

VDM Projektbericht

Tim Köhne

Module: Virtualisierung und Dienstarchitekturen (Master)
Dozent: Prof. Dr. Roman Majewski

12. Juli 2025

Zusammenfassung

In dieser Arbeit wird die Bereitstellung und der Betrieb eines Kubernetes-Clusters demonstriert, in dem sowohl eine zustandslose als auch eine zustandsorientierte Anwendung implementiert wurden. Ziel war es, die Unterschiede in Architektur, Skalierbarkeit und Wartbarkeit beider Anwendungstypen zu untersuchen. Als zustandslose Anwendung wurde ein modulares Web-Frontend mit Backend zur Bildbearbeitung realisiert, während für die zustandsorientierte Anwendung die Open-Source Publishing Plattform Ghost mit einer MariaDB Datenbank aufgesetzt wurde.

Der Bericht beschreibt den Aufbau des Clusters, die Konfiguration der einzelnen Komponenten, die Einbindung von Persistent Volumes über einen NFS-Server sowie die Anwendung von Kubernetes-Ressourcen wie Deployments, StatefulSets, Services und dem Horizontal Pod Autoscaler. Darüber hinaus werden die Herausforderungen und Grenzen bei der Durchführung von Updates, insbesondere bei zustandsorientierten Diensten, erläutert.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziel des Projekts	1
1.2	Vorgehensweise	1
2	Aufbau des Kubernetes-Clusters	1
2.1	Clusterarchitektur	1
2.2	Installation und Konfiguration des Clusters	2
3	Zustandslose Anwendung	4
3.1	Applikationsbeschreibung	4
3.2	Was bedeutet zustandslos?	6
3.3	Auto-Scaling	6
3.4	RBAC-Konzept	7
4	Zustandsorientierte Anwendung	8
4.1	Applikationsbeschreibung	8
4.2	Was bedeutet zustandsorientiert?	10
5	Updates der Anwendungen	10
5.1	Update der zustandslosen Anwendung	10
5.2	Update der zustandsorientierten Anwendung	12
6	Fazit	12
6.1	Reflexion	12
6.2	Herausforderungen	13
6.3	Weiterentwicklungspotenziale	13
	Quellenverzeichnis	14
A	Manifeste	15
A.1	Stateless Manifeste	15
A.2	Stateful Manifeste	22

1 Einleitung

1.1 Ziel des Projekts

Ziel dieses Projekts war es, ein funktionierendes Kubernetes-Cluster zu implementieren und darauf sowohl eine zustandslose als auch eine zustandsorientierte Anwendung bereitzustellen. Die Aufgabenstellung wurde im Rahmen des Moduls Virtualisierung und Dienstarchitekturen vorgegeben und diente dazu, ein praktisches Verständnis von Containerisierung, Clusterverwaltung sowie dem Betrieb von Cloud-nativen Anwendungen zu vermitteln.

Die Motivation liegt darin, zentrale Konzepte moderner IT-Infrastruktur, wie Skalierbarkeit, Ausfallsicherheit und Automatisierung, an einem realitätsnahen Beispiel zu erproben. Durch die Gegenüberstellung von zustandslosen und zustandsorientierten Anwendungen sollen Unterschiede im Betrieb, in der Skalierung und bei Updates herausgearbeitet werden.

1.2 Vorgehensweise

Nach dem initialen Setup der Cluster-Infrastruktur wurden die beiden Anwendungen schrittweise entwickelt bzw. konfiguriert. Die zustandslose Anwendung wurde als Eigenentwicklung umgesetzt, um vollständige Kontrolle über Architektur und Verhalten zu haben. Für die zustandsorientierte Anwendung wurde Ghost (Ghost Foundation 2025) als eine bestehende Open-Source Publishing-Plattform genutzt. Beide Anwendungen wurden in separaten Namespaces bereitgestellt und mit geeigneten Kubernetes-Ressourcen konfiguriert.

2 Aufbau des Kubernetes-Clusters

2.1 Clusterarchitektur

Als Grundlage für das gesamte Kubernetes-Cluster dienen drei Virtuelle Maschinen. Diese werden über VirtualBox (Oracle 2025) zur Verfügung gestellt. Dabei wird eine mit dem Hostnamen master bezeichnet, welche unter anderem die Kubernetes-Control-Plane zur Verfügung stellt. Die anderen beiden VMs werden worker1 und worker2 genannt und dienen dazu, die Anwendungen im Cluster auszuführen.

Außerdem wird das NAT-Netzwerk 10.0.2.0/24 aufgesetzt, welches die Kommunikation der VMs ermöglicht und diesen gleichzeitig Internetzugriff über den VM-Host zur Verfügung stellt. Es wird auch genutzt, um eine Portweiterleitung von den Ports 30080 und 30081 von einer der Workernodes auf den Host bereitzustellen. Dies ermöglicht später die Bereitstellung der Anwendungen des Clusters auf dem Host. Der Aufbau dieses Netzes ist in Abbildung 1 dargestellt.

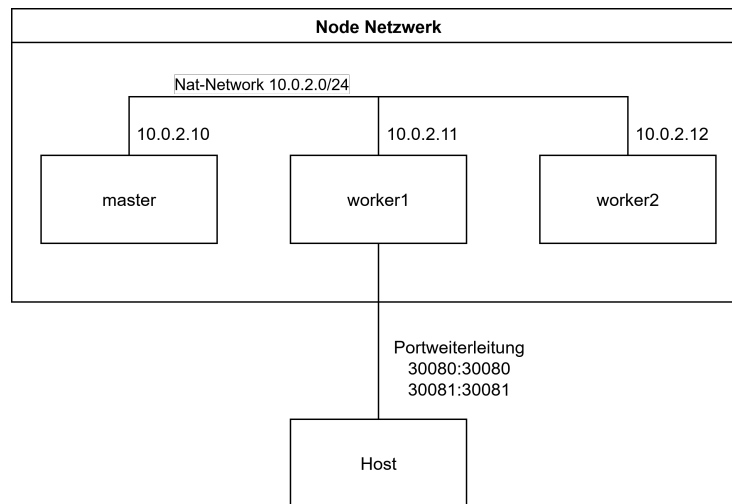


Abbildung 1: Übersicht des Netzwerks mit den drei VMs, dem Host und den eingerichteten Verbindungen inklusive NAT und Port-Weiterleitungen.

2.2 Installation und Konfiguration des Clusters

Die VMs basieren auf Ubuntu Server 24.04.2 LTS und sind jeweils mit zwei Prozessorkernen, zwei Gigabyte RAM und 25GB SSD-Speicher ausgestattet. Für den erfolgreichen Aufbau des Kubernetes-Clusters sind mehrere Installations- und Konfigurationsschritte notwendig. Einige betreffen alle VMs gleichermaßen, während andere spezifisch für die jeweilige Rolle im Cluster (Master oder Worker) sind.

