Pedagogická fakulta
Faculty of Education

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v ČB

Pedagogická fakulta

Katedra informatiky

Využití technologií Kubernetes a Docker pro běh webové aplikace Bobřík informatiky

The use of Kubernetes and Docker technologies to run the web-based application Bebras Challenge

Bakalářská práce

Vypracoval: Tomáš Marek

Vedoucí práce: Mgr. Václav Šimandl, Ph.D.

České Budějovice 2023

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Pedagogická fakulta

Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Tomáš MAREK**
Osobní číslo:	**P200038**
Studijní program:	**B7507 Specializace v pedagogice**
Studijní obor:	**Informační technologie a e-learning**
Téma práce:	**Využití technologií Kubernetes a Docker pro běh webové aplikace Bobřík informatiky**
Zadávající katedra:	**Katedra informatiky**

Zásady pro vypracování

Cílem této bakalářské práce je návrh, realizace a evaluace prostředí založeného na technologiích Kubernetes a Docker určeného pro běh webové aplikace Bobřík informatiky. V teoretické části student popíše technologie Kubertenes a Docker, jejich výhody a nevýhody a také vzájemné rozdíly. V rámci praktické části práce student provede analýzu aktuálního prostředí webové aplikace Bobřík informatiky včetně komponent infrastruktury, jako je například databáze. Následně vytvoří návrh nové infrastruktury řešené pomocí technologií Kubernetes a Docker, na nějž naváže realizací daného řešení s využitím serverové kapacity odpovídající současnému produkčnímu serveru ibobr.cz. Řešení založené na nových technologiích bude spuštěno souběžně s aktuálním produkčním řešením, aby bylo možné provést výkonnostní testy proti oběma prostředím. Výkonnostní testy student realizuje pomocí Open source software gatlig, který umožní provádět plošné zátěžové testy, simulující podle definovaného scénáře chování reálných uživatelů využívajících webovou aplikaci Bobřík informatiky. Zjištění vzešlá z těchto výkonnostních testů student analyzuje a učiní závěry o výhodách a nevýhodách využití technologií Kubertenes a Docker pro běh webové aplikace Bobřík informatiky.

Rozsah pracovní zprávy:	**40**
Rozsah grafických prací:	**CD ROM**
Forma zpracování bakalářské práce:	**tištěná**

Seznam doporučené literatury:

1. Ibryam, B. Kubernetes Patterns: Reusable Elements for Designing Cloud Native Applications. Sebastopol, CA: ? O'Reilly Media, 2019. ISBN 978-1492050285.
2. Leszko, R. Continuous Delivery with Docker and Jenkins. Birmingham: ? Packt Publishing, 2017. ISBN 978-1787125230.
3. Matthias, K., Kane, S.P. Docker: Up & Running: Shipping Reliable Containers in Production. 2. vydání. Sebastopol, CA: ? O'Reilly Media, 2018. ISBN 978-1492036739.
4. Moore, J.D. Kubernetes: The Complete Guide To Master Kubernetes. 2019. ISBN ? 978-1096165774.
5. Rajput, D. Hands-On Microservices – Monitoring and Testing: A performance engineer's guide to the continuous testing and monitoring of microservices. Birmingham: ? Packt Publishing, 2018. ISBN ? 978-1789133608.

Vedoucí bakalářské práce: **Mgr. Václav Šimandl, Ph.D.**
Katedra informatiky

Datum zadání bakalářské práce: **11. dubna 2022**
Termín odevzdání bakalářské práce: **30. dubna 2023**

L.S.

doc. RNDr. Helena Koldová, Ph.D.
děkanka

doc. PaedDr. Jiří Vaníček, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 11. dubna 2022

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 17. dubna 2023.

Tomáš Marek

Abstrakt

Cílem bakalářské práce je provést porovnání technologií pro provoz soutěžního webového systému Bobřík informatiky, jak z pohledu uživatele (rychlost odezvy aplikace), tak z pohledu provozu aplikace jako takové (náklady na provoz, dynamické vs statické alokování zdrojů). V teoretické části je uvedena základní terminologie a architektura nově použité technologie Kubernetes a Docker, následně pak v praktické části je analyzováno současné řešení, a navrženo řešení pomocí Kubernetes a Docker, které je implementováno na nový server a následně jsou provedeny srovnávací testy oproti stávajícímu řešení včetně automatického nasazení aplikace z repozitáře Gitlab a následně je doporučeno řešení infrastruktury pro další ročník soutěže, které vychází z provedených testů. Práce by mohla sloužit i jako návod pro začínající uživatele Kubernetes, nebo pro administrátory implementující aplikaci v této technologii.

Klíčová slova

Kubernetes, Docker, PHP, Ceph, MySQL, Gatling, zátěžové testy, výkonostní testy

Abstract

The aim of the bachelor thesis is to compare technologies for the operation of the competitive web system Bebras contest, both from the user's point of view (application response speed) and from the point of view of the application operation as such (operation costs, dynamic vs. static resource allocation). In the theoretical part the basic terminology and architecture of the newly used Kubernetes and Docker technology is presented, then in the practical part the current solution is analyzed and a solution using Kubernetes and Docker is proposed and implemented on a new server, followed by comparative tests against the existing solution, including automatic deployment of the application from the Gitlab repository, and then an infrastructure solution is recommended for the next year of the contest based on the tests performed. The work could also serve as a guide for novice Kubernetes users or for administrators implementing an application in this technology.

Keywords

Kubernetes, Docker, PHP, Ceph, MySQL, Gatling, load testing, performance testing

Poděkování

Děkuji Mgr. Václavovi Šimandlovi, Ph.D. za vedení diplomové práce, odborný pohled, cenné podněty a připomínky, které v rámci vedení poskytoval. Dále děkuji i rodině za její podporu v průběhu psaní práce.

Obsah

1 Úvod

1.1 Cíl práce

Cílem bakalářské práce je provést porovnání technologií pro provoz soutěžního webového systému Bobřík informatiky, jak z pohledu uživatele (rychlá odezva aplikace), tak z pohledu provozu aplikace jako takové (náklady na provoz, dynamické vs statické alokování zdrojů). V teoretické části bude uvedena základní terminologie a architektura nově použité technologie Kubernetes a Docker, následně pak v praktické části bude analyzováno současné řešení, bude navrženo řešení pomocí Kubernetes a Docker, které bude implementováno na nový server a následně budou provedeny srovnávací testy oproti stávajícímu řešení včetně automatického nasazení aplikace z repozitáře Gitlab a následně doporučeno řešení infrastruktury pro další ročník soutěže, které bude vycházet z provedených testů. Práce by mohla sloužit i jako návod pro začínající uživatele Kubernetes, nebo pro administrátory implementující aplikaci v této technologii.

1.2 Metoda práce

V první fázi bude připraveno nové prostředí postavené na fyzickém serveru, který odpovídá aktuálnímu produkčnímu serveru aplikace Bobřík informatiky. Na tomto serveru bude za pomocí QEMU a Proxmoxu, jakožto virtualizačního Hypervizora vytvořeno několik oddělených virtuálních strojů, na které bude nainstalován operační systém Debian. Následně za pomoci automatizačního nástroje od společnosti RedHat, jménem Ansible a nad ním postaveného OpenSource projektu Kubespray provedu instalaci orchestračního nástroje Kubernetes. Jakmile bude orchestrační nástroj nainstalován a funkční, bude potřeba připravit vhodnou infrastrukturu uvnitř nového clusteru. Pro účely této práce budu využívat jako vstupní proxy aplikaci Nginx, která je rozšířena a upravena přímo pro Kubernetes. Jako další aplikaci budeme instalovat cert-manager, který uvnitř clusteru automaticky spravuje certifikáty a také je pří-

padně automaticky obnovuje podle potřeby. Tato komponenta není nutná a je její funkci možné suplovat manuálním zásahem. Dále je možné cluster rozšířit o komponenty jako fluentbit nebo prometheus, které se využívají k zajištění monitorování clusteru, nebo logování clusteru i aplikací.

Jakmile bude cluster připraven, přijde čas na přípravu procesu nasazení aplikace Bobřík informatiky do nově vytvořeného clusteru. Pro tento účel budeme k nasazení využívat OpenSource balíčkový nástroj nazvaný Helm. Tento nástroj umožňuje nadefinovat šablony, které se následně aplikují do cílového clusteru. Jelikož aktuálně je aplikace částečně do Dockeru převedena, nebudu zde popisovat vytvoření Docker obrazu, ale pouze provedu lehké úpravy pro provoz v tomto orchestračním nástroji. Následně využijeme funkcionality nástroje GitLab a to CI/CD, které nám umožní dynamicky sestavit nový image a pak nasadit za pomoci nástroje Helm na produkci. Až bude aplikace nasazena v orchestračním nástroji Kubernetes a ve stejné verzi jako na aktuální produkci u poskytovatele Web4U, je možné přejít k další fázi práce, a to k porovnání starého a nového prostředí. Na toto porovnání jsem se rozhodl využít OpenSource nástroj Gatling, který umožňuje za pomoci předpřipravených scénářů provádět automatizované zátěžové testy webových aplikací. Aby bylo možné mít relevantní výsledek, bude potřeba vytvořit několik testovacích scénářů s různým zaměřením (JavaScript, Složitější PHP operace, Přihlášení/Registrace a další..).

2 Teoretická část

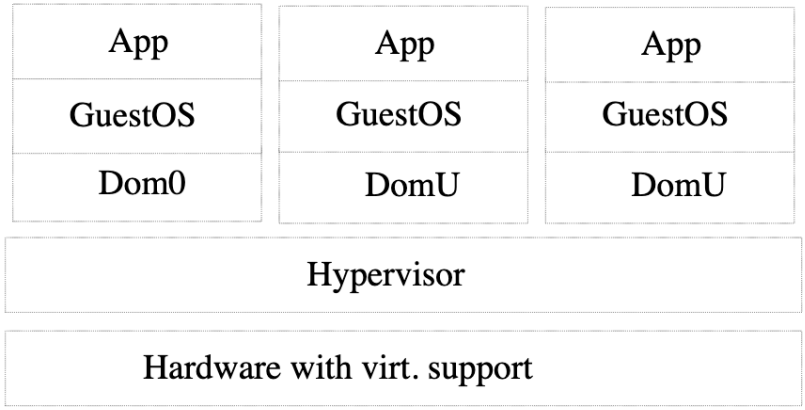
2.1 Virtualizace

Virtualizace je proces, který umožňuje vytvoření neomezeného množství virtuálních počítačů s rozdílnými operačními systémy na jednom fyzickém počítači. V závislosti na druhu virtualizační technologie rozlišujeme, jak daná virtualizace přistupuje k fyzickému vybavení počítače (CPU, RAM). V dnešní době se virtualizace používá stále častěji, hlavně díky velkému rozmachu aplikačních kontejnerů (blíže bude popsáno v sekci kontejnerová virtualizace) a také potřebě snižovat náklady na provoz v datových centrech, kdy společnosti přesouvají většinu fyzických serverů do virtuální podoby [3].

2.1.1 Plná virtualizace (nativní virtualizace)

Plná virtualizace (jinak řečeno nativní virtualizace) se označuje jako Hardwarová (dále jen HW) virtualizace, která jak nám název napovídá, musí být podporována již na samotném HW. Následně při vytvoření virtuálního stroje dochází k alokaci fyzických HW zdrojů (CPU, RAM), což znamená, že je možné vytvářet virtuální stroje pouze dokud máme dostatek dostupných zdrojů. Dále je zde také důležité zmínit, že nativní virtualizace díky přístupu k HW je závislá na architektuře procesoru. Pokud by Hypervisor [1] měl na stroji hosta například procesor od společnosti AMD, tak by nebylo možné spustit virtuální stroj podporující pouze procesor Intel. Celkovou správu virtuálních strojů zajištuje právě zmíněný Hypervisor, jak můžeme pozorovat na obrázku 1. Také je zde dobře viditelné, že virtuální stroje jsou vzájemně izolované [2] [1].

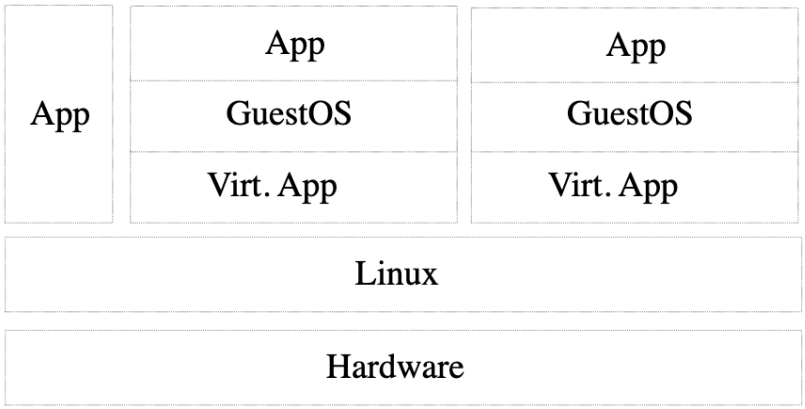
[1] Hypervisor, také označovaný jako VMM nebo monitor virtuálního počítače, je program, který slouží k vytváření a provozování virtuálních počítačů (VM).



Obrázek 1: Princip plné virtualizace [4]

2.1.2 Emulace (simulace)

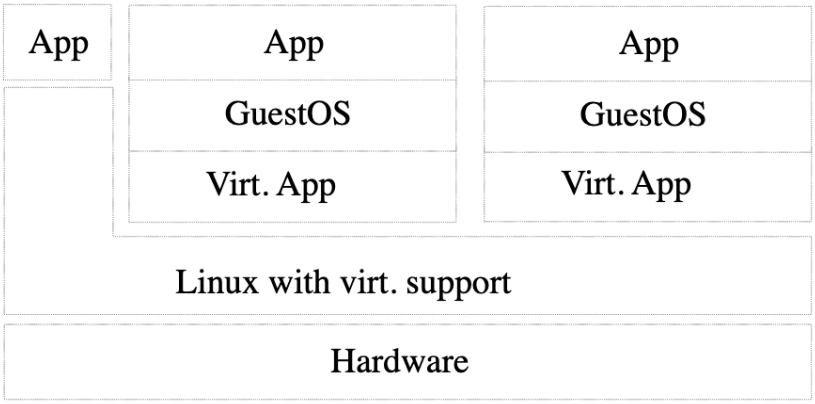
Jako emulace (jinak řečeno také simulace) se označuje druh virtualizace, který je velmi podobný nativní virtualizaci. Taktéž umožňuje provádět plnou izolaci hostovaného OS a to ovšem s jednou výjimkou. Jak jsem již zmínil při nativní virtualizaci sdílíme fyzické vybavení počítače a zároveň nás toto limituje, protože jsme například omezení architekturou procesoru. Emulace tento problém řeší, i když na úkor výkonu virtualizovaného stroje. Emulátor totiž vytváří pro hostovaný OS iluzi například architektury procesoru nebo počtu vláken a následně to překládá na fyzické HW vybavení. Ovšem z pohledu uživatele využívající virtuální stroj, je vidět pouze emulovaný HW. Toto řešení se například hodí, pokud na amd64 architektuře potřebujeme spustit arm operační systém, nebo opačně. Jak můžeme pozorovat na obrázku 2, tak je možné na serveru provádějícím simulaci pustit také jakoukoliv aplikaci bez nutnosti spouštění emulace [1].



