A STUDY ON JUSTIFICATION FOR HIGH-QUALITY KUBERNETES SYSTEMS

A STUDY ON JUSTIFICATION FOR HIGH-QUALITY KUBERNETES SYSTEMS

BY

ZHEXUAN LYU, B.Eng.

a Report

submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in partial fulfilment of the requirements

for the degree of

Master of Engineering

Master of Engineering (2025) McMaster University
(Department of Computing and Software) Hamilton, Ontario, Canada

TITLE:	A Study on Justification for High-Quality Kubernetes Systems
AUTHOR:	Zhexuan Lyu B.Eng. (Network Engineering), Hangzhou Dianzi University, Hangzhou, China
SUPERVISOR:	Dr. Sébastien Mosser
NUMBER OF PAGES:	xiii, 65

Lay Abstract

Have you ever wondered how to deploy a high-quality application on a K8s cluster? While K8s has become popular for managing containerized workloads, its complexity can make it challenging to configure robust, secure, and efficient systems at both the application and cluster levels. This research explores how justification diagrams can guide best practices across multiple domains to create more secure and durable K8s deployments. By providing a systematic and flexible approach, these diagrams enable engineers to maintain recommended practices in different environments, ultimately improving the reliability and resilience of K8s-based applications and services.

Abstract

Kubernetes, also known as K8s, has become the de facto foundation for modern computing infrastructures, playing an essential role whether deployed in the cloud or on-premises, and it consistently serves as the core of business operations. However, configuring a robust K8s environment often requires experienced DevOps or platform engineers, and human errors can significantly undermine overall quality. Although numerous resources discuss best practices for K8s, few tools offer tangible mechanisms to help engineers detect and remediate misconfigurations. In this study, we employ the concept of the justification diagram to investigate K8s best practices across multiple domains, including high availability, performance, security, and maintainability. We also propose a novel, keyword-driven operational framework that extends the traditional justification diagram with executable functionality in combination with practical strategies. We then apply the integrated approach to ten specific best practices in the context of GitHub Actions pipelines to show how these verify and improve K8s configurations. The results confirm the feasibility and effectiveness of this approach, improving the quality and reliability in industrial contexts where deployment can be an issue and being systematic and extensible to ensure best practices in diverse environments.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, **Dr. Sébastien Mosser**. I am very grateful to him for bringing up the topic of this project and for his valuable feedback regarding the structure of this work. His continuous guidance throughout my MEng studies and during the code and report review have been instrumental to the completion of this work.

I would also like to extend my special thanks to my friend **Kai Sun** for his patience and thoughtful suggestions throughout this project. I also wish to thank the developers and maintainers of the open-source project, jPipe, since their open-source implementations were really helpful in building the core components of this work.

My parents and family have always been my backbone for necessary and constant financial support and endless emotional encouragement. This feat would not have been possible without them.

Finally, I would like to thank McMaster University, especially the Department of Computing and Software, for providing practical curricula and a project-based learning environment; both have been instrumental in shaping my engineering capabilities.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviations

CI	Continuous Integration
CD	Continuous Deployment/Delivery
JD	Justification Diagram
HA	High Availability
K8s	Kubernetes
AST	Abstract Syntax Tree
LSP	Language Server Protocol
DAG	Directed Acyclic Graph
PDB	Pod Disruption Budget
HPA	Horizontal Pod Autoscaler
VPA	Vertical Pod Autoscaler

PSP	Pod Security Policy
PSA	Pod Security Admission
PSS	Pod Security Standards
SLA	Service Level Agreement
SLO	Service Level Objective
CNCF	Cloud Native Computing Foundation
JSON	JavaScript Object Notation
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

Kubernetes, often abbreviated as K8s, was first developed and launched by Google in 2014, building on its experience with application-oriented APIs for managing durable and reliable containerized distributed systems inherited from its predecessor, Borg [4]. It was then donated to the *Cloud Native Computing Foundation* (CNCF)[1] two years later. After graduating from the CNCF in 2018, it has rapidly become one of the top open-source projects in the world with over 74k contributors [6], as well as the leading industry standard for container orchestration. According to a recent report [41], more than 75% of developers or organizations have experience with Kubernetes or are running Kubernetes clusters in a variety of use cases, including hybrid cloud, cloud-native applications, and modernization of existing services. However, despite K8s' high availability, scalability, and other features, many users, including experienced developers, still find that configuring and managing a high-quality Kubernetes deployment is resource-intensive and time-consuming.

Although there are a number of books, articles, and technical blogs [3, 5, 18, 43]

[1] `https://www.cncf.io/`

discussing how to configure K8s effectively, few have translated these best practices into systematic methods for verifying and validating a given configuration. As a result, K8s misconfigurations are still common and can potentially lead to service vulnerabilities, disruptions, or even failures [33]. While there are some static analysis tools, such as kube-score and kube-linter, which check YAML configurations to ensure certain best practices or rules are applied, they are still in the early stages of development and only work on Kubernetes configuration files. In addition, these tools lack the ability to verify cluster-wide settings and cannot "justify" these checks—i.e., they do not provide systematic reasoning behind each practice or track future updates. Other open-source K8s management tools, such as Helm[2], do make it easier to deploy K8s applications in different environments and reduce repetitive tasks, but they were not designed to validate Kubernetes configurations.

Therefore, it is valuable to design and implement an approach that can demonstrate whether a set of K8s best practices has been properly followed during configuration and maintenance. This approach should meet the following requirements:

1. At the theoretical level, it should be able to extract the core components of K8s best practices and justify the rationale behind them.

2. At the practical level, it should be able to automate these best practices and integrate seamlessly into existing industrial workflows.

To address the above needs, we propose using the *Justification Diagram* (JD) for these best practices, transforming each best practice into a comprehensive and well-justified validation workflow. Furthermore, we propose an optimized and innovative

[2]`https://helm.sh/`

operational justification framework called jPipe Runner, which not only justifies the rationale of each practice but also validates the justification process using programming functions or scripts in a logically structured order. By combining these methods, our approach can be easily extended to any of Kubernetes best practices and run separately via the jPipe Runner operational justification framework or integrated into the pipeline for *Continuous Integration and Continuous Delivery* (CI/CD), ultimately enhancing the overall quality of Kubernetes systems.

In this report, we focus primarily on answering the following three questions:

1. How can a practice be extracted and transformed into a Justification Diagram?
2. How does the operational justification framework make JD files executable?
3. How can Justification Diagrams be used with these best practices to enhance the quality of a K8s system?

These questions are addressed in the following chapters, providing more insight and detail into our work. Chapter 2 offers a preliminary introduction to the technical background, including K8s, CI/CD pipelines, and Justification Diagram. Next, Chapter 3 discusses the best practices collected from multiple domains, including high availability, performance, security, and maintainability, and explains how they are transformed into Justification Diagrams. Then, Chapter 4 introduces the motivation, design and implementation of the jPipe Runner operational justification framework, and Chapter 5 examines a real-world scenario to demonstrate how these best practices can be represented and tracked using Justification Diagrams in CI/CD pipelines to enhance K8s system quality. Finally, Chapter 6 summarizes the conclusion of this project and discusses potential future work.

Chapter 2

Background

This chapter provides a preliminary background introduction to help readers better understand the technical concepts and knowledge mentioned in this report, including Kubernetes, CI/CD pipelines, and Justification Diagrams.

2.1 Introduction to Kubernetes

Kubernetes, an open-source platform for container orchestration, was initially conceived and developed by Brendan Burns, Joe Beda, and Craig McLuckie at Google [4]. It automatically deploys, scales, and manages containerized applications, significantly simplifies complex workflows, and increases system reliability, quickly becoming the most popular and widely used tool for cloud container orchestration. Some K8s clusters with high *Service Level Agreements* (SLAs) may feature multiple control planes to increase the availability of API controllers and servers, and to reduce potential downtime and service disruptions. However, as shown in Figure 2.1, a typical K8s cluster usually consists of a single control plane and a group of worker machines running containerized applications [14]. A control plane typically has several key components—API server, etcd, scheduler, and controller manager—which together detect, control, and respond to various cluster events. There are three core concepts in K8s: pod, node, and cluster. In a K8s environment, a pod is usually the minimum unit of deployment, and a node generally refers to a physical or virtual machine that provides necessary resources such as CPU and memory to run containerized pods. A cluster is a group of nodes in a K8s environment that orchestrates containerized applications and services [38]. All control plane and master/worker nodes work collaboratively, forming a complex, state-of-the-art Kubernetes cluster.

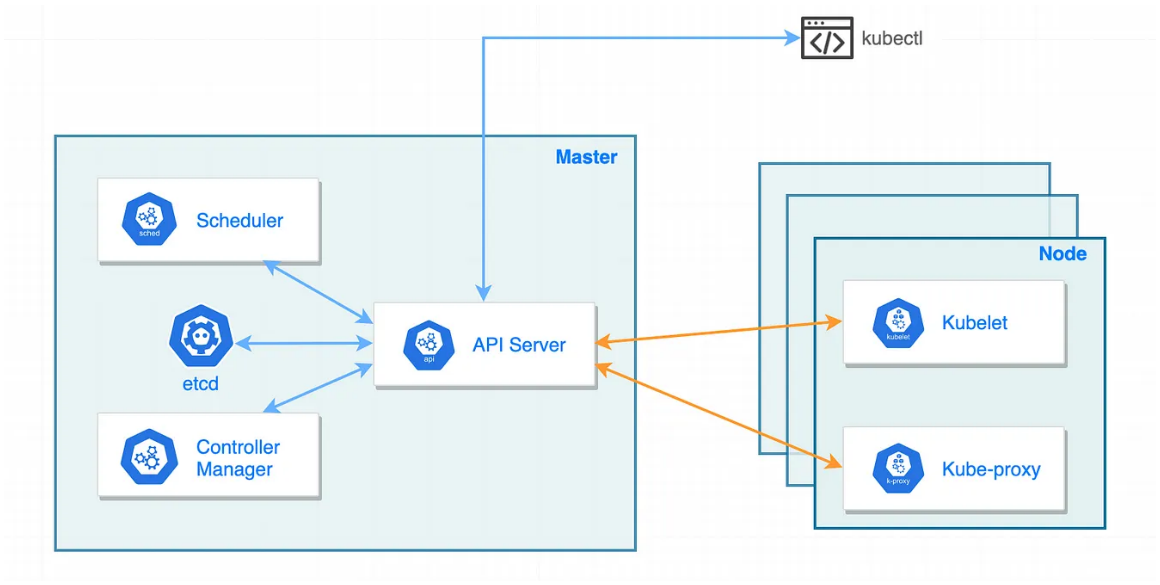


Figure 2.1: Kubernetes cluster components [37]

2.1.1 Configuration Challenges

With a growing number of Kubernetes users and organizations, misconfigurations have become more apparent, leading to many potential risks and issues. Due to the inherent complexity of K8s itself, as well as cloud infrastructure considerations and other requirements for scalability, security, and reliability, configuring a high-quality application in a K8s cluster can be challenging. A 2022 study [36] shows that despite K8s providing many benefits, such as easy cloud-based interfacing, *Service Level Objective* (SLO)-based scalability, and auto-recovery applications, there are still numerous challenges when dealing with K8s. For example, the study specifically mentions that a lack of security practices and tools has led to increased security vulnerabilities in K8s installations and deployments, and these security issues have caused significant delays in K8s-based software deployment. Moreover, the study indicates that maintenance-related challenges associated with K8s resource management demand significant time and effort from developers, while system, network, and performance configurations further increase the overall burden. A report from Red Hat [32] reveals that nearly 60% of survey respondents have experienced a misconfiguration incident in their environment that could lead to data breaches and hacks. In addition, almost 47% of respondents expressed concerns about their environments being exposed due to misconfigurations. Therefore, we need not only a guide to K8s best practices but also a mechanism for checking and justifying them. Chapter 3 provides more best practices across different K8s domains to address these misconfiguration challenges.

2.1.2 Kubernetes Distributions

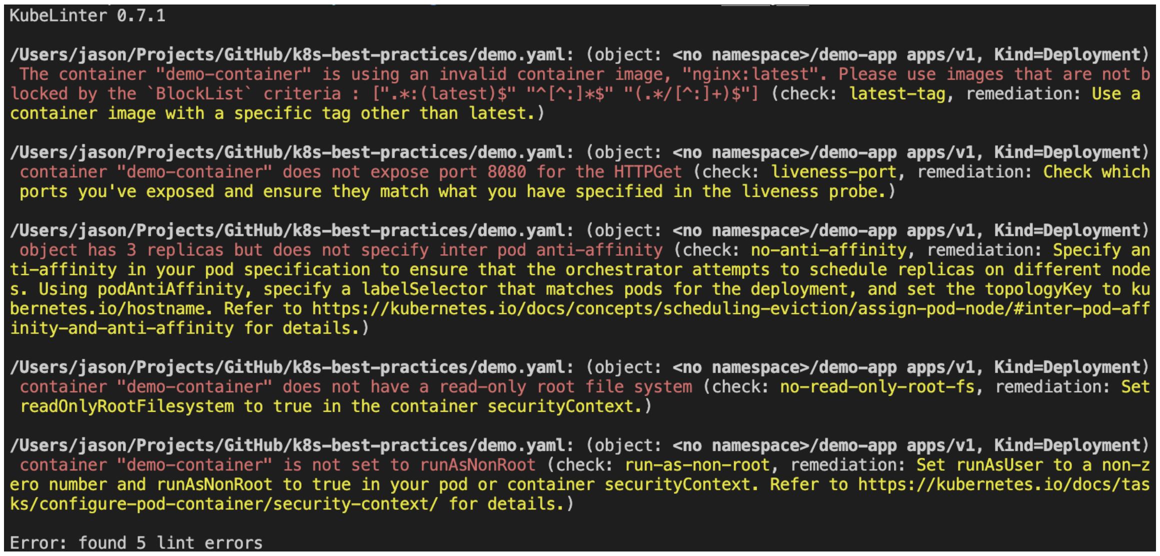
Similar to Linux distributions, Kubernetes also has various distributions to tackle different usage requirements. In addition to the official K8s, which is a general-purpose container orchestrator, there are several well-known lightweight distributions, such as MicroK8s, K0s, K3s, and minikube [11], that serve different purposes across various scenarios. For example, K0s and K3s are both CNCF-certified K8s distributions that offer easy-to-configure cluster solutions [21], especially for resource-constrained edge devices. A 2024 paper [1] presents a comparative study of the security and performance aspects of these three distributions (K0s, K3s, and K8s) and concludes that K0s and K8s deliver better performance and have fewer security vulnerabilities. Minikube [26], on the other hand, is not intended for production-grade workloads but is designed for even lighter use in testing and experimentation. Since it is compatible with all the essential features of K8s, such as scalability and multi-node support[1], and because performance is not our primary concern, minikube becomes an ideal tool for running and validating K8s practices in this project.

