Packaging of Kubernetes Applications

Andreas Baur ✉
Ulm University, 89081 Ulm, Germany

Abstract

Modern distributed applications and services usually consist of multiple containerized components across multiple tiers. To orchestrate these sets of containers, Kubernetes has emerged as the de-facto standard for deploying and managing these containerized applications. While Kubernetes provides many features to make deployment and management of container infrastructures a lot easier, it focuses less on the applications themselves, leaving the challenge of treating applications as an interdependent unit. Therefore, multiple tools and approaches exist that are trying to solve this challenge. This paper aims at collecting and comparing those different tools and approaches by defining desired features and comparing them using the previously defined requirements.

2012 ACM Subject Classification Software and its engineering → Distributed systems organizing principles; Software and its engineering → Software configuration management and version control systems; Software and its engineering → Middleware; Computer systems organization → Cloud computing

Keywords and phrases Application Management, Deployment Lifecycle, Infrastructure-as-Code, Kubernetes, Helm, Kustomize, ytt, kapp

1 Introduction

Modern distributed applications impose many challenges on developers and system administrators alike. To satisfy user needs, applications need to be continuously deployed and dynamically scaled, for example based on user traffic. Therefore, modern distributed software architectures often revolve around splitting the application into multiple parts or services. One such architectural style is the microservice approach, where applications are divided into small and lightweight services that perform a specific business function [3], with services being independently deployable and loosely coupled while communicating through APIs [2].

This comes with a set of advantages: First of all, splitting up an application into microservices unlocks the opportunity to use different programming languages and database technologies for different services, which allows for selecting the best solution for each service on a case-by-case basis. Deploying and rolling out new software versions can also be sped up, since only the affected microservice can be specifically targeted for an update, instead of building and rolling out a whole new monolith [8]. It is also more amenable for the *Continuous Delivery* approach, where new software versions are rolled out to the market automatically and therefore more quickly, improving efficiency with more reliable releases [7]. As shown in [6], maturing deployment pipelines and more automation can lead to 95% reduction in person-hours spent releasing and 86% reduction in release cycle time, making it possible to increase the number of releases per month 2.6-fold. Microservice applications can also be scaled more dynamically [8]. Furthermore, with every module being encapsulated in one service, principles of good modularity within the software architecture are more enforced, which can lead to reduced system coupling [11] and better productivity, since development and maintenance of one service can be assigned to one team, respectively.

Thus, many IT companies like Amazon, Netflix and Spotify are switching to the microservice approach for their cloud-native applications [10]. However, introducing the microservice architectural style comes with its own challenges [3]. Among other challenges, it becomes increasingly hard to configure, deploy and manage applications as an interdependent unit during its whole lifecycle, since it is divided into several services on different tiers. This

Proceedings of the 2020 OMI Seminars *Research Trends in Data Centre Operations*, *Selected Topics in Data Centre Operations*, and *Research Trends in the Internet of Things* (PROMIS 2020).
Editor: Jörg Domaschka; Article No. 1; pp. 1:1–1:11
TIL:DR Today I Learnt: Doing Research series
OPARU Open Access Repository – Ulm University, Germany

is also the case for other non-microservice applications consisting of multiple components and services.

With the rise of containerization technologies in the last few years, these aforementioned (micro-)services are often packaged into containers [14] to decouple them from the systems on which they run. To orchestrate these sets of containers, Google published *Kubernetes* in 2014 [5], an „open-source system for automating deployment, scaling, and management of containerized applications“ [1]. While Kubernetes provides many useful features for administering container infrastructures, there is still missing support for treating and managing the different components and services of an application as a unit.

After covering the basics of containers and Kubernetes in Section 2, requirements for tools to solve this issue are defined in Section 3. Then, multiple tools and approaches are discussed and compared (Sections 4 and 5). The paper concludes in Section 6.

2 Background

2.1 Containers

Containers provide a level of abstraction and isolation from the host OS when running applications inside them, similar to VMs. Unlike VMs, though, containers share the OS with the host machine. While this prohibits using a different operating system inside a container, it also makes container technology more light-weight, allowing for the operation of hundreds of containers on one physical host machine. Additionally, restarting a container does not require restarting the OS itself.

