

UNIVERZITA HRADEC KRÁLOVÉ  
FAKULTA INFORMATIKY A MANAGEMENTU  
KATEDRA INFORMAČNÍCH TECHNOLOGIÍ

## DIPLOMOVÁ PRÁCE

Migrace legacy aplikace do kontejnerové virtualizace

**Autor:** Pavel Čížinský

**Studijní obor:** Aplikovaná informatika

**Vedoucí práce:** doc. Ing. Vladimír Soběslav, Ph.D

Hradec Králové, 2019

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a uvedl jsem všechny použité prameny a literaturu.

V Jaroměři dne 31. července 2019

Pavel Čížinský

Rád bych poděkoval doc. Ing. Vladimíru Soběslavovi, Ph.D. za odborné vedení práce, podnětné rady a čas, který mi věnoval.

## **Anotace**

Diplomová práce pojednává o migraci tradiční aplikace do kontejnerového prostředí. Migrace aplikací do kontejnerového řešení je v dnešním světě mikroslužeb a cloudu obrovské téma. Kontejnerový přístup řeší spoustu kritických problémů spojených s životním cyklem aplikace. S migrací přicházejí změny, jako je nutná úprava architektury a nasazovacího procesu celé aplikace. Hlavním cílem práce je zanalyzovat možnosti migrace tradiční aplikace do kontejnerového prostředí. V praktické části je vybraná aplikace zkontejnerizována a spuštěna v kontejnerovém prostředí za pomoci představených technologií.

## **Klíčová slova**

Kontejnery, Legacy aplikace, Docker, Kubernetes, OpenStack

## **Annotation**

Title: Migration of legacy application to container environment

This Master's thesis describes migration of legacy application to container-based virtualization environment. Migration of legacy applications is a huge topic in today world of clouds and microservices, because container-based virtualization solves a problems, which are connected with application lifecycle management. The main goal of this thesis is introduction to a concept of container-based virtualization and analyze selected legacy application, which will be containerized. In practical part selected application is containerized and migrated to container environment.

## **Key words**

Containers, Legacy application, Kubernetes, OpenStack

# Obsah

<b>1. Úvod</b>	<b>1</b>
<b>2. Kontejnerizace</b>	<b>4</b>
2.1. Historie kontejnerové virtualizace . . . . .	4
2.2. Kontejnery . . . . .	5
2.3. Mikroslužby . . . . .	9
2.4. Orchestrátory . . . . .	10
2.5. Stavové a bezstavové aplikace . . . . .	11
2.6. Porovnání orchestrátorů . . . . .	12
2.6.1. Docker orchestrátory . . . . .	12
2.6.2. Kubernetes . . . . .	13
2.6.3. Apache Mesos . . . . .	15
2.6.4. Ostatní . . . . .	16
2.6.5. Souhrn . . . . .	17
2.7. Orchestrace workloadu . . . . .	17
2.7.1. Helm . . . . .	18
2.7.2. Jsonnet . . . . .	20
2.7.3. Kustomize . . . . .	22
2.7.4. Kubernetes operátor . . . . .	23
2.7.5. Srovnání . . . . .	24
<b>3. Vymezení legacy aplikace</b>	<b>26</b>
3.1. Definice . . . . .	26
3.2. Monolitická architektura . . . . .	26
3.3. Problémy . . . . .	28
3.3.1. Metodologie vývoje . . . . .	28
3.3.2. Hardware . . . . .	30
3.3.3. Udržování aplikace . . . . .	30
3.3.4. Aktualizace a CI/CD . . . . .	30
<b>4. Analýza migrované aplikace</b>	<b>32</b>
4.1. Mirantis Cloud Platform . . . . .	32
4.1.1. OpenStack . . . . .	33
4.1.2. DriveTrain – CI/CD . . . . .	33
4.1.3. Stacklight – Logování a monitoring . . . . .	34

4.2.	Architektura . . . . .	35
4.2.1.	Orchestrace (SaltStack) . . . . .	38
4.2.2.	Metadata model . . . . .	39
4.2.3.	Instalace . . . . .	41
4.2.4.	Správa . . . . .	42
4.3.	Problémy . . . . .	43
4.3.1.	Složitost . . . . .	43
4.3.2.	Life cycle management a aktualizace . . . . .	44
4.3.3.	Vývoj a testování . . . . .	45
<b>5.</b>	<b>Návrh budoucí architektury kontejnerové aplikace</b>	<b>46</b>
5.1.	Současné možnosti . . . . .	46
5.2.	Definice požadavků . . . . .	48
5.3.	Návrh budoucí architektury kontejnerové aplikace . . . . .	49
<b>6.</b>	<b>Implementace kontejnerového řešení</b>	<b>53</b>
6.1.	Keystone . . . . .	53
6.2.	Operator SDK . . . . .	55
6.3.	Dockerfile . . . . .	57
6.4.	Implementace logiky operátoru . . . . .	59
<b>7.</b>	<b>Porovnání klasického a kontejnerového řešení</b>	<b>67</b>
7.1.	Definice testovacích prostředí . . . . .	67
7.1.1.	Testovací scénáře . . . . .	67
7.1.2.	Instalace řešení . . . . .	68
7.1.3.	Vysoká dostupnost . . . . .	68
7.1.4.	Zátěž . . . . .	68
7.1.5.	Škálování . . . . .	70
7.1.6.	Lifecycle management . . . . .	70
7.2.	Porovnání . . . . .	71
<b>8.</b>	<b>Závěr</b>	<b>73</b>
	<b>Literatura</b>	<b>74</b>
<b>9.</b>	<b>Přílohy</b>	<b>1</b>
<b>A.</b>	<b>Cizí výrazy</b>	<b>I</b>

# 1. Úvod

V dnešní době mobilních technologií a cloudových aplikací je kladen velký důraz na rychlý vývoj a doručování aplikací koncovým uživatelům. Tento trend můžeme sledovat především u dynamicky se rozvíjejících firem a startupů, které jsou schopny vydávat i několik verzí svých aplikací za měsíc. Vývoj aplikací se za posledních deset let dramaticky zrychlil. K vytvoření aplikace už není potřeba složitě designovat infrastrukturu, datacentra či ladit a optimalizovat databáze. Většina populárních technologií je vydávána jako open source a je částečně dostupná pro vývojáře zdarma. Infrastruktura v podobě fyzických serverů je často nahrazována cloudem, který je možno pronajmout a provozovat na něm téměř vše od databází až po sítě. Uživatel v cloudu platí pouze za využití zdroje, to znamená, že ušetří finanční prostředky spojené s provozem vlastního hardwaru. Využívání cloudu otevírá vývojářům nové možnosti pro řešení problému. Díky dostupnosti cloudových zdrojů se mohou vývojáři zaměřovat především na psaní kódu, místo řešení problému s infrastrukturou.

Naopak na druhé straně jsou ve firmách provozovány aplikace označované jako legacy. Tyto aplikace jsou součástí velkých systémů, které zajišťují kritickou funkcionalitu. Jedná se například o velké bankovní, armádní či informační systémy. Obrovským problémem těchto aplikací je jejich architektura. Tyto systémy byly navrhovány před několika desítkami let, proto je pro jejich správný běh nutný zastaralý operační systém a hardware. Aplikace bývají napsány v již nepodporovaných verzích programovacích jazyků a jejich zdrojové kódy bývají velmi složité na udržování. Jedná se o komplexní systémy, které byly upravovány spoustou rozdílných vývojářů. Dalším důležitým faktorem je dokumentace, která také nemusí být jednotná pro stejné verze aplikace. Problematické bývá testování aktuální verze a change management, protože tyto komplexní systémy bývají často napojeny na další systémy. Pak může i jedna malá změna v aplikaci změnit výsledné chování systému jako celku. Pro tvorbu těchto systémů byly používány neagilní vývojové metodiky, které neumožňovaly dostatečnou flexibilitu a komunikaci během vývoje.

Hlavní motivací proč migrovat tyto aplikace na novější technologie a architektury je jejich udržitelnost. Jak již bylo zmíněno výše, tyto legacy aplikace převážně zajišťují chod důležitých systémů pro korporace a velké organizace. Čím je technologie starší, tím je nutné investovat více finančních prostředků k jejímu udržení, což se pro-

---

jeví i na zvýšených nákladech na dodatečné rozšíření aplikace. Pro udržování těchto aplikací jsou pak organizovány specializované týmy, které se zabývají pouze udržováním běhu aplikace a nasazováním nových či opravených verzí. Tyto verze bývají ovšem nasazovány velmi pomalu, především kvůli složitosti a náročnosti provést aktualizaci bez výpadků. Problém s pomalým nasazováním nových verzí vznikl v minulosti, kdy se světy vývojářů aplikací a systémových administrátorů začaly od sebe víc a více vzdalovat. Každý specializovaný tým se zabýval pouze svou danou částí, například programátoři končili svoji práci vytvořením instalátoru a otestováním aplikace v testovacím prostředí. Vývojáři se dále nezabývali problémem, jak funguje nasazená aplikace na produkčních serverech s reálnými daty. Naopak druhá strana, kterou zastávali systémoví inženýři, byla k novým verzím skeptická. Důvodem byla především komplexita a nové problémy (bugy), se kterými přicházely nové verze, které byly nasazovány. Problémy poté měli administrátoři především při údržbě aplikací a infrastruktury, protože jejich týmy byly odpovědné za funkčnost aplikací na produkčních serverech. Tento proces vedl k celkově obtížnému udržování aplikace. Proto se zhruba v letech 2006–2007 začaly v IT komunitě objevovat hlasy a argumenty poukazující na skutečnost, že současné řešení není optimální a že je potřeba vymyslet nový způsob propojení práce systémových administrátorů a vývojářů aplikací.

DevOps je přístup k vývoji a správě softwaru. Vznikl spojením dvou anglických slov: development (vývoj) a operation (správa). Tento přístup je založen na komunikaci mezi vývojáři a administrátory, kteří aplikaci v produkci spravovali. Celý tento přístup je založen na zrychlení vývoje aplikace a na snížení času, který je nutný k doručení nové verze aplikace či aktualizace ke koncovým zákazníkům. Vznik tohoto přístupu je datován kolem roku 2008 a je spjat s rozmachem cloudu a přístupem jako služba (as a service). Na rozdíl od agilního vývoje DevOps nemá žádný oficiální manifest nebo definici, každá firma či tým si ho vykládá po svém. DevOps přístup bourá komunikační bariéru mezi vývojáři softwaru a lidmi, kteří se o daný běh softwaru starají. Jde především o zodpovědnost týkající se provozu daného softwaru aplikace, už není možno označit, že za chyby v aplikaci může pouze jedna strana, veškeré problémy by měly být řešeny formou spolupráce a dialogu mezi týmy. Pomocí tohoto přístupu lze mnohem efektivněji zaručit rychlé řešení problémů na produkci.

Cílem této diplomové je zanalyzovat a přesunout vybranou legacy aplikaci do kontejnerového řešení. Výsledek práce spočívá ve vytvoření funkčních požadavků na kontejnerovou aplikaci, navržení architektury a vytvoření funkčního prototypu dané aplikace v kontejnerovém prostředí. V druhé kapitole je představen koncept kontejnerizace a základní technologie spojené s tímto typem virtualizace. Třetí kapitola se zabývá pojmem legacy aplikace a jsou zde popsány problémy monolitického přístupu, ten je poté porovnán s kontejnerovým řešením. Ve čtvrté kapitole je představena aplikace, která bude



---

převedena do kontejnerového řešení. V páté kapitole jsou uvedeny funkční požadavky na kontejnerovou aplikaci a její architekturu. Zároveň se tato kapitola zabývá implementací aplikace do kontejnerového prostředí. V rámci šesté kapitoly je provedeno testování a porovnání aplikace na jednotlivých architekturách. V závěru diplomové práce je provedeno závěrečné shrnutí.

V práci se vyskytuje mnoho anglických výrazů a termínů. Jedná se zejména o slova, která nemají český ekvivalent, nebo jejich překlad nedává v kontextu smysl. Proto tato slova nebudou přeložena a budou ponechána v původním znění tak, jak se běžně v praxi používají. Komplikovanější termíny a zkratky budou vysvětleny v slovníku cizích výrazů, který je přiložen v závěru práce.

## 2. Kontejnerizace

Kontejnerizace je jeden z typů přístupu k virtualizaci. Nejedná se o klasickou virtualizaci, pomocí které jsou emulovány hardwarové zdroje, ale o tzv. virtualizaci na úrovni operačního systému. Celý princip kontejnerové virtualizace je postaven na izolaci jednotlivých zdrojů operačního systému. Tato virtualizace používá k izolaci systémové zdroje v jádru operačního systému. Na Linuxu se jedná převážně o cgroups, které slouží k izolaci CPU a paměti, pro izolaci procesů pak slouží namespaces. Tento typ virtualizace se stal v posledních letech velmi oblíbený především kvůli své jednoduchosti a rychlosti, a to hlavně díky projektu Docker. S nárůstem popularity kontejnerové virtualizace se začala znovu více používat architektura mikroslužeb.

### 2.1. Historie kontejnerové virtualizace

I když kontejnery zažívají v posledních třech letech obrovský rozmach, nejedná se zdaleka o nejnovější technologii. První náznaky kontejnerů se objevily během roku 1979, kdy byl do UNIXU v7 naimplementován program chroot [1], který jako jeden z prvních ukázal, jak efektivně izolovat procesy a odlišné souborové systémy. Z UNIXU se chroot dostal na systém BSD v roce 1982. O téměř dvacet let později v roce 2000 se objevil projekt FreeBSD jail [2], který byl postaven nad konceptem chrootu, rozšiřoval jeho možnosti práce s procesy a souborovým systémem. Bohužel díky malému rozšíření operačního systému FreeBSD se tento projekt neujal. V roce 2001 byl představen projekt Linux VServer [3], který byl postaven na stejném principu jako předchozí projekt FreeBSD jails. Projekt se sice nestal v komunitě populární, ale je v linuxovém jádře udržován dodnes. V roce 2004 se kontejnerovou virtualizací začala zabývat také firma Oracle, která ve svém operačním systému Solaris vytvořila projekt Solaris Containers často označován jako Solaris Zones [4]. Celý projekt byl postaven nad konceptem zón, pomocí kterých byla tvořena jednotlivá separace. Mezi roky 2004-2008 se pokoušela spousta projektů prorazit se svým kontejnerovým konceptem. Hlavními hráči zde byly OpenVZ. Jednalo se o projekt, který byl opět postaven na linuxovém jádru [5]. Ovšem zdrojové kódy které pro své řešení použili, se nikdy oficiálně nedostaly do linuxového jádra, bylo nutno tedy jádro patchovat. Další projekt Process Containers, byl vytvořen společností Google a byl opět postavený na linuxovém jádře. V roce 2007 byl tento pro-

jekt oficiálně přejmenován na control groups (cgroups) [6], převážně kvůli dvojsmyslu slova kontejner, které bylo použito v předchozím názvu. Tato změna byla přidána do linuxového jádra během roku 2007 a v lednu roku 2008 byl projekt oficiálně zahrnut do verze jádra 2.6.24. Projekt Process Contraines určil směr, jak by měla izolace procesů v linuxovém jádře vypadat. Ve stejném roce se objevil projekt LXC (LinuX Containers) [7], kterým byla první kompletní lehkotonažní technologie zaměřená na správu kontejnerů. Technologie byla postavená nad cgroupami a namespaces a přidána do linuxového jádra, takže pro svůj běh nebylo třeba žádných externích změn. V roce 2013 se na kontejnerovou scénu opět vrací Google, který otevřel zdrojové kódy ke svému kontejnerovému řešení zvanému Let Me Contain That For You (LMCTFY) [8]. Cílem tohoto projektu bylo poskytnout abstrakci na kontejnery a možnost spravovat je skrze API. V roce 2015 bylo rozhodnuto, že veškerá logika a koncepty tohoto projektu budou sloučeny s projektem Docker a knihovnou libcontainer, kterou Docker používal. Samotný Docker projekt, který masově rozšířil kontejnerovou virtualizaci a nastartoval tak novou éru vývoje softwaru, vznikl jako projekt Solomona Hykese, který tento projekt vytvořil ve firmě dotCloud. Firma posléze převzala projekt za svůj, změnila svoje jméno na Docker a začala se věnovat pouze této technologii a orchestraci okolo ní [9]. Docker se stal standardem a vybudoval si téměř monopol. V následujících letech se začala objevovat další řešení jako například, rkt od firmy CoreOS (dnešní Red Hat), která vyvinula svůj systém kontejnerů. Rkt fungoval i jako runtime a dokázal spouštět i Docker kontejnery. Volba na tvorbu vlastního kontejner enginu se nezrealizovala díky dlouhému čekání na schvalování patchů do Docker projektu. V roce 2015 pak vznikla organizace Open Container Initiative (OCI) [10], která nastavila a popsala standard pro kontejnery. Snahou bylo mít kontejnery univerzální s možností jednoduché změny jednotlivých řešení. S rozmachem orchestrátoru Kubernetes se začaly v roce 2017 objevovat kontejnery a runtime určené pouze pro tento orchestrátor, jednalo se o projekty jako např. Containerd a Podman. Ve stejném roce Docker oznámil, že bude také přispívat do Kubernetes a nebude upřednostňovat pouze svůj orchestrátor Swarm. Během nadcházejícího roku se Kubernetes stal populárním tak, že ho do svých produktů zařadily všechny tři hlavní firmy působící v oblasti veřejného cloudu (Azure, Amazon a Google).

## 2.2. Kontejnery

Princip kontejnerů a kontejnerové virtualizace je založen na izolaci jednotlivých částí aplikace na sobě nezávisle fungujících celků. Pro svůj běh kontejnery používají izolované zdroje, jako jsou CPU, disk nebo síť, pomocí kterých je kontejner oddělen od zbytku systému. Kontejner si lze velmi jednoduše představit jako klasický kontejner

na přepravu zboží. Je vždy uzavřen a měl by fungovat sám o sobě. Do kontejneru lze velmi jednoduše zabalit nainstalované aplikace i s jejich podpůrnými knihovnami a nastavením, celé kontejnery pak lze velmi snadno přenášet mezi různými prostředími.

Obrovskou výhodou kontejnerů je možnost spuštění několika stejných služeb, jako jsou například databáze, v jednom prostředí bez možnosti, aby se služby navzájem ovlivňovaly. Další výhodou je bezpečnost veškerých akcí, které jsou spuštěny v kontejneru. Akce jsou odděleny od operačního systému hosta. Kontejnery také řeší aplikační problém se závislostmi. Díky kontejnerovému řešení lze nainstalovat potřebné balíčky a knihovny přímo do kontejneru. Je tedy možné aplikaci v produkčním nastavení vyzkoušet přímo na stroji, na kterém je aplikace vyvíjena. Tato skutečnost značně zjednodušuje práci vývojářům, kteří si mohou velmi snadno a rychle otestovat svou aplikaci. Obdobně pomáhá při práci administrátorům, kteří se nemusí bát o zpětnou kompatibilitu a nestabilitu aplikace.

Kontejnery bývají často srovnávány s klasickými virtuálními stroji. Ovšem jedná se o úplně jiný princip virtualizace. V kontejneru by měl být spuštěn pouze jeden proces jako PID 1. Oproti plně virtualizovanému serveru, kde je proces PID 1 `init` nebo `systemd`, který má další procesy jako své potomky. Z tohoto důvodu je spuštění virtuálního stroje mnohem pomalejší a může trvat řádově v minutách, naopak spuštění kontejneru je záležitostí několika vteřin. Další výhodou kontejnerového přístupu je práce se zdroji, kontejner dokáže ke zdrojům přistupovat dynamicky na rozdíl od virtuálního stroje, který si musí před svým startem zdroje předalokovat, tyto změny poté většinou nelze jednoduše upravovat. Chybné počáteční nastavení zdroje pro virtuální stroj může mít negativní vliv na alokované zdroje, které reálně nikdy nevyužije. Díky tomuto principu jsou virtuální stroje vhodnější pro jiný typ workloadu než kontejner, například pro stavové aplikace, kde je dopředu zřejmé, jaké zdroje bude aplikace potřebovat. Při použití virtuálních strojů je spuštěn nový operační systém s vlastním jádrem. Ten je mnohem jednodušší zabezpečit [11]. Na druhé straně ve světě virtuálních strojů existuje stále ještě řada problémů, např. řešení vysoké dostupnosti aplikací nebo řešení konfigurace virtuálních strojů a jejich škálování.

**Tabulka 2.1.:** Porovnání kontejneru a VM

	Virtuální stroje (VM)	Kontejnery
Velikost:	Velká	Nízká
Škálovatelnost:	Pomalá - nutnost škálovat celé virtuální stroje	Rychlá - škálované jsou pouze kontejnery
Nasazení:	Pomalé - nasazení vyžaduje nastavit aplikaci a operační systém	Rychlé - aplikace již je nainstalována v images, které jsou spouštěny
Izolace:	Úrovni systému	Úrovni procesu
Přenositelnost:	Nižší	Vysoká
Granularita komponent:	Vysoká	Nízká
Bezpečnost:	Vysoká	Nízká

Pro kontejnerizaci existuje velké množství technologií, nejznámější z nich je Docker. Docker přišel s jednoduchým a srozumitelným konceptem pro snadné používání kontejnerů. Docker ovšem není jediná dostupná kontejnerová technologie. Do vývoje Dockeru se snažili přispívat vývojáři nejen z firmy Docker, ale i z jiných firem. Proto vznikaly nové projekty, např. projekt rkt od firmy CoreOS, který se technologií Docker nechal velmi inspirovat. Tento projekt se ve srovnání s Dockerem zaměřoval na bezpečnost, zároveň byl s Dockerem kompatibilní, to znamená, že v rkt prostředí bylo možno též spouštět image vytvořené pro Docker. Během roku 2015 byla založena iniciativa OCi, která měla za úkol standardizovat svět kontejnerů. Zakládající členové byly společnosti Docker a CoreOS. Hlavní standardy, které jsou uveřejněny pod OCi, jsou standardy pro image a runtime [10]. Image standard popisuje, jak by měla vypadat image, ze které je možné spustit kontejner. Runtime popisuje, jak má být kontejner spuštěn a jak má vypadat jeho životní cyklus. Tyto dva standardy otevřely cestu projektům, jako jsou Crio-O a podman, které byly převážně vyvíjené společností Red Hat, která se snažila kolem kontejnerů postavit svůj vlastní business. Zajímavostí je, že podle zdroje [12] Red Hat byl již v roce 2016 v kontejnerovém businessu finančně mnohem úspěšnější a dokázal na technologii Docker vydělat mnohem více peněz než právě firma Docker. Od začátku bylo cílem Red Hatu dostat Docker ze svých produktů úplně, aby měl nad svými produkty větší kontrolu. V současnosti již je možné vytvářet a spouštět kontejnery bez použití technologie Docker.

Z průzkumů prováděného zdroji [13] a [14] vyplývá, že nejrozšířenější a nejpoužívanější kontejnerizační řešení je Docker. Architektura Docker kontejneru je postavena

na systému vrstev. Každá vrstva vždy obsahuje pouze data z akce, která byla na dané vrstvě vykonána. Všechny vrstvy kromě nejvyšší (tzv. kontejnerovou vrstvy) jsou v režimu Read-Only, to znamená, že do těchto vrstev není možno zapisovat data, ale pouze z nich číst. Veškeré akce, které jsou prováděny v kontejneru po jeho spuštění, probíhají v kontejnerové vrstvě. Tento vrstvý systém lze použít díky unionFS [15], speciálnímu typu souborového systému, který Docker využívá. Pro tvorbu jednotlivých kontejnerů se používají takzvané Docker image. Image je předpis jenž obsahuje aplikaci se všemi závislostmi na spuštění, například MySQL, Redis nebo runtime pro programovací jazyky (JRE, Node). Prostřednictvím těchto imgs se poté distribuují jednotlivé kontejnery. Jedná se o samostatnou spustitelnou jednotku, která obsahuje aplikaci nebo část aplikace i s jejími závislostmi. Pokud je potřeba vytvořit vlastní image, lze to pomocí speciálního souboru zvaného Dockerfile.

**Kód 2.1:** Dockerfile, zdroj: vlastní tvorba

```
1 FROM openjdk:8-jre-alpine3.8
2
3 COPY ./target/file-service.jar /app/
4 RUN mkdir /filevol
5 RUN mkdir /logs
6 CMD ["java", "-Xmx200m", "-jar", "/app/file-service.jar"]
7
8 EXPOSE 7000
```

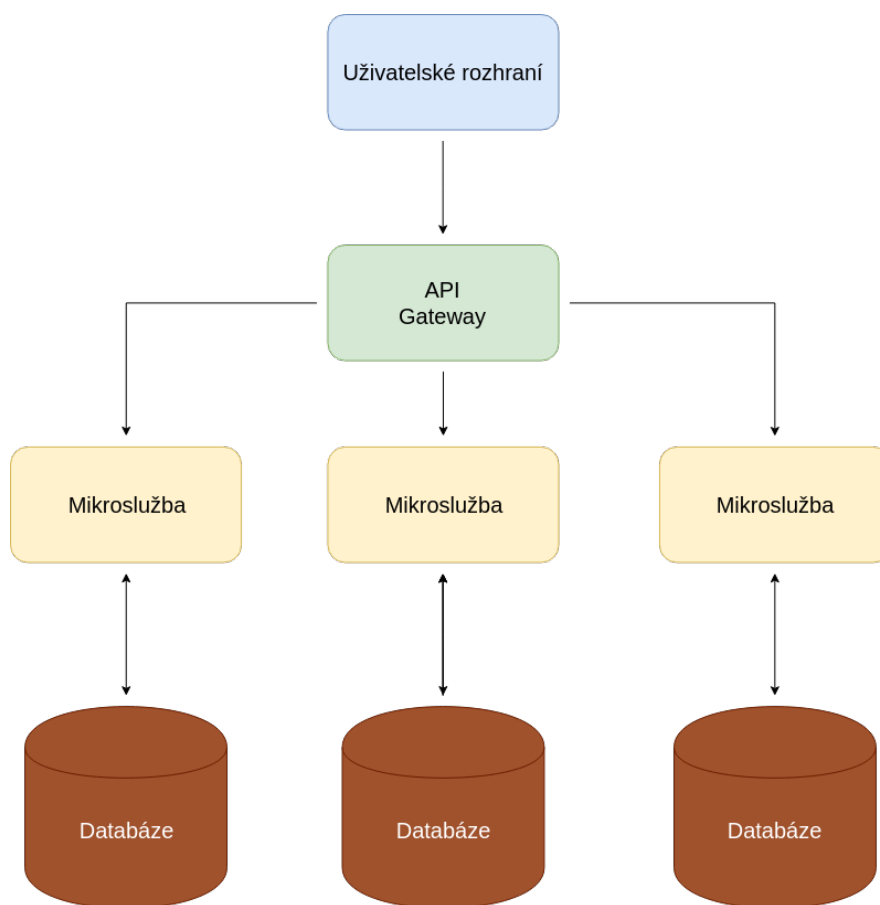
Základním stavebním kamenem Docker, jak již bylo zmíněno, je Dockerfile, jehož strukturu lze vidět v ukázce kód 2.1. Jedná se o textový soubor, který má pomocí klíčových slov provádět akce potřebné k sestavení image. Každý Dockerfile začíná klíčovým slovem *FROM*, které specifikuje základní image, ze které se vychází. Hlavním požadavkem na tyto image je především malá velikost a potřeba dávat do kontejnerů pouze data nezbytně nutná pro jejich běh. Jako základní image pro kontejnery se často využívá Alpine Linux, což je lehkotónážní linuxová distribuce, která má v základní verzi pouze 5 Mb [16]. Tato distribuce je zúžena o všechny nepotřebné knihovny a nástroje. Ve zmiňované ukázce je použit Alpine, který má již v sobě předinstalované prostředí pro spouštění jar souborů. Tato image pochází z repozitáře Docker Hub, jenž je nejpopulárnějším repozitářem pro Docker image. Ve druhém kroku pomocí příkazu *COPY* dojde k překopírování souborů z lokálního souborového systému do image. V příkladu je nakopírován jar balíček obsahující aplikaci. Jako další je použito klíčové slovo *RUN*, které slouží ke spuštění různých shellových příkazů, v příkladu je pomocí něho volán příkaz *mkdir*, který vytváří adresáře vyžadované aplikací. V praxi se tento příklad často používá na instalaci balíčků z repozitářů jednotlivých linuxových distribucí. Předposlední příkaz *CMD* určuje, jak bude probíhat start kontejneru, v příkladu pak spouští

překopírovaný jar soubor pomocí JRE. Pokud je nutné pro aplikaci před spuštěním vykonat nějakou složitější logiku, například zkontrolovat práva na souborech v kontejneru či připravit aplikaci (např. dokonfigurovat), tak se používá příkaz *ENTRYPOINT*. Entrypoint dává možnost ještě před startem spustit shellový skript s přidanou logikou. Poslední příkaz *EXPOSE* poté slouží k určení, na jakém portu bude aplikace vystavena. V příkladu je použit port 7000. Dále lze specifikovat i protokol, který bude ke komunikaci použit. Je možné vybrat z tcp nebo udp, ale pokud protokol není nadefinován, je vždy vybrán tcp. Image lze pak sestavit dvěma různými způsoby. První je pomocí Docker CLI a příkazem *docker build .*, druhý pomocí orchestrátoru Compose. Jak už bylo zmíněno výše, image jsou ukládány a přenášeny pomocí repozitářů, výchozí je Docker Hub. Práce s repozitářem je velmi jednoduchá, pracuje se s ním také pomocí Docker CLI a struktura příkazů je značně podobná verzovacímu systému Git.

