ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

FAKULTA
BIOMEDICÍNSKÉHO
INŽENÝRSTVÍ

DIPLOMOVÁ PRÁCE

2021

BC. TOMÁŠ
KRAJČA

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA BIOMEDICÍNSKÉHO INŽENÝRSTVÍ

Katedra biomedicínské informatiky

Architektura systému pro sběr dat a detekci anomálií ze senzorů

The architecture of the system for sensors data collection and anomaly detection

Diplomová práce

Studijní program:	Biomedicínská a klinická technika
Studijní obor:	Biomedicínská informatika

Autor bakalářské práce:	Bc. Tomáš Krajča
Vedoucí bakalářské práce:	Mgr. Radim Krupička, Ph.D.
Konzultant:	Ing. Miroslav Uller

Kladno 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Krajča**	Jméno:	**Tomáš**	Osobní číslo:	**465559**
Fakulta:	**Fakulta biomedicínského inženýrství**				
Garantující katedra:	**Katedra biomedicínské informatiky**				
Studijní program:	**Biomedicínská a klinická informatika**				

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Architektura systému pro sběr dat a detekci anomálií ze senzorů

Název diplomové práce anglicky:

The architecture of the system for sensors data collection and anomaly detection

Pokyny pro vypracování:

Cílem diplomové práce je vytvořit a implementovat architekturu pro sběr dat ze senzorů a predikci anomálií. Vstupem systému jsou časové řady ze senzorů a konfigurace prediktoru. Tyto informace jsou předány prediktoru, který za využití umělé inteligence a HW náročných algoritmů upozorňuje na anomálie. Systém navrhněte tak, aby bylo možné modulárně měnit způsob komunikace s prediktorem (webové API, streamy, databáze), bylo možné provozovat celý systém v cloudu, a aby bylo možné systém řídit přes webové API a přiřazovat mu serverové prostředky. Predikční software by měl být nezávislý na způsobu řízení a způsobu komunikace, navíc by měl umožňovat snadné ladění a implementaci prediktoru. V rámci diplomové práce vytvořte funkční demoverzi systému.

Seznam doporučené literatury:

[1] Christina Terese Joseph, K. Chandrasekaran, Straddling the crevasse: A review of microservice software architecture foundations and recent advancements, Software: Practice and Experience, ročník 49, číslo 10, 2019
[2] Robert C. Martin, Clean Architecture - A Craftsman's Guide to Software Structure and Design, ed. 1, Pearson, 2018, ISBN 978-0-13-449416-6

Jméno a příjmení vedoucí(ho) diplomové práce:

Mgr. Radim Krupička, Ph.D.

Jméno a příjmení konzultanta(ky) diplomové práce:

Ing. Miroslav Uller

Datum zadání diplomové práce: **15.02.2021**
Platnost zadání diplomové práce: **18.09.2022**

doc. Ing. Zoltán Szabó Ph.D.	prof. MUDr. Jozef Rosina, Ph.D., MBA
podpis vedoucí(ho) katedry	podpis děkana(ky)

PROHLÁŠENÍ

Prohlašuji, že jsem bakalářskou práci s názvem Architektura systému pro sběr dat a detekci anomálií ze senzorů vypracoval samostatně a použil k tomu úplný výčet citací použitých pramenů, které uvádím v seznamu přiloženém k diplomové práci.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů.

V Kladně 13.5.2021 …..……..……..

Bc. Tomáš Krajča

PODĚKOVÁNÍ

Rád bych tímto poděkoval vedoucímu práce Mgr. Radimu Krupičkovi, Ph.D. za cenné rady, připomínky a trpělivost v průběhu vytváření práce. Dále bych rád poděkoval konzultantovi práce Ing. Miroslavu Ullerovi za cenné rady a za rozšíření obzorů v oblasti implementace softwaru. A v neposlední řadě bych rád poděkoval mé rodině a přítelkyni, kteří mě při studiu podporují.

ABSTRAKT

Diplomová práce se zabývá problematikou návrhu a vývoje softwaru jak z teoretického, tak i z praktického hlediska, jehož součástí je návrh a implementace demoverze architektury systému vhodného pro sběr dat v reálném čase a predikci anomálií. Cílem teoretické části je objasnění základní terminologie a vybraných metodik používaných při vývoji softwaru, věnuje se životnímu cyklu vývoje softwarového produktu, současným softwarovým architekturám a soustředí se na kontejnerizaci aplikací včetně nástrojů pro orchestraci. Praktická část obsahuje návrh, implementaci a dokumentaci architektury systému, která vychází z požadavků práce, je v konceptu mikroslužbové architektury, realizuje kontejnerizaci a orchestraci prostřednictvím platformy Docker.

Klíčová slova

Docker, Docker swarm, softwarová architektura, kontejnerizace a kontejnery, Kubernetes, InfluxDB Tick Stack

ABSTRACT

The main goal of this master theses is to clarify issues during software development both in theoretical way and in practical way which include implementation of system architecture, which is eligible for real time data streaming and predicting anomalies. Main goal of theoretical work is to describe general terminology and chosen methods both ordinally used in software development. It contains general information of software development life cycle, software architectures and it is focused on containerization with orchestration of software products. Practical part of this theses contains design, implementation and documentation of system architecture which is based on defined requirements and it is in concepts of microservices with use of container and orchestration tools from Docker ecosystem.

Keywords

Docker, Docker swarm, Software Architecture, Containers, Kubernetes, InfluxDB Tick Stack

Obsah

Seznam symbolů a zkratek

Seznam zkratek

Zkratka	Význam
VM	Virtuální počítač (*Virtual Machine*)
OS	Operační systém *(Operating system)*
HW	Hardware, veškeré fyzické komponenty zařízení
SW	Software, veškeré programové vybavení zařízení
API	Aplikační rozhraní *(Application Programming Interface)*
SOA	Servisně orientovaná architektura *(Service Oriented Architecture)*
UML	Jednotný jazyk pro tvorbu diagrmů *(Unified Modelint Language)*
MB	Mega bajt, jednotka velikosti *(Mega Byte)*

1 Úvod

Mezi hlavní cíle této práce patří objasnění metodik a vybraných softwarových technologií používaných při návrhu a implementaci softwarových produktů, a praktický vývoj demoverze architektury systému vhodného pro zpracování dát v reálném čase a predikci anomálií. Práce se skládá z rešerše informačních zdrojů a z praktické implementace softwarového řešení, která vychází ze zadaných požadavků. Pro teoretickou část byly vybrány zajímavé oblasti z návrhu a vývoje softwaru, softwarových architektur a kontejnerizace. Součástí práce je vysvětlení životního cyklu softwarového řešení, aktuálních používaných metodik a konkrétní technologická řešení pro kontejnerizaci a orchestraci aplikací včetně samotných definic. V rámci praktické části je cílem vytvořit funkční demoverzi architektury systému vhodného pro sběr dat v reálném čase a predikci anomálií. Zároveň má systém podporovat správu prostřednictvím grafického rozhraní, být konfigurovatelný a umožnit vizualizaci dat. Praktická část mimo samotné výsledné implementace architektury systému, bude obsahovat popis procesu návrhu a bude dokumentovat jak samotnou implementaci a její zajímavá místa, tak i způsoby nasazení a základní práci se systémem.

2 Přehled současného stavu

Tato kapitola je věnována vymezení základních pojmů, které se v práci hojně vyskytují, a konceptům v oblasti softwarového inženýrství, s důrazem na softwarové architektury a kontejnerizaci aplikací včetně orchestrace. Pro kapitoly jsou vybrány koncepty a technologie, které jsou použity v návrhu a implementaci demoverze architektury systému podle zadání práce.

2.1 Vymezení pojmů

Informační systém

Pro informační systém existuje mnoho obecných definic, které se navzájem mírně liší. Ale shodují se v definici, že se jedná o softwarový produkt, který přijímá vstupní data, zpracovává je prostřednictvím procesů a následně transformuje na data výstupní. V běžném hovoru se termín používá pro pojmenování podnikových aplikací, které splňují již zmíněnou definici. V akademické sféře je informační systém možné definovat jako soubor hardwarových a softwarových prostředků, které přijímají a poskytují data uživatelům informačního systému. [1]

Metodika

Metodika, často užívané jako metoda, je označení pro komplexní pracovní postup. V kontextu softwarového vývoje se jedná o pracovní postup používaný pro vývoj softwarových produktů. [2]

Metodologie

Metodologie je vědní disciplína, která se zabývá tvorbou, aplikací a porovnáváním metodik. Definuje pravidla a postupy pro návrh, plánování a řízení vývoje softwaru. [3]

Softwarová komponenta

Za komponentu se v oblasti softwarového vývoje považuje funkční programový modul, který je samostatný a rozlišitelný. Konkrétní představu o komponentách lze nalézt v praktické části této práce.

Softwarový produkt

Softwarový produkt, nebo softwarové řešení je názvem pro finální podobu vyvíjeného softwarového systému.

Softwarová technologie

Je obecný pojem označující konkrétní vývojové metody, programovací jazyk nebo nástroje, které se používají při vývoji softwaru. Příkladem softwarové technologie je .NET 5.

Služba *(Service)*

Služba je v informatice název pro proces, který vykonává danou množinu funkcí, obvykle však pouze jednu. Běží na pozadí, běžně nemá žádné uživatelské rozhraní a je volána prostřednictvím aplikačního rozhraní.

2.2 Návrh a vývoj software

Tato kapitola se věnuje metodám a postupům, které se používají při návrhu a vývoji softwarových produktů. Jsou zde popsány důvody vzniku disciplíny softwarové inženýrství a základní koncepty pro návrh a vývoj software. V počátcích vývoje softwaru byl člověk limitován výpočetními prostředky, v současném světě je už člověk limitován lidskými ambicemi a očekáváním. Právě člověk je zde tím stěžejním bodem, který je zodpovědný za kvalitu výsledného softwarového produktu. Aby bylo možné dosáhnout co největší kvality, tak je potřeba vyvíjet software podle osvědčených metodik.

Softwarové inženýrství je informatická disciplína, která vznikla ve snaze reagovat na nedostatky při návrhu a implementaci softwarových produktů. Zahrnuje v sobě prvky z oblasti managementu, inženýrství a informatiky. Při vývoji a následné údržbě informačních systémů dochází časem k potřebě aktualizací, mezi aktualizace pro uvedení příkladu může patřit nová funkcionalita nebo oprava chyb. Při takovém zásahu může docházet k nepředvídatelným reakcím programu, nová funkcionalita může ovlivnit již existující součásti systému a způsobit nestabilitu systému. Pro předcházení těchto nežádoucích problémů se vyvinula disciplína softwarové inženýrství, která definuje koncepty pro přístup při vývoji a následné udržitelnosti informačních systémů nebo aplikací. Zabývá se analýzou požadavků klienta, návrhem, vývojem a testováním softwarových aplikací.

2.2.1 Životní cyklus softwarového produktu

Životní cyklus vývoje softwaru je proces, jenž lze popsat jako množinu nezbytných fází směřujících k úspěšnému a efektivnímu návrhu, vývoji, nasazení a provozu softwarového produktu. Mezi fáze životního cyklu patří zadání a analýza požadavků, návrh a implementace, verifikace, provoz a údržba. V rámci této kapitoly jsou vysvětleny fáze životního cyklu software a také modely, které definují konkrétní posloupnost realizací jednotlivých fází cyklu.

Zadání a analýza požadavků

Každému vlastnímu vývoji softwaru musí předcházet analýza požadavků, ta se vytváří na základě rozhovoru se zákazníkem. Při rozhovorech se musí zohlednit případná neznalost zákazníka v oblasti informatiky, je proto důležité zvolit správnou metodiku rozhovoru pro získání nejrelevantnějších dat. Po získání požadavků následuje odborná analýza analytikem, který na základě vyhodnocení analýzy rozhodne další postup. Dalším postupem v tomto kontextu může být opakovaný rozhovor s klientem pro vyladění nejasností nebo předání finálních specifikací vývojářskému týmu. Na základě sesbíraných požadavků se obvykle definují grafická znázornění požadavků, mezi které se řadí strukturní diagramy, diagramy chování a diagramy interakce. Sbírané požadavky se dělí do dvou kategorií na funkční a nefunkční požadavky.

- **Funkční požadavky**

Funkčními požadavky jsou konkrétní funkcionality systému, v etapě návrhu klient definuje, co musí systém umět. Jedná se o nejkritičtější kategorii požadavků, analytik musí zadání klienta analyzovat a v případě neúplnosti či rozporů v požadavcích navrhnout lepší řešení. V této fázi je kladen velký důraz na komunikaci se zákazníkem pro zajištění kvalitního řešení. Výsledkem je celkový popis funkčností aplikace, který ale nezahrnuje použití konkrétních technologií nebo detailní popis architektury systému.

- **Nefunkční požadavky**

Nefunkční požadavky jsou definovány jako omezení systému nebo jiné okolnosti požadovaného řešení. Běžně nepatří do předepsané metodiky pro prvotní analýzu požadavků, berou se v úvahu až v etapě návrhu. Mezi nefunkční požadavky omezující systém patří požadavky na specifické technologie, uživatelské prostředí nebo efektivitu a paměťovou náročnost řešení. Jinými okolnostmi řešení, v kontextu nefunkčních

požadavků, mohou být cenová omezení, časová omezení nebo případně i omezení daná legislativou. [4]

- **Diagramy**

Pro grafické znázornění požadavků se vytvářejí diagramy, jejichž cílem je zachytit a specifikovat funkčnosti softwarových produktů do vizuální podoby. Diagramy se zpravidla vytvářejí prostřednictvím jazyka UML *(Unified modeling language)*, který definuje standardní způsob tvorby diagramů. V rámci UML není jednotná metodika, podle které by se měly vytvářet analýzy, ale je prostředkem pro grafické znázornění analýzy. Existuje řada diagramů popisující objekty a vztahy, v praxi se klade důraz pouze na několik stěžejních.

Návrh a implementace

Na základě analýzy požadavků vzniká návrh ve formě dokumentu, který obsahuje jak detailní informace vztahující se ke konkrétní implementaci produktu, tak informace o ceně, časovém harmonogramu a stranách, které uzavírají smluvní kontrakt.

Po podepsání smlouvy se zahájí vlastní implementace produktu, která vychází ze specifikací návrhového dokumentu.

Verifikace

V této fázi dochází k testování softwarového produktu. Testují se všechny reakce produktu na vstupní data. A kontroluje se, zdali jsou dosaženy všechny požadované cíle, které byly vytyčené při návrhu a specifikaci produktu. Zároveň další úlohou této fáze je odhalit a opravit případné nedostatky produktu. Testování se provádí v lokálním testovacím prostředí, protože testování v produkčním prostředí po nasazení by bylo nežádoucí a mohlo vést k řadě problémů.

Provoz a údržba

Po úspěšné verifikaci softwarového produktu následuje nasazení produktu do produkčního prostředí. Nasazením je myšlena instalace potřebných technologií, instalace samotného produktu a případné propojení s ostatními existujícími komponentami. Jedná se tedy o plné zprovoznění softwarového produktu. Po nasazení je nutné poskytnout dokumentaci a školení uživatelům produktu. Následuje zkušební provoz, kdy je poskytovatel produktu připraven poskytnout podporu a opravy v případě problémů.

Závěrečnou fází je běžný provoz a údržba produktu. Do údržby se řadí aplikování nových nebo úprava existujících funkcionalit, které by ovšem neznamenaly změnu celého produktu. V případě potřeby rozsáhlých změn je nutné se vrátit opět k první fázi životního cyklu vývoje softwarového produktu.

2.2.2 Modely životního cyklu software

Cílem modelů životního cyklu je přinést řád do vývoje softwarových produktů.

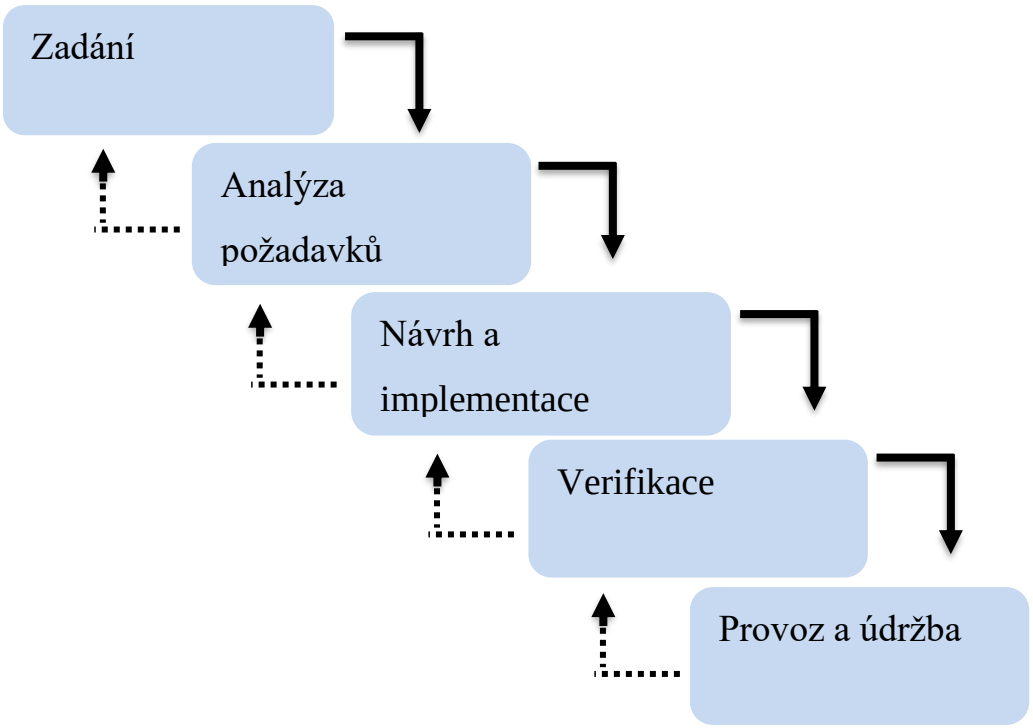
Vodopádový model

Vodopádový model je jedním z nejstarších modelů, člení softwarový proces na jednotlivé fáze, které na sebe sekvenčně navazují. Název pro tento model vychází právě ze sekvenční návaznosti, kdy se po kompletním ukončení jedné fáze životního cyklu přechází plynule na další.

Obecně je vodopádový model považován za velmi jednoduchý a užívá se obvykle pro menší projekty, které nekladou důraz na pozdější změny funkcionalit. Aktuálně existují modifikace vodopádového modelu, které řeší nedostatky původního vodopádového modelu, například umožňují návrat zpět právě o jednu fázi cyklu, pokud je to nutné. Přestože u hodnocení vodopádového modelu vůči aktuálním novým metodám převažují nevýhody, stále je vhodnější jej použít než nenásledovat žádný z modelů.

Mezi výhody užití vodopádového modelu patří především jeho jednoduchost, je vhodný pro menší projekty nebo u projektů, kde je přesně známý problém a jsou přesně specifikovány možnosti jeho řešení.

Jednou z nevýhod modelu je doba nutná pro vznik prvních verzí funkčního produktu. Tyto verze vznikají až po implementaci celého produktu, klient tak nemá možnost reagovat na stav objednávky v průběhu vývoje a ani po jeho ukončení, protože by to vedlo k návratu do první fáze cyklu. Nejkritičtější fází je v tomto modelu zadání a analýza požadavků, pokud v této fázi nebyly funkcionality dostatečně specifikované, tak jejich pozdější úprava nebo přidání povede k celkovému zvýšení nákladů.



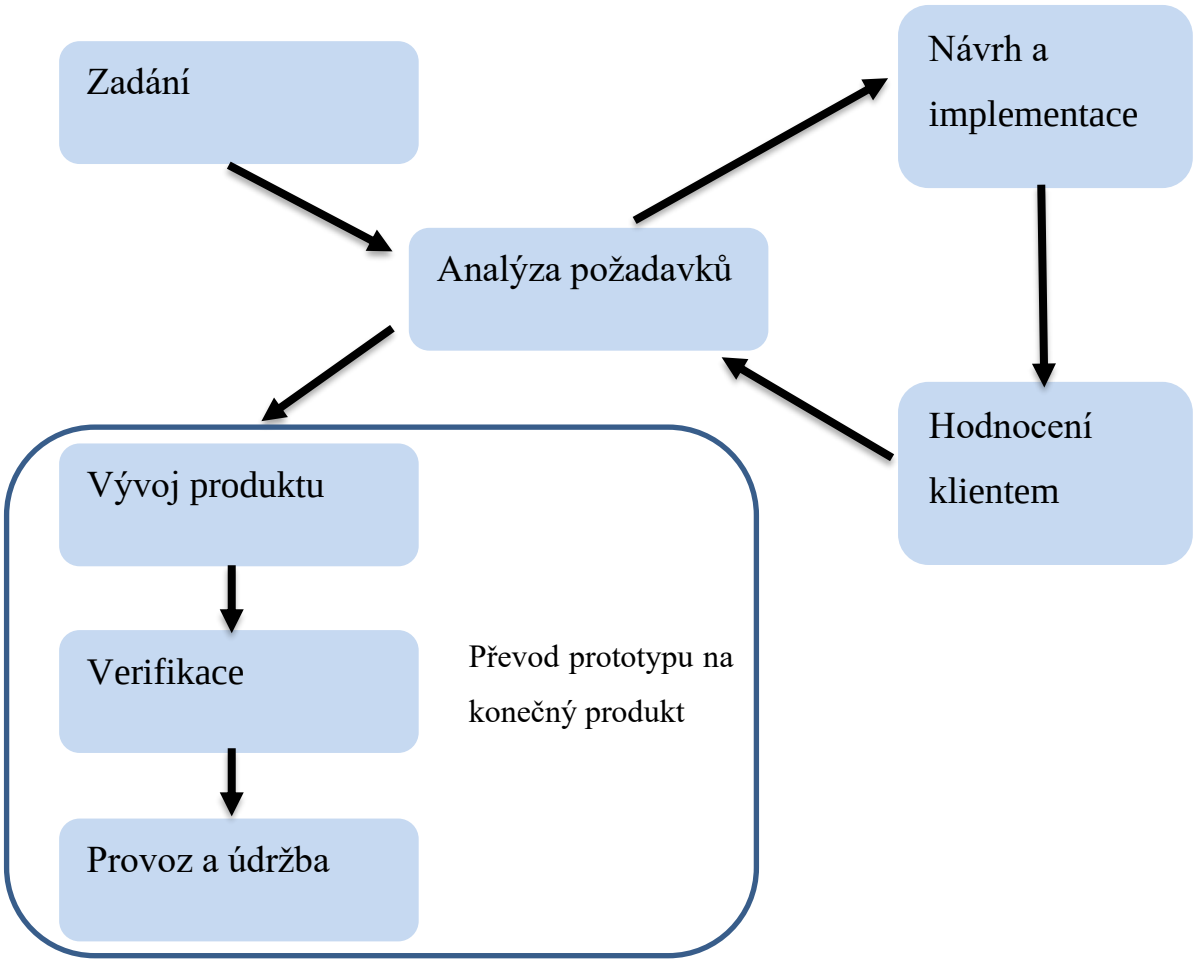
Obrázek 1 Vodopádový model

Prototypový model

Prototypový model eliminuje nejvýraznější nedostatek vodopádového modelu tím, že v průběhu tvorby produktu vznikají funkční částečné implementace produktu, a díky tomu má zákazník možnost reagovat na vznikající produkt ještě před předáním. Použití tohoto modelu je vhodné, pokud při zadání produktu nejsou specifikovány všechny funkcionality. Model vychází ze známých fází životního cyklu produktu. Po sběru a analýze požadavků dochází k tvorbě jednoduchých prototypů pro následné zhodnocení klientem. Vývojáři poté reagují na klientovo hodnocení a iterativně se opakuje předchozí krok s tvorbou a hodnocení prototypu do doby, než je prototyp ve finální podobě a ke spokojenosti klienta. Po tomto kroku dochází k finální implementaci produktu, jeho verifikaci, nasazení do produkčního prostředí a údržbě.