For creating and managing containers, *Docker* has emerged as a commonly used tool. A Docker container runs all processes in isolation from the host, with its own process tree, file system and even its own IP address for networking purposes. Containers are created using a base image, containing OS fundamentals and sometimes also prebuilt applications, based on which instructions can be given on how to extend it, for example in a *Dockerfile*. Those instructions include mounting and copying folders outside the container into it (e.g. for persistent storage) and executing commands for installing dependencies and starting the application itself [4]. Once finished, container images can then be shared and pushed into a remote repository, from where it can be pulled again on other machines as well. This makes it easy to build, deploy and distribute containerized applications and services. With a fixed base image and all the instructions for installing necessary dependencies extending the fixed container image, it is ensured that the application inside the container behaves the same, even if it is run on different host machines, without worrying about technical details and manually installing dependencies on every target system [5].

2.2 Kubernetes

When applications are distributed across multiple containers with multiple instances running at the same time, it becomes increasingly hard to manage them on a cluster. Therefore, Kubernetes was introduced by Google in 2014, and has grown to be one of the largest open-source projects in the world [5]. It is built upon years of Google's experience with their cluster management system Borg, with a few improvements from lessons learned while using it [13].

[1] `https://kubernetes.io/`

Listing 1 Snippet of a Kubernetes Deployment object for a MySQL server declared in a .yaml file as shown in [1].

```
metadata:
  labels:
    app: wordpress
    tier: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: secret
    - name: MYSQL_DATABASE
      value: wordpress_data
    - name: MYSQL_USER
      value: wordpressUser
    - name: MYSQL_PASSWORD
      value: wordpressPassword
```

2.2.1 Core Goals and Principles

Among the core goals of Kubernetes are supporting higher velocity (in terms of deploying updates), easier scaling and more efficiency (in terms of using resources on a cluster) of distributed cloud-native applications, while maintaining high availability. Every Kubernetes object can be defined declaratively in a .json or .yaml file containing the desired state of the object (for example including the container image that should be deployed), while Kubernetes tries to ensure that the desired state matches the actual state. This also provides a self-healing system, with Kubernetes constantly monitoring all objects in the cluster and for example attempting to restart containers when they die unexpectedly, in case they should be running according to declarative configuration files. Scaling can therefore be done by changing the amount of desired replicas that should be simultaneously running [5].

2.2.2 Clusters

A Kubernetes cluster consists of *Nodes* that can be added by the user, representing virtual or physical machines. On every cluster there are master nodes (one created by default when creating a cluster) which are responsible for management, contain the API server and scheduler, and worker nodes responsible for running actual applications. The master nodes then schedule groups of containers (called *Pods*) across worker nodes [5].

2.2.3 Kubernetes Objects

Applications on a Kubernetes cluster are organized in and managed by different objects. They can be imperatively created or declaratively defined inside JSON or YAML-files and deployed using the command line interface `kubectl`. Objects on a cluster can be organized with *Labels* and *Annotations*, providing loose coupling between them.

The core objects within Kubernetes clusters are *Pods*. Pods group multiple containers and *Volumes* running in the same execution environment into one unit. Processes inside a Pod share the same IP address and can have access to the same file system using volumes. Most

of the time, there is only one container running inside a Pod. Along with the desired image, the amount of required resources and resource limits can be specified for every container.

For defining how many replicas of a Pod should be running at the same time and managing the release of new versions, *Deployment* objects are used. The strategy with which an update should be performed can also be defined. Those strategies include killing all Pods and restarting them with the new container image (*Recreate*), or updating only a few Pods at a time to ensure availability during the rollout process (*RollingUpdate*) [5].

Finally, *Services* allow for exposing Pods to other nodes in the cluster or globally, so that it can be accessed by end users. When defining a Service, a set of labels can be specified to determine which Pods provide it. When an application is exposed, Kubernetes automatically load-balances traffic across all Pods within a Service by default [2].

With these basic objects, a cloud-native application can be deployed on a Kubernetes cluster. There are more advanced objects that can be created in a Kubernetes environment, but are not further discussed here.

