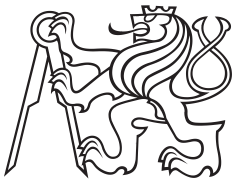
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Architektura distribuovaného systému pro těžení dat z internetu

Bc. David Stražovan

Vedoucí: Ing. Josef Smolka
Květen 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Stražovan** Jméno: **David** Osobní číslo: **457796**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Architektura distribuovaného systému pro těžení dat z internetu

Název diplomové práce anglicky:

Architecture of distributed system for internet data mining

Pokyny pro vypracování:

Cílem práce je návrh nové softwarové architektury systému pro těžení dat z internetu, která plní požadavky klíčových vlastníků systému. Součástí návrhu architektury je:
1. analýza stavu současného řešení, identifikace problematických částí a návrh možného zlepšení,
2. analýza požadavků na systém,
3. identifikace/výběr klíčových požadavků, které formují architekturu,
4. rešerše možných přístupů (architektonické styly, vzory, normy/standardy, technologie),
5. návrh architektury, včetně návrhu klíčových komponent a jejich zodpovědností, technologií a návrhu procesu vývoje a provozu (postupů, rolí, metod, nástrojů),
6. hodnocení architektury z pohledu požadavků, včetně výčtu nutných kompromisů v návrhu a jejich odůvodnění,
7. ověření architektury pomocí prototypu, který demonstruje užití klíčových technologií a ověří naplnění vybraných kvalitativních atributů (např. odolnost proti výpadku, horizontální škálovatelnost), funkčně se zaměří na distribuci práce mezi agenty, vzájemnou koordinaci agentů a ošetřování chybových stavů při zpracování dílčích úloh.

Seznam doporučené literatury:

- Len Bass , Paul Clements , Rick Kazman - Software Architecture in Practice (4th edition)
- Martin Kleppmann - Designing Data-Intensive Applications
- Humberto Cervantes, Rick Kazman - Designing Software Architectures: A Practical Approach
- Mark Richards - Software Architecture Patterns

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Josef Smolka katedra softwarového inženýrství FJFI

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **11.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

Ing. Josef Smolka podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Poděkování

Chtěl bych poděkovat svému vedoucímu bakalářské práce Ing. Josefu Smolkovi za cenné rady, připomínky a vstřícnost při zpracování této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje.

V Praze, 19. 5. 2022

Abstrakt

Práce se zabývá analýzou existující architektury systému, který těží a zpracovává data z internetu. Systém začíná narážet na své limity ve výkonnosti, udržovatelnosti, monitorování a DevOps.

Hlavní náplní práce je návrh nové architektury na základě analýzy současného stavu. V rámci návrhu nové architektury jsou popsány nároky, které jsou na ni kladeny, architektonické vzory, které dané podmínky a problémy řeší a nakonec samotný návrh nové architektury.

V rámci práce vznikne proof of concept, na kterém jsou výsledky práce demonstrovány.

Klíčová slova: softwarová architektura, distribuovaný systém, vývoj software, škálovatelnost

Vedoucí: Ing. Josef Smolka

Abstract

The thesis deals with the analysis of the existing architecture of a system that extracts and processes data from the Internet. The system is starting to hit its limits in performance, maintainability, monitoring and DevOps.

The main focus of the work is the design of a new architecture based on the analysis of the current state. As part of the design of the new architecture the demands that are placed on it, the architectural patterns that address the given requirements and problems, and finally the design itself.

A proof of concept is created to demonstrate the results of the work.

Keywords: software architecture, distributed system, software development, scalability

Title translation: Architecture of a distributed system for extracting data from the Internet

Obsah

Obrázky

Tabulky

Kapitola 1
Úvod

Architektura systému je jednou z jeho klíčových vlastností systému. Má vliv na jeho výkon, použitelnost, dostupnost, škálovatelnost a flexibilitu v reagování na byznysové požadavky. Jedná se o sadu kritických rozhodnutí, která mohou mít zásadní vliv na úspěch či neúspěch softwarového produktu.

Pokud se jedná o systémy pro těžbu dat z internetu, je klíčové, aby byly vysoce škálovatelné, monitorovatelné a bylo možné rychle reagovat na změny. Zde vidíme, že správně zvolená architektura, podporující tyto aspekty, je jednou z klíčových vlastností pro takový systém.

Motivací pro tuto práci je existence systému pro těžbu dat z internetu, jehož architektura začíná narážet na své limity ve dříve zmíněných aspektech. Vlastníci systému se tak rozhodli pro návrh nové architektury, která by netrpěla problémy současné architektury a umožnila by další rozvoj, škálování, podporovala automatizaci provozních procesů a centrální monitorování celého systému. Krom samotné struktury systému a rolí jednotlivých komponent je třeba myslet i na vývoj a vývojový team. Práce se proto dotýká i návrhu procesu vývoje, verzování kódu a způsobů distribuce artefaktů samotných. Nová architektura a procesy se snaží v maximální možné míře využít aktuální znalosti týmu, jejich zkušenosti s technologiemi a infrastrukturu, na které je současná architektura provozována tak, aby náklady na implementaci nové architektury byly co nejnižší.

První část práce popisuje aktuální architekturu systému a její nedostatky. Následuje rešerše architektonických stylů, vzorů, taktik a technologií. Poté je představena metodika pro návrh nové architektury a samotný návrh nové architektury, kde jsou zaznamenána všechna rozhodnutí; výstupem návrhu je současně i dokumentace pro další rozvoj. V rámci návrhu v poslední části je provedeno ověření naplnění požadavků novou architekturou pomocí prototypu.

Kapitola 2

Zadání

Cílem práce je návrh nové softwarové architektury systému pro těžení dat z internetu, která plní požadavky klíčových vlastníků systému. Součástí návrhu architektury je analýza současného stavu, požadavků na systém, identifikace/výběr klíčových požadavků, které formují architekturu, výčet nutných kompromisů a jejich odůvodnění, návrh klíčových komponent a jejich zodpovědností, návrh technologií a návrh procesu vývoje a provozu (postupů, rolí, metod, nástrojů).

Požadavky na novou architekturu:

- V maximální možné míře automatizuje prvky testování, sestavování a nasazování dílčích částí aplikace.
- Je provozována takovým stylem, který:
  - Agenta abstrahuje od podkladového serverového prostředí (fyzický server, VPS).
  - Poskytuje jasný a ucelený pohled na stav celého systému.
  - Je horizontálně škálovatelný.
  - Je vysoce dostupný.
  - Je robustní - počítá s výpadky částí systému, je tzv. fault-tolerant.
- Agent je naprosto transparentní vůči lokaci, ve které je provozován. V rámci provozního prostředí může být libovolně přesouván.
- Minimalizuje nároky/náklady na výpočetní výkon.
- Minimalizuje nároky/náklady na lidskou obsluhu.
- Bezpečně komunikuje s centrálním systémem.
- Dokáže flexibilně reagovat na změny v čase (ať už na úrovni těžených zdrojů, tak i na úrovni centrálního systému).

Návrh se mj. dotkne témat:

- správa a distribuce konfigurace agentů,
- monitorování agentů a správa logů,

- distribuce práce mezi agenty, sběr a kompletace výsledků,
- vzájemná koordinace,
- ošetření chybových stavů (zadaná práce se nemůže ztratit).

Práce se soustředí na návrh architektury distribuovaného agenta a klade předpoklady na centrální systém. Současně s návrhem architektury bude vytvořen prototyp (Proof of Concept), který bude demonstrovat reakci architektury na klíčové požadavky.

Kapitola 3

Analýza současného řešení

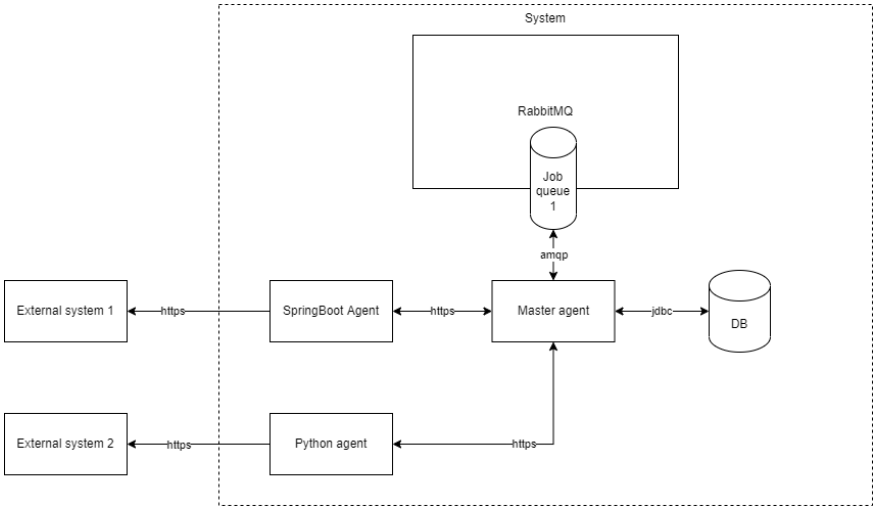
Současné řešení je podčástí většího systému, který můžeme rozdělit na *centrální systém* a *roboty*. *Centrální systém* se stará o zpracování dat a plánování práce (zde je přesah do části robotů) a roboti na základě úloh naplánových při zpracování dat provádějí úlohy.

Tato kapitola obsahuje analýzu části systému s roboty a komponenty, které s ní nesouvisí, zde nejsou uvedeny.

3.1 Hlavní komponenty a komunikace

3.1.1 Diagram komunikace a komponent

Následující diagram zobrazuje high-level pohled na architekturu současného řešení, na kterém jsou vyznačené hlavní komponenty systému a komunikace mezi nimi. Master agent běží vždy jen jeden, ovšem agentů může být několik instancí. Tento fakt není pro tento diagram zásadní a je zde pro zjednodušení vynechán. Podobně počet front na RabbitMQ.



Obrázek 3.1: Komponenty a konektory současné architektury

3.1.2 RabbitMQ a fronty

Systém používá pro plánování práce RabbitMQ fronty. V systému existuje několik front (cca 50, ne všechny se ovšem týkají práce agentů), pomocí kterých si master agent plánuje práci. Tyto fronty jsou perzistentí a prioritizované[1].

3.1.3 Master agent

Master agent je komponenta, která systém řídí. Plánuje práci (pomocí front), komunikuje s roboty, databází a s okolním světem. Master agent je komponenta sdílená s *Centrálním systémem*, z jehož pohledu se jedná o master node. Jsou na něm spouštěny interní úlohy pro práci s daty, které mohou rozhodnout o potřebě spustit úlohy na agentu. V takovém případě je úloha naplánovaná pomocí RabbitMQ. Zároveň poskytuje agentům jejich konfiguraci. Pro tyto potřeby (práce agentů, její výsledky a poskytování konfigurace) poskytuje REST API. Pro autentizaci agentů implemetuje OAuth endpoint.

Master agent kontroluje souběh úloh tak, aby se nestalo, že v případě duplicitních úloh ve frontě se budou zpracovávat duplicitní úlohy paralelně.