Mezi výhody tohoto modelu patří především schopnost reagovat na změny ze strany zákazníka v průběhu tvorby produktu, je tedy zapojen v celém procesu vývoje, a to může včasně eliminovat případné zvýšení nákladů na vývoj produktu.

Nevýhodou je použití tohoto modelu u rozsáhlých systémů. Z toho důvodu je dán přesný harmonogram, který určuje množství a termíny pro opakování prototypů.



Obrázek 2 Prototypový model

2.3 Softwarové architektury

Obecná definice softwarové architektury není jednoznačná, často dochází k tomu, že si ji každý vyloží po svém a mírně se navzájem liší. Obvykle je definována jako soubor softwarových komponent s jejich funkcionalitami, vlastnostmi a vzájemnými vztahy. Je také dobré zmínit, že jednotlivé softwarové architektury nepředstavují konkrétní technologii, ale představují určitý koncept nebo postupy při návrhu a tvorbě aplikací. Časem se v každém odvětví modernizuje, užívají nové technologické postupy a oblast softwarové architektury v tomto není výjimkou. K prvním programům se přistupovalo jako k velkým samostatným blokům, které obsahovaly všechny potřebné součásti systému, to naráželo na problémy v případě potřeby komunikace s okolím, nebo aktualizaci funkcí původního systému. Časem vznikaly nové přístupy, které umožňovaly komunikaci mezi ostatními bloky na síťové vrstvě, a umožňovaly opakovanou použitelnost již funkčních komponent pro více aplikací. Současným trendem je přístup,

kdy je každá funkcionalita systému jako samostatná služba a celý program je tvořen právě z množiny těchto služeb. Při takovém návrhu je jednoduché modifikovat jednotlivé součásti bez nutnosti modifikace celého programu. Aktuálně se v praxi užívá mnoho druhů softwarových architektur, ovšem v této práci budou popsané jen vybrané architektury.

2.3.1 Monolitická architektura

Monolitická architektura je považována za původní, klasický koncept při vývoji informačních systémů nebo aplikací. Základem monolitické architektury je „jeden samostatný blok“, dále označován jako monolit, obsahující všechny součásti systému, které jsou na stejné úrovni, navzájem závislé a provázané. Typická monolitická aplikace je tvořena třemi vrstvami, mezi které patří uživatelské rozhraní, business logika a datová vrstva. Bussiness logika je v tomto vztahu oddíl zodpovědný za zpracování dat získaných buď prostřednictvím uživatelského rozhraní nebo dat získaných z datové vrstvy. Datová vrstva je zodpovědná za komunikaci s databází. [5]

Vývoj systému nebo aplikace na základě konceptu monolitické architektury je obecně jednodušší a rychlejší, oproti jiným konceptům je veškerá funkcionalita aplikace v jednom monolitu a není potřeba řešit komunikace vně monolitu. Při vývoji je také důležité zvolit vhodnou technologickou platformu, pokud je monolitická aplikace vyvíjena v .NET Core, tak je pak nemožné přejít na jinou technologickou platformu.

Údržba a aktualizace monolitických systémů je díky své struktuře náročnější. Postupem času při vývoji roste objem kódu aplikace a případné aktualizace nebo přidávání funkcí či knihoven může znamenat časově a objemově náročný refactoring.

Modularita

Kvůli provázanosti jednotlivých součástí monolitu může přidání nových funkcionalit nebo aktualizace triviální knihovny vyústit v nefunkčnost celé aplikace a následnému objemově a časově náročnému refactoringu. U větších projektů pak může docházet k ignorování potřeby refactoringu a následnému přiklonění k použití záplat, díky kterým může být kód aplikace nepřehledný.

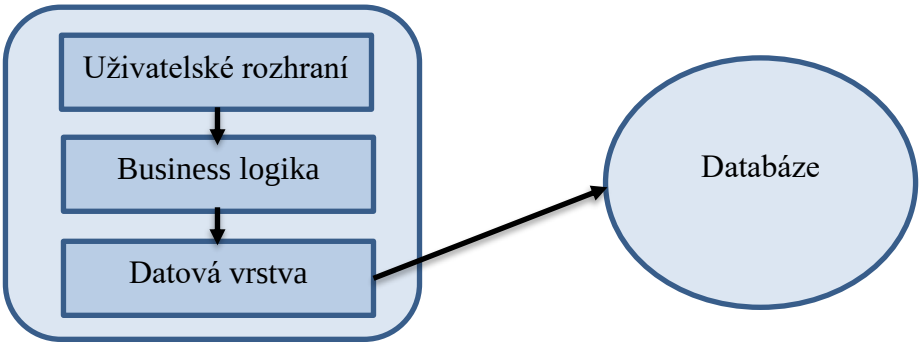
Nasazení a testování

Systém nebo aplikace v konceptu monolitické architektury obsahuje všechny komponenty a závislosti uvnitř monolitu, a právě díky tomu je relativně snadné nasazení

aplikace na testovací nebo produkční prostředí. S postupným vývojem monolitické aplikace dochází ke zvýšení objemu kódu, což může vést ke zvyšování doby potřebné pro nasazení aplikace. Testování monolitické aplikace probíhá při spuštění celé instance aplikace, to u větších projektů může být náročné na výpočetní prostředky i na čas.

Škálování a správa

Pro škálování komponent monolitické aplikace je nutné spuštění instance celé aplikace naráz, není možné spustit pouze jednu komponentu. Pro efektivní škálování je poté potřeba spustit několik instancí aplikace, což má za následek vysoké nároky na výpočetní prostředky. Z povahy monolitu může při selhání jedné komponenty spadnout celá aplikace, to pak vede i k časové náročnosti.



Obrázek 3 Monolitická architektura

2.3.2 Servisně orientovaná architektura (SOA)

Koncept servisně orientované architektury je evolucí v oblasti softwarové architektury. Lze jej označit za nástupce monolitické architektury, neboť řeší nedostatky, které vyplívají z konceptu monolitu. Jak již název napovídá, tak tato architektura je orientovaná na služby, kdy jednotlivé služby představují funkcionality aplikace, které bývají navzájem provázané a dohromady tvoří větší funkční bloky. Služby jsou navrhovány tak, aby byly navzájem nezávislé, mohly mezi sebou komunikovat prostřednictvím komunikačních protokolů po síti, mohly být opakovaně použitelné, v praxi jsou konkrétní služby poskytnuty několika komponentám, a aby byly jednoduše modifikovatelné. Služby nemusejí být technologicky závislé, z čeho vyplívá, že

jednotlivé služby mohou být vytvořeny na základě různých technologických platforem. [6; 7]

V konceptu vystupují tři role, mezi které patří konzument služby, poskytovatel služby a registr služeb. Všechny tyto role spolu komunikují prostřednictvím definovaných zpráv nezávislých na technologické platformě. Registr lze definovat jako katalog dostupných služeb, zároveň se stará o zajištění rychlé a spolehlivé komunikace mezi poskytovatelem služeb a konzumentem. Poskytovatel služby se stará o registraci služeb a komunikaci jak s registrem, tak s konzumentem služeb. Konzument v tomto vztahu vyhledává služby v registru a posílá požadavky na služby poskytovateli, vytvoří kontrakt a následně poskytovatel vrací odpověď konzumentovi. [8]

Modularita

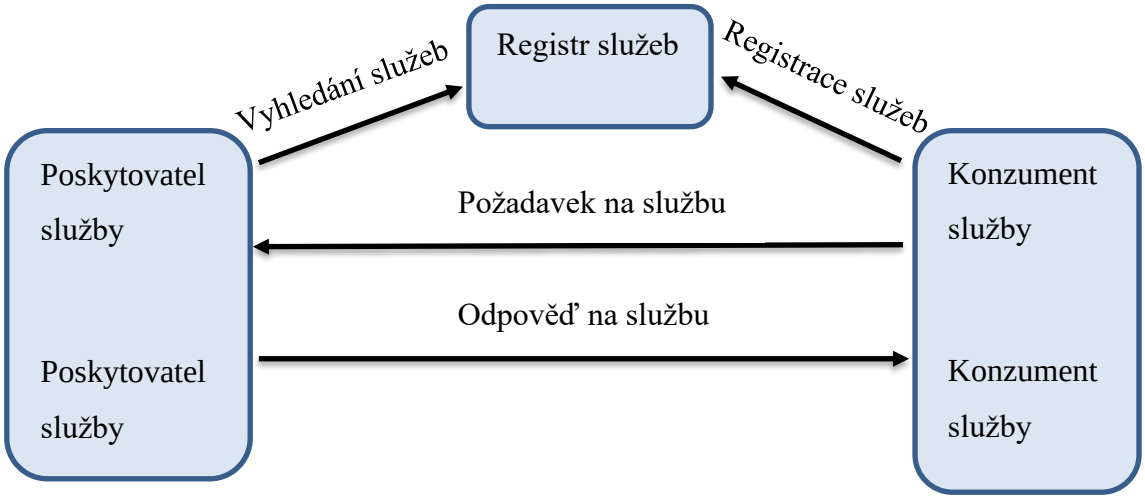
Jak již bylo zmíněno, tak servisně orientovaná architektura řeší nedostatky monolitické architektury. Pokud se dodrží pravidla pro návrh a implementaci na základě tohoto konceptu, pak SOA umožňuje snadnější modifikaci jednotlivých služeb, většinou jsou jen slabě provázané a změna jedné služby neovlivní celou aplikaci. V nejhorším možném případě bude nutné modifikovat několik provázaných služeb. Zároveň aplikace v tomto konceptu umožňuje komunikaci s okolím a není izolována, jako tomu je v případě monolitu.

Nasazení a testování

Náročnost nasazení do lokálního testovacího nebo do produkčního řešení závisí na použité technologii. Vzhledem k tomu, že se při konceptu SOA běžně nevyužívá kontejnerizace a zároveň se nejedná o monolit, tak může být výsledné nasazení složitější. Při testování konkrétní služby není nutné pouštět celou aplikaci, lze testovat právě tu konkrétní službu nebo služby, které jsou navzájem provázané.

Škálování a správa

Škálovat lze jednotlivé služby, což umožňuje nižší nároky na výpočetní zdroje nežli v případě, že je pro škálování nutné pustit celou instanci aplikace. Pro spravování služeb lze použít orchestrační nástroje, které zjednodušují veškerou správu.



Obrázek 4 Servisně orientovaná architektura

2.3.3 Mikroslužbová architektura

Mikroslužbová, známá také jako mikroservisní, architektura je v současném světě velmi často skloňovaný termín v oblasti softwarové architektury nejen kvůli svým výhodám a aktuálnímu trendu, ale také kvůli častějším migracím monolitických systémů na mikroslužby. Koncept této architektury je evolucí Servisně objektové architektury, s kterou má několik společných principů. Hlavní myšlenkou je vytvoření nezávislých, vzájemně neprovázaných služeb, které spolu komunikují a dohromady tvoří celý systém nebo aplikaci. Oproti konceptu monolitické architektury, kdy jsou všechny komponenty monolitu, se jedná o přesný opak čili dekompozici monolitu na samostatné služby. Samotný návrh systému v konceptu mikroslužbové architektury je náročný z hlediska zajištění plnohodnotné komunikace mezi službami, ale pokud se při návrhu a implementaci dodrží základní pravidla mikroslužeb, pak je takový systém velmi dobře škálovatelný, jednoduše nasaditelný a především modulární. Každá služba pracuje na principu jedné odpovědnosti, to v praxi znamená, že se každá služba soustředí na právě jednu funkcionalitu. [9]

Modularita

Mezi největší přednosti této architektury patří již zmíněná modularita. Díky principu jedné odpovědnosti je v případě přidávání nových služeb nebo změny současných jasné, kde je potřeba udělat úpravy, protože jsou jasně vymezené hranice služby. Není tedy potřeba přepisovat všechny služby systému. Zároveň nejsou služby omezené na použitou softwarovou technologii, je tedy možné různé služby vyvíjet na základě různých softwarových technologiích. Komunikace mezi jednotlivými službami se nejčastěji

odehrává prostřednictvím odlehčených protokolů, kterými jsou http, REST nebo Thrift API. [9]

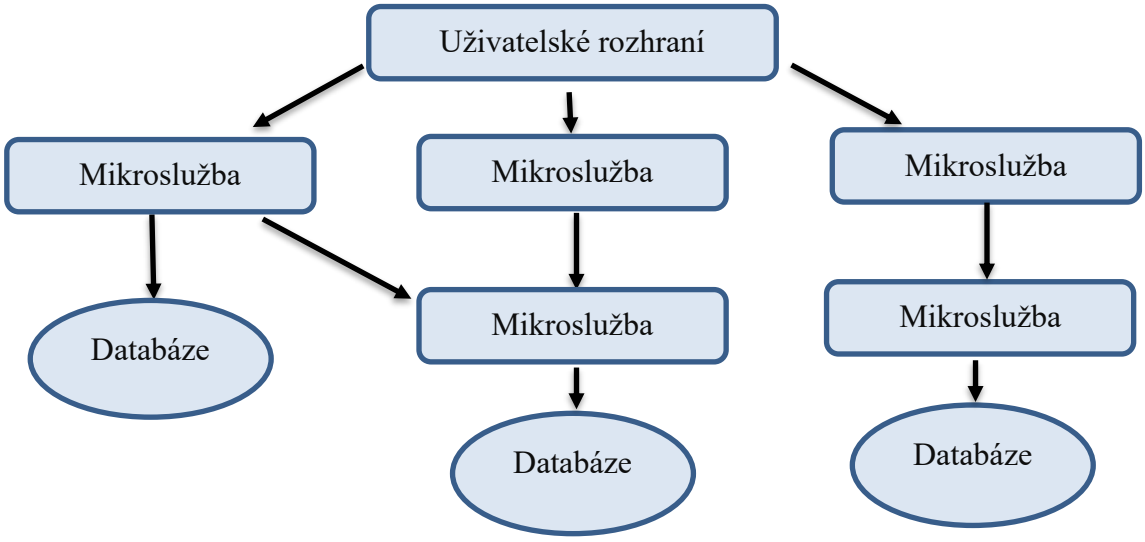
Nasazení a testování

Systémy v konceptu mikroslužbové architektury velmi často užívají principu kontejnerizace. Není to pravidlem, ale díky výhodám, které kontejnery přináší, se často mikroslužby užívají právě v kontextu mikroslužeb. Kontejnerizaci a kontejnerům je v této práci věnována samostatná kapitola, lze již v této fázi tvrdit, že obecně užití kontejnerů znamená jednodušší nasazení na různé OS a různá zařízení.

V případě změny služby v aplikaci postavené na mikroslužbách, není nutné testovat celý systém, ale postačí testování a poté nasazení právě té změněné služby. To je výhodou oproti monolitické aplikaci, kdy je nutné otestovat a následně nasadit celou aplikaci.

Škálování a správa

Díky dekompozici větších celků na malé služby zajišťující právě jednu funkcionalitu, je mnohem jednodušší škálování a správa jednotlivých služeb. V případě škálování mikroslužbové aplikace postačí vytvořit replikaci konkrétní služby, bez nutnosti spuštění nové instance celého systému. To umožňuje nižší nároky na výpočetní zdroje a čas. Pokud by aplikace byla nasazená v cloudu, pak se díky nástrojům pro orchestraci škálují jednotlivé služby automaticky, oproti tomu u velkých monolitických aplikací nelze tento přístup použít, protože škálování je možné pouze v případě, že je celá aplikace spuštěna.



Obrázek 5 Příklad mikroslužbové architektury

2.3.4 Porovnání architektur

Výběr vhodné architektury pro vývoj softwarového produktu není jednoznačný. Výběr se realizuje na základě analýzy požadavků, zároveň zde hraje významnou roli zkušenost týmu, který bude produkt vyvíjet a v neposlední řadě také ekonomické aspekty. Výběr je kritickým krokem a není dobré jej podcenit.

Z hlediska jednoduchosti vývoje jednoznačně vítězí aplikace v konceptu monolitické architektury. Odpadá zde nutnost definování komunikace s vnějšími moduly a vývoj takové aplikace je zpravidla ekonomicky výhodnější. Nutno ale poznamenat, že tyto výhody platí v případě monolitické aplikace, která nemá vizi v budoucnu vylepšovat a přidávat nové funkcionality a je již ve fázi zadání plně specifikována.

Pokud se po nasazení aplikace do produkčního řešení počítá s její modifikací v průběhu času, pak je použití monolitické aplikace nežádoucí a vítězí zde společně servisně orientovaná architektura s mikroslužbovou architekturou. Při výběru mezi servisně orientovanou a mikroslužbovou architekturou je dobré brát v potaz velikost projektu, přenositelnosti funkcionalit, komunikaci a ekonomické aspekty.

Z pohledu správy a škálovatelnosti vítězí SOA a mikroslužbová architektura oproti monolitické. Vždy je jednodušší škálovat a spravovat menší funkční bloky, v tomto případě služby zajišťující funkcionalitu než služby monolitické aplikace, která pro škálování vyžaduje spuštění celé aplikace. Pokud by aplikace byla nasazená v cloudu, pak se díky nástrojům pro orchestraci škálují jednotlivé služby automaticky, oproti tomu u velkých monolitických aplikací nelze tento přístup použít, protože škálování je možné pouze v případě, že je celá aplikace spuštěna.

2.4 Kontejnerizace a kontejnery

S rozšířením servisně orientované a mikroslužbové architektury bylo potřeba zajistit izolaci jednotlivých služeb a všech jejich závislostí. Jedním ze způsobu vytvoření izolovaného aplikačního prostředí pro běh služeb je vytvoření virtuálního stroje na základě virtualizace prostřednictvím hypervizoru. Nicméně tradiční pojetí virtualizace, při které vznikají samostatné VM na fyzickém hardwaru, nese některé důsledky, které mohou být pro konkrétní softwarové řešení nevhodné. Kontejnerizace nepotřebuje VM a jejím cílem je vytvoření izolované funkční jednotky s nejmenší možnou velikostí a jednoduchou přenositelností. Stává se tak vhodnějším prostředím pro projekty, které pro

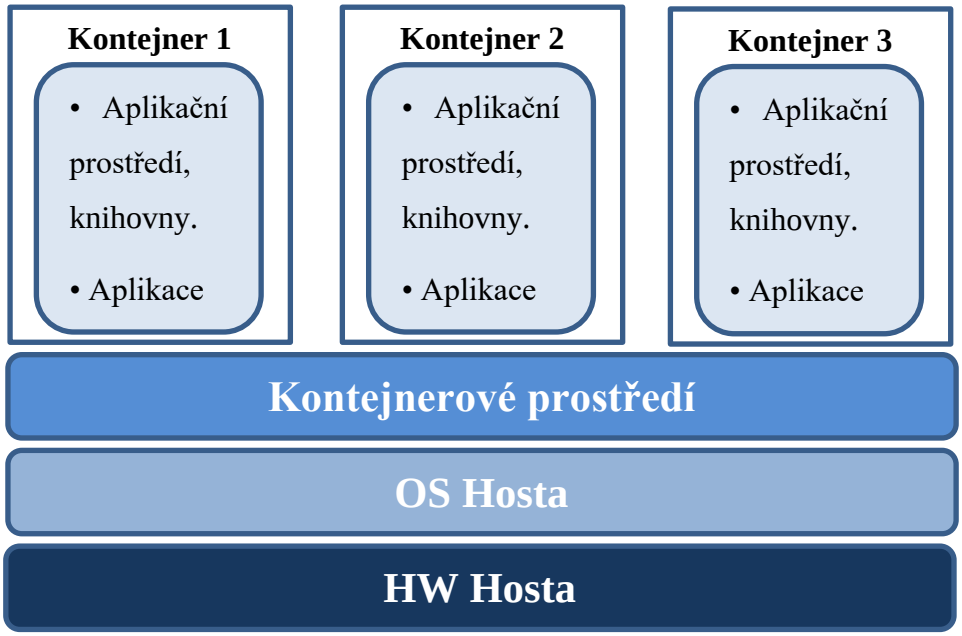
své řešení nevyžadují tradiční hardwarově náročnou virtualizaci. Rozdílům mezi tradiční VM a kontejnerizací bude věnován v této práci větší prostor.

