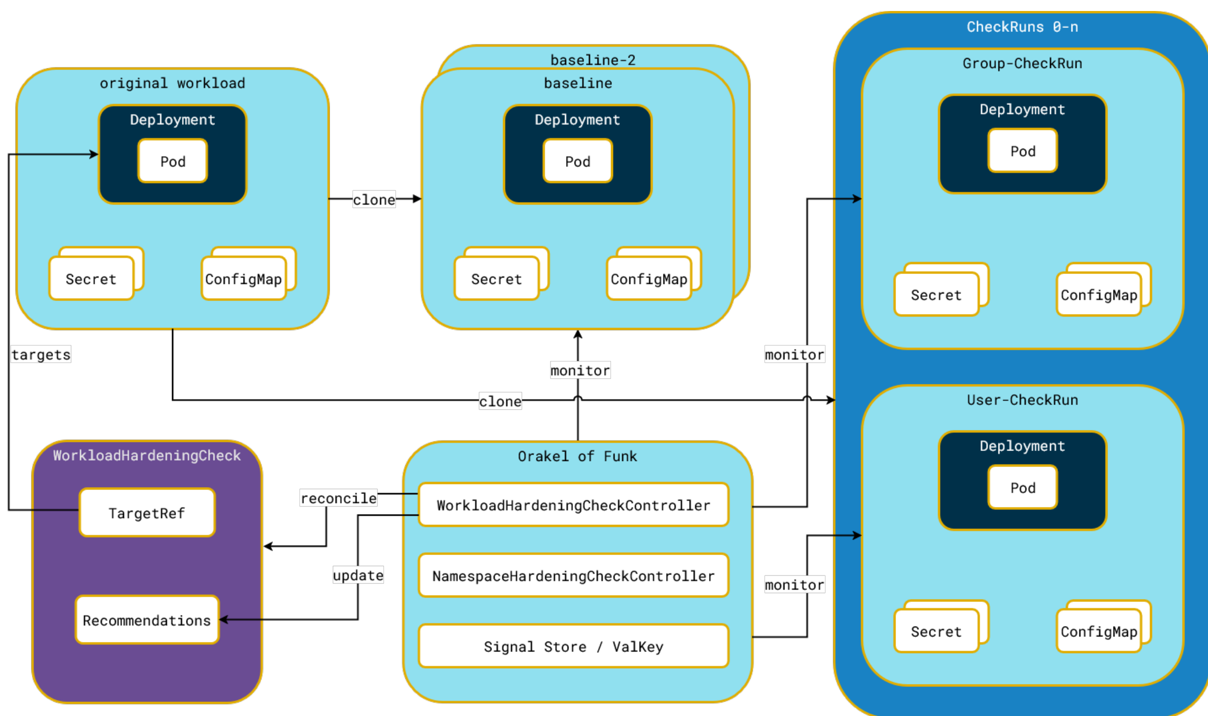
University of Applied Sciences and Arts Northwestern Switzerland
School of Computer Science

Automated Kubernetes Workload Hardening Using a Functionality Oracle

Bachelor Thesis

Windisch, September 2025



Student	Mathias Petermann mathias.petermann@gmail.com
Expert	Romano Roth romano.roth@zuehlke.com
Academic Supervisor	Sebastian Graf sebastian.graf@fhnw.ch
Project Sponsor	Sebastian Graf sebastian.graf@fhnw.ch
Projectnumber	25FS_IMVS28

University of Applied Sciences and Arts Northwestern Switzerland, School of Computer Science

Abstract

Kubernetes allows workloads to be deployed in sandboxed environments using containerization, offering fine-grained runtime restrictions via securityContext configurations. However, determining whether an application continues to function correctly under increasingly restrictive settings remains a challenge, especially when conventional testing methodologies require application-specific knowledge or integration tests. This thesis presents a methodology for automated workload hardening, without requiring internal knowledge of the workload under test.

A Kubernetes Operator was developed that iteratively restricts container runtime permissions and evaluates the functional correctness of workloads using a set of heuristics. These heuristics rely on telemetry signals such as container logs, and resource metrics, gathered during controlled check runs and compared against a recorded baseline. Log analysis is performed using the Drain algorithm, while time series data are evaluated through statistical summaries. The operator clones the workload's `Namespace` to preserve isolation, executes checks for different configurations, and synthesizes a recommended *securityContext* configuration based on the outcome.

Real-world workloads were used for evaluation, alongside custom workloads targeting specific runtime constraints. The results demonstrate that functionality-based hardening is feasible with minimal assumptions, and that log-based heuristics are particularly effective for detecting deviations. The operator-based approach integrates seamlessly with Kubernetes environments and supports developer workflows by providing actionable hardening recommendations.

Keywords:

Kubernetes, Workload Hardening, Runtime Security, Heuristic Testing, Functionality Oracle, Log Analysis, Anomaly Detection

Contents

List of Figures v

List of Tables v

Acronyms vi

Glossary vi

1 Introduction 1

1.1 Problem statement . . . 1

1.2 Research Objectives . . . 1

1.3 Scope and Limitations . . . 2

2 State of the Art & Related work 3

2.1 Kubernetes workload execution and runtime constraints . . . 3

2.2 Software classifications . . . 5

3 Classification of workloads 6

3.1 Definition and Characteristics of Workloads . . . 6

3.2 Classification of software applications . . . 6

3.3 Classification of Kubernetes workloads . . . 9

3.4 Conclusion . . . 9

4 Heuristics for Verifying Workload Functionality 11

4.1 Oracles in software testing . . . 11

4.2 Attributes used by Kubernetes . . . 11

4.3 Attributes used for anomaly detection . . . 13

4.4 Heuristic attributes for Kubernetes Workloads . . . 14

4.5 Conclusion . . . 15

5 Functionality Oracle for Kubernetes Workload 16

5.1 Attributes . . . 16

5.2 Sources . . . 17

5.3 Processing . . . 18

5.4 Heuristic Evaluation Strategies . . . 18

5.5 Analyzing Timeseries . . . 19

5.6 Direct Comparison 19
5.7 Analyzing Logs 21
5.8 Heuristic to determine Workload functionality 24

6 System Architecture **25**
6.1 Where to run 25
6.2 Solution Execution Flow 26

7 Implementation and Evaluation **28**
7.1 Choice of Framework: Operator SDK 28
7.2 Operator Execution Flow 28
7.3 Signal Collection and Storage 29
7.4 Evaluation Heuristics and Test Cases 30
7.5 Check Design 31
7.6 Execution Modes and Configuration 32
7.7 Results Reporting 33
7.8 Evaluation Setup 34
7.9 Observations and Limitations 36
7.10 Summary 37

8 Conclusion and Future Work **38**
8.1 Key Insights and Contributions 38
8.2 Strengths and Limitations of the Operator-based Approach 38
8.3 Evaluation Highlights 39
8.4 Opportunities for Technical Improvements 39
8.5 Future Research Directions 40

References **41**

Declaration of Authenticity **43**

Appendices **45**
A Index of Auxiliary Tools 45
B Source Code Artifacts 46
C Orakel of Funk Project README 46

List of Figures

3.1 Software application categories according to Sommerville 7
5.1 Overlaying two CPU recordings for Prometheus 19
5.2 Applying Dynamic Time Warping to Prometheus CPU usage recording 20
6.1 Graph outlining operator based oracle architecture 26
7.1 CheckRun StatusCondition state diagram 29
7.2 WorkloadHardeningCheck and NamespaceHardedningCheck example 32
7.3 Recommended securityContext in a WorkloadHardeningCheck 34

List of Tables

2.1 Kubernetes securityContext fields for limiting Linux container runtime capabilities 4
4.1 Common Attributes for Anomaly Detection in Kubernetes Workloads 13
5.1 Heuristic Attributes available in Kubernetes 16
5.2 Statistical Summary on Prometheus resource usage 21
5.3 Functionality test cases for Kubernetes workload 24

Acronyms

API Application Programming Interface.

CI/CD pipeline Continuous Integration/Continuous Delivery pipeline.
CLI Command Line Interface.
CRD Custom Resource Definition.

DTW Dynamic Time Warping.

LLM Large Language Model.

OOM Out Of Memory.

RBAC Role-Based Access Control.

Glossary

baseline A known working state of an application. In the context of our work this references an unmodified copy of the target workload..

CrashLoopBackOff A Kubernetes Pod state indicating repeated failures and restarts of a container..

Drain (algorithm) A log parsing algorithm that extracts structured templates from unstructured logs using a tree-based approach..

implicit oracle "An implicit test oracle is one that relies on general, implicit knowledge to distinguish between a system's correct and incorrect behaviour."[1].

kubelet The kubelet is the primary "node agent" that runs on each node..
Kubernetes API The Kubernetes API is used to manage resource objects in a cluster. It is the central interface how to interact with Kubernetes..

ValKey A high-performance, in-memory key-value store used for temporary storage of logs and metrics..

WorkloadHardeningCheck A custom resource that defines the execution of security context checks for a specific workload in the Kubernetes operator..

1 Introduction

1.1 Problem statement

Container-based sandboxing results in applications having no knowledge about the underlying infrastructure. This isolation means that runtime restrictions, such as those imposed by Kubernetes *securityContext* or resource limits, can lead to unexpected application behavior.

As the impact of these restrictions only becomes apparent during runtime, they cannot be detected through conventional functional testing. Thus, applications must be tested in a Kubernetes cluster to ensure they function correctly under restrictive conditions. However, in practice, this is often neglected due to the significant effort required to conduct such tests. Moreover, there is no generic solution to ensure that an application operates correctly under these constraints.

1.2 Research Objectives

The goal of this project is to provide a software solution which iteratively restricts the container runtime for Kubernetes based workload, and with each change verifies the functionality before creating a recommendation how to configure the workload for maximum security without breaking it.

1.2.1 Classification of workloads

To verify the functional correctness of an application, we usually run integration or end-to-end tests. But this can be quite time consuming, and depending on the development tools used, can not be easily done in Kubernetes. In addition, we want to create a solution which is language- and framework-independent to allow usage with a large number of applications.

To achieve this, we need a method to verify functionality in a more generic manner. One possible method is to categorize workloads based on the interfaces they expose and the ways in which they can be interacted with. This leads to our first research question.

What classification methods exist to group workloads into categories that enable standardized testing?

1.2.2 Using heuristics to verify functional correctness

One strategy to verify functional correctness of an application is to use telemetry generated by the workload and the surrounding `Pod` to and compare this to a known working state.

This is similar to what is done in anomaly detection, and we want to leverage research done in that field to get a set of data points that we can use to create a heuristic process to verify the applications functionality, which is reflected in our second research question.

What heuristics can be used to evaluate workload functionality when runtime constraints are modified?

1.2.3 Using heuristics in an iterative approach

Based on the outcome of the previous research questions we will develop a software solution that can gather the necessary telemetry data for an application and restrict the workloads container runtime dynamically. It should then use the identified heuristics to verify functional correctness before continuing to limit the runtime.

How can the identified heuristics be utilized to automate iterative runtime restrictions and systematically verify workload functionality after each change?

1.2.4 Software architecture and integration into DevOps process

The software tool we develop should integrate easily into existing DevOps workflows, so it is important we evaluate the best place to integrate it, which will also impact the software architecture.

Integrating such a tool into a Continuous Integration/Continuous Delivery pipeline (CI/CD pipeline) seems like the best place as it is run often and we could even leverage end-to-end tests that are written for a specific software. But if you are only using the software and not developing it, this could limit the applicability of our solution.

Another option is to write a Kubernetes operator which performs the iterative approach to harden a given workload. This would allow anyone running the target application to run these tests, but would limit the integration into a regular CI/CD pipeline. To investigate this further and balance the trade-offs between the different implementations we have our final research question.

What is the most effective architectural approach to balance automation in CI/CD pipelines with the flexibility of a Kubernetes operator for developer-driven adjustments?

1.3 Scope and Limitations

The focus of this thesis is hardening the workload through the container runtime, and the restrictions we can apply to it. We will make use of Kubernetes built-in capabilities, like the *securityContext* and resource requests and limits, that can be defined for each pod individually.

In Kubernetes there is also Role-Based Access Control (RBAC) which is applied to a workload through the assigned Service Account. For this project we explicitly exclude hardening the RBAC configurations.

For classifying workloads we will focus on software applications and ignore system software and compilers as they are not usually run in Kubernetes.

Even though Kubernetes supports Windows and Linux based worker nodes, it is still uncommon to run worker nodes on Windows. Consequently Windows specific configurations are excluded.

2 State of the Art & Related work

2.1 Kubernetes workload execution and runtime constraints

Containers provide a lightweight and portable mechanism for running software in isolated environments. At their core, containers are Linux processes with constrained access to host resources, enforced through Linux kernel primitives such as *namespaces*, *cgroups*, and *capabilities*. However, containers are not secure by isolation alone, as they can be run privileged.

Kubernetes uses *securityContext* configurations to fine-tune runtime restrictions. It can be be defined both at `Pod` and `Container` levels and allows users to limit the container's access to the underlying host system.

By leveraging the *securityContext*, DevOps engineers and platform operators can enforce security best practices such as running containers as non-root user, dropping Linux capabilities, or mounting file systems as read-only, thereby significantly reducing the attack surface of workloads running in Kubernetes[2].

2.1.1 Kubernetes securityContext options

Table 2.1 summarizes the *securityContext* options provided by Kubernetes, which will be the basis for restricting the container runtime [3]. [1]

For all options applicable at Pod and Container level, the configuration at container level takes precedence. This allows running multiple Containers with different settings in a single pod.

2.1.2 Misconfigurations in Helm Charts

At KubeCon EU 2024 Minna and Blaise presented a pipeline that combines static scanning of Helm charts, with automated template fixes, and a runtime functionality oracle to ensure changes do not break workloads. The oracle deploys a baseline, tightens permissions, and accepts a change if *Readiness-Probes* succeed and logs remain comparable.

Their oracle aligns with our black-box approach, but our method targets systematic restriction of *securityContext* settings, whereas theirs evaluates template-level fixes driven by scanners.

While they released an article outlining their work [4], no public source code for the oracle and no dedicated paper describing it were released, this was confirmed via personal e-mail correspondence with F. Minna.

