DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Games Engineering

Static Source Code Analysis of Container Manifests

Janik Noah Müller

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Games Engineering

Static Source Code Analysis of Container Manifests

Statische Quelltextanalyse von Containerisierungsmanifesten

Author:	Janik Noah Müller
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Jürgens, Timo Pawelka, Roman Haas
Submission Date:	October 19th 2020

I confirm that this master's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, October 19th 2020 Janik Noah Müller

Acknowledgments

I thank my advisers Roman Haas, Timo Pawelka, Dr. Elmar Jürgens as well as Prof. Dr. Dr. h.c. Manfred Broy from the department of Software and Systems Engineering of the Technical University of Munich for enabling me to write this thesis about static source code analysis of container manifests.
I am especially indebted to Roman Haas and Timo Pawelka, who put a lot of time and effort into this project. Thank you for your continuous support.
I am grateful to all employees of CQSE GmbH for integrating me into their company and providing the underlying tools (Teamscale) necessary for the implementation.

Abstract

Over the past years the usage of container and cloud native applications has grown significantly. Kubernetes is the most popular option for container orchestration and deployment at scale. Kubernetes provides a lot of powerful features but many companies cannot keep up with its rapid evolution and misconfigured Kubernetes manifests are the consequence. In 2020, StackRox carried out a survey where 69% of the respondents reported that they detected misconfiguration in their Kubernetes environment over the past 12 months which led to security incidents or issues [45]. It is the most common vulnerability for Kubernetes emphasizing the need for an approach to detect misconfiguration in Kubernetes manifests early and automatically. In this thesis, a static source code analysis tool is implemented which detects misconfiguration in Kubernetes manifests and Helm charts caused by the violation of best practices. The implemented tool detects violations of 28 Kubernetes related best practices. The best practices cover the topics security, resource management, availability and structure. On average, the tool detects a findings density of 31.87 findings per 1,000 Lines of Code. It is able to detect misconfiguration in practice, especially in the security context. The implemented tool produced a false-positive rate of 24.44%. This serves as a first estimate because the sample size of the assessed findings was insufficient. The most false-positives occurred due to the use of external software affecting Kubernetes. It made some checks obsolete, so that they could only produce false-positive findings. In that case, they should be disabled before the analysis. It is recommended to configure the tool according to the project's special circumstances if some best practice violations are tolerated, for example because of the usage of external software. If the checks which could only produce false-positives due to project's special circumstances are excluded, the false-positive rate for the implemented tool is 2.86%.

Contents

1 Introduction

Cloud native technologies and containerization have raised in popularity over the past years. The latest survey of the Cloud Native Computing Foundation reports that 84% of the respondents are using containers in production [18]. Kubernetes does container orchestration and deployment at scale. It is not the only software for this task, but the "CNCF Survey 2019" report indicates that 78% of the respondents use Kubernetes in production [18]. It is a significant development compared to the previous year where only 58% reported that they are using Kubernetes in production. Kubernetes establishes itself as dominant player for container orchestration. Kubernetes had its 6th birthday on June 7th, 2020. Over this relatively short period of time over 90k commits were contributed to the project [31]. According to GitHub's report "The State of the Octoverse", 6.9k contributors worked on the Kubernetes project putting it in 7th place in the category Number of Contributors to Open Source Projects [20]. These numbers demonstrate the relevance and the IT community's interest in Kubernetes.
But the rapid growth of the software also entails risks. The orchestration of containers and applications is an important part in production. It comprises the scaling of the applications and security mechanisms like network access control, authorization and authentication. Kubernetes provides powerful features to handle these aspects but developers need to understand them and learn how to use them correctly. Keeping up with the fast pace Kubernetes is evolving can be challenging. Best practices need to be established and developers need to understand and adhere to them. In their latest report, StackRox has stated that 69% of the respondents reported that they have detected misconfiguration in their Kubernetes and container environment causing security incidents in the past 12 months [45]. It depicts that misconfiguration of manifests is the most common type of vulnerability in Kubernetes. StackRox's finding demonstrates the need for an approach to automatically detect misconfiguration and best practice violations as early as possible in the development stage.

In this thesis a static source code analysis tool is implemented which analyzes Kubernetes manifests and detects violations of best practices. In total, 34 current best practices for Kubernetes configurations are presented. They determine how to set up a Kubernetes cluster securely and efficiently. Consequently, if they are adhered to, they prevent misconfiguration in Kubernetes manifests. The best practices are partitioned in

the groups Security (16 best practices), Resource Management (7), Availability (7) and Structure (4). The implemented tool includes checks detecting violations for 26 out of the 34 best practices. The static source code analysis approach is an effective method because it can perform the analysis automatically and detects defects early in the development stage before the changes are deployed in production. The implemented tool analyzes all Kubernetes manifests within a project. It also detects Helm charts and locally renders them to a Kubernetes manifest in order to analyze them as well.
The evaluation results show that the average findings density the implemented tool detects is 31.87 findings per 1,000 Lines of Code. The Security best practice group is violated most often, but it also is the largest group. The most findings producing checks are Set Limits (see Subsection 4.2.2), Set Requests (see Subsection 4.2.1) and Use Labels (see Subsection 4.4.1). Project's special circumstances can limit the effectiveness of the implemented tool. Determined special circumstances for the study objects of this thesis are the toleration of a certain best practice violation and the usage of external software affecting Kubernetes. One study object requires multiple files providing values to locally render the Helm chart's templates to a Kubernetes manifest. As a consequence, 24 of 29 Helm charts could not be rendered locally and thus, could not be analyzed for best practice violations. The determined false-positive rate for the implemented tool is 24.44%, but 20 out of 22 detected false-positives are caused due to the study object's special circumstances. If the usage of external software makes some checks obsolete or some best practice violations are tolerated on purpose, then the corresponding checks should be disabled. If the false-positives of these checks are excluded, the tool has a false-positive rate of 2.86%. However, the false-positive rates are derived from a small sample size and are only a first estimate. The false-positive rate evaluation must be replicated on a larger sample size. The evolution of a study object regarding the number of findings and the project size has been evaluated by executing the implemented tool on the code base of every commit in the repository's history. The results show that the findings density decreases over time. Thus, the implemented tool is the most useful in the early stage of a project where it detects the highest findings density and supports the developers to adhere to best practices from the beginning.

The thesis has the following outline. Chapter 2 explains the fundamental terms and definitions. Chapter 3 discusses scientific work related to this thesis. Chapter 4 presents best practices for Kubernetes partitioned in the four categories Security, Resource Management, Availability and Structure. In Chapter 5, the implemented tool is explained. Chapter 6 presents the evaluation of the implemented tool. Chapter 7 depicts approaches for future work improving upon this thesis and Chapter 8 gives the conclusion of this thesis.

2 Terms and Definitions

This clarifies all relevant terms and definitions for this thesis.

2.1 Finding

Finding is a term for quality defects. In the scope of this thesis, it means a violation of a Kubernetes best practice. A finding contains the description of the quality defect, the area where the defect is located at in the code and a recommendation how to fix the quality defect. The finding is categorized as error or as warning depending on the relevance of the defect.

2.2 Static Kubernetes Manifest

Static Kubernetes manifests are Yaml or Json files which contain specifications of one or more Kubernetes resources. The Kubernetes resources must be complete and static. The static Kubernetes manifest's content is not allowed to depend on templating software like Helm offers it.

2.3 Kubernetes Resource

The terms Kubernetes Resource and Kubernetes Object are used interchangeably in the scope of this thesis. On the official Kubernetes website Donnelly, Wang, and Bannister [16] defined the term Kubernetes Object like this:

> *Kubernetes objects* are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. [...] A Kubernetes object is a "record of intent"–once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's *desired state*.

2.4 Namespace

Sayfan [40] defines the term Namespace as follows:

> A namespace is a virtual cluster. You can have a single physical cluster that contains multiple virtual clusters segregated by namespaces. Each virtual cluster is totally isolated from other virtual clusters, and they can only communicate through public interfaces.

2.5 Pod

Sayfan [40] has the following definition for Pod:

> A pod is the unit of work in Kubernetes. Each pod contains one or more containers. Pods are always scheduled together (always run on the same machine). All the containers in a pod have the same IP address and port space; they can communicate using localhost or standard inter-process communication. In addition, all the containers in a pod can have access to shared local storage on the node hosting the pod. The shared storage will be mounted on each container. Pods are important feature of Kubernetes. [...] Pods provide a great solution for managing groups of closely related containers that depend on each other and need to co-operate on the same host to accomplish their purpose. It's important to remember that pods are considered ephemeral, throwaway entities that can be discarded and replaced at will. Any pod storage is destroyed with its pod. Each pod gets a unique ID (UID), so you can still distinguish between them if necessary.

2.6 Deployment

On Google Cloud an article about the Deployment resource object in Kubernetes defines it like this:

> Deployments represent a set of multiple, identical Pods with no unique identities. A Deployment runs multiple replicas of your application and automatically replaces any instances that fail or become unresponsive. In this way, Deployments help ensure that one or more instances of your application are available to serve user requests. Deployments are managed by the Kubernetes Deployment controller. (Google [22])

2.7 Service

Sayfan [40] gives the following definition for Service in the Kubernetes context:

> Services are used to expose some functionality to users or other services. They usually encompass a group of pods, usually identified by [...] a label. You can have services that provide access to external resources, or to pods you control directly at the virtual IP level. Native Kubernetes services are exposed through convenient endpoints. Note that services operate at layer 3 (TCP/UDP). Kubernetes 1.2 added the Ingress object, which provides access to HTTP objects. [...] Services are published or discovered via one of two mechanisms: DNS, or environment variables. Services can be load-balanced by Kubernetes. But, developers can choose to manage load balancing themselves in case of services that use external resources or require special treatment.

2.8 Helm

Helm is the Kubernetes package manager. When using Helm, even the most complex application can be described in charts. A chart consists of the three key components template, values and chart file. The chart file itself contains metadata about the chart. The templates are Kubernetes components like in Kubernetes manifests, but values can be replaced by placeholders. These placeholders have unique names and they get the actual values assigned either from the value.yaml file or during run time. Dependencies can be added manually in the allocated directory or they are only specified in the requirements.yaml file for Helm v1 or in the Chart.yaml file for Helm v2.

3 Related Work

In this chapter, research related to this thesis is introduced. The sources are partitioned on the basis of the way they relate to this thesis. Sources analyzing code quality of files for Kubernetes deployment are represented in the first section. The other section contains related research regarding static source code analysis.

3.1 Code Quality Control for Kubernetes

Scientific research presented in this section target best practices for Kubernetes manifests, Helm charts or container images and most of them evaluate a tool to detect vulnerabilities with regards to the best practices.
Shamim, Bhuiyan, and Rahman [43] conducted a grey literature review to determine security practices and systematize that knowledge in order to help practitioners in securing Kubernetes installations. They analyzed 104 internet artifacts like blog posts or video presentations containing experiences, recommendations and best practices for Kubernetes security. As a result, Shamim, Bhuiyan, and Rahman [43] identified 11 Kubernetes security best practices. A similarity to this thesis is that internet artifacts are considered valuable sources to determine best practices. In contrast to this thesis, Shamim, Bhuiyan, and Rahman [43] limited their field of research to the security aspect. They determined best practices at a higher level of abstraction than it is the case in this thesis where only best practices on code-level are considered. As a consequence, one security practice identified by Shamim, Bhuiyan, and Rahman [43] is "Vulnerability scanning" meaning that container images and deployment configurations should be inspected before they are released to the public. This demonstrates the need for a static source code analysis tool to detect vulnerabilities and emphasizes the relevance of this thesis. The security practices "Implementing Kubernetes-specific Security Policies", "Namespace separation" and "Limit CPU and memory quota" are also elaborated in this thesis and the implemented tool detects violations of these best practices.
In his thesis, Schwegler [42] elaborated on security best practices for Kubernetes and evaluated security scanners for Docker. He describes security mechanisms in Kubernetes and gives practical examples how to use them. He compares security scanners for Dockers regarding the detection of known vulnerabilities and introduces the tools "KubeSec" and "kube-bench" which detect violations of security best practices for static

Kubernetes manifests. He evaluates the economic costs to operate Kubernetes clusters in the cloud versus on-premise as well as evaluating costs for cluster per customer versus multiple customers in one cluster. One difference to this thesis is that Schwegler focuses only on security best practices, but he integrates Docker and the economic context in his research. Another difference is that Schwegler gives examples how to configure a cluster securely following best practices in the first place whereas this thesis is focused on the implementation of a tool which reports violations of best practices. Instead of the introduced tools "KubeSec" and "kube-bench" which cover only security best practices, "Kube-Score" is integrated in the tool implemented in this thesis because it covers multiple areas and more best practices in total.
Spillner [44] developed and evaluated the quality assessment tool HelmQA which automatically detects quality insufficiency in Helm charts. The tool detects duplicated values and replaces them in order to use Helm's templating feature to full effect. The study object is the "KubeApps Hub" repository which is the official platform to share and retrieve Helm charts. The "stable" directory of that repository is also a study object in this thesis. The evaluation was conducted by daily executing the tool on the study object over four months. The tool's results were sent to the chart maintainers and the acceptance of the findings were evaluated over the time period. Spillner concluded that the tool HelmQA has a sufficiently low false-positive rate and the recommendations were accepted, but the quality improvement over the evaluation period of four months increased less than expected. The major difference between the implemented tools is that HelmQA only analyzes Helm charts and reports quality insufficiency if Helm features are not used according to best practices. The tool implemented in this thesis is capable to analyze Helm charts, but it only analyzes the Kubernetes resources created by the chart and reports violations of Kubernetes best practices. The methodology used by Spillner [44] differs to the one used in this thesis. In this thesis, the implemented tool is executed on multiple study objects. The effectiveness of the tool is measured by the findings density it produces and the false-positive rate is determined by manually validating the correctness of randomly picked findings. In contrast, the methodology of Spillner [44] was to apply the tool on the study object over a certain period of time and evaluate the effectiveness and false-positive rate on the basis of the maintainer feedback and quality improvement of the analyzed charts.