3.1.4 SpringBoot/Python agent

Agent je samostatná aplikace, která implementuje úlohy a provádí extrakci dat z externích systémů. Svou konfiguraci a práci získává z master agenta pomocí REST API. Komunikace mezi agenty a master agentem je autentizovaná a zabezpečena komunikací přes HTTPS, není zde žádné další šifrování na úrovni aplikační vrstvy. Původně byli agenti implementováni v Javě a frameworku SpringBoot a v současné době probíhá přepis do Pythonu. Typy úloh jsou:

- *webový crawler* - prochází síť webů a mapuje jejich propojení,
- *webový scrapper (různé specializace)* - těží konkrétní typ dat z webu,
- *WHOIS resolver* - těží základní údaje o doméně z doménového registru,
- *DNS resolver* - těží základní údaje o doménovém názvu (IP, jmenné servery),
- *health check* - kontroluje základní funkci webu,
- *RSS/Atom downloader* - stahuje RSS a Atom feedy.

3.1.5 DB

Z pohledu systému agentů je důležitá hlavně pro uchovávání jejich konfigurace. Větší pozornost zde komponentě nebude věnována (je důležitější z pohledu *Centrálního systému*).

[1] https://www.rabbitmq.com/priority.html

3.2 Vývoj

Aktuální vývoj využívá následující nástroje:

- GitLab
- Ansible
- Zabbix
- Trello (pro SCRUM)
- IntelliJ Idea community edition.

3.2.1 GitLab

GitLab[2] je nyní využíván pro dva účely:

- Git
- CI/CD pipelines.

V prvním případě se používá jako hosting pro git repozitář projektu a verzování kódu. V druhém se jedná o základně nakonfigurované CI/CD pipelines pro sestavení a nasazení aplikace.

3.2.2 Ansible

Ansible[3] je nástroj pro automatizaci a management konfigurace IT infrastruktury. Umožňuje mít tzv. "infrastructure as code", což umožňuje infrastrukturu verzovat, stejně jako kód. Ansible je použito pro konfiguraci fyzických serverů a VPS.

3.2.3 Zabbix

Zabbix[4] je opensource nástroj pro monitorování infrastruktury a notifikace. V aktuální architektuře je použit pro monitorování serverů a VPS a notifikování v případě výpadků.

3.2.4 Trello

Trello[5] je nástroj pro řízení projektů používající systém Kanban[6]. V projektu je použita pro organizaci a řízení vývoje agilní metodika (SCRUM). Z pohledu architektury není zajímavý nástroj samotný, ale agilní vývoj, a ten by nová architektura měla podporovat v maximální možné míře.

[2] https://gitlab.com/
[3] https://www.ansible.com/
[4] https://www.zabbix.com/
[5] https://trello.com/
[6] Vizuální systém organizace práce

3.2.5 IntelliJ Idea community edition

IntelliJ Idea[7] je aktuálně využívána k vývoji komponent v jazyce Java. Z pohledu architektury nemá tento nástroj vliv.

3.3 Nedostatky

3.3.1 Distribuce a nasazení

Ve stávajícím řešení jsou artefakty po sestavení zabaleny jako DEB balíčky[8], které jsou následně distribuovány na cílové prostředí pomocí interního repozitáře. Na stanici se pak artefakt nainstaluje a spouští pomocí startup skriptů. Jednou z nevýhod tohoto řešení je závislost na platformě. Není tak možné změnit platformu na jinou, která nepodporuje DEB balíčky. Manuální instalace a spouštění artefaktů neumožňuje vyspělejší DevOps procesy.

3.3.2 Komunikace a škálování

Jak je vidět v diagramu komunikace mezi komponentami (obrázek 3.1), komunikace mezi agenty a master agentem je realizována pomocí synchronních volání REST API master agenta. Synchronní komunikace je jednodušší na představu a na vývoj (je méně komplexní a je v ní méně prostoru pro chyby), ale také hůře neškáluje. Při navyšování počtu agentů se zvyšují nároky na master agenta, který musí zvládat obsluhovat velké množství požadavků (výsledky práce agentů). Stává se tak úzkým hrdlem a zároveň je to single-point-of-failure. V aktuálním řešení není mezi agenty a master agentem žádná mezilehlá komponenta, která by umožňovala zátěž na master agenta regulovat.

3.3.3 Provoz

Aktuálně není provoz a nasazování systému dostatečně řízeno. Je tak velice složité dozvědět se celkový obraz systému (jaké jsou v systému verze agentů). Aktuální řešení používá Zabbix pro dohled nad servery, neřeší ovšem celkový pohled nad samotným systémem. V architektuře navíc chybí řešení pro sběr logů, jejich analýzu a vizualizaci. V aktuálním řešení také dále chybí jakákoliv forma service registry, ve kterém by byl přehled všech běžících robotů, jejich verzí a dalších informací (v případě více agentů na jednom serveru monitorovaném Zabbixem). Tento stav snižuje efektivitu provozních činností a může být zdrojem dalších problémů (např. souběh nekompatibilních verzí komponent vedoucí k selhání systému).

[7] https://www.jetbrains.com/help/idea/discover-intellij-idea.html
[8] https://wiki.debian.org/Packaging

3.4 Metriky a zátěž aktuálního systému

Pro řešení není kritický výkon jednotlivých agentů, jelikož ten je u většiny úloh limitován omezeními a propustností externích systémů. Zároveň s tím samotné úlohy nemají vysoké nároky a jedná se spíše o větší počet instancí jednotlivých úloh. Pro příklad - systém uchovává údaje o doménách v řádech milionů domén (druhého a třetího řádu) a pro ně je potřeba s určitou periodou spouštět úlohu *webového crawleru*.

Vzhledem k tomu, že výkon samotných agentů není kritický a je pro nás důležitý spíše jejich počet, je prioritou minimalizace konzumace systémových prostředků (hlavně spotřeba paměti). Toto je důležitý faktor ceny provozu řešení, jelikož agenty je potřeba provozovat na VPS, kde jsou ceny určené konfigurací dostupných systémových prostředků.

Kapitola 4

Rešerše

4.1 Architektonické styly

4.1.1 Vrstevnatá architektura

Jedná se o nejrozšířenější architektonický styl. Výhodou stylu je, že kopíruje podnikovou organizaci většiny společností.

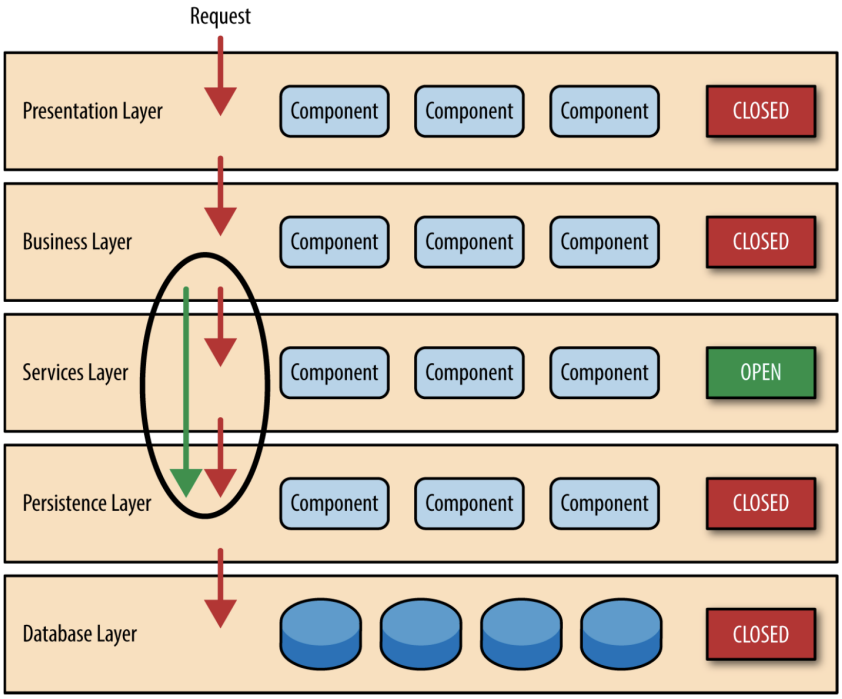
Komponenty jsou v tomto stylu organizovány ve vrstvách, kde každá má specifickou roli. Počet vrstev není nijak omezen.

Klíčovými koncepty jsou[7]:

- **Zapouzdření logiky**. Každá vrstva má svou zodpovědnost a zapouzdřuje její logiku.
- **Otevřené a uzavřené vrstvy**. Princip uzavřených vrstev znamená, že každá vrstva může komunikovat pouze s vrstvou přímo pod ní. Existují ovšem případy, kdy tutu podmínku chceme relaxovat. V takovém případě můžeme umožnit komunikaci i s nižší vrstvou, pokud je vrstva mezi nimi označena jako otevřená. Důvodem pro vytvoření takové vrstvy může být například přidání mezivrstvy s pomocnými komponentami. Pokud by taková vrstva byla uzavřená, znamenalo by to implementovat v ní kód, který by sloužil jen pro propouštění požadavku do nejdůležitějších vrstev, ale nepřidával by žádnou novou logiku.

Styl je znázorněn na obrázku 4.1. Obrázek ilustruje princip uzavřených a otevřených vrstev.

Výhodou tohoto stylu je jeho jednoduchost. Vývoj je jednoduchý, testování je jednoduché. Problémem stylu je, že většina jeho implementací je (a má tendenci být) monolitických. Proto je těžší provádět změny a ty následně nasazovat (většinou je nutné nasadit celou aplikaci). Škálovatelnost zde také není moc dobrá. Pokud máme plně monolitickou implementaci, je potřeba duplikovat celé instance aplikace. Pokud nasazujeme jednotlivé vrsty separátně, můžeme replikovat ty, ovšem granularita replikace zůstává stále široká a škálování je nákladné.



Obrázek 4.1: Vrstevnatá architektura[7]

4.1.2 Událostmi řízená architektura

Událostmi řízenou architekturu můžeme popsat jako asynchronní, distribuovanou architekturu, kde služby reagují na změny stavu. Těmto změnám říkáme události. Ty mohou nést buďto nějaký stav nebo představovat notifikaci. Obecněji lze událostmi řízenou architekturu definovat takto[12]:

> V architektuře řízené událostmi se uvnitř nebo vně firmy vyskytne významná událost, která se okamžitě rozšíří mezi všechny zainteresované strany (lidské nebo automatizované). Zainteresované strany vyhodnotí událost a případně provedou nějakou akci. Akce v událostmi řízení architektuře může může zahrnovat vyvolání služby, spuštění obchodního procesu a/nebo další zveřejnění informací.

V rámci této architektury rozlišujeme následující styly zpracovávání:

- Simple event processing - Tzv. *Simple event* je vytvořen, pokud nastane změna nějakého stavu nebo podmínky. Může se jednat například o detekci změny tlaku v pneumatikách, změnu teploty vzduchu apod. Na základě takové události lze následně například upozornit uživatele indikací na monitorovací obrazovce.
- Complex event processing - V některých případech je potřeba uvažovat složitější zpracování událostí. Můžeme se dostat do situací, kdy pro vyvolání další události čekáme na sérii různých událostí. Jinými slovy,

čekáme, než se naplní nějaká kritéria a až následně pokračujeme se zpracováním (vytvořením další události). Tento styl se často využívá k detekci anomálií, podvodů a různých vzorů chování. Příkladem může být událost zavedení nového uživatele do systému, která čeká na událost regitrace a následné aktivace účtu.

- Event stream processing - Zpracovávání streamu událostí. Události jsou zapisovány do logu v pořadí výskytu. Konzumenti z tohoto logu mohou číst. Tento styl je využívám vzorem Event sourcing.
- Pub/Sub - klasický publisher-subscriber vzor komunikace. Na vstupu má jednoho (případně více) producentů, kteří generují události a na druhé straně odběratele, kteří události dostávají a zpracovávají.