3 Requirements

Since every Deployment and Service object only corresponds to one part of the application, every artifact needs to be deployed separately (for example using `kubectl apply -f mysql.yaml`, with *mysql.yaml* being the file shown in Listing 1). Even though directories containing multiple .yaml files can be deployed with the same command at once, no dependencies between those Kubernetes objects can be specified, nor exists a tangible Kubernetes object representing all the components of an application by default. Often values like environment variables of containers or service names need to be consistent across multiple objects, creating redundant declarations with potential human error when manually copying configurations.

To solve the issue, several tools and approaches have emerged to support managing the lifecycle of applications on a Kubernetes cluster. In this paper, they are discussed on the basis of the following requirements:

- Deploy and upgrade applications as a unit (R1)
- Verbose output when deploying and upgrading an application (R2)
- View all running application instances and their respective status on the cluster (R3)
- View logs from all Pods of an application (R4)
- View application rollout history and perform rollbacks to earlier releases (R5)
- Delete application instances as a unit from the cluster (R6)
- Externalize configuration options to unify configuration management for common values across artifacts making up the application (R7)
- Provide overlays to further tweak and override application configurations, regardless of what configuration values have been externalized (R8)
- Browse and install existing Kubernetes applications from remote repositories (R9)

4 Tools and Approaches

There is a vast landscape of Kubernetes-related tools that is continually evolving. The following concentrates on rather well-known community-adopted tools.

[2] `https://kubernetes.io/docs/concepts/services-networking/service/`

4.1 Helm

One popular tool meeting many of the above defined criteria is *Helm*. It has been proposed in 2016 to „find, share and use software built for Kubernetes“ [3].

To achieve this, Helm introduces so-called *Charts* to package multiple Kubernetes objects into one deployable, manageable unit. On the client side, Helm provides a command line interface for creating and managing those Charts. Like Docker images, they can be pushed and pulled from a remote repositories, allowing for easy access to available Charts. They can also be searched for pre-configured Charts containing whole applications (for example, a readily configured Wordpress application with a MariaDB backend is available in the official Helm stable chart repository [4]) [12]. Alternatively, Charts can be installed from a local directory or an already packaged *.tar.gz*-file.

4.1.1 Installing and Managing Applications

When installing a Helm Chart containing a preconfigured application, a new instance of it is created, called a *Release*. Helm then automatically creates and deploys all Kubernetes objects needed for the application to run, as defined in the Chart. Releases have a distinct name, which can either be assigned explicitly by the user or randomly generated, representing the respective instance of the deployed Chart. To configure a specific release, additional configuration parameters (as declared in the Chart definition) can be passed as arguments, both mandatory and optional, either by directly accessing them with flags in the command or by providing a YAML-file containing all desired key-value configuration pairs. Along with status information, all currently deployed Releases on the cluster and their current configurations can be displayed. Like Kubernetes itself, Helm also provides the ability to upgrade and roll back Releases. Using `helm history`, the history of a given Release with automatically assigned revision numbers can be retrieved. Rollbacks to a given revision number of a Release can be performed using `helm rollback`. Releases can be uninstalled with one single command, with Helm automatically deleting all respecting Kubernetes objects from the cluster [5].

4.1.2 Helm Charts

As previously mentioned, the core artifact introduced by Helm is a Chart: a „collection of files that describe a related set of Kubernetes resources“ [6].

Within these files, placeholders can be specified using the Go template language. Values for these placeholders can either be provided in the *values.yaml* file as the default configuration, or set and overridden when installing the concrete Chart Release.

When used correctly, redundant configuration can therefore be avoided by defining variables using templates and setting actual values at one place, with Helm substituting all values automatically (as shown in Figure 1). Hence, changing configuration of an application stack is less error prone, because copying values that should be consistent across multiple files is done by Helm itself, rather than manually by maintainers with high possibility for human error.