3.2 Static Source Code Analysis

This section describes related work which evaluate or use the same approach applied in this thesis.
In his paper, Bardas [3] explained the essential constituents of static code analysis,

evaluated its strengths and limitations and presented well-known static source code analysis tools. The most common techniques used for static code analysis are pattern matching and data-flow analysis. The static code analysis approach used in this thesis solely focuses on pattern matching because the Kubernetes manifest specify a configuration and do not contain executable code which could be analyzed regarding its data-flow. Bardas [3] stated that it is impossible for static code checkers to assure correctness of a program and also false-positives can occur in static code analyses. The false-positive rate is an important factor for determining the usefulness and developer's acceptance of the tool. Thus, evaluating the false-positive rate for the tool implemented in this thesis is a research question in this thesis' evaluation. According to Bardas [3], it is an important constituent of static code analysis that the resulting report is easy to understand for all developers and that it clearly presents the detected weaknesses, its severity and location. The implemented tool fulfills this constituent as it reports findings which contain a description of the weakness, its location, even gives recommendations for fixing the issue and specifies the severity by categorizing the finding as error or warning. The strengths of static code analysis depicted by Bardas [3] are that it can be performed on modules and unfinished code and does not require a finished or executable application. It is independent of any particular execution and dynamic user input. Static code analysis can be performed early in the development stage before the software is released and while the detected vulnerabilities are still relatively cheap to fix. Bardas identified the fixed number of checks as the major limitation of static code analysis. The analysis can only detect known weaknesses specified in the tool's rule database. Weaknesses for which no rules have been defined cannot be detected. The tool's database containing rules for known weaknesses has to be updated every time a new type of vulnerability is detected. The other limitation mentioned by Bardas is that only vulnerabilities in the implementation level can be detected by static code analysis and not violations of design or architecture requirements. One part of this thesis deals with the elaboration of Kubernetes related best practices and the implemented tool contains checks for these current best practices. However, if new best practices are established in the future, the implemented tool's checks need to be extended by new checks for the new best practices. The fact that static code analysis automatically detects weaknesses early in the development stage is the decisive strength making static code analysis the method of choice to solve the problem of misconfiguration in Kubernetes manifests. Bardas [3] came to the conclusion that "[s]tatic analyzers should be a key part of every software development process". This emphasizes the relevance of a static source code analysis tool for Kubernetes configurations and consequently, the relevance of this thesis.

Zampetti et al. [53] studied the usage of static code analysis tools in continuous integration pipelines. The evaluation was conducted on 20 Java open source projects.

The research questions were the identification of concrete static code analysis tools being used and how they are configured, secondly which type of issues are causing build failures and thirdly how long it takes to resolve broken builds. The results of Zampetti et al. [53] were that almost all static source code analysis tools are configured according to the project and that this configuration changed a limited number of times over the observed period of time. One reason they named for the modification of the configuration is that some checks became irrelevant. This observation fits to the finding of this thesis that the implemented tool has to be configured for every project in order to maximize the tool's effect. If some checks are obsolete due to external software or because the best practice violation is tolerated, then these checks should be disabled in the tool's configuration. In their second research question, Zampetti et al. [53] detected a limited amount of broken builds and a high percentage of warnings. The build failures are mainly caused when coding standards were violated and warnings were reported for checks with a high number of false-positives. The results of their third research question were that most broken builds were resolved within eight hours by actually fixing the reported issue instead of adapting the building script or the static code analysis tool's configuration. Combining the results of the second and third research question it shows that developers accept the static code analysis results if it reports violations of coding standards. Best practices are one form of coding standards, so it can be expected that developers accept the results of the implemented tool reporting violations of best practices as well. Requirements are that the reported violations contain a low number of false-positives and if the tool is not integrated in the continuous integration pipeline causing broken builds, developers might not fix the issues as quickly as in the study of Zampetti et al. [53] because it might be less urgent to them. Zampetti et al. [53] concluded that "[f]orcing the adherence to coding guidelines is one of the most useful applications of [automatic source code analysis tools]". The adherence to best practices is very similar to the adherence to coding guidelines. It can be assumed that static source code analysis is an effective method to enforce best practices and prevent misconfiguration.

4 Solution Approach

The goal of this thesis is the creation of a tool which does static source code analysis in order to detect violations of best practices in Kubernetes manifests. The purpose of this chapter is building the foundation for the tool by elaborating on best practices for Kubernetes manifests. The focus is set on best practices whose adherence can be examined with static analysis. The main source for this chapter is the book "Kubernetes Best Practices" written by Burns et al. [7]. Articles and blog entries by experienced Kubernetes developers are also valuable sources because they often give specifications how to adhere to or violate the best practice on the implementation level. These specifications can be used to implement the static analysis checks which test if the best practice is violated.
The best practices are partitioned into the categories Security, Resource Management, Availability and Structure. Within these categories a paragraph about the meaning and importance of each best practice is given.

4.1 Security

Kubernetes is a flexible system which manages low-level resources in a generic way [40]. Kubernetes has access to system critical resources like networking and resource allocation and it runs container images which contain unknown source code. Kubernetes needs to isolate components especially the system critical resources. It needs to execute the black box container images which could potentially contain malicious code. Kubernetes should grant only the minimal access rights and capabilities necessary. Kubernetes provides features to address these potential threats, but it is the developer's responsibility to use them and configure the Kubernetes manifests correctly following best practices.

4.1.1 Stable API Version

The API version has to be specified for every resource in Kubernetes manifests. It is best practice to use the latest stable API version [10]. It is a potential threat to use a deprecated version because these versions are not supported or may lose support

in the future [23]. When Kubernetes resources use a deprecated API version it is recommended to migrate them to the latest stable release.

4.1.2 Read-only Root File System

Usually a container only needs write access to a mounted volume which stores data and the current state not only for this container instance but also all its replicas [6]. Configuring the containers to have read-only root file system rights decreases the attack surface and can prevent malicious processes to store or manipulate data inside a container [2]. Otherwise an attacker who compromised the container could write an executable file and run it inside the container [39].

4.1.3 Run as Non-root

By default all processes in a container run as the root user which often comprises more permissions than the workload requires [6, 35]. If that is the case and the Kubernetes configuration does not explicitly set runAsNonRoot to true, an attacker who compromised the container and managed to escape from it immediately has root access on the host [38]. Enabling this setting gives an extra layer of protection because the attacker would need an additional attempt to get root access [5].

4.1.4 Run as High User

The previous best practice recommends to run as non-root which is equivalent to the user identification 0. This best practice extends the previous one by adding the condition that the user identification value should be set to a value above 10,000. This avoids conflicts with the host's user table and common values [49].

4.1.5 Disable Privileged Setting in Security Context

Containers are defined as privileged if their container user identification of 0 is mapped to the host's user identification of 0 [6]. Privileged containers have effectively the same permissions as root access on the host which are more capabilities than they usually need [38, 39]. Considering the general security principle Least Privileges [39] the goal is to limit container's capabilities to the bare minimum necessary to operate correctly. Setting the privileged option to false where possible is best practice and one step of creating a secure security context [7].

4.1.6 Disallow Privilege Escalation in Security Context

The next step towards creating a secure security context is disabling the setting to allow privilege escalation. Even if the capabilities have been assigned correctly to the container, as long as privilege escalation is enabled processes inside the container can gain more privileges constituting a security risk [36]. Although privilege escalation is enabled by default, there is rarely a situation where an application requires more permissions during run time than it received at the start [2].

4.1.7 Set Image Pull Policy to Always

The default image pull policy would store the pulled image in the node's cache which can be accessed by all pods on that node [39]. This configuration has the following known security issue. Pod A has the necessary credentials for image S, pulls it and consumes the secret. The image is stored in the node's cache. Pod B located on the same node does not have the necessary credentials but uses the cached image without the need to pull the image again. It circumvents the container registry security [46]. Setting the value "always" for the image pull policy prevents the usage of a cached image because every pod needs to pull the image whenever it wants to use it and this requires to pass the registry credentials check [51].

4.1.8 Use Tags for Containers but Not *:latest*

Having no tag for containers defaults to using the *:latest* tag. The latest container image will be pulled and that may or may not be the expected version [5]. Looking at Docker as a popular software to build containers, there the *:latest* tag is nothing more than a tag. Naturally, it is expected to be always the most recent pushed version of the container but that is not the case. The *:latest* tag has no additional functionality. It is the default if no other tag is given, which makes it error-prone for human mistakes [47]. A developer who accidentally forgets to tag his pushed container version would immediately overwrite the *:latest* image which could lead to unexpected behavior or even failing containers being deployed to your production cluster if it pulls the *:latest* image version. The solution is to use unique and descriptive tags for example using the Git hashes is a common good practice [7, 41].

4.1.9 Enforce Role-based Access Control

An authorization mode should be configured because otherwise any user would have cluster-admin privileges [30]. The role-based access control supersedes the now deprecated attribute-based access control. It is the most used security mechanism to

implement a fine-grained permission structure for actions performed against the API by users or groups [7]. It is best practice to create custom roles on the application-level [48]. Consequently, for every application there is a service account specified in the corresponding pod and the service account only holds the permissions the application needs considering the Least Privilege principle. Creating customized roles for many applications and service accounts requires high administrative effort. The trade off between administrative effort and adhering to the Least Privileges principle has to be chosen for every project individually.

4.1.10 Disable Automatic Mount of Service Account Token

All pods and containers without an explicit service account get the default service account automatically assigned [25]. The default service account has various permissions and most applications do not even need to talk to the API [39]. Therefore, the option to automatically mount an API token should be disabled for service accounts.

4.1.11 Disable Host PID

If a container runs in the host's PID namespace, it can snoop on processes running outside the container. If the container also has ptrace capabilities, it could be exploited to escalate privileges outside of the container [36].

4.1.12 Disable Host IPC

When host IPC is enabled the container can interact with processes outside the container circumventing this layer of isolation [36].

4.1.13 Disable Host Network

A container running in the host's network namespace has access to the local loopback device. This way it could access network traffic to and from other pods [36].

4.1.14 Use Secrets for Confidential Information

Confidential information like credentials, tokens or keys should always be stored in Kubernetes secret resources [7, 25]. Furthermore, sensitive Configmap and Secret resources should be encrypted by external tools [35].

4.1.15 Require Network Policies

With network policies the communication between all Kubernetes components and also with endpoints outside the network can be controlled [7, 25]. Network policies can create firewall-like protection between pods running on the same Kubernetes cluster [37]. If an attacker could compromise a container, the attacker would try to explore the network to compromise other containers and hosts as well [39]. Restricting network access and communication path would isolate different components, making it more difficult for the attacker to infiltrate the entire system. The official Kubernetes website recommends to deny all traffic in the network and then incrementally add the necessary routes as a best practice [35].

4.1.16 Deny Not Trusted Registries

Containers are black boxes for Kubernetes. They are trustfully executed without inspecting them for malicious code or other security vulnerabilities. Due to this limitation, it is best practice to allow containers from trusted registries only [7]. The Kubernetes admission controller can be used to enforce image registry governance policies which automatically deny all images from not trusted registries [30].