2.4.1 Historie kontejnerizace a kontejnerů

Za počátek vzniku kontejnerů se považují 80. léta minulého století, kdy byl představen chroot, což je unixový příkaz pro změnu kořenového adresáře. Tato funkcionalita umožňuje změnit kořenový adresář pro procesy a tím omezit přístup procesů pouze do vybraných adresářů. V počátku 20. století došlo k vydání nové verze operačního systému FreeBSD *(FreeBSD 4)* a představení funkcionality jails (vycházející z konceptu chroot), která dovoluje rozdělení počítače s operačním systémem FreeBSD na několik dalších nezávislých systémů, což umožňuje izolování služeb od uživatelského prostředí. V průběhu let 20. století vznikaly další funkcionality zaměřené na izolaci služeb a aplikačního prostředí především pro operační systémy založené na linuxovém jádru. Pomyslným milníkem se v kontejnerizaci označuje rok 2013 a příchod Dockeru, kterému bude v této práci věnován širší prostor. Od této doby vzrostla popularita kontejnerizace ruku v ruce s orchestračními nástroji umožňující správu kontejnerů. [10]

2.4.2 Kontejnery

Kontejner představuje samostatnou virtuální jednotku s izolovaným aplikačním prostředím, která umožňuje nasazení a běh aplikace nebo její části. Zpravidla kontejnery obsahují jen to nejnutnější pro nasazení a běh služby, konkrétně se tak jedná o prostředí pro aplikaci a zdrojový kód aplikace včetně všech závislostí, konfigurací a knihoven potřebných pro chod aplikace. V kontejnerech by se tak nemělo nacházet nic zbytečného, cokoliv navíc odporuje pravidlům pro kontejnery a zbytečně zvyšuje velikost kontejneru. Kontejnery na rozdíl od standardních VM nevirtualizují hardware čili nevytváří nezávislé virtuální stroje, ale běží nad jádrem operačního systému hostitelského zařízení a sdílí jeho zdroje. Aktuálně existuje celá řada poskytovatelů kontejnerizační technologie, mezi nejznámější patří Docker, Amazon Web Services (AWS), Linux containers (LXC), Java containers a Hyper-V containers. [11]



Obrázek 6 Schéma příkladu kontejnerů

Výhody kontejnerizace

Kontejnerizace umožňuje snadné nasazení napříč různými operačními systémy. Pokud aplikace využívající kontejnerizaci pracuje správně v lokálním prostředí, poté ji bude možné nasadit a provozovat i v produkčním prostředí. Samozřejmě záleží na použité technologii pro kontejnery, protože některé z nich mají požadavky na specifický software, který je obvykle dostupný pro většinu operačních systémů. Díky jednoduchosti nasazení je kontejnerizace vhodná pro softwarové produkty, které budou nasazené v cloudu.

Kontejnery jsou výhodným použitím při paralelním běhu funkcionalit aplikace, kontejnery lze při nasazení rozdělit do klastrů pro zajištění optimálního výkonu a škálování. Škálování a správa kontejnerů se obvykle provádí prostřednictvím orchestračních nástrojů, díky nim lze spravovat jednotlivé kontejnery i jednotlivé klastry, případně lze nechat škálování automaticky. O orchestračních nástrojích se lze dočíst v této práci (*2.5 Nástroje pro správu kontejnerů*). [12]

Kontejnery nejsou samostatnými virtuálními stroji, z toho vyplívá, že nepotřebují celý operační systém pro svůj chod. Díky tomu nejsou tak náročné na výpočetní zdroje.

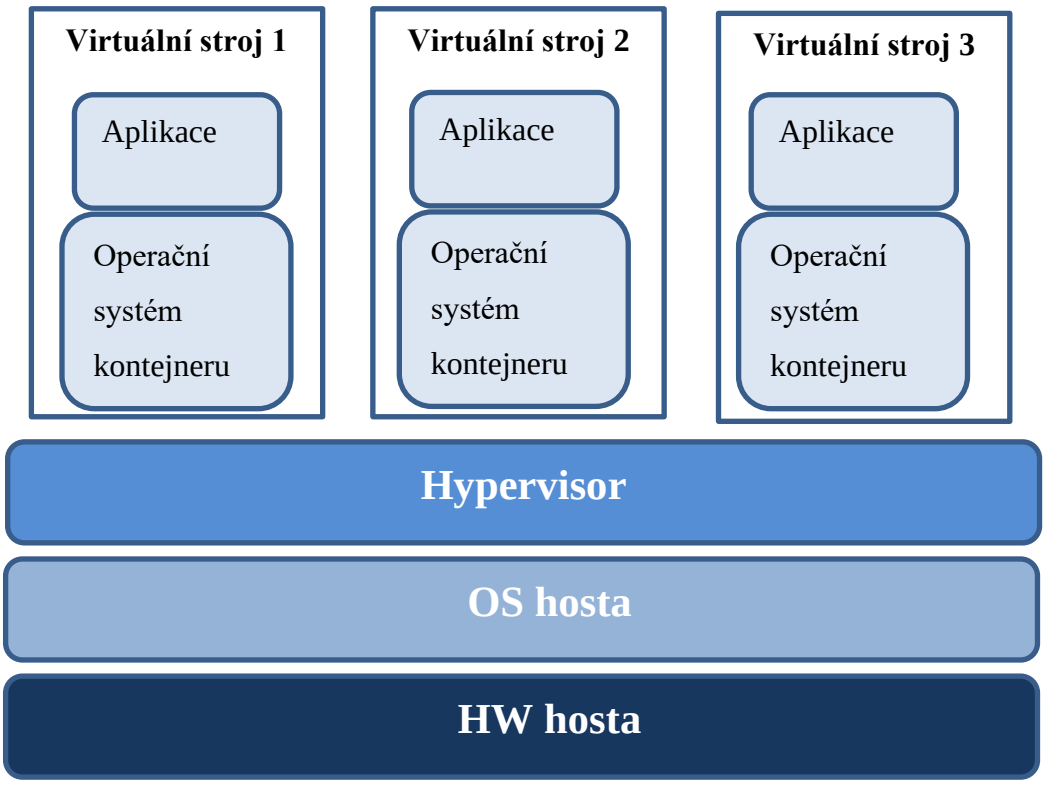
Využití kontejnerizace umožňuje, při správně navrženém softwarovém produktu, jednodušší přidávání nových nebo modifikaci aktuálních funkcionalit.

Nevýhody kontejnerizace

Hlavní nevýhodou kontejnerizace je nedostatečná izolace kontejnerů od hostitelského operačního systému. Jak již bylo zmíněno, tak kontejnery nevytváří virtuální stroje založené na hardwaru, ale běží nad jádrem operačního systému hostitelského zařízení a sdílí jeho prostředky. Vždy existuje bezpečnostní hrozba, že může dojít k poškození hostitelského operačního systému, ta by mohla ovlivnit i nasazené kontejnery. Z hlediska vývoje je nevýhodou cena a kvalita návrhu softwarového produktu, který bude využívat kontejnerizace. Přestože kontejnerizace je aktuálně trendem, tak není moc kvalitních kontejnerizačních architektur ani mnoho kvalitních vývojářů, kteří se v kontejnerizace perfektně vyznají, díky tomu kvalita kontejnerizovaných produktů není vždy dobrá. Softwarový produkt využívající kontejnerizace se běžně od tradičního řešení může výrazně prodražit, právě kvůli komplikovanému návrhu a celkově náročné implementaci. [12]

2.4.3 Porovnání kontejnerů a virtuálních strojů

Před samotným porovnáním technologií je nutné se seznámit s definicí virtuálních strojů. Virtuální stroj je označení pro software, který simuluje fyzické prostředí běžného zařízení (počítač, server). Z toho vyplívá, že v rámci jednoho hostitelského fyzického zařízení lze vytvořit několik dalších virtuálních strojů, které se chovají jako běžné výpočetní jednotky. Každý virtuální stroj má vlastní operační systém a je izolován od ostatních virtuálních strojů i v případě, že jsou na stejném hostitelském fyzickém zařízení. Virtuální stroje se obvykle používají na serverech, ale je možné je provozovat i na běžných osobních počítačích, jejichž CPU *(centrální procesorová jednotka)* podporuje virtualizaci. Virtuální stroje mají dlouhou historii, datují se již od 70. let minulého století, kdy byly používané u sálových počítačů. V té době také vznikla definice, že virtuální stroje jsou efektivní izolovanou kopií skutečného počítače. Mezi populární poskytovatele technologie virtuálních strojů aktuálně patří VMware vSphere, VirtualBox, Hyper-v a Zen. [11]



Obrázek 7 Schéma virtuálních strojů

Virtuální stroje i kontejnery se používají k tvorbě izolovaných virtuálních prostředí, které umožňují vývoj a testování softwarových aplikací. Navzájem se liší z několika pohledů.

Princip

Virtuální stroje sdílí výpočetní prostředky hostitelského zařízení *(host)*. Jsou tedy jeho návštěvníky *(guest)*, kteří se vytvářejí na základě hypervisoru. [13]

Kontejnery na rozdíl od VM virtualizují pouze operační systém čili nevytváří virtuální izolované stroje v hostitelském zařízení, ale virtuálně izolují aplikace nebo služby, přičemž sdílí operační systém a výpočetní prostředky s hostitelským zařízením. [13]

Operační systém

Virtuální stroje umožňují v rámci hostitelského zařízení vytvářet zcela izolované instance strojů s jiným operačním systémem, než je na hostitelském zařízení. Díky tomu může na jednom hostitelském zařízení běžet mnoho VM s různými operačními systémy.

Oproti tomu kontejnery se mohou obejít bez plné verze operačního systému a vystačí si s upravenou odlehčenou verzí operačního systému pro kontejnery nebo se sdílenou

verzí hostitelského zařízení, díky tomu jsou kontejnery objemově mnohem menší než VM.

Náročnost na zdroje

Virtuální stroje spotřebují větší množství výpočetních prostředků. Je tomu především kvůli velikosti operačního systému a jeho závislostí. Nelze spustit pouze specifickou část VM, ale je nutné spustit celou její instanci.

Kontejnery mají nejmenší možnou velikost, v extrémních případech se může pohybovat již v řádech desítek MB. Díky tomu je možné spouštět několik běžících instancí kontejneru naráz i při zachování nízkých požadavků na výpočetní prostředky.

Bezpečnost

Vzhledem k tomu, že VM je zcela izolované od hostitelského zařízení, tak je vhodným prostředkem pro testování, bez rizika ohrožení hostitelského zařízení nebo dalších instancí VM. [13]

Kontejnery sdílí jádro hostitelského operačního systému, tím pádem nejsou od něj plně izolované a jsou vystaveny potencionálním bezpečnostním hrozbám, které mohou ovlivnit běžící kontejnery. [13]

Škálování aplikací a správa

Škálování aplikací běžících ve VM je výpočetně náročnější především z toho pohledu, že je potřeba vytvořit několik běžících instancí celého VM.

Z výše uvedených rozdílů vyplívá, že virtuální stroje jsou z pohledu vývoje softwarových aplikací vhodnějším přístupem, pokud aplikace potřebuje pro svůj běh kompletní operační systém, který může být zároveň odlišný od operačního systému hostitelského zařízení. Také z důvodu bezpečnosti a plné izolovanosti od ostatních aplikací. A v neposlední řadě, pokud je potřeba nasadit více aplikací na jednom hostitelském zařízení. [11]

Kontejnery jsou vhodnějším přístupem z hlediska vývoje softwarových aplikací, pokud je potřeba pouštět více instancí konkrétní aplikace. Pokud nepotřebují plnou verzi operačního systému, tak se kontejnery načítají mnohem rychleji a stávají se tím efektivnějšími. Nebo pokud má výsledný softwarový produkt běžet na jakékoliv základní infrastruktuře. [13]

Tabulka 1 Porovnání kontejnerů a VM

Hledisko	Kontejner	VM
Operační systém	Nepotřebuje k činnosti celý operační systém. Vystačí si s odlehčenou kopií OS.	K činnosti je potřebná plná verze OS.
Velikost	Díky absenci OS, je velikost kontejnerů malá.	VM kvůli plné verzi OS zabírají více místa.
Výpočetní nároky	Menší náročnost na výpočetní prostředky. Požadavky pouze na SW.	Virtualizace potřebuje vyšší výpočetní prostředky a specifický HW.
Správa	Správa jednotlivých kontejnerů. Umožňuje kontejnery replikovat, odstraňovat, vytvářet.	Správa celé VM prostřednictvím hypervizoru.
Rychlost	Rychlejší start díky absenci plné verze OS.	Pomalejší start samotného OS.
Bezpečnost	Bezpečnostní hrozby díky sdílenému jádru hostitelského OS. A kvůli nedostatečné datové integritě.	Bezpečnost je daná konkrétním VM, který je plně izolovaný.
Perzistence dat	Po ukončení práce kontejneru, dojde k odstranění dat vzniklých činností kontejneru. Nutné data ukládat vně kontejneru v hostu.	VM si udržuje data.

2.4.4 Docker

Docker je jak název pro společnost, tak název pro open-source kontejnerizační platformu určenou pro vývoj, distribuci a běh kontejnerizovaných aplikací. První verze Docker byla vydána v roce 2013 a pro kontejnery používala LXC *(Linux containers),* které později nahradila vlastním řešením Libcontainer. V průběhu času společnost rozšiřovala své portfolio nástrojů pro práci s kontejnery a aktuálně se dá považovat za standard v oblasti kontejnerizace. Docker není pouze nástrojem pro vytváření kontejnerů, ale obsahuje také nástroje pro správu kontejnerů a vytváření klastrů pro kontejnery. [10]

Obecně docker poskytuje a definuje pravidla pro kontejnerizaci čili zabalení aplikace do izolovaného kontejneru se všemi jeho závislostmi, a definuje způsob správy kontejnerů prostřednictvím dockeřího ekosystému. Tím, že jsou kontejnery od sebe izolovány, tak může zároveň běžet velké množství kontejnerů. Myšlenkou dockeru je vývoj aplikací a jejich komponent s použitím kontejnerů, které se následně stanou jednotkou pro vývoj, distribuci a testování. Na konci životního cyklu vývoje softwarového produktu stačí jednoduše nasadit kontejnerizovanou aplikaci do produkčního prostředí jako kontejner nebo orchestrovanou službu. [14]

Architektura Docker

Docker je postaven na principu architektury Klient-Server *(Client-Server)*. Celá architektura dockeru zahrnuje docker klient *(Docker Client),* docker host *(Docker Host)* a docker registr *(Docker registry)*. [14]

- **Docker Daemon**

Daemon je obecně název pro perzistentní čili trvalý proces, který běží na pozadí. Z pohledu dockeru je jeho cílem správa kontejnerů v hostitelském zařízení. Je to zcela soběstačné prostředí, které umožňuje správu všech docker objektů. Naslouchá požadavkům z REST API, které přicházejí od docker klienta a na jejich základě provádí operace s kontejnery. Daemon také může komunikovat s cizími docker daemony, pokud existuje více hostitelských zařízení spravující službu nebo klastr. [15]

- **Docker Klient**

Klient je primárním přístupovým bodem pro uživatele dockeru, zajišťuje komunikuje s docker daemonem, který se stará o již zmíněnou správu kontejnerů zahrnující sestavení, běh a distribuci. Klient spolu s daemonem může běžet na stejném

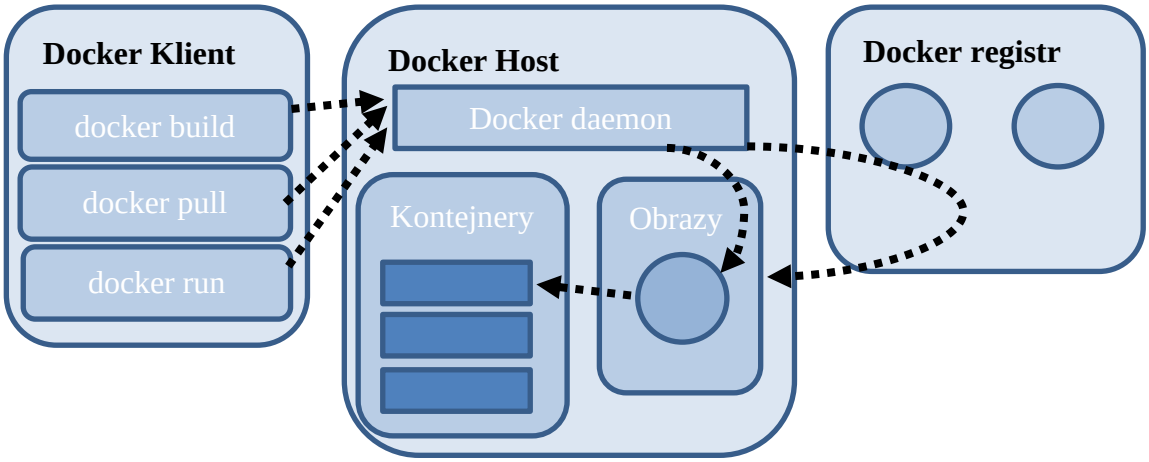
hostitelském zařízení nebo se lze klientem připojit i ke vzdáleným daemonům. Mezi klientem a daemonem se zprostředkovává komunikace prostřednictvím REST API, unixových socketů nebo síťovým rozhraním. [14]

- **Docker Host**

Host představuje prostředí potřebné pro spuštění a běh kontejnerizovaných aplikací. Skládá se z docker objektů a docker daemona, jeho prostřednictvím se vykonávají příkazy, které posílá klient.

- **Docker Registr**

Docker registr se stará o online ukládání nebo stahování docker obrazů. Docker zahrnuje v rámci svého ekosystému službu Docker hub, což je jejich vlastní volně dostupný repozitář určený pro sdílení obrazů kontejnerů. Umožňuje přístup všem uživatelům a poskytuje jim mnoho funkcí zdarma. Pro komerční prostředí nebo pro náročné uživatele Docker Hub poskytuje nadrámcové služby, které jsou ovšem zpoplatněny ve formě měsíčního předplatného. [14]



Obrázek 8 Schéma Docker architektury

- **Docker objekty**

Dockerfile

Dockerfile není definován jako samostatný objekt, ale je základním stavebním kamenem pro ostatní docker objekty. Definice a následná stavba obrazů vychází z dockerfile, což je soubor sloužící jako šablona pro definování instrukcí vedoucích

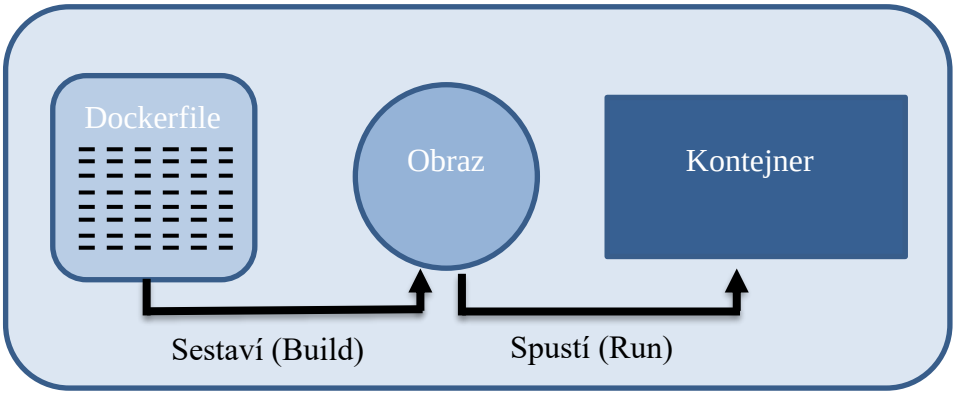
k sestavení požadovaného docker obrazu. Obsahuje tedy všechny kroky a závislosti pro sestavení obrazu který je předpokladem ke správnému běhu kontejnerů. [16]

Docker obrazy *(Docker Images)*

Docker obrazy slouží jako šablony obsahující set instrukcí pro vytvoření kontejneru. Sestavují se na základě definovaného dockerfilu, sám o sobě je obraz pouze ke čtení čili jakákoliv modifikace existující image není možná. V případě potřeby změn je potřeba přepsat instrukce v dockerfile a vytvořit nový obraz. [14]

Docker kontejnery *(Docker Containers)*

Kontejnery jsou spustitelné instance docker obrazů. Lze s nimi provádět celou řadu operací od vytváření, spouštění, zastavování, vypínání, migrace nebo mazání. Tyto operace se provádí na úrovni docker hostu. Docker kontejnery splňují všechny charakteristiky standardních kontejnerů čili obsahují jen to nejnutnější pro chod aplikace a jsou izolované od ostatních kontejnerů i od hostitelského zařízení. Zde je dobré připomenout, že s hostitelským zařízením stále sdílí jádro operačního systému, takže nejsou zcela izolované. Z hlediska perzistence dat je potřeba přihlédnout k tomu, jakým způsobem kontejnery vznikají, prvotní definice pochází již z dockerfile, z kterého se sestavuje obraz a jeho spuštěním vzniká běžící kontejner. Čili veškerá data vzniklá během činnosti kontejneru zaniknou s jeho ukončením. Pro zachování integrity dat v případě, kdy je potřeba navázat na data již vzniklá předchozí činností kontejneru, je nutné definovat jiné způsoby pro ukládání dat. Tyto způsoby jsou vysvětleny níže. [14]



Obrázek 9 Schéma tvorby docker obrazů a kontejnerů

- **Docker konfigurace sítě *(Docker Networking)***

Docker sítě *(Docker networks)* se řadí mezi docker objekty, ale zároveň mají svůj význam pro samotnou architekturu, proto lze konkrétní nastavení považovat za objekty, ale samotné způsoby konfigurace spadají pod architekturu dockeru.

Výhodou u docker kontejnerů je možnost propojení kontejnerů mezi sebou, nebo s jinými aplikacemi v rámci hostitelského zařízení. Docker podporuje několik síťových ovladačů, které umožňují komunikaci.

