Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks

Ahmed Zerouali
Ahmed.Zerouali@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Ruben Opdebeeck
Ruben.Denzel.Opdebeeck@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Coen De Roover
Coen.De.Roover@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

***Abstract*—Using Kubernetes for the deployment, management and scaling of containerized applications has become a common practice. To facilitate the installation and management of these applications, practitioners can use the Helm package manager to assemble their configuration files into charts. The latter are reusable packages of pre-configured Kubernetes resources that can be deployed as a unit. In this paper, we aim to support chart developers and users by carrying out a comprehensive study on publicly available charts. For 9,482 charts that are distributed via the Artifact Hub repository, we mine and collect the list of their metadata, versions, dependencies, maintainers and container images. Then, we carry out an empirical analysis to assess the state and evolution of charts, as well as the outdatedness and security risks of their images. We found that the ecosystem forming around Helm charts is growing fast. However, most of the charts are not official with no popularity and no license. We also observed that charts tend to release multiple versions, but around half of them are still in the initial development phase. When looking at the container images used in charts, we found that around half of them are outdated and 88.1% of them are exposed to vulnerabilities, jeopardizing 93.7% of the charts.**

***Index Terms*—Kubernetes, Helm, Software Ecosystem, Infrastructure-as-code, Evolution, Security**

I. Introduction

Over the last decade, containerisation has become a common practice in deploying software. Containers provide a lightweight solution to provisioning multiple software systems on a single host. Each system runs in an isolated container, which includes all dependencies, binaries and configuration files required by the system. Varying the number of container instances at run time facilitates realising elasticity for contemporary software architectures.

The most popular containerisation technology is Docker. It is often used in tandem with Kubernetes, the most popular container management and orchestration tool [1]. According to the 2022 Stack Overflow Developer Survey [2], both Docker and Kubernetes are in first and second place as the surveyed developer's most loved and wanted tools.

Kubernetes enables deploying containerized applications on a distributed cluster consisting of a number of compute nodes [3]. In the spirit of Infrastructure as Code, cluster deployments are entirely managed through declarative configuration files. Each file declares the configuration of the resources needed for the cluster such as the containers, pods, deployments, secrets, service and volumes.

Specifying all configurations and verifying their correctness can be cumbersome, especially for complex setups of large clusters. To ease the Kubernetes configuration burden, the Cloud Native Computing Foundation (CNCF) released Helm [1] as a package manager for Kubernetes applications in 2015. Helm enables assembling Kubernetes configuration files into packages that can be shared through local and remote repositories for reuse. These reusable packages are called *charts*. It is possible to install, update or uninstall a chart with a single command, abstracting away the underlying control plane interactions. Properly versioned, charts evolve over time. According to a 2020 survey of practitioners by the CNCF [4], Helm is the most popular tool for packaging Kubernetes applications and is used by 63% of respondents.

In this paper, we collect and analyse the charts made publicly available through the Artifact Hub repository [2]. The latter is considered as the largest repository of Helm charts. Our goal is to obtain insights about the current state of the ecosystem that is forming around Helm charts, and its evolution over time. In particular, we look at the characteristics and growth of the charts, as well as the outdatedness and security of the container images used in them. Existing studies have already looked at container images that are hosted on Docker Hub [5], but did not look yet at the clients in which these images are used, such as Kubernetes applications. Inspired by the work of Wittern et al. [6], this work presents a comprehensive study that focuses on six aspects of the ecosystem of Helm charts:

- A_1 **Chart growth trends:** We study how the ecosystem around Helm charts has arisen over time in terms of growth in the number of charts, their repositories and their versions.
- A_2 **Chart reusability characteristics:** We investigate the reusability characteristics of charts, including their popularity, license, and types (i.e., application or library, verified or official, etc). We also look at the number of maintainers of each chart, as well as whether those maintainers are primarily individuals or organisations. This provides us insights about the risk of the ecosystem and individual charts being abandoned.

[1] https://helm.sh/

[2] Artifact Hub is a Cloud Native Computing Foundation sandbox project https://www.cncf.io/projects/artifact-hub/

- A_3 **Chart dependencies:** We investigate how often Helm charts include others as dependencies, and study the practices that govern dependency version selection. This provides us insights about how interconnected the ecosystem is.
- A_4 **Deployed container images:** In addition to Docker Hub, new registries from GitHub, Google and Quay are emerging with thousands of images hosted on them. For each chart, we identify the container images it manages and present descriptive statistics about their number and registry of origin. We also look at the tags that are used to specify these images in the configuration files.
- A_5 **Outdatedness of deployed container images:** Container images coming from registries like Docker Hub can contain outdated packages that lack more recently introduced bug fixes and features. We track the images used in Helm charts back to their source registries and then identify whether they are outdated or not. We carry out this study and evaluate the outdatedness of chart images at two different time points: 1) at the analysis date (17-10-2022); and 2) at the date of the chart's latest release. The latter will give us insights about how careful chart maintainers are about the images they use when releasing their charts, while the former will give us a snapshot of the state of the ecosystem at the analysis date, i.e., we assess the quality of used images as if the charts were deployed on the analysis date.
- A_6 **Security of deployed container images:** Charts deploying older container images might unwittingly deploy outdated packages with known security vulnerabilities. We therefore scan the images used in charts and identify the known vulnerabilities to which the chart is exposing the application. We study their number, type, severity and how they could be avoided.

The insights gained through this study will be useful to the practitioners including the community managers, the contributors, and the users of Helm charts.

II. Background

Helm charts are a packaged collection of files comprising all of the resources required to manage and deploy an application on a Kubernetes cluster. Charts can be stored in a repository, and can be easily installed, upgraded, and uninstalled using the Helm package manager. The files packaged by a chart are typically organised as follows:

- `Chart.yaml`: a file that contains metadata about the chart, such as its name, version and description.
- `values.yaml`: a file that contains default values for the configuration parameters of the chart, including the image repository and image tag that should be used for the application's pods.
- `templates/`: a directory that contains one or more Kubernetes manifest files that define the resources that should be deployed. Templates in this folder may be combined with the `values.yaml` file.
- `charts/`: a directory that contains the charts upon which the chart depends.

In addition to these required files and directories, a chart may include optional files such as a Readme, License and other configuration values files. When a user installs a chart, they have the option to override the default values specified in the `values.yaml` or any of the other values files. This makes it easy for users to customise the configuration of their application without having to modify the chart itself.