2.1.3 Kube-Linter

Kube-linter is a configuration analysis tool for Kubernetes written in Golang [40]. It performs static analysis and best practice compliance checks on Kubernetes YAML files and Helm charts to identify potential issues and improve configuration quality. While it is still in its early development stage and lacks systematic practice reasoning, it provides nearly 60 lint checks that significantly simplify our work by reducing the time needed to implement specific checking rules from scratch, such as minimum-replicas and latest-tag checks. Therefore, Kube-linter is widely used in this project as the underlying implementation for certain practice checks. A sample output from kube-linter is shown in Figure 2.2, including various lint errors and remediation tips. However, tools like kube-linter are sometimes insufficient for solely assisting in K8s

[1] Minikube introduced multi-node support in v1.9.0 (CHANGELOG)

configuration, as they do not provide detailed rationales behind each best practice, lack explanations of the potential consequences of not following a practice, and, most importantly, cannot systematically track evolutionary updates to a configuration.



```
KubeLinter 0.7.1

/Users/jason/Projects/GitHub/k8s-best-practices/demo.yaml: (object: <no namespace>/demo-app apps/v1, Kind=Deployment)
 The container "demo-container" is using an invalid container image, "nginx:latest". Please use images that are not b
locked by the `BlockList` criteria : [".*:(latest)$" "^[^:]*$" "(.*/[^:]+)$"] (check: latest-tag, remediation: Use a
container image with a specific tag other than latest.)

/Users/jason/Projects/GitHub/k8s-best-practices/demo.yaml: (object: <no namespace>/demo-app apps/v1, Kind=Deployment)
 container "demo-container" does not expose port 8080 for the HTTPGet (check: liveness-port, remediation: Check which
 ports you've exposed and ensure they match what you have specified in the liveness probe.)

/Users/jason/Projects/GitHub/k8s-best-practices/demo.yaml: (object: <no namespace>/demo-app apps/v1, Kind=Deployment)
 object has 3 replicas but does not specify inter pod anti-affinity (check: no-anti-affinity, remediation: Specify an
ti-affinity in your pod specification to ensure that the orchestrator attempts to schedule replicas on different node
s. Using podAntiAffinity, specify a labelSelector that matches pods for the deployment, and set the topologyKey to ku
bernetes.io/hostname. Refer to https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-aff
inity-and-anti-affinity for details.)

/Users/jason/Projects/GitHub/k8s-best-practices/demo.yaml: (object: <no namespace>/demo-app apps/v1, Kind=Deployment)
 container "demo-container" does not have a read-only root file system (check: no-read-only-root-fs, remediation: Set
 readOnlyRootFilesystem to true in the container securityContext.)

/Users/jason/Projects/GitHub/k8s-best-practices/demo.yaml: (object: <no namespace>/demo-app apps/v1, Kind=Deployment)
 container "demo-container" is not set to runAsNonRoot (check: run-as-non-root, remediation: Set runAsUser to a non-z
ero number and runAsNonRoot to true in your pod or container securityContext. Refer to https://kubernetes.io/docs/tas
ks/configure-pod-container/security-context/ for details.)

Error: found 5 lint errors
```

Figure 2.2: Kube-linter sample output

2.1.4 Helm-based Configuration

Among all the K8s management tools available in the market, Helm is the most popular and widely used for modern projects or services that require parameterized deployment. Before Helm-based configuration was introduced, deploying a K8s application across multiple environments was challenging and tedious at times. For example, as shown on the left side of Figure 2.3, even for minor modifications, developers had to maintain multiple copies of the same K8s configuration or manage separate branches for different environments, resulting in significant maintenance burdens and potential inconsistencies between configurations.

Helm effectively solves this problem by separating K8s configuration into multiple

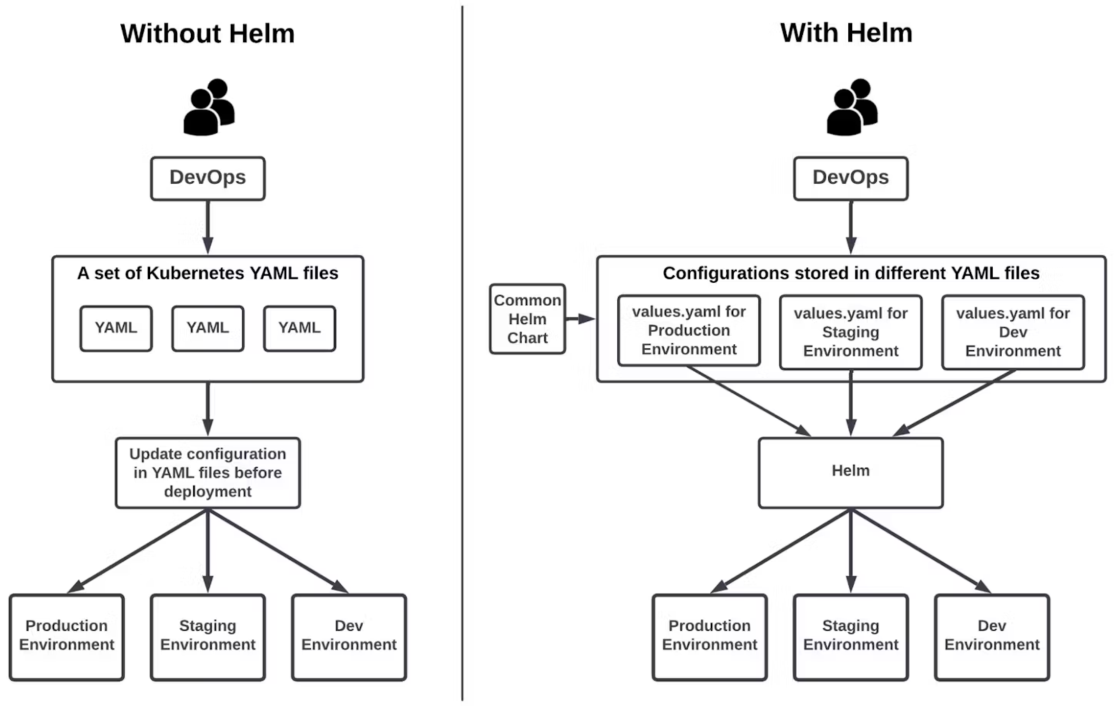

Without Helm
DevOps
A set of Kubernetes YAML files
YAML
YAML
YAML
Update configuration in YAML files before deployment
Production Environment
Staging Environment
Dev Environment
With Helm
DevOps
Configurations stored in different YAML files
Common Helm Chart
values.yaml for Production Environment
values.yaml for Staging Environment
values.yaml for Dev Environment
Helm
Production Environment
Staging Environment
Dev Environment

Figure 2.3: K8s deployment with vs. without Helm [35]

parts and rendering them into a final configuration as needed using the Go templating engine[2]. Helm introduces the concept of a Chart, a package that contains all the required resources—such as deployments, services, and secrets—for deploying an application in a K8s environment [35]. Furthermore, a Helm chart is also designed to be used as a dependency for other charts since all charts are mandatorily versioned and can be easily downloaded from a Helm repository, similar to a Docker registry. A Helm chart typically consists of three key components: a `Chart.yaml` file, which contains metadata such as the Helm version and chart dependencies; a `templates/` folder, which stores all deployable resource information; and one or more YAML files that define values for resource configuration. The right side of Figure 2.3 shows the

[2]`https://pkg.go.dev/text/template`

deployment workflow when using a Helm chart, in which a set of `values.yaml` files is used to configure different environments, and the chart is finally rendered by Helm before deployment, significantly simplifying the Kubernetes management process.

As one of the best practices for facilitating maintainability, parameterized management using Helm is essential for industrial K8s application deployment. Further details on Helm's role in maintainability practices are provided in Chapter 3.

2.2 DevOps and CI/CD

DevOps is an essential concept in software development today. It is a methodology that combines 'Dev' from software development and 'Ops' from information technology operations [2]. It also serves as a cultural approach that bridges the development and operations teams. On the other hand, CI/CD, often seen as Continuous Integration and Continuous Deployment/Delivery, are core practices in the DevOps workflow. The classical CI/CD process is based on three principles: continuous integration, continuous testing, and continuous deployment. Figure 2.4 illustrates the process where software is built, integrated, and tested in a development environment and released, deployed, and monitored in an operational context. These steps allow developers to locate and fix problems in the whole software development cycle. With DevOps integrated into CI/CD, developers can accelerate software delivery, increase collaboration, and decrease deployment failures [7].

There are many CI/CD implementations available on the market, such as Travis CI, Jenkins, and GitHub Actions[3], targeting various usage scenarios and purposes. For instance, Jenkins is a self-hosted automation server widely used for CI/CD in

[3]`https://github.com/features/actions`

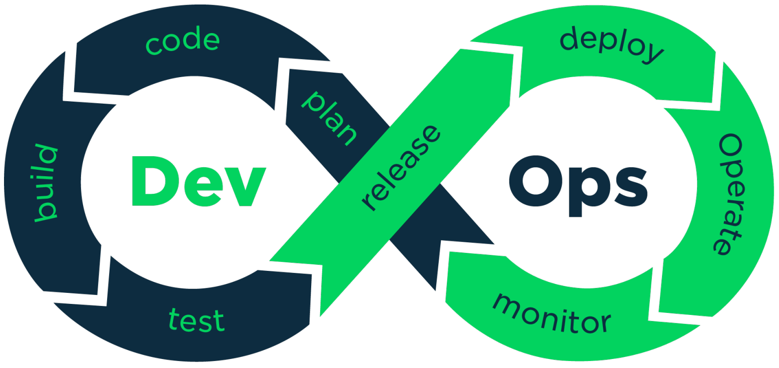


Figure 2.4: DevOps workflow [44]

many enterprises. It uses a `Jenkinsfile` to enable declarative or scripted workflow configuration. However, Jenkins requires substantial customization and dedicated server maintenance, making it inconvenient and burdensome for small or experimental environments. Therefore, since our project code is hosted on the GitHub Repository and GitHub Actions is highly integrated with GitHub toolchains, we chose GitHub Actions as our CI/CD platform to run and verify our best practices. GitHub Actions uses GitHub Workflows, an automated process associated with Actions that can be configured using YAML files. Workflows are able to be triggered either manually or by different events, such as Git pushes or Git pull requests. In this project, each practice quality check is delivered as a CI/CD pipeline within GitHub Workflows, which can be reproduced or tested via manual workflow dispatch or pull request events.

2.3 Justification Diagram

The concept of the *Justification Diagram* (JD) was first introduced in a 2016 paper [29] by Thomas Polacsek, aimed at providing confidence in verification and validation requirements. It is based on the Toulmin argument model [30], an argumentation pattern that visualizes the key components of a product and establishes trust in the outcome. This approach transforms abstract or poorly organized documentation into a transparent representation that justifies why a particular outcome or decision is acceptable, thereby improving quality assurance in complex systems. The jPipe language [22], a specific JD language implemented by the jPipe compiler, defines several key elements, including evidence, strategy, sub-conclusion, and conclusion, and these elements establish distinct dependency relations and constraints.

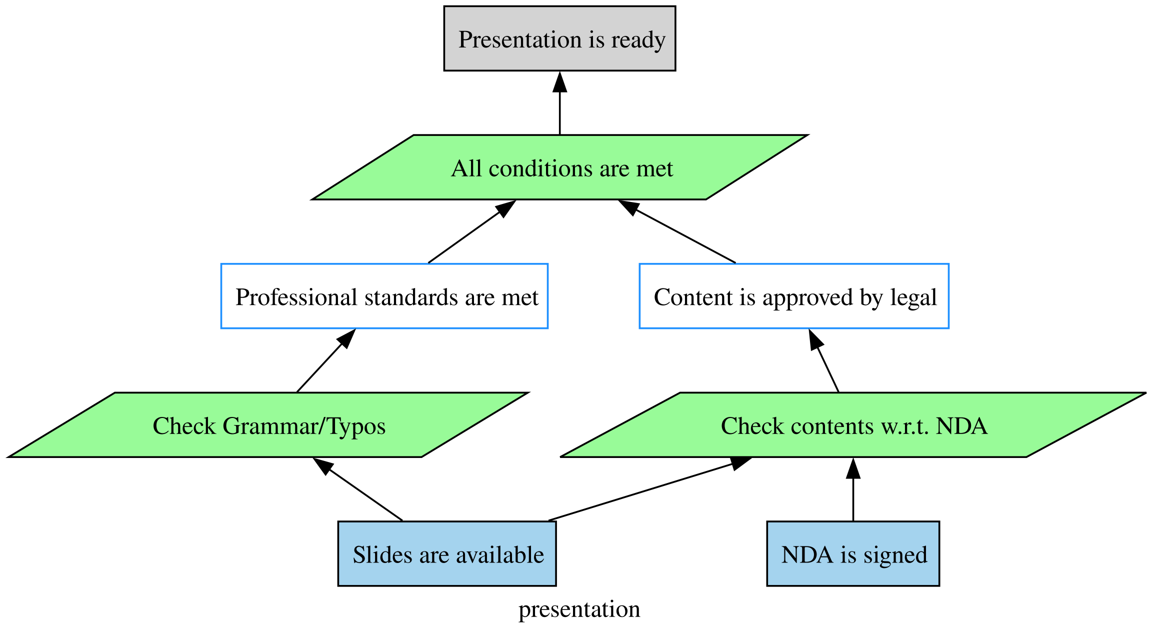


Figure 2.5: Justification Diagram for presentation readiness [23]

As shown in Figure 2.5, this Justification Diagram is used to justify readiness for presentations. The two light blue rectangular nodes, *"Slides are available"* and *"NDA is signed"*, are evidence elements that support the two strategy elements in pale green parallelogram nodes. The arrows represent the support relationships between elements. A strategy relies on one or more evidence elements and can lead to either a sub-conclusion or a conclusion, while a single evidence element can support multiple strategies. However, each strategy can infer only one conclusion. For example, the *"NDA is signed"* evidence supports the *"Check contents w.r.t. NDA"* strategy, and the *"Slides are available"* evidence supports both *"Check Grammar/Typos"* and *"Slides are available"*. A conclusion is represented by light grey rectangular nodes, while a sub-conclusion is represented by dodger blue outlined rectangular nodes. For instance, the *"Check contents w.r.t. NDA"* strategy leads to the *"Content is approved by legal"* sub-conclusion, while the *"Check Grammar/Typos"* strategy leads to the *"Professional standards are met"* sub-conclusion. These sub-conclusions further support the *"All conditions are met"* strategy, which functions as a logical AND operation, eventually leading to the conclusion *"Presentation is ready"*.

Therefore, the entire justification process is straightforward and can be effectively used for validating and tracking changes with concrete evidence in Justification Diagrams. For more detailed explanations and examples of jPipe and JD language, please refer to the jPipe tutorial repository [23]. Similar to previous studies [31, 42] that have demonstrated the feasibility of using JD to justify various system qualities, this project exploits and extends a similar approach to justify the K8s system quality.

