and the proxy server is used to contact services that are behind a firewall, such as an internal container registry. For user authentication in the application, a separate authentication provider is needed, but the connection is required only when a user is trying to authenticate.

A minimum containerized B2SHARE deployment does not require Flower or Vault agent to run and API, files, and Celerys are combined into one B2SHARE container. Secrets can be managed as environment variables and Flower is used only for monitoring and debugging of Celery.

The B2SHARE deployment containers are managed with Docker compose, which is a tool, where a set (a stack) of containers are defined and configured in a YAML file. Docker compose provides commands to run these configured container stacks and the commands are used to run and manage the B2SHARE container stack. Defining the containers in the YAML file means which container image to use and run, and how these containers are

Container	Description
Application API	Handles API requests
Application Files	Serves files
Nginx	Web server, routes requests to API/files
Vault agent	Fetches secrets from Vault instance
Redis	Memory cache for sessions etc.
ElasticSearch	Search engine
RabbitMQ	Message queue to queue background tasks
Celery beat	Schedules background tasks
Celery worker	Runs background tasks
Flower	A debug tool for Celery

Table 2.1: Containers running on a B2SHARE instance. Application API and Application Files are duplicates of the B2SHARE container to reduce the number of processes running on a single container. The web server NGINX is used to route connections to API or files container based on the HTTP request URL.

configured. The configuration of a container can be environment variables, disk mounts, network port mappings, definition of secrets etcetera. All containers listed in table 2.1 are defined, configured, and run with Docker compose. The containers interact with the HTTP protocol in the local container network on the virtual machine.

The container stack runs on a virtual machine that is either a VMWare or a cPouta instance. VMWare is used for production instances and cPouta for development and testing instances. Currently, three production B2SHARE deployments are running on separate virtual machines. These deployments are a public B2SHARE, a public B2SHARE for training and testing purposes, and a premium B2SHARE deployment for a customer. For development purposes, there are a lot of cPouta virtual machines with B2SHARE running. Development instance configurations are usually kept as close as possible to production instance configurations for easier debugging of production deployment faults.

The current B2SHARE deployment uses out-of-date dependencies and Python modules. Most of the dependencies have reached the end of life, which means, that issues found on these dependencies are not fixed anymore. For example, a B2SHARE deployment uses ElasticSearch version 2.4.6, which reached its end of life in 2018 [10]. The B2SHARE development team has noticed from preliminary tests, that B2SHARE system requirements can be lowered if some of the outdated dependencies are updated. For example, the amount

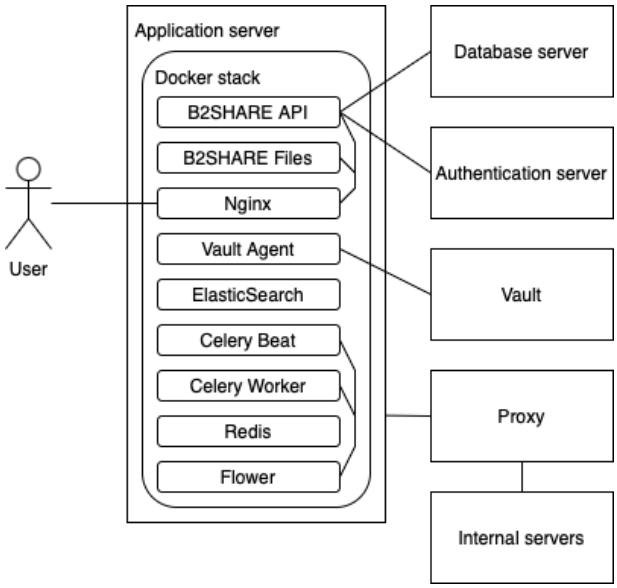

Figure 2.5: B2SHARE container architecture and external services are required to run the application. Connections between the B2SHARE API container and other containers are left out to make the diagram easier to read.

of memory required can be brought down by updating RabbitMQ from 3.8 to 3.9. Some of the update processes however require updating some of the B2SHARE modules, which required the update of other modules and so on.

A virtual machine running a B2SHARE container deployment for development purposes most commonly has 4 CPU cores and 7.8 GiB of memory.

3 Methods

This chapter describes the methods used in the research. Section 3.1 begins with an explanation of Design Science Research and continues by explaining the research process regarding implementation and briefly explains, how different implementations are evaluated. For easier understanding and referencing, each implementation and possible implementation is labeled. In this chapter, a high availability implementation or a high availability deployment means level three of high availability defined in section 2.3.

In this and following chapters units of gibibyte (GiB) and tebibyte (TiB) are used. One GiB is 1024^3 bytes (1073741824 bytes). One TiB is 1024^4 bytes.

For this and the following chapters, zero downtime means downtime that can not be noticed (under half a second) and near-zero downtime means downtime, that is less than the normal application update time of approximately 50 seconds.

3.1 Design Science Research

Design science research is a cyclic method for researching different implementations of information science. It is intended to create and evaluate new artifacts meant to solve organizational problems. The artifacts should be defined and described in a way, that the implementation of the artifacts is possible. These artifacts can be for example models, methods, and instantiations. The evaluation of an artifact should prove the quality, utility, and efficiency of the artifact. Design science research method assumes, that the problem that is researched is important for the organizational point of view [19]. From the point of this thesis research, the goal is to implement methods and instantiations that should lead to near-zero downtime deployments of B2SHARE.

Design science research cycle consists of two main stages: implementation, and evaluation. The cycle and Design science research are described in figure 3.1. During the implementation phase, an artifact or a process is implemented. In the evaluation phase, the created implementation is evaluated[19].

Nowadays more and more DevOps teams work with some kind of cyclic process, such as scrum [35]. This is the case with the B2SHARE development team. Thus, a cyclic research

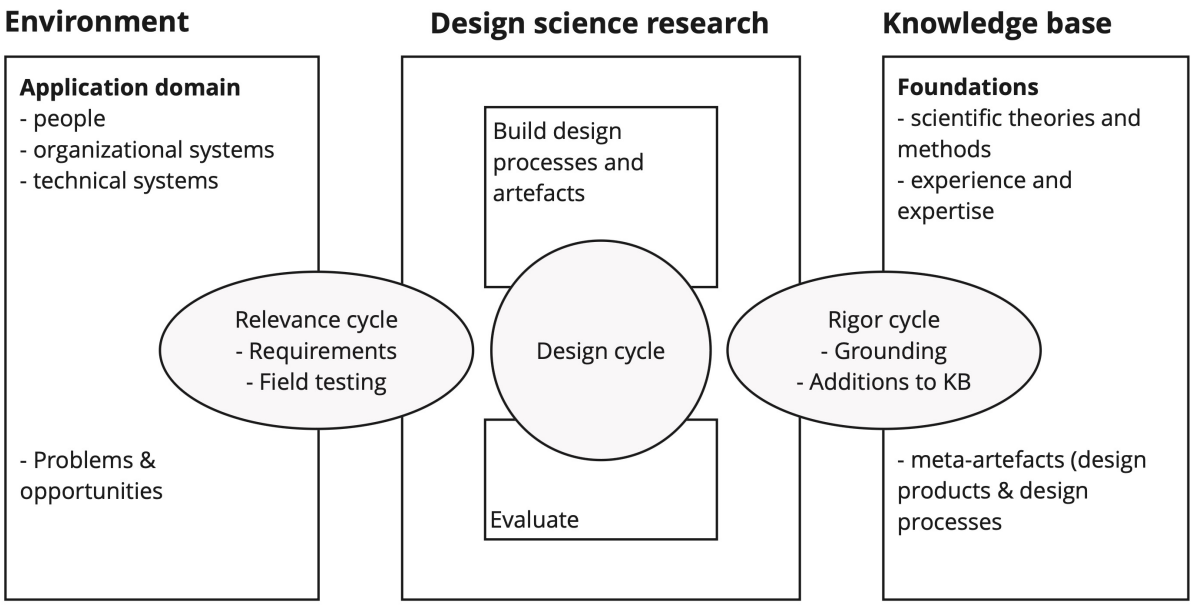

Figure 3.1: Design science research as desgribed by [19]. The important part of the method is the design sycle.

method fits well for this research.

The research consists of defining the environment, where the problem is defined and usable systems are evaluated, multiple design cycles, where tools are selected and the selected tools are individually implemented and evaluated, and the Design cycle is repeated for a tool until a good enough result is achieved. The design cycle leads to an artifact, which can be for example a deployment declaration, or to a process, which can be for example an update method. After each Design cycle, the artifact or the process is stored and it can be used as a basis for the next cycle. After the research is the Rigor cycle, where the artifacts and processes generated by the research are presented to the team and possibly implemented for the general deployments of B2SHARE [18].

3.1.1 Defining the environment

The problems that should be solved are reducing the amount of downtime during application updates and server maintenance while minimizing the cost of the deployment environment and minimizing the required work preparing for a service break and implementing one. In other words, how to do a cost-effective high availability deployment of B2SHARE. The problem defines the goals for the research: how to achieve near-zero downtime updates of B2SHARE and how to achieve near-zero downtime server maintenance

with B2SHARE. The goals do not contain high availability deployments of the B2SHARE application, but it comes as a side effect of deploying and replicating the application in a cluster.

Defining the technical systems included evaluating platforms that can be used for the research and possibilities for using different distributed data management tools for the research implementations and the possible production implementations. CSC provides a container platform called Rahti, which is a Kubernetes and OpenShift environment used to deploy containers. However, cPouta was selected for the implementations because it provides virtual machines that can be used to implement different implementations, it is used for the development instances and it is similar to the production environment.

Using cPouta brings some limitations to the implementations. The research project can use at any given time 8 CPU cores, 32.2 GB memory, 2 floating IPs, and 8 instances.

For evaluating different implementations metrics are collected from each implementation. The goal for each implementation is to have near-zero downtime for deployments and the secondary goal is to have near-zero downtime during server maintenance or server downtime. Metrics are collected from other factors, which include the cost of infrastructure, the time it takes for an updated version to be usable, and the amount of changes required to do the implementation. The cost is measured in billing units and calculated based on tables 3.1 and 3.2. The calculation is done with the formula 3.1. The metrics lead to analytical and experimental evaluation methods proposed in [19]. The subsection 3.1.2 describes how the measurements are made with different implementations.

$$cost/h = no.instances * BU/h + no.floatingIP * BU/h + storage * BU/h \quad (3.1)$$

Cores	Memory in GiB	Disk in GB	Cost in BU/h
1	0.9	80	0.25
2	1.9	80	0.5
3	3.9	80	1
4	7.8	80	2

Table 3.1: Price of instances in billing units [9].

To measure downtime and to evaluate different implementations, measurements are made with a Python script that sends an HTTP request over the network approximately every half a second to the B2SHARE API root, and the responses are logged into a CSV file.

Type	Unit	Cost in BU/h
Floating IP	Amount	0.2
Storage Volumes	TiB	3.5

Table 3.2: Other infrastructure costs for deployments in billing units [9].

The resulting list of response statuses is the measurement. The measurement resolution is 0.5 seconds. The HTTP request measurement method ensures, that the application is reachable. The method does not test the usability of the application. Detailed descriptions of update time measurements are provided in each cycle. Each measurement is done at least ten times. The HTTP measurement Python script is in appendix B.