The Helm tool itself provides a way to version and manage the installed charts. It is, for instance, possible to upgrade to a newer version or to rollback to a previous version of a chart.

Helm charts can be distributed through any code repository. Artifact Hub[3] forms a centralised index, enabling users to search for charts across repositories, to browse charts by category, and to inspect basic information about charts such as their README and version history. Users can also host a repository for their Helm charts on Artifact Hub itself.

The Helm tool supports interacting with Artifact Hub through commands such as "`helm repo add`" which adds a chart repository to the list of locally known repositories or "`helm search`" to find a particular chart. Once a repository has been added, users can use the "`helm install`" command to install charts from that repository.

III. Related Work

Our empirical analysis is inspired by the work of Wittern et al. [6], who were the first to conduct a comprehensive study into the inter-package dynamics of the then-emerging npm ecosystem, including JavaScript package descriptions, dependencies, download metrics, and usage in publicly available GitHub repositories. They discovered that the number of npm packages and their updates are increasing at a super-linear rate. Additionally, they found that packages are becoming more and more dependent on each other. In a similar vein, our work represents the first comprehensive study on the emerging ecosystem of Helm charts.

Ibrahim et al. [7] carried out a study on 4,103 open-source Github projects that use Docker Compose. The latter is a tool for defining and running multi-container applications on a single host using a simple YAML file that contains configuration details (e.g., services, networks, and volumes) for each container. On the other hand, Kubernetes is a more robust and advanced platform for container orchestration, it allows for automatic management and scaling of containerized applications in a clustered environment, across multiple hosts. The goal of the study was to understand how Docker Compose is used in the wild. The authors found that over a quarter (26.8%) of the considered projects use Docker Compose to build their applications from a single image. They also found that Docker Compose is used with basic options instead of advanced ones, and that users tend to use earlier versions of this tool. Docker Compose and Kubernetes are both tools

[3]https://artifacthub.io

for managing containerized applications, and they share some similarities in terms of functionality and concepts.

Baur [8] collected and compared the different tools available for packaging Kubernetes applications including Helm, Kustomize and ytt. A recent survey of 1,164 organisations [9] found that up to 30% of low-performing organizations apply changes manually to their Kubernetes configurations, while a higher percentage of top-performing organizations use a packaging solution like Helm or Kustomize (performance was computed based on best practices and industry standards that each organisation has). They also found that 70.67% of the respondents consider Kubernetes security to be very important. Bose et al. [10] quantified the frequency at which security defects appear in Kubernetes manifests. Shamim et al. [11] analysed 104 internet blog posts and video presentations that discuss Kubernetes security. They identified 11 security practices for Kubernetes, including vulnerability scanning and continuous updates. Our study goes beyond the work by quantitatively analysing the outdatedness and security of images used in Helm charts of Kubernetes applications.

In a previous work [5], we analysed the relation between outdated system packages in Debian-based Docker images, their associated security vulnerabilities, and bugs. We quantified the gap between the outdated packages and their most recent versions using the technical lag metric [12], [13]. We found that even the latest versions of Debian-based container images published on Docker Hub feature packages with known vulnerabilities and bugs. Our current study is different since we focus on the container images used within Helm charts. As charts are intended as packages of reusable Kubernetes deployments, one can reasonably assume their container images to be scrutinised further by the chart developers. Moreover, the images stem from additional registries such as GitHub and Quay, next to Docker Hub.

IV. Data Extraction

We analyse the ecosystem forming around reusable Kubernetes packages distributed as Helm charts through Artifact Hub. To this end, we collect information about their number, their versions, their dependencies, and the container images they deploy on the Kubernetes cluster. Then, we analyze image outdatedness and Common Vulnerabilities and Exposures (CVE) affecting the system and third-party packages installed therein. Various sources need to be mined to this end.

A. Helm chart corpus

First, we query the *Artifact Hub* repository index to retrieve its list of publicly available Helm charts. Using the index's API, we extract the published versions of each chart and the version control repository hosting its latest version.

Using this link, we download the latest version of each Helm chart and analyse its `Chart.yaml` file to extract the chart's maintainers and dependencies. Listing 1 provides an example of the relevant information stored in such a file. The result is a corpus of 9,482 Helm charts hosted in 1,921 *Artifact Hub* repositories as of October 17^{th}, 2022.

Listing 1: Excerpt of an example `Chart.yaml` file

```
apiVersion: v2 # The chart's API version
name: MyChart # The chart's name
version: 1.0.0 # A SemVer compatible version for the chart
appVersion: 3.0.0 # The version of the upstream application
kubeVersion: ^1.8.0-0 # A range of Kubernetes versions
description: This is my chart # A brief description of the chart
type: application # The chart's type
home: https :// example.com/ # The chart's homepage
sources: https :// example.com/ # The chart's repository or source
dependencies: # List of dependencies
- name: nginx
  version: 1.2.3
maintainers: # List of maintainers
- name: maintainer1
  email: maintainer1@email
```

B. Deployed container images

In order to analyse the container images deployed by each Helm chart, we "render" each chart template (i.e., instantiate its template by substituting configuration values for template variables) using Helm commands (`helm template` [4]) and extract the names, tags and repositories of the images referenced in the chart. For each discovered container image, we trace it back to its source registry (e.g., Docker Hub, Quay) and query the registry's API for the image's version tags and their release dates.

To find the list of vulnerabilities affecting the images retrieved from each chart, we rely on Artifact Hub's security scanner which is built on top of Trivy [5], a popular open source tool that scans container images for security issues.

C. Vulnerabilities affecting deployed container images

For each identified vulnerability, we use its CVE [6] number to trace it back to the National Vulnerability Database [7] and extract its CVSS score. The latter is a metric that captures the severity of a vulnerability as an integer between 0-10. In addition, for each CVE, we look for its corresponding Common Weakness Enumeration (CWE) [8]. CWE is a standard way used to classify and describe vulnerabilities (CVE).

V. Statistical Analysis

In this section, we use the dataset collected in Section IV to analyse the dynamics of the Helm ecosystem in terms of the aspects introduced in the introduction. We limit ourselves to the concrete results of the analysis here, and discuss their implications in Section VI.

To enable reproducibility of our work, we provide a replication package with the code and the data used for the analysis [9].

