Projektdokumentation für das Fach Trainingsfirma Systemintegration

Fachrichtung Fachinformatiker für Systemintegration

PostgreSQL-Cluster

Installation eines ausfallsicheren PostgreSQL Clusters auf K3s mit Backup-Lösung

Bearbeitungszeitraum:

1. Halbjahr 2024/2025

11.09.2024 - 15.01.2025

Bearbeiter:

Heinrich Thaler

www.it-thaler.de

info@it-thaler.de

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bezeichnung
CLI	Command Line Interface
K8s	Kubernetes
LXC	Linux Container
PVCs	Persistent Volume Claims
PVs	Persistent Volumes
SQL	Structured Query Language
iSCSI	internet Small Computer System Interface
SCSI	Small Computer System Interface
KVM	Kernel-basierte Virtual Machine

1. Einleitung

1.1. Projektziel

Das Ziel des Projektes ist die Implementierung eines hochverfügbaren PostgreSQL-Clusters, das auf K3s (einer minimalen Kubernetes-Distribution) läuft. Es soll sicherstellen, dass die PostgreSQL-Datenbank, die kritische Daten für den Schulalltag wie Stundenpläne und Termine enthält, bei Ausfällen nicht unerreichbar wird. Zudem muss ein Backup eingerichtet werden, um Datenverlust zu verhindern und eine schnelle Wiederherstellung zu ermöglichen.

Abbildung 1: Logo von K3s

1.2. Projektbegründung

PostgreSQL ist eine der weltweit führenden Open-Source-Datenbanklösungen, die sich durch ihre Zuverlässigkeit, Skalierbarkeit und Flexibilität auszeichnet. Da die Verwaltung von Datenbanken eine zentrale Rolle in modernen IT-Infrastrukturen spielt, ist es essenziell, deren Verfügbarkeit und Sicherheit zu gewährleisten.

Die Wahl von K3s als Plattform ermöglicht es, eine minimalistische und ressourcenschonende Kubernetes-Umgebung zu nutzen, die optimal für kleinere Infrastrukturen geeignet ist. Gleichzeitig bietet die Backup-Lösung einen Schutz vor Datenverlust, was für den Umgang mit sensiblen und unersetzlichen Daten von entscheidender Bedeutung ist.

Dieses Projekt bietet mir die Möglichkeit, mich intensiv mit modernen Technologien wie Container-Orchestrierung, Hochverfügbarkeit und Datensicherung auseinanderzusetzen. Es wird meine Kenntnisse in der IT-Systemadministration vertiefen und mir einen wertvollen Vorteil in der Berufswelt verschaffen, da Kenntnisse in Virtualisierung, Containerisierung und Datenhaltung zunehmend gefragt sind.

1.3. Projektumfeld

Das Projekt wird an den EDV-Schulen des Landkreises Deggendorf im Rahmen der Ausbildung zum Fachinformatiker für Systemintegration umgesetzt. An der Berufsfachschule erfolgt die Ausbildung praxisnah, mit einem starken Fokus auf die Anwendung der Kenntnisse im späteren echten Berufsleben. Für die Umsetzung der Projekte an der Schule darf die Infrastruktur der Trainingsfirma mitbenutzt werden. Es sind Computer, Server und Netzwerke vorhanden, welche den Schülern während der Durchführung des Projektes bereitstehen.

2. Projektplanung

2.1. Projektphasen

Diese Tabelle enthält eine grobe Zeitplanung zu den verschiedenen Projektphasen:

Projektphase	Geplante Zeit
Einarbeitung in Kubernetes	6 Stunden
IST-Analyse	2 Stunden
SOLL-Analyse	3 Stunden
Installation von K3s	2 Stunden
Installation von Metallb	2 Stunden
Einrichtung PostgreSQL Operator	6 Stunden
Entwicklung Testscripts	4 Stunden
Ausfallsicherheit testen	2 Stunden
Backup einrichten	4 Stunden
Backup testen	1 Stunden
Dokumentation	8 Stunden
Gesamt	**40 Stunden**

2.2. Ressourcenplanung

Es werden mehrere Computer benötigt, um den Cluster aufzusetzen. Da nicht für jedes Schülerprojekt mehrere Server ausgehändigt werden können, soll das Projekt auf LXC-Containern eingerichtet werden. Es werden keine leistungsstärkeren Rechner benötigt, da K3s sehr Ressourceneffizient ist. Zudem wurde die bestehende Infrastruktur wie z.B. das Netzwerk der Schule benutzt.

2.3. Entwicklungsprozess

Um möglichst flexibel zu bleiben, wurde dieses Projekt mit einem agilen Entwicklungsprozess bearbeitet. Der Fortschritt wurde regelmäßig mit Herrn Grallinger abgesprochen, damit das Projekt auf Kurs bleibt. Nach jedem Schritt in der Implementierungsphase wurde das Umgesetzte sofort getestet und dokumentiert.

3. Analysephase

3.1. Ist-Analyse

Bisher greifen die KlaTab App und andere schulinterne Anwendungen auf die „sqlprod“ Datenbank zu. Diese wird aktuell auf nur einem einzigen Server bereitgestellt und stellt dadurch einen „Single Point of Failure“ dar. Dadurch ist sie anfällig für Ausfälle, welche vermieden sollen, da diese von vielen Schülern und Lehrern rund um die Uhr verwendet wird. Diese Daten sind sehr wichtig für den reibungslosen Ablauf der Verwaltung und Organisation. Ein längerer Ausfall dieses Systems würde den Schulalltag beeinträchtigen, da als Folge unter anderem der Stundenplan, Prüfungen und Termine nicht mehr verfügbar wären.

3.2. Nutzwertanalyse

Auf die Daten der Datenbank greifen ausschließlich an der EDV-Schule entwickelte Anwendungen zu. Diese Anwendungen stehen nur schulinternen Personen zur Verfügung, können jedoch über das Internet erreicht werden, um beispielsweise von zu Hause aus den Stundenplan oder anstehende Prüfungen einzusehen. Der geplante Cluster soll sicherstellen, dass ein Großausfall aller Systeme verhindert wird und die Verfügbarkeit der Anwendungen auch bei technischen Problemen gewährleistet bleibt.

3.2.1. Bedeutung für die Nutzer

Schüler A:
„Als Benutzer der KlaTab-App möchte ich den Stundenplan der EDV-Schule rund um die Uhr abrufen können. Sollte die App plötzlich nicht mehr funktionieren, wäre das sehr ärgerlich, da ich darüber nicht nur den Stundenplan, sondern auch Termine und Prüfungen einsehen kann. Ohne die App müsste ich diese Informationen anderweitig organisieren, was zusätzlichen Aufwand bedeutet.“

Sekretärin B:
„Als Sekretärin bin ich darauf angewiesen, Termine in die Datenbank einpflegen zu können. Dieses System ermöglicht es, sämtliche anstehenden Ereignisse zentral und übersichtlich zu organisieren. Ein längerer Systemausfall würde uns zwingen, die Termine manuell zu koordinieren und herauszugeben, was zu einem erheblichen Mehraufwand und potenziellem Chaos führen könnte.“

3.3. Kosten-Nutzen-Faktor

In diesem Projekt kommt ausschließlich Open-Source-Software zum Einsatz. Dadurch entfallen Lizenzkosten, und die finanziellen Aufwendungen beschränken sich auf die Betriebskosten der Trainingsfirmenserver, wie Strom- und Wartungskosten. Trotz des geringen finanziellen Aufwands bietet das System erhebliche Vorteile, wie eine erhöhte Ausfallsicherheit, bessere Skalierbarkeit und eine zuverlässige Datenverfügbarkeit. Somit ergibt sich ein hoher Kosten-Nutzen-Faktor, da das System mit minimalen Kosten einen großen Mehrwert für die Schule schafft.

