Enhancing Availability of Microservice Architecture

A Case Study on Kubernetes Security Configurations

Nadin Habbal

Systems Sciences, bachelor's level
2020

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

Abstract

The objective of this report is to enhance availability in a microservice architecture for critical systems by studying Kubernetes security configurations.
As a theoretical framework, various categories of failure (crash, omission, value, timing and byzantine) and Kubernetes' security actions (securing the cluster, managing authorization and authentication, implementing a trusted software supply chain, securing workloads in runtime and managing secrets) were applied to target relevant data. A case study method was adopted and the data was collected by performing an interview and meetings and, also, by sending questionnaires.
The conclusion indicated that Kubernetes security configuration could solve some of the availability concerns in a microservice architecture for critical systems, but not entirely all categories of failure and not to a full extent.

Keywords: Cloud Computing, Microservices, Microservices Architecture, Availability, Critical Systems, Kubernetes, Cloud Security, Monitoring, IT-security

Acknowledgements

I would like to thank my advisors, Mr. Svante Edzen and Mr. Ingemar Andersson for their guidance and support. Their consultation has been helpful and vital to completing my Bachelor’s degree at Lulea University of Technology.

Also, I would like to thank Skatteverket and especially Mr. Fredrik B. Andersson for giving me the fine opportunity to write my thesis in collaboration with Skatteverket. A special thanks to my supervisor Ms. Ülker Kayhan at Skatteverket for her vast efforts, time, and great encouragement. Her support was excellent. I would like to thank Mr. Claudio Meneses Marshall at Skatteverket, who has given me central knowledge of microservice architecture and its functionality. His skillful input and generous knowledge sharing were invaluable and a wonderful experience. I would also like to thank Mr. Sadmir Halilovic for his vital contribution, time, and important insights on Kubernetes and microservices. Finally, I would like to thank Mr. Jan Videsäter at Skatteverket for his interesting presentation on the platform and precious time.

Stockholm, 2020, June
Nadin Habbal

Contents

List of Figures and Tables

1.0 Research Problem

Cloud-based business solutions have reached exponential growth in the last ten years and are expected to grow further. A set of computing resources (hardware, CPU, storage, software, etc.) are managed by cloud service providers. They offer their services in the form of infrastructure, platform, and software to its consumers, mostly over the Internet using multi-tenancy and resource virtualization techniques (Kumar & Goyal, 2019).
The unique characteristics of a cloud e.g. pooled sharable resources, on-demand scalability, and customized self-service, have accelerated the growth of cloud-based business use cases and applications. The cloud-based applications and services have attracted research communities, both from industry and academia, to find innovative solutions for more consumer-friendly, cost-effective, technologically efficient, and secure cloud systems (Kumar & Goyal, 2019).

One of the cloud-enabling technology, web services, and service-oriented architecture (SOA) provides an architectural framework that enables the communication between systems through interacting services implemented by the systems. Some of the most frequently utilized technologies for implementing the web services are; Hypertext Transfer Protocol, HTTP; Simple Object Access Protocol, SOAP; Representational State Transfer, REST; eXtensible Markup Language, XML; Web Services Description Language, WSDL and Universal Description, Discovery and Integration, UDDI (Kumar & Goyal, 2019). Service-Oriented Architecture (SOA) is a close relative of microservices (Yarygina, 2018).

The microservice architecture implements modular structure and strict separation of concerns that allows creating highly scalable and flexible distributed systems. The process of gradually transitioning from a modular non-distributed system towards microservices is shown in Figure 1 (Yarygina, 2018).

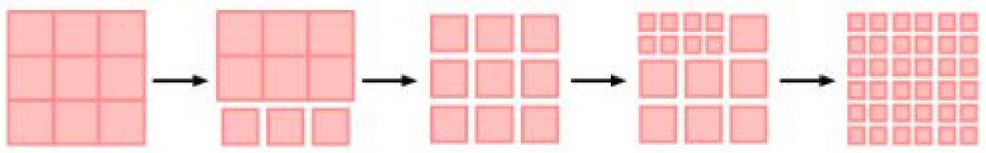


Figure 1: Transitioning to Microservices. The granularity of components increases from left to right (Yarygina, 2018).

However, microservices have some challenges in regards to fault tolerance, software testing, distributed transactions, and data consistency, and infrastructure complexity. to handle. (Yarygina, 2018).

Microservices architecture encompasses several layers and therefore this brings more complex security concerns. The table below provides an understanding of the security issues in each layer (table 1) (Yarygina, 2018).

LAYER	**THREAT EXAMPLES**
Orchestration	Management, coordination, and automation of service-related tasks, including scheduling and clustering of services. Microservice network structure may change continuously due to services being stopped, started, and moved around; service discovery provides a DNS-like central point for locating services. Attacks include: compromising discovery service and registering malicious nodes within the system, redirecting communication to them.
Application	Typical and still very common application-level security problems are

	SQL injection flaws, broken authentication and access control, sensitive data exposure, Cross-Site Scripting (XSS), insecure deserialization, general security misconfiguration. The ten most critical web application security risks are published annually by OWASP.
Communication	Classic attacks on the network stack and protocols; attacks against protocols specific to the service integration style (SOAP, RESTful Web Services. Attacks include: eavesdropping (sniffing), identity spoofing, session hijacking, Denial of Service (DoS), and Man-in-the-Middle (MITM).
Cloud	Cloud computing brings a myriad of security concerns, including unlimited control of cloud provider over everything it runs; there are few technical options to prevent disruption or attacks from a malicious provider.
Virtualization	Deployment affects security; OS processes offer little separation from other services in the same system; containers and VMs offer more protection against compromised ser-

	vices. Attacks include sandbox escape, hypervisor compromise, and shared memory attacks; also, the use of malicious and/or vulnerable images is another serious security concern.
Hardware	Hidden under abstraction, but still a reliability and security concern; hardware bugs are extremely dangerous because they undermine security mechanisms of other layers; hardware backdoors.

Table 1: Decomposition of Microservices Security Issues into Layers (Yarygina, 2018).

While having these security concerns in mind, FRA, the National Defence Radio Establishment in Sweden concludes that cyberattacks targeting different Swedish locations have increased. In 2017 FRA discovered 10 000 foreign activities/month, aiming for Swedish organizations (Olsson, 2019, January 15). Moreover, citizens in developed nations are more likely to become victims of cybercrime. Some key reasons are:

- Higher-income economies
- More advanced technological infrastructure
- Greater urbanization
- Greater digitalization

Among the countries at the greatest risk was Iceland at the top, followed by Sweden. Sweden took top place because it was highest among all analyzed countries in the internet, smartphone, and Instagram penetration. Also, Sweden came in second on Facebook penetration (Whitney, 2020, May 27).
It could be concluded that IT-security in general terms and cloud security in specific terms are considered to be some of the top priorities for critical systems.

Cloud solutions require comprehensive thought and effort in particular when it concerns enforcing cloud security (Kumar & Goyal, 2019).
Hence the objective of this paper is to study how current technology may strengthen cloud security. Kubernetes and its security features were selected for this purpose as Kubernetes is the most widely used management system for containers along with a massive community behind it. Kubernetes is currently provided to deploy scalable microservices. Nowadays, almost all engineering positions from software engineers to site reliability engineers one way or another deal with Kubernetes (Taherizadeh & Grobelnik, 2020).

As previously mentioned cyber attacks have increased in recent years (Olsson, 2019, January 15) and therefore it is relevant to study how Kubernetes may preclude security issues in a microservice architecture for critical systems.
However, cloud security has multiple aspects and for this paper, availability is selected as the key aspect of cloud security. Critical systems require high availability as the lack of availability would seriously disrupt society (Aven, 2009). Availability is, therefore, relevant to study and it has been widely discussed to understand how to strengthen a microservice architecture for critical systems. Although, Kubernetes' security actions as a solution to availability concerns in for critical systems deployed as a microservice architecture have not been previously studied (Smith, Trivedi, Tomek, Ackaret, 2008., Brendan, Lefebvre, Meyer, Feeley, Hutchinson, Warfield, 2008., Machida, Andrade, Kim Seong, Trivedi, 2011., Machida, Kim Seong, Trivedi, 2009).

However, Vayghan, Saied, Toeroe, and Khendek (2019) evaluated microservices-based applications from the availability perspective by utilizing the default Kubernetes configuration for healing (Vayghan, Saied, Toeroe, Khendek, 2019), but the study did not take into account how specifically Kubernetes' security actions may enhance the availability of a critical system with a microservice architecture. This paper focused on each Kubernetes' security action and if it enabled availability by solving the diverse types of failure. Also, this report aimed at studying to what extent each Kubernetes' security action resolved different types of failure affecting availability in a microservice architecture for critical systems. Furthermore, the results indicated how critical it is for business when a specific failure in a critical system set up as a microservice architecture is not dissolved by Kubernetes' security action.

Taherizadeh and Grobelnik (2020) presented influencing factors in the dynamic management of scalable resources provided by Kubernetes. Also, the study evaluated the choices of such factors to develop the optimum scaling strategy to be used and analyzing the way how they dynamically influence the impact of reactive auto-scaling rules. Finally, the results demonstrated the way to tune the auto-scaling of containerized applications orchestrated by Kubernetes concerning diverse workload patterns. The report did not present if and to what extent each Kubernetes' security action could solve types of failures affecting availability in a microservice architecture for critical systems. Moreover, Taherizadeh and Grobelnik (2020) did not indicate how severe it is for a specific business with critical systems having unresolved failures (Taherizadeh & Grobelnik, 2020). These aspects are studied in this paper.

1.1 Research Objective and Research Questions

Hence, there is a knowledge gap to fill and the objective is to enhance the availability of microservice architecture for critical systems by studying Kubernetes security configurations.

The objective is broken down into the following research question:

- How could Kubernetes security configurations enhance the availability of microservice architecture for critical systems?

1.2 Layout

Initially, this report describes some central concepts about distributed systems to clarify the context. The second section presents the research design, which is followed by a section on the theoretical framework. The theoretical framework offers an understanding of the diverse types of failure, the levels of severity and, also Kubernetes' security actions are explained. In the result section, the different types of failure at Skatteverket are described. The failures are labelled to set a level of severity. The section on analysis indicates if Kubernetes' security configurations fully solve, partially solve, or not at all the diverse categories of failure. The analysis is summarized by presenting a table. As a final section, the results are discussed and a conclusion is specified.

1.3 Definitions

1.3.1 Availability

In simple terms the National Institute of Standards and Technology, NIST defines availability as the extent to which an organization's full set of computational resources is accessible and usable. Availability can be affected temporarily or permanently, and a loss can be partial or complete. Denial of service attacks, equipment outages, and natural disasters are all threats to availability. The concern is that most downtime is unplanned and can impact the mission of the organization. NIST is a physical sciences laboratory and a non-regulatory agency of the United States Department of Commerce. Its mission is to promote innovation and industrial competitiveness (National Institute of Standards and Technology, 2017).

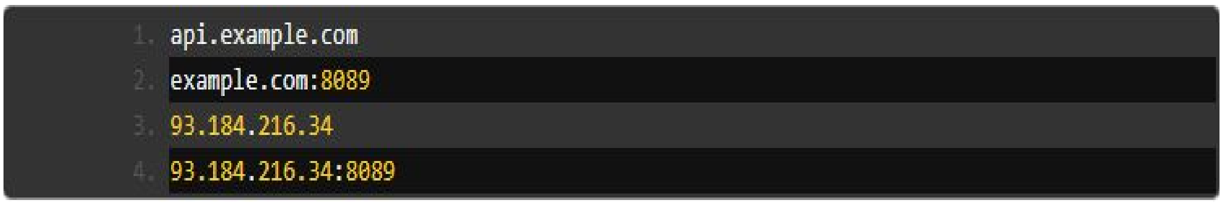
According to Phaphoom, Wang and Abrahamsson availability refers to a percentage of time that the services are up and available for use. SLA contracts might use a more strict definition of availability by counting on uptime that respects the quality level specified in the SLA (Phaphoom, Wang, Abrahamsson, 2013).

High availability configurations generally aim to eliminate as many single points of failure as possible (Loveland., Dow., LeFevre., Beyer., Chan, 2008).
Availability is also explained as the capability of guaranteeing continuous access to data and resources by authorized clients (Ardagna et al., 2015).

In this paper NIST definition of availability has been applied, but the definition has not the authorization aspect, therefore, availability is referred to accessible and usable resources to authorized clients (Ardagna et al., 2015).

1.3.2 A host

A host is the domain name or IP address (IPv4) of the host that serves the API. It may include the port number if different from the scheme's default port (80 for HTTP and 443 for HTTPS). Note that this must be the host only, without http(s):// or sub-paths (Figure 2) (Swagger, n.d.).



```
api.example.com
example.com:8089
93.184.216.34
93.184.216.34:8089
```

Figure 2: Valid Hosts (Swagger, n.d.).

1.3.3 A Service

Service in this study is defined as Yarygina described it in her thesis:

"A service is a self-contained unit of business functionality that can be accessed remotely and may consist of other underlying services. Communication between service occurs through network calls rather than system calls."
(Yarygina, 2018, p. 22).

1.3.4 A Service-Level Agreement, SLA

A service-level agreement (SLA) is a commitment between a service provider and a client. Particular aspects of the service – quality, availability, responsibilities – are agreed between the service provider and the service user. The most common component of an SLA is that the services should be provided to the customer as agreed upon in the contract (Service-Level Agreement, 2020).

1.3.5 Docker

Docker is a container platform (platform as a service, PaaS) for application/microservices development and delivery (Docker, n.d.). Docker uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their software, libraries, and configuration files. Furthermore, the containers communicate with each other through defined channels. All containers are run by a single operating system kernel and therefore use fewer resources than virtual machines (Docker, 2020).

1.3.6 Kubernetes, K8s

Kubernetes (also known as k8s or “kube”) is an open-source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications. It is possible to cluster together groups of hosts running Linux containers, and Kubernetes assists in managing those clusters (Red Hat, n.d.). Kubernetes was developed to enable deployment of multiple containers to multiple hosts, which was not possible when utilizing Docker (Wallen, 2017, May 10). Kubernetes was originally developed and designed by engineers at Google. Google was one of the early contributors to Linux container technology as Google generates more than 2 billion container deployments a week. Red Hat was one of the first companies to work with Google on Kubernetes. Moreover, in 2015 Google donated the Kubernetes project to the newly formed Cloud Native Computing Foundation, CNCF (Red Hat, n.d.).
Kubernetes terminology is quite extensive and therefore the most frequent components of Kubernetes are described below:

- A *master* is a machine, which controls Kubernetes nodes. The master determines all task assignments (Red Hat, n.d.).