Obrázek 2: Princip emulace [4]

2.1.3 Paravirtulizace

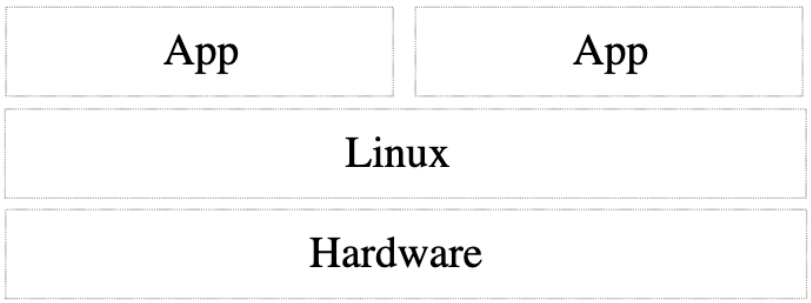
Paravirtualizace je jedním z dalších typů virtualizace, který se ovšem liší od již zmíněných typů. Tento typ virtualizace vyžaduje při instalaci mít již upravený operační systém tak, aby zvládl využívat tzv. Hypercalls, které jsou optimalizovány pro rychlý a efektivní přístup k fyzickým prostředkům. Hypercalls umožňují komunikaci mezi operačním systémem a hypervizorem a poskytují operačnímu systému přístup k řízení využití prostředků. Tyto instrukce jsou následně nahrazovány operačním systémem a tím se zajišťuje rychlejší přístup k prostředkům. Platí zde také stejně jako u nativní virtualizace, potřeba aktivní podpory na úrovni hardwarového vybavení počítače. Jak můžeme pozorovat na obrázku 3, tak je zde opět znázorněna možnost spuštění aplikace přímo na fyzickém hypervisoru, stejně jako izolace celého operačního systému [1].



Obrázek 3: Princip paravirtulizace [4]

2.1.4 Virtualizace na úrovni OS (kontejnerová virtualizace)

Virtualizace na úrovni OS (dnes známá také jako kontejnerová virtualizace). Tato virtualizace se úplně liší od předchozích typů virtualizace a to proto, že nevytváří virtuální stroje, jak je známe. Tento typ virtualizace nabízí izolaci určité aplikace do tzv. kontejneru, který obsahuje pouze binární soubory té zmíněné aplikace a potřebné základní knihovny. Ostatní potřebné komponenty se vždy berou z hosta, protože aplikační kontejner nikdy neobsahuje vlastní Linuxové nebo jiné jádro. Vždy sdílí dané jádro s počítačem/virtuálním strojem, na kterém je provozován. Kontejner jako takový nemá bez potřebných povolení přístup k fyzickému vybavení. Jak jsem již zmínil je možné tyto kontejnery spustit v tzv. privilegovaném módu, který tento přístup umožní. Toto se může hodit například, pokud máme kontejner, který obsahuje VPN server a potřebuje vytvořit na síťové kartě hosta virtuální most. Jak jsem již zmínil, tak je také viditelné na obrázku 4, kde jsou nad operačním systémem hosta provozovány rovnou aplikační kontejnery. Příkladem virtualizace kontejnerů je dnes hojně používaný open source software Docker, nebo například Podman, který sdílí společný formát kontejneru, tzv. OCI, který bude popsán v další části [1].



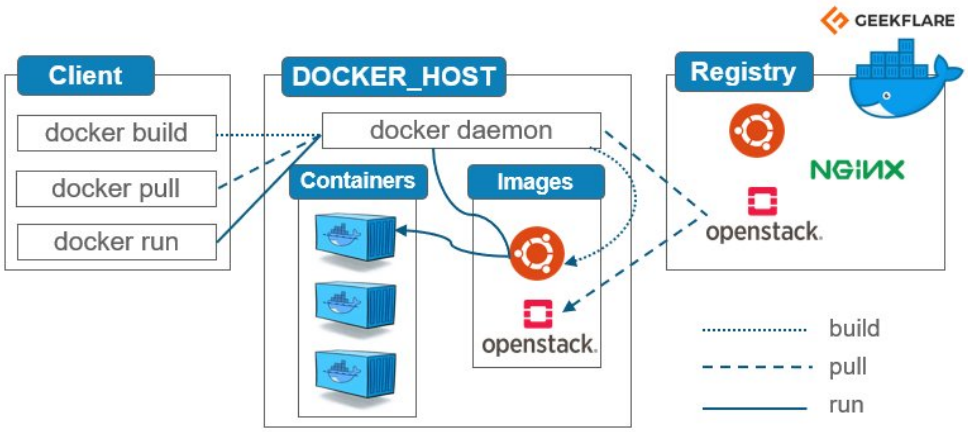
Obrázek 4: Princip Virtualizace na úrovni OS [4]

2.2 Docker

Docker je open-source platforma, která slouží k vývoji, nasazení a spouštění aplikací. Docker dokáže oddělit aplikace od infrastruktury, což se provádí pomocí virtualizace prostředí a nástrojů, které umožňují snadnou správu a nasazení aplikací. Docker používá kontejnery, které využívají podporu virtualizace v jádře a poskytují menší, rychlejší a přenositelnější alternativu k nezávislým kontejnerům z OpenVZ, které jsou již nějakou dobu zastaralé. Aplikace v Dockeru sdílejí jádro s hostitelským počítačem, což přináší další výhody díky obalování základní virtualizace jádra [6].

2.2.1 Architektura Dockeru

Na obrázku 5 je možné pozorovat základní architekturu Dockeru a jeho jednotlivé komponenty, které budou následně více rozebrány v následujících sekcích.



Obrázek 5: Architektura dockeru [8]

2.2.2 Docker Engine

Docker Engine je klíčová komponenta v rámci celého Docker systému. Pracuje na principu klient-server a je instalován přímo na hostitelském stroji. Zde jsou komponenty, ze ktrých se skládá.

- "Server s dlouhodobě běžícím procesem démona **dockerd**.
- API, které určuje rozhraní, přes která mohou programy komunikovat s démonem Docker a vydávat mu pokyny.
- Klientské rozhraní příkazového řádku (CLI)."[8]

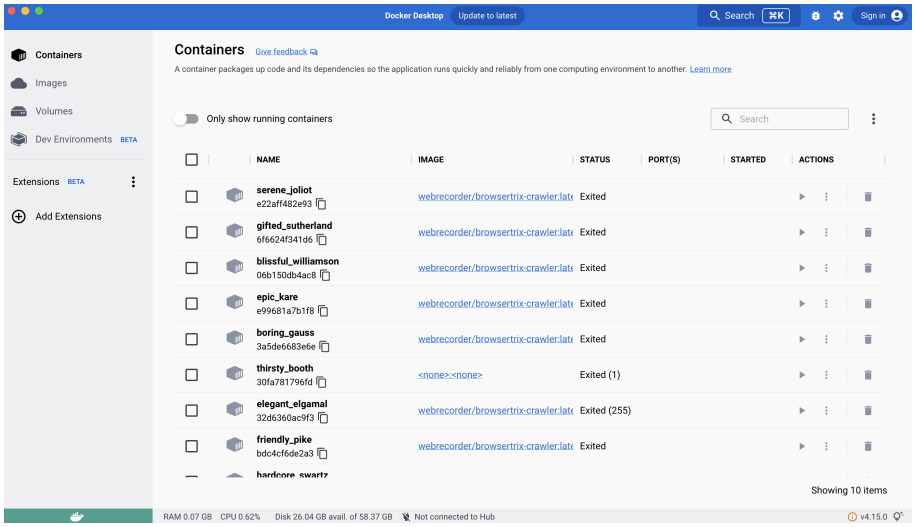
Docker API se používá k ovládání nebo interakci s démonem Docker, prostřednictvím skriptů nebo přímých příkazů CLI. [6].

2.2.3 Démon

Démon Docker (dockerd), jedná se o démona, který poslouchá a čeká na požadavky, které uživatel za pomocí Docker Klienta vyšle ke zpracování. Tento démon se stará o veškeré procesy v rámci docker, mezi které řadíme například stahování docker image, nebo například vytvoření nového kontejneru, nebo vytvoření nového volume (neboli svazku).[6].

2.2.4 Klient

Klient Docker je primární způsob, jak komunikovat s Docker API. Při použití příkazů, jako je například docker run, nebo docker ps. Klient tyto příkazy odešle do docker démonu pomocí již zmíněného API požadavku. Aktuálně existuje velké množství CLI i GUI, které využívají tohoto klienta pro správu Dockeru. Jedním z nejpoužívanějších je aktuálně oficiální Docker GUI, které umožňuje uživatelsky přívětivou správu Docker objektů. Prostředí je vyobrazeno níže na obrázku 6 [6].



Obrázek 6: Docker GUI

2.2.5 Docker objekty

Docker image neboli obraz dockeru.

Docker image je klíčovým prvkem v platformě Docker pro provoz kontejnerových aplikací. Jakmile je docker image sestavený, není již možné v něm provádět jakékoliv úpravy. Jedná se o seznam instrukcí pro vytvoření image, ze kterého se pak spouští kontejner.

K vytvoření vlastního image je třeba použít soubor instrukcí, který se nazývá Dockerfile. Tento soubor definuje jednotlivé instrukce, které vedou k sestavení samotného docker image. Nový image může být vytvořen na základě

již existujícího image, což umožňuje jeho přizpůsobení specifickým požadavkům. Pokud je upraven Dockerfile existujícího image, budou znovu sestaveny pouze změněné kroky. Po sestavení daného image bude dostupný pouze na zařízení, které toto sestavení provedlo. Pokud budeme chtít tento image distribuovat mezi lidi, bude potřeba tento image nahrát na nějaký veřejný docker repozitář. Mezi nejznámější repozitář patří **hub.docker.com** [6].

Na obrázku 7 je zobrazen příklad jednoduchého **Dockerfile**, který jak je zobrazeno v horní části souboru, je z veřejně dostupného image percona DB a je rozšířen o další instrukce.

```
# --------------------------------
# Build argumnets | configuration
# --------------------------------
# Specifices, which base image will be used
ARG DB_IMAGE=percona:5.7

# --------------------------------
# Prepare customized db image
# --------------------------------
# Package predefined databse state, if  & allow customization via .sh script
FROM $DB_IMAGE

USER root

RUN chmod -R 777 /tmp

USER mysql

# Let maraidb entrypoint handle .sh .sql .sql.gz database preparation
COPY docker/services/db/docker-entrypoint-initdb.d /docker-entrypoint-initdb.d

COPY docker/services/db/my.cnf.d /etc/my.cnf.d
```

Obrázek 7: Modifikovaný image pro percona MySQL

Docker container neboli docker kontejner. Jakmile budeme chtít spustit docker image, vytvoříme tzv. docker container. Tento docker container je po vytvoření možné ovládat právě pomocí Docker klienta (Docker API), přes kterého ho můžeme zastavit, restartovat, v případě selhání opět spustit a také samozřejmě odstranit. Níže je zobrazený příklad, který spustí jednoduchý databázový kontejner vycházející z odštěpené verze MySQL a to **mariadb**[2]. [7]

[2]Klíčové slovo latest označuje, že při spuštění kontejneru se má použít poslední dostupná verze v docker registru

```
docker run --detach --name db --env MARIADB_USER=example-user
↪  --env MARIADB_PASSWORD=my_cool_secret mariadb:latest
```

Příkaz v příkladu spustí kontejner obsahující MariaDB databázi, které je možné skrze proměnou prostředí předat volitelný argument MARIADB_USER a také MARIADB_PASSWORD, které v rámci spuštění kontejneru inicializují tohoto uživatele do databáze.

Network neboli síťování. Docker také nabízí svým uživatelům možnost vytvářet virtuální sítě, do kterých následně umožňuje připojovat vytvořené kontejnery. Pokud není explicitně definováno docker použije síť, kde nese název default pro všechny spuštěné kontejnery. Tato možnost síťování přímo v rámci docker nám umožňuje vytvořit na jednom stroji několik síťově izolovaných skupin kontejneru, které budou ve vlastních sítích a vzájemně se nebudou ovlivňovat. [6].

Volume neboli svazky. Jakmile se vytvoří nový kontejner, jsou perzistentní data uchována pouze po dobu životnosti kontejneru, jakmile se kontejner restartuje, nebo smaže, tak dojde také k odstranění veškerých dat, které za dobu běhu vygeneroval. Pokud z aplikačního kontejneru potřebujeme některé informace zachovat i po restartu, jsou zde takzvané Docker volumes. Tyto objekty v základu reprezentují nějakou složku na disku, kam se při použití správného přepínače namapuje specifická složka v kontejneru. Na příkladu níže je ukázáno, jak do kontejneru devtest namapovat vytvořený volume objekt s názvem **vol1**. Tento objekt se namapuje v rámci kontejneru na pozici /**app** [6].

```
docker run -d --name devtest -v vol1:/app nginx:latest
```

2.2.6 Docker registry

Docker Registry je distribuovaný systém pro ukládání a distribuci Docker obrazů s unikátními identifikátory. Každý obraz může mít několik verzí, z nichž každá má své vlastní značky. Registry Dockeru jsou rozděleny do úložišť Docker, kde jsou uloženy všechny verze modifikací obrazu. Uživatelé mohou vy-

užívat registry Dockeru k načítání obrazů lokálně a k nahrávání nových obrazů do registru s odpovídajícími přístupovými právy. Docker Registry je serverová aplikace, která ukládá a distribuuje Docker obrazy. Je bezstavová a umožňuje extrémní škálování [10].

2.2.7 Open Container Initiative (OCI)

Projekt Open Container Initiative (OCI) byl založen v roce 2015 pod záštitou Linux Foundation s cílem standardizovat kontejnerové technologie a formát kontejnerů. Hlavním cílem projektu je poskytnout uživatelům a vývojářům jednotný standard pro kontejnery, který by nezávisel na konkrétní implementaci.

OCI staví na technologiích, jako jsou Docker a App Container (appc), a snaží se vytvořit společný základ pro budoucí vývoj. Projekt definuje specifikace pro formát kontejnerů, runtime a distribuci kontejnerů.

Specifikace formátu kontejneru stanoví, jak by kontejner měl být sestaven a jaké zdroje by měl být schopen využívat. Runtime specifikace definuje, jak kontejner běží a jak je prováděna izolace mezi kontejnery a hostitelským systémem. Specifikace distribuce kontejnerů definuje, jakým způsobem se mají kontejnery přenášet mezi různými systémy.

OCI je podporováno mnoha významnými hráči v oblasti kontejnerových technologií, jako jsou například Docker, Red Hat, Microsoft a Google. Díky této podpoře se OCI stalo široce používaným standardem pro kontejnery a umožnilo lepší interoperabilitu mezi různými kontejnerovými řešeními.

Standardizace kontejnerových technologií pomocí projektu OCI má mnoho výhod. Uživatelé mohou využívat různá řešení kontejnerů, která jsou kompatibilní s daným standardem. Vývojáři mohou pak vytvářet aplikace a kontejnery, které jsou snadno přenosné mezi různými systémy. Tím se snižují náklady a časové nároky na vývoj a nasazení aplikací v prostředí s kontejnery [11].