[4] https://helm.sh/docs/helm/helm_template/

[5] https://github.com/aquasecurity/trivy

[6] Common Vulnerabilities and Exposures is a popular numbering system used to identify publicly acknowledged vulnerabilities; https://cve.mitre.org.

[7] https://nvd.nist.gov/

[8] https://cwe.mitre.org/

[9] https://doi.org/10.5281/zenodo.7552697

A_1 *Chart growth trends*

The first Helm chart was created in October 2016. Helm charts and repositories were initially hosted by Google, before being moved to Artifact Hub in November 2020. The latter contains more than 9,482 charts maintained by different organizations and users. In this part of the study, we investigate the evolution of this ecosystem since its creation. We look at the number of new charts, versions, and repositories registered on Artifact Hub over the observation period.

Figure 1 shows the evolution over time of the cumulative number of Helm repositories, charts and their versions. The Y-axis is in a log scale, revealing a large difference between the number of repositories, charts, and versions. We observe that the cumulative number for each artifact is increasing exponentially over time, confirming that the Helm ecosystem is growing. This is statistically confirmed by a regression analysis using linear and exponential growth models. Table I summarizes the R^2 values reflecting the fit.

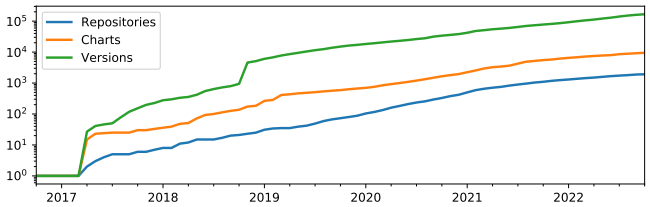

Fig. 1: Evolution of the cumulative number of Helm repositories, charts and their versions available in Artifact Hub.

TABLE I: R^2-values of regression analysis on the evolution of the number of Helm repositories, charts and versions.

	Repositories	Charts	Versions
Linear	0.72	0.73	0.76
Exponential	**0.98**	**0.92**	**0.85**

As explained before, a Helm repository contains different charts and each chart has many versions. We found that each repository has an average and median of 1 and 4.9 charts, respectively. Similarly, we found that each chart has an average and median of 17.6 and 4 versions, respectively. We also computed the number of versions released between the first and the latest chart version. We found that charts release an average and median of 27.6 and 5 versions per month. This difference between the mean and median is due to the age of charts since more than 80% of them are less than 2 years old and have less than 11.7 months between their first and latest versions.

As a versioning scheme, Helm requires version numbers of charts to be compliant with the semantic versioning v2.0.0 standard (SemVer) [10]. A SemVer-compatible release uses a version number composed of a *major*, *minor* and *patch* number. An optional pre-release label can also be added to the version numbers as follows *Major.Minor.Patch-pre*. This format enables ordering releases and indicates the importance of each new release. We found that 4% of the chart releases are major (i.e., a change happened to the major version number), 16.5% are minor (i.e., a change happened to the minor version number but not to the major number), 77.5% are patch (i.e., a change happened to the patch version number only), and 2% are pre-releases having the format *X.Y.Z-pre* instead of *X.Y.Z*. We also found that on average, Helm charts released 0.8 major releases, 3.6 minor releases and 17.3 patch releases, while the median is 0 major, 1 minor and 4 patches. This means that more than half of them (i.e., 72.1% to be precise) has not changed their major version number since their creation. In fact, we found that 48.3% of the charts are still in major version 0 (*0.Y.Z*). This means that the artifacts are still in their initial development. This is the case for 36.8% of the verified charts and 56.2% of the unverified ones.

Summary: The ecosystem of Helm charts is growing. The number of charts, their repositories, and their versions are exponentially increasing over time. Helm charts released 27.6 versions on average with a median of 5. More than 80% of the charts have been created in the last two years. According to SemVer specifications, 48.3% of the charts are still in the initial development.

A_2 *Chart reusability characteristics*

1) Chart types: Helm charts are classified into two types, application and library charts. The former are the standard charts describing a packaged Kubernetes deployment for a specific application, while the latter define charts intended for inclusion in other Helm charts. In our dataset, we found that 53% of the charts are declared as applications, 46.4% do not have any classification, and only 0.59% (56) are explicitly declared as library charts in the `Chart.yaml` manifest. Note that the unclassified charts might in reality be either.

We also found that only 1.3% (120) are official charts published by the owners of the deployed application, e.g., the `prometheus` chart [11] is coming from the Prometheus project. Furthermore, 59% (5,599) of the charts are verified, i.e., published on Artifact Hub by the actual owner of the git repository hosting it. Note that this owner is not necessarily the developer of the deployed application, e.g., the `redis`, `mysql`, `postgresql` and `wordpress` charts are all packaged and published by Bitnami.

We also found that official charts are the oldest, meaning that the first Helm charts distributed online were coming from official organisations.

Summary: Most of the charts within the ecosystem are declared of the application type, with only a tiny proportion being of the library type. Merely 1.3% of the charts are official, while 59% of all charts are verified.

2) Chart licenses: We investigated the licenses for each chart and found that 91.9% of the charts do not include any license file. As published code is under exclusive copyright by default, not including a license with a chart might cause

[10] SemVer 2: https://semver.org/spec/v2.0.0.html

[11] https://artifacthub.io/packages/helm/prometheus-community/prometheus

license compatibility problems when they are deployed[12]. For the other charts, we found that 5.2% of them have an *Apache-2.0* license and 2.3% have an *MIT* license, while the rest (0.6%) have other licenses. This will be discussed further in Section VI.

Summary: Only 8.1% of the charts have a license. Helm charts tend to use permissive licenses.

3) Chart popularity: Few charts are truly popular. We found that 81.2% of charts has not received any star, 91.9% does not have any subscribers and 97.1% does not have any production users. The latter is a measure of how many users are deploying a particular chart in production environments. Looking at an aggregated popularity metric, $popularity = stars + subscribers + production\ users$, we found that 79.6% of the charts have zero popularity. Only 2.5% of the charts have a popularity higher than or equal to 10, while the maximum popularity we found is 608.

Summary: Few charts are popular. Only 20.4% has at least one star, subscriber, or a production user.