The Python script is run on an Apple laptop with an Apple M2 Pro processor, 16 GB of memory, and a wireless connection to a cable network. The theoretical network speed is 1000 Mbps for download and 100 Mbps for upload. The measured network speed is approximately 650 Mbps for download and 95 Mbps for upload. The approximate distance from the measurement laptop to the virtual machines is 460 km and the measured network roundtrip time is approximately 31 ms. The roundtrip latency measurement is made with ping to an instance running in the same data center as the implemented clusters.

The baseline measurements are made with a Docker Compose deployment with a B2SHARE application stack described in table 3.3. This deployment is similar to the normal B2SHARE deployment. The basis of the deployment is one virtual machine with 4 CPU cores, 7.8GiB of memory, 80GB of root disk, and Almalinux 9 as an operating system. The downtime measurement is done only with the HTTP request method. The update command is described in appendix A.1.

To achieve the goals and make measurements with different implementations, a cluster of virtual machines is created and the application is deployed to the cluster. To achieve the goal for near-zero application downtime server maintenance, a data management system should be implemented, because the application relies on stored data. To make measurements of deployment update downtime, a rolling update should be done and the downtime caused measured. To measure downtime during server maintenance, one server in the cluster should be brought down and back up or it should be simulated with another method. The used method is described for each implementation.

It is important to note, that later in this chapter and the following chapters one stack or multiple stacks of B2SHARE are deployed to a cluster. Multiple stacks mean, that

Container/Pod	Description
B2SHARE	Handles API, Files and background tasks
REDIS	Memory cache
ElasticSearch	Search engine
Postgres	Database
RabbitMQ	Message queue
Nginx	Web server as reverse proxy

Table 3.3: Minimum working B2SHARE application stack deployment deployed with Docker Swarm and Kubernetes. With Kubernetes Nginx was deployed as an ingress controller.

two or more unique instances of the B2SHARE application are deployed. For example, a deployment with two stacks can have one stack for customer A and one stack for customer B. These application stacks do not depend on each other.

The starting point for each new implementation is a cluster of virtual servers running on the OpenStack platform with Almalinux 9 as an operating system and containerd as a container runtime. The full specification of instances in each cluster implemented is described in the subsection 3.1.2 . The deployed application stacks are not the same as the production stacks described in section 2.6. The deployed stacks are minimum working deployments of B2SHARE described in table 3.3.

3.1.2 Cycles

The first cycle included a comparison of all major container orchestration tools. These tools are Docker Compose, Docker Swarm [48], Kubernetes [22], and HashiCorp Nomad [31]. The comparison was made to select two candidates for implementation and further testing. The comparison concerned the maturity, usability, features, costs, learning curve, scalability, cluster support, and available documentation of the tools.

The two tools selected for implementation were Docker Swarm and Kubernetes. Docker swarm was selected, because there is already some knowledge of the tool in the team, it supports high availability and replication, is easy to use in high availability environments, high availability deployments are cost-effective and the deployment declarations are very similar to Docker compose declarations. Kubernetes was selected because it has a lot of features concerning configuration, high availability, and replication support and it is widely used for deploying cloud applications. The comparison was made with online

documentation [48, 22, 31] and based on [26, 50, 47].

Implementation cycle for Kubernetes.

Kubernetes implementation (labeled **K1**) started with three virtual machines. One virtual machine is a dedicated control plane node with 2 CPU cores, 2GiB of RAM, 80GB disk, and a floating IP, and two virtual machines as worker nodes with 3 CPU cores, 3.9GiB of RAM, and 80GB root disk. One shared volume with 100GB of storage space was created to function as a data disk and it was attached to both worker nodes. Initial Kubernetes implementation is described in figure 3.2. The cluster was created with kubeadm [22].

The control plane node hosts the Kubernetes API server, which is used to distribute pods to worker nodes. Based on Kubernetes documentation about best practices, the control plane node does not run any application pods. The worker nodes are used to run application pods. However, it is possible to run application pods in the control plane nodes [22].

Data management was done with a Cinder multi-attach volume, which is a volume that can be attached to multiple virtual machines and is built into the OpenStack platform. This 100GB volume provides a good method for sharing data between nodes without a need for synchronization or implementing a separate data management cluster. However, if the multi-attach volume is not configured correctly with a clustered file system, which coordinates operations for the shared volume, there is a possibility for read errors and data corruption [9]. For a production environment, a SAN (Storage Area Network) volume or other distributed filesystem described in section 2.5 can be used. OpenStack does not offer SAN disks, thus a Cinder multi-attach volume was used.

To deploy the B2SHARE application to Kubernetes, definitions for pods and deployments, volumes, networks, services, etcetera had to be created. These definitions are used to manage the cluster and the application and define the required resources for the application to work correctly in the cluster. The definitions are the main artifact created by the Kubernetes implementation cycle. A Kubernetes deployment is a definition for a set of pods to run an application, the replication of the pods, and other configurations of these pods [22].

A liveness probe and a startup probe were configured for each Kubernetes deployment. The liveness probe checks periodically, if the application in a pod is running correctly. The startup probe is checking periodically during the pod startup if the application in the pod is ready [22].

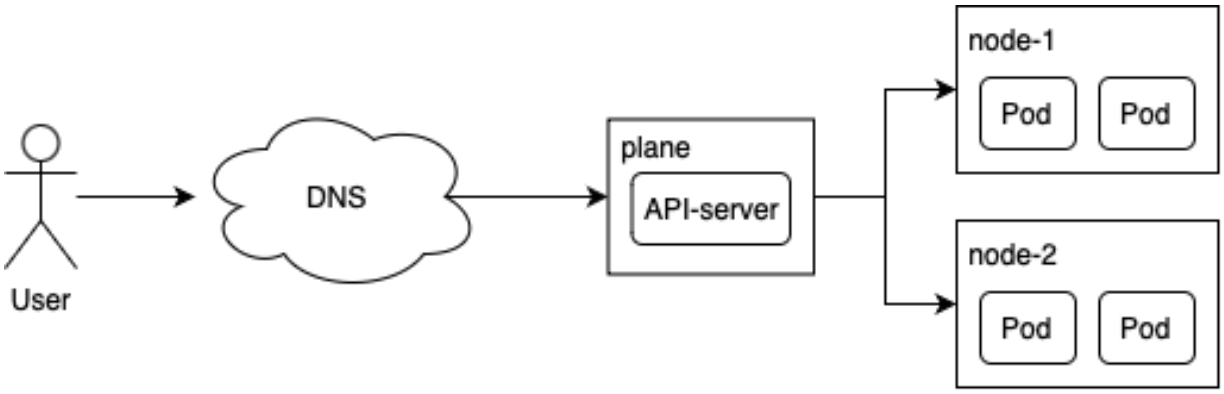

Figure 3.2: Initial implementation of Kubernetes, where all requests go to the control plane, which redirects traffic to the correct application Pod.

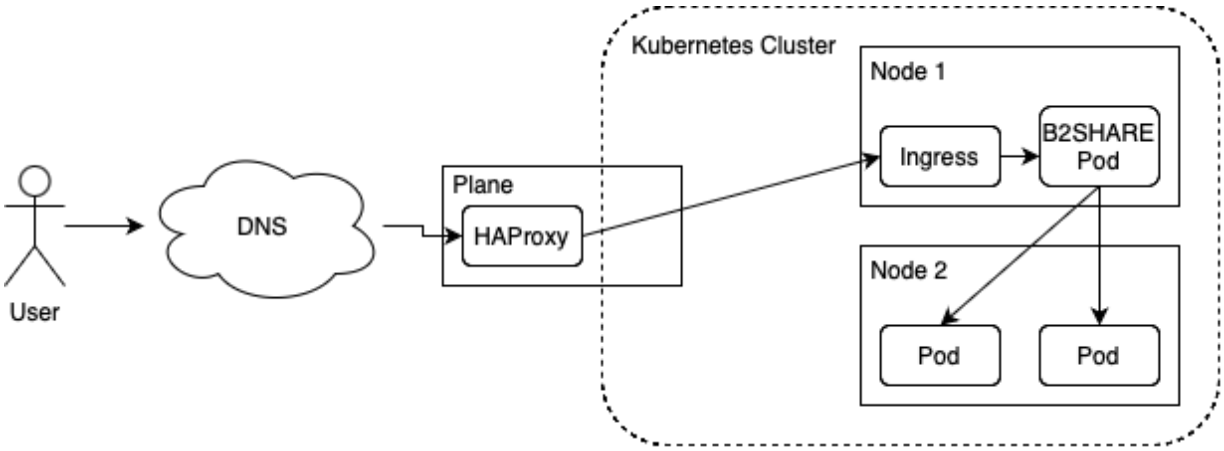

Figure 3.3: A rough network diagram of the Kubernetes cluster deployment. When a user sends a request, it goes to the control plane, where HAProxy proxies the request to the ingress controller. The ingress controller forwards the request to the application pod.

To be able to reach the application from the internet, a proxy was added to one of the nodes in the cluster to handle incoming traffic. The selected proxy was HAProxy and it was deployed on the control plane node. The control plane node has a public floating IP address accessible from the internet and the proxy listens to HTTP and HTTPS traffic and proxies the traffic to the ingress controller, which is a part of the application deployment in the Kubernetes cluster. Figure 3.3 describes the flow of a request from a user to the application and between the pods.

With the Kubernetes implementation, only one set of the application stack was deployed. Each part of the application stack was its own Kubernetes deployment. This implementation was considered good enough because Kubernetes provides a lot of functionality to manage the application deployment and for recreating deployed pods on a different worker

node to handle node downtime [22].

The implementation (**K1**) made does not allow for the high availability of the application. When the control plane node goes down, the application is not usable. For the application to have high availability, at least three control plane nodes have to exist to handle the downtime of other control plane nodes. A highly available implementation also requires at least three worker nodes based on Kubernetes documentation [21].

Calculating the cost of a high availability B2SHARE deployment with a Kubernetes cluster is done with two different possible implementations of a cluster. These were not implemented. The first method (labeled **KP1**) is to run one B2SHARE deployment per worker node requiring two worker nodes to run two replicas of B2SHARE making it possible for one worker node from these two nodes to go down without causing downtime. With this implementation, each worker node requires 3 CPU cores and 3.9 GiB of memory. The cost of virtual machines for this implementation is calculated with equation 3.2. This cluster implementation requires at least three worker nodes and three control plane nodes. This cluster implementation is described in figure 3.4.

The second method (labeled **KP2**) is to have worker nodes with more processing power, where one worker node can run two B2SHARE application stacks at once. The application is replicated to two nodes. Each worker node in this implementation has 4 CPU cores and 7.8 GiB of memory. The cost of this implementation is caluclated with equation 3.3. Both methods can handle the downtime of at least one worker node and the downtime of one control plane node without any application downtime. This cluster implementation is described in figure 3.5.