2.3 Kubernetes

Kubernetes, známý také jako K8s, je nástroj pro řízení kontejnerů, který je určen pro prostředí s více servery. Tento open-source software byl vyvinut společností Google a nyní je spravován Cloud Native Computing Foundation. Kubernetes nabízí možnost správy a orchestrace kontejnerů vytvořených pomocí Dockeru, bez ohledu na to, na kterém serveru nebo cloudu běží.

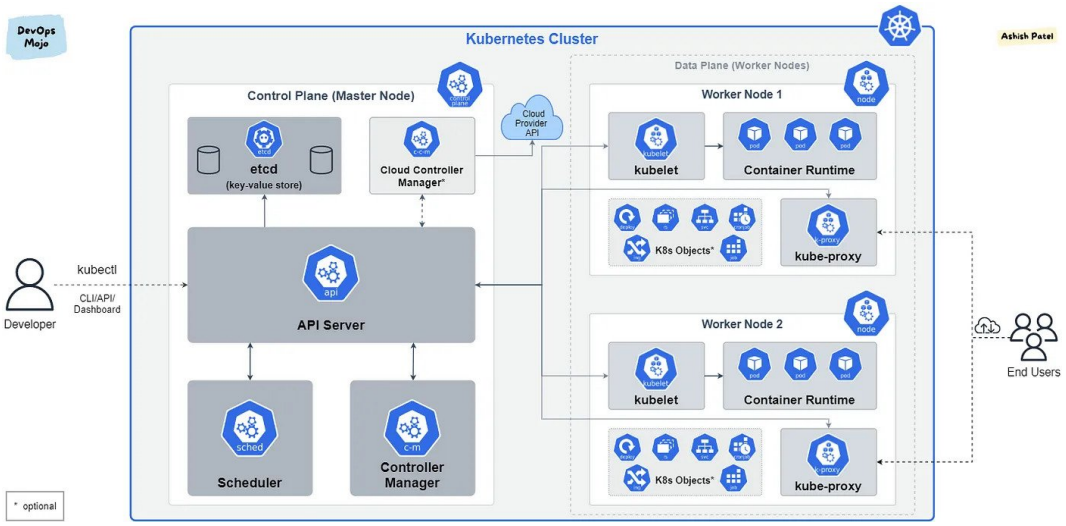
Kubernetes umožňuje spravovat a orchestrovat kontejnery tak, aby byly spolehlivé a škálovatelné. Používá pro to několik konceptů, jako jsou "pody", "deploymenty"a "servisy"(dále používáno jako pojem služby). Pody jsou základními stavebními bloky pro kontejnery a dále se spravují pomocí deploymentů, které umožňují definovat požadovaný stav a strategii aktualizací aplikace. Servisy pak poskytují přístup ke kontejnerům, které jsou běžící na různých serverech, jako by byly na jednom místě.

Kubernetes nabízí mnoho výhod pro správu a provoz kontejnerových aplikací. Například umožňuje horizontální a vertikální škálování aplikací, snižuje náklady a zvyšuje dostupnost pomocí automatického obnovování, detekování chyb a přesouvání kontejnerů. Kubernetes také umožňuje snadné nasazení aplikací do cloudu, což vede ke snížení nákladů a zjednodušení správy.

Kubernetes je open-source software, což umožňuje mnoho komunitních i komerčních podpůrných řešení a služeb. Vzhledem k tomu, že se jedná o standard pro orchestraci kontejnerů, je široce používán mnoha organizacemi a firmami po celém světě [12] [9].

2.3.1 Architektura

Na obrázku 8, můžeme pozorovat všechny důležité komponenty, které Kubernetes obsahuje a následně budou více rozebrány v dalších kapitolách.



Obrázek 8: Architektura nástroje Kubernetes [13]

2.3.2 Cluster

Cluster v Kubernetes představuje skupinu počítačů (nebo také "nodů"), které společně poskytují prostředky pro běh aplikací. Tyto nody mohou být umístěny na různých fyzických serverech, virtuálních serverech nebo dokonce v cloudových prostředích, jako jsou například Google Cloud, Amazon Web Services, nebo Azure.

Cluster v Kubernetes umožňuje efektivní využití zdrojů, jako jsou procesorový čas, paměť a uložiště, a poskytuje tak vyšší výkon a odolnost aplikací proti pádu. Cluster vytváří také možnost automatického škálování aplikací, kdy se podle aktuálního vytížení automaticky spouští nebo ukončují instance aplikací. Každý Cluster v Kubernetes obsahuje jeden nebo více Master nodů (také někdy v terminologii je možné najít jako control-plane), které spravují stav celého Clusteru, včetně spouštění a ukončování aplikací, monitorování stavu nodů a řízení škálování. Master nody také poskytují API rozhraní prostřednictvím kube-apiserveru pro řízení a monitorování Clusteru.

Kromě Master nodů obsahuje Cluster v Kubernetes také jedno nebo více Worker nodů, které běží na fyzických serverech nebo virtuálních strojích a poskytují výpočetní zdroje pro běh aplikací. Každý Worker nod provozuje vlastní

sadu kontejnerů, které tvoří aplikace. Kubernetes automaticky rozmisťuje tyto kontejnery na jednotlivé Worker nody, aby byl co nejefektivněji využit procesorový čas a paměť. Cluster v Kubernetes také poskytuje řadu dalších funkcí, jako je automatické opravování nodu, plánování úloh a správa konfigurace.

2.3.3 Node

Klíčovou součástí clusteru jsou nody master a worker, které mohou být implementovány v podobě fyzického či virtuálního stroje.

Master node

Hlavním řídícím prvkem clusteru je Master node (také někdy v terminologii je možné najít jako control-plane), který provozuje v rámci clusteru několik velice důležitých komponent, jako je API server, jakožto hlavní řídící bod K8S (o kterém bude popsáno níže v sekci kube-apiserver), scheduler, který je potřeba k tomu, aby cluster zvládl korektně plánovat umístění kontejneru na nody, a controller-manager, který řídí veškeré události, mezi které patří kontrola stavu jednotlivých nodů a další aktivity. Navíc může zastávat i roli worker node a být schopen provozovat některé z plánovaných podů, nicméně to není úplně korektní řešení, protože jakýkoliv provoz na worker nodu může negativně ovlivnit důležité kontrolní prvky celého clusteru[14].

Všechna data v rámci Kubernetes clusteru jsou uchovávána v etcd uložišti, což je konzistetní a vysoce dostupná key-values databáze. [15].

Worker node

Jako worker node označujeme server, který v kubernetes clusteru zastává pouze roli hostitele aplikací. Neprovozuje žádné komponenty, které jsou důležité pro chod samotného clusteru (api server, scheduler a další). Výjimku tvoří pouze kubelet (bude pospáno detailněji níže).[14].

Kubelet Je klíčovou komponentou na všech worker nodách. Tato komponenta navazuje spojení s API serverem, který je provozován na master nodu a přijímá úkoly, jako například spusť tento pod nebo smaž nějaký jiný pod. Dále taky kubelet reportuje veškeré informace o dění na daném worker nodu, jako například stavy daných kontejnerů, nebo stav daného nodu. V případě, že by například kubelet oznámil masteru, že není schopen spustit další pod, bude automaticky z clusteru vyřazen. [16].

2.3.4 ETCD

Etcd je distribuované key-value uložiště, které slouží k ukládání konfigurace a stavu aplikací v rámci clusteru. Je to důležitá součást architektury systému Kubernetes, který využívá etcd k ukládání konfigurace, stavu a metadata o

všech objektech v clusteru, jako jsou například kontejnery, uzly, služby nebo úlohy. Etcd je navržen tak, aby byl velmi rychlý, konzistentní a odolný vůči poruchám. Tento nástroj byl vyvinut v Go jazyce a je open-source projekt, který je vyvíjen a udržován komunitou [15].

2.3.5 kube-apiserver

Kube-apiserver je centrální komponentou Kubernetes architektury, která poskytuje rozhraní pro řízení a správu Kubernetes clusteru pomocí REST API. Jeho hlavní funkcí je zpracovávání požadavků na API, které přicházejí z různých zdrojů a následné řízení komunikace s ostatními komponentami, jako jsou etcd uložiště nebo kube-scheduler. Kube-apiserver také zajišťuje bezpečnost a autorizaci pro přístup k API a jeho objektům [17].

2.3.6 Síťování v Kubernetes

Koncept sítě v Kubernetes spočívá v poskytování způsobů pro komunikaci mezi kontejnery v různých uzlech. To se děje prostřednictvím síťových pluginů, které jsou zodpovědné za poskytování síťového rozhraní kontejnerům a komunikaci mezi nimi.

Jedním z příkladů síťových pluginů v Kubernetes je Flannel. Tento plugin vytváří virtuální síťovou vrstvu, která umožňuje komunikaci mezi uzly v clusteru. Každý uzel má přiřazenou IP adresu z této virtuální sítě, která se používá pro komunikaci s jinými uzly. Kontejnery jsou připojeny k této virtuální síti a mohou tak komunikovat s jinými kontejnery v různých uzlech pomocí těchto IP adres.

Dalším příkladem síťového pluginu v Kubernetes je Calico. Tento plugin poskytuje síťování na úrovni třetí vrstvy (Network Layer 3) a podporuje konfiguraci sítě na základě politik. Umožňuje například vytvářet síťová pravidla, která omezují komunikaci mezi kontejnery na základě různých kritérií, jako jsou například IP adresy, porty nebo protokoly [18].

2.3.7 Pod

V Kubernetes je nejmenší jednotkou nasazení aplikace v rámci clusteru označován koncept pod (angl. "pod"). Pod představuje abstrakci jedné či více spuštěných instancí kontejnerů a sdílí s nimi stejný prostor v rámci nodu. Toto řešení umožňuje jednoduše a flexibilně spravovat aplikace v rámci clusteru, např. při škálování nebo aktualizaci aplikace. Pod je také zodpovědný za přidělování IP adres a portů pro jednotlivé kontejnery. Níže přikládám příklad definice objektu Podu [19].

```
"apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80"
```

2.3.8 ReplicaSet

ReplicaSet v Kubernetes je nástroj pro řízení počtu replik aplikací běžících v Kubernetes clusteru. Jeho úkolem je zajistit, aby byl vždy dosažen požadovaný počet replik aplikace a při výpadku některé z nich ji znovu spustit. ReplicaSet umožňuje také aktualizaci aplikací bez výpadků, přičemž zajistí postupné nahrazování starých replik novými. Je třeba mít na paměti, že ReplicaSet není zodpovědný za správu stavu aplikace, ale pouze za správný počet replik [20].

2.3.9 Deployment

Deployment v Kubernetes slouží k deklarativní správě aplikací v Kubernetes clusteru, přičemž zajistí, aby byl vždy dosažen požadovaný stav aplikace bez ohledu na stav aktuálních replik. Deployment řídí podřízený objekt ReplicaSet, který jsem popsal výše. Níže přikládám příklad definice objektu Deploymentu [21].

```
"apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80"
```

[21]

2.3.10 StatefullSet

StatefulSet je mechanismus pro řízení stavových aplikací v Kubernetes clusteru, který zajišťuje, že repliky aplikace jsou spouštěny a zastavovány v přesně definovaném pořadí a mají přidělené unikátní identifikátory. To je důležité pro aplikace, které ukládají stav, jako jsou databáze. StatefulSet také poskytuje stabilní síťové identity pro jednotlivé repliky aplikace, což umožňuje konzistentní přístup k datům bez ohledu na umístění replik. StatefulSet podporuje aktualizaci aplikací bez výpadků stejně jako Deployment, ale vyžaduje určité úpravy v návrhu aplikace, aby byla odolná vůči změnám topologie sítě a restartům replik. Níže přikládám příklad definice objektu StatefullSetu [22].

```
"apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  minReadySeconds: 10 # by default is 0
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
```

```
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi"
```

[22]

2.3.11 DaemonSet

DaemonSet v Kubernetes je objekt, který zajišťuje spuštění jedné instance aplikace na každém nodu v Kubernetes clusteru. Tímto způsobem lze použít DaemonSet pro různé účely, například pro sběr dat nebo monitorování stavu nodů v clusteru. Tento objekt poskytuje také několik možností pro aktualizaci aplikace bez výpadku služby. Je však důležité mít na paměti, že aplikace musí být navržena tak, aby byla nezávislá na ostatních aplikacích a zdrojích v clusteru, protože každý uzel bude mít vlastní instanci aplikace. Níže přikládám příklad definice objektu DaemonSetu [23].

```
"apiVersion: apps/v1
kind: DaemonSet
```

```
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log"
```

[23]

2.3.12 Service

Služba neboli Service v Kubernetes je objekt, který umožňuje komunikaci mezi aplikacemi v Kubernetes clusteru. Service poskytuje konzistentní způsob pro přístup k aplikacím, i když se repliky aplikace mění, přidávají nebo odebírají.

Service má přidělenou IP adresu a port, který umožňuje přístup k aplikaci. Pokud je v Kubernetes clusteru spuštěno více replik aplikace, Service zajistí, že požadavky jsou rovnoměrně rozděleny mezi tyto repliky. Tím se zajišťuje dostupnost aplikace, jelikož jedna nefunkční replika neovlivní dostupnost aplikace pro ostatní požadavky.

Existuje několik typů Service v Kubernetes, jako například ClusterIP, NodePort, LoadBalancer a ExternalName, které se liší způsobem přístupu k aplikaci a typem použitého rozhraní.

Service také poskytuje možnosti pro zabezpečení přístupu k aplikaci pomocí definování síťových pravidel a dalších bezpečnostních funkcí. Níže přikládám příklad definice objektu Služby (Service) [24].

```
"apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376"
```

[24]

2.3.13 Ingress

Ingress v Kubernetes je objekt, který umožňuje přistupovat k aplikacím z vnější sítě, zpravidla z internetu. Ingress slouží jako vrstva mezi vnější sítí a službami (Services) v Kubernetes clusteru, a umožňuje řídit přístup k různým službám pomocí definice pravidel a směrování provozu. Ingress definuje pravidla pro přístup k jednotlivým službám na základě URL, HTTP metody, či jiných kritérií. Díky tomu lze jednoduše přistupovat k různým službám v clusteru přes jednu IP adresu a port. Ingress také umožňuje použití šifrování pomocí protokolu TLS pro zabezpečení komunikace mezi klientem a Ingressem.

Ingress je často používán jako nástroj pro směrování provozu mezi různými verzemi aplikace, nebo pro distribuci provozu mezi více služeb v clusteru. Dále umožňuje jednoduše řídit přístup k aplikaci z různých zdrojů, jako jsou například IP adresy nebo síťové segmenty.