Zunächst müssen auf allen VMs zentrale Softwarekomponenten installiert werden:

- Docker dient sowohl als Container-Runtime auf allen Nodes als auch auf der Master-Node als Basis für eine private Docker-Registry. Die Installation erfolgte gemäß der offiziellen Docker-Dokumentation für Ubuntu (Docker Inc. 2025).
- cri-dockerd fungiert als Schnittstelle zwischen Kubernetes und Docker, um Docker als Runtime nutzen zu können (Mirantis 2025).
- kubeadm, kubectrl und kubelet werden als die grundlegenden Kubernetes-Werkzeuge zur Cluster-Erstellung, Verwaltung und Container-Orchestrierung genutzt (The Linux Foundation 2025).
- nfs-common wird benötigt, um den Zugriff auf den NFS-Server für die zustandsorientierte Anwendung zu ermöglichen.

Zusätzlich erfolgen mehrere Konfigurationsschritte:

- Jedem Node wird per netplan eine statische IP-Adresse zugewiesen, außerdem werden die Hostnamen der Maschinen eindeutig festgelegt.

- Docker wird so konfiguriert, dass eine unsichere Verbindung zur lokalen Registry auf der Master-Node erlaubt ist.
- Swap wird auf allen VMs deaktiviert, da Kubernetes dies voraussetzt.
- Die Ports 30080 und 30081 werden mithilfe von ufw in der Ubuntu-Firewall freigeschaltet, um den externen Zugriff auf Anwendungen über NodePort-Services zu ermöglichen.

Auf der Master-Node werden darüber hinaus zusätzliche Dienste eingerichtet:

- Eine lokale Docker-Registry wird bereitgestellt, um selbst erstellte Images im Cluster verfügbar zu machen. Die Images werden auf dem Host mit `docker save` exportiert, über einen VirtualBox Shared Folder übertragen, mit `docker load` importiert, neu getaggt und in die Registry gepusht.
- Ein NFS-Server wird aufgesetzt, um die persistente Speicherung der zustandsorientierten Anwendung zu ermöglichen.

Nach Abschluss dieser Schritte wird das Cluster mithilfe von `kubeadm` initialisiert. Für die Kommunikation zwischen Pods wird das Pod-Netzwerk `192.168.0.0/16` mit Calico konfiguriert (Tigera Inc. 2025). Zusätzlich wird die Kubernetes Metrics Server API installiert (SIG Instrumentation 2025), um später die automatische Skalierung über den Horizontal Pod Autoscaler zu ermöglichen.

Nachdem das Cluster erfolgreich initialisiert wurde, wurden die beiden Worker-Nodes dem Cluster hinzugefügt. Der Aufbau des Pod-Netzwerks ist in Abbildung 2 dargestellt. Abbildung 3 zeigt die initialisierte Clusterstruktur mit einer Master-Node und zwei Worker-Nodes, welche als betriebsbereit angezeigt werden.

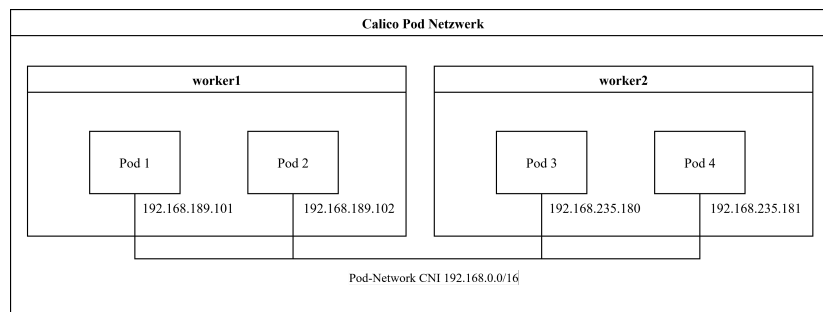


Abbildung 2: Pod-Netzwerk zur internen Kommunikation zwischen Kubernetes-Pods, exemplarisch mit vier Pods dargestellt.

```

tim@master:~/stateful$ kubectl get nodes
NAME       STATUS   ROLES    AGE   VERSION
master     Ready    control-plane  3d21h  v1.33.2
worker1    Ready    <none>      3d21h  v1.33.2
worker2    Ready    <none>      3d21h  v1.33.2
  
```

Abbildung 3: Finaler Clusteraufbau: Die Master-Node fungiert als Control-Plane, die beiden Worker-Nodes wurden erfolgreich eingebunden.

3 Zustandslose Anwendung

3.1 Applikationsbeschreibung

Für die zustandslose Anwendung wurde eine einfache, eigens entwickelte Webanwendung bereitgestellt, die das Hochladen und Bearbeiten von Bildern ermöglicht. Dazu wird ein Webserver eingesetzt, der verschiedene Bildbearbeitungsoperationen anbietet.

Als Frontend dient ein nginx-Server, der statische HTML-, CSS- und JavaScript-Dateien ausliefert und gleichzeitig als Reverse-Proxy fungiert, um Anfragen an das Backend weiterzuleiten. Das Backend besteht aus einem FastAPI-Server in Python, der eingehende Anfragen verarbeitet und je nach gewählter Bearbeitungsart an einen spezialisierten Bildbearbeitungsdienst weiterleitet.

Derzeit stehen zwei Bearbeitungsoptionen zur Verfügung, die jeweils durch eigene Microservices realisiert sind:

- Ein Go-basierter Dienst, der Bilder in Schwarz-Weiß umwandelt.
- Ein weiterer FastAPI-Service, der mithilfe der Python-Bibliothek rembg (Gatis 2025) den Hintergrund eines Bildes entfernt.

Im weiteren Verlauf werden diese Komponenten als Frontend, Backend, Image-Editor-BW und Image-Editor-Rembg bezeichnet. Einen schematischen Überblick über den Aufbau der Anwendung zeigt Abbildung 4. Abbildung 5 zeigt exemplarisch das Web-Frontend nach erfolgreicher Ausführung einer Schwarz-Weiß-Konvertierung (5a) sowie einer Hintergrundentfernung (5b).

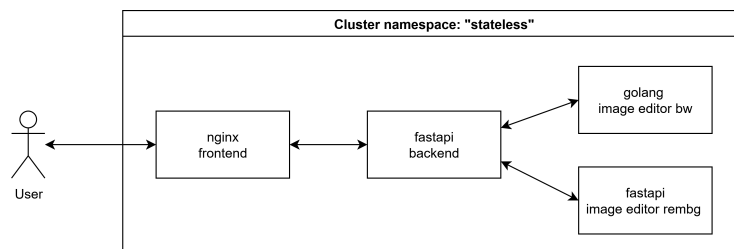


Abbildung 4: Architektur der zustandslosen Anwendung mit Frontend, Backend und zwei Bildbearbeitungsdiensten.