Chapter 3

Kubernetes Best Practices and Justification

As mentioned in Chapter 2, misconfigurations in K8s have been causing risks and issues due to its increasing popularity and complexity. Although numerous studies have proposed best practices for K8s configuration, the lack of effective and systematic tools that allow developers to consistently apply these practices remains an issue. Consequently, there is a strong need to address this gap by justifying these best practices and transforming them into reasonable, executable configuration quality checks. This need has led us to adopt the Justification Diagram, an ideal and valuable tool that enables us to justify the rationale behind these best practices and convert them into actionable quality checks. Moreover, the best practices in this report are collected and designed to address different aspects of various K8s domains; therefore, they are independent and, theoretically, can be applied either individually or in combination to the K8s environment.

In this chapter, we examine K8s best practices across four distinct domains and focus on addressing **how each practice can be extracted and transformed into a Justification Diagram**. Each domain includes two to three practices, for a total of ten best practices. The four domains are High Availability (Section 3.1), Maintainability (Section 3.2), Performance (Section 3.3), and Security (Section 3.4). All resources presented in this chapter are available in the *"k8s-best-practices"* repository [25], providing guidance for readers to understand each best practice as well as the implementation of each configuration quality check.

3.1 High Availability

High Availability (HA) is one of the most crucial features provided by K8s and is essential for achieving high SLAs in many enterprises. A properly configured K8s application should be able to avoid service disruptions and remain available even when encountering sudden pod crashes or node failures. In this section, three best practices are presented to achieve HA: anti-affinity rules, deployment replicas, and pod disruption budget.

3.1.1 Anti-affinity Rules

When a K8s application is deployed with replicas, multiple copies of the application are created and run in different pods. However, if all these pods are running on the same node, a single node failure will cause the application to stop serving, regardless of the number of replicas allocated. To achieve HA, one best practice is to ensure that replicated pods are deployed on different available nodes. Several books and articles [5, 8, 27] suggest applying anti-affinity rules that control pod placement based on specific requirements to ensure that pods are separated across nodes, thereby avoiding a single point of failure and enhancing high availability.

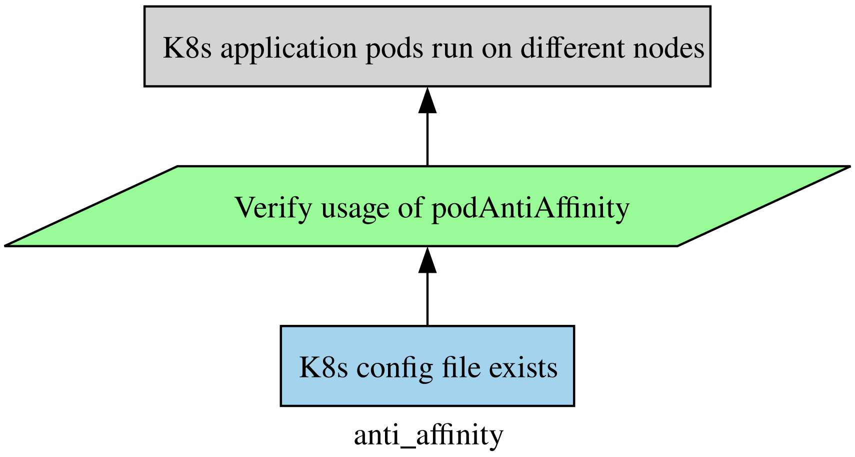

K8s application pods run on different nodes
Verify usage of podAntiAffinity
K8s config file exists
anti_affinity

Figure 3.1: Justification Diagram for anti-affinity rules

Figure 3.1 shows the Justification Diagram for the anti-affinity rules. Ensuring proper use of the `podAntiAffinity` specification allows pods to be distributed on the desired nodes during deployment replication; therefore, the practice conclusion is *"K8s application pods run on different nodes"*. The evidence for this practice, on which the conclusion is based, is *"K8s config file exists"*, considering that our definition of ready requires verifying the specific settings in the YAML configuration file from a testing perspective. This evidence is also shared across most of the remaining diagrams of best practices. The strategy here is *"verify usage of podAntiAffinity"*, which connects the evidence and the conclusion and establishes the reasoning between them. Finally, once the configuration file is ready, we can build the quality check by first verifying its existence and then confirming that the configuration has proper anti-affinity rules set.

3.1.2 Deployment Replicas

Replication is a common technique widely used in computer-based systems to achieve high availability by ensuring a specified number of replicas are always running [20]. Therefore, we extract another best practice from the book [5]: using replicas of Deployment-kind objects for applications. The Justification Diagram is represented in Figure 3.2. It has three strategies: *"verify deployment has replicas"*, *"verify ReplicaSet is not used"*, and *"all replica requirements are met"*.

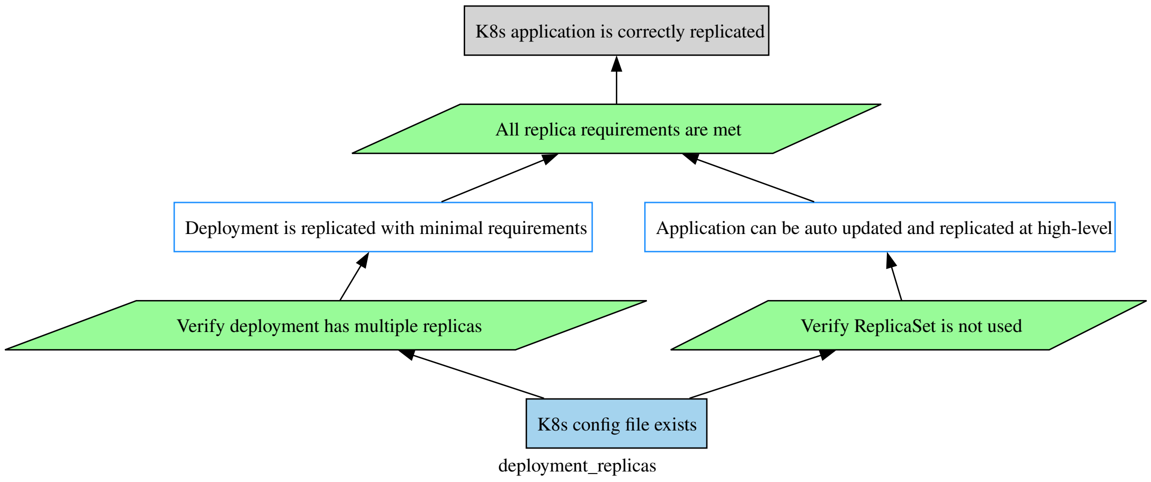

K8s application is correctly replicated
All replica requirements are met
Deployment is replicated with minimal requirements
Application can be auto updated and replicated at high-level
Verify deployment has multiple replicas
Verify ReplicaSet is not used
K8s config file exists
deployment_replicas

Figure 3.2: Justification Diagram for deployment replicas

The first strategy is to ensure that, in the K8s configuration, Deployment-kind objects have the replicas field properly configured. The number of replicas usually depends on the real-world application requirements, but a minimum of three replicas is recommended. This strategy leads to the sub-conclusion *"Deployment is replicated with minimal requirements"*. The second strategy is to prevent direct use of the ReplicaSet feature because, while a ReplicaSet can manage pod replication, it lacks the high-level management features of Deployments, such as rolling updates and rollback capabilities, and may cause other updating or backup issues. Consequently, the sub-conclusion of the second strategy is *"Application can be automatically updated and replicated at a high level"*. The third strategy is a special method that behaves like an AND logical operator in the JD language, connecting the previous two sub-conclusions and leading to the *"K8s application is correctly replicated"* conclusion when both sub-conclusions are reached.

3.1.3 Pod Disruption Budget

In Kubernetes, pods may be evicted from a host at some point, either voluntarily or involuntarily. Involuntary evictions are usually caused by various failures, such as hardware issues, network loss, or kernel panics, which can typically be prevented by multi-node replicas, as mentioned in the previous section. Voluntary disruptions, on the other hand, can be caused by cluster maintenance or pod template updates. In this case, two articles [5, 19] recommend setting a *PodDisruptionBudget* (PDB) to minimize the impact on our K8s applications.

A `PodDisruptionBudget` can be applied to a set of deployments by specifying a label selector, which allows users to set a policy on the minimum number of replicas that must remain available or the maximum number of replicas that can be unavailable during voluntary disruptions. For example, one can specify that a maximum of 50% of the pods for a targeted application can be evicted and updated at a given time, ensuring the required uptime of the application. As illustrated in Figure 3.3, once we verify that a PDB is enabled in the configuration, we can conclude that *"K8s application is available during voluntary disruptions"*. Although this practice does not directly contribute to runtime high availability, it is a powerful approach to enhance availability from a precautionary perspective.

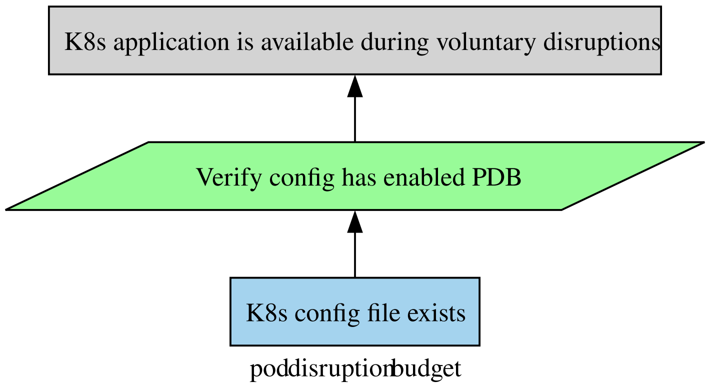

K8s application is available during voluntary disruptions
Verify config has enabled PDB
K8s config file exists
poddisruptionbudget

Figure 3.3: Justification Diagram for pod disruption budget

3.2 Maintainability

Maintaining a high-quality K8s application can be a challenging task even for a group of experienced DevOps engineers, given the complexity of today's K8s environment. Although numerous best practices exist to facilitate the K8s maintenance process, due to the limited scope of this report, in this section we primarily focus on two practices: versioned image tagging and parameterized management.

3.2.1 Versioned Image Tagging

As a container orchestration platform, K8s applications rely heavily on pulling container images from container registries such as Docker Container Registry and Amazon Elastic Container Registry. When pulling images from a container registry, many users are inclined to use the "latest" tag to acquire the newest dependent images for simplicity or convenience. A 2024 blog from Docker [34] states that using the "latest" tag is not advisable because it is not a mandatory standard and, therefore, does not always point to the highest version, which may result in running applications with outdated images. The book [5] also suggests that we should avoid using "latest" as an image tag, as such a tag is not a valid version and could lead to difficulty in identifying which code change corresponds to the rolled-out image, thereby increasing the difficulty of troubleshooting and maintenance.

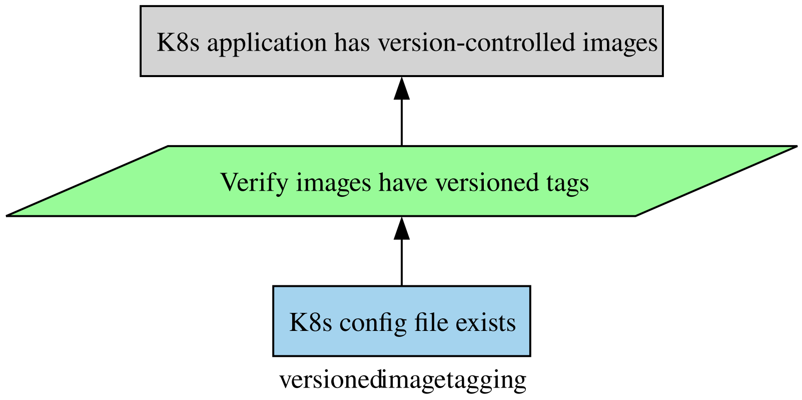

K8s application has version-controlled images
Verify images have versioned tags
K8s config file exists
versionedimagetagging

Figure 3.4: Justification Diagram for versioned image tagging

Alternatively, the book proposes multiple tagging strategies, such as using *BuildID*, *Git Hash*, or a combination of *BuildID* and *Git Hash* as a unique tag, to easily locate the code changes corresponding to a specific release version or Git commit. However, due to the variety of tagging strategies available, it is unrealistic to implement a feasible test to check all possible tag names. Therefore, in this practice, we only impose the strategy *"verify images have versioned tags"* by applying a blocklist that prevents the direct use of images with "latest" tags or untagged images. The diagram can be viewed in Figure 3.4, and for a more detailed implementation of this blocklist, please refer to the repository [25]. Eventually, the conclusion *"K8s application has version control of images"* can be reached after the strategy is verified.

Thus, the overall workflow of this configuration quality test is to first check the existence of the K8s configuration file and then ensure that the image tags do not violate the above blocklist.

3.2.2 Parameterized Management

As discussed in the background chapter, real-world K8s application deployments typically involve multiple steps, including internal alpha testing, external beta testing, and production release. Each of these steps requires a dedicated K8s environment due to security, isolation, and debugging concerns. For example, developers may want to specify different numbers of replicas in different environments, such as using a small number in the test environment and more in production. One initial option is simply to copy configuration files from one place to another, which is impractical and can result in significant maintenance overhead. Therefore, two studies [5, 45] endorse using Helm for this type of parametrization as the best practice.

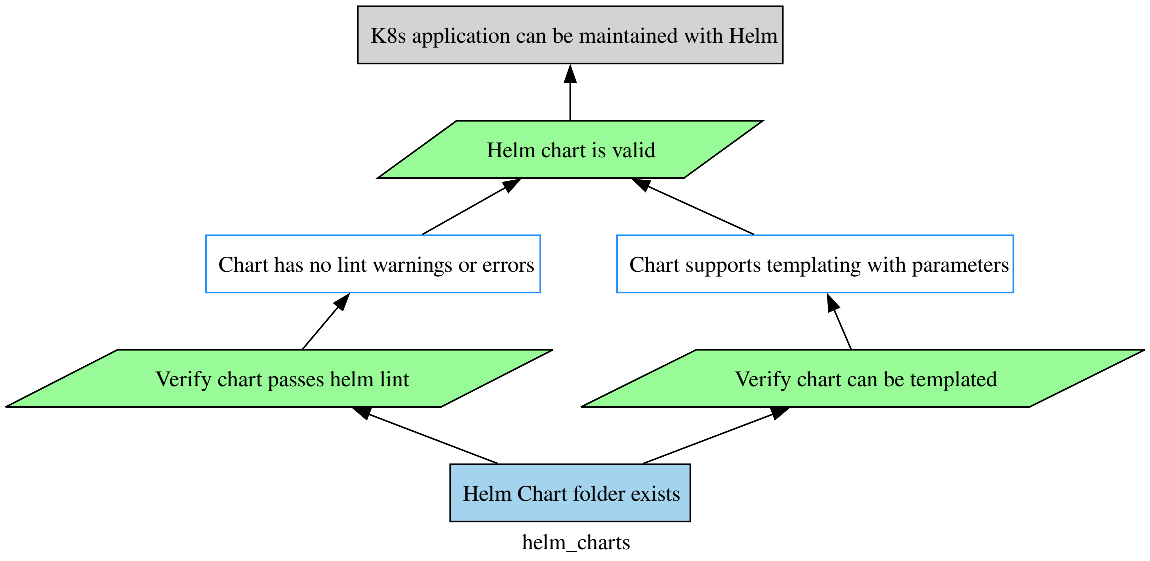

K8s application can be maintained with Helm
Helm chart is valid
Chart has no lint warnings or errors
Chart supports templating with parameters
Verify chart passes helm lint
Verify chart can be templated
Helm Chart folder exists
helm_charts

Figure 3.5: Justification Diagram for parameterized management

In this practice, ensuring the proper use of Helm charts facilitates parameterized management, easing the Kubernetes configuration and maintenance burden. As shown in Figure 3.5, similar to previous practice evidence, the evidence here is *“Helm*

Chart folder exists" since each Helm chart is a structured folder. To consider a Helm chart valid, two conditions should be met:

1. The Helm chart has no lint warnings or errors, which eliminates potential issues.
2. The Helm chart can generate parameterized and templated K8s configurations.