Ingress se skládá z Ingress Controlleru a Ingress Resource. Ingress Controller je komponenta, která poskytuje Ingress služby a implementuje pravidla pro směrování provozu. Ingress Resource je YAML konfigurační soubor, který definuje pravidla pro směrování provozu a další nastavení pro Ingress Controller. Níže přikládám příklad definice objektu Ingress [25].

```
"apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
```

```
- path: /testpath
  pathType: Prefix
  backend:
    service:
      name: test
      port:
        number: 80"
```

[25]

2.4 Helm

Helm je nástroj pro správu balíčků v Kubernetes, který zjednodušuje proces instalace, aktualizace a odinstalace aplikací v Kubernetes clusteru. Pomocí "charts"(neboli balíčky), které obsahují manifesty, konfigurace, závislosti a další soubory. Helm poskytuje také možnost parametrizace chartu pro jednoduché přizpůsobení aplikace pro různá prostředí a konfigurace. Je to oblíbený nástroj v Kubernetes komunitě, který umožňuje snadnou správu a nasazení aplikací v Kubernetes [26].

2.4.1 Koncept helmu

Helm poskytuje standardizované balíčky (nazývané "charts") obsahující všechny potřebné soubory pro správu aplikací v Kubernetes. Tyto balíčky zahrnují manifesty Kubernetes, konfigurační soubory a další potřebné soubory. Strukturu balíčku helm je možné vidět na obrázku 9. Helm také umožňuje snadnou parametrizaci chartů pro různá prostředí a konfigurace, což usnadňuje nasazení aplikací do Kubernetes clusteru a správu jejich závislostí. Celkově lze říci, že koncept balíčkovacího nástroje Helm usnadňuje proces instalace, aktualizace a odinstalace aplikací v Kubernetes [27].

Obrázek 9: Struktůra Helm chartu [27]

2.4.2 Šablony

Helm umožňuje definovat takzvané template funkce (nazývané "templates function"), které umožňují některé opakované operace provádět skrze pouhé zavolání šablony. Níže je názorně zobrazena funkce šablonování. V klasickém manifestu je definováno pouze klíčové slovo "template"[28].

```
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "mychart.labels" }}
```

Následně se volá funkce, která se nachází v souboru **helpers.tpl**.

```
{{/* Generate basic labels */}}
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
```

2.4.3 Verzování

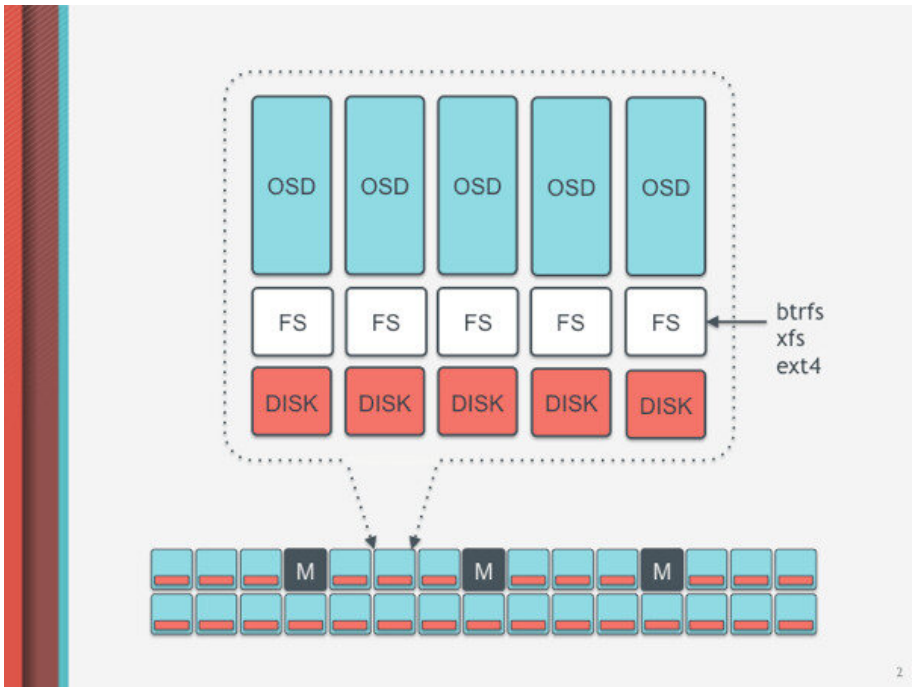
Jakmile provedeme přes nástroj Helm jakoukoliv operaci (install/upgrade), dojde k automatickému vytvoření release verze a s každým dalším nasazením se tato verze automaticky posouvá. Díky tomuto chování je možné v případě jakéhokoliv problému se posouvat jednoduše zpět nebo dopředu v historii nasazení [29].

2.5 Ceph

Ceph je bezplatné softwarově definované uložiště dat (SDS), které implementuje objektové uložiště na clusteru. Je navrženo tak, aby nebylo závislé na jediném bodě, který může selhat a byl odolný vůči chybám (díky replikaci dat RADOS), a to pomocí běžného hardwaru. Je dodavatelsky agnostický a nevyžaduje specifickou hardwarovou podporu. Je sebe opravující díky speciálním démonům sledujícím stav clusteru (ceph-mon), kteří sledují aktuální a selhávající nody clusteru [30].

2.5.1 Architektura

Na obrázku (Obrázek č.10) máme architekturu softwaru Ceph, což je distribuovaný systém pro ukládání dat. Celý systém se skládá z několika komponent, které spolupracují na ukládání a správě dat. Mezi tyto komponenty patří Monitor (Monitor Daemon), Object Storage Daemon (OSD), Metadata Server Daemon (MDS) a klienti.



Obrázek 10: Ceph architektura [30]

2.5.2 OSD

OSD v softwarově definovaném úložišti Ceph je zkratka pro Object Storage Device a zodpovídá za ukládání datových bloků a jejich replikaci pro vysokou dostupnost. Tento koncept umožňuje Ceph být odolný proti selhání v jednom bodě a tolerovat chyby díky replikaci dat pomocí RADOS. K ukládání datových bloků na OSD se využívá btrfs, což umožňuje využít jeho vlastnosti copy-on-write. Pro efektivní využití Ceph se doporučuje oddělit OSD data, OSD journal a OS na samostatné disky a zajistit dostatečné systémové zdroje [30].

2.5.3 RADOS Block Device

Rados Block Device (RBD) je funkce, která umožňuje ukládání blokových dat na objektovém uložišti Ceph. RBD tedy slouží jako rozhraní pro práci s blokovými zařízeními nad Ceph uložištěm. RBD poskytuje distribuované blokové uložiště s možností snapshotů, replikace dat a dynamického přidělování uložiště. Tyto vlastnosti jsou zprostředkovány přes jeho architekturu postavenou na objektovém uložišti Ceph a umožňují aplikacím přístup k ukládání dat jako by to bylo tradiční blokové úložiště [31].

2.6 Ansible

Ansible je open-source nástroj, nebo platforma pro automatizaci používaná pro IT úkoly, jako je správa konfigurace, nasazování aplikací, orchestrace intraservisu a provisioning. Automatizace je v dnešní době klíčová, protože IT prostředí jsou příliš složitá a často musí být rychle škálována. Systémoví administrátoři a vývojáři by nedokázali všechno manuálně zvládnout. Automatizace zjednodušuje složité úkoly, nejenže usnadňuje práci vývojářů, ale umožňuje jim zaměřit pozornost na další úkoly, které přinášejí organizaci hodnotu. Jinými slovy, uvolňuje čas a zvyšuje efektivitu [32].

2.6.1 Playbook

Ansible playbooks jsou jako instrukční manuály pro úkoly. Jsou to jednoduché soubory napsané v YAML, což je zkratka pro "YAML Ain't Markup Language", tedy lidsky čitelný jazyk pro serializaci dat. Playbooks jsou v jádru toho, co dělá Ansible tak populárním, protože popisují úkoly, které mají být provedeny rychle a bez nutnosti, aby uživatel znal nebo si pamatoval určitou syntaxi. Playbooks nejenom, že mohou deklarovat konfigurace, ale mohou také orchestrovat kroky manuálně řazeného úkolu a mohou provádět úkoly současně nebo v různých časech.

Každý playbook je složen z jednoho nebo více "plays"a cílem "play"je ma-

povat skupinu hostitelů na dobře definované role, které jsou reprezentovány úkoly [33].

2.6.2 Kubespray

Kubespray je open-source nástroj pro automatizované nasazení a správu clusteru Kubernetes. Je navržen pro snadnou instalaci a konfiguraci Kubernetes clusteru na širokou škálu infrastruktury, včetně bare metal, virtuálních strojů a cloudových prostředí.

Kubespray používá Ansible jako svůj konfigurační management nástroj, což umožňuje snadné a opakovatelné nasazování clusterů s minimálním úsilím. Jeho architektura umožňuje uživatelům snadno přidávat nebo odebírat nody z clusteru a snadno upravovat konfiguraci Kubernetes a přidávat další komponenty.

Kubespray je dodáván s rozsáhlou knihovnou předdefinovaných konfigurací, které lze použít pro běžné úkoly v Kubernetes, jako jsou například správa nodů, balancování zátěže, instalace síťových pluginů, vytváření a správa přiřazení IP adres, správa uzlů a řešení výpadků.

Kubespray také podporuje konfiguraci a instalaci dalších aplikací a komponent, jako jsou například nástroje pro správu záznamů (logging), monitorování, průběžné doručování (continuous delivery) a další. Vzhledem k tomu, že Kubespray je open-source a poskytuje širokou škálu funkcí a konfigurací, stal se velmi oblíbeným nástrojem mezi komunitami, které používají Kubernetes jako svou platformu pro orchestraci a nasazení aplikací [34].

2.7 Gatling

Gatling je open-source nástroj pro testování výkonu webových aplikací, který umožňuje simulační testování při různých zatíženích a pomáhá identifikovat případné problémy. Jeho základním konceptem je popsat chování uživatele skrze tzv. scénáře a definovat jejich kroky. Gatling poskytuje detailní výstupní

data včetně výkonnosti, chyb a grafického zobrazení. Tento nástroj je napsán v jazyce Scala a spouští se z příkazové řádky nebo z webového rozhraní. Gatling se používá hlavně v prostředí DevOps a automatizace testování aplikací [35].

2.7.1 Koncept

Gatling je založen na architektuře Actor model, což umožňuje efektivní zpracování vysokého počtu uživatelských interakcí s aplikací. Gatling používá jazyk Scala pro psaní testovacích scénářů, což umožňuje vývojářům psát testovací scénáře s použitím moderních programovacích technik. Gatling také poskytuje přehledné a uživatelsky přátelské rozhraní pro zobrazení výsledků testů a pro analýzu statistik výkonu aplikace [35].

Gatling také umožňuje testování různých druhů webových aplikací, včetně aplikací napsaných v různých programovacích jazycích, jako jsou například Java nebo Ruby. Gatling je rovněž schopen simulovat různé typy uživatelských interakcí s aplikací, jako je klikání na odkazy, vyplňování formulářů nebo posílání požadavků na server.

2.7.2 Scénáře

Scénář v nástroji Gatling je soubor kódu napsaného v jazyce Scala, který definuje kroky a chování, které má být testováno při simulaci zátěže.

Každý scénář v Gatlingu začíná definicí protokolu (HTTP, FTP, atd.), který má být použit pro komunikaci s testovaným serverem. Poté se definuje seznam kroků, které se mají provést v rámci simulace. Tyto kroky mohou zahrnovat HTTP požadavky, dotazy do databáze, úpravy proměnných, atd.

Scénář v Gatlingu se skládá ze tří hlavních částí:

- SetUp - v této části se inicializují proměnné a nastavují hodnoty, které jsou potřebné pro běh simulace.
- Scenario - tato část definuje kroky, které mají být provedeny v rámci simulace. Každý krok v této části musí být definován jako funkce, která

vrací objekt typu "ChainBuilder".

- Teardown - v této části se provádí úklid po skončení simulace.

Technologicky je Gatling navržen tak, aby byl výkonný, škálovatelný a snadno použitelný pro testování vysoko-výkonných aplikací. Gatling používá asynchronní vysílání HTTP požadavků, což umožňuje zpracování velkého počtu požadavků za krátký čas. Gatling také umožňuje nastavení zátěže na základě počtu uživatelů, kteří simulaci spouštějí, což umožňuje testovat výkonnost aplikace v prostředí podobném reálnému světu [35].

2.8 Porovnání technologií Docker a Kubernetes

Přestože Docker a Kubernetes jsou navrženy pro podobné účely, existuje mezi nimi několik rozdílů. Docker se zaměřuje na vytváření a správu kontejnerů, což je ideální pro vývojáře, kteří potřebují izolované prostředí pro své aplikace a snadné nasazení do různých prostředí. Na druhé straně Kubernetes je navržen jako nástroj pro řízení a orchestraci kontejnerů, umožňuje správu větších počtů kontejnerů a jejich nasazení do různých prostředí. Kubernetes také umožňuje snadné škálování aplikací a vyvažování zátěže, což pomáhá udržovat vysokou dostupnost a výkon aplikací. Zatímco Docker je často používán jako základní stavební kámen pro aplikace, Kubernetes je často používán pro vytváření a správu komplexních aplikací v cloudu. Celkově vzato, oba nástroje mají svá specifická použití, ale v kombinaci poskytují velmi silný nástroj pro vývoj, nasazení a správu moderních aplikací v cloudu.

3 Praktická část

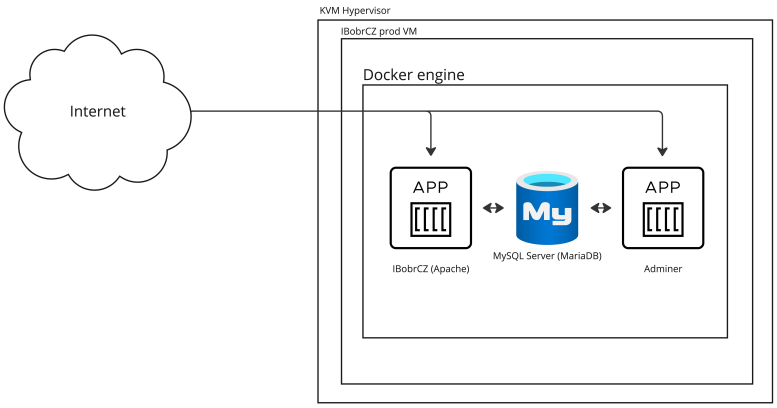
3.1 Analýza současného řešení

Pro potřeby sestavení návrhu nové infrastruktury na technologii Kubernetes je potřeba provést analýzu aktuálního řešení.

Aktuální řešení je fakultě poskytováno společností Web4U s.r.o., která dodává pro potřeby provozu jeden virtualizovaný server, který je vybaven 64GB RAM paměti a 12 virtuálními CPU. Pokud je ovšem potřeba, je možné nechat zdroje navýšit, ale pro naše potřeby budeme počítat se základními zdroji.

Provoz webové aplikace je řešen pomocí webového serveru Apache a několika podpůrnými moduly. Mezi zmíněné moduly patří například php, gd, mysqli, zip a další potřebné moduly. V rámci provozu je spuštěn pouze jeden webový server. Pro ukládání stavových dat se využívá MySQL databázový server, který je provozován na stejném virtuálním serveru jako webový server. Součástí provozovaného PHP frameworku je také mechanismus cache, ale ten je ukládán na souborový systém a nevyužívá žádnou další službu.