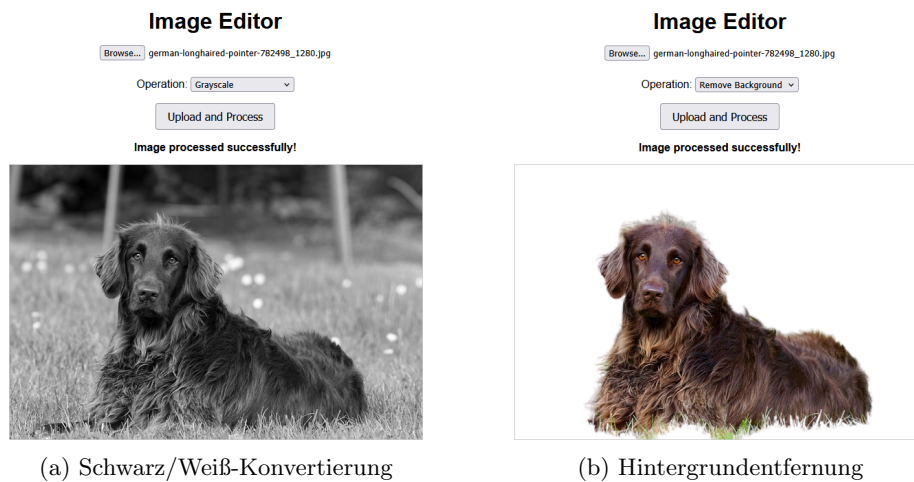


Abbildung 5: Darstellung der zustandslosen Bildbearbeitung im Webinterface. Bildquelle: (Pixabay 2015)

Insgesamt wurden vier eigene Docker-Images erstellt und in die lokale Registry der Master-Node übertragen, um sie im Kubernetes-Cluster bereitzustellen. Die gesamte Anwendung wurde im Namespace stateless ausgeführt.

Für jede Komponente wurde ein Deployment definiert, welches den jeweiligen Container verwaltet. Kubernetes erstellt daraus automatisch Pods und erlaubt eine Skalierung der Replika. Zusätzlich wurden für jede Komponente eigene Services eingerichtet, die den Komponenten eine interne Kommunikation ermöglichen. Dies ist insbesondere wichtig für:

- Die Verbindung des Frontends mit dem Backend,
- Die Kommunikation zwischen dem Backend und den Bildbearbeitungsdiensten,
- Die Lastverteilung bei mehreren Replika.

Der nginx-Service wurde als NodePort-Service konfiguriert, sodass die Anwendung über Port 30080 auch von außerhalb des Clusters erreichbar ist. Abbildung 6 zeigt die laufenden Komponenten der zustandslosen Anwendung im Kubernetes-Cluster. Alle definierten Deployments haben erfolgreich ihre zugehörigen Pods erstellt und in Betrieb genommen. Die Manifeste aller Komponenten der zustandslosen Anwendung sind im Anhang A.1 zu finden.

```
tim@master:~/stateful$ kubectl -n stateless get pods
NAME                                READY   STATUS    RESTARTS   AGE
backend-5b4f6fb6cf-5w4f2            1/1     Running   1 (168m ago)  3h31m
frontend-58b6446fd9-sctnc           1/1     Running   2 (168m ago)  3h31m
image-editor-bw-7779847cc9-zpn86    1/1     Running   1 (168m ago)  3h31m
image-editor-rembg-85b59b6758-c2p8q 1/1     Running   1 (168m ago)  5h10m
```

Abbildung 6: Laufende Pods der zustandslosen Anwendung im Kubernetes-Cluster nach erfolgreicher Ausführung der zugehörigen Deployments.

3.2 Was bedeutet zustandslos?

Bei der bereitgestellten Anwendung handelt es sich um eine zustandslose Applikation, da keine ihrer Komponenten anfrageübergreifende Zustände speichert oder verwaltet. Jede eingehende Anfrage wird unabhängig von vorherigen oder parallelen Anfragen verarbeitet. Dies ermöglicht eine nahezu unbegrenzte horizontale Skalierbarkeit, ob eine oder tausend Instanzen des Frontends existieren hat keinen Einfluss auf die Funktionsweise der Anwendung.

Zum Vergleich: Zustandsorientierte Anwendungen wie Datenbanken speichern Informationen dauerhaft. Eine Anfrage kann dort den Zustand für nachfolgende Anfragen verändern, etwa indem sie Daten hinzufügt oder löscht. Solche Anwendungen lassen sich nicht ohne Weiteres auf mehrere Instanzen verteilen, da alle Replikate synchronisiert sein müssten, um konsistente Daten bereitzustellen.

Die Komponenten der hier umgesetzten Anwendung wurden bewusst so entworfen, dass sie ohne persistente Speicherung arbeiten. Anfragen werden entgegengenommen, verarbeitet und weitergeleitet ohne Seiteneffekte oder dauerhafte Datenhaltung. Dadurch lassen sich problemlos mehrere Replika definieren und betreiben. In Kombination mit dem Horizontal Pod Autoscaler kann die Skalierung auch automatisiert anhand der aktuellen Systemauslastung erfolgen.

3.3 Auto-Scaling

Der Horizontal Pod Autoscaler (HPA) ermöglicht es, die Ressourcenlast einzelner Pods kontinuierlich zu überwachen und die Anzahl ihrer Replika dynamisch anzupassen. So kann eine Anwendung zunächst mit einem einzigen Replika starten und bei steigender Auslastung automatisch skaliert werden, um eine gleichmäßige Lastverteilung sicherzustellen.

Im Rahmen dieses Projekts wurde der HPA für alle Komponenten der zustandslosen Anwendung implementiert. Die Skalierung erfolgt basierend auf der jeweiligen CPU-Auslastung, wobei alle Komponenten unterschiedlich dimensionierte CPU-Anforderungen stellen. Die Replikationsanzahl kann dabei zwischen einem und fünf Pods variieren.

Die Funktionsweise des HPA wird in Abbildung 7 am Beispiel der Komponente Image-Editor-Rembg veranschaulicht. Das zugehörige Deployment-Manifest definiert eine CPU-Anforderung von 1000m (entspricht einem vollen CPU-Kern) pro Pod. Sobald die CPU-Auslastung 80% dieser Kapazität überschreitet, greift der HPA ein.

Im Ruhezustand zeigt (7a) eine geringe CPU-Auslastung. Steigt die Last durch parallele Anfragen, erhöht sich die Auslastung deutlich, wie in (7b) zu sehen ist. Der HPA reagiert darauf mit der Skalierung: In (7c) wurde ein zusätzliches Replikat erzeugt, um die Last zu verteilen.


```
tim@master:~/stateless$ kubectl -n stateless top pods
NAME                                CPU(cores)   MEMORY(bytes)
backend-5b4f6fb6cf-x2x8w            2m           50Mi
frontend-58b6446fd9-jqgsd          1m           2Mi
image-editor-bw-7779847cc9-k426t    1m           5Mi
image-editor-rembg-85b59b6758-vgc4r 2m           872Mi
```

(a) Der Pod befindet sich im Idle-Zustand.

```
tim@master:~/stateless$ kubectl -n stateless top pods
NAME                                CPU(cores)   MEMORY(bytes)
backend-5b4f6fb6cf-x2x8w            2m           48Mi
frontend-58b6446fd9-jqgsd          1m           3Mi
image-editor-bw-7779847cc9-k426t    1m           4Mi
image-editor-rembg-85b59b6758-vgc4r 949m         1676Mi
```

(b) Hohe Last durch viele parallele Anfragen.

```
tim@master:~/stateless$ kubectl -n stateless top pods
NAME                                CPU(cores)   MEMORY(bytes)
backend-5b4f6fb6cf-x2x8w            2m           49Mi
frontend-58b6446fd9-jqgsd          1m           2Mi
image-editor-bw-7779847cc9-k426t    1m           5Mi
image-editor-rembg-85b59b6758-9m854 726m         172Mi
image-editor-rembg-85b59b6758-vgc4r 271m         799Mi
```

(c) Der HPA skaliert das Deployment automatisch und erstellt ein weiteres Replikat.