3.4. Qualitätsanforderungen

3.4.1. Funktionalität

Die Datenbank muss vom Schulnetz aus erreichbar sein, da Anwendungen wie Stundenpläne und Terminübersichten rund um die Uhr benötigt werden. Bestehende Anwendungen sollen unverändert bleiben, um eine reibungslose Migration zu gewährleisten. Automatisierte Backups sichern Daten gegen Verluste durch Fehler oder Sicherheitsvorfälle.

3.4.2. Ausfallsicherheit

Fällt eine Node aus, muss der Betrieb sofort von anderen Nodes übernommen werden, um die Verfügbarkeit sicherzustellen. Zudem sollen durch Backups verloren gegangene Daten ohne großen Aufwand wiederherstellbar sein, was die Systemstabilität und Benutzerzufriedenheit erhöht.

3.4.3. Wartbarkeit

Eine einfache Skalierung der Datenbank sollte möglich sein, falls die Nutzung in der Zukunft ansteigt. Künftige Schüler müssen das System verwenden können und bei Bedarf sollte die Möglichkeit bestehen es abzuändern.

3.4.4. Effizienz

Durch den dezentralen Clusterbetrieb mit K3s und Patroni entsteht ein erhebliches Overhead. Trotzdem sollten Ressourcen nicht unnötig verschwendet werden, um unter anderem Strom zu sparen und Antwortzeiten zu erhöhen.

3.5. Einarbeitung

Ich wollte mich außerschulisch bereits mit Kubernetes auseinandersetzen. Ich habe aber sehr schnell gemerkt, dass es sehr Komplex ist. Bisher fehlte mir die Gelegenheit, mich intensiver damit zu beschäftigen. Dieses Projekt bietet jedoch die perfekte Möglichkeit, grundlegende Kenntnisse über Kubernetes zu erwerben und praxisnah anzuwenden.

Zu Beginn meiner Einarbeitung habe ich Zuhause insgesamt über fünf Stunden Videomaterial angesehen [1], um die grundlegenden Konzepte und Funktionsweisen von Kubernetes zu verstehen. Dabei habe ich mich auf die zentralen Themen wie Cluster-Struktur, Ressourcenverwaltung und die Bedeutung von Containern im Kontext von Kubernetes konzentriert.

Im Anschluss daran habe ich gezielt recherchiert, welche Technologien und Kubernetes-Komponenten für die Realisierung des Projekts im Kapitel Error: Reference source not found geeignet sind. Dabei habe ich die folgenden Kernkomponenten identifiziert, die für das Verständnis und die praktische Umsetzung wesentlich sind:

3.5.1. Kubernetes Komponenten

3.5.1.1. Pods

Pods sind die kleinsten und einfachsten Einheiten in Kubernetes, die einen oder mehrere Container umfassen. Sie bilden die Basis für die Ausführung von Anwendungen im Cluster.

3.5.1.2. Nodes

Nodes sind die einzelnen Rechner, die Teil des Kubernetes-Clusters sind. Sie übernehmen die Ausführung der Workloads und stellen die benötigten Ressourcen wie CPU und Speicher bereit.

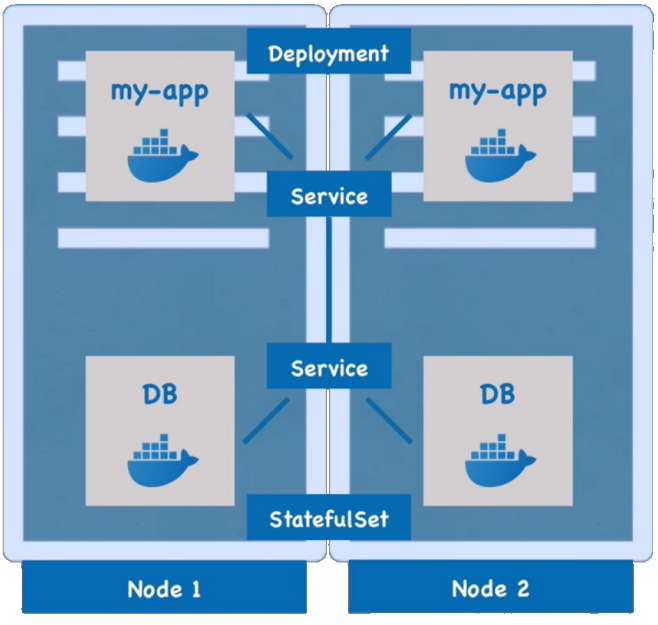

Abbildung 2: Übersicht der K8s Komponenten [1]

3.5.1.3. Deployments

Deployments ermöglichen die Verwaltung und Skalierung von Anwendungen. Sie bieten Mechanismen wie Rollouts und Rollbacks, um Änderungen an Anwendungen sicher durchzuführen.

3.5.1.4. StatefulSets

StatefulSets sind wie Deployments, aber für Anwendungen welche einen Zustand haben. Jeder Pod erhält eine fortlaufende Nummer, mit der sie identifiziert werden können. Sie werden beispielsweise für Datenbanken verwendet und sind daher für dieses Projekt besonders relevant.

3.5.1.5. Services

Services stellen sicher, dass Pods, die dynamisch erstellt oder entfernt werden, zuverlässig angesprochen werden können. Sie ermöglichen eine stabile Kommunikation innerhalb des Clusters und mit externen Clients.

3.5.1.6. ConfigMaps

ConfigMaps dienen der Verwaltung von Konfigurationsdaten, die unabhängig vom Anwendungscode gespeichert werden. Sie ermöglichen die Anpassung von Anwendungen ohne erneutes Erstellen von Images.

3.5.1.7. Persistent Volumes & Claims

Ein **Persistent Volume** ist eine von Kubernetes bereitgestellte Speicherressource, die unabhängig von der Lebensdauer eines Pods existiert. PVs werden in der Regel von einem Administrator erstellt und basieren auf einem zugrunde liegenden Speicher (z. B. lokale Festplatten, NFS, AWS EBS, oder Azure Disk).

Ein **Persistent Volume Claim** ist eine Anfrage eines Pods nach einem bestimmten Speicherplatz. PVCs werden von Entwicklern definiert und spezifizieren die Anforderungen wie Größe, Zugriffsmodus und ggf. die gewünschte StorageClass.

3.5.2. Kubectl

kubectl ist das Kommandozeilen-Tool (CLI) zur Interaktion mit Kubernetes-Clustern. Es dient der Verwaltung von Ressourcen wie Pods, Services und Deployments.