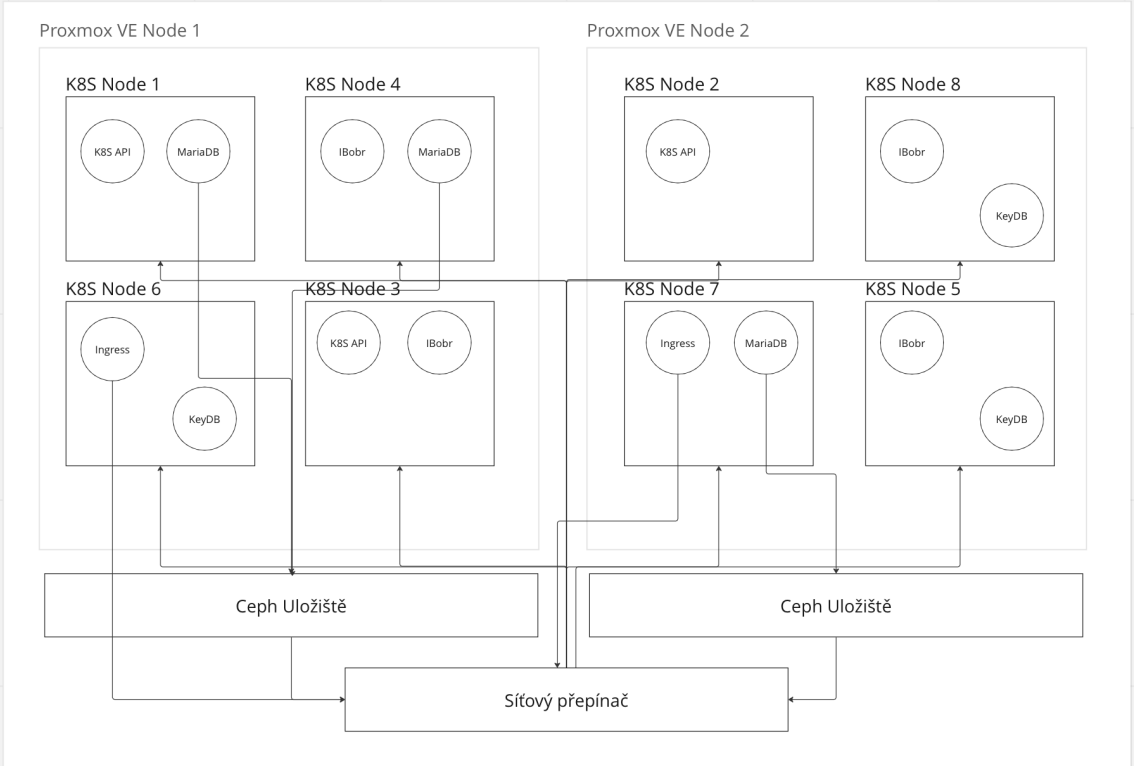
Následné nasazování aplikace je řešeno částečnou kontejnerizací za pomocí docker wrapperu docker compose. V rámci nasazení je potřeba naklonovat na cílový server celý repozitář, který je následně připojen do kontejneru jako volume. Tato architektura neodpovídá konceptu kontejnerizace.



Obrázek 11: Aktuální architektonické řešení

3.2 Návrh nového řešení na technologii Kubernetes

Na základě předchozí analýzy vytvořím návrh takového řešení, které bude odpovídat architektonickému modelu orchestračního systému Kubernetes. Orchestrační systém Kubernetes byl ze své podstaty vytvořen pro případ rychlého škálování a vysoké dostupnosti, proto bude nutné provést designové úpravy aktuálního řešení, již na úrovni Virtualizace (emulace).



Obrázek 12: Nový architektonický návrh

Pro potřeby nového řešení budu využívat dva fyzické servery, které budou vyhrazeny pouze pro tuto práci. Nové řešení bude postaveno na Bare-Metal[3], neboli na fyzických serverech. Také je možné provést nasazení například do Cloudu (AWX, Google Cloud a jiné), kde odpadává práce s fyzickým HW. Jelikož zde provádím srovnání s již použitou architekturou, bude využití fyzického HW ve finálních zátěžových testech více vypovídající, než kdyby byl

[3]Bare-Metal - Reprezentuje technologický pojem, který označuje servery provozované mimo cloudové poskytovatele

použit Cloud. Součet zdrojů, které máme dostupné na dvou fyzických serverech, odpovídá aktuálnímu řešení, které je poskytováno pro běžící instanci soutěže Bobřík informatiky.

Na připravené fyzické servery budu instalovat virtualizační systém Proxmox v clusterovém řešení, na kterém budeme dále implementovat orchestrační nástroj Kubernetes. Než se ovšem pustím do implementace orchestračního nástroje Kubernetes bude potřeba provést instalaci několika podpůrných komponent. Aby bylo možné zachovat plnou dostupnost i v případě selhání jednoho fyzického serveru, bude využívat pro ukládání perzistentní data[4]. Následně již vytvořím samotné virtuální stroje a poté provedu instalaci nástroje Kubernetes.

Před samotným nasazením aplikace Bobřík informatiky budu muset provést kontrolu aplikačního kódu, jestli je možné aplikaci provozovat ve více replikách a v případě potřeby budu muset provést úpravy aplikace tak, aby to možné bylo.

Následně bude nutné provést instalaci infrastrukturních komponent Kubernetes clusteru jako je například Nginx reverzní proxy server nebo například databázový cluster, což bude popsáno v pozdější fázi implementace.

3.3 Realizace nového řešení

Pro potřeby nového řešení mám dva fyzické servery, které zdroji odpovídají konfiguraci aktuálního virtuálního serveru Bobříka informatiky. Použitím dvou serverů chceme simulovat možnost distribuce zátěže přes několik fyzických serveru bez toho, aniž by uživatel poznal, že se nachází na jiném serveru. Pro provoz virtualizace použiji na fyzických serverech systém Proxmox, který využívá emulátor quemu.

[4]Perzistetní data - Označujeme tak data kontejneru nebo aplikací, kterou jsou stavová

3.3.1 Instalace virtualizačního systému Proxmox

Pro instalaci Proxmox ISO download [37]. Pro instalaci systému je několik možností. Je možné provést překopírování daného ISO na flashdisk, nebo je možné použít vzdálený systém IPMI, který budu využívat v našem případě [5].

Dále je instalován Proxmox VE, který nevyžaduje složitější konfiguraci. Po dokončení instalace na prvním serveru, následně opakuji instalační akci také na druhém serveru. Samozřejmě je potřeba při instalaci zvolit korektní IPv4 adresy pro servery, aby se následně bylo možné připojit do systému webové správy.

Po dokončení instalace druhého serveru bude virtualizační prostředí připraveno.

3.3.2 Připojení fyzických serverů do clusteru

Nyní jsou připraveny dva fyzické servery s nainstalovaným virtualizačním OS Proxmox VE. Zatím každý server funguje sám za sebe a pro korektní provoz je potřeba propojit tyto dva servery do Proxmox VE Clusteru. To provedenu pomocí Proxmox GUI. Nejdříve provedu incializaci clusteru, což zabere několik minut. Jakmile je na prvním nodu připravený cluster, připojím druhý server dle dostupných instrukcí, které jsou zobrazeny v GUI.

3.3.3 Konfigurace síťového FS na platformě Ceph

V této části bude popsán poslední krok k dokončení nastavení systému pro virtualizaci, který začal v předchozí kapitole.

Jakmile je plně inicializovaný Proxmox VE cluster, je možné začít konfigurovat CEPH sdílený filesystem. Veškeré úkony je možné dělat v Proxmox WebUI

V rámci všech Proxmox serverů bude potřeba provést instalaci potřebných

[5] IPMI - vzdálená správa fyzických serverů, umožňující ovládání, nebo například připojení ISO souboru přímo z PC pomocí TCP/IP bez nutnosti kopírování na flash/CD médium

Ceph komponent. Všechny tyto balíčky se nacházejí ve veřejných Linux repozitářích a je také možné provést instalaci přes WebUI. Jakmile jsou nainstalované potřebné balíčky, je možné inicializovat cluster. Pro naše potřeby je ponecháno nastavení clusteru základní, je ale samozřejmě možné využít pokročilá nastavení. Následně do čerstvě vytvořeného clusteru přidám všechny fyzické disky, připravené pro síťový FS. Je potřeba dbát na to, aby disková kapacita a její rozložení byla stejná na všech fyzických serverech. Instalaci by sice bylo možné provést i s rozdílnou kapacitou, ale v případě selhání jednoho ze serverů by došlo k zastavení funkce celého systému.

Následně vytvořím na nově inicializovaném clusteru dva RBD oddíly. Jeden nazvaný "k8s"a druhý "vms". Tyto oddíly budou sloužit pro perzistentní data systému Kubernetes a pro diskové obrazy virtuálních strojů.

3.3.4 Instalace virtuálních strojů

Další krok je vytvoření a následná instalace 8 nových virtuálních strojů, které budou rozmístěny rovnoměrně na oba servery. Na servery nainstaluji operační systém Linux GNU, přesněji distribuci Ubuntu, která je v době psaní této práce 22.04 (Jammy Jallyfish). Tento operační systém je plně kompatibilní s technologií Kubernetes. Pro snadnější konfiguraci použiji již hotové obrazy pro instalaci, které nesou příznak Cloud-init. Tato technologie/aplikace umožňuje spustit VM bez nutnosti instalace z ISO souboru a ulehčuje konfiguraci nasazeného OS pomocí Cloud-init [6]. Technologie Cloud-init je plně podporována ze strany virtualizační platformy Proxmox, takže veškerá základní nastavení virtuálního stroje provedu pomocí této zmíněné technologie.

Technologie Cloud-init poskytuje velké množství možností nastavení. Pro stěžejní běh budou potřeba dvě, a to jsou SSH klíč a následně konfiguraci síťových rozhraní. Počítejme, že jsou všechny virtuální stroje propojeny, a proto přistoupím ke konfiguraci IPv4 adres. Jelikož se na síti nenachází DHCP server, použiji statické adresování. Pro adresaci je možné zvolit adekvátní rozsah

[6] Cloud-init - Jedná se o sadu nástrojů a aplikací, které umožňují nasadit připravený OS

v privátní podsíti (subnetu) například 10.50.50.0/24. Následně do tohoto rozsahu umístím všech 8 virtuální strojů, taktéž je potřeba všude umístit SSH veřejný klíč své stanice, ze které bude provedena instalace, která bude popsána v dalším kroku.

V poslední řadě před spuštěním virtuálního stroje je potřeba provést korektní alokaci diskového prostoru na Ceph cluster. Jak jsem již popisoval v předchozím kroku, byl vytvořen RBD oddíl "vms", na kterém bude disk virtuálního stroje umístěn.

3.3.5 Instalace K8S clusteru pomocí nástroje ansible (Kubespray)

Tato sekce se bude věnovat samotné instalaci orchestračního nástroje Kubernetes. Pro instalaci použiji opensource projekt, který je poskytován na stránce github.com pod názvem Kubespray [36]. Kubespray umožňuje za pomoci automatizačního nástroje Ansible provést instalaci na předem definovaných nodách.

Jelikož je aplikace Kubespray napsána v automatizačním nástroji Ansible, je potřeba vytvořit soubor hosts, který bude obsahovat IP adresy námi vytvořených virtuálních serveru z předchozích kroků. Aby se nemusel tento soubor vytvářet manuálně, tak projekt Kubespray vytvořil pomocný skript, který s touto aktivitou pomůže. Stačí pouze provést deklaraci IPv4 adres do proměnné a následně zavolat pomocný skript.

```
declare -a IPS=(10.50.50.1 10.50.50.2 10.50.50.3 10.50.50.4
↪  10.50.50.5 10.50.50.6 10.50.50.7 10.50.50.8)
CONFIG_FILE=inventory/mycluster/hosts.yaml python3
↪  contrib/inventory_builder/inventory.py ${IPS[@]}
```

Touto sadou příkazů se vytvořil soubor hosts.yaml, který je již možné použít pro nasazení Kubernetes. Nicméně je ještě dobré před samotným nasazením provést kontrolu konfiguračního souboru clusteru, který se bude nasazovat. Tato konfigurace je umístěna na cestě.

```
/inventory/sample/group_vars/k8s_cluster/k8s-cluster.yml
```

Tento konfigurační soubor určuje, jakou verzi clusteru si přejeme nainstalovat, nebo například jaký síťový plugin je možné použít, nebo jaké budou rozsahy pro spuštěné kontejnery. Například:

```
kube_config_dir: /etc/kubernetes
kube_script_dir: "{{ bin_dir }}/kubernetes-scripts"
kube_manifest_dir: "{{ kube_config_dir }}/manifests"
kube_cert_dir: "{{ kube_config_dir }}/ssl"
kube_token_dir: "{{ kube_config_dir }}/tokens"
kube_api_anonymous_auth: true
kube_network_plugin: calico
kube_network_plugin_multus: false
kube_service_addresses: 10.233.0.0/18
kube_pods_subnet: 10.233.64.0/18
kube_version: v1.26.1
```

Pro naše účely zůstane konfigurace v základním tvaru a použiji poslední stabilní verzi K8S a to již výše popsanou 1.26.1.

Po provedení kontroly konfiguračních souborů je možné přistoupit k samotnému nasazení systému Kubernetes. Pokud je vše korektně nastaveno dle předchozích kroků, tak stačí spustit tento příkaz, který inicializuje sadu Ansible playbooku na všech vytvořených virtuálních serverech a provede kompletní nastavení.

```
ansible-playbook -i inventory/mycluster/hosts.yaml  --become
↪  --become-user=root cluster.yml
```

Kompletní instalace clusteru zabere cca 30 min. Je velice důležité neztratit pro instalaci připojení k síti. To by mohlo mít za následek nekorektní instalaci a nutnosti provést reinstalaci všech virtuálních serverů.

3.3.6 Test nainstalovaného K8S clusteru

V následující části budu popisovat, jak otestovat, zda instalace Kubernetes clusteru, která byla provedna v předchozí části proběhla korektně a zda veškeré systémové komponenty fungují správně. Pro tuto aktivitu využiji nástroj, který byl vytvořen společně s Kubernetes pro jeho řízení s názvem kubectl, který je možné stáhnout na tomto odkazu kubectl.

Ze základního konceptu fungování klientské aplikace kubectl, které popisuji v teoretické části je jasné, že pro korektní fungování tohoto nástroje bude potřeba konfigurační soubor, který bude obsahovat přístupový token, adresu a uživatelské jméno pro přístup do clusteru. V rámci instalace pomocí nástroje Ansible došlo k vytvoření administrátorského přístupového konfiguračního souboru, který se v základní konfiguraci umístí na první master node vytvořeného clusteru. Pro aktivaci přístupu z naší stanice, tento konfigurační soubor přenesu. Pro instalaci a konfiguraci používám operační systém MacOS, který je Linux like OS, proto lze použít níže zmíněný příkaz pro přenesení souboru.

```
scp root@10.50.50.1:/root/.kube/config ~/.kube/
```

V případě použití například Windows, existuje alternativa v podobě programu WinSCP.