Abbildung 7: Automatische Skalierung mit dem HPA am Beispiel der Komponente Image-Editor-Rembg. Nach Überschreiten des festgelegten Schwellenwerts wird ein zusätzliches Replikat erzeugt, um die Last zu verteilen.

3.4 RBAC-Konzept

Das Role-Based Access Control (RBAC) in Kubernetes ermöglicht eine fein granulare Zugriffskontrolle innerhalb eines Clusters. Dabei werden Rollen definiert, die bestimmten Ressourcen und Aktionen innerhalb eines Namespaces bestimmte Berechtigungen zuweisen. Über sogenannte RoleBindings können diese Rollen anschließend einzelnen Nutzern oder Nutzergruppen zugeordnet werden. So lassen sich beispielsweise vollständige Zugriffsrechte für Entwickler definieren, die aktiv Änderungen vornehmen, während gleichzeitig restriktive Berechtigungen für Rollen wie Qualitätssicherung oder sogar Praktikanten vergeben werden können, welche lediglich Einblick in die Umgebung benötigen, ohne selbst Veränderungen vornehmen zu müssen.

Im Rahmen dieses Projekts wurden genau diese beiden Rollen definiert, der Developer und der Viewer.

- Der Developer besitzt umfassende Zugriffsrechte auf Ressourcen wie Pods, Services, ConfigMaps, Deployments und CronJobs. Der Zugriff auf Secrets ist jedoch explizit ausgeschlossen und ausschließlich dem Kubernetes-Administrator vorbehalten.
- Der Viewer verfügt über die gleichen Ressourcenrechte wie der Developer, jedoch ausschließlich mit Leseberechtigung. Diese Rolle eignet sich insbesondere für Benutzer, die das System einsehen, aber nicht verändern dürfen.

Abbildung 8 veranschaulicht die Auswirkungen der RBAC-Konfiguration. Es wird geprüft, ob ein Nutzer jeweils die Berechtigung besitzt, Pods zu lesen, Deployments zu löschen und Secrets einzusehen. Während der Viewer ausschließlich Lesezugriff auf Pods erhält, kann der Developer zusätzlich Deployments löschen. Nur der Administrator ist berechtigt Secrets einzusehen.

```
tim@master:~$ kubectl config use-context viewer-context
Switched to context "viewer-context".
tim@master:~$ kubectl auth can-i get pods
yes
tim@master:~$ kubectl auth can-i delete deployments
no
tim@master:~$ kubectl auth can-i get secrets
no
tim@master:~$ kubectl config use-context developer-context
Switched to context "developer-context".
tim@master:~$ kubectl auth can-i get pods
yes
tim@master:~$ kubectl auth can-i delete deployments
yes
tim@master:~$ kubectl auth can-i get secrets
no
tim@master:~$ kubectl config use-context kubernetes-admin@kubernetes
Switched to context "kubernetes-admin@kubernetes".
tim@master:~$ kubectl auth can-i get pods
yes
tim@master:~$ kubectl auth can-i delete deployments
yes
tim@master:~$ kubectl auth can-i get secrets
yes
```

Abbildung 8: Rollenbasierte Zugriffskontrolle in der zustandslosen Anwendung: Zugriffsrechte für Viewer, Developer und Administrator im Vergleich.

Die Umsetzung erfolgte über SSL-Zertifikate für zwei Benutzer in unterschiedlichen Gruppen, die Zertifikate wurden von der Certificate Authority des Kubernetes-Clusters signiert. Die Vergabe der Zugriffsrechte erfolgte über RoleBindings, durch die den jeweiligen Gruppen passende Rollen zugewiesen wurden. Anschließend wurde die kubectl-Konfiguration so angepasst, dass zwischen den definierten Benutzerkontexten innerhalb des Clusters gewechselt werden kann.

4 Zustandsorientierte Anwendung

4.1 Applikationsbeschreibung

Als zustandsorientierte Anwendung wurde die Open-Source Publishing-Plattform Ghost (Ghost Foundation 2025) eingesetzt. Ghost stellt ein Content-Management-System bereit, welches das Erstellen und Veröffentlichen von Blogs und Online-Publikationen ermöglicht. Dabei ähnelt es Plattformen wie WordPress, legt jedoch besonderen Wert auf Performance und Benutzerfreundlichkeit. Als Datenbank kommt MariaDB zum Einsatz, welche die Daten dauerhaft über einen NFS-Server speichert. Zusätzlich wird ein Nginx-Server als Reverse-Proxy verwendet, dieser leitet Nutzeranfragen an Ghost weiter und dient gleichzeitig als Cache für statische Inhalte.

Sämtliche Bestandteile liegen als fertige Docker-Images vor, sodass ein eigener Build nicht erforderlich war. Als Grundlage für die Implementierung wurde ein

von Ghost bereitgestelltes Beispiel (Docker Library 2025) herangezogen, welches die Bereitstellung von Ghost und MySQL mittels Docker-Compose veranschaulicht. Abbildung 9 veranschaulicht den groben Aufbau der gesamten Anwendung sowie die Kommunikation zwischen den einzelnen Komponenten.

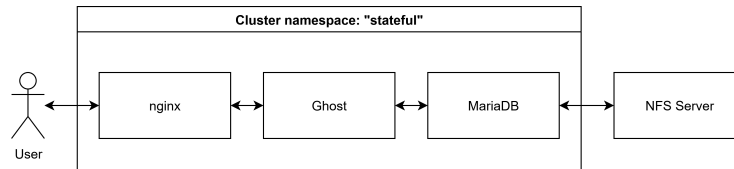


Abbildung 9: Architektur der zustandsorientierten Anwendung

In Abbildung 10 ist das Web-Frontend der Anwendung zu sehen. Es handelt sich um das standardmäßige Design von Ghost mit etwas eigener Konfiguration, etwa zur Festlegung des Plattformnamens. Zwei Blog-Posts sind sichtbar, einer wurde vor und einer nach einem Neustart aller Anwendungskomponenten erstellt, um die persistente Speicherung der Inhalte darzustellen.