Most *(Bridge)*

Most je základním síťovým ovladačem, který se automaticky nastavuje jako výchozí, pokud není definováno jinak. Používá se především pro ty případy, kdy samostatné kontejnery komunikují se stejným docker hostem. [17]

Host *(Host)*

Ovladač host slouží pro samostatné kontejnery, které nevyžadují izolaci od docker hosta

Překryvné sítě *(Overlay)*

Překryvné sítě vytváří distribuovanou síť napříč množinou docker hostů, to znamená, že umožňuje propojení několika docker daemonů dohromady. Díky tomu zajišťuje síťové propojení mezi kontejnery, běžícími na různých hostech. Komunikaci je zároveň možné šifrovat a tím zajistit, že bude bezpečná. [14]

Sítě na základě MAC adresy *(Macvlan)*

Při tomto přístupu se nastavuje kontejnerům MAC adresa, díky tomu se poté v síti jeví jako samostatné fyzické zařízení. [17]

Bez síťové komunikace *(None)*

Pokud je zvolen síťový ovladač jako žádný, tak kontejner nemá žádnou možnost komunikace. Je tedy zcela izolovaný. [14]

- **Docker úložiště *(Docker Storage)***

Kontejnery vzhledem k jejich životnímu cyklu nejsou datově perzistentní. Kontejnery vznikají na základě docker obrazu a mohou běžet, zastavovat se, rušit a mazat. Aby kontejnery vždy běžely stejně, tak jsou spouštěny z jejich obrazů vzniklých na

základě definovaného dockerfile. Proto nemá běžící kontejner žádný vliv na obraz, který by čistě hypoteticky mohl uchovávat data z běhu kontejnerů. Pro zajištění perzistence dat kontejneru, například pro získání dat vzniklých v předchozí instanci kontejneru, existuje několik přístupů pro sdílení. [14]

Obsah dat *(Data Volume)*

Obsah *(Volume)* je jeden ze způsobů pro zajištění perzistence dat. Dochází k propojení adresáře kontejneru s hostitelským adresářem. V praxi dochází ke sdílení obsahu na obou stranách, díky tomu je možné dostat data jak do kontejneru, tak z něj do hosta. Zároveň lze do jednoho konkrétního sdíleného adresáře napojit i jiné kontejnery. [14]

Vázané propojení *(Bind mounts)*

Vázané propojení *(Bind mounts)* existují již od prvních verzí dockeru, jsou ovšem oproti obsahu dat *(data volumes)* značně limitované, protože závisí na souborovém systému hostitelského zařízení. Funguje na principu navázání souboru nebo adresáře do kontejneru prostřednictvím úplné cesty v hostitelském zařízení, sdílení je tak omezeno na přesnou adresářovou strukturu hostitelského zařízení. Mezi nevýhody patří nemožnost správy tohoto způsobu propojení pomocí docker příkazů v příkazovém řádku. Z hlediska zabezpečení existuje bezpečnostní hrozba, díky které se může měnit souborový systém hostitelského zařízení v důsledku činnosti kontejneru. [14]

Docker compose

Docker compose je nástroj, který je určen pro automatizaci nasazení aplikací skládajících se z mnoha kontejnerů. Umožňuje jejich sestavení a běh v definovaném pořadí, může jim přiřazovat konfiguraci, vystavovat porty nebo nastavit sdílené datové úložiště kontejneru a hostitelského zařízení. Tyto instrukce pro sestavení a běh se definují ve speciálním souboru *(docker-compose.yml)* ve formátu YAML, soubor má svá pevná pravidla určující strukturu. Použití a funkčnost nástroje je podmíněna třemi procesy. V prvním kroku musí existovat základní dockerfile, který představuje šablonu pro vznik kontejneru, v druhém kroku dochází k definování služeb v docker-compose.yml pro jednotlivé kontejnery a v posledním kroku se může přistoupit k následnému finálnímu sestavení.

Použití a výhody Dockeru

Použití Dockeru zefektivňuje celý životní cyklus vývoje softwarového produktu. Umožňuje vývojářům pracovat v jejich oblíbeném technologickém prostředí při zachování lokálního prostředí. Pro srozumitelnost je dobré připomenout, že pokud kontejnerizovaná aplikace běží v lokálním podmínkách, pak bude běžet i v produkčním prostředí. Kontejnery jsou také vhodnou volbou v případě využití průběžné integrace *(Continous Integration - CI)*. [14]

Docker kontejnery zajišťují, že aplikace uvnitř kontejnerů jsou izolované jak od hostitelského systému, tak od ostatních běžících kontejnerů. To zjednodušuje škálování jednotlivých kontejnerů a definuje pevné hranice kontejnerů.

Přenositelnost a snadná nasaditelnost kontejnerů je jednou ze stěžejních výhod kontejnerizace. V případě dockeru, který obsahuje široké spektrum nástrojů pro spouštění, nasazení a správu kontejnerů je kompatibilita nástrojů v jiných zařízeních podmíněná prostředím pro docker a eventuálně prostředím pro další služby spojené například s orchestrací kontejnerů. [18]

Mnoho aplikací v konceptu servisně orientované nebo mikroslužbové architektury mají oddělené komponenty, představující služby aplikace, které ve výsledku dávají dohromady funkční aplikaci. Docker obsahuje nástroj, který dokáže nasadit všechny potřebné kontejnery v požadovaném pořadí se všemi závislostmi a vystavuje aplikaci jako samostatnou jednotku s přístupem k jednotlivým službám. To znamená, že aplikace složená z množiny služeb se sestaví na základě definovaného postupu a lze k ní přistupovat jak z pohledu samostatné jednotky, tak k jednotlivým službám jednotky. [18]

Pro orchestrací a škálování kontejnerů existuje řada poskytovatelů služeb orchestrace. Orchestračním nástrojům je v této práci věnován prostor, tudíž zde se nachází pouze jednoduchá definice. Orchestrační nástroje slouží k automatizovanému nasazení, škálování, propojení s ostatními kontejnery a ke škálování. Samotný docker má vlastní orchestrační nástroj, který se nazývá Docker Swarm a ten je určen pouze pro práci s docker kontejnery. Pro uvedení příkladu nástroje třetí strany je vhodné zmínit Google's Kubernetes, který je často skloňovaným nástrojem v oblasti orchestrace kontejnerů.

Nástroj Docker compose, který patří do dockeřího ekosystému slouží k nasazení množiny docker kontejnerů. Na základě definované souboru pro docker compose se spustí množina kontejnerů se všemi závislostmi stejně, jako kdyby se pouštěli kontejnery

po jednom. To usnadňuje nasazení aplikace, která je postavena na více kontejnerech a eventuálně potřebuje spouštět kontejnery v určitém pořadí.

Nevýhody Dockeru

Standardně obrazy docker kontejnerů musí být sestavené na cílové platformě. To znamená, že Windows kontejner nebude fungovat na Linuxu a naopak. Nicméně stále existuje možnost, že se sestavení docker kontejnerů provede až na cílové platformě, nebo se kontejnery nasadí do virtuálního stroje cílového zařízení, takže existují možnosti pro požadovanou přenositelnost na jiné operační systémy. [18]

Docker kontejnery si neuchovávají data, která vznikla při jejich běhu. Pokud dojde ke spuštění kontejneru, jeho stopnutí a opětovnému spuštění, tak si kontejner neuchová žádná data z minulé existence. Pro tento případ je možné nastavit sdílení dat vně kontejneru nebo nastavit ukládání dat do databáze, obvykle na hostitelském zařízení. Data, která vzniknou v průběhu existence kontejneru se poté ukládají do sdíleného prostředí hosta a kontejneru, pokud se kontejner stopne a znova spustí, tak si může ze sdíleného prostředí načíst data a pokračovat, kde přestal.

Přestože docker nabízí celou řadu užitečných funkcionalit pro vytvoření, správu a nastavení kontejnerů, tak samotná implementace služeb, které mohou těžit z výhod dockeru, může být složitá pro nezkušeného uživatele.

2.5 Nástroje pro správu kontejnerů

Z definice kontejnerů je patrné, že při jejich použití může celá aplikace potřebovat ke své plné činnosti jednotky, stovky nebo tisíce běžících kontejnerů naráz. Při těchto počtech je náročná manuální správa a orientace mezi všemi kontejnery. Pro řízení je možné používat nejen automatizované skripty, ale i nástroje označené jako orchestrátory. Orchestrátory jsou obecně nástroje pro automatické řízení aplikace nebo systému a jejich komponent. V kontextu kontejnerů a kontejnerizace jsou orchestrátory nástrojem, který zjednodušuje práci s kontejnery, umožňuje automatizaci procesů s kontejnery, v případě detekce problému s kontejnerem jej dokážou restartovat, zároveň lze prostřednictvím orchestrátoru vstupovat do kontejnerů, pouštět a zastavovat kontejnery, přidělovat jim výpočetní kapacitu, monitorovat jejich stav nebo třeba měnit konfigurační soubory. Uvedeno je pouze několik funkcionalit, které nám orchestrace může poskytnout. Existuje

několik služeb pro orchestraci kontejnerů a každá z nich umožňuje rozdílné funkcionality, aktuálně mezi hojně používané patří Kubernetes, Docker Swarm, Amazon ECS a OpenShift. V této podkapitole se níže nachází popis vybraných služeb pro orchestraci kontejnerů a v závěru jejich porovnání. [19; 20]

2.5.1 Kubernetes

Kubernetes, znám také pod označením k8s, je open-source nástroj pro orchestraci kontejnerů, který byl vytvořen Googlem a nyní jej spravuje Linux Foundation. Kubernetes je komplexním systémem a celý jeho ekosystém zahrnuje ve svém portfoliu řadu funkcionalit jejichž cílem je poskytnout ucelenou platformu určenou pro správu kontejnerizovaných aplikací a služeb s důrazem na automatizaci procesů. [21]

Architektura Kubernetes

Kubernetes je postaven na architektuře klient-server a jeho základem je kubernetes klastr, který může být vytvořen prostřednictvím řady nástrojů. Ty se dělí především na klastry určené pro lokální nasazení a na produkční nasazení Kubernetes. Lokální klastry se používají při vývoji a testování a poskytují je následující nástroje Minikube, Kind, K3S nebo integrovaný základní Kubernetes klastr dostupný v Docker desktop. Pro produkční prostředí se používají nástroje Kops, Kubespray nebo kubeadm, které umožňují specifičtější konfiguraci vhodnou právě pro produkční prostředí.

Základní kubernetes klastr se skládá z jednoho hlavního uzlu *(master node)* a množiny vedlejších pracovních uzlů *(worker nodes)*. Vzájemný vztah mezi uzly je takový, že hlavní uzel je zodpovědný za řízení klastru a pracovních uzlů.

Kubectl

Kubectl je oficiálním Kubernetes nástrojem pro ovládání aplikačního rozhraní Kubernetes. Lze jeho prostřednictvím spravovat veškeré objekty klastru a také monitorovat jejich stav. [22]

Komponenty hlavního uzlu *(Master node components)*

- **Etcd klastr**

  Databáze na principu klíč-hodnota, používá se k ukládání informací o stavu Kubernetes klastru, o objektech aplikačního rozhraní a detailech o službách. Je přístupný pouze prostřednictvím API serveru. [23]

- **Kube-apiserver**

  Kubernetes API slouží pro interakci uživatelů s Kubernetes klastrem. Umožňuje správu, vytváření a konfiguraci klastru na základě požadavků, které validuje a případně vykoná a změny zaznamená do etcd databáze. [21]

- **Kube-controller-manager**

  Kontrolér je proces běžící na pozadí, monitoruje stav klastru prostřednictvím api-serveru a stará se o udržování definovaného stavu. [21]

- **Kube-scheduler**

  Organizuje zařazování nově vytvořených Podů do různých pracovních uzlů v závislosti na dostupných výpočetních prostředcích. Plánovač má veškerý přehled o celkově dostupných a aktuálně využívaných výpočetních prostředcích. [23]

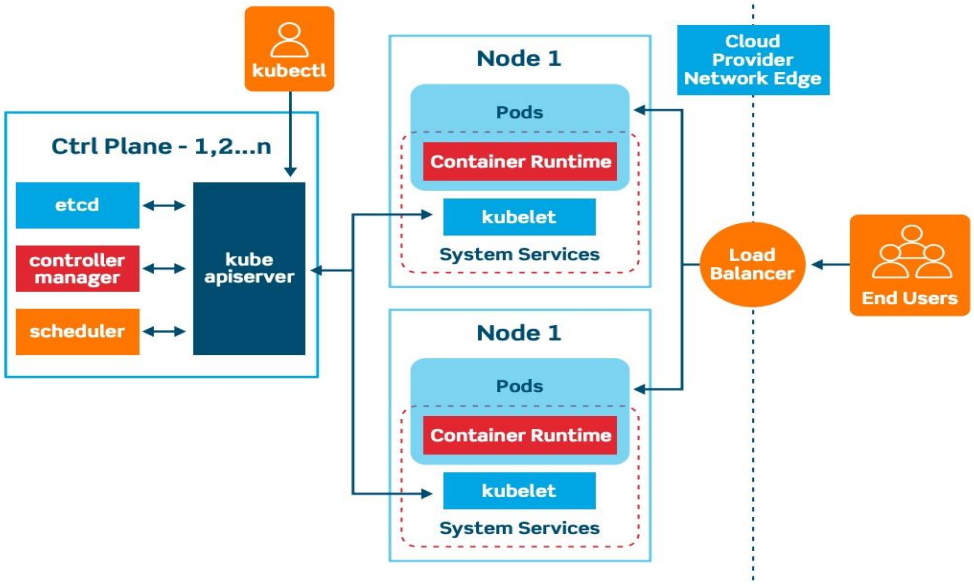
Komponenty pracovních uzlů *(Worker node components)*

- **Kubelet**

  Kubelet je hlavní službou pracovního uzlu, je zodpovědný za řízení kontejnerů a monitoring jejich stavu. Tato komponenta spolupracuje s hlavním uzlem a průběžně informuje o aktuálním stavu.

- **Kube-proxy**

  Síťová proxy, která běží na každém pracovním uzlu, definuje pravidla a zprostředkovává síťovou komunikaci Podů jak uvnitř, tak i vně klastru. [21]



Obrázek 10 Schéma Kubernetes Klastru

Objekty Kubernetes

Pod

Pod je nejmenším funkčním prvkem v Kubernetes, reprezentuje jeden nebo množinu kontejnerů běžících ve stejném prostředí. To znamená, že všechny kontejnery nacházející se v konkrétním Podu se zároveň nachází na stejném fyzickém zařízení. Kontejnery běžící v Podu navzájem sdílí stejnou síť a výpočetní prostředky a jsou od ostatních Podů izolovány. Pody nejsou navrženy pro dlouhodobý běh, pokud dojde k jejich zrušení nebo chybě, tak se smažou a následně se musí vytvořit nové. Kvůli této vlastnosti se při deploymentu nastavují repliky Podů, které určují kolik totožných kopií Podů bude naráz běžet, repliky slouží především k udržení funkčnosti služeb v Podech, i v případě přetížení některé z replik nebo jejího výpadku. [21]

Uzel *(Node)*

Uzly jsou fyzické nebo virtuální stroje, které jsou základem Kubernetes klastru a jsou zodpovědné za veškerou jeho činnost.

Služby *(Services)*

Kubernetes služby jsou definovány jako abstrakce nad Pody, jenž zprostředkovávají síťovou komunikaci a přiřazují neměnné IP adresy a DNS záznamy Podům. Služby mají využití především v těch případech, kdy dojde k restartování nebo vytvoření replik Podu. Standardně by se po takové akci každému Podu přidělila nová adresa, ale Kubernetes služby zajistí unikátní neměnnou IP adresu, která bude přidělena všem replik totožného Podu. [24]

Deployment

Deployment lze charakterizovat jako návod pro nasazení a běh kontejnerizovaných aplikací v Podech, vystupují ve formě souboru formátu YAML. Deployment je právě tím místem, kde se obvykle definuje počet replik konkrétních Podů, způsoby škálování a obrazy pro inicializaci kontejnerů. [21]

Výhody Kubernetes

Orchestrace prostřednictvím Kubernetes zefektivňuje celý proces nasazení kontejnerizovaných aplikací. Zároveň obvykle nedochází k výpadkům služeb, protože Kubernetes provádí pravidelnou aktualizaci kontejnerů, pouští jejich nové verze nebo

repliky, čeká až se nové kontejnery začnou podílet na práci a staré kontejnery ukončí. [24]

Díky automatizovanému škálování dokáže Kubernetes udržet kontejnery v činnosti i v případě, že některý z nich využívá nadměrné výpočetní prostředky a nestíhá odpovídat na všechny požadavky. V reakci na to se automaticky vytvoří nové repliky využívaného kontejneru, aby nedošlo k celkovému výpadku služby, a až v původní replice dojde ke snížení nároků na výpočetní prostředky, pak se nově vytvořené repliky ukončí a tím uvolní prostředky pro další kontejnery. [24]

Obecně používání orchestračních nástrojů vede k vyšší efektivitě. V kontextu Kubernetes dochází díky automatizaci procesů k rovnoměrnějšímu a tím efektivnějšímu využívání výpočetních prostředků, což v případě tradičních nebo cloudových serverů vede k finančním úsporám.

Nevýhody Kubernetes

Navzdory značným výhodám, které může použití Kubernetes přinést, je důležité zmínit fakt, že samotný vývoj aplikací, které se budou spravovat prostřednictvím Kubernetes, je technologicky náročný a značně nákladný jak z ekonomického, tak i z časového hlediska. Je tomu především z důvodu komplexity celého Kubernetes.

Nasazení a správa kontejnerů prostřednictvím příkazového řádku předpokládá u Kubernetes použití jejich vlastní specifické syntaxe kubectl. Která se liší oproti jiným syntaxím a její osvojení může být časově náročné. Existují tu však alternativy vůči příkazovému řádku v podobě webového API s grafickým rozhraním, ale většinou nepokrývají veškeré funkcionality. [19]

Jak již bylo zmíněno, samotné Kubernetes je velmi komplexní a má mnoho výhod. Ovšem jeho použití pro malé aplikace, které nemají vysoké nároky na výpočetní prostředky, je zbytečná a vede k časovým a ekonomickým ztrátám.

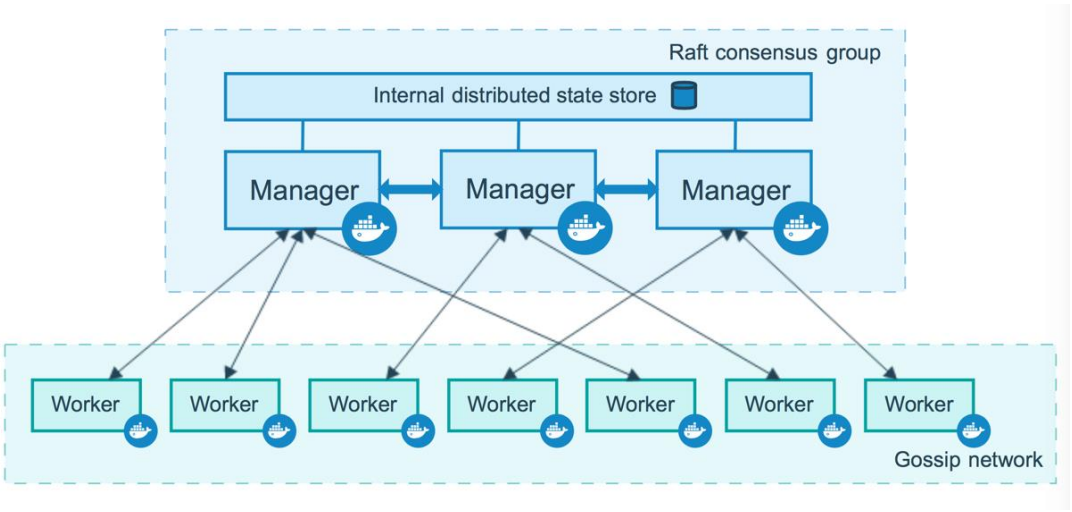
2.5.2 Docker Swarm

Docker swarm je nástroj pro orchestraci Docker kontejnerů. Integrován přímo do Docker prostředí a díky tomu nevyžaduje další instalační balíčky, zároveň umožňuje interakci se všemi komponentami Docker ekosystému. To je považováno za velkou výhodu, protože pokud uživatelé pracují s Docker kontejnery, tak se předpokládá, že jsou

zvyklí na toto prostředí a případné ovládání swarm klastru je poté jednoduché, protože pro zajištění orchestrace pak odpadá nutnost učit se jinému prostředí. [19]

Architektura Docker Swarm

Docker swarm se skládá z množiny fyzických nebo virtuálních strojů, které navzájem spolupracují v jednom swarm klastru. Klastr je pak tvořen manažerskými uzly *(managers)* a pracovními uzly *(workers)*. Manažeři se starají o správu klastru a správu jednoho nebo množiny pracovních uzlů. Pracovní uzly v tomto vztahu vykonávají úlohy, které přijímají od manažerů. Pro zajištění konzistentního stavu celého klastru se využívá algoritmus *Raft*, což je konsensuální protokol a v kontextu swarm slouží ke komunikaci a distribuci požadavků napříč manažerskými uzly klastru. [14]



Obrázek 11 Schéma Docker swarm, Dostupné z [14]

Objekty Docker swarm

- **Manažerské uzly *(Manager nodes)***

Manažerské uzly řídí veškerou činnost v klastru, mezi nejvýznamnější úlohy uzlu patří zpracování požadavků uživatele, replikace úloh, posílání požadavků na úlohy pracovním uzlům, zajištění orchestrace, zajištění síťové komunikace a vystavování portů, monitorování stavu celého klastru a plánování úloh. Pro použití v praxi není dán limit manažerských uzlů na jeden klastr, je však doporučeno volit lichý počet a maximálně sedm manažerských uzlů na klastr. Platí, že více manažerských uzlů neznamená lepší výkon, ba právě naopak. [19]

- **Pracovní uzly *(Worker nodes)***

Obecně řídící uzly slouží pouze k řízení činnosti kontejnerů, nepodílí se na komunikaci v rámci Raft ani nezajišťují plánování úloh. Sám o sobě nemůže řídící uzel existovat v klastru existovat, vždy zde musí být alespoň jeden manažerský uzel. Komunikuje pouze s manažerským uzlem, kterému hlásí aktuální stav nebo reporty o zadaných úlohách. Řídící uzly se dají vytvářet i při běhu klastru, případně lze pracovní uzly povýšit na uzly manažerské, to se může využít v případě, že některý z řídících uzlů vypoví službu. [14; 19]

- **Služby a úlohy *(Services and tasks)***

  Službou je v tomto kontextu myšleno provedení specifikované úlohy. Znamená to, že služba se skládá z jedné nebo množiny úloh.

  Úloha je definována jako konkrétní práce samotného kontejneru nebo kombinace kontejnerů. Úlohy jsou tedy jednotkami, které definují činnost. Pro uvedení příkladu existuje kontejner zajišťující generování dat a příkazem pro tento kontejner je neustálý běh pro zajištění kontinuálního generování, dohromady se jedná o úlohu.

Výhody Docker Swarm

V kontextu orchestrace Docker kontejnerů je swarm ideální volbou, protože je integrován do Docker prostředí, nevyžaduje tak další instalaci produktů z jiných stran.

Swarm umožňuje snadné nasazení aplikací, které jsou sestavovány pomocí nástroje Docker compose, ten zajišťuje nasazení definovaného počtu kontejnerů se všemi jejich závislostmi. V praxi lze tedy nasadit do swarmu celou aplikaci, která je definována v compose souboru, jediným příkazem.