Po přenesení konfiguračního souboru je potřeba provést úpravu endpointu v konfiguraci, kde je uvedena lokální IPv4 (127.0.0.1). Úpravu jsem provedl hned v první sekci konfiguračního souboru, viz níže.

```
- cluster:
    certificate-authority-data: <base64 token>
    server: https://10.50.50.1:443
  name: ibobr-cluster
```

Jakmile je soubor uložen, otestuji propojení s clusterem. Pro test lze využít například příkaz

```
kubectl version
```

Tento příkaz by měl vrátit výstup podobný tomuto.

```
Client Version: version.Info{Major:"1", Minor:"25",
↪ GitVersion:"v1.25.2",
↪ GitCommit:"5835544ca568b757a8ecae5c153f317e5736700e",
↪ GitTreeState:"clean", BuildDate:"2022-09-21T14:33:49Z",
↪ GoVersion:"go1.19.1", Compiler:"gc",
↪ Platform:"darwin/arm64"}
Kustomize Version: v4.5.7
Server Version: version.Info{Major:"1", Minor:"24",
↪ GitVersion:"v1.24.9-gke.3200",
↪ GitCommit:"92ea556d4e7418d0e7b5db1ee576a73f8fc47e91",
↪ GitTreeState:"clean", BuildDate:"2023-01-20T09:29:29Z",
↪ GoVersion:"go1.18.9b7", Compiler:"gc",
↪ Platform:"linux/amd64"}
```

Pokud je zobrazen podobný výsledek znamená to, že Kubernetes cluster korektně odpovídá na volání a je možné pokračovat k dalšímu kroku.

3.3.7 Instalace podpůrných systémových aplikací

Tato sekce je zaměřena na instalaci potřebných komponent pro provozování webové aplikace v Kubernetes.

Pro potřeby webové aplikace budou potřeba následující komponenty, které jsou všechny volně dostupné a pod licencí Open Source. Instalaci těchto komponent provedu pomocí balíčkovacího nástroje Helm:

- Ingress kontroler - Aplikace, která slouží jako reverzní proxy a korektně směřuje webový provoz na základě ingress pravidel.
- Certmanager - Jedná se o komponentu, která se stará o automatické obměňování SSL certifikátů a pracuje ve spojení s Ingress kontrolerem (pro náš test budeme využívat Lets Encrypt certifikáty, které jsou vydávány zdarma).
- Ceph CSI Storage operator - Jedná se komponentu, díky které je možné vytvářet PVC (Persistent Volume Claim) objekty přímo ve vytvořeném Ceph clusteru.
- MetalLB - Jedná se o komponentu, která poskytuje Load Balancer as a service. (Tato komponenta se využívá pouze jako pomocná aplikace pro Ingress kontroler, kde zprostředkuje cloud loadbalancer.)

Pro spuštění instalace je potřeba mít nainstalovaný Helm client na svém počítači. Tuto instalaci provedu pomocí návodu, který se nachází na domovské stránce nástroje Helm helm.sh/docs/intro/install/helm.

MetalLB - pro účely load balanceru pro bare metal cluster (bare metal - Servery provozované mimo cloud ve vlastním DC a na vlastním hardware). Balíček metallb použiji z veřejně dostupného repozitáře na adrese artifacthub.io/packages/helm/metallb/metallb. Pro instalaci vytvořím konfigurační soubor **values.yaml** obsahující následující kód.

```
configInline:
  address-pools:
   - name: default
     protocol: layer2
     addresses:
     - 10.50.50.10/32 #Jedná se o adresu která nesmí být
      ↪ přiřazena na jiný stroj a bude použita pro HTTP a
      ↪ HTTPS komunikaci do clucteru
```

Jakmile je připraven konfigurační soubor, provedu instalaci do clusteru

```
helm repo add metallb https://metallb.github.io/metallb
helm install -f values.yaml -n metallb --wait
↪  --create-namespace metallb metallb/metallb
```

Tento příkaz provede instalaci balíčku MetalLB do Kubernetes cluster a namespace metallb.

Ceph CSI Storage operator - pro účely mapování Ceph RBD oddílů jako Kubernetes PVC využijeme balíček, který je dostupný ve veřejném repozitáři s názvem ceph-csi/ceph-csi-rbd na adrese artifacthub.io/packages/helm/ceph-csi/ceph-csi-rbd.

Před instalací je nutné získat potřebná data z aktuálně nainstalovaného clusteru. Přihlásím se pomocí SSH na jakýkoliv z virtualizačních serverů a spustím následující příkaz.

```
root@pve-1:/# ceph mon dump
dumped monmap epoch 3
epoch 3
fsid 79179d9d-98d8-4976-ab2e-58635caa7235
last_changed 2023-02-11T10:56:42.110184+0000
created 2023-02-11T10:56:22.913321+0000
min_mon_release 16 (pacific)
0: [v2:10.40.40.1:3300/0,v1:10.40.40.1:6789/0] mon.a
1: [v2:10.40.40.2:3300/0,v1:10.40.40.2:6789/0] mon.b
```

Následně vytvořím soubor **ceph-csi-rbd-values.yaml**, který bude sestaven z dat, které jsou zobrazeny na výstupu výše.

```
csiConfig:
  - clusterID: "79179d9d-98d8-4976-ab2e-58635caa7235"
    monitors:
      - "10.40.40.1:6789"
      - "10.40.40.2:6789"
provisioner:
  name: provisioner
  replicaCount: 2
```

Následně provedu instalaci ceph operátoru a přiložím vytvořený soubor, díky kterému dojde k prvotní inicializaci operátoru.

```
helm repo add ceph-csi https://ceph.github.io/csi-charts
helm repo update
helm install --namespace --wait ceph-csi-rbd --create-namespace
↪  ceph-csi-rbd ceph-csi/ceph-csi-rbd --values
↪  ceph-csi-rbd-values.yaml
```

Touto sadou příkazů došlo k instalaci, Ceph CSI operátora, který má základní konfiguraci a napojení na Ceph cluster. Zatím ale není možné vytvářet perzistentní volume, protože operátor zatím nemá žádný účet, pomocí kterého by mohl provádět komunikaci s Ceph clusterem. Tento účet je možné získat pomocí příkazů níže.

```
# Vytvoření uživatele a klíče pro přístup k RBD oddílu
↪  (následný klíč je v base64):
ceph auth get-or-create-key client.k8s_admin mon "allow r" osd
↪  "allow class-read object_prefix rbd_children, allow rwx
↪  pool=k8s" | tr -d '\n' | base64
QVFDZ0R5VmdyRk9KREJBQTJ5b2s5R1E2NUdSWExRQndhVVBwWXc9PQ==
```

```
# Převedeme také uživatelské jméno na base64:
echo "k8s_admin" | tr -d '\n' | base64
azhzX2FkbWlu
```

Následně vytvořím soubor **ceph-admin-secret.yaml**, který bude obsahovat právě získané přístupové heslo a také přístupový klíč. Tento soubor, který je typu Kubernetes Secret, bude veškerá citlivá data obsahovat ve formátu base64.

```
apiVersion: v1
kind: Secret
metadata:
  name: ceph-admin
  namespace: default
type: kubernetes.io/rbd
data:
  userID: azhzX2FkbWlu
  userKey:
  ↪ QVFCK0hDVmdXSjQ1T0JBQXBrc0VtcVhlZFpjc0JwaStIcmU5M3c9PQ==
```

Následně tento soubor aplikuji do kubernetes pomocí kontrolního nástroje kubectl.

```
kubectl apply -f ceph-admin-secret.yaml
```

Jakmile tento soubor aplikuji do kubernetes, využiji přístupový token, díky kterému existuje plný přístup k RBD oddílu k8s, který je vytvořený v Ceph clusteru. Aby mohlo docházet k automatickému vytváření PVC v rámci kubernetes, ještě vytvořím objekt, který se jmenuje StorageClass, který bude využívat právě nainstalovaný ceph operátor. Pro tento účel je potřeba soubor **ceph-rbd-pod-pvc-sc-allinone.yaml**, který bude obsahovat následující:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ceph-rbd-sc
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: rbd.csi.ceph.com
parameters:
   clusterID: 79179d9d-98d8-4976-ab2e-58635caa7235
   pool: k8s
   imageFeatures: layering
   csi.storage.k8s.io/provisioner-secret-name: ceph-admin
   csi.storage.k8s.io/provisioner-secret-namespace: default
   csi.storage.k8s.io/controller-expand-secret-name: ceph-admin
   csi.storage.k8s.io/controller-expand-secret-namespace:
   ↪ default
   csi.storage.k8s.io/node-stage-secret-name: ceph-admin
   csi.storage.k8s.io/node-stage-secret-namespace: default
reclaimPolicy: Delete
allowVolumeExpansion: true
mountOptions:
   - discard
```

Následně tento soubor opět aplikuji pomocí stejného příkazu, jako předchozí Secret.

```
kubectl apply -f ceph-rbd-pod-pvc-sc-allinone.yaml
```

Tímto byl vytvořen objekt StorageClass, který slouží jako šablona pro tvorbu PVC a bude možné je dynamicky vytvářet, mazat a upravovat.

Certmanager - pro účely automatické správy a obnovy certifikátu. Balíček cert-manager bude použit z veřejně dostupného repozitáře na adrese artifacthub.io/packages/helm/cert-manager/cert-manager. Instalaci tohoto balíčku provedu opět pomocí nástroje Helm.

```
helm repo add jetstack https://charts.jetstack.io
helm install cert-manager --create-namespace --wait --namespace
↪  cert-manager --version v1.11.0 jetstack/cert-manager
```

Po dokončení instalace bude cert-manager připraven k použití. V základním nastavení slouží cert-manager pouze k ukládání certifikátu. Aby bylo možné nechávat vystavovat certifikáty automaticky přes protokol ACME [7], bude potřeba na clusteru vytvořit objekt typu ClusterIssuer, který zaregistruje ACME endpoint a umožní vystavení.

Vytvořím si soubor **cluster-issuer-le.yaml**, který bude obsahovat následující:

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
  annotations:
    "helm.sh/hook": post-install,post-upgrade
spec:
  acme:
    email: marekt04@jcu.cz
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
```

[7] ACME - Jedná se o protokol, který komunikuje přímo s CA a umožní vydat po kontrole certifikát, který je podepsaný

```
    - http01:
        ingress:
          class: nginx
```

Následně opět aplikuji přes nástroj kubectl.

```
kubectl apply -f cluster-issuer-le.yaml
```

Nyní je připravený cert-manager, který bude sloužit k automatické obsluze a vydávání Lets Encrypt certifikátu pro webový provoz.

Ingress kontroler - bude sloužit jako reverzní proxy pro všechny komponenty a aplikace, které budou provozovány na daném Kubernetes clusteru. Balíček ingress-nginx bude použit z veřejně dostupného repozitáře na adrese artifacthub.io/packages/helm/ingress-nginx/ingress-nginx. Instalace tohoto balíčku bude provedena opět pomocí nástroje Helm.

```
helm repo add ingress-nginx
↪ https://kubernetes.github.io/ingress-nginx
helm repo update
helm install ingress -n ingress --create-namespace --wait
↪ ingress-nginx/ingress-nginx
```

Pokud veškeré předchozí kroky byly provedeny korektně, tak by po dokončení instalace, měl být na adrese **http://10.50.50.10** zobrazen HTTP kód, který bude hlásit "nic nenalezeno"(404 Not Found)

3.3.8 Instalace DB clusteru

V této sekci popsuji instalaci databázového clusteru, který využiji pro provoz aplikace Bobřík informatiky. Jak bylo zmíněno výše, na aktuálním řešení je databázový server provozován na odštěpené verzi MySQL a tou je MariaDB. V rámci aktuálního řešení je databázový server provozován v jedné instanci a v případě selhání není dostupný fail over. Pro potřeby nasazení nové infrastruktury budu také využívat verzi MariaDB, která je zabalená pod clustrovací technologií Galera.

Instalaci databázového clusteru provedu stejně jako u ostatních komponent pomocí balíčkovacího nástroje Helm. Balíček mariadb-galera použiji z veřejně dostupného repozitáře na adrese artifacthub.io/packages/helm/appuio/mariadb-galera. Pro naše potřeby bude dostačující základní konfigurace, proto není potřeba vytvářet modifikační soubor a stačí pouze spustit instalaci pomocí následujícího příkazu.

```
helm repo add appuio https://charts.appuio.ch
helm repo update
helm install -n ibobr-galera-db --create-namespace --wait
↪  mariadb-galera appuio/mariadb-galera
```

Jakmile je příkaz dokončen, tak je DB připravena k použití a nahrání dumpu.

3.3.9 Úprava aplikace Bobřík informatiky pro běh v Kubernetes

Tato sekce se zaměřuje na úpravu aplikace Bobřík informatiky, tak aby bylo možné aplikaci spustit ve více instancích najednou, které budou sdílet stejnou cache a také stejnou session storage. Aplikace v aktuálním nastavení neumožňuje izolované spuštění docker kontejneru (vyžaduje existenci zdrojového kódu na souborovém systému a docker kontejner využívá pouze jako PHP a Apache runtime), což je pro provoz v orchestračním systému Kubernetes nepřípustné a je potřeba mít provedenou kompletní izolaci aplikace do Docker obrazu.

Pro potřeby sdílení cache a session použiji key-value in-memory databázi jménem Redis (a její alternativy, jako třeba KeyDB). Tuto databázi budu instalovat taktéž přímo v clusteru. V tomto řešení využiji fork aplikace Redis, který byl zmíněn výše a to KeyDB. Použiji balíček keyDB z veřejně dostupného repozitáře na adrese artifacthub.io/packages/helm/appuio/enapter/keydb. Následně provedu instalaci se základními hodnotami pomocí následující sady příkazů.

```
helm repo add enapter https://enapter.github.io/charts/
helm repo update
```

```
helm install -n ibobr-test --create-namespace --wait keydb
↪ enapter/keydb
```

V dalším kroku implementuji připojení na Redis (KeyDB) z frameworku Nette, na kterém je aplikace Bobřík informatiky postavena. Pro tyto potřeby slouží veřejně dostupná knihovna Kdyby/Redis na adrese github.com/Kdyby/Redis Instalaci provedu pomocí nástroje composer, která je uvedena v přiložené dokumentaci projektu Kdyby/Redis. Následně se musí provést inicializace použité knihovny v Nette frameworku. K tomu bude sloužit soubor **redis.php**, který umístím ve složce **app/config/**. Tento soubor obsahuje inicializační nastavení a také to, do jaké DB se budou ukládat jaká data. Konfigurační soubor by měl vypadat takto.

```
<?php
use Kdyby\Redis\DI\RedisExtension;
if (extension_loaded('redis')) {
        return [
                'extensions' => [
                        'redis' =>
                        ↪ Kdyby\Redis\DI\RedisExtension::class
                ],
                'redis' => [
                        'host' => 'REDIS_SESSION_HOSTNAME',
                        'database' => 1, #Journal,Cache a DB
                        ↪ cache budou ukládány do DB 1 na
                        ↪ instanci keydb
                        'journal' => true,
                        'storage' => true,
                        'session' => [
```

```
                                'database' => 2 #Session
                                ↪ storage (určující například
                                ↪ že je daný uživatel na
                                ↪ daném pc přihlášený k
                                ↪ testu) bude ukládána do DB
                                ↪ 2 na instanci keydb
                        ]
                ]
        ];
}
```

Jakmile je soubor vytvořen, tak je ho nutné ještě zaregistrovat pomocí include v souboru **app.neon**. Tím je připojení na in-memory databázi připravené. Nette od té doby začne veškerou cache a session ukládat na připravený DB server.