4) Chart maintainers: Despite its young age, the ecosystem around Helm charts is growing. Helm being awarded the graduated status by Cloud Native Computing Foundation (CNCF) shows that it is becoming more popular amongst Kubernetes users. Thus, these users should be aware of the risks brought about by the use of shared charts. For instance, a used chart might have already been abandoned or be at risk of being abandoned by their maintainers. Using charts with a low number of (key) maintainers (i.e., bus factor) can also be risky in this light. Abandoned charts no longer benefit from bug fixes and vulnerability patches, and may configure Kubernetes or the application they deploy on a Kubernetes cluster in a suboptimal or out of date manner.

For each chart in our dataset, we parsed the *Chart.yaml* file from its latest version and extracted the list of maintainers. We found that 64.1% (6,078) of the charts coming from 52.7% of the repositories (1,012) specify the username or email of at least one maintainer in this manifest. We also observed that 53.1% of the charts that do declare their maintainers are verified.

Maintainer information is important and should always be included in the manifest so that users can report bugs to or request bug fixes from a responsible person or organization. Focusing only on the subset of charts with declared maintainers, we found that they have on average (and median of) a single maintainer. The highest number of maintainers per chart that we found is 9. Based on the username, we found a unique number of 1,750 maintainers in total for the whole set of charts. It is worth mentioning that maintainer information declared in the charts does not always refer to a person but in many cases to an organization or team.

Next, we computed the number of charts per maintainer and repository. The average number of charts that each maintainer is responsible for is 4.4, the median is 1, while the maximum is 968 charts maintained by a single organisation. This skew means that there is a subset of maintainers that is responsible for most of the charts. Indeed, we found that 30.4% of the maintainers (532) are responsible for more than 80% of the charts. We also looked at the number of repositories per chart. We found that on average, maintainers are responsible for charts originating from 1.7 repositories, while more than half of the maintainers contribute to charts coming from one single repository. The maximum is 63 repositories for a single maintainer. Table II shows more information about the distribution of the number of maintainers per chart and repository.

TABLE II: Distribution of the number of maintainers per (/) chart and repository, and vice versa.

Distribution	Mean	Min	Median	Max
# maintainers / chart	1.3	1	1	9
# maintainers / repository	2.9	1	1	233
# charts / maintainer	4.4	1	1	968
# repositories / maintainer	1.7	1	1	63

Summary: The majority (64.1%) of charts includes the name or email of their maintainers in their `Chart.yaml` file. Charts have a mean and median of one maintainer. Conversely, more than half of the maintainers is responsible for a single chart. On average, each maintainer is responsible for 4.4 charts. Organizations like AppsCode, TrueCharts and Bitnami maintain the highest numbers of charts.

A_3 *Chart dependencies*

It is possible for charts to depend on other charts. A chart's dependencies should be listed in the `Chart.yaml` file as mentioned in Section IV and put in the `charts/` directory. Each dependency declared in this manifest should be either in a local or remote repository so the Helm tool can pull and copy it to the `charts/` directory; otherwise, it should be placed manually in this directory. By specifying versions in `Chart.yaml`, developers can restrict the releases of the charts on which they depend. Version specifications take the form of a single SemVer version or of a constraint resulting in a range of versions. Constraints are resolved by the Helm tool when a chart is deployed, downloading the latest available release satisfying the dependency constraint. Examples of these constraints include the "*" wildcard, comparison symbols (e.g., $>= 1.2.x$), tilde ($\sim$ 1.2.3), caret ($\hat{}$1.2.3), etc.[13]

Relying on many dependencies may bring about a high monitoring effort as the upstream of each needs to be monitored for new releases and needs to be configured correctly. In our dataset, we found a total number of 4,761 dependencies used by only 3,004 charts (31.7%). The mean and median number of dependencies used by Helm charts are 1.6 and 1, respectively. The maximum is 22 dependencies used by two unverified charts. The maximum number found for verified charts is 18, while the maximum for official charts is 4.

[12]https://choosealicense.com/no-permission/

[13]See https://helm.sh/docs/chart_best_practices/dependencies/ for details.

Looking at the constraints used for declaring chart dependencies, we found that 62.3% of the dependencies are used with a strict version (e.g., 1.2.3), while the rest is declared with a constraint (e.g., $>= 0.1.0$, $1.x.x$, etc).

Summary: Only 31.7% of the charts have dependencies on other charts. The median number of dependencies that charts have is one. Chart dependencies tend to be used with strict version constraints, i.e., only 37.7% of the dependencies were specified with a constraint resolving to a version range.

A_4 *Deployed container images*

Helm charts package files required for the control plane of a Kubernetes cluster, and declare the container images to be deployed thereon. We investigate which images are the most frequently used, and look at the source registry hosting them. We expect most of the images to stem from Docker Hub, which is with over 10 million images the most popular registry for Docker.

After rendering the charts (i.e., substituting configuration values for template variables) in our dataset using the `Helm template` command, we found a total number of 12,743 (non-unique) images used in 63.4% (6,014) of the charts. Note that this command does not render configuration values used for testing, including image names. Thus, we only rendered images used for actual chart deployments. Moreover, several charts could not be rendered because they require or miss some configuration values that are to be provided locally. Additionally, although it is not common, it is possible for charts to exist without including images. Such charts might include configuration files that are reused by other charts. For the remainder of the study, we focus on the charts that could be rendered and that have at least one image.

We found that the median number of images used in each chart is 1, while the average is 2.12. The maximum number of images that were used in one chart is 181 (coming from a verified chart). These numbers decrease to 1, 2.1 and 51, respectively, when only focusing on unverified charts, showing that verified charts have more images than unverified ones. Next, we looked at the most popular images since one image can be used by many charts. Indeed, we found that all used images are actually corresponding to 8,935 unique images, some of them being used in hundreds of charts. Table III shows the top 10 images used in Helm charts.

TABLE III: Top 10 container images used by Helm charts stored in Artifact Hub.

Container image	Registry	# Charts
truecharts/multi-init	GitHub	769
busybox	Docker Hub	267
bitnami/postgresql	Docker Hub	265
mysql	Docker Hub	160
nginx	Docker Hub	142
bitnami/redis	Docker Hub	123
postgres	Docker Hub	96
dellcloud/pages	Docker Hub	85
alpine	Docker Hub	78
truecharts/postgresql	GitHub	69

Looking at the origin of these images, we found that they come from different registries, some of them private. Table IV shows the top 5 container image registries used by Helm charts. Perhaps surprisingly, Docker Hub is only the source for 60.5% of all used images. The second-most popular registry is GitHub with 16.7% of all images, and the third-most is Quay with 6.1% of the images. Note that many images stemming from GitHub were originally hosted on TrueCharts, and only later moved to the TrueCharts repository on GitHub as packages [14]. In fact, 8.9% of the images declared in the charts still use the old location of TrueCharts [15]. All charts using the old location are still active with at least one version update in the past year.