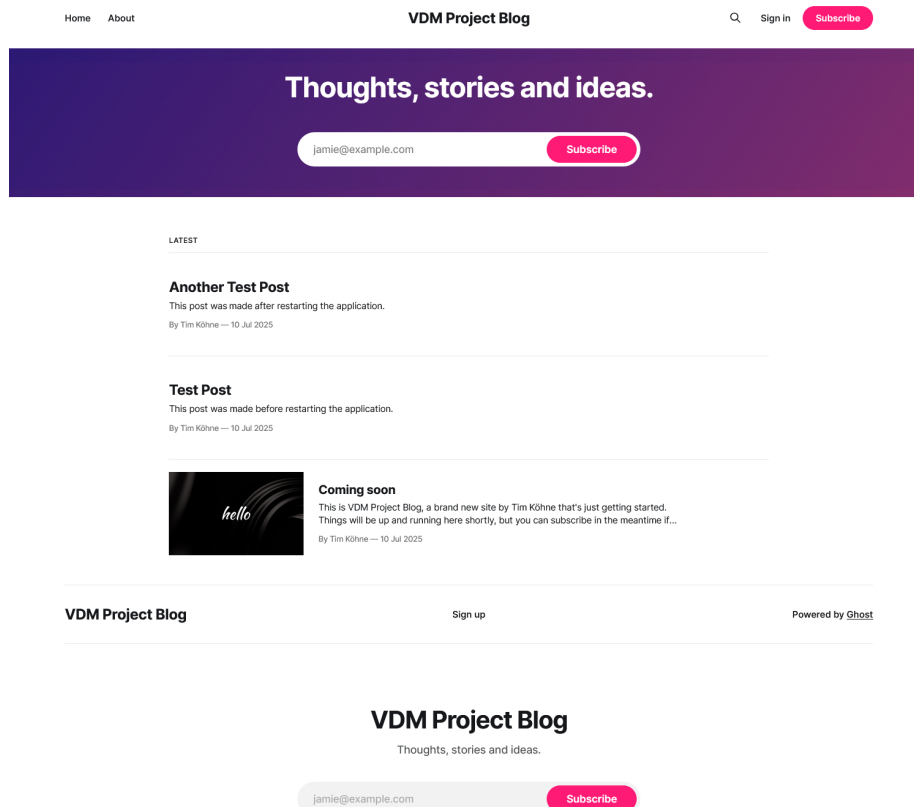


Abbildung 10: Frontend der zustandsorientierten Anwendung mit Blogposts.

Für die Bereitstellung der Anwendung im Kubernetes-Cluster wurden entspre-

chende Ressourcen-Definitionen in Form von Manifesten erstellt. Nginx und Ghost wurden jeweils als Deployments konfiguriert, während die Datenbank MariaDB mithilfe eines StatefulSet implementiert wurde. Dies erlaubt es der Datenbank eine feste Identität zuzuweisen und persistente Speicherressourcen zu binden. Der Speicherzugriff erfolgt über ein PersistentVolumeClaim, das einem PersistentVolume zugeordnet ist. Dieses wiederum nutzt einen NFS-Server der auf der Master-Node betrieben wird, um die Daten dauerhaft zu speichern. Für MariaDB und Ghost wurden jeweils interne Services konfiguriert, um die Kommunikation zwischen den Komponenten innerhalb des Clusters zu ermöglichen. Zusätzlich wurde ein NodePort-Service für Nginx eingerichtet, über den die gesamte Anwendung extern über Port 30081 erreichbar ist. Alle Ressourcen dieser Anwendung wurden im Namespace `stateful` eingerichtet. Die Manifeste aller Komponenten der zustandsorientierten Anwendung sind im Anhang A.2 zu finden.

Abbildung 11 zeigt die aktuell laufenden Pods von Nginx, Ghost und MariaDB im Kubernetes-Cluster.

```
tim@master:~/stateful$ kubectl -n stateful get pods
```

NAME	READY	STATUS	RESTARTS	AGE
ghost-7478546578-55v59	1/1	Running	1 (7m30s ago)	7m33s
mariadb-0	1/1	Running	0	7m33s
redis-698496554f-d7trs	1/1	Running	0	7m33s

Abbildung 11: Laufende Pods der zustandsorientierten Anwendung

4.2 Was bedeutet zustandsorientiert?

Im Gegensatz zur zustandslosen Architektur handelt es sich bei dieser Anwendung um eine zustandsorientierte Applikation. Nutzeranfragen können sich gegenseitig beeinflussen, da Inhalte dauerhaft gespeichert und für andere Benutzer zugänglich gemacht werden müssen. Ein veröffentlichter Blog-Post soll beispielsweise auch bei späteren Seitenaufrufen angezeigt werden. Damit greifen alle Anfragen auf dieselbe zugrunde liegende Datenbank zu, welche wiederum ihre Daten auf einem persistenten Speicher ablegt. Diese Abhängigkeit vom gespeicherten Zustand erschwert die horizontale Skalierbarkeit, insbesondere im Datenbankbereich, da nicht beliebig viele unabhängige Instanzen betrieben werden können, ohne die Datenkonsistenz zu beeinflussen. Es wird also nur eine Instanz der Datenbank genutzt, welche alle Anfragen bearbeiten muss. Dies könnte Probleme bereiten wenn die Anwendung für viele Nutzer angeboten wird. In dem Fall sollte darüber nachgedacht werden, die Datenbank durch eine Lösung zu ersetzen, die verteilten Betrieb unterstützt und auf horizontale Skalierung ausgelegt ist, beispielsweise durch den Einsatz von Datenbanken, die Replikation, Partitionierung oder verteilte Konsistenzmechanismen unterstützen.

5 Updates der Anwendungen

5.1 Update der zustandslosen Anwendung

Nach der Bereitstellung der zustandslosen Anwendung zeigte sich, dass die vorhandenen Container keine sinnvolle Unterstützung für Kubernetes Zustand-

sprüfung mit `readinessProbes` und `livenessProbes` hatten. Um dies zu verbessern, wurde ein Update der Anwendung durchgeführt. Hierzu wurden die Kubernetes-Manifeste entsprechend erweitert, sodass regelmäßige Statusabfragen an die einzelnen Komponenten gesendet werden können. Gleichzeitig mussten die Container-Images angepasst werden, um geeignete Schnittstellen für diese Prüfungen bereitzustellen.

Beispielsweise wurde im FastAPI-Backend eine neue HTTP-Schnittstelle unter `/healthz` implementiert, die im funktionsfähigen Zustand stets mit `{"status": "ok"}` antwortet. Ähnliche Schnittstellen wurden ebenfalls in den Images des Frontends sowie der beiden Bildbearbeitungskomponenten ergänzt.

Die Durchführung des Updates gestaltet sich bei zustandslosen Anwendungen unkompliziert, da Kubernetes durch das Deployment-Objekt bereits standardmäßig eine automatische Rollout-Strategie bietet. Zudem bestehen keine Abhängigkeiten zu bestimmten Instanzen anderer Komponenten, was die Flexibilität bei Änderungen erhöht.