Dále v rámci architektury rozlišujeme následující komponenty:

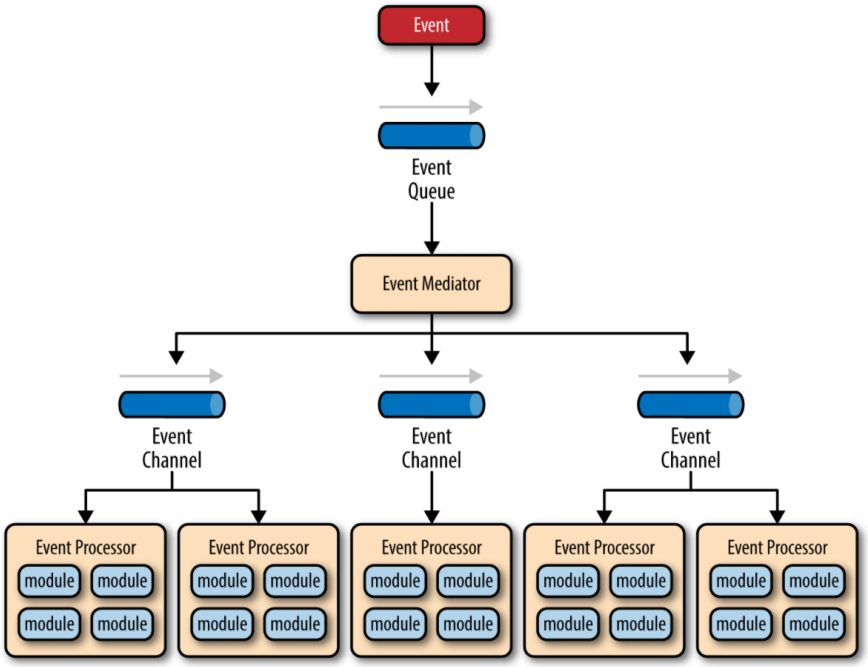
- Událost (event) - Změna stavu objektu (či podmínky) a data, která s touto změnou souvisí. Připojeným datům se říká "payload".
- Producent událostí - Každá událost má nějaký zdroj, který ji vygeneroval. Může se jednat o uživatele, který kliknul na tlačítko, senzor, který detekoval změnu teploty, či například o agenta, který těží databázový transakční log.
- Zpracovatel (konzument) události - Komponenta, která zprácovává výskyt události.
- Kanál - Základní komponenta pro přeposílání zpráv mezi producenty a konzumenty.
- Event loop - Spravuje interakci mezi událostmi a konzumenty.
- Event store - Databáze pro uložení událostí. Ty jsou zde uloženy pro pozdější použití. Události z event store jsou následně zveřejněny na event stream či na message broker. Často použito ve spojení se vzory Event sourcing a Command Query Responsibility Segregation (dále jen CQRS).
- Event mediator - Pokud zpracování události vyžaduje více kroků, mediator pošle další asynchronní události. Komponenta zajišťuje následnou orchestraci.
- Fronta - Komponenta pro příjem zpráv od klienta, ze které je následně čte mediator.

Na základě potřeby orchestrace následně rozdělujeme událostmi řízené architektury na dvě možné topologie:

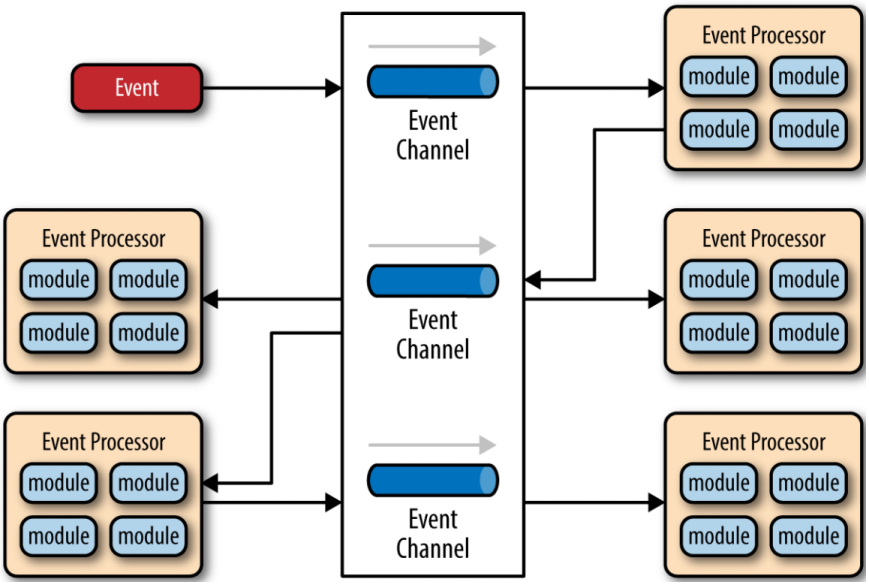
- Mediator topologie - Topologie hodící se pro komplexní zpracování událostí. Skládá se ze čtyř hlavních komponent - fronta, mediator, kanály a procesory. Události přicházejí do fronty událostí, na kterou reaguje mediator. Ten vytvoří příslušné události, které kanály pošle příslušným

procesorům, a dále orchestruje zpracování. Zde je důležité si uvědomit, že mediator samotný neprovádí žádnou byznysovou logiku, pouze rozpad na podúlohy a jejich orchestraci. Příklad topologie je uveden na obrázku 4.2.

- Broker topologie - Topologie hodící se pro jednodušší zpracování událostí. Odpadají zde komponenty fronty a mediátoru a topologie obecně je jednodušší, než v případě mediatoru. Zde pracujeme pouze s konzumenty a komponentou broker. Komponenta broker je centralizovaná či federovaná a obsahuje kanály událostí. Příklad topologie je uveden na obrázku 4.3.



Obrázek 4.2: Topologie mediator[7]



Obrázek 4.3: Topologie broker[7]

Tato architektura velmi dobře škáluje a umožňuje škálovat jednotlivé procesory (konzumenty) samostatně. Nasaditelnost je zde také dobrá, ovšem při výběru topologie mediator můžeme narazit na větší potřebu synchronizace nasazení jednotlivých komponent. Jelikož procesory jsou zde zcela nezávislé, změny provedené v jednom procesoru neovlivňují další a to zjednodušuje reakce na změnové požadavky.

Tyto výhody jsou ovšem vykoupeny složitějším vývojem a testováním, které vychází z asynchronnosti a distribuovanosti. Při vývoji je potřeba myslet na to, že pořadí zpráv nemusí být správné, že zprávy mohou přijít vícekrát apod. Dalé je také náročnější analyzovat selhání, kdy je potřeba analyzovat asynchronní komunikaci.

4.1.3 Microservices

Tento styl můžeme popsat následujícím způsobem[8]:

> Vývoj aplikace jako malých, samostatně běžících, služeb, které spolu komunikují pomocí lightweight mechanismů (HTTP/REST, RPC). Tyto služby mají své byznysové zaměření a jsou samostatně nasaditelné.

Klíčovými koncepty této architektury jsou[7]:

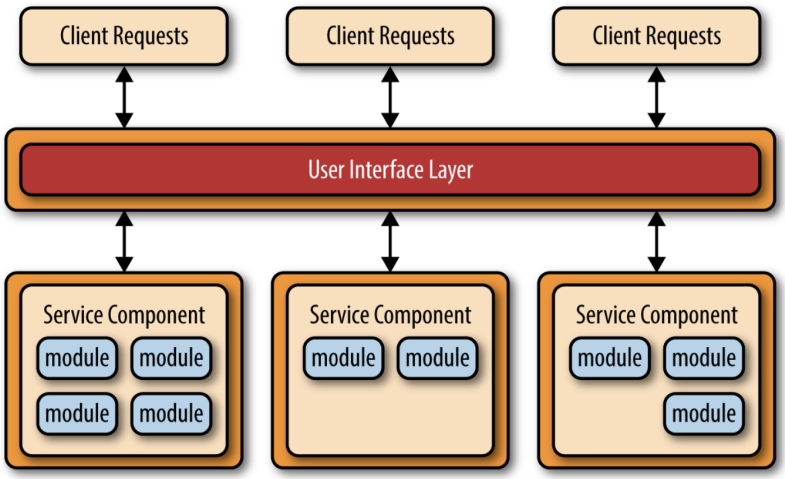
- **Samostatně nasaditelné jednotky**. Každá komponenta je samostatně nasaditelnou jednotkou. Zde se klade důraz i na možnost automatizace.
- **Servisní komponenta**. V rámci vývoje jednotlivé služby uvažujeme jako servisní komponenty s různou mírou granularity. Každá servisní

komponenta pak obsahuje moduly, které představují nezávislou část aplikace.

- **Distribuovanost**. Jedná se o distribuovanou architekturu, která s sebou nese všechny výhody i nevýhody distribuovaných architektur.

Z konceptu servisních komponent nám vychází, že je dělena do komponent dle jejich byznysových zaměření. Tento přístup koresponduje s osou Y na *ScaleCube*[1]. Rozdělení na komponenty se správnou granularitou je na tomto stylu to nejnáročnější. Pokud bude zaměření jednotlivých komponent moc široké, ztrácíme výhody tohoto stylu. Pokud budou naopak úzce zaměřené, bude architektura klást větší nároky na komunikaci a bude potřeba zavádět nějaké centrální prvky pro orchestraci. Poté už nám taková architektura spadá spíše do stylu SOA.

Styl je znázorněn na obrázku 4.4. Obrázek ilustruje obecnou podobu tohoto stylu. Některými z dalších možností je použití API (API Gateway) namísto User Interface Layer, případné přidání přidání komponenty pro messaging.



Obrázek 4.4: Microservices architektura[7]

Jednou z výhod tohoto stylu, která vychází z jeho klíčových konceptů, je možnost nasazovat samostatně jen změněné části systému. Pokud máme služby A,B,C,D, upravíme A a B, tak můžeme nasadit jen ty. Tato vlastnost zvyšuje dostupnost systému, protože není nutné dělat velké odstávky (pokud je spuštěných více komponent A a B, je možné dělat plně bezodstávkové nasazení). S tím ovšem přichází také zvýšená komplexita procesu nasazení, protože se jedná o více komponent.

Architektury s tímto stylem velmi dobře reagují na změny. Tyto architektury také dobře škálují a můžeme škálovat na úrovni jednotlivých servisních komponent, což nám umožní škálovat například jen ty funkcionality, které

[1] https://microservices.io/articles/scalecube.html

jsou vysoce vytíženy. Zároveň díky rozdělení na více procesů, pokud jedna z komponent selže, neselžou i další.

Co se vývoje týče, separace logiky ulehčuje vývoj a testování. Ovšem to, že se jedná o distribuovanou architektruru, přidává velké množství komplexity a situací, které je potřeba v rámci vývoje vyřešit (například nedostupnost komponenty, zpoždění, nedoručení zpráv, atd.).

4.1.4 SOA

Servisně orientovaná architektura, podobně jako microservisní, cílí na rozdělení řešení do menších, specializovaných služeb. Formálně ji můžeme definovat následovně[9]:

> SOA je forma technologické architektury, která lpí na principech servisní orientace. Pokud je realizována prostřednictvím technologické platformy webových služeb, SOA vytváří potenciál pro podporu a vylepšení těchto principů pomocí řídícího procesu a automatizace domén podniku.