TABLE IV: Top 5 container image registries used by Helm charts stored in Artifact Hub.

Registry	# Container images	Cumulative % images
Docker Hub	5,406	60.5
GitHub	1,496	77.2
Quay	541	83.3
Google Container Registry	301	86.7
Azure Container Registry	166	88.5

Dockerfiles typically create a new container image from an existing one using the `FROM` instruction, e.g., `FROM debian:latest`. For this command, it is recommended to always use a specific tag such as `FROM debian:bullseye-20221205` instead of the `latest` or a blank tag. Not pinning the image version down is considered a code smell [14], and flagged as such by Dockerfile linters such as Hadolint (i.e., rules DL3006 and DL3007). In a similar vein, images used in Kubernetes clusters should be specified in the configuration manifests with a fixed tag. Looking at our dataset, we found that 8.1% of the images are declared with the `latest` tag and 3.2% are declared without any tag. However, we only found 17.3% of the charts participating in this bad practice. When differentiating between verified and unverified charts, we found that verified charts have a higher tendency to include images with fixed tags, i.e., 92.5% of the images used in verified charts are declared with a fixed tag compared to 86.2% for unverified charts.

Summary: More than 63% of the charts deploy container images. Charts deploy a mean and median of 2.12 and 1 images, respectively. 60.5% of the images used in charts are hosted on Docker Hub. 88.7% of the images used by charts are declared with a pinned tag or version. Only 17.3% of the charts make use of images with unpinned tag.

A_5 *Outdatedness of deployed container images*

Usually, container images used in Helm charts and coming from repositories registered within container registries contain applications with the dependencies, binaries, and configuration files needed to run them. As these applications evolve, their images evolve too. A container image of an application can

[14]https://github.com/orgs/truecharts/packages

[15]For example, *tccr.io/truecharts/multi-init* instead of *ghcr.io/truecharts/multi-init*. Both links work.

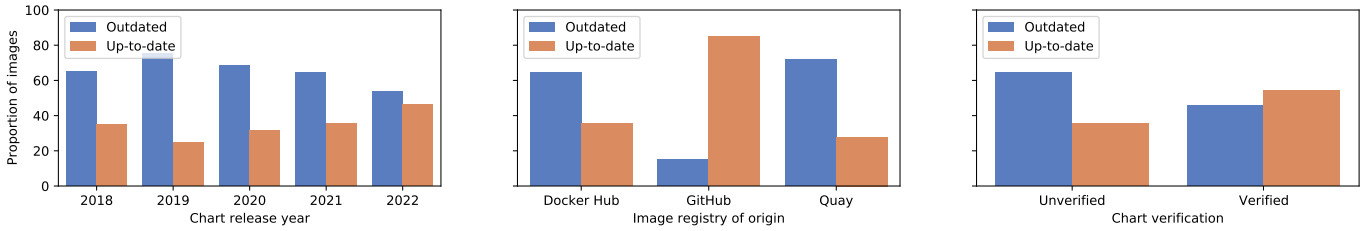

Fig. 2: Proportion of outdated and up-to-date container images grouped by their charts' release year, container registry and chart verification, at the analysis date. Each group constitutes 100% of the images belonging to it.

have different versions with multiple tags. Each image version can correspond to a specific version of the application. For example, `node:19.4.0-alpine3.17` [16] is an image that includes the version *19.4.0* of *Node.js* and it is based on the base image of *Alpine v3.17*. Users are in general encouraged to deploy up-to-date images for an application. We therefore expect Helm charts to follow this best practice too.

We investigated the outdatedness of container images used by Helm charts by computing the difference between an image's release date and the release date of the latest available image in the container registry. For this analysis, we only focus on the container images stemming from Docker Hub, GitHub and Quay. As mentioned before in Table IV, images from these registries represent 83.3% of all unique images referenced in Helm charts. Including images that are used in multiple charts, these registries offer 85.3% of the used images.

We found that 61.5% of the used images is released in 2022, 19.7% is released in 2021, while the rest (18.8%) is released before 2021. We also observed that images from Quay and Docker Hub are relatively older than GitHub images. These observations mean that some outdated images might be in use. To quantify this outdateness, we carry out the analysis at two different time points: 1) at the analysis date (i.e., 17-10-2022); and 2) at the release date of the charts.

Summary: More than 61.5% of the images used in charts were released in the last year. Images coming from GitHub are more recent.

a) ***Outdatedness at the analysis date****:* We first compute the outdatedness of images as if they are going to be deployed now (at the analysis date). Thus, we simply retrieve the latest available image version from the container registry and compare it to the used image. We start by exploring how many images within Helm charts are up-to-date, i.e., with the latest available version/tag. Figure 2 shows the proportion of outdated and up-to-date images grouped by their charts' year of release, their registry of origin and the chart's verification. At the first glance, we observe that there are many outdated images in each group. Considering all images without grouping them, we found that 56.6% of them are outdated. As expected, the figure reveals that more recent charts have fewer outdated images. We also observe that images from GitHub are more up-to-date than other images which is due to the fact that the majority of used GitHub images were released in 2022. Additionally, we found that verified charts have more up-to-date images than unverified charts.

To quantify how outdated container images are, we use the time lag metric defined by Gonzalez-Barahona et al. [12], [13]. More specifically, we compute the time lag as the time between the images used in the charts and the latest versions of these images that are available in their corresponding container registries. Focusing only on the 56.6% of the images that are outdated, we found that the median time lag for images used in charts is 218 days while the average is 362 days. Figure 3 shows the time lag incurred by outdated images used in Helm charts. We observe that Docker Hub images have higher lag than other images while GitHub images have the lowest lag values. We also observe that verified charts have lower image lag values than unverified charts.

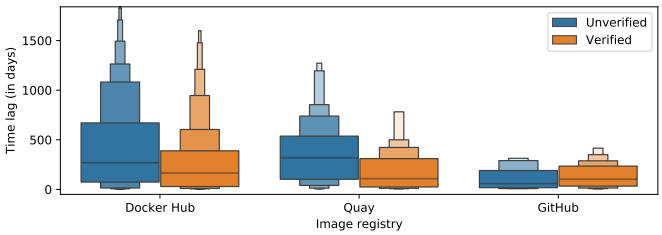

Fig. 3: Time lag of outdated images used by Helm charts, grouped by their registry of origin and chart's verification, at the analysis date.