Das Update wird durch Ausführen von `kubectl apply` auf die angepassten Manifeste eingeleitet. Die genaue Vorgehensweise beim Rollout kann dabei innerhalb der Manifeste definiert werden, in diesem Fall wurde jedoch auf die Standardkonfiguration zurückgegriffen, da diese den Anforderungen genügt. Kubernetes führt daraufhin ein Rolling Update durch. Bestehende Pods mit dem alten Image werden schrittweise durch neue Pods ersetzt, wobei stets eine Mindestanzahl lauffähiger Instanzen aufrechterhalten bleibt. Falls nur ein einzelnes Replika existiert, wird vor dem Austausch automatisch ein weiteres erstellt, um die Verfügbarkeit der Anwendung sicherzustellen.

Abbildung 12 zeigt den Ablauf des Updates, es ist erkennbar, dass alle Komponenten nacheinander aktualisiert werden und die Rolling-Update-Strategie konsequent eingehalten wird.

```

minimaster: /stateless$ kubectl apply -f frontend.yaml; kubectl -n stateless rollout status deployment/frontend
deployment.apps/frontend configured
service/frontend unchanged
horizontalpodautoscaler.autoscaling/frontend unchanged
Waiting for deployment "frontend" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "frontend" rollout to finish: 1 old replicas are pending termination...
deployment "frontend" successfully rolled out
minimaster: /stateless$ kubectl apply -f backend.yaml; kubectl -n stateless rollout status deployment/backend
deployment.apps/backend configured
service/backend unchanged
horizontalpodautoscaler.autoscaling/backend-hpa unchanged
Waiting for deployment "backend" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "backend" rollout to finish: 1 old replicas are pending termination...
deployment "backend" successfully rolled out
minimaster: /stateless$ kubectl apply -f image-editor-bw.yaml; kubectl -n stateless rollout status deployment/image-editor-bw
deployment.apps/image-editor-bw configured
service/image-editor-bw unchanged
horizontalpodautoscaler.autoscaling/image-editor-bw-hpa unchanged
Waiting for deployment "image-editor-bw" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "image-editor-bw" rollout to finish: 1 old replicas are pending termination...
deployment "image-editor-bw" successfully rolled out
minimaster: /stateless$ kubectl apply -f image-editor-rembg.yaml; kubectl -n stateless rollout status deployment/image-editor-rembg
deployment.apps/image-editor-rembg configured
service/image-editor-rembg unchanged
horizontalpodautoscaler.autoscaling/image-editor-rembg-hpa unchanged
Waiting for deployment "image-editor-rembg" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment spec update to be observed...
Waiting for deployment "image-editor-rembg" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "image-editor-rembg" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "image-editor-rembg" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "image-editor-rembg" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "image-editor-rembg" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "image-editor-rembg" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "image-editor-rembg" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "image-editor-rembg" rollout to finish: 1 old replicas are pending termination...
deployment "image-editor-rembg" successfully rolled out
minimaster: /stateless$

```

Abbildung 12: Durchführung des Updates der zustandslosen Anwendung. Es werden alle Komponenten nacheinander aktualisiert. Dabei wird jeweils die Rolling-Update Strategie verwendet, um die Anwendung durchgehend verfügbar zu halten.

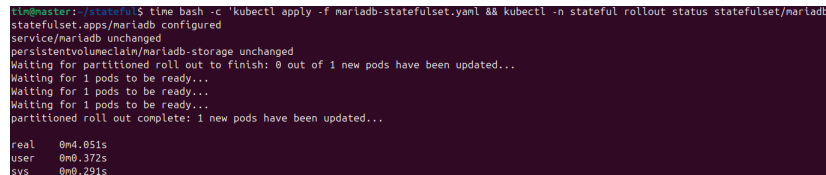
5.2 Update der zustandsorientierten Anwendung

Beim Update der zustandsorientierten Anwendung lag der Fokus auf der Datenbank, da sie den relevanten Unterschied zur zustandslosen Architektur zeigt. In diesem Fall wurde ein Versionswechsel der MariaDB-Version von 10.11 auf 11.8 durchgeführt.

Die Datenbank wird als StatefulSet betrieben, da in diesem Szenario eine einzigartige und persistente Identität erforderlich ist. StatefulSets unterstützen grundsätzlich Rolling Updates und auch die Skalierung auf mehrere Replika, der Einsatz dieser Mechanismen setzt jedoch voraus, dass die zugrunde liegende Anwendung, selbst für den verteilten Betrieb ausgelegt ist und beispielsweise Replikation, Konsistenzmechanismen oder Leader-Election unterstützt. Ein unterbrechungsfreies Update ist daher nur möglich, wenn eine verteilte Datenbank eingesetzt wird, die für genau solche Szenarien konzipiert ist.

MariaDB bietet diese Funktionalität nicht standardmäßig. Das unterstützte Master-Slave-Modell reicht nicht aus, da Schreibzugriffe ausschließlich auf dem Master erfolgen können. Bei einem Update dieser zentralen Instanz wäre somit die Verfügbarkeit weiterhin nicht vollständig gewährleistet.

Das durchgeführte Update konnte daher nur unter kurzfristiger Unterbrechung der Verfügbarkeit erfolgen, Abbildung 13 zeigt den Ablauf des Rollouts. Während des Updates war die Datenbank für etwa vier Sekunden nicht erreichbar. In hochverfügbaren Umgebungen oder bei kritischen Anwendungen wäre eine solche Unterbrechung problematisch und müsste durch den Einsatz verteilter Datenbanksysteme vermieden werden. Im hier betrachteten Fall, einer CMS-Plattform, ist eine kurze Unterbrechung in der Regel akzeptabel, zumal Nginx und Ghost als vorgeschaltete Komponenten dienen und kurzfristige Ausfälle der Datenbank weitgehend abschirmen.



```
timemaster@timemaster:~$ time bash -c 'kubectl apply -f mariadb-statefulset.yaml && kubectl -n stateful rollout status statefulset/mariadb'
statefulset.apps/mariadb configured
service/mariadb unchanged
persistentvolumeclaim/mariadb-storage unchanged
Waiting for partitioned roll out to finish: 0 out of 1 new pods have been updated...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
partitioned roll out complete: 1 new pods have been updated...

real    0m4.051s
user    0m0.372s
sys     0m0.291s
```

Abbildung 13: Update-Rollout von MariaDB bei der zustandsorientierten Anwendung.

6 Fazit

6.1 Reflexion

Im Rahmen dieses Projekts wurde erfolgreich ein Kubernetes-Cluster aufgebaut und sowohl eine zustandslose als auch eine zustandsorientierte Anwendung bereitgestellt. Dabei konnten die grundlegenden Unterschiede in Bezug auf Architektur, Skalierbarkeit und Wartbarkeit praktisch nachvollzogen werden. Besonders deutlich wurde, wie stark sich der Charakter einer Anwendung auf deren Betriebs- und Updatefähigkeit innerhalb eines Clusters auswirkt.