Einige der wichtigsten kubectl-Befehle sind:

- **Pods auflisten:**
  Um alle Pods im aktuellen Namespace anzuzeigen, verwendet man den folgenden Befehl:
  `kubectl get pods`
  Mit dem Parameter `-n` kann ein bestimmter Namespace angegeben werden, z.B.:
  `kubectl get pods -n <namespace>`
- **Services auflisten:**
  Um alle Services im aktuellen Namespace aufzulisten, kann folgender Befehl verwendet werden:
  `kubectl get services`
- **Ressourcen aus einer Datei anwenden:**
  Um eine Konfigurationsdatei (z. B. eine YAML-Datei) auf das Cluster anzuwenden,

verwendet man:
`kubectl apply -f <dateiname>.yaml`
Dies wird häufig genutzt, um Deployments, Services oder andere Ressourcen zu erstellen oder zu aktualisieren.

3.6. Technologieentscheidungen

3.6.1. PostgreSQL-Operator

Der PostgreSQL Operator bietet eine automatisierte und vereinfachte Verwaltung von PostgreSQL-Datenbanken in Kubernetes-Clustern. Zu den wesentlichen Vorteilen zählen:

- **Automatisierte Skalierung und Verwaltung:** Der Operator übernimmt komplexe Aufgaben wie das automatische Anlegen, Verwalten und Skalieren von PostgreSQL-Clustern, was den manuellen Aufwand erheblich reduziert.
- **Hochverfügbarkeit durch Patroni:** Patroni, ein integraler Bestandteil des PostgreSQL Operators, gewährleistet Hochverfügbarkeit, indem es automatische Failover-Mechanismen und eine kontinuierliche Überwachung der Nodes bietet.
- **Kubernetes-Integration:** Da der Operator speziell für Kubernetes entwickelt wurde, nutzt er native K8s-Funktionen wie StatefulSets, ConfigMaps und PersistentVolumeClaims. Dies erhöht die Effizienz und vereinfacht die Konfiguration.

Im Gegensatz zu einer manuellen PostgreSQL-Installation spart der PostgreSQL Operator Zeit, verbessert die Zuverlässigkeit und bietet ein skalierbares Datenbankmanagement, das sich nahtlos in die K3s-Umgebung integriert.

3.6.2. Loadbalancer

MetalLB vs. Keepalived
Im Projekt wurden zwei Loadbalancer-Technologien untersucht: MetalLB und Keepalived.

- **MetalLB:**
    - MetalLB ist ein speziell für Kubernetes entwickelter Loadbalancer, der externe IP-Adressen in Bare-Metal-Clustern bereitstellt.
    - **Vorteile:**
        - Nahtlose Integration in Kubernetes-Cluster: MetalLB arbeitet mit den Kubernetes-Netzwerkressourcen wie Services und Endpoints zusammen.
        - Unterstützt sowohl Layer 2 (ARP) als auch Layer 3 (BGP) Modi.
        - Automatische Konfiguration und Verwaltung über Kubernetes-Manifestdateien.
    - **Einschränkungen:**
        - MetalLB ist ausschließlich in Kubernetes-Umgebungen verwendbar und nicht für andere Netzwerkarchitekturen geeignet.
- **Keepalived:**

- Keepalived ist eine universelle Loadbalancer-Lösung, die nicht von Kubernetes abhängig ist und auch in klassischen Netzwerken eingesetzt werden kann.
- **Vorteile:**
    - Unterstützt Virtual Router Redundancy Protocol (VRRP) zur Bereitstellung von Hochverfügbarkeit in Netzwerken.
    - Kann unabhängig von Kubernetes betrieben werden, was in hybriden Umgebungen vorteilhaft ist.
- **Einschränkungen:**
    - Die Integration mit Kubernetes ist weniger nahtlos und erfordert zusätzlichen Aufwand zur Verwaltung.

Entscheidung für MetalLB:
MetalLB wurde gewählt, da es speziell für Kubernetes entwickelt wurde und sich optimal in die K3s-Umgebung integriert. Durch die einfache Konfiguration und Verwaltung direkt über Kubernetes-Manifestdateien bietet MetalLB eine klare, effiziente und ressourcenschonende Lösung für Bare-Metal-Umgebungen. Keepalived wurde verworfen, da die zusätzliche Komplexität in einem Kubernetes-Cluster nicht gerechtfertigt war und MetalLB die Anforderungen des Projekts vollständig erfüllte.

3.6.3. Kubernetes-Distribution

K3s ist eine leichtgewichtige, zertifizierte Kubernetes-Distribution, die speziell für Edge-Computing, IoT und ressourcenbeschränkte Umgebungen entwickelt wurde. Es bietet eine Reihe von Vorteilen, die es zur idealen Wahl für das Projekt machen:

- **Geringe Ressourcenanforderungen:** K3s benötigt deutlich weniger Speicher und CPU als andere Kubernetes-Distributionen und ist daher ideal für virtuelle Umgebungen.
- **Einfache Installation:** Als Single Binary gestaltet, vereinfacht K3s die Installation und reduziert den Administrationsaufwand.
- **Integrierte Komponenten:** K3s enthält vorinstallierte Tools wie containerd, SQLite und Loadbalancer-Unterstützung (z. B. MetalLB). Häufig genutzte Features wie Helm sind bereits integriert.

Vergleich mit Alternativen:

- **Standard-Kubernetes (z. B. via Kubeadm):**
    - Bietet zwar maximale Flexibilität, erfordert jedoch eine manuelle Konfiguration und Verwaltung, was den Administrationsaufwand erhöht.
    - Ressourcenintensiver, was für die Anforderungen des Projekts ungeeignet ist.
- **MicroK8s:**
    - Ebenfalls eine leichtgewichtige Kubernetes-Distribution, jedoch primär für Ubuntu-Umgebungen optimiert und weniger flexibel in heterogenen Betriebssystemlandschaften.

- Im Vergleich zu K3s komplexer, wenn es um die Konfiguration von externen Loadbalancern oder Datenbanken geht.

Entscheidung für K3s:
K3s wurde gewählt, da es eine ideale Balance aus Funktionalität, Einfachheit und Ressourcenschonung bietet. Im Kontext des Projekts war K3s daher die effektivste Lösung.

3.7. Failover Mechanismus

Ein Failover bezeichnet den Wechsel von einer primären Instanz auf eine sekundäre Instanz, wenn die primäre Instanz ausfällt. Im Folgenden wird beschrieben, wie der Failover im Hintergrund abläuft:

3.7.1. Failover Ablauf

PostgreSQL mit Patroni:

Patroni überwacht kontinuierlich die Verfügbarkeit der PostgreSQL-Instanzen. Im Falle eines Ausfalls der Primary-Instanz initiiert Patroni einen Leader-Election-Prozess, um eine neue Primary zu bestimmen. (Abb. 3) Die neue Primary wird aktiviert, und die verbleibenden Replicas werden entsprechend aktualisiert, um die Datenreplikation von der neuen Primary zu übernehmen.

Datenreplikation:

PostgreSQL-Replicas sind über synchrone oder asynchrone Replikation auf dem neuesten Stand, wodurch beim Failover keine oder minimale Datenverluste auftreten.

Service Umleitung:

Nach einem Failover aktualisiert Patroni die Endpunkte des Services, sodass der Kubernetes-Service den Netzwerkverkehr automatisch zur neuen Primary-Instanz umleitet. Sobald der Failover-Prozess abgeschlossen ist, wird der Netzwerkverkehr nahtlos umgeleitet.