Summary: 56.6% of the images used in the latest chart versions available at the analysis date are outdated. Verified charts have more up-to-date images and lower time lag. On average, outdated images are one year behind their latest available versions.

b) ***Outdatedness at the charts' release date****:* We now look at the outdatedness of images at the time when their charts were released. For each chart, we identify its list of images and then we retrieve the last version of each image right before the release of the chart. This will give us an idea about the quality of the images that developers include in their Helm charts. Figure 4 shows the proportion of outdated and up-to-date images grouped by their charts' year of release, their registry of origin and the chart's verification, computed at the charts' release date. We observe a difference with Figure 2

[16]https://github.com/nodejs/docker-node/blob/28ad5e0e5d0e80df44d897c9057ffd6419a3c7a5/19/alpine3.17/Dockerfile

in the sense that images are more up-to-date. However, we can still find outdated images. Considering all images without grouping them, we found that 41.3% of them are outdated. We also observe that in recent years, chart developers include more up-to-date images than outdated ones. We also observe that the majority of the images coming from GitHub are up-to-date. As previously observed, verified charts have more up-to-date images than unverified ones. This shows once again that verified images have better quality characteristics. Focusing only on the 41.3% of the images that were outdated at the charts' release date, we find that the median time lag of images used in charts is 168 days while the average is 299 days. Our other observations are similar to those made in Figure 3, e.g., Docker Hub images have the highest lag values.

Summary: Chart developers tend to use up-to-date images, especially in recent years. Overall, at the release date of the charts, 41.3% of the referenced images were outdated with around 10 months of time lag, on average.

A_6 *Security of deployed container images*

Assuring the security of image containers used in Helm charts is essential since vulnerable images can be exploited in attacks on both the deployed application and on the cluster onto which the image is deployed. We therefore analyse whether the Helm chart images are vulnerable in terms of known vulnerabilities of the applications and packages installed therein.

To this end, the Artifact Hub vulnerability scanner managed to scan 93.7% (5,634) of the charts that we already inspected for container images. We found 88.1% (4,965) of these charts to include container images with vulnerable packages. The total number of vulnerabilities affecting these images is 2,532,897, while the number of vulnerability types (i.e., unique CVEs) is 12,812. The total number of container images affected by these vulnerabilities is 6,893. Figure 5 shows the proportion of unique vulnerabilities affecting Helm charts, grouped by their severity, while Figure 6 shows the proportion of all vulnerabilities. We observe that most of the vulnerabilities are of a medium or high severity. We also observe a significant proportion (10.9%) of unique vulnerabilities for which we could not retrieve the severity from NVD (i.e., CVSS is not determined yet by NVD).

Considering all vulnerabilities, we found an average of 510.2 vulnerabilities in each chart, and a median of 204 vulnerabilities. Similarly, the mean and the median number of vulnerabilities we found in each container image is 258.9 and 96, respectively. We also found that 40.8% of all vulnerabilities have already been fixed. Table V shows the top vulnerabilities (CVEs) to which images in Helm charts are exposed. We observe that *CVE-2022-2097* is the most widespread vulnerability affecting 51.67% of the images and 62.46% of the charts. This vulnerability stems from the widely-deployed *openssl* package, but is of a medium severity. In contrast, the *zlib*-affecting *CVE-2022-37434* is of a critical severity and is widely-deployed through Helm charts.

While browsing the homepages of some popular images from Docker Hub, we noticed that Docker Hub warns its users about the *Log4Shell (CVE-2021-44228)* vulnerability in particular. This vulnerability was publicly disclosed on December 2021. Indeed, because of its critical severity, *Log4Shell* should be patched quickly wherever it is deployed. We therefore looked for this particular vulnerability in our dataset, and we found that the majority of the charts are safe, with only 75 charts still deploying images with packages affected by this CVE. However, when we looked for the more recent *Spring4Shell (CVE-2022-22965)* vulnerability, we found 251 charts exposed to it through their container images. This critical vulnerability is a remote code execution vulnerability that affects Spring Framework. It is considered as one of the most exploited security vulnerabilities in 2022 [17].

TABLE V: Top 5 common vulnerabilities and exposures (CVE) exposing container images used in Helm charts.

CVE	CWE ID	% charts	% images
CVE-2022-2097	326	62.46	51.67
CVE-2022-37434	787	61.9	53.78
CVE-2021-39537	787	60.12	46.65
CVE-2022-29458	125	57.74	45.35
CVE-2017-11164	674	57.52	43.19

We retrieved the common weakness enumeration (CWE) for each vulnerability that is deployed by the charts in our dataset. Table VI shows the top CWEs of the vulnerabilities found in this study. We observe that 82.24% of the charts deploying a vulnerability, has at least one vulnerability of CWE type `Out-of-bounds Write`. The latter is considered as the most frequent and dangerous CWE (with the highest number of CVEs) [18]. However, we also observe that `Inadequate Encryption Strength` vulnerabilities are prevalent across images of Helm charts, although this CWE is not in the top 25 list of CWEs. Chart and image maintainers should be aware of the prevalence of unpopular CWEs like this one in their applications.

TABLE VI: Top 5 common weakness enumeration (CWE) affecting container images used in Helm charts.

CWE	% charts	% images
Out-of-bounds Write (787)	83.24	76.48
Out-of-bounds Read (125)	75.43	66.75
Use After Free (416)	75.1	65.4
Inadequate Encryption Strength (326)	72.59	61.7
Uncontrolled Resource Consumption (400)	72.2	59.1

Summary: Around 9 out of 10 charts deploy at least one vulnerability. Images used in charts are affected by up to 12,812 different vulnerabilities (with a unique CVE). 9.3% of the vulnerabilities deployed by charts are of a critical severity. Charts have an average of 510.2 vulnerabilities and a median of 204. Image containers used in charts have an average of 258.9 vulnerabilities and a median of 96.