- A *node, worker node* is the machine that performs the requested, assigned tasks. The Kubernetes master controls all nodes (Red Hat, n.d.).
- A *pod* is a group of one or more containers deployed to a single node. All containers in a pod share an IP address, IPC, hostname, and other resources. Pods abstract network and storage from the underlying container (Red Hat, n.d.). In other words, a pod simply indicates one single instance of an application which can be replicated, if more instances are helpful to handle the increasing workload (Taherizadeh & Grobelnik, 2020).
- The *Replication controller* controls how many identical copies of a pod should be running somewhere on the cluster (Red Hat, n.d.).
- A *Service* decouples work definitions from the pods. Kubernetes service proxies automatically get service requests to the right pod, even though the pod might be moved in the cluster or replaced by another pod (Red Hat, n.d.).
- A *kubelet* runs on nodes reads the container manifests and ensures the defined containers are started and running (Red Hat, n.d.).
- A *kubectl* is the command-line configuration tool for Kubernetes (Red Hat, n.d.).

A *cluster* is referred to as the working Kubernetes deployment and it consists of two parts: the control plane (includes the master node or nodes) and the compute machines (encompass the worker nodes).
The worker nodes run pods, which are grouped in multiple containers. Each node has its environment (e.g. Linux), which could be a physical or virtual machine.
The master node is responsible for maintaining the desired state of the cluster, e.g. which applications are running and which container images they use. Worker nodes execute the desired state and run therefore the applications and workloads. In other words, it is automatically decided by the master node which node is best suited for executing the task. Subsequently, the master node allocates resources and assigns the selected node to fulfil the requested work. The figure below visualizes on an aggregated level the architecture of Kubernetes (Red Hat, n.d.).

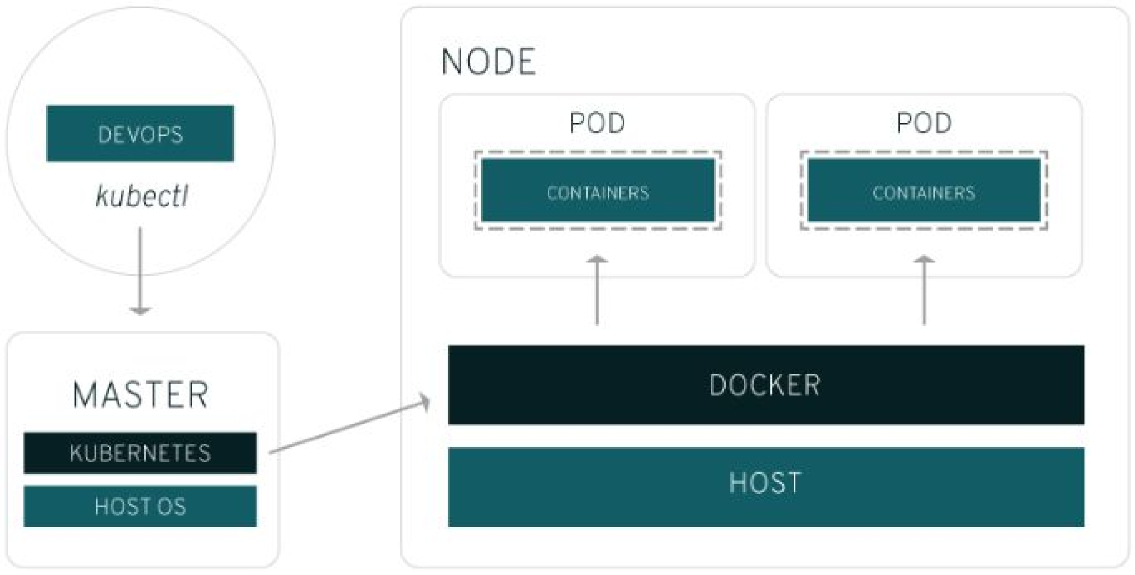


Figure 3: Overview of Kubernetes Architecture (Red Hat, n.d.)

However, as the actual work is processed inside the cluster it is important to understand how the components of a cluster interact. Initially, the components are explained below.

- The *Control plane* consists of the master node along with its data on the cluster's state and configuration (Red Hat, n.d.).
- A *kube-apiserver* is the front end of the Kubernetes control plane and handles therefore all internal and external requests. The API server determines if a request is valid for processing. The API is accessible through REST calls, through the kubectl command-line interface, or other command-line tools such as kubeadm (Red Hat, n.d.).
- A *kube-scheduler* manages the health of the cluster. Concerns such as if the number of containers is sufficient and where they are needed are handled by the kube-scheduler. Moreover, the scheduler considers the resource needs of a pod, e.g.CPU, or memory, and then it schedules the pod to an appropriate worker node (Red Hat, n.d.).
- A *kube-controller-manager* runs the cluster contains several controller functions in one. One controller consults the scheduler to ensure the correct number of running pods. If a pod fails and stops to process its work, another

controller notices the failure and responds. A controller connects services to pods to send requests to the right endpoints. Also, there are controllers for creating accounts and API access tokens (Red Hat, n.d.).

- The *etcd* is a key-value store database, which holds all configuration data and information about the state of the cluster (Red Hat, n.d.).
- The *Container runtime engine* runs the containers as each worker node has a container runtime engine. Docker is one example (Red Hat, n.d.).
- A *kubelet* is contained inside each worker node. A kubelet is an application that communicates with the master node. The kubelet ensures that the containers are running in a pod and execute the assignment specified by the master node (Red Hat, n.d.).
- The *kube-proxy* is encompassed by a worker node along with a kube-proxy and a network proxy for facilitating Kubernetes networking services. The kube-proxy manages network communications inside or outside the cluster (Red Hat, n.d.).
- The *Persistent storage* manages the application data attached to a cluster. Persistent volumes are, therefore, specific to a cluster, rather than a pod (Red Hat, n.d.).
- A *Container registry* stores the images that Kubernetes relies on (Red Hat, n.d.).

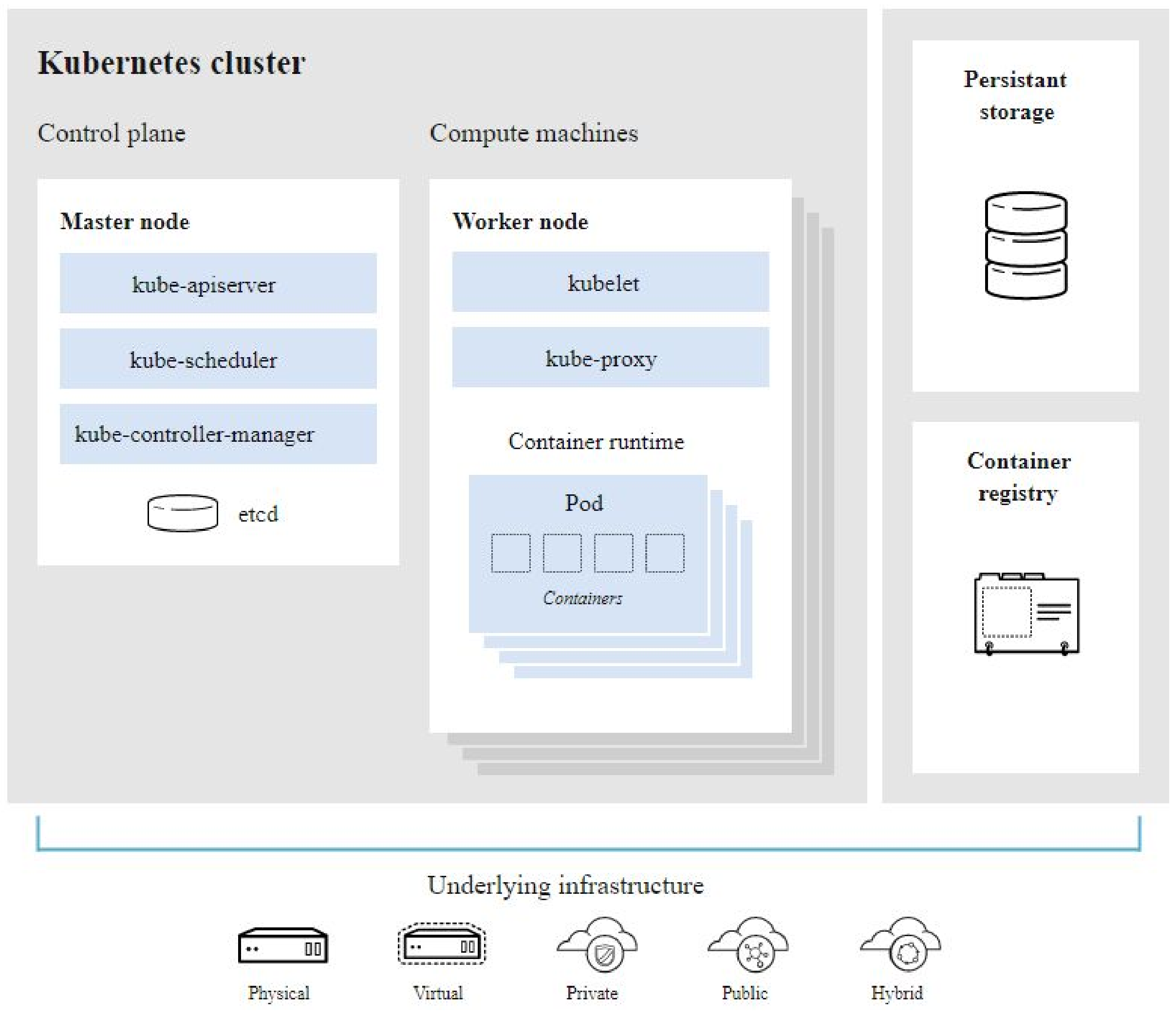


Figure 4: Kubernetes Cluster (Red Hat, n.d.)

One of Kubernetes key benefits is that it could be implemented on diverse types of infrastructure e.g. virtual machines, public cloud providers, private clouds, and hybrid cloud environments (Red Hat, n.d.).

1.3.7 Messaging in a Distributed System

In a distributed system, messaging plays a pivotal role. Data flows from one system to another through messages. Different protocols and formats are used to share data within a distributed system as messages or events. Another key aspect of messaging is the nature of communication. Mainly, there are 2 styles of messaging.

- Synchronous (request-response) messaging (Fernando, 2019).

- Asynchronous (pub-sub, competing consumer, event processing, batch processing) messaging (Fernando, 2019).

Synchronous messaging means that the system which is sending the message expects an immediate response from the target system as indicated in the figure below. Request-Response style of messaging is another name for the same (Fernando, 2019).

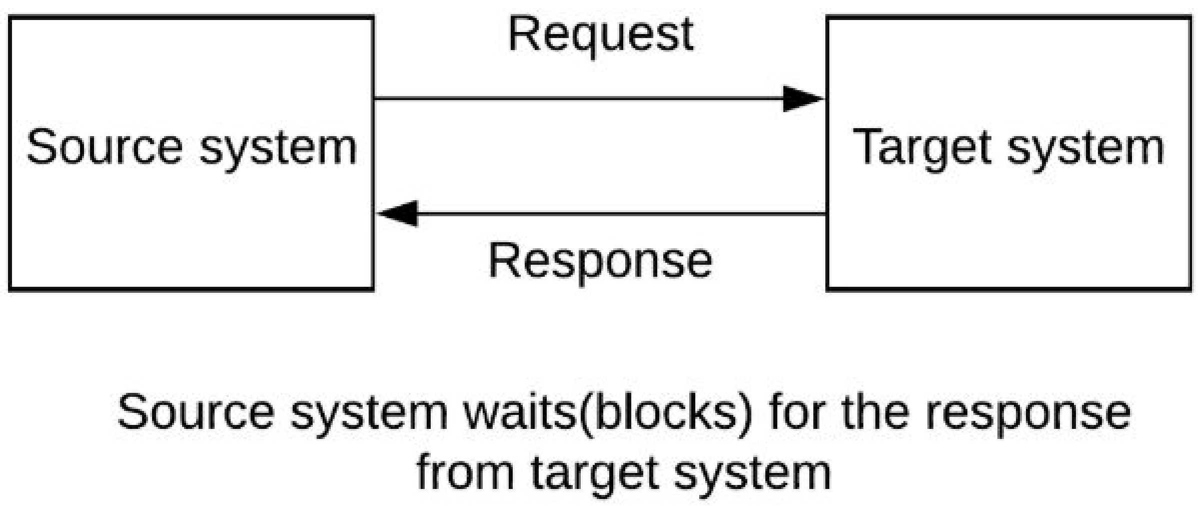

Request
Source system
Target system
Response
Source system waits(blocks) for the response from target system

Figure 5: Synchronous Messaging (Fernando, 2019).

Asynchronous messaging are in some cases more useful as there cases when data flows through systems at really high rates and it is complicated to respond synchronously. The message sources, on the other hand, do not expect an immediate response or sometimes does not expect a response at all (Fernando, 2019).
However, there are primary application server elements required for asynchronous messaging; publication broker, publication contractor, and subscription contractor services as presented in the figure below. The publication broker service routes the workload to both contractor server processes.
The publication contractor updates the publication contract with the status of subscription processing (Done or Retry) when the publication broker service has performed an HTTP post of the publication service operation to the integration gateway. Finally, the subscription contractor runs the appropriate notification to

update the subscription contract concerning the status of the subscription processing (Oracle, n.d.).

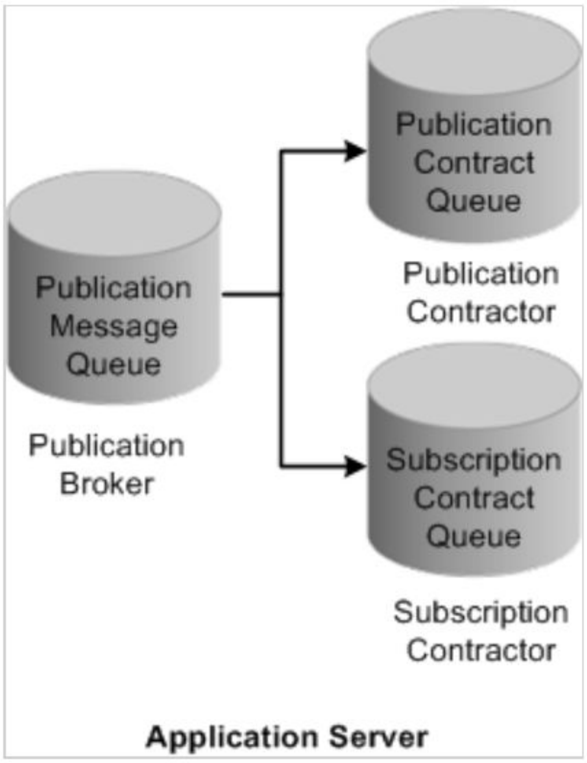


Figure 6: Brokers, Contractors, and Queues (Oracle, n.d.)

1.3.8 Monolithic Architecture and Microservice Architecture

A monolithic application is when all the functionalities of a project exist in a single codebase. The application is designed in various layers, e.g.presentation, service, and persistence and subsequently, the codebase of the application is deployed as single jar/war file. The term “mono” represents the single codebase containing all the required functionalities (Bhadauria, Raman. n.d.).