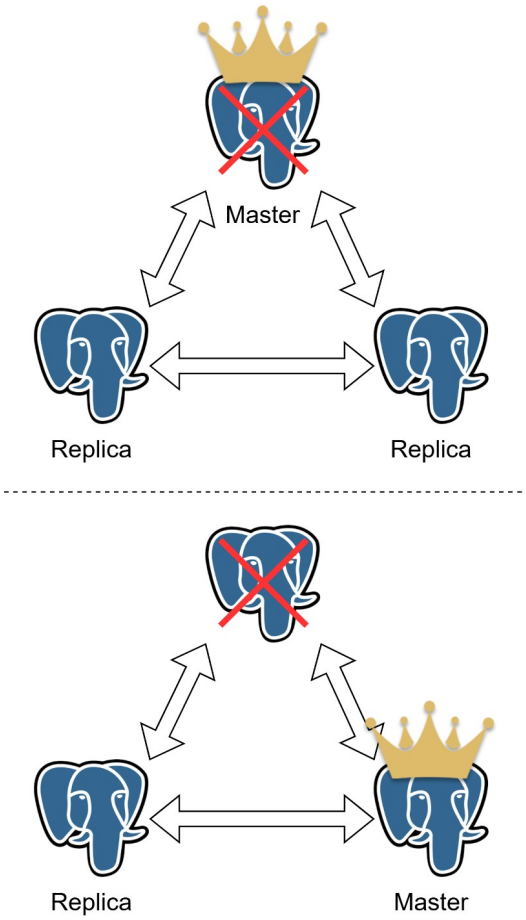

Abbildung 3: PostgreSQL Failover mit Patroni

3.7.2. Auswirkungen auf den Benutzer

Idealfall :

- **Keine spürbaren Auswirkungen:** Der Failover erfolgt schnell, sodass Benutzer keine Unterbrechung bemerken und Anwendungen reibungslos weiterlaufen.
- **Minimaler Performance-Einbruch:** Kurzzeitige Verzögerungen könnten auftreten, wenn Anfragen erneut geroutet werden.

Worst-Case-Szenario:

- **Ausfall des Services:** Die Datenbank ist nicht mehr erreichbar und dadurch fallen alle abhängigen Anwendungen aus.

- **Fehlerhafte Datenzustände:** Veraltete Daten oder Schreibfehler, falls die Synchronisation nicht richtig abgeschlossen war.

4. Entwurfsphase

4.1. Topologie

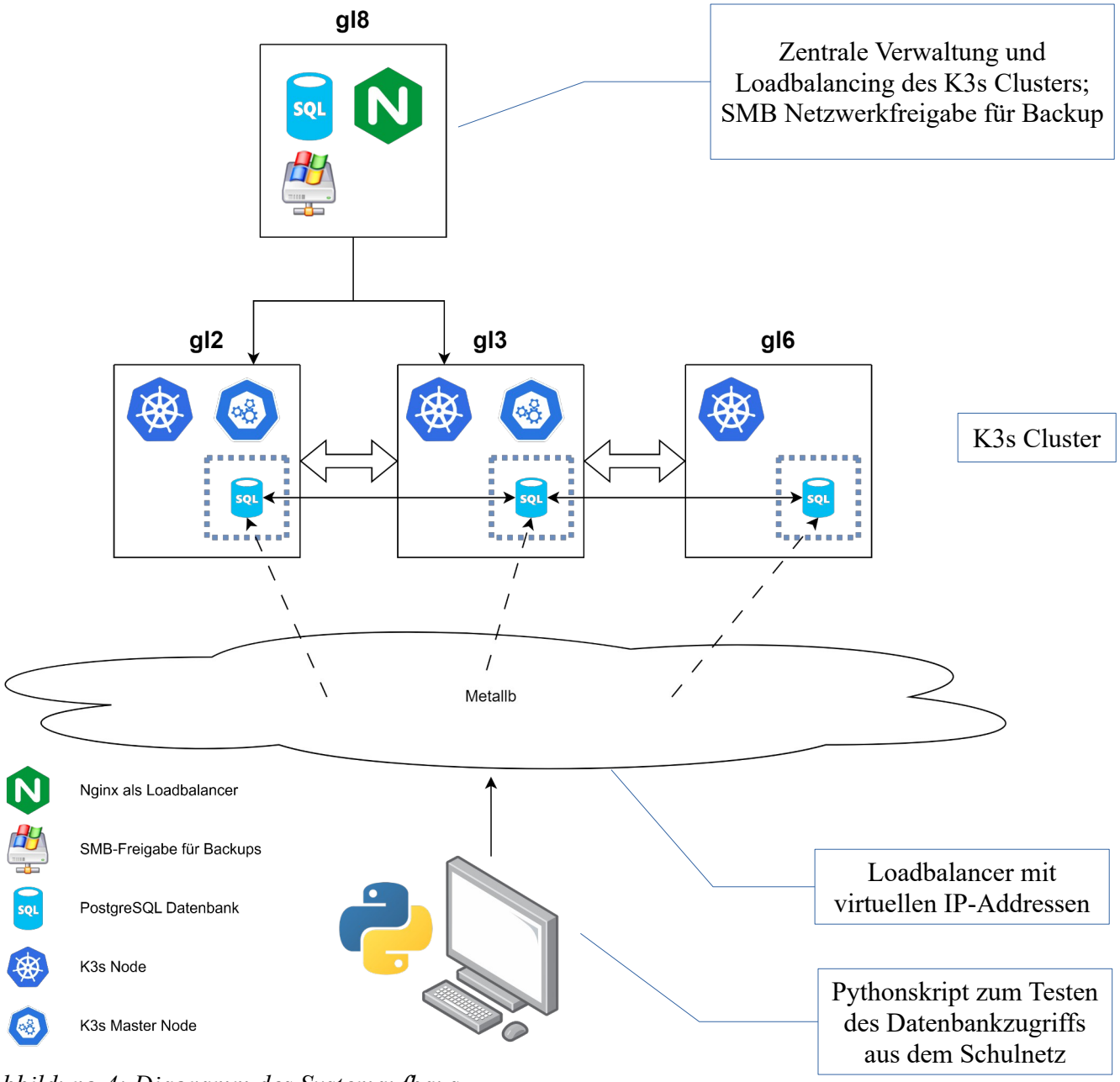

Abbildung 4: Diagramm des Systemaufbaus

4.2. Virtuelle Maschinen

Folgende virtuelle Maschinen mit den jeweiligen Rollen werden benötigt:

Maschine	Rolle
gl2	K8s Master & Worker
gl3	K8s Master & Worker
gl6	K8s Worker
gl8	Loadbalancer für K8s, Datenbank für K8s, Dateifreigabe für Backups

gl6 wird als eine **Worker-Node** eingestuft. Worker-Nodes führen die Workloads aus, die von Kubernetes orchestriert werden, wie beispielsweise Pods und andere Ressourcen, die die Applikationen bereitstellen.

gl2 und **gl3** fungieren zusätzlich als **Master-Node**. Dort laufen die Kern-Komponenten von Kubernetes, wie der **API-Server**, der **Controller Manager**, der **Scheduler** und das **etcd**-Speichersystem. Diese Komponenten steuern den gesamten Cluster, treffen Entscheidungen über die Platzierung von Workloads und verwalten den Cluster-Zustand.

gl8 ist nicht teil des Kubernetes-Clusters, übernimmt aber mehrere wichtige Aufgaben:

1. **Loadbalancer für Kubernetes:** Eingehender Traffic auf die K8s API wird gleichmäßig auf die Master-Nodes (gl2 und gl3) und stellt sicher, dass der Cluster auch bei einem Ausfall einer Master-Node verfügbar bleibt.
2. **Datenbank für Kubernetes:** Für die Nutzung der eingebetteten etcd mit Hochverfügbarkeit wären mindestens drei Master-Knoten erforderlich. Um die Komplexität zu reduzieren wurde die Datenbank für dieses Projekt ausgelagert, auch wenn das theoretisch einen weiteren Single Point of Failure darstellt.
3. **SMB-Freigabe**: Ermöglicht die Sicherung der PostgreSQL-Daten durch eine zentrale Dateifreigabe.