## 2.3. Mikroslužby

Princip architektury mikroslužeb (Microservices) je velmi podobný Unixové filozofii, která vychází z konceptu mnoha malých jednoúčelových programů, kde každý vykonává pouze jednu činnost, zato spolehlivě. Z praktického hlediska to znamená, že aplikace, která používá princip mikroslužeb, by měla být rozdělena do jednotlivých zapouzdřených komponent, které vždy vykonávají pouze jednu akci. V rámci webové aplikace to může být například služba na přihlášení či vyhledávání. Základní rozdíl mezi mikroslužbami a monolitickým přístupem je dělení aplikace na jednotlivé části. Aplikace jsou lehce uchopitelné, mnohem lehce spravovatelné a nezávislé. Služby by měly být maximálně nezávislé, ať už se jedná o jejich zdrojový kód, nebo o možnost nasazení [17]. Jednotlivé služby by měly být navrženy tak, aby bylo možné jejich nové verze nasazovat a testovat nezávisle na ostatních službách, se kterými jsou propojeny. S konceptem mikroslužeb je nutné veškeré procesy automatizovat, protože se velmi jednoduše může stát, že počet jednotlivých mikroslužeb bude velmi rychle narůstat. Důležité je mít pod kontrolou jejich nasazování, testování a hlavně monitoring [18]. Jednotlivé služby mohou být nasazovány i nezávisle a při nasazení může i malá chyba v jedné službě způsobit globální problém v celé aplikaci. V chybně navržené architektuře může i chyba jedné služby způsobit výpadek celé aplikace. Proto je nutné jednotlivé komponenty držet co nejvíce izolované. Dále je nutné na celé řešení pohlížet jako na decentralizovaný systém, kde by v ideálních podmínkách výpadek jedné služby neměl ovlivnit zbytek. Často nebývá tento koncept pochopen správně a bývá pro všechny mikroslužby používána jedna databáze. V ideálním případě je nutné databáze rozdělit tak, aby mohly být lépe decentralizované, což bývá často problematické, protože je nutno zachovat jednotlivé vazby mezi databázemi. K dalším nevýhodám přístupu mik-

roslužeb patří problémy s API, obtížnost architektury aplikace a jednotlivé ovlivňování mikroslužeb navzájem. Čím víc se API dané mikroslužby používá, tím obtížnější je ho změnit. Nekompatibilní změna do API poté ovlivní veškeré další mikroslužby. Dalším problémem je často chybný návrh architektury. Architektura mikroslužeb není vhodná pro všechny typy aplikací. Především pro menší systémy, které nepotřebují dynamické škálování, je stále vhodnější monolitická architektura. I když svět mikroslužeb není zcela dokonalý, výhody převyšují. Velkou výhodou výsledku vývoje aplikace je jednoduchost, izolovatelnost, testování, vlastnictví a škálovatelnost. Aplikace postavené na konceptu mikroslužeb se výrazně lépe škálují, lze u nich škálovat i pouze jednotlivé části. Viz schéma architektury mikroslužeb na obrázku 2.1.



Obrázek 2.1.: Schéma mikroslužeb, zdroj: vlastní tvorba

## 2.4. Orchestrátory

Hlavní problém v kontejnerové virtualizaci byla správa a lifecycle management (LCM) samotných kontejnerů. Jelikož kontejnerová aplikace bývá často složena z několika de-



sítek kontejnerů, je nutné proces správy automatizovat co nejvíce, protože není možné jednoduše řídit kontejnerové aplikace ručně. Bylo nutno vyřešit, jak naimplementovat orchestrační vrstvu nad kontejnerovým řešením. Na kontejnery byl kladen požadavek na vyšší abstrakci nad jednotlivými zdroji za účelem zjednodušení principu ovládání a nastavování oproti klasickým virtuálním strojům. Důraz byl kladen především na networking, aby komunikace mezi jednotlivými kontejnery probíhala pouze přes DNS jména, která lze naimplementovat pomocí DNS záznamů. Záznamy slouží ke zjednodušení práce vývojářů, kteří pak nemusí složitě řešit síťové problémy. Dalším důvodem pro používání orchestrátorů je spojování kontejnerů do vyšších logických prvků. Například databázi je často vhodné spouštět spolu s dané databázovou cache. Takže při škálování dané komponenty je služba škálována také s podpůrnou službou. Dále vznikla potřeba orchestrovat i stavové aplikace. Je tedy potřeba vyřešit problém připojování svazků dat do příslušných kontejnerů. A to nejen při běhu, ale také při výpadku kontejneru.

V orchestrátoru lze odlišit dva základní termíny, a to orchestrace a scheduling. Scheduling je algoritmus zodpovědný za spouštění kontejnerů a orchestrace slouží k jejich řízení. Scheduler vždy bývá implementován jako součást orchestrátoru a zajišťuje rovnoměrné spouštění kontejnerů po hypervisorech a má přehled o zdrojích, které a kde spustit. Scheduler by měl fungovat i v rámci disaster recovery scénáře. Pokud se na hypervisoru vyskytne chyba, scheduler to zjistí a automaticky spustí kontejnery na jiném serveru. Orchestrace dále zajišťuje konzistenci v clusteru a sleduje počet kontejnerů. Pokud v clusteru nějaký chybí, tak spustí nový, pokud přebývá, tak ho zastaví. Z výše uvedených důvodů vyplývá, že ve velkém aplikačním clusteru provoz kontejnerů bez orchestrátoru nemůže fungovat. Samotný kontejner je pouze stavební jednotka většího celku. Stejný problém řešila i firma Google, která s kontejnery experimentovala ještě před kontejnerovým boomem v roce 2014. Podle zdroje [19] již v roce 2014 bylo v Googlu spouštěno každý týden přes dvě miliardy kontejnerů, proto se společnost Google rozhodla vybudovat vlastní systém na orchestraci kontejnerů zvaný Borg. V současné době firma Google provozuje své veškeré aplikace v kontejnerech.

## 2.5. Stavové a bezstavové aplikace

S orchestratory a kontejnery přichází další problém, který bylo potřeba vyřešit, a tím jsou stavové aplikace. Stavové aplikace jsou typ aplikací, které si pro svůj běh potřebují držet data. Může se jednat například o databáze, které drží data pro webový backend. Data v databázích jsou průběžně aktualizována se změnami ve webové aplikaci, proto je důležité tato data perzistentně ukládat mimo kontejner, aby nedošlo k jejich ztrátě. Kontejnery by měly být bezstavové a jít jednoduše restartovat či převytvořit bez jaké-

koliv ztráty dat. Oproti stavovým aplikacím jsou postaveny bezstavové aplikace, které si žádný aktuální stav nedrží, pracují s daty, které si aplikace bere z externího API. Obvykle jsou závislé na datech třetích stran, která nejsou uložena v aplikaci. Bezstavová aplikace tedy není závislá na žádných datech uložených v aplikaci. Tento fakt dává bezstavovým aplikacím ohromnou výhodu, dají se velmi rychle a efektivně horizontálně škálovat bez zásahu do aplikace. Dále lze velmi efektivně provádět rollbacky aplikace. Bezstavové aplikace jsou mnohem lépe udržitelné, což je velmi výhodné v kombinaci s architekturou mikroslužeb a kontejnerovým přístupem.

## 2.6. Porovnání orchestrátorů

Před porovnáním orchestrátorů je nutné představit ty nejpoužívanější. Je potřeba zmínit, že v současnosti všechny nástroje okolo kontejnerové virtualizace, ať už se jedná o orchestrátory, runtime či kontejnery samotné, jsou open source a jsou dostupné zdarma.

### 2.6.1. Docker orchestrátory

I samotná firma Docker zjistila, že provoz kontejnerů a kontejnerových aplikací bez orchestrátorů je velmi obtížný. Proto se rozhodla vytvořit hned dvě řešení, pomocí kterých je možné kontejnery ovládat. Prvním řešením je Docker Compose, tento nástroj není oficiálně označován Dockerem jako orchestrátor a splňuje nadefinované požadavky pro orchestrátory. Tento projekt je oproti ostatním velmi jednoduchý, nemá žádnou složitou architekturu, skládá se pouze z jednoho binárního souboru, pomocí kterého jsou spouštěny Compose soubory. Compose soubor je dokument ve formátu yaml, ve kterém je popsáno, jak by daná kontejnerová aplikace měla být vytvořena a jak by jednotlivé zdroje měla používat (volumes, síťové porty). Z Compose definice lze též přímo volat sestavování souboru Dockerfile. Compose díky své jednoduchosti není určen pro běh kontejneru na produkci, ale slouží primárně pro vývojáře, kteří si s jeho pomocí jsou schopni spustit vyvíjenou aplikaci. Často také bývá Compose, díky své rychlosti, zaintegrovan do testovacích prostředí.

**Kód 2.2:** Docker Compose, zdroj: vlastní

```
1 version: '3'
2 services:
3   database:
4     image: mysql
5     restart: always
6     ports:
7       - 3306:3306
8       - 33060:33060
```

```
9     volumes:
10     - ./data:/var/lib/mysql
11     environment:
12     MYSQL_ROOT_PASSWORD: password
```

Ukázku Compose souboru je možno vidět na ukázce kódu číslo 2.2. Compose musí začínat verzí Composu, poté je vždy definován blok služeb, pod kterým jsou nadefinované jednotlivé komponenty aplikace a jejich vlastnosti. V příkladu je pro demonstraci použita databázová aplikace MySQL. Je specifikováno, že jako image je využita MySQL, restart určuje, kdy se má kontejner restartovat. Pod ports je specifikován list portu a na jaké hosty porta mají být namapovány. V příkladu jsou použity stejné porty jak pro hosta, tak pro kontejnerovou aplikaci. Volumes specifikují část kontejneru, která má ukládat své soubory na souborový systém hosta. V příkladu je namapovaná cesta `/var/lib/mysql`, ve které jsou uloženy data databáze. Takže při použití nejsou data z kontejneru databáze po restartu ztracena. Poslední část aplikační definice tvoří environmentální proměnné, které slouží pro konfiguraci samotné aplikace. Jediná proměnná, kterou je nutné definovat, aby se databáze spustila, je `MYSQL_ROOT_PASSWORD`, což je heslo pro uživatele root.

Druhým řešením je Docker Swarm. Jak již název napovídá, projekt se zaměřuje na orchestraci kontejnerů a byl vytvořen společností Docker. Na rozdíl od Docker Compose se Swarm zaměřuje na běh kontejnerových aplikací v produkčních prostředích. Je složitější a obsahuje spoustu důležitých vlastností, které Compose nemá, jako jsou například škálování, load balancing atd. Architektura Swarm clusteru je rozdělena na dvě základní komponenty, a to na workers nody a managers nody. Workers nody slouží k provozu kontejnerů a managery pak pro jejich správu. Jednotlivé role se dají kombinovat. Server může obsahovat manager i worker roli zároveň. Pouze přes manager lze přistupovat do clusteru a spouštět v něm jednotlivé příkazy přes Docker CLI. Manager zajišťuje dále několik rolí: scheduling (spouštění kontejnerů), škálování, health check, DNS a networking. Bezpochyby největší výhodou tohoto řešení je jeho provázanost s Docker ekosystémem. Swarm orchestrátor lze velmi rychle a lehce nainstalovat a spravovat. Samotná instalace probíhá pomocí docker-enginu, Swarm je totiž součástí instalačního balíčku s Dockerem. Jednotlivé produkty se dají jednoduše spojovat a ovládat. Například pomocí Docker Compose lze spouštět předdefinované compose soubory přímo ve Swarmu [20].

## 2.6.2. Kubernetes

Dalším vybraným orchestátorem je Kubernetes (zkráceně k8s), tento nástroj byl vytvořen Joe Bedou [21] za jeho působení ve firmě Google. Předlohou pro tento nástroj byl

interní Google orchestrátor zvaný Borg. K8s je také jediný z orchestrátorů, který je členem CNCF (Cloud Native Computing Foundation), tato organizace spolu s velkou vývojářskou otevřeností vybudovala okolo tohoto orchestrátoru obrovskou uživatelskou komunitu. Komunita měla značný vliv na celkový rozvoj orchestrátoru. K8s je velmi modulární a nabízí hned několik výměnných modulů ať pro síťové řešení (CNI - Container Network Interface), nebo pro storage (CSI - Container Storage Interface). V k8s clusteru jsou servery rozděleny do dvou skupin, master a minion. Master je složen z několika komponent, hlavní z nich je API server, který slouží ke komunikaci. API je využíváno například k8s CLI klientem, další komponenta je Etcd, key-value databáze, která si ukládá aktuální stav clusteru, dále je zde manager se schedulerem, které zajišťují spouštění a běh kontejnerů. Na straně minionu se nachází kube-proxy, který spravuje networking pro pody, a kubelet služba, která slouží k propojení minionu s api serverem. Nad těmito základními komponentami jsou spouštěny k8s objekty. Základní jednotkou v k8s není jednotlivý kontejner, ale pod. Pod je logická jednotka složená z kontejnerů, tyto pody určují logické celky, které jsou poté spolu škálovány (např webová aplikace a cache). Daný pod je bezstavový, je spouštěn vždy na jednom serveru, kontejnery v něm sdílejí síťové zdroje, IP adresu a komunikují mezi sebou pomocí portu. Do kontejnerů lze také připojovat volume s daty. Jednotlivé pody jsou provázány pomocí k8s služeb, které zajišťují nejen komunikaci uvnitř clusteru, ale je možno pomocí loadbalanceru a ingress služby vystavit také mimo cluster do internetu. Pro stavové aplikace je v k8s objekt statefulset, který má při vytvoření nadefinovaný datový volume s cestou, do které se připojí. Volume je perzistentní, to znamená, že při ztrátě či smazání kontejneru spuštěného statefulsetu nebude smazán. Pro spuštění nových kontejnerů a škálování slouží komponenta replica set (dříve replication controller), která sleduje aktuální počet kontejnerů v clusteru a ukládá si stav do Etcd. Pokud nějaký server s kontejnery postihne výpadek, tak to replicaset zjistí a spustí kontejnery, které byly ztraceny na jiném serveru. Jednotlivé aplikace jsou v k8s odděleny pomocí namespaces. Namespace si lze představit jako další virtuální cluster, ve kterém pody mohou komunikovat pouze v rámci namespace. K identifikování objektů v clusteru slouží label, ty ve formátu klíč–hodnota slouží k popisování a propojování objektů. Jednotlivé definice s objekty jsou pro k8s nadefinované prostřednictvím takzvaných manifestů. Manifest je soubor ve formátu yaml, ve kterém jsou blokově definovány jednotlivé objekty. Díky popularitě v komunitě se k8s stal standardem na poli kontejnerové virtualizace a rozšířil se do většiny velkých firem. Pro běh clusteru totiž není potřeba žádný dedikovaný tým, jak tomu většinou je u privátního cloudu, kdy jeden tým zajišťuje podporu pro ostatní vývojáře. V k8s bylo celé řešení navrhováno tak, aby fungovalo out-of-box (bez zásahu z venku) s cílem, aby se vývojáři mohli soustředit pouze na vývoj aplikací a nikoliv na infrastrukturu.

**Kód 2.3:** Kubernetes Manifest - Nginx deployment, zdroj: [22]

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5    labels:
6      app: nginx
7  spec:
8    replicas: 3
9    selector:
10     matchLabels:
11       app: nginx
12  template:
13    metadata:
14      labels:
15        app: nginx
16    spec:
17      containers:
18        - name: nginx
19          image: nginx:1.7.9
20          ports:
21            - containerPort: 80

```

Na ukázce kódu číslo 2.3 je zobrazen vzorový k8s manifest. V tomto případě jde o deployment manifest. Každý manifest má předem danou strukturu a obsahuje čtyři základní stavební bloky. Prvním je `apiVersion` a `Kind`, který popisuje objekt, jenž bude v k8s vytvořen, a verzi API, které k této akci využije. Dalším blokem jsou `metadata`, která slouží k unikátnímu označení objektu, pod `metadata` lze například vytvářet různé `labels` či anotace ve formátu klíč–hodnota. Podle těchto `metadata` lze dále objekty filtrovat nebo propojovat navzájem. Posledním blokem v manifestu souboru je `spec` část, neboli část týkající se specifikací objektu. Tato část je vždy pro každý objekt odlišná a obsahuje uvnitř další bloky s dalšími parametry. Vybraný objekt na příkladu je deployment pro Nginx. Ve `spec` části je možné sledovat, že Nginx bude spuštěn ve třech replikách, dále je zde patrné namapování `labelu` a definice pro `image`, která má být spuštěna.

### 2.6.3. Apache Mesos

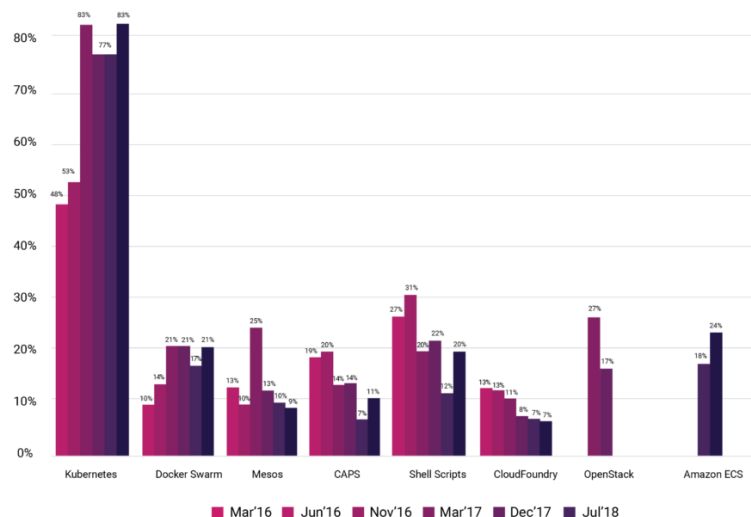
Apache Mesos oproti ostatním projektům není klasickým orchestrátorem. Jedná se spíše o operační systém pro datacentra, který poskytuje určitý level abstrakce nad výpočetními zdroji (CPU, RAM, disk, atd). Pomocí Mesos lze spravovat nejen zdroje na fyzických serverech, ale také na virtuálních strojích. Mesos funguje na architektuře

master-minion, kdy je jeden stroj určen k ovládání těch druhých. Jednotlivá izolace na této vrstvě je taktéž implementována pomocí cgroups. Komunikace mezi minion a master probíhá pomocí Rest API. Masteři jsou spuštěni v kvóru, které je zajištěno pomocí technologie ZooKeeper. Vždy je aktivní pouze jeden master. Masteři na základě dat z mimonu mají informace o jejich vytíženosti a na podkladě těchto informací rozdělují úkoly mezi jednotlivé miniony.

Pro kontejnerovou orchestraci se používá framework Marathon, což je kontejnerový orchestrátor, který je spuštěn nad Mesosem [23]. Mesos neslouží pouze jako orchestrátor pro Docker kontejnery, lze v něm spouštět nativně Mesos kontejnery, které jsou také jako Docker postaveny nad linuxovými technologiemi cgroups a namespaces. Obrovskou výhodou Marathonu je jeho provázanost s Mesosem, pro kontejnery například lze připojovat storage přímo z diskových zdrojů. Součástí Marathonu jsou dále služby pro service discovery, load balancing a metrics, které slouží jako metering endpoint, přes který lze monitorovat cluster. Mesos využívá v produkci řada velkých firem, jako jsou Samsung, Autodesk, eBay či Twitter.

#### 2.6.4. Ostatní

Jak již bylo zmíněno, hlavní příklon k orchestrátorům proběhl se začátkem k8s. Již v době, kdy k8s ještě nebyl dostupný, velké startupy už měly potřebu nějaké orchestrace, a vznikaly proto vlastní implementace orchestrátorů. Například projekt Helios od švédské firmy Spotify, který v současné době už není ve fázi vývoje, ale je pouze udržován a jsou do něj implementovány jen opravy. Projekt bude udržován pouze po dobu, než ve Spotify naplno přestoupí na k8s. S obdobným projektem na orchestraci přišla i firma CoreOS, jednalo se o projekt Fleet, který byl postaven nad technologiemi Etcd a systemd. I vývoj tohoto projektu byl začátkem roku 2018 po akvizici CoreOS [24] zastaven a bylo doporučeno přejít na k8s.



Obrázek 2.2.: CNCF průzkum využití orchestrátorů, zdroj: [25]

### 2.6.5. Souhrn

Orchestrátory lze porovnávat z mnoha různých úhlů, jako jsou například množství vlastností, nebo kolik dokáží spustit kontejnerů najednou. Nejdůležitější faktorem je však komunita, která se okolo dané technologie pohybuje. Čím větší komunita, tím je nástroj rychleji vyvíjen a je kladen větší důraz na jeho stabilitu. V tomto ohledu jednoznačně dominuje k8s, jenž je nyní brán jako standard pro orchestraci a je běžně součástí komerčních produktů. V současné době má k8s na githubovém repozitáři téměř 52 000 hvězd uživatelů a celkový počet kontributorů do projektu přesáhl 2 100 vývojářů. Je velmi těžké zjistit přesné číslo, kolik firem využívá daný orchestrátor v produkčním prostředí, poskytovatelé cloudů takové statistiky nezveřejňují. Organizace CNCF každoročně pořádá průzkum týkající se Cloudu a kontejnerů, kde je patrné viz obrázek 2.2, že k8s je nejpopulárnější orchestrátor. V červenci roku 2018 byl třikrát více používánější než konkurenční Docker Swarm. Z tohoto průzkumu také vzešlo, že k8s je nejpoužívanější projekt z CNCF, který je provozován v produkčním nasazení. Proto bude v této práci použit k orchestraci migrované aplikace právě orchestrátor k8s.

## 2.7. Orchestrace workloadu

K8s je nástroj na provoz a lifecycle kontejnerů, ale sám o sobě neřeší definici a provoz jednotlivých aplikací. Standardní cestou pro vytváření aplikací a objektů v k8s jsou manifesty. Manifest je soubor ve formátu yaml. Přístup a práce s manifesty není složitá, ale s přibývajícím počtem komponent v aplikaci a objektů je velmi obtížné tyto manifesty

přehledně udržovat a spravovat. Pro vyřešení tohoto problému bylo zapotřebí vytvořit další úroveň abstrakce nad k8s, která zjednoduší jak verzování, tak provoz aplikací na produkčních clusterech. Celá myšlenka abstrakcí je postavená nad spojování aplikací do funkčních celků, například pokud chce uživatel použít nějakou stávající komponentu, například databázi, nemusí si všechny definice vytvořit sám, ale může použít již částečně hotové řešení. Tato problematika byla řešena v různých projektech s odlišnými řešeními. Nástroje, které tyto operace umožňují používat, by se daly označit jako konfigurační management pro k8s. O zjednodušení zápisu manifestů se opět snaží řada projektů, celkem existuje přes 60 projektů, které se zabývají konfigurováním workloadu. Do výběru byly vybrány projekty s největší komunitou. Správný nástroj by měl splňovat následující požadavky:

- **Deklarativnost** – Nástroj by měl vytvářet konfiguraci, která je jednoznačná a nezávislá na platformě, či závislá na systému [26].
- **Čitelnost** – Čitelnost předpisu aplikace: manifest jako takový je sice vhodné řešení, ale pokud má aplikace delší list, je snadné se v předpisu ztratit. Definice by měla být lehce čitelná nejen pro stroj, ale i pro člověka.
- **Flexibilita** – Nástroj by měl podporovat několik způsobů užití.
- **Udržovatelnost** – Nástroj by měl být jednoduchý na používání a kód, který je nástrojem vytvořen, by měl být lehce upravitelný a znovupoužitelný.
- **Verzovatelnost** – Ve světě mikroslužeb je nutno posuzovat prostředí a aplikace jako živoucí celek, který se v závislosti na čase mění. Proto je nezbytné uchovávat historii, především z důvodu výpadků a chybám aplikace či clusteru. Podle správného verzování je možné snadno zjistit, kdo v clusteru udělal nějakou změnu a co se stalo předtím, než aplikace přestala správně pracovat.

### 2.7.1. Helm

Helm je projekt, který je často označován jako balíčkovací systém pro k8s, který měl velký vliv na jeho rozšíření. Díky helmu lze velmi jednoduše instalovat aplikace do k8s clusteru. Projekt byl vyvinut firmou Helm, kterou v roce 2017 koupil gigant Microsoft [27]. Je složen ze dvou hlavních částí, klientská část je označována jako helm a tiller funguje jako serverová část. Tiller běží v k8s a vytváří a řídí aplikace. Právě serverová část helmu vzbuzuje v k8s komunitě velkou kritiku, protože tiller je spouštěn v clusteru s plnými administrátorskými právy. To znamená, že pokud se do clusteru dostane někdo s neoprávněným přístupem, může dojít k ohrožení bezpečnosti chodu k8s clusteru.



**Kód 2.4:** Kubernetes Manifest - Grafana service, zdroj vlastní

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    labels:
5      heritage: "{{ .Release.Service }}"
6      release: "{{ .Release.Name }}"
7      chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
8      app: {{ template "grafana.fullname" . }}
9      name: {{ template "grafana.fullname" . }}
10 {{- if .Values.service.annotations }}
11   annotations:
12   {{ toYaml .Values.service.annotations | indent 4 }}
13 {{- end }}
14 spec:
15   ports:
16     - name: http
17       port: {{ .Values.port }}
18       {{- if eq .Values.service.type "NodePort" }}
19       {{- if .Values.service.nodePort }}
20       nodePort: {{ .Values.service.nodePort }}
21       {{- end -}}
22       {{- end }}
23   selector:
24     app: {{ template "grafana.fullname" . }}
25     type: "{{ .Values.service.type }}"
26 {{- if eq .Values.service.type "LoadBalancer" }}
27   {{- if .Values.service.loadBalancerIP }}
28   loadBalancerIP: {{ .Values.service.loadBalancerIP }}
29   {{- end -}}
30   {{- if .Values.service.loadBalancerSourceRanges }}
31   loadBalancerSourceRanges:
32   {{- range .Values.service.loadBalancerSourceRanges }}
33   - {{ . }}
34   {{- end }}
35   {{- end -}}
36 {{- end -}}

```

Hlavní myšlenkou helmů bylo dodat do k8s stejnou flexibilitu v instalování aplikací, jako tomu je u balíčkovacího systému v Linux. Stačí napsat jeden příkaz a Helm dokáže nainstalovat jakoukoliv aplikaci. Aplikace helmů jsou poté zabaleny do chartu. Chart si lze představit jako yaml šablonu. Vzorovou šablonu (helm šablonu) lze vidět na ukázce kódu číslo 2.4, na ukázce je zobrazena šablona pro k8s službu pro aplikaci Grafana.