4.2 Resource Management

Managing resources includes allocation of low-level resources like memory and CPU. These resources can be assigned on namespace-, pod- or container-level specifying the minimal amount needed for the component and the maximum amount of resources the component may claim. These values among others are important specification for the Kubernetes scheduler to manage and distribute the components efficiently between the nodes. Additionally, managing resources means supervising that all components fulfill the task they are intended for.

4.2.1 Set Requests

The request value determines the amount of resources the container or pod is guaranteed to get [6]. The scheduler will only mount the pod or container on a node that can provide the required resources [7, 13]. Setting the requests correctly avoids the threat of container throttling due to the lack of resources [25].

4.2.2 Set Limits

The limit value determines the maximum of resources the container or pod can claim. Specifying limits prevents resource hogging by misbehaving containers and thus, hinders Denial of Service attacks through this application [25]. Further, it serves the scheduler as a metric to choose the best suited node for the current pod [37]. Limits can be set for memory and CPU as resources. If a container exceeds its CPU limit, Kubernetes starts throttling the container because CPU is considered a compressible resource [7]. This can result in a worse performance, but does not cause the pod to terminate. Memory cannot be artificially compressed in any way leading to a termination of the pod if an internal container exceeds its memory limit [13].

4.2.3 Set CPU Request Less or Equal to One

Unless the application executes highly computational tasks and is designed to use multiple cores, it is best practice to set the CPU request to one or below [37, 41]. Running more replicas of the container compensates for the restriction of this best practice and gives the system more flexibility and reliability [13].

4.2.4 Set Namespace Quotas

Similar to the three previous best practices requests and limits can be specified for namespaces. This feature is called namespace quotas and sets the requests and limits for all pods and all its containers within one namespace [7, 37]. If the namespace quotas are not defined, "noisy neighbor" scenarios could arise [29]. Without resource quotas one developer team could hog more resources than their fair share on the cluster reducing the resources and performance for other teams to less than their fair share on the cluster [13].

4.2.5 Use Ingress over LoadBalancer

Using the type LoadBalancer for services provides high availability and high performance, but the cloud provider also allocates more resources charging more money for the service [5]. The Ingress component is able to route multiple services through a single external endpoint. Ingress, however, is not the preferable choice in every scenario as it only supports the HTTP(S) protocol and not UDP or TCP [7].

4.2.6 Do Not Specify Host Port

It is recommended to leave the host port unspecified, so that it is chosen dynamically when being scheduled. Otherwise the options to schedule the pod are limited as the combination of host IP, host port and protocol has to be unique for every pod that is scheduled on a node [10]. The host IP defaults to 0.0.0.0 and the protocol defaults to TCP if not specified differently. This leaves the host port as only flexible part in this unique combination. If the host port is set to a fixed value, it can conflict with other pod's host port, host IP, protocol combination making it difficult to find a suiting node especially if other aspects such as resource requests and limits are taken into account. It certainly makes scheduling and resource management less efficient. Thus, it should be avoided to set the host port to a fixed value.

4.2.7 Set Dependent Components

In order to operate as intended Kubernetes components require certain information when they are defined in a Kubernetes manifest. The required information are different for the various Kubernetes components. Some examples are that Ingress components must target at least one service, ConJob components must configure a deadline, Service components must target at least one pod and HorizontalPodAutoscaler components must target a valid object [51].

4.3 Availability

A key feature of Kubernetes and a main purpose of all orchestration tools is the scalability and availability it provides to the orchestrated applications. This section comprises best practices regarding scalability, availability of resources and failure recovery.

4.3.1 Horizontal Pod Autoscaler

The HorizontalPodAutoscaler is a Kubernetes component which automatically scales the number of pod replicas up or down based on the current usage of the respective pod during run time [37]. It uses CPU, memory or custom metrics for its evaluation whether a pod should be scaled up/down [7, 25]. It is important that deployments do not have a replica count set configured statically if a HorizontalPodAutoscaler is used to change this value dynamically [51].

4.3.2 Cluster Autoscaler

The cluster autoscaler has a similar approach as the HorizontalPodAutoscaler, although it does not target scaling on pod-level but scaling on node-level. It determines the utilization of nodes and scales the cluster accordingly by requesting new nodes from the cloud provider or removes idle nodes [25]. Another signal that more nodes are needed is if there are pending Pods [7]. This can occur when the Pod's request value is too high for any node to provide the requested resources, see Subsection 4.2.1.

4.3.3 Odd Number of Master Nodes

It is necessary to have an odd number of master nodes equal or higher as three in order to have a clear majority in case of a network split [40]. Following this best practice the cluster can overcome the failure of one master node and by choosing an odd number of master nodes the cluster has the same tolerance as the next higher even number of master nodes [25].

4.3.4 Configure Pod Anti Affinity

Replicas of Pods have the purpose to increase availability and reliability. Assigning all replicas on the same node reduces the effectiveness of this measure because the failure of this one node would cause all replicas to be unavailable [41]. Configuring the PodAntiAffinity can guarantee an even distribution of replicas across all nodes and ensures that enough replicas are available even when one node fails [7].

4.3.5 Specify Pod Disruption Budget

The PodDisruptionBudget specifies the number of replicas that must remain available upon a voluntary disruption [7, 41]. A voluntary disruption could be a rolling update. Instead of shutting down all Pod replicas simultaneously, the PodDisruptionBudget prevents the specified amount of replicas to shut down to assure this amount of availability even during an update. The remaining old versions of the Pod can be updated as soon as updated versions of the Pod are up and running.

4.3.6 Set Termination Grace Period

For a voluntary disruption Kubernetes sends a sigterm signal to the Pod as first step in the termination process. Sending the sigterm signal starts the termination grace period which defaults to 30 seconds [41]. During this period of time the pod has to do tasks like saving data and its state, closing network connections, finishing their current request or

task [15]. If the Pod finished its graceful termination, Kubernetes immediately continues with the termination process without waiting for the full 30 seconds to pass [15]. But if the time is up and the Pod did not finish its graceful termination, Kubernetes sends a sigkill signal forcing the Pod to shut down. A force shut down could lead to memory leaks or data loss which would be fatal. Thus, knowing the maximum time needed for Pods to gracefully terminate and setting the termination grace period correctly is a critical best practice. Despite the fact that Kubernetes immediately continues when the Pod finished its termination, it is not suitable to set termination grace period infinitely high. A Pod could run in a deadlock during termination making the force shut down necessary. An infinitely high termination grace period would cause the system to get stale and cancel the failure recovery mechanisms. Thus, setting the termination grace period correctly is important for a graceful shut down and performance reasons.

4.3.7 Create Health Checks

Configuring health checks for distributed systems is almost mandatory and Kubernetes is no exception [14]. By default Kubernetes monitors if processes are running or not to determine whether they are healthy [5]. The default behavior can be optimized by creating custom health checks in order to improve the speed and accuracy of detecting unhealthy processes. The two checks that should be defined are readiness probes and liveness probes.

Readiness Probes

The purpose of the readiness probe is to detect whether a container is ready to receive traffic [7, 41]. Without the readiness probe Pods could be terminated or receive user requests during the initialization process conveying unavailability to the user [6]. When the probe starts failing Kubernetes stops sending traffic to the respective Pod until it continues passing the readiness probe [14, 25].

Liveness Probes

The liveness probe's purpose is to detect when containers are in a broken state they cannot recover from like a deadlock [7, 25]. When the probe fails Kubernetes removes the Pod with the failed container and creates a new replica as a replacement [14, 37].

Readiness and liveness probes should never use the same check because the reaction when the check fails differs. Readiness probe failure leads to a temporary stop of traffic until the Pod is ready again whereas liveness probe failure leads to an immediate

termination of the Pod [1]. During initialization or when a Pod is busy with a request the readiness probe should fail because the Pod cannot take another request on. But the liveness probe should pass while the Pod is in one of these healthy states because they are doing meaningful work and are not stuck.
Also readiness and liveness probes should not call dependent services to avoid cascading failure [41].

4.4 Structure

This section contains all best practices which target an organizational or structural concern. It includes syntax rules, usage of the Labels feature in Kubernetes as well as the usage of the Kubernetes package manager Helm.

4.4.1 Use Labels

Labels are a powerful feature of Kubernetes. The user can create custom labels and assign arbitrary values to them. This helps to identify and organize Kubernetes components into groups. Further, it allows bulk operations on a group of Kubernetes components determined by querying on label values [25]. It is best practice to use plenty of descriptive labels and use them on as many Kubernetes components as possible to maximize their effect [5].

Recommended Labels

The official Kubernetes website recommends the usage of the following labels: name, instance, version, component, part-of and managed-by [17]. They have the shared prefix app.kubernetes.io so that they do not conflict with custom user labels. In order to use these labels to full effect, they should be specified for every Kubernetes component without exception [7, 24].

Prohibit No Labels

It is recommended to create a list of required labels, a list of should-have labels and a list of optional labels [24]. This provides clear guidelines to the developers across teams and guarantees a consistent labeling convention. Based on this method the best practice of prohibiting Kubernetes components with no labels assigned is deduced. The sole exceptions are the Kubernetes components ReplicaSet and ReplicationsController. In case that these Kubernetes components have no labels assigned themselves, their

labels are the same as the labels of the Pod that the ReplicaSet or ReplicationController manages [32].

4.4.2 Follow Syntax Rules

The Kubernetes OpenAPI specification defines syntax rules for Kubernetes manifests. For every option it specifies the type and the size of the value acceptable for the corresponding field. Labels names and label values for example have a maximum character count of 63, they must begin and end with an alphanumeric character and in addition to alphanumeric characters only dashes, underscores and dots are allowed as special characters [11, 24].

4.4.3 Do Not Set Namespace to Default in Manifests

The default namespace is created by Kubernetes itself and as the name indicates it is used as default if no other namespace is specified [32]. Using the default namespace only makes sense if the project is small enough that it is the only namespace being used [8]. In that case, it is best practice to never specify a namespace in Kubernetes manifest to have a clean and uniform style. If the project requires multiple namespaces, the default namespace should not be used but only custom namespaces because adding objects to the default namespace makes role-based access control more difficult [7, 36]. Further, using the default namespace makes the system error prone to human mistakes as developers may forget to explicitly set the namespace which adds the Kubernetes component to the default namespace and that could lead to overwriting or disrupting other Kubernetes components in that namespace [12]. If the project is big enough that it requires multiple namespaces, then it is best practice to create separate namespaces for individual teams, projects or customers [25]. Using multiple namespaces does not add a performance penalty [12]. In contrary, it can even improve performance because requests and limits can be defined on a namespace-level, see Subsection 4.2.4. Also, creating multiple namespaces is the first level of isolation providing separation of concern and decreasing the attack surface [19].

4.4.4 Use Helm

Helm is the Kubernetes package manager and the base structure is explained in 2.8. Helm should be used as a best practice because it improves managing complexity, update strategy, sharing Kubernetes manifest data and safe rollbacks [40]. Despite the fact that the actual values for the placeholder used in the templates can be assigned during run time it is best practice to assign default values for every placeholder in

the value.yaml file, so that the deployment does not fail if no new values are assigned during run time. In-place upgrades and custom hooks simplify the update strategy [40]. There is a official repository to share preconfigured Helm charts with all necessary dependencies with the community [5]. It contains Helm charts for popular software components which can be deployed to any cluster out of the box. Helm provides the functionality to rollback a set of recent changes to a cluster with a single command [40].

5 Implementation

After elaborating on best practices for Kubernetes manifests the following part of this master thesis is describing the implementation of a prototypical static source code analysis tool which implements most of the identified best practices from Chapter 4.

5.1 Covered Best Practices

The focus of the implementation are checks for the most relevant best practices and checks for best practices which could be implemented in a reasonable amount of time. Table 5.1 shows for every best practice whether the implemented tool detects violations of the best practice. If a best practice is not covered by the implemented tool, the Table specifies a short reason for it, but in the following paragraph these reasons are explained in more detail.
The prototype is able to detect violations of 11 best practices for the Security category, 6 for the Resource Management category, 5 for the Availability category and 4 for the Structure category. The best practice Use Helm 4.4.4 is integrated in the prototype as it not only supports static Kubernetes manifests but also Helm charts as input. Adhering to the Run as High User 4.1.4 best practice already implies compliance with the Run as Non-root User 4.1.3 best practice. Regarding the Disallow Privilege Escalation in Security Context 4.1.6 best practice there is the option to change the default value with the PodSecurityPolicy resource. Given the fact that it is possible to influence the behavior in multiple ways makes it more difficult to test if the best practice is adhered to. The best practice Use Secrets for Confidential Information 4.1.14 cannot be tested efficiently with a static source code analysis tool because of the limitation to determine what data in non-Secret Kubernetes resources corresponds to confidential information. Deny Not Trusted Registries 4.1.16 and Enforce Role-based Access Control 4.1.9 are two best practices which highly depend on project's individual requirements making it difficult to create generally applicable checks. Additional software and service providers like Rancher and NeuVector could take over cluster deployment and full life cycle container security. Using them makes best practices like Cluster Autoscaler 4.3.2, Set Namespace Quotas 4.2.4 and Odd Number of Master Nodes 4.3.3 obsolete because the external software handles the orchestration regarding these aspects. Developers do not need to specify a configuration for these aspects in Kubernetes manifests.