Microservices are specified as small autonomous services built around the seven principles below: model [services] around business concepts, adopt a culture of automation, hide internal implementation details, decentralize all things, isolate failure, and make services independently deployable and highly observable (Yarygina, 2018).
Furthermore, microservices are defined as a specialized variation of service-oriented architecture that emphasizes the fine-grained separation of concerns, continuous

delivery, and virtualization (Yarygina, 2018).

Microservices depend on the principles below to enable a microservice architecture.

- Separation of concerns (Yarygina, 2018).
- Continuous delivery and DevOps (Yarygina, 2018).
- Continuous integration (Yarygina, 2018).
- Cloud, virtualization, and containerization (Yarygina, 2018).

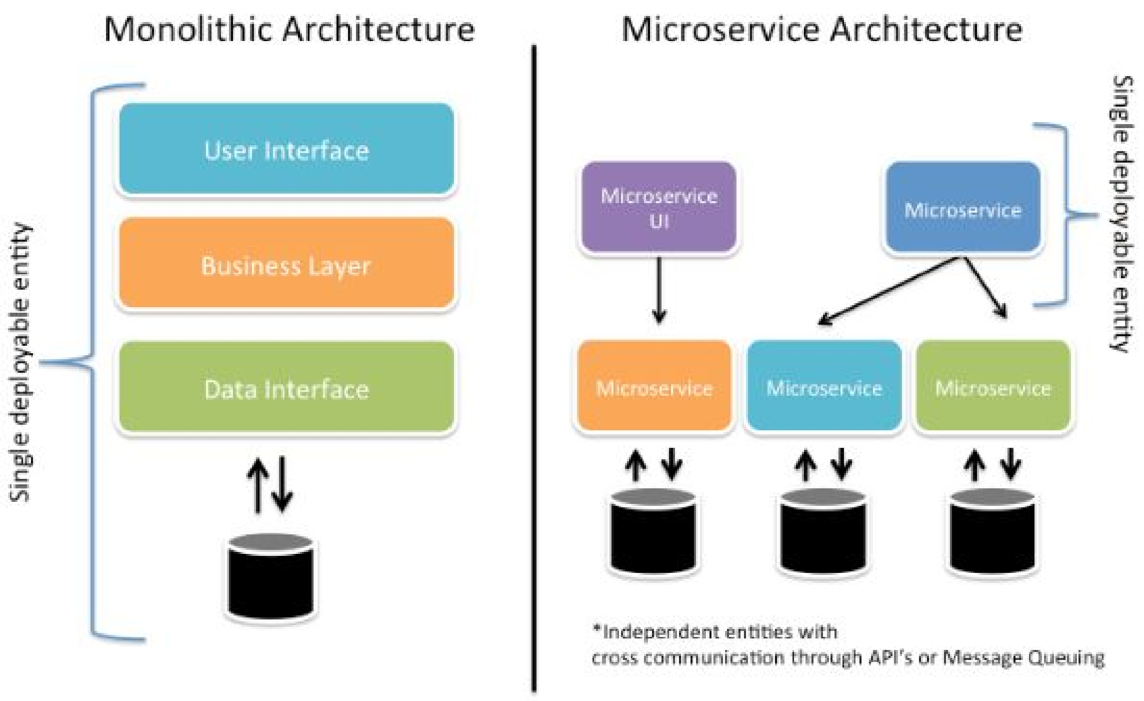


Figure 7: Monolithic Architecture vs Microservice Architecture (Management & Solutions, n.d.)

1.3.9 Red Hat OpenShift Container Platform

OpenShift is a family of containerization software developed by Red Hat. Its main product is the OpenShift Container Platform—an on-premises platform as a service built around Docker containers orchestrated and managed by Kubernetes on a foundation of Red Hat Enterprise Linux.

The Openshift User Interface, UI has various functionalities, allowing one to monitor the container resources, container health, the nodes the containers reside on, IP addresses of the nodes, etc. The key store can be accessed via the Secrets in Openshift (OpenShift, 2020).

1.3.10 REST-API

A RESTful API is an application program interface, API that uses HTTP requests to GET, PUT, POST, and DELETE data.
An API for a website is a code that allows two software programs to communicate with each other. A RESTful API, also referred to as a RESTful web service or REST API is based on representational state transfer (REST), an architectural style and approach to communications often used in web services development.

The REST used by browsers can be thought of as the language of the internet. In a cloud context, APIs are being used by cloud consumers to expose and organize access to web services. REST is a logical choice for building APIs that allow users to connect to, manage, and interact with cloud services flexibly in a distributed environment. RESTful APIs are used by such sites as Amazon, Google, LinkedIn, and Twitter. As the calls are stateless, REST is useful in cloud applications. Stateless components can be freely redeployed if something fails, and they can scale to accommodate load changes. This is because any request can be directed to any instance of a component; there can be nothing saved that has to be remembered by the next transaction. That makes REST preferable for web use, but the RESTful model is also helpful in cloud services because binding to a service through an API is a matter of controlling how the URL is decoded. Cloud computing and microservices are assumed to make RESTful API design the rule in the future (Rouse, 2020, April).

1.3.11 Virtualization Technology and Containerization

The literature concludes that containerization and virtualization are key-enabling technologies of a microservice architecture (Ardagna et al., 2015, Yarygina, 2018),

Docker containers are common as an option for microservice deployment nowadays (Yarygina, 2018).
However, it is possible to combine both containerization and virtualization when running microservices. As shown in figure 3. it is possible to run microservices on top of (i) a physical machine running an operating system, (ii) a machine running a container engine, (iii) a machine running a virtualized environment (in this setting the hypervisor is mapped as the operating system), or (iv) a machine running a container engine on top of a virtualized environment (Di Francesco et al., 2019).

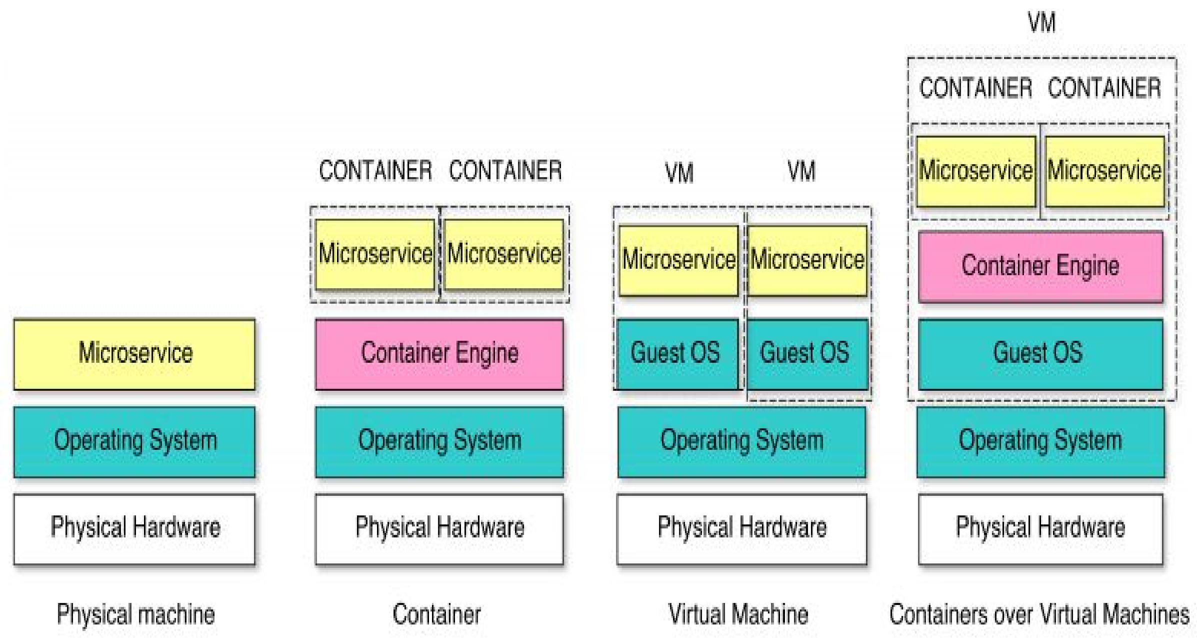


Figure 8: Abstraction Layers (Di Francesco et al., 2019)

2.0 Research Design

The objective of the research is about explaining how Kubernetes may solve availability concerns in microservice architecture for critical systems. Hence, qualitative methodology is a proper choice as the focus of this report is about explaining how Kubernetes security configurations may support availability for a critical system deployed as a microservice architecture.
A quantitative method aims at testing theories by examining the relationships among variables. In quantitative studies, collected data includes statistics which in turn is analyzed to generalize and replicate findings. (Creswell, 2008). A quantitative method would not offer a proper toolbox considering the objective of the research.

2.1 Case Study as a Method

The findings of this study were explanatory and descriptive to fulfil the objective of research and a quantitative methodology would therefore not offer the proper toolbox. A case study is a relevant option and Skatteverket, the Swedish Tax Agency is selected as a single case. Skatteverket is interesting in this context because of its essential function in Swedish society. Skatteverket is the core organization to ensure funding of the public sector and it is a relevant option considering the objective of this study; (how Kubernetes' security configurations may enhance availability in a microservice architecture for critical systems).
In the appropriation directions, it's declared that Skatteverket shall contribute to assuring the funding of the public sector and to strengthen a well-functioning society while preventing crime (Regeringsbeslut Fi2018/02509/RS (delvis)).
Thus, Skatteverket has critical systems deployed as microservices and they must be available at all times. If the availability of these systems is severely affected due to a crash or failure it would have a profound impact on the Swedish society and economy.

To properly describe Skatteverket as a case some questions were sent by e-mail to Mr. Claudio Meneses Marshal, a consultant at Skatteverket/a system architect (Appendix F).
In the result section, a case description of Skatteverket is provided.

A common criticism of the case study is its dependency on a single case exploration, making it difficult to reach a generalizing conclusion.
The results of this paper could be generalized as availability issues are encountered in various contexts as mentioned in the literature (Vayghan, Saied, Toeroe, Khendek, 2019., Smith, Trivedi, Tomek, Ackaret, 2008., Brendan, Lefebvre, Meyer, Feeley, Hutchinson, Warfield, 2008., Machida, Andrade, Kim Seong, Trivedi, 2011., Machida, Kim Seong, Trivedi, 2009., Taherizadeh, Grobelnik, 2020) and therefore the results of this paper could be applied to other critical systems with high-availability requirements. However, generalizations must always be modestly and carefully considered and applied as this report is a single-case-study.
Furthermore, case studies are often accused of a lack of rigour (Zainal, 2007). Yin (2009) notes that:
"too many times, the case study investigator has been sloppy, and has allowed equivocal evidence or biased views to influence the direction of the findings and conclusions" (Yin, 2009).
To address this concern the conclusions and collection of data have been guided by a theoretical framework (table 2). The theories applied in this study has also limited the production of documentation to select relevant data.

2.2 Collection of Data

To approach the problem a questionnaire was sent to the systems architect. It provided important insights on the purpose of metrics. Metrics indicate when failure arises and in turn, failure has an impact on the availability of the microservice architecture for critical systems (Appendix A).
The benefit of sending a questionnaire by email is that the respondent is able to send a reply at his/her convenience in written form. Although, one limitation is that it

is not possible to pose additional questions in real-time, which is time-consuming. This shortcoming did not have an impact on the conclusion and the time limit has been taken into account when planning.

Two meetings were performed to understand the microservice architecture at Skatteverket and to capture Skatteverket's perspective on Kubernetes. Both sessions were recorded to ensure the accuracy of the received information. In the first session on the 10th March 2020, Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect explained on high-level failures that might arise in a microservice architecture on the infrastructure level at Skatteverket and which metrics are applied to discover these diverse failures (Appendix B).
The purpose of the second session on the 16th March 2020 with Mr. Sadmir Halilovic, solutions architect at Skatteverket was to target the general features of Kubernetes and its benefits and limitations (Appendix C).
The meetings provided a foundation to specify the objective and questions of research. One disadvantage, being aware of is the subjectivity of knowledge. Consequently, knowledge is based on prior understanding and our interpretation of the real world (Marsh, 2002). However, the objective of this paper does not assume that there is a reality independent of our assumptions and interpretations. The objective is based upon the premise that there is a problem within a specific context, which may be solved by applying a particular technology. The meetings offered the foundation to set an objective and questions of this paper.

Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, Ms. Ülker Kayhan, a supervisor (main expertise is testing) and Mr. Sadmir Halilovic, a solutions architect were selected as interviewees due to their expertise within software testing, microservice architecture, and Kubernetes. Their experience was interesting as it provided vital data to fulfil the objective of this paper; studying how Kubernetes may enhance availability in microservice architecture for critical systems.

Data was collected by performing one telephone interview with Mr. Claudio Meneses Marshall (Appendix D). The interview aimed to create an understanding of

the diverse types of failure that has an impact on availability within a microservice architecture for critical systems. Also, Mr. Claudio Meneses Marshall and Ms. Ülker Kayhan were asked to set levels of severity to each type of failure; (severe, moderate, and minor) as a manner of indicating the impact of failure on operations at Skatteverket. The questions were semi-structured as the theory on types of failure provided a structure and therefore semi-structured questions supported the objective of this report. Also, semi-structured questions would provide comparable replies to analyze (Lantz, 1993).
The interview was recorded to ensure the correctness of the responses. The questions were sent by email before the interview to give the respondent time to prepare. Also, the responses were summarized and sent to the interviewee to avoid erroneous information. This is a manner of securing reliability of responses, which means the responses reflect the interviewee's perspective and therefore the responses are possible to analyze by applying the theoretical framework (Lantz, 1993).

Also, two questionnaires were sent by email to Ms. Ülker Kayhan and Mr. Sadmir Halilovic (Appendix D). The questionnaires included the same questions as the first interview with Mr. Claudio Meneses Marshall. It is beneficial to send questionnaires by email as the respondents can reply at their convenience. However, by adopting such a method for collecting data there is no possibility to directly pose additional questions when responses are ambiguous. This concern is addressed by sending additional questions to bring clarity to the responses. Also, the responses were summarized and sent by email to the interviewees to confirm reliability.
Moreover, a meeting was arranged to describe the case of Skatteverket to set Kubernetes and microservice architecture into the context of the case (Appendix F). The case description is outlined in the section of the result.

Another aspect to consider when performing research is the validity of conclusions. Yin discusses internal validity and it refers to the credibility of the arguments made. Yin argues that the boundary of the study should be defined in such a way that it can be seen that information beyond the boundary is of decreasing relevance. Also, the

reader should be convinced that very little relevant evidence remains untouched (Yin, 2009).
The conclusions in this paper are valid as they are guided by the theoretical framework. Thus, the collected data present an understanding of types of failure affecting availability in a microservice architecture for critical systems. Subsequently, Kubernetes' security actions were studied to indicate whether or not they may preclude failure.