Klíčovými koncepty SOA jsou[9]:

- Volná vazba - minimalizace vazeb služeb, vyžadujeme pouze, aby o sobě služby věděly.
- Kontrakt služeb - služby zachovávají definovaný kontrakt na komunikaci.
- Samospráva - služby mají kontrolu nad logikou, kterou zapouzdřují.
- Abstrakce - služby mají zapouzdřenou svoji logiku a dostupné je jen to, co definuje kontrakt.
- Znovupoužitelnost - zapouzdřená logika podporuje znovupoužitelnost.
- Kompozice - služby lze uspořádat a složit tak, aby vytvořily spojené služby.
- Bezstavovost - služby minimalizují potřebu uchovávat stav.
- Zjistitelnost - služby jsou navržené tak, že je lze vyhledat a zhodnotit dostupným vyhledávacím mechanismem.

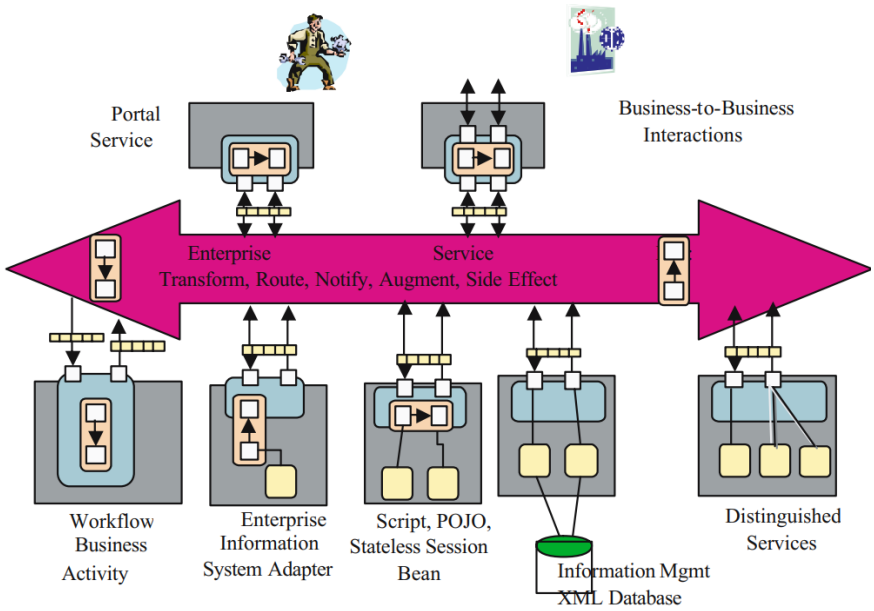
Jak můžeme vidět, podoba s microservisní architekturou je opravdu velká. O to důležitější je znát rozdíly mezi těmito dvěma styly. Jedním z rozdílů je rozsah řešení. Microservisní architektura cílí na architekturu aplikace, zatímco SOA cílí na podnikovou architekturu. To můžeme vidět i ve výše popsané definici, která mluví o "automatizaci domén podniku". Pokud se na to díváme tímto způsobem, můžeme říci, že v rámci SOA můžeme použít microservisní architekturu pro jednotlivé aplikace.

Dalším rozdílem, který ze zvýše zmíněného plyne, je rozdílný způsob komunikace. V kapitole Microservices je uvedeno, že služby mezi sebou komunikují pomocí lightweight mechanismů. V SOA je větší důraz kladen na integraci

s chytrými routovacími mechanismy napříč celým podnikem[10]. Zde můžeme mluvit například o technologii ESB, která umožňuje krom routování navíc i transformace, správu byznysových procesů a další.

Jednou z výhod SOA je, že pro ni existuje velké množství standardních technologií a formátů. Ať už se jedná o WSDL pro popis služeb a jejich registraci pomocí UDDI, přes transakce (WS-Transaction) až po zabezpečení (WS-Security).

Styl je znázorněn na obrázku 5.2.



Obrázek 4.5: SOA[11]

4.2 Architektonické vzory a taktiky

Architektonické vzory můžeme definovat následujícím způsobem[3]:

> Architektonický vzor popisuje opakující se designový problém vyskytující se v určitém kontextu a poskytuje osvědčené architektonické řešení pro tento problém. Toto řešení je specifikováno popisem rolí komponent, ze kterých se skládá, jejich zodpovědností, vztahů a způsobu komunikace.

Tento popis je velice podobný popisu klasických návrhových vzorů, které jsou známé v softwarovém inženýrství, rozdílem zde je jejich zaměření. Architektonické vzory se zabývají problémy architektury, tj. celkové struktury systému a ne jen konkrétních modulů.

Architektonické taktiky jsou **designová rozhodnutí** vedoucí k dosažení požadavků na atributy kvality. Na architektonický vzor se můžeme dívat také jako na balíček architektonických taktik. Taktiky jsou důležité, protože ne

vždy existují vzory, které bychom mohli použít, a tak je vhodné znát i samotné taktiky, kterými můžeme našeho cíle dosáhnout. Příkladem taktiky může být *Heartbeat*, který do systému zavádí periodickou komunikaci mezi systémovým monitorem a komponentou a zajišťuje detekci selhání komponenty. Příkladem vzoru je *Circuit breaker*, který v sobě zakomponovává taktiku opakovaných pokusů o komunikaci. Tento vzor brání systému pokoušet se o komunikaci donekonečna a po určitém počtu opakování se rozhodne, že se jedná o selhání. V této kapitole budou zmíněny pouze návrhy a taktiky relevantní pro tuto práci, nejedná se o úplný výčet.

4.2.1 Aplikační metriky

Problémem, který tento vzor řeší, je sběr údajů tak, aby bylo možné říci, co se v systému děje. Řešením je přidat do služby mechanismus pro sběr statistik[39]. Následně do systému přidáme službu, která bude sloužit jako kolektor a úložiště těchto statistik. Tato služba následně může poskytovat také upozornění a přehledy. Pro kolekci metrik je možné využít dva způsoby komunikace - pull a push. V prvním případě podpůrná služba sama stahuje metriky ze služeb, v druhém posílají služby metriky do podpůrné služby.

Nevýhodou vzoru je nutnost přidávat sběr statistik do byznysové logiky a potřeba zavedení další služby, což může klást vyšší nároky na infrastrukturu.

4.2.2 Agregace logů

Při použití microservisní architektury máme několik služeb, které běží na různých strojích a generují logy ve standardním formátu[39]. Problém, který tento vzor řeší, je jak pochopit, co se v systému děje a jak porozumět problémům, které se v systému vyskytnou. Řešením je použití centrální služby, která logy z jednotlivých služeb agreguje. Tato služba následně poskytuje prostředky pro analýzu logů a jejich prohledávání. Případně může umožňovat upozornění na základě automatické analýzy logů.

Nevýhodou vzoru jsou vyšší nároky na infrastrukturu. Je potřeba zavedení dalších podpůrných služeb a navýší se objem komunikace. Dle retence logů mohou být kladeny také větší nároky na úložiště.

4.2.3 Distributed tracing

Pokud máme požadavek, který je zpracováván více službami, není jednoduché dohledat a přehledně vidět průběh jeho zpracování. Řešením je vygenerování unikátního ID, které požadavku na začátku zpracování přidělíme a následně ho posíláme spolu s daty všem službám, které ho dále zpracovávají[39]. Všechny služby toto externí ID vloží do všech logů týkajících se zpracování tohoto požadavku. To nám následně poskytuje přehled zpracování napříč službami. Pokud použijeme i vzor Agregace logů, můžeme pro daný požadavek vidět zpracování přehledně na jednom místě.

Nevýhodou vzoru jsou vyšší nároky na infrastrukturu. Díky dodatečné informaci se nám zvyšují nároky na úložiště logů.

4.2.4 Health check API

Health check API vzor řeší problém detekce schopnosti služby zpracovat požadavek. Služba může běžet, ovšem nemusí být schopna zpracovat požadavek (například pro nedostatek zdrojů). Na službu v tomto stavu nechceme posílat další požadavky.

Řešením je zavedení API, které umožní kontrolu stavu služby. Ta může poskytovat informace nejen o prostředcích, ale i o možnosti komunikace s dalšími službami či specifické údaje spojené s konkrétní aplikací. Toto API mohou využívat další podpůrné služby jako monitoring, load balancer či service registry. Na základě poskytnutých údajů je možno určit, zda je možné na službu dále posílat požadavky či zda mají být směrovány jinam.

Nevýhodou vzoru je způsob dotazování na stav služby. To bývá zpravidla implementováno periodickým dotazováním. Informace o stavu služby tak není kontinuální a může se stát, že mezi dvěma dotazy dojde k neschopnosti služby plnit požadavky a my se o tom nedozvíme.

4.2.5 Service registry

Problémem, který tento vzor řeší, je nutnost znalosti dostupných instancích, pokud chceme použít klientské nebo serverové service discovery[39].

Řešením je zavedení služby, která bude fungovat jako databáze dostupných služeb. Do ní se budou služby při spuštění registrovat a při vypínání se odregistrují. Tato databáze může být dále použita klientskou/serverovou service discovery. Pro případ selhání služby bez odregistrování může service registry používat health check API (pokud je tento vzor použit) pro detekci dostupnosti služby a její schopnosti plnit požadavky.

Nevýhodou vzoru je, že nově zavedená služba je kritická a je potřeba věnovat velkou pozornost její dostupnosti. V případě nedostupnosti této služby nebude možná komunikace mezi službami v rámci systému. Toto se dá mitigovat zavedením cache, ovšem při delším výpadku služby to není dostatečné řešení.

4.2.6 Klientské service discovery

Problémem, který tento vzor řeší, je způsob, jakým jedna služba může volat jinou, resp. jak zjistit její lokaci[39].

Řešením je volání service registry (service registry vzor je prerekvizita pro tento) pro získání lokace volané služby. Následně pak může odeslat požadavek na cílovou službu. Pro optimalizaci komunikace je možné zavést na straně klienta cache.

Nevýhodou je vazba klienta a service registry, dále také nutnost implementovat tuto logiku pro všechny použité frameworky a jazyky.

4.2.7 Serverové service discovery

Tento vzor řeší stejný problém jako předchozí vzor. Vzor zavádí novou komponentu, která má známou lokaci pro všechny služby a všechny požadavky

ze služeb na jiné služby jsou prováděny přes ní. Tato služba je následně také ta, která komunikuje se service registry.

Nevýhodou vzoru je zavedení nové komponenty, pro kterou je důležitá vysoká dostupnost. Zavedení prostředníka pro komunikaci také zvyšuje počet síťových skoků při komunikaci a může zvýšit latenci. Není ovšem potřeba implementovat logiku na klientovi.

4.2.8 Databáze per služba

Problémem, který tento vzor řeší, je alokace databází ke službám[39]. Ideálně chceme, aby data služby byly nezávislé, nezávisle škálovatelné a nezávisle nasaditelné.

Řešením je zavedení databáze pro každou službu. Tím jsou data každé služby privátní a přístup k nim je možný jen na základě rozhraní poskytovaného službou. Toto řešení poskytuje vysokou úroveň zapouzdření a flexibilitu. Je možné například pro každou službu použít přesně ten typ databáze, který se hodí pro její data.