[1] Options only available on Windows nodes are excluded

Field	Description
`allowPrivilegeEscalation`	Controls whether a process can gain more privileges than its parent, such as via setuid binaries. Container level only.
`capabilities`	Allows fine-grained control over Linux capabilities (e.g., `NET_ADMIN`, `SYS_TIME`). Container level only.
`fsGroup`	Defines a group ID used for setting group ownership of mounted volumes. Pod-level only.
`fsGroupChangePolicy`	Defines behavior of changing ownership and permission of the volume before being exposed inside Pod. Pod-level only.
`privileged`	Grants the container full access to the host, disabling most isolation mechanisms. Only available at the Container level.
`readOnlyRootFilesystem`	Mounts the containers root file system as read-only. Prevents write access to root-level paths. Container level only.
`runAsGroup`	Specifies the GID to run the container process. Useful for filesystem permissions. Applicable at Pod or Container level.
`runAsNonRoot`	Ensures the container does not run as root (user ID 0). Kubernetes will reject the Pod if no user ID is set. Applicable at Pod or Container level.
`runAsUser`	Specifies the UID to run the entrypoint of the container process. Applicable at Pod or Container level.
`seLinuxOptions`	Specifies SELinux labels for process confinement. Requires SELinux-enabled hosts. Usable at Pod or Container level.
`seccompProfile`	Applies a Seccomp profile to limit available syscalls. Usable at Pod or Container level.
`supplementalGroups`	List of additional GIDs the container will be part of. Useful for shared volume access. Pod-level only.
`supplementalGroupsPolicy`	Defines how supplemental groups are calculated. Promoted to Beta in Kubernetes 1.33.
`sysctls`	Defines kernel parameters for the Pod. Only safe, allow-listed sysctls are permitted. Pod-level only.

Table 2.1: Kubernetes *securityContext* fields for limiting Linux container runtime capabilities [3]

2.2 Software classifications

Software can be categorized according to a multitude of characteristics. A commonly used attribute is the copyright status of software, which can be seen as a hierarchical taxonomy, with free and non-free software as the major categories, and lower level categories based on the specific license used. For our purpose, the license of the application under test provides no relevant information and we exclude it for our research.

In the software industry, an often used taxonomy is their functionality, grouping them into general categories like database, middleware, backend and frontend. These are very broad categories and do not provide information about the functional correctness of an application, and depending on the exact architectural style of an application, each category can contain a number of components [5]. While these categories tell us something about the inner workings of an application, we are not able to infer functional correctness from them alone.

If we look at cloud offerings, we see that they offer their services for different categories of applications. AWS for example groups their EC2 instances into compute-, memory-, storage- and AI-optimized categories.[2] Considering they need to provide different hardware for each category, it makes sense that they choose the kind of resource which is most relied on as the attribute to group by. Even though we can not rely solely on resource usage, we might be able to use it to detect functional correctness.

[2] https://aws.amazon.com/ec2/instance-types/

3 Classification of workloads

3.1 Definition and Characteristics of Workloads

As mentioned in Section 1.3 we are focusing exclusively on software applications, and exclude other types of software that are not commonly run on Kubernetes. In Section 2.2 we already showed ways how software is categorized but these are not suitable to verify functional correctness of an application.

A Kubernetes workload typically consists of one or more `Pods` running containers withing a single Kubernetes `Namespace`. These pods are orchestrated by higher level Kubernetes resources, such as `CronJob`, `Job`, `Deployment`, `DaemonSet`, or `StatefulSet`. The higher level resources, are used to determine how many and on which nodes the corresponding `Pods` are spawned, but the pod specifications within each resource Kind are identical.

Each workload in Kubernetes encapsulates the application logic, its execution lifecycle, and its scalability properties. These workloads are described declaratively using manifests and are managed through the Kubernetes API. Workloads also define interaction with other cluster components through networking, storage, configuration (e.g., `ConfigMaps` and `Secrets`), and policies (e.g., affinity rules, tolerations, and *securityContext*) [2].

For the classification of workloads in regard to automated testing, we will focus on the `Pod` level which contains the containers running the application logic, but when specifying tests that can be used to determine functional correctness we will need to take the higher level kind into consideration as they can behave differently.

The Kubernetes platform standardizes deployment and execution environments, abstracts away domain-specific hosting concerns, and often treats workloads uniformly regardless of their stage in the software lifecycle. As such, we do not need to know about the *application domain* or the *lifecycle phase* an application is in.

3.2 Classification of software applications

Application software can be categorized based on several classification schemes, depending on the perspective taken.

In *Software Engineering* Sommerville touches on all the categories seen in Figure 3.1, illustrating how many different options there are to classify software [6]. He also touches on categories based on licensing model but as we excluded that characteristic earlier we removed it from the tree.

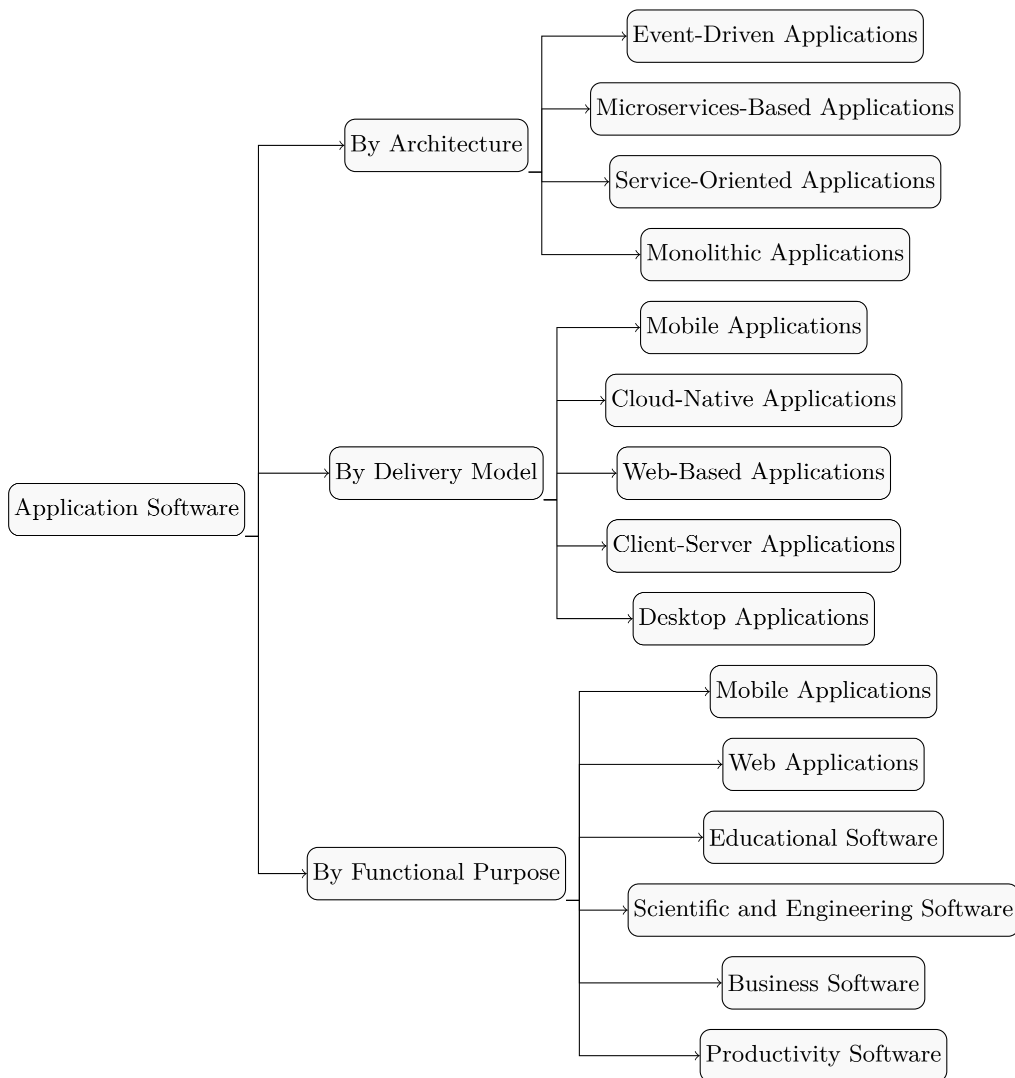


Figure 3.1: Software application categories according to Sommerville[6]

But even in this limited tree, we have two more classification attributes, that can not be used for a black-box approach, since we do not know an applications functional purpose, without knowing about its internals, and also the delivery model can not be inferred as we might have a partial view of the software in Kubernetes. This leaves only the architecture of a service that can be used to infer generic tests.

3.2.1 Classification by Architecture

As shown in Figure 3.1 the architecture of an application can be used to group them independent of its functional purpose, and without knowing much about the internals of an application or its business logic.

In *Fundamentals of Software Architecture* many common architecture styles are explained in great detail, and they state "Service-based architecture is a hybrid of the microservices architecture style" [5]. Based on this we will limit ourselves to the following architectural styles for further investigations:

- Monolithic

- (Micro-)Service-Based
- Event-Driven

3.2.2 Monolithic application

Monolithic applications consist of a single process implementing all the functionality, and often only interact with their own database. This makes it simpler for generic testing since only a single component must be tested, but at the same time it is difficult to test all the possible inputs to be sure that everything is functional.

As for the testing approach the only option is to test the endpoints the application exposes, and rely on the return codes of these endpoints. Usually a monolithic application will only have a limited amount of endpoints exposed, and rarely one of those implements a proper health check that can be relied on.

3.2.3 Event-Driven applications

Event-driven applications consist of multiple loosely coupled components that communicate through events, often using a message broker. Each component typically reacts to specific types of events and may produce its own events in response. This architecture introduces complexity for generic testing, as the flow of information is asynchronous and non-linear, making it harder to trace complete execution paths or guarantee full coverage through testing.

For a testing approach, it would be useful to observe the event streams and verify whether events are emitted and consumed in a timely and expected manner.

To gather insights into the health of an event-driven application we need to leverage the metrics of the event-broker, detailing the number of events generated and consumed. Depending on the producer and consumer components, we might also get metrics how many messages were produced and consumed, which is a great indicator for the functionality.

3.2.4 Service-based applications

Service-based applications are structured as a collection of distinct, independently deployable services [7]. Each service typically communicates with others via APIs over the network, which allows tracking interaction between services by deep-packet inspection, but also makes testing more complex as each component should be tested and hardened individually to isolate any changes influencing the other components.

Health check endpoints are more common in service-based applications, and each service may provide its own health and readiness endpoints that can be used to infer overall system health. However, the increased number of external dependencies and communication channels means that failures might propagate in subtle ways, making it critical to also monitor interdependencies.

3.2.5 Classification by input/output

One more classification method that can be used, is classifying an application by its exposed interfaces. While this is implied by the *delivery model* for certain application types (e.g., Web-Based or Client-Server applications), it is not common to characterize applications based on the ports or protocol that they expose.

But for our testing purposes this can be very useful. In Kubernetes each container can define certain ports that are exposed, and map them to a `Service` which then makes it available in the

cluster. Additionally ports can also be used to perform probes on a container to determine its running state, and it is even possible to provide a script which is run within the container as a probe.

During security audits it is common to run a port scanner against a target host which reports all open ports and the protocol used on each port. In Kubernetes it is considered best practice to configure the *containerPorts* attribute on each container, but it is not required to route traffic to a `Pod`, so we can not rely on this configuration to find all open ports.

3.3 Classification of Kubernetes workloads

When trying to classify workloads for testing purposes, it is essential to use categories that affect runtime behavior or observability.

Three dimensions that play a central role are *interaction mode*, *system architecture*, and *input/output types*. These directly influence how a workload behaves at runtime, what kind of telemetry it produces, and how its health and correctness can be inferred using Kubernetes-native observability tools.

- **Interaction Mode:** Distinguishes between batch jobs, interactive applications, event-driven services, and stream-processing workloads, and it affects the structure of observability signals. For example, batch jobs may only produce logs once per run and have no persistent state, while event-driven services may require high-resolution metrics and tracing data to capture responsiveness [8].
- **Architecture:** Workloads may follow monolithic, service, or event-driven patterns. In Kubernetes, this is reflected in pod-level communication, which can be monitored using a service mesh [9].
- **Input/Output types:** Whether a workload handles textual, binary, structured, or sensor-derived data affects both how input is simulated in testing and how output is evaluated. Applications with structured output (e.g., JSON over HTTP) allow easier log-based assertions and Prometheus-based metrics scraping, while binary or opaque data may require heuristic anomaly detection on resource usage or error logs [1].

These characteristics are the most useful to classify Kubernetes workload in regards to black box testing of functionality.

3.4 Conclusion

While the classification of application workloads along *architecture*, *interaction mode*, and *input/output* dimensions provides a structured framework for understanding software deployed in Kubernetes environments, this approach alone proves insufficient for the construction of truly generic functional tests. The core limitation arises from the abstraction gap between observable workload characteristics and the specific functional expectations of the software.

Even within narrowly defined categories, such as software architecture, the diversity in application behavior, and runtime dependencies makes it impractical to derive a meaningful test suite that validates correctness without incorporating application-specific knowledge. For instance, knowing that a workload communicates asynchronously via events does not reveal the semantics of those events, the correctness criteria of responses, or the expected causal chains within the system. Similarly, classifying a system as “RESTful” or “stream-processing” does not expose the internal business rules that govern response validity.

This conclusion aligns with findings in the software testing literature. In *Introduction to Software Testing* Ammann and Offutt emphasize that “test requirements must be derived from the

specification or code” and that only through knowledge of expected behavior can meaningful assertions be made in a test oracle [8].

Therefore, while workload classification remains a valuable tool for structuring testing requirements, it does not resolve the fundamental challenge of generating workload-specific assertions required for functional correctness testing.

4 Heuristics for Verifying Workload Functionality

4.1 Oracles in software testing

In software testing an *oracle* is defined as the component which describes the correct output for a test, based on its input [10]. Such an oracle can be as simple as a expected value in a unit test, or something more complex like multiple assertions that verify if a list of integers was sorted. In some cases the *oracle* can even be a human for example when performing manual tests.

The concept of a test oracle was established very early on in software testing and we operate under the assumption that the oracle can accurately predict the tests outcome, but this is not necessarily the case [10].

In the case of functional correctness we already know that there is no oracle which can provide us with a reliable prediction for each software, but we can use an implicit oracle [1] that operates on assumptions made about any software. For instance, a Kubernetes pod in the *CrashLoopBackOff* state is always non-functional. Conversely, a pod in the *Running* state may appear healthy, but in the absence of correctly configured *Liveness-* or *Readiness-Probes*, it could still experience continuous errors without triggering a restart or failure state [11].

4.2 Attributes used by Kubernetes

As discussed in Section 3.3 workloads can be classified based on the interaction mode, the architecture as well as the input/output types. Each criteria results helps us define the signals that we should look at to determine if a workload is functionally correct or not.

Additionally we can take inspiration from anomaly detection techniques to determine if a workload is performing as expected. But unlike in anomaly detection where data is gathered over a time frame of multiple months, we are limited to a much smaller test duration and we need to establish a known working baseline that we can compare against.