In the equations 3.2 and 3.3 *no.applications* is the amount of unique B2SHARE application stacks deployed.

$$cost/h = MAX(no.applications * 2 * 1BU/h, 3 * 1BU/h) + 3 * 0.5BU/h \tag{3.2}$$

$$cost/h = MAX(3 + \lceil \frac{no.applications - 4}{2} \rceil * 2BU/h, 3 * 2BU/h) + 3 * 0.5BU/h \tag{3.3}$$

For having near-zero downtime node updates without guaranteed high availability, a set of three control plane nodes and the amount of deployed unique application stacks of worker nodes plus one worker node is required, where each worker node has a minimum of 3 CPU cores and 3.9 GiB of memory. This is because, during worker node updates, the workload

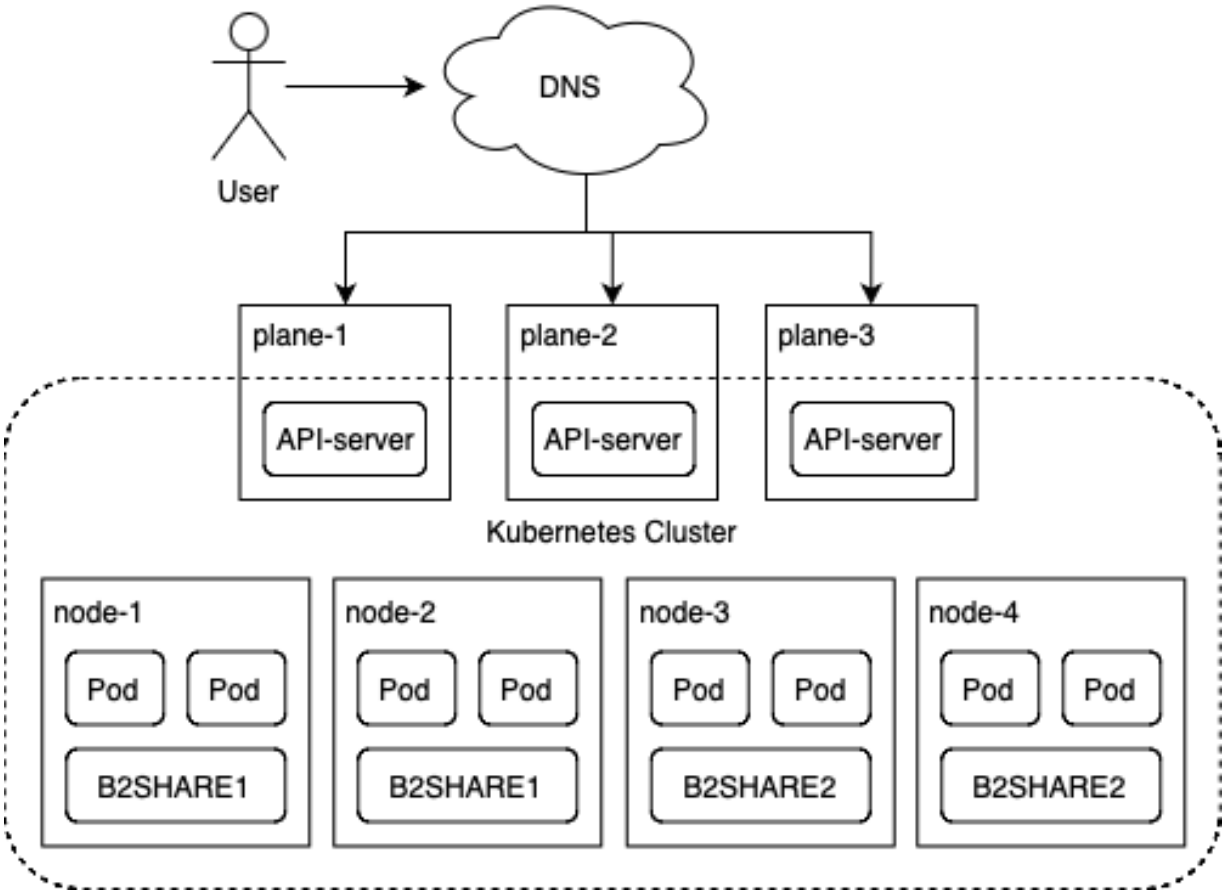

Figure 3.4: A rough description of a high availability Kubernetes cluster (**KP1**) with two unique B2SHARE application stacks (B2SHARE1 & B2SHARE2) deployed and replicated. DNS is used as a load balancer and ingress HTTP traffic is routed via control plane nodes.

of a worker node can be transferred to the extra worker node. After the workload is transferred, the worker node without workload can be brought down without application downtime. However, the limit of a minimum of three worker nodes still applies. This implementation is not a high availability implementation: it does not handle unplanned node downtime with near-zero application downtime. This method is labeled **KP3**.

The application update time is measured by analysing a description of the application pod, where Kubernetes provides a log of statuses for the pod. The update time is the time elapsed from the start of a new pod to the moment, when the pod is declared ready. An example log for a pod status is in appendix A.2. This measurement relies on the startup probe, which is configured to check the application readiness in a pod every ten seconds. The update time might get better, if the startup probe is configured with a smaller check interval.

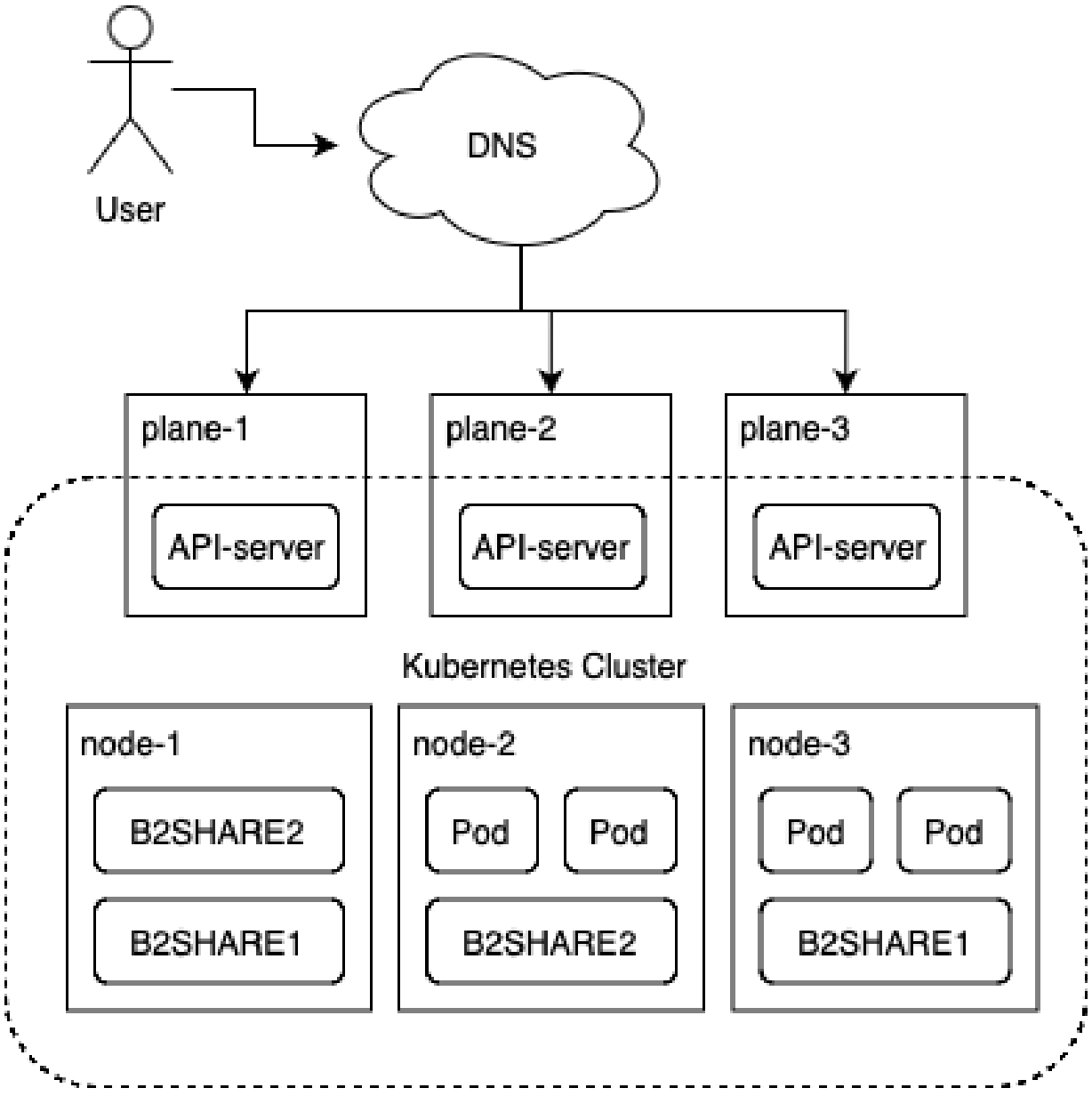

Figure 3.5: A rough description of a high availability Kubernetes cluster (**KP2**) with two unique B2SHARE application stacks (B2SHARE1 & B2SHARE2) deployed and replicated. DNS is used as a load balancer and ingress HTTP traffic is routed via control plane nodes.

Implementation cycle for Docker Swarm.

Comparing to Kubernetes, Docker Swarm does not require a dedicated leader node that does not run application-related containers [48]. The Swarm cluster was set up with two virtual machines. The virtual machines have each 4 CPU cores, 7.8GiB of memory, 80GB disk, and a unique floating IP address. One node was selected to be the leader node and the other node was a normal worker node. Docker Swarm supports single node operation as well, but to test possible high availability deployments and near-zero downtime server maintenance deployments, two nodes are used. This deployment can not handle downtime of the leader node: a fault-tolerant deployment requires at least three nodes, where each node is a manager node [48].

A leader node in a Docker Swarm cluster manages the cluster and the containers in the cluster and a manager node is a node, that can function as a leader node. For deploying new containers, updating application deployments, and cluster management is done with the leader node. In a Docker Swarm cluster, multiple nodes can be promoted to be a manager node, but only one of these manager nodes at any given time is the leader node. If the currently active leader node goes down, a reachable manager node takes over as a leader node [48].

Initial Docker Swarm cluster implementation (labeled **S1**) can be seen in figure 3.6. With the Swarm implementation, the idea is to have round-robin load balancing with DNS, that balances traffic between the instances. An HTTP reverse proxy is used to route traffic into the correct application container based on the domain name. Only one database container was deployed to simulate an external database server. Each deployed application has its own database in the database container.

Measuring downtime during application deployments is done with a configuration that implements a start-first method, where a new container is started before the old container is stopped [48]. Measurement methods and commands are in appendix A.3. To fine-tune the update time, the configuration was modified and deployed until reaching a good enough result. The update time relies on health check configuration, which is configured to check if the container is healthy every ten seconds.

Implementation cycle of replicated containers with Docker Swarm.