Nevýhodou řešení je správa několika různých druhů databází a implementace byznysových transakcí. Neprimitivními se stávají také dotazy, které potřebují data z více služeb. Vliv na infrastrukturu a zdroje se odvíjí od způsobu řešení zapouzdření dat. Zde je několik variant, kde každá přináší jiné množství přidané zátěže (od nejvyšší po nejnižší):

- databázový server per služba,
- schéma per služba,
- privátní tabulky per služba.

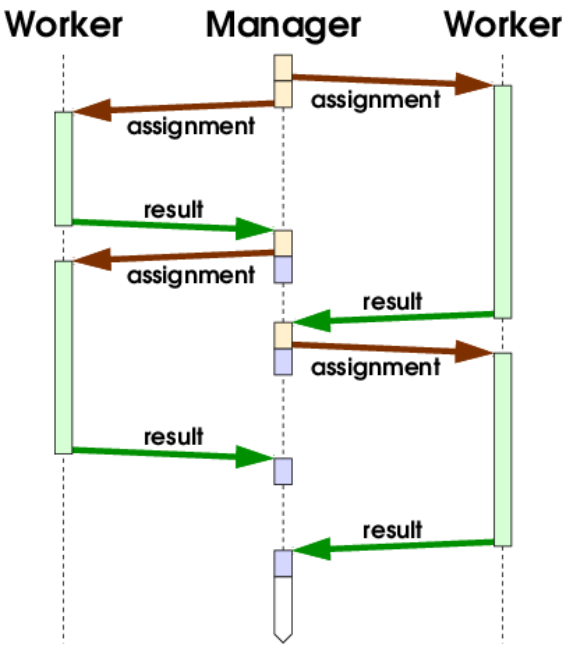
4.2.9 Sdílená databáze

Vzor řeší stejný problém jako předchozí. Řešením je jedna sdílená databáze pro všechny služby, kde služby mohou používat a manipulovat s daty jiných služeb pomocí lokálních transakcí[39]. Nevýhodou je nižší flexibilita, zvýšená závislost na implementaci jednotlivých služeb a potřebná koordinace vývoje a nasazení. Stejně tak se mohou služby navzájem ovlivňovat výkonnově.

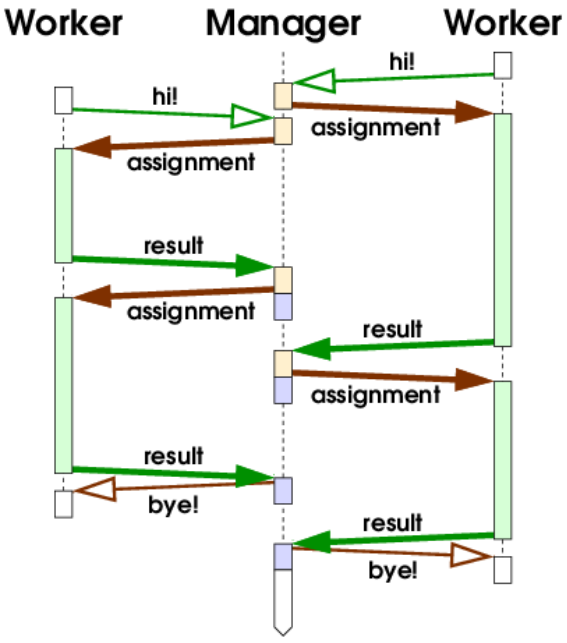
4.2.10 Manager-Worker

Problémem, který tento vzor řeší, je rozdělení práce mezi více tzv. workerů[40]. Jedná se o architektonický vzor pro paralelní zpracování dat.

Tento vzor zavádí komponenty Manager a Worker. Manager je zodpovědný za správné rozdělení práce a sada identických workerů za její odvedení. Tento vzor má dvě základní podoby - push a pull. Push je znázorněn na obrázku 4.6. Jedná se o variantu, kdy manager ví o jednotlivých instancí komponent worker a posílá jim proaktivně práci. Varianta pull je na obrázku 4.7. V této variantě nemusí manager vědět dopředu o lokaci workerů a práci jim přidělí, když si o ni řeknou.



Obrázek 4.6: Varianta push[41]



Obrázek 4.7: Varianta pull[41]

4.2.11 Heartbeat

Hearbeat je mechanismus pro detekci selhání komponenty, který do systému zavádí periodickou výměnu zpráv mezi monitorovaným procesem a hlídajícím

procesem [3]. Pokud stavíme škálovatelné systémy, je možné heartbeat zakomponovat do jiných zpráv, abychom zbytečně nezatěžovali komunikační kanál. Pokud zpráva nepřijde do určitého časového limitu, hlídající komponenta označí proces za selhaný a signalizuje tuto skutečnost.

4.2.12 12-factor application

Nejedná se přímo o architektonický vzor, spíše o metodologii a řadu doporučení, jak navrhovat aplikace jako služby tak, aby podporovaly automatizaci a škálování[42].

Kód

Aplikace splňující 12 faktorů je vždy verzována ve verzovacím systému (Git, SVN, ..). Aplikace je vždy verzována jen v jednom repozitáři a má mnoho nasazení. Nasazení je běžící instance aplikace, například v testovacím či produkčním prostředí. Tato nasazení sdílí kód, ovšem může se jednat o různé verze (z pohledu verzování).

Závislosti

Aplikace splňující 12 faktorů má vždy explicitně vyjádřené všechny své závislosti a nese si je s sebou. Aplikace není závislá na závislostech nainstalovaných na běhovém prostředí a nepoužívá je, používá své, přibalené.

Konfigurace

Aplikace splňující 12 faktorů má vždy striktně oddělenou konfiguraci a kód. Konfigurací jsou například přístupové údaje, adresy, apod. Tuto konfiguraci aplikace dostává formou proměnných prostředí.

Podpůrné služby

Aplikace splňující 12 faktorů nerozlišuje lokálně provozované podpůrné služby od podpůrných služeb spravovaných třetími stranami. V obou případech je vnímá jako prostředky, ke kterým přistupuje pomocí URL a přístupových údajů poskytnutých pomocí konfigurace. Je tak možné mezi lokálními službami a službami třetích stran přecházet bez nutnosti zásahu do kódu.

Podpůrnými službami se myslí například databáze, STMP server, messaging služby či systémy pro caching.

Sestavení, vydání a spuštění

Aplikace splňující 12 faktorů striktně rozděluje fázi sestavení, vydání a spuštění aplikace. Sestavení transformuje kód do spustitelného balíčku. Vydání je fáze, kdy se kombinuje sestavený balíček s konfigurací pro konkrétní prostředí. Výsledkem této fáze je tzv. realease. Poslední fází je spuštění. Jedná se o spuštění konkrétního realease.

Procesy

Aplikace splňující 12 faktorů je spouštěna formou procesů, které spolu nic nesdílejí. Data, která chce aplikace ukládat, musejí být ukládány v podpůrných službách.

Vazba s portem

Aplikace splňující 12 faktorů je plně soběstačná a nespoléhá na poskytnutí webového serveru z prostředí pro poskytování webové služby. Aplikace sama poslouchá na daném portu a poskytuje na něm HTTP či jiné služby. Příkladem poskytnutí webového serveru z prostředí může být například provoz Java webové aplikace na aplikačním serveru.

Concurrency

Pro aplikaci splňující 12 faktorů je přirozené použití procesů. Různé úkoly mohou být delegované na různé procesy (například dlouhodobé úlohy či zpracování HTTP požadavků). Tyto je pak možné nezávisle na sobě škálovat.

Disposability

Aplikace splňující 12 faktorů jsou navrženy tak, aby je bylo možné kdykoliv rychle spustit a vypnout. Takto je možné rychle reagovat na potřeby nového nasazení, změny konfigurace, elastické škálování a obnovení po selhání.

Parita prostředí

Aplikace splňující 12 faktorů jsou navrženy pro continous deployment a tak, aby rozdíly mezi prostředími byly minimální. Pro aplikace splňující 12 faktorů je typické, že jsou nasazovány s periodou hodin, aplikace je implementována a nasazována stejnými lidmi a diference mezi různými prostředími je minimální.

Logy

Aplikace splňující 12 faktorů se nestará o své logy. Aplikace pouze posílá logy na standardní výstup a následné zpracování, routování, ukládání a podobné operace má na starosti běhové prostředí.

Správcovské procesy

Aplikace splňující 12 faktorů používá pro administrativní úlohy procesy. Administrativními procesy se myslí například migrace databáze, opravy dat a další. Tyto úlohy jsou obvykle implementovány jako skripty, které jsou verzovány spolu s kódem aplikace tak, aby byly synchronizovány.

4.3 Technologie

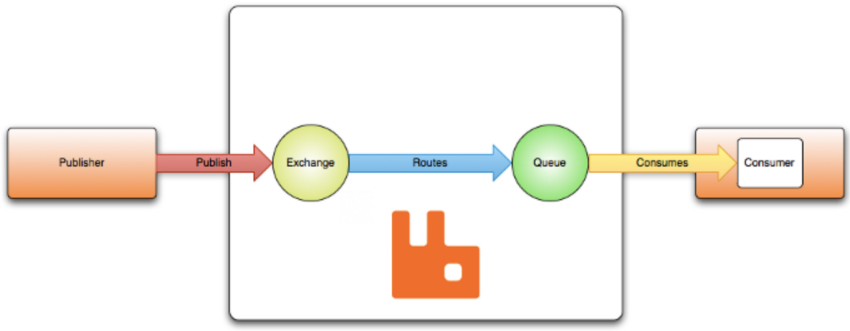
4.3.1 Komunikace

RabbitMQ

Jedná se o open source message broker implementující AMQP protokol[14] a dnes i další protokoly, jako například STOMP. Podporuje vysokou dostupnost díky replikacím a je možné jej provozovat v clusteru a škálovat přidáváním nových uzlů. RabbitMQ je navrženo pro téměř prázdné fronty a v případě akumulace velkého množství zpráv jeho výkonnost značně degraduje[13]. High level design RabbitMQ se skládá z následujících komponent:

- publisher - ten, kdo zprávy odesílá na brokera,
- broker - přijímá zprávy od publishera a provádí routing ke konzumentům,
- consumer - aplikace konzumující zprávy.

Zprávy jsou odeslané na tzv. exhange, ze které jsou dále pomocí routovacích pravidel rozkopírovány na příslušné fronty. Broker poté zajistí doručení zpráv z front ke konzumentům (ať už pomocí pull nebo push mechanismu). Výše popsané je vyobrazeno na obrázku4.8.



Obrázek 4.8: Příklad komunikace na RabbitMQ[17]

Mezi výhody RabbitMQ patří jeho rozšířenost (používán velkým množstvím vývojářů), má klientské knihovny pro většinu mainstream programovacích jazyků a dobrou dokumentaci.

Apache Kafka

Apache Kafka je škálovatelná a vysoce dostupná platforma pro distribuované streamování dat[15]. Zprávy jsou uchovávány na serverech a organizovány do tzv. *topics*. Topic je v podstatě log, do kterého se dá pouze přidávat na konec (proto se také kafka někdy nazývá "distributed message log"). Apache Kafka má několik případu užití, mezi jinými například[16]:

- tracking aktivity uživatelů,
- messaging,
- kolekce metrik a logů,
- commit log - například sledování WAL[2] a odesílání změn na kafku,
- streamové zpracování dat.