4.2.1 Pod healthiness

In Kubernetes a `Pod` represent the smallest component that can be managed through a resource definition, but it is considered bad practice to configure a *naked* `Pod`. Instead users define a resource that manages the `Pod` and its lifecycle. Two common options how a `Pods` are managed, is either through a `ReplicaSet` or a `Job`, both of which can be managed by higher level resources. For example a `Deployment` for the `ReplicaSet` or a `CronJob` for the `Job`.

For the user it makes a huge difference if they define a `Deployment`, a `StatefulSet` or a `DaemonSet` but if we look at it on the `Pod` level, the pod specification used is identical.

The lifecycle on the other hand varies greatly based off the higher level resource. `Pods` scheduled by a `Job` are expected to exit after finishing whatever processing they need to do, and are only restarted if the `Job` exited with an error code. While `Pods` managed by a `ReplicaSet` are always restarted, and are expected to run indefinitely. This drastically changes how the functional correctness of a `Pod` is defined.

To define the indicators used if a `Pod` is functioning correctly we will thus differentiate between regular `Pods` and `Jobs` even if we evaluate both on the `Pod` level.

Before we define the heuristics for regular `Pods` we need to look at how Kubernetes evaluates pod healthiness.

In the resource definition of a `Pod` there are so called *Probes* which are evaluated to determine each `Containers` healthiness. Those are the *Startup-*, *Liveness-* and *Readiness-Probes*. For simplicities sake, we are going to assume each `Pod` only contains a single `Container`.

4.2.2 Startup-Probe

As the name indicates, the *Startup-Probe* is executed as soon as a `Container` is started, and can be used to delay the execution of the other probes. This is helpful if the pod takes a long time to start. [3]
As soon as the probe is successful it will no longer be run and the *Liveness-* and *Readiness-Probes* take over.

4.2.3 Liveness-Probe

The *Liveness-Probe* determines a `Pods` running state. If a `Pod` is failing this probe repeatedly, it will be restarted. This is intended to detect issues that are unrecoverable.

4.2.4 Readiness-Probe

The designed functionality of the *Readiness-Probe* is to fail if a time-consuming task is being performed, ensuring that the `Pod` temporarily does not receive traffic. If the *Readiness-Probe* fails, Kubernetes will remove it from any matching services, until the probe is successful again, at which point it will resume sending traffic to it.
This can be used to remove a `Pod` from a service while refreshing caches or loading files, and resume normal interactions afterwards, without having to restart the pod.
Defining a proper *Readiness-Probe* is difficult and many applications reuse the *Liveness-Probe* endpoint with a lower failure threshold, but unless the `Pod` is somehow able to recover itself thanks to the reduced load, this is insufficient [11].

4.2.5 Pod state

Based on the *Probes*, Kubernetes determines the state of each `Container` and as soon as all of them are *started* and *ready* the Pod itself is considered *ready* as well.
If a `Container` in a `Pod` crashes, it will be restarted but with a exponential delay, and the `Pod` will move to the *CrashLoopBackOff* state until all `Containers` are ready again.

For `Jobs` the lifecycle is a bit different, as they are expected to finish after performing their tasks, and the `Pod` state will proceed to *Succeeded* at that point. They way this is determined is solely based on the exit code of the `Containers`.

Based on this we can assume that even though a `Pods` healthiness can be determined by the probes, we can not rely on them alone for the functional correctness, even if all of them are configured and we can only rely on them to indicate failure of a `Pod`.
Additionally for `Pods` belonging to a `Job` we should not try to determine the success based on the `Pod` itself, but on the `Job` resource. There are different type of `Jobs`, which can still succeed even if a single `Pod` failed, most notably, indexed jobs which are often used for machine-learning workloads. [4]

[3] `https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/`
[4] `https://kubernetes.io/docs/tasks/job/indexed-parallel-processing-static/`

4.3 Attributes used for anomaly detection

Based on the fact that the `Pod` state can not be relied on to test for functional correctness we decided to look into other disciplines to find approaches to determine a workloads functionality.

Anomaly detection operates on the assumption that incorrect behavior often manifests through observable deviations in execution characteristics, even if those deviations cannot be directly mapped to a violation of functional requirements. These characteristics may include unexpected spikes in resource usage, changes in user interaction patterns, shifts in input/output distributions, or alterations in control and data flows. In systems that produce logs, metrics, or traces, these signals can be monitored over time and compared against statistical models, heuristics, or historical baselines to identify anomalies [12].

Category	**Attribute**	**Description**
Resource Metrics	CPU Usage	CPU usage over time
	Memory Usage	Memory consumption (RSS, working set)
	Disk I/O Rate	Bytes read/written per second
	Network I/O	Network throughput, latency
Kubernetes Events	Pod Restarts	Restart count of containers
	CrashLoopBackOff	Failure states like OOMKilled
	Probe Failures	Liveness/readiness probe errors
Logs and Errors	Log Volume	Number of log lines per window
	Error Frequency	Occurrences of “error”, “fail”, etc.
	Log Anomaly Score	Outlier score for log patterns
App Metrics	HTTP Error Rate	Ratio of 4xx/5xx to all responses
	Request Latency	Distribution/spikes in latency
	Request Rate	Number of requests per second
	Queue Length	Buffer or queue depth
Temporal Features	Metric Volatility	Variance or instability over time
	Diurnal Deviation	Deviation from daily patterns
	Anomaly Frequency	Count of detected anomalies
System Health	Node Availability	Node taints or unschedulability
	CPU Throttling	Time throttled by cgroup limits
Behavioral Patterns	API Call Mix	Distribution of endpoints accessed
	Dependency Drift	Changes in service interactions

Table 4.1: Common Attributes for Anomaly Detection in Kubernetes Workloads

Table 4.1 summarizes common system and application attributes that serve as inputs for anomaly detection techniques. These include resource-centric metrics such as CPU usage or disk I/O, operational signals like application restarts or log volume, and behavioral indicators such as request latency or API usage patterns. Log-based anomaly detection, for instance, has been shown to be effective even in the absence of log parsing, through the use of statistical or deep

learning models that detect structural deviations in log sequences [13]. Similarly, temporal attributes, such as volatility or periodic deviation, have been used in time-series-based models to detect anomalies in streaming or real-time applications.

Anomaly detection is particularly useful in circumstances where the correctness of output is either domain-specific or difficult to predict in a long term observational scenario. Techniques such as clustering, and statistical outlier detection allow these systems to be monitored and evaluated with minimal domain knowledge. While these methods cannot guarantee correctness in a formal sense, they can significantly reduce the time to detect faults and help direct human attention to potentially erroneous states [12].

4.4 Heuristic attributes for Kubernetes Workloads

Based on the signals shown in Section 4.3 we want to create heuristics that allows us to determine a workloads functional correctness.

To be able to define this heuristic we first need to identify which signals we get from Kubernetes and which ones require more tools to gather them, or even more configuration to be able to use them.

Looking at the categories in Table 4.1, we can exclude the Behavioral Patterns, System Health and Temporal Features, since during our tests we will not have any users accessing the workload and the time frame during which we gather metrics is too short to detect any abnormal deviations. Instead we will focus on Resource Metrics, Kubernetes Events, Logs and Errors as well as App Metrics.

4.4.1 Resource Metrics

Resource metrics consist of CPU and memory usage for each container, as well as disk and network I/O. Those metrics are gathered by Kubernetes by default, through the kubelet on each `Node`, but they are not exposed as a resource on the Kubernetes API.

We will need to find a way to access these metrics from our solution, without accessing the underlying container runtime.

4.4.2 Kubernetes Event

`Events` are a Kubernetes native resource, which represents the report of an action that happened in the cluster, usually some state change. They have very limited retention time and do not have a state them self, so we do not know if the underlying trigger was resolved or just did not retry again. Even in the official documentation it is recommended to only use them as supplemental data. [5]

While there are certain `Events` that could be useful to determine why a certain `Pod` is failing, the more interesting information will be in the logs of the `Pod`. But for `Pods` that are crash-looping before the container is created, analyzing the `Events` will be the only option to gather more information about the failure.

4.4.3 Logs and Errors

The best practice for any container is to write the logs of its process to *stdout* as this can be easily handled by the container runtime engine. In Kubernetes logs written to *stdout* can

[5] `https://kubernetes.io/docs/reference/kubernetes-api/cluster-resources/event-v1/`

be fetched through the Kubernetes API just like a regular resource, even though they are not reflected by a resource definition.

As in anomaly detection it does not make sense to compare the logs verbatim but instead to create metrics on these logs. For example the log volume should always be roughly the same for a workload. Similarly one can count the occurrence of certain known words (like "error", "fail" etc.) and compare them with a known baseline to make an assumption on the workload.

There are also projects that use machine learning to analyze logs and predict if there are anomalies in the logs. Some of them are based on a semantic approach where the logs are parsed, similarly to the word count described above [14] [15], while others aim perform a pattern matching approach, where log messages are parsed into templates and consecutive messages are matched against those templates [16]. With the improvements in machine learning and Large Language Model (LLM) those technologies are now also applied to detect anomalies in logs [13], but due to the limited amount of logs collected during the short lifetime of our analysis a deep learning based approach is unlikely to produce reliable results.

4.4.4 App Metrics

In Kubernetes it is common for any workload to expose application specific metrics, on a dedicated endpoint or path. This is usually done using the *OpenMetrics 1.0* [6] format, which can be scraped by a monitoring solution (e.g., Prometheus) and only contains metrics deemed relevant by the application developers.

The metrics exposed here, vary greatly as it could be the an HTTP error rate for a web-service exposing an API, or the number of consumed or produced events in an event-driven architecture.

These metrics are the most relevant to determine the functional correctness of a workload, but also the hardest to interpret, since each application comes with its own set of metrics.

4.5 Conclusion

Relying on Kubernetes Pod phase and probe outcomes alone does not establish functional correctness, since a `Pod` marked `Ready` may still fail to perform its intended work. It does however work as a hard failure gate. The heuristic basis in this project therefore focuses on container logs and resource usage metrics, which together provide workload-agnostic signals that are straightforward to collect.

Kubernetes Events are treated only as auxiliary context for cases where workloads cannot be scheduled, and are excluded from the core evaluation. Although application-level metrics can be highly indicative, their interpretation depends on application-specific knowledge and instrumentation, therefore they are not prioritized within this work.

[6] https://prometheus.io/docs/specs/om/open_metrics_spec/

5 Functionality Oracle for Kubernetes Workload

5.1 Attributes

As highlighted in Section 4.5 we will not rely on pod healthiness as a successful indicator for the functional correctness of our workload and instead make use of the metrics and signals that are also used in anomaly detection.

To gather these metrics and signals we can use the Kubernetes API, and try to find application specific metrics which would give us an in-depth insight into the application.

For each category of attributes we can define a *source* where we can get these attributes from, as well as some *processing* that needs to be done on the collected values.

Category	Source	Processing
Resource Metrics	kube-api / Prometheus	time series / Prometheus
Kubernetes Events	kube-api	text analysis
Logs and Errors	kube-api	text analysis
App Metrics	application / ServiceMonitor	time series / Prometheus

Table 5.1: Heuristic Attributes available in Kubernetes

Table 5.1 shows that we can gather most attributes directly through the Kubernetes API, and only require a more complicated process to gather the *App Metrics*.

Processing the collected data will be more complicated, but the number of different approaches we need to implement is also limited.

5.1.1 Resource Metrics

Resource Metrics are automatically gathered by Kubernetes on each `Node` but are not made available as a specific resource. To collect the metrics there are two possibilities.

- **metrics-server**: The metrics-server provides a Kubernetes native API endpoint from which the metrics can be scraped in a Prometheus compatible format [17]. The metrics-service is widely adopted and relatively lightweight.
- **Kubelet API**: The kubelet manages containers in Kubernetes and exports metrics for them on the specific `Node` on a REST API endpoint. This endpoint is also used by the metrics-server as its source, so we could query this directly.

Either way is a feasible solution and needs to be evaluated in detail, before deciding which option to chose.

5.1.2 Kubernetes Events

`Events` are available directly through the Kubernetes API and do not need further tooling to collect them.

Since most of the `Events` will be from Kubernetes itself, the wording of them is static and analyzing them can be done with some regex matching. `Events` that do not match the known patterns, can still be compared between the baseline and each deployment.

Due to the short lifetime of `Events` (the default lifetime is one hour), we will need to collect them continuously while the tests are running.

5.1.3 Logs

Logs are available through the Kubernetes API and will persist until a container is removed. In the case of crashing containers, the logs for the current and the previous container are available, so these should be collected regularly to have more information why a container failed.

The format of the logs is inconsistent and we need to analyze them as generically as possible. To find a generic way, we will need to evaluate different techniques and create a proof of concept implementation.

5.1.4 App Metrics

Even though these would be the most relevant metrics for us, there is no standardized way to get them from an application. To get the location of these metrics we can use `ServiceMonitor` resource in the cluster if they are present, or fallback to trying different paths that are commonly used. For example in Spring Boot they are at `/actuator/prometheus` by default [7], but it is not feasible to check every possible path for metrics so a simplified way to determine the location of metrics is necessary.

To make the most of app metrics, it would required to expose a configuration option to the user specifying where the metrics are, and which metrics to compare between the baseline and checks.

5.2 Sources

5.2.1 Kubernetes API / kube-api

The *kube-api* source refers to the Kubernetes REST API which is the single point of entry for all interactions with Kubernetes.

Using attributes exposed by the Kubernetes API is considered easy as their are well documented libraries for any API interaction, and even Kubernetes internally uses this API extensively.

5.2.2 Prometheus / ServiceMonitor

For the Resource- and App- Metrics we could consider querying Prometheus directly, if one is present in the Kubernetes cluster, but Prometheus is limited in regards to processing the time series it scrapes.

The `ServiceMonitor` resource is a special `Custom Resource Definition (CRD)` which only exists if the Prometheus-Operator is installed on the cluster. This `CRD` contains information where exactly the App Metrics can be found.

Each `ServiceMonitor` has a *scrapeInterval* configured, which we would need to update when we clone our baseline. Additionally we also want to use metrics provided by the kubelet which is usually exposed through the *metrics-server*, which again has a *scrapeInterval* configured, but also an *interval* parameter, which determines how often the metrics-server queries the kubelet.

[7] `https://docs.spring.io/spring-boot/reference/actuator/metrics.html#actuator.metrics.export.prometheus`

On Azure for example, we do not have the option to configure the metrics-server and will only get new data every 60 seconds.

Instead of trying to work around these limitations, it is more practical to gather the metrics directly from the Kubernetes API, where the metrics are updated every 15 seconds. For the app metrics, we will need to scrape the `Pod` metrics endpoint directly, as they are not automatically scraped.

5.3 Processing

5.3.1 Time Series

All collected metrics need to be compared to a baseline to help determine if the workload is functional. As we are only collecting data for a limited time frame, the time series we end up with, are quite short.

To compare the data there are a multitude of options that can be taken. For this project we decided to look into statistical summary comparison as well as Dynamic Time Warping (DTW).