Two strategies verify these conditions and lead to the corresponding sub-conclusions. Thus, the conclusion is *"K8s application can be maintained with Helm"* given that the conditions are verified by the logical strategy *"Helm chart is valid"*.

3.3 Performance

A high-quality K8s application should remain highly responsive while handling sudden request spikes or heavy workloads and reduce resource consumption during idle periods. In this category, we introduce three best practices—load balancing, pod autoscaling, and resource management—to demonstrate how to achieve high performance in Kubernetes deployments.

3.3.1 Load Balancing

Load balancing is crucial in Kubernetes for maintaining the overall performance of applications. A properly configured load balancing component ensures that traffic is distributed evenly and efficiently across pods, preventing any single pod from becoming a bottleneck while ensuring that no pod is underutilized [12]. The book [5] recommends using the Service-kind object, a built-in TCP load balancer in K8s, to implement L4 load balancing for either internal or external traffic.

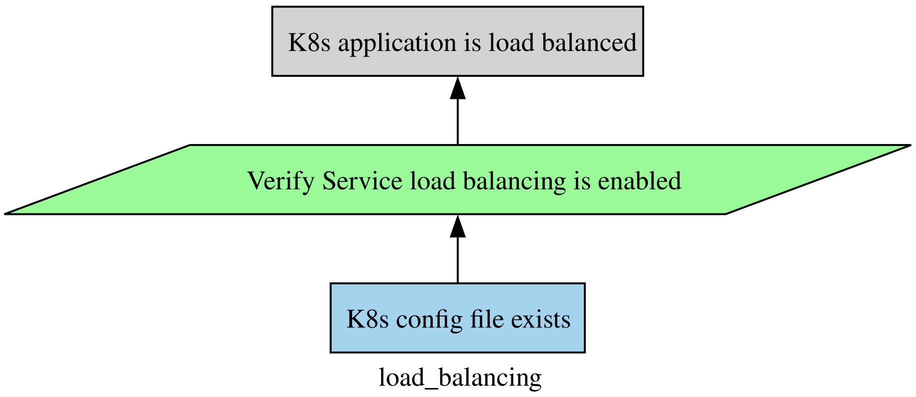

K8s application is load balanced
Verify Service load balancing is enabled
K8s config file exists
load_balancing

Figure 3.6: Justification Diagram for load balancing

Figure 3.6 exhibits the flow of this practice configuration check. The strategy *"Verify Service load balancing is enabled"* is supported by evidence similar to that used in previous practices. It simply checks if Service-kind objects exist in the given configuration file and have a proper service type set. There are typically four service types with different effects, as shown in Table 3.1, and the actual load balancing is performed by the kube-proxy[1] as part of K8s networking. Finally, once the Service objects are verified, the conclusion is reached that *"K8s application is load balanced"*.

Table 3.1: Service types for K8s load balancing

Type	**Description**
ClusterIP	Uses an internal IP so the service is only accessible within the cluster.
NodePort	Opens a fixed port on every node, allowing external access to the service.
LoadBalancer	Assigns an external IP to distribute traffic to an external load balancer, e.g., *Amazon Elastic Load Balancing* (ELB).
ExternalName	Maps the service to a given DNS name without creating a proxy.

[1]`https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/`

3.3.2 Pod Autoscaling

Autoscaling improves application performance by dynamically allocating resources, handling workloads that are not fixed or experience sudden spikes, and reducing response latency. *Horizontal Pod Autoscaler* (HPA) and *Vertical Pod Autoscaler* (VPA) are two kinds of autoscaling pod schedulers. HPA scales by deploying more pods in response to increasing loads, and VPA scales by assigning more resources, such as memory or CPU, to the pods. While VPA can be beneficial in certain scenarios, several sources [5, 39, 10] recommend not adopting this feature for production deployments due to its slow response in adjusting resource requests and potential disruptions to service consistency. Therefore, as suggested by the book [5], we only apply HPA as the best practice for pod autoscaling.

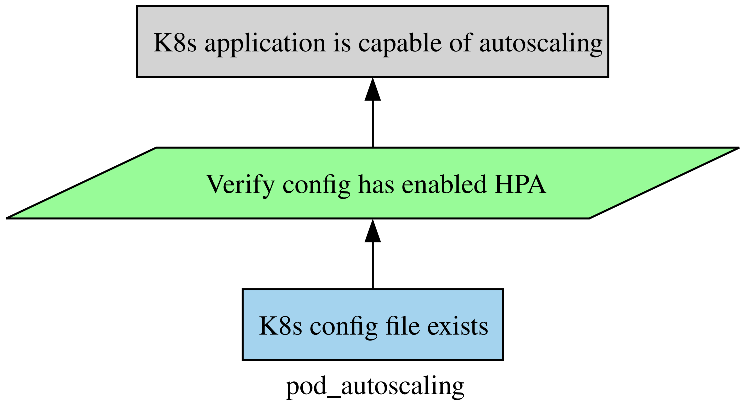

K8s application is capable of autoscaling
Verify config has enabled HPA
K8s config file exists
pod_autoscaling

Figure 3.7: Justification Diagram for pod autoscaling

The diagram illustrates the check steps in Figure 3.7. It first checks the existence of the given K8s configuration file and validates the evidence *"K8s config file exists"*. Second, the strategy *"Verify config has enabled HPA"* verifies that the HPA objects are correctly enabled with reasonable replica ranges. If the configuration conforms to the quality check, it can be concluded that *"K8s application is capable of autoscaling"*.

3.3.3 Resource Management

Resource management is a crucial approach to prevent containers from contending for resources, ensure fair use of each resource, and improve overall system quality and performance. There are typically two types of resources—CPU and memory—that should be considered for each container. There are also two ways to define resource allocation: requests and limits. The "requests" refers to the reservation, which is the guaranteed minimum amount of a resource that can be allocated to a container, while the "limits" refers to the maximum resource usage allowed for a container. This practice needs to ensure that both of these resources are properly configured with reasonable resource requests and limits.

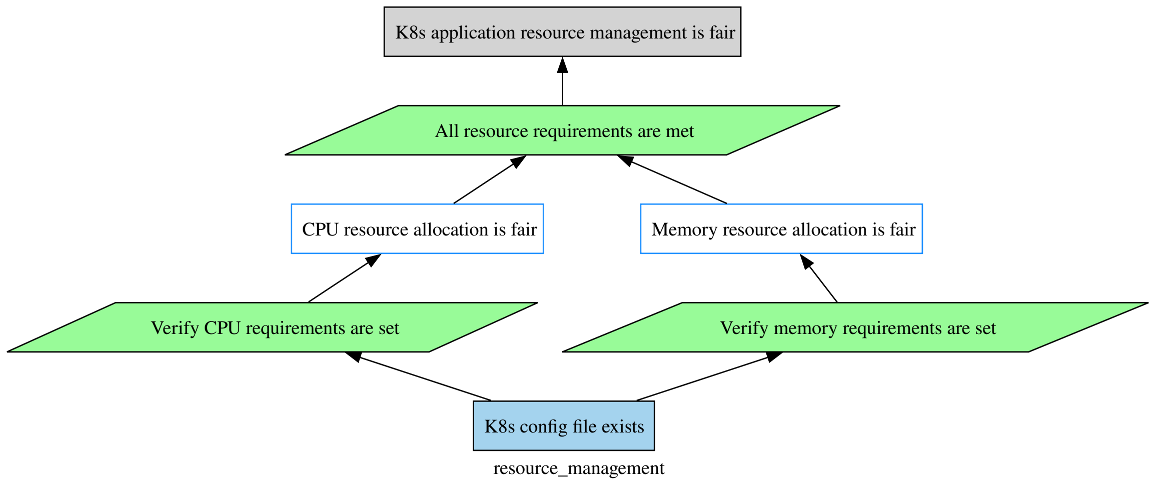

K8s application resource management is fair
All resource requirements are met
CPU resource allocation is fair
Memory resource allocation is fair
Verify CPU requirements are set
Verify memory requirements are set
K8s config file exists
resource_management

Figure 3.8: Justification Diagram for resource management

As illustrated in Figure 3.8, we need a K8s configuration file to begin the check, so the evidence is *"K8s config file exists"*. Then, we need to check that two types of resources are set using two strategies: *"Verify CPU requirements are set"* and *"Verify memory requirements are set"*. After validation, both strategies reach their sub-conclusions, *"CPU resource allocation is fair"* and *"Memory resource allocation is* *fair"*. These sub-conclusions are combined and verified by the strategy *"All resource requirements are met"* for a logical AND operation. Once the strategy is verified, we can conclude that *"K8s application resource management is fair"*.

3.4 Security

Security is a key criterion when assessing K8s system quality. In this section, we focus on two pod security best practices from different perspectives. The first, the security context practice, is a pod-level approach intended to enhance the security of a single pod, and the second, the pod security admission practice, focuses on improving pod security at the cluster namespace level.

3.4.1 Security Context

In Kubernetes, there is a security mechanism, `SecurityContext`, that can be used to apply and enforce certain operating system security settings for pods and containers. The security context can be specified either as a pod-level specification that applies to all containers within that pod or as a per-container specification that only affects a certain container. The security context is considered a best practice to restrict container access to filesystems or OS capabilities, isolating processes in a controlled and limited environment, hence strengthening overall system security. There are various settings supported by the security context, such as user group permissions, filesystem access, and Linux capabilities. Please refer to the Kubernetes security context documentation [15] for detailed specifications. In this practice, we adopt three key security settings to improve pod security: non-root user, non-privileged permission, and read-only filesystems.

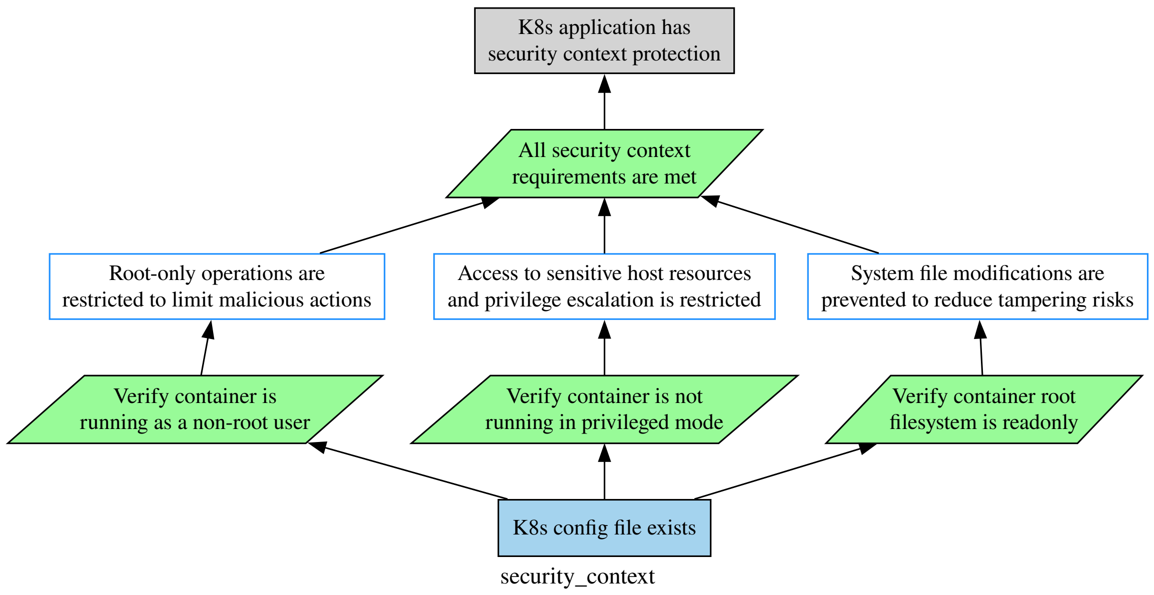

K8s application has security context protection
All security context requirements are met
Root-only operations are restricted to limit malicious actions
Access to sensitive host resources and privilege escalation is restricted
System file modifications are prevented to reduce tampering risks
Verify container is running as a non-root user
Verify container is not running in privileged mode
Verify container root filesystem is readonly
K8s config file exists
security_context

Figure 3.9: Justification Diagram for security context

Therefore, we can conclude that *"K8s application has security context protection"* after verifying these three security settings. As clearly presented in Figure 3.9, three strategies are supported by the evidence *"K8s config file exists"*: the strategy *"Verify container is running as a non-root user"* ensures that a container is not run as root; the strategy *"Verify container is not running in privileged mode"* ensures that no privileged permissions are granted within the container; and the strategy *"Verify container root filesystem is readonly"* restricts write permissions for a container to prevent malicious overwriting of system files. These strategies further lead to their sub-conclusions, and another logical strategy *"All security context requirements are met"* joins these sub-conclusions and yields the final conclusion.

3.4.2 Pod Security Admission

As mentioned at the beginning of this section, the security context is a pod-level security practice that only applies to a single pod, but what if we want a higher-level security approach that enforces security settings across all pods? The answer is *Pod Security Admission* (PSA). It's worth mentioning that the book [5] endorses using the *Pod Security Policy* (PSP) feature to secure pods, but this feature was removed in Kubernetes v1.25 [13] and replaced by the similar but easier-to-use PSA mechanism. PSA works in conjunction with *Pod Security Standards* (PSS), which define three security isolation levels—from loose to strict—namely, Privileged, Baseline, and Restricted. PSA also has three modes that can be used individually or together to enforce these levels: enforce, audit, and warn. For more detailed explanations and specifications of these levels and modes, please refer to the corresponding documentation [16, 17]. In this practice, we focus on adopting PSA for a namespace-level security configuration.