Ze zvýše zmíněných případů užití můžeme vyvodit, že se jedná hlavně o platformu pro práci s událostmi, kterou je možno použít i pro messaging. Mezi výhody kafky patří vysoká propustnost a nízká latence i při persistenci zpráv (narozdíl od RabbitMQ).

Pulsar

Pulsar je distribuovaný publish-subscribe messaging systém s nízkou latencí, bezztrátovostí dat a serverless framework pro streamové zpracování dat[18]. Pulsar mimo jiné nabízí možnost dlouhodobého uložení zpráv, uživatelsky definovaných transformací přímo v Pulsaru, bez potřeby dalších aplikací a real-time messaging. Podobně jako jiné messaging systémy, i Pulsar používá koncept tzv. *topics*. Topic je zde ovšem až na konci komplexnější hieararchie konceptů:

- Tenant - může prezentovat například konkrétní aplikaci (jedná se obecně o multitenantní systém). Každý tenant může mít vlastní autentizační a autorizační schéma, TTL zpráv či nastavení úložiště.
- Namespace - Prostředek pro logické seskupení topiců.
- Topic - Pojmenovaný kanál pro přenos zpráv mezi producenty a konzumenty zpráv.

Pulsar je relativně nová technologie (open-source od roku 2016) a proto není ještě tak rozšířená (hlavně z pohledu komunity vývojářů). Nabízí ovšem vysokou dostupnost, elastické škálování a byl navrhován pro cloudová prostředí.

4.3.2 Topologie a service discovery

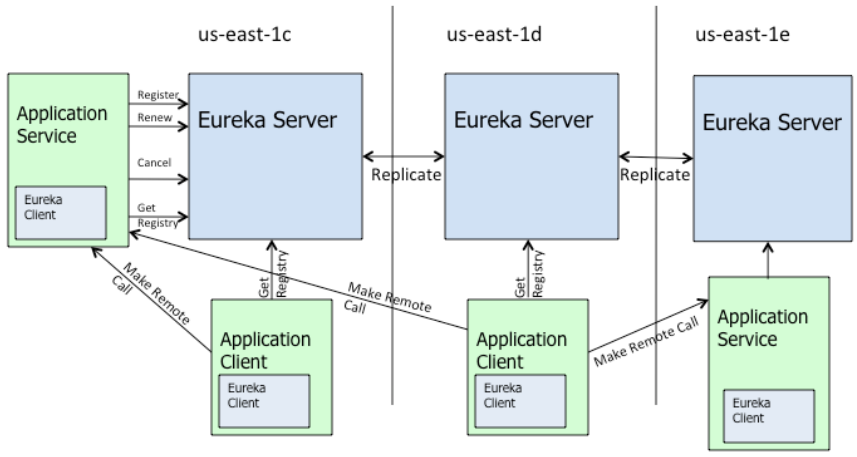
Netflix Eureka

Eureka je služba založená na protokolu REST (Representational State Transfer), která se v AWS cloudu používá především k vyhledávání služeb pro účely vyvažování zátěže a převzetí služeb při selhání serverů střední vrstvy[23]. Krom serverové služby (tzv. Eureka Server) poskytuje i Java knihovnu pro klientské aplikace (tzv. Eureka Client). Load balancing je možné provádět i na straně klienta. Velkou výhodou je jednoduchá integrace v rámci frameworku

[2] https://www.postgresql.org/docs/current/wal-intro.html

Spring[3]. Pokud při implementaci není použit framework Spring nebo jazyk Java, je možné se na Eureka Server integrovat manuálně, pomocí jeho REST API.

Princip Netflix Eureka je následující. Služby se registrují na Eureka Serveru a následně posílají heartbeat každých 30s. Pokud server od služby heartbeat nedostane v daném časovém limitu, odebere ji z registru. Tento registr (stav) je replikován napříč všemi Eureka servery v rámci clusteru. Klient se pak může následně zeptat Eureka serveru na umístění služby, kterou chce volat a poté ji může zavolat. Princip je znázorněn na obrázku 4.9.



Obrázek 4.9: Princip fungování Eureka service discovery[23]

Consul

Consul od společnosti HashiCorp je service mesh[4] řešení poskytující service discovery, konfiguraci a segmentaci[19]. Mezi poskytované funkcionality patří:

- service discovery,
- healt check (například kontrola, zda je služba "naživu", či zda není spotřeba zdrojů na uzlu moc vysoká),
- KV store - consul poskytuje jednoduché HTTP API pro přístup ke key-value úložišti, které je možné použít mj. například pro konfiguraci,
- generování a distribuce TLS certifikátů pro bezpečnou komunikaci.

Consul je distribuovaný systém mířící na vysokou dostupnost. Při použití jsou na jednotlivé uzly v systému nasazeny tzv. *Consul agenti*. Další komponentou je *Consul server*. Agenti zprostředkovávají health checking, servery si vedou

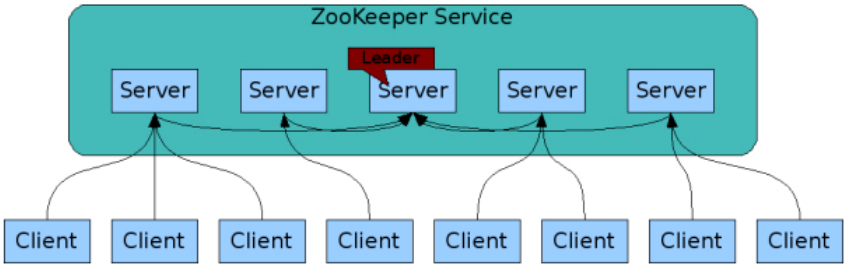
[3]https://docs.spring.io/spring-cloud-netflix/docs/current/reference/html/
[4]https://buoyant.io/service-mesh-manifesto/

katalog služeb a uchovávají data. Servery mohou být, pro dosažení vysoké dostupnosti, nasazeny v clusteru, ve kterém následně probíhá leader election.

Apache ZooKeeper

ZooKeeper je vysoce výkonná koordinační služba pro distribuované aplikace. V jednoduchém rozhraní vystavuje běžné služby, jako je pojmenování, správa konfigurace, synchronizace a skupinové služby, takže je není nutné psát od začátku[20]. ZooKeeper je replikovaný4.10 a používá leader election pro zvolení Leader uzlu. Procesu spolu sdílejí data přes ZooKeeper pomocí hieararchického namespace, který je organizovaný podobně jako souborový systém (cesta, "/"použito jako separátor). Tento Namespace se skládá z tzv. znodes, což jsou registry dat a jsou podobné složkám a souborům na souborovém systému. Data jsou držena v paměti, pro dosažení nízké latence a vysoké propustnosti.

ZooKeeper samotný tak není službou pro service discovery, ale je možné ji pomocí něj implementovat. K tomu je možné využít například Apache Curator, což je klientská knihovna poskytující předpřipravené implementace a rozšíření pro ZooKeeper, jako napříkald service discovery[21].



Obrázek 4.10: Replikace v ZooKeeper[20]

Nativní kubernetes SD

Pokud aplikaci nasazujeme na kubernetes cluster, je možné, že se naše aplikace bude přesouvat mezi pody (např. po pádu) a bude se měnit její IP adresa. V takovém případě je potřeba mít service discovery. Na toto ovšem myslí již samotné kubernetes a poskytuje nástroj, tzv. Service, pomocí kterého lze tuto situaci řešit. Service je v systému Kubernetes abstrakce, která definuje logickou sadu modulů Pod a politiku přístupu k nim (někdy se tento vzor nazývá mikroslužba)[22].

Kubernetes následně poskytuje možnosti, jak ke službám přistupovat. Zde zmíníme 2 způsoby:

- Dotazování API serveru a ten nám následně vrátí endpointy. Ty jsou aktualizovány pokaždé, když se změní množina Podů, kterou zastřešuje.
- DNS - pokud máme v našem clusteru DNS server, který s ním umí pracovat (komunikuje s API serverem), bude naší službě přiřazeno DNS

jméno, které můžeme následně používat pro komunikaci se službou.

4.3.3 Monitoring, správa logů

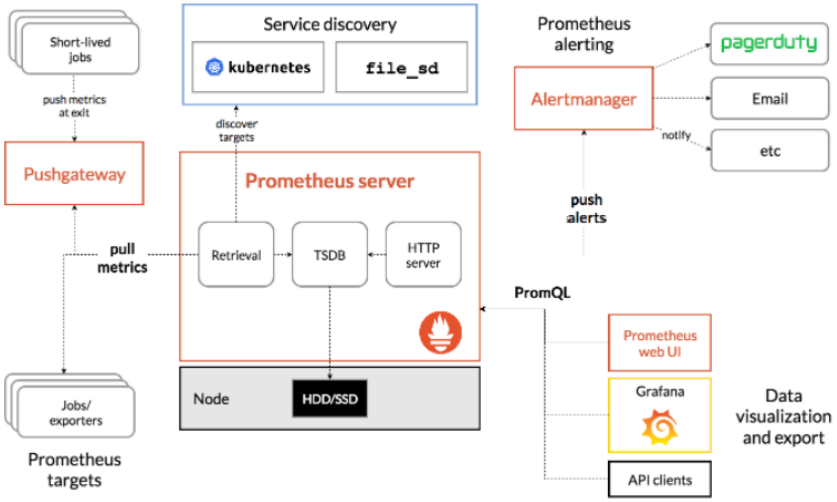
Grafana

Grafana je open source vizualizační a analytický software. Umožňuje dotazování, vizualizaci, upozorňování a zkoumání metrik, protokolů a sledovacích záznamů bez ohledu na to, kde jsou uloženy. Poskytuje nástroje, které umožňují proměnit data z databáze časových řad (TSDB) v přehledné grafy a vizualizace[24]. Obecně umožňuje číst data z různých zdrojů a má nativně podporu pro Prometheus, InfluxDB, Loki, různá RDBMS a mnoho dalších. Hlavním prvkem uživatelského rozhraní je tzv. dashboard, který se skládá z panelů. Pro panel následně můžeme nakonfigurovat dotaz na data a jejich vizualizaci. Nástroj je flexibilní a je možné pomocí něj vytvořit přehled stavu celého systému.

Prometheus

Prometheus je open-source toolkit pro monitorování systémů a upozorňování[25]. Sbírá data pomocí dotazů (pull, push, tj. odesílání metrik do promethea, je možný, avšak nedoporučovaný) a ukládá je jako časové řady, tj. informace jsou označeny časovou značkou a následně key-value páry informací. Těm se říká *labels*. Krom manuální konfigurace služeb, které má monitorovat, podporuje taky jejich automatické objevování přes service discovery. Na data je následně možné se doptávat pomocí jazyka PromQL. Prometheus také umožňuje nastavit pravidla pro upozornění, která mohou být dále odeslána do komponenty AlertManager, která může přes další kanály (email, Slack, ..) distribuovat.

High level design Promethea je zobrazen na obrázku 4.11.