2.3 Analysis of Data

The collected data was analyzed by connecting the failures (crash, omission, value, timing and byzantine) affecting availability in a microservice architecture for critical systems to Kubernetes' security configurations (securing the cluster, managing authorization and authentication, implementing a trusted software supply chain, securing workloads in runtime and managing secrets). This was performed to achieve the objective of this paper.

In the result and analysis section, the data is presented and analyzed by presenting the types of failure at Skatteverket. In the next step, Kubernetes' security actions are analyzed to study whether or not they might solve the various failures. Finally, the analysis is summarized in table 4. A failure might be fully, partially, or not solved at all by a specific Kubernetes' security action. Also, the respondents at Skatteverket were asked to set a level of severity when failure was partially or not solved at all by Kubernetes' security actions.

2.4 Ethics

In regards to ethical concerns, no confidential information has been presented and recordings of interviews and meetings were stored locally on the devices (PC and smartphone). Furthermore, the respondents at Skatteverket have continuously the contents of this report.

2.5 Source Evaluation

The sources used for this study has been primarily written by researchers within cloud computing and cloud security to capture the complexity of the field in research terms and to avoid a profound bias. Keywords as cloud computing, monitoring, microservices, microservices architecture, availability, Kubernetes, critical systems, cloud security, and IT-security were searched for. Also, articles from ResearchGate, have been used to find additional information on the topic but also on methodology. Databases such as EBSCOHOST, ScienceDirect, and Google Scholar have been useful sources.

The articles were chosen because of their relevance while having the objective in mind. Few articles have qualitatively examined in-depth the failures affecting availability in a microservices architecture, which makes the categories of failure too wide to comprehend in a tangible manner. However, the wideness of each category offers flexibility to implement in several contexts and is therefore fruitful to apply as a theoretical framework.

Also, some articles from professionals within the cloud computing industry were used to determine the concepts in the section of definitions. One limitation of such sources is its simplistic view on a specific concept. However, the concepts are not part of the theoretical framework and, therefore, should not have an impact on the conclusion of this report. Also, the concepts were selected based on the boundary of the objective. In other words, a wide definition of a certain concept would not provide an appropriate context for the objective.

3.0 Theoretical Framework

This section defines a critical system and explains the types of failures that might arise in a distributed system, which is a microservice architecture in this paper. Moreover, Kubernetes' security actions are described.

3.1 What is a Critical System?

In this context, a critical system, physical or cyber is to a Nation so vital that their incapacity or destruction would have a debilitating impact on national economic security, and/or public health or safety (Aven, 2009).

3.2 Types of Failure affecting Availability

Let us consider a distributed system model in which multiple hosts with no shared memory are connected by a network that provides an unreliable unordered communication mechanism with no upper bound on worst-case message transmission time. Hosts can only interact by sending and receiving messages through this communication network. Also, let us assume that the communication network does not experience permanent failures and that partitions are not considered.

Given this context, a faulty host can only affect other hosts by not sending a message when it should, by sending a message when it should not, or by sending an incorrect message. An erroneous host may exhibit the incorrect behaviour only to a subset of the other hosts.

These kinds of incorrect behaviours are mapped to failure model definitions, including crash, omission, timing, value, and Byzantine (Hiltunen., Immanuel., Schlichting, 1999).

3.2.1 Crash

A host may permanently stop and fail to send messages. If a host is in the process of sending a message when it crashes, some of the intended receivers may not receive

the message (Hiltunen., Immanuel., Schlichting, 1999).
A host may also stop responding to any request in this scenario (Balazinska., Hwang., Shah, 2017).

3.2.2 Omission

A host may repeatedly and irregularly fail to send a message to all or some of the intended receivers, or fail to receive messages (Hiltunen., Immanuel., Schlichting, 1999).

3.2.3 Timing

A host may send a message earlier or later than expected. The network delaying a message longer than expected may result in a host appearing to have a late timing failure (Hiltunen., Immanuel., Schlichting, 1999).

3.2.4 Value

A host may send a message with incorrect content. Let us assume, that if a value faulty host sends a multicast message, all receivers will receive the same message inaccurate contents (Hiltunen., Immanuel., Schlichting, 1999).

3.2.5 Byzantine

A host may do anything. This means that in addition to value and timing failures, a faulty host may deliberately attempt to confuse other hosts by sending different versions of a message to different receivers or by impersonating another host (Hiltunen., Immanuel., Schlichting, 1999).

3.3 Kubernetes, K8s and Security

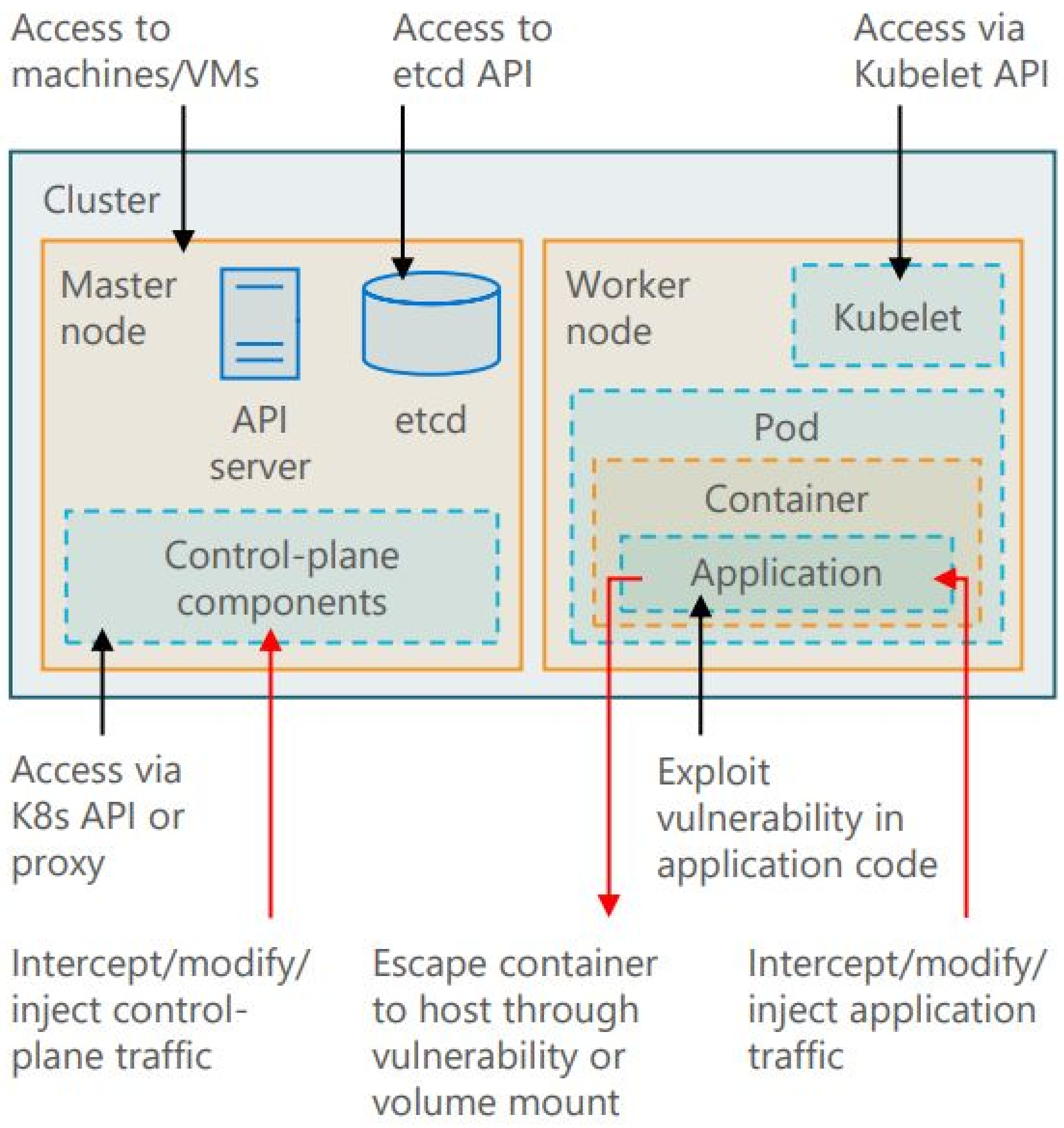
To implement Kubernetes professionals could use Kubernetes API objects to describe a cluster's desired state: what applications or other workloads should run, what container images are used by the cluster, the number of replicas, what network and disk resources should be available, etc. The desired state is defined by creating objects using the Kubernetes API, e.g. via the command-line interface, kubectl. It is also possible to utilize Kubernetes API directly to interact with the cluster and set or modify your desired state (Kubernetes, n.d.).

Kubernetes is presented to offer several gains:

- Microservices by breaking an application into smaller, manageable, scalable components that could be used by groups with different requirements (Vohra, 2016).
- Fault-tolerant cluster in which if a single Pod replica fails (due to node failure, for example), another is started automatically (Vohra, 2016).
- Horizontal scaling in which additional or fewer replicas of a Pod could be run by just modifying the “replicas” setting in the Replication Controller or using the replicas parameter in the kubectl scale command (Vohra, 2016).
- Higher resource utilization and efficiency (Vohra, 2016).
- Separation of concerns. The Service development team does not need to interface with the cluster infrastructure team (Vohra, 2016).

Although, security issues have expanded while the adoption of Kubernetes increased.
Rice and Burns wrote security guidelines to approach and resolve the security concerns when utilizing Kubernetes to deploy and run applications on a large scale (Rice & Burns, 2019).
The figure below presents the diverse points of vulnerabilities and therefore, the authors bring up five key actions to enhance security:

1. Securing the cluster (Rice & Burns, 2019).
2. Managing authorization and authentication (Rice & Burns, 2019).
3. Implementing a trusted software supply chain (Rice & Burns, 2019).
4. Securing workloads in runtime (Rice & Burns, 2019).
5. Managing secrets (Rice & Burns, 2019).



Access to machines/VMs
Access to etcd API
Access via Kubelet API
Cluster
Master node
API server
etcd
Control-plane components
Worker node
Kubelet
Pod
Container
Application
Access via K8s API or proxy
Exploit vulnerability in application code
Intercept/modify/ inject control-plane traffic
Escape container to host through vulnerability or volume mount
Intercept/modify/ inject application traffic

Figure 9: Kubernetes Architecture (Rice & Burns, 2019).

3.3.1 Securing the Cluster

The Master Node controls the configuration and operation of the entire cluster and is, therefore, a key area to secure.

The API server offers REST API access to control the cluster. The kubeadm installer disables the API server's insecure port so that API access is restricted to encrypted TLS connections made over a secured port by default, but is not limited by default to authenticated users Internet (Rice & Burns, 2019). To further limit access to the API server, it is possible to:

- Prevent unauthenticated users from accessing it. This means that all API server access, including health checks and service discovery, must be authenticated (Rice & Burns, 2019).
- Permit unauthenticated user access but limit it using role-based access control (RBAC). By default, the RBAC setting permits very limited access to anonymous users, so that a client can make health checks and service discovery can be performed without providing certificates. This is the default approach, but it does rely on you maintaining sensible RBAC policies that restrict what anonymous users can do (Rice & Burns, 2019).
- Further protect API server access with additional measures, such as a traditional firewall or VPN (Rice & Burns, 2019).
- If the Kubernetes Dashboard is installed then it should be used to connect to the API server while restricting restricted access to avoid exposure on the Internet (Rice & Burns, 2019).

Kubernetes stores cluster configuration and state information in a distributed key-value store named etcd. Unauthorized access to etcd may threaten the entire cluster, which is why access to it should be strictly limited (Rice & Burns, 2019).

The kubelet is an agent that runs on each worker node and interacts with the container runtime to launch pods and report node and pod status. Unauthorized access to a kubelet can allow starting and stopping pods, as well as executing unauthorized code (Rice & Burns, 2019).

Also, the Center for Internet Security (CIS) publishes benchmark lists with more than 100 recommended configurations. The open-source kube bench can automate the checking task and provide pass and fail results on all the benchmark tests (Rice & Burns, 2019).

3.3.2 Managing Authorization and Authentication

Kubernetes is a distributed system, therefore, it is vital to use authentication for multiple components (not just the API server) to prevent unwanted users or service accounts from accessing cluster components and data (such as kubelets, kube proxies, and secrets). Also, authorization controls could be used to prevent authenticated users from having blanket access to unneeded capabilities (Rice & Burns, 2019).

Implementing authentication models is one method to authenticate access. Also Role-Based Access Control, RBAC supports fine-grained access over authorization and access to manage authorization on multiple levels (Wallen, 2020, March 17). Kubernetes RBAC model uses several objects to govern resource authorization.

- Entity: A user, group, or service account (Rice & Burns, 2019).
- Resource: Something the entity will access, like a pod, secret, or service (Rice & Burns, 2019).
- Role: Used to define rules specifying a set of actions that are permitted on a set of resources (Rice & Burns, 2019).
- RoleBinding: Attaches a role to an entity, defining the actions the entity can perform on resources (Rice & Burns, 2019).

3.3.3 Implementing a Trusted Software Supply Chain

Kubernetes is utilized to run the software in the form of containers, which means that the contents and interaction of containers have an impact on the security of Kubernetes applications. It is vital to implement controls across the pipeline to ensure that what goes in is validated, and that code integrity is maintained in all phases and levels. This could be achieved by:

- Implementing source control (Rice & Burns, 2019).
- Image scanning (Rice & Burns, 2019).
- Avoiding the use of root user (Rice & Burns, 2019).
- Applying image integrity controls (Rice & Burns, 2019).
- Enforcing the use of trusted images (Rice & Burns, 2019).
- Securing the registry (Rice & Burns, 2019).

3.3.4 Securing Workloads in Runtime

It is also important to place boundaries and controls to limit an application's authorization and thus what it can do in runtime. This configuration may limit the damage of an attack or prevent an intruder from getting past their initial intrusion point. Kubernetes has several native policies that, may bolster a secure environment. This could be accomplished by:

- Setting a security context to define privileges and access control at the pod or container level (Rice & Burns, 2019).
- Setting a pod security policy to secure the cluster context (Rice & Burns, 2019).
- Setting a network policy place guardrails on pod network traffic to prevent unwanted traffic between nodes on a cluster and traffic between the pod and other layers or external resources (Rice & Burns, 2019).

3.3.5 Managing Secrets

Secrets, such as private keys or passwords, are often needed for a container to access services or data. The challenge is ensuring that the secret is accessible only from the intended container. Also, it is possible to expose credentials to specific team members. This could be attained by:

- Avoiding some practices as hard-coding or embedding secrets in images and using unencrypted environment variables (Rice & Burns, 2019).
- Storing secrets in Etcd and third party vaults (Rice & Burns, 2019).
- Passing secrets to containers using two methods; environment variables and volume mount (Rice & Burns, 2019).