Table 5.1: Presents whether the implemented tool detects violation for the best practice. The Comment column either gives the reason why the best practice is not covered or it specifies whether the check is provided by Kube-Score or an own implementation.

Best Practice	In Prototype	Comment
Stable API Version	✓	Provided by Kube-Score
Read-only Root File System	✓	Provided by Kube-Score
Run as Non-root	✗	Included in Run as High User
Run as High User	✓	Provided by Kube-Score
Disable Privileged Setting in Security Context	✓	Provided by Kube-Score
Disallow Privilege Escalation in Security Context	✗	Multiple ways for adherence
Set Image Pull Policy to Always	✓	Provided by Kube-Score
Use Tags for Containers but Not *:latest*	✓	Provided by Kube-Score
Enforce Role-based Access Control	✗	Project specific differences
Disable Automatic Mount of Service Account Token	✓	Own implementation
Disable Host PID	✓	Own implementation
Disable Host IPC	✓	Own implementation
Disable Host Network	✓	Own implementation
Use Secrets for Confidential Information	✗	Difficult to identify confidential information
Require Network Policies	✓	Provided by Kube-Score
Deny Not Trusted Registries	✗	Project specific differences
Set Requests	✓	Provided by Kube-Score
Set Limits	✓	Provided by Kube-Score
Set CPU Request Less or Equal to One	✓	Provided by Kube-Score
Set Namespace Quotas	✗	External software could make best practice obsolete
Use Ingress over LoadBalancer	✓	Own implementation
Do Not Specify Host Port	✓	Own implementation
Set Dependent Components	✓	Provided by Kube-Score
Horizontal Pod Autoscaler	✓	Provided by Kube-Score
Cluster Autoscaler	✗	External software could make best practice obsolete
Odd Number of Master Nodes	✗	External software could make best practice obsolete
Configure Pod Anti Affinity	✓	Provided by Kube-Score
Specify Pod Disruption Budget	✓	Provided by Kube-Score
Set Termination Grace Period	✓	Own implementation
Create Health Checks	✓	Provided by Kube-Score
Use Labels	✓	Own implementation
Follow Syntax Rules	✓	Provided by Kube-Score
Do Not Set Namespace to Default in Manifests	✓	Own implementation
Use Helm	✓	Own implementation

5.2 Technology Used

In this section, every external tool used in the prototype is introduced by explaining its general purpose and its role regarding the created prototype.

5.2.1 Teamscale

The client/server application Teamscale serves as underlying framework and tool for this thesis. Teamscale is developed by CQSE GmbH. It is a software quality analysis tool, which performs incremental analysis [26]. Teamscale analyzes and stores the history of a system. The tool provides several repository connectors to version control systems like Git, SVN and TFS. Teamscale uses a variety of known quality measures such as structure metrics, clone detection and code anomaly [21].
For this prototype, Teamscale is used to maintain the project. It automatically fetches changes to the file system or the version control system repository and launches the analysis. It provides the delta which contains all files that have been added, changed or deleted during the recent change. It also provides the content index containing all files that belong to the project.

5.2.2 Kube-Score

Kube-Score is an open-source static source code analysis tool owned by Gustav Westling. It receives the content of Kubernetes manifests as input and returns a report with recommendations to make the application more secure and resilient based on best practices [50]. Figure 5.1 shows an example report created by Kube-Score. There are other tools which also perform static source code analysis for Kubernetes manifests, for example KubeSec or Kubeval. But Kube-Score is the most extensive tool providing checks which target a large variety of best practices from all categories as defined in Chapter 4.

5.2.3 Helm

As mentioned in Section 4.4.4, it is best practice to use Helm for the described benefits. Thus, it is important that the implemented prototype is capable to analyze Helm charts. The Helm software is used to render the Kubernetes manifests locally by replacing the placeholder in the templates with actual values from the value.yaml file. At this point, the best practice to specify default values for every placeholder variable is important for this Helm feature to work properly. Another crucial aspect for a successful local rendering process is providing the chart dependencies. The required charts can be provided directly in the allocated directory or they are only specified

```
[ {
      "object_name": "Deployment/apps/v1/default/web2",
      "type_meta": {
          "kind": "Deployment",
          "apiVersion": "apps/v1"
      },
      "object_meta": {
          "name": "web2",
          "namespace": "default",
          "creationTimestamp": null
      },
      "checks": [
          {
              "check": {
                  "name": "Label values",
                  "id": "label-values",
                  "target_type": "all",
                  "comment": "Validates label values",
                  "optional": false
              },
              "grade": 10,
              "skipped": false,
              "comments": null
          },
          {
              "check": {
                  "name": "Container Image Pull Policy",
                  "id": "container-image-pull-policy",
                  "target_type": "Pod",
                  "comment": "Makes sure that the pullPolicy is set to Always.
                  This makes sure that imagePullSecrets are always validated.",
                  "optional": false
              },
              "grade": 1,
              "skipped": false,
              "comments": [
                  {
                      "path": "web2",
                      "summary": "ImagePullPolicy is not set to Always",
                      "description": "It's recommended to always set the
                      ImagePullPolicy to Always, to make sure that the
                      imagePullSecrets are always correct, and to always
                      get the image you want."
                  }
              ]
          }
      ]
  } ]
```

Figure 5.1: An example for a Kube-Score report including only two checks.

in the requirements.yaml file for Helm v1 or the Chart.yaml file for Helm v2. If the dependencies are only specified by name, version and repository, an additional Helm command is necessary to download archives for these charts. It is not the common case that charts have dependencies and an additional execution of an external software which downloads data is very time-consuming. In order to minimize the cost, the command to download dependencies is neglected at first. Only when the process to locally render the chart failed and the error indicates that the cause is missing dependencies the Helm command to download the dependencies gets executed and another attempt to locally render the chart starts. This course of action keeps the feedback time low because for the most charts only one Helm command is executed. But in the case that dependencies are specified, three Helm commands are executed: the failing local render attempt, downloading the missing dependencies, rendering the chart locally again. This sequence of commands has high performance cost and should be avoided. Thus, it is recommended to add relevant charts to the designated folder manually for better performance.

5.3 Prototype

In this section, the key aspects of the prototype are explained. The prototype is capable to analyze projects where static Kubernetes manifests and Helm charts are used interchangeably. First, the files and Helm charts which need to be analyzed have to be detected. Then, the Helm charts need preliminary work to output the static Kubernetes manifest needed for the analysis. Kube-Score and the custom checks implemented as part of this thesis analyze the Kubernetes manifests and report violations of best practices.

5.3.1 Identification of Source Files

The first step is the identification of source files within the project which need to be analyzed. Teamscale provides the delta between two sequential versions of the manifests and only these changed or added files may contain new findings. All other files have already been analyzed in their current version. Among the files in delta only Kubernetes manifests are targets for the analysis. Filtering files according to their extension is not enough as yaml and json files can also be used for other purposes than Kubernetes manifests. Additionally to the extension, the file's content is checked for the base structure of Kubernetes resources. Every Kubernetes resource must define apiVersion, metadata and kind as fields which serves as an identifier.
For the remaining files it needs to be determined whether they are belonging to a Helm chart. Helm charts have a fixed structure and it is required to have the file named

Chart.yaml at the first level in the chart folder. Thus, a Kubernetes manifest which has a Chart.yaml file in the same directory or a parent directory is part of a Helm chart.

5.3.2 Assembling Analysis of Helm Charts

The analysis takes static Kubernetes manifests as input and not Helm charts itself. Helm charts need to locally render the Kubernetes manifests. This procedure is described in Section 5.2.3. Teamscale stores the project data in a database. It provides the content of all files but not the files themselves stored on the local machine. In contrast, Helm operates on the actual chart represented as files on the local machine. The prototype circumvents this mismatch by rebuilding the chart in a temporary directory with temporary files. For the execution of Helm it is important to restore the chart with all its files and not only the Kubernetes manifest in yaml or json format. Custom definitions can be stored in files with the .txt extension and without them, the attempt to locally render the chart into Kubernetes manifests would fail.
Helm renders all template files into one static Kubernetes manifest splitting the contained Kubernetes resources by "- - -". The fact that only one Kubernetes manifest displays the content of the entire chart makes it difficult to attach the findings of the analysis to the responsible file within the chart. As backup and default setting, all findings are attached to the Chart.yaml file and the developers need to do the extra step to locate the source for this finding within the chart's template files. Helm itself provides the solution for this issue by writing the responsible source file's path as a comment in the first row for every Kubernetes resource. The prototype looks for this information and updates the source from the generic default option to the correct file for all findings found in this Kubernetes resource.

5.3.3 Analysis of Kubernetes Manifests

The final step is the analysis of the Kubernetes manifests. Kube-Score takes the manifest as input and returns a report. Kube-Score gives grades for every implemented check (see Figure 5.1, line 21). Additionally, the report contains a general description of the best practice tested with the corresponding check (see Figure 5.1, line 30) as well as comments which specify the concrete issue in the manifest and provide recommendations how to fix it (see Figure 5.1, line 36-45). The prototype parses the report and creates a finding for every failed check. The best score for a grade is 10. If the check has the grade 10, it passed and the best practice is adhered to. Grades between 4 and 9 are warnings, whereas grades below 4 are considered critical errors. Kube-Score offers a human-readable output where only violated checks are displayed and the violations are categorized as warnings or errors. The threshold of 4 has been determined by

comparing the grades for the checks and the assigned category in the human-readable output.
As part of this thesis, checks are implemented in order to extend the coverage of best practice for this prototype. Table 5.1 shows which best practices are covered by checks implemented as part of this thesis and which are covered by Kube-Score's checks. As a preparation for the execution of the checks, the Kubernetes manifests are split by the divider "- - -" resulting in a list of Kubernetes resources. The Kubernetes resources are parsed with the SnakeYaml library to get a navigable structure. The best practices target a specific Kubernetes resource type or component like metadata. The type is given in the "kind" field which must be defined for every Kubernetes resource. The components like metadata or Pods can be defined at multiple places within a Kubernetes resource. Pods can be defined as a Kubernetes resource itself or as a nested object in the "template" field. Metadata is defined for every Kubernetes object whether it is a Kubernetes resource or only defined as a nested object within another Kubernetes resource. It is important to detect all occurrences of these components and execute the checks targeting these components for every instance.
For every metadata instance it is checked whether the "namespace" field is defined and set to "default". In that case, a warning is reported, see also Section 4.4.3. Further, the metadata must contain a non-empty "labels" field. The only exception to this rule are metadata instances belonging to Kubernetes resources of the type ReplicaSet or ReplicationController, see also Section 4.4.1. Every Pod object has a "spec" field and the assigned PodSpec must specify a value for the "terminationGracePeriodSeconds" field as recommended by the best practices described in Section 4.3.6. If the PodSpec object has objects assigned to the "containers" or "initContainers" fields, these objects are checked. If an object defined in their "ports" field defines the "hostPort" field, a warning is reported, see also Section 4.2.6. If the fields "hostPID", "hostIPC" or "hostNetwork" are defined and set to true in the PodSpec, warnings are reported by the responsible checks, see also Section 4.1.11, Section 4.1.12 and Section 4.1.13. Similarly, the same fields are checked for Kubernetes resources of type PodSecurityPolicy in its PodSecurityPolicySpec object assigned to the "spec" field. The PodSpec or Kubernetes resources of type ServiceAccount should define the field "automountServiceAccountToken" and set it to false as explained in Section 4.1.10. Otherwise a warning is reported. A warning is also reported if a Kubernetes resource of kind Service specifies LoadBalancer as the value for the field "type" in the service's "spec" object, see also Section 4.2.5.

5.4 Limitations

The implemented static source code analysis tool has limitations explained in the following subsections.