4.3. Netzwerk

Das von der Schule bereitgestellte TrFi Netzwerk hat einen Internetzugang und DHCP Server. Den Containern sollen trotzdem folgende statische IP-Adressen zugeteilt werden:

Maschine	IP-Adresse
gl2	10.0.145.110
gl3	10.0.145.111
gl6	10.0.145.112
gl8	10.0.145.113

Die Dienste welche über Metallb bereitgestellt werden, sollen virtuelle IP-Adressen aus dem folgenden Bereich zugewiesen bekommen:

10.0.145.130 – 10.0.145.149

5. Implementierungsphase

5.1. LXC-Container

Es waren bereits zwei LXC-Container vorhanden. Jedoch wurden vier Benötigt. Die weiteren wurden durch Klonen einer Debian-Vorlage erstellt. Damit die Container miteinander kommunizieren können, benötigen diese einen Eintrag in der Hosts-Datei auf jeder Maschine.

5.2. K3s Cluster

Mithilfe der offiziellen Anleitung von K3s wurde es installiert und eingerichtet. Hier entstand das erste Problem: K3s konnte nicht Starten und gab folgende Fehlermeldung aus: `/dev/kmsg: no such file or directory`
Der Fehler wurde mithilfe von Herrn Grallinger und diesem Beitrag im Proxmox-Forum behoben: [2]. Der Container hatte nicht genügend rechte, um auf /dev/kmsg zuzugreifen, obwohl es kubelet benötigte. Dies führte zu einem Absturz mit der bereits genannten Fehlermeldung.

Zwei der Nodes wurden zum Master hochgestuft, sodass bei einem Ausfall der andere übernehmen kann.

```
hans@gl8:~$ kubectl get nodes
NAME   STATUS   ROLES                  AGE   VERSION
gl2    Ready    control-plane,master   52d   v1.30.4+k3s1
gl3    Ready    control-plane,master   52d   v1.30.5+k3s1
gl6    Ready    <none>                 52d   v1.30.5+k3s1
hans@gl8:~$
```

Abbildung 5: Auflistung der Nodes im K3s Cluster

Um K3s Hochverfügbar zu machen, müssen zwei Vorkehrungen getroffen werden: Auf Gl8 wurde Nginx eingerichtet (Anhang 9.2), um als Load-Balancer einen ausfallsicheren Zugriff auf den Cluster zu gewährleisten. Zudem wurde darauf eine PostgreSQL-Datenbank installiert, welche die Konfiguration von Kubernetes speichert.

5.3. Helm

Der PostgreSQL-Cluster wurde mithilfe von Helm installiert. Helm ist ein Open-Source Tool, welches als Paketmanager für Kubernetes dient. Es hilft, Anwendungen und deren zugehörige Ressourcen auf Kubernetes einfacher bereitzustellen, zu verwalten und zu aktualisieren.

Helm kann mithilfe eines Befehls von der Webseite installiert werden. Um eine Helm-Chart zu installieren, muss zuerst das Repo hinzugefügt werden. Anschließend kann mithilfe dieses Befehls PostgreSQL Operator installiert werden:
`helm install postgres-operator postgres-operator-charts/postgres-operator`

5.4. PostgreSQL Operator

PostgreSQL Operator automatisiert die Bereitstellung, Verwaltung und Skalierung von PostgreSQL-Datenbanken. Mithilfe der Postgesql Operator UI (Abb. 6) wurde die Konfigurationsdatei wichtige-daten-3.yaml (Anhang 9.3) erstellt. Es enthält Einstellungen wie den Namen des Clusters, die Standarddatenbank und die Anzahl der Replicas. Mithilfe von `kubectl apply -f` wird die

Konfiguration angewandt oder aktualisiert. Indem man die Anzahl der Replicas erhöht, kann man die Performance des Clusters erheblich steigern.

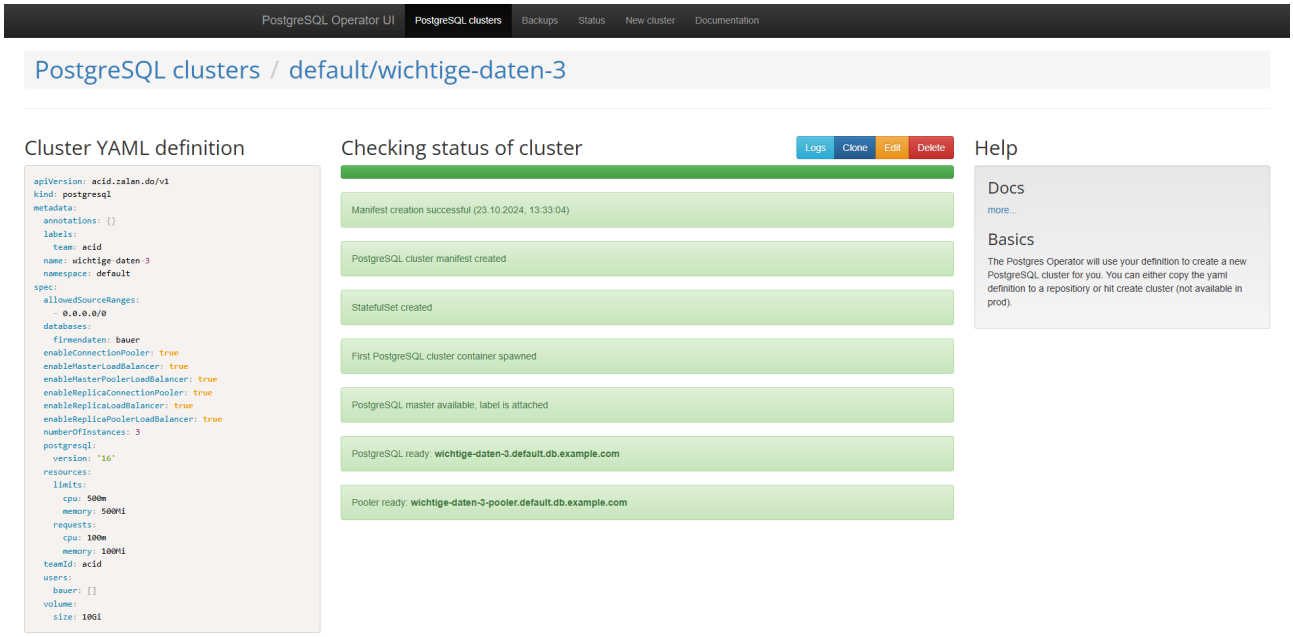

Abbildung 6: Konfiguration des PostgeSQL-Clusters in der UI

Etwa eine Minute nach der Aktivierung, je nachdem wie viele Replicas man angegeben hat, ist der PostgreSQL-Cluster einsatzbereit. Mithilfe eines temporären Portforwardings kann bereits eine Verbindung aufgebaut werden. Das Passwort des Admin-Benutzers „postgres“ wird vom System beim Erstellen selbst erstellt. Dieses kann mit dem folgenden Befehl ausgelesen werden:

```
kubectl get secret postgres.wichtige-
daten-3.credentials.postgresql.acid.zalan.do -n default -o
jsonpath="{.data.password}" | base64 --decode
```

Schreibzugriffe können leider immer nur von der Master-Node ausgeführt werden, da es sonst zu inkonsistenten Daten kommen könnte.