3.4 Summary

The theories above are applied by studying the types of failure (crash, omission, value, timing and byzantine) in the case, Skatteverket to analyze whether or not each separate Kubernetes' security action (securing the cluster, managing authorization and authentication, implementing a trusted software supply chain, securing workloads in runtime and managing secrets) may mitigate the failure. The ability of Kubernetes to handle failure is indicated by the categories; fully solved by Kubernetes, partially solved by Kubernetes or not solved by Kubernetes. The matrix below illustrates the application of the theoretical framework, which will be presented in the section of analysis. Furthermore, the respondents at Skatteverket have graded the severity of unsolved failures by applying levels of severity; severe (red), moderate (yellow), and minor (green).

Fully solved by Kubernetes = ●

Partially solved by Kubernetes = ◑

Not solved by Kubernetes = ○

Levels of severity

- *Severe* is a "critical problem" (Red) — the product is unusable or an error severely impacts an End-User's operation, and there are no workarounds to restore product functionality. A severity level of severe requires maximum effort to resolve a critical problem (Lawinsider, n.d.).
- *Moderate* is a "major problem" (Yellow) — significant product functionality is not working according to product definitions, or significant business objectives cannot be met (Lawinsider, n.d.).
- *Minor* (Green) — minor product functionality is not working according to product definitions, or minor business objectives cannot be met (Lawinsider, n.d.).

THE TYPES OF FAILURE / **KUBERNETES' SECURITY ACTIONS**	CRASH	OMISSION	TIMING	VALUE	BYZANTINE
SECURING THE CLUSTER					
MANAGING AUTHORIZATION AND AUTHENTICATION					
IMPLEMENTING A TRUSTED SOFTWARE SUPPLY CHAIN					
SECURING WORKLOADS IN RUNTIME					
MANAGING SECRETS					

Table 2: The Application of the Theoretical Framework

4.0 Result: The Case Skatteverket

Firstly, the case Skatteverket is described to set the respondents' replies into context. The case description is about explaining why availability is such a vital security aspect for Skatteverket's critical systems. Also, the case description includes Mr. Claudio Meneses Marshall's replies on Kubernetes and microservice architecture at Skatteverket.
Secondly, the responses of Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, Ms. Ülker Kayhan, a supervisor, and Mr. Sadmir Halilovic, a solutions architect are summarized to explain the categories of failure affecting availability.

4.1 Skatteverket - Case Description

Mr. Claudio Meneses Marshall emphasized that most business processes at Skatteverket are vital to continuously enable and maintain a functioning society. Consequently, Skatteverket handles all matters of taxes and population. The Swedish government would not have any income if there were no information on taxes, population, and companies (Appendix F).
In the appropriation directions, it's declared that Skatteverket shall contribute to assuring the funding of the public sector and to strengthen a well-functioning society while preventing crime (Regeringsbeslut Fi2019/04080/S3 (delvis)).
The technical platforms and architectural patterns including microservices are utilized to support all business processes at Skatteverket (Appendix F). Also, it signifies that all IT-systems bolster the achievement of business goals declared in the appropriation directions. This implies that all IT-services are crucial in Skatteverket's business and therefore, the high-availability of critical systems is indispensable. If these critical systems fail then it would seriously disrupt society and it would have an impact on the national economic security and/or public health or safety. The high-availability feature of a critical system is crucial in this context (Appendix F).

The system architect and consultant at Skatteverket, Mr. Claudio Meneses Marshall explained that Kubernetes and Red Hat OpenShift Container Platform, OCP are implemented at Skatteverket. This implementation is a strategic decision to shift to “The Next Generation” of architecture; from a monolithic architecture to a microservice architecture. A microservice architecture implies a container-based platform where Kubernetes is utilized. Other Swedish authorities will most likely follow Skatteverket and perform the same transition to a microservice architecture (Appendix F).

Furthermore, Mr. Claudio Meneses Marshall indicated that Skatteverket has critical systems for those business areas that are declared in the appropriation directions Regeringsbeslut Fi2019/04080/S3 (delvis). The business areas imply that Skatteverket must provide and manage information on taxes, population registration and estate inventories for both individuals, businesses, and employers.
For instance, the services of “Navet” are consumed by quite many Swedish organizations within the public and private sectors. Navet has all information on taxes. Another example is “Kommunavräkning”, which is one of the most critical systems according to the system architect. Kommunavräkning manages and transfers taxes to the Swedish municipalities (Skatteverket, n.d.).
Mr. Claudio Meneses Marshall indicated that procurement and analysis of product and technology options were performed prior to the implementation of microservices and Kubernetes three to four years ago.
Moreover, specified that currently, Kubernetes is utilized as a part of the Red Hat OpenShift Container Platform, OCP. All future applications will be developed and implemented by utilizing Kubernetes. Also, Kubernetes is applied as technology during system testing and, acceptance testing and in production. This implies that Kubernetes is defined by Skatteverket as the vital and sole platform for the deployment of new applications. Monolithic systems as Navet will be part of the legacy systems and are, therefore, in the the-end-of-life phase of the system development life cycle. Also, it signifies that the future development environment will be distributed, which will enable developers to locally developed and perform system

tests on their PCs. System testing includes verification of the flow of request calls and it would be a complete implementation of the Red Hat OpenShift Container Platform.
One application that has been recently developed is TAIS, which provides services to all Swedish citizens and countries within the EU. TAIS is developed in the OCP-based environment and is, therefore, an application within a microservice architecture. The application, TAIS consumes information services by communicating through the integration layer with Skatteverket’s critical and monolithic systems, e.g. NAVET (Appendix F).

4.2 Failures affecting Availability of Microservice Architecture at Skatteverket

Mr. Claudio Meneses Marshall was asked to describe some failures that affect the availability of hosts when interacting. The interviewee replied that hosts communicate not only by sending messages but also through integrations, e.g. REST-APIs or queue management systems. This means that a failure could appear due to an issue on the integration layer or on the DB-layer, which does not involve network errors as indicated in the quote (Appendix E - Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, personal communication, 2020, March 31).
Another error that might arise is inaccurate management of information. The host is unable to manage information accurately or it could have lost authorization keys. The authorization keys could have got corrupted due to an update of a recipient host. Thus, the authorization keys are no longer available. The authorization keys are essential when communicating with other external hosts. As a result of failing to update the tables of hosts, a queue on the integration layer emerges to manage the failure.

Ms. Ülker Kayhan indicated that from a testing perspective, frequent failure is the downtime of services. Otherwise, the services “skatteberäkning-fysisk” and “skatteberäkning-juridisk” are designed to function properly as calls are sent to their wrappers. Wrappers encompass a library of several functions that enhance response delivery when the services “skatteberäkning-fysisk” and “skatteberäkning-juridisk” receive a call.

When reflecting in general terms of various failures the Mr. Sadmir Halilovic accentuated that a vast period of unsynchronized data is not beneficial for data consistency. In other words, hosts have not sent messages on updated data and therefore data is not consistent. However, it is possible to apply patterns to mitigate failure, but it is a more complex solution.
Hosts sending messages to other hosts when it should not is a failure causing availability concerns, which require excessive SLA-conditions. Another solution is designing asynchronous communication on the integration layer, which could be achieved by a broker. The communication between the services provided by the broker is synchronous but not necessarily simultaneously.
When synchronous communication between services, hosts, or pods on the integration layer is not functioning as intended the unavailability cause data loss. Data loss could be avoided by implementing patterns (retry or outbox) in producing services.
When a host sends an incorrect message to another host the failure indicates that the producing service (host) has a bug, which also has a negative impact on other hosts. This kind of error could have severe consequences on business if not solved immediately. In this scenario, the solutions architect emphasized that it is beneficial to break down the system into separate microservices to isolate failure and thereby avoid failure of other microservices as implied in the quote (Appendix E - Mr. Sadmir Halilovic, solutions architect, personal communication, 2020, April 10). Moreover, the microservices would have a shorter time to market as the scope of the regression test is limited.

4.2.1 Crash

Mr. Claudio Meneses Marshall explained that the task of a host is to send messages to other hosts or to make certain information available.

For instance, consider a fictional scenario with a Walt Disney domain. Mickey Mouse is assigned to publish an API with updates on cartoon series. This information is consumed by an external service or a consumer, but the information is not received by the recipient. The following checkpoints should be performed to handle a crash; firstly always check the functionality of the network and secondly, check the API-gateway on the integration layer (Appendix E - Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, personal communication, 2020, March 31).

A crash in this context means that a host waiting for a response is not receiving the requested information. The host receives an error message.
In sum, there are three scenarios when a crash occurs. Firstly, a host that is assigned to get certain information is unable to authenticate accurately. Secondly, a host is not getting contact with the API-Gateway as it has downtime or the API gateway is missing. Thirdly, API gateway might send null as a response, meaning that the error arises between the API gateway and the REST-API - publishing host.

Ms. Ülker Kayhan discussed crash failure from the testing perspective, crash failure might also arise on Skatteverket's Openshift Container Platform, OCP. In this scenario, the hosts hosting the services recover by restarting automatically. The technical tester replied that crash failure rarely occurs, but when hosts do crash it due to lack of RAM. In such a scenario the crash is due to the services' Java-based processes have consumed all RAM. Further, Ms. Ülker Kayhan stressed that failure arises due to human error as explained in the quote (Appendix E - Ms. Ülker Kayhan, supervisor, personal communication, 2020, April 7).

Mr. Sadmir Halilovic confirmed that a crash has occurred and resultantly the business transactions are interrupted. This type of failure could be smoothly handled or not at all depending on the policy between the provider (container platform) and consumer (containerized service/application). The provider's task (container platform) is to send back a response to the service's/application's request. The consumer (containerized service/application) is assigned to determine the desired state of its service. Having this in mind, the platform’s configuration could for some reason result in a transfer of one or multiple pods from one worker node to another worker node within the same cluster. This event could generate unintended crash of a specific instance of a service, but if the pods/applications are properly designed they would be able to handle a crash. In this scenario, the services would continue its process either by creating a new instance of the service or by transferring the pods to a different worker node. This is also known as the self-healing or resilient capabilities of a pod. Services can handle such exceptions if patterns e.g. as outbox are implemented when designing the services. Another option when handling failures is to create services that manage the orchestration of workflows.

4.2.2 Omission

Mr. Meneses Marshall considered a scenario, where hosts within a container had shared resources e.g. memory, storage, CPU, etc, but all of a sudden failure of a host arise. A new host replaces the faulty one. This system behaviour seems to have no obvious reason, but it appears as the erroneous host is overloaded or unresponsive. A new host replaces the inaccurate one. There are several reasons for this irregular system behaviour; the host has defective code, the host is poorly designed or it is erroneously configured.
Omission failure may occur when a certain call is performed or a certain delivery is requested. To set an example we could refer to the Walt Disney scenario. Let us assume that there is a host that delivers updates on new cartoons and the host appears to function appropriately until there are updates on Pluto cartoons. The host is unable to publish cartoons on Pluto, even though updates on Mickey Mouse cartoons are published without any failures. Irregular system behaviour as this one implies a system exception and it is important to understand when the failure occurs. The question of when a failure emerges is about identifying the event generating the failure. Also, Mr. Meneses Marshall accentuated that a consequence analysis must include a review of both code and architecture. Moreover, this scenario is the worst case as expressed in the quote (Appendix E - Mr Claudio Meneses Marshall, a consultant at Skatteverket/system architect, personal communication, 2020, March 31).

Omission failure signifies that failure isolation is vital and therefore a host is assigned one function. Hosts within a microservice architecture are built with few dependencies to trace and isolate failures. Finally, the systems architect concluded that is the most complex scenario and it emerges daily as specified in the quote (Appendix E - Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, personal communication, 2020, March 31).

According to Ms. Ülker Kayhan omission failure from a testing point of view emerges when hosts have inaccurate parameters (e.g. date of birth) to complete their service task. This failure is corrected by assuring that the correct input parameters are sent to the services hosting a particular function.

Mr. Sadmir Halilovic confirmed that omission has emerged and subsequently, transactions of a business are interrupted. These interruptions could be poorly or adequately handled depending on the elements of the contract between the provider and the customer. Furthermore, omission failure of a service is probably a result of not applying patterns to mitigate failures as explained in the quote (Appendix E - Mr. Sadmir Halilovic, a solutions architect, personal communication, 2020, April 10). If patterns are applied failures are mitigated without the interruption of business transactions.

4.2.3 Timing

Mr. Meneses Marshall replied that timing failures do appear as indicated in the quote (Appendix E - Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, personal communication, 2020, March 31).
A host sends a message or it replies to a get-request. In this context, it is important to determine whether the communication between hosts should be synchronous or asynchronous. Also, it is vital to understand what communication mechanisms should be utilized, e.g. update, post. The business dependency of up-to-date information when operating specific communication mechanisms and also the type of communication. A service should be configured to deliver certain information at a given time and therefore it should not be left to independently specify when certain

items should be added to a queue. The integration layer manages message deliveries in various manners. One method is to create a queue and then adding items to the queue. When a host is available the queue item is collected by a host to process the item. Secondly, a service sends a message through its REST-API to a different REST-API.
A message is exposed to timing failure when network latency appears. The message is subsequently added as an item to a queue and as soon as a host is available it collects the item. This type of communication is asynchronous as the message is not received in real-time. A thoroughly designed integration layer could preclude the negative effect of network latency. However, some businesses might be depended on real-time information. Aircraft flight control systems must receive exact real-time information. A pilot needs up-to-the-minute information on speed, altitude, wind resistance, availability of runway, etc to safely land the aircraft.
In sum, services are configured depending on business requirements on the degree of real-time information. Once the requirements are set the services are configured to communicate synchronously or asynchronously.

Ms. Ülker Kayhan underlined that timing failures in a testing context emerge when a host with a specific service might send a delayed response because of network latency. A delayed response could also be a result of high workloads.

According to Mr. Sadmir Halilovic timing failures have probably arisen because of network issues or heavy workload on the producing services. There might be another possible cause for timing failure; the logic of service could be based on a workflow, where calls or events are received in an unexpected order as specified in the quote (Appendix E - Mr. Sadmir Halilovic, a solutions architect, personal communication, 2020, April 10).
As a result timing failure appears in the application, but if the application is accurately designed then it would be able to handle timing failure (Appendix E - Mr. Sadmir Halilovic, a solutions architect, personal communication, 2020, April 10).