Díky helmům se dají jednotlivé konfigurační možnosti parametrizovat tak, že pokud je potřeba daný parametr pro aplikaci nastavit, musí být parametrizován. Parametrizování je největší výhoda a zároveň nevýhoda helm šablonování. Jednotlivé charty jsou uloženy v repozitáři, odkud je pak helm instaluje. Helmový repozitář je velmi podobný linuxovému a pracuje s podobnými příkazy jako balíčkovací systém apt. Helm má na GitHubu oficiálně repozitář komunitních helm chartů, ve kterém jsou většinou open source aplikace jako Mariadb, Redis, Tomcat atd. A právě v tomto repozitáři lze najít chart pro Nginx [28], z kterého je patrné, jak nepraktické může být šablonování v praxi. Nginx má parametrizovanou většinu svých konfigurovatelných parametrů a než se uživatel v šabloně zorientuje, jaké parametry vlastně potřebuje, bude potřebovat mnohem víc času, než by si napsal vlastní manifest či helm chart. Proto si většinou každá komunita/společnost udržuje vlastní sadu helm chartů, které mají standardizované parametry pro jejich produkční využití.

Většina aktuálních problémů, se kterými se komunita Helmu v současnosti potýká, je spojena právě s komponentou tiller, všechny tyto problémy by měly být vyřešeny v nové verzi Helmu 3 [29]. Tato verze ve svém návrhu již nepočítá se serverovou částí a má přidat řadu nových vlastností, jako jsou lepší lifecycle pro aplikace, Lua skriptování atd. Verze Helmu 3 je v současnosti v rané části vývoje a ještě nebylo oznámeno žádné pevné datum, kdy by měla vyjít první stabilní verze.

### 2.7.2. Jsonnet

U Jsonnetu se nejedná o nástroj na tvorbu manifestů, ale spíše o programovací jazyk. Tento jazyk byl vyvinut Googlem, zpopularizován byl až s nástupem k8s. Jsonnet vychází z klasického JSONu a je rozšířen například o podmínky, cykly, proměnné atd. Díky své velké flexibilitě se stal hned součástí několika nástrojů, jako je Ksonnet, KubeCFG či Kapitan. Obrovskou výhodou je jeho použitelnost při šablonování. Pomocí něho je možno ve velmi krátké době napsat téměř cokoliv. Nevýhodou je složitá syntaxe a operátor bez praxe se jsonnetem jen těžko na první pohled zjistí, co daná šablona dělá.

**Kód 2.5:** Kapitan komponenta - využívající Jsonnet, zdroj: [30]

```
1 local kube = import "lib/kube.libjsonnet";
2 local kap = import "lib/kapitan.libjsonnet";
3 local inv = kap.inventory();
4 local server_containers =
    inv.parameters.gerrit.server.deployment.containers;
5 local gerrit_pvcs = import "./pvcs.jsonnet";
6 local gerrit_volume =
    kube.HostPathVolume(server_containers.gerrit.vol.rev.path)
```

```

7
8 local gerrit_container = kube.Container("gerrit") {
9   image: server_containers.gerrit.image.registry + "/" +
10     server_containers.gerrit.image.name + ":" +
11     server_containers.gerrit.image.tag,
12
13   volumeMounts: [
14     {
15       mountPath: "/var/gerrit/review_site",
16       name: "reviewsite",
17     },
18
19   {
20     GerritDeployment(name): kube.Deployment(name) {
21       spec+: {
22         template+: {
23           spec+: {
24             containers_+: {
25               gerrit: gerrit_container
26             },
27             volumes_+: {
28               reviewsite: gerrit_volume
29             },
30           },
31         },
32       },
33     },
34   }

```

Na ukázce kódu 2.5 je vidět implementace kapitan komponenty postavená na Jsonnetu. Kód uvedený v příkladu lze pomocí kapitan kompilátoru přeložit na k8s manifest. Tento manifest poté obsahuje deployment pro aplikaci Gerrit. V samotném jsonnetu lze používat rozšiřující knihovny. Knihovny jsou napsané v Jsonnetu, mají stejný formát, ale jsou uloženy v souborech označených libsonnet. V uvedeném případě se jedná především o knihovny spojené s k8s objekty. Knihovny jsou nadefinovány pomocí klíčového slova `import`. Pro zjednodušení a neopakování kódu jsou používány proměnné, které začínají klíčovým slovem `local`. Například na řádce 8 je vytvořena proměnná `gerrit_container`, která slouží pro ukládání definice kontejnerové images. Je složena ze třech dalších parametrů, které je možné konfigurovat a přetěžovat, jedná se o parametry kontejnerový repozitář, jméno image a tag. Jednotlivé JSON objekty lze spojovat do vyšších logických celků jako na řádce 20, kde je vytvářen již deployment objekt. Zde je vidět volání funkce z *kube.libsonnet* knihovny. Zápis deployment funkce je

strukturou velmi podobný tradičnímu manifestu, na rozdíl od klasického manifestu je možné používat nadefinované proměnné jako výše zmíněný `gerrit_container`. Jednotlivé JSON bloky jsou propojeny pomocí znaku `+`.

### 2.7.3. Kustomize

Kustomize je postaven nad myšlenkou DAM (Declarative Application Management) [26] a je zaměřen na zjednodušení práce s k8s manifesty. Projekt vznikl pod SIG-cli [31] a již od počátku byl vyvíjen tak, aby jeho syntaxe byla co nejbližší ke k8s. Na rozdíl od projektu helm se nejedná o šablonovací jazyk. Kustomize je jednoduchý nástroj na spojování jednotlivých manifestů rozdělených do logických celků. Tyto celky jsou mnohem přehlednější než jeden dlouhý manifest. Nad jednotlivými celky lze jednoduše tvořit úpravy. Základem každé aplikace používající kustomize je `kustomization.yml` soubor. V tomto souboru lze nadefinovat cesty manifestů. `kustomization.yml` může jednotlivé parametry v manifestech přepsat. Taková logika funguje i napříč adresáři, je možné například vytvořit adresář `base`, který bude obsahovat základní aplikaci, a nad ním zadefinovat například další adresáře `production` a `stage`, které budou dané manifesty přepisovat. Toto řešení je vhodné především pro vývojáře nebo pro práci více týmů na jedné aplikaci. Další vlastností, kterou lze v kustomize využít, je `patch` systém. `Patches` jsou části manifestu, které upravují část původní aplikace. Tyto `patches` lze jednoduše přidat do Kustomize a ten je při renderování propíše do výsledného manifestu. Ukázku `kustomization` souboru je možné sledovat na ukázce kódu číslo 2.6. V příkladu je použit kustomize pro wordpress aplikaci, v sekci `base` je vidět, že výsledný manifest bude složen ze dvou adresářů, které obsahují další `kustomization` soubory s manifesty. Dále jsou zde `patchesStrategicMerge` a `vars`. První jsou `patches`, které přepíše hodnoty z `base` adresářů, a `vars` poté určuje základní proměnné pro `patch` soubor. Jednotlivé sestavení výsledného manifestu probíhá pomocí příkazu `kustomize build <cesta na adresář s kustomization.yml>`. Toto je velmi výhodné řešení, pokud do provozu aplikace zasahuje více lidí. Kustomize plně podporuje integraci s gitem, lze tedy nasazovat aplikace přímo z gitu. Projekt je součástí projektů Ship a Argo. Od verze 2.0.3 se stal kustomize součástí nástroje kubectl [32], není proto nutné instalovat žádné rozšiřující nástroje.

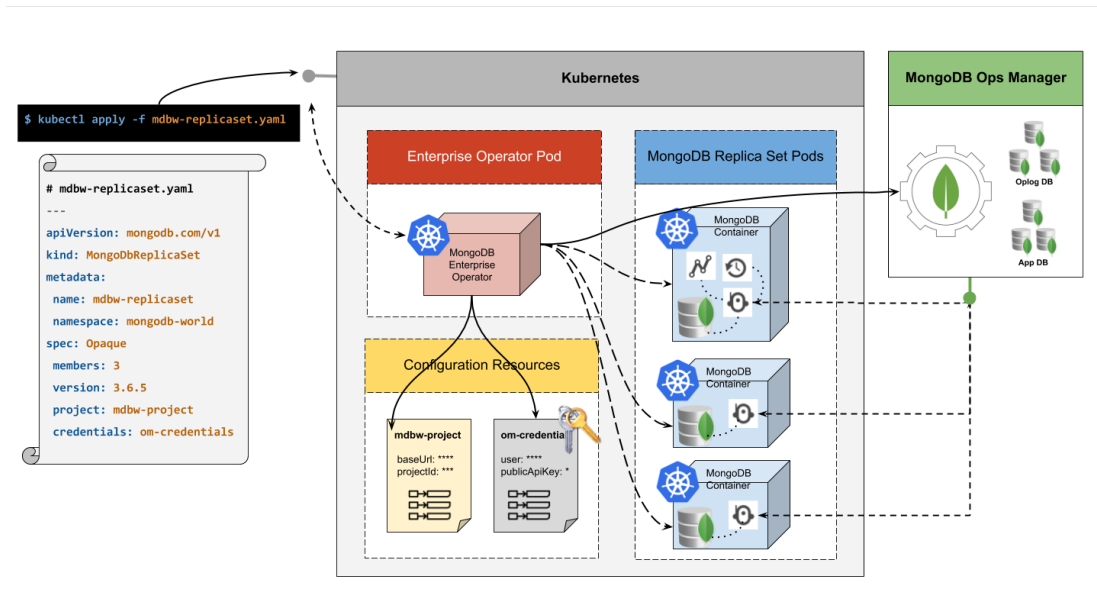
**Kód 2.6:** `kustomization.yml` soubor, zdroj: [33]

```
1 bases:
2   - wordpress
3   - mysql
4 patchesStrategicMerge:
5   - patch.yaml
```

```
6 namePrefix: demo-
7
8 vars:
9   - name: WORDPRESS_SERVICE
10     objref:
11       kind: Service
12       name: wordpress
13       apiVersion: v1
14   - name: MYSQL_SERVICE
15     objref:
16       kind: Service
17       name: mysql
18       apiVersion: v1
```

#### 2.7.4. Kubernetes operátor

Koncept operátorů je postaven na orchestraci aplikace přímo přes k8s API. V k8s totiž lze využívat objekty zvané CRD (Custom Resource Definition). Pomocí těchto CRD lze rozšířit k8s API o vlastní objekty s vlastní logikou. V praxi to vypadá tak, že v k8s clusteru běží kontejner zvaný operátor, který dokáže nové objekty s nadefinovaným CRD spouštět a vykonávat na nich požadované akce. Operátor nemá žádnou standardizovanou architekturu, jedná se o aplikaci, která komunikuje s k8s API. Aplikace může být napsaná v jakémkoliv jazyce. Pomocí operátoru lze mnohem lépe řešit životní cyklus aplikace, lze prostřednictvím něj sledovat stav jednotlivých zdrojů v clusteru a na základě změn reagovat v aplikaci. S vývojem konceptu operátor začala především firma CoreOS, která napsala základní funkcionalitu a otevřela zdrojové kódy pro nástroj zvaný operator-sdk. Tento nástroj slouží k psaní operátorů. Sdk je dostupné pouze pro programovací jazyk go. Nástroj zjednodušuje především vytvoření nových k8s API a controlleru pro nové objekty, zároveň lze pomocí něj generovat boilerplate kód. Princip operátorů se používá převážně u větších aplikací a systémů, kde je nutné kombinovat více faktorů dohromady, například před upgradem aplikace je nutné zálohování, spuštění nových služeb a podobně. Nevýhodou oproti ostatním řešením může být složitost implementace operátoru. Operátor nelze nastavovat pouze změnou yamllů, logiku celého operátoru je nutné naprogramovat.



Obrázek 2.3.: Schéma MongoDB Enterprise operátoru, zdroj:[34]

Na obrázku 2.3 je vidět schéma Kubernetes operátoru pro aplikaci MongoDB. Dále je zde patrné, jak funguje komunikace v rámci operátoru. Operátor je spuštěn jako kontejner běžící v Kubernetes clusteru, ten dokáže číst CRD. Příklad CRD je v levé části obrázku, jedná se o objekt MongoDBReplicaSet. Operátor dokáže tento k8s objekt rozpoznat a vykonat akce, které jsou v logice operátoru popsány. Samotný MongoDB operátor se v tomto případě nestará pouze o instalaci, ale o celý životní cyklus databáze, operátor totiž dokáže komunikovat i s MongoDB Ops Managerem. Tato aplikace slouží k ovládání životního cyklu aplikace, pomocí manageru lze řešit automatizaci, zálohy či monitoring databáze. Součástí podů je kromě volume a databáze i ops agent, který slouží ke komunikaci mezi jednotlivými databázemi a Ops managerem.

### 2.7.5. Srovnání

Z následujícího přehledu lze vyvodit, že neexistuje žádný nejlepší přístup k orchestraci workloadu. Žádné z řešení není univerzální, každé má své výhody a nevýhody a je určeno ke specifickému užití. Helm se nespíše díky své jednoduchosti udržel jako nejpopulárnější balíčkový manager, nástroje postavené nad jsonnetem zde budou pro pokročilejší uživatele. Kustomize působí jako vyplnění pro stávající manifestová řešení, například je součástí projektu Ship, kde pomocí kustomizem lze upravovat Helm charty. Operátory zde zůstávají jako alternativa pro komplexní aplikace, které potřebují pro svoji orchestraci komplikovanější logiku. Nutno zmínit, že toto srovnání se týká nástrojů v současné době, všechny nástroje se rychle vyvíjejí a je jen těžko odhadnutelné,

jakým směrem a způsobem se bude vývoj přístupu k orchestraci workloadu ubírat v dalších letech.

## 3. Vymezení legacy aplikace

V této kapitole bude nadefinován termín legacy aplikace. První část kapitoly bude věnována vymezení legacy aplikace a bude představena její architektura. V druhé části kapitoly jsou popsány problémy, které vznikly během potřeby udržovat tento typ aplikací v provozu.

### 3.1. Definice

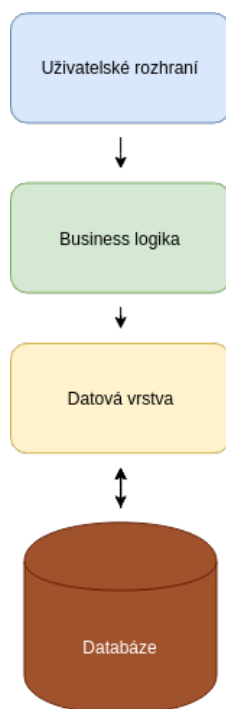
Termín legacy aplikace není jednoduché nadefinovat. Podle zdrojů [35] a [36] bývá vykládán a používán v mnoha různých kontextech. Obecně se dá říci, že se jedná o aplikaci, která je postavena na zastaralém hardwaru a provozována na neaktuálních operačních systémech. Tyto systémy jsou v současné době stále používány a udržovány v běhu na původním hardwaru. Problémem je často značná složitost aplikace a nedostatečná či nepřesná dokumentace. Některé legacy aplikace běží na mainframech. Dále se jedná o aplikace se zastaralou architekturou či napsané ve starých programovacích jazycích. Hovoříme zejména o rozlehlé monolitické aplikace, které jsou velmi těžko spravovatelné. Důvodem, proč tyto staré systémy bývají udržovány, je jejich kritická funkcionality. Například v organizaci NASA jsou stále systémy tohoto typu využívány pro vesmírný výzkum [37]. NASA ve svém výzkumu používá i některé technologie a systémy, které byly vytvořeny během 70. let minulého století.

### 3.2. Monolitická architektura

Co se týká architektury, je velmi těžké stanovit jeden typ, jelikož mnoho legacy aplikací je postaveno na rozdílných architekturách. Často bývá monolitická architektura označována jako legacy. Obecně lze říci, že monolitická aplikace je složena z jednoho velkého balíku zdrojového kódu, který obsahuje všechny komponenty. Implementace monolitické architektury bývá zprvu jednoduchá a lépe se s ní pracuje z důvodu sdíleného kódu. S přidáváním komponent začne být aplikace složitější z důvodu soudržnosti zdrojového kódu. Například refaktorování zdrojového kódu může negativně ovlivnit aplikaci. Dalším problematickým faktorem je testování aplikace. Aplikace je jeden logický celek, kterému je nutné přizpůsobit funkční testy, které bývají mnohem složi-



tější a časově náročnější než testování jednotlivých komponent aplikace [39]. Největším problémem monolitické aplikace je její nasazení a provoz. Pro nasazení nových verzí aplikace je nutné nasazovat aplikaci jako celek. Proces aktualizace často vyžaduje velké množství skriptu a manuálních procesů, které je velmi obtížné zautomatizovat. Stejný problém je se škálováním aplikace, protože není možné škálovat jednotlivé části aplikace ve více kopiích. Naopak je nutno spouštět celou repliku aplikace jako celek. Tato skutečnost vede ke plýtvání se zdroji. Při naškálování zdrojů bude server s aplikací konzumovat větší množství zdrojů, které nebudou plně využity. Jak již bylo zmíněno, tato architektura má řadu nevýhod. I přes tuto skutečnost bývá tento typ architektury používán především u menších aplikací, které nepotřebují škálovat a neobsahují složitou business logiku.



**Obrázek 3.1.:** Schéma monolitické architektury, zdroj: vlastní tvorba

Při porovnání použití architektury mikroslužeb a monolitické architektury se nedá jednoznačně stanovit, která architektura je složitější. Co se týká granularity a velikosti mikroslužeb, bývají tyto díky svému dělení mnohem rozsáhlejší než monolitická architektura. Jednotlivé služby však bývají od sebe izolovány, a to umožňuje jednotlivým komponentám nezávislost, což je výhodné především při testování a nasazování nových verzí. Tato výhoda se zpravidla projevuje u instalací, kdy není potřeba celou aplikaci znovu instalovat, jak tomu bývá u monolitické architektury. Z pohledu provozu monolitické aplikace bývají všechny funkce zabaleny do jednoho procesu, který poté

spouští celou aplikaci. Problém jednoho procesu se projeví u škálování, protože oproti mikroslužbám není možné spustit více replik části aplikace a je nutné aplikaci duplikovat celou. Porovnání architektur je v tabulce číslo 3.1.

**Tabulka 3.1.:** Porovnání monolitické architektury a mikroslužeb

	Monolitická architektura	Mikroslužby
Zdrojový kód:	Jednotný kód	Kód je rozdělen do více na sobě nezávislých celků
Programovací jazyky:	Aplikace je napsaná v jednom programovacím jazyce.	Každá služba je oddělená aplikací, která může být napsána v různých programovacích jazycích. Nutná je pouze standardizace komunikace mezi službami.
Granularita kódu:	Vysoká	Nízká
Škálování:	Aplikaci je nutno škálovat jako celek je nezbytné škálovat i služby, které nebudou plně využívány.	Lze škálovat pouze jednotlivé části aplikace nezávisle na sobě.

### 3.3. Problémy

V následující části kapitoly jsou popsány problémy legacy aplikací. U jednotlivých problémů je naznačeno, jak je lze vyřešit pomocí současných řešení a technologií.

#### 3.3.1. Metodologie vývoje

Obtížnou práci s legacy aplikacemi může způsobit kromě architektury i model vývoje softwaru, který byl použit. V těchto aplikacích probíhalo řízení projektu pomocí tzv. vodopádového vývoje (waterfall). Tento model byl nadefinován Winstonem W. Roycem, který ho zveřejnil ve svém článku „Managing the Development of Large Software System“. Již v roce 1970 byl samotným autorem tento model označen jako příklad nefungujícího modelu. Model je postaven na základním principu sedmi vrstev složených ze specifikace požadavků, návrhu, implementace, integrace, testování a ladění (validace), instalace a údržba. Mezi jednotlivými vrstvami nelze přecházet, pokud nejsou jednotlivé úkoly v dané vrstvě dokončeny. Mezi jednotlivými vrstvami se lze pohybovat

pouze vpřed. Model klade důraz především na plánování, a proto je nutné mít všechny funkční požadavky připravené ještě před samotnou tvorbou aplikace. Nevýhodou vodopádového přístupu je jeho malá flexibilita a pomalý vývoj, protože je nutné mít jednotlivé kroky předem naplánované z důvodu, že později už není prostor na změny. Aby byl model flexibilnější, začaly ho jednotlivé firmy upravovat dle svých potřeb. I přes veškeré modifikace tento model vývoje nebyl úspěšný natolik, aby mohl být v současnosti masově využíván. V dnešní době je potřeba rychlého vývoje aplikací. Nové verze nástrojů a technologií vychází každý týden, a je tedy nutné tuto skutečnost zohlednit již při vývoji aplikace samotné. I přes obrovskou jednoduchost vodopádového přístupu je model jen zřídka využíván v praxi. Slouží převážně k teoretické výuce softwarového inženýrství nebo je použit pouze ve velkých organizacích, které mají předem určený rozpis požadavků a kroků. (NASA [37], vládní projekty).

Přesným opakem vodopádového přístupu jsou agilní metodiky, které jsou založeny na iterativním přístupu k problému. Tento způsob plánování byl vytvořen především pro potřeby rychlého vývoje softwaru. V dnešní době se agilní přístup využívá také v dalších oborech a odvětvích, jako je například marketing. Stavební kámen agilní metodiky je manifest agilního programování [38], který byl navržen a vytvořen v roce 2001 na setkání odborníků softwarového plánování. Největším rozdílem mezi vodopádovým a agilním přístupem je čas reakce na změnu požadavků. Pokud se zákazník rozhodne něco změnit v průběhu implementace v agilním přístupu, je velmi jednoduché na změnu reagovat, lze ji zahrnout do nadcházející iterace. Agilní metodika v dnešní době slouží i jako odrazový můstek pro tvorbu dalších přístupů ve vývoji. Jedná se o modely jako XP, Scrum, FDD, TDD. Jedním z přístupů, který za posledních pár let vzrostl na popularitě, je Scrum. Scrum systém je založen na iteračních cyklech, které se nazývají sprint. Sprints se opakují v pevně daném časovém období, zpravidla čtrnáctidenních cyklech. Na začátku sprintu jsou vždy vymezeny problémy, které se budou ve sprintu řešit. Pro každý problém je pevně určen vlastník problému, který obvykle zastupuje v procesu roli zákazníka, ale nezasahuje do vývojového procesu. Poté jsou jednotlivé problémy rozděleny mezi vývojový tým, který je vždy zodpovědný za vyřešení daného úkolu nebo přidání nové funkcionality. Na konci sprintu by měla být nová funkcionality implementovaná ve funkční podobě. Vývojové týmy bývají složeny z menšího počtu osob, cca do deseti vývojářů. V menším počtu jsou vývojáři schopni efektivněji spolupracovat a pracovat flexibilněji. Jednotlivé vývojové týmy jsou víceúčelové a spolupracují od analýzy přes vývoj až po dokumentaci. Poslední entita, která figuruje v scrum procesu, je tzv. scrum master, který hledá překážky ve vývoji a pracuje na jejich odstranění. Zároveň kontroluje, zda se dodržuje scrum proces a zda vše probíhá tak, jak bylo naplánováno. Podobně jako u vodopádového přístupu, tak i u scrumu některé firmy, které koncept přebírají, jej upravují podle svých potřeb. Například švéd-

ská firma Spotify si vystavěla svůj Squad framework [40], který celý scrum proces mění a přidává další koncepty. Největší výhodou scrumu a agilního vývoje obecně je obrovská rychlost vývoje a schopnost doručovat zákazníkům nové vlastnosti rychleji.

### 3.3.2. Hardware

První problém, který souvisí s provozem legacy aplikací, je hardware. Zdrojový kód legacy aplikace je často přizpůsoben k provozu na konkrétní hardwarové platformě. Dříve byly tyto platformy spravovány pouze jediným vendorem, což vedlo k uzamčení se na jednoho výrobce. Pokud pochází hardware od specializovaného vendedora, je velmi nákladné tento hardware spravovat. Důvodem je především dostupnost uživatelské podpory ze strany poskytovatele hardwaru. Pokud je podpora pro hardware dostupná, je zpravidla velmi drahá.

Problémy se závislostí na vendoru lze vyřešit pomocí komoditního hardwaru, který je složen z komponent třetích stran. Tyto komponenty jsou velmi jednoduché na výměnu. Posléze pak není složité vyměnit celý server za jiný. Tento typ hardwaru je oproti specializovanému hardwaru výrazně levnější.

### 3.3.3. Udržování aplikace

V této části budou uvedeny problémy spojené s udržováním legacy aplikací. Pokud je třeba aplikaci rozšířit nebo upravit, je nutné aplikaci dodatečně doladit. Proto je u starých aplikací složených z většího počtu komponent nezbytné udržovat aktuální dokumentaci, aby se do dodatečného vývoje aplikace mohli zapojit i programátoři, kteří aplikaci nevyvíjeli. Důležitým faktorem pro udržování a rozšiřování jsou aktuální testy aplikace, ty bývají ve velké části případů vývoje zanedbávány [41]. Ať už se jedná o funkční či unit testy, je nutné mít alespoň základní sadu scénářů, se kterou lze vyvíjenou aplikaci vyzkoušet. Další komplikací, která vede k neefektivní práci vývojářů, jsou také vývojová prostředí. Velké množství komponent a tříd vede ke zpomalení vývojových nástrojů (IDE).

### 3.3.4. Aktualizace a CI/CD

Dalším problémem může být doručování aktualizací a opravných záplat do již běžící aplikace. Staré aplikace byly většinou navrhovány pro jeden způsob užití. Byly nasaženy do produkčního prostředí a, pokud byly funkční, nebylo nutno se o ně starat. Podle zdroje [42] byly některé legacy systémy v provozu i několik desítek let. I když by daná aplikace nevyžadovala žádné nové vlastnosti, je nutné zaručit alespoň bezpečnost záplat. Tyto aktualizace se netýkají pouze aplikace, ale i operačního systému, na kterém je aplikace provozována.