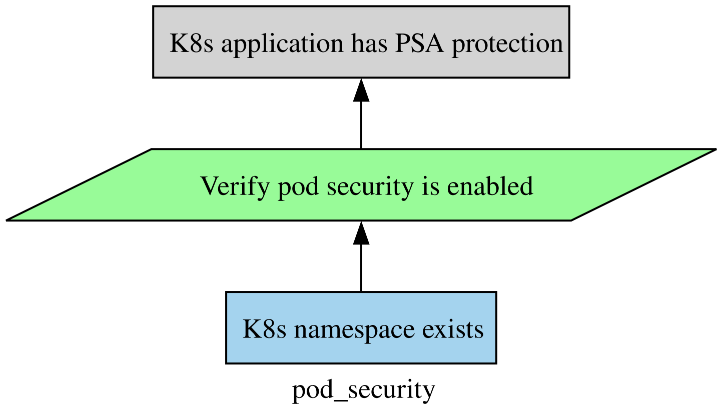

K8s application has PSA protection
Verify pod security is enabled
K8s namespace exists
pod_security

Figure 3.10: Justification Diagram for pod security admission

The justification workflow is exhibited in Figure 3.10. Since PSA is not always configured in K8s configuration files but is sometimes directly labeled in a namespace using kubectl[2], in this case, the evidence is *"K8s namespace exists"*, which checks for the existence of the namespace in a K8s cluster. Next, the strategy *"Verify pod security is enabled"* is applied to ensure that PSA labels are enabled. There are many combinations of PSA and PSS, so we can only check whether this feature is used at the moment. Finally, if the strategy passes, the conclusion *"K8s application has PSA protection"* is reached.

3.5 Conclusion

In this chapter, we explore and analyze ten common best practices for Kubernetes configuration from four distinct categories. Figure 3.11 presents the feature model of these K8s best practices. We have successfully demonstrated the meaning and rationale behind each practice, as well as the methodology for transforming them into Justification Diagrams. This analysis also concludes that despite the size and complexity of a model, the Justification Diagram is fully capable of modeling these practices into coherent and standalone quality checks.

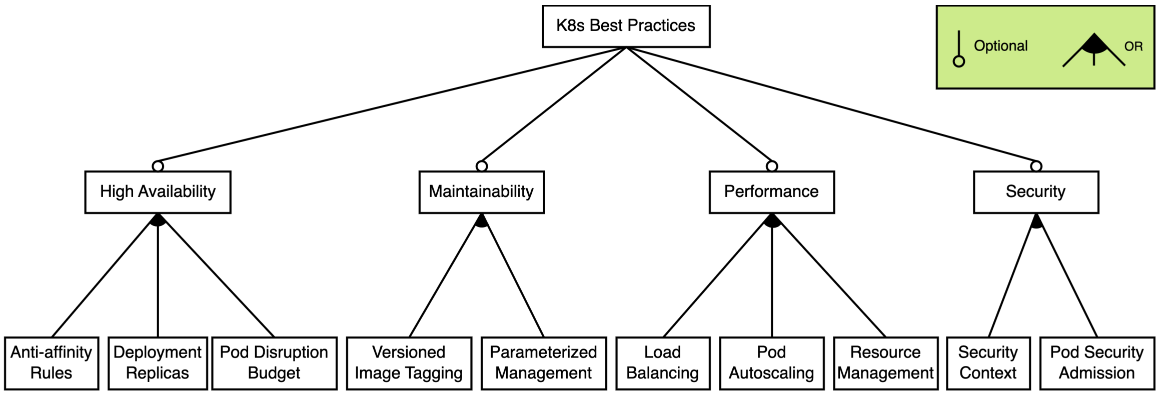


Figure 3.11: Feature model for K8s best practices

[2] `https://kubernetes.io/docs/reference/kubectl/`

Chapter 4

jPipe Runner: A New Operational Justification Framework

In the last chapter, we discussed ten best practices for K8s and transformed them into Justification Diagrams. This chapter aims to answer the second question, "**How does the operational justification framework make JD files executable?**", by introducing a novel keyword-driven justification framework called *jPipe Runner* that renders Justification Diagrams operational and enables easy integration into CI/CD pipelines. First, in Section 4.1, we discuss the problems with the existing operational justification diagram and the motivation behind our new approach. Next, in Section 4.2, we explain the design and architecture of the new framework, and in Section 4.3, we describe its internal technical implementation. Usage is demonstrated through a quick example in Section 4.4 to illustrate the workflow of this new tool. Finally, we conclude with the outcomes of the framework in Section 4.5.

4.1 Problem and Motivation

As discussed in the background chapter, the Justification Diagram provides a visual and transparent representation of a product, offering the ability to trace changes and enhancing confidence throughout the software development process. However, there is a significant gap between the abstract pattern of the JD and its concrete implementation: while the primary focus of the JD is on describing what justification to perform and the rationale linking justification and conclusion, it lacks the capacity to execute the justification process in practice. Therefore, the concept of the Operational Justification Diagram was introduced by Jean-Michel Bruel and later implemented by Deesha Patal in an earlier version of the jPipe compiler [28]. However, the original operational justification diagram has been deprecated and is no longer supported by the latest jPipe compiler due to several key drawbacks:

1. It cannot automatically and explicitly bind the semantic JD components such as evidence and strategy with the corresponding operation functions.
2. It extends the operational capability of the JD only conceptually, without mandating the execution of JD components through a truly executable mechanism.
3. It significantly overcomplicates the JD language syntax by introducing numerous keywords such as `‘probe’`, `‘operation’`, and `‘expectation’`, resulting in maintenance burdens for the compiler itself.

To address the problems of the original operational justification diagram, we are inspired by the Robot Framework[1], an open-source automation framework for acceptance testing, and designed a new operational justification framework, *jPipe Runner*,

[1]`https://robotframework.org/`

independent of the jPipe compiler. This framework has a keyword-driven approach that maps each JD evidence or strategy to a Python function for execution, which strictly follows the justification steps of a JD and aborts the justification process if the result of any evidence or strategy is not satisfied. This approach transforms the Justification Diagram from a purely visual reasoning tool into a fully automated CI/CD-style justification framework.

4.2 Architecture

The jPipe Runner is designed to be a lightweight and out-of-the-box framework that can be run as a command-line tool or easily integrated into a CI/CD pipeline. As depicted in Figure 4.1, the overall architecture of jPipe Runner is plain and simple, consisting of three core components: parser, runtime, and engine.

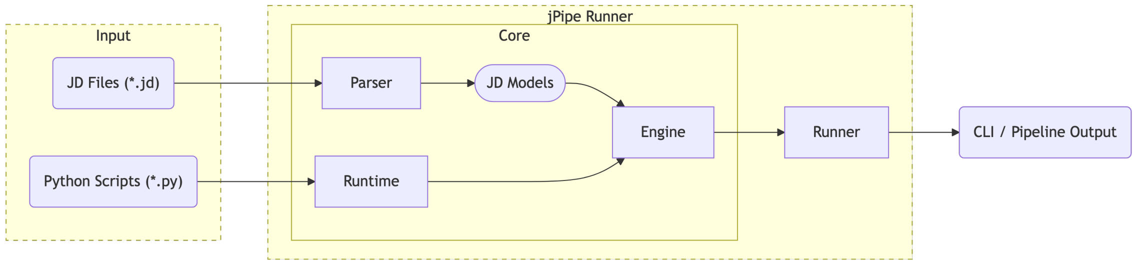


Figure 4.1: Architecture of jPipe Runner

The parser checks the grammar and syntax of a given JD file and parses it into an abstract, machine-readable justification model that can be used within the framework. The runtime, on the other hand, is responsible for linking functions from the input Python library scripts and dynamically setting variables and executing functions. Finally, the engine connects the parser and the runtime. It constructs a directed justification graph based on the abstract justification model produced by the parser, orchestrates the justification order from evidence nodes to the conclusion node by strictly following the justification logic, and calls the runtime to execute the corresponding functions. The framework runner then prints the output, showing every step of the justification process as well as the status of each step. A more detailed example can be found in Section 4.4.

This architecture ensures maximal compatibility with the jPipe compiler and facilitates easy maintenance with minimal manpower.

4.3 Implementation

As we outlined the architecture in the last section, this section aims to demonstrate how these core components are implemented. The jPipe Runner is implemented in Python for flexibility, as by design it requires dynamic loading of external library functions to execute, which makes it impractical to implement in other statically-typed languages like Java or Rust. The framework is open-sourced under an MIT license on the *"jpipe-runner"* repository [24]. For the specific code-related implementation, please refer to the repository. It is worth mentioning that this framework was started as a prototype project and is still in its early development, so the following subsections apply only to the code implementation as of the time of writing, which is version *0.0.1*. In Table 4.1, we summarize the source code structure of the *"jpipe-runner"* project by dividing it into four categories. The implementation of each core component is explained in the following subsections.

Table 4.1: Source code structure of jPipe Runner

Type	**Name**	**Description**
Parser	jpipe.lark	Defines the grammar of the JD language.
	parser.py	Parses JD source code into JD models.
	transformer.py	Transforms Lark AST into custom models.
Runtime	runtime.py	Provides the capability to set global variables and execute external Python functions.
Engine	jpipe.py	Implements core logical justification functionality.
	enums.py	Defines enumerated keywords for the JD models.
	models.py	Contains model definitions for the JD models.
Miscellaneous	exceptions.py	Defines exception classes for error conditions.
	runner.py	Provides the command-line entry point.
	utils.py	Contains helper and utility functions.

4.3.1 Parser

To be maximally compatible with the current Justification Diagram language, the parser needs to fully support all the syntax of JD in the jPipe compiler. Since the language has a custom grammar, a grammar parser is required to parse the JD source code and generate an *Abstract Syntax Tree* (AST) for further use. In the jPipe compiler, two grammar parsers are used for different purposes: ANTLR[2] is originally applied in the compiler and Langium[3] is used to support the *Language Server Protocol* (LSP) for the jPipe VS Code extension. However, neither is applicable or reusable for the jPipe Runner, as they are not supported in Python. Therefore, the jPipe Runner utilizes another grammar parser, Lark[4], to achieve the same functionality. It should be mentioned that using different grammar parsers across projects for

[2]`https://www.antlr.org/`
[3]`https://langium.org/`
[4]`https://github.com/lark-parser/lark`

the same language grammar can lead to inconsistencies and increased maintenance overhead across all jPipe projects; this problem and its possible solution are further discussed in the conclusion and future work chapter.

```
start: model

model: load_stmt* class_def*

load_stmt: "load" STRING

class_def: CLASS_TYPE ID ("implements" ID)? (justification_pattern |
↪  composition)

justification_pattern: "{" (variable | instruction | support )+ "}"
variable: VARIABLE_TYPE ID instruction
instruction: "is" STRING
support: ID "supports" ID

VARIABLE_TYPE: "evidence"
               | "strategy"
               | "sub-conclusion"
               | "conclusion"
               | "@support"

CLASS_TYPE: "justification"
            | "pattern"
            | "composition"
```

Listing 4.1: Source code of JD Lark grammar

To implement a JD language parser in Python, the first step is to define the grammar using Lark. Listing 4.1 shows part of the definition of the Justification Diagram expression in Lark. The expression is derived from the Langium version of the JD expression in the jPipe compiler, which treats each curly-bracket-closed block as a class definition, supporting three class types: '`justification`', '`pattern`', and '`composition`'. Then, each line in the justification pattern class is treated as one of two types. The first type is variable assignment, which binds an instruction to a variable by assigning it an ID. Five variable types are supported, including '`evidence`', '`strategy`', '`sub-conclusion`', '`conclusion`', and the abstract support '`@support`'. The second type is relationship definition, which defines the support relationship between two IDs. Other directives, such as the '`load`' statement, are also supported to ensure compatibility with the existing jPipe compiler.

```
jpipe_parser = Lark(grammar=JPIPE_GRAMMAR,
                    start='start',
                    parser='lalr')

def parse_jd(source: str) -> ModelDef:
    try:
        tree: ParseTree = jpipe_parser.parse(text=source)
        model: ModelDef = JPipeTransformer().transform(tree)
        return model
    except (UnexpectedCharacters, UnexpectedToken) as e:
        raise SyntaxException(
            'parse error: invalid JD source code') from e
```

Listing 4.2: Source code of JD parser function

Once the JD grammar is defined in Lark, a transformer class is used to convert the AST of the Justification Diagram source code, as parsed by the Lark parser, into Python data types. For a detailed transformer implementation, please refer to the `JPipeTransformer` class in the `transformer.py` code file. Finally, a parse function, as shown in Listing 4.2, combines these two modules and returns the JD model that is recognized internally as Python data classes.

4.3.2 Runtime

The runtime is a crucial component that grants the jPipe Runner executability within the framework. The current implementation of the runtime is a thin wrapper around an abstract runtime interface that primarily defines three key methods:

1. **load_files**: This method loads all the given external Python files into the runtime instance. The core functionality is implemented in a private method, `_import_file`, as shown in Listing 4.3. Each Python file is treated as a Python module, imported via `importlib`, and stored in a private `_modules` variable. However, loading executable files can pose security risks, so we will discuss potential solutions in the Future Work section of Chapter 6.

2. **set_variable**: This method sets the module-level variable using the built-in `setattr` function for each module loaded from the `load_files` method. It has a variant method, `set_variable_literal`, which converts the variable's value into a Python literal structure via the `ast.literal_eval` function, thereby extending the variable data types.

3. **call_function**: When the jPipe Runner engine starts a justification process and needs to validate an evidence or strategy, it calls the runtime's `call_function` method to execute the corresponding function for that evidence or strategy.

4.3.3 Engine

As a core component of the jPipe Runner framework, the engine is typically responsible for constructing justification graphs and executing justification processes. There

```python
def _import_file(self, file_path: str) -> None:
    if not os.path.isfile(file_path):
        raise FileNotFoundError(f"File not found: {file_path}")
    module_name, _ = os.path.splitext(
        os.path.basename(file_path))
    spec = importlib.util. \
        spec_from_file_location(module_name, file_path)
    module = importlib.util. \
        module_from_spec(spec)
    spec.loader.exec_module(module)
    self._modules.append(module)
```

Listing 4.3: Source code of runtime import method

are two essential classes in the engine's implementation: the `Justification` class and the `JPipeEngine` class.

The `Justification` class inherits from the `DiGraph`[5] class of the NetworkX [9] package, a Python package for building and operating complex graph networks. This class has two key methods: `validate` and `layered_traverse`. The `validate` method checks the validity of a justification graph—a `Justification` instance represented as a special directed graph—by strictly enforcing the following constraints:

1. The justification graph must be a *Directed Acyclic Graph* (DAG).
2. There must be only one conclusion node, and it should not have any child nodes.
3. Evidence nodes must have an in-degree of zero and point only to strategy nodes.
4. Strategy nodes must point to only one sub-conclusion or conclusion node.
5. Sub-conclusion nodes can only point to strategy nodes.