5.4.1 Delta Split

As mentioned in Subsection 5.2.1, Teamscale creates the delta containing the files that have been added or changed with the currently analyzed commit. For commits with a large amount of added or changed files, these files are divided into multiple deltas. The tool is initiated for every delta separately. This procedure is meant to improve parallelism and threading, leading to a better performance. But, for the execution of this tool, it causes the opposite when Helm charts are involved. The files are assigned randomly to the deltas. If a chart consists of 10 Kubernetes manifests and these 10 files are all assigned to different deltas, the Helm chart would be rendered and analyzed 10 times giving redundant results. This limitation causes a significant drop in performance for the analysis of large commits. A solution for this issue would be either to adapt Teamscale by assigning files in the same directory to the same delta or to extend the tool by tracking which charts have already been analyzed for a commit.
Teamscale provides the option to set the delta size in the project settings. When the implemented tool is executed, it is recommended to set the delta size higher than the number of files in the project, so that Teamscale always creates only one delta per commit even for the initial commit where all project files are considered as newly added.

5.4.2 Detection of Kubernetes Manifests

Kubernetes manifest are specified in yaml or json files. Files with these extensions can also be used for other purposes than Kubernetes. The implemented tool addresses this issue by looking for the Kubernetes base structure within the file's content as mentioned in Subsection 5.3.1. Every Kubernetes manifest must have the base structure. As a consequence, the occurrence of false-negatives is impossible. However, the search is not specific enough to only accept Kubernetes manifests. It is possible to have false-positives. The tool would try to analyze the false-positives, most likely leading to execution error findings. This mechanism is meant to be an additional control step to filter non Kubernetes manifest files, but it is still the responsibility of the developer to exclude paths which do not contain Kubernetes manifests upon project creation in Teamscale.

5.4.3 Support for External Tools

The implemented tool supports the use of the external tool Helm in order to organize Kubernetes manifest in charts. Helm is the package manager of Kubernetes and commonly used. But there are other external software that would need the same level of support by modifying the tool internally, for example if they offer an alternative templating feature for Kubernetes manifests. Missing support for an external tool may obstruct the implemented tool of this thesis in successfully analyzing projects relying on the external tool.

5.4.4 Number of Checks

The tool is limited to the checks implemented or provided by Kube-Score. It is not capable to detect all violations of best practices as the checks do not cover all best practices elaborated on in Chapter 4 (see Figure 5.1). Furthermore, Kubernetes is an evolving software and that is why the best practices for working with Kubernetes are evolving as well.

6 Evaluation

This chapter presents the evaluation of the aforementioned static source code analysis tool. In the first section of this chapter, the research questions are defined, followed by a description of the study setup and design. Then the study objects are introduced. Afterwards, the results are presented and in the discussion section the results are assessed. The final section of this chapter deals with the threats to validity regarding this evaluation.

6.1 Research Questions

This section defines the research questions and their purpose.

6.1.1 RQ1: What finding densities can be observed when running our tool on the study subjects?

The number of findings detected by the tool are measured. In order to generalize the result across the study objects, the number of findings is given in proportion to the study object's size. The purpose of this research question is the evaluation of the effectiveness of the tool. It depicts whether findings can be found in practice.

6.1.2 RQ2: Which checks produce the most findings and which best practice group is violated most often?

The best practices are grouped into the categories introduced in Chapter 4. The total number of findings per best practice and per category are evaluated. These numbers indicate for which category the prototype performs the best. This research question identifies best practices which are violated most commonly and also identifies the best practice group which is violated the least. Our hypothesis is that the Security category is violated the least. Adhering to the security best practices is system critical and usually a special focus is put on providing a secure application and protecting the application against as many known vulnerabilities as possible. The assumption is that the developer teams creating the study objects already integrated mechanisms to adhere to these best practices without using the tool presented in this work.

6.1.3 RQ3: What false-positive rate needs to be expected from our tool? Which checks produce the most false-positives?

Best practices provide guidelines how to do certain things in a usually correct way. But it is not always the only correct way and in some special cases it might even be false. Sometimes best practices are violated on purpose because of the special circumstances in the certain case. If best practices are violated, findings are reported giving recommendations what to do in order to adhere to the best practice and improve the Kubernetes configuration. But, as mentioned before, there are exceptions to the rules defined by the best practices. Thus, the implemented tool is expected to have some noise in its reported findings. The findings produced by the checks differ in importance. They are partitioned in errors and warnings. Errors are violations of best practices which should be adhered to in most cases and are rarely violated on purpose. Warning findings are expected to produce more noise. They are referring to best practices where violations on purpose happen more frequently. But the warnings give the developer the opportunity to double check whether the current version is meant to be like it is.
The purpose of this research question is the evaluation of the tool's proneness to errors and noise. False-positives are all reported findings which are incorrect or invalid because of the special case.

6.1.4 RQ4: How did large projects evolve over time with regard to the number of findings and size?

In this research question the evolution of the study object is analyzed with regard to the number of findings and the project's size in order to evaluate when the tool is the most useful. Our hypothesis is that the findings density decreases over time. At the beginning of a project, the focus is usually on getting the application running. Over time the team has more capacity to check more best practices and extend the application to adhere to them. Another hypothesis is that the correlation of decreasing number of findings and passing time is less pronounced with regard to the security best practices. Extending the hypothesis mentioned in Subsection 6.1.2 the findings density should be consistently low for the Security category.

6.2 Study Setup

The technologies necessary are described in Section 5.2. The versions used for this evaluation are Teamscale 6.1, Kube-Score 1.8.1 and Helm 3.3.1. Teamscale has been modified to a custom version by implementing the prototype internally. For every

study object a Teamscale project is created. The study objects are added via the Teamscale Git repository connector and the entire development history is analyzed. The "master" branch is analyzed for every study objects. Only the Zalando study object is an exception as its "dev" branch is analyzed because it is declared to be the main branch in the repository's description.

6.3 Study Design

This section describes the methodology used to conduct the evaluation. For every research question an explanation is given how the results are obtained.

6.3.1 RQ1: What finding densities can be observed when running our tool on the study subjects?

With regard to this evaluation, only Kubernetes manifests are relevant for determining the project size of study objects. Including all files of the project would distort the results because non Kubernetes resources are not analyzed and cannot generate findings. Thus, they have no meaningfulness for this evaluation and are not considered for the project size.
The metric Number of Manifests is not suitable to describe the project size. It is up to the developer team whether they decide to create a new manifest for every Kubernetes resource or to include many resources in a single manifest. This can lead to significantly varying manifest sizes so that they cannot be counted equally in the metric to determine the project size. To circumvent this inequality the project size is determined by the metric Lines of Code, which sums up the number of lines in manifests.
The finding density is defined as the total number of findings detected for the study object per 1,000 Lines of Code. The total number of findings only contain the findings detected by the static source code analysis tool as described in Chapter 5.

6.3.2 RQ2: Which checks produce the most findings and which best practice group is violated most often?

In order to answer this research question, the implemented tool analyzes the study objects and the reported findings are assessed on a group-level and a check-level. The best practice groups are the categories introduced in Chapter 4. The most often violated best practice group is the category comprising the most findings. Furthermore, the number of findings detected per check are assessed in order to identify the 3 checks which produce the most findings.
The checks implemented as part of this thesis cover exactly one best practice each. Most

of Kube-Score's checks also cover one best practice with two exceptions. The check with the identifier "container-resources" covers the best practices Set Requests (Section 4.2.1), Set Limits (Section 4.2.2) and Set Dependent Components (Section 4.2.7). When setting requests or limits, the check distinguishes between the resources memory and CPU. The check with the identifier "container-security-context" covers the best practices Read-only Root File System (Section 4.1.2), Run as High User (Section 4.1.4) and Disable Privileged Setting in Security Context (Section 4.1.5). Kube-Score distinguishes between violations of the Run as High User best practice if the user ID or the group ID equals 10,000 or below.
Kube-Score reports findings for the checks "container-resources" and "container-security-context". In the "comments" section of the checks, details about the exact violations are given (see Subsection 5.3.3). For the "container-resources" check the comments can contain messages of the types "Memory request not specified", "CPU request not specified", "Memory limit not specified", "CPU limit not specified" or "No containers defined". For the "container-security-context" check the comments can contain messages of the types "Container has no security context", "Writable root filesystem", "Container is privileged", "UserID above 10,000 is recommended" or "GroupID above 10,000 is recommended". In order to make all checks comparable, each check has to cover the same amount of best practices. The checks "container-resources" and "container-security-context" are split by their comment types to create checks that only cover one best practice. The types "Memory request not specified" and "CPU request not specified" are combined to cover Set Requests and "Memory limit not specified" and "CPU limit not specified" for Set Limits respectively. "UserID above 10,000 is recommended" and "GroupID above 10,000 is recommended" are added to cover the best practice Run as High User. All violations reported by the "container-security-context" check with the comment "Container has no security context" are added to the Run as High User best practice because it is the only best practice which is violated by the default settings if no custom security context is defined.
Another important factor taken into consideration are the errors occurring prior to the analysis. There are two types of possible errors. The first type comprises errors reported by Helm during the process of locally rendering the chart. Such an error would mean that neither Kube-Score nor the implemented checks in this thesis would be executed for the chart. The second type covers errors that occur while parsing the content of static Kubernetes manifests. The manifest content is parsed per Kubernetes resource. A failure would prevent the execution of the checks implemented in this thesis for the parsed Kubernetes resource but not the entire manifest.

6.3.3 RQ3: What false-positive rate needs to be expected from our tool? Which checks produce the most false-positives?

For every check that is determined as one of the 3 most findings producing check for a study object as described in the study design of RQ2, the false-positive rate is evaluated. 10 findings per check are picked randomly and examined regarding validity. Developers of the study objects Alpha and Beta are included in the examination process. They contribute expertise in working with Kubernetes and comprehensive knowledge about their projects.
The false-positive rate is determined for every check. The overall false-positive rate of the implemented tool is estimated by averaging the results of all checks. Further, the average false-positive rates are given for checks categorized as errors and warnings respectively.

6.3.4 RQ4: How did large projects evolve over time with regard to the number of findings and size?

Requirements for the study objects to be suitable candidates for this research question are project coherence and longevity. The first requirement eliminates the study objects Helm Stable and Bitnami as suitable candidates because they are platforms to share and retrieve charts rather than coherent projects. The second requirement eliminates the study objects Alpha and Karch because the analyzed Kubernetes manifests have been developed over a relatively short period of time. Due to the results of RQ2 described in Subsection 6.5.2 that 83% of the charts fail to render locally the Beta study object is rejected as candidate in this research question.
The Zalando study object is a coherent project evolving since October 2016. The analysis period for this research question is from February 1st, 2017 until September 30th, 2020 because the analyzed directory has been created on January 25th, 2017. The number of findings and the project size represented by the number of lines of code are measured over time. The findings density is defined as the number of findings per 1,000 lines of code. It is determined for the whole project, as well as for every best practice category as defined in Chapter 4.

6.4 Study Objects

The evaluation is conducted on six study objects. The study objects comprise four open source projects published on GitHub, as well as two closed source projects. The most important criterion for the selection of study objects is project size with regard to Kubernetes manifests. A balanced ratio between study objects using Helm charts and

study objects only using static Kubernetes manifests is another important factor for the selection process. The characteristic information of each study object is summarized in Table 6.1.

6.4.1 Helm Stable

Helm itself provides an open source repository which serves as a platform for everybody to share their Helm charts publicly [27]. These charts contain popular software applications which can be deployed with Kubernetes out of the box. The repository was created in October 2015 and 13,014 commits were made until September 27th, 2020. It includes a directory called "stable" containing the latest stable version of the shared charts. This directory is analyzed as a study object for this evaluation. It contains 282 charts consisting of 3,415 Kubernetes manifests and configuration files with 267,546 lines of code.

6.4.2 Bitnami

Bitnami is a company offering application packaging in order to enable customers to quickly deploy software on any platform. Similar to Helm, it provides an open source repository with ready to launch Helm charts [4]. The repository was created in March 2016 and comprises 6,459 commits until September 27th, 2020. It contains 75 charts consisting of 1,205 Kubernetes manifests and configuration files with 158,643 lines of code.

6.4.3 Zalando

In October 2016, Zalando published an open-source repository called "Kubernetes on AWS" on GitHub. It contains custom configuration templates for the specific Zalando use case to provision Kubernetes on Amazon Web Services [52]. The configuration templates are all static Kubernetes manifests but some are modified by a templating software assigning placeholders to fields. These placeholders must be replaced by actual values before deployment. The project description already tells us that the configuration templates are not in a ready to launch state because of missing values. It also explains that the configuration depends on four components. Some of them are developed in-house at Zalando and not available as open source software at this moment [9]. One component this project depends on is the Cluster Lifecycle Manager (CLM) developed by Zalando. The CLM's main purpose is polling the Cluster Registry and updating the cluster to the desired state [28]. The desired state is described in Kubernetes manifests which are stored in repositories like the "Kubernetes on AWS" repository.