[3] `https://helm.sh/`
[4] `https://github.com/helm/charts/tree/master/stable/wordpress`
[5] `https://helm.sh/docs/intro/using_helm/`
[6] `https://helm.sh/docs/topics/charts/`

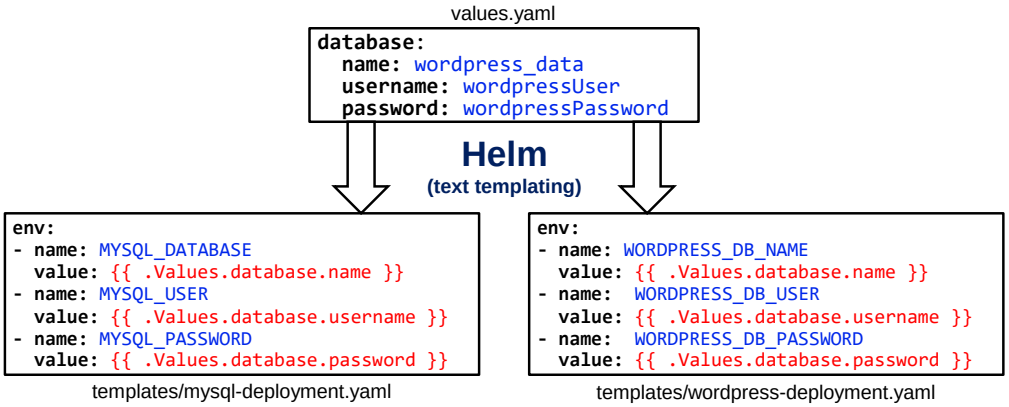

Figure 1 Managing redundant configurations with Helm in a Wordpress example deployment using templates. Helm substitutes these placeholders (in red) with the concrete values provided in the *values.yaml* file.

4.2 Kustomize

An alternative way of managing the lifecycle of applications on a Kubernetes cluster is *Kustomize* [7]. While it is still available as a standalone binary, it has now been integrated into version 1.14 of the Kubernetes-native command line tool `kubectl`. Unlike Helm, Kustomize provides a template-free way of creating and deploying Kubernetes resources by using *bases* and *overlays* to deploy and configure the application for different environments, while also providing ways to configure applications as a unit [8].

4.2.1 Configuring and Applying Applications

This is achieved by declaring a new *kustomization.yaml* file, containing configuration options and references to all related Kubernetes resources of an application. For central configuration of multiple Kubernetes objects, *ConfigMaps* and *Secrets* can be generated within this file. These are standard Kubernetes objects containing configuration options (mostly key-value pairs), with ConfigMaps storing general data and Secrets made for more confidential data [9]. Pods can then reference and read those configuration values, for example when declaring environment variables for containers, thus making it possible to configure multiple Pods as a unit with one single entry in the *kustomization.yaml* file. Additionally, along with other features like name prefixes, a set of common labels which should be added to every referenced Kubernetes object can be defined within the file. Applying the application can then be done with one single command, using the `-k` flag when applying a directory with `kubectl` [10].

4.2.2 Kustomize Overlays

Going further, multiple layers can be introduced to alter configurations for all Kubernetes objects that make up an application. This is especially useful when changing configurations

[7] `https://kustomize.io/`
[8] `https://github.com/kubernetes-sigs/kustomize`
[9] `https://kubernetes.io/docs/concepts/configuration/configmap/`
[10] `https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/`