Vytvoření swarm klastru se provádí prakticky jediným příkazem a není k němu potřeba používat nástroje pro vytváření klastrů ze třetích stran. Zároveň lze nové uzly k existujícímu klastru jednoduše připojit.

Nevýhody Docker Swarm

V porovnání s jinými orchestračním nástrojům má omezené funkcionality. Nejvýznamnějším nedostatkem je absence automatického škálování, lze jej ovšem definovat ručně jako službu, nebo pro dosažení automatického škálování využít produktů třetích stran. Docker swarm také nedisponuje vlastním webovým rozhraním pro ovládání klastru.

2.5.3 Porovnání Kubernetes vs Docker Swarm

- **Nasazení aplikace**

  Kubernetes – Nasazení aplikace do Kubernetes klastru je možné jako kombinaci služeb nebo mikroslužeb, kombinaci deploymentů a podů. Díky komplexnosti celého Kubernetes může nasazení a škálování kontejnerů být časově náročnější. [25]

  Docker Swarm – Aplikace se nasazují jako samostatné služby nebo mikroslužby do swarm klastru. Lze také aplikaci nasadit pomocí docker compose, který obsahuje definici pro všechny kontejnery a jeho závislosti. Samotná rychlost nasazení kontejnerů je značně vyšší a umožňuje i rychlejší škálování. [25]

- **Škálování a dostupnost**

  Kubernetes – Kubernetes umožňuje automatické škálování již v základu. Pody a jejich repliky jsou rozmístěny napříč všemi zařízeními v Kubernetes klastru, díky tomu jsou rychle dostupné. Zároveň díky automatizované detekci poškozených Podů dokáže rychle reagovat a vytvořit jejich nové repliky pro zajištění funkčnosti. [25]

  Docker Swarm – Swarm neumožňuje automatické škálování kontejnerů. Služby zde mohou být replikovány a tím zajišťují rychlou dostupnost. [25]

- **Instalace**

  Kubernetes – Kubernetes vyniká svojí komplexností, která se negativně projevuje již při instalaci samotného prostředí a následné konfigurace komponent.

  Docker Swarm – Je integrován do prostředí Docker, inicializace swarm klastru je triviální a připojování nových uzlů je také jednoduché.

- **Grafické rozhraní pro ovládání**

  Kubernetes – Kubernetes disponuje vlastním grafickým rozhraním jehož prostřednictvím lze ovládat činnost klastru.

  Docker Swarm – Pro ovládání swarm klastru prostřednictvím webového rozhraní je nutné použít nástroj, který je poskytován třetími stranami. Docker sám o sobě žádným takovým rozhraním nedisponuje.

Docker swarm lze použít pouze pro kontejnerizované aplikace prostřednictvím Docker. Je vhodné jej použít pro menší aplikace, nebo v případě, že je žádoucí orchestrace bez rozšířenějších funkcionalit. Vyniká především svojí jednoduchostí.

Kubernetes je oproti tomu vhodné použít v těch případech, kdy je potřeba zajistit orchestraci řádů stovek až tisíců kontejnerů. Značnou nevýhodou je náročnost konfigurace celého systému pro produkční prostředí. Kubernetes pro svoji definici používá vlastní kubectl, jehož syntax se liší oproti tradiční syntaxi příkazové řádky. Kubernetes vyniká svojí komplexitou a možnostmi, ale je náročnější ho pochopit a zvládnout.

2.5.4 Webové služby pro ovládání orchestračních nástrojů

Tato podkapitola popisuje vybrané nástroje, které poskytují webové grafické rozhraní, jehož prostřednictvím lze spravovat kontejnerizované aplikace. Tyto nástroje umožňují správu buď samotných kontejnerů, nebo celého klastru.

Portainer

Portainer je open-source projekt poskytující nástroj pro správu kontejnerizovaných aplikací pomocí grafického rozhraní. Na trhu má speciální místo, protože podporuje správu kontejnerizovaných aplikací bez ohledu na způsob jejich nasazení. Není omezen pouze na klasicky nasazené Docker kontejnerizované aplikace, ale umožňuje správu kontejnerizovaných aplikací prostřednictvím docker compose a aplikací nasazených ve Swarm nebo Kubernetes klastru.

Portainer poskytuje jednoduché řešení pro správu kontejnerizovaných aplikací, jeho pomocí lze pouštět, restartovat nebo zastavovat kontejnery, prohlížet záznamy kontejnerů, umožňuje spravování Swarm služeb, nebo dokáže měnit konfigurace jednotlivých kontejnerů. V základní verzi poskytuje omezené funkce, plná verze i s rozšiřitelnými funkcionalitami je zpoplatněná.

Kubernetes dashboard

Kubernetes disponuje vlastním grafickým rozhraním pro ovládání Kubernetes klastru. V základu není součástí nově vytvořeného Kubernetes klastru a je nutné jej do klastru přidat. Slouží k monitorování aktuálního stavu klastru a jednotlivých uzlů, lze jeho

prostřednictvím vytvářet nové deploymenty a spravovat škálování. Jeho prostřednictvím lze spravovat většinu funkcionalit, které kubectl nabízí.

Kubernetes dashboard se hodí pro začínající uživatele s kubernetes, protože umožňuje ovládání už tak komplexního Kubernetes prostřednictvím webové aplikace s grafickým rozhraním. Samo o sobě pokrývá široké spektrum funkcionalit, které kubectl nabízí. Pro správu Kubernetes klastru existují alternativy, mezi které patří již zmíněný Portainer, dále stojí za zmínku K8Dash a Octant. [21]

3 Cíle práce

Cílem práce je navrhnout architekturu a vytvořit funkční demoverzi vhodnou pro sběr dat ze senzorů a následnou predikci anomálií. Systém má být navržen tak, aby mohl být provozován na lokálním serveru nebo v cloudu. Při návrhu architektury je nutné se zaměřit na modulárnost celého systému, aby v případě potřeby mohly být jednoduše implementovatelné nové funkcionality nebo prediktory.

V rámci demoverze je nutné navrhnout generátor dat, který bude simulovat kontinuální data pro prediktor a zajistit vizualizaci příchozích dat včetně predikce anomálií.

Je potřeba zajistit orchestraci jednotlivých služeb systému, aby bylo možné celý systém řídit prostřednictvím webového API, a také jeho prostřednictvím přiřazovat hardwarové prostředky službám systému nebo umožnit změnu konfigurace jednotlivých služeb.

4 Návrh aplikace

Tato kapitola obsahuje analýzu požadavků vycházející ze zadání práce a konzultací během semestru. Vysvětluje návrh architektury systému a případů použití, vybrané části jsou vizualizovány prostřednictvím diagramů.

4.1 Funkční požadavky

Funkční požadavky vychází ze zadání, které se nachází na prvních stranách práce, jehož shrnuté cíle jsou v kapitole *3 Cíle práce*.

- Modulární predikční platforma nezávislá na konkrétní technologické platformě (python, .NET core)
- Běh systému v cloudu nebo lokálně na serveru
- Možnost přiřazovat a monitorovat výpočetní prostředky
- Možnost řídit nastavení systému a monitorování systému
- Možnost vizualizace vstupních dat a hodnot z prediktoru
- Systém bude snadno nasaditelný
- Umožnit modulárně přidávat nové prediktory
- Možnost konfigurace služeb při běhu systému
- Příchozí a predikovaná data je nutné ukládat

4.2 Případy užití

Vzhledem k nespecifikování požadavků na jednotlivé uživatelské role zde jako jediným aktérem vystupuje uživatel, který je schopen celý systém nasadit, spouštět, monitorovat a spravovat.

Nasazení systému

Jednoduché nasazení systému při splnění požadavků pro softwarové prerekvizity bude možné aplikovat na operační systémy Windows i distribuce Linux.

Spouštění systému

Celý navrhovaný systém bude po nasazení jednoduché spustit. Ideálně uvést všechny součásti systému do provozu jedním příkazem.

Konfigurace služeb

Uživatelé by měli mít možnost konfigurovat jednotlivé služby systému, buď před jejich spuštěním nebo při jejich běhu.

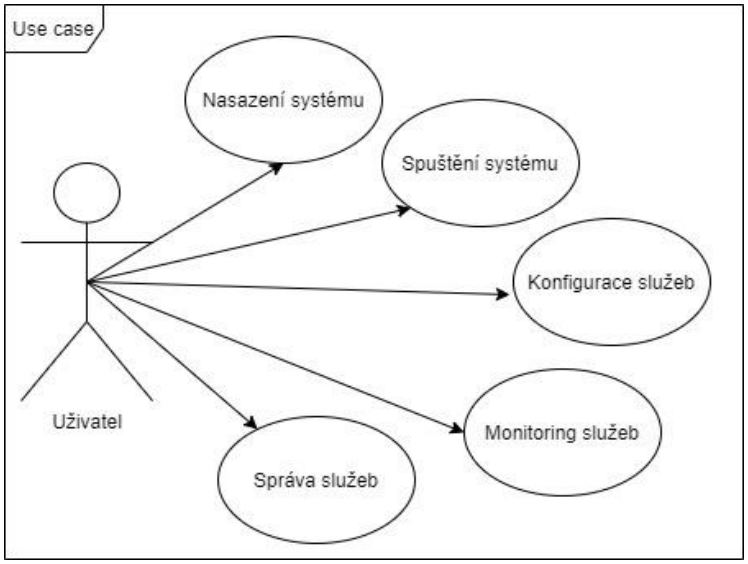
Monitoring služeb

Všechny služby systému by mělo být možné monitorovat a zároveň vizualizovat jak z hlediska jejich stavu, tak z hlediska jejich nároků na výpočetní prostředky.

Správa služeb

Služby systému musí být jednoduše spravovatelné. Jedná se tak o spouštění nebo zastavování služeb, jejich konfigurace nebo přidělování výpočetních prostředků.

Diagram případů užití



Obrázek 12 Use Case diagram

4.3 Řešení požadavků

Řešení požadavků popisuje koncepty a technologie pro implementaci, které vyplívají na základě analýzy funkčních požadavků.

- **Mikroslužbová architektura**

Z hlediska volby softwarové architektury bude na základě zadání vybrána mikroslužbová architektura, která je vhodná vzhledem k požadavkům na na modularitu, orchestraci a na možnost provozu systému na serveru nebo v cloudu.

- **Použití kontejnerizace pro služby**

Pro zajištění izolace jednotlivých služeb, bude využitá kontejnerizace, která je zároveň vhodným řešením pro systém z hlediska modularity, izolace a jednoduchosti nasazení systému.

- **Databáze optimalizovaná pro časové řady**

Vstupními daty pro prediktor jsou časové řady, které lze definovat jako chronologicky uspořádaná data. Z povahy vstupních dat vyplívá, že bude nutné zvolit vhodnou databázi, která je žádoucí pro časové řady a je velmi rychlá, vstupní data mohou proudit v řádech milisekund.

- **Standardizování komunikace**

Vzhledem k povaze vstupních dat je nutné zvolit vhodný jednotný datový formát pro streamování dat mezi komponentami systému.

- **Vizualizace dat a monitoring**

Jedním z požadavkům je možnost vizualizace dat z databáze, vizualizace využitých výpočetních prostředků a ovládání prostřednictvím grafického rozhraní.

- **Simulace dat ze senzorů a prediktor**

Pro simulaci kontinuálních příchozích dat v reálném čase bude vytvořen jednoduchý konfigurovatelný generátor dat, který bude představovat příchozí data ze senzorů. Pro predikci bude vytvořen triviální prediktor, který bude přijímat zpracovávat a predikovat data. Návrh kvalitního prediktoru nebo sběr dat z reálných senzorů není předmětem této práce.

- **Záloha dat ze senzorů a prediktoru**

Simulovaná data ze senzorů je potřeba ukládat do databáze pro budoucí tvorbu prediktorů. Množina sesbíraných dat poté může posloužit jako trénovací data pro učení prediktoru.

4.4 Diagram návrhu komponent

Na základě funkčních požadavků byl vytvořen diagram, který znázorňuje propojení komponent. Navrhovaný systém tak byl rozdělen na hlavní komponenty správa služeb a zpracování dat.

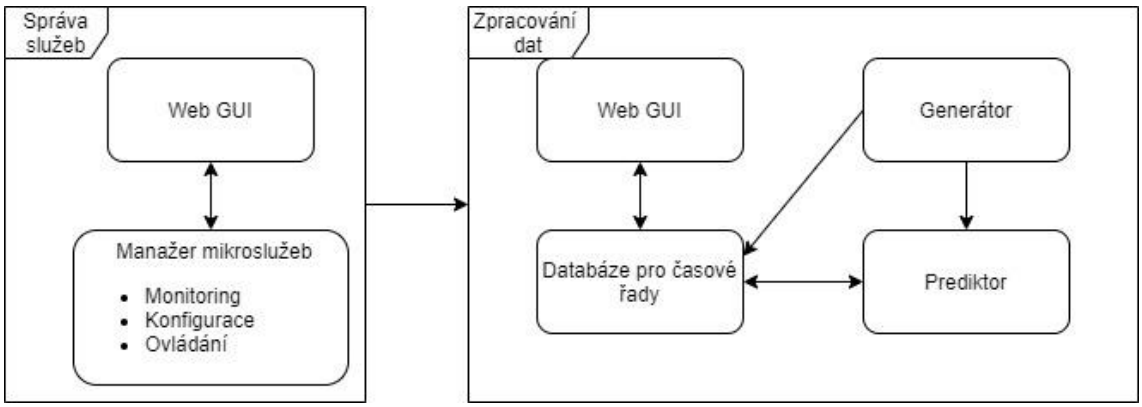
Správa služeb představuje samotný nástroj pro orchestraci mikroslužeb a webové grafické rozhraní, kterým lze nástroj ovládat.

- **Web GUI** – Grafické rozhraní poslouží pro ovládání orchestrátoru. Jehož prostřednictvím bude možné pouštět, restartovat a zastavovat služby, konfigurovat služby a spravovat výpočetní prostředky.
- **Manažer mikroslužeb** – Konkrétní nástroj pro orchestraci mikroslužeb, který bude možné ovládat jak přes příkazovou řádku, tak přes webové grafické rozhraní.

Zpracování dat obsahuje služby pro simulování reálných dat, prediktor, databázi pro časové řady a grafické webové rozhraní určené pro monitoring a správu databáze.

- **Generátor –** Počítá se s tím, že generátor bude komunikovat nejen s prediktorem, ale i s databází, kvůli ukládání příchozích dat pro učení prediktoru. Hlavní úlohou generátoru je konfigurovatelná simulace dat v reálném čase.
- **Prediktor** – Triviální konfigurovatelná predikce se vytváří na základě dat poskytnutých generátorem nebo dat z databáze.
- **Databáze pro časové řady** – Databáze slouží k ukládání časových řad z generátoru i prediktoru dat.

Web GUI – Grafické rozhraní pro vizualizaci dat z databáze.



Obrázek 13 Návrh komponent

5 Implementace

Tato kapitola popisuje technologie, které byly použité pro implementaci architektury navrhovaného systému. Obsahuje také nový přepracovaný diagram komponent, nyní již jako evoluci návrhového diagramu se zakomponováním použitých konkrétních technologií. Poté jsou jednotlivé komponenty systému a jejich funkcionalita obecně shrnuty, v případě nejasností se lze podívat na samotný kód, který je komentovaný. Pro implementaci je využito objektově orientovaného programování za účelem jednoznačného oddělení funkčních celků, jednodušší implementaci nových funkcionalit a obecně pro větší přehlednost.

5.1 Použité technologie

Následující technologie byly vybrány na základě analýzy požadavků systému. Při výběru byl kladen důraz na jednoduchost a tvrzení „nevymýšlet kolo“, z toho důvodu se vybraly některé existující, spolehlivé nástroje.

Python

Python je open-source vyšší skriptovací programovací jazyk, jehož první verze se objevila již v roce 1991. Aktuálně je velmi rozšířený a poskytuje obrovské množství knihoven, které jsou předpokladem pro kvalitní zpracování dat nebo tvorbu informačních systémů.

V rámci zadání této práce nebyl specifikován jazyk a kvůli zkušenostem autora a existenci kvalitních knihoven určených pro zpracování dat, byl tedy zvolen jako primární jazyk právě Python.

Docker

Vzhledem k volbě mikroslužbové architektury pro navrhovaný systém je vhodné každou službu systému izolovat a pevně definovat případnou komunikaci s ostatními službami. Z těchto důvodu se využije principu kontejnerizace a jako technologická platforma je vybrán Docker.

Pro automatizované nasazení celé aplikace byl použit nástroj docker-compose, který je součástí Docker ekosystému. Jeho prostřednictvím bude docházet k automatizovanému nasazení celého systému naráz a bude obsahovat veškeré závislosti.

Z ekosystému Docker byl použit ještě nástroj umožňující orchestraci kontejnerizovaných aplikací. Aplikace tedy bude běžet jak v podobě samostatných kontejnerů, tak i ve swarm klastru, pro zajištění automatické opravy služeb, možnosti konfigurace a přidělování výpočetních prostředků.

InfluxDB TICK Stack

InfluxData je vývojářská společnost, která stojí za vznikem databáze a řady dalších open-source nástrojů, které usnadňují práci s časovými řadami. V rámci jejich portfolia je nejznámějším produktem InfluxDB databáze, která se na aktuálním trhu řadí mezi nejuniverzálnější a mezi vývojáři k nejpopulárnější databázi určené pro časové řady. [26]. V rámci této práce bude použit jejich InfluxDB TICK stack (dále jako Influx balíček), což je balík služeb, které slouží k ukládání, upozorňování a vizualizaci dat. Součástmi balíčku jsou následující komponenty:

- InfluxDB databáze optimalizovaná pro časové řady, zároveň komunikuje s dalšími součástmi Influx balíčku a poskytuje jim data.
- Kapacitor, který je určen primárně k vytváření událostí a dále umožňuje v reakci na tyto události streamování dat a spouštění skriptů.
- Chronograf je webovým grafickým rozhraním pro monitoring a konfiguraci některých dostupných služeb v Influx balíčku.
- Telegraf je zodpovědný za sběr dat a komunikaci s databází.

Portainer

Pro monitorování stavu nasazené aplikace, pouštění, restartování nebo zastavování kontejnerů nebo pro změnu konfigurace prostřednictvím grafického rozhraní, bude využit open-source nástroj Portainer. Jeho prostřednictvím bude možné spravovat jak automatizovaně sestavené kontejnery přes docker-compose, tak kontejnery nasazené ve swarm klastru.

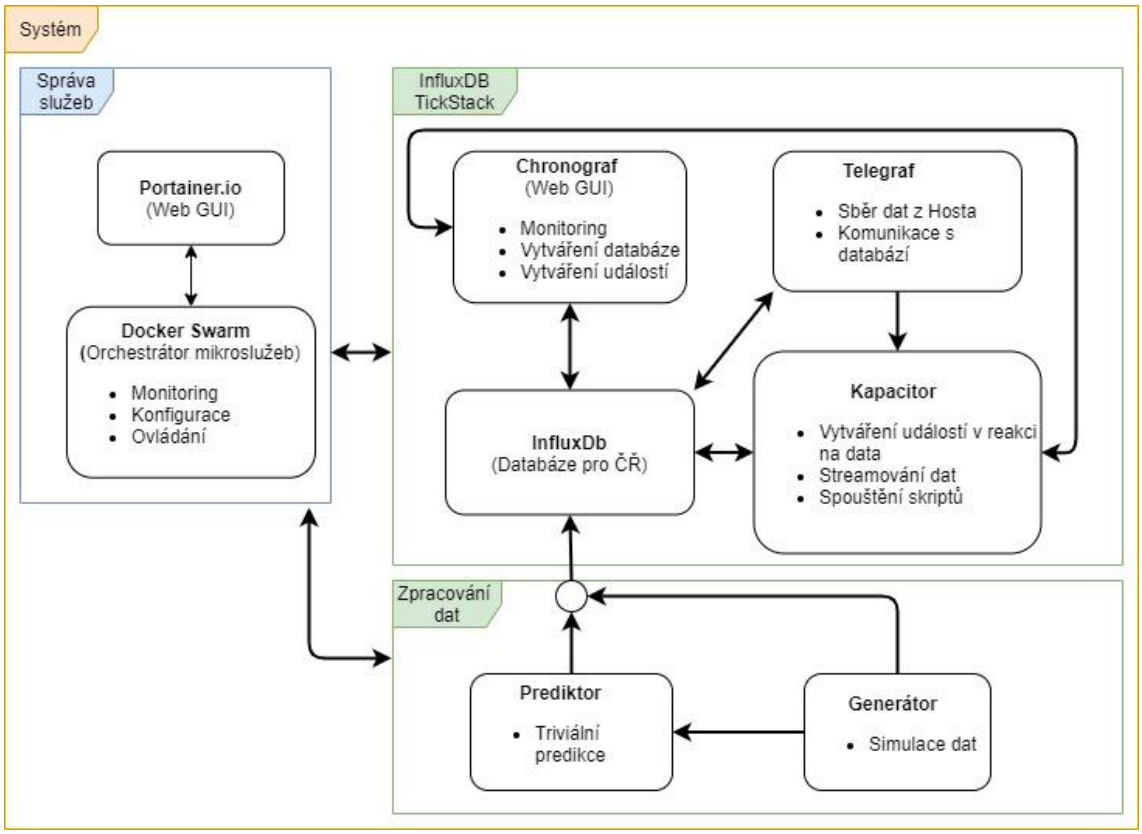
5.2 Architektura řešení

Konkrétní architektura řešení vzešla z diagramu, který znázorňuje návrh komponent systému, a z vybraných technologií pro jejich realizaci. Výsledná architektura je tedy založena na původním návrhu, ale v tomto bodě již obsahuje konkrétní použité nástroje a jejich popis.

- **InfluxDB TickStack**

Pro ukládání časových řad, zaznamenávání a reakci na události, sběr dat z hostitelského zařízení a pro vizualizaci dat prostřednictvím webového grafického rozhraní byl využit Influx balíček. Komunikace mezi jednotlivými službami je pevně definována strukturou oficiální verze od InfluxData. Úlohy jednotlivých služeb balíčku jsou obecně shrnuty v kapitole popisující použité technologie.