The statistical summary comparison is based on values that can be calculated from the time series data, for example the *mean* of a single series, and then compared with each other. The difference can then be used to determine if both series are similar. This method is robust for timing inconsistencies, but does not capture temporal patterns or shifts.

DTW is useful for time series which are not aligned, but the pattern is the same or stretched over time. For large datasets this method is considered slow as it requires more computational resources, but as our datasets are very small the performance should be sufficient [18].

Independently of the method used, we can compare the raw data, or normalize it first. Normalizing the data is useful since our data points are not measured at exactly the same time relative to the startup and we have to assume we do not measure at the peak of each spike. For DTW it also helps us to only focus on patterns [19]. Normalizing time series can also distort the results, especially when using statistical summaries. The *min/max* values of a normalized series will always be *0* or *1* respectively, to counter this we need to define for each calculation if we do it on the raw data or the normalized one.

5.3.2 Text analysis

Logs as well as Events are text based signals and need to be analyzed and compared in a way that does not require any domain knowledge for the application under test.

In *Tools and Benchmarks for Automated Log Parsing* Zhu et al. compared the accuracy of 13 different log parser on 16 unstructured datasets and came to the conclusion that Drain is the most accurate log parser achieving a high accuracy across 9 of the 16 datasets [16][20].

To determine if Drain is also accurate in our use case where we have a limited amount of logs available, we decided to perform a proof of concept using Drain on logs gathered manually from known running and failing configurations.

5.4 Heuristic Evaluation Strategies

Before designing our oracle we need to decide on our strategy to log parsing as well as time series analysis. Since both types of data are very different two different testing procedures were evaluated.

For our tests we decided to use *Prometheus* and *ArgoCD* as the target workloads. Prometheus is a cloud-native monitoring tool which was developed with Kubernetes in mind. ArgoCD is a GitOps tool which allows to manage workloads on a Kubernetes cluster entirely from Git. Both tools are widely adopted in the Kubernetes ecosystem and follow the best practices in regards to configuration of the workloads as well as how they interact with Kubernetes.

5.5 Analyzing Timeseries

To record pod resource usage, we developed a lightweight Command Line Interface (CLI) tool that restarts a pod and captures its resource consumption from the moment it starts.. While this is not strictly necessary for evaluating different approaches to analyzing the data gathered it reduces outliers caused by long running pods, when we want to compare the different recordings.

As outlined in Section 5.2.2 the high interval configured in the metrics-server, lead to the decision to query each kubelet directly for metrics. But we quickly realized that even the kubelet only refreshes the resource usage for each container every 15 seconds. Even though this change increased the granularity of our recordings, it will still only result in 20 data points over 5 minutes which we determined a reasonable default runtime for a check. But unless we implement the metrics gathering ourself, directly accessing the container runtime, there is no way to get a higher resolution than 15 seconds.

Using our proof-of-concept CLI we created a set of time series for *Prometheus* and *ArgoCD* and started to compare them.

5.6 Direct Comparison

In Figure 5.1 we see a first comparison where two recordings of the same pod were printed onto a relative timeline. While this shows that our assumption that the resource usage of a workload follows a similar pattern on each startup it also visualizes that a direct comparison between each data point is not possible. There are however spikes on a regular interval (which is to be expected from this workload) and if we find a way to correlate these it could be a solution to our problem.

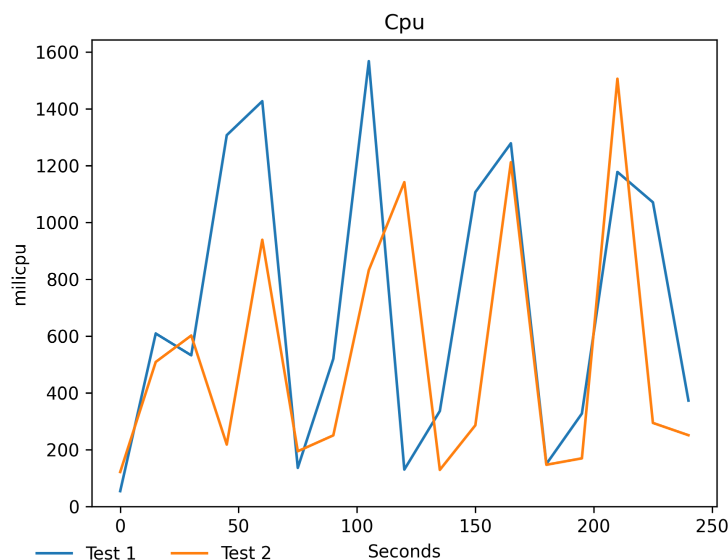


Figure 5.1: Overlaying two CPU recordings for Prometheus

5.6.1 Dynamic Time Warping

In Section 5.3 we already mentioned Dynamic Time Warping (DTW) as an option to compare two time series which contain a time shift. For DTW to work the best it is recommended to normalize the data first [18], so we applied DTW to a normalized version of the same time series and graphed the results.

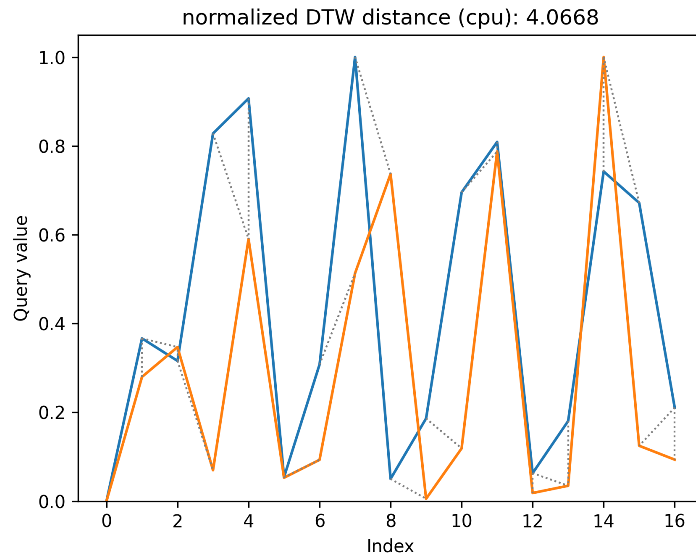


Figure 5.2: Applying Dynamic Time Warping to Prometheus CPU usage recording

As shown in Figure 5.2 Dynamic Time Warping successfully matched each spike and gave us a resulting DTW distance of 4.0668. Creating more recordings for the same pod, showed a variance of the DTW distance between 3.5 and 5 when referencing our baseline recording. The large deviation in the DTW distance shows that a single baseline recording will not be enough and it will be necessary to record it multiple times to make it reliable.

If our metrics could be collected at a higher frequency using DTW would certainly give us a good indicator if the resource usage pattern of a check run matches the ones we recorded in the baseline. But would still require interpretation of the DTW distance.

5.6.2 Statistical Summary

Another way to compare two timeseries, is to calculate statistical summaries and compare those. Albeit a more primitive approach it is something we considered early on as it is quite easy to calculate for each metric and, depending on the amount of data points we have, can give us an understandable way to compare those timeseries.

As we already normalized the time series for the Dynamic Time Warping calculation, we decided to also use the normalized data for the statistical summary, except for minimal and maximal values as they lose all their relevance during normalization.

Table 5.2 shows the statistical summary of our test recordings of Prometheus. As highlighted the standard deviation and variance in the calculated summary is quite low, we were able to confirm this observation, when we created more recordings and compared the resulting statistical summaries.

	CPU			Memory		
	Baseline	**Test**	**Diff**	**Baseline**	**Test**	**Diff**
mean	0.436	0.286	0.148	0.792	0.809	0.016
median	0.315	0.118	0.197	0.857	0.971	0.114
standard deviation	0.333	0.310	0.023	0.261	0.285	0.024
variance	0.111	0.096	0.014	0.068	0.081	0.013

Table 5.2: Statistical Summary on Prometheus resource usage

5.6.3 Metrics Based Heuristic

In our research we have identified that while Dynamic Time Warping is a well established process to compare timeseries, but in our use-case the recordings are quite limited making the calculation of a reference DTW distance more complex.

At the same time, we demonstrated that statistical summaries offer a simple yet effective method for comparing time series. Given their ease of calculation and interpretability, especially when dealing with short recording intervals, we selected statistical summaries as the primary approach for evaluating resource usage metrics during check runs.

But neither of these approaches is enough to reliably determine if a workload is functional or not on its own. So we need to make sure to also use other heuristics in our functionality oracle.

5.7 Analyzing Logs

Recording logs from a Kubernetes pod is rather simple as the container best practices define that each pod should write to *stdout* which is then collected by the container runtime and accessible through the Kubernetes API.

For all our test recordings we queried the container logs directly from the Kubernetes API without integrating it into our proof-of-concept CLI.

5.7.1 Drain

Drain is an algorithm which parses logs into templates and stores them as a fixed depth tree [16]. While it was first created as a research project, it was also adopted by IBM to analyze logs for the IBM Cloud, where syslog events of thousands of network devices were processed using Drain [21]. They also improved the algorithm further and published it as *Drain3*, which is available as an open source project. [8]

For our first analysis we used the log output of the first 5 minutes of runtime for a Prometheus instance, the instance was already well used and contained multiple gigabytes of time series data.

In our initial comparison we realized that each entry, which only existed once in our baseline, resulted in an anomaly when comparing to a test run. Investigating the templates generated by Drain we realized that, even though the logs are semi-structured, timestamps were not detected as tokens and the same log message with a different timestamp could not be matched successfully.

For example the following log entries were part of the baseline logs:

[8] `https://github.com/logpai/Drain3`

```
time=2025-07-25T15:58:22.588Z level=INFO source=main.go:725 msg="Starting Prometheus Server"
    mode=server version="(version=3.4.2, branch=HEAD,
    revision=b392caf256d7ed36980992496c8a6274e5557d36)"
time=2025-07-25T15:58:22.592Z level=INFO source=main.go:1266 msg="Starting TSDB ..."
time=2025-07-25T15:58:22.817Z level=INFO source=head.go:862 msg="WAL replay completed"
    component=tsdb checkpoint_replay_duration=42.820087ms wal_replay_duration=158.587917ms
    wbl_replay_duration=138ns chunk_snapshot_load_duration=0s mmap_chunk_replay_duration=4.466189ms
    total_replay_duration=205.908907ms
```

Listing 5.1: Baseline logs

And the following logs were part of a test run:

```
time=2025-07-25T15:59:42.428Z level=INFO source=main.go:725 msg="Starting Prometheus Server"
    mode=server version="(version=3.4.2, branch=HEAD,
    revision=b392caf256d7ed36980992496c8a6274e5557d36)"
time=2025-07-25T15:59:42.432Z level=INFO source=main.go:1266 msg="Starting TSDB ..."
time=2025-07-25T15:59:42.657Z level=INFO source=head.go:862 msg="WAL replay completed"
    component=tsdb checkpoint_replay_duration=40.269496ms wal_replay_duration=164.763458ms
    wbl_replay_duration=131ns chunk_snapshot_load_duration=0s mmap_chunk_replay_duration=1.826402ms
    total_replay_duration=206.89316ms
```

Listing 5.2: Test logs

Even a quick glance makes it obvious to a human reader that those logs are identical, but as the timestamp and other structured data like *checkpoint_replay_duration* only showed up once in the baseline, the algorithm was not able to tokenize the log entries resulting in the following log templates:

```
time=2025-07-25T15:58:22.588Z level=INFO source=main.go:725 msg="Starting Prometheus Server"
    mode=server version="(version=3.4.2, branch=HEAD,
    revision=b392caf256d7ed36980992496c8a6274e5557d36)"
time=2025-07-25T15:58:22.592Z level=INFO source=main.go:1266 msg="Starting TSDB ..."
time=2025-07-25T15:58:22.817Z level=INFO source=head.go:862 msg="WAL replay completed"
    component=tsdb checkpoint_replay_duration=42.820087ms wal_replay_duration=158.587917ms
    wbl_replay_duration=138ns chunk_snapshot_load_duration=0s mmap_chunk_replay_duration=4.466189ms
    total_replay_duration=205.908907ms
```

Listing 5.3: Drain Templates initial tests

The templates in Listing 5.3 are identical to the baseline input we provided which should have been expected as the algorithm is only able to generate templates based on comparing similar log entries.

Since we are looking for a solution which does not need any parameters to parse the templates we decided to use a second set of logs to train the algorithm, before trying to match the logs of a test run.

```
<*> level=INFO source=main.go:725 msg="Starting Prometheus Server" mode=server
    version="(version=3.4.2, branch=HEAD, revision=b392caf256d7ed36980992496c8a6274e5557d36)"
<*> level=INFO source=main.go:1266 msg="Starting TSDB ..."
<*> level=INFO source=head.go:862 msg="WAL replay completed" component=tsdb <*> <*> <*>
    chunk_snapshot_load_duration=0s <*> <*>
```

Listing 5.4: Drain Templates with two baseline recordings

In Listing 5.4, the timestamps as well as other dynamic fields are now detected by the algorithm and replaced with placeholder values. Using these templates to match the third set of logs, which we consider the test run logs, the algorithm no longer reported any anomalies as the logs could be matched to an existing template.

To verify that anomalies are found if a workload is failing for some reason we also created a Prometheus deployment which was missing the `ClusterRoleBinding` to monitor Kubernetes resources. This results in a running pod which is considered healthy by Kubernetes standards since the *Liveness-* and *Readiness-Probes* are successful, but in the logs we can see a lot of *Error* level logs which indicate a failure.