For achieving near-zero downtime server maintenance and high availability, replication of the application container and it's dependent containers is required. This is done by replicating the containers to each instance in the cluster. The implementation (labeled

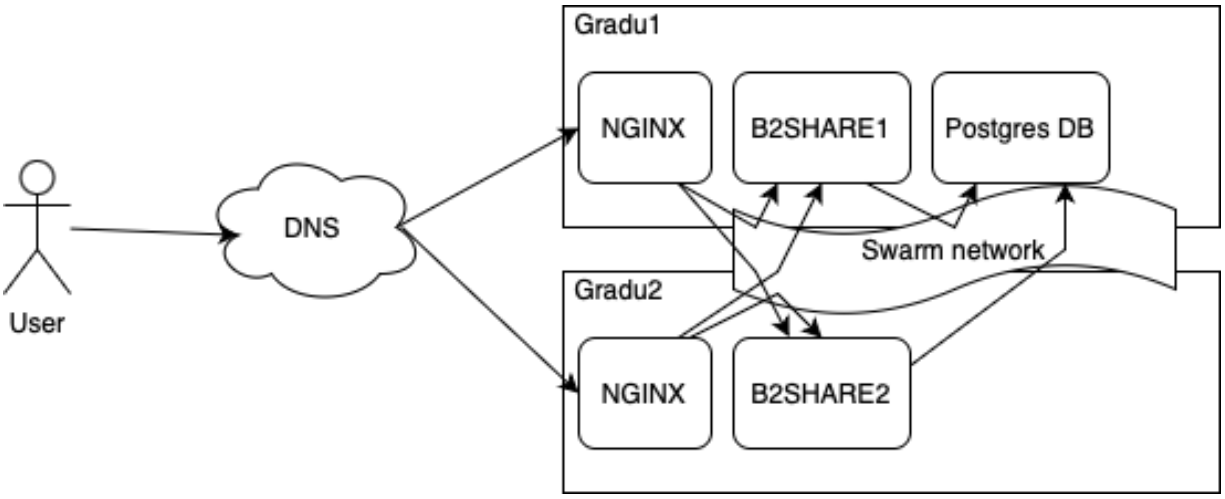

Figure 3.6: Initial implementation of Docker Swarm. The DNS routes traffic with round-robin load balancing to either one of the instances, where NGINX reverse proxy redirects traffic to the correct application container.

S2) with replicated containers is described in figure 3.7.

Data management is done with a Cinder multi-attach disk, which is a disk that can be attached to multiple servers at once. The disk is attached to both virtual machines in the cluster. The same kind of multi-attach volume was used with the Kubernetes implementation. To prevent read errors or data corruption, a clustered file system is used and the selected file system is Global File System (GFS2 for short) [9].

Measurements of application downtime during server maintenance are done by draining a node. This action removes all running containers from the specified node. The node type (leader/manager/worker) does not affect its capability to be drained [48]. This is done to simulate server downtime. The measurements are made with the same methods as not replicated implementation and the commands and methods are described in appendix A.3. The update time relies on health check configuration, which is configured to check if the container is healthy every ten seconds.

The cost for high availability deployments of B2SHARE with Docker Swarm is calculated with equations 3.4 and 3.5. The first equation 3.4 is used to calculate the cost of deployment (labeled **SP1**) with virtual machines, which each have 3 CPU cores and 3.9 GiB of memory. This kind of virtual machine can run only one deployment of B2SHARE at once. Due to this limitation, there have to be two virtual machines for each deployment for the application to be replicated for high availability, which can withstand the outage of one node.

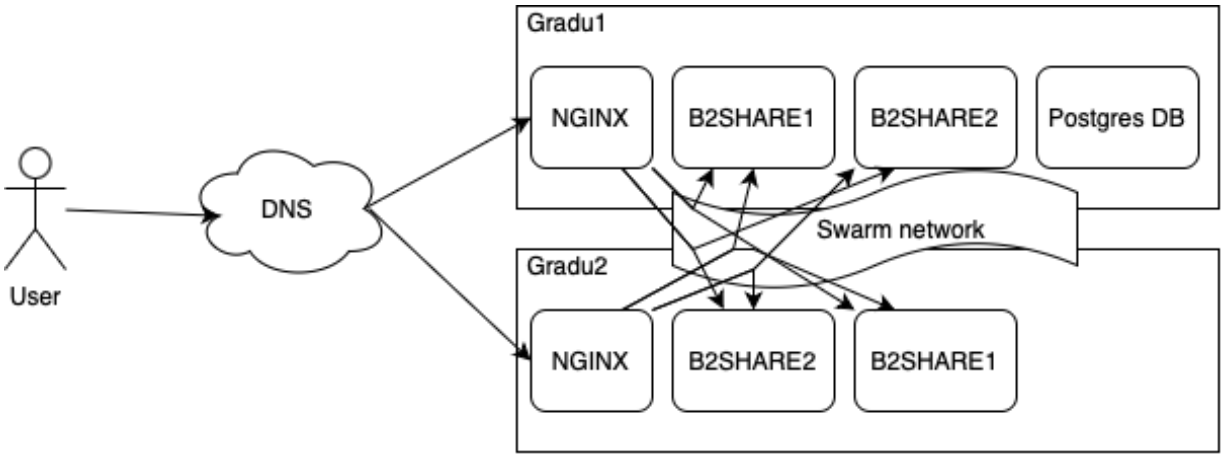

Figure 3.7: Replication of application containers to each node in the cluster to achieve near-zero downtime server updates. Most of the containers are left out from the figure, but containers that the application depends on are replicated as well. The full list of containers is in table 3.3. Containers B2SHARE1 and B2SHARE2 are part of different application stack deployments.

The second equation 3.5 is used to calculate the cost of a deployment (labeled **SP2**) with virtual machines, that each has 4 CPU cores and 7.8 GiB of memory. This kind of implementation can have two B2SHARE deployments running on one virtual machine. The minimum number of nodes in high availability Docker Swarm clusters is three for the cluster to tolerate the downtime of a leader node. The fault tolerance for leader node downtime increases by one node with every two manager nodes added [48]. Both deployment methods can handle unplanned downtime of one node - leader, manager or worker - without any application downtime.

In the equations 3.4 and 3.5 *no.applications* is the amount of unique B2SHARE application stacks deployed.

$$cost/h = MAX(no.applications * 2 * 1BU/h, 3 * 1BU/h) \tag{3.4}$$

$$cost/h = MAX(3 + \lceil \frac{no.applications - 4}{2} \rceil * 2BU/h, 3 * 2BU/h) \tag{3.5}$$

For near-zero downtime node updates, the amount of unique application stacks plus one node is required. These nodes can be as small as suitable. With B2SHARE, it's 3 CPU cores and 3.9 GiB of memory. Before a node update, the node should be drained and the workload transferred to the extra node. After that, the drained node can be brought down. This implementation is not a high availability implementation: this cluster can not handle unplanned downtime of a node without application downtime.

4 Results

This chapter presents the results of different implementations described in the chapter 3 and measurements of downtime of these implementations. First, the methods for analyzing the results are presented in the section 4.1. After that, the baseline measurements with Docker Compose deployment are presented in the section 4.2. After the baseline, the results of two proposed solutions described in section 3.1.2 are given. First for the Docker Swarm cluster in section 4.3 and then for the Kubernetes cluster in section 4.4. All the measurements, deployment configurations, and deployment guides are published in B2SHARE[†]. The measurements and configurations published are with the minimum achieved downtime.

For all the results, it's important to note, that if the deployed application has a lot of data, the system requirements might change. For example, it's noticed from B2SHARE production deployments, that ElasticSearch takes more memory compared to deployment with only a few records, and an instance with 8GiB of memory is required to run the application. On the other hand, REDIS, ElasticSearch, and RabbitMQ can be shared between multiple B2SHARE deployments.

4.1 Analyzing the results

The implementations created two different measurements. What is the measured downtime when sending HTTP requests over the network and how long does an application update take, or in other words, how long does it take for a new container to start and take over the old container. Two different methods were used to measure update time: analyzing the log files and using the time command.

The results of over-the-network measurements are analyzed based on HTTP status codes. The application is considered to have downtime, when the status code is something else than 200 OK or consecutive Timeout [13]. Two other cases were noticed in the measurements: HTTP status code 502 Bad Gateway and a request exception failed to establish a new connection. The downtime was measured from the last HTTP status code 200 in a consecutive series of HTTP status code 200 to the first HTTP status code 200 of a

[†]https://doi.org/10.23728/b2share.0ab94117e43b45baa64773625b52fa33

consecutive series of these codes. If the measurement series included a series of timeouts, it was considered, that the application is usable. The results of update time measurements were analyzed based on the elapsed time. The mean update time was calculated for each implementation. If it is stated, that zero downtime was achieved, it means, that all the HTTP request measurements are either a full set of HTTP status code 200 OKs or there can be Timeouts in the measurement set.

To be clear, a Timeout means that the Python script did not receive an answer from the application in 0.5 seconds. There are many reasons for this to happen, thus it is considered, that the application is responding.

The measurements with HTTP requests were made to analyze, if the goals of near-zero application downtime updates and near-zero application downtime server maintenance were achieved. The measurements of application update time were made to analyze if one deployment method is faster in updating the application compared to other methods.

For research questions one and two, the infrastructure changes are from implementations and from online documentation For research question three the costs are calculated based on the implementations and online documentation regarding the goal.

4.2 Baseline

Measurements of downtime of the baseline deployment (Docker Compose) application updates resulted in a mean downtime of 46,4 seconds. The maximum recorded downtime was 47,4 seconds and the minimum was 45,7 seconds. Measurements for application downtime during server downtime were not made, because it is directly correlated to the duration of the server downtime and has a lot of variability due to manual work.

The cost for the baseline deployment is 2 BU/h for the virtual machine and 0,2 BU/h for one floating IP totaling to 2,2 BU/h. However, Kubernetes deployment did prove, that the B2SHARE application can run with a virtual machine with less processing power and memory bringing the cost to 1,2 BU/h.

4.3 Docker Swarm

Zero downtime application updates were achieved with the Docker Swarm implementations. This applies to both non-replicated (**S1**) and replicated (**S2**) implementations.

However, during the updates with implementation **S1**, there is some variability in response times over the network. The maximum measured response time during application updates was 3.2 seconds. This measurement was made by calculating the time elapsed between the last 200 OK before a series of Timeouts and the first 200 OK after the series of Timeouts. This means, that the request did not result to an error, but a request timeout would not have happened if the timeout limit had been greater than the elapsed time. However, the Timeouts can be caused by normal network latency and there are many other factors, that can affect the response time, for example, how quickly the reverse proxy notices that traffic can be routed to the new container and the route to be updated.

For an application update to be ready and all requests routed to the new application container, the mean update time was 55.02 seconds with Docker Swarm non-replicated deployment (**S1**). Without update parallelism, the update time for replicated application deployment (**S2, SP1, SP2**) can be roughly calculated with multiplying the update time without replication and the amount of replicated containers.

Zero application downtime during server maintenance was achieved with the Docker Swarm replicated implementation (**S2**). The implementation required the application container to run on a global mode, which replicates each container in the application stack to each instance. With more instances in the cluster, it is sufficient to deploy the application to two instances or to move the application container from one instance to another with zero downtime before planned instance downtime.

For research question one, to achieve near-zero downtime application updates no infrastructure or architecture chages are required compared to Docker Compose deployments. A single node Docker Swarm deployment can be used to achieve zero downtime deployments. For research question three, a single node deployment with Docker Swarm does not bring additional costs compared to a Docker Compose deployment. The system requirements are the same. This was simulated in the implemented Docker Swarm cluster by running the application on one instance.