Tento balíček obousměrně komunikuje s komponentou správa služeb a vůči komponentě zpracování dat je komunikace jednosměrná. Služby z komponenty zpracování dat komunikují pouze s databází, do které ukládají generovaná a predikovaná data.



Obrázek 14 Architektura řešení

- **Zpracování dat**

V rámci komponenty zpracování dat je zajištěna jednosměrná komunikace mezi službami, kdy generátor zasílá data prediktoru. Toto zasílání je řešeno prostřednictvím sdílené linuxové fronty, do které má generátor práva pro zápis a prediktor práva pouze pro čtení. Zároveň obě služby komunikují jednosměrně s InfluxDB, které zasílají data. Jediná obousměrná komunikace je s komponentou správa služeb. Pro zasílání dat byl zvolen formát JSON, ale implementace služeb umožňuje kdykoliv přidat funkcionalitu, která bude zahrnovat i jiné formáty.

- **Správa služeb**

Komponenta správa služeb je tvořena webovým aplikačním rozhraním s grafickou nadstavbou pro orchestraci služeb. Zároveň je zde zahrnut také orchestrátor, který lze ovládat právě prostřednictvím webového rozhraní. Tato komponenta komunikuje obousměrně s veškerými službami systému a umožňuje jejich správu.

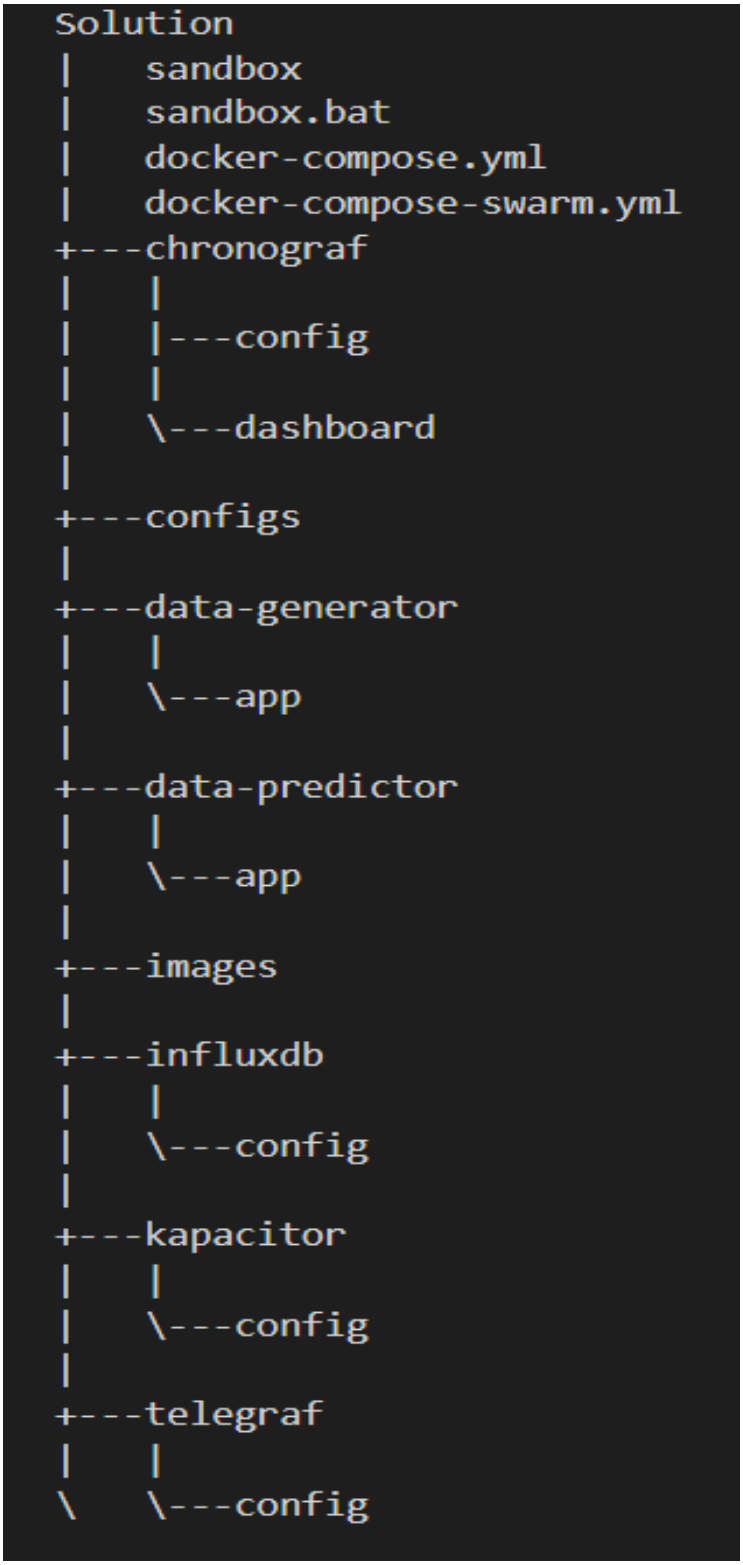
5.3 Struktura řešení

V této části je popis adresářové struktury řešení *(Obrázek č. 16 Adresářová struktura řešení)*, pro zjednodušení byly vybrány pouze stěžejní adresáře.

- V kořenovém adresáři se vyskytují skripty pro spuštění, docker-compose soubory pro nasazení a spuštění aplikace zvlášť pro Windows a Ubuntu.
- Adresář **chronograf** obsahuje podadresář dashboard, ve kterém se nachází exportované přednastavené zobrazení grafů pro chronograf.
- **Configs** adresář slouží k uchovávání konfiguračních souborů pro prediktor a generátor.
- Adresář **data-generator** obsahuje Dockerfile pro vytváření obrazu generátoru, a obsahuje podadresář app, ve kterém je celá aplikační logika generátoru.
- Adresář **data-predictor** obsahuje Dockerfile pro vytváření obrazu prediktoru, a obsahuje podadresář app, ve kterém je celá aplikační logika generátoru.
- Adresáře **influxfdb**, **kapacitor** a **telegraf** slouží k zajištění perzistence dat mezi běžícím kontejnerem a hostujícím zařízením. Zároveň mají

podadresář config, ve kterém jsou konfigurační soubory pro jednotlivé služby.

- Adresář **images** obsahuje definici verzí obrazů pro Influx balíček.



Obrázek 15 Adresářová struktura řešení

5.4 Postup implementace

Na základě analýzy požadavků byly vybrány technologické platformy, zde je třeba zohlednit, že se použilo několik již existujících nástrojů a byly upraveny pro vyvíjený systém.

5.4.1 Implementace služeb

Generátor

Pro simulování kontinuálních dat v reálném čase byl vytvořen generátor, který posílá data do databáze a zároveň je zasílá do linuxové fronty, kde data čekají na vyzvednutí prediktorem.

Třídy a metody generátoru jsou navrženy tak, aby vytvářeli co nejobecnější prostředí a změny jeho chování byly řízeny pouze konfiguračním souborem. Pro uvedení příkladu takové obecné metody je dobré zmínit část kódu právě pro načítání konfiguračních souborů. Na základě koncovky vstupního souboru, který může být buď ve formátu JSON nebo YAML, je zadaný soubor řádně načten do objektu, ke kterému se už snadno přistupuje jako k listu textových řetězců. Vzorek kódu reprezentující třídu pro načítání konfiguračních souborů je možné vidět na obrázku (*Kód 1 Ukázka Třídy pro načtení konfigurace)*. Mimo samotného generování a čtení dat ze souborů také ještě generátor posílá data do databáze v Influx balíčku a zařazuje je do linuxové fronty. V jednom kroku poté zašle data jak prostřednictvím aplikačního rozhraní databáze, tak je přidá i do fronty.

Pro frontu byla použita linuxová fronta *(FIFO)*, která funguje na principu pojmenované pajpy *(Named Pipe)*. Z důvodu, že jsou kontejnery od sebe izolované bylo nutné vytvořit komunikaci, která je v souladu s pravidly dockeru a zároveň neodporuje principům mikroslužeb. Z toho důvodu byly využity docker volumes, které zajišťují definované sdílení dat.

Data, která generátor produkuje jsou buď náhodná data, která se nahrávají do souboru, nebo existující data z konkrétního souboru. Tyto data jsou čistě pouze hodnotami, pro jejich zaslání ve standardizovaném formátu je použitá třída, která zpracuje hodnotu včetně časové známky a vytvoří JSON objekt, který je pak nosičem časových řad. Vytváření JSON objektu je aktuálně částečně konfigurovatelné, ale je vše připraveno pro implementaci dalších funkcionalit.

```python
class Configuration(object):

    def load_config(self,filepath=None):
        filepath = filepath or "config/config.yaml"
        file, ext = os.path.splitext(filepath)
        if ext==".yaml":
            return self.load_config_yaml(filepath)
        elif ext==".json":
            return self.load_config_json(filepath)
        else:
            return None

    def load_config_json(self,filepath):
        with open(filepath,'r',encoding="utf-8") as jsonfile:
            config=json.load(jsonfile)
        return config

    def load_config_yaml(self,filepath):
        with open(filepath,'r',encoding="utf-8") as yamlfile:
            config=yaml.load(yamlfile,yaml.FullLoader)
        return config
```

Kód 1 Ukázka Třídy pro načtení konfigurace

Struktura generátoru

Obrázek k adresářové struktuře generátoru se nachází na další straně *(Kód 2 Adresářová struktura prediktoru)*. Adresář data-generátor obsahuje dockerfile, který slouží k sestavení jeho obrazu včetně všech jeho závislostí.

V jeho podadresáři `/app` jsou poté skripty, které obsahují veškerou aplikační logiku generátoru. Zároveň je zde skript pro shell, který definuje spuštění generátoru s předpřipraveným argumentem, který určuje cestu k základnímu konfiguračnímu souboru. Nachází se zde ještě textový soubor, ve kterém jsou názvy všech potřebných python knihoven pro běh generátoru.

```
+ - - -data-generator
|   |   Dockerfile
|   |
|   + - - -app
|   |   |   main.py
|   |   |   requirements.txt
|   |   |   startscript.sh
|   |   |   tasks.py
|   |   |   __init__.py
|   |   |
|   |   \---data
|   |   |       generated.txt
```

Kód 2 Adresářová struktura generátoru

Shrnutí

Vstupem generátoru je datový a konfigurační soubor, ve kterém je specifikováno připojení k databázi, generování dat a možnost zvolit, zdali se data budou posílat do prediktoru prostřednictvím fronty.

Výstupem generátoru jsou časové řady, které vždy obsahují časovou známku a jednu měřenou veličinu. Výstup se ukládá do databáze a přidává se do fronty pro prediktor.

Prediktor

Návrh a implementace kvalitního prediktoru není součástí zadání této diplomové práce. Ovšem nutno říct, že kvalitní prediktor by sám o sobě obstál samostatné diplomové práci a je vhodným námětem pro budoucí studenty. Pro simulaci prediktoru a zakomponování do návrhu architektury systému byla implementována služba, která přijímá příchozí data, předzpracovává je pro matematickou operaci představující predikci a výsledek odesílá do databáze. Příchozí a odchozí komunikace probíhá ve standardizovaném formátu JSON.

Veškerá aplikační logika implementovaného prediktoru je opět navržena tak, aby byla co nejvíce obecná a podporovala rozmanitou konfiguraci. Některé funkcionality, které byly již zmíněny u generátoru, byly implementovány současně i do prediktoru. Pro

uvedení příkladu lze zmínit logiku pro načítání konfigurace. Kvůli předpokladu, že může existovat několik různých prediktorů, které by mohly být naráz spuštěny v rámci jednoho obrazu, byla logika konkrétní predikce izolována mimo kořenový adresář. Vhodnějším řešením by byla kompletně oddělená implementace prediktorů čili každý prediktor by měl vlastní obraz a byl zcela oddělený od ostatních prediktorů. Pro tento případ je předpokladem dodržení pravidel pro komunikaci v jednom formátu, čtení z fronty a odesílání do databáze.

Implementovaný prediktor přijímá data ve frontě z adresáře, který sdílí společně s generátorem. Díky použití linuxové fronty lze dosáhnout velmi rychlé komunikace. To je vhodné pro případy, kdy data budou proudit v řádech milisekund. Data z frontě jsou přijímána ve formě textového řetězce splňující formát JSON. Tato data jsou poté předzpracována do formátu, ve kterém bude s daty možné pracovat jako s hodnotami, následně jsou odesílány do konkrétního predikčního skriptu. Tento skript násobí příchozí hodnotu násobkem definovaným v konfiguračním souboru a následně hodnotu včetně časové známky převádí do objektu formátu JSON a nahrává jej do databáze.

```
+ - - -data-predictor
|   |   Dockerfile
|   |
|   \- - -app
|       |   main.py
|       |   requirements.txt
|       |   startscript.sh
|       |   tasks.py
|       |   __init__.py
|       |
|       \- - -predictors
|       |      predictor1.py
|       |      __init__.py
```

Kód 3 Adresářová struktura prediktoru

Struktura prediktoru

Obrázek popisující adresářovou strukturu prediktoru lze nalézt na *Kód 3 Adresářová struktura prediktoru*. Adresář data-predictor obsahuje dockerfile, který slouží k sestavení jeho obrazu včetně všech jeho závislostí.

Podadresář `/app` obsahuje skripty, které obsahují veškerou aplikační logiku prediktoru a skript v shellu, který inicializuje základní spuštění celého prediktoru i s argumentem definující cestu ke konkrétnímu konfiguračnímu souboru. Stejně jako u generátoru, je i zde soubor definující všechny potřebné python knihovny pro spolehlivý běh prediktoru. Podadresář /predictors je místem pro implementaci různých prediktorů. Aktuálně se zde vyskytuje pouze jeden prediktor, jehož funkčnost byla popsána výše.

Shrnutí

Vstupem prediktoru jsou data z linuxové fronty, která je sdílená s generátorem, a vstupem je konfigurační soubor, ve kterém jsou informace o připojení k databázi, a konfigurace samotného prediktoru – v tomto případě hodnota pro násobení.

Výstupem prediktoru jsou časové řady, které se posílají do databáze Influx balíčku.

Úprava InfluxDB Tick stack

Influx balíček je volně ke stažení přímo od Influx Data a obsahuje všechny potřebné závislosti pro jeho provoz. Bylo tedy nutné nastudovat celý balíček a správně jej propojit s komponentami pro zpracování dat a správu služeb. Jedinou změnou je definice verze InfluxDB databáze. V době implementace architektury systému byla poslední verze této databáze označená jako 1.7.11, v současné době byla vydána novější verze 2.0, která v systému nebyla použita především z toho důvodu, že se s ní v původním návrhu nepočítalo a kvůli množství změn oproti minulé verzi. Pro komunikaci s databází byla použita knihovna přímo od Influx Data určená pro python, kompatibilní s vybranou verzí databáze.

5.4.2 Kontejnerizace služeb

Kontejnery se vytvářejí na základě definovaných obrazů, které lze získat prostřednictvím globálního nebo lokálního repozitáře obrazů, nebo je lze sestavit přímo na lokálním zařízení. Služby z Influx balíčku jsou dostupné na DockerHub čili jejich obrazy není potřeba lokálně ukládat, ale postačí vybrat jejich verzi. Ostatní služby

systému, mezi které patří generátor a prediktor, mají dostupný obraz taktéž na DockerHub, ale kvůli vlastní implementaci těchto služeb jsou pro nasazení prostřednictvím docker-compose sestavovány lokálně.

Definice lokálních obrazů

Vytváření lokálních obrazů se týká pouze generátoru a prediktoru. Jejich obrazy se vytvářejí na základě shodných Dockerfilů (Docker soubory obsahující všechny závislosti pro sestavení obrazů).

```
FROM python:3

# set the working directory in the container
WORKDIR /app
# copy all local files from app
COPY /app /app
# install python libraries
RUN python3 -m pip install -r requirements.txt
# run start script inside container
CMD [ "sh", "./startscript.sh" ]
```

Obrázek 16 Ukázka Dockerfile pro prediktor/generátor

Obrazy pro tyto služby se vytvářejí v prostředí python, kvůli použitému jazyku při implementaci služeb a jednoduší následné instalaci knihoven pro python. Zároveň je zde definována kořenová složka obrazu `/app`, která obsahuje všechny skripty ve stejnojmenné složce na lokálním úložišti. Další příkaz slouží ke spuštění instalace python knihoven potřebných pro běh skriptů a posledním příkazem je vykonání shell skriptu, ve kterém je předdefinováno spuštění služby.

Definice obrazů z repozitáře Docker Hub

Obrazy pro ostatní služby systému jsou dostupné na globálním docker repozitáři Docker Hub, pro vytvoření instance kontejneru je potřeba lokálně specifikovat pouze konkrétní verzi, která je v základu pro většinu služeb nastavena na poslední vydanou verzi. Jediná služba, která se svojí verzí liší je databáze InfluxDB, byla pro ni vybrána verze 1.7.11.

Definice pro docker-compose

Veškeré služby a jejich závislosti jsou definované v souboru „docker-compose.yml“ podle pravidel verze 3.0. Soubor se nachází v kořenovém adresáři systému.

- **Definice ostatních závislostí**

Služby Influx balíčku mají nastavené závislosti podle oficiální verze. Mezi tyto závislosti patří vystavování portů a vymezení síťových přístupů pro komunikaci s ostatními službami. Pro služby je nastaveno pořadí, ve kterém se kontejnery nasazují a spouští. Některé služby totiž čekají s vlastním spuštěním v závislosti na běhu jiné služby.

- **Komunikace mezi kontejnery a sdílení dat**

Pro sdílení dat mezi službami je využito Docker volumes (Obsahy) a mezi komponentami Influx balíčku je komunikace navíc zajištěná aplikačním rozhraním jednotlivých komponent (Síťová komunikace).

Volumes sdílení probíhá buď ve vztahu Host – Kontejner nebo Kontejner – Kontejner. Příkladem vztahu Host - Kontejner je adresář pro konfigurační soubory, ty jsou dostupné vždy na hostitelském zařízení a umožňují definování nových konfiguračních souborů ještě před samotným sestavením obrazů a vytvořením kontejnerů. Speciálním případem je sdílení ve vztahu Kontejner – Kontejner, které slouží konkrétně ke komunikaci generátoru a prediktoru. Generátor zasílá data do linuxové fronty (FIFO) do adresáře `/fifo/dataIN` , který je součástí jeho kontejneru, následně tento adresář sdílí s prediktorem, který jej má taktéž ve stejném adresáři v rámci vlastního kontejneru. Použití fronty umožňuje velmi rychlý přenos dat, který je žádoucí pro případné produkční řešení, kdy mohou data přicházet v řádech milisekund.

Perzistence dat

Z povahy kontejnerů je zřejmé, že vzniklá data během činnosti kontejnerů zanikají spolu s kontejnerem. Pro zajištění perzistence dat je tak nutné nastavit sdílení dat z kontejnerů na hostitelské zařízení. Takto jsou ukládána data z databáze a nastavení chronografu.

5.4.3 Orchestrace služeb

Pro zajištění orchestrace všech služeb byl použit nástroj Docker swarm, který je součástí Docker ekosystému. V prvé řadě je inicializován swarm klastr a poté jsou všechny služby do něj připojeny. Pro nasazení služeb do klastru se v příkazovém řádku používá příkaz docker-stack, jehož prostřednictvím je možné nasadit kontejnery, které jsou definovány v klasickém docker-compose souboru. Použitím compose souboru odpadá nutnost definovat odlišný způsob nasazení. Pokud by byl použit původní docker-compose soubor, který standardně slouží k automatizovanému nasazení samostatných kontejnerů, tak se budou ignorovat závislosti, které nejsou v rámci swarm klastru podporovány. Mezi tyto nepodporované závislosti patří například sestavení obrazů, proto se musí definovat existující obrazy nacházející se v lokálním nebo globálním repozitáři. Pro zajištění větší přehlednosti byl vytvořen nový docker-compose-swarm.yml soubor, který se nachází v kořenovém adresáři řešení a obsahuje pouze potřebné závislosti pro spolehlivé nasazení kontejnerů do klastru. Orchestrátor lze ovládat buď napřímo z příkazové řádky nebo pomocí webového grafického rozhraní Portainer, který podporuje veškeré potřebné funkcionality, které tento systém vyžaduje.

5.4.4 Nasazení a spouštění systému

Jednoduché nasazení a spouštění systému obstarávají spustitelné soubory, které jsou jak pro prostředí Windows, tak pro distribuce Linux. Tyto soubory přijímají vstupní argument/y a na jeho základě provedou požadovanou činnost. Obsahují legendu pro jednotlivé vstupní argumenty prostřednictvím tradičního argumentu help.

```
up            -^> Nasazení kontejnerů nástrojem docker-compose
                  a spuštění systému.
down          -^> Zrušení všech kontejnerů systému. Vypnutí
systému.
upswarm       -^> Inicializace swarm klastru a nasazení
                  kontejnerů do klastru.
downswarm     -^> Odstranění kontejnerů z klastru a vypnutí
                  klastru
restart       -^> Restart systému (jen pro compose)
influxdb      -^> InfluxDB příkazová řádka

#Následující příkazy fungují jen v sestavení compose.
enter <název kontejneru> -^> vstup do vybraného kontejneru
log <název kontejneru>   -^> logy vybraných kontejnerů (jen
                             pro Influx balíček)
delete-data  -^> Smazání všech dat, které činností balíčku
                  vznikly.
docker-clean -^> Zastaví kontejnery a vymaže všechny docker
                  obrazy.
```

Obrázek 17 Legenda k ovládání systému

Pro nasazení a spuštění systému v operačním systému Windows je určen spustitelný soubor „sandbox.bat“ a spouští se příkazem: `./sandbox.bat <arg1> <arg2>`

Pro nasazení a spuštění systému v operačním systému s Linuxovým jádrem je určen spustitelný skript „sandbox.sh“ a spouští se příkazem: `./sandbox <arg1> <arg2>`

5.5 Testování Implementace

Pro systém nebyly vytvořeny specifické testy jednotlivých tříd nebo metod, ale byla otestována veškerá funkčnost jednotlivých komponent. V prvé řadě se během vývoje testovaly komponenty v Linuxového prostředí s pomocí testovacích konfiguračních souborů a dat. Po otestování funkčnosti samotných služeb komponent se přešlo ke kontejnerizaci těchto služeb a následné otestování vzájemné interakce s ostatními kontejnery.