Problém s aktualizacemi byl vyřešen pomocí konceptu CI/CD (Continuous integration/Continuous delivery) společně s agilním vývojem aplikací. Koncept CI je založen na průběžné integraci nových změn ve zdrojovém kódu aplikace. Tyto změny jsou průběžně testovány s produkčním kódem, který je nasazen v produkčním prostředí. Součástí CI jsou procesy jako testování zdrojového kódu a sestavování aplikace. Po úspěšném dokončení těchto procesů bývá změna přidána do zdrojového kódu připraveného na nasazení do produkce. Všechny kroky v CI pipeline by měly být spouštěny automaticky za sebou bez asistence člověka. Pomocí správně zavedeného CI lze velmi jednoduše odhalit chyby ve zdrojovém kódu nebo konflikty s již existujícím kódem. Celý CI proces se poté odehrává okolo repozitáře se zdrojovými kódy, většinou se jedná o gitový repozitář. Na něj je napojen CI nástroj, který sleduje změny v repozitáři a na základě těchto změn spouští jednotlivé CI procesy. I když tento proces dokáže fungovat automaticky, přichází s mnohem náročnějšími podmínkami na hardwarové zdroje, na kterých jsou jednotlivé akce spouštěny. Princip Continuous Delivery je postaven na průběžném dodávání nových vlastností a opravných záplat do již existujících prostředí. Pokud má být CD dostatečně efektivní a rychlé, je nutné mít celý proces automatizovaný [43]. U velkých firem bývají aplikace vyvíjeny agilními přístupy, avšak vydávání nových verzí vyžaduje spoustu manuálních kroků: například různé typy testování. Při správně naimplementovaném CI/CD by měly být změny nasazovány přímo na testovací prostředí (často označováno jako stage). Toto prostředí by mělo být téměř identické jako produkční. Pokud změna na stage prostředí funguje správně, je možno přejít k nasazování do produkčního prostředí. Je-li CI/CD proces dobře nastavený, není potřeba žádný další tým k nasazování nových verzí aplikace. Důležité je tento proces uchovat jednoduchý, aby ho bylo možné flexibilně měnit a škálovat. Staré CI systémy jsou často složeny ze skriptů, které lze velmi složitě měnit, protože mají pevně dané hodnoty pouze pro jeden způsob užití.

## 4. Analýza migrované aplikace

Jako tradiční aplikace pro migraci do kontejnerové virtualizace byl zvolen OpenStack, konkrétně distribuce od firmy Mirantis, která je dodávána jako součást produktu Mirantis Cloud Platform (MCP). Tato Aplikace by se dala označit jako tradiční aplikace, nejedná se přímo o monolitickou aplikaci, ale některými problémy je velmi tradičním aplikacím blízká. MCP distribuce byla vybrána z důvodu spolupráce s firmou Mirantis. Ta se zabývá nasazováním cloudových řešení pro enterprise zákazníky. Hlavním cílem a požadavkem ze strany Mirantisu bylo prozkoumat možnosti řešení problémů, kterými současný produkt trpí. Především otestovat principy kontejnerizace OpenStacku. V počátku této kapitoly je stručně popsán produkt MCP, poté je představena současná architektura. V závěrečné části kapitoly jsou zmíněny problémy, se kterými se platforma potýká. Součástí samotného MCP je i několik dalších podpůrných komponent Stacklight (systém pro monitoring), DriveTrain (CI/CD systém) a Ceph (storage systém). Kromě MCP existuje na trhu i řada dalších projektů a komerčních produktů postavených okolo OpenStacku, těmi nejznámějšími produkty jsou Red Hat OpenStack, SUSE OpenStack, z projektů se jedná například o TripleO, DevStack či OpenStack-Ansible. Jednotlivé produkty se pak liší v architektuře a způsobu ovládání.

### 4.1. Mirantis Cloud Platform

MCP je software pro tvorbu cloudových řešení. Používá se především pro privátní cloudová řešení. Celé řešení je open source, dá se tedy nainstalovat a provozovat zdarma. Hlavní byznys model je založen na podpoře a integraci platformy s okolními službami. MCP je složeno z velkého množství komponent a nabízí řešení nejen pro klasické virtuální stroje (OpenStack), ale i pro kontejnery (k8s a CaaS). V rozsahu praktické práce bude pod pojmem MCP označováno pouze OpenStack řešení bez jeho podpůrných komponent. Zbylé komponenty nejsou pro význam této diplomové práce důležité. Ovšem pro zařazení do kontextu je nutné alespoň stručně představit jednotlivé části platformy a jejich funkce.

### 4.1.1. OpenStack

OpenStack je software na tvorbu privátních cloudů a public cloudů, je složen z několika open source projektů, které byly postupně převedeny pod OpenStack. Projekt vznikl v roce 2010, kdy se spojila firma Rackspace Hosting s NASA a otevřely zdrojové kódy [44] ke svým projektům, ze kterých se stala nova (compute engine) a swift (object storage), a tím začal vznikat projekt OpenStack. Později začalo do projektu přispívat více kontributorů ze společností Intel, Ubuntu, Cern, IBM atd. Z OpenStacku se stal standard pro privátní cloud a předběhl i konkurenční řešení Apache Cloudstack, který je rovněž open source.

OpenStack nabízí širokou škálu projektů, které řeší mnoho problémů. Jeden z projektů, který je zaměřen na jednu speciální oblast, je projekt zvaný Sahara. Projekt řeší oblast Big Dat a lze pomocí něho instalovat a spravovat nástroje pro ně určené: Hadoop, Spark atd. Projekty typu Sahara bývají používány jen zřídka v poměru k počtu instalací do datacenter. Nejpoužívanější způsob užití pro OpenStack je virtuální infrastruktura. Pod slovy virtuální infrastruktura si lze představit několik serverů (nejčastěji Linuxových), které jsou propojeny pomocí virtuální sítě. Tato síť pak může být routovaná dál po datacentru nebo může směřovat do internetu, záleží na způsobu využití cloudu. OpenStack bývá často využíván jako náhrada za komerční cloudová řešení typu VMware. K tomuto účelu se většinou používá spojení základních sedmi komponent Nova, Keystone, Neutron, Glance, Cinder, Heat, Horizon. Nova slouží ke spuštění virtuálního workladu. Keystone je služba poskytující API pro autentifikaci, autorizaci a service discovery. Glance je služba sloužící pro ukládání image pro virtuální instance. Neutron slouží k síťování a propojení jednotlivých zdrojů. Cinder je služba, pomocí které je možno připojit do OpenStacku blokovou storage. Heat je orchestrátor, který slouží k ovládání OpenStack workloadu. Základem Heatu jsou šablony, pomocí kterých lze přistupovat k workloadu IaaS přístupem. Poslední službou je Horizon, což je webové rozhraní, pomocí kterého lze ovládat všechny výše popsané komponenty.

### 4.1.2. DriveTrain – CI/CD

Jak již bylo zmíněno na začátku, DriveTrain je komponenta, která se stará o instalaci MCP a následný lifecycle management. Celé řešení je spuštěno ve virtuálních strojích. Nad těmito virtuálními stroji je celý CI/CD stack spuštěn v kontejnerové virtualizaci, konkrétně v Dockeru. Pro orchestraci je využit Docker Swarm, pro ukládání perzistentních dat je pak mezi CI/CD servery nainstalován souborový systém GlusterFS. DriveTrain komponenta je složena ze čtyř hlavních komponent Jenkins, Gerrit, OpenLdap a Aptly. Jenkins je CI nástroj, který původně sloužil k sestavování softwarů a spouštění jednotlivých testů, ale díky velkému množství rozšíření je možno Jenkins využít i k dal-

ším činnostem. Dalo by se říci, že Jenkins je CI framework, který lze pomocí rozšíření přizpůsobit pro různé úkoly. V DriveTrinu se používá na instalaci a lifecycle management cloudu. Stavebním pluginem je pipeline plugin. Pipeline si lze představit jako strom propojených úloh, které jsou spuštěny v předem daném pořadí. Pipeline jsou psané v programovacím jazyce Groovy, který je velmi podobný jazyku Java, ale díky jeho syntaxi je mnohem vhodnější na skriptování. Jednotlivé pipeline jsou složeny ze dvou základních komponent: pipeline-library [45] a mk-pipelines [46]. Pipeline-library je repozitář obsahující funkce, ze kterých jsou následně složeny pipeline umístěné v druhém repozitáři mk-pipelines. Tyto repozitáře jsou umístěny v Gerritu spolu s meta-dovým modelem. Gerrit je webová aplikace sloužící jako gitový repozitář pro ukládání zdrojových kódů. Pro regulaci přístupu do Gerritu a Jenkinse pak slouží OpenLdap. OpenLdap je implementace protokolu LDAP. Aplty slouží jako lokální obraz pro debian balíčků pro případ, že by nebyly dostupné upstream repozitáře s balíčky. Je velmi časté, že zákazníci mají cloudové prostředí bez přístupu k internetu.

Správa cloudu v praxi funguje tak, že pro každou akci prováděnou na control plane cloudu, jako jsou třeba restart služby, aktualizace balíčku či obnovení záloh databáze, je napsaná pipeline. Pipeline zapouzdřuje vždy logiku předdefinované akce. Například při obnovení databáze je třeba udělat několik kroků. Je nutno ověřit, zda existují zálohy z vybraného dne, zda není aktivní MySQL služba, poté je potřeba zálohy přesunout ze zálohovacího serveru na server s databází a až následně provést obnovení databáze. V závěrečném kroku je nutno zkontrolovat funkčnost databáze, dále zkontrolovat spojení s clusterem a dostupnost databáze pro ostatní služby. Tyto všechny akce lze spustit pomocí příkazu přes shell, pipeline tyto operace zlehčuje a šetří čas, jelikož operátor nemusí být na danou akci doménový expert. Tento koncept pipeline výrazně ulehčuje práci operátorům, kteří nemusí jednotlivé akce provádět manuálně, či si psát pro každou akci shellový skript, který může potenciálně cloud poškodit.

#### 4.1.3. Stacklight – Logování a monitoring

Pro správný chod všech serverů a služeb je nutné celou infrastrukturu monitorovat. Za tímto účelem je v MCP naimplementován monitorovací systém. Ten stejně jako CI/CD běží v Docker Swarmu. Monitoring se dá rozdělit na dvě základní části: logování a monitoring. Logování je postavené na technologiích Fluentd a Fluent bit, které běží jako daemon na všech serverech, ze kterých sbírají logy a posílají je do časové databáze Elasticsearch, kde jsou logy po dobu 30 dnů uloženy. Pro zobrazení a lepší práci s logy je nad Elasticsearchem spuštěn nástroj Kibana. Kibana je program, který slouží k vizualizaci dat z Elasticsearch. Pomocí Kibany lze pohodlně sledovat a filtrovat pomocí různých dotazů veškeré logy v reálném čase. Druhá část monitoringu je postavená na technologii Prometheus, která sbírá jednotlivé metriky ze serverů. Tyto metriky jsou



do Prometheusa posílány prostřednictvím programu Telegraf, který je vyčítá z jednotlivých serverů a služeb. Telegraf dokáže vyčítat velkou část metrik od vytížení CPU přes procesy až po velikost volného místa na disku. Hlavním rozdílem Prometheusa oproti ostatním monitorovacím nástrojům je jeho architektura, je totiž založen na principu sbírání a ukládání metrik, na rozdíl třeba od populárního Nagiosu. Díky uloženým datům v databázi lze pomocí Alertmanageru, což je část Prometheusa, vytvářet jednotlivá upozornění. Prometheus dále obsahuje několik rozšiřujících modulů. Například integraci s různými notifikačními systémy, což znamená, že přestane-li nějaká služba pracovat, Prometheus to pomocí metriky zjistí a zašle upozornění například do Pagerduty, Slacku či na email. Pro lepší vizualizaci si lze metriky zobrazit nejen v grafickém rozhraní Prometheusa, ale také v Grafaně. Grafana je nástroj, pomocí kterého, jak už název napovídá, lze jednoduše s využitím JSON jazyka vytvářet grafy.

## 4.2. Architektura

Hardwarovou infrastrukturu pro privátní cloud lze rozdělit na dva základní typy fyzických serverů: na KVM servery a compute. Servery, na kterých jsou spuštěny služby starající se o ovládání cloudů, se nazývají control plane. A protože v MCP jsou všechny tyto služby spuštěny na virtuálních strojích pomocí KVM virtualizace, jedná se tedy o virtuální control plane. Servery, které slouží k provozu VM vytvořených OpenStackem, se označují jako compute. Pro oba typy těchto serverů (fyzické i virtuální) je používán operační systém Linux, konkrétně distribuce Ubuntu v LTS verzi. V současné době se jedná o verzi 16.04 xenial. Celé MCP řešení je postaveno na otevřeném softwaru, většinou jsou používány upstream balíčky distribuce Ubuntu, pro služby jako OpenStack jsou použity downstream balíčky, které obsahují opravy/úpravy kódu pro jednotlivé zákazníky. Tyto změny jsou pak dostupné na Mirantis GitHubu a Gerritu.

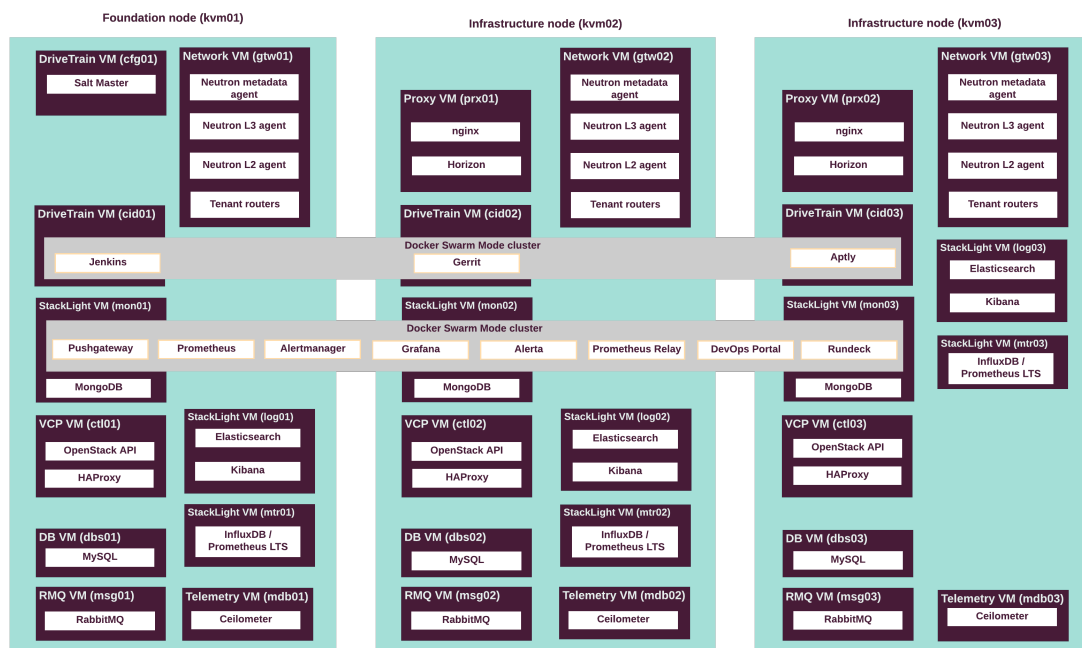
Jak již bylo zmíněno, celý control plane OpenStacku je spuštěn ve virtuálních strojích. Pro virtuální stroje bývá jako hypervisor používán KVM. Jednotlivé virtuální stroje jsou logicky rozčleněny dle svých základních rolí. K fyzickým KVM serverům bývá připojena storage, která slouží primárně pro disky virtuálních strojů. Každá role zahrnuje logickou část OpenStacku. Schéma virtuálního control plane je na obrázku 4.1. K ovládání Linuxových serverů, na kterých jsou spuštěny OpenStack služby, slouží VM označené jako (cfg01), které také bývá často označováno jako salt-master. Jedná se o konfigurační server, který slouží jako hlavní ovládací konzole pro konfigurační management SaltStack, kterým je ovládáno celé řešení. Je na něm uložen metadatový model, který popisuje celou infrastrukturu. Součástí cfg nody je také aplikace MaaS (Metal as a Service), která slouží především v prvotní fázi pro instalaci fyzických serverů. MaaS dokáže na jednotlivé servery nainstalovat operační systém pomocí PXE bootu. To zna-

mená, že pro instalaci stačí ručně nainstalovat pouze jeden operační systém, na kterém je spuštěn virtuální stroj s MaaS a salt-masterem.

První typem nodů, které jsou spouštěny, jsou control node (ctl), které slouží pro správu a komunikaci mezi jednotlivými OpenStack službami. Na nich je nainstalovaný Keystone a serverové role některých OpenStack projektů jako například Heat, Cinder či Glance. Jednotlivé služby, jakými jsou Glance či Cinder, dokáží pracovat s různými typy backendů. V MCP se pro ukládání Glance image používají dva standardní backendy, Glusterfs nebo Ceph. Pro Cinder lze pak použít opět softwarovou storage Ceph, Lvm nebo i hardwarové řešení, pokud jimi zákazník disponuje. Další typ nodů jsou database nodes (dbs). Na těchto serverech je spuštěna databáze, která uchovává všechna data o uživateli OpenStacku a workloadu, který je v OpenStacku spuštěn. Pro tento účel se používá MySQL databáze, v MCP je použito řešení Galera Cluster od firmy Percona. Galera oproti klasickému MySQL clusteru má lepší vlastnosti (v oblastech replikace a škálování). Dalšími typy nodů, které jsou v řešení použity, jsou message nodes (msg). Ty slouží pro software RabbitMQ, který plní funkci message brookera (fronty), a veškerá komunikace mezi OpenStack službami prochází přes tuto frontu. Pro síťování jsou použity speciální dedikované nody, ty mohou mít dva odlišné názvy (ntw nebo gtw). Záleží na tom, jaké síťovací řešení je použito. Pro OpenvSwitch, dále jako OVS, jsou použity gateway nody. Samotné OVS může být nakonfigurováno v několika různých režimech. Zde záleží na modelu routování komunikace mezi jednotlivými compute. Pro správu síťování v MCP lze také použít další SDN (Software Defined Networking) řešení implementované pomocí neutron pluginů. V současnosti hlavním SDN, které je v MCP oficiálně podporováno, je řešení původem od Juniper Network, Tungsten Fabric (dříve OpenContrail). Toto řešení implementuje dva základní typy nodů (ntw), které slouží jako controllery pro SDN. Controllery slouží ke správě virtuálních sítí. Pro ukládání informací o činnostech ve virtuálních sítích slouží analytické nody (nal). Obrovskou výhodou Tungsten Fabricu v porovnání se základním neutronem jsou vlastnosti, které Tungsten Fabric nabízí. Neutron v OVS režimu tyto vlastnosti potřebuje doinstalovat, případně dokonfigurovat prostřednictvím dalších OpenStack projektů. Jedná se o služby jako FaaS (Firewall as a Service) či LBaaS (Load balancer as a Service). Posledním typem virtuálních serverů z control planu jsou proxy nody (prx). Tyto nody slouží pro externí přístup do OpenStacku. Je na nich nainstalovaný Horizon a je zde především Nginx, který slouží jako reverse proxy pro endpointy jednotlivých OpenStack služeb a zároveň je zabezpečuje pomocí certifikátů.

Speciálním typem serveru v MCP jsou OpenStack compute (cmp). Jedná se o fyzické servery, které jsou výkonnější než KVM a je na nich spuštěn OpenStack workload – VM spravované OpenStackem. OpenStack podporuje pro compute mnoho typů hypervisorů a různé typy virtualizace, například Hyper-V, VMware vSphere či LXC. V

praxi pak bývá pro virtualizaci používáno klasické QEMU/KVM. Jedná se o nejrozšířenější typ linuxové virtualizace, pro jeho používání není nutná licence pro hypervisoru. Ke compute bývá často připojovaná i bloková storage, která slouží k uchování perzistentních volume pro disky instancí. Nejčastěji je pro tento typ storage v MCP využit Ceph, který dokáže poskytnout jak blokovou storage, tak i ephemeral disky. Tento typ disku je možno použít jako boot disk instance vytvořené OpenStackem.



Obrázek 4.1.: MCP architektura s OVS networkingem, zdroj: [47]

Pro zajištění vysoké dostupnosti jsou všechny control plane servery spuštěny ve více replikách pro případ, že by se některá služba stala nedostupnou. Počet replik se odvíjí od počtu compute nodů v OpenStacku. Vysoká dostupnost je řešena pomocí dvou služeb, Keepalived a Haproxy. Služby jsou nakonfigurované tak, že Keepalived vytvoří virtuální ip adresu (VIP) vždy pro jeden set nodů, například control. Jednotlivé služby vždy komunikují přes VIP a daný port. Jednotlivé requesty jsou rozdělovány pomocí Haproxy mezi jednotlivé servery. Pokud jedna služba na serveru přestane fungovat, chod nebude ovlivněn, protože Haproxy bude stále rozdělovat požadavky mezi zbývajících dva. Pokud je ztracen nod, který má aktivní VIP, Keepalived zjistí pomocí broadcastu, že nod nereaguje a adresa není dostupná. VIP se poté přepne na další dostupný server, záleží na prioritě, kterou má nakonfigurovanou v konfiguračním souboru.

### 4.2.1. Orchestrace (SaltStack)

SaltStack je software pro konfigurační management, který používá deklarativní přístup, což znamená, že vždy popisuje stav, v jakém má být prostředí či software nakonfigurován. Jednotlivé akce, které má SaltStack vykonat, jsou popsány pomocí stavů. Jednotlivé stavy jsou složeny z modulů. Tyto moduly jsou součástí SaltStacku a napsány v programovacím jazyce Python, díky tomu lze jednotlivé formule jednoduše rozšířit o další externí moduly. Veškerá instalace a konfigurace je v MCP prováděna pomocí tzv. formulí, což je sada stavů zapouzdřená do jedné logické složky. Formule pro tyto stavy může využívat různé externí moduly. Řada těchto modulů je předdefinována přímo v SaltStacku. Jedná se především o moduly pro ovládání základních služeb či moduly pro běžné akce na souborovém systému. Pro pokročilejší akce je nutné vytvářet vlastní moduly. V MCP jsou jednotlivá metadata a formule logicky rozděleny po službách či systémech. Například pro základní konfiguraci linuxových serverů lze použít `salt-formula-linux`, která umí nainstalovat a nakonfigurovat jednotlivé části systému, jako jsou síť nebo souborový systém. Služby typu `ssh`, `ntp` či `nfs` mají opět svou vlastní formuli. Toto rozdělení do logických částí poskytuje mnoho výhod. První z nich je rozdělení complexity mezi jednotlivé formule. Každá se poté specializuje pouze na danou oblast a služba je mnohem lépe udržitelná. Celkové řešení používá dohromady přes 100 různých formulí, které by se daly rozdělit do dalších několika kategorií.

- Infrastrukturní (Linux, OpenSSH, Salt)
- Deployment služby (MaaS, Libvirt)
- Podpůrné služby (MySQL, Nginx, Apache)
- OpenStack služby (Keystone, Nova, Cinder)
- Kontejnerové služby (Docker, Kubernetes, Helm)
- mMonitoring (Elasticsearch, Kibana, Prometheus, Sensu)
- Integration (Jenkins, Aptly, Packer, Gitlab)
- Programovací jazyky (Java, Node, Python, Ruby)

V praxi je nutno na jednotlivých serverech, ať už virtuálních nebo fyzických, provést konfiguraci několika typů těchto služeb. Pro definici, jakou roli má mít každý server, je použit projekt Reclass (recursive external node classification), který rekurzivně spojí všechny yml soubory nainportované na jednotlivé servery v modelu. Díky tomu

může mít nadefinovaný nod víc typů různých metadat. Díky Reclassu lze též přetěžovat a přepisovat jednotlivé části modelu, což zaručuje velmi vysokou modelovací flexibilitu. S flexibilitou souvisí komplexnost a složitost modelů. Často se stane, že je daný Reclass parametr potřeba změnit. Parametry bývají častokrát v modelu přetěžovány, proto je velmi náročné najít místo, kde se nachází daný parametr, který je nutné přetížit. Parametry lze přebírat a přetěžovat napříč importy tříd modelu. Projekt Reclass je dostupný pro řadu dalších konfiguračních nástrojů, jako jsou Ansible či Puppet.

Jednotlivé salt-formule jsou vyvíjeny jako open source. Důvodem, proč byly vytvořeny nové formule a nebyly použity komunitní spravované samotným SaltStackem, jsou jednotná metadata. Formule pod hlavičkou SaltStacku (na GitHubu jako SaltStack-formulas) jsou tvořeny komunitou okolo technologie a nemají žádnou pevnou strukturu. Při nasazení většího množství formulí dohromady je nutné mít pro větší přehlednost jednotný styl metadat. V MCP je tedy využíván projekt salt-formula, který je řízen převážně Mirantisem a komunitou používající MCP. Z počátku byl projekt vyvíjen primárně na GitHubu, po čase však bylo rozhodnuto, že formule budou přesunuty a spravovány nezávisle na GitHub komunitě. Formule jsou nyní dostupné přes systém gerrit [48], který poskytuje lepší možnosti spolupráce a code review než ostatní nástroje.

#### 4.2.2. Metadata model

Současné MCP bylo od začátku navrhováno v konceptu IaC, infrastructure-as-a-code. Znamená to, že celé cloudové řešení je postaveno nad metadatovým modelem. Model lze lehce upravovat a pomocí něho měnit prostředí. Model je složen ze tří základních typů metadat: servisních, systémových a produktových (clustrových). Tyto tři skupiny jsou od sebe izolované, a díky tomu lze jednoduše jednotlivé typy metadat kombinovat. Model je tvořen pomocí SaltStack pillaru. Těmi je popsáno, jak mají jednotlivé služby být nainstalované a nakonfigurované. Systémová a servisní část modelu je univerzální a jsou v ní předdefinovány univerzální konfigurace a parametry, tato část modelu je pro všechny instalace postavené na MCP stejná. Cluster model je hlavní část, ve které jsou nadefinovány konkrétní parametry, které jsou pro každého zákazníka či cloud unikátní. Jedná se o parametry, jako jsou hesla, jména serverů, ip adresy či dodatečná konfigurace spojená s integracemi na další systémy. Na této úrovni metadat jsou zpravidla přetěžovány parametry, které jsou definované v systémové části modelu. Všechny soubory v modelu jsou yaml soubory. Tyto soubory jsou většinou mezi sebou provázány pomocí jednotlivých importů.