4.2.4 Value

Mr. Meneses Marshall indicated that value failures do arise and it is a matter of design as explained in the quote (Appendix E - Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, personal communication, 2020, March 31).
Value failure occurs when a service has several deliveries or dependencies. The question to pose is whether a service is correctly configured or not. In addition to faulty service configurations, it is important to check if the 1s and 0s in DB have the right format. For instance, a service requests all digits from 1 to 6, but the DB, in turn, provides only the digit 7. In this case, the service is not accurately configured. A second example is when a bank account suddenly receives a large amount of capital which was not previously there. The value delivered by the host is not validated. There must be some control mechanisms to check deposits larger than a certain figure. Both examples illustrate that a value sent by a certain host is not validated. Also, value failures could mean that the value is incorrectly registered in the DB or the information is collected from an inappropriate table. Although it is important to stress that this failure is not part of the microservice architecture, rather it is a question for the system architect and DB team to look into.

Mr. Sadmir Halilovic concluded that value failure could emerge due to an error in the service, which means that the service has sent faulty data to other internal services. This in turn, would results in inconsistent data in several services as emphasized in the quote (Appendix E - Mr. Sadmir Halilovic, a solutions architect, personal communication, 2020, April 10).
This category of failure requires corrections in both the software and databases.

4.2.5 Byzantine

Mr. Meneses Marshall introduced the topic by affirming that this category of failure must not happen as argued in the quote (Appendix E - Mr. Claudio Meneses

Marshall, a consultant at Skatteverket/a system architect, personal communication, 2020, March 31).
The respondent explained byzantine failure by having the Walt Disney scenario in mind. A Byzantine failure indicates that all Walt Disney characters can authenticate with the Mickey Mouse service and the service Donald Duck and Walt Disney are confirming the authentication. Byzantine failure mustn’t occur and it would not pass a unit test in a microservice architecture, but a Byzantine failure may emerge within monolithic applications.
To explain this category further the hosts are not accurately identifying themselves in their interaction. One host sends its client id and client secret and request for acceptance of credentials by the other host. The responding host accepts the credentials and asks the requesting host what it would like to read. If the hosts do not authenticate when interacting the whole system is fraudulent. The authentication keys are in this case missing and the very same keys are accessible to and utilized by anyone within an organization. A microservice architecture bolsters the separation of concerns to enhance failure isolation and the impact of the failure would not affect the whole system. The separation of concerns is achieved by configuring the hosts to have one specific function and each host has its client id and client secret. Interaction between hosts is approved only for a specific group of hosts and consequently, credentials are shared by interacting hosts only. When the inaccuracy has occurred it is due to human error.

In regards to impersonation one host does not impersonate another host as the system would crash. Two hosts may not have the same credentials, but several hosts could send contradictory information as the client id and client secret are accessible to all hosts.

In a testing context, byzantine failures arise very scarcely according to Ms. Ülker Kayhan.

According to Mr. Sadmir Halilovic all types of failures mentioned above are mitigated by applying best practices and patterns. If one developer of service intentionally or unintentionally neglects all or part of the patterns and best practices then the hosts will indicate various types of failure. It is common to implement patterns and best practices to preclude frequent types of failures. However, it is difficult to adopt precautions when failures are unknown and have not previously appeared. Recently, several frameworks have supported developers to handle failures. Some patterns adopted by developers; circuit breakers, retries with exponential backoff, bulkheading, etc. These patterns are adopted by the consuming services/applications (consuming either APIs or events) but the producing services/applications (producing APIs or events) might be exposed to multiple instances of the same synchronous request or asynchronous event. The producing services must be able to handle such exceptions. In other words, unexpected operations must have idempotent and commutative features, which means the result of operations must stay the same regardless of the number of repetitive operations or sequences.

4.3 How severe are the failures?

Mr. Claudio Meneses Marshall applied levels of severity to demonstrate the impact of failures on operations at Skatteverket. Crash, omission, and, byzantine failures are labelled as severe (red). A value failure is considered to be a moderate (yellow) failure, while timing has the lowest level of severity, minor (green). This is presented in the table below and the levels of severity are specified as the following:

- *Severe* is a “critical problem” (Red) — the product is unusable or an error severely impacts an End-User’s operation, and there are no workarounds to restore product functionality. A severity level of severe requires maximum effort to resolve a critical problem (Lawinsider, n.d.).
- *Moderate* is a “major problem” (Yellow) — significant product functionality is not working according to product definitions, or significant business objectives cannot be met (Lawinsider, n.d.).

- *Minor* (Green) — minor product functionality is not working according to product definitions, or minor business objectives cannot be met (Lawinsider, n.d.).

The Types of Failure	CRASH 🟥	OMISSION 🟥	TIMING 🟩	VALUE 🟨	BYZANTINE 🟥

Table 3: The Types of Failure and the Levels of Severity

In the analysis below table 4 is complemented with information on the capacity of Kubernetes security configurations mitigating failures, which affect the availability of microservice architecture for critical systems at Skatteverket.

5.0 Analysis: Kubernetes Security Configurations and Failure Handling

In this section Kubernetes' security actions are analyzed to determine whether or not availability might be enhanced by studying how diverse types of failure are handled and to what extent (fully solved, partially solved, or not at all).
Each category of failure address the following security actions in Kubernetes:

1. Securing the cluster
2. Managing authorization and authentication
3. Implementing a trusted software supply chain
4. Securing workloads in runtime
5. Managing secrets

Further, the severity of unresolved (partially or completely) failures in Skatteverket's critical systems is graded by the respondents at Skatteverket by adopting three severity levels; severe, moderate, and minor. The findings of the analysis are illustrated in table 4.

When Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect, Ms. Ülker Kayhan, a supervisor and Mr. Sadmir Halilovic, a solutions architect were asked to reflect on failures affecting availability in general terms they addressed downtime, data inconsistency, the host's inaccurate management of information, loss of authorization keys and hosts with various bugs.
When a crash failure occurs due to the Java-based processes' overconsumption of RAM Kubernetes could mitigate the crash by configuring the desired state of worker nodes in Kubernetes. The desired state determines what resources should be available. Kubernetes functionality strives towards fulfilling the specified desired state, which enables fault-tolerance, higher resource utilization, efficiency, and

horizontal scaling. In other words, these features could preclude a crash due to human error.

5.1 Crash

5.1.1 Securing the Cluster

The issue might be resolved by accurate authentication on the integration layer. Kubernetes kubeadm offers configuration possibilities of the API server to limit API access. This is accomplished by restricting API access to encrypted TLS connections made over a secured port. However, It is not clear if issues as API-Gateway downtime and erroneous connection between API-Gateway and REST-API publishing host (null response as a result) might be solved by securing Kubernetes cluster functionality. Although, when implementing Kubernetes it is possible to determine a cluster's desired state by creating objects using the Kubernetes API. This configuration could offer self-healing capabilities when API-Gateway has downtime and the REST-API publishing host sends a null-response.

The etcd limits unauthorized access which could resolve some of the API authentication problems in a microservice architecture for critical systems. However, it depends on the cause of the authentication problems of the API. Etcd configurations do not resolve crash failures involving API connections.
The benchmark list provided by the Center for Internet Security, CIS could offer some important insight to prevent a crash due to API-Gateway or faulty connection between API-Gateway and REST-API publishing host.
Also, the etcd is a component that could aid in crash prevention by limit unauthorized access to the cluster. However, to know in detail how this could be achieved further information on crash failure is needed e.g. how and when a crash emerged.

A kubelet includes container interaction at runtime to check pod health, which indicates that crash failures could be avoided by limiting access to the kubelet.

Unauthorized access to a kubelet could result in a crash as pods could start, stop, and execute unauthorized code in an unintended manner. However, this is another point that needs further research to set exactly how a kubelet could avoid crash failure.

5.1.2 Managing Authorization and Authentication

This aspect is important as it covers authentication and authorization configurations not solely for the API server but also kubelets, kube proxies, and secrets. It is essential to properly configure authentication and authorization of all components to ensure that API connections and REST-APIs function as intended. Authentication models bolster several methods to authenticate access and thereby prevent a crash due to erroneous authentication of hosts. RBAC objects (entity, resource, role, and role binding) offer fine-grained access over authorization and access on multiple levels, which could solve the authentication issues of a host.
However, it is not evident how API-Gateway downtime/missing API-Gateway and null-responses of API-Gateway could be solved by managing authorization and authentication. More details on the crash and configurations are required for this purpose.

5.1.3 Implementing a Trusted Software Supply Chain

This point concerns the containers and their applications and as each worker node in Kubernetes has a Kubelet API a crash might be avoided. Interrupted business transactions could be avoided by implementing source control, image scanning, avoiding the use of root users, applying image integrity controls, and enforcing the use of trusted images.
All components within a microservice architecture are connected in various ways and a deeper understanding of the crash context is required.

5.1.4 Securing Workloads in Runtime

The best practices to secure workloads in runtime include boundaries and controls that limit what an application within a container can do. In other words, this action includes container-internal security actions, therefore securing workloads in runtime would not prevent crash failures due to connection issues between API-Gateways and REST-APIs and inaccurate authentication of hosts hosting certain services. However, interrupted business transactions because of an unintended transfer of one or multiple pods from one worker node to another worker node within the same cluster could be mitigated by preventing unwanted traffic between nodes on a cluster and between the pod and other layers of external resources.

5.1.5 Managing Secrets

This security action encompasses solutions to enhance external and internal container security, which indicates that avoiding crash failures due to API-Gateways and REST-APIs connection issues and downtime of an API-Gateway is not resolved by such a security action. Authentication issues of a host could be mitigated by avoiding incorrect practices (such as hard-coding), storing secrets in etcd, and third party vaults. Also, utilizing environment variables and/or volume mount methods to pass secrets to containers are rewarding methods to adopt.
Moreover, it is possible to store secrets in etcd and it might offer some support in regards to credentials storage. It could be fruitful to specifically look into the topic in future research.

5.2 Omission

5.2.1 Securing the Cluster

Omission failure consists of errors within a container and, therefore, it is vital to secure the cluster by accurate authorization access to the kubelet. This would prevent unintended starting and stopping of pods, but also avoiding the execution of unauthorized code. CIS’s benchmark list could also support the automation of task checking. The benchmark list provides results of a pass and fail, which would prohibit defective code and poorly and erroneously designed services within hosts. Also, the etcd would prevent omission failure as etcd access could be limited.

5.2.2 Managing Authorization and Authentication

Omission failure caused by defective code could be restricted on multiple levels by using the Kubernetes authentication model and RBAC objects. Although, more research is required to understand this security action in depth. Omission failures are ambiguous and it is not evident how poorly and erroneously designed services within hosts could be avoided by this security action.

5.2.3 Implementing a Trusted Software Supply Chain

Such a security action is interesting due to the provided control access functionality, which covers the whole pipeline. This point ensures incoming data is validated before utilization, which could ensure accurate input parameters. Furthermore, code integrity is maintained in all phases and levels. This is accomplished by implementing source control, image scanning, avoiding the use of root users, applying image integrity controls, enforcing the use of trusted images, and securing the registry.

However, it is not clear how omission failures due to poorly designed or configured services within hosts would be avoided. Further research is required on the topic.

5.2.4 Securing Workloads in Runtime

This security action is essential to look into as it would set boundaries and controls to limit what applications within a container can do at runtime. Unwanted traffic between nodes on a cluster and traffic between a pod and other layers or external resources is limited. This action would keep the code adequate. Code integrity could also be maintained by setting a pod security policy and a security context (to define privileges and access control at the pod or container level). Although, it is not clear how omission failures due to poorly designed or configured services within hosts would be limited.

5.2.5 Managing Secrets

Managing secrets for a container is a vital security action as it would maintain code integrity. However, it would not limit omission failures due to architectural and programming errors.

5.3 Timing

Timing failure could be caused by a heavy workload on the producing services. Kubernetes provide horizontal scaling and higher resource utilization and efficiency by defining the desired state of the cluster.

5.3.1 Securing the Cluster

Timing failures are not a question of cluster security. The reasons for such failures involve communication methods; synchronous or asynchronous. Also, timing failures are a question of what communication mechanisms to adopt; e.g. update.

5.3.2 Managing Authorization and Authentication

Managing authorization and authentication would not solve timing issues. Timing failures are about the communication of services within various hosts.

5.3.3 Implementing a Trusted Software Supply Chain

This security action does not offer a solution to timing failures as timing failures arise when communication mechanisms of services are not properly configured or specified.

5.3.4 Securing Workloads in Runtime

This point consists of actions to set access boundaries and controls for applications within a container. Timing failures due to heavy workloads and network latency might be restricted by setting a network policy guardrails on pod traffic. This action would prevent unwanted traffic between nodes on a cluster and traffic between a pod and other layers or external resources. However, this topic should be carefully looked into in future studies.

5.3.5 Managing Secrets

Further, managing secrets is a matter of access control on a container-level, therefore timing failures would not be solved. Timing failures are caused by communication errors of services within hosts.

5.4 Value

5.4.1 Securing the Cluster

Value failure would not be solved by securing the cluster as a value failure is caused by inaccurately designed services within hosts. In such a scenario the services have several deliveries or dependencies. In general terms, Kubernetes might solve the issue by separating the concerns of diverse teams and by dividing an application into smaller manageable, scalable components. These components could be used by groups with diverse requirements.

5.4.2 Managing Authorization and Authentication

Value failure is am issue of design as indicated by the respondents, therefore, a value failure would not be completely prevented. However, inadequately managed authorization and authentication of hosts would severely affect the data within the cluster. All containers and their components would have faulty values.

5.4.3 Implementing a Trusted Software Supply Chain

All components within a microservice architecture are connected and to prevent value failure it is vital to complete a trusted software supply chain. This means enabling control access throughout the pipeline to validate that data input and code integrity is ensured.

5.4.4 Securing Workloads in Runtime

Kubernetes offer several native policies to enhance environment security, which could limit value failure. This implies that applications have boundaries and controls in runtime. Unwanted access and damage attacks are limited by this security action. Although, value failure due to inaccurate design of software and databases would not be mitigated by implementing this security action.

5.4.5 Managing Secrets

Value failure could also be avoided by avoiding to applying insecure practices, for example, encrypted environment variables and hard-coded secrets. In general terms, it is essential to limit user access to secrets. However, erroneously designed software and database would not be restricted by managing secrets.

5.5 Byzantine

5.5.1 Securing the Cluster

The API server in Kubernetes yields clusters controls to prevent insecure ports and unauthenticated API access. The etcd and the kubelet are vital components as secure configurations are provided. The benchmark list of 100 recommendations could bring important insights to hinder byzantine failure.

5.5.2 Managing Authorization and Authentication

This point must not be overlooked as it is one of the core actions when preventing byzantine failure. Authentication models and RBAC prohibit unwanted access to multiple components and not just the API server. Managing this aspect restricts certain capability access to authorized users, which precludes a fraudulent microservice architecture.

5.5.3 Implementing a Trusted Software Supply Chain

This action enhances the validation of data input of containers and their applications to maintain code integrity and accurate data input. Securing control throughout the pipeline is vital to prohibit byzantine failure. This is accomplished by implementing source control, image scanning, avoiding the use of root users, applying image integrity controls, enforcing the use of trusted images, and securing the registry.