Systém byl otestován pro operační systém Windows při použití Docker desktop, a byl otestován pro distribuci Linuxu Ubuntu 20.04. Při testování se kladl důraz na samotnou funkčnost systému i při změnách konfiguračních souborů.

6 Uživatelská dokumentace

Tato kapitola popisuje požadavky a postup pro nasazení systému ve vybraných prostředích. Zároveň je zde vysvětleno spouštění a správa celého systému včetně orchestrace.

6.1 Nasazení a spouštění ve Windows 10

Pro otestování demoverze architektury systému pro operační systém Windows 10 je nutné nainstalovat následující prerekvizity:

- Windows 10 (Home/Professional)
- WSL 2 (dostupné z: https://docs.microsoft.com/en-us/windows/wsl/install-win10)
- Docker Desktop (dostupné z: https://docs.docker.com/docker-for-windows/install/)
- Verzovací systém GIT (dostupné z: https://git-scm.com/downloads)
- Samotný systém, který je předmětem této práce (dostupné z: https://github.com/jedimik/Diplomova_prace)

Po instalaci výše zmíněných prerekvizit a vybrání WSL 2 za hlavní distribuci WSL se lze uchýlit ke stažení samotného systému. Systém je nahraný na repozitáři Github a díky nainstalovanému gitu, který je uveden jako prerekvizita, je možné jednoduše naklonovat obsah celého repozitáře.

```
git clone https://github.com/jedimik/Diplomova_prace.git
```

Kód 4 Naklonování repozitáře

Po vstupu do adresáře, který obsahuje veškerý kód aplikace, je možné aplikaci nasadit pomocí příkazové řádky. Jednotlivé možnosti spuštění a práce s aplikací pomocí příkazové řádky jsou uvedeny v kapitole 5.4.4 Nasazení a spouštění systému . Pro nasazení a spuštění je nutné, aby cesta v příkazové řádce korespondovala s cestou kořenové adresáře stáhnutého systému. Poté lze systém spustit jako automatizované nasazení kontejnerů, nebo jako vytvoření swarm klastru a následné připojení systému do klastru.

```
#Pro compose, automatizované nasazení kontejnerů
./sandbox.bat up
#Pro vytvoření swarm klastru a připojení systému
./sandbox.bat upswarm
```

Kód 5 Spouštění systému ve Windows

Po spuštění se automaticky otevřou ve webovém prohlížeči dvě okna, každé z nich bude v lokální síti, ale s odlišným portem. Adresa localhost:8888 představuje webové rozhraní chronografu, který slouží k monitoringu dat, a adresa localhost:9000 představuje webové rozhraní pro portainer, kterým lze spravovat nasazený systém.

Kde najít jednotlivé služby

Spuštění ./sandbox.bat up a po swarm

6.2 Nasazení a spouštění na Ubuntu

Pro otestování demoverze architektury systému pro distribuci Linuxu Ubuntu je nutné nainstalovat následující prerekvizity:

- Ubuntu 20.04
- Aplikační balíčky (python3, docker, git)
- Samotný systém, který je předmětem této práce (dostupné z: https://github.com/jedimik/Diplomova_prace)

Po instalaci výše zmíněných prerekvizit je možné stáhnout/naklonovat systém z repozitáře github.

```
git clone https://github.com/jedimik/Diplomova_prace.git
```

Kód 6 Naklonování repozitáře

Po vstupu do adresáře, který obsahuje veškerý kód aplikace, je možné aplikaci nasadit pomocí příkazu. Jednotlivé možnosti spuštění a práce s aplikací pomocí příkazů jsou uvedeny v kapitole 5.4.4 Nasazení a spouštění systému . Pro nasazení a spuštění je nutné, aby uživatel byl přímo v kořenovém adresáři naklonovaného systému. Poté lze systém spustit jako automatizované nasazení kontejnerů, nebo jako vytvoření swarm

klastru a následné připojení systému do klastru. Před samotným prvotním spuštěním je ještě potřeba jednorázově přiřadit spouštějícímu skriptu práva pro exekuci.

```
#Povolení exekuce skriptu
sudo chmod +x sandbox
#Pro compose, automatizované nasazení kontejnerů
./sandbox up
#Pro vytvoření swarm klastru a připojení systému
./sandbox upswarm
```

Kód 7 Spouštění systému v Ubuntu

Pokud distribuce Ubuntu disponuje grafickým rozhraním, tak se po spuštění automaticky otevřou ve webovém prohlížeči dvě okna, každé z nich bude v lokální síti, ale s odlišným portem. Pokud je Ubuntu provozováno přes WSL2, pak k adresám lze přistoupit v hostitelském zařízení, pokud je Ubuntu externě nebo ve virtuálním stroji, tak je nutné vystavit síť pro přístup zvenčí. Adresa localhost:8888 představuje webové rozhraní chronografu, který slouží k monitoringu dat, a adresa localhost:9000 představuje webové rozhraní pro portainer, kterým lze spravovat nasazený systém.

6.3 První spouštění a konfigurace

Předpokladem pro práci s nástroji je nasazení a první spuštění systému. Bez ohledu na vybraný přístup buď přes docker-compose, nebo jako balíček připojený do swarm klastru, je nutné některé služby konfigurovat. Po spuštění systému se automaticky otevřou webová grafická rozhraní pro chronograf (localhost:8888) a portainer (localhost:9000).

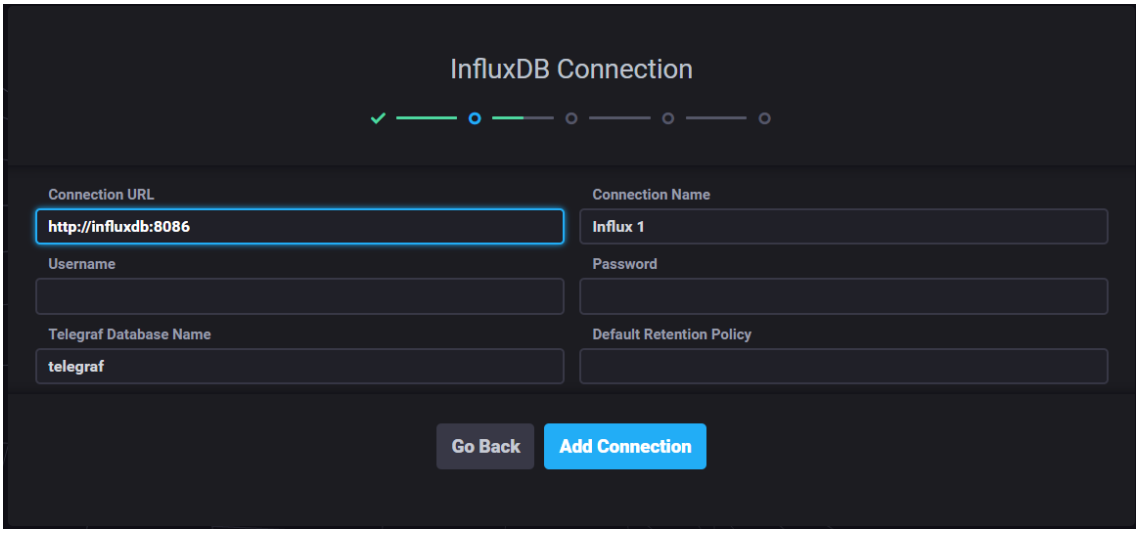
6.3.1 Chronograf

Při prvním spuštění je potřeba konfigurovat připojení k vybrané databázi, výběr základního dashboardu a připojení ke kapacitoru. V prvním kroku se definuje připojení k databázi InfluxDB, která standardně naslouchá na portu 8086, čili se chronograf bude připojovat na toto umístění. Přihlašovací jméno a heslo není v prvotním nastavení požadováno, protože nebyli vytvořeni noví uživatelé pro databázi. (Url: influxdb:8086)

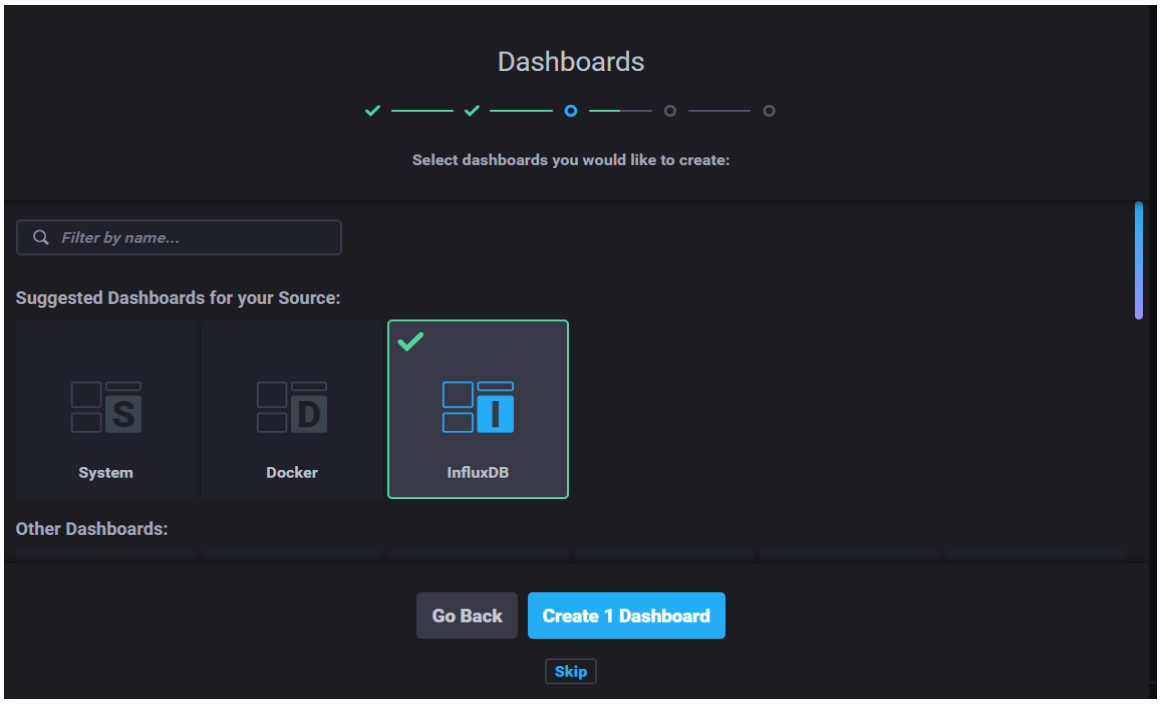
V druhém kroku je možné vybrat dashboard pro konkrétní zdroj či zdroje dat, nebo lze tento krok přeskočit. V základu jsou zde tři možnosti, Systém pro monitoring

výpočetních požadavků, Docker pro monitoring kontejnerů a InfluxDB pro monitoring dat příchozích do databáze.

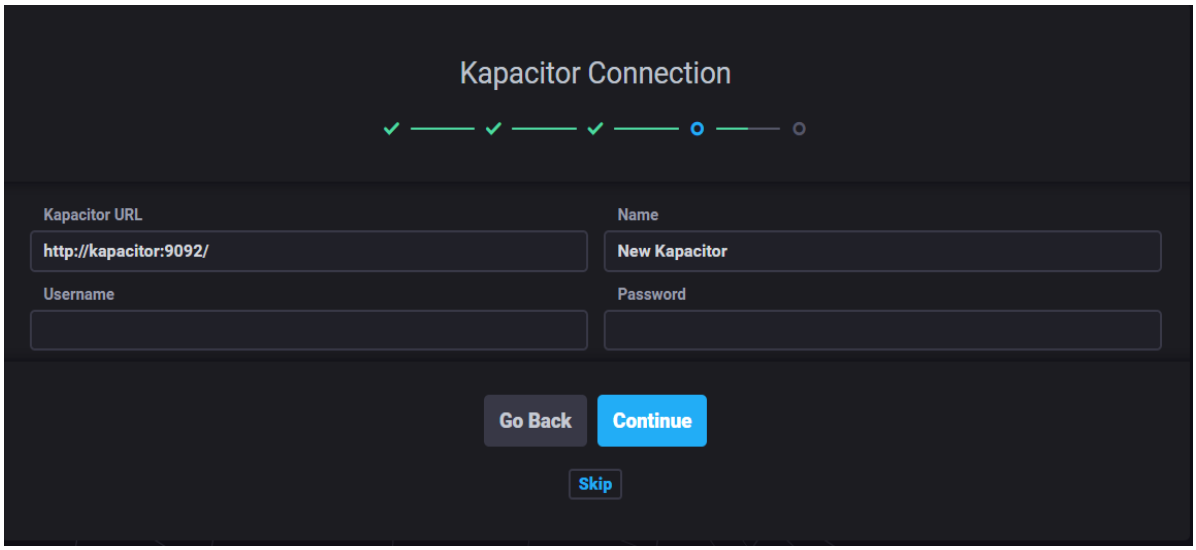
Ve třetím kroku se definuje připojení ke kapacitoru, který je zodpovědný za upozorňování na události. Tento krok lze také přeskočit, ale není to doporučeno. Rozhodně je vhodnější sen vyhnout chybovým hláškám. (Url: kapacitor:9092)



Obrázek 18 První krok konfigurace prvního spuštění chronografu



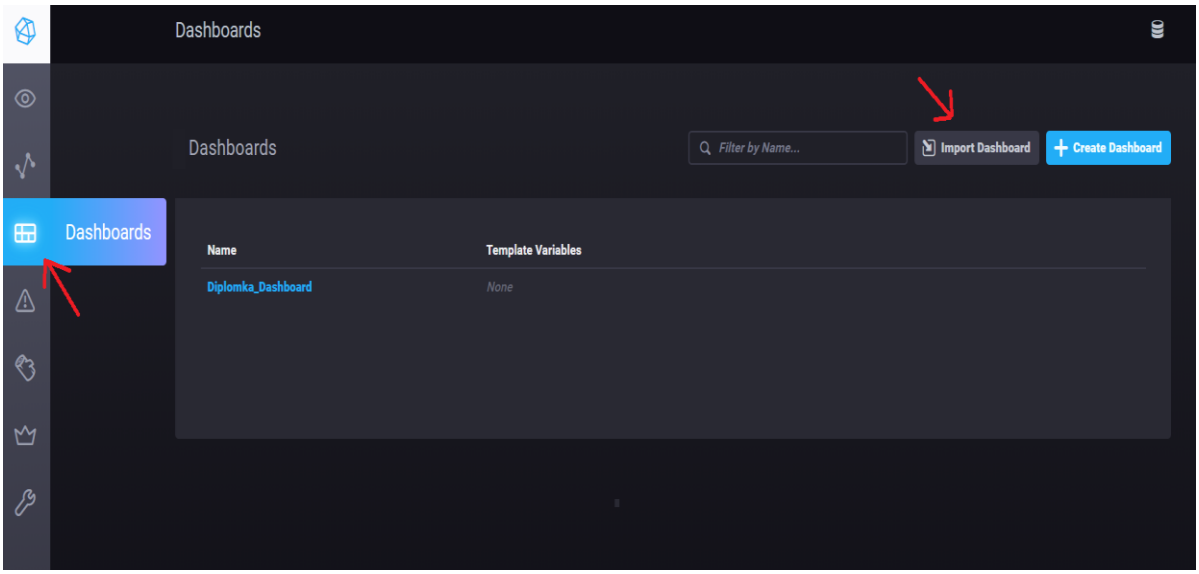
Obrázek 19 Druhý krok konfigurace prvního spuštění chronografu



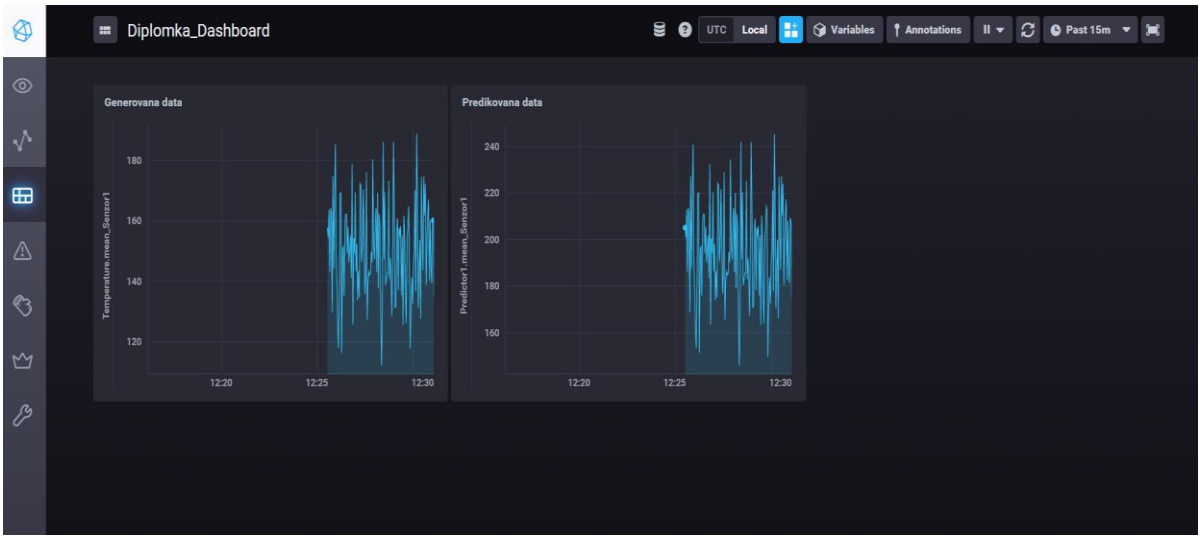
Obrázek 20 Třetí krok konfigurace prvního spuštění chronografu

Import vlastního dashboardu

Pro demonstraci funkčnosti celého systému byl vytvořen vlastní dashboard, který zobrazuje příchozí data z generátoru a predikci z prediktoru. Tento dashboard je exportován v adresáři `/chronograf/dashboard/Diplomka_dashboard.json` . Importovat jej lze prostřednictvím webového rozhraní chronografu viz návod v obrázku. Poté lze vybrat importovaný dashboard, který v základu zobrazuje příchozí data do databáze.



Obrázek 21 Import Dashboardu do Chronografu



Obrázek 22 Zobrazení importovaného dashboardu

6.3.2 Portainer

Při prvním spuštění portaineru je uživatel vyzván pro zadání uživatelského jména a hesla. Pro testovací účely bylo vybráno uživatelské jméno: admin, a heslo: adminadmin . Nastavení přístupových údajů je jediným nezbytným krokem k používání Portaineru, pro popis jeho praktického užití je věnován prostor v kapitole Uživatelská dokumentace.

6.4 Použití Chronografu

Prostřednictvím nástroje chronograf je možné spravovat řadu funkcionalit Influx balíčku, pomocí přehledného grafického rozhraní. Mezi tyto funkcionality patří:

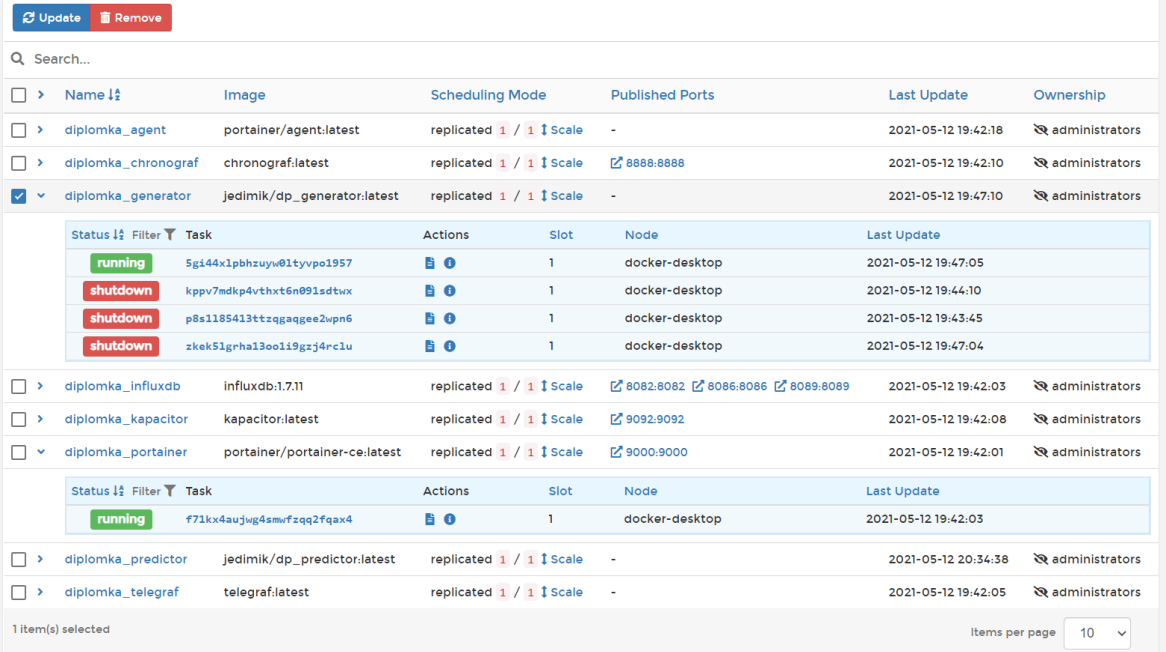
- Host list uchovává zobrazuje statistiky o využití výpočetních prostředků připojeného hostitelského zařízení.
- Data explorer čili prohlížeč dat je nástrojem pro vizualizaci dat uložených v InfluxDB databázi. Dokáže reagovat na dotazy a zobrazovat výsledky v přehledných grafech. Pro dotazy je použit InfluxQL jazyk.
- Dashboardy slouží jako šablony pro vizualizaci vybraných dat pomocí grafů. Lze si je vytvořit, importovat či exportovat, nebo využít předdefinovaných dashboardů, které jsou podporovány nástrojem Telegraf.

- Kapacitor jenž slouží k nastavování upozornění na data podle vybraných metrik. Sleduje data a v případě splnění podmínky provede definovanou akci. V praxi tak může například při překročení prahové hodnoty spustit upozornění, nebo spustit skript. Lze jej použít i pro streamování dat, kdy za periodicky bude posílat data do skriptu a pouštět jej.