**Kód 4.1:** Metadata model pro Galera databázi, zdroj: vlastní tvorba

```
1 classes:
2 - system.linux.system.repo.mos9
```

```

3 - system.linux.system.repo.saltstack
4 - system.galera.server.cluster
5 - system.galera.server.database.aodh
6 - system.galera.server.database.ceilometer
7 - system.galera.server.database.cinder
8 - system.galera.server.database.glance
9 - system.galera.server.database.grafana
10 - system.galera.server.database.heat
11 - system.galera.server.database.keystone
12 - system.galera.server.database.nova
13 - system.galera.server.database.neutron
14 - cluster.cloudlab_cluster
15 parameters:
16   _param:
17     keepalived_vip_interface: eth1
18     keepalived_vip_virtual_router_id: 80
19     galera_server_cluster_name: cloudlab_cluster
20     cluster_vip_address: ${_param:openstack_database_address}
21     cluster_local_address: ${_param:single_address}
22     cluster_node01_hostname: dbs01
23     cluster_node01_address:
24       ${_param:openstack_database_node01_address}
25     cluster_node02_hostname: dbs02
26     cluster_node02_address:
27       ${_param:openstack_database_node02_address}
28     cluster_node03_hostname: dbs03
29     cluster_node03_address:
30       ${_param:openstack_database_node03_address}
31 linux:
32   network:
33     interface:
34       eth2:
35         enabled: true
36         type: eth
37         proto: static
38         address: ${_param:single_address}
39         netmask: ${_param:control_network_netmask}

```

Na ukázce kódu číslo 4.1 je znázorněn metadatový model pro Galera databázi. Tento soubor ve formátu yaml je složen ze tří základních stavebních bloků. Prvním je blok `classes`, který určuje yaml soubory s třídami, které budou na nodu aplikovány. Třídy by se daly přirovnat k importům klasických tříd z objektově orientovaných jazyků. Tyto třídy udávají cestu na další yaml soubory, které jsou pomocí `Reclassu` vyrendrovány

a spojeny v jeden velký saltstack pillar. Všechny yaml soubory začínající slovem systém pak odkazují na systémový model [50], na část, která je stejná pro všechny cloudy. Pillary je možné následně jednotlivými formulami aplikovat. Jednotlivé soubory tříd se mohou navzájem přetěžovat, přetěžování funguje směrem od shora dolů. Druhou složkou jsou parametry, jejichž definice začíná na řádku 16. Tyto parametry slouží převážně k přepisování parametrů z nadefinovaných tříd, v tomto případě se jedná například o konfiguraci služby Keepalived, která slouží k zajištění vysoké dostupnosti. Poslední částí je pillar, který je unikátní pro danou službu. V ukázce se jedná o konfiguraci pro linuxový interface, řádek 28. Zde je definice pro interface eth2. Tato definice pillaru pak přetěžuje, veškeré definice přicházející ze tříd uvedených v bloku classes. Na cluster modelu se takto specifikují pouze pillary, které jsou unikátní pro jednotlivá prostředí.

### 4.2.3. Instalace

V předchozích částech kapitoly byl představen produkt MCP a jeho komponenty. Nyní bude popsáno, jak se tento produkt nasazuje k zákazníkům. Pro instalaci projektu je nutné mít připraveny fyzické servery, které jsou navzájem propojeny třemi odlišnými sítěmi. Jednotlivé sítě mají vždy svou úlohu. První je management, který slouží pro řízení serverů, druhý je control plane, který slouží pro komunikaci OpenStack služeb, a poslední je dataplane, což je síť, po které komunikují instance vytvořené OpenStackem. Základem každé instalace je metadatový model, který je popsán v předchozí kapitole. Ten je vygenerován interním nástrojem postaveným nad projektem Cookiecutter [51]. Cookiecutter je šablonovací systém, který umožňuje v yaml souborech používat logiku jazyku Jinja2. To umožňuje vytvářet podmínky, proměnné, parametry či cykly. Tyto parametry jsou plněny ze souboru, který je nazýván context, jenž je ve formátu JSON. Pomocí Cookiecutteru contextu je určeno, jaké budou parametry daného cloudu a jak bude cloud nakonfigurován. U parametrů jde především o IP adresy, hesla, hostname či základní konfigurace služeb. Vygenerovaný model je nutné ještě ručně upravit pro každé prostředí, protože ne všichni zákazníci mají stejnou síťovou konfiguraci serverů nebo používají jiné storage systémy pro KVM nody. Po vygenerování modelu lze přistoupit k samotné instalaci cloudu. Instalaci operačního systému Ubuntu je nutno provést manuálně, protože operační systém pro první KVM server nelze automatizovat. Manuální část se týká také síťové konfigurace, je nutné nakonfigurovat linuxové bridge, pomocí kterých je připojena síť do konfiguračního serveru. Jakmile jsou linuxové bridge připravené, lze spustit cfg server. Disk tohoto serveru je totiž vždy stejný, a proto je pro cfg01 předpřipravená image [49]. Této image je při spuštění předána též config.iso, která obsahuje vygenerovaný model. Libvirt si při spuštění image přečte data ze souboru config.iso a pomocí cloud-initu spustí a nakonfiguruje cfg server již

podle modelu. Po spuštění cfg serveru již lze veškerou konfiguraci provádět pomocí SaltStacku. Po spuštění konfiguračního serveru je nutné nainstalovat operační systém na zbylé servery pomocí MaaSu. Ten přes síť, která je určena pro management, začne poskytovat DHCP z předem nadefinovaného rozsahu. Hardwarové servery jsou nakonfigurovány tak, že se v nekonečné smyčce pokouší o PXE boot. Jakmile obdrží IP adresu, tak se zaregistrují do MaaSu. Ten dokáže pomocí PXE na operačním systému nainstalovat operační systém Ubuntu a spustit na něm cloud-init. V rámci cloud-initu se dají provádět další poinstallační úlohy. V rámci MCP to je instalace a konfigurace salt-minionu, ten spouští stavy z salt-mastera. Jakmile jsou servery zaregistrovány pod salt-masterem, může začít instalační proces, který probíhá pomocí Jenkins pipeline. Tato pipeline je spouštěna z Jenkinsu, který je předinstalován na konfiguračním serveru. Na rozdíl od DriveTrain Jenkinse tento Jenkins obsahuje pouze jednu pipeline a ta slouží pouze k instalaci cloudového prostředí. Jenkins je po úspěšném dokončení pipeline zastaven a smazán. Pro kontrolu, zda je prostředí správně nainstalováno a zda jsou všechny služby dostupné, slouží projekt Rally a Tempest. Tyto projekty však bývají spouštěny prostřednictvím CVP pipeline, která se nachází v již zmiňovaném DriveTrain Jenkinsovi. Tato pipeline dokáže vygenerovat HTML reporty o tom, jak jsou služby dostupné a jakou mají odezvu.

#### 4.2.4. Správa

Udržování cloudu a lifecycle management okolo něj jsou velmi důležité. Po instalaci, pokud nejsou vyžadovány nějaké změny, je cloud pozorován skrze monitorovací systém Stacklight, který je sám schopen na základě svých upozornění ukazovat aktuální stav. Tato upozornění bývají přeposílána do zvoleného CRM systému nebo do e-mailu. Stacklight nemonitoruje pouze stavy jednotlivých služeb, ale také metriky a logy, které jsou produkovány ostatními službami. Pokud je vyžadována nějaká změna v konfiguraci cloudu, je nutné použít následující postup. Nejprve je třeba provést změnu v meta-datovém modelu a vytvořit commit se změnou a nahrát změnu do gitového repozitáře s modelem. Změnu je poté nutné na straně cloudu stáhnout a aplikovat. Lze ji aplikovat pomocí Jenkins pipeline nebo manuálně ručním voláním SaltStacku. Tento postup je doporučený především kvůli přehledu, ze kterého je patrné, co se v infrastruktuře stalo a kdo změnu provedl. Z důvodu úplnosti a správnosti změn není vhodné změny dělat na konfiguračním serveru. Zejména u větších operations týmů není přehled, kdo změny vykonal a jaké změny byly na serverech aplikovány.



## 4.3. Problémy

I když MCP je oproti jiným cloudovým řešením od ostatních vendorů velmi flexibilní a nabízí řadu možností, které se týkají integrace a jednotlivých konfigurací, přesto se jak pracovníci Mirantisu, tak jejich zákazníci setkávají s problémy, které je nutné každodenně řešit. V počátcích OpenStacku neumělo mnoho firem OpenStack správně nainstalovat a nakonfigurovat. V současnosti se řeší spíše problémy spojené s úlohami, které se objeví až po instalaci cloudů, jako jsou bezpečnosti aktualizace a upgrady. Následující část bude věnována nejproblematictější částí MCP z pohledu vývojáře a správce.

### 4.3.1. Složitost

Prvním z problémů je správa cloudového řešení. Nejedná se jenom o správu workloadu, který je spravován OpenStackem, ale o samotné cloudové řešení. Problémem je jeho složitost a množství komponent, které je potřeba znát a ovládat. Open source cloudové řešení pro svou správu vyžaduje širokou znalost technologií, od databází přes operační systém Linux až po síťování. Pro takovýto technologický stack je velmi obtížné zajistit zaměstnance nebo partnery, kteří by chtěli do MCP integrovat svůj produkt. Díky vysoké učební křivce může seznámení s řešením trvat i v řádu měsíců. Každý OpenStack vendor má své řešení postaveno trochu jinak a používá k jeho orchestrování odlišné nástroje. Pro koncového zákazníka je téměř nemožné tak složité prostředí udržovat a spravovat pouze vlastními pracovníky. Většina vendorů nabízí možnost placené podpory, která pomůže zákazníkům operovat služby, nad kterými je OpenStack spuštěn. Zákazník se poté stará pouze o workload a konfigurace OpenStacku samotného.

Složitost modelu se odvíjí dle požadavků jednotlivých zákazníků. Obecně platí pravidlo: čím více komponent a integrací, tím komplikovanější model. Protože cloudy jsou živé prostředí, mohou být jednotlivé modely upravovány v čase. Pro aktualizaci cloudu je nutné projít celou metadatovou strukturu. Jelikož jsou metadata uložena na souborovém systému v jednotlivých yaml souborech, je velmi těžké zajistit jejich standardizovanou strukturu. Během aktualizací se běžně stávalo, že některé komponenty nefungovaly správně, protože inženýr, který aktualizaci prováděl, přehlédl chybu v metadatovém modelu. Dalším problémem, se kterým se inženýři u zákazníků setkali, byly ručně provedené změny v konfiguračních souborech. Celé MCP řešení je totiž navrhováno tak, aby jednotlivé služby byly konfigurovatelné metadatovým modelem z jednoho místa. Pokud je v modelu odlišná konfigurace, než je na cílovém serveru, může být tato chybná změna nechtěně aplikovaná bez jakéhokoliv vědomí.

### 4.3.2. Life cycle management a aktualizace

Vzhledem k velkému množství rutin a repetitivních záležitostí, které jsou na MCP cloudech, se dospělo k závěru, že je nutné tento typ akce automatizovat. Aby nebylo nutné všechny příkazy volat ručně, jsou tyto akce volány skrze Jenkins pipeline. Jedná se o úlohy, jako jsou obnovení či záloha Galera databáze, výměny Ceph osd disku či aktualizace na novější MCP verzi. Lifecycle probíhal způsobem, kdy byla provedena změna v modelu a nová verze modelu byla stažena na konfigurační server. A poté byla spuštěna pipeline z DriveTrine Jenkinse. Jednotlivé pipeline volaly jednotlivé stavy z saltformuli nebo shell příkazy, například ověření výstupu. Bohužel ne všechny úlohy jsou takto automatizované, například různé konfigurace služeb jako IDM nikdy nebyly zařazeny do pipeline a je stále potřeba provádět je přes SaltStack manuálně. Neautomatizované služby jsou především služby, které nebyly zákazníky často vyžadovány. Používáním se osvědčilo, že tato abstrakce logiky do Jenkins pipeline je vhodná pouze na jednoduché repetitivní úlohy (výše zmiňovaná výměna osd disku pro Ceph). Pro pokročilejší akce, jako jsou především aktualizace na nové verze, slouží například komponent OpenStack, Ceph či OpenContrail.

Pro tyto komponenty byla napsána univerzální pipeline. Jedním z problémů univerzální pipeline bylo přidávání možnosti aktualizovat na novější verzi. Nové verze OpenStacku přicházely se změnami, které byly těžko automaticky implementovatelné, jednalo se například o změny databázových schémat pro nové služby. Dalším problémem těchto univerzálních postupů je fakt, že zpočátku nepočítaly se zákaznickými integracemi. Ve většině případů pipeline na aktualizaci cloudu skončily neúspěšně, protože pipeline narazila na nějaký nepředvídatelný problém. Zprvu byla iniciativa upravovat aktualizací pipeline vždy pro každého zákazníka a vývojový tým musel pokaždé vytvořit totožné vývojové prostředí jako zákaznický cloud, a to včetně rozšiřujících integrací. Bohužel průběh replikace prostředí byl velmi těžkopádný a pomalý. Proto přišel nový požadavek, který měl zkusit rozumně tyto integrace v pipeline zohlednit. V praxi se ovšem jednalo o ještě složitější proces, od kterého nakonec bylo ustoupeno. Nakonec bylo rozhodnuto, že se aktualizace budou provádět pouze ručně servisní inženýrem, který má aktualizací manuál. Ten se skládá z univerzálních kroků, ze kterých pipeline vycházela. Aktualizační manuál byl složen z jednotlivých kroků, které se měly během aktualizace cloudového prostředí provádět. Tento list kroků byl následně inženýrem doplněn o další manuální kroky, které bylo nutno při aktualizaci vykonat. Nevýhodou je, že tvorba takového plně funkčního aktualizacího manuálu je časově náročná a komplikovaně škálovatelná.

### 4.3.3. Vývoj a testování

Problémem, který v důsledku současné složité architektury vznikl, je pomalé vydávání nových verzí platformy a doručování těchto verzí zákazníkům. Vývoj je prováděn pomocí malých virtuálních prostředí, vytvořených ve vývojovém cloudu postaveném také na OpenStacku. Aby vývojářům bylo umožněno co největší pohodlí, tak je vytváření labů implementováno pomocí Jenkins pipeline. Tato pipeline je tvořena ze tří hlavních částí. První je generování modelu, poté jsou vytvořeny virtuální zdroje pomocí heat šablony a závěrečný krok je spuštění instalace prostředí. Instalace probíhá pomocí další pipeline, která spouští jednotlivé salt stavy pomocí salt API. Salt API je součástí konfiguračního serveru. Tento způsob automatizace je velmi vhodný, jelikož pomocí něj lze spustit prostředí, které je v mnohých parametrech velmi identické s reálnými produkčními clustery. Navíc laby samy k instalaci používají stejnou instalační proceduru, která je používána u zákazníků. Nevýhodou těchto virtuálních labů pro vývoj je především jejich velikost. I přestože jsou laby zmenšené, zabírají velké množství zdrojů. Například i pro vývoj drobné funkcionality do linux formule je nutno spustit lab, ve kterém nebudou využity všechny komponenty.

Samotné testování je tvořeno dvěma základními druhy. Prvními testy, kterým změny či opravy procházejí, jsou zabudované CI testy. Testují především syntaxi a možnost sestavení instalačního balíčku, tato část testování je prováděna automaticky Jenkinsem po commitu. Hlavní testování přichází při testování komponenty. To bývá prováděno na úrovni vývojového týmu, kdy je testována funkčnost požadovaných změn či vlastností komponenty. Druhý druh testování je nazýván integrační a slouží k testování jednotlivých komponent a změn najednou. Tyto integrační testy jsou dále rozděleny do tří skupin: testování na virtuálních labech, fyzickém hardware a tzv. scale testování. Integrační testování bývá prováděno speciálním QA týmem, který má k dispozici již hardwarové zdroje. Nevýhodou celého tohoto cyklu je doba testování chyb a nových vlastností. Každá změna musí být před svým zařazením do nové verze otestována s ostatními komponentami, většina nových vlastností vyžaduje i ruční kontrolu. Testování jedné malé změny může zabrat i několik hodin času QA inženýra.

## 5. Návrh budoucí architektury kontejnerové aplikace

V předchozích kapitolách byl představen koncept legacy aplikace a migrované aplikace. V této kapitole budou uvedeny směry, kterými lze řešení dále vyvíjet. Zároveň bude představen návrh a architektura nového řešení s důrazem kladeným na problémy zmíněné v kapitole číslo 4. Migrovaná aplikace částečně používá zastaralou monolitickou architekturu. I přesto, že jsou služby logicky rozděleny na jednotlivé role a komponenty, je samotné řešení velmi složité na provoz a vývoj. Při škálování virtuálního control plane je nutné vytvářet nové virtuální stroje, které konzumují hardwarové zdroje, které nejsou efektivně využity.

### 5.1. Současné možnosti

V této části kapitoly budou představeny současné možnosti, pomocí kterých lze vybranou aplikaci (MCP) dále zlepšit a udržovat tak, aby byla stabilnější a jednodušší jak pro zákazníky, tak i pro servisní inženýry a vývojáře, kteří aplikaci vyvíjejí. Prvním řešením je udržet a stabilizovat současné řešení, včetně životního cyklu aplikace. Obrovskou výhodou původního řešení je již zmiňovaná flexibilita, na kterou jsou zákazníci zvyklí a již nemají u konkurenčních řešení. Prvním bodem pro stabilizaci současného řešení je nutnost standardizovat model, jak již bylo zmíněno v popisu instalace aplikace. Každé prostředí je postavené na vygenerovaném modelu, který je následně upravován inženýrem, jenž prostředí instaluje. A jelikož chybí systém osvědčených postupů, tak si každý inženýr upravuje model způsobem, aby byl pro něho přehledný. V praxi lze totiž namodelovat výslednou konfiguraci mnoha různými způsoby. Problémy odlišných modelů se následně stávají spornými během upgradu cloudu, kdy jednotlivé modely musí být opět upraveny, aby přidávaly nebo odebíraly jednotlivou funkcionalitu. V současnosti se tento problém řeší přesunem metadat z produktové (cluster) úrovně na systémovou, aby ubylo parametrů a produktová úroveň byla přehlednější. Model souvisí i s Jenkins pipeline, pomocí kterých jsou jednotlivé akce vykonány. Koncept Jenkins pipeline byl popsán v kapitole 4. Na tento problém bohužel neexistuje jedno univerzální řešení, které by bylo v případě Mirantisu potřeba. Jak už bylo zmíněno v předchozí kapitole,

OpenStack je globální projekt, který je vyvíjen mnoha vývojáři napříč firmami. Většinu těchto firem spojuje nejen zdrojový kód, který vývojáři píší, ale také vznikající problémy propojené s provozem a správou OpenStack cloudů. Z toho důvodu začalo nezávisle na sobě vznikat několik projektů, jak tyto problémy vyřešit. Kontejnerizace OpenStacku je zcela nový směr a způsob, jak provozovat OpenStack. S myšlenkou kontejnerizovaného OpenStacku začaly firmy experimentovat během prvních verzí k8s v roce 2015. Ani jedna firma v tu dobu nebyla schopna nabídnout žádný stabilní projekt či produkt. Z tohoto důvodu vzniklo hned několik projektů, které problém řeší. Pro ilustraci byly vybrány projekty s největší uživatelskou základnou:

- **Kolla** – První OpenStack projekt zabývající se sestavováním a nasazením OpenStacku do kontejnerového prostředí. K nasazování je použit konfigurační management Ansible [52]. Dříve existoval projekt Kolla i pro Mesos a Kubernetes.
- **OpenStack-Helm** – Další projekt spravovaný OpenStack Foundation, který se zabývá tvorbou a nasazením kontejnerů. Jak již název napovídá, jedná se o sadu Helm chartů, pomocí kterých lze OpenStack nainstalovat a spravovat.
- **stackanetes** – Projekt, který vznikl ve spolupráci mezi firmami CoreOS a intel. Stackanetes se zabývá pouze instalací základních OpenStack služeb, používá především nástroje vyvinuté firmou CoreOS, např. kpm. Po akvizici CoreOS Red Hatem byl tento projekt pozastaven a ve vývoji se již nepokračuje.
- **AirShip** – Nejnovější řešení, které se stalo součástí OpenStack Foundation. Skládá se z kolekce několika projektů. Podle zdroje [53] jde spíše o framework, který řeší kompletní životní cyklus nejen OpenStacku, ale také operačního systému hostů a hardwaru. Celé AirShip řešení bylo vytvořeno firmami AT&T, South Korea Telecom (SKT) a Intel, které pomocí něho chtějí provozovat svou infrastrukturu pro 5G.
- **loci** – Projekt loci, na rozdíl od předchozích řešení, poskytuje pouze lehkotónážní Docker image, ovšem neřeší jejich nasazení a lifecycle management. Pomocí loci projektu lze jednoduše sestavovat image pro široké spektrum linuxových distribucí s odlišnými balíčkovacími systémy (Debian, CentOS, Suse, atd.).

I přestože vývojáři Mirantisu přispívají do zmíněných projektů, v současné době žádný z projektů není zcela optimální pro nahrazení současného MCP. Hlavním cílem Mirantisu je mít nástroj, který vyřeší zmíněné problémy uvedené v kapitole 4 a ulehčí operátorům správu prostředí. Dále je nutné, aby kontejnerizované MCP bylo možné nasadit do jakéhokoliv k8s clusteru. To je důležité především kvůli uživatelské komunitě, která by si mohla řešení otestovat sama a podílet se na vývoji. S kontejnerizací

přichází také řada nových problémů, které je nutno vyřešit. Prvním z nich je dodávání opravných záplat. V současné době jsou opravy dodávány debian balíčky. Balíčky jsou uloženy v repozitáři, který je unikátní pro každou verzi. Odlišení verzí je implementováno pomocí tagu. Tag je vždy unikátní pro danou verzi. V případě Mirantisu jsou vydané verze označeny rokem, měsícem a opravnou revizí, kdy byla verze vydána (např. 2018.4.1). Při použití kontejnerizace verzování balíčků už není potřeba, jelikož všechny komponenty budou spouštěny v rámci kontejnerů. Bezpečnostní záplaty budou fungovat následovně. Objeví-li se chyba v produktu, bude opraven zdrojový kód a sestaví se nový Docker image se stejným tagem, jaký byl použit pro vybranou verzi. Zákazník poté nebude muset stále kontrolovat, zda má nainstalované nejnovější balíčky a jejich závislosti. Veškeré aktualizace budou prováděny pomocí nasazení nových kontejnerů, které budou nasazeny formou rolling aktualizace, což znamená, že jednotlivé záplaty bude možno aplikovat bezvýpadkově. Největší problém se týká migrace stávajících zákazníků. Jedná se především o migraci workloadu ovládaného OpenStackem. Tento typ migrace je technicky velmi složitý. Aby migrace byla co nejméně bezvýpadková, je nutno vytvořit vedle současného clusteru také cluster kontejnerový, kde bude nové řešení naimplementováno. Postupně k němu budou připojeny compute servery s migrovaným workloadem. Velmi důležité je umístění disků jednotlivých instancí. Jsou-li disky uloženy v externí storage mimo cloud, tak migrace nebude obtížná. V tomto případě stačí pouze přesunout databázi, kterou lze převést pomocí obnovení záloh, které jsou periodicky vytvářeny. V databázi jsou uvedeny jednotlivé cesty na disky ve storage. Stačí připojit storage ke kontejnerovému řešení a instance znovu spustit. Pro disky ukládané přímo na souborový systém compute serveru bude tento proces časově i technicky náročnější. Proto se zvažuje několik možných řešení, jak s těmito prostředím nakládat. Tím nejpravděpodobnějším je nemigrovat a udržovat prostředí pomocí současných nástrojů. Zákazníkům budou dodávány pouze bezpečnostní záplaty. Nové vlastnosti budou dostupné až v rámci kontejnerového řešení, upgrade na kontejnerové řešení by se řešil přeinstalací celého prostředí.

## 5.2. Definice požadavků

V předchozí části kapitoly byla představena současná řešení, která ukazují jak představenou aplikaci MCP spravovat a vyvíjet dál. Pro jednodušší provoz OpenStacku nakonec byla zvolena možnost migrace do kontejnerového řešení. Orchestrátor je schopen zajistit spoustu úloh, které se musely řešit v předchozí architektuře externími nástroji, například vysoká dostupnost. Budoucí platforma musí být navržena tak, aby splňovala všechny funkční požadavky, které jsou od nového řešení očekávány.

- Prvním požadavkem je komplexnost. Podporuje stejné množství vlastností jako

předchozí řešení a je velmi snadno konfigurovatelné. To platí o rozšiřitelnosti řešení, která by měla podporovat stejné spektrum integrací jako současné řešení postavené na virtuálních strojích.

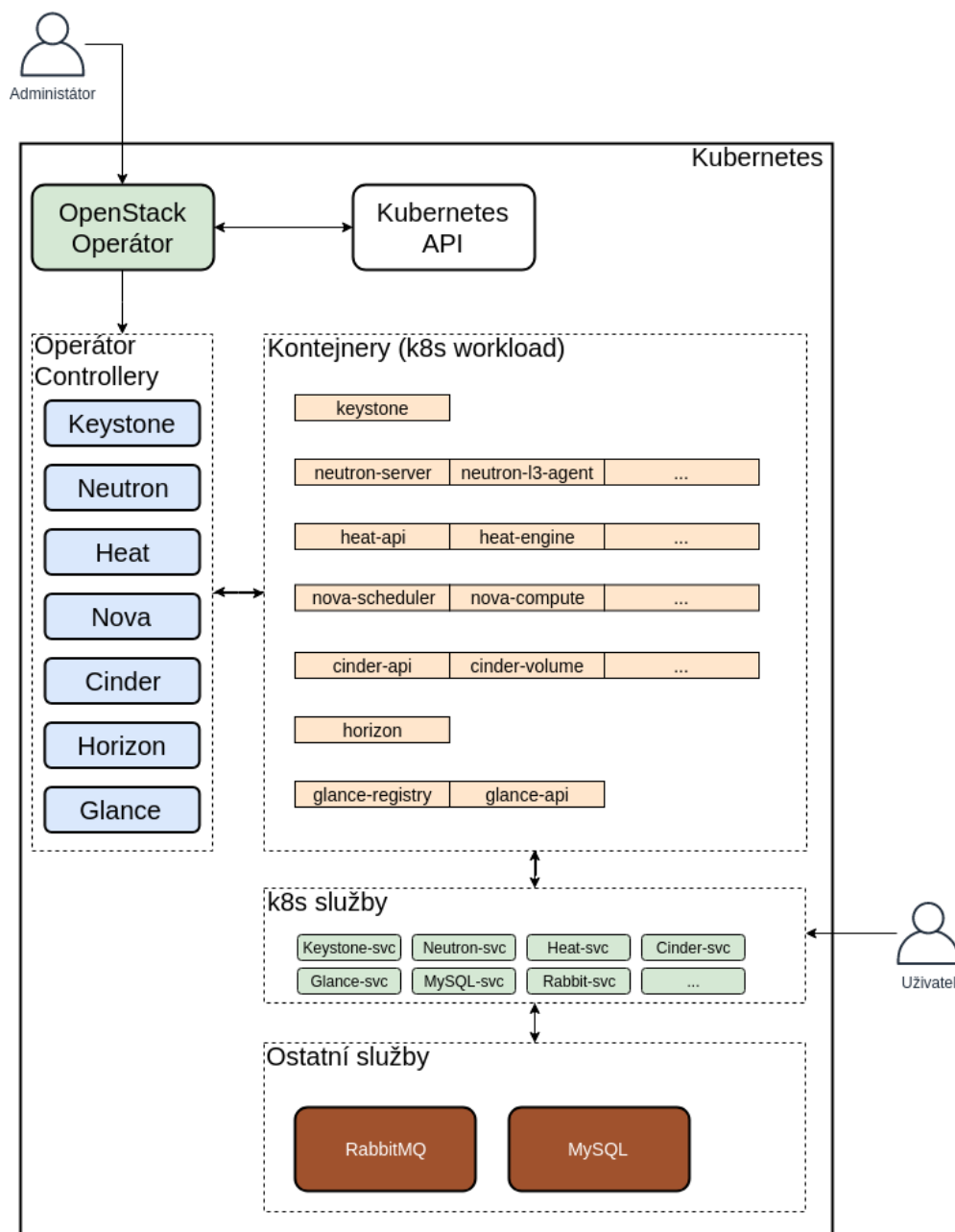
- Dalším aspektem je nasazování nového řešení. Řešení by nemělo být závislé pouze na jedné instalační metodě a mělo by podporovat více operačních systémů. Pro tento případ je výhodné používat k8s, který může být spuštěn nad jakoukoliv Linuxovou distribucí. Kontejnery jsou pak závislé jenom na zdrojích jádra. To znamená, že aplikace, která běží v kontejnerech, je závislá pouze na knihovnách, které jsou součástí kontejneru. Dále se otevírá možnost pro instalaci řešení do k8s clusterů třetích stran, které nejsou spravovány Mirantisem, jelikož OpenStack control plane se může brát jako klasická kontejnerová aplikace.
- Nejdůležitějším požadavkem pro nové řešení je mít systém, pomocí kterého lze jednoduše ovládat životní cyklus aplikace. Aplikaci by mělo jít jednoduše spravovat z jednoho místa a veškeré akce by měly být automatizované tak, že abstrahují operátory aplikace od komplexnosti.
- Dalším požadavkem je mít spuštěno pouze tolik replik OpenStack služeb, které jsou v daný moment potřeba. Efektivně využít hardwarové zdroje bez nutnosti spuštění replikovaných služeb. Díky k8s lze provádět rovněž dynamické škálování jednotlivých služeb. Pokud budou jednotlivé služby vytížené, automaticky se spustí další repliky.
- Řešení by mělo být modulární, aby do něho bylo možné jednoduše přidávat další projekty a integrace na další služby. Výhodou modulárních řešení z pohledu OpenStacku je kombinace několika odlišných verzí komponent mezi sebou. Často se stává, i když to není v dokumentaci doporučováno, že je spuštěno několik OpenStack komponent se staršími verzemi. Jedná se většinou o komponenty Cinderu, která zajišťuje block storage. Zákazníci totiž chtějí používat jejich storage systémy, které mají kompatibilní ovladače pouze se zastaralými verzemi Cinderu. V tomto případě je tedy nutné Cinder spouštět ve starší verzi.
- Poslední požadavek se týká testování. Díky odlehčenému modelu by pro testovací prostředí měly stačit mnohem menší clustery, které nezabírají tolik zdrojů a jejich spuštění trvá méně času.