Die zustandslose Anwendung ließ sich mit vergleichsweise geringem Aufwand skalieren, aktualisieren und überwachen. Funktionen wie der Horizontal Pod

Autoscaler oder Rolling Updates konnten hier direkt und ohne Zusatzkonfiguration genutzt werden. Die zustandsorientierte Anwendung hingegen erforderte deutlich mehr Planung, insbesondere im Hinblick auf Persistenz, Netzwerkzugriff und Updateverfahren.

6.2 Herausforderungen

Zu den größten Herausforderungen zählten die persistente Speicherung von Zustandsdaten sowie das Update der Datenbank innerhalb eines StatefulSet. Der Betrieb einer einzelnen MariaDB-Instanz ohne verteilte Replikation machte deutlich, dass klassische Datenbanken ihre Grenzen haben, wenn es um hochverfügbare, unterbrechungsfreie Updates geht.

Auch die initiale Konfiguration von RBAC, Services und Volumes erforderte ein gutes Verständnis der Kubernetes-Ressourcen und ihrer Zusammenspiele. Dennoch konnten alle Anforderungen gelöst werden.

6.3 Weiterentwicklungspotenziale

Ein mögliches Zukunftsszenario besteht in der Ablösung der einzelnen MariaDB-Instanz durch ein verteiltes Datenbanksystem wie Vitess, CockroachDB oder einem Galera Cluster, um echte Hochverfügbarkeit und ausfallsichere Updates zu ermöglichen.

Auch das Thema Observability ließe sich ausbauen, etwa durch den Einsatz von Prometheus, Grafana oder OpenTelemetry, um Metriken, Logs und Traces zentral zu erfassen und auszuwerten. So könnte die Verfügbarkeit von Diensten über längere Zeiträume analysiert werden.

Insgesamt zeigt das Projekt, wie leistungsfähig und flexibel Kubernetes im Umgang mit unterschiedlichen Anwendungstypen ist, vorausgesetzt, die eingesetzten Komponenten sind entsprechend gestaltet und integriert.

Quellenverzeichnis

Docker Inc. 2025

DOCKER INC.: *Ubuntu* — *docs.docker.com*. <https://docs.docker.com/engine/install/ubuntu/>, 2025. — [Accessed 07-07-2025]

Docker Library 2025

DOCKER LIBRARY: *Dockerhub Ghost*. https://hub.docker.com/_/ghost/, 2025. — [Accessed 11-07-2025]

Gatis 2025

GATIS, Daniel: *GitHub - danielgatis/rembg: Rembg is a tool to remove images background* — *github.com*. <https://github.com/danielgatis/rembg>, 2025. — [Accessed 11-07-2025]

Ghost Foundation 2025

GHOST FOUNDATION: *Ghost: The #1 open source headless Node.js CMS* — *ghost.org*. <https://ghost.org/docs/>, 2025. — [Accessed 11-07-2025]

Mirantis 2025

MIRANTIS: *Install* — *mirantis.github.io*. <https://mirantis.github.io/cri-dockerd/usage/install/>, 2025. — [Accessed 09-07-2025]

Oracle 2025

ORACLE: *Oracle VirtualBox* — *virtualbox.org*. <https://www.virtualbox.org/>, 2025. — [Accessed 11-07-2025]

Pixabay 2015

PIXABAY: *German Longhaired Pointer Dog Pet - Free photo on Pixabay* — *pixabay.com*. <https://pixabay.com/photos/german-longhaired-pointer-dog-pet-782498/>, 2015. — [Accessed 10-07-2025]

SIG Instrumentation 2025

SIG INSTRUMENTATION: *GitHub - kubernetes-sigs/metrics-server: Scalable and efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines.* — *github.com*. <https://github.com/kubernetes-sigs/metrics-server/?tab=readme-ov-file>, 2025. — [Accessed 09-07-2025]

The Linux Foundation 2025

THE LINUX FOUNDATION: *Installing kubeadm* — *kubernetes.io*. <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/>, 2025. — [Accessed 09-07-2025]

Tigera Inc. 2025

TIGERA INC.: *Calico quickstart guide* — *Calico Documentation* — *docs.tigera.io*. <https://docs.tigera.io/calico/latest/getting-started/kubernetes/quickstart>, 2025. — [Accessed 07-07-2025]

Alle relevanten Dateien sind auch verfügbar unter: <https://github.com/timkoehne/VDM-Kubernetes-Setup>

A Manifeste

A.1 Stateless Manifeste

backend.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: stateless
  name: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: 10.0.2.10:5000/imageprocessing-backend:vdm5
          ports:
            - containerPort: 8005
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "500m"
              memory: "256Mi"
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8005
            initialDelaySeconds: 5
            periodSeconds: 10
            failureThreshold: 3
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8005
            initialDelaySeconds: 10
            periodSeconds: 15
            failureThreshold: 3
```

```
---
apiVersion: v1
kind: Service
metadata:
  namespace: stateless
  labels:
    app: backend
  name: backend
spec:
  ports:
    - port: 8005
      targetPort: 8005
  selector:
    app: backend
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
  namespace: stateless
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 80
```

frontend.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: stateless
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
```

```
    app: frontend
spec:
  containers:
  - name: frontend
    image: 10.0.2.10:5000/imageprocessing-frontend:vdm6
    ports:
    - containerPort: 80
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "500m"
        memory: "256Mi"
    readinessProbe:
      httpGet:
        path: /healthz
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
      failureThreshold: 3
    livenessProbe:
      httpGet:
        path: /healthz
        port: 80
      initialDelaySeconds: 10
      periodSeconds: 20
      failureThreshold: 3
---
apiVersion: v1
kind: Service
metadata:
  namespace: stateless
  name: frontend
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30080
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: frontend
```

```
namespace: stateless
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: frontend
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
```

image-editor-bw.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: stateless
  name: image-editor-bw
spec:
  replicas: 1
  selector:
    matchLabels:
      app: image-editor-bw
  template:
    metadata:
      labels:
        app: image-editor-bw
    spec:
      containers:
      - name: image-editor-bw
        image: 10.0.2.10:5000/imageprocessing-image-editor-bw:vdm3
        ports:
        - containerPort: 8000
        resources:
          requests:
            cpu: "100m"
            memory: "128Mi"
          limits:
            cpu: "500m"
            memory: "256Mi"
        readinessProbe:
          httpGet:
            path: /healthz
            port: 8000
          initialDelaySeconds: 5
```