In addition to the logs shown in Listing 5.2 the failed run contains log messages like this:

```
time=2025-07-26T07:49:48.940Z level=INFO source=reflector.go:569
    msg="pkg/mod/k8s.io/client-go@v0.32.3/tools/cache/reflector.go:251: failed to list *v1.Node:
    nodes is forbidden: User \"system:serviceaccount:monitoring:prometheus-server\" cannot list
    resource \"nodes\" in API group \"\" at the cluster scope" component=k8s_client_runtime
time=2025-07-26T07:49:48.940Z level=ERROR source=reflector.go:166 msg="Unhandled Error"
    component=k8s_client_runtime logger=UnhandledError
    err="pkg/mod/k8s.io/client-go@v0.32.3/tools/cache/reflector.go:251: Failed to watch *v1.Node:
    failed to list *v1.Node: nodes is forbidden: User
    \"system:serviceaccount:monitoring:prometheus-server\" cannot list resource \"nodes\" in API
    group \"\" at the cluster scope"
time=2025-07-26T07:49:51.318Z level=INFO source=reflector.go:569
    msg="pkg/mod/k8s.io/client-go@v0.32.3/tools/cache/reflector.go:251: failed to list *v1.Service:
    services is forbidden: User \"system:serviceaccount:monitoring:prometheus-server\" cannot list
    resource \"services\" in API group \"\" at the cluster scope" component=k8s_client_runtime
```

Listing 5.5: Prometheus error logs

Matching the error logs with Drain, trained on two baseline sets, yielded the expected results that all *Error* log entries could not be matched and reported as anomalies. Since an error is reported every second, the amount of anomalies is overwhelming and we decided that we need a way to minimize the amount of anomalies reported. As we established previously matching the logs against other entries does not make sense, so we passed the reported anomalies through a new instance of the Drain miner, to match the anomalies against each other.

```
<*> level=INFO source=reflector.go:569
    msg="pkg/mod/k8s.io/client-go@v0.32.3/tools/cache/reflector.go:251: failed to list <*> <*> is
    forbidden: User \"system:serviceaccount:monitoring:prometheus-server\" cannot list resource <*>
    in API group \"\" at the cluster scope" component=k8s_client_runtime
<*> level=ERROR source=reflector.go:166 msg="Unhandled Error" component=k8s_client_runtime
    logger=UnhandledError err="pkg/mod/k8s.io/client-go@v0.32.3/tools/cache/reflector.go:251:
    Failed to watch <*> failed to list <*> <*> is forbidden: User
    \"system:serviceaccount:monitoring:prometheus-server\" cannot list resource <*> in API group
    \"\" at the cluster scope"
```

Listing 5.6: Prometheus error log templates

This resulted in just two additional templates, instead of hundred of error messages, while retaining the most important information about the errors. In this case that the `ServiceAccount` used by Prometheus does not have the permissions required to query the Kubernetes API server.

5.7.2 Log Based Heuristic

As we have seen in our tests it is possible to create a heuristic based on logs using existing algorithms which are able to be trained on a limited amount of logs.

As *Drain3* is an open source project, it should also be easily possible to port it to Go if there is no Go module yet.

For the log analysis it is important that we record more than one baseline, otherwise the log templates can not be extracted, and before reporting the anomalies to the user it is also useful to run them through Drain to only report the templates of an error.

5.8 Heuristic to determine Workload functionality

Based on our findings in Section 5.5 and 5.7 we need a baseline to compare our signals to. If we just take the already running workload, we might encounter a workload that is already running for multiple hours and if we try to compare the logs that we gathered from a freshly started workload they will be different. The same issue exists for metrics gathered from an already running application, as that application might have a memory leak, or just a constant allocation that will look differently if it has just started.

One way to avoid those issues, is to restart the workload and get a fresh baseline for all attributes. This gets rid of any interference from what is already running and can be sure that we compare the same logs and Events over the lifetime of a `Pod`.

Ideally we are able to clone an existing application deployment to generate baseline recordings, and then clone each test case with adjusted configurations, to avoid any side effects from failing test cases. This also allows to run multiple test in parallel and reducing the runtime for the full check.

5.8.1 Functionality Test Cases

To confirm functionality we need to define test cases which are run against each pod after changing a configuration.

ID	Title	Description	Datasource
TC-01	Pod Stability	Detect if the container terminated unexpectedly during execution.	Probes
TC-02	Pod Readiness	Identify if the container fails its healthiness probe during the test window.	Probes
TC-03	Log Pattern Matching	Compare runtime logs to baseline and detect anomalies.	Logs
TC-04	Resource Usage Patterns	Verify whether CPU or memory usage patterns match the baseline.	Metrics

Table 5.3: Functionality test cases for Kubernetes workload

These test cases are ordered by their reliability and the resulting priority. If a container crashes during startup it is save to assume that the applied configuration does not work, and the *Health-* and *ReadinessProbes* never succeed. If a container never passes the probes, it will not be flagged as ready. In those cases we can consider the check failed even without analyzing the logs, but we can still use log analysis to help the user find the failure and hopefully adapt their application to work given the new restrictions.

In Section 5.6.3 we concluded that metrics alone are not reliable enough to confirm if a workload is functional, that is why we give them the lowest priority and only use them to detect outliers, but if the previous checks were all successful flagging a check as failed just due to the metrics requires careful consideration.

6 System Architecture

To test a workloads security context and automatically provide a recommendation how to further restrict the `Pods` capabilities it is necessary to start the workload in Kubernetes, and then gradually restrict the workload runtime environment.

In Section 5.8 we concluded that the most viable heuristics to validate a workloads functionality is by using a combination of built-in signals, namely the *Liveness*- and *Readiness-Probes*, and container logs which can be analyzed using Drain.

For all heuristics it is necessary to integrate with the Kubernetes API, so our most important requirement for our solution is being able to access the API without interference or rate limits.

6.1 Where to run

Evaluating an applications security restrictions, is something that needs to be done on a regular basis since each code change could result in changes to the capabilities needed to run the application. Usually tasks that should be run on each code change, are run in a CI/CD pipeline, but as we require a Kubernetes cluster to deploy the application and then test its permutations, this will require access to a Kubernetes cluster from the CI/CD pipeline

Another option would be to run the application as an operator within Kubernetes. This ensures that we are already in a cluster, and allows the solution to easily make use of Kubernetes native resources.

6.1.1 CI/CI Pipeline

CI/CD pipelines are often used to automate recurring tasks, especially those related to code changes. This makes them an invaluable tool in the life of developers. But developers also want their pipelines to be fast and robust, to quickly get feedback if their code change has an unexpected impact. If a pipelines takes too long, developers end up switching context and work on different tasks. Coming back and debugging the pipeline is again another context switch which often reduces productivity.

The checks we intend to perform, are inherently long running, as we need to collect metrics and logs for at least a few minutes to have enough data to conclude if the workload is functional. Accordingly a job like this, should not be run on each commit, but only whenever a merge is to be performed.

While this pattern would allow us to run the checks on each change, its implementation highly depends on the CI/CD software used. The most universal approach would be to write a CLI, which can be triggered from the job definition, based on manifests present in the repository. But this will still require access to a Kubernetes cluster where the tests can be performed and does not solve the issue of the execution duration.

6.1.2 Kubernetes Operator

In Kubernetes, operators are the most common way to introduce custom functionality, that interacts with multiple resources in the cluster. It allows us to define a Custom Resource Definition (CRD) that contains all configurations necessary to start a workload runtime hardening check, which are then interpreted by our operator which performs the heavy lifting. [11]

The advantage of this approach is that the application creating, monitoring and deleting the checks, is already running in a Kubernetes cluster, and no additional cluster is necessary. But

it also requires that the operator was installed on the cluster previously. Depending on the setup within a company, installing an Operator needs to be done by the platform team, which is responsible for cluster operations.

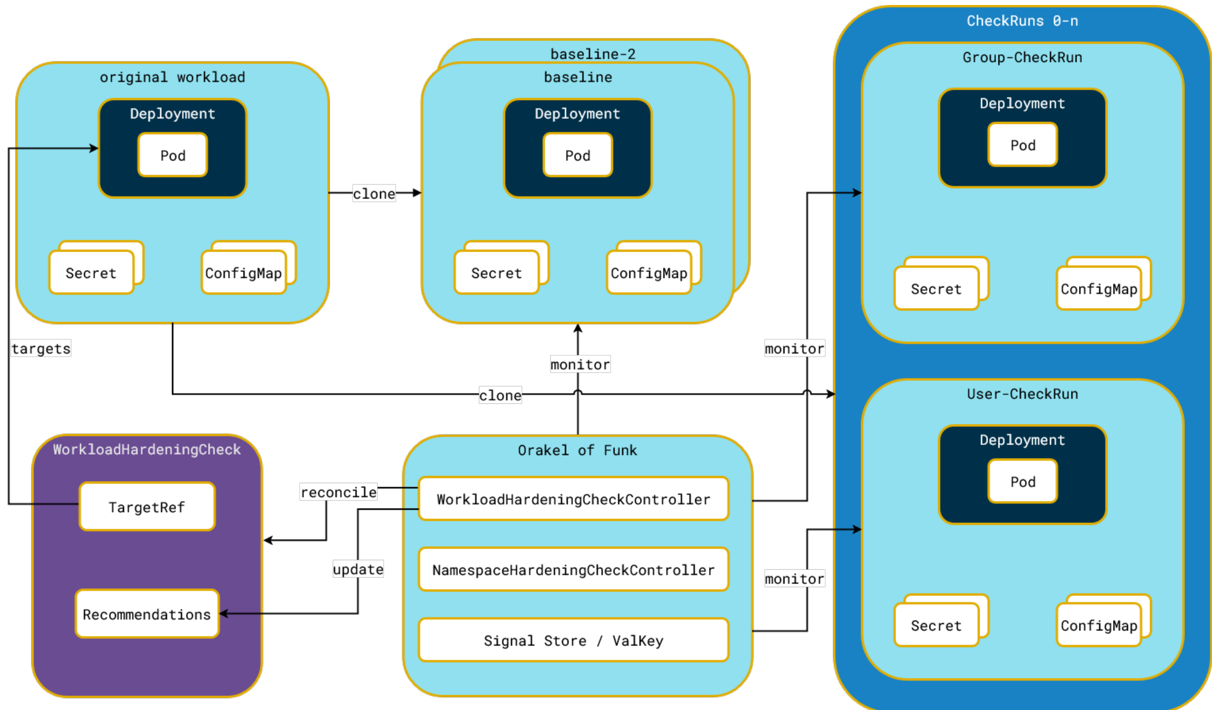


Figure 6.1: Graph outlining operator based oracle architecture

Figure 6.1 shows a high-level view, how the operator interacts with the `WorkloadHardeningCheck` resource to perform `CheckRuns` on a target workload.[9]

As the `NamespaceHardeningCheck` resources operates on a `Namespace`-level, it can reuse the `WorkloadHardeningCheck` by analyzing the target `Namespace` and create a `WorkloadHardening Check` for each workload which is not managed by another operator. After all `WorkloadHardening Checks` are finished, the `NamespaceHardeningCheckController` only needs to gather the recommended *securityContext* and report them to the user.

6.2 Solution Execution Flow

The core of our solution is to automate the hardening of the workload through dynamic runtime adjustments and evaluation. As documented in Section 5.8 we need a baseline to compare each check against, which is achieved by recording the unmodified workload twice.

Depending on the existing *securityContext* on a workload not all checks need to be executed, so the solution needs to analyze the current deployment and plan what checks are necessary.

Each check should run independently and report its state back to the operator in a simple way, that can be checked each time the CRD is reconciled.

After running the checks, the operator should generate the recommended *securityContext* which is the combination of all flags configured in the successful checks. But to make the most of the operator, we should also perform a final check run where the recommended *securityContext* is used to validate our recommendation.

[9] Figure 6.1 was updated after the implementation phase and already contains the `ValKey` instance used to store signals as described in Section 7.3.3

This results in the following execution flow.

1. **Baseline Recording:** The Namespace containing the original workload is cloned and signals (logs and metrics) are recorded.
2. **Check Planning:** Based on the workload specification, the operator determines which `securityContext` attributes should be tested.
3. **Check Execution:** For each required check, a separate clone of the original Namespace is created. The workload is deployed with the check-specific changes and, logs and metrics are recorded.
4. **Signal Comparison:** After all check runs complete, the recorded signals are compared to the baseline.
5. **Recommendation Synthesis:** The `securityContext` modifications for all passed checks are aggregated.
6. **Final Verification:** A final Namespace clone is created, in which the workload is deployed with the aggregated recommendations.

This modular execution flow ensures that each restriction is validated, that recommendations are based on observable behavior, while avoiding any assumptions about the workloads internal logic. All evaluation steps are performed in cloned `Namespaces`, preserving isolation and preventing any unintended side effects on production deployments.

7 Implementation and Evaluation

7.1 Choice of Framework: Operator SDK

The implementation of the workload hardening logic is encapsulated in a Kubernetes Operator. For this project, the *Operator SDK*[10] was chosen. The Operator SDK provides a structured and well-documented approach to building, packaging, and deploying Kubernetes Operators, while integrating with existing Kubernetes tooling.

Several factors influenced this decision:

- **Integration with Kubernetes API Machinery:** The Operator SDK builds on the `controller-runtime` library and client-go modules, providing a consistent to implement reconciliation logic, manage resource events, and handle status updates.
- **Scaffolding and Code Generation:** The framework offers scaffolding commands for quickly generating boilerplate code for Custom Resource Definition (CRD), controllers, and webhook configurations. This reduced the initial setup effort and ensured that the generated code follows established best practices.
- **CRD Schema Management:** With built-in support for OpenAPIv3 schema annotations, the Operator SDK simplifies the definition and maintenance of CRDs. This is particularly useful when new fields need to be added during development.
- **Local Development and Testing:** The SDK provides tools for running the operator locally against a remote or in-cluster API server, facilitating rapid development and debugging cycles with the exception of webhooks where it is not possible to perform the callback to a locally running controller.
- **Community Support and Ecosystem:** As a widely adopted CNCF project, the Operator SDK benefits from a large user base, active development, and comprehensive documentation, reducing the risk of adopting unsupported or unstable tooling.

Given these advantages, the Operator SDK was the most suitable choice for implementing a complex, resource-driven workflow. It allowed the focus to remain on the domain-specific logic of signal collection, evaluation, and recommendation synthesis, rather than on low-level controller infrastructure.

7.2 Operator Execution Flow

The operator reacts to custom resources of kind `WorkloadHardeningCheck`, each of which targets a specific workload. The assumption is that a user creates a dedicated check resource for each workload they want to harden. Upon reconciliation, the operator executes the steps described in 6.2.

To track the progress during the `Check Execution` stage, we decided to use the *StatusConditions* on the `WorkloadHardeningCheck` resources. For each `CheckRun` a *StatusCondition* is created, and the progress is tracked through the *Status* and *Reason* attributes.