Obrázek 4.11: Prometheus high level design[25]

InfluxDB a TICK stack

InfluxDB je open-source databáze časových řad bez schématu s volitelnými closed-source komponentami vyvinutá společností InfluxData[26]. Nabízí SQL-like dotazovací jazyk pro dotazy nad daty. V rámci InfluxDB je několik základních konceptů:

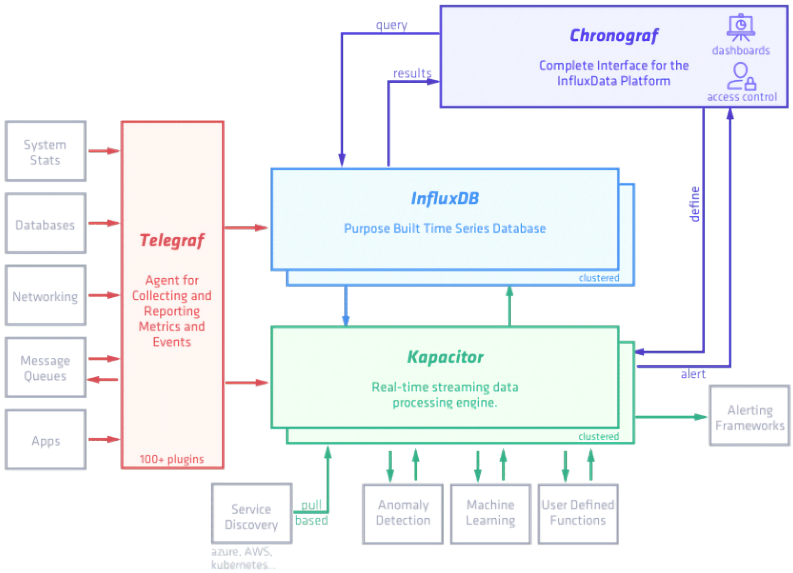
- čas - sloupec s časovou značkou je se všemi daty,
- pole (field) - atribut, skládá se z klíče (můžeme si představit název sloupce) a hodnoty (hodnota v daném sloupci v daném čase),
- tag - popisem a strukturou identické s polem.

Ač pole a tag jsou strukturou identické entity, rozdílem je, že tagy jsou indexovány a pole ne. Tagy se tedy hodí pro vyhledávání. Dále časová značka a tagy tvoří unikátní klíč záznamu.

InfluxDB můžeme doplnit dalšími komponentami, který spolu s ní tvoří tzv. TICK stack. Těmito komponentami jsou:

- Telegraf - agent pro kolekci metrik a jejich odesílání do datových úložišť (IndexDB v tomto případě, ale podporuje i mnoho jiných),
- Chronograf - nástroj pro administraci a vizualizaci (alternativa k nástroji Grafana),
- Kapacitor - engine pro následné zpracování (streamové i dávkové) dat z InfluxDB s podporou uživatelské logiky.

TICK stack je zobrazen na obrázku 4.12.



Obrázek 4.12: TICK stack[26]

ELK stack a beats

ELK stack je kompletní řešení pro analýzu logů, postavené na kombinaci třech open-source nástrojů - Elasticsearch, Logstash a Kibana[27].

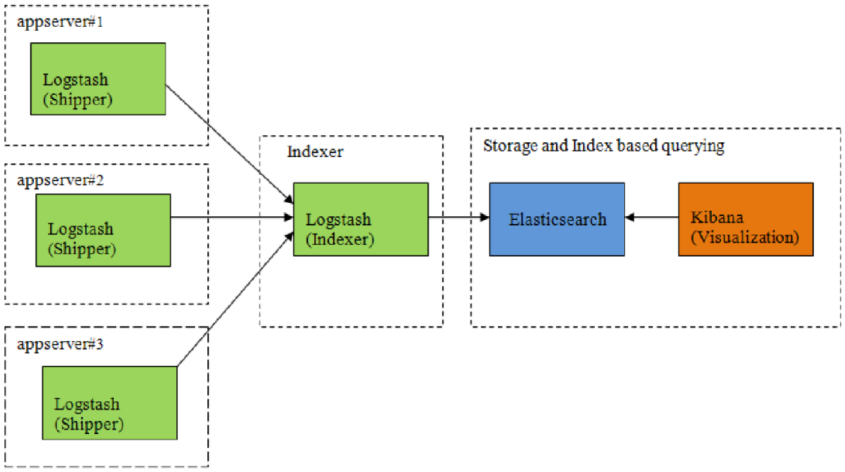
První komponenta, Elasticsearch, je vyhledávací engine založený na technologii Apache Lucene[5]. Je škálovatelná[27] a poskytuje REST API.

Logstash je komponenta, která pomáhá sbírat, parsovat, analyzovat a transformovat data. Její architektura umožňuje používat různé pluginy pro různé vstupní zdroje a cíle výstupu (např. logy na vstupu a Elasticsearch na výstupu).

Kibana je nástroj pro vizualizaci strukturovaných a nestrukturovaných dat. Celý nástroj je napsán v HTML a JavaScriptu. Poskytuje různé pohledy na data (histogramy, koláčové grafy, mapy, apod.). Zároveň umožňuje i uživatelem definované vizualizace. Typické nasazení ELK stacku je zobrazeno na obrázku 4.13.

Na obrázku 4.13 vidíme, že na každém uzlu, ze kterého chceme sbírat logy, je nainstalován Logstash, který data odesílá do jiné instance Logstashe, která data dále transformuje a filtruje, před odesláním do Elasticsearch. V době cloudového nasazení a škálování může být takovéto nasazení náročnější na zdroje a v roli "Shipper" nejsou využity, a ani nejsou potřebné, všechny možnosti Logstashe. Dle současného trendu se na uzlech Logstash nahrazuje komponentami, kterým se říká *Beats*. Jedná se o kolekci open-source odesílatelů logů, kteří fungují jako agenti nainstalovaní na uzlech, odkud odesílají logy a metriky[28]. Mezi tyto Beats patří například Filebat (pro odesílání aplikačních logů), Packetbeat (pro zachycení a odesílání síťové komunikaci)

[5] https://lucene.apache.org/



Obrázek 4.13: ELK stack[27]

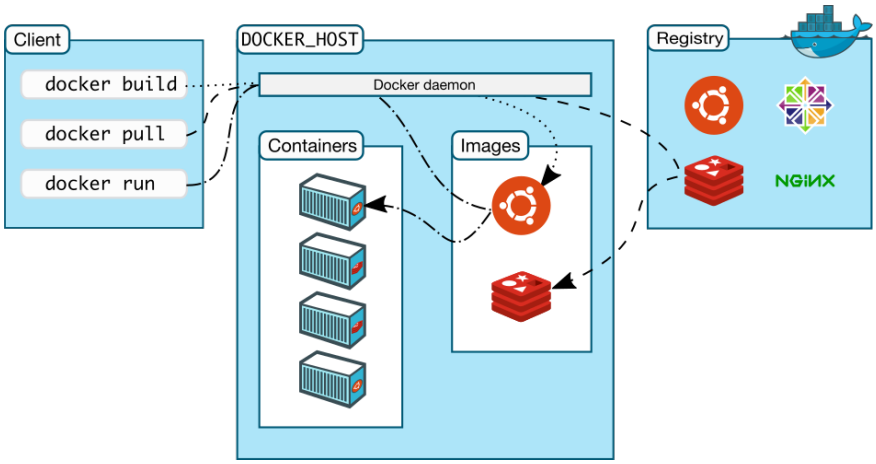
či Metricbeat (pro odesílání metrik systému).

4.3.4 DevOps a infrastruktura

Docker

Docker je open-source platforma pro vývoj, dodání a provoz aplikací[29]. Umožňuje zabalení aplikací to tzv. kontejnerů - spustitelný balíček obsahující naší aplikaci a její závislosti.

Používá client-server architekturu. Klienti komunikují s Docker *daeomn* komponentou, která zajišťuje úkony okolo sestavení, spouštění a distribuce kontejnerů. Ke komunikaci používají REST API, unixové sockety či síťové rozhraní (klient a server nemusí být nutně spuštěny na stejném stroji). Architektura je znázorněna na obrázku 4.14.



Obrázek 4.14: Docker architektura[29]

Na obrázku 4.14 jsou krom klienta a *daemona* také další objekty:

- image - šablona pro vytvoření kontejneru,
- kontejner - běžící instance konkrétního image; krom image je definován konfigurací poskytnutou při vytvoření/spuštění.

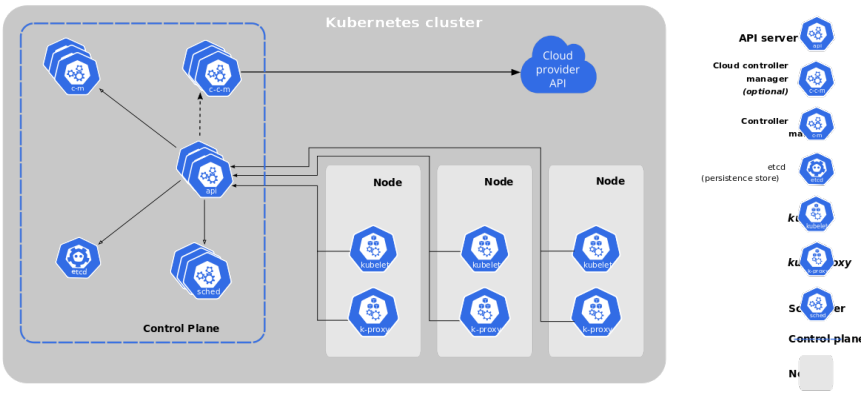
Kubernetes

Kubernetes je přenosná, rozšiřitelná open source platforma pro správu kontejnerových úloh a služeb, která umožňuje deklarativní konfiguraci a automatizaci[30]. Pokud pro nasazení našich aplikací používáme kontejnery, máme jich velké množství, snažíme se o škálovatelné a vysoce dostupné řešení, jejich manuální správa přestává být možná. Nástroje pro orchestraci kontejnerů, jako je Kubernetes, umožňují mnohé úlohy automatizovat. Mezi tyto úlohy patří:

- nasazování kontejnerů,
- dostupnost a redundance,
- automatické škálování na základě zátěže,

- přesun kontejnerů mezi uzly,
- load balancing a service discovery,
- správa konfigurace (včetně citlivých údajů, jako například hesel).

Architektura kubernetes je znázorněna na obrázku 4.15.



Obrázek 4.15: Kubernetes architektura[30]

Základem architektury a clusteru jsou *worker* zařízení, kterým se říká uzly, na kterých jsou spouštěny kontejnery. Na těchto uzlech existují následující komponenty:

- Pod - nejmenší nasaditelná jednotka v Kubernetes. Jedná se o skupinu kontejnerů, které jsou spuštěny pospolu ve sdíleném kontextu.
- Kubelet - agent, který dohlíží na běh a zdraví kontejnerů v Podu.
- Kube-proxy - síťová proxy, udržuje síťová pravidla na uzlu. Tato pravidla umožňují komunikaci mezi pody v rámci clusteru.
- *Container runtime* - běhové prostředí pro kontejnery. Zodpovídá za spouštění a běh kontejnerů. Může se jednat například o Docker.

Dále je zde řídící část clusteru, tzv. *Control plane*. Ta dělá rozhodnutí o tom, co se má stát v clusteru (například spuštění nového kontejneru) a detekuje události v clusteru (například selhání kontejneru). Obsahuje následující komponenty:

- API Server - poskytuje REST API pro ovládání clusteru.
- etcd - vysoce dostupné key-value úložiště, které je použito pro ukládání stavu clusteru a konfigurace.
- Scheduler - sleduje nově vytvořené Pody a pokud nejsou přiřazeny k žádnému uzlu, nalezne pro ně vhodný uzel (bere v potaz nároky specifikované uživatelem jako jsou třeba HW zdroje či preference některých uzlů před jinými).