```
        periodSeconds: 10
        failureThreshold: 3
    livenessProbe:
        httpGet:
            path: /healthz
            port: 8000
        initialDelaySeconds: 10
        periodSeconds: 15
        failureThreshold: 3
---
apiVersion: v1
kind: Service
metadata:
    namespace: stateless
    name: image-editor-bw
spec:
    selector:
        app: image-editor-bw
    ports:
        - protocol: TCP
          port: 8000
          targetPort: 8000
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
    name: image-editor-bw-hpa
    namespace: stateless
spec:
    scaleTargetRef:
        apiVersion: apps/v1
        kind: Deployment
        name: image-editor-bw
    minReplicas: 1
    maxReplicas: 5
    metrics:
        - type: Resource
          resource:
            name: cpu
            target:
                type: Utilization
                averageUtilization: 80
```

image-editor-rembg.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
    namespace: stateless
    name: image-editor-rembg
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: image-editor-rembg
  template:
    metadata:
      labels:
        app: image-editor-rembg
    spec:
      containers:
        - name: image-editor-rembg
          image:
            ↪ 10.0.2.10:5000/imageprocessing-image-editor-rembg:vdm4
          ports:
            - containerPort: 8000
          resources:
            requests:
              cpu: "1000m"
            limits:
              cpu: "1500m"
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8000
            initialDelaySeconds: 60
            periodSeconds: 10
            failureThreshold: 3
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8000
            initialDelaySeconds: 70
            periodSeconds: 15
            failureThreshold: 3
      ---
apiVersion: v1
kind: Service
metadata:
  namespace: stateless
  name: image-editor-rembg
spec:
  selector:
    app: image-editor-rembg
  ports:
    - protocol: TCP
      port: 8000
      targetPort: 8000
  ---
apiVersion: autoscaling/v2
```

```

kind: HorizontalPodAutoscaler
metadata:
  name: image-editor-rembg-hpa
  namespace: stateless
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: image-editor-rembg
  minReplicas: 1
  maxReplicas: 3
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80

```

role-developer.yaml:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: stateless
  name: role-developer
rules:
  - apiGroups: [""]
    resources: ["pods", "services", "configmaps",
      ↪ "persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "create", "update", "delete"]
  - apiGroups: ["apps"]
    resources: ["deployments", "replicasets", "statefulsets"]
    verbs: ["get", "list", "watch", "create", "update", "delete"]
  - apiGroups: ["batch"]
    resources: ["jobs", "cronjobs"]
    verbs: ["get", "list", "watch", "create", "update", "delete"]
  - apiGroups: ["autoscaling"]
    resources: ["horizontalpodautoscalers"]
    verbs: ["get", "list", "watch", "create", "update", "delete"]

```

role-viewer.yaml:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: stateless
  name: role-viewer
rules:
  - apiGroups: [""] # Core API group

```

```

resources: ["pods", "services", "configmaps",
  ↪ "persistentvolumeclaims"]
verbs: ["get", "list", "watch"]
- apiGroups: ["apps"]
  resources: ["deployments", "replicasets", "statefulsets"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["batch"]
  resources: ["jobs", "cronjobs"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["autoscaling"]
  resources: ["horizontalpodautoscalers"]
  verbs: ["get", "list", "watch"]

```

roleBinding-developer.yaml:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: roleBinding-developer
  namespace: stateless
subjects:
- kind: Group
  name: developer
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: role-developer
  apiGroup: rbac.authorization.k8s.io

```

roleBinding-viewer.yaml:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: roleBinding-viewer
  namespace: stateless
subjects:
- kind: Group
  name: viewer
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: role-viewer
  apiGroup: rbac.authorization.k8s.io

```

A.2 Stateful Manifeste

ghost-deployment.yaml:

```

apiVersion: apps/v1
kind: Deployment

```



```

metadata:
  name: ghost
  namespace: stateful
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ghost
  template:
    metadata:
      labels:
        app: ghost
    spec:
      containers:
        - name: ghost
          image: ghost:5-alpine
          ports:
            - containerPort: 2368
          env:
            - name: url
              value: http://localhost:30081
            - name: database__client
              value: mysql
            - name: database__connection__host
              value: mariadb
            - name: database__connection__user
              valueFrom:
                secretKeyRef:
                  name: ghost-secret
                  key: mariadb-user
            - name: database__connection__password
              valueFrom:
                secretKeyRef:
                  name: ghost-secret
                  key: mariadb-password
            - name: database__connection__database
              value: ghost
            - name: cache__type
              value: redis
            - name: cache__host
              value: redis
      ---
apiVersion: v1
kind: Service
metadata:
  name: ghost
  namespace: stateful
spec:
  type: ClusterIP
  selector:

```

```
  app: ghost
  ports:
    - port: 2368
      targetPort: 2368
```

mariadb-statefulset.yaml:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mariadb
  namespace: stateful
spec:
  serviceName: mariadb
  replicas: 1
  selector:
    matchLabels:
      app: mariadb
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
        - name: mariadb
          image: mariadb:11.8
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: ghost-secret
                  key: mariadb-root-password
            - name: MYSQL_DATABASE
              value: ghost
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  name: ghost-secret
                  key: mariadb-user
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: ghost-secret
                  key: mariadb-password
          volumeMounts:
            - mountPath: /var/lib/mysql
              name: mariadb-storage
      volumes:
        - name: mariadb-storage
          persistentVolumeClaim:
```

```
        claimName: mariadb-storage
---
apiVersion: v1
kind: Service
metadata:
  name: mariadb
  namespace: stateful
spec:
  ports:
    - port: 3306
  selector:
    app: mariadb
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mariadb-storage
  namespace: stateful
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: nfs
  resources:
    requests:
      storage: 5Gi
```

nginx-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: stateful
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          ports:
            - containerPort: 80
          volumeMounts:
            - name: nginx-config-volume
```

```

        mountPath: /etc/nginx/nginx.conf
        subPath: nginx.conf
    volumes:
    - name: nginx-config-volume
      configMap:
        name: nginx-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  namespace: stateful
data:
  nginx.conf: |
    events {}

    http {
      proxy_cache_path /tmp/nginx_cache levels=1:2
        ↪ keys_zone=ghost_cache:10m max_size=100m inactive=5m
        ↪ use_temp_path=off;

      server {
        listen 80;

        location / {
          proxy_cache ghost_cache;
          #proxy_ignore_headers Cache-Control Expires;
          proxy_cache_valid 200 5m;
          proxy_cache_use_stale error timeout updating http_500
            ↪ http_502 http_503 http_504;
          add_header X-Cache $upstream_cache_status;

          proxy_pass http://ghost:2368;
          proxy_set_header Host $host;
          proxy_set_header X-Real-IP $remote_addr;
          proxy_set_header X-Forwarded-For
            ↪ $proxy_add_x_forwarded_for;
          proxy_set_header X-Forwarded-Proto $scheme;
        }
      }
    }
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: stateful
spec:
  type: NodePort
  selector:

```

```
  app: nginx
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30081
```

pv.yaml:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mariadb-pv0-nfs
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: nfs
  nfs:
    path: /srv/nfs/shared/mariadb_0
    server: 10.0.2.10
```

secrets.yaml:

```
apiVersion: v1
kind: Secret
metadata:
  name: ghost-secret
  namespace: stateful
type: Opaque
stringData:
  mariadb-user: "ghostuser"
  mariadb-password: "password"
  mariadb-root-password: "password"
```