The state diagram in Figure 7.1 models the lifecycle of a single check run. A check run always starts in state `1`, indicating that it has been registered but execution has not yet begun. When the controller initiates the check, the `CheckRunner` transitions it to state `2`, representing an active recording phase. From here, three outcomes are possible:

[10] `https://sdk.operatorframework.io/`

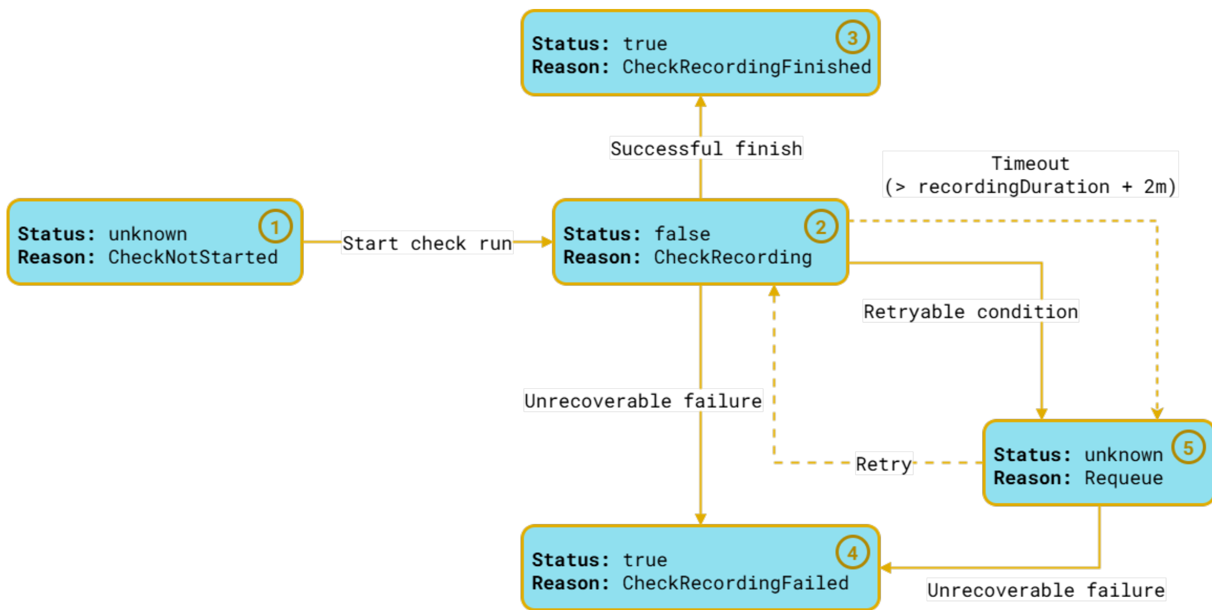


Figure 7.1: CheckRun `StatusCondition` state diagram

3. **Successful finish:** The `recording` was successful, but this does not yet indicate that the `CheckRun` was successful as the results are not analyzed yet. This is a terminal state.
4. **Unrecoverable failure:** During recording there was an unrecoverable error, and the `CheckRun` is flagged as failed. This is a terminal state.
5. **Recoverable failure:** A recoverable error occurred and the `CheckRun` should be retried. This can happen if a `CheckRun` does not finish in time, for example if the operator was restarted, or if the pods could not be scheduled in time due to limited cluster resources. This is not a terminal state.

The state transitions depicted as dashed arrows, are performed by the `WorkloadHardening Controller`, while the solid arrows indicate state transitions issued by the `CheckRunner`.

Using these state transitions allows tracking each *CheckRun* including the baseline as well as the final check run.

7.3 Signal Collection and Storage

To determine whether a workload remains functional under different runtime restrictions, the operator collects the two categories of telemetry data described in Section 5.5 and 5.7 : container logs and pod resource metrics. The processes established during those tests, were successfully integrated into our operator.

These signals form the basis for comparing the original workload behavior with the behavior observed under modified *securityContext* settings.

7.3.1 Metrics Collection

As established during our proof of concept implementation, gathering the resource metrics directly from each `Node` is the best solution. But since there is a delay between a `Pod` starting and the kubelet reporting its resource usage, there are some cases, where no metrics can be collected. For example `Pods` which crash in the first few seconds, will not get reported by the kubelet. But since they already fail *TC-01* we do not have to rely on the metrics to establish that they are not functional.

The provided metrics, consisting of CPU usage and memory consumption, are collected periodically and stored for later analysis.

7.3.2 Log Collection

Logs are collected via the Kubernetes API by streaming the output of each container in the pod. This method is compatible with all workloads that write to standard output and does not require sidecar injection or external logging infrastructure. The operator opens a log stream to each container during the recording phase of each run. The logs for each container are stored and analyzed according to the heuristic process described in Section 5.7.2 separately to avoid interference.

7.3.3 Storage Backend

All collected signals, both logs and metrics, are stored in a `ValKey` instance that is deployed alongside the operator. `ValKey` offers low-latency access and simple key-value semantics, making it well suited for storing time-series and log data with minimal operational complexity.

To avoid excessive storage usage, each set of recorded signals is assigned a one-day expiry. The expiry semantics are enforced using `ValKey's` built-in key expiration mechanism, ensuring that outdated data is automatically removed.

7.4 Evaluation Heuristics and Test Cases

Each check run applies a modified *securityContext* to the workload and observes whether the workload continues to function correctly. To determine success or failure, the four test cases established in Section 5.8.1 are evaluated against the signals recorded during each check run.

7.4.1 TC-01: Pod Stability

The first condition is that the pod must not crash or enter a *CrashLoopBackOff* state during the evaluation period. This is verified by inspecting the pod status via the Kubernetes API. A crashing pod immediately invalidates the corresponding check run.

7.4.2 TC-02: Pod Readiness

The second condition requires that the pod reaches the *Ready* state according to its configured *Liveness*- and *Readiness-Probes*. This ensures that the application inside the container is not only running, but also responsive according to the workload's health check configuration. These probes are evaluated natively by Kubernetes.

7.4.3 TC-03: Log Pattern Matching

To further verify that the application behavior has not changed under the modified runtime constraints, logs from each check run are compared against a previously recorded baseline. As outlined in Section 5.7.2 the baseline is collected twice ensuring sufficient variety for meaningful template generation by the Drain algorithm which is used for log comparison.

For this project, a Go implementation [11] of the Drain algorithm was used to extract templates from the baseline and match them against the check run logs. Anomalous log lines that do not match any known template are considered indicators of functional deviation.

[11] `https://github.com/faceair/drain`

7.4.4 TC-04: Resource Usage Patterns

A final check is performed on resource usage metrics. The operator compares the statistical properties of CPU and memory consumption during the check run and the baseline. Normalized values are used to compute metrics such as variance and standard deviation, while the raw values were used for minimum/maximum. Large deviations may indicate altered workload behavior, potentially due to failing internal retries, disabled features, or other effects of the enforced restrictions.

These four test cases are applied in sequence to determine whether the check run passed. If any of *TC-01*, *TC-02* or *TC-03* fail, they cause the check run to be marked as failed and corresponding *securityContext* modifications to be excluded from the final recommendation.

7.5 Check Design

The set of checks performed by the operator is derived from the available fields in the Kubernetes *securityContext* and *podSecurityContext* specifications as described in Table 2.1. Each check evaluates whether the workload remains functional when one or more of these attributes are set. Where possible, checks were designed to isolate individual settings, but in some cases, interdependencies between fields required grouped evaluations.

Each check sets one or more *podSecurityContext* attributes, but only after ensuring that the target workload does not have that attribute set. In case of grouped checks, the attributes are copied from the `Pod` to `Container` level to ensure consistent configurations.

7.5.1 Grouped Checks

Certain fields are semantically or functionally dependent on each other. For example, setting *runAsGroup* at the container level often requires the corresponding *fsGroup* to be set at the pod level, especially for file system access involving shared volumes. Additionally the container sandbox requires the *runAsUser* attribute to be configured if the *runAsGroup* attribute is set. As a result, these fields are combined into a single *GroupCheck*, which sets:

- *fsGroup* (pod)
- *runAsGroup* (pod & container)
- *runAsUser* (pod & container)

Another grouped example is the *UserCheck*, which sets both:

- *runAsUser* (pod & container)
- *runAsNonRoot* (pod & container)

These combinations are necessary to ensure that the workload does not fail due to incomplete or inconsistent securityContext definitions.

7.5.2 Isolated Checks

For attributes without known dependencies, dedicated checks were created. Examples include:

- *readOnlyRootFilesystem*
- *allowPrivilegeEscalation*
- *capabilities.drop*

Each of these is evaluated independently, allowing the operator to identify the minimal set of constraints that do not interfere with application functionality.

7.6 Execution Modes and Configuration

To accommodate different operational environments, we wanted to give the user flexibility how the checks are performed. This requires exposing configurations which we have done using the `WorkloadHardeningCheck` and `NamespaceHardeningCheck` custom resources.

7.6.1 RunMode: Parallel vs Sequential

The execution of hardening checks can be configured to run either sequentially or in parallel. This is controlled through the *runMode* field, which accepts two values:

- `Sequential`: Checks are executed one after the other. This mode is preferred when shared resources (e.g., persistent volumes or external systems) could be impacted by concurrent execution.
- `Parallel`: All checks are launched concurrently. This significantly reduces the overall evaluation time but requires more cluster resources during execution.

Each check is executed isolated to prevent blocking the reconciliation loop. Synchronization and result collection are handled internally within the operator logic.

In either case the two baseline recordings are run in parallel with a slight delay between the first and second one. This delay ensures that timestamps in log messages are not identical which would be a problem for the log analysis.

7.6.2 Workload Selection

Each `WorkloadHardeningCheck` targets a specific workload by specifying a `Namespace` the kind to check and its name.. The operator determines which resources match the selector and identifies the top-level owner objects for cloning. This ensures that only controller-managed workloads (e.g., `Deployments`, `StatefulSets`) are evaluated, avoiding lower-level `ReplicaSets` or pods.

```yaml
apiVersion: checks.funk.fhnw.ch/v1alpha1
kind: WorkloadHardeningCheck
metadata:
  name: nginx-unprivileged
  namespace: oof-nginx-unprivileged
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-unprivileged
  recordingDuration: 1m
```

`WorkloadHardeningCheck`

```yaml
apiVersion: checks.funk.fhnw.ch/v1alpha1
kind: NamespaceHardeningCheck
metadata:
  labels:
    app.kubernetes.io/name: podtato
  name: podtato
spec:
  targetNamespace: podtato-kubectl
  recordingDuration: 1m
```

`NamespaceHardeningCheck`

Figure 7.2: `WorkloadHardeningCheck` and `NamespaceHardedningCheck` examples

For the `NamespaceHardeningCheck` the user only configures the *targetNamespace* and the workloads in the `Namespace` are automatically detected. For each detected workload a `Workload HardeningCheck` is created.

Currently only `Deployment`, `StatefulSet` and `DaemonSet` resources are supported, but extending it for `Jobs` and `CronJobs` should be easily possible as those also rely on the *PodSpec*.

Custom resources like the Prometheus resource provided by the Prometheus-Operator, are more complicated to handle, and thus were excluded for the initial implementation.

7.6.3 Evaluation Duration

The duration of each evaluation run is controlled via the *recordingDuration* field, with a default value of 5 minutes. During this window, logs and metrics are recorded for both the baseline and all check executions. The duration must be sufficiently long to allow the application to start and stabilize, but short enough to limit resource consumption and enable a short feedback loop.

7.7 Results Reporting

The outcome of a `WorkloadHardeningCheck` is reported through the `status` field of the corresponding custom resource. This includes both the progress of each individual check and the final recommendation, allowing users to track execution and retrieve results easily.

7.7.1 Status Tracking

As each phase of the hardening process progresses, the operator updates the resource's status with structured metadata. This includes:

- Current phase (e.g., `BaselineRecording`, `RunningChecks`, `Completed`)
- List of executed checks and their state (e.g., `Pending`, `Running`, `Passed`, `Failed`)
- References to any detected anomalies or failed test cases

This information provides visibility into the internal state of the operator and facilitates debugging in case of misconfiguration or unexpected workload behavior.

7.7.2 Recommended Configuration

After all checks have been executed and evaluated, the operator synthesizes a recommended *securityContext*. This is constructed by aggregating all the modifications from successful checks. The recommendation is separated into two parts:

- *podSecurityContext*: for pod-level attributes
- *securityContext*: for container-level attributes

To validate the correctness of the combined configuration, the `Namespace` containing the target workload is cloned one more time and the recommended *securityContext* is applied. This *FinalCheckRun* must pass all test cases before the recommendation is published in the *status* of the `WorkloadHardeningCheck` resource.

```yaml
conditions:
- lastTransitionTime: "2025-08-13T16:51:36Z"
  message: Finished, recommendation ready
  reason: Finished
  status: "True"
  type: Finished
recommendation:
  containerSecurityContexts:
    allowPrivilegeEscalation: false
    capabilities:
      drop:
      - ALL
    readOnlyRootFilesystem: true
    runAsGroup: 1000
    runAsUser: 1000
  podSecurityContext:
    fsGroup: 1000
    runAsGroup: 1000
    runAsNonRoot: true
    runAsUser: 1000
```

Figure 7.3: Recommended *securityContext* in a `WorkloadHardeningCheck`

For the `NamespaceHardeningChecks` an additional *FinalCheckRun* is created, where the recommended *securityContext* is applied to all workloads and they are tested for functional correctness. If this *FinalCheckRun* is successful, the `NamespaceHardeningCheck` is considered successful and the recommendation for all tested workloads is published in the *status* similar to the `WorkloadHardeningCheck`

This staged and modular approach to hardening allows for precise attribution of failures and avoids applying unnecessary or conflicting restrictions.

7.8 Evaluation Setup

To validate the correctness, robustness, and applicability of the implementation, a combination of real-world workloads and purpose-built test cases was used. This ensured both broad coverage and fine-grained control over expected behaviors.

7.8.1 Real-World Workloads

Three commonly deployed Kubernetes applications and one showcase application were selected as representative test subjects:

- **Prometheus**: A monitoring stack with persistent volumes and custom user permissions.
- **ArgoCD**: A multi-component GitOps deployment tool with fine-grained RBAC and sidecars.
- **MariaDB**: A stateful database workload with strict volume mount and process ownership requirements.
- **Podtato-Head**: A cloud-native application built to colorfully demonstrate delivery scenarios using different tools and services. [12]

All applications were deployed using their official Helm charts with default `values.yaml` configurations. These workloads represent realistic deployment patterns and provide a baseline for evaluating the operator's ability to apply restrictions without compromising functionality.

Podtato-Head was chosen as an additional example workload, as it reflects a microservice pattern where each service can be hardened individually. At the same time it is a very simple workload without any RBAC configurations as it does not interact with the Kubernetes API.

7.8.2 Modified NGINX Scenarios

To test specific edge cases, a set of three workloads was created using the official `nginx` image:

- A default deployment running as `root`, with write access to system directories.
- A modified version running as a non-root user, serving traffic on an unprivileged port, with logs and runtime files moved to `/tmp`.
- An extended version that mounts an `emptyDir` volume at `/tmp`, allowing compatibility with `readOnlyRootFilesystem`.

These scenarios were designed to validate how specific securityContext attributes interact with file system layout, user permissions, and process ownership.

7.8.3 Purpose-Built Workloads

Seven minimalist workloads were constructed using single-container Pods running custom Bash scripts. Each workload explicitly exercises a security boundary:

- **Chown**: Attempts to invoke `chown`, requiring the `CHOWN` capability.
- **AllowPrivEscalation**: Attempts to enable privilege escalation.
- **WriteFs**: Writes to `/tmp` with and without `emptyDir`.
- **RunAsRoot**: Binds to privileged ports and examines `UID`.
- **BindPort**: Attempts to bind to port 80 and port 1000, revealing runtime differences.