### 5.3. Návrh budoucí architektury kontejnerové aplikace

Architektura systému je zobrazena na obrázku 5.1. Nová architektura bude postavena nad orchestrátorem k8s, jenž bude využíván pro orchestraci workloadu konceptu ope-

rátoru. Operátor se bude starat jak o nasazení aplikace, tak o její životní cyklus. Jednotlivé OpenStack komponenty budou rozděleny do kontejnerů. Jeden kontejner bude obsahovat vždy jednu službu komponenty. Například nova služba bude mít kontejnery pro služby nova-api, nova-scheduler, nova-conductor, nova-compute atd. Tyto kontejnery je nutné rozdělit do logických skupin (podů), v rámci kterých budou kontejnery spouštěny a škálovány. Protože podle zdroje [54] není vhodné mít všechny kontejnery od jedné služby v jednom podu. Při vytížení jednotlivých komponent, např. zmíněné nova, je nutné převážně škálovat api [55], nikoliv celou komponentu. Proto v rámci operátoru bude každá OpenStack komponenta mít svůj vlastní controller, který bude sloužit k provozu a hlídání nadefinovaných akcí. Tento controller rozšiřuje možnosti k8s reagovat na vytvořené CR (Custom Resource). CR je k8s objekt, je stejný jako k8s předdefinované objekty (např. pod, services, deployment, atd). Pokud je vytvořen požadavek s tímto typem objektu, k8s zkontroluje, zda existuje CRD předpis, který udává, zda daný cluster takový typ objektu zná. Pokud ano, je tento objekt poslán na operátor, který už podle zadaných parametrů dokáže provést manifestem stanovené akce. Zároveň lze pomocí controllerů sledovat jednotlivé stavy objektů a reagovat na ně. Při instalaci lze například ověřit, zda některé objekty existují, nebo zda je nutné počkat, než se vytvoří. Výhodou použití operátoru je především flexibilita přístupu. Samotný operátor je aplikací, jejíž logiku lze naprogramovat. Z obrázku se schématem je patrné, jak jednotlivé CR objekty spravují kontejnery s jednotlivými službami. Služby jsou mezi sebou propojeny pomocí cluster IP v rámci k8s služby. OpenStack je pro koncové uživatele dostupný přes Keystone public endpointy, které jsou vystavené na k8s službě jako externí ip adresa. Tato externí adresa je implementována formou k8s load balanceru. S těmito endpointy už může koncový uživatel pracovat stejně jako u klasického OpenStacku. Služby jsou pro přístup dostupné pomocí keystonec souboru a OpenStack CLI klienta nebo přes grafické rozhraní Horizon.





**Obrázek 5.1.:** Návrh kontejnerové architektury, zdroj: vlastní tvorba

Jak již bylo zmíněno v požadavcích, navrhované řešení je koncipováno, aby ho bylo možné nainstalovat do jakéhokoli k8s clusteru. Nezáleží na nástroji, jakým byl cluster nainstalován. Je nutno zmínit, že vybraný k8s cluster, do něhož bude OpenStack nainstalován, musí mít připojený nějaký typ storage. Pro některé komponenty, např. MySQL databáze či Keystone, je nutné uchovávat perzistentní data mimo kontejner. Instalace samotného OpenStacku je velmi jednoduchá. Celá aplikace bude instalována

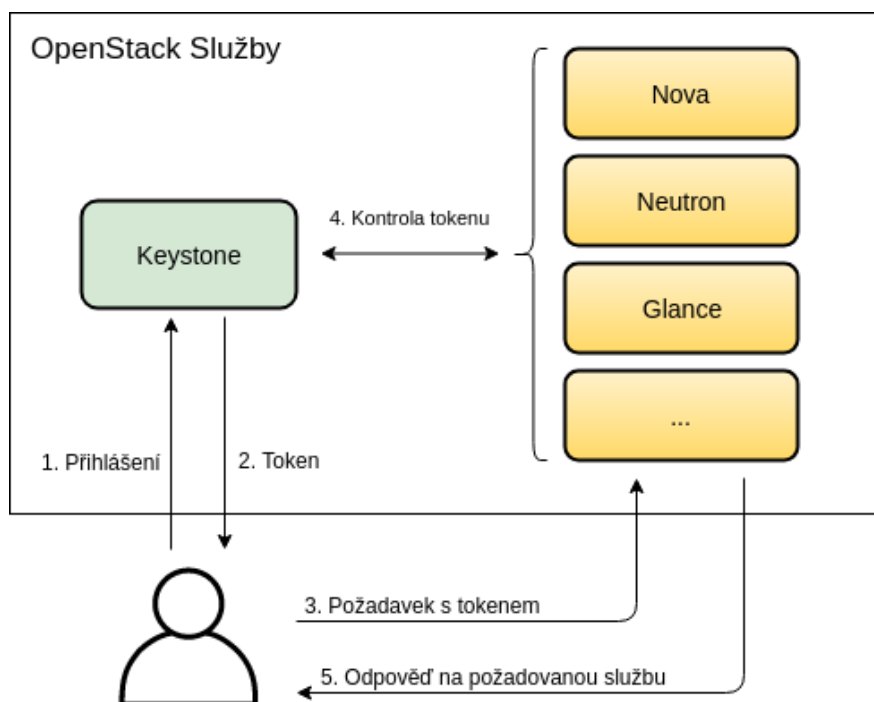
a řízena pomocí OpenStack operátoru. Tento operátor lze nainstalovat dvěma způsoby. Prvním způsob je instalace pomocí manifestu, tedy nejprve bude do k8s aplikován předpis, který spustí kontejner s operátorem. Ten vytvoří sadu CRD, a bude tak možno použít manifesty s CR, které jsou již specializované pro OpenStack. Před aplikací jednotlivých CR je nutné spustit podpůrné služby, jako jsou MySQL databáze a RabbitMQ, které jsou OpenStackem vyžadovány. Tyto dvě aplikace opět mohou být instalovány několika způsoby. Jelikož se jedná o aplikace, které mohou být provozovány nezávisle, nebyly proto přidány do OpenStack operátoru. Pro obě aplikace existují vlastní operátory. Po spuštění a konfiguraci těchto dvou aplikací lze popořadě aplikovat manifesty s CR, které spustí postupně control plane. Druhým způsobem je použití helm chartu pro OpenStack operátor, který má jednotlivé manifesty pro CRD a CR uložené v šablonách.

## 6. Implementace kontejnerového řešení

V předchozí kapitole byla popsána architektura aplikace po migraci do k8s clusteru. V této kapitole bude představena praktická implementace OpenStack operátoru, která používá zmíněnou architekturu. OpenStack je aplikace složená z několika komplexních komponent. V práci nebude popsána architektura a logika operátoru pro všechny základní OpenStack služby. Bude zde uvedena pouze ukázka implementace pro službu Keystone, která slouží uživatelům pro autorizace a ostatním službám k vzájemnému propojení. V úvodu kapitoly bude představena Keystone služba, vysvětleno, jakou roli v OpenStacku plní a jak standardně probíhá její instalace. Poté bude představen nástroj operator-sdk a nakonec bude popsána implementace Keystone v rámci OpenStack operátoru.

### 6.1. Keystone

Keystone je autentifikační a autorizační komponenta OpenStacku. Keystone neslouží pouze pro ověřování přístupu externích uživatelů, ale také pro OpenStack služby. Keystone je sám o sobě jeden velký API server, který slouží k propojování služeb a řízení přístupu. Jako backend používá Keystone SQL databázi, oficiální podpora existuje pro MariaDB, Postgres a MySQL. Existují i návody, jak například spustit Keystone i s modernějšími databázemi. Pro plnou funkcionalitu těchto databází je nutné upravit části keystonu, protože ne všechny patche byly přijaty komunitou. Keystone je autentifikační komponenta a často je vyžadováno integrovat ji s existujícími AAA systémy. Díky široké podpoře protokolů lze Keystone spojit se službami podporujícími OAuth, SAML či LDAP. Tyto integrace bývají vyžadovány převážně velkými zákazníky, kteří mají všechny své uživatele uložené a spravované prostřednictvím jednoho IDM systému.



**Obrázek 6.1.:** Schéma komunikace v rámci Keystone služby, zdroj: vlastní tvorba

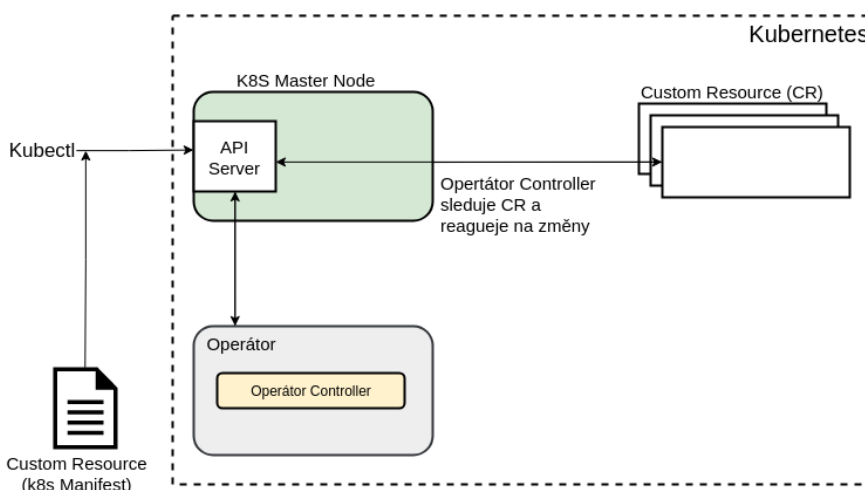
Schéma fungování komunikace v rámci Keystone je zobrazeno na obrázku číslo 6.1. Při přístupu do OpenStacku se uživatel ověří svými přihlašovacími údaji proti keystone. Pokud uživatel již existuje v databázi nebo IDM, tak mu je vygenerován token. Pomocí tohoto tokenu se lze dotazovat v rámci OpenStack služeb. Při dotazu na službu je vždy kontrolována validita tokenu. Při základním nastavení je token validní pouze 3600 vteřin. Pomocí validního tokenu lze vytvořit požadavek na danou službu. Komunikace mezi službami je v rámci OpenStacku reprezentována Keystone endpointy. Endpoint je tvořen pomocí url adresy s portem definovaným vždy pro daný typ služby. Na základě endpointu jednotlivé služby uvnitř cloudu komunikují. Existují tři základní typy endpointu: internal, public, admin. Jak již název napovídá, každý endpoint slouží k jinému typu přístupu. Proces komunikace je vždy stejný, ať uživatel používá Horizon rozhraní či OpenStack CLI klienta.

Instalace Keystone probíhá standardně pomocí debian balíčků podobně jako u všech ostatních komponent. Keystone je nutné spustit v režimu *mod\_wsgi*, obvykle se pro toto spuštění používá Apache, Nginx či jiný HTTP server. Keystone se liší od ostatních služeb svou prvotní konfigurací. Pro Keystone je potřeba provést proces zvaný bootstrap, který vytvoří endpoint, admin uživatele, doménu a službu. Tyto zdroje je nutné mít připravené dříve, než se pod Keystone začnou registrovat další služby. Pro bootstrap je zapotřebí mít již připravenou a připojenou Keystone databázi s předvytvořeným uží-

vatelem. Poté lze provést bootstrap. Existují dvě základní metody bootstrap: přes environmentální proměnné a spuštění jednotlivých příkazů nebo přes keystone-manage nástroj. První zmíněný způsob podle zdroje [56] není bezpečný. Druhý bootstrap pomocí keystone-manage je velmi jednoduchý, stačí tomuto příkazu pouze předat parametry a on všechny akce provede automaticky. Tento typ bootstrapu je uveden v ukázce kódu 6.5 v implementaci Keystone controlleru na řádce 27.

## 6.2. Operator SDK

Koncept operátoru byl již představen ve druhé kapitole, týkající se kontejnerizace. Operátor je koncept, pomocí kterého lze skrze k8s API vytvářet CRD. V CRD lze nadefinovat vlastní typy objektů a pomocí operátoru pak naprogramovat logiku a chování objektu v clusteru. Operátor by se dal označit jako aplikace, která skrze k8s API ovládá jinou aplikaci. Tyto operátory lze psát pomocí různých programovacích jazyků. Jazyk musí obsahovat pouze knihovnu na komunikaci s k8s API. V rámci práce byl pro tvorbu operátoru vybrán nástroj operátor-framework, který standardizuje tvorbu operátorů. Nástroj vytvořila společnost CoreOS (nyní udržován Red Hatem). Operátor-framework podporuje pro tvorbu operátoru pouze jediný programovací jazyk, tím je Golang, dále uváděn jako Go. Go je jazyk od americké společnosti Google, byl vytvořen v roce 2007, v roce 2009 byl vydán veřejně. Go je kompilovaný statický typovaný jazyk. Výsledek zkompilevaného Go programu je jeden linkovaný binární soubor, který je možné spustit na širokém spektru platform. Pro běh aplikace není nutné instalovat specializované prostředí, jako například pro jazyk Java (JRE). Důvodem, proč byl SDK nástroj napsán v Go, je fakt, že veškeré kontejnerové technologie využívají tento jazyk, např. Docker, Kubernetes, Prometheus.



**Obrázek 6.2.:** Schéma komunikace v rámci k8s operátoru, zdroj: vlastní tvorba

Operátor-framework je složen z několika projektů. Hlavním z nich je projekt operátor-sdk, který slouží k vygenerování kostry projektu a následnému sestavení operátoru. Struktura operátoru-sdk je uvedena v příloze A. Sdk se ovládá pomocí operátor-sdk CLI. Sdk je složeno ze dvou základních částí: controlleru a API. Na schématu 6.2 je zobrazena komunikace v rámci operátoru. Operátor se poté vytváří pomocí příkazu `operator-sdk new <Název operátoru>`. Tento příkaz vygeneruje základní kostru pro tvorbu operátoru. Hlavní částí je soubor `main.go`, pomocí kterého je operátor spouštěn a zároveň slouží k registraci controlleru a čtení z parametrů z CR API. Dále jsou vygenerovány manifesty pro instalaci service account a role operátoru samotného. Součástí předvygenerované kostry je také `Dockerfile`, podle kterého se při příkazu `operator-sdk build <Název operátor image>` sestaví binární soubor a vytvoří Docker image pro operátor.

Generování CR API slouží k vytvoření api adresáře a CR. Pro vytvoření k8s API je nutné operátoru-sdk sdělit dva parametry: api verzi a druh api. CR je yaml manifest, jímž se definují nové k8s objekty, které jsou operátorem rozpoznány. V api adresáři je dále vygenerováno několik go souborů. Hlavním z nich je soubor pro definování parametrů pro CR, tento soubor se jmenuje stejně jako vytvořený druh API, např. `nginx_types.go`. V souboru se pomocí struktur nadefinují proměnné, které se dají pro vytvoření k8s objektu vyplnit. Rozlišují se dva základní typy proměnných. Prvním z nich jsou proměnné typu Spec. To jsou proměnné, jež jsou naplněny CR souborem. Proměnné typu Status se plní během doby, kdy je operátor spuštěn. Funkcionalita status proměnných bude vysvětlena v části kapitoly, kde bude popsána implementace Keystone role. Po změně tohoto souboru, který obsahuje jednotlivé struktury proměnných, je nutné přegenerovat DeepCopy metody uložené v souboru `zz_generated.deepcopy.go`. Tento soubor je unikátní v rámci api adresáře. Přegenerování se provádí pomocí operátor-sdk příkazu `operator-sdk generate k8s`.

Druhou částí je controller. Jeho generování probíhá stejným systémem jako API. Na rozdíl od API má controller pouze jeden jediný go soubor, který obsahuje logiku operátoru pro daný druh API. Základní částí jakéhokoliv controlleru je Reconcile funkce. V této funkci je napsána celá logika akcí, které mají být spuštěny. Reconcile si lze představit jako nekonečný cyklus, který postupně prochází zdrojový kód reconcile funkce od shora dolů a spouští požadované akce. Rozdíl od nekonečného cyklu je ten, že Reconcile není spuštěn periodicky, ale vždy po změně ve zdrojích sledovaných operátorem. Změna může být vyvolána například aplikací nové konfigurace v CR. Pokud operátor zjistí změnu, spustí funkci Reconcile znovu a při správném naprogramování bude provedena pouze daná změna a všechny akce provedené již v předchozích cyklech Reconcile funkce jsou ignorovány. V rámci controlleru lze nadefinovat tzv. watch funkce, které slouží ke sledování změn jednotlivých zdrojů. Pomocí watche lze sledovat veš-

keré změny v k8s API. Je tedy možné dynamicky reagovat i na změny, které se dějí uvnitř k8s clusteru. Návod jak operátor-sdk nainstalovat a zprovoznit je dostupný na [57].

### 6.3. Dockerfile

Celé řešení je navrhováno tak, aby bylo možné použít i upstream loci image. Z důvodu kontroly jednotlivých částí kontejnerizace a bezpečnosti aplikace bylo rozhodnuto sestavit vlastní Docker image. Zdrojové Dockerfily jsou k dispozici na GtiHubu [58]. Jak doporučuje oficiální Docker dokumentace [59], je pro jednotlivé image komponenty zvoleno použití konceptu základních imagí. Tedy každá služba má jednu globální image, která je společná pro všechny služby v rámci komponenty. V základní image jsou nainstalovány všechny balíčky potřebné pro spuštění aplikace. Tento koncept je vhodný především u komponent, jež obsahují velké množství služeb, například služba nova. Výhoda se projeví především při doručování opravných aktualizací, ty jsou aplikovány na úrovni základní image. Po změně v této image je nutné znovu sestavit všechny kontejnery. Protože Docker využívá systém vrstev a unionFS, tak sestavování image postavených na této základní imagi je téměř okamžité.

**Kód 6.1:** Základní Dockerfile pro Keystone image, zdroj: vlastní tvorba

```

1 FROM ubuntu:16.04
2
3 RUN apt update && \
4     apt install curl apt-transport-https -y
5
6 RUN echo "deb [arch=amd64] https://mirror.mirantis.com/
   /2019.2.0/openstack-queens/xenial xenial main" >>
   /etc/apt/sources.list.d/mcp_openstack.list &\
7     echo "deb [arch=amd64] https://mirror.mirantis.com/
   /update/2019.2.0/openstack-queens/xenial xenial main" >
   /etc/apt/sources.list.d/mcp_openstack.list
8
9 RUN curl -fsSL http://mirror.mirantis.com/
   /2019.2.0/openstack-queens/xenial/archive-queens.key |
   apt-key add -
10
11 RUN apt update && \
12     apt install uwsgi uwsgi-plugin-python -y &&\
13     apt install keystone python-openstackclient -y
14
15 COPY entrypoint.sh /entrypoint.sh

```

```
16 RUN chmod +x /entrypoint.sh
```

Na ukázce 6.1 je zobrazen base Dockerfile pro Keystone. Jako základní zdrojová image je použita distribuce Ubuntu ve verzi 16.04. Vybrána byla především z důvodu, že celé MCP používá pro doručování svých verzí OpenStacku debian balíčky. Na řádce číslo 3 je uvedena instalace nutných nástrojů k přidání externího http repozitáře. Přidání repozitáře je implementováno pomocí programu echo. V Dockerfile jsou přidány dva typy repozitářů. Základní repozitář, ve kterém jsou uloženy stabilní verze balíčků, a repozitář update, jenž obsahuje balíčky s bezpečnostními záplatami. Oba dva repozitáře jsou downstream a jsou spravovány firmou Mirantis. Poté je pomocí nástroje curl stažen GPG klíč, který slouží pro apt. Na řádce číslo 11 je zobrazena instalace samotného Keystonu a programů, které jsou nezbytné k jeho spuštění. Pro kontejnerizaci je využit OpenStack z vydání MCP 2019.2.0, což je verze OpenStack Queens. Queens je zároveň nejnovější podporovanou verzí OpenStacku v MCP [60]. Na dalších řádcích je přesun a úprava práv bashového skriptu, který slouží jako entrypoint pro kontejner, viz ukázka kódu 6.2. Ve skriptu kromě jiných příkazů je nejdůležitější samotné spuštění Keystone služby na řádce 12. Jelikož Keystone je od verze Liberty, je nutné spouštět ho jako http server s mod\_wsgi módem. V původním řešení založeném na VM byl Keystone implementován pomocí Apache http serveru. Pro kontejnerové řešení je zvolen projekt uWSGI, který zaručuje stejnou funkcionalitu pro spuštění s následujícími parametry. Prvním parametrem je plugin, jelikož uWSGI neumí v základu spouštět python kód. Druhým je port, na kterém bude aplikace vystavena. Zde byl zvolen port 5000, jenž je standardní pro Keystone službu. Dále je nutné specifikovat počet workeru. Počet workeru je vypočítán na základě počtu cpu, které má kontejnerový host k dispozici. Výpočet je uveden na řádce 4. Posledním parametrem je identifikace Keystone WSGI služby. Ta je implementována pomocí programu which, který slouží k ukázkání cesty na spustitelný soubor příkazu keystone-wsgi-public.

**Kód 6.2:** Entrypoint pro Keystone image, zdroj: vlastní tvorba

```
1  #!/bin/bash
2
3  if [[ -f /proc/cpuinfo && -z "$WORKERS" ]]; then
4      WORKERS=$(( $(grep -c processor /proc/cpuinfo) * 2 + 1 )
5  fi
6
7  BIND_PORT=5000
8
9  echo 'export PS1="\u@keystone" \W # "' >> /root/.bash_profile
10
11 echo "Starting keystone ${COMPONENT} service"
```



```

12 uwsgi --plugin http --plugin python --http :${BIND_PORT}
    --workers ${WORKERS} --wsgi-file $(which
    keystone-wsgi-public)

```

V práci není použita služba keystone-wsgi-admin, jež využívá port 35357. Celá funkcionalita této služby je převedena pod keystone-wsgi-public. Podle zdroje [61] bude ve vydání OpenStacku Stein služba keystone-wsgi-admin z projektu odstraněna.

## 6.4. Implementace logiky operátoru

V této části kapitoly je popsána praktická implementace Keystonu v rámci OpenStack operátoru. Zdrojový kód aplikace je dostupný v GitHub repozitáři [62]. OpenStack operátor využívá Go ve verzi 1.12 a používá Go.mod pro udržování jednotlivých závislostí v rámci aplikace. OpenStack včetně Keystonu potřebuje mít pro svůj správný běh i podpůrné služby, jedná se především o databázi a frontu. V řešení jsou použity technologie MySQL a RabbitMQ. Tyto dvě technologie nejsou součástí samotného operátoru. Obě technologie byly do k8s clusteru nainstalovány pomocí stabilního helm chartu a pro každou z nich byl pro OpenStack vytvořen specializovaný list uživatelů.

**Kód 6.3:** Ukázka části Keystone CR, zdroj: vlastní tvorba

```

1  apiVersion: openstack.lotharkatt.io/v1alpha1
2  kind: Keystone
3  metadata:
4    name: keystone
5  spec:
6    release: Queens
7    replicas: 1
8    image: quay.io/lotharkatt/keystone-wsgi-public
9    config: |
10     [DEFAULT]
11     transport\__url =
12         rabbit://openstack:cloudlab@msg-rabbitmq:5672/
13     [database]
14     connection =
15         mysql+pymysql://keystone:cloudlab@mysql/keystone
16     [token]
17     provider = fernet
18     region: RegionOne
19     service:
20       - name: nova
21         type: compute

```

```

20     description: "Compute Service"
21     endpoint:
22     - port: 8774
23       protocol: http
24       availability: public, internal, admin
25   - name: cinder
26     type: volume
27     endpoint:
28     - port: 8776
29       protocol: http
30       availability: public, internal, admin
31     description: "Block Storage"
32   users:
33   - name: nova
34     password: cloudlab
35   - name: cinder
36     password: cloudlab

```

Na ukázce kódu číslo 6.3 je možné vidět část CR pro objekt Keystone. Pomocí tohoto manifestu je služba spuštěna a konfigurována. CR manifest předává data controlleru, který vykonává naprogramovanou logiku. Počáteční struktura je velmi podobná klasickému manifestu, je zde specifikováno API, Kind a Metadata. Odlišná je především sekce Spec. Spec obsahuje parametry, jako jsou počet replik a jaká image bude použita. Pod parametrem config se nachází konfigurace, která bude použita v konfiguračním souboru keystone.conf. V sekci Spec jsou uvedeny další parametry, pomocí nichž jsou registrovány jednotlivé OpenStack služby pod Keystone. Na ukázce číslo 6.3 je zobrazena definice pouze pro služby Nova a Cinder.

#### Kód 6.4: Ukázka struktury KeystoneSpec, zdroj: vlastní tvorba

```

1 type KeystoneSpec struct {
2   Replicas int32          `json:"replicas"`
3   Image    string         `json:"image"`
4   Config   map[string]string `json:"config"`
5   Service []KeystoneService `json:"service"`
6   Users   []KeystoneUser   `json:"users"`
7 }

```

Jak již bylo naznačeno v části operátor sdk, CR manifest je generován pomocí souboru types, jenž je tvořen v rámci api. Struktury ze souboru types slouží k mapování CR do controlleru, viz ukázka kódu číslo 6.4. Struktura KeystoneSpec slouží pro mapování Spec parametrů v CR manifestu. Ve struktuře je vždy zobrazen název proměnné, datový typ a pojmenování proměnné v CR manifestu. U proměnné Service je možné vi-

dět, že v rámci struktury lze použít i nově nadefinované datové typy. V případě Service se jedná o pole struktur. Pomocí vnořených struktur lze následně vytvořit komplexní definici, se kterou je možné ve zdrojovém kódu controlleru velmi efektivně pracovat.