Furthermore, the CLM contains the templating component which is responsible for filling the templates with actual values [34]. Executing this templating software of CLM and having access to all values would be mandatory to analyze all Kubernetes manifests of this project. But even when manifests with parameterized values are excluded this project remains the biggest collection of solely static Kubernetes manifests discovered on GitHub during the search of study objects for this thesis.
The repository's default branch is "dev" and its cluster/manifests directory which contains the Kubernetes manifests is analyzed as the study object for this evaluation. From the creation date until September 27th, 2020 2,357 commits were made on "dev". The study object comprises 190 Kubernetes manifests and configuration files with 17,993 lines of code. Excluding files with parameterized values 110 Kubernetes manifests and configuration files with 9,139 lines of code remain.

6.4.4 Karch

Karch is an open source project which simplifies sharing Kubernetes cluster topologies [33]. It was created in August 2017 and comprises 80 commits until June 17th, 2019. The project contains a test cluster with static Kubernetes manifests located at aws/test/k8s. These 13 Kubernetes manifests with a total of 2,440 lines of code form the study object analyzed in this evaluation. Despite the fact that it is only a test cluster which was created over a short period of time and is not actively worked on anymore, it is still the second biggest collection of only static Kubernetes manifests discovered on GitHub during the search of study objects for this thesis.

6.4.5 Alpha

Alpha is the name given to a closed source project within the scope of this thesis. It is a project of an industry partner from the reinsurance domain. Alpha is a company internal project which uses Kubernetes for orchestration and deployment. The project was started in May 2019 and 1,367 commits were made until September 27th, 2020. At the point of evaluation Alpha consists of 464 Kubernetes manifests and configuration files with 31,755 lines of code. The project contains 2 Helm charts.

6.4.6 Beta

Within the scope of this thesis Beta is the name for a closed source project owned by an industry partner from the reinsurance domain. In Beta, Kubernetes is used for orchestration and deployment. The project was created in May 2018 and 1,154 commits were made until September 27th, 2020. At the point of evaluation it has 184 Kubernetes

manifests and configuration files with 6,246 lines of code. Beta includes 29 Helm charts representing the largest part of the Kubernetes related resources within the project.

Table 6.1: The characteristics of the study objects.

Name	Creation Date	Commits	Lines of Code	Helm Charts	Open Source
Helm Stable	October 2015	13,014	267,546	282	✓
Bitnami	March 2016	6,459	158,643	75	✓
Zalando	October 2016	2,357	9,139	0	✓
Karch	August 2017	80	2,440	0	✓
Alpha	May 2019	1,367	31,755	2	✗
Beta	May 2018	1,154	6,246	29	✗

6.5 Results

This section presents the results of this evaluation for each research question.

6.5.1 RQ1: What finding densities can be observed when running our tool on the study subjects?

The values for the Lines of Code metric are presented in Table 6.1. The number of findings and the findings densities for each study object are presented in Table 6.2. The findings density for half of the study objects is in the range of 19.59 and 24.4. High findings densities are detected for Karch (47.54) and especially Alpha (66.76). The lowest findings density has Bitnami with 11.95. The average findings density considering the results of all study objects is 31.87 findings per 1,000 lines of code.

Table 6.2: The Number of Findings and the Findings Density for each Study Object.

Project	Findings	Findings Density
Helm Stable	5,240	19.59
Bitnami	1,895	11.95
Zalando	223	24.4
Karch	116	47.54
Alpha	2,120	66.76
Beta	131	20.97

6.5.2 RQ2: Which checks produce the most findings and which best practice group is violated most often?

Table 6.3 presents the checks that produce the most findings. Table 6.4 displays the number of errors and findings per best practice group for each study object, whereas Table 6.5 summarizes the number of findings for each best practice group.
Zalando, Karch and Alpha have 0 errors and Bitnami has 1 error. A lot of errors occurred during the analysis of Beta (24) and Helm Stable (134). Use Labels is the check producing the most findings for one half of the study objects. The other half has Set Limits as the most findings producing check, followed by Set Requests on the second place. Nine different checks made it among the 3 checks which produce the most findings for at least one study object. Seven out of these 9 checks report findings if the Kubernetes default settings are used. The most violated best practice group is Security for 5 out of the 6 study objects. Only Zalando has a different best practice group being violated the most with 100 findings for the Structure category.

6.5.3 RQ3: What false-positive rate needs to be expected from our tool? Which checks produce the most false-positives?

Table 6.6 summarizes the results for this research question. Nine different checks have been determined among the 3 most findings producing checks for all study objects. For the checks covering the best practices Set Requests (Section 4.2.1), Set Limits (Section 4.2.2), Use Labels (Section 4.4.1), Set Termination Grace Period (Section 4.3.6), Stable API Version (Section 4.1.1) and Disable Automatic Mount of Service Account Token (Section 4.1.10) the determined false-positive rate is 0%. The checks for Do Not Set Namespace to Default in Manifests (Section 4.4.3) and Require Network Policies (Section 4.1.15) have a false-positive rate of 100%. The result for the Run as High User (Section 4.1.4) check's false-positive rate is 20%.
The overall estimated false-positive rate for the implemented tool is 24.44%. The false-positive rate for errors is 24% and for warnings 25%.

6.5.4 RQ4: How did large projects evolve over time with regard to the number of findings and size?

The results are visualized in two charts. Figure 6.1 displays the metrics Lines of Code, Number of Findings and Findings Density for the Zalando project. Figure 6.2 shows the findings densities for the best practice categories Security, Resource Management, Availability and Structure.
The project size increased from 1,138 lines of code in February 2017 to 11,565 lines of code on September 18th, 2020 before it dropped down and ended at 9,159 lines of

Table 6.3: The 3 Checks producing the most Findings for each Study Object.

Project	Most Findings Producing Check		
	1	2	3
Helm Stable	Set Limits 821	Set Requests 735	Run as High User 565
Bitnami	Set Limits 268	Set Requests 220	DNSNtDiM[1] 195
Zalando	Use Labels 100	DAMoSAT[2] 34	Set Termination Grace Period 13
Karch	Use Labels 31	DAMoSAT[2] 18	Stable API Version 11
Alpha	Use Labels 358	Require Network Policies 234	Run as High User 205
Bet	Set Limits 18	Set Requests 18	Run as High User 15

[1] Abbreviation for Do Not Set Namespace to Default in Manifests
[2] Abbreviation for Disable Automatic Mount of Service Account Token

Table 6.4: The Number of Errors and Findings per Best Practice Category for each Study Object.

Project	Error[1]	Category			
		Security	Resource Management	Availability	Structure
Helm Stable	134	2,272	1,665	794	375
Bitnami	1	824	535	288	217
Zalando	0	83	21	14	100
Karch	0	60	12	13	31
Alpha	0	781	481	498	358
Beta	24	51	37	10	9

[1] The Errors produced during the Helm local rendering process or while parsing the Yaml Kubernetes resources

Table 6.5: The Number of Findings reported for each Best Practice Category.

Category	Findings
Security	4,071
Resource Management	2,751
Availability	1,617
Structure	1,090

Table 6.6: The false-positive rates for the checks producing the most findings, the false-positive rates for error findings, warning findings and an estimation for the entire tool.

Group	False-Positive Rate
Use Labels	0%
Set Limits	0%
Set Requests	0%
Run as High User	20%
Stable API Version	0%
Require Network Policies	100%
Set Termination Grace Period	0%
Disable Automatic Mount of Service Account Token	0%
Do Not Set Namespace to Default in Manifests	100%
Error Findings	24%
Warning Findings	25%
Implemented Tool	24.44%

code. The number of findings started with 129 and increased until it reached its peak with 322 findings on December 2nd, 2019. Then it decreases and is at 219 findings on September 30th, 2020. The findings density for the project decreases almost continuously disregarding minor spikes. It started with a value of 116.2 and ended with 23.9 findings per 1,000 lines of code.
The findings density values for the categories Security (66.3 to 6.8), Resource Management (28.1 to 1.8) and Availability (14.1 to 1.1) also decreased consistently over time. On November 22nd, 2017 was a drop from 19.7 to 9.2 in the findings density metric for the Resource Management category. The findings density for the Structure category stayed in the range of 5.1 and 9.5 from February 1st, 2017 until November 29th, 2018. Then it was between 14.9 and 26.3 until August 19th, 2020 before it dropped back down and is at 10.6 on September 30th, 2020.

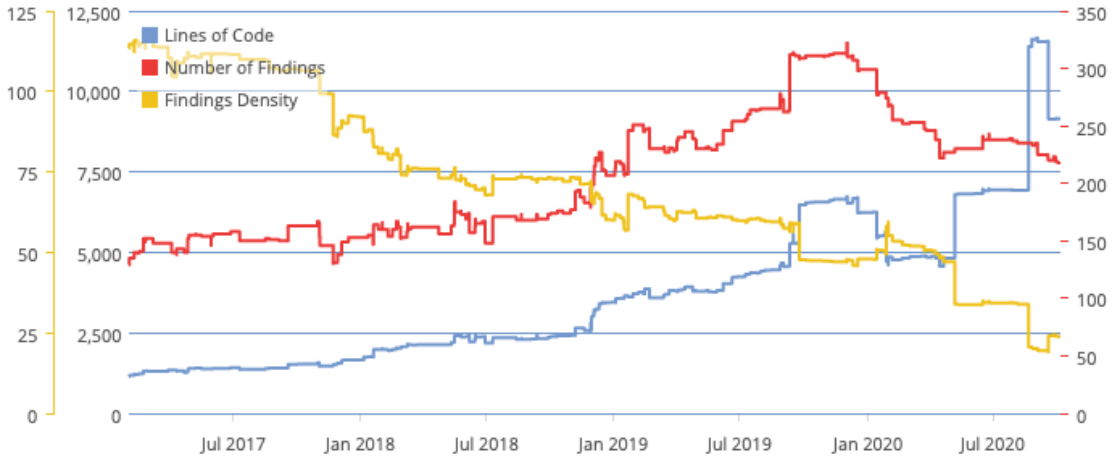

Figure 6.1: The findings trend for the study object Zalando

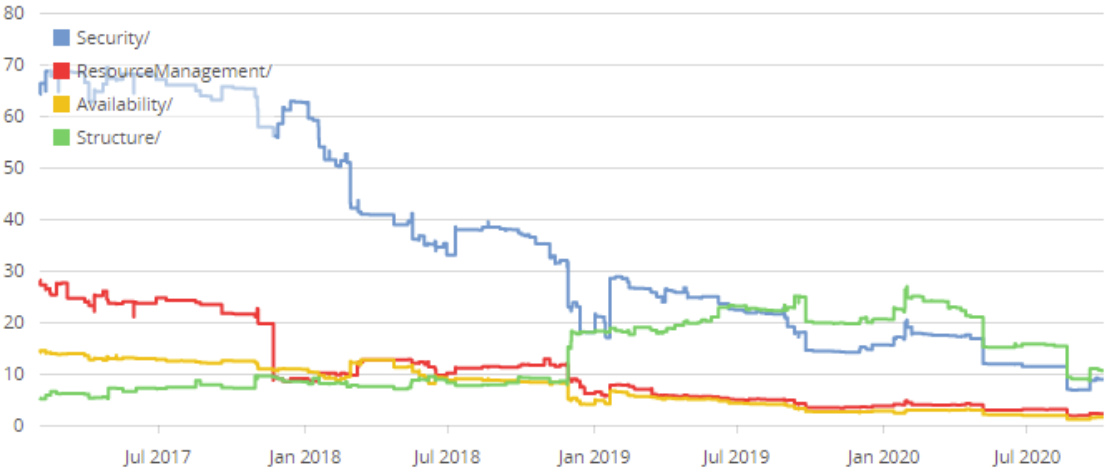

Figure 6.2: Findings densities for the best practice categories as defined in Chapter 4

6.6 Discussion

In this section the results are discussed. For every research question the results are interpreted, comparisons are made among the study objects and the results are put into context if necessary.