[17] https://www.dynatrace.com/news/blog/anatomy-of-spring4shell-vulnerability/

[18] https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

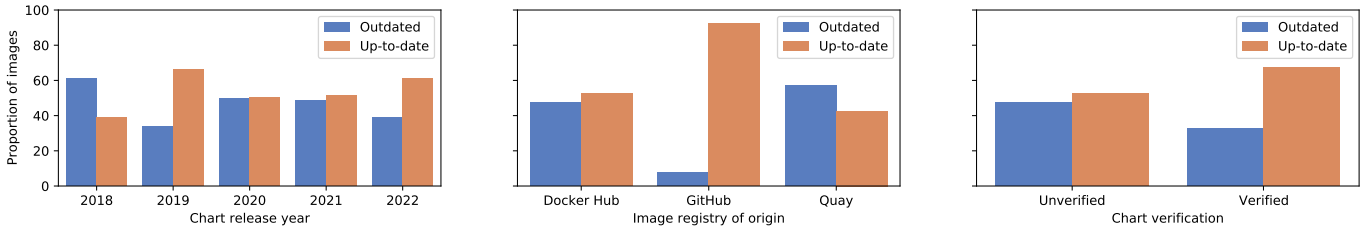

Fig. 4: Proportion of outdated and up-to-date container images grouped by their charts' release year, container registry and chart verification, at the charts' release date. Each group constitutes 100% of the images belonging to it.

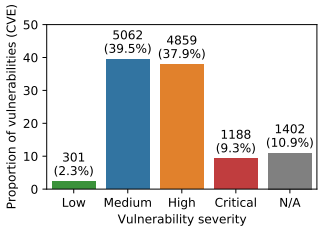

Fig. 5: Proportion of unique vulnerability CVEs affecting Helm charts, grouped by their severity.

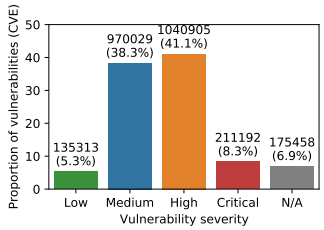

Fig. 6: Proportion of all vulnerabilities affecting Helm charts, grouped by their severity.

VI. Discussion

Helm charts render deploying and configuring applications on a Kubernetes cluster a systematic and repeatable process. Charts can be reused across concrete environments, and an ecosystem has started forming around the Artifact Hub repository for shared charts. Our paper is the first to study this emerging ecosystem in terms of growth, reuse, inter-chart dependencies, the freshness of the container images they deploy, and the known vulnerabilities of packages therein.

In A_1, we found that the ecosystem around Helm charts is growing in an exponential manner. However, nearly half of the charts are still in the initial development phase and should be considered unstable according to the SemVer specification. We also observed that chart releases span all types of version increments (i.e., patch, minor and major). However, the meaning of these version increment types within the context of charts is not yet clear. For Ansible, which is another Infrastructure-as-Code (IaC) technology, Opdebeeck et al. [15], [16] found that developers of Ansible roles (i.e., packages) do not adhere to the same rules when selecting the type of version increment for a release. Future work could consider the same topic for Helm charts.

In A_2, we found that the majority of the charts have zero popularity. This might mean that users do not tend to rate the Helm charts they rely on, or that only a small subset of charts enjoys widespread use. However, it is well-known that software popularity metrics that reflect real-world popularity are far from trivial to devise [6]. A follow-up study could look at the GitHub repository of each chart and extract the popularity metrics of GitHub and compare them to the metrics from Artifact Hub. Nevertheless, such a study should take into account the fact that multiple charts can share the same GitHub repository.

We also found that most of the charts are not official, and therefore provide Kubernetes deployments of applications that belong to third parties. This is different from other Infrastructure-as-Code ecosystems such as Docker Hub where organisations are the ones sharing and taking care of the maintenance of official images for their applications. We believe that organisations that already provide official images in Docker Hub should also provide official Helm charts. This will instil potential chart users more confidence, as official charts are more trustworthy and seen to be well maintained and secure.

In addition, we found that most of the charts do not have a license. However, for the small subset of charts that have one, they tend to use permissive licenses such as *MIT* and *Apache-2.0*. This sample of charts with a license gives us an idea about the permissive climate of the Helm charts ecosystem. The Helm community can learn from other communities like npm and RubyGems and start providing a default permissive license with each chart. This will promote a permissive climate and facilitate the chart reuse. Moreover, it will push the chart maintainers to either accept the default license or provide their own license.

We also observed that charts tend to declare only one maintainer in their `Chart.yaml` file which can be an individual or an organisation. Many organisations maintain several charts for multiple applications while only specifying official email or username. This makes it difficult for users to assess the bus factor of the charts they would want to use. The community should devise a means to quantify the long-term maintenance support that can be expected for such a chart, without endangering the privacy of its maintainers.

In A_3, we found that dependencies tend to be used with strict version constraints. This means that chart developers are less interested in having automatic updates of their dependencies compared to developers in other general-purpose ecosystems, e.g., *RubyGems* [17]. A follow-up study could look at the impact of these constraints on the outdatedness of depended upon Helm charts. However, this finding confirms that different ecosystems have different habits and values, as previously stated by Bogart et al. [18].

In A_4, we found that more than half of the charts only make use of a single image. Charts can use a single image for a Kubernetes cluster and run multiple containers from that

image. Each container on a node in the cluster is running the same image, but has different configuration or environment variables. The usage of a single image seems to be more prevalent in Helm charts compared to Docker-Compose GitHub projects where Ibrahim et al. [7] found that 26.8% of the projects they studied use Docker Compose to build their applications from a single image.

We also found that many charts do not include any image at all. Helm charts provide a way to package and manage Kubernetes resources, and can therefore also be used to deploy resources such as ConfigMaps and Services besides container images. It is also possible for a chart to package no Kubernetes resources at all, and to just consist of files or values intended for reuse by other charts. This seems in contrast with the low number of library charts that we found. Charts that are not deploying any application are considered as libraries. In fact, none of the library charts we have in our dataset make use of container images. Thus, we think that charts should mention their correct type in the `Chart.yaml` file so that users can better differentiate between charts.

In A_4, we also found that 82.7% of images in charts are used with a strictly pinned down version tag. This is a good practice from a safety point of view, but it can also easily lead to outdatedness, as it is not always easy to keep up with the rate at which the used images update their tags. Indeed, in A_5 we found that if we deployed the studied charts at the analysis date (17-10-2022), we would find that 56.6% of the used images are outdated in terms of both versions and time. More importantly, we also observed that images were already outdated at the release date of their charts. However, we believe that researchers can treat time lag in infrastructure ecosystems different from time lag in e.g., development library ecosystems. For Docker Hub, it was observed that different images providing the same application and package versions can be created within a short period of time (e.g., create a slim version of an image after one week) [19].