[5]`https://networkx.org/documentation/stable/reference/classes/digraph.html`

The `layered_traverse` method is used to iterate the justification graph nodes in a layered order. For example, in a justification graph, traversal must start with all evidence nodes regardless of their order, and a strategy node must be iterated only after all of its incoming nodes (such as evidence or sub-conclusion nodes) have been iterated, thereby ensuring the logical correctness of the justification process.

On the other hand, the `JPipeEngine` class combines all the components in the framework into a cohesive whole. First, it calls the `load_jd_file` function from the parser module and converts the JD models into one or more justification graphs. Next, it checks the validity of the justification graphs by calling the `validate` method provided in the Justification class. Finally, when running the justification process, it acquires the justification order by calling the `layered_traverse` method and verifies each justification node in that order within a for-loop on a *keyword-driven* basis. For example, for an evidence or strategy node, it extracts the keyword from that node and converts it into a function name in snake case format, such as **snake_cased** ("*Check contents w.r.t. NDA*") $\Rightarrow$ "*check_contents_wrt_nda*". Then, it calls the runtime's `call_function` method to execute this function by its name and checks if the function returns a non-false result; otherwise, its successor nodes will be skipped and the process will be aborted and marked as failed.

4.4 Example

In this section, we use a quick example to introduce how to use jPipe Runner in CI/CD pipelines, demonstrating the capability and flexibility provided by this operational justification framework. Figure 4.2 presents a simple Justification Diagram for justifying the professionalism of slides.

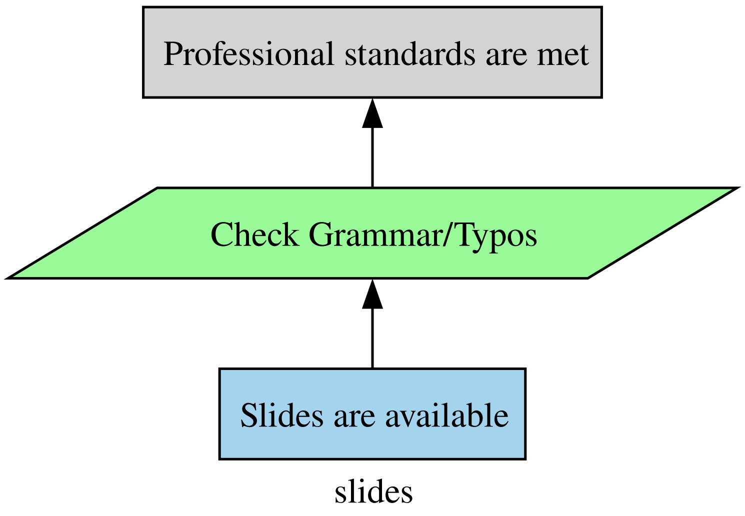


Figure 4.2: Justification Diagram for slides professionalism

First, to justify the Justification Diagram in Figure 4.2, we need to implement two functions—`slides_are_available` and `check_grammar_typos`—representing the behavior for the evidence and the strategy, respectively. jPipe Runner converts the keywords of all evidence and strategy nodes in that diagram into functions with snake cased names, reflecting the keyword-driven principle of the operational framework. Table 4.2 shows the corresponding keywords mapped to function names in snake case.

Table 4.2: Keywords to mapped functions

No.	**Type**	**Keyword**	**Mapped Function**
1	Evidence	*Slides are available*	`slides_are_available`
2	Strategy	*Check Grammar/Typos*	`check_grammar_typos`
3	Conclusion	*Professional standards are met*	/

Second, a Python script named ‘`slides.py`’ needs to be created to implement the specific functions within. As shown in Listing 4.4, all functions should accept zero arguments and return a boolean-like value indicating the result status of the corresponding function. A global variable '`available`' is declared at the top of the example code; it is used to configure the required value for different cases and is set by jPipe Runner dynamically when executing the justification process.

```
# This var will be set from CLI
available = None

def slides_are_available() -> bool:
    return available is not None

def check_grammar_typos() -> bool:
    # Do some real grammar/typo checks
    return True
```

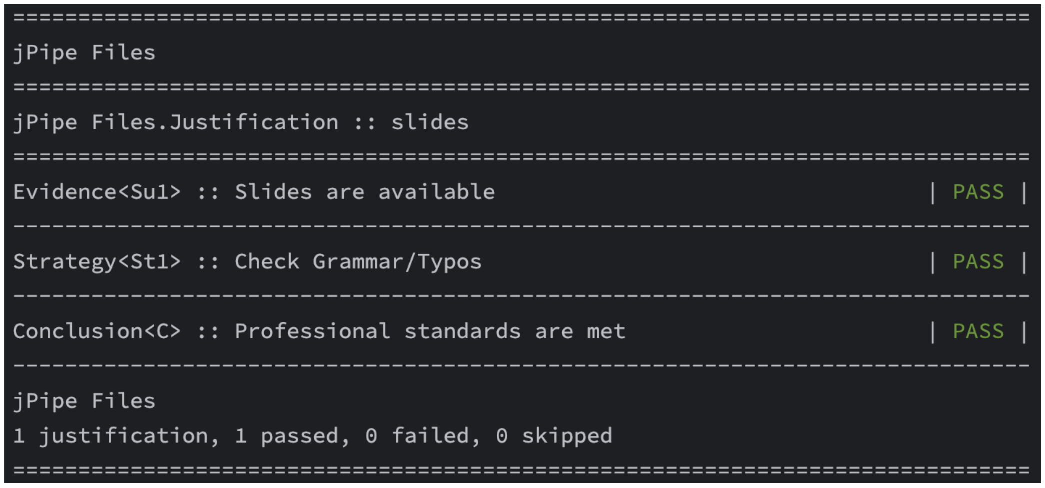
Listing 4.4: Example source code for slides check functions

Next, in order to use jPipe Runner in the GitHub Workflow pipeline, the YAML workflow configuration shown in Listing 4.5 must be set with the correct options. The '`jd_file`' option specifies which Justification Diagram file is to be used in the justification step, and the '`variable`' and '`library`' options are responsible for providing the targeted Python scripts with the required variables as input arguments.

```
  - name: Justify Slides
    uses: ace-design/jpipe-runner@main
    with:
      jd_file: "./slides.jd"
      variable: |
        available:ready
      library: |
        ./slides.py
```

Listing 4.5: Example source code for slides check workflow

Finally, a detailed table-based output is printed by jPipe Runner in the CLI or a CI/CD pipeline, as shown in Figure 4.3. The justification process strictly follows the logical justification order, starting from evidence and eventually reaching the conclusion step by step. The `‘PASS’` status indicates the success of a justification step; if any step fails with a `‘FAIL’` status, the remaining unreached steps will be marked as `‘SKIP’`, thereby yielding a complete record of the justification process.



```
================================================================================
jPipe Files
================================================================================
jPipe Files.Justification :: slides
================================================================================
Evidence<Su1> :: Slides are available                                   | PASS |
--------------------------------------------------------------------------------
Strategy<St1> :: Check Grammar/Typos                                    | PASS |
--------------------------------------------------------------------------------
Conclusion<C> :: Professional standards are met                         | PASS |
--------------------------------------------------------------------------------
jPipe Files
1 justification, 1 passed, 0 failed, 0 skipped
================================================================================
```

Figure 4.3: Example output of jPipe Runner

4.5 Conclusion

The jPipe Runner framework demonstrates its capability to replace the existing operational justification diagram and serve as a novel approach for justifying workflows in a programmable and keyword-driven manner. The simplicity and flexibility it offers may significantly change the current usage of the Justification Diagram in CI/CD pipelines and provide insights for future operational justification studies.

Chapter 5

Case Study of K8s Best Practices in Mastodon Helm Chart

As explored in Chapter 3, we examined ten best practices of K8s and demonstrated how to transform them into Justification Diagrams. However, these best practices are sometimes added or removed from a K8s configuration, making it difficult to track and analyze these changes and their impact on specific K8s deployments. Therefore, this chapter analyzes the evolution of real-world Kubernetes configurations and demonstrates **how Justification Diagrams can be used with these best practices to enhance the quality of a K8s system** through a comprehensive case study.

First, we discuss the objectives of this case study and its corresponding methodology in Section 5.1. Next, Sections 5.2 and 5.3 present observations and key findings from the Mastodon Helm charts. Section 5.4 provides a detailed demonstration of using Justification Diagrams to represent best practices in the Mastodon Helm chart and how they integrate with jPipe Runner to track changes and enhance Kubernetes configuration quality. Finally, we conclude this case study in Section 5.5.

5.1 Objectives and Methodology

Adapting Helm to parameterize Kubernetes configurations and standardize package management is a good practice for improving the overall quality of a K8s system from a maintenance perspective. However, Helm charts can grow rapidly during project development and regular maintenance. The templates for Helm charts also evolve constantly to accommodate new best practices by updating resources and options in the K8s application. Therefore, continuously tracking and validating these changes in each Helm version through Git commits is a crucial task for maintaining the K8s application in a stable and manageable state. The main objective of this case study is to explore how Justification Diagrams can address this problem. By leveraging the composition and decomposition ability of Justification Diagrams, we can justify each version update in the Helm chart and integrate this process with jPipe Runner into a CI/CD pipeline to automate the tracking and validation, ultimately enhancing the quality of Kubernetes systems.

In this case study, we primarily focus on best practice-related uses and changes in K8s configurations maintained with Helm charts. By comparing various popular open-source projects that use Helm to manage their K8s deployments, we eventually choose the Mastodon Helm chart for this case study for the following reasons:

1. Mastodon is a popular open-source platform with a large community, having over 900K monthly active users and more than 8K servers[1] as of now.

2. It has over 200 Git commits in its Helm chart history, indicating active updates, continuous development, and ongoing maintenance. Therefore, we can extract

[1]`https://joinmastodon.org/about`

and analyze useful information from its Helm chart codebase.

3. The Helm chart of Mastodon has undergone several major refactorings. Such significant changes can be a positive sign of an active community, responsive development, and the adoption of state-of-the-art features in its configurations, providing valuable information for our case study.

We conduct and structure this case study into the following essential steps to ensure an accurate analysis and a smooth transition to Justification Diagrams.

- Step 1: Retrieve all relevant Helm charts of Mastodon and extract detailed commit information.
- Step 2: Generate a set of complete K8s configurations for all commits using their default `values.yaml` settings.
- Step 3: Iterate through each configuration to document and analyze the applied best practices of Kubernetes.
- Step 4: Compare each configuration with its previous commit to identify best practice-related updates and transform them into semantic explanations.
- Step 5: Use the Justification Diagram to represent the justification for each change documented in the previous step.

In Steps 1 and 2, we used a Python script with the GitPython[2] library to extract Git commit information from the cloned Git repositories on our local machine and generated K8s configurations using the `helm template` command, respectively.

[2]`https://gitpython.readthedocs.io`

It is worth mentioning that generating configurations is necessary to facilitate the comparison of each commit or version, as it merges all Helm template resources and configurations into a single final YAML configuration. This approach significantly reduces the workload by eliminating the need to examine various template changes across multiple locations within a single commit. In Steps 3 and 4, we first created a script using Kube-Linter and a YAML parser to automate the best practice compliance check for each configuration. Then, we manually performed a double-check and conducted an in-depth analysis of each configuration and its corresponding commit, documenting the results in CSV sheets available in the *"k8s-best-practices"* repository under `case-studies/mastodon` directory [25]. Finally, in Step 5, we organized and presented these details in Justification Diagrams.

5.2 Observations and Analysis

The Helm chart for Mastodon involves four GitHub repositories and over 200 commits in total. Our primary focus is on best practice-related changes, such as adding a *Pod Disruption Budget* (PDB) resource to the chart template or using a Load Balancer for deployment. Therefore, instead of reviewing each commit individually and analyzing the differences from its previous commits, we used automation tools to identify and label the implementation of each practice in every commit and manually verified their correctness afterward, which significantly reduced repetitive work. Next, we only needed to examine the commits and their neighboring commits that showed different practice applications, such as adding or removing a certain practice. This approach allowed us to focus on extracting the meaning and reasoning behind these commits and representing these changes in Justification Diagrams.

We observed and analyzed four relevant repositories—*Ladicle/mastodon-chart*[3], *johnschultz/helmadon*[4], *mastodon/mastodon*[5], and *mastodon/chart*[6]—and their updates from April 2017 to February 2025. The Helm chart underwent significant refactoring during this period and has now become stable and consistent, making it an ideal study case for analyzing practice evolution.

5.3 Key Findings from Observations

After analyzing the Mastodon Helm chart, we have identified several key findings throughout its evolution. In this section, we discuss these findings and the key events in its evolution, along with their underlying meaning and rationale. These documented changes provide valuable insights for constructing Justification Diagrams in the next section.

The evolution of the Mastodon Helm chart has three major stages, during which it underwent significant refactoring and updates at each stage transition. The first stage occurred before Mastodon officially supported deploying its services via a Kubernetes Helm chart. In April 2017, two repositories, *Ladicle/mastodon-chart* and *johnschultz/helmadon*, individually added support for the Mastodon Helm chart. Their development processes were primarily concentrated in that month and stopped development afterward, resulting in their Helm chart versions remaining at v0.1.0 without further updates.

The first repository initially implemented three best practices: versioned image tagging, load balancing, and resource management. It later adopted deployment

[3]`https://github.com/Ladicle/mastodon-chart`
[4]`https://github.com/johnschultz/helmadon`
[5]`https://github.com/mastodon/mastodon`
[6]`https://github.com/mastodon/chart`

replicas in commit `0fce8fc`. The other repository, by contrast, initially adopted only the resource management practice during its early development stage. A load balancer was introduced after five commits for the Mastodon web service. Subsequently, the author started using versioned Mastodon images in the `values.yaml` configuration, improving the maintainability of the Helm chart.

In June 2020, the second stage began when an *"Add Helm Chart"* pull request[7] was opened and merged into the Mastodon source repository, officially supporting the Helm chart under the `chart/` directory. This pull request was built on the two previous individual Mastodon Helm charts and implemented additional best practices, including anti-affinity rules, *Horizontal Pod Autoscaler* (HPA), and pod security context. Subsequently, the Helm chart entered a regular development and maintenance process, upgrading from version v0.1.0 to v2.3.0. No best practice-related changes were made during this period, and most modifications mainly focused on functional improvements in the chart, such as database migrations and S3 storage support.

After two years of development and maintenance, the third stage started with the upgrade of the Mastodon Helm chart from v2.3.0 to v3.0.0 in November 2022, including a major refactor that optimized deployments and environment variables through a pull request[8]. However, this version was soon deprecated, and the entire Helm chart was relocated to a separate *mastodon/chart* repository, where commits from v0.1.0 to v2.3.0 were cherry-picked into the new repository.