For research question two, to achieve near-zero downtime server maintenance with Docker Swarm, at least three nodes and a storage system for distributed computing are required. A DNS can be used as a load balancer to divide traffic between the nodes removing the requirement for an additional dedicated load-balancing node. The Docker Swarm cluster implementation with replication however requires a good solution for data management; Docker Swarm does not have built-in data management tools for deployments, which are

replicated to multiple nodes. One possibility is to use a disk, that can be attached to multiple nodes.

Zero downtime server maintenance for a worker node was achieved with the Docker Swarm replicated implementation (**S2**) with two nodes. However, this deployment can not handle downtime of the leader node due to the minimum requirement of three nodes for high availability Docker Swarm cluster implementations [48]. With two nodes, the worker node can be brought down without causing application downtime.

For research question three, deployments with multiple virtual machines cause additional infrastructure costs for a single deployment. With Docker Swarm and a minimum requirement of three nodes, where each node acts as a leader node, the total cost is roughly three times compared to a single node deployment due to the required nodes. It is important to note, that the nodes might not have to have as much computing power compared to a single node deployment, because the workload can be shared with all nodes in the cluster. However, to achieve high availability, at least two replicas of each container have to be running at any given time.

The cost of the implemented Docker Swarm cluster with replication (**S2**) is 4 BU/h for virtual machines, 0,75 BU/h for two floating IPs:s and storage totaling 4,75 BU/h. For a high availability deployment of one B2SHARE stack, three virtual machines with 3 CPU cores and 3.9 GiB of memory each are required with a floating IP assigned to at least two of the virtual machines. This deployment costs 3 BU/h for the virtual machines, 0,75 BU/h for three floating IPs:s and storage totaling to 3,75 BU/h. The smaller cost is due to less requirements for processing power and memory per node.

High availability deployment for two B2SHARE deployments with Docker Swarm requires either four instances with 3 CPU cores, 3.9 GiB of memory, a multi-attach volume, and two floating IPs totaling 4,75 BU/h (**SP1**) or three virtual machines with 4 CPU cores, 7,8 GiB of memory, a multi-attach volume and two floating IP:s totaling 6,75 BU/h (**SP2**). Figure 4.1 compares the costs of these two methods for high availability Docker Swarm with n amount of B2SHARE deployments.

A comparison of the implemented Docker Swarm cluster and two implemented deployments (**S1, S2**) and proposed high availability clusters (**SP1, SP2**) is in table 4.1.

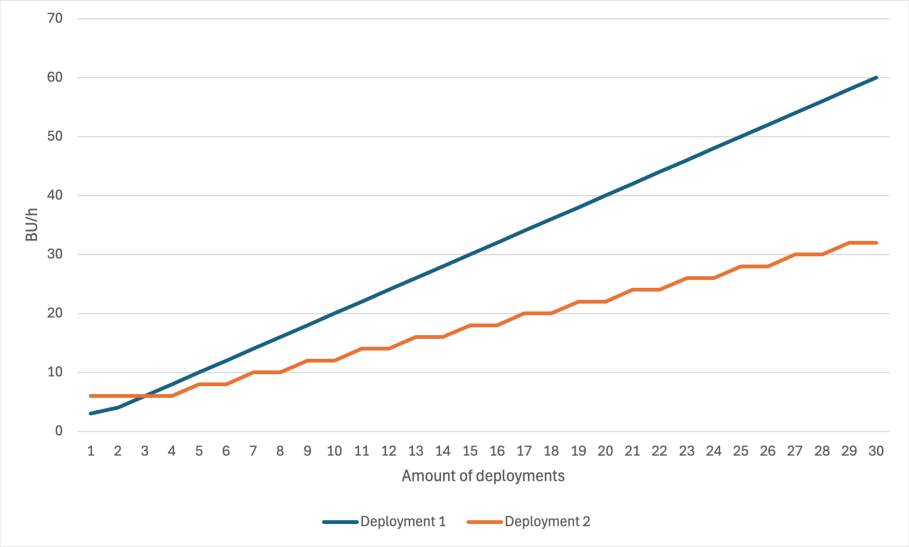

Figure 4.1: Comparison of costs for different methods for achieving high availability for B2SHARE deployments with Docker Swarm cluster. Deployment 1 (**SP1**) has worker nodes with 3 CPU cores and 3.9 GiB of memory. Deployment 2 (**SP2**) has worker nodes with 4 CPU cores and 7.8 GiB of memory. The cost does not include floating IPs and storage.

4.4 Kubernetes

Implemented Kubernetes cluster (**K1**) proved to have zero application downtime during application updates and worker node maintenance. However, as previously described in 3, the implemented cluster can not handle downtime of the control plane node.

The implemented Kubernetes cluster (**K1**) does not have enough processing power to run more than one unique deployment of the B2SHARE application stack during worker node updates. For this, either more worker nodes are required or the existing worker nodes have to have more processing power and memory. One deployment of B2SHARE utilizes almost all of the memory on a worker node (4GiB) when all pods required are running on the same node.

During an application update in the implemented cluster (**K1**), it took approximately 56 seconds for the new pod to take over. However, this relates to the startup probe configuration. The startup probe is used to check if a pod is ready to handle connections. The new pod takes over as soon as the startup probe notices, that the new pod is ready.

For research question one based on Kubernetes documentation [22], at least two virtual machines, where one is a control plane node and the other is a worker node are required for near-zero downtime application updates. The single worker node has to have enough

Label	**Nodes**	**Node spec**	**High availability**	**Cost in BU/h**	**Amount of unique B2SHARE stacks**	**Near-zero downtime**
S1	2	4 CPU, 8 GiB Memory	No	4.75	2	Updates only
S2	2	4 CPU, 8 GiB Memory	No	4.75	2	Updates and maintenance
SP1	4	3 CPU, 4 GiB Memory	Yes	4,75	2	Updates and maintenance
SP2	3	4 CPU, 8 GiB Memory	Yes	6,75	2	Updates and maintenance

Table 4.1: Comparison of implemented Docker Swarm cluster and proposed implementations. **Amount of unique B2SHARE stacks** column is the amount of supported unique B2SHARE application stack deployments while near-zero application downtime server maintenance is possible. **NOTE: S1** does not support near-zero application downtime server maintenance. **S1 & S2** does not support maintenance of the leader node.

processing power and memory to run the application replicated. For near-zero downtime application updates changes for the application architecture are not required.

For research question two based on the implementation (**K1**) and Kubernetes documentation [22], a storage method for distributed computing and at least six virtual machines are required for near-zero application downtime server maintenance. In this six-node cluster, three of the virtual machines act as a control plane node and the remaining three virtual machines are worker nodes [21].

For the following results of Kubernetes cluster cost, a control plane node has 2 CPU cores and 1.9 GiB of memory. The cost of a single control plane node is 0.5 BU/h.

For research question three based on the implementation (**K1**), the cost for a single B2SHARE deployment that can tolerate worker node downtime is 2,5 BU/h for the virtual machines and 0,55 BU/h for a floating IP and storage totaling 3,05 BU/h. Each worker

node has 3 CPU cores and 3.9 GiB of memory. This cluster can be used for a deployment of two B2SHARE applications, but then worker node downtime can not be tolerated.

A Kubernetes cluster that can handle planned worker node downtime with two unique B2SHARE deployments requires either three worker nodes with 3 CPU cores and 3.9 GiB of memory and a control plane node based on **KP3** or two worker nodes with 4 CPU cores and 7.8 GiB of memory and a control plane node based on **KP2**. The first implementation has a cost of 3,5 BU/h for instances, 0,55 BU/h for a floating IP and storage totaling to 4,05 BU/h, and the cost for the implementation with two worker nodes with more processing power and memory is 4,5 BU/h for instances, 0,55 BU/h for a floating IP and storage totaling to 5,05 BU/h.

A truly high available deployment of one B2SHARE application stack deployment with a Kubernetes cluster costs 4,5 BU/h for instances, 0,75 BU/h for two floating IPs, and storage totaling 5,25 BU/h. This deployment has three control plane nodes, where two of them have a floating IP, and three worker nodes with 3 CPU cores and 3.9 GiB of memory. This implementation is based on **KP1**.

A high availability deployment of two B2SHARE deployments has two possible implementations: more nodes and nodes with more processing power and memory. The total cost of the first high availability implementation (**KP1**) is 6,25 BU/h with two floating IP addresses. This implementation has four worker nodes, where one node runs a single replica of the application, and three control plane nodes. The cost of virtual machines in this implementation is 5,5 BU/h. In this cluster, each worker node has 3 CPU cores and 3.9 GiB of memory.

The total cost of the second high availability implementation (**KP2**) is 8,25 BU/h with two floating IP addresses. This implementation has three worker nodes with 4 CPU cores, 7.8 GiB of memory, and three control plane nodes. One worker node runs a replica of both B2SHARE deployments. The cost of the virtual machines for this cluster is 7,5 BU/h. Figure 4.2 compares the costs of these two high availability deployment methods with Kubernetes. The same notice about system requirements growing with Docker Swarm deployment applies with Kubernetes deployments, that more records require more memory.

A comparison of the implemented Kubernetes cluster (**K1**) and proposed high availability cluster implementations (**KP1, KP2**) is in table 4.2. Proposal (**KP3**) because it is the same as **KP1** when one or two unique B2SHARE application stacks are deployed.

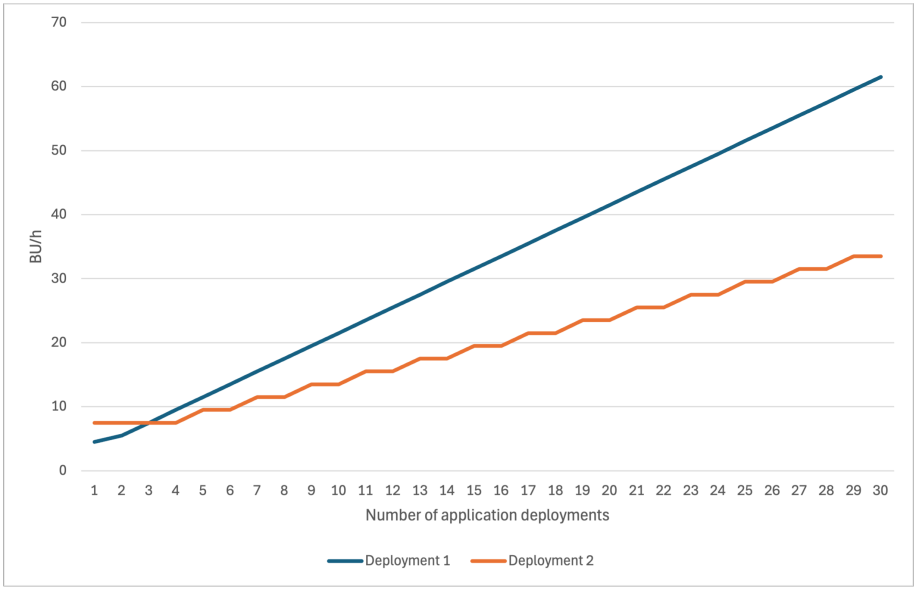

Figure 4.2: Comparison of costs for different methods for achieving high availability for B2SHARE deployments with Kubernetes cluster. Deployment 1 (**KP1**) has worker nodes with 3 CPU cores and 3.9 GiB of memory. Deployment 2 (**KP2**) has worker nodes with 4 CPU cores and 7.8 GiB of memory. The cost does not include floating IPs and storage, but it includes three control plane nodes.