Jako další krok v úpravách aplikace následuje modifikace sestavení Docker Image. Jak bylo již zmíněno výše, existující Dockerfile pro Bobříka informatiky se využívá pouze částečně a je potřeba tento Dockerfile přepracovat. Nový Dockerfile proto rozdělím do čtyř fází a to **Install common**, **Composer dependencies**, **Frontend**, **Runtime** a postupně rozvedu každou fázi.

Fáze **Install common**. Obsah této fáze byl z velké většiny převzat z přechozího kroku. Navíc jsem pouze nainstaloval PHP modul redis.

Fáze **Composer dependencies**. V rámci této fáze ze zdrojového image composeru převezmu binární soubor composer a následně nad projektovým souborem **composer.json**, který obsahuje potřebné balíčky provedu composer install. Pro svůj běh využívá již nainstalované knihovny z přechozí fáze **Install common**.

Fáze **Frontend**. Tato fáze již byla jako celá přidána a obsahuje kompletní sestavení Frontend javacriptu a také CSS souborů pomocí nástroje Gulp.

Fáze **Runtime**. Tato fáze byla také jako celá přidána a obsahuje sesta-

vení vytvořených souborů do výsledného Docker image. Postupně zde provedu vykopírování souborů z jednotlivých fází. Jako první se kopíruje sestavený backend ze sekce **Composer dependencies** a následně sestavené frontend CSS a JS z fáze **Frontend**. Následně provedu ještě dodatečnou konfiguraci webového serveru apache a aktivuji rewrites. Poté už jen zaregistruji entrypoint bash script, který spouští proces webového serveru po spuštění kontejneru.

3.3.10 Implementace procesu automatického nasazení přes nástroj GitLab

V této sekci budu popisovat proces automatického nasazení aplikace Bobřík informatiky do Kubernetes clusteru. Jelikož ve školním GitLabu mám oprávnění pouze jako host, vytvořil jsem pro tyto potřeby vlastní instanci GitLabu, kde jsem zapracoval potřebné změny. Pro potřeby tohoto nově vytvářeného řešení budu uvažovat pouze jedno prostředí v orchestračním nástroji Kubernetes a to produkční. Ostatní vývoj bude prováděn lokálně a proto nebude třeba zavádět STAGE prostředí [8]. Pro zachování korektního verzování a také možnosti rollback strategie, bude implementováno automatické nasazení pouze po vytvoření tagu ve verzovacím nástroji GIT. Také bude potřeba narozdíl od aktuálního řešení, provádět sestavení Docker image před samotným nasazením a následně uložit do GitLab Docker Registry, nikoliv až na cílovém serveru. Pro tyto potřeby budu využívat **Gitlab CI/CD Pipelines**.

Aby bylo možné začít používat GitLab Pipelines, je nejprve potřeba mít připravený soubor **.gitlab-ci.yml** kterým bude následně plněn požadovaný obsah. Obsahem souboru se rozumí definice pipeline, která se zadává ve značkovacím jazyce YAML a obsahuje připravené řídící prvky, které jsou dále popsané přímo v dokumentaci GitLabu [38]. Pro naši potřebu zavedu několik úkolů. Bude se jednat o **docker-build**, **template-helm-package**, **test-helm-package**, **production-deploy**. Následně postupně popíši jednotlivé úkoly.

[8] STAGE prostředí - jedná se o vývojové prostředí, které kopíruje produkční podmínky a umožňuje test aplikace před uvedením do produkce.

Úkol **docker-build**. Jedná se o proces, který byl dříve spouštěn přímo na serveru při nasazení. Aby bylo možné aplikaci nasadit bez nutnosti opětovného sestavení, připravím již celý Docker image a následně nahraji do uložiště docker images.

Úkol **template-helm-package**. Tento proces provádí jednoduchou kontrolu správnosti YAML definice pro nasazení.

Úkol **test-helm-package**. Jedná se o ekvivalentní úkol, jako předchozí, jenom se jedná o test funkčnosti jako celku.

Úkol **production-deploy**. Tento úkol slouží již k přímému nasazení do Kubernetes clusteru podle definice přes balíčkovací nástroj Helm.

3.3.11 Nasazení aplikace do prostředí Kubernetes

V této sekci provedu nasazení celé aplikace do nového prostředí. Dle navržené architektury po vytvoření GIT tagu se spustí automatický úkol, který provede nasazení aplikace do nového prostředí. Nová instance je po tomto kroku dostupná do několika minut.

3.4 Zátěžové testy

V této sekci budu popisovat přípravu a následnou realizaci zátěžových testů, které byly prováděny proti nově navrženému řešení a také oproti aktuálně provozovanému řešení.

3.4.1 Příprava testovacích scénářů

Pro přípravu zátěžových testů jsem využil nástroj, který poskytuje software Gatling a to Gatling Recorder. Tento nástroj umožňuje zaznamenat reálnou aktivitu uživatele v dané aplikaci. Aktivitu je možné zaznamenávat dvěma způsoby. Jako první je nahrání HAR souboru[9] a jako druhá možnost je HTTP

[9]HAR soubor - Jedná se o export síťové aktivity přímo z webového prohlížeče. Pro korektní zaznamenání je nutné mít aktivní vývojářskou konzoli.

proxy, kterou jsem zvolil pro zaznamenání uživatelské aktivity. Značnou výhodou využití této proxy je schopnost zaznamenávat pauzu uživatele (když student zvažuje jakou odpověď zvolí) a tím pádem simulovat reálné chování studenta.

Proxy server je spuštěn na stanici, kde jsem také spustil nástroj Gatling Recorder. Následně jsem spuštěný proxy server nastavil do svého prohlížeče, aby bylo možné zaznamenat moji aktivitu. Jakmile byl server přidán, začal jsem se simulováním uživatele.

Nyní je vše připraveno pro nahrání testu. Otevřu nově navrženou instanci aplikace Bobřík informatiky a provádím klasické vyplnění testu, jako by prováděl standardní student. Mezitím se veškeré mé kroky a požadavky na servery zaznamenávají a ukládají pomocí proxy serveru. Jakmile jsem dokončil celý test nástroj Gatling Recorder z celého průběhu automaticky vygeneroval testovací scénář ve formátu scala. Stejný postup opakujeme také pro stávající řešení.

3.4.2 Průběh testů

Nyní již mám připravené simulační scénáře, které jsem si připravil v předchozím kroku a jako poslední krok před spuštěním testů bude potřeba implementovat chování daného scénáře (počet souběžně aktivních uživatelů, délka testu), neboli Gatling Injection. Pro naše potřeby bude využívat dvě sady chování scénáře.

Jako první test budeme uvažovat konstantní počet uživatelů a to 400 po dobu běhu 30 minut, který je definován následujícím způsobem.

```
setUp(
  scn.inject(
    constantConcurrentUsers(400).during(600.seconds)
  ).protocols(httpProtocol)
)
```

Jako druhý test budeme uvažovat náhodně generující provoz v rozpětí 200 až 250 aktivních uživatelů po dobu běhu 30 minut, který je definován následujícím způsobem.

```
setUp(
  scn.inject(
    rampUsers(200).to(250).during(30.minutes).randomized
  ).protocols(httpProtocol)
)
```

Po přidání sady chování daného scénáře je možné test spustit. Test spustíme následujícím příkazem a můžeme pozorovat jak test postupuje přímo na obrazovce.

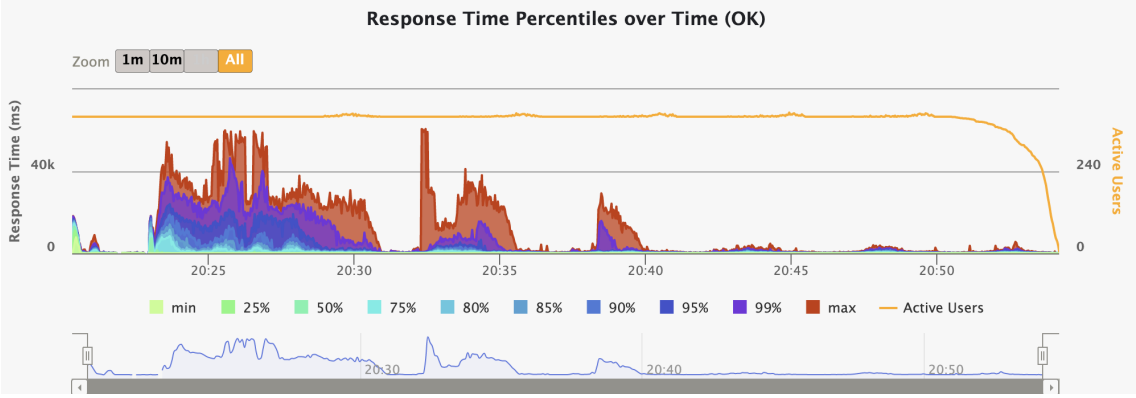
```
./bin/gatling.sh -s <název simulace> --run-mode local
```

Při pokusu spustit test s větším počtem uživatelů (600rq) na aktuálním řešení, jsem bohužel narazil na problém u poskytovatele Web4U, který má implementovanou ochranu proti DDoS útoku [10] a postupně blokoval IPv4 adresy, ze kterých jsem test prováděl.

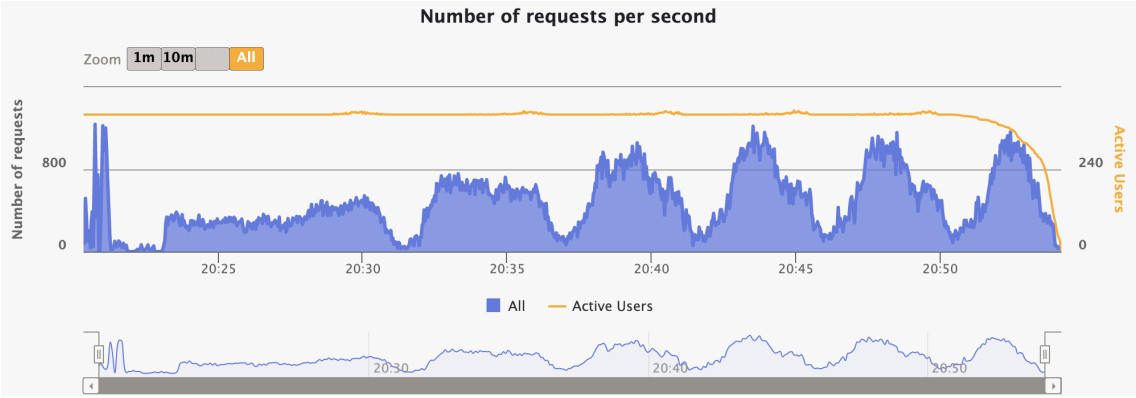
[10] DDoS je zkratka pro Distributed Denial of Service. Jedná se o útok, při kterém je cílový server zaplaven obrovským množstvím požadavků.

3.4.3 Konstantní zátěž na novém řešení (400rq)

Zde je možné pozorovat znatelný nárůst času odezvy v začátku testu, což bylo způsobeno časem, který byl potřeba na automatické naškálování. Následně se čas odpovědi vrátil do normálu.



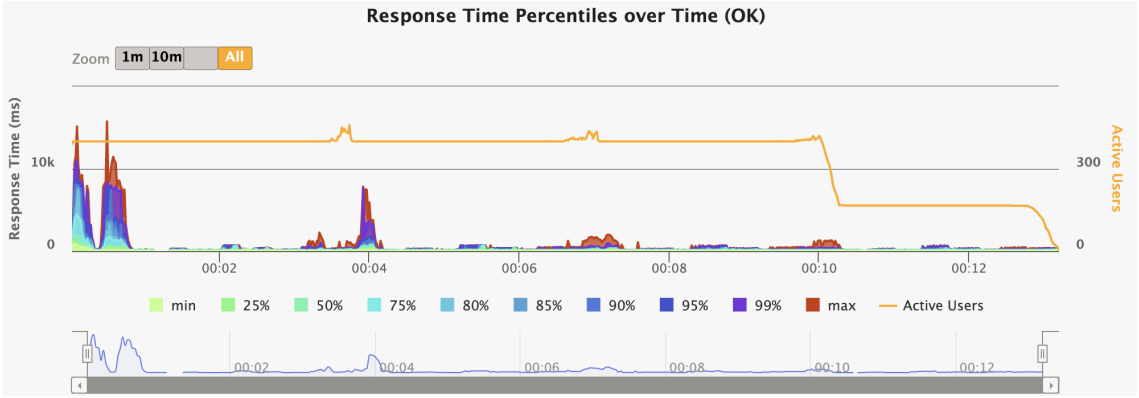
Obrázek 13: RT Konstatní 400 požadavků (nové řešení)



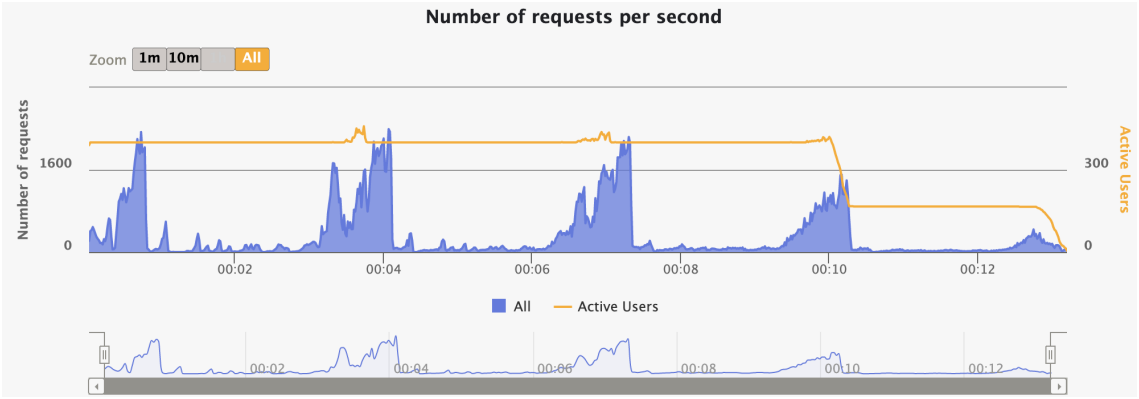
Obrázek 14: RQ Konstatní 400 požadavků (nové řešení)

3.4.4 Konstantní zátěž na stávajícím řešení (400rq)

Zde je možné pozorovat znatelný nárůst času odezvy v začátku testu, což bylo způsobeno spouštěním vláken na webovém serveru, následně nedocházelo k dalšímu kolísání a počet aktivních spojení byl stále stejný. Po stabilizaci čas odpovědi opět klesl. Je nutné podotknout, že server byl na toto zatížení v době testu manuálně naškálován.