Notably, in the new Helm chart repository, v3.0.0 was completely removed, and v4.0.0 directly followed v2.3.0 without a clear explanation. It appears that v3.0.0

[7] `https://github.com/mastodon/mastodon/pull/14090`
[8] `https://github.com/mastodon/mastodon/pull/20733`

and v4.0.0 shared some common changes, as both versions removed the HPA practice. However, the HPA resource was not removed because it was considered an inefficient practice but rather because it was broken and had been ineffective during previous development, as indicated by the commit message[9]. Although the contributor mentioned reinstating the HPA practice in future versions, it remains unavailable at the time of writing. After v4.0.0, the Mastodon Helm chart underwent a smooth and regular development and maintenance process, with the PDB practice added for the Mastodon web and streaming services in v5.1.1. The Helm chart evolved stably afterward without major practice-related changes.

From the above analysis and findings, it is evident that maintaining and tracking updates to K8s best practices can be challenging. Developer teams may sometimes forget which practices they have changed in their Helm chart, resulting in issues such as failing to reinstate the HPA after its removal. Therefore, we need to explore how Justification Diagrams can be used to continuously track and reflect on changes during configuration updates, ultimately enhancing the quality of the K8s system.

5.4 Justification for Mastodon K8s Configuration

Justification Diagrams can be used to represent K8s best practices applied in each version of the Mastodon Helm chart. These diagrams illustrate which practices are implemented in a specific Helm chart and the outcomes they produce. They can also be composed or decomposed to reflect changes in applied practices throughout the evolution of the Mastodon Helm chart by adding or removing specific nodes. The analysis documented in the *"case-studies/mastodon"* sheets [25] lists key transitions

[9]`https://github.com/mastodon/mastodon/commit/cddcafe`

in these charts and their intended purposes. In the following subsections, we demonstrate the potential benefits of integrating Justification Diagrams and jPipe Runner into the evolution of the Mastodon Helm chart.

5.4.1 Modular Practice Diagrams

When implementing a Helm chart for Mastodon applications, the best practices applied in the chart are often unclear and difficult to track. The primary benefit of using the Justification Diagram is the ability to easily represent and modularize practice diagrams, visualizing the rationale behind the overall K8s configuration.

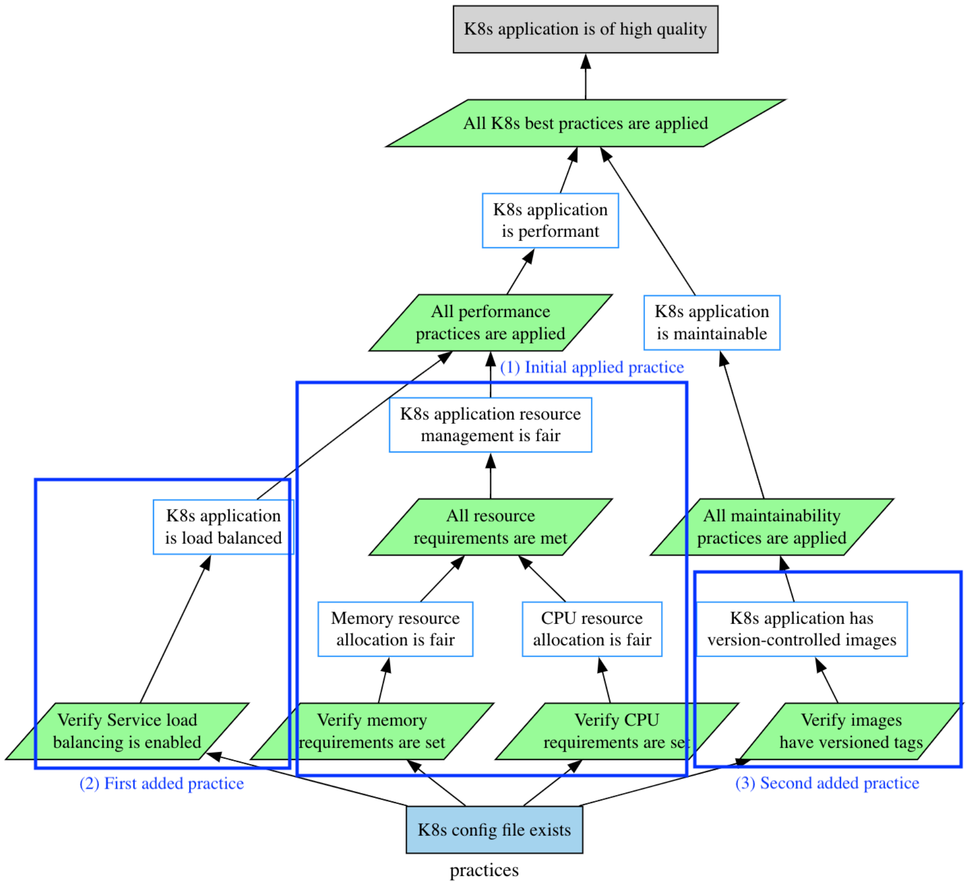

K8s application is of high quality
All K8s best practices are applied
K8s application is performant
All performance practices are applied
K8s application is maintainable
(1) Initial applied practice
K8s application resource management is fair
K8s application is load balanced
All resource requirements are met
All maintainability practices are applied
Memory resource allocation is fair
CPU resource allocation is fair
K8s application has version-controlled images
Verify Service load balancing is enabled
Verify memory requirements are set
Verify CPU requirements are set
Verify images have versioned tags
(2) First added practice
(3) Second added practice
K8s config file exists
practices

Figure 5.1: Justification Diagram for applied practices in *johnschultz/helmadon*

For example, the goal of applying practices from different domains is to deploy high-quality K8s applications. As shown in Figure 5.1, the initial Justification Diagram of applied practices in the *johnschultz/helmadon* Helm chart represents a configuration where only resource management was initially applied to the Mastodon web application. In subsequent updates, load balancing was added as an additional performance-related best practice, followed by versioned image tagging as the next applied best practice to enhance maintainability. This change can be clearly represented in the Justification Diagram by merging the corresponding practice diagram with the existing one. While we can only manually merge Justification Diagrams for now, the upcoming version of the jPipe compiler will support justification merge statements, as shown in Listing 5.1, to facilitate this process.

```
composition {
    justification final_practices is merge(load_balancing, practices) {
        // merge options...
    }
}
```

Listing 5.1: Example code for Justification Diagram merging

With Justification Diagrams, we can easily represent which practices are applied in the Mastodon Helm chart as well as the purpose and effectiveness of each practice. This approach can significantly help developer teams track and document practice changes in K8s configurations.

5.4.2 Pipeline-Automated Checks

Through our case study on the Mastodon Helm chart, we recognize that relying solely on Justification Diagrams to document and track changes is somewhat insufficient.

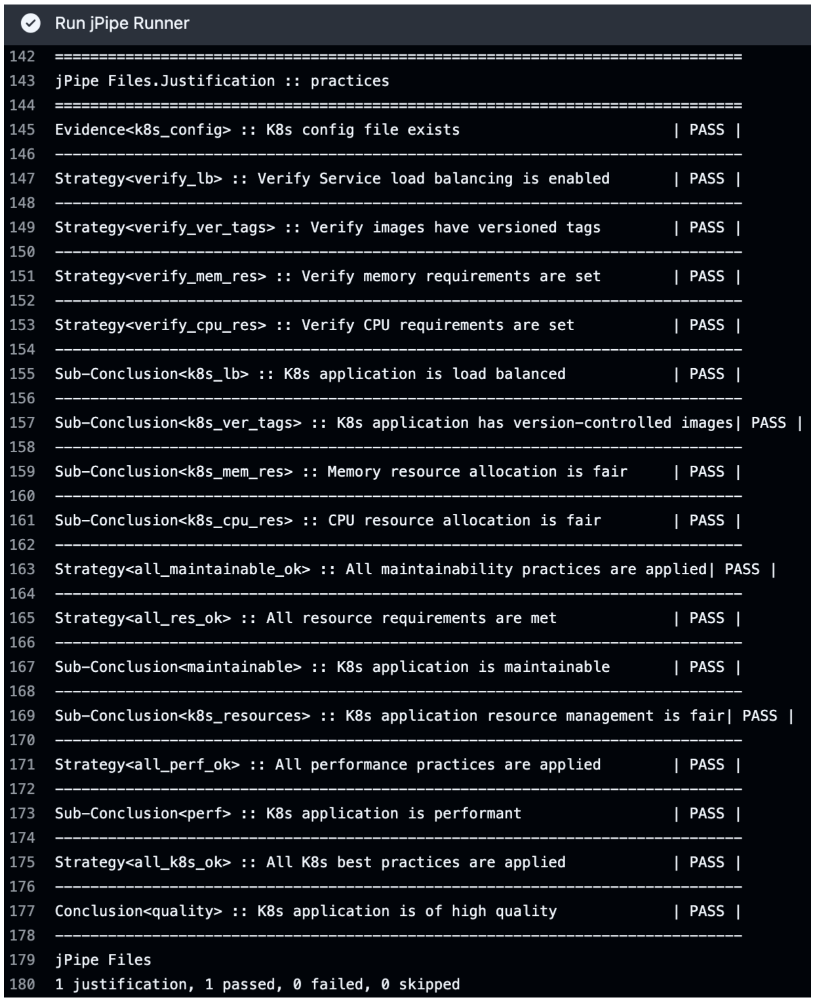
It is also important to ensure that each Git commit complies with the best practices represented in the diagrams. Therefore, jPipe Runner is necessary to operationalize the best practice diagrams and ensure that each practice is properly applied in the Mastodon K8s configuration. Furthermore, we can integrate jPipe Runner into a designated CI/CD pipeline to incorporate K8s best practice checks and automate the justification process.

```yaml
jobs:
  check-mastodon-k8s-practices:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.13"
      - name: Install Python dependencies
        run: |
          pip install -r requirements.txt
      - name: Set up kube-linter
        uses: ./.github/actions/setup-kube-linter
        with:
          kube-linter-version: latest
      - name: Run jPipe Runner
        uses: ace-design/jpipe-runner@main
        with:
          jd_file: ${{ github.event.inputs.jd_file || env.DEFAULT_JD_FILE }}
          variable: |
            k8s_config_path:${{ github.event.inputs.k8s_config_path ||
             ↪  env.DEFAULT_K8S_CONFIG_PATH }}
          library: |
            python-libraries/*.py
            case-studies/mastodon/python-libraries/*.py
```

Listing 5.2: GitHub workflow for automated Mastodon K8s practice checks [25]

In Listing 5.2, we demonstrate the use case of jPipe Runner to automate best practice checks for a targeted Justification Diagram in GitHub workflow pipelines.

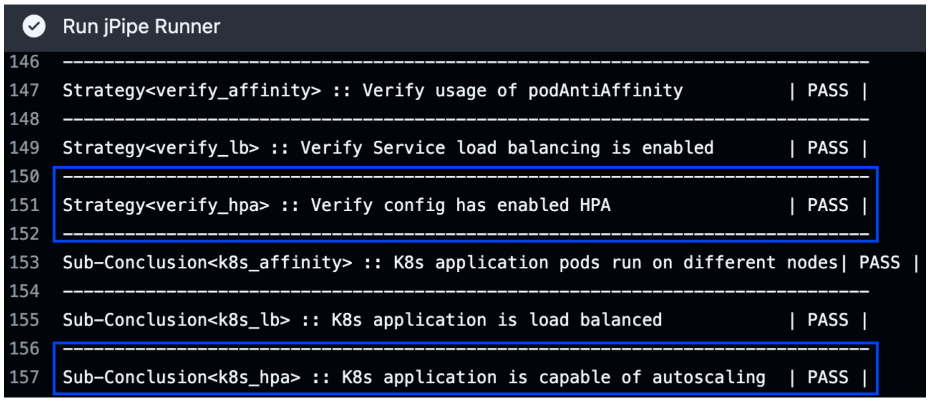
The workflow configuration is structured into two simplified parts: one part sets up the necessary environments and tools, and the other runs the jPipe Runner action.



```
==============================================================================
jPipe Files.Justification :: practices
==============================================================================
Evidence<k8s_config> :: K8s config file exists                        | PASS |
------------------------------------------------------------------------------
Strategy<verify_lb> :: Verify Service load balancing is enabled       | PASS |
------------------------------------------------------------------------------
Strategy<verify_ver_tags> :: Verify images have versioned tags        | PASS |
------------------------------------------------------------------------------
Strategy<verify_mem_res> :: Verify memory requirements are set        | PASS |
------------------------------------------------------------------------------
Strategy<verify_cpu_res> :: Verify CPU requirements are set           | PASS |
------------------------------------------------------------------------------
Sub-Conclusion<k8s_lb> :: K8s application is load balanced            | PASS |
------------------------------------------------------------------------------
Sub-Conclusion<k8s_ver_tags> :: K8s application has version-controlled images| PASS |
------------------------------------------------------------------------------
Sub-Conclusion<k8s_mem_res> :: Memory resource allocation is fair     | PASS |
------------------------------------------------------------------------------
Sub-Conclusion<k8s_cpu_res> :: CPU resource allocation is fair        | PASS |
------------------------------------------------------------------------------
Strategy<all_maintainable_ok> :: All maintainability practices are applied| PASS |
------------------------------------------------------------------------------
Strategy<all_res_ok> :: All resource requirements are met             | PASS |
------------------------------------------------------------------------------
Sub-Conclusion<maintainable> :: K8s application is maintainable       | PASS |
------------------------------------------------------------------------------
Sub-Conclusion<k8s_resources> :: K8s application resource management is fair| PASS |
------------------------------------------------------------------------------
Strategy<all_perf_ok> :: All performance practices are applied        | PASS |
------------------------------------------------------------------------------
Sub-Conclusion<perf> :: K8s application is performant                 | PASS |
------------------------------------------------------------------------------
Strategy<all_k8s_ok> :: All K8s best practices are applied            | PASS |
------------------------------------------------------------------------------
Conclusion<quality> :: K8s application is of high quality             | PASS |
------------------------------------------------------------------------------
jPipe Files
1 justification, 1 passed, 0 failed, 0 skipped
```

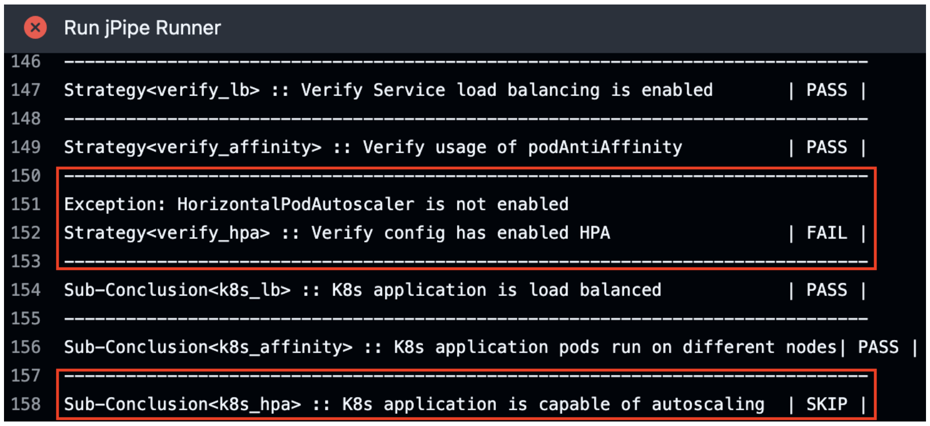
Figure 5.2: jPipe Runner output for Mastodon K8s configuration workflow