5.5.4 Securing Workloads in Runtime

Byzantine failure involves different aspects, but authorization and authentication are key points to look into. This action inhibits byzantine failure by setting boundaries and controls to application actions. Only authenticated applications are authorized to perform certain actions. This is achieved by Kubernetes setting security context, determining pod security policy, and defining a network policy place guardrails on pod network traffic.

5.5.5 Managing Secrets

Access to secrets such as private keys and passwords is restricted to intended containers by avoiding insecure practices (hard-coding secrets and using unencrypted variables), encrypted storage of secrets in etcd, and limiting access of secrets to authenticated containers (volume mount method).

5.6 Summary of Analysis

The table below summarizes the findings of this report. Each of Kubernetes' security actions is evaluated based on its ability to solve the various types of failure. The failure could be fully solved, partially solved, or not solved at all as illustrated below. Furthermore, the respondents at Skatteverket have graded the severity of unsolved failures by applying levels of severity; severe (red), moderate (yellow), and minor (green).

Fully solved by Kubernetes = ●

Partially solved by Kubernetes = ◑

Not solved by Kubernetes = ○

Levels of severity

- *Severe* is a “critical problem” (Red) — the product is unusable or an error severely impacts an End-User’s operation, and there are no workarounds to restore product functionality. A severity level of severe requires maximum effort to resolve a critical problem (Lawinsider, n.d.).
- *Moderate* is a “major problem” (Yellow) — significant product functionality is not working according to product definitions, or significant business objectives cannot be met (Lawinsider, n.d.).
- *Minor* (Green) — minor product functionality is not working according to product definitions, or minor business objectives cannot be met (Lawinsider, n.d.).

	CRASH	OMISSION	TIMING	VALUE	BYZANTINE
SECURING THE CLUSTER	🟥 ◑	🟥 ●	🟩 ○	🟨 ◑	🟥 ●
MANAGING AUTHORIZATION AND AUTHENTICATION	🟥 ◑	🟥 ◑	🟩 ○	🟨 ◑	🟥 ●
IMPLEMENTING A TRUSTED SOFTWARE SUPPLY CHAIN	🟥 ◑	🟥 ◑	🟩 ○	🟨 ●	🟥 ●
SECURING WORKLOADS IN RUNTIME	🟥 ◑	🟥 ◑	🟩 ◑	🟨 ◑	🟥 ●
MANAGING SECRETS	🟥 ◑	🟥 ◑	🟩 ○	🟨 ◑	🟥 ●

Table 4: Summary of Analysis

6.0 Discussion and Conclusion

The objective of this paper was to enhance the availability of microservice architecture for critical systems by studying Kubernetes security configurations. Hence, the following question of the research was posed:

- How could Kubernetes security configurations enhance the availability of microservice architecture for critical systems?

Vayghan, Saied, Toeroe, and Khendek (2019) evaluated microservices-based applications from the availability perspective by utilizing the default Kubernetes configuration for healing (Vayghan, Saied, Toeroe, Khendek, 2019)
Taherizadeh and Grobelnik (2020) presented influencing factors in the dynamic management of scalable resources provided by Kubernetes. Also, the study evaluated the choices of such factors to develop the optimum scaling strategy to be used and analyzing the way how they dynamically influence the impact of reactive auto-scaling rules. Finally, the results demonstrated the way to tune the auto-scaling of containerized applications orchestrated by Kubernetes concerning diverse workload patterns (Taherizadeh & Grobelnik, 2020).

Both reports did not take into account how specifically Kubernetes' security actions may enhance the availability of microservice architecture for critical systems. This paper focused on each Kubernetes security action and to what extent it enabled availability of by solving the types of failure. This was performed by setting the labels; fully solved, partially solved, or not solved at all. Further, it is determined how severe the failure would be for a microservice architecture for a critical system. This was accomplished by asking the respondents at Skatteverket to specify the level of severity; severe, moderate, and minor.

The results in this paper demonstrated that a crash, omission, timing, and value do occur. However, the results indicated that the failure byzantine does not emerge and

it must not emerge in a microservice architecture for critical systems as this would have a severe impact on business and society. A byzantine failure is usually discovered early in the testing phase and is prohibited early in a system's life cycle as Mr. Claudio Meneses Marshall mentioned.
Kubernetes security configurations (securing the cluster, managing authorization and authentication, implementing a trusted software supply chain, securing workloads in runtime and managing secrets) could partially solve a crash and an omission. Both failures are crucial to mitigate in a microservice architecture for critical systems, and the failures are indicated as severe problems in table 4. The table also demonstrates that the security action; securing the cluster could fully avoid a certain aspect of omission. A value failure could be fully solved by implementing a trusted software support chain. However, the remaining Kubernetes' security actions could partially avoid a value failure. This category of failure is considered as a moderate severity by Skatteverket. Timing failure is not precluded at all by Kubernetes' security actions, which is graded as a minor level of severity. In this case, there is one exception as timing failure could be partially avoided by securing workload in runtime. Byzantine failure could be completely prevented by Kubernetes' security actions as presented in table 4.
Finally, Kubernetes security configuration could solve some of the availability concerns in a microservice architecture for critical systems, but not entirely all categories of failure and not to a full extent. Also, it is important to keep in mind that distributed systems include several layers as indicated in table 1, which means that failures precluded by Kubernetes security configurations concern only the orchestration layer.

Although this report presents some of Kubernetes' benefits and limitations concerning availability issues in a microservice architecture for critical systems on a conceptual level additional research is needed on a technical level.
Further research on diverse types of failures that might occur in a distributed system is necessary to understand the nature of failures. Also, the theory development by studying multiple cases would be fruitful as it could offer an understanding of the impact on availability. Moreover, Kubernetes' security actions require a prototype

and a testing environment including use cases and scenarios to understand the advantages and disadvantages of Kubernetes security configurations on a technical level. This would contribute to a profound understanding of the Kubernetes architecture in regards to availability in specific terms and to security in general terms.

Availability as a concept is difficult to capture as it could include availability for different components on multiple levels in a microservice architecture for critical systems. Also, it is not obvious how to separate integrity and availability as inconsistency of data does have an impact on the availability of data. Further research on data availability and data consistency would be interesting to perform.
For future work, it would also be relevant to study how architectural patterns affect availability in a microservice architecture for critical systems.

In terms of generalization, the conclusion presented in this paper is interesting for other microservice architectures for critical systems as most critical systems require high availability.

References

Alshuqayran, Nour Ali., Evans, Roger. (2016). A Systematic Mapping Study in Microservices Architecture. Conference: *Service-Oriented Computing and Applications (SOCA)*. 2016 IEEE 9th International Conference, Macau, China, 4-6 November, 2016.
DOI: 10.1109/SOCA.2016.15

Ardagna, C, A., Asal, R., Damiani, E & Hieu Vu, Q., (2015). From Security to Assurance in the Cloud: A Survey. *ACM Computing Surveys*, 48(1), Article 2 (July 2015). DOI: 10.1145/2767005

Aven, Terje. (2009). Identification of safety and security critical systems and activities. *Reliability Engineering and System Safety*, 94, 404-411.
DOI:10.1016/j.ress.2008.04.001

Balazinska, Magdalena., Hwang, Jeong-Hyon., Shah, A. Mehul. (2017). Fault Tolerance and High Availability in Data Stream Management Systems. *Encyclopedia of Database Systems*.
DOI: 10.1007/978-1-4899-7993-3_160-2

Basyildiz, B. (2019, August 19). *A Brief History of Container Technology*. Retrieved 2020, January from
https://www.section.io/engineering-education/history-of-container-technology/

Bertolino, A., De Angelis, G., Gallego, M., Garcia, B., Gortazar, F., Lonetti, F., & Marchetti, E. (2019). A Systematic Review on Cloud Testing. *ACM Computing Surveys*, 52(5), Article 93 (September 2019).
DOI: 10.1145/3331447

Bhadauria, Raman. (n.d.). *Monolithic vs Microservices architecture*. Retrieved 2020, May 22 from
https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/

Burstein, F. V., Gregor, S., (1999). The Systems Development or Engineering Approach to Research in Information Systems: An Action Research Perspective, Conference: 10th Australasian Conference on Information Systems, December 1-3, 1999,Wellington New Zeeland.
DOI:10.4225/03/57DA300758D88

Creswell, W. John. (2009). Research design: Qualitative, quantitative, and mixed methods approaches (3rd Ed). Sage Publications.

Cully, Brendan., Lefebvre, Geoffrey., Meyer, T. Dutch., Feeley, Mike., Hutchinson, Norman., Warfield, Andrew. (2008). Remus: High Availability via Asynchronous Virtual Machine Replication. Conference: *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI* 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings. Retrieved from https://www.usenix.org/legacy/events/nsdi08/tech/full_papers/cully/cully.pdf

Deduktiv metod. Nationalencyklopedin, Retrieved 2020, March 20 from http://www.ne.se.proxy.lib.ltu.se/uppslagsverk/encyklopedi/lång/deduktiv-metod

Di Francesco, Paolo., Lago, Patricia., Mavolta, Ivano. (2019). Architecting with microservices: A systematic mapping study. *The Journal of Systems and Software*, 150, 77-97.
DOI: 10.1016/j.jss.2019.01.001

Docker. (n.d.). Retrieved 2020, May 18 from https://www.docker.com/

Docker. (2020). Retrieved 2020, May 18 from https://en.wikipedia.org/wiki/Docker_(software)

Dubé, Line., Paré, Guy. (2003). Rigor in Information Systems Positivist Case Research: Current Practices, Trends, and Recommendations. *Management Information Systems and Research Center, MIS Quarterly*, 27(4), (December, 2003), 597-636. Retrieved from https://www.jstor.org/stable/30036550

Fernando, Chanaka. (2019, June 26). A futuristic view of building distributed systems with messaging [Blog Post]. Retrieved from https://blog.usejournal.com/a-futuristic-view-of-building-distributed-systems-with-messaging-560d0652513a

Hiltunen, A. Matti., Immanuel, Vijaykumar., Schlichting, D. Richard. (1999). Supporting Customized Failure Models for Distributed Software. *Distributed System Engineering*, 103 (6).
DOI: 10.1088/0967-1846/6/3/302

Hogan, Michael., Liu, Fang., Sokol, Annie., Tong, Jin. (2013). *NIST Cloud Computing Standards Roadmap*, USA: National Institute of Standards and Technology, Special Publication 500-291, Version 2. Retrieved 2020, February 11 from https://www.nist.gov/publications/nist-sp-500-291-nist-cloud-computing-standards-roadmap

Kim Seong, Dong., Machida, Fumio., Trivedi, Kishor S. (2009). Availability Modeling and Analysis of a Virtualized System.
Conference: *2009 15th IEEE Pacific Rim International Symposium on Dependable, Computing*, 16-18 Nov. 2009, Shanghai, China
DOI: 10.1109/PRDC.2009.64

Kubernetes. (n.d.). Kubernetes. Retrieved 2020, April 13 from https://kubernetes.io

Kubernetes. (n.d.) Reterieved 2020, April 13 from https://kubernetes.io/docs/concepts/

Kumar, Rakesh., Goyal, Rinkaj. (2019). On cloud security requirements, threats, vulnerabilities and countermeasures: A survey. *Computer Science Review*, 33, 1-48.
DOI: 10.1016/j.cosrev.2019.05.002

Lantz, Annika. (1993). *Intervjumetodik: den professionellt genomförda intervjun*. Lund:Studentlitteratur.

Lawinsider. (n.d.). Retrieved 2020, April from
https://www.lawinsider.com/contracts/1C477Q8DqoUpqKAiumD8Gl/sourcefire-inc/0/2010-08-05#severity-level

Loveland, S., Dow, E. M., LeFevre, F., Beyer, D., Chan, P. F., (2008). Leveraging virtualization to optimize high-availability system configurations. *IBM Systems Journal*, 47(4).
DOI: 10.1147/SJ.2008.5386515

Machida, Fumio., Andrade, Ermeson., Kim Seong, Dong., Trivedi, Kishor S. (2011). Candy: Component-based Availability Modeling Framework for Cloud Service Management Using SysML. Conference: *30th IEEE Symposium on Reliable Distributed Systems* (SRDS 2011), Madrid, Spain, October 4-7, 2011. Proceedings of the IEEE Symposium on Reliable Distributed Systems.
DOI:10.1109/SRDS.2011.33

Management & Solutions. (n.d.). Retrieved 2020, May 22 from https://www.mandsconsulting.com/oracle-cloud-devops-and-cloud-native-application s/

Mankoff, Jennifer., Rode, Jennifer, A., Faste, Haakon, (2013). Looking Past Yesterday's Tomorrow: Using Futures Studies Methods to Extend the Research Horizon. Conference: *CHI 2013, Computer Science Economics*, April 27–May 2, Paris, France
DOI: 10.1145/2470654.2466216

Marsh, David. (2002). *Theory and Methods in Political Science*. Palgrave: Macmillan.

National Institute of Standards and Technology. (2020). Retrieved 2020, March 3 from
https://sv.wikipedia.org/wiki/National_Institute_of_Standards_and_Technology

Olsson, Jonas. (2019, January 15). FRA: Cyberangrepp mot Sverige ökar. *Svt Nyheter*. Retrieved 2020, January 14 from
https://www.svt.se/nyheter/fra-cyberangreppen-mot-sverige-okar

Pesonen, Lauri. I.W., Eyers, David., Bacon, Joy., (2006). A capability-based access control architecture for multi-domain publish/subscribe systems. Conference: International Symposium on Applications and the Internet, 2006. SAINT 2006, January 23-27, Phoenix, AZ, USA.
DOI: 10.1109/SAINT.2006.1

Oracle. (n.d.). Retrieved 2020, May 4 from
https://docs.oracle.com/cd/E41633_01/pt853pbh1/eng/pt/tibr/concept_Asynchronous Messaging-07656e.html

Phaphoom, Nattakarn., Wang, Xiaofeng., Abrahamsson, Pekka. (2013). Foundations and Technological Landscape of Cloud Computing. *ISRN Software Engineering*.
DOI: 10.1155/2013/782174

Red Hat. (n.d.). Retrieved 2020, May 18 from
https://www.redhat.com/en/topics/containers/what-is-kubernetes

OpenShift. (2020). Retrieved 2020, May 22 from
https://en.wikipedia.org/wiki/OpenShift

Regeringsbeslut Fi2018/02509/RS (delvis). (2018, July 5) *Regleringsbrev för budgetåret 2018 avseende Skatteverket*. Retrieved 2020, January 14 from https://www.esv.se/statsliggaren/regleringsbrev/?RBID=19255

Regeringsbeslut Fi2019/04080/S3 (delvis). (2019, December 19) *Regleringsbrev för budgetåret 2019 avseende Skatteverket*. Retrieved 2020, May 21 from https://www.esv.se/statsliggaren/regleringsbrev/?RBID=20413

Rice, Liz., Burns, Brendan. (2019). *The Definitive Guide to Securing Kubernetes*. Retrieved from https://www.techrepublic.com/resource-library/whitepapers/the-definitive-guide-to-securing-kubernetes/

Rouse, Margaret. (2020, April). *What is RESTful API (REST API)?* Retrieved 2020, April 30 from
https://searchapparchitecture.techtarget.com/definition/RESTful-API

Ruparelia, Nayan. (2016). *Cloud Computing*, USA: The MIT Press Essential Knowledge Series.