5.5. Metallb

Damit die Datenbank dauerhaft von außerhalb benutzt werden kann, wird ein Loadbalancer wie Metallb benötigt. Es ist eine Lösung für Kubernetes-Cluster, die es ermöglicht, externe virtuelle IP-Adressen bereitzustellen und eingehenden Traffic auf die entsprechenden Kubernetes-Services zu verteilen.

Mithilfe der offiziellen Anleitung wurde es Installiert und anschließend konfiguriert (Anhang 9.4). IP-Adressen aus dem Bereich 10.0.145.130 – 149 werden mit dem layer2 Protokoll angeworben. Das heißt, dass ARP-Pakete verschickt werden, welche eine MAC-Adresse für die IP des Services beinhaltet.

5.6. Testscript

Mithilfe der Python-Bibliothek *Faker* wurde die Datenbank mit zufälligen Werten befüllt. Anschließend wurde mit dem im Anhang 9.5 zu sehenden Python-Skript db-test-multi.py ein Stresstest der Datenbank durchgeführt. Es verwendet Multithreading, um mehrere Anfragen parallel zu senden. Dadurch wird getestet, wie viel das Cluster aushalten kann.

```python
if __name__ == "__main__":
    # Run 30 concurrent read loops
    with concurrent.futures.ThreadPoolExecutor(max_workers=600) as executor:
        # Launch 30 threads
        futures = [executor.submit(read_loop, thread_id) for thread_id in range(300)]

        try:
            # Wait for all threads to complete
            concurrent.futures.wait(futures)
        except KeyboardInterrupt:
            print("Main program exiting...")
```

Abbildung 7: Codeausschnitt des Testscripts für Multithreading

Das Programm gibt bei hohen Threadzahlen regelmäßige Fehler aus, da die Datenbank keine erfolgreiche Verbindung mit allen Verbindungsanfragen aufbauen kann.

```
Thread- 26 | Random ID:  90, Name: Joshua Moreno       , Num_col:  71
Thread-201 | Random ID:  62, Name: Kathy Black         , Num_col:  91Thread-215 | Random ID
0.0.145.135", port 5432 failed: FATAL:  sorry, too many clients already' Retrying...Thread-
me: Theresa Bowers      , Num_col:  72



Thread-126 | Random ID:  61, Name: Brett Blackwell     , Num_col:  45
Thread- 13 | Random ID:  78, Name: Theresa Bowers      , Num_col:  72

Thread-271 | Random ID:  91, Name: Melissa Bowen       , Num_col:  26Thread-177 | Random ID

Thread-154 | Random ID:  19, Name: Jeffrey Johnson     , Num_col:  35Thread- 10 | Random ID
0.0.145.135", port 5432 failed: FATAL:  sorry, too many clients already' Retrying...

Thread-235 | Random ID:  54, Name: Donna Pineda        , Num_col:  49Thread- 98 | Random ID
0.0.145.135", port 5432 failed: FATAL:  sorry, too many clients already' Retrying...
Connection failed: 'connection to server at "10.0.145.135", port 5432 failed: FATAL:  sorry



Connecting...Connecting...Connecting...

Thread- 43 | Random ID:  73, Name: Charles Marshall    , Num_col:  20
Thread- 33 | Random ID:  72, Name: Roberto Parker      , Num_col:  67Connection failed: 'co
dy' Retrying...Thread-138 | Random ID:  92, Name: Samuel Barker       , Num_col:  48
```

Abbildung 8: Ausgabe des Testprogramms

Wenige Replicas werden durch die hohe Anzahl der Anfragen sehr schnell ausgelastet. Dank PostgreSQL-Operator ist es aber sehr leicht, auf z.B. 20 Replicas zu skalieren. Dazu ändert man die Anzahl in der Konfigurationsdatei und wendet sie mit kubectl an. Dann kann die Datenbank nicht mehr von einem einzelnen PC verlangsamt werden.

5.7. Backup

Eine Samba Freigabe wurde auf der gl8 eingerichtet. Diese wird als der Speicherort für die Backups ausgewählt. In einem Cronjob (Anhang 9.6) wird regelmäßig der pg_dump befehl ausgeführt.

Mit dem psql Befehl kann das Backup wieder eingespielt werden:

```
psql --set=sslmode=require -h 10.0.145.138 -p 5432 -U postgres -d firmendaten -f /home/hans/backup/backup_20250115_132205.sql
```

5.8. Longhorn

Longhorn ist eine Open-Source-Lösung zur Verwaltung von verteiltem Speicher in Kubernetes-Clustern. Es ermöglicht eine dezentrale Speicherung, indem Daten redundant auf verschiedenen Nodes verteilt werden. Dadurch wird die Verfügbarkeit und Ausfallsicherheit der Daten erhöht, selbst wenn eine Node ausfällt.

Um dieses Projekt zu erweitern, sollen Backups nicht nur auf einem zentralen Speicher, sondern dezentral auf allen Nodes des Clusters gespeichert werden. Longhorn kann direkt in den Kubernetes-Cluster integriert werden und bietet optional eine benutzerfreundliche Weboberfläche zur Verwaltung. Mit der Implementierung von Longhorn wird sichergestellt, dass Backups sicher und hochverfügbar gespeichert sind, wodurch die Zuverlässigkeit des Systems weiter verbessert wird.

Die Installation von Longhorn verlief ohne Probleme, und die Weboberfläche konnte erreicht werden. Im Dashboard wurden alle Nodes und der verfügbare Speicher angezeigt. Allerdings wurde der verfügbare Speicher dreimal so hoch angezeigt, wie er tatsächlich war. Dieser Fehler resultiert daraus, dass die LXC-Container auf derselben physischen Festplatte gespeichert sind. Longhorn interpretiert den Speicher jeder Node als unabhängig, da es keine Informationen darüber hat, dass alle Nodes auf dieselbe zugrunde liegende Festplatte zugreifen. In Umgebungen mit physisch getrennten Nodes wäre diese Darstellung korrekt, in einem Szenario mit gemeinsam genutztem Speicher jedoch irreführend.