Label	**Nodes**	**Worker node spec**	**High availability**	**Cost in BU/h**	**Amount of unique B2SHARE stacks**	**Near-zero downtime**
K1	1+2	2 CPU, 2 GiB Memory	No	3.05	1	Updates and maintenance
KP1	3+3	2 CPU, 2 GiB Memory	Yes	6.25	2	Updates and maintenance
KP2	3+3	3 CPU, 4 GiB Memory	Yes	8.25	4	Updates and maintenance

Table 4.2: Comparison of implemented Kubernetes cluster and proposed implementations. **Nodes** column is the amount of control plane nodes + amount of worker nodes. **Amount of unique B2SHARE stacks** column is the amount of supported unique B2SHARE application stack deployments while near-zero application downtime server maintenance is possible. **NOTE: K1** does not support maintenance of control plane node without application downtime.

5 Discussion

First in this chapter methods for achieving near-zero downtime application updates and near-zero application downtime server maintenance are presented in the section 5.1. After that, we'll compare the three orchestrator tools implemented (Docker Compose, Docker Swarm, and Kubernetes) in depth in section 5.2. At last, in section 5.3 we present a state-of-the-art high availability deployment for B2SHARE based on theory, that can be implemented with usable infrastructure.

5.1 Achieving near-zero downtime

The research proved, that near-zero downtime application updates are achievable for B2SHARE with the current infrastructure and requires only minor changes to the declarations using Docker Swarm. For achieving near-zero downtime deployments the easiest method is to use Docker Swarm in a single node mode. However, this change would require modifications to the existing deployment methods with automation.

Achieving near-zero downtime application updates is possible with a Kubernetes implementation as well. If the implementation does not follow best practices about Kubernetes clusters, it is possible to deploy B2SHARE to a single node Kubernetes. Comparing this implementation with a Docker Swarm implementation, the single-node Kubernetes cluster requires a virtual machine with more processing power and memory, because the Kubernetes API-server takes more computational resources compared to Docker Swarm.

For near-zero application downtime server maintenance, the application has to be deployed in a cluster. For a Docker Swarm cluster, the minimum amount of virtual machines for a single deployment is three. For a Kubernetes cluster, the minimum amount of virtual machines is six [21]. This makes Docker Swarm a cheaper option for near-zero application downtime server maintenance deployments but the cost difference is small if the Kubernetes control plane nodes have minimum system requirements. The difference is 1.5 BU/h. However, with larger clusters, the cost for the required three control plane nodes in a Kubernetes cluster is small compared to the cost of worker nodes in the cluster.

An important part of the implementations is the ability for virtual machines in the cluster

to be maintained with operating system updates and other package updates without causing downtime for the application. Both Kubernetes and Docker Swarm implementations proved that this is possible. Kubernetes has more safeguards than Docker Swarm when workloads are removed from an instance. Operators can define the minimum amount of pods running and being ready in a deployment, and Kubernetes will enforce it. Docker Swarm does not care, if a replacement container is ready or not, and the old container is removed just after the new container has started.

For high availability implementations, the data storage should also be fault-tolerant. If the application runs in two different data centers, the data should be replicated alongside the application. Thus, if one data center goes offline, the application still has access to all data.

For high availability deployments of any application, it is important, that each component the application depends on, is deployed with high availability in mind. The state of high availability of an application deployment is as weak as the weakest link. With the B2SHARE deployment, this would be the external database server, that is not implemented to be high availability. Thus, it is impossible for B2SHARE to have 100% availability without modifications to the database deployment. One good solution is to bring the database into the application cluster and replicate the database.

The calculations and descriptions of high availability cluster implementations had one assumption: the deployment can withstand at least one fault. For each deployment, it is important to set a limit, how many fault at minimum should the implementation handle correctly. For the proposed clusters the cost and required infrastructure is different, if the limit is at least two faults or at least three faults, and so on.

For load balancing requests DNS round-robin load balancing was used. This eliminated the requirement for external load balancer servers, which were described in section 2.3 in figure 2.4. For the implementations, the DNS load balancing works well, but it might have some issues with some users due to differences of how browsers or other user agents handle the DNS response.

DNS load balancing does not perform as a good load balancer when some of the target servers are down. If one target server out of three target servers is unreachable, 66.66...% of the requests goes to one server, and 33.33...% of the requests goes to the other reachable server. Thus, for high availability deployments using a set of dedicated load balancer servers is a good idea. Another method is to update the DNS records every time before a node is drained for maintenance and after the maintenance is completed. Before a

level three or four of high availability deployment is done for B2SHARE, I recommend for testing different load balancing methods and testing DNS load balancing more thoroughly.

5.2 Comparing the implementations

Both Docker Swarm and Kubernetes proved to be powerful tools for managing a cluster of virtual machines for containerized applications and for achieving near-zero downtime application updates and server maintenance. When comparing to the baseline Docker Compose deployment, deploying B2SHARE with Docker Swarm requires only small modifications to the already existing declarations making Docker Swarm a good candidate for fast and easy transition to near-zero downtime application updates. For Kubernetes, all the declarations had to be rewritten. For the rewrite process old Docker Compose declarations can be used as a reference and good basis.

Both implementations showed, that using containers is a good idea and the tools for orchestrating the containers are mature. The implementations proved, that achieving near-zero downtime application updates, near-zero application downtime server maintenance and high availability does not require in the best case any modifications to the application code. That was the case with B2SHARE. In the other hand all the containers in the application stack have to be replicable. For example, there were issues with the database container which did not support replication with the used configuration. However, there are existing declarations for replicating all the required containers in the B2SHARE stack.

For fine-tuning the application update time with the two implemented clusters, we could use the mean update time of 46.4 seconds from the Docker Compose deployment as a basis for determining the readiness of the application. With Kubernetes this means, that the startup probe is configured for example with an initial delay of 40 seconds, a check period of one second, and a failure threshold of 30 times. For the Docker Compose deployment, the configuration for the health check could be for example a start period of 40 seconds, a check interval of one second, and a maximum amount of retries of 30 times. Both of these configurations first waits 40 seconds and after the wait, the readiness is checked once every second. The check will result in a container/pod recreation after 30 failed checks. The Docker health check has the issue, that it will continue to do the health check once a second for the lifetime of the container. Kubernetes does not have this issue, because after the pod is ready, the startup probe finishes, and the health of the pod is determined by a liveness probe.

The HTTP measurement method turned out to be a good solution for measuring application downtime. The method did not measure if the application is fully functional or not. However, a test that tests the functionality of the application would not achieve the same test interval compared to the used method. A test that tests functionality takes longer to run each time.

As stated before in chapter 4, the resolution of downtime measurements is half a second. Measured roundtrip time to the dataceneter and back is approximately 31 ms as presented in section 3.1.1. Thus, the network latency could have affected the measurements by approximately 6%.

Some differences were noticed in the research made between the Docker Swarm cluster implementation and the Kubernetes cluster implementation. Setting up the Kubernetes cluster did take longer compared to Docker Swarm. Configuring a Kubernetes cluster was more complex. Kubernetes is a lot more configurable by the cluster operator compared to Docker Swarm. These configurations include cluster networking, setting up data volumes, configuring ingress to the cluster and applications running in the cluster, and more. Docker Swarm on the other hand provides a more "out of the box" solution.

After the Kubernetes cluster was successfully created and the application deployed, both implementations performed similarly. Infrastructure for Kubernetes is a bit more expensive compared to Docker Swarm due to the requirement of an additional control plane node.

No automation or different storage methods were implemented for this thesis. Based on theory and online documentation for all three container orchestrators, Kubernetes has the most possibilities for automation integrations and distributed storage methods. GlusterFS and Longhorn have turn-key deployments for Kubernetes and GitOps can be integrated to Kubernetes easily. It is important to note, that GitOps integrations exist for Docker Swarm as well. Database systems, such as PostgreSQL and MySQL have replicated deployments for Kubernetes.

It could be argued, that Kubernetes is a better option in most implementations for clusters running containerized applications when compared to Docker Swarm. This is because Kubernetes is designed to run multiple application deployments in a single cluster, while Docker Swarm is designed for deploying one application stack to a cluster.

With Docker Swarm, the operator has to define, how requests are routed in the cluster. Kubernetes does not require this, as long as a request reaches the cluster, the cluster

handles the request routing. Kubernetes does this based on the domain name and ingress controllers. A sufficient setup for a Kubernetes cluster is to have two or three ingress nodes (that can be for example the control plane nodes), and these nodes route all incoming HTTP traffic to the ingress controller. With Docker Swarm, the request routing configuration is done with a reverse proxy, such as nginx.

With smaller deployments especially with Docker Swarm cluster implementations, a multi-attach data disk is a good enough solution for managing data between instances. For large-scale deployments with multiple applications and for Kubernetes clusters, multi-attach disks can be hard to manage due to a lot of virtual volumes. NFS, GlusterFS, Longhorn, or other sophisticated storage methods for distributed computing should be used with these cluster implementations.

Based on the research, online documentation, and previous research, it is argued, that Kubernetes is generally the best option for achieving near-zero downtime application updates and near-zero application downtime server maintenance with containerized application deployments. Kubernetes is becoming the industry standard for orchestrating containers in clusters if it has not yet become the industry standard. The amount of deployment configuration provided by Kubernetes is a lot, and the amount of ready-made deployment configurations for different applications for Kubernetes is overwhelmingly more compared to Docker Swarm. In my opinion these reasons justifies the small cost increase for Kubernetes cluster implementations compared to other container orchestrators.

The research was done using B2SHARE as an example of an uWSGI application. I would argue, that the results and the methods presented in this thesis apply to other uWSGI applications as well. For an uWSGI application to achieve near-zero downtime application updates and near-zero application downtime server maintenance, it is recommended, by this thesis, to containerize the application and deploy the containerized application into a cluster.

This thesis provided production-ready declarations for deploying B2SHARE into a Kubernetes cluster or into a Docker Swarm cluster.

5.3 A truly high availability deployment

For the B2SHARE deployment to be fully high availability (level four of high availability), the deployment cluster should have instances running in at least two data centers. In the

context of CSC, these are in Kajaani and in Espoo. For the deployment to withstand an outage of one data center, the cluster should have control plane nodes in both locations, data should be replicated to both locations using a distributed file system such as Longhorn and databases should be replicated to both locations.

For state-of-the-art high availability deployments, Kubernetes should be used. As presented before, Kubernetes has better functionality for high availability deployments. This applies to deployments to multiple data centers as well. The following description assumes a Kubernetes cluster with three production B2SHARE applications deployed.

