November 28, 2025

A3: Operate and Monitor Kubernetes

1 Assignment

This assignment will focus on deploying your own application stack either to your self-provisioned Kubernetes cluster or a local Minikube installation.

Migrate from Docker Compose to Kubernetes: You have already containerized your project and orchestrated it in Docker compose. Now you will move on and migrate the application to Kubernetes and illustrate that you know how to make use of the different custom resources of Kubernetes. At the most basic level, you will introduce `Deployments`, `Services`, and an `Ingress` to deploy your application to a cluster. More advanced solution would take this further, use `Secrets` and `ConfigMaps`, and generally provide a bit of flexibility when it comes to installing the application to other Kubernetes clusters.

Please be aware that your manifests should not contain any sensitive information (e.g., SMTP passwords). Place them in a `Secret` that is automatically generated in a Helm chart. The chart should provide an illustrative placeholder and allow to exchange the values during installation. Excellent solutions will also find a way to use a shared VirtualBox folder to create shared storage across all *Pods*.

Helm Chart: Create *one* Helm chart for the complete deployment of this (and other) assignments. This Helm chart should grow over time and, at the end of the course, cover all the various aspects of your application that are deployed into the Kubernetes cluster, including future tasks on, for example, traffic management or continuous experimentation. You do not need to release the Helm chart, just store it as a folder in the `operations` repository.

The Helm chart should be the primary way to install all your project deliverables into *a* cluster (not necessarily your own). There should be no need for a detailed install description, however, minor things can always happen, so it is recommended to at least include a basic README that explains how to avoid issues that your peer reviewers have identified with your deployment. We will highly appreciate this, as it will also support our grading efforts!

It is important for you to draw a clear line between the provisioning of the server and the installation of your app. The expectation is that your Helm chart is independent of your provisioned server and that it is *not* installed during the automated provisioning. The chart should be installable into other clusters as well, for example, a Minikube cluster. You can assume though that the target cluster has Istio installed and that both a suitable *Ingress Controller* and an *Istio Gateway* have already been setup and are functional (relevant for Assignment 4).

The app is supposed to be manually deployed through a simple Helm install. The installation has several meaningful default values in a `values.xml` file that can be overridden to customize the installation. For example, to support the grading, it is possible to configure the hostnames during installation through which the stable version of the app is reached (and the pre-release version that is being experimented on in Assignment 4).

Enable Monitoring: Prometheus is a powerful solution to modern monitoring needs. We want you to use Prometheus to automatically pick-up your application metrics. Istio installations often come with their own Prometheus instance, but we do not want you to rely on this. Instead, install your own Prometheus instance through your Helm script and introduce `ServiceMonitor`s that bind your services to this instance.

You application must introduce at least three metrics in your that give insights into the *usability* of your application. Do not just report memory usage, but instrument existing interactions or add new ones that can create monitorable events. They need to allow you to reason about how your users are using the system. Include all the different metric types, i.e., *Gauge*, *Counter*, and *Histograms*, and break down the metrics with *Labels*.

Alerting: Excellent groups will configure ↗alerting capabilities *in Prometheus* (not in Grafana). They will configure the `AlertManager` and define at least one `PrometheusRule` that can warn developers through a channel of their choice (e.g., email). The rule does not need to be complex, for example, it would be enough to send an email alert when the service receives more than 15 requests per minute for two minutes straight.

> Caution
> Do not put passwords into your public files (source code, deployment files), instead, require a pre-defined `Secret`!

Grafana: Create a Grafana dashboard that shows *your custom metrics* (and a second one to support the decision process in Assignment 4). The Grafana UI allows you to import/export JSON files to share/create a dashboard. At the basic level, you will put your two dashboards as JSON files into the `operations` repository and explain in the `README.md` how they can be manually installed. Good groups will advance their dashboard beyond the basic introduction of metrics and use several different visualizations to illustrate deep understanding of Grafana: they use the specialized visualizations for *Gauges* and *Counters*, use timeframe selectors, and apply functions (e.g., `rate` or `avg`) to create non-standard plots.

It should not be required to manually import the dashboard. Custom dashboards can be automatically added during your webapp installation by providing a `ConfigMap` for the Grafana deployment. You can take inspiration from the Grafana templates that are automatically added by the →prometheus-operator chart.

> Info
> The goal of this assignment is to create a self-contained deployment of an app that can be installed to an existing cluster via the Helm script (which is not necessarily the cluster that you have provisioned yourself). We consider, for example, Prometheus, Grafana, and required `Ingress`es as part of the application, while for example an `IngressController` setup should already be provided by the cluster.