The implementation of each best practice check is open-sourced in the *"k8s-best-practices"* repository [25] under the `python-libraries` directory. Similar to the modular Justification Diagrams used for K8s best practices, these Python-implemented practice checks are also reusable and can be easily integrated into the pipeline without additional modifications. Figure 5.2 shows the workflow output of jPipe Runner validating the Mastodon K8s configuration, indicating that these best practices have been properly followed from commit `28bd9df` to `fcecc39`.



```
Run jPipe Runner
--------------------------------------------------------------------------------
Strategy<verify_affinity> :: Verify usage of podAntiAffinity          | PASS |
--------------------------------------------------------------------------------
Strategy<verify_lb> :: Verify Service load balancing is enabled       | PASS |
--------------------------------------------------------------------------------
Strategy<verify_hpa> :: Verify config has enabled HPA                 | PASS |
--------------------------------------------------------------------------------
Sub-Conclusion<k8s_affinity> :: K8s application pods run on different nodes| PASS |
--------------------------------------------------------------------------------
Sub-Conclusion<k8s_lb> :: K8s application is load balanced            | PASS |
--------------------------------------------------------------------------------
Sub-Conclusion<k8s_hpa> :: K8s application is capable of autoscaling  | PASS |
```

(a) Before the removal of HPA



```
Run jPipe Runner
--------------------------------------------------------------------------------
Strategy<verify_lb> :: Verify Service load balancing is enabled       | PASS |
--------------------------------------------------------------------------------
Strategy<verify_affinity> :: Verify usage of podAntiAffinity          | PASS |
--------------------------------------------------------------------------------
Exception: HorizontalPodAutoscaler is not enabled
Strategy<verify_hpa> :: Verify config has enabled HPA                 | FAIL |
--------------------------------------------------------------------------------
Sub-Conclusion<k8s_lb> :: K8s application is load balanced            | PASS |
--------------------------------------------------------------------------------
Sub-Conclusion<k8s_affinity> :: K8s application pods run on different nodes| PASS |
--------------------------------------------------------------------------------
Sub-Conclusion<k8s_hpa> :: K8s application is capable of autoscaling  | SKIP |
```

(b) After the removal of HPA

Figure 5.3: Comparison of jPipe Runner output before and after HPA removal

However, as mentioned in the previous section, HPA was removed during the third stage of the Mastodon Helm chart's evolution but was intended to be reinstated. We present a comparison of the workflow output before and after the commits that removed HPA in Figure 5.3, where we can clearly notice that HPA is no longer enabled in the configuration and the corresponding strategy has failed. This allows us to understand the impact of such removal, as the Mastodon application is no longer capable of autoscaling, which may affect the overall K8s application performance.

Therefore, if this approach is applied to the entire evolution of Mastodon's K8s Helm chart, it can significantly improve configuration development and maintenance. This is achieved by not only helping developers track configuration and best practice changes but also explaining why it should be adopted and what consequences may arise if it is not adopted, through automated pipeline checks.

5.5 Conclusion

After examining over 200 commits across four Helm chart repositories, we studied the evolution of practice usage in the Mastodon Helm chart and analyzed the reasoning and meaning behind these changes. Based on this case study, we demonstrated that the Justification Diagram is a valuable and practical tool for tracking practice-related changes within the Helm chart and capturing the intentions behind each update. Furthermore, by integrating jPipe Runner into the chart repository to operationalize Justification Diagrams, this approach can further automate the tracking and validation of changes for Helm charts in CI/CD pipelines, and ultimately enhance the overall quality of the Kubernetes system.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize the results of this project and discuss future work.

6.1 Summary

In conclusion, this report demonstrates the effectiveness of using the Justification Diagram together with jPipe Runner to enhance the quality of Kubernetes configuration within CI/CD pipelines. By incorporating various best practice quality checks, this approach enables developers to effectively identify possible issues and threats in K8s configurations. Moreover, it provides a holistic methodology for constructing a unified, automated workflow—from specifying the justification to executing the practice—with valuable observations in Kubernetes system quality research.

Our study began by addressing the growing need for identifying and reducing potential K8s misconfigurations and for achieving robustness and reliability in K8s applications. In Chapter 3, we examined ten K8s best practices from four distinct domains, demonstrating the capability of the Justification Diagram to represent and visualize various K8s best practices. We also showed how to extract and transform these best practices from semantic content into Justification Diagrams, providing the foundation for the subsequent implementation of the K8s quality checks. In Chapter 4, we introduced a new operational justification framework that replaced the existing operational justification diagram and elevated the concept of justification operations by enabling keyword-driven and programmable features in the jPipe Runner framework, thereby filling the gap between plain Justification Diagrams and programmed, functioning justification operations. Finally, we studied Mastodon's open-source K8s configurations by tracking the evolution of its best practice usage using Justification Diagrams in Chapter 5. This demonstrates the practicality and alignment of our work, highlighting its potential to provide a comprehensive justification for K8s configurations and enhance the overall quality of Kubernetes systems.

6.2 Future Work

In this section, we discuss potential future work to enhance our approach from two perspectives: K8s best practices and the jPipe Runner framework.

6.2.1 K8s Best Practices

The best practices selected for this project primarily focus on the breadth of K8s configurations but lack depth in each area due to the limited scope and the inherent complexity of Kubernetes. Therefore, more in-depth best practices should be gathered and analyzed to enhance our approach. For example, in the context of high availability, further exploration of practices related to pod liveness and readiness checks is recommended, as these mechanisms determine an application's health status, affect its restart policies and traffic management, and ultimately impact the overall reliability of a Kubernetes deployment.

Additionally, while the selected best practices apply to general cases, some of them may not be suitable for certain edge scenarios. For example, deployment replicas may not always be advisable for stateful applications such as databases and message queues because they could cause data consistency issues and access conflicts. Therefore, further detailed studies and clear guidelines on selecting best practices for specific use cases are needed to refine our approach and cover a broader range of scenarios.

6.2.2 jPipe Runner Framework

The jPipe Runner framework is a promising tool for implementing justification operations, but it is still a prototype experimental project at present. There are several areas that need to be further improved. The current keyword-driven approach converts the name of an element of a justification diagram into a snake-cased function name. While this works in most cases, it can become a problem if the label contains an extremely long description, resulting in an unwieldy function name that affects both readability and conciseness. A more sophisticated keyword-to-function mapping mechanism should be supported in the framework, such as using a decorator to explicitly bind a function to a specific justification element.

To address the security concerns raised by the `load_files` method in the runtime implementation section of Chapter 4, a sandbox mechanism or restrictions on the permissions granted to the loaded code could be advisable and valuable solutions for isolating code execution from the main runner process and enhancing the security of the jPipe Runner's justification process.

Furthermore, as discussed in the parser implementation section of Chapter 4, at least three different grammar parsers are used across the jPipe projects. This inconsistency can lead to significant maintenance challenges between jPipe grammar implementations. Standardizing the grammar parser into a single implementation is necessary to mitigate these issues. For example, the jPipe compiler could export a justification diagram in JSON format and pass it to the jPipe Runner, which may provide a feasible solution to avoid redundant grammar definitions across projects.

Bibliography

[1] Pedro Ascensão, Luís Filipe Neto, Karima Velasquez, and David Perez Abreu. 2024. Assessing kubernetes distributions: A comparative study. In *2024 IEEE 22nd Mediterranean Electrotechnical Conference (MELECON)*. IEEE, 832–837.

[2] Andreas Brunnert, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, et al. 2015. Performance-oriented DevOps: A research agenda. *arXiv preprint arXiv:1508.04752* (2015).

[3] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: up and running.* " O'Reilly Media, Inc.".

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue* 14, 1 (2016), 70–93.

[5] Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson. 2023. *Kubernetes Best Practices.* " O'Reilly Media, Inc.".

[6] CNCF. 2023. Project Journey Report. `https://www.cncf.io/reports/kubernetes-project-journey-report/`

[7] Codefresh. [n. d.]. CI/CD vs. DevOps: Key Differences and How They Work Together. `https://codefresh.io/learn/ci-cd-concepts/ci-cd-vs-devops-key-differences-and-how-they-work-together/`

[8] RS Dittakavi. 2023. Achieving the Delicate Balance: Resource Optimization and Cost Efficiency in Kubernetes. *Eduzone: International Peer Reviewed/Refereed Multidisciplinary Journal* 12, 2 (2023), 125–131.

[9] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11–15.

[10] Sara Hong, Yeeun Kim, Jaehyun Nam, and Seongmin Kim. 2024. On the Analysis of Inter-Relationship between Auto-Scaling Policy and QoS of FaaS Workloads. *Sensors* 24, 12 (2024), 3774.

[11] Heiko Koziolek and Nafise Eskandani. 2023. Lightweight kubernetes distributions: A performance comparison of microk8s, k3s, k0s, and microshift. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 17–29.

[12] Kubecost. 2025. Load Balancing in Kubernetes. `https://www.kubecost.com/kubernetes-best-practices/load-balancer-kubernetes/`

[13] Kubernetes. 2022. Pod Security Policies. `https://kubernetes.io/docs/concepts/security/pod-security-policy/`

[14] Kubernetes. 2024. Cluster Architecture. `https://kubernetes.io/docs/concepts/architecture/`

[15] Kubernetes. 2024. Configure a Security Context for a Pod or Container. `https://kubernetes.io/docs/tasks/configure-pod-container/security-context/`

[16] Kubernetes. 2024. Pod Security Admission. `https://kubernetes.io/docs/concepts/security/pod-security-admission/`

[17] Kubernetes. 2024. Pod Security Standards. `https://kubernetes.io/docs/concepts/security/pod-security-standards/`

[18] Vikash Kumar. 2023. Kubernetes Best Practices 2024: Essential Tips for Modern Cluster Management. `https://www.getambassador.io/blog/kubernetes-best-practices`

[19] Nicolas Labrot. 2024. Optimizing Application Resilience: A Deep Dive into Kubernetes Pod Disruption Budgets and Rollout Strategies. `https://blog.spikeseed.cloud/k8s-resilience-pdb-rollout/`

[20] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)* 10, 4 (1992), 360–391.

[21] Natalie Lunbeck. 2024. K0s vs K3s vs K8s: Comparing Kubernetes Distributions. `https://shipyard.build/blog/k0s-k3s-k8s/`

[22] Sébastien Mosser. [n.d.]. *jPipe Language.* `https://github.com/ace-design/jpipe`

[23] Sébastien Mosser and Nirmal Chaudhari. [n.d.]. Modelling Justification Diagrams Using jPipe. `https://github.com/ace-design/jpipe-demo`

[24] Sébastien Mosser and Jason Lyu. [n.d.]. *jPipe Runner.* `https://github.com/ace-design/jpipe-runner`

[25] Sébastien Mosser and Jason Lyu. [n.d.]. k8s-best-practices. `https://github.com/ace-design/k8s-best-practices`

[26] Ruchika Muddinagiri, Shubham Ambavane, and Simran Bayas. 2019. Self-hosted kubernetes: Deploying docker containers locally with minikube. In *2019 international conference on innovative trends and advances in engineering and technology (ICITAET)*. IEEE, 239–243.

[27] OpenShift. 2024. Advanced Scheduling and Pod Affinity and Anti-affinity. `https://docs.openshift.com/container-platform/3.11/admin_guide/scheduling/pod_affinity.html`

[28] Deesha Patel. 2024. *A STUDY ON JUSTIFYING PLATFORM-INDEPENDENT CI/CD PIPELINES.* Technical Report. McMaster University. `http://hdl.handle.net/11375/29326`

[29] Thomas Polacsek. 2016. Validation, accreditation or certification: a new kind of diagram to provide confidence. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–8.

[30] Thomas Polacsek, Sanjiv Sharma, Claude Cuiller, and Vincent Tuloup. 2018. The need of diagrams based on Toulmin schema application: an aeronautical case study. *EURO Journal on Decision Processes* 6, 3-4 (2018), 257–282.

[31] Corinne Pulgar. 2022. Eat your own DevOps: a model driven approach to justify continuous integration pipelines. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 225–228.

[32] Redhat. 2021. The State of Kubernetes Security. `https://www.redhat.com/en/blog/state-kubernetes-security`

[33] Eoghan Russell and Kapal Dev. 2024. Centralized Defense: Logging and Mitigation of Kubernetes Misconfigurations with Open Source Tools. *arXiv preprint arXiv:2408.03714* (2024).

[34] Jay Schmidt. 2024. Docker Best Practices: Using Tags and Labels to Manage Docker Image Sprawl. `https://www.docker.com/blog/docker-best-practices-using-tags-and-labels-to-manage-docker-image-sprawl/`

[35] Jacob Schmitt. 2023. What is Helm? A complete guide. `https://circleci.com/blog/what-is-helm/`

[36] Shazibul Islam Shamim, Jonathan Alexander Gibson, Patrick Morrison, and Akond Rahman. 2022. Benefits, Challenges, and Research Topics: A Multi-vocal Literature Review of Kubernetes. *arXiv preprint arXiv:2211.07032* (2022).

[37] Ashish Singh. 2024. Kubernetes Architecture and Components Explained. `https://aws.plainenglish.io/kubernetes-architecture-and-components-explained-d3e7213f255f`

[38] Cody Slingerland. 2024. Kubernetes Node Vs. Pod Vs. Cluster: Key Differences. `https://www.cloudzero.com/blog/kubernetes-node-vs-pod/`

[39] Abaied Sopi and Plotoaga Andrei. 2023. Comparing various methods for improving resource allocation on a single node cluster in Kubernetes.

[40] StackRox. 2024. KubeLinter. `https://github.com/stackrox/kube-linter`

[41] Andy Suderman. 2025. Kubernetes in 2025: Are You Ready For These Top 5 Trends & Predictions. `https://www.fairwinds.com/blog/kubernetes-2025-top-5-trends-predictions`

[42] Kai Sun. 2024. *A STUDY OF JUSTIFICATION ON JUPYTER NOTEBOOK QUALITY & FAIRNESS*. Technical Report. McMaster University. `http://hdl.handle.net/11375/29641`

[43] Tigera. [n. d.]. Kubernetes Security: Risks, Security Controls & Best Practices. `https://www.tigera.io/learn/guides/kubernetes-security/`

[44] Thad West. 2020. Why Service Management is a Critical Piece of DevOps. `https://blog.isostech.com/why-service-management-is-a-critical-piece-of-devops`

[45] Ahmed Zerouali, Ruben Opdebeeck, and Coen De Roover. 2023. Helm charts for Kubernetes applications: Evolution, outdatedness and security risks. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 523–533.