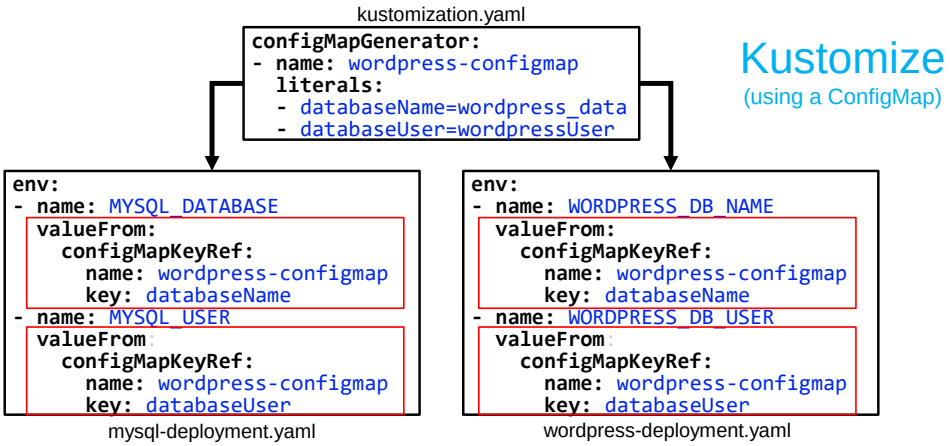

Figure 2 Managing redundant configurations with Kustomization using a ConfigMap. The key *databasePassword* can be referenced accordingly.

for different environments, like development and production, with both having similarities but also some differences in configuration. Therefore, a directory tree can be created, with one directory containing all base YAML-files (along with a *kustomization.yaml*) and other (sub-) directories containing overlay YAML-files, also along with a *kustomization.yaml*. Within the latter, references to all overlay files and a reference to the base directory can be added, often with one overlay file for every smaller configuration option. In the overlay files, metadata has to be provided to identify the objects on which the overlay should be applied, while the `spec` section contains concrete configurations. When applied to a cluster, Kustomize then renders new YAML output using all defined base objects while merging and overriding them with the overlay files [11].

4.3 Application Object

Since the Kubernetes API allows for the creation of custom resources [12], a user-defined type could be introduced to represent an application consisting of multiple components and aggregate them as a unit.

This idea has already been implemented by a Kubernetes working group [13], providing an Application object that can be defined like any other Kubernetes object (preferably in a .yaml-file) which consists of multiple standard Kubernetes objects (like Deployments and Services). Like any other Kubernetes object, the Application object can then be applied, queried and managed using the built-in CLI `kubectl`, showing how many components are ready and providing the possibility of application-level health checks.

Within the declaration of this Application object, additional metadata can be added. This includes the type of application in general (like *Wordpress*), and the name of the concrete instance of the application (like *wordpress-1*). Therefore, multiple running instances of the same application can be displayed and managed. To specify what Kubernetes objects belong to an application, a Label selector has to be provided in the declaration of the

[11] `https://blog.viadee.de/kubernetes-deployments-mit-kustomize` [German]
[12] `https://kubernetes.io/docs/concepts/extend-kubernetes/#user-defined-types`
[13] `https://github.com/kubernetes-sigs/application`

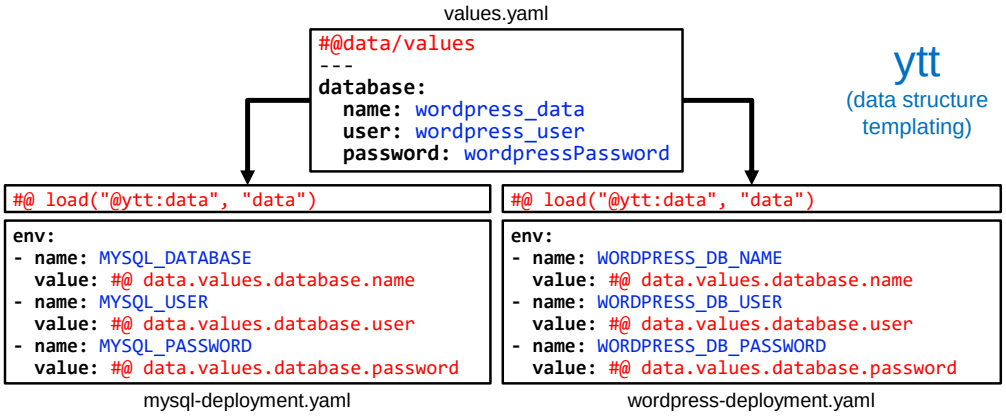

Figure 3 Managing redundant configurations with ytt using templates.

Application object. Every Kubernetes object containing the set of Labels specified in the Label selector is then coupled to the Application object [14]. When deleting the Application object from the cluster, all related Kubernetes objects (matching the specified Label selector, like Deployments and Services) are deleted along with it, if desired.