These workloads allow isolated testing of specific constraints, enabling deterministic outcomes and easier root cause analysis.

7.8.4 Container Runtime Variance

An unexpected outcome occurred during the `BindPort` tests, where the container was able to bind to port 80 even when running as a non-root user. Investigation revealed that certain container runtimes (e.g., Docker and containerd) modify the sysctl parameter `net.ipv4.ip_unprivileged_port_start` to `0`, allowing unprivileged processes to bind to traditionally restricted ports. [13] [14]

[12] https://github.com/podtato-head/podtato-head-app
[13] https://github.com/containerd/containerd/issues/6924
[14] https://github.com/moby/moby/pull/41030

This behavior is not universal across all container runtime engines and highlights a portability consideration when interpreting check results.

The combination of realistic and synthetic evaluation targets provides strong confidence that the operator performs as expected under a variety of runtime conditions and workload configurations.

7.9 Observations and Limitations

Throughout the development and evaluation of the operator, several practical challenges and limitations were encountered. These informed design decisions and highlight areas for future improvement or caution when adopting the tool.

7.9.1 Concurrency Model

Each check run is executed in a dedicated Go routine to prevent blocking the main reconciliation cycle. While this model works well in practice, it introduces concurrency challenges, particularly when updating shared resources such as the parent custom resource. Synchronization and error propagation require careful handling.

An alternative design, using dedicated Kubernetes `Job` resources for each check, was considered. However, this would necessitate a second controller to run each check and report back, significantly increasing complexity. Furthermore, the issue of race conditions during status updates would not be inherently resolved by this model.

7.9.2 Resource Update Semantics

Kubernetes enforces strict versioning of resources via the *resourceVersion* field. When performing updates to existing resources, such as tracking progress in the `WorkloadHardeningCheck` status, this requires fetching the most recent version of the resource, applying changes, and retrying in case of conflicts.

This pattern led to a substantial amount of boilerplate code within the operator, due to the concurrency model described above. To mitigate this, the implementation relies on the Kubernetes client-go retry helpers to manage transient failures and ensure eventual consistency.

7.9.3 Signal Volume and API Server Load

Early prototypes attempted to store all collected logs and metrics directly in the status field of the custom resource. This approach proved imprractival for workloads with extensive log output or long evaluation durations, as it led to large resources and potential API server degradation.

As a result, the `status` field now contains only metadata and summary information. The full logs and metrics remain accessible from ValKey, but are no longer persisted in the Kubernetes API.

This separation balances visibility and scalability, ensuring that users have access to actionable recommendations without risking performance penalties for large custom resource definitions.

7.9.4 Stateful and Multi-Component Workloads

Certain workloads, such as databases or distributed systems (e.g., ArgoCD), may behave differently across cloned namespaces due to external dependencies, persistent volume claims, or

coordination mechanisms. While the operator avoids modifying the original `Namespace`, replication fidelity is not guaranteed for complex topologies.

This limitation suggests that the tool is best suited for stateless or loosely coupled workloads, but as can be witnessed from our tests using MariaDB as a test workload, it is also possible to harden stateful applications using our functionality oracle.

As shown in 5.7 missing `RoleBindings` for a `ServiceAccount` can lead to failing workloads. In our tests this affected both Prometheus as well as ArgoCD. As we only cloned namespace-scoped resources, the `ClusterRoleBinding` they rely on, did not get cloned and did not include the `ServiceAccounts` in the cloned `Namespaces`. After adding a specific implementation to ensure `ClusterRoleBindings` were cloned as well, the tests for Prometheus and ArgoCD could be concluded successfully.

7.10 Summary

The implementation demonstrates a practical and modular approach to automatically hardening Kubernetes workloads by observing and evaluating their runtime behavior under security constraints. By leveraging the extensibility of the Kubernetes operator pattern, the tool performs deep introspection on application behavior without requiring knowledge of internal logic or test suites.

The core contribution lies in implementation of the functionality oracle which dynamically evaluates *securityContext* attributes across isolated namespace clones, using our multi-layered heuristics:

- Kubernetes-native readiness and stability indicators
- Structural log analysis using the Drain algorithm
- Statistical comparison of resource usage metrics

The system has been designed with flexibility and observability in mind. Users can tune execution modes and durations, while results are exposed through structured status fields and external telemetry storage. The separation of signal collection from control-plane resources ensures operational safety and scalability.

Real-world workloads and purpose-built test cases validated the tool's effectiveness across a diverse range of scenarios. While some runtime-specific behaviors and Kubernetes API semantics introduced complexity, the design proved resilient and adaptable.

The operator is particularly well-suited to ensure best practices during development by continuously hardening the application, as well as facilitating the introduction of third party software by providing a simple way to validate and harden deployment configurations provided by the supplier.

In conclusion, the implementation provides a reusable and portable foundation for security hardening at the workload level, enabling secure-by-default deployments without relying on handcrafted test logic.

8 Conclusion and Future Work

8.1 Key Insights and Contributions

The work presented in this thesis contributes to the domain of Kubernetes workload security and runtime verification by proposing a functionality oracle for hardening workloads. Several key insights emerged during the development and evaluation of the heuristic processes:

- **Security hardening is non-trivial:** Precisely because Kubernetes offers comprehensive mechanisms to define and restrict container runtime behavior via `securityContext` settings, it is difficult for users to anticipate their impact on application functionality, particularly for Linux capabilities, which demand both Kubernetes-specific and operating system-level expertise.
- **Baseline recording enables functionality inference:** By recording the behavior of a known functional deployment and comparing it against subsequent executions with increasing restrictions, it is possible to infer which constraints break or preserve functionality. This approach reduces the need for prior knowledge about the application.
- **Log-based heuristics are effective:** The black-box methodology employed by the operator relies solely on observable signals, such as container logs, without requiring instrumentation or internal knowledge of the workload. Within this context, log-based anomaly detection demonstrated high reliability in detecting deviations from expected behavior, and proved to be a robust mechanism for determining functional correctness.

These insights support the thesis that functionality-based hardening is feasible, and can be automated through the combination of operator mechanisms, baseline inference, and runtime signal analysis.

8.2 Strengths and Limitations of the Operator-based Approach

The implementation of the solution as a Kubernetes Operator proved to be a suitable architectural choice. It provided native integration with the cluster environment and enabled comprehensive observation and manipulation of workloads during the hardening process.

One of the main strengths of the operator-based approach is its proximity to all relevant components. It can observe application behavior in real time, clone namespaces, manage security settings, and collect relevant signals, all from within the cluster. This facilitates a seamless and automated feedback loop for iterative hardening.

Another significant benefit is the use of custom resources for coordination and user interaction. By encoding each check as a `WorkloadHardeningCheck` resource, the operator leverages Kubernetes-native patterns for declarative interaction, status tracking, and event-driven execution.

However, some limitations were also identified. The most prominent challenge lies in managing concurrent status updates to the custom resource. Since the operator updates the CRD status with the outcome of each individual check, concurrent execution can lead to conflicts or inconsistencies. While the current implementation handles these scenarios functionally, it increases the size and complexity of the resource, and complicates readability for users.

Finally, from a platform compatibility perspective, the implementation was kept deliberately generic. By relying on direct queries to the Kubernetes API instead of metrics-server or distribution-specific components, the operator maintains broad compatibility across clusters. The only requirement is the ability to install CRDs, which is typically available in standard Kubernetes environments.

8.3 Evaluation Highlights

The evaluation phase provided a range of practical insights into the behavior of different workload types under the proposed approach. These observations showed both the effectiveness and the limitations of the current implementation.

Stateless applications proved to be the most straightforward category to harden. Their simplicity in deployment and lack of persistent state made it easy to replicate their runtime environment and monitor behavior. However, they introduced a subtle risk: if the application only exposes functionality under specific runtime conditions, such as handling requests or processing events, a test execution that does not trigger those conditions may lead to an overly restrictive recommendation. This illustrates the need for ensuring realistic workload activity during the evaluation.

Multi-container applications presented a more fundamental challenge. For the initial implementation the `WorkloadHardeningCheck` targets a single workload in isolation, but applications often consist of multiple interdependent workloads. The addition of a `NamespaceHardeningCheck` turned out to be relatively simple, as we were able to scan the target `Namespace` for all workloads and create `WorkloadHardeningChecks` for each one. This showed that the architecture decisions made are a solid foundation to add more features and checks in the future.

One unexpected finding emerged while implementing the check for privileged port binding as described in Section 7.8.4. This resulted in behavior that contradicted initial assumptions, but as we established the result reported by the operator was correct, which proved that the heuristic process works.

Overall, the evaluation confirmed the viability of the baseline-based hardening approach and exposed opportunities for extending its scope to better support complex or tightly integrated workloads.

8.4 Opportunities for Technical Improvements

Several avenues for improving the current implementation remain. These improvements range from architectural refinements to functional extensions that would broaden the scope and applicability of the solution.

One potential improvement lies in exposing the functionality through a dedicated Command Line Interface (CLI). This CLI would allow teams to run hardening checks in a Kubernetes cluster independently of the operator, providing more flexibility for integrating the solution into different workflows. This would be particularly valuable in CI/CD environments, where a lightweight and self-contained tool could trigger hardening checks and return results without requiring controller installation.

Other improvements are the consolidation the baseline recording for `NamespaceHardeningChecks`. As they currently rely on creating a `WorkloadHardeningCheck` for each supported resources, they also create two baseline recordings each. Instead the `NamespaceHardeningCheck` should create baseline recordings for each workload and pass them to the `WorkloadHardeningCheck` reducing the resources used for testing.

In the current implementation the recommended *securityContext* is shown as part of the same resource the user creates to trigger the testing. For the `NamespaceHardeningChecks` this is not an issue, as the resource does not contain much information otherwise. In case of the `WorkloadHardeningCheck` it is less ideal, since that resource already contains a lot of information about each check run performed. To reduce the information shown to the user, the

`WorkloadHardeningCheck` could generate a new Kind of resource called `WorkloadHardening Report` which only shows the recommendation and results of failed *CheckRuns*.

These enhancements would contribute both to the robustness and the accessibility of the hardening framework, making it more suitable for a wider variety of deployment models.

8.5 Future Research Directions

Beyond immediate technical improvements, several avenues for future research emerged that could build upon the concepts introduced in this work. These directions focus on enhancing the precision, generality, and applicability of the functionality oracle.

A potential area is the integration of large language models (LLMs) for log analysis. The current heuristic-based approach to log comparison has proven to be effective, but remains limited to structural similarity and tokenized pattern matching. Leveraging LLMs could enable a deeper semantic understanding of log content, allowing the system to detect subtler forms of functional change. Future research should first investigate the amount and type of signals required, to reliably model application behavior.

Another area of interest is the extension of the functionality oracle for regression testing during application upgrades. Given a known-good baseline, the same signal analysis techniques could be applied to compare the behavior of a new version of the workload. However, challenges arise due to typical changes in log output—such as version strings or configuration messages, which could trigger false positives. Investigating log normalization techniques or adaptive heuristics that allow controlled variance could make this use case practical. This scenario also introduces the need to manage multiple workload versions concurrently, either through side-by-side deployments or declarative upgrade instructions.

Finally, broader adoption of the approach would benefit from a focus on complex workloads composed of multiple, interdependent components as well `CustomResourceDefinitions`. Research into how such systems can be tested, while preserving the principles of minimal prior knowledge and black-box evaluation, would help generalize the methodology for a larger class of Kubernetes applications.

Together, these directions aim to extend the solution into a more adaptive, intelligent, and widely applicable framework for workload evaluation and hardening.

References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015. DOI: 10.1109/TSE.2014.2372785.

[2] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*. O'Reilly Media, 2022, ISBN: 9781098110178.

[3] "Configure a Security Context for a Pod or Container," Kubernetes. (Nov. 19, 2024), [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/ (visited on 03/21/2025).

[4] F. Minna, A. Blaise, F. Massacci, and K. Tuma, "Automated security repair for helm charts," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion '24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 412–413, ISBN: 9798400705021. DOI: 10.1145/3639478.3643534. [Online]. Available: https://doi.org/10.1145/3639478.3643534.

[5] M. Richards and N. Ford, *Fundamentals of Software Architecture, An Engineering Approach*. O'Reilly Media, 2020, ISBN: 9781492043423. [Online]. Available: https://books.google.ch/books?id=xa7MDwAAQBAJ.

[6] I. Sommerville, *Software Engineering* (International computer science series). Pearson, 2011, ISBN: 9780137035151.

[7] S. Newman, *Building Microservices, Designing Fine-Grained Systems*, Second edition. O'Reilly Media, 2021, ISBN: 9781492034025.

[8] P. Ammann and J. Offutt, *Introduction to Software Testing* (Introduction to Software Testing). Cambridge University Press, Dec. 2016, ISBN: 9781107172012. DOI: 10.1017/9781316771273.

[9] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering, Achieving Production Excellence*. O'Reilly Media, 2022, ISBN: 9781492076445.

[10] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, Nov. 1982, ISSN: 0010-4620. DOI: 10.1093/comjnl/25.4.465. eprint: https://academic.oup.com/comjnl/article-pdf/25/4/465/1045637/25-4-465.pdf.

[11] B. Ibryam and R. Huß, *Kubernetes patterns, Reusable elements for designing cloud native applications*, Second edition, R. Huß, Ed. O'Reilly Media, 2023, 266 pp., ISBN: 9781098131654.

[12] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, Jul. 2009, ISSN: 0360-0300. DOI: 10.1145/1541880.1541882.

[13] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21, Melbourne, Australia: IEEE Press, 2022, pp. 492–504, ISBN: 9781665403375. DOI: 10.1109/ASE51524.2021.9678773.

[14] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298. DOI: 10.1145/3133956.3134015.

[15] T. van Ede, H. Aghakhani, N. Spahn, *et al.*, "DeepCASE: Semi-Supervised Contextual Analysis of Security Events," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, IEEE, 2022. DOI: 10.1109/sp46214.2022.9833671.

[16] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 33–40. DOI: 10.1109/ICWS.2017.13.

[17] "Resource metrics pipeline," Kubernetes. (Aug. 31, 2024), [Online]. Available: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/ (visited on 04/28/2025).

[18] M. Müller, *Dynamic Time Warping*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 69–84, ISBN: 9783540740483. DOI: 10.1007/978-3-540-74048-3_4.

[19] R. H. Shumway and D. S. Stoffer, *Time Series Analysis and Its Applications: With R Examples* (Springer Texts in Statistics). Springer New York, 2006, ISBN: 9780387362762. DOI: 10.1007/978-3-031-70584-7.

[20] J. Zhu, S. He, J. Liu, *et al.*, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 121–130. DOI: 10.1109/ICSE-SEIP.2019.00021.