The cluster should have three control plane nodes. At least one control plane node has to be in Kajaani and one in Espoo. The location of the third control plane node does not matter. The control plane nodes can be used as ingress nodes: each control plane node has a public IP address. This way one data center can go offline without affecting the management of the cluster.

The cluster has six worker nodes. Three in Kajaani and three in Espoo. A worker node should have sufficient resources to run on the production deployment of B2SHARE. Each B2SHARE application stack is replicated to Espoo and Kajaani. Thus, all deployments can handle an outage of one data center. Because each deployment is replicated to two nodes, the deployment can withstand the planned downtime of a worker node.

Because the cluster is distributed to two data centers, good cluster configuration is important. The configuration of databases and storage for distributed computing is especially important due to data recovery in case of a data center outage. Two scenarios are presented next. Both scenarios have different recovery requirements.

Scenario one: Datacenter one goes offline due to a power outage and all nodes in that datacenter shut down. All requests are routed to the deployments in datacenter two. Database operations and disk write operations happen in datacenter two. When datacenter one is back online, data can be copied from datacenter two to datacenter one, because data in datacenter one is at the last known common state

Scenario two: Datacenter one goes offline due to a network problem. All requests are routed to the deployments in data center two which has network connectivity. Deployments in the data center two make normal database operations etcetera. In this scenario, nodes in datacenter one are isolated, but running. Thus, some scheduled tasks may run in datacenter one leading to database changes or other changes in data. When datacenter one goes back online, databases etcetera. have diverged from the last known common

state. The data has to be merged somehow. One solution is for the cluster to check connectivity to the internet. If the connectivity is lost, no data operations are done.

6 Conclusions

This thesis evaluated and compared two different container orchestrators for cluster deployments. The two container orchestrators are Kubernetes and Docker Swarm. The evaluation and comparison was made to find out, if B2SHARE - a containerized uWSGI application - can have near-zero downtime application updates and near-zero application downtime server maintenance, and what are the infrastructure costs of these two cluster implementations. The implementations were evaluated against the baseline B2SHARE deployment with Docker Compose.

The measurements done in the research in this thesis were done with a simple HTTP request script. The measurements have resolution of 0.5 seconds. The effect of network latency between the location measurements were made and the datacenter was not major.

This thesis proved, tha near-zero downtime application updates and near-zero application downtime server maintenance can be implemented easily and efficiently with Docker Swarm and Kubernetes. As a side effect, this thesis provided methods for achieving resilience for unplanned server downtime with B2SHARE deployments.

For load balancing and redirecting requests to servers that are functioning, DNS round robin load balancing was used. This method turned out to be a good enough solution for load balancing requests to the cluster. Thus, the implemented clusters did not have any external load balancer servers to redirect requests to virtual machines in the cluster. However, both implemented clusters have some king of load balancers to route traffic inside the cluster.

Both implemented clusters - Kubernetes and Docker Swarm - are cost effective methods for deploying containerized application when comparing infrastructure costs. Docker Swarm is marginally cheaper compared to Kubernetes and Docker Swarm has approximately the same infrastructure cost as the baseline deployment with Docker Compose. Thus, if a goal is for a Docker Compose application deployment to have near-zero application updates, it is easy to convert it to Docker Swarm deployment.

In the research process, it was noticed, that Docker Swarm is more user friendly and offers "turn key" cluster implementations for containerized applications. Kubernetes in the other hand is more configurable by the cluster operators and offers a lot more features. Based

on online documentation and previous research, Kubernetes is becoming the industry standard and it is preferred over Docker Swarm for cluster implementations.

Data management for implementations in this thesis was done with a Cinder multi-attach disk. A multi-attach disk is a storage device that can be attached to multiple virtual machines at once. The thesis described other data management methods for distributed computing, such as SAN and GlusterFS, that are potential data management systems for clustered production implementations.

The research was done with Design Science Research methodology. Design Science Research proved it self for being a good research method for doing research with infromation systems. Design Science Research is a cyclic research method that is used to create design artifacts from each design cycle. Evaluating each artifact after each design cycle created a possibility for fine tuning the implementations on following design cycles.

For evaluating the implementations, a simple HTTP request python script was implemented and it was seen good enough for this thesis. This method was used to measure, if the application has downtime or not by querying the API root of B2SHARE. However, the HTTP request test did not test, if the application is fully functional or not.

Based on the amount of documentation and previous work, it could be stated, that Kubernetes is a better option compared to Docker Swarm. However, the research done for this thesis did not find such evidence for small scale cluster deployments. In the other hand, for large scale cluster deployments and deployments with multiple applications, using Kubernetes is required.

For future work, it would be a good idea to implement production grade clusters with Docker Swarm and Kubernetes with more virtual machines, comparing these implementations and evaluating high availability of these implementations. For the Kubernetes implementation three control plane nodes and at least three worker nodes should be sufficient. For the Docker Swarm at least three manager nodes should be sufficient.

Another future research topic is to compare different methods of storage for distributed computing and how well different storage methods integrate with containerized application deployments in clusters. The comparison could evaluate the fault tolerance of different storage methods and suitability for high availability deployments.

As a last future research topic proposal, comparison of DNS load balancing with dedicated load balancing servers could be done. The research could compare load balancing implementations with both methods.

6.1 Acknowledgements

I wish to acknowledge and thank CSC – IT Center for Science, Finland, for generous computational resources and for funding the research part. I want to thank Harri Hirvonsalo for brainstorming the topic for this thesis. Thank you for the whole B2SHARE development team at CSC for supporting me during this research.

6.2 Usage of large language models

No large language model have been used in this thesis.

Bibliography

[1] R. Alimi, A. Rahman, and Y. Yang. *RFC 6392: A Survey of In-Network Storage Systems*. USA, 2011.

[2] F. Beetz and S. Harrer. "GitOps: The Evolution of DevOps?" In: *IEEE Software* 39.4 (2022), pp. 70–75. DOI: 10.1109/MS.2021.3119106.

[3] N. Bhaia, L.-H. Hung, R. Cordingly, and W. Lloyd. "Understanding Container Isolation: An Investigation of Performance Implications of Container Runtimes". In: *Proceedings of the 9th International Workshop on Container Technologies and Container Clouds*. WoC '23. Bologna, Italy: Association for Computing Machinery, 2024, pp. 7–12. ISBN: 9798400704598. DOI: 10.1145/3631311.3632400. URL: https://doi.org/10.1145/3631311.3632400.

[4] A. Bozhikov. "Business Downtime and its Impact on Business Organizations". In: *CONFERENCE PROCEEDINGS - INTERNATIONAL CONFERENCE ON APPLICATION OF INFORMATION AND COMMUNICATION TECHNOLOGY AND STATISTICS IN ECONOMY AND EDUCATION (ICAICTSEE-2012)* (Jan. 2012), pp. 546–552.

[5] T. Brisco. *RFC1794: DNS Support for Load Balancing*. USA, 1995.

[6] B. Carver, R. Han, J. Zhang, M. Zheng, and Y. Cheng. "FS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ASPLOS '23. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 394–411. ISBN: 9798400703942. DOI: 10.1145/3623278.3624765. URL: https://doi.org/10.1145/3623278.3624765.

[7] A. M. Caulfield and S. Swanson. "QuickSAN: a storage area network for fast, distributed, solid state disks". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. New York, NY, USA: Association for Computing Machinery, June 2013, pp. 464–474. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485962. URL: https://dl.acm.org/doi/10.1145/2485922.2485962 (visited on 11/03/2024).

[8] M. L. Chin, C. E. Tan, and M. I. Bandan. "Efficient load balancing for bursty demand in web based application services via domain name services". In: *8th Asia-Pacific Symposium on Information and Telecommunication Technologies*. 2010, pp. 1–4.

[9] *Docs CSC, User guides and tutorials*. URL: https://docs.csc.fi/ (visited on 09/28/2024).

[10] *Elastic's Version Policy and Product End of Life Dates*. URL: https://www.elastic.co/support/eol (visited on 11/24/2024).

[11] P. T. Endo, M. Rodrigues, G. E. Gonçalves, J. Kelner, D. H. Sadok, and C. Curescu. "High availability in clouds: systematic review and research challenges". In: *Journal of Cloud Computing* 5.1 (Oct. 2016), p. 16. ISSN: 2192-113X. DOI: 10.1186/s13677-016-0066-8. URL: https://doi.org/10.1186/s13677-016-0066-8.

[12] F. Eyvazov, T. E. Ali, F. I. Ali, and A. D. Zoltan. "Beyond Containers: Orchestrating Microservices with Minikube, Kubernetes, Docker, and Compose for Seamless Deployment and Scalability". In: *2024 11th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2024, pp. 1–6. DOI: 10.1109/ICRITO61523.2024.10522382.

[13] R. Fielding, M. Nottingham, and J. Reschke. *RFC 9110: HTTP Semantics*. USA, 2022. DOI: 10.17487/RFC9110. URL: https://doi.org/10.17487/RFC9110.

[14] *Chapter 1. GFS2 Overview*. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/global_file_system_2/ch-overview-gfs2 (visited on 11/10/2024).

[15] *Gluster Docs: Architecture*. URL: https://docs.gluster.org/en/latest/Quick-Start-Guide/Architecture/ (visited on 11/10/2024).

[16] B. T. Hasan and D. B. Abdullah. "Real-Time Resource Monitoring Framework in a Heterogeneous Kubernetes Cluster". In: *2022 Muthanna International Conference on Engineering Science and Technology (MICEST)*. 2022, pp. 184–189. DOI: 10.1109/MICEST54286.2022.9790264.

[17] T. Haynes and D. Noveck. *RFC 7530: Network File System (NFS) Version 4 Protocol*. USA, 2015.

[18] A. Hevner. "A Three Cycle View of Design Science Research". In: *Scandinavian Journal of Information Systems* 19 (Jan. 2007).

[19] A. R. Hevner, S. T. March, J. Park, and S. Ram. "Design science in information systems research". In: *MIS Q.* 28.1 (Mar. 2004), pp. 75–105. ISSN: 0276-7783.

[20] D. A. Kimber, X. Zhang, P. H. Franklin, and E. J. Bauer. "Modeling planned downtime". In: *Bell Labs Technical Journal* 11.3 (2006), pp. 7–19. DOI: 10.1002/bltj.20174.

[21] *Creating Highly Available Clusters with kubeadm*. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/#before-you-begin (visited on 12/14/2024).

[22] *Kubernetes documentation*. URL: https://kubernetes.io/docs/home/ (visited on 10/20/2024).

[23] *Longhorn*. URL: https://longhorn.io (visited on 12/14/2024).

[24] P. Macko and J. Hennessey. "Survey of Distributed File System Design Choices". In: *ACM Trans. Storage* 18.1 (Mar. 2022). ISSN: 1553-3077. DOI: 10.1145/3465405. URL: https://doi.org/10.1145/3465405.

[25] A. Malhotra, A. Elsayed, R. Torres, and S. Venkatraman. *Evaluate Solutions for Achieving High Availability or Near Zero Downtime for Cloud Native Enterprise Applications*. en-US. DOI: 10.1109/ACCESS.2023.3303430. URL: https://ieeexplore.ieee.org/document/10214005 (visited on 06/30/2024).