Mapování parametrů ze struktury na kontejner probíhá ve spuštěné funkci Reconcile. Pro vytvoření jednotlivých k8s objektů je potřeba vytvořit vybraný objekt a naplnit ho požadovanými daty. Pro tvorbu objektů byly použity k8s api knihovny, které jsou součástí operátor sdk a obsahují již předvytvořené k8s objekty. Jelikož se tvorba těchto objektů v operátoru často opakuje, bylo rozhodnuto vytvořit specifickou funkci, která je použitelná napříč všemi controllery pro OpenStack služby. Ukázka funkce pro tvorbu Keystone deploymentu je zobrazena na kódu číslo 6.5. Objekt je poté předán operátoru pomocí funkce Create(), která objekt v k8s vytvoří. Create je součástí operátor clientu, jenž je ve struktuře ReconcileKeystone, která vlastní Reconcile funkci pro daný controller. Operátor client slouží také k ověřování skutečnosti, zda byl daný objekt již vytvořen.

**Kód 6.5:** Ukázka funkce createDeploymentAPI(), zdroj: vlastní tvorba

```

1 func createDeploymentAPI(cr *keystonev1alpha1.Keystone, name
    string, port int32, image string, label string)
    *appsv1.Deployment {
2   replicas := cr.Spec.Replicas
3   ls := common.CreateLabel(label)
4   return &appsv1.Deployment{
5     ObjectMeta: metav1.ObjectMeta{
6       Name:      name,
7       Namespace: cr.Namespace,
8     },
9     Spec: appsv1.DeploymentSpec{
10      Replicas: &replicas,
11      Selector: &metav1.LabelSelector{
12        MatchLabels: ls,
13      },
14      Template: corev1.PodTemplateSpec{
15        ObjectMeta: metav1.ObjectMeta{
16          Labels: ls,
17        },
18        Spec: corev1.PodSpec{
19          InitContainers: []corev1.Container{
20            {
21              Name: name + "-init",
22              Image: "quay.io/lotharkatt/keystone-wsgi-public",
23              Command: []string{

```

```

24     "sh",
25     "-c",
26     "keystone-manage fernet_setup --keystone-user root
      --keystone-group root ;" +
27     "keystone-manage db_sync ;" +
28     "keystone-manage bootstrap --bootstrap-password
      cloudlab --bootstrap-username admin
      --bootstrap-project-name admin
      --bootstrap-role-name admin
      --bootstrap-service-name keystone
      --bootstrap-region-id RegionOne
      --bootstrap-admin-url http://keystone-svc:5000
      --bootstrap-public-url http://keystone-svc:5000
      --bootstrap-internal-url http://keystone-svc:5000",
29     },
30     },
31     },
32     Containers: []corev1.Container{
33     {
34         Name:         name,
35         Image:         image,
36         ImagePullPolicy: corev1.PullAlways,
37         VolumeMounts: []corev1.VolumeMount{
38         {
39             Name:         "keystone-cm",
40             MountPath:    "/etc/keystone/keystone.conf",
41             SubPath:       "keystone.conf",
42         },
43         },
44         Ports: []corev1.ContainerPort{
45         {
46             Name:         name,
47             ContainerPort: port,
48         },
49         },
50     },
51     },
52     Volumes: []corev1.Volume{
53     {
54         Name: "keystone-cm",
55         VolumeSource: corev1.VolumeSource{
56             ConfigMap: &corev1.ConfigMapVolumeSource{
57                 LocalObjectReference: corev1.LocalObjectReference{

```

```

58         Name: "keystone-configmap",
59     },
60 },
61 },
62 },
63 },
64 },
65 },
66 },
67 }
68 }

```

V části Keystone bylo zmíněno, že služba potřebuje před svým spuštěním projít procesem bootstrapu. V rámci operátoru je bootstrap naimplementován pomocí init kontejneru. Init kontejner je speciální typ kontejneru, který je spuštěn pouze jednou (a to při vytvoření podu) a má za úkol vykonat akci, která připraví prostředí na spuštění kontejneru s aplikací. Kontejner danou akci spustí, a pokud je správně vykonána, tak se vypne a spustí aplikační kontejner. Pomocí init kontejnerů jsou často implementovány akce, jako jsou připojení volumnů či inicializační konfigurace. Definice init kontejneru je uvedena na řádce 19 v ukázce 6.5. Z ukázky pro init kontejner je patrný název a kontejnerová image, která bude spouštět list uvedených akcí. Pro spuštění akcí je nutné definovat shell, ve kterém budou příkazy spuštěny. První příkaz slouží pro vygenerování fernet tokenu, což je typ tokenu, který se používá pro autorizaci a autentifikaci. Poté jsou pomocí příkazu *keystone-manage sync\_db* vytvořeny tabulky pro Keystone službu v databázi. Poslední příkaz provede bootstrap proces, ve kterém se vytvoří uživatel admin, služba a endpointy pro OpenStack. Tyto informace jsou následně uloženy do databáze. Díky využití databáze není nutné tyto údaje sdílet napřímo s aplikačním kontejnerem, který si po spuštění potřebné informace zjistí sám z databáze.

Podle zdroje [63] by se aplikace vyvíjené pro k8s měly konfigurovat skrze environmentální proměnné. Problémem Keystonu je, že tato služba nebyla od začátku vyvíjena na běh v kontejnerovém prostředí a pro svůj běh a integrace používá ke konfigurování pouze konfigurační soubory. Pro konfiguraci takovýchto aplikací v k8s slouží objekt – tzv. ConfigMapy, který dokáže doručovat konfiguraci jak přes environmentální proměnné, tak přes konfigurační soubor. V praxi to funguje tak, že konfigurační soubor je připojen jako volume. Konfigurace připojení ConfigMapy je v ukázce kódu 6.5 na řádce 39. Samotná ConfigMapa je vytvořena pomocí funkce *createConfigMap()*, viz ukázka kódu 6.6. K8s API očekává data, která mají tvořit ConfigMapu v datovém formátu mapa, tzn. klíč-hodnota. Tento formát je velmi vhodný, protože do jedné ConfigMapy lze zapouzdřit více konfiguračních souborů. Každý konfigurační soubor má v rámci ConfigMapy svůj unikátní klíč, pomocí kterého lze ConfigMapu rozdělit na

více konfiguračních souborů. U této metody, kdy je použito více konfigurací v jedné ConfigMapě, je vždy nutné specifikovat parametr SubPath. Tomuto parametru je nutno specifikovat klíč konfiguračního souboru. V projektu je konfigurace doporučována prostřednictvím CR manifestu, viz ukázka 6.3.

**Kód 6.6:** Ukázka funkce pro generování ConfigMap, zdroj: vlastní tvorba

```

1 func createConfigMap(cr *keystonev1alpha1.Keystone)
    *corev1.ConfigMap {
2     return &corev1.ConfigMap{
3         ObjectMeta: metav1.ObjectMeta{
4             Name:      cr.Name + "-configmap",
5             Namespace: cr.Namespace,
6         },
7         Data: cr.Spec.Config,
8     }
9 }

```

Při instalaci OpenStacku a konfiguraci Keystonu se stále opakují podobné úlohy, proto bylo nutné najít cestu jak ovládat a konfigurovat Keystone objekty služby prostřednictvím CR. Bylo rozhodnuto doplnit strukturu ReconcileKeystone o další client, který dokáže ovládat OpenStack. Tento client se jmenuje osclient a je implementován pomocí souboru knihoven zvaných Gophercloud [64]. Gophercloud poskytuje OpenStack SDK pro jazyk Go a lze pomocí něho ovládat OpenStack. Problémem, který bylo v rámci logiky operátoru nutno vyřešit, je přihlášení do již vytvořeného Keystonu. Pro vytvoření instance osclienta je nutno specifikovat několik povinných parametrů, většina z nich je statická, jen parametr IdentityEndpoint je dynamický. Statické parametry mají pevně stanovené hodnoty, změna je možná pouze prostřednictvím CR manifestu. Parametr IdentityEndpoint očekává IP adresu a port, na kterém je Keystone vystaven. Tato adresa je v k8s clusteru umístěná na k8s službě, která je s kontejnerem propojena. Pro Keystone je použita adresa externího typu, tzn. že adresa je dostupná i mimo k8s cluster. Je to především z důvodu, aby byla možnost připojení na OpenStack také pomocí keystonec souboru. Zmiňovaná adresa je poskytována SDN řešením a je po každé jiná, proto je nutné zaručit, aby tento parametr byl do operátoru předán dynamicky. Na uložení adresy je použita výše zmiňovaná proměnná typu Status, do které lze uložit hodnotu, která po doběhnutí Reconcilu nebude smazána. Prostřednictvím vytvořeného objektu pro Keystone k8s službu lze jednoduše IP adresu přechíst a nahrát do Status proměnné. Pokud jsou všechny parametry pro instanci validní, je vytvořen osclient, se kterým lze ovládat OpenStack. Použití osclienta je zřejmé na ukázce kódu 6.7. Z ukázky je patrné vytváření uživatelů pro OpenStack. Pomocí *instance.Spec.Users* si operátor vyčítá list uživatelů z CR, poté je list uložen do proměnné *users*. Cyklus

prochází listem a v rámci cyklu je volána funkce `CreateUser()`, která uživatele z manifestu vytvoří. Vstupní parametr této funkce je pouze objekt `User`. Stejná funkcionalita je naimplementována pro tvorbu keystone služeb a endpointů. OpenStack operátor umí OpenStack objekty z CR pouze vytvářet, nikoliv mazat. Tato funkcionalita nebyla implementována z důvodu, že by nezkušený uživatel mohl svou neopatrnou úpravou v Keystone CR manifestu nevratně poškodit konfiguraci celého OpenStack cloudu.

**Kód 6.7:** Ukázka tvorby Keystone uživatelů, zdroj: vlastní tvorba

```
1 // Create user
2 users := instance.Spec.Users
3 for _, user := range users {
4   err = r.osclient.CreateUser(&osclient.User{user.Name,
5     user.Password, "admin", "default"})
6 }
```

Na obrázku číslo 6.3 je vyobrazena Keystone služba nasazená do k8s clusteru. List jednotlivých podů obsahuje kromě keystonu také operátor, MySQL databázi a RabbitMQ. V přehledu k8s služeb lze vidět zmiňovanou externí adresu pro Keystone. V implementovaném prototypu je pro každou OpenStack službu nakonfigurována externí ip adresa. V ostrém nasazení by se problém s externím přístupem řešil přidáním k8s API gateway (např. Ambassador), která by zajistila lepší správu a především větší bezpečnost.

NAME	READY	STATUS	RESTARTS	AGE
pod/database-mysql-68b77ff666-95djh	1/1	Running	0	2d
pod/keystone-bd8887f6c-dfs6z	1/1	Running	0	45h
pod/msg-rabbitmq-0	1/1	Running	0	5m23s
pod/openstack-operator-7f96cd99b7-rw7dk	1/1	Running	0	45h

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/database-mysql	ClusterIP	10.98.175.230	<none>	3306/TCP	2d
service/keystone-svc	ClusterIP	10.102.79.254	34.65.103.135	5000/TCP	45h
service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	2d1h
service/msg-rabbitmq	ClusterIP	10.108.19.38	<none>	4369/TCP, 5672/TCP, 25672/TCP, 15672/TCP	5m23s
service/msg-rabbitmq-headless	ClusterIP	None	<none>	4369/TCP, 5672/TCP, 25672/TCP, 15672/TCP	5m23s
service/openstack-operator	ClusterIP	10.111.80.98	<none>	8383/TCP	45h

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/database-mysql	1/1	1	1	2d
deployment.apps/keystone	1/1	1	1	45h
deployment.apps/openstack-operator	1/1	1	1	45h

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/database-mysql-68b77ff666	1	1	1	2d
replicaset.apps/keystone-bd8887f6c	1	1	1	45h
replicaset.apps/openstack-operator-7f96cd99b7	1	1	1	45h

NAME	READY	AGE
statefulset.apps/msg-rabbitmq	1/1	5m23s

**Obrázek 6.3.:** Spuštěná služba Keystone v k8s, zdroj: vlastní tvorba





## 7. Porovnání klasického a kontejnerového řešení

V této kapitole bude kontejnerizovaný OpenStack porovnán a otestován se stávajícím VM řešením. Kapitola je rozdělena na tři základní části. V první části jsou představena jednotlivá testovací prostředí, na kterých bude testování prováděno. Druhá část je zaměřena na testování jednotlivých řešení podle daných scénářů. Tyto scénáře vychází z funkčních požadavků stanovených návrhem nové architektury. Ve třetí závěrečné části je provedeno porovnání zmíněných architektur a jejich implementace.

### 7.1. Definice testovacích prostředí

Pro testovací prostředí byla vytvořena dvě odlišná prostředí, první pro současnou a druhé pro kontejnerovanou verzi. Je nutné zmínit, že obě prostředí byla z důvodu nedostatku hardwarových zdrojů simulována virtuálně. Obě řešení byla spuštěna ve virtuálních strojích nad OpenStackem.

Pro VM řešení byl použit virtuální MCP lab, který se v Mirantisu standardně využívá pro vývoj a testování. Tento lab byl vytvořen projektem OpenStack Heat, základní konfigurace byla provedena prostřednictvím SaltStacku. Celé prostředí nebylo doinstalováno, jednalo se pouze o vytvoření virtuálních zdrojů jako je networking a instalace SaltStacku, která je potřebná pro MCP.

Kubernetes prostředí, které je vyžadováno pro kontejnerovaný OpenStack, bylo též vytvořeno prostřednictvím Heatu. Samotný Kubernetes byl nainstalován pomocí SaltStack formule. Pro networking bylo použito SDN Tungsten Fabric. Storage pro k8s clusteru byla řešena pomocí sdíleného souborového systému GlusterFS, ten byl prostřednictvím projektu Heketi připojen do k8s clusteru. Heketi projekt byl zvolen z důvodu dynamického vytváření perzistentních volumů pro k8s.

#### 7.1.1. Testovací scénáře

V této části kapitoly jsou uvedeny jednotlivé testovací scénáře, na kterých bude popsáno, jak daný testovací scénář probíhá v obou prostředích. Jednotlivé testovací scénáře jsou vybrány z funkčních požadavků na kontejnerizaci stanovených v kapitole 5.

### 7.1.2. Instalace řešení

Cílem prvního testovacího scénáře je ověřit možnosti instalace a vyhodnotit rychlejší instalaci. Pro instalaci Mirantis OpenStacku byla použita sada instalačních nástrojů, které jsou používány v praxi. Instalace proběhla standardním instalačním postupem popsaným v kapitole 4.2.3 skrze Jenkins pipeline. Průběh instalace trval hodinu a dvacet minut.

Instalace kontejnerizované aplikace probíhala aplikováním jednotlivých manifestů na k8s prostředí. Před aplikováním manifestů pro jednotlivé komponenty OpenStack operátoru bylo nutné nainstalovat a nakonfigurovat do prostředí podpůrné služby MySQL a RabbitMQ. Tyto dvě služby byly nainstalovány pomocí helmu, jejich konfigurace byla provedena příkazy z přílohy B. Poté již bylo možné aplikovat jednotlivé instalační manifesty, jejich instalace trvala cca 15 minut.

### 7.1.3. Vysoká dostupnost

Cílem druhého scénáře bylo ověřit, jak bude OpenStack reagovat při vypínání jednotlivých služeb a zda testovaná prostředí odolají výpadku jednotlivých serverů, nad kterými jsou spuštěny. Tento test byl v VM řešení prováděn manuálně. Jako první byly na serveru ctl01 vypnuty všechny OpenStack služby. Dostupnost byla ověřována opakovaným dotazováním se na API jednotlivých OpenStack služeb. Druhý test spočíval v restartování jednotlivých ctl serverů, bylo zajištěno, aby vždy alespoň jeden server zůstal zapnutý. Testování se opět provádělo pomocí dotazů na API. Oba testy pro OpenStack s virtuálním control planem proběhly úspěšně. Jediná problematická část, která měla potíže s výpadky, byly dbs servery, na kterých byla spuštěná databáze. Ta vždy při opětovném připojení do clusteru musela replikovat nová data. Jelikož na testování v testovacím prostředí byl OpenStack nainstalován nově, tak se v databázi vyskytovalo málo záznamů a replikace byla téměř okamžitá.

V kontejnerovém řešení jsou jednotlivé služby nasazovány v rámci k8s podů. Tyto pody jsou nasazovány prostřednictvím Deploymentu. Test vypínání služeb byl v k8s prostředí vykonán pomocí odstraňování jednotlivých podů. Druhý test byl zinscenován restartem jednotlivých compute hostů, na nichž byly kontejnery spuštěny. Verifikace dostupnosti API byla shodná jako na VM prostředí. Oba dva testy proběhly bez problémů. Pody byly po smazání nasazovány ReplicaSetem zpět do prostředí. Objekt ReplicaSet byl vytvořen v rámci k8s Deploymentu.

### 7.1.4. Zátěž

Třetí scénář testuje dostupnost jednotlivých API. Pro test byl použit dotaz na Keystone API, který vrací list uživatelů, kteří jsou v daném prostředí vytvořeny. Toto testování

bylo v obou prostředích provedeno stejným způsobem. Pro tento typ testování bylo potřeba vytvořit validní token, pomocí kterého je možno odeslat dotaz na API. Token byl vytvořen pomocí příkazu *openstack token issue* na předem danou dobu. S tímto tokenem již bylo možné vytvořit dotaz proti API, dotaz byl simulován prostřednictvím příkazu *curl*. Tímto příkazem lze do dotazu uložit potřebný token, viz ukázka kódu 7.1. Požadavek je zobrazen na řádce 11. Pro lepší formátování vrácené odpovědi byla použita python utilita *-mjson.tool*, jenž slouží k parsování JSON odpovědi. Odpověď je poté přehlednější a lépe čitelná.

**Kód 7.1:** Token issue a dotaz na Keystone API, zdroj: vlastní tvorba

```

1 root@ctl01:~# openstack token issue
2 +-----+-----+
3 | Field | Value |
4 +-----+-----+
5 | expires | 2019-07-15T09:06:30+0000 |
6 | id | gAAAAABdLDQGHjWh4fEUcDKdnSNSiJiIzM |
7 | project_id | 79269c93c7444f9d9957748d98882bc8 |
8 | user_id | 5249ee2b007f4d569e72550ca0abc17b |
9 +-----+-----+
10
11 root@ctl01:~# curl -H "X-Auth-Token:
    gAAAAABdLDQGHjWh4fEUcDKdnSNSiJiIzM"
    "http://keystone-svc:5000/v3/users" | python -mjson.tool
12
13 root@ctl01:~# ab -n 100 -c 50 -H "X-Auth-Token:
    gAAAAABdLDQGHjWh4fEUcDKdnSNSiJiIzM"
    "http://keystone-svc:5000/v3/users"

```

Pro testování zátěže byl použit v obou prostředích jednořádkový příkaz z řádku 13, který využívá nástroj Apache Benchmark. Pomocí něho lze vygenerovat počet dotazů, které poté budou odeslány na server, zároveň lze specifikovat kolik dotazů má být zasláno současně. V obou prostředích byl tento test aplikován pro sto, tisíc a deset tisíc dotazů, z toho vždy jedna desetina dotazů byla odeslána současně. Ukázka výsledku z testovacího scénáře pro sto tisíc dotazů je zobrazena na kódu 7.2.

**Kód 7.2:** Ukázka benchmark testu pro sto tisíc dotazů, zdroj: vlastní tvorba

```

1 Concurrency Level: 1000
2 Time taken for tests: 37.549 seconds
3 Complete requests: 10000
4 Failed requests: 98
5 (Connect: 0, Receive: 0, Length: 98, Exceptions: 0)
6 Non-2xx responses: 98

```

```

7 Requests per second: 266.32 [#/sec] (mean)
8 Time per request: 3754.935 [ms] (mean)
9 Time per request: 3.755 [ms] (mean, across all concurrent
  requests)
10 Transfer rate: 908.98 [Kbytes/sec] received
11
12 Connection Times (ms)
13 min mean[+/-sd] median max
14 Connect: 0 3 10.7 0 51
15 Processing: 89 3556 1853.1 3297 18448
16 Waiting: 89 3556 1853.1 3296 18448
17 Total: 139 3560 1850.5 3297 18448

```

### 7.1.5. Škálování

Čtvrtý scénář měl prověřit škálování OpenStack control plane. Ve scénáři byly testovány možnosti škálování API jednotlivých OpenStack služeb. Jak již bylo zmíněno, celé MCP řešení je navrženo jako IaaS. Jeho hlavní částí je metadatový model, jenž popisuje, jak má prostředí vypadat. U původního prostředí bylo nutné při škálování model upravit a pro jednotlivá API OpenStack služeb spustit nový virtuální server zvaný ctl04. Ten bylo nutno nakonfigurovat a nainstalovat na něj nejen balíčky pro API OpenStack služeb, ale také programy zaručující vysokou dostupnost (HAproxy, Keepalived). Celá procedura není automatizovaná, a tak změny v modelu a jejich aplikování trvalo cca 20 minut.

Kontejnerizovaná OpenStack služba je do k8s prostředí nasazována jako k8s Deployment, viz část o vysoké dostupnosti. Škálování pak probíhá pomocí změny parametru replicas v jednotlivých CR manifestech. Počet replik z CR manifestu je předán přes operátor ReplicaSetu a ten vytvoří požadovaný počet replik. V k8s lze využít vlastnost autoscaling, to znamená, že se nastaví pravidla, při kterých může sám orchestrátor automaticky spustit další repliky služeb nezávisle na lidském zásahu do prostředí. Proces škálování v rámci kontejnerového prostředí proběhl v rámci několika vteřin.

### 7.1.6. Lifecycle management

V posledním testovacím scénáři byla testována aktualizace a upgrade jednotlivých řešení. V původním MCP prostředí byl tento test aplikován pomocí sady manuálních kroků. Aktualizace i upgrade byl testován na komponentě Keystone. Obě dvě procedury si jsou velmi podobné, jedná se vždy o instalaci debian balíčku, který obsahuje nejnovější verzi či opravenou chybu. Před instalací balíčku bylo nutné na testovacím prostředí vypnout službu HAproxy, aby tento krok aktualizace proběhl bez výpadku.

Požadavky na jednotlivé API pak nebyly směřovány na tento server. Balíček s novou verzí softwarů byl nainstalován ručně přes apt, po té byla daná služba restartována a znovu zapnuta HAproxy služba. Proces se opakoval pro všechny zbylé cti servery.

U kontejnerizované aplikace byly tyto aktualizací procesy prováděny odlišně. Pro aktualizaci Keystone komponenty je nutné mít předem připravený Docker Image, který bude nahrazovat stávající kontejner. Pro upgrade byl využit k8s rollout, což je funkcionality přímo v k8s, pomocí které lze jednoduše měnit verze kontejnerů. Orchestrátor si pak stáhne vyžadovanou verzi image a postupně nahrazuje kontejnery se starou verzí za kontejnery verzí novou. Při tomto procesu není potřeba řešit postupné load balancování, jak tomu bylo u předchozího řešení, vše je řízeno samotným orchestrátorem.

## 7.2. Porovnání

Z uvedených testů, které byly provedeny na základě funkčních požadavků lze odvodit, že instalace kontejnerové aplikace, která obsahovala manuální kroky, je mnohem rychlejší než instalace OpenStacku postaveného na virtuálních strojích. Nutno zmínit, že do tohoto testu nebyl započten čas strávený na přípravě v prostředí. V praxi pro použití testované instalační metody pro kontejnerové prostředí je třeba předpřipravit hardwarové servery včetně operačního systému a nakonfigurované sítě, dále je třeba nainstalovat orchestrátor. Doba konfigurace hardwaru a instalace podpůrných programů během nasazování MCP trvá servisnímu inženýrovi zhruba týden. Pokud je aplikace kontejnerizovaná a nasazena v k8s clusteru, časová náročnost nasazení OpenStacku je mnohem menší, jelikož se orchestrátor dokáže starat o síťovou logiku sám bez jakéhokoli lidského zásahu. V testu vysoké dostupnosti obě dvě řešení obstála. Obě fungují na podobném principu, kdy jednotlivé služby jsou připojeny pomocí jedné adresy, která směřuje dotazy mezi požadované služby. Při výpadku jedné služby, zůstává API stále aktivní, protože služba, na kterou jsou směřovány dotazy je spuštěna ve více replikách. U k8s je díky izolaci jednotlivých služeb možno kontejner po pádu znovu spustit. Tato skutečnost ušetří čas podpoře, jelikož při nedostupnosti služeb není nutné se přihlašovat do prostředí jako u současného řešení. K8s orchestrátor si dokáže částečně aplikaci udržovat sám. Ve třetím testu byla prověřována dostupnost API v obou prostředích. Tento test dopadl pro obě prostředí shodně. V případech testující desetitisíce dotazů se vždy některé dotazy ztratily nebo selhaly, jednalo se však o nižší desítky dotazů, což je v kontextu těch úspěšně přijatých zanedbatelné číslo. Ve škálovacím testu bylo ověřeno, že kontejnerové řešení nabízí oproti původnímu MCP větší možnosti škálování. V kontejnerovém prostředí lze služby škálovat nezávisle na sobě. K8s dokáže při správném nastavení škálovat služby automaticky, takže není potřeba asistence inženýra. Kontejnerové řešení dominuje především v rychlosti škálování jednotlivých služeb, které do-

káže spouštět v jednotkách vteřin, narozdíl od původního MCP, kde spuštění nového virtuálního serveru s API trvá až desítky minut. Poslední test se týkal lifecycle managementu, a to konkrétně upgradu a aktualizací. Testem prošlo lépe kontejnerizované řešení, které má aktualizací proces vyřešený pomocí samotného orchestrátoru. Proces je oproti současnému MCP plně automatizovaný a nevyžaduje žádné skripty či pipeline. Díky oddělené logice jednotlivých služeb do kontejnerů lze velmi jednoduše aktualizovat a upravovat pouze části OpenStacku. Například pokud je nutné otestovat kompatibilitu storage a cinderu, je velmi jednoduché vyupgradovat pouze cinder kontejnery. Pokud by OpenStack s touto novou částí nefungoval, lze jednoduše a téměř okamžitě spustit znovu staré kontejnery pomocí k8s rolloutu. Porovnání jednotlivých řešení je uvedeno v souhrnné tabulce číslo 7.1.

**Tabulka 7.1.:** Porovnání prostředí

	VM prostředí	Kontejnerové prostředí
Doba instalace:	85 min	15 min
Vysoká dostupnost:	Ano	Ano
Dostupnost API:	99,99%	99,99%
Škálování:	Manuální, v rámci desítek minut	Manuální i automatizované, v rámci vteřin
Lifecycle management:	Manuální	Automatizovaný

## 8. Závěr

Kontejnerizace je dnes jedno z velmi aktuálních témat, především pak ve spojení s cloudovými technologiemi. Je to hlavně z důvodu konceptu zapouzdření logiky do funkčních celků, které poskytují vývojářům a systémovým administrátorům větší flexibilitu. Kontejnerizované služby lze zároveň díky své izolaci velmi jednoduše a rychle nasazovat do produkce, což zkracuje dobu, během které jsou vývojáři schopni doručovat zákazníkům nové verze aplikací.