4.4 ytt

With most manifests of a Kubernetes application written in YAML, a YAML templating tool can be used to manage application configurations on a cluster. One such tool for this job is *ytt* [15], a VMware-backed open-source project developed by the Carvel team [16].

4.4.1 ytt Templates

Unlike Helm, ytt works with *data structure templating* instead of *text templating*. With text templating, users have to constantly worry about the correct YAML output, often manually adding quotation marks or worry about text indentations, since YAML files are treated as plain text without considering the underlying data structure [9]. Ytt in contrast decodes every input file into a tree of YAML nodes before performing data operations and modifications on them. Once done, the tool encodes all nodes into YAML-formatted text output again, ensuring that - if there are no errors during parsing - the rendered output is syntactically correct. Since every ytt directive is embedded in special comments, users constantly write valid YAML manifests.

Like with Helm, centrally configurable common values between Kubernetes objects can be externalized, so separate YAML files with all desired configuration values can be provided to manage application configurations as a unit.

4.4.2 ytt Overlays

Additionally, ytt also supports overlays, with a rich set of selectors (to select which resources the overlay should be applied to) and operations (to specify how the targeted resources

[14] https://github.com/kubernetes-sigs/application/blob/master/docs/api.md
[15] https://carvel.dev/ytt/
[16] https://carvel.dev/

should be modified) that can be used to tweak configurations, providing extra configurability since not only the values that have been externalized within templates can be changed [17].

As mentioned in [9], with Helm exclusively providing templating and value substitution, chart authors faced the problem of having to decide which values should be made configurable. When users demand increased configurability of charts, every possible desired change has to be made configurable in the *values.yaml* file, making it more complicated and less readable. In ytt, however, while commonly changed values can be externalized in a similar way, overlays can be additionally used to further modify YAML resources, without template authors having to provide an externalized configuration option for every possible desired change.

4.5 kapp

Another lightweight tool developed by the Carvel team is *kapp* [18], which focuses more on the deployment and management of applications during runtime rather than managing configuration. Like ytt, kapp is also a VMware-backed open-source project.

With kapp, YAML manifests can be applied as a unit with an application name, while YAML input can be provided either from files and directories or directly using Linux shell pipelining. This especially makes it possible to combine kapp with other tools capable of directly rendering YAML to the standard terminal output (like ytt, Kustomize and Helm).

4.5.1 Deploying Applications

Before deploying (or upgrading, if the same application name already exists on the cluster), an overview of all Kubernetes objects that will be created, modified or deleted is shown, with an extra prompt whether the changes should be applied. If desired, a detailed *Diff* can be printed out (similar to Git) before deployment, showing exactly which lines of YAML manifests are being added and removed once the changes have been applied. During deployment, the tool gives verbose output on what Kubernetes objects are currently being created, deleted or modified, providing more transparency compared to `kubectl`, and individually waits until one resource is ready before deploying the next one [9].

4.5.2 Managing Running Applications

All currently running applications can be queried with one simple command. Furthermore, a specific running application can be inspected, displaying all related Kubernetes objects (optionally in a tree view showing simple parent-child resource relationships). To monitor running applications, kapp provides a command to stream logs from all Pods associated with an application to the terminal output. Like during deployment, all resources associated with a given application can be deleted from the cluster as a unit with one command, displaying a similar verbose output of the operations that will be performed and a prompt for confirmation. [19]

Table 1 Feature comparison of the discussed tools.

Requirement	Helm	Kustomize	App Object	ytt	kapp
R1: Deploy/Upgrade	yes	yes	-	-	yes
R2: Verbose output	-	yes	-	-	detailed
R3: View instances	yes	-	yes	-	yes
R4: Stream Pod logs	-	-	-	-	yes
R5: History/Rollback	yes	-	-	-	-
R6: Delete	yes	yes	yes	-	yes
R7: Unified config	advanced	limited	-	advanced	-
R8: Overlays	-	yes	-	advanced	-
R9: Browse repos	yes	-	-	-	-

5 Comparison

While Helm provides many features in one tool, it lacks transparency and overlays. It includes application packaging with Charts and is the only solution fulfilling R5 and R9. However, as shown by Spillner in [12], the quality of Charts in public Helm repositories is not always ensured. Helm Charts are highly customizable (R7), but configuration templating can be tedious compared to ytt, especially with more complex use cases, since YAML manifests are treated as plain text rather than data structures [9].