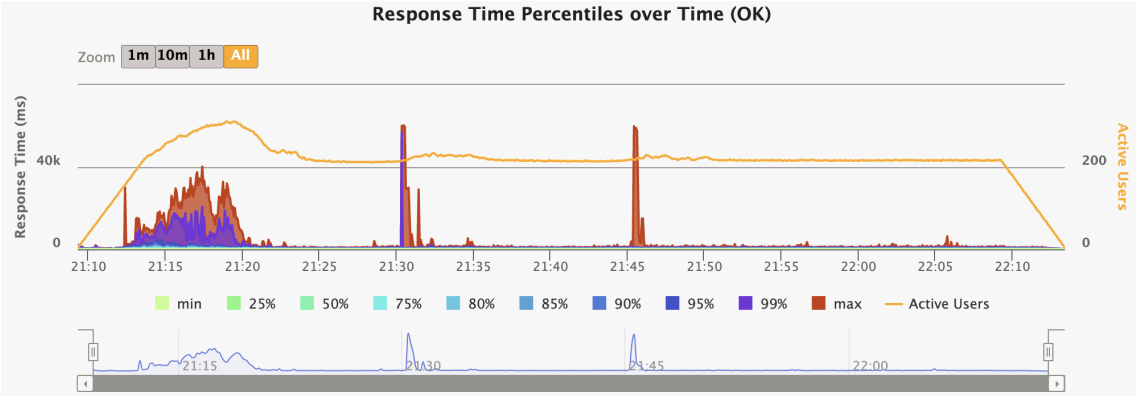
Obrázek 15: RT Konstatní 400 požadavků (stávající řešení)



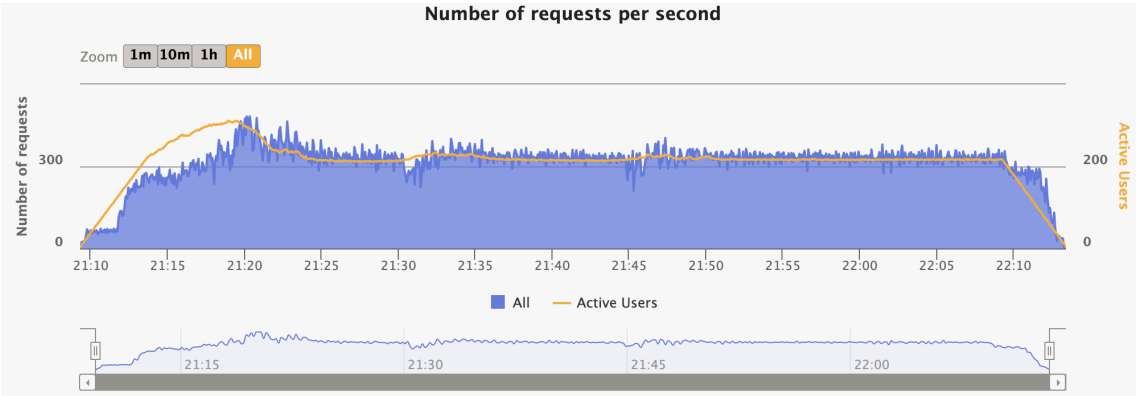
Obrázek 16: RQ Konstatní 400 požadavků (stávající řešení)

3.4.5 Náhodná zátěž na novém řešení (200 - 250rq)

Zde je možné pozorovat znatelný nárůst času odezvy v začátku testu, což bylo způsobeno stejně jako u předchozího testu časem, který byl potřeba na automatické naškálování. Následně se čas odpovědi vrátil do normálu. Další zvýšení času odezvy opět značí automatické škálování.



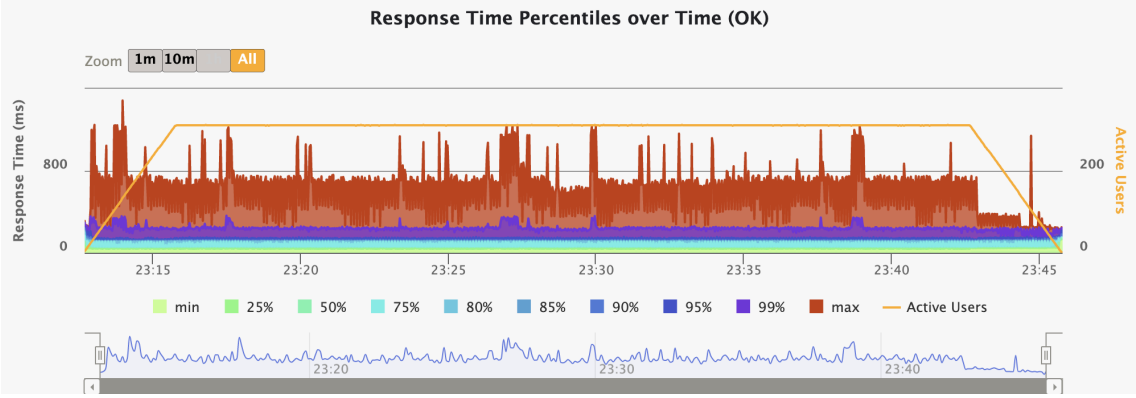
Obrázek 17: RT Náhodných 200-250 požadavků (nové řešení)



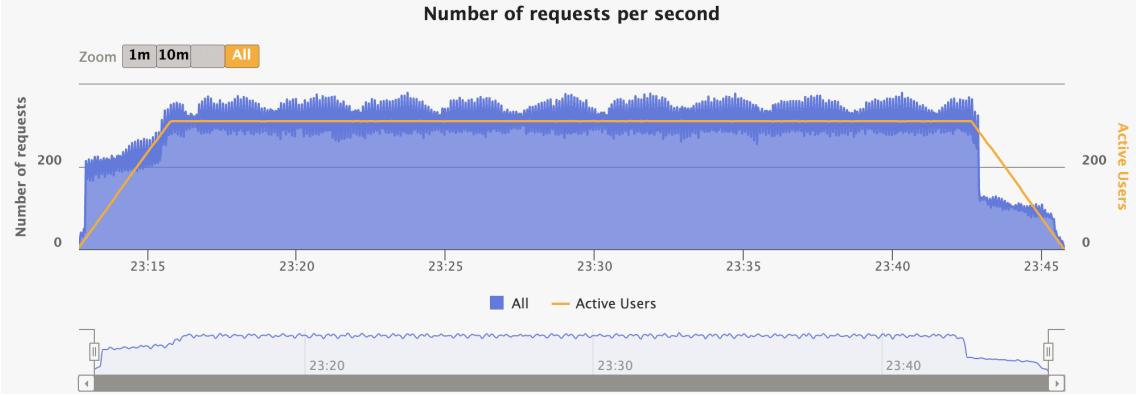
Obrázek 18: RQ Náhodných 200-250 požadavků (nové řešení)

3.4.6 Náhodná zátěž na stávajícím řešení (200 - 250rq)

Zde je možné pozorovat, po colou dobu trvání testu, znatelné kolísání času odpovědi při zvětšování počtu uživatelů.



Obrázek 19: RT Náhodných 200-250 požadavků (stávající řešení)



Obrázek 20: RQ Náhodných 200-250 požadavků (stávající řešení)

3.4.7 Výsledky testů

V této sekci se zaměříme na výsledky výkonnostních testů aplikačních prostředí Bobříka informatiky. Dle dostupných dat a grafů, které jsou zobrazeny v předchozí kapitole, je možné pozorovat následující chování.

V případě, že se jedná o náhodně generující zátěž, která nebyla předem predikovatelná, je možné pozorovat zvyšující se čas odezvy na aktuálním prostředí, který odpovídá náporu náhodně připojovaných uživatelů. Tento jev by mohl mít ve větším měřítku za následek úplné zahlcení webového serveru. V případě nového řešení je možné pozorovat znatelné zvýšení odezvy webového serveru na začátku testu. Toto zvýšení zapříčinil čas, který potřebuje orchestrace Kubernetes k tomu, aby provedla navýšení replik aplikace Bobřík informatiky. Následně je možné pozorovat opětovné snížení odezvy.

V případě, že se jedná o konstantní zátěž, můžeme na aktuálním prostředí pozorovat zvýšení odezvy pouze na začátku při navázání komunikace, následně se čas odezvy opět vrátil k normálu. Při konstantní zátěži nepozorujeme znatelné nárůsty času odezvy v průběhu testu. V případě nového řešení je možné pozorovat stejný průběh chování, jako u předchozího scénáře, kdy na začátku testu došlo ke zvýšenému času odezvy, ve kterém došlo k automatickému naškálování aplikačních podů.

4 Závěr

V závěru práce bych chtěl říci, že nemohu přesně určit, zda je nové řešení lepší nebo horší než aktuálně provozované řešení.

Aktuálně provozované řešení dle zátěžových testů nevykazuje problémové chování, pokud je provoz na aplikaci Bobřík informatiky predikovatelný a server je na tento provoz dostatečně horizontálně naškálován. Ovšem vykazuje nedostatečný výkon při náhodně generující zátěži, kde dochází ke zvyšování času odezvy serveru, které by při větším kolísání mohlo mít za následek zahlcení serveru. Také zde musím vyzdvihnout některé výhody stávajícího řešení, a to například náklady, které jsou na údržbu znatelně nižší, než u nově navrženého řešení. Nicméně musím zde zmínit i několik nevýhod aktuálního řešení, mezi které patří například nutnost restartu celého serveru při provádění horizontální škálování, nebo nedostatečná odolnost při selhání fyzického serveru.

V případě nově navrženého řešení, je zde chování při obou scénářích totožné. Na začátku testu detekuje orchestrační nástroj zvýšenou zátěž na aktuálně spuštěných replikách a začne spouštět automaticky další repliky aplikace Bobřík informatiky. Tento stav je způsoben tím, že orchestrační nástroj Kubernetes je nastaven tak, aby v případě klidového stavu snížil počet replik aplikace na minimální počet (v našem případě se jednalo o 2 repliky) které mají značně snížené zdroje. Nespornou výhodou tohoto řešení je tady vysoká flexibilita při reagování na náhodnou zátěž aplikace Bobřík informatiky, nebo také odolnost vůči selhání fyzického serveru, nebo například také virtuálního serveru.

Dle výše popsaného můžeme tedy konstatovat, že pro **predikovatelný** provoz je aktuální řešení provozu Bobříka informatiky dostačující (s nutností úpravy dle predikce před spuštěním), kdežto při **náhodně generující** zátěži by bylo lepší využít navrhované řešení, které je schopné se flexibilně vypořádat s nárazovými změnami.

Reference

[1] Server Virtualization [online]. Simplex Tech Articles [cit. 2023-04-14]. Dostupné z: https://www.simplex.com.cy/tech_articles/5-Server-Virtualization-Technologies

[2] About Hypervisor [online]. VMWare [cit. 2023-04-14]. Dostupné z: https://www.vmware.com/topics/glossary/content/hypervisor.html

[3] What is virtualization [online]. HPE Blog [cit. 2023-04-14]. Dostupné z: https://www.hpe.com/pl/en/what-is/virtualization.html

[4] Virtualizace [online]. Brno: Fakulta informatiky Masarykovy univerzity, 2016 [cit. 2023-04-05]. Dostupné z: https://www.fi.muni.cz/ kas/pv090/referaty/2016-podzim/virt.html

[5] HALAMÍČEK Petr, Techniky a možnosti virtualizace výpočetního prostředí [online]. Brno: Mendelova zemědělská a lesnická univerzita v Brně, 2009 [cit. 2023-04-05]. Dostupné z: https://docplayer.cz/20806815-Techniky-a-moznosti-virtualizace-vypocetniho-prostredi.html

[6] Docker overview [online]. Docker Docs [cit. 2023-04-05]. Dostupné z: https://docs.docker.com/get-started/overview

[7] What is docker [online]. Microsoft Learn [cit. 2023-04-05]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/architecture/containerized-lifecycle/what-is-docker

[8] Docker Architecture [online]. Docker Arch Guide [cit. 2023-04-05]. Dostupné z: https://geekflare.com/docker-architecture/

[9] Nasazování kontejnerizovaných aplikací do cloudu [online]. Hynek Proske [cit. 2023-04-05]. Dostupné z: https://theses.cz/id/hy6hk8/STAG89046.pdf

[10] What is docker registry [online]. Docker registry docs [cit. 2023-04-05]. Dostupné z: https://www.geeksforgeeks.org/what-is-docker-registry/

[11] About OCI [online]. OCI, oficiální web [cit. 2023-04-05]. Dostupné z: https://opencontainers.org/about/overview/

[12] About Kubernetes [online]. Google Cloud, What is Kubernetes [cit. 2023-04-05]. Dostupné z: https://cloud.google.com/learn/what-is-kubernetes

[13] Kubernetes architection guide [online]. Medium DevOps Mojo [cit. 2023-04-05]. Dostupné z: https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd

[14] Kubernetes components [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/concepts/overview/components/

[15] Quickstart [online]. ETCD [cit. 2023-04-05]. Dostupné z: https://etcd.io/docs/v3.5/quickstart/

[16] kubelet [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[17] kube-apiserver [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

[18] Kubernetes networking guide [online]. Calico [cit. 2023-04-05]. Dostupné z: https://www.tigera.io/learn/guides/kubernetes-networking/

[19] What is pod in Kubernetes [online]. RedHat Docs [cit. 2023-04-05]. Dostupné z: https://www.redhat.com/en/topics/containers/what-is-kubernetes-pod

[20] ReplicSet [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

[21] Deployment [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[22] StatefullSet [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[23] DaemonSet [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

[24] Service [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/

[25] Ingress [online]. Kubernetes Docs [cit. 2023-04-05]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/ingress/

[26] Quickstart [online]. Helm Docs [cit. 2023-04-05]. Dostupné z: https://helm.sh/docs/intro/quickstart/

[27] What is Helm? Helm and Helm Charts Explained [online]. Phoenix-NAP KB [cit. 2023-04-05]. Dostupné z: https://phoenixnap.com/kb/what-is-helm

[28] Named Templates [online]. Helm Docs [cit. 2023-04-05]. Dostupné z: https://helm.sh/docs/chart_template_guide/named_templates/

[29] Rollback [online]. Helm Docs [cit. 2023-04-05]. Dostupné z: https://helm.sh/docs/helm/helm_rollback/

[30] Ceph overview [online]. Cloudinfrastack Blog [cit. 2023-04-05]. Dostupné z: https://cloudinfrastack.com/blog/ceph/overview_of_ceph/

[31] RADOS Block Device [online]. TechTarget Blog [cit. 2023-04-05]. Dostupné z: https://www.techtarget.com/searchstorage/definition/RADOS-Block-Device-RBD

[32] How Ansible works [online]. RadHat Ansible [cit. 2023-04-05]. Dostupné z: https://www.ansible.com/overview/how-ansible-works

[33] What is Ansible And How to Use Ansible in Docker [online]. SimpliLearn Tutorials [cit. 2023-04-05]. Dostupné z: https://www.simplilearn.com/tutorials/ansible-tutorial/what-is-ansible

[34] README.md [online]. GitHub Kubespray projekt [cit. 2023-04-05]. Dostupné z: https://github.com/kubernetes-sigs/kubespray

[35] Docs [online]. Gatling Web [cit. 2023-04-05]. Dostupné z: https://gatling.io/docs/gatling/

[36] Kubespray [online] GitHub [cit. 2023-04-05], Dostupné z: https://github.com/kubernetes-sigs/kubespray.

[37] Proxmox ISO [online] Proxmox Docs [cit. 2023-04-05], Dostupné online na Web

[38] YAML [online]. GitLab Docs [cit. 2023-04-05]. Dostupné z: https://docs.gitlab.com/ee/ci/yaml/

Seznam obrázků