6.6.1 RQ1: What finding densities can be observed when running our tool on the study subjects?

The results show that the Alpha study object has the significantly highest findings density. One fact regarding the study object is that Alpha is a relatively young project. This could support the hypothesis made in Subsection 6.1.4 that the findings density is higher at the beginning of a project and decreases for long-living projects.
The findings density for study object Karch is also above average. An explanation for this result is that the analyzed files represent a test cluster for test purposes only. Given these circumstances it might be justifiable to disregard some best practices.
The other study objects are below average and especially the Bitnami study object has a low findings density of 11.95 indicating a high quality of the Kubernetes resources. An average of 31.87 findings per 1,000 Lines of Code shows that the tool is able to provide the developer a significant amount of feedback regarding the analyzed Kubernetes configuration.

6.6.2 RQ2: Which checks produce the most findings and which best practice group is violated most often?

In this section the results of RQ2 are interpreted. First the errors are discussed followed by the most violated best practice groups. Then the checks producing the most findings are discussed.
The expected amount of errors occurring during the execution of the prototype is zero. That is the case for 50% of the study objects. During the local rendering process of 75 Helm charts of the Bitnami study object one error occurred. This amount is tolerable as it indicates a misconfiguration in the chart and not a limitation in the prototype. The Helm Stable study object containing 282 charts produced 134 errors. Out of these errors, 124 are deprecation warnings and these warnings do not cause the local rendering process to fail. Thus, they do not hinder the analysis afterwards. The remaining errors are due to missing values. That is a best practice violation as defined in Subsection 4.4.4. However, it can be explained by Helm Stable's special situation. Helm Stable is not the typical project which uses Kubernetes to orchestrate and deploy one or multiple applications. Instead, it is a platform to publicly share Helm charts which can be used

as templates for other projects. These templates need customization when added to the actual project. It does not make sense to specify a default value for a password while it is only meant as a template on a public platform. In the contrary, leaving out some values which are required but are project dependent can help the end user to notice these values. A surprising result is that local rendering failed for 24 out of 29 Helm charts in the Beta project. The reason is that the 24 charts depend on multiple files to provide values for the local rendering process. It would require a modified Helm command specifying the extra files. The prototype is only capable to render charts with the basic structure. A solution to analyze these Helm charts would be to render the charts up front and provide the resulting static Kubernetes manifest as input for the prototype.
The hypothesis that the Security best practice is the least violated best practice group did not hold. In the contrary, it is the most violated best practice group for 5 out of 6 study objects. This result can be explained by the number of checks implemented per best practice group. From the Security category 11 best practices are covered whereas the second largest category Resource Management has 6 best practices being checked. The related work introduced in Chapter 3 shows that research has been done especially regarding security vulnerabilities and best practices in the Kubernetes context. This research seems to have an effect on the number of defined best practices where the Security category exceeds the other categories. But it apparently did not cause developer teams to pay special attention to the security best practices for Kubernetes so that they integrate mechanisms to adhere to them.
The study objects Helm Stable and Bitnami affect the special circumstance that they are a platform providing Helm charts as templates and that they are not actual projects being deployed themselves. For both projects the checks for Set Limits 4.2.2 and Set Request 4.2.1 produced the most findings. An explanation for this result could be that the chart creators neglect this configuration part based on the fact that they do not know the resource capabilities of the cluster the chart is deployed on in the end. But they do know the minimal amount of resources required for their application which is specified in requests. They can also specify an appropriate limit in order to adhere to best practices and serve the end user as a guideline. The result that the best practice Do Not Set Namespace to Default in Manifests 4.4.3 is violated so often for the Bitnami project can be explained by the project's special circumstance as well. For Helm charts it is good practice to specify the namespace in all template files and assign it a placeholder. In the value.yaml file, the actual namespace name should be provided. In Bitnami's case where the charts function as templates, the actual namespace name the chart will be deployed to is unknown. At this point it is best practice to specify a default value in order to enable a successful local rendering of the chart, but at the same time it violates the best practice Do Not Set Namespace to Default in Manifests

under the given special circumstances.
The best practices Set Limits and Set Requests show a strong correlation. For 3 out of the 6 study objects they are the two most violated best practices with nearly the same amount of findings being reported for each of them. It indicates that either both are defined correctly or none are defined at all.
Another interesting result is that the check for Use Labels 4.4.1 produced the most findings for the study objects Zalando, Karch and Alpha. For the other 3 study objects the check is never among the 3 checks producing the most findings. Adhering to the best practice is a single conceptual decision which affects the entire project. The results show that if the developer team of a project does not adhere to the best practice, the check produces a high amount of findings because Labels must be defined for every type of Kubernetes resources.
Nine checks made it among the 3 checks which produce the most findings for at least one study object. Seven out of these 9 checks report findings if the Kubernetes default settings are used. From these numbers, it can be deduced that best practices where the developer has to actively specify some setting in order to adhere to it tend to be violated more often. Possible explanations for this observation are that developers do not know about the best practices they are violating with the default setting or they have forgotten to specify this part of the configuration. If they specify a setting which violates a best practice, it is less likely that they will revise their configuration because they have concisely chosen this option.

6.6.3 RQ3: What false-positive rate needs to be expected from our tool? Which checks produce the most false-positives?

The false-positive rate of 100% for the checks Do Not Set Namespace to Default in Manifests 4.4.3 and Require Network Policies 4.1.15 are surprising results. One would assume that they indicate that the checks are not working, but the result is caused by special circumstances of the study objects where the analyzed files have been reported. For the Do Not Set Namespace to Default in Manifests check the explanation for the high false-positive rate is given in Subsection 6.6.2. The fact that Bitnami is not a project being deployed but only providing Helm charts as templates makes it good practice to specify the default namespace as default value. The cause for the 100% false-positive rate for findings reported by the Require Network Policies check in the Alpha project is the following. The Alpha developer explains that the external software NeuVector is used to manage full life cycle container security. In NeuVector, policies can be defined to control communication in the cluster making the need for network policies obsolete. In both cases, special circumstances make it impossible to detect true-positives for the corresponding check. In these situations, the check should be disabled for the project.

The remaining 2 false-positives detected for findings produced by the Run as High User 4.1.4 check are due to the use of external software as well. Alpha uses Kustomize which is a tool to modify Kubernetes manifests. The base file is a Kubernetes manifest shared by multiple parties. Each party can create a patch file containing modifications for the base file. It enables the party to customize the base file without forking. The false-positives have been detected in patch files. These patch files are not complete Kubernetes manifests but only contain the modifications. The final base files created after applying the modifications specified in the patch files are configured correctly and thus, the reported findings are false-positives.
The estimated false-positive rate for the implemented tool of 24.44% is higher than assumed. The false-positive rate for error findings is smaller than the rate for warning findings, but the difference between them is less than expected. If the false-positives produced by the checks for the best practices Do Not Set Namespace to Default in Manifests and Require Network Policies are excluded, the overall false-positive rate for the implemented tool would be 2.86%. The large discrepancy between the false-positive rates once including all detected false-positives and once excluding false-positives, where true-positives are impossible, demonstrate the importance to configure the implemented tool correctly adapting to special circumstances of the analyzed project. It further emphasizes the insufficiency of the sample size evaluated for this research question. Consequently, the estimated false-positive rates for the entire tool and for error findings and warning findings respectively are not meaningful.

6.6.4 RQ4: How did large projects evolve over time with regard to the number of findings and size?

The hypothesis that the findings density increases over time holds for the analyzed study project Zalando. It consistently decreased over the time of development. The same development is observed in the categories Security, Resource Management and Availability. It shows that the implemented tool should be used especially in the early stages of a project. It helps the developer to detect weaknesses early on and making it effortlessly easier to adhere to best practices from the beginning.
The hypothesis that the findings density for the Security category correlates less like the overall findings density is rejected. The Security findings density decreases over time as well and differs compared to the other categories only in the way that the findings density is higher. This supports the conclusion described in 6.6.2 that existing research led to more defined best practices resulting in more detected findings and a higher findings density. But the fact that neglecting these best practices means to be more vulnerable for attacks did not strengthen the importance of the security best practices over availability or resource management related best practices. Developer teams do

not focus more on security best practices as anticipated.
Structure's finding density differs from the ones of the other categories. The findings density stays on the same level within a small range for a long period of time. One finding in the Structure category was that the default namespace is used. The rest are findings reporting that a Kubernetes resource has no labels assigned. That indicates that the policy which Kubernetes resources to label stayed the same. The larger changes in the findings density are caused by adding or removing large chunks of code. It demonstrates that the labels and the labeling guideline were not improved during the time of the analysis. The changes in the findings density were only side effects of other actions.

6.7 Threats to Validity

This sections depicts the threats to validity for this thesis. They are split in the two groups internal validity and external validity. Internal validity describes threats to validity regarding the prototype and the evaluation concept. External validity comprises external factors threatening the validity of this evaluation regarding generalization.

6.7.1 Internal Validity

One threat to internal validity is the potential of undiscovered defects in the implementation of the source code analysis tool. Despite the extensive testing of the prototype, it cannot be guaranteed that the documented limitations (see Section 5.4) are the only ones in the implementation.
Another threat to internal validity is the choice of the study design. There may be better suitable procedures to execute the evaluation. The number of checks and the number of findings per check being analyzed to determine the false-positive rate is too small. There is not sufficient data to derive the overall false-positive rate of the implemented tool.
Measures were taken to prevent the misinterpretation of results. Experts reviewed the findings regarding correctness. However, it remains possible that some interpretation of results may be wrong.

6.7.2 External Validity

One threat to external validity is the insufficient size of the study objects. The choice of study objects is a threat to external validity as well because there are special circumstances regarding the study objects which need to be considered. Helm Stable and Bitnami are projects where different developer teams can share their Helm charts. The

charts are templates with some parts being generalized. They are not intended to be deployed in this version. Alpha and Beta rely on external software and this factor is disregarded by the source code analysis tool. Zalando uses unpublished software to parameterize values in their Kubernetes manifests. These special circumstances could have an unforeseen influence on the evaluation results and due the small number of study object the impact could be significant. The findings of RQ4 cannot be generalized because the study was conducted on a single study object.
Another threat to external validity is the proneness to human error regarding RQ3. The findings validity was checked manually by the author of this thesis and Kubernetes developers of the projects Alpha and Beta. They answered to the best of their knowledge, but some assessments may be incorrect.

7 Future Work

In this thesis, a static source code analysis tool was implemented. It analyzes Kubernetes manifests and Helm charts in order to detect best practice violations. The limitations (see Section 5.4) and the threats to validity (see Section 6.7) depict the weaknesses of the implemented tool and the evaluation of it. Accordingly, we provide suggestions for future work to improve upon this thesis.
The implemented tool can be extended by identifying more Kubernetes related best practices and implementing checks which detect if they are violated. Another way to improve the implemented tool is to refine on the method to identify relevant source files for the analysis. Currently, the basic search finds all Kubernetes manifests but it also accepts some non Kubernetes files and the algorithm can be more specific and go in to more detail to automatically select Kubernetes manifests more accurately. The third option to extend the implemented tool is making it recognize the use of popular external software and react accordingly. The integration of Helm can be used as an example, so that a similar approach can be used for the integration of other external software like Kustomize. Taking Kustomize as an example, the tool should be able to detect patch files, locate the targeted base file, execute Kustomize to create the merged Kubernetes manifest and analyze it to detect best practice violations in this final version.
The evaluation conducted in this thesis should be replicated in a larger scale on many diverse study objects. The metrics used for the evaluation could be optimized, for example by replacing the metric Lines of Code for the metric Source Lines of Code. This would remove whitespace and comments when determining the project size. Whitespace and comment lines cannot cause any findings being reported by the implemented tool and thus, they are irrelevant for this evaluation. They should be excluded in order to minimize the gap of differences in development that some developer teams use a lot of whitespace and comments for structure and organization while others do not. This results in a more fine-grained and accurate calculation of the findings density which is an important factor in this evaluation. For replication studies of this work, the study design could be improved by adding a review process for the assessment of false-positives. Shamim, Bhuiyan, and Rahman [43] use a methodology which includes a manual assessment, followed by a verification step to mitigate the first author's bias. This approach could be integrated similarly in this evaluation reducing the threat to external validity that the evaluation results are prone to human error.

One interesting follow-up research improving upon this thesis would be to conduct a long-term study where developer teams actively use the implemented tool in order to assess and improve their Kubernetes project. Spillner [44] uses a similar methodology to evaluate the false-positive rate and the developer's acceptance of the tool HelmQA which recommends quality improvements for Helm charts. The developers should be integrated in the study. They provide comprehensive knowledge about their projects and should be responsible to configure the implemented tool correctly. This configuration includes aspects like disabling irrelevant checks for the project due to external software being used. During the evaluation period the developers could flag every finding they looked at either as false-positive or as accepted true-positive. This approach could lead to a sufficient amount of data to determine a meaningful false-positive rate. Interesting research questions with the described study design would be evaluating the effect of the tool regarding Kubernetes quality improvement over a period of time and determining the false-positive rates for the tool, individual checks and best practice groups.