Kustomize is more light-weight and already integrated in newer Kubernetes versions. When combined with the Application object it can be a powerful and template-free solution, able to manage applications as a unit (R1, R3, R7) with verbose output and overlays. However, configurability is limited (R7), since only Kubernetes-native objects like ConfigMaps and Secrets can be used for managing common values rather than being able to arbitrarily template YAML files like Helm or ytt.

With ytt and kapp focusing on different aspects (configuration and deployment, respectively), a combination of both can also be a powerful but still light-weight solution, meeting every requirement except R5 and R9. While ytt provides advanced templating and overlays, kapp simplifies the deployment process with verbose output and helps monitoring applications by being able to stream all Pod logs of an application to the terminal. Kapp is therefore the only discussed tool meeting R4. The gap of installing applications from repositories (R9) can be closed with another light-weight tool by Carvel called *imgpkg* [9]. However, ytt templates can be very complex to write and maintain, and kapp lacks the ability to roll back application instances as a unit.

6 Conclusions

When comparing the aforementioned tools, it becomes obvious that there is no single best solution. As shown in Table 1, every tool provides a different incomplete set of desired features, sometimes with different quality. Since Helm, Kustomize and ytt are capable of rendering YAML output to the terminal and kapp being able to deploy YAML from the

[17] `https://carvel.dev/ytt/#playground`
[18] `https://carvel.dev/kapp/`
[19] `https://carvel.dev/kapp/#playground`

standard input, they can be easily combined using Linux shell pipelines. Furthermore, the Application object can also be combined with every other tool, as it is declared and deployed like any other Kubernetes object. Depending on the requirements and personal preference, one or a combination of the previously discussed tools can provide a sufficient solution. To date, no universal single tool is known that meets all the above defined requirements. However, like the ecosystem of available tools, Kubernetes itself is constantly growing and evolving, and we might see some features that are now exclusively provided by external tools being integrated into Kubernetes in the future.

References

1 Example: Deploying wordpress and mysql with persistent volumes. URL: `https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/`.

2 L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 176–185, 2019. `doi:10.1109/QRS.2019.00034`.

3 N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51, 2016. `doi:10.1109/SOCA.2016.15`.

4 D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014. `doi:10.1109/MCC.2014.51`.

5 Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. O'Reilly Media, 2019.

6 M. Callanan and A. Spillane. Devops: Making it easy to do the right thing. *IEEE Software*, 33(3):53–59, 2016. `doi:10.1109/MS.2016.66`.

7 L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015. `doi:10.1109/MS.2015.27`.

8 Martin Fowler. Microservices - a definition of this new architectural term, 2014. URL: `https://martinfowler.com/articles/microservices.html`.

9 Dmitriy Kalinin and Shatarupa Nandi. Managing applications in production: Helm vs. ytt & kapp, 2020. KubeCon and CloudNativeCon Europe 2020. URL: `https://www.youtube.com/watch?v=WJw1MDFMVuk`.

10 F. Rossi, V. Cardellini, and F. L. Presti. Hierarchical scaling of microservices in kubernetes. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37, 2020. `doi:10.1109/ACSOS49614.2020.00023`.

11 M. Song, Q. Liu, and H. E. A mirco-service tracing system based on istio and kubernetes. In *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pages 613–616, 2019. `doi:10.1109/ICSESS47205.2019.9040783`.

12 Josef Spillner. Quality assessment and improvement of helm charts for kubernetes-based cloud applications. *arXiv preprint arXiv:1901.00644*, 2019.

13 Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

14 J. Zhang, R. Ren, C. Huang, X. Fei, W. Qun, and H. Cai. Service dependency based dynamic load balancing algorithm for container clusters. In *2018 IEEE 15th International Conference on e-Business Engineering (ICEBE)*, pages 70–77, 2018. `doi:10.1109/ICEBE.2018.00021`.