6.5 Praktické použití nástroje Portainer

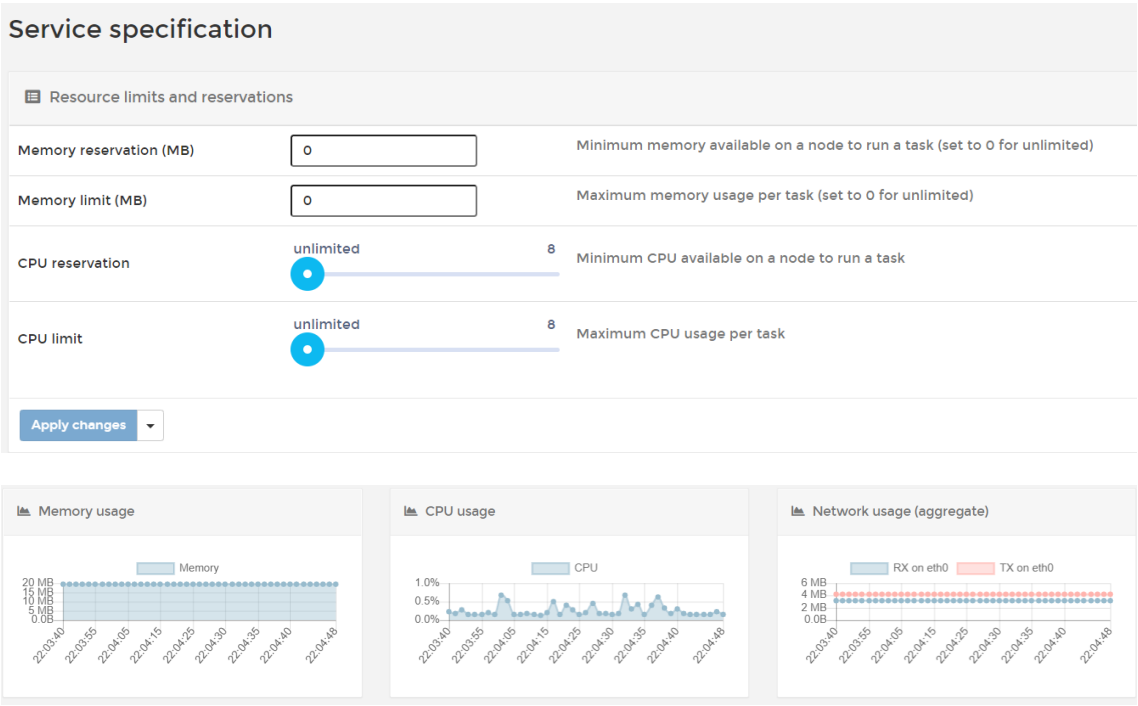
Nástroji Portainer bylo už v této práci věnováno dostatečně prostoru, tato kapitola bude sloužit pro ukázku vybraných funkcí tohoto nástroje.

Správa služeb v klastru



Obrázek 23 Správa služeb v Klastru

Na obrázku č. 24 je zobrazení nasazeného a spuštěného systému v Portaineru, konkrétně se jedná o záložku navigačního menu s názvem Stacks. Zde je možné odstraňovat, aktualizovat a škálovat kontejnery. Jsou zde zobrazeny i minulé repliky kontejnerů, které třeba v době běhu systému spadly a lze si, pokud kontejnery logují, nalézt důvod jejich pádu. V tomto konkrétním případě se jednalo o změnu konfigurace čili vypnutí původní repliky a start nově konfigurované repliky kontejneru byl žádoucí.



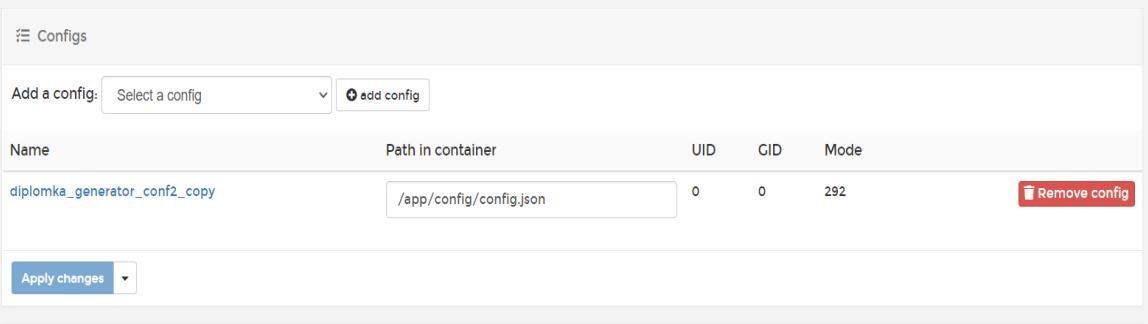
Obrázek 24 Přiřazování a monitoring výpočetních prostředků

Při vybrání jednotlivých služeb je možné definovat rozmezí pro maximální využití výpočetních prostředků. Zároveň je možné nároky na prostředky konkrétních služeb monitorovat. Nastavení maximálních výpočetních prostředků je možné pomocí záložky Services a vybrání konkrétní služby, pro monitoring aktuálně využívaných výpočetních prostředků je potřeba z navigačního menu zvolit položku Containers, poté vybrat konkrétní kontejner a zobrazit jeho stav (*Stats*).

Změna konfigurace služeb

Velmi zajímavým a užitečným nástrojem je změna konfigurace za běhu systému. Princip, jakým změna probíhá je popsán v podkapitole Změna konfigurace automatizovaná. Zde je uveden konkrétní postup pro změnu.

- V navigačním menu vybrat záložku Configs (konfigurační soubory)
- Přidat nový konfigurační soubor, nebo naklonovat původní a změnit v něm hodnoty
- Vybrat v navigačním menu záložku Services (služby)
- Vybrat konkrétní službu
- Najít nastavení Configs (konfigurace), třetí nastavovací blok odspoda
- Přidat konfigurační soubor a nastavit cestu do kontejneru.
- Aplikovat změny



Obrázek 25 Automatizovaná změna konfigurace

Po vybrání nového konfiguračního souboru a aplikování změn dojde ke zahození původní instance kontejneru a vytvoří se jeho nová replika, která ovšem pracuje již s novým konfiguračním souborem. Prerekvizitou pro změnu konfigurace je znát cestu ke konfiguračním souborům v kontejneru, mít definovanou konfiguraci v rámci swarm a v poslední řadě znát strukturu měněného konfiguračního souboru.

Další možnosti nastavení

V základu je snaha o zaručení neustálého běhu definovaných kontejnerů, to je žádoucí pro případ, kdy neočekávaně jeden z kontejnerů vypoví službu a orchestrátor Docker swarm vytvoří jeho novou repliku. Portainer v tomto kontextu umožňuje sledovat stav kontejnerů a eviduje případné spadnutí kontejnerů čímž dává zpětnou vazbu o kvalitě implementace.

Prostřednictvím webového rozhraní lze také spravovat Síť, ve které jsou kontejnery, definovat nové obrazy pro aktualizaci kontejnerů, spravovat volumes (sdílené obsahy) a další různé funkcionality. Samotný popis všech funkcionalit Portaineru by byl velmi rozsáhlý, na svém webu má aktuální a dobře popsanou dokumentaci a v případě nejasností je dobré se na ní obrátit.

6.6 Změna konfigurace služeb

Změna konfigurace manuální

Manuální změna konfigurace služeb je vhodná pro případ, kdy je aplikace nasazená a spuštěná nástrojem docker-compose, který nepodporuje definování konfigurací v compose souboru. Nová konfigurace pro službu, ale neumožňuje okamžité aktualizaci kontejneru a při změně kontejner spadne. Po změně konfigurace je tedy vždy nutné kontejner restartovat. Samotné změny konfigurace je pak možné docílit dvěma způsoby.

První způsob spočívá v přidání nového konfiguračního souboru do sdíleného úložiště (volume) mezi kontejnerem a hostem. Adresářem pro přidání konfiguračních souborů je `/configs/<nazev_sluzby>/` zároveň je potřeba zmínit, že kontejnery se sestavují na základě obrazu a v tomto případě je pro sestavení spuštěn skript (`startscript.sh`), který inicializuje spuštění služby s argumentem obsahujícím cestu ke konfiguračnímu souboru. V případě přidání nového konfiguračního souboru bude potřeba přepsat cestu ke konfiguračnímu souboru ve spouštěcím skriptu a sestavit nový obraz pro celou službu.

Druhý způsob spočívá v nahrazení původního konfiguračního souboru novým. Musí mít název shodný s minulým konfiguračním souborem a zároveň to musí být právě ten soubor, který je definován ve spouštěcím skriptu. Vzhledem ke skutečné implementaci je tedy nutné se rozhodnout, jestli se budou používat konfigurační soubory ve formátu YAML nebo JSON a držet se zvoleného formátu.

Změna konfigurace automatizovaná

Kontejnery nasazené ve swarm klastru umožňují, oproti nástroji docker-compose, automatizovanou změnu konfigurace. To znamená, že se plynule změní konfigurace služby a automaticky se restartuje běžící instance kontejneru pro novou konfiguraci. Tato změna je možná prostřednictvím příkazového řádku nebo pomocí webového nástroje Portainer – konkrétní návod je zmíněn v kapitole 6.5 Praktické použití nástroje Portainer.

7 Diskuse

Hlavním cílem této práce bylo objasnit základní problematiku návrhu a vývoje softwaru a příbuzných oblastí. Témata popsaná v teoretické části práce byla vybrána tak, aby korespondovaly s navrhovanou architekturou systému v praktické části práce. Informace o této problematice lze snadno nalézt jak na webových stránkách, v tištěných zdrojích, tak i prostřednictvím přednášek během studia. Je dobré klást důraz na tuto oblast informatiky, přestože je většině vývojářům a informatikům známá, tak málokdo svědomitě dodržuje přesně předepsané metodiky. Zpracování teoretické části této práce probíhalo až po samotné implementaci architektury systému, přestože se během této doby dokumentoval návrh i samotná implementace, tak prakticky nebyla zcela dodržena doporučená metodika pro návrh a vývoj, která je v teoretické části práce doporučena. Tato skutečnost přinesla mnoho problémů při vytváření praktické části diplomové práce a potvrzuje, že je opravdu vhodné následovat doporučenou metodiku a díky tomu se vyhnout problémům v budoucnu.

Návrh a implementace praktické části probíhala paralelně, přestože se nedá definovat konkrétní použitou metodiku, tak samotný vývoj byl v konceptu iterativního modelu, kdy průběžně vznikaly nové verze systému a následně byly konzultovány s vedoucím práce. Již v názvu práce je zmíněná detekce a predikce anomálií, která je ve skutečné verzi systému představována jednoduchou násobičkou. Bylo tak rozhodnuto z důvodu, že návrh a vytvoření modelu pro kvalitní prediktor by bylo značně náročné, a proto nebude součástí této práce. Místo toho má systém sloužit jako šablona pro budoucí prediktory. Proto byl systém vytvořen v konceptu mikroslužbové architektury s využitím kontejnerizace, ve skutečnosti se jedná o velmi silnou dvojici, která si dokáže při kvalitní orchestraci poradit i s vysokými nároky na výpočetní výkon, a vysoké nároky se u kvalitního prediktoru, který bude zpracovávat data v reálném čase, očekávají. Značnou část kapitoly implementace tvoří pouze obecné vysvětlení funkčnosti a případně ukázka zajímavých částí v praxi, kód je dobře okomentován a jeho třídy včetně metod mají intuitivní názvy, proto samotné implementaci nebyl věnován tak velký prostor. V zadání práce byl požadavek na možné nasazení v cloudu, přestože tento způsob nebyl otestován, tak na základě použitých technologií, které tuto skutečnost potvrzují i v teoretické části práce, by hypoteticky neměl být žádný problém s nasazením systému do cloudu.

Výsledný systém a jeho aktuální funkčnosti nejsou zcela obecné a stále je prostor pro vylepšení. Významnou otázkou je implementace nových prediktorů, aktuálně se počítá s prediktory v odděleném adresáři v rámci jednoho predikčního obrazu, což odporuje konceptu mikroslužeb. Vzhledem k navržené architektuře systému je možné vytvářet nové obrazy prediktorů nezávislých na technologické platformě a zajistit jejich funkčnost dodržením pravidel pro komunikaci ve standardizovaném formátu a způsobu komunikace přes frontu, včetně pravidel pro zasílání dat do InfluxDB databáze pomocí knihoven pro specifickou technologickou platformu. Aktuálně je komunikace prediktoru s databází InfluxDB jednosměrná, prediktor posílá data do databáze. Je vhodné se zamyslet nad obousměrnou komunikací, při které by prediktor zasílal databázi dotazy na data a vytvářel další predikce na základě získaných dat.

Zakomponování InfluxData Tick Stack se ukázalo jako výborná volba, díky balíčku nebylo nutné pracovat na vlastní implementaci webového grafického rozhraní a zároveň použití balíčku rozšíří obzory čtenářům této práce, pokud se s ním ještě nesetkali. Moudré rčení „nevymýšlet kolo“ v tomto kontextu plně vystihuje důvody použití Influx balíčku. Během implementace systému byla od InfluxData vydána nová verze databáze InfluxDB, která má nové aplikační rozhraní a novou ovládací knihovnu pro python. Z toho důvodu je pro systém použita starší verze InfluxDB databáze a to přesně 1.7.11. Do budoucna je vhodné popřemýšlet o úpravách pro podporu nové verze databáze.

Pro nasazení do swarm klastru je nutné použít existující obrazy kontejnerů a nelze je lokálně sestavovat během nasazení. Tyto obrazy lze sice lokálně sestavit a pak na ně odkazovat v rámci lokálního repozitáře, ale přineslo by to více problémů než užitku. Kvůli tomu byly vytvořeny repozitáře pro obrazy prediktoru a generátoru na globálním Docker repozitáři DockerHub, který je zároveň propojen s GitHub repozitářem tohoto projektu a při každé nové změně, která se nahraje standardně do GitHub repozitáře, dojde k sestavení nových obrazů na DockerHub. Tato integrace je užitečná a dokáže ušetřit čas.

Ve výsledném systému je stále prostor pro vylepšení, splňuje cíle vytyčené při zadání práce a může posloužit jako vhodná architektura pro budoucí predikce časových řad v reálném čase.

8 Závěr

V diplomové práci se podařilo splnit vytyčené cíle, kterými bylo objasnění problematiky návrhu a vývoje softwaru s důrazem na kontejnerizaci aplikací a jejich orchestraci, a vytvoření funkční demoverze architektury systému pro sběr dat v reálném čase a predikci anomálií. Teoretická část čerpala z relevantních informačních zdrojů, kterými byly především zahraniční články a oficiální dokumentace zmíněných produktů. Praktická část obsahuje návrh, implementaci a uživatelskou dokumentaci, která je v souladu s aktuálním stavem architektury systému. Systém stále obsahuje prostor pro vylepšení, je vhodné ho otestovat na reálných datech s reálným modelem prediktoru, aby se zjistila efektivita tohoto řešení a mohlo se rozhodnout, jakým směrem se s tímto systémem lze vydat.

Seznam použité literatury

[1] VONDRÁŠEK, Jiří. *Metody strukturované analýzy a návrhu informačních systémů*. Jindřichův Hradec, 2007. Diplomová práce. Vysoká škola ekonomická v Praze, Fakulta managementu.

[2] ZEMEK, Petr. Zaměňované pojmy v oblasti SW inženýrství. *Petr Zemek | O věcech, které mě baví* [online]. Brno, 2008-2020 [cit. 2021-04-26]. Dostupné z: https://cs-blog.petrzemek.net/2010-01-17-zamenovane-pojmy-v-oblasti-sw-inzenyrstvi

[3] Metodika vývoje softwaru. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001 [cit. 2021-04-26]. Dostupné z: https://cs.wikipedia.org/wiki/Metodika_v%C3%BDvoje_softwaru

[4] ŠIMON, Bc. Vladimír. *Návrh informačního systému pro správu VT pomocí UML*. Praha, 2013. Diplomová práce. Bankovní institut vysoká škola Praha, katedra matematiky, statistiky a informačních technologií. Vedoucí práce Doc. Ing. Bohumil Miniberger, CSc.

[5] KANJILAL, Joydip. Pros and cons of monolithic vs microservices architecture. *TechTarget* [online]. Atlanta: Whatls.com, 1999-2021 [cit. 2021-03-30]. Dostupné z: https://searchapparchitecture.techtarget.com/tip/Pros-and-cons-of-monolithic-vs-microservices-architecture

[6] SOA (Service-Oriented Architecture). *IBM Cloud Learn Hub* [online]. Armonk, New York: IBM [cit. 2021-04-06]. Dostupné z: https://www.ibm.com/cloud/learn/soa

[7] SOA (Service Oriented Architecture). *ManagementMania.com* [online]. Wilmington (DE), 2011-2021 [cit. 2021-04-15]. Dostupné z: https://managementmania.com/cs/service-oriented-architecture

[8] BIH, Joseph. Service oriented architecture (SOA) a new paradigm to implement dynamic e-business solutions. *Ubiquity* [online]. 2006, **2006**(), 1-1 [cit. 2021-04-21]. ISSN 1530-2180. Dostupné z: doi:10.1145/1162511.1159403

[9] KALSKE, Miika, Niko MÄKITALO a Tommi MIKKONEN. Challenges When Moving from Monolith to Microservice Architecture. *Current Trends in Web Engineering*. Cham: Springer International Publishing, 2018, , 32-47. Lecture Notes in Computer Science. ISBN 978-3-319-74432-2. Dostupné z: doi:10.1007/978-3-319-74433-9_3

[10] OSNAT, Rani. A Brief History of Containers: From the 1970s Till Now. *Aqua Blog* [online]. India: Aqua Security Software [cit. 2021-04-27]. Dostupné z: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

[11] SHAW, Keith. What is a virtual machine, and why are they so useful?. *NetworkWorld* [online]. IDG Communications, Inc., 2020 [cit. 2021-04-28]. Dostupné z: https://www.networkworld.com/article/3583508/what-is-a-virtual-machine-and-why-are-they-so-useful.html

[12] LINTHICUM, David. Containers: The pros and cons you may not know about. *InfoWorld* [online]. IDG Communications, Inc., 2019 [cit. 2021-04-28]. Dostupné z: https://www.infoworld.com/article/3342165/containers-the-pros-and-cons-you-may-not-know-about.html

[13] SIMIC, Sofija. Containers vs Virtual Machines (VMs): What's the Difference?. *PhoenixNAP, Global It Services* [online]. Phoenix, USA: Global IT Services [cit. 2021-04-29]. Dostupné z: https://phoenixnap.com/kb/containers-vs-vms

[14] *Docker: Docker docs* [online]. Palo Alto, California: Docker, Inc., 2013 [cit. 2021-04-04]. Dostupné z: https://docs.docker.com/

[15] CHEN, Jay. Attacker’s Tactics and Techniques in Unsecured Docker Daemons Revealed: Docker Daemon. *Paloalto NETWORKS, UNIT 42* [online]. California, US: Palo Alto Networks Inc., 2020 [cit. 2021-04-30]. Dostupné z: https://unit42.paloaltonetworks.com/attackers-tactics-and-techniques-in-unsecured-docker-daemons-revealed/

[16] Docker Architecture. *Knoldus Blogs* [online]. India: Knoldus Inc., 2020 [cit. 2021-04-30]. Dostupné z: https://blog.knoldus.com/docker-architecture/

[17] Docker Architecture. *JavaTpoint* [online]. JavaTpoint, 2011-2018 [cit. 2021-05-01]. Dostupné z: https://www.javatpoint.com/docker-architecture

[18] YEGULALP, Serdar. What is Docker? The spark for the container revolution. *InfoWorld* [online]. IDG Communications, Inc., 2019 [cit. 2021-04-29]. Dostupné z: https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html

[19] MORAVCIK, Marek a Martin KONTSEK. Overview of Docker container orchestration tools. In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)* [online]. IEEE, 2020, s. 475-480 [cit. 2021-04-03]. ISBN 978-1-6654-2226-0. Dostupné z: doi:10.1109/ICETA51985.2020.9379236

[20] Container Orchestration Tools. *OpsWorks* [online]. 2017-2021 [cit. 2021-04-04]. Dostupné z: https://opsworks.co/container-orchestration-tools

[21] *Kubernetes.io* [online]. The Linux Foundation', 2021 [cit. 2021-03-31]. Dostupné z: https://kubernetes.io/

[22] BURNS, Brendan, Joe BEDA a Kelsey HIGHTOWER. *Kubernetes: Up and Running*. 2. vydání. United States: O'Reilly Media, Inc., 2019. ISBN 9781492046530.

[23] Kubernetes Architecture. *Aqua Cloud Native Academy* [online]. Aqua Security Software Ltd., 2021 [cit. 2021-05-02]. Dostupné z: https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-architecture/

[24] ARUNDEL, John a Justin DOMINGUS. *Cloud Native DevOps with Kubernetes* [online]. 1. vydání. Sebastopol, CA: O'Reilly Media, 2019 [cit. 2021-05-02]. ISBN 9781492040750. Dostupné z: https://learning.oreilly.com/library/view/cloud-native-devops/9781492040750

[25] RAVINDRA, Savaram. Kubernetes vs. Docker Swarm: What's the Difference?. *TheNewStack* [online]. Portland, USA: The New Stack, 2015-

2021 [cit. 2021-05-06]. Dostupné z: https://thenewstack.io/kubernetes-vs-docker-swarm-whats-the-difference

[26] 7 Powerful Time-Series Database for Monitoring Solution. *GEEKFLARE* [online]. England, 2015-2021 [cit. 2021-05-08]. Dostupné z: https://geekflare.com/time-series-database/

Seznam Obrázků

Seznam Tabulek

Seznam Kódů

Seznam Příloh

Příloha 1: Odkazy na zdrojové kódy

Odkaz na zdrojový kód diplomové práce na GitHub:

https://github.com/jedimik/Diplomova_prace.git

Odkaz na zdrojový kód InfluxData Tick Stack na GitHub:

https://github.com/influxdata/sandbox

Odkaz na obraz prediktoru z Docker Hub:

https://hub.docker.com/repository/docker/jedimik/dp_predictor

Odkaz na obraz generátoru z Docker Hub:

https://hub.docker.com/repository/docker/jedimik/dp_generator

Odkaz na obraz InfluxDB databáze z Docker Hub:

https://hub.docker.com/_/influxdb

Odkaz na obraz kapacitoru z Docker Hub:

https://hub.docker.com/_/kapacitor

Odkaz na obraz chronografu z Docker Hub:

https://hub.docker.com/_/chronograf

Odkaz na obraz telegrafu z Docker Hub:

https://hub.docker.com/_/telegraf