Service-Level Agreement. (2020). Retrieved 2020, April 30 from https://en.wikipedia.org/wiki/Service-level_agreement

Skatteverket. (2017, December). *2018–2020 Verksamhetsplan för Skatteverket.* Retrieved 2020, January 14 from https://www.skatteverket.se/download/18.4a4d586616058d860bc4f4e/1515598042648/verksamhetplan-for-skatteverket-2018-2020-skv190-utgava22.pdf

Skatteverket. (n.d.). Retrieved 2020, May 21 from https://www.skatteverket.se/foretagochorganisationer/myndigheter/utbetalningavkommunalskattemedel.4.361dc8c15312eff6fd1a79d.html

Smith, W. E., Trivedi, K. S., Tomek, L. A., Ackaret, J. (2008). Availability analysis of blade server systems. *IBM Systems Journal*. 47(4). DOI:10.1147/SJ.2008.5386524

Soldani, Jacopo., Tamburri, Damian, Andrew., Van den Heuvel, Willem-Jan. (2018). The Pains and Gains of microservices: A Systematic Grey literature review. *The Journal of Systems and Software*. 146, 215-232.
DOI: https://doi.org/10.1016/j.jss.2018.09.082

Swagger. (n.d.). Retrieved 2020, April 30 from https://swagger.io/docs/specification/2-0/api-host-and-base-path/

Taherizadeh, Salman., Grobelnik, Marko. (2020). Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Advances in Engineering* Software, 140, 102734.
DOI: https://doi.org/10.1016/j.advengsoft.2019.102734

Wallen, Jack. (2017, May 10). Kubernetes: A Cheat Sheet. *TechRepublic*. Retrieved 2020, May 18 from
https://www.techrepublic.com/article/kubernetes-the-smart-persons-guide/

Vohra, Deepak. (2016). *Kubernetes with Docker*. White Rock, British Columbia, Canada.

Vayghan, A. Leila., Saied, A.Mohamed.,Toeroe, Maria., Khendek, Ferhat. (2019). *Kubernetes as an Availability Manager for Microservice Applications.* (Master's Thesis). Concordia University, Gina Cody School of Engineering and Computer Science, Montreal, Quebec, Canada.

Wallen, Jack. (2020, March 17). Kubernetes Security Guide. *Techrepublic*. Retrieved 2020, March 17 from
https://www.techrepublic.com/resource-library/downloads/kubernetes-security-guide-free-pdf/

Whitney, Lance (2020, May 27). Why developed countries are more vulnerable to cybercrime. *TechRepublic*. Retrieved 2020, June 1 from
https://www.techrepublic.com/article/why-developed-countries-are-more-vulnerable-to-cybercrime/

Xiao, Zhifeng., Xiao, Yang. (2013). Security and Privacy in Cloud Computing. *IEEE Communications Surveys & Tutorials*, 15 (2).
DOI:10.1109/SURV.2012.060912.00182

Yargina, Tetiana. (2018). *Exploring Microservice Security* (Doctoral dissertation, The University of Bergen, Norway). Retrieved from http://bora.uib.no/handle/1956/18696

Yin, R. K. (2009). *Case study research: Design and methods* (4th Ed.). Thousand Oaks, CA: Sage.

Zainal, Zaidah, (2007). Case study as a research method. *Jurnal Kemanusiaan*. 5(1). Retrieved from https://www.researchgate.net/publication/41822817_Case_study_as_a_research_method

Zhang, Qi., Cheng, Lu., Boutaba, Raouf. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1, 7-18.
DOI 10.1007/s13174-010-0007-6

Appendix

Appendix A

Questionnaire sent by email to Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a System Architect, 2020, February 20

1. Vilka säkerhetstester ska genomföras?
2. Vilka problem möter ni när säkerhetstester genomförs?
3. Finns det use case gällande säkerhetskrav och säkerhetstester för den interna och externa appen?
4. Jag behöver ha bakgrund till problemen.
5. Vilka metrics är relevanta att undersöka för att säkerställa att den interna och externa appen är rätt skapade ur säkerhetsperspektiv?
6. Vilka fel uppstår i microservice - arkitekturen när det gäller den interna och externa appen?

Appendix B

Meeting Agenda, 2020, March 10

Participants: Mr. Claudio Meneses Marshall, a Consultant at Skatteverket/System Architect, Ms. Ülker Kayhan, a Supervisor, Mr. Fredrik B. Andersson, a Solutions Architect,

Inledning - Tre availability - krav på infrastrukturnivå (Machida, Andrade, Kim Seong, Trivedi, 2011).

Krav 1 - Fault tolerance (Machida, Andrade, Kim Seong, Trivedi, 2011).

Krav 2 - Load-Balancing (Machida, Andrade, Kim Seong, Trivedi, 2011).

Krav 3 - Automatic Scale-Up (Machida, Andrade, Kim Seong, Trivedi, 2011).

Appendix C

Meeting Agenda, March 2020, March 16

Participants: Mr. Sadmir Halilovic, a Solutions Architect, Ms. Ülker Kayhan, Supervisor, Mr. Fredrik B. Andersson, a Solutions Architect

1. Jag har förstått att Kubernetes inte erbjuder ett monitoring tool utan snarare erbjuder Prometheus monitoring på virtualization layer, dvs att man kan skala upp CPU och memory så nodes/workers matchar desired state i deployment file. Så jag ser det som en viss monitoring funktion och jag vill gärna veta mer om den och på vilket sätt den funktionen automatiseras?
2. Red Hat Openshift Container Platform använder Kubernetes komponenter som exempelvis Prometheus och Grafana. Kan du berätta mer om det ur ett arkitekturperspektiv?
3. Vilka fördelar och nackdelar ser du med Kubernetes?
4. Vilka behov och krav uppfyller Kubernetes och vilka krav och behov kan inte tillgodoses av teknologin?
5. Hur kommer Kubernetes användas vid förändringar av behov och krav?
6. Vilka use case är aktuella för Kubernetes?
7. Kan Kubernetes användas för att förbättra availability i en mikroservicearkitektur och automatisera monitoring och i sådana fall på vilket sätt?

Appendix D

Interview Questions on Types of Failure

Types of Failure / Feltyper

Tänk dig en mikroservicearkitektur, där det finns flera noder som inte delar minne och är anslutna i ett nätverk som tillhandahåller en opålitlig kommunikations-mekanism.
Noder kan bara interagera genom att skicka och ta emot meddelanden via kommunikationsnätverket. Anta också att kommunikationsnätverket inte har permanenta fel.
En nod kan endast påverka andra noder genom att inte skicka ett meddelande när det borde, genom att skicka ett meddelande när det inte ska, eller genom att skicka ett felaktigt meddelande (Matti A. Hiltunen, A., Immanuel, Vijaykumar., Schlichting, D. Richard, 1999)
/
Let us consider a microservices architecture, where several nodes do not share memory and are connected to a network that provides an unreliable communication mechanism.
Nodes can only interact by sending and receiving messages through the communication network. Also, assume that the communication network does not have permanent faults.
A node can only affect other nodes by not sending a message when it should, by sending a message when it should not, or by sending an incorrect message.
(Matti A. Hiltunen, A., Immanuel, Vijaykumar., Schlichting, D. Richard, 1999)

1. Vilka fel kan uppstå med noderna och i deras interaktion?
2. Crash
   En nod kan krascha och därmed misslyckas med att skicka meddelanden. Om en nod håller på att skicka ett meddelande när det kraschar, kan det vara så att vissa av de avsedda mottagarna inte får meddelandet.

a. Har det hänt att en nod har kraschat?
b. På vilket sätt har det kommit i uttryck?
c. Vad hände?

3. Omission

En nod kan vid upprepade tillfällen och oregelbundet misslyckas med att skicka ett meddelande till alla eller till några av de avsedda mottagarna, eller misslyckas med att ta emot meddelanden.

a. Har det hänt att en nod har vid upprepade tillfällen och/eller oregelbundet misslyckats med att skicka ett meddelande till alla eller till några önskade mottagare?
b. På vilket sätt har det kommit i uttryck?
c. Vad hände?

4. Timing

En nod kan skicka ett meddelande tidigare eller senare än väntat. Ett nätverk med oväntade fördröjningar Ett meddelande som blir oväntat fördröjt på grund av ett långsamt nätverk kan leda till att en nod verkar ha ett timing-fel.

a. Har det hänt att en nod har skickat ett meddelande tidigare eller senare än väntat?
b. På vilket sätt har det kommit i uttryck?
c. Vad hände?

5. Value

En nod kan skicka ett meddelande med felaktigt innehåll. Om en nod med fel skickar ett multicast-meddelande så kommer alla mottagare att få samma felaktiga meddelande.

a. Har det hänt att en nod har skickat ett meddelande med felaktigt innehåll?
b. På vilket sätt har det kommit i uttryck?
c. Vad hände?

6. Byzantine

Här kan en nod uppvisa olika typer av fel på en och samma gång. Det innebär att utöver felen value och timing så kan en nod förvirra andra noder genom att skicka olika versioner av ett meddelande till olika mottagare eller genom att utge sig för att vara en annan nod.

a. Har det hänt att en nod har skickat ett meddelande med felaktigt innehåll vid en tidigare eller senare tillfälle än väntat, skickat flera versioner av ett meddelande till olika mottagare samt utgivit sig för att vara en annan nod?
b. På vilket sätt har det kommit i uttryck?
c. Vad hände?

7. Fler Fel?

Kan du komma på andra fel som inte tagits upp här?

Appendix E

The quotes of the respondents.

Failures affecting Availability

Mr. Claudio Meneses Marshall, a Consultant at Skatteverket/a System Architect. Interview by phone, 2020, March 31

“Du frågar efter en tabelluppdatering och så får du inte svar tillbaka. Då måste du ha en mekanism för att hantera det. Så du har en övervakning där som säger: “Här har vi ett svar som vi inte får”. Då är det en indikation på att någonstans är fel med databaskommunikationen. Då måste det generera någon typ av alarm. Där har du något som ligger utanför nätverket i sig. Det ligger i integrationslagret, i databaslagret.”

Mr. Sadmir Halilovic, a Solutions Architect. Interview questions sent by email, 2020, April 10

“ I dessa fall kan det vara fördelaktigt att "systemet" är nedbrutet i separata tjänster (microservices) då enbart den eller dem påverkade tjänsterna behöver stoppas/rättas till.”

Crash

Mr. Claudio Meneses Marshall, a Consultant at Skatteverket/a System Architect. Interview by phone, 2020, March 31

“Kontrollera alltid nätverket först. Två, kontrollera integrationslagret. Fungerar API-Gateway på rätt sätt? Troligtvis ligger svaret där.”

Ms. Ülker Kayhan, a Supervisor, Interview questions sent by email, 2020, April 7

“Problemet med våra applikationer är 99/100 ggr vårt eget fel.”

Omission

Mr. Claudio Meneses Marshall, a Consultant at Skatteverket/a System Architect. Interview by phone, 2020, March 31

“Som konsekvensanalys så måste man titta på hur noden är uppbyggd utifrån arkitektur- och kodsynpunkt. Det här är det värsta som kan hända.”

“Det är det mest komplexa scenario som finns och det är det som händer dagligen. OBS! överallt”

Mr. Sadmir Halilovic, a Solutions Architect. Interview questions sent by email, 2020, April 10

“Sannolikt har det berott på tjänsten inte använder mönster som mitigerar en crash, network failure eller dylikt.”

Timing

Mr. Claudio Meneses Marshall, a Consultant at Skatteverket/a System Architect. Interview by phone, 2020, March 31

“Det här ska inte förekomma. Observera ska inte förekomma. Här pratar vi om en tjänst som ska skicka ett meddelande, send eller get, svara på send- eller get-kommando. Det där måste verifieras från början. Ska det vara synkron? Ska det vara asynkron, just-in-time? Ska det vara genom update. Det är olika mekanismer som vi har för att säga: “Uppdatera mig, informera mig när du har fått någonting nytt”.

Mr. Sadmir Halilovic, a Solutions Architect. Interview questions sent by email, 2020, April 10

“Möjligtvis även där tjänstelogik bygger på ett flöde (workflow) och tar emot anrop eller events i oväntat ordning”.

Value

Mr. Claudio Meneses Marshall, a Consultant at Skatteverket/a System Architect. Interview by phone, 2020, March 31

“Ja, det förekommer och det förekommer överallt.”
“Och det är också en designfråga”

Mr. Sadmir Halilovic, a Solutions Architect. Interview questions sent by email, 2020, April 10
“Förmodligen leder det till att data i ett system som består av flera tjänster blir inkonsistent.”

Byzantine

Mr. Claudio Meneses Marshall, a Consultant at Skatteverket/a System Architect. Interview by phone, 2020, March 31
“Det här kan inte förekomma.”
“Noderna måste autentisera sig sinsemellan, annars är hela systemet utsatt för manipulation. Så det ska inte förekomma i en mikrotjänstarkitektur. Det blir väldigt väldigt svårt. Då är det inte systemvetenskap. Det här är klantighet”
“Man har tappat bort nycklarna. Det skulle det kunna vara. Så hela företaget har tillgång till en och samma nycklar.”

Appendix F

Case Description - Questions part 1, 2020, April 20

These questions were sent by e-mail to Mr. Claudio Meneses Marshall, a consultant at Skatteverket/a system architect.

1. Vilka verksamhetsprocesser är samhällskritiska?
2. Vilka samhällskritiska verksamheter ska poddar och appar inom en mikroservicearkitektur ge systemstöd för?
3. Vilka verksamhetsmål ska apparna ge systemstöd för och på så vis hjälpa verksamheten att uppnå sina mål?
4. Varför är availability viktig för Skatteverkets samhällskritiska system/ mikroservicearkitektur?

Case Description - Questions part 2, 2020, May 15

1. För vilka tjänster har Skatteverket kritiska system? Skattedeklaration? Fler (både internt och externt)?
2. När det gäller mikrotjänster hur länge har Skatteverket haft dessa?
3. Vilka erfarenheter har Skatteverket av Kubernetes?
4. I vilken fas befinner sig Skatteverket när det gäller implementeringen av Kubernetes? Är det i design, utveckling, test eller drift?
5. Hur länge har Skatteverket implementerat Kubernetes och i vilken omfattning?
6. Vilka komponenter av Kubernetes används?
7. Vilka inslag av Kubernetes finns i Skatteverkets kritiska system?