Cílem práce bylo ověření, zda je možné převést tradiční aplikaci postavenou nad plnou virtualizací do kontejnerového prostředí a otestovat, zda budou vyřešeny problémy, se kterými se potýkala předchozí verze aplikace. V práci byl představen koncept kontejnerové virtualizace, byly zde porovnány jednotlivé architektury aplikací a zároveň představeny technologie, které jsou s kontejnerizací spojeny. V další části diplomové práce byla analyzována aplikace a byly stanoveny funkční požadavky, ze kterých vznikla nová architektura aplikace. Ta byla prakticky naimplementována a otestována. Z výsledku testování lze usoudit, že tradiční aplikaci do kontejnerového prostředí je možné zmigrovat. Aby migrace do kontejnerového prostředí byla úspěšná, je třeba připravit aplikaci, kde je nutno udělat spoustu změn. Vzhledem k množství a složitosti úprav aplikace pak ne vždy bývá tento postup migrace do kontejnerového prostředí efektivní. Pro migraci je pak vhodné vybrat správný nástroj, který přesunutí do kontejnerového prostředí zjednoduší a tím ulehčí práci nejen vývojářům ale i operátorům.

Jako vhodný předmět pro další výzkum v rámci kontejnerizace je možné označit testování kontejnerizované aplikace na škále. Pomocí, kterého by se dalo zjistit, jak se chová v produkčním prostředí. Popřípadě se zamyslet nad tématem migrace více obecněji, zamyslet se nad univerzálním frameworkem, který by měl za cíl zjednodušit migraci tradičních aplikací do kontejnerového prostředí. Pomocí tohoto frameworku by pak mohl být značně ulehčen čas administrátorům a vývojářům, kteří musí aplikaci pro kontejnery optimalizovat.

# Literatura

- [1] MATTEO, Riondato. *FreeBSD Handbook Chapter 15 Jails*. *freebsd.org. The FreeBSD Project*. [online]. 2014 [cit. 2019-04-21] Dostupné z: [https://www.freebsd.org/doc/en\\_US.ISO8859-1/books/handbook/jails.html](https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html)
- [2] INDRASIRI, Kasun., P. SIRIWARDENA. *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress, 2018. ISBN 9781484238578.
- [3] *Linux Vserver Overview*. *Linux Vserver*. [online]. 2006 [cit. 2019-04-21]. Dostupné z: <http://linux-vserver.org/Overview>
- [4] BOTTOMLEY, James. *Solaris Internals wiki:Zones*. [online]. 2007 [cit. 2019-04-21]. Dostupné z: <https://solarisinternals.com/wiki/index.php/Zones>
- [5] *Openvz history*. *Openvz*. [online]. 2008 [cit. 2019-04-21]. Dostupné z: <https://wiki.openvz.org/History>
- [6] CORBET, Jonathan. *Notes from a container*. [online]. 2007 [cit. 2019-04-21]. Dostupné z: <https://lwn.net/Articles/256389/>
- [7] MARQUEZ, Ell. *The History of Container Technology*. [online]. 2018 [cit. 2019-04-21]. Dostupné z: <https://linuxacademy.com/blog/containers/history-of-container-technology/>
- [8] LMCTFY *GitHub repozitář*. *GitHub*. [online]. 2016 [cit. 2019-04-21]. Dostupné z: <https://github.com/google/lmctfy/commit/0b317d7eb625d1877a8a0aaf2f46f770d9a5a50f>
- [9] GLOUB, Ben. *DotCloud, Inc. is Becoming Docker, Inc.* [online]. 2013 [cit. 2019-04-21]. Dostupné z: <https://blog.docker.com/2013/10/dotcloud-is-becoming-docker-inc/>
- [10] POLVI, Alex. *Making Sense of Container Standards and Foundations: OCI, CNCF, appc and rkt*. [online]. 2015 [cit. 2019-04-23]. Dostupné z: <https://coreos.com/blog/making-sense-of-standards.html>



- 
- [11] GUO, Jianing. *Demystifying container vs VM-based security: Security in plaintext*. [online]. 2017 [cit. 2019-04-23]. Dostupné z: <https://cloud.google.com/blog/products/gcp/demystifying-container-vs-vm-based-security-security-in-plaintext>
- [12] ASAY, Matt. *Why Red Hat makes more money on Docker than Docker does*. [online]. 2016 [cit. 2019-04-23]. Dostupné z: <https://www.techrepublic.com/article/why-red-hat-makes-more-money-on-docker-than-docker-does/>
- [13] CARTER, Eric. *2018 Docker usage report*. [online]. 2018 [cit. 2019-05-02]. Dostupné z: <https://sysdig.com/blog/2018-docker-usage-report/>
- [14] SHIMEL, Alan. *ClusterHQ and DevOps.com survey show Containers poised for mass adoption*. [online]. 2015 [cit. 2019-05-02]. Dostupné z: <https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2015.pdf>
- [15] *Docker OverlayFS Dokumentace*. Docker. [online]. 2016 [cit. 2019-05-02]. Dostupné z: <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>
- [16] *Alpine Linux - official Image*. Docker Hub. [online]. 2019 [cit. 2019-05-05]. Dostupné z: [https://hub.docker.com/\\_/alpine](https://hub.docker.com/_/alpine)
- [17] WOLFF, Eberhard. *Microservices: Flexible Software Architectures*. [online]. 2016 [cit. 2019-05-03]. Dostupné z: <https://microservices.io/patterns/microservices.html>
- [18] RICHARDSON, Chris. *Microservice architecture pattern*. [online]. 2017 [cit. 2019-05-03]. Dostupné z: <https://microservices.io/patterns/microservices.html>
- [19] BEDA, Joe. *Containers At Scale*. [online]. 2017 [cit. 2019-05-05]. Dostupné z: <https://speakerdeck.com/jbeda/containers-at-scale>
- [20] *Deploy a stack to a swarm*. Docker Doc. [online]. 2019 [cit. 2019-05-05]. Dostupné z: <https://docs.docker.com/engine/swarm/stack-deploy/>
- [21] BEDA, Joe. *First commit*. Kubernetes GitHub. [online]. 2014 [cit. 2019-05-05]. Dostupné z: <https://github.com/kubernetes/kubernetes/commit/2c4b3a562ce34cddc3f8218a2c4d11c7310e6d56>
- [22] *Deployments - Kubernetes*. Kubernetes Docs. [online]. 2019 [cit. 2019-05-14]. Dostupné z: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

- 
- [23] *Marathon Docs. Mesos Sphere* [online]. [cit. 2019-05-05]. Dostupné z: <https://mesosphere.github.io/marathon/>
- [24] *Red Hat, Inc. Red Hat to Acquire CoreOS, Expanding its Kubernetes and Containers Leadership* [online]. 2018 [cit. 2019-05-05]. Dostupné z: <https://www.redhat.com/en/about/press-releases/red-hat-acquire-coreos-expanding-its-kubernetes-and-containers-leadership>
- [25] *BARNARD, Kaitlyn. CNCF Survey: Use of Cloud Native Technologies in Production Has Grown Over 200%*. [online]. 2018 [cit. 2019-05-22]. Dostupné z: <https://www.cncf.io/blog/2018/08/29/cncf-survey-use-of-cloud-native-technologies-in-production-has-grown-over-200-percent/>
- [26] *Declarative application management in Kubernetes. Kubernetes GitHub*. [online]. 2019 [cit. 2019-05-05]. Dostupné z: <https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/declarative-application-management.md>
- [27] *GUTHRIE, Scott. Microsoft to acquire Deis to help companies innovate with containers*. [online]. 2017 [cit. 2019-05-14]. Dostupné z: <https://blogs.microsoft.com/blog/2017/04/10/microsoft-acquire-deis-help-companies-innovate-containers/>
- [28] *Helm Nginx repository. Helm GitHub*. [online]. 2018 [cit. 2019-05-14]. Dostupné z: <https://github.com/helm/charts/tree/master/stable/nginx-ingress>
- [29] *Helm 3 Design Proposal. Helm GitHub*. [online]. 2018 [cit. 2019-05-14]. Dostupné z: <https://github.com/helm/community/blob/master/helm-v3/000-helm-v3.md>
- [30] *Kapitan templates for Kubernetes services. Richard Felkl GitHub*. [online]. 2019 [cit. 2019-05-20]. Dostupné z: <https://github.com/richardfelkl/kapitans-fleet>
- [31] *CLI Special Interest Group. Kubernetes GitHub*. [online]. 2019 [cit. 2019-05-14]. Dostupné z <https://github.com/kubernetes/community/tree/master/sig-cli>
- [32] *Kubernetes SIG CLI. Kubernetes GitHub*. [online]. 2019 [cit. 2019-05-14]. Dostupné z: <https://github.com/kubernetes/community/tree/master/sig-cli>

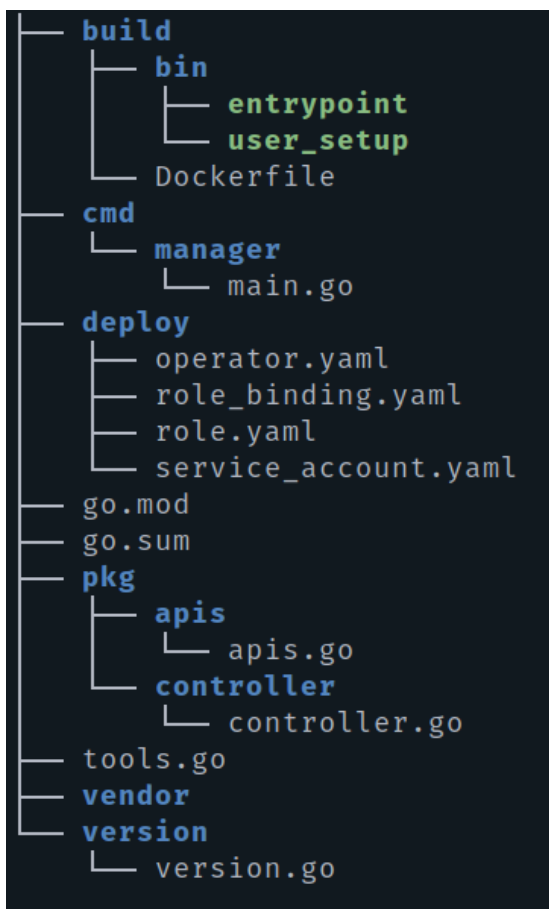
- [33] *Kapitan templates for Kuberne services. Kubernetes GitHub*. [online]. 2018 [cit. 2019-05-20]. Dostupné z: <https://github.com/kubernetes-sigs/kustomize/blob/master/examples/wordpress/kustomization.yaml>
- [34] ARJARAPU, Shyam. *Getting started with MongoDB Enterprise Operator for Kubernetes*. [online]. 2018 [cit. 2019-05-20]. Dostupné z: <https://hackernoon.com/getting-started-with-mongodb-enterprise-operator-for-kubernetes-bb5d5205fe02>
- [35] *Migrating to Azure: transforming legacy applications into scalable cloud-first solutions*. New York, NY: Springer Science+Business Media, 2018. ISBN 9781484235843.
- [36] WOOTTON, Benjamin. *What Are Legacy Systems and What Should You Do with Them?* [online]. 2017 [cit. 2019-05-24]. Dostupné z: <https://www.contino.io/insights/what-are-legacy-systems-and-what-should-you-do-with-them>
- [37] ORREGO S., MUNDY, E. Andres, Gregory. *A study of software reuse in NASA legacy systems*. [online]. 2007, ISSE. 3. 167-180. 10.1007/s11334-007-0027-y.[cit. 2019-5-5]
- [38] BECK, K., BEEDLE, M., van BENNEKUM, A., COCKBURN, A., CUNNINGHAM, W., FOWLER, M., Grenning, J., Highsmith, J., Hunt, A., JEFFRIES, R., KERN, J., MARRICK, B., MARTIN, R. C., MELLOR, S., SCHWABER, K., SUTHERLAND, J. a THOMAS, D. (2001). *Manifesto for Agile Software Development Manifesto for Agile Software Development*.
- [39] MALLADI, Srinivas. G. RAMAKRISHNA. a R RAJASEKHARA a E. Babu. *Analysis of Legacy System in Software Application Development: A Comparative Survey*. 2016. *International Journal of Electrical and Computer Engineering (IJECE)*. 6. 292-297. 10.11591/ijece.v6i1.8367.
- [40] FERNANDES, Thaisa. *Spotify Squad framework — Part I* [online]. 2017 [cit. 2019-05-24]. Dostupné z: <https://medium.com/productmanagement101/spotify-squad-framework-part-i-8f74bcfcd761>
- [41] KESSLER, David. *Automated Testing Strategy for Legacy Systems*. [online]. 2012 [cit. 2019-06-01]. Dostupné z: <https://www.sourceallies.com/2012/04/automated-testing-strategies-for-legacy-systems/>
- [42] LAMB, John. *Legacy systems continue to have a place in the enterprise*. [online]. 2008 [cit. 2019-06-01]. Dostupné z: <https://www.computerweekly.com/feature/Legacy-systems-continue-to-have-a-place-in-the-enterprise>

- [43] AZERI, Izzy. *What Is CI/CD?* [online]. 2017 [cit. 2019-06-02]. Dostupné z: <https://dzone.com/articles/what-is-cicd>
- [44] RYAN, Paul. *NASA and Rackspace part the clouds with open source project.* [online]. 2010 [cit. 2019-06-10]. Dostupné z: <https://arstechnica.com/information-technology/2010/07/nasa-and-rackspace-part-the-clouds-with-open-source-project/>
- [45] *Mirantis/pipeline-library* repozitář. *GitHub*. [online]. 2017 [cit. 2019-06-10]. Dostupné z: <https://github.com/Mirantis/pipeline-library/>
- [46] *Mirantis/mk-pipeline* repozitář. *GitHub*. [online]. 2017 [cit. 2019-06-10]. Dostupné z: <https://github.com/Mirantis/mk-pipelines>
- [47] *Mirantis Dokumentace: OpenStack compact cloud architecture.* *Mirantis Doc*. [online]. 2018 [cit. 2019-06-10]. Dostupné z: <https://docs.mirantis.com/mcp/q4-18/mcp-standard-configuration/os-scaling/os-compact-cloud-arch.html>
- [48] *MCP Gerrit Review page.* *Mirantis Gerrit*. [online]. 2017 [cit. 2019-06-11]. Dostupné z: <https://gerrit.mcp.mirantis.com/>
- [49] *Mirantis Dokumentace: Deploy salt master node.* *Mirantis Docs team*. [online]. 2019 [cit. 2019-06-11]. Dostupné z: <https://docs.mirantis.com/mcp/q4-18/mcp-deployment-guide/deploy-mcp-drivetrain/deploy-salt-master-node.html>
- [50] *MCP Reclass system model.* *Mirantis GitHub*. [online]. 2016 [cit. 2019-06-12]. Dostupné z: <https://github.com/Mirantis/reclass-system-salt-model>
- [51] *Cookiecutter Repository.* *Cookiecutter GitHub*. [online]. 2016 [cit. 2019-06-12]. Dostupné z: <https://github.com/audreyr/cookiecutter>
- [52] *Kolla dokumentace.* *OpenStack*. [online]. 2016 [cit. 2019-06-24]. Dostupné z: <https://docs.openstack.org/kolla-ansible/latest/>
- [53] *Whitepaper - A New Open Infrastructure Project for OpenStack.* *AT&T Cloud Platform Development team*. [online]. 2018 [cit. 2019-06-24]. Dostupné z: [https://about.att.com/ecms/dam/inside\\_connections\\_blogdocs/Whitepaper%20-%20Airship%20a%20New%20Open%20Infrastructure%20Project%20for%20OpenStack%20v1.0.pdf](https://about.att.com/ecms/dam/inside_connections_blogdocs/Whitepaper%20-%20Airship%20a%20New%20Open%20Infrastructure%20Project%20for%20OpenStack%20v1.0.pdf)

- 
- [54] HECK, Joseph. *Kubernetes for Developers*. Packt Pub, 2018. ISBN 9781788834759.
- [55] BELOVA, Dina, A. SHAPOSHNIKOV a M. SIMONIN. *Chasing 1000 nodes scale*. [online]. 2016 [cit. 2019-06-24]. Dostupné z: <https://www.openstack.org/assets/presentation-media/Chasing-1000-nodes-scale.pdf>
- [56] *OpenStack Bootstrapping Identity*. OpenStack. [online]. 2019 [cit. 2019-07-04]. Dostupné z: <https://docs.openstack.org/keystone/queens/admin/identity-bootstrap.html>
- [57] *Getting started with the Operator SDK*. OKD docs. [online]. 2019 [cit. 2019-07-05]. Dostupné z: <https://docs.okd.io/latest/operators/osdk-getting-started.html>
- [58] *LotharKAtt/openstack-images repozitář*. GitHub. [online]. 2019 [cit. 2019-07-09]. Dostupné z: <https://github.com/LotharKAtt/openstack-images>
- [59] *Docker Create a base image*. Docker docs. [online]. 2019 [cit. 2019-07-10]. Dostupné z: <https://docs.docker.com/develop/develop-images/baseimages/>
- [60] *MCP Release Support Status*. Mirantis. [online]. 2019 [cit. 2019-07-10]. Dostupné z: <https://www.mirantis.com/software/openstack/releases/>
- [61] *OpenStack Rocky Series Release Notes*. OpenStack. [online]. 2019 [cit. 2019-07-11]. Dostupné z: <https://docs.openstack.org/releasenotes/puppet-keystone/rocky.html>
- [62] *LotharKAtt/openstack-operátor repozitář*. GitHub. [online]. 2019 [cit. 2019-07-11]. Dostupné z: <https://github.com/LotharKAtt/openstack-operator>
- [63] SAYFAN, Gigi. *Mastering Kubernetes*. Packt Pub, 2017. ISBN 9781786461001.
- [64] *Gophercloud package dokumentace*. Golang Docs. [online]. 2019 [cit. 2019-07-15]. Dostupné z: <https://godoc.org/github.com/gophercloud/gophercloud>

## 9. Přílohy

### A. Stuktura Operátor SDK projektu



---

## B. Příkazy pro konfiguraci MySQL a RabbitMQ

```
// MySQL
CREATE DATABASE cidner;
CREATE DATABASE glance;
CREATE DATABASE keystone;
CREATE DATABASE neutron;
CREATE DATABASE nova;
CREATE DATABASE nova_api;
CREATE DATABASE nova;
CREATE DATABASE nova_cell0;

GRANT ALL PRIVILEGES ON *.* TO 'nova' IDENTIFIED BY 'cloudlab';
GRANT ALL PRIVILEGES ON *.* TO 'nova_cell0' IDENTIFIED BY 'cloudlab';
GRANT ALL PRIVILEGES ON *.* TO 'nova_api' IDENTIFIED BY 'cloudlab';
GRANT ALL PRIVILEGES ON *.* TO 'cinder' IDENTIFIED BY 'cloudlab';
GRANT ALL PRIVILEGES ON *.* TO 'glance' IDENTIFIED BY 'cloudlab';
GRANT ALL PRIVILEGES ON *.* TO 'keystone' IDENTIFIED BY 'cloudlab';
GRANT ALL PRIVILEGES ON *.* TO 'neutron' IDENTIFIED BY 'cloudlab';

//RabbitMQ
rabbitmqctl add_user openstack cloudlab
rabbitmqctl set_user_tags openstack administrator
rabbitmqctl set_permissions -p / openstack ".*" ".*" ".*"
```

## A. Cizí výrazy

- **AAA (Authentication, Authorization, Accounting protocol)** - Sada bezpečnostní protokolů sloužící k autentizaci a autorizaci.
- **Bug** - Chyba v programu
- **CaaS (Container as a Service)** - CaaS je služba, která pomáhá vývojářům a operátorům vytvářet a ovládat kontejnery.
- **CLI (Command Line Interface)** - Textové uživatelské rozhraní
- **Cloud-init** - Program, který se používá u cloudových serverů, slouží ke spuštění nadefinovaných akcí hned po spuštění instance. Bývá používán k inicializačnímu nastavení instance.
- **Cluster** - Seskupení skupiny serverů, které spolu spolupracují a chovají se jako jeden server.
- **cgroups, namespaces** - Zdroje zabudované v Linuxovém jádře, pomocí kterých je zajištěna izolace kontejnerů. Cgroups slouží k izolaci CPU, paměti a diskových zdrojů, namespaces pak slouží k izolaci procesů.
- **Commit** - Základní stavební jednotka verzovací systému git. Commit udržuje záznamy změn v zdrojových kódech aplikace.
- **CRM (Customer relationship management)** - Software sloužící ke komunikaci mezi firmou a zákazníky např. ServiceNow, Salesforce.
- **Development** - Vývojářský tým
- **Downstream repozitář** - Downstream je kopie upstream repozitáře, který obsahuje změny a nové vlastnosti, které nejsou v upstreamu dostupné. Downstream tvoří společnosti, které mají potřebu do projektu dodat novou funkcionalitu či opravnou záplatu a nechtějí čekat na komunitní procesy. Do downstream větví mohou přispívat pouze kontributoři, kteří mají do repozitáře přístup.
- **Git** - Git je nástroj na správu verzí zdrojových kódů aplikace.



- 
- **GitHub** - Github je služba pro sdílení zdrojových kódů aplikací vezovaných gitem. Většina open source projektů je hostována na serveru GitHub.com
  - **Hypervisor** - Server, který slouží jako host pro virtuální stroje.
  - **IDM (Identity Management)** - IDM je služba, zajišťující správu identity. Jedná se o služby, které spravují uživatele a jejich oprávnění např. FreeIPA, AD.
  - **Linuxové jádro (kernel)** - Monolitické jádro, které ovládá a spravuje hardwarové zdroje na linuxových distribucích.
  - **Open source** - Otevřený zdrojový kód, tento kód je dostupný na internetu pod otevřenou licencí např. Apache2, GNU GPL.
  - **Operations** - Tým zodpovědný za provoz infrastruktury a správu aplikací.
  - **PagerDuty** - Platforma sloužící ke správě notifikací z infrastruktury.
  - **Private cloud** - Private cloud je určený jen pro určitou skupinu uživatelů a dostupný pouze z jejich sítě. Je hostován a provozován na vlastním hardwaru. Nejpopulárnější technologie pro private cloud jsou Microsoft Azure Pack a OpenStack.
  - **Public cloud** - Public cloud neboli veřejný cloud je poskytovatel, který nabízí pronájem virtuálních zdrojů prostřednictvím internetu. Největšími poskytovateli public cloudu jsou firmy Amazon (AWS), Google (GCE) a Microsoft (Azure).
  - **QA (Quality assurance)** - Sada procesů sloužící k testování a udržování aplikace.
  - **REST API (Representational State Transfer)** - REST je architektura komunikačního rozhraní, pomocí kterého lze pomocí HTTP volání provádět CRUD akce.
  - **Runtime** - Runtime je sada systémových prostředků a knihoven, které slouží pro spouštění aplikací, např. JRE (Java Runtime Environment)
  - **Slack** - Komunikační nástroj sloužící k posílání zpráv, volání a sdílení souborů.
  - **Upstream repozitář** - Upstream repozitář, obsahuje zdrojové kódy projektu, které jsou veřejně přístupné. Vývoj v upstream repozitáři je řízen buď majitelem repozitáře nebo komunitou okolo projektu.
  - **VM (Virtual Machine)** - Virtuální stroj je produkt plné virtualizace. Pomocí emulovaného hardwaru je vytvořen virtualizovaný hardware, na kterém je pak možné spustit operační systém.
  - **Vendor** - Společnost, která prodává či poskytuje podporu pro danou technologii.

## Seznam obrázků

2.1. Schéma mikroslužeb, zdroj: vlastní tvorba . . . . .	10
2.2. CNCF průzkum využití orchestrátorů, zdroj: [25] . . . . .	17
2.3. Schéma MongoDB Enterprise operátoru, zdroj:[34] . . . . .	24
3.1. Schéma monolitické architektury, zdroj: vlastní tvorba . . . . .	27
4.1. MCP architektura s OVS networkingem, zdroj: [47] . . . . .	37
5.1. Návrh kontejnerové architektury, zdroj: vlastní tvorba . . . . .	51
6.1. Schéma komunikace v rámci Keystone služby, zdroj: vlastní tvorba . . . .	54
6.2. Schéma komunikace v rámci k8s operátoru, zdroj: vlastní tvorba . . . . .	55
6.3. Spuštěná služba Keystone v k8s, zdroj: vlastní tvorba . . . . .	65

# Seznam tabulek

2.1. Porovnání kontejneru a VM . . . . .	7
3.1. Porovnání monolitické architektury a mikroslužeb . . . . .	28
7.1. Porovnání prostředí . . . . .	72

## Seznam ukázek kódu

2.1. Dockerfile, zdroj: vlastní tvorba . . . . .	8
2.2. Docker Compose, zdroj: vlastní . . . . .	12
2.3. Kubernetes Manifest - Nginx deployment, zdroj: [22] . . . . .	15
2.4. Kubernetes Manifest - Grafana service, zdroj vlastní . . . . .	19
2.5. Kapitan komponenta - využívající Jsonnet, zdroj: [30] . . . . .	20
2.6. Kustomization.yml soubor, zdroj: [33] . . . . .	22
4.1. Metadata model pro Galera databázi, zdroj: vlastní tvorba . . . . .	39
6.1. Základní Dockerfile pro Keystone image, zdroj: vlastní tvorba . . . . .	57
6.2. Entrypoint pro Keystone image, zdroj: vlastní tvorba . . . . .	58
6.3. Ukázka části Keystone CR, zdroj: vlastní tvorba . . . . .	59
6.4. Ukázka struktury KeystoneSpec, zdroj: vlastní tvorba . . . . .	60
6.5. Ukázka funkce createDeploymentAPI(), zdroj: vlastní tvorba . . . . .	61
6.6. Ukázka funkce pro generování ConfigMap, zdroj: vlastní tvorba . . . . .	64
6.7. Ukázka tvorby Keystone uživatelů, zdroj: vlastní tvorba . . . . .	65
7.1. Token issue a dotaz na Keystone API, zdroj: vlastní tvorba . . . . .	69
7.2. Ukázka benchmark testu pro sto tisíc dotazů, zdroj: vlastní tvorba . . . . .	69

**Podklad pro zadání DIPLOMOVÉ práce studenta**

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Čížinský Pavel	Ruská 273, Jaroměř - Josefov	I1700483

**TÉMA ČESKY:**

Migrace legacy aplikace do kontejnerové virtualizace

**TÉMA ANGLICKY:**

Migration of legacy application to container environment

**VEDOUcí PRÁCE:**

Ing. Vladimír Soběslav, Ph.D. - KIT

**ZÁSADY PRO VYPRACOVÁNÍ:**

Osnova:

1. Úvod
2. Kontejnerizace
3. Vymezení legacy aplikace
4. Analýza migrované aplikace
5. Návrh budoucí architektury kontejnerové aplikace
6. Implementace kontejnerového řešení
7. Porovnání klasického a kontejnerového řešení
8. Závěr

**SEZNAM DOPORUČENÉ LITERATURY:**

Migrating to Azure: transforming legacy applications into scalable cloud-first solutions. New York, NY: Springer Science+Business Media, 2018. ISBN 9781484235843.

HIGHTOWER, Kelsey, Brendan BURNS a Joe BEDA. Kubernetes: up and running: dive into the future of infrastructure. Sebastopol, CA: O'Reilly Media, 2017. ISBN 9781491935675.

FONSECA, Nelson L. S. da. a Raouf BOUTABA. Cloud services, networking, and management. Hoboken, New Jersey: Wiley, 2015. ISBN 9781118845943.

SUBRAMANIAN Sriram, Sreenivas VORUGANTII. Software-Defined Networking (SDN) with OpenStack.~ Birmingham, United Kingdom: Packt Publishing Limited, 2016. ISBN 9781786465993

Podpis studenta:

  
.....

Datum: 18.2.2019

Podpis vedoucího práce:

  
.....

Datum: 28.2.2019