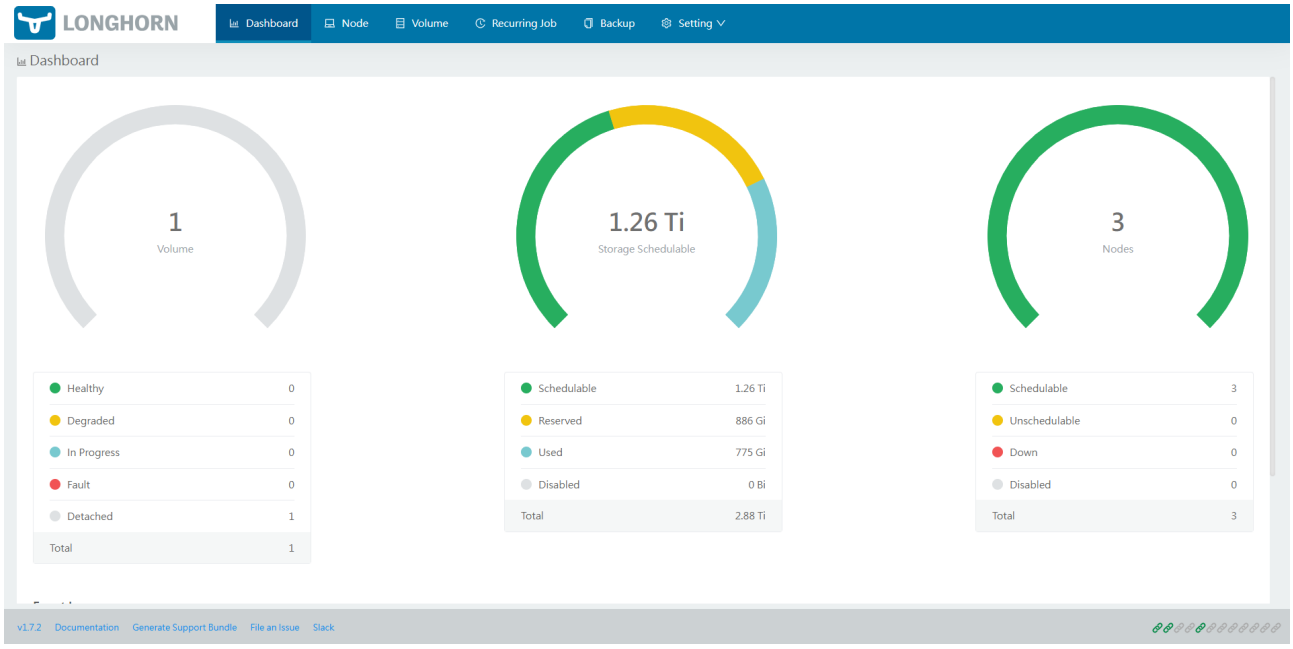

Abbildung 9: Longhorn UI Dashboard

Mithilfe der UI wurde ein Volume mit zugehörigem PV und PVC erstellt. Anschließend wurde die folgende Zeile des Cronjobs bearbeitet, um den neu erstellten Claim zu verwenden:

claimName: postgres-bak

Beim Testen blieb der pod des Jobs jedoch im Status „ContainerCreating“, und in der UI wurde zwischen dem Status „Attaching“ und „Failed“ gewechselt. Die Fehlermeldungen in der UI war relativ nichtsaussagend. Über die Installationsanleitung von Longhorn [3] stieß ich auf iSCSI. Ich hatte es zwar installiert, jedoch bietet Longhorn seinen eigenen Installer. Diese Container konnten allerdings nicht starten. Der Befehl `modprobe iscsi_tcp` folgenden Fehler:

```
hans@gl3:~$ sudo modprobe iscsi_tcp
modprobe: FATAL: Module iscsi_tcp not found in directory /lib/modules/6.1.0-28-amd64
```

Longhorn benötigt dieses Modul, um die Volumes mit den Containern zu verbinden. Eine Internetrecherche ergab sich leider erfolglos. Jemand anderes hatte bereits im Proxmox Forum das exakt selbe Problem. [4] Leider wird iSCSI zurzeit in LXC-Containern nicht unterstützt. Anstatt LXC hätte man KVM verwenden können, was den Fehler beheben würde. Ein nachträglicher Umzug ist für dieses Projekt jedoch nicht mehr machbar.

6. Abnahmephase

Das Projekt wurde Herrn Grallinger am 15.01.2025 vorgestellt und es wurden mögliche Verbesserungen besprochen. Eine Woche später wurde das Projekt der Klasse präsentiert. Alle in Kapitel 5 bearbeitete Anforderungen wurden nach der Bearbeitung sofort auf getestet und die Ergebnisse notiert. Abschließend wurde mithilfe der Testscripts wie bereits in Kapitel 5.6 beschrieben die Funktion und Ausfallzeiten überprüft.

7. Dokumentation

Jede Tätigkeit wurde nach Datum sortiert aufsteigend und mit der benötigten Zeit in Joplin im Markdown-Format dokumentiert. Zudem wurden verwendete Befehle oder Quellen notiert. Die gesammelten Notizen flossen in dieses Dokument detailgetreu mit ein. Anschließend wurde eine Präsentation erstellt, damit das Projekt und dessen Bearbeitung vor der Klasse präsentiert werden kann.

8. Fazit

8.1. Soll-/Ist-Vergleich

Der K3s Cluster und die Datenbank befindet sich in einem funktionsfähigen Zustand und alle Anforderungen wurden bearbeitet. Lediglich die Erweiterung durch Longhorn schlug fehl, da das iSCSI Modul auf den LXC-Containern fehlte. Mithilfe der Notizen in Joplin wurde die tatsächlich benötigte Zeit für die verschiedenen Arbeitsschritte berechnet. Diese Tabelle enthält die tatsächlich aufgewendete Zeit für jeden Projektabschnitt:

Projektphase	Geplante Zeit	Benötigte Zeit
Einarbeitung in Kubernetes	6 Stunden	6 Stunden
IST-Analyse	2 Stunden	2 Stunden
SOLL-Analyse	3 Stunden	3 Stunden
Installation von K3s	2 Stunden	4 Stunden
Installation von Metallb	2 Stunden	1 Stunden
Einrichtung PostgreSQL Operator	6 Stunden	5 Stunden
Entwicklung Testscripts	4 Stunden	4 Stunden
Ausfallsicherheit testen	2 Stunden	2 Stunden
Backup einrichten	4 Stunden	4 Stunden
Backup testen	1 Stunden	1 Stunden
Dokumentation	8 Stunden	8 Stunden
Gesamt	**40 Stunden**	**40 Stunden**

Da bei der Installation von K3s (Kapitel 5.2) der KMSG-Fehler auftrat, wurden dort 2 Stunden mehr benötigt. Diese Zeit konnte dank der Einfachen und Schnellen Installation von Metallb wieder eingeholt werden.

Für Longhorn als Erweiterung wurden zusätzlich 5 Stunden zur Installation und Fehlerfindung benötigt.

8.2. Persönliches Fazit

Dank dieses Projektes durfte ich mich intensiv mit modernen Technologien wie Hochverfügbarkeit, Container-Orchestrierung und Datensicherung auseinandersetzen. Verteilte Datenhaltung und zuverlässige Hochverfügbarkeit sind ein sehr schwieriges und komplexes Thema. Trotzdem ist es sehr wichtig und in der Praxis in fast allen Bereichen notwendig, wo mit hohen Benutzerzahlen gerechnet werden.

Anstatt LXC hätte für dieses Projekt KVM verwendet werden sollen. Manche Module die für K3s oder Longhorn gebraucht werden fehlen bei LXC-Containern. KVM emuliert die Hardware, weshalb die Probleme dann nicht auftreten würden.