[26] N. Marathe, A. Gandhi, and J. M. Shah. "Docker Swarm and Kubernetes in Cloud Computing Environment". In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. 2019, pp. 179–184. DOI: 10.1109/ICOEI.2019.8862654.

[27] M. Moravcik, M. Kontsek, P. Segec, and D. Cymbalak. "Kubernetes - evolution of virtualization". In: *2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. Oct. 2022, pp. 454–459. DOI: 10.1109/ICETA57911.2022.9974681. URL: https://ieeexplore.ieee.org/document/9974681 (visited on 11/17/2024).

[28] U. Naseer, L. Niccolini, U. Pant, A. Frindell, R. Dasineni, and T. A. Benson. "Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 529–541. ISBN: 9781450379557. DOI: 10.1145/3387514.3405885. URL: https://doi.org/10.1145/3387514.3405885.

[29] N. C. Nelson. "Downtime procedures for a clinical information system: a critical issue". In: *Journal of Critical Care* 22.1 (2007), pp. 45–50. ISSN: 0883-9441. DOI: 10.1016/j.jcrc.2007.01.004. URL: https://www.sciencedirect.com/science/article/pii/S088394410700010X.

[30] N. T. Nguyen and Y. Kim. "A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes Cluster". In: *2022 27th Asia Pacific Conference on Communications (APCC)*. 2022, pp. 651–654. DOI: 10.1109/APCC55198.2022.9943782.

[31] *Nomad by HashiCorp*. URL: https://www.nomadproject.io (visited on 11/10/2024).

[32] M. O'Keefe, S. Soltis, and T. Ruwart. "The Global File System". In: *High Performance Mass Storage and Parallel I/O: Technologies and Applications* (Oct. 2000).

[33] *About OCFS2*. URL: https://docs.oracle.com/en/operating-systems/oracle-linux/6/admin/ol_about_ocfs2.html (visited on 11/10/2024).

[34] *Open Container Initiative*. URL: https://opencontainers.org (visited on 11/17/2024).

[35] P. Ounsrimuang and S. Nootyaskool. "Introducing scrum process optimization". In: *2017 International Conference on Machine Learning and Cybernetics (ICMLC)*. Vol. 1. 2017, pp. 175–181. DOI: 10.1109/ICMLC.2017.8107761.

[36] C. Pahl. *Containerization and the PaaS Cloud*. en-US. DOI: 10.1109/MCC.2015.51. URL: https://ieeexplore.ieee.org/document/7158965 (visited on 07/21/2024).

[37] R. Queiroz, T. Cruz, J. Mendes, P. Sousa, and P. Simões. "Container-based Virtualization for Real-time Industrial Systems—A Systematic Review". In: *ACM Comput. Surv.* 56.3 (Oct. 2023), 59:1–59:38. ISSN: 0360-0300. DOI: 10.1145/3617591. URL: https://dl.acm.org/doi/10.1145/3617591 (visited on 09/01/2024).

[38] *Understanding Containers*. URL: https://docs.openshift.com/container-platform/4.12/nodes/containers/nodes-containers-using.html (visited on 11/17/2024).

[39] C. K. Rudrabhatla. "Comparison of zero downtime based deployment techniques in public cloud infrastructure". In: *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. 2020, pp. 1082–1086. DOI: 10.1109/I-SMAC49090.2020.9243605.

[40] *opencontainers/runc*. URL: https://github.com/opencontainers/runc/tree/v1.2.2 (visited on 11/17/2024).

[41] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. "Design and implementation of the Sun network filesystem". In: *Innovations in Internetworking*. USA: Artech House, Inc., 1988, pp. 379–390. ISBN: 0890063370.

[42] F. Semchedine, L. Bouallouche-Medjkoune, O. Sayeh, S. Ayoub, and D. Aïssani. "DNS-based load balancing with Cache for geographically distributed Web server systems". In: *2014 Global Summit on Computer Information Technology (GSCIT)*. 2014, pp. 1–6. DOI: 10.1109/GSCIT.2014.6970100.

[43] V. Sharma, H. K. Saxena, and A. K. Singh. "Docker for Multi-containers Web Application". In: *2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*. 2020, pp. 589–592. DOI: 10.1109/ICIMIA48430.2020.9074925.

[44] M. Šimon, L. Huraj, and N. Búčik. "A Comparative Analysis of High Availability for Linux Container Infrastructures". en. In: *Future Internet* 15.8 (Aug. 2023). Number: 8 Publisher: Multidisciplinary Digital Publishing Institute, p. 253. ISSN: 1999-5903. DOI: 10.3390/fi15080253. URL: https://www.mdpi.com/1999-5903/15/8/253 (visited on 07/23/2024).

[45] *[MS-SMB]: Server Message Block (SMB) Protocol*. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb/f210069c-7086-4dc2-885e-861d837df688 (visited on 11/10/2024).

[46] S. R. Soltis, G. M. Erickson, K. W. Preslan, M. T. O'Keefe, and T. M. Ruwart. "The Global File System: A File System for Shared Disk Storage". In: *IEEE Transactions on Parallel and Distributed Systems* (1997).

[47] M. Straesser, J. Mathiasch, A. Bauer, and S. Kounev. "A Systematic Approach for Benchmarking of Container Orchestration Frameworks". In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. ICPE '23. Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 187–198. ISBN: 9798400700682. DOI: 10.1145/3578244.3583726. URL: https://doi.org/10.1145/3578244.3583726.

[48] *Administer and maintain a swarm of Docker Engines*. URL: https://docs.docker.com/engine/swarm/admin_guide/ (visited on 09/28/2024).

[49] *The uWSGI project*. URL: https://uwsgi-docs.readthedocs.io/en/latest/ (visited on 12/14/2024).

[50] A. Valantasis, N. Makris, and T. Korakis. "Orchestration Software for Resource Constrained Datacenters: an Experimental Evaluation". In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 2022, pp. 121–126. DOI: 10.1109/NetSoft54395.2022.9844043.

[51] *Understanding VMFS Datastores*. URL: https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.storage.doc/GUID-5EE84941-366D-4D37-8B7B-767D08928888.html (visited on 11/10/2024).

[52] *What is Storage Area Network (SAN)? | VMware Glossary*. en. URL: https://www.vmware.com/topics/storage-area-network-san (visited on 11/10/2024).

[53] S. Volpert, S. Winkelhofer, S. Wesner, and J. Domaschka. "An Empirical Analysis of Common OCI Runtimes' Performance Isolation Capabilities". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. London, United Kingdom: Association for Computing Machinery, 2024, pp. 60–70. ISBN: 9798400704444. DOI: 10.1145/3629526.3645044. URL: https://doi.org/10.1145/3629526.3645044.

[54] J. Xiang and L. Chen. "A Method of Docker Container Forensics Based on API". In: *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*. ICCSP 2018. New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 159–164. ISBN: 978-1-4503-6361-7. DOI: 10.1145/3199478.3199506. URL: https://dl.acm.org/doi/10.1145/3199478.3199506 (visited on 09/01/2024).

[55] B. Yang, A. Sailer, S. Jain, A. E. Tomala-Reyes, M. Singh, and A. Ramnath. *Service Discovery Based Blue-Green Deployment Technique in Cloud Native Environments | IEEE Conference Publication | IEEE Xplore*. DOI: 10.1109/SCC.2018.00031. URL: https://ieeexplore.ieee.org/document/8456417 (visited on 09/01/2024).

Appendix A Measurements

Measureming application downtime is done with a Python script that sends a HTTP request approximately every 0.5 seconds. The script is provided in appendix B.

A.1 Baseline

Baseline update time measurements were done ten times to get a good understanding of downtime during application updates. The baseline deployment is a Docker Compose deployment. To simulate downtime, application container was restarted and measurements were made based on HTTP return codes. The application container was restarted with:

```
docker compose restart b2share
```

A.2 Kubernetes

Kubernetes application update time measurements are made by analysing the pod description of B2SHARE pod. In the pod description Kubernetes provides a status history for a pod. The update time is the time elapsed between status types "PodScheduled" and "Ready". The command for describing a pod is:

```
kubectl describe pod <pod identifier> -o yaml
```

The describe command gives a description of the selected pod. Status changes of the pod is logged in the description:

```
status:
  conditions:
    - lastProbeTime: null
      lastTransitionTime: "2024-11-02T11:32:41Z"
      status: "True"
      type: PodReadyToStartContainers
    - lastProbeTime: null
```

```
      lastTransitionTime: "2024-11-02T11:32:40Z"
      status: "True"
      type: Initialized
    - lastProbeTime: null
      lastTransitionTime: "2024-11-02T11:33:40Z"
      status: "True"
      type: Ready
    - lastProbeTime: null
      lastTransitionTime: "2024-11-02T11:33:40Z"
      status: "True"
      type: ContainersReady
    - lastProbeTime: null
      lastTransitionTime: "2024-11-02T11:32:40Z"
      status: "True"
      type: PodScheduled
```

A.3 Docker Swarm

The application update time for both (not-replicated and replicated) Docker Swarm implementations were made with time command:

```
time docker service update --force b2share
```

The time command measures, how long the proceeding command takes to run completely. In this case it measures, how long it takes to update the application. The update command finishes after Docker has verified, that the created container is working correctly. The elapsed time is the total elapsed time provided by the time command. A sample output for time command:

```
docker service update --force b1_b2share2 \
    0.37s user 0.19s system 1% cpu 54.676 total
```

Appendix B Python script for HTTP measurements

Application downtime is measured based on HTTP status codes. If the application is working correctly the status code is 200, if the service is unavailable, the status code is 502. Timeout requests are not counted as downtime. Python script used for measuring downtime:

```
import requests
import time
import sys
from datetime import datetime

results = []

iterations = 240
timeout = 0.5
first = 0
last = 0

URL = "https://gradu3.usoft.fi"

prev_code = 200
try:
    while True:
        start = time.time()
        try:
            r = requests.get(URL, timeout=timeout)
            # print(r.status_code, time.time())
            results.append(str(r.status_code) + "," + str(time.time()))
            stat = r.status_code
        except requests.Timeout:
            results.append("Timeout,"+str(time.time()))
            stat = 502
        except Exception as e:
```

```
            results.append("Exception,"+str(time.time()))
            print(e)
            stat = 502
        if not prev_code == 200 and stat == 200:
            last = time.time()
        if prev_code == 200 and not stat == 200:
            first = time.time()
        prev_code = stat
        end = time.time()
        elapsed = end-start
        if elapsed < timeout:
            time.sleep(timeout-elapsed)
except KeyboardInterrupt:
    pass

fname = sys.argv[1]
dt = datetime.now().strftime('%Y-%m-%d_%H:%M:%S')
split = fname.split('/')
split[-1] = dt + '-' + str(len(results)) + '-' + split[-1]
fname = '/'.join(split)
f = open(fname, 'w')
f.write('status, time\n')
for l in results:
    f.write(l)
    f.write('\n')
f.close()

print("First non 200", first)
print("First 200", last)
print("Downtime", last-first)
```

The script labels requests with Timeout with status code 502 for calculating elapsed time between 200 OK responses. All provided datasets were checked and if dataset contained Timeouts, those were subtracted from calculated downtime.