[21] D. Ohana. "Use open source drain3 log-template mining project to monitor for network outages," IBM Developer. (May 12, 2020), [Online]. Available: https://developer.ibm.com/blogs/how-mining-log-templates-can-help-ai-ops-in-cloud-scale-data-centers/ (visited on 07/25/2025).

Declaration of Authenticity

I hereby declare that any individual work submitted for assessment is entirely the product of my own effort,

- that I have correctly cited all text passages that do not originate from me/us, in accordance with standard academic citation rules[15], and that I have clearly mentioned all sources used;
- that I have declared in footnotes or in an index of auxiliary tools all aids used (AI assistance systems such as chatbots[16], translation[17], paraphrasing[18], or programming applications[19], and indicated their use at the corresponding text passages;
- that I have acquired all intangible rights to any materials I may have used, such as images or graphics, or that these materials were created by me/us;
- that the topic, the thesis or parts of it have not been used in an assessment of another module, unless this has been expressly agreed with the lecturer in advance and is stated as such;
- that I am aware that my work may be checked for plagiarism and for third-party authorship of human or technical origin (artificial intelligence);
- that I am aware that the FHNW School of Engineering will pursue a violation of this declaration of authenticity and that disciplinary consequences (reprimand or expulsion from the study program) may result from this.

Ostermundigen, 24 September 2025

Name: Mathias Petermann

Signature:

[15] e.g. APA oder IEEE
[16] e.g., ChatGPT
[17] e.g., DeepL
[18] e.g., Quillbot
[19] e.g., Github Copilot

Appendix

A Index of Auxiliary Tools

OpenAI ChatGPT (GPT-4o Model)

ChatGPT was used as an auxiliary tool during the preparation and writing of this thesis. Its use was limited to the following support tasks:

- **Language and Style Refinement:** Assisted in improving sentence structure, grammar, and academic tone throughout. Used for rewording technical passages, ensuring clarity, and maintaining consistent terminology.
- **LaTeX Formatting Assistance:** Provided guidance on LaTeX syntax for structuring tables, figures, lists, and document sections. Assistance included formatting multi-column tables, figure captions, and handling bibliographic references.
- **Content Structuring and Organization:** Supported the organization of chapters and subsections. ChatGPT was used to evaluate the logical coherence of section transitions and to suggest headings and outlines.
- **Research Support:** Aided in locating bibliographic references and summarizing academic papers. All references included in the final thesis were verified for correctness by the author.
- **Sparring Partner for Idea Development:** ChatGPT was also used as an interactive discussion partner to refine and challenge conceptual ideas. It proved to be a valuable tool for fostering reflective thinking and to articulate the thought process.

All outputs from ChatGPT were critically reviewed and manually adapted by the author. ChatGPT was not used to generate implementation code or unreviewed content.

GitHub Copilot (based on OpenAI Codex)

GitHub Copilot was used in the development of the Kubernetes Operator that forms the core implementation of this thesis. Its use was limited to the following supportive activities:

- **Code Completion and Suggestion:** Used to assist with writing boilerplate code and repetitive patterns, particularly when working with the Operator SDK, Custom Resource Definitions (CRDs), and Kubernetes API interactions in Go.
- **Syntax Assistance:** Provided suggestions for correct syntax when using Go programming constructs and Kubernetes client libraries. Copilot was particularly helpful in recalling function signatures and standard library usage.
- **Non-Automated Use and Manual Validation:** All code written with Copilot assistance was critically reviewed and tested by the author. Copilot was not used to generate complete functions or architectural decisions, but only for low-level coding support.

The author retained full control over the implementation, including the design, logic, and correctness of the software.

B Source Code Artifacts

The following Git repositories were used, and referenced in this thesis, each repository contains a tag which is relevant for the final submission.

Name	Description	URL	Tag
Orakel of Funk	This repository contains the operator developed for this project.	https://github.com/fhnw-imvs/fhnw-kubeseccontext	v0.6.3
	Container image	https://github.com/fhnw-imvs/fhnw-kubeseccontext/pkgs/container/fhnw-kubeseccontext	0.6.3
	Helm chart	https://github.com/fhnw-imvs/fhnw-kubeseccontext/pkgs/container/fhnw-kubeseccontext%2Forakel-of-funk	0.6.3
Python LogDrain	Proof of concept using Python Drain3 implementation	https://git.home.mpetermann.ch/peschmae/fhnw-ip6-log-drain-py	0.1.0
Go LogDrain	Proof of concept using Go Drain implementation	https://git.home.mpetermann.ch/peschmae/fhnw-ip6-log-drain-golang	0.1.0
Go resource collector	Proof of concept implementation for resource collection	https://git.home.mpetermann.ch/peschmae/fhnw-ip6-runtime-cmdline-poc/	0.1.0
Python resource comparison	Python based resource comparison for DTW and statistical summaries	https://git.home.mpetermann.ch/peschmae/fhnw-ip6-resource-comparison-poc/	0.1.0

C Orakel of Funk Project README

Orakel of Funk

Keep your Kubernetes workloads in tune — secure and functional.

WIP: This project is part of a Bachelor thesis at FHNW called *Automated Kubernetes Workload hardening using a Functionality Oracle*.

What is the Orakel of Funk?

The Orakel of Funk automates the testing and validation of Kubernetes workload hardening through controlled runtime experiments. It supports both namespace-wide and workload-specific hardening tests, leveraging Kubernetes-native `securityContext` fields such as `runAsNonRoot`, `readOnlyRootFilesystem`, `seccompProfile`.

When a user creates a `WorkloadHardeningCheck` or `NamespaceHardeningCheck` resource, the operator:

- Clones the target namespace (excluding network resources like Ingress).
- Gathers baseline logs and metrics over a configurable `recordingDuration`.
- Iteratively applies individual runtime restrictions to cloned workloads, isolating the impact of each `securityContext` attribute.
- Compares the behavior of each modified workload against the baseline, producing recommended `securityContext` configuration for each `WorkloadHardeningCheck`.
- In case of a `NamespaceHardeningCheck` the recommendations are propagted to the resources, and a final check run is performed where each workload is hardened according to its recommendations.

This approach allows developers and operators to assess the impact of runtime hardening on their workloads in a systematic, reproducible and automated manner, enabling safer adoption of security best practices without introducing disruptions.

Getting Started

To install the Operator in your cluster a Helm chart is provided, which deploys the operator and a ValKey instance to your cluster.

To generate the Certificates used by the webhooks, the helm chart relies on `cert-manager`, and assumes it is already present in the cluster. If you do not have `cert-manager` installed, check-out the cert-manager documentation for installation instructions.

Installation

The Helm chart is available from the Github package registry: Orakel of Funk Helm Chart

Since this is an OCI registry, you need to have at least Helm 3.8.0

To install the chart, you can use the following command:

```
helm install orakel-of-funk oci://ghcr.io/fhnw-imvs/fhnw-kubeseccontext/orakel-of-funk:0.1.0
```

As long as the repository and packages repository are private, you will need to use a personal access token with the `read:packages` scope to authenticate to the registry. You can do this by running the following command:

```
helm registry login ghcr.io -u <your-username>
```

Since the container image is also hosted on the same registry, you can use the same credentials to pull the image used by the operator, but need to create the ImagePullSecret manually, as the intention is to make both the chart and the image available to the public.

Just make sure to set the `imagePullSecrets` in the `global` section of the values file, or use the `--set global.imagePullSecrets[0].name=<your-secret-name>` flag when installing the chart.

Usage

To use the operator, you need to create a `WorkloadHardeningCheck` or `NamespaceHardeningCheck` resource.

WorkloadHardeningCheck

This resource targets a single workload (currently Deployment, StatefulSet or DaemonSet) and applies the hardening checks to it. The operator will clone the workload and apply the hardening checks to the cloned workload.

An example of a `WorkloadHardeningCheck` resource is shown below:

```yaml
apiVersion: checks.funk.fhnw.ch/v1alpha1
kind: WorkloadHardeningCheck
metadata:
  name: write-fs-emptydir
  namespace: oof-write-fs-emptydir
  labels:
    app.kubernetes.io/part-of: orakel-of-funk
    app.kubernetes.io/component: test
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: write-fs-emptydir
  recordingDuration: 1m
```

Since this resource is namsepace-scoped, it will assume that the target workload is in the same namespace as the `WorkloadHardeningCheck` resource. The operator will clone the workload and apply the hardening checks to the cloned workload.

NamespaceHardeningCheck

This resource targets a whole namespace and applies the hardening checks to all workloads in the namespace.

An example of a `NamespaceHardeningCheck` resource is shown below:

```yaml
apiVersion: checks.funk.fhnw.ch/v1alpha1
kind: NamespaceHardeningCheck
metadata:
  labels:
    app.kubernetes.io/name: orakel-of-funk
  name: podtato-namespace-hardening
spec:
  targetNamespace: podtato-kubectl
  recordingDuration: 1m
```

The resource is cluster-scoped, and will create `WorkloadHardeningCheck` resources for each workload in the `targetNamespace`.

Example Workloads

To test the operator, multiple example workloads are provided in the `examples` directory. These workloads can be used to test the operator and see how it works.

Podtato App

The [Podtato-Head](#) is a Demo App by the TAG App Delivery. It consists of multiple microservices, running in a single namespace.

This example contains a `NamespaceHardeningCheck` resource that targets the `podtato-kubectl` namespace.

MariaDB

The MariaDB example contains rendered manifests, based on the MariaDB Helm chart using the default `values.yaml`.

It contains a `WorkloadHardeningCheck` resource that targets the `mariadb` deployment in the `mariadb` namespace.

Nginx based workloads

There are multiple examples based on simple Nginx containers. They are intended to show how different configurations of a workload can affect the hardening checks and the recommendations made by the operator.

Custom Workloads

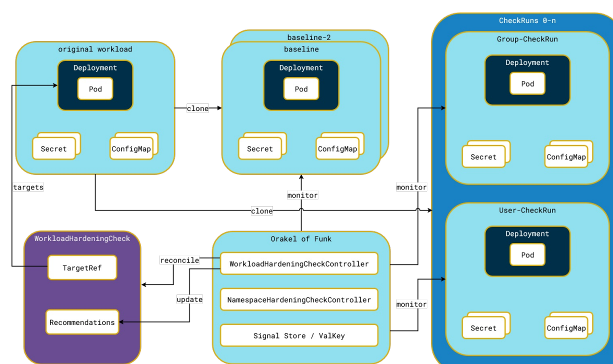
Last but not least, there are a few custom workloads, based on simple bash scripts to test specific scenarios/ `securityContext` configurations. These are intended to verify the operator's behavior in specific scenarios, such as:

- A workload that tries to write to the container filesystem.
- The same workload, but adapted to use an `emptyDir` volume.
- A workload trying to `chown` a file to a different user. This requires the `chown` capability to be set in the `securityContext`.
- A workload trying to escalate his privileges.

Development

This Operator is developed in Go, using the Operator SDK. The code is structured in a way that allows for easy testing and development.

Architecture



The operator is structured around the `Controller` pattern, where each controller is responsible for a specific resource type. The two main controllers are:

- `WorkloadHardeningCheckController`: Responsible for handling `WorkloadHardeningCheck` resources.
- `NamespaceHardeningCheckController`: Responsible for handling `NamespaceHardeningCheck` resources.

In addition both CRDs also have mutating and validating webhooks, which are used to validate the resources

and mutate them before they are created.

Webhooks

The mutating webhooks, only add a `suffix` to both the `WorkloadHardeningCheck` and `NamespaceHardeningCheck` resources, if none is provided. This suffix is used to create a unique name for the cloned namespaces, so that it does not conflict with the original workload. In case of the `NamespaceHardeningCheck` the suffix is also used to create extended suffixes for the `WorkloadHardeningCheck` resources.

The validating webhook for the `NamespaceHardeningCheck` ensures that the `targetNamespace` exists, but does not check if the namespace is empty.

The validating webhook for the `WorkloadHardeningCheck` ensures that the `targetRef` points to an existing workload, which is in the `ready` state. If you apply the `WorkloadHardeningCheck` at the same time as your workload, the webhook will not be able to find the workload, and will reject the request. In this case you need to wait until the workload is in the `ready` state and apply it again.

CheckRuns / Types

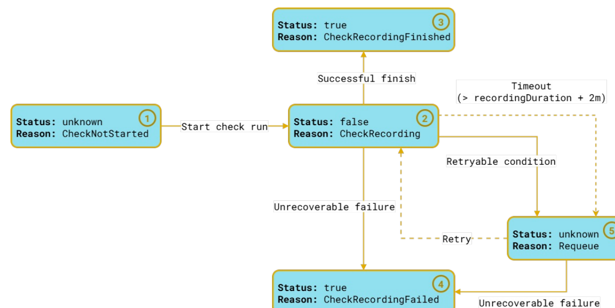
To run checks a `CheckRunner` is used, which is responsible for running the checks and collecting the results. The `CheckRunner` is implemented in `internal/runner/runners.go`.

The available checks are implemented based on the `CheckInterface` in `pkg/checks/check.go`, and new checks can be added by implementing this interface. They need to register themselves using the `RegisterCheck` function in `pkg/checks`.

The `CheckInterface`, which defines two methods:

1. `ShouldRun` to check if the `podSpec` does not yet define the necessary `securityContext` attributes
2. `GetSecurityContextDefaults` to apply the `securityContext` required for the specific check.

The `CheckRunner` will then run the checks and collect the results into a `recording.WorkloadRecording` struct, which is serialized and stored in ValKey. To track the progress of each check, the `CheckRunner` creates a `status.Condition` for each check, which is updated as the check progresses.



Recording signals

The `CheckRunner` uses a `Recorder` per signal. Those are implement in `internal/recording/` and explicitly called in the `CheckRunner`.

Some signals are only recorded if the pod reaches the `ready` state, while others are recorded regardless of the pod state. The `Recorder` is responsible for deciding which signals to record based on the pod state and the signal type.

Functionality Oracles

There are two independent oracles implemented in the operator, which are used to compare the behavior of the workloads:

1. `LogOrakel`: This oracle analyzes the logs of the recorded baselines and compares them to logs of the `checkRuns`. The oracle is based on the Drain3 algorithm as implemented by faceair/drain. Logs that are not present in the baseline, are considered anomalies, thus indicating a change in the workload's

behavior. If any anomalies are detected, the oracle will mark the check as failed, and store the detected anomalies in the `CheckRun` status.

2. `MetricsOrakel`: This oracle analyzes the metrics of the recorded baselines and compares them to metrics of the `checkRuns`. The oracle calculates the mean and standard deviation of the metrics and compares them to the baseline. If the metric is outside of the range of the baselines, it is considered a deviation and flagged on the `checkRun`.

The oracles are implement in `pkg/orakel/` from where they can be included into other golang projects as well.