Dank dieses Projektes konnte ich mich sehr stark mit Kubernetes und ein paar Unterprojekten wie PostgreSQL-Operator und Metallb auseinandersetzen. Ich habe viel wissen erlangt welches für mein zukünftiges Berufsleben sehr nützlich sein wird. Dies wurde während diesem Projekt umfassend theoretisch, gepaart von praktischer Übung, umgesetzt. Meine Kenntnisse gehen nun weit über die Grundlagen hinweg, sodass ich jetzt perfekt für zukünftige Projekte gewappnet bin. Am wichtigsten ist aber, dass mir bis auf die Dokumentation das gesamte Projekt sehr viel Spaß gemacht hat. Daraus lässt sich schließen, dass ich den richtigen Beruf gewählt habe.

Abbildungsverzeichnis

Literaturverzeichnis

1: TechWorld with Nana, Kubernetes Tutorial for Beginners [FULL COURSE in 4 Hours], Abgerufen am: 25.11.2024, https://www.youtube.com/watch?v=X48VuDVv0do
2: Blais, Kubernetes : sharing of /dev/kmsg with the container, Abgerufen am: 18.11.2024, https://forum.proxmox.com/threads/kubernetes-sharing-of-dev-kmsg-with-the-container.61622/post-291514
3: Longhorn Authors, Quick Installation, Abgerufen am: 16.12.2024, https://longhorn.io/docs/1.7.2/deploy/install/#installing-open-iscsi
4: pgagarinov, Iscsi im LXC container, Abgerufen am: 16.12.2024, https://forum.proxmox.com/threads/iscsi-im-lxc-container.114598/post-509558

9. Anhang

9.1. Glossar

LXC (Linux Containers)	Virtualisierungstechnik auf Betriebssystemebene, welche isolierte Linux-Systeme auf einem Host ausführt.
KVM (Kernel-basierte Virtual Machine)	Open-Source-Lösung zum Ausführen von virtuellen Maschinen auf der Hardware mithilfe des Linux-Kernels.
K8s (Kubernetes)	Open-Source System zur Verwaltung von Container-Anwendungen. Ermöglicht sehr hohe Skalierung und High Availability.
K3s	Minimalistische, leichtgewichtige Distribution von Kubernetes.
Helm	Ein Paketmanager für Kubernetes.
Helm-Charts	Vorlagen aus dem Internet ermöglichen schnelle Bereitstellung von Anwendungen.
Metallb	Load-Balancer-Lösung für ein K8s Cluster, welches Services mit einer externen IP versorgt.
Patroni	Eine Vorlage für Hochverfügbare PostgreSQL Lösungen.

Cronjobs	Cronjobs sind wiederkehrende Aufgaben, die automatisiert auf unixartigen Betriebssystemen ausgeführt werden.
Multithreading	Gleichzeitiges Arbeiten mehrerer Threads innerhalb eines Prozesses, um Aufgaben Parallel abzuarbeiten.
iSCSI	Ein Verfahren, welches die Nutzung des SCSI-Protokolls über TCP ermöglicht.
SCSI	Eine Familie von standardisierten Protokollen und Schnittstellen für die Verbindung und Datenübertragung zwischen Peripheriegeräten und Computern.

9.2. nginx.conf

```
load_module /usr/lib/nginx/modules/ngx_stream_module.so;
events {}

stream {
  upstream k3s_servers {
    server gl2:6443;
    server gl3:6443;
  }

  server {
    listen 6443;
    proxy_pass k3s_servers;
  }
}
```

9.3. wichtige-daten-3.yaml

```
kind: "postgresql"
apiVersion: "acid.zalan.do/v1"

metadata:
  name: "wichtige-daten-3"
  namespace: "default"
  labels:
    team: acid

spec:
  teamId: "acid"
  postgresql:
    version: "16"
  numberOfInstances: 3
  enableMasterLoadBalancer: true
  enableReplicaLoadBalancer: true
  enableConnectionPooler: true
  enableReplicaConnectionPooler: true
  enableMasterPoolerLoadBalancer: true
  enableReplicaPoolerLoadBalancer: true
  maintenanceWindows:
```

```
  volume:
    size: "10Gi"
  users:
    bauer: []
  databases:
    firmendaten: bauer
  allowedSourceRanges:
  - 0.0.0.0/0
  resources:
    requests:
      cpu: 100m
      memory: 100Mi
    limits:
      cpu: 500m
      memory: 500Mi
```

9.4. metallb-config.py

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
        - 10.0.145.130-10.0.145.149
```

9.5. db-test-multi.py

```
import time
import os
import psycopg2
from psycopg2 import OperationalError
from faker import Faker
import random
import concurrent.futures

# Initialize Faker
fake = Faker()

def connect_db():
    while True:
        try:
            print("Connecting...")
            conn = psycopg2.connect(
                host="10.0.145.135",
                port=5432,
                database="firmendaten",
                user="postgres",

password="lILoRG64I5rbp8iJcUxUXQogYOl6npkXk8O8oi3uaWqEiuhPZgOxq3UzaAp5CdEo",
                sslmode='require',
                connect_timeout=2
```

```
            )
            print("Connected to the database.")
            return conn
        except OperationalError as e:
            print(f"\033[91mConnection failed: '{str(e).strip()}' Retrying...\033[0m")
            time.sleep(0.5)

def read_random_id(conn, thread_id):
    try:
        with conn.cursor() as cursor:
            cursor.execute("SELECT id, name, num_col FROM testdaten ORDER BY RANDOM() LIMIT
1;")
            result = cursor.fetchone()
            if result:
                print(f"\033[92mThread-{thread_id:3d} | Random ID: {result[0]:3d}, Name: {result[1]:<20},
Num_col: {result[2]:3d}\033[0m")
            else:
                print("No records found.")
    except Exception as e:
        print(f"\033[91mError during query execution: '{str(e).strip()}'\033[0m")

def read_loop(thread_id):
    conn = connect_db()
    print(f"Thread-{thread_id} started.")
    try:
        while True:
            if conn is None or conn.closed:
                conn = connect_db()
            read_random_id(conn, thread_id)
            time.sleep(0.5)
    except KeyboardInterrupt:
        print(f"Thread-{thread_id} exiting...")
    finally:
        if conn:
            conn.close()

if __name__ == "__main__":
    # Run 30 concurrent read loops
    with concurrent.futures.ThreadPoolExecutor(max_workers=600) as executor:
        # Launch 30 threads
        futures = [executor.submit(read_loop, thread_id) for thread_id in range(300)]

        try:
            # Wait for all threads to complete
            concurrent.futures.wait(futures)
        except KeyboardInterrupt:
            print("Main program exiting...")
```

9.6. postgres-backup.yaml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: postgres-backup
spec:
  schedule: "0 2 * * *"  # Daily at 2 AM
  jobTemplate:
```

```
8   spec:
9     template:
10      spec:
11        containers:
12        - name: backup
13          image: postgres:16
14          volumeMounts:
15          - name: smb-backup
16            mountPath: /mnt/smb-backup
17          - name: backup-script
18            mountPath: /scripts
19          command: ["/bin/bash", "/scripts/backup.sh"]
20        restartPolicy: OnFailure
21        volumes:
22        - name: smb-backup
23          persistentVolumeClaim:
24            claimName: smb-backup-pvc
25        - name: backup-script
26          configMap:
27            name: postgres-backup-script
```

Heinrich Thaler

www.it-thaler.de

info@it-thaler.de

19.01.2025