In A_6, we found that the container images included in charts are affected by hundreds of vulnerabilities, including widespread and critical vulnerabilities like Log4Shell. This means that despite all the work done on the outdatedness and security of software artefacts, more effort is still required. One easy way to reduce this high number of vulnerabilities affecting packages installed by images in Helm charts is to use the available vulnerability fixes. In fact, we found that 40.8% of all vulnerabilities can be avoided by just adopting their available fixes. This should be considered by both image and chart maintainers.

VII. Threats to Validity

Given the empirical nature of our study, its findings are exposed to some potential threats to validity. We present these threats following the classification and recommendations of Wohlin et al. [20].

The main threat to *construct validity* comes from imprecisions in the data sources we used to obtain the list of publicly-available Helm charts. We only relied on the catalog of charts registered on Artifact Hub. We might have missed other charts that are not distributed via Artifact Hub, but are made available through GitHub or other version control services. The Helm tool supports installing such charts as well. However, we think that Artifact Hub contains the most relevant charts that are supposed to be reused. In fact, the Helm homepage refers the users searching for charts to Artifact Hub [19].

Another threat to construct validity stems from the fact that when rendering the chart template using Helm, we could not process all charts. Some of them might require manual input for or substitution of template variables. This means that we might have missed some of the charts that are relevant in the ecosystem. In addition, we relied on the Artifact Hub scanner that is based on Trivy to find the list of vulnerabilities affecting the packages within images used in Helm charts. It might be possible that this tool missed some vulnerabilities. To mitigate this threat, we inspected five images using Snyk's scanner and compared the results. We found that both tools reported the same list of vulnerabilities.

Internal validity concerns choices and factors internal to the study that could influence the observed results. When looking at the outdatedness of images used in Helm charts, we only focused on those coming from Docker Hub, GitHub and Quay registries. While one may argue that this subset is not representative of all images, the selected images represent 85.3% of all images used in the charts. Moreover, these images are the most popular ones (see Table III). We therefore consider the selected set of images to be representative for most of the chart maintainers and users.

Conclusion validity concerns the degree to which the conclusions we derived from our data analysis are reasonable. When extracting the list of maintainers and the type of each chart, we found that many charts do not include this information. It could be possible that some charts are maintained by a large number of maintainers but that they do not specify this in their `Chart.yaml` file. In a similar vein, 46.4% of the charts do not specify their type. This might have led us to draw conclusions about the A_2 that are impacted by the missing data.

As a threat to *external validity*, our findings cannot be generalised beyond Helm charts and the ecosystem forming around them.

VIII. Conclusion

We presented an empirical analysis of the ecosystem forming around the Artifact Hub registry of Helm charts, which package the resources needed to deploy containerized applications on Kubernetes clusters in a repeatable and reusable manner. We investigated the growth, reuse aspects, and dependencies of Helm charts. For the container images used within charts, we quantified image freshness and the known vulnerabilities of the packages within.

Considering only the charts listed in the Artifact Hub repository, we studied 9,482 charts. We observed that the ecosystem

[19] https://helm.sh/docs/helm/helm_search_hub/

of charts is growing in an exponential manner. However, the majority of the charts are not official with no popularity and no declared license. We also found that charts tend to only use few dependencies with strict dependency constraints. When looking at the container images used in these charts, we found that more than half of the charts only make use of one image. However, around half of the images are outdated, at the release time of the charts and at the analysis date. In addition, around 9 out of 10 charts are exposed to vulnerabilities affecting their images. Community managers, developers, and users of Helm charts can use our results and insights to better improve the state of their charts and ecosystem. As future work, we plan to extend our study by detecting potential misconfigurations in and fixes of charts.

ACKNOWLEDGMENTS

This research was partially funded by the "Cybersecurity Initiative Flanders" project and the Research Foundation Flanders (FWO) under Grant No. 1SD4321N.

REFERENCES

[1] Cloud Native. Cloud native annual survey 2021. https://www.cncf.io/reports/cncf-annual-survey-2021/.

[2] Stack Overflow. 2022 stack overflow developer survey. https://survey.stackoverflow.co/2022/.

[3] Kubernetes. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/, 2022.

[4] Cloud Native. Cncf survey 2020. https://www.cncf.io/cncf-cloud-native-survey-2020.

[5] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the relation between outdated Docker containers, severity vulnerabilities, and bugs. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 491–501. IEEE, Feb 2019.

[6] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 351–361, 2016.

[7] Md Hasan Ibrahim, Mohammed Sayagh, and Ahmed E Hassan. A study of how docker compose is used to compose multi-component systems. *Empirical Software Engineering*, 26(6):1–27, 2021.

[8] Andreas Baur. Packaging of kubernetes applications. In *Proceedings of the 2020 OMI Seminars (PROMIS 2020)*, volume 1, pages 1–1. Universität Ulm, 2021.

[9] Humanitec. Kubernetes benchmarking study 2022. https://humanitec.com/whitepapers/kubernetes-benchmarking-study-2022.

[10] Dibyendu Brinto Bose, Akond Rahman, and Shazibul Islam Shamim. 'under-reported'security defects in kubernetes manifests. In *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pages 9–12. IEEE, 2021.

[11] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. In *2020 IEEE Secure Development (SecDev)*, pages 58–64. IEEE, 2020.

[12] Jesus M Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. Technical lag in software compilations: Measuring how outdated a software deployment is. In *IFIP International Conference on Open Source Systems*, pages 182–192, 2017.

[13] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer, 2018.

[14] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, 8:34127–34139, 2020.

[15] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. Does infrastructure as code adhere to semantic versioning? an analysis of ansible role evolution. In *2020 IEEE 20Th international working conference on source code analysis and manipulation (SCAM)*, pages 238–248. IEEE, 2020.

[16] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. On the practice of semantic versioning for Ansible galaxy roles: An empirical study and a change classification model. *J. Syst. Softw.*, 182, December 2021.

[17] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240, 2019.

[18] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2016.

[19] Max de Visser. A look at how often docker images are updated. https://anchore.com/look-often-docker-images-updated/, 2017.

[20] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.