8 Conclusion

In this thesis, 34 best practices regarding security, resource management, availability and structure of Kubernetes manifests were elaborated on. A static source code analysis tool was implemented. It detects violations of 26 out of the 34 presented best practices (see Table 5.1). The tool analyzes Kubernetes manifests and Helm charts and reports findings categorized as error or warning.
As a first step, the implemented tool identifies all Kubernetes manifests and Helm charts within the project. Then, it executes Helm to locally render the charts to static Kubernetes manifests. Kube-Score and the checks implemented in this thesis analyze all Kubernetes manifests and report findings if best practices are violated. A finding describes the issue, specifies the location where it occurred and gives recommendation how to fix the best practice violation.
The implemented tool reported on average 31.87 findings per 1,000 Lines of Code. It shows that the tool is able to provide the developer a significant amount of feedback and recommendations regarding the analyzed Kubernetes configuration. Security was the most violated best practice group for 5 of the 6 study objects. An explanation for this result is the fact that it is the largest best practice group comprising 11 best practices being checked by the tool. The check producing the most findings for one half of the study objects was Use Labels (see Subsection 4.4.1) because the violation of this conceptual best practice leads to a high amount of findings. Set Limits (see Subsection 4.2.2) and on the second place Set Requests (see Subsection 4.2.1) were the most findings producing checks for the other half of the study objects indicating a high correlation between these two best practices. The local rendering failed for 24 out of the 29 Helm charts of the Beta study object. The reason is that the charts depend on multiple files providing values for the parameterized fields in the templates, but this special behavior is not supported by the tool. The tool only executes the basic Helm command to render the charts locally and it expects it to work as an adherence to the best practice described in Subsection 4.4.4. The determined false-positive rate for the tool was 24.44% if the tool is not configured according to the analyzed project's special circumstances. 20 of the 22 identified false-positives were reported by checks which could not produce true-positive findings because of the project's special circumstances. If these false-positives are excluded, then the implemented tool would have a false-positive rate of 2.86%. The sample size the false-positive rates are based on was too

small in order to give meaningful results. Monitoring the evolution of a study object showed that the findings density decreases over time. The implemented tool is the most valuable in the early stages of a project where it detects the highest findings density. It lets the developers detect the violations effortlessly and supports them to adhere to best practices from the beginning.

In sum, the implemented static source code analysis tool successfully analyzes static Kubernetes manifests and detects best practice violations in practice especially in the security context. The usage of external software can limit the effectiveness of the tool. The correct configuration of the tool is crucial in order to minimize errors during the execution and false-positives among the reported findings. The configuration includes disabling checks which are obsolete due to special circumstances of the project like using external software. It may also comprise the preparation of the Kubernetes data, so that the implemented tool receives only input it can handle which are static Kubernetes manifests and Helm charts.

List of Figures

List of Tables

Bibliography

[1] Mohamed Ahmed. *Kubernetes and Containers Best Practices - Health Probes*. July 21, 2019. URL: https://www.magalix.com/blog/kubernetes-and-containers-best-practices-health-probes. (accessed: September 2nd 2020).

[2] Gabriel Avner. *Kubernetes Pod Security Policy Best Practices*. Apr. 2, 2019. URL: https://resources.whitesourcesoftware.com/blog-whitesource/kubernetes-pod-security-policy. (accessed: August 28th 2020).

[3] Alexandru G Bardas. "Static code analysis." In: *Journal of Information Systems & Operations Management* 4.2 (2010), pp. 99–107.

[4] Bitnami. *The Bitnami Library for Kubernetes*. URL: https://github.com/bitnami/charts/tree/master/bitnami. (accessed: August 15th 2020).

[5] Anita Buehrle. *Top 5 Kubernetes Best Practices From Sandeep Dinesh (Google)*. Mar. 6, 2018. URL: https://www.weave.works/blog/kubernetes-best-practices. (accessed: August 28th 2020).

[6] Jim Bugwadia. *10 Kubernetes Best Practices You Can Easily Apply to Your Clusters*. Nov. 14, 2019. URL: https://thenewstack.io/10-kubernetes-best-practices-you-can-easily-apply-to-your-clusters/. (accessed: August 31st 2020).

[7] Brendan Burns et al. *Kubernetes Best Practices*. First Edition. Sebastopol: O'Reilly Media, Inc., 2019. ISBN: 9781492056478.

[8] Derek Carr, Mike Danese, and Tim Hockin. *Namespaces*. July 17, 2020. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. (accessed: August 28th 2020).

[9] Alex Contini. *Case Study: zalando. Europe's Leading Online Fashion Platform Gets Radical with Cloud Native*. URL: https://kubernetes.io/case-studies/zalando/. (accessed: August 28th 2020).

[10] Mike Danese. *Configuration Best Practices*. July 17, 2020. URL: https://kubernetes.io/docs/concepts/configuration/overview/. (accessed: August 28th 2020).

[11] Mike Danese. *Labels and Selectors*. Aug. 5, 2020. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/. (accessed: August 28th 2020).

[12] Sandeep Dinesh. *Kubernetes best practices: Organizing with Namespaces*. Apr. 27, 2018. URL: https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-organizing-with-namespaces. (accessed: August 28th 2020).

[13] Sandeep Dinesh. *Kubernetes best practices: Resource requests and limits*. May 11, 2018. URL: https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits. (accessed: August 28th 2020).

[14] Sandeep Dinesh. *Kubernetes best practices: Setting up health checks with readiness and liveness probes*. May 4, 2018. URL: https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-setting-up-health-checks-with-readiness-and-liveness-probes. (accessed: August 28th 2020).

[15] Sandeep Dinesh. *Kubernetes best practices: terminating with grace*. May 18, 2018. URL: https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-terminating-with-grace. (accessed: August 28th 2020).

[16] Devin Donnelly, Shuang Wang, and Tim Bannister. *Understanding Kubernetes Objects*. July 27, 2020. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. (accessed: October 13th 2020).

[17] Matt Farina. *Recommended Labels*. July 10, 2020. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/. (accessed: August 31st 2020).

[18] Cloud Native Computing Foundation. *CNCF Survey 2019*. 2019.

[19] Connor Gilbert. *9 Kubernetes Security Best Practices Everyone Must Follow*. Jan. 14, 2019. URL: https://www.cncf.io/blog/2019/01/14/9-kubernetes-security-best-practices-everyone-must-follow/. (accessed: August 28th 2020).

[20] Github. *The State of the Octoverse*. 2019.

[21] Nils Göde et al. *Qualität in Echtzeit mit Teamscale*. Universität Siegen, 2014.

[22] Google. *Deployment*. Sept. 28, 2020. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/deployment. (accessed: October 13th 2020).

[23] Brian Grant, Daniel Smith, and Tim Hockin. *Kubernetes Deprecation Policy*. Aug. 30, 2020. URL: https://kubernetes.io/docs/reference/using-api/deprecation-policy/. (accessed: August 31st 2020).

[24] Hasham Haider. *9 Best Practices and Examples for Working with Kubernetes Labels*. Sept. 28, 2018. URL: https://www.replex.io/blog/9-best-practices-and-examples-for-working-with-kubernetes-labels. (accessed: August 28th 2020).

[25] Hasham Haider. *Kubernetes Production Readiness and Best Practices Checklist*. replex, 2019.

[26] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. "Teamscale: Software Quality Control in Real-time." In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India, 2014, pp. 592–595. ISBN: 978-1-4503-2768-8.

[27] Helm. *Helm Charts*. URL: https://github.com/helm/charts. (accessed: August 15th 2020).

[28] Henning Jacobs, Muaaz Saleem, and Sandor Szücs. *Running Kubernetes in Production*. URL: https://kubernetes-on-aws.readthedocs.io/en/latest/admin-guide/kubernetes-in-production.html. (accessed: August 28th 2020).

[29] Amir Jerbi and Michael Cherny. *Security Best Practices for Kubernetes Deployment*. Aug. 31, 2016. URL: https://kubernetes.io/blog/2016/08/security-best-practices-kubernetes-deployment/. (accessed: August 28th 2020).

[30] Ajmal Kohgadai. *Docker Container Security 101: Risks and 33 Best Practices*. Sept. 13, 2019. URL: https://www.stackrox.com/post/2019/09/docker-security-101/. (accessed: August 28th 2020).

[31] Kubernetes. *Kubernetes*. URL: https://github.com/kubernetes/kubernetes. (accessed: October 14th 2020).

[32] *Kubernetes API*. Version 1.18.0. Kubernetes. Apr. 13, 2020.

[33] Etienne Lafarge. *karch - A terraform module to spawn Kubernetes clusters - test*. URL: https://github.com/elafarge/karch/tree/master/aws/test. (accessed: August 15th 2020).

[34] Mikkel Larsen, ed. *Continuously Deliver your Kubernetes Infrastructure*. KubeCon. (May 2–4, 2018). Copenhagen, Denmark. URL: https://kccnceu18.sched.com/event/Dque/continuously-deliver-your-kubernetes-infrastructure-mikkel-larsen-zalando-se-advanced-skill-level-slides-attached.

[35] Andrew Martin. *11 Ways (Not) to Get Hacked*. July 18, 2018. URL: https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/. (accessed: August 28th 2020).

[36] Rory Mccune and Liz Rice. *CIS Kubernetes Benchmark*. CIS Benchmarks, 2020.

[37] Daniele Polencic. *Kubernetes production best practices. A curated checklist of best practices designed to help you release to production*. URL: https://learnk8s.io/production-best-practices#application-development. (accessed: August 28th 2020).

[38] Liz Rice, ed. *Running with Scissors*. KubeCon. (May 2–4, 2018). Copenhagen, Denmark. URL: https://kccnceu18.sched.com/event/EMyr/keynote-running-with-scissors-liz-rice-technology-evangelist-aqua-security-slides-attached.

[39] Liz Rice and Michael Hausenblas. *Kubernetes Security*. First Edition. Sebastopol: O'Reilly Media, Inc., 2018. ISBN: 9781492039068.

[40] Gigi Sayfan. *Mastering Kubernetes*. Birmingham: Packt Publishing Ltd., 2017. ISBN: 978-1-78646-100-1.

[41] Rémy-Christophe Schermesser. *8 Algolia-Tested Best Practices for Kubernetes*. June 13, 2019. URL: https://blog.algolia.com/8-algolia-tested-best-practices-kubernetes/. (accessed: August 28th 2020).

[42] Sebastian Schwegler. "Security Best-Practices in Kubernetes und Evaluation eines geeigneten Security-Scanners." Bachelor's Thesis. Duale Hochschule Baden-Württemberg Karlsruhe, 2019.

[43] Md. Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. *XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices*. 2020.

[44] Josef Spillner. *Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications*. 2019.

[45] StackRox. *The State of Container and Kubernetes Security*. 2020.

[46] Thomas Stringer. *Kubernetes' AlwaysPullImages Admission Control - the Importance, Implementation, and Security Vulnerability in its Absence*. 2018. URL: https://trstringer.com/kubernetes-alwayspullimages/. (accessed: August 28th 2020).

[47] Vladislav Supalov. *What's Wrong With The Docker :latest Tag?* URL: https://vsupalov.com/docker-latest-tag/. (accessed: August 31st 2020).

[48] Eric Tune, David Eads, and Jordan Liggitt. *Using RBAC Authorization*. July 23, 2020. URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/. (accessed: August 31st 2020).

[49] Author Unknown. *CONTAINERS[] .SECURITYCONTEXT .RUNASUSER > 10000*. URL: https://kubesec.io/basics/containers-securitycontext-runasuser/. (accessed: August 28th 2020).

[50] Gustav Westling. *Kube-Score*. Version 1.8.1. Aug. 11, 2020. URL: https://github.com/zegl/kube-score.

[51] Gustav Westling, Andre Hilsendeger, and Jörg Lenhard. *README_CHECKS*. Aug. 5, 2020. URL: https://github.com/zegl/kube-score/blob/master/README_CHECKS.md. (accessed: August 28th 2020).

[52] Zalando. *Kubernetes on AWS*. URL: https://github.com/zalando-incubator/kubernetes-on-aws. (accessed: August 15th 2020).

[53] Fiorella Zampetti et al. "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines." In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. Buenos Aires, Argentina, 2017, pp. 334–344. ISBN: 9781538615447.