2 Submission

In this course, you will use a Git organization with several repository to collaborate within your team. This assignment will be assessed by checking your team repositories and the interactions on GitHub. To register your a submission, you must submit a *submission document* to the REMLA course on →Peer. The detailed assessment process is described in the *assessment overview* document that you can find on Brightspace.

> Info
> Please note that we will also check individual contributions to validate that all team members have passed the knock-out criteria of the course.

3 Assessment

This assignment needs to comply with the *Basic Requirements* as laid out in the *assessment design* document. Each of the following subsections presents a partial grade, which ranges from *Insufficient* (1) to *Excellent* (10), representing the weighted average of all corresponding rubric items. Grades can be affected by a *Pass/Fail* in the *Data Availability* (see Assignment 1). If information is unavailable or unclear, the respective grades will be *Insufficient*.

Positive rubric results are defined incrementally: Grades higher than level N can only be achieved when all items of level N have been achieved. Partial completions will be graded by considering the state of the lowest incomplete level. For example, a full completion of *Sufficient* and partial completions of *Good* will end up in the range between 6-8. Any contributions on the *Excellent* level are ignored, until *Good* has been fully completed. In contrast to positive results, we only explain by example which delivered state of a rubric item we will consider as *Insufficient* and *Poor*.

> Caution
> The assessment will not be performed on your cluster, but on a "known-working" cluster. Your helm package must be self-contained and no further setup steps are required to install your app into an Istio-enabled cluster.

3.1 Graded Requirements

Kubernetes Usage:

Insufficient The deployment does not contain a `Deployment` or a `Service` or deploying to Kubernetes fails.

Poor The deployment is successful, but can only be accessed through a node port or a service tunnel.

Sufficient
- The application can be deployed to a Kubernetes cluster.
- The relevant deployment files contain *at least* a working `Deployment` and a working `Service`.
- The app is accessed through an `Ingress`. The cluster provisioning creates a suitable `IngressController`.
- All aspects are installed through the central Helm chart.

Good
- The deployed application defines the location of the model service through an environment variable.
- The model service can be *relocated* just by changing the Kubernetes config.
  For example, changing the name of the service or the port on which it is running.
- The deployed application successfully uses a `ConfigMap` and a `Secret`.
  Define one to show that you know how, even though your application might not need one.
- The hostname/URL of the app can be changed through the `value.xml` of the Helm chart.
  To support grading, I want to be able to pick the URLs that I use to access the stable/pre-release versions.

Excellent
- All VMs mount the same shared VirtualBox folder as `/mnt/shared` into the VM.
- The deployed application mounts this path as a *hostPath* `Volume` into at least one `Deployment`.

App Monitoring:

Insufficient The app defines less than three metrics or Prometheus does not automatically collect the values.

Poor
- The app metrics are lacking an example for either `Gauge` or `Counter`.
- The app metrics are exposed through a framework (create the `/metrics` endpoints yourself).

Sufficient
- The app has 3+ *app-specific* and UI-related metrics for reasoning about user behavior or the perception of the app from the perspective of users.
- The metrics are automatically discovered and collected by Prometheus through `ServiceMonitor`.
- The content of the `/metrics` endpoint is created by yourself (not by a library).
- All aspects are installed through the central Helm chart.
- The `/metrics` endpoint of the `frontend-service` is reachable through the Ingress.
  This is a measure to support grading of setups with separate containers for the web UI and the API gateway.

Good
- The exposed metrics include a `Gauge`, a `Counter`, and an *app-specific* `Histogram` metric.
- Each metric type has at least one example, in which the metric is broken down with labels.

Excellent
- An `AlertManager` is configured with at least one non-trivial `PrometheusRule`.
- A corresponding `Alert` is raised in any type of channel (e.g., via email).
- The deployment files and the source code must not contain credentials (e.g., SMTP passwords).
  Instead, require pre-deployed `Secret`s and use them through environment variables.

Grafana:

Insufficient No dashboard is defined or it is trivial.

Poor A serious dashboard attempt exists, but it is incomplete or can only be imported with errors.

Sufficient
- A basic Grafana dashboard exists that illustrates all app-specific metrics.
- A second Grafana dashboard exists that supports the experimental decision of A4.
- Both dashboards are defined in a JSON file and can be manually imported in the Web UI.
- The operations repository contains a `README.md` that explains the manual installation (if required).

Good
- The basic dashboard contains at least two different visualizations that are not just time series.
  For example, a visual *Gauge* representation or a bar chart for *Histograms*.
- The basic dashboard employs variable timeframe selectors to parameterize the queries.
- The basic dashboard applies functions (like `rate` or `avg`) to enhance the plots.

Excellent Both Grafana dashboards are automatically installed by the central Helm chart.