- Controller manager - kontroluje stav clusteru (pomocí API serveru), porovnává ho s chtěným stavem. Pokud stav neodpovídá, provádí kroky, které ke chtěnému stavu povedou.

Helm

Helm je open source správce balíčků pro Kubernetes[37]. Helm používá pro balíčkování tzv. charts. Je to jednotka, kterou umí instalovat (resp. odinstalovat) do (z) Kubernetes. Dále spravuje životní cyklus balíčků, které byly pomocí Helm nainstalovány.

Základní koncepty Helm jsou:

- Chart - balíček informací potřebných k nasazení aplikace do Kubernetes,
- Configuration - obsahuje informace, které mohou být spojeny s charty k vytvoření nasaditelný objekt,
- Release - spuštěná instance chartu s konkrétní configuration. Helm si tyto objekty interně verzuje a je tak možné se vracet k předchozím verzím.

Dále existují dvě základní komponenty Helm:

- Helm Library - Komponenta provádějící logiku operací nástroje. Komunikuje s Kubernetes API, vytváří release z chart a configuration a poskytuje následné operace pro práci s release (upgrade, odinstalování).
- Helm Client - klientská aplikace pro koncové uživatele. Umožňuje lokální vývoj chartů, správu repozitářů, správu realeasů a komunikaci s Helm Library.

4.3.5 Jazyky a frameworky pro tvorbu komponent

V rámci této práce uvažujeme jen jazyky Java a Python. S oběma jazyky má tým zkušenosti a nepůsobí problémy s výkonem systému.

Spring Boot

Spring Boot umožňuje snadno vytvářet standalone, produkční aplikace založené na frameworku Spring[31]. Spring Boot je velmi pohodlný framework pro vývoj a poskytuje konfiguraci a integraci pro mnoho knihoven třetích stran out-of-the-box (z cloudových technologií například pro eureku či endpoint pro monitoring pomocí Prometheus).

Nevýhodou frameworku je, že dependency injection a vyhledávání existujících implementací probíhá při startu aplikace a pomocí reflexí a využívá Proxy pattern ve velké míře. Toto má dopad na paměťové nároky a dobu startu aplikace, což může mít vliv u cloud-native aplikací, které jsou zpoplatněny na základě zdrojů a doby běhu.

Micronaut

Micronaut je moderní, na platformě JVM založený, full stack Java framework určený pro vytváření modulárních, snadno testovatelných JVM aplikací s podporou jazyků Java, Kotlin a Groovy.[32]. Podobně jako Spring Boot, i Micronaut poskytuje konfiguraci a integraci pro mnoho nástrojů a knihoven třetích stran. Zároveň poskytuje vlastní CLI nástroj pro tvorbu projektů a modifikaci existujících (například tvorba nové Singleton bean). Jednou z výhod frameworku Micronaut, který jej odlišuje od Spring Boot, je minimální používání vzoru Proxy a dependency injection realizovaná v době kompilace. Tím snižuje nároky na paměť a zrychluje čas startu aplikace (výměnou za delší čas kompilace).

Quarkus

Quarkus je Kubernetes-native Java framework přizpůsobený pro GraalVM a HotSpot, vytvořený z nejlepších knihoven a standardů Javy[33]. Tento framework optimalizuje Javu tak, aby mohla být efektivně provozována v serverless, cloudových prostředích a na Kubernetes. Je možné v něm používat další frameworky a knihovny, jako například Spring, JPA, RESTEasy apod.

Quarkus umožňuje aplikace spouštět buďto v JVM módu či v kompilovaném módu (pro GraalVM). I v JVM módu mají aplikace nižší paměťové nároky a čas startu, než aplikace používající například Spring Boot. Podobně jako Micronaut se Quarkus snaží část práce přenést již do sestavování aplikace (například zpracování anotací, dependency injection, parsování konfigurace).

Nativní mód využívá tzv. ahead-of-time kompilace (za pomocí GraalVM), kdy kompiluje náš kód přímo do nativního kódu, místo do bytecode. Tím odpadá overhead použití JVM a JIT kompilace. V tomto módu jsou nároky na paměť a čas startu ještě nižší.

4.3.6 Databáze

PostgreSQL

PostgreSQL je výkonný objektově-relační databázový systém s otevřeným zdrojovým kódem, který využívá a rozšiřuje jazyk SQL v kombinaci s mnoha funkcemi, které bezpečně ukládají a škálují nejsložitější datové úlohy[34]. Jedná se o velmi rozšířenou a používanou technologii, která je bez licenčních poplatků. Podporuje standardní relační ukládání dat v tabulkách i ukládání a dotazování nad JSON daty.

Podporuje replikaci dat single-master i multi-master. V prvním případě pouze jedna instance provádí změny, které se pak replikují na další instance. V druhém případě mohou být změny prováděny na více instancích, je ale potřeba aplikovat techniky řešící kolize. Dále je zde možnost replikace provádět v synchronním či asynchronním módu. V synchronním módu uzel čeká, než repliky přijmou změnu a až poté klientovi potvrdí úspěšný zápis, zatímco v asynchronním módu se na přijetí změn nečeká[35].

Jedná se tedy o databázový systém, kde je možné dosáhnout vysoké dostupnosti a databázi horizontálně škálovat, ovšem nebyla pro toto škálování od začátku navržena a NoSQL databázové systémy jsou pro tyto účely lepší.

Redis

Redis je open source (pod licencí BSD), in-memory úložiště datových struktur, které se používá jako databáze, cache, message broker a streaming engine[36]. Jedná se o tzv. key-value store, kde jsou data uložena ve formě klíč-hodnota, a klíč je zároveň unikátním identifikátorem záznamu. Jako hodnoty Redis podporuje například řetězce, hashes (strukturované objekty), seznamy, množiny, řazené množiny a další. Nad těmito hodnotami je možné provádět atomické operace.

Redis dosahuje vysoké výkonnosti tím, že se jedná o in-memory úložiště, má však i podporu pro periodické ukládání na disk, či ukládání provedených operací do logu. Dále podporuje asynchronní replikaci (master-replica) pro dostupnost a sharding pro škálování při provozu v clusteru.

Kapitola 5

Návrh nové architektury

Cílem práce je návrh nové architektury, která řeší nedostatky aktuální. Architektura má maximalizovat využití DevOps a automatizace, soustředit se hlavně na škálovatelnost, monitoring, dostupnost a odolnost proti chybám. V této kapitole budou požadavky rozpadeny do větší míry detailu formálnějším a ověřitelným způsobem tak, aby bylo možné architekturu vyhodnotit a ověřit, do jaké míry a jakým způsobem požadavky řeší.

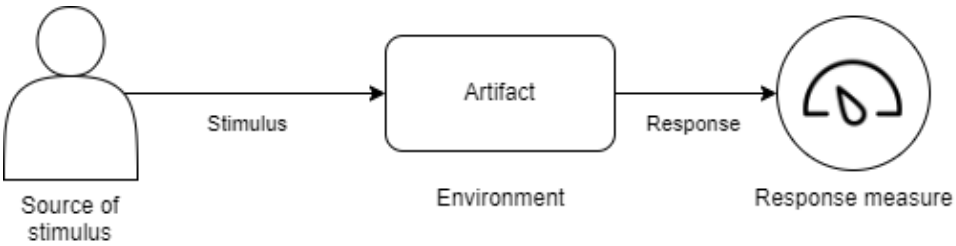
5.1 Způsob definice požadavků na architekturu

Požadavky na architekturu budou prezentovány a formalizovány pomocí tzv. Quality atributes (atributů kvality) a Quality attributes scenarios (scénáře atributů kvality).

Atribut kvality je měřitelná nebo testovatelná vlastnost systému, která indikuje, jak dobře systém splňuje požadavky stakeholderů. Scénář atributu kvality je formou zachycení požadavků na atribut kvality[3].

Tato formalizace umožní výslednou architekturu vyhodnotit a říci, zda požadavky splňuje.

5.1.1 Scénář atributů kvality



Obrázek 5.1: Obecná šablona pro scénář atributů kvality

Níže je uvedena tabulka (5.1) s popisem jednotlivých částí scénáře.

Část	Popis
Stimulus	Stimulus označuje událost, která se objeví v našem systému. Pro různé atributy může být stimulus různý. Například pro výkonnost se jedná o událost, pro bezpečnost je to útok a pro udržovatelnost se jedná o požadavek na změnu.
Source of stimulus	Stimulus je vyvolán nějakou entitou (systém, útočník, stakeholder). Zdroj stimulu může ovlivnit, jak by na něj měl systém reagovat.
Artifact	Stimulus jako událost se dostane k nějaké části systému. Zde můžeme zvolit různou míru detailu, ovšem čím přesněji, tím lépe. Na chybu databáze budeme reagovat nejspíše jinak než na chybu monitoringu.
Environment	Environment (prostředí) nám říká, za jaké situace se stimulus objevil. Většinou se jedná o stav systému (běžící, v chybě, vypínající se apod.), ovšem může se jednat i o stav systému z pohledu projektu (jinak se budeme chovat k požadavku na změnu při vývoji a jinak po nasazení).
Response	Response popisuje, jak se artefakt zachová po přijetí stimulu. Úkolem architekta je zajištění této části. Response může být opět různá v závislosti na daném atributu kvality. Například pro výkonnost to může být vygenerování odpovědi po přijetí požadavku, pro údržbu to může být implementace požadavku na změnu.
Response measure	Response by měla být nějakým způsobem měřitelná tak, abychom mohli ověřit, zda bylo dosaženo požadované odpovědi. Pro výkonnost se může jednat o latenci, pro vývoj to může být například doba pro implementaci změny.

Tabulka 5.1: Popis částí scénáře atributů kvality[3]

5.1.2 Atributy kvality

Availability (dostupnost)

ISO/IEC 25010[2] definuje dostupnost následovně:

> Stupeň, do kterého je systém, výrobek nebo součást funkční a přístupný v okamžiku, kdy je vyžadován pro použití.

Dostupnost je spojena do jisté míry s robustností (chyby nemají způsobit nedostupnost) a s bezpečností (například Denial of Service útoky ovlivňují

dostupnost). Dostupnost systému můžeme měřit jako pravděpodobnost, že systém bude poskytovat službu v požadovaných mezích v nějakém časovém intervalu[3]. Pro toto se často používá následující vzorec:

$$MTBF/(MTBF + MTTR)$$

kde MTBF je průměrný čas mezi selháním (Mean Time Between Failures) a MTTR je průměrný čas opravy (Mean Time To Repair). MTBF si můžeme představit jako "uptime"a $MTBF + MTTR$ jako celkový čas. Dostupnost se často vyjadřuje v počtu devítek za desetinou čárkou ve výsledku předchozího vzorce (5.2). Termínem *vysoká dostupnost* se obvykle označuje dostupnost s alespoň pěti devítkami, tj. 99.999%.

Dostupnost	**Nedostupnost v 90 dnech**	**Nedostupnost v roce**
99.0%	21 hod 36 min	3 dny 15.6 hod
99.9%	2 hod 10 min	8 hod 0 min 46 s
99.99%	12 min 58 s	52 min 34 s
99.999%	1 min 18 s	5 min 15 s
99.9999%	8 s	32 s

Tabulka 5.2: Dostupnost dle počtu devítek za desetinnou čárkou

V rámci práce míříme na vysokou dostupnost, tj. alespoň 99.999%.

Security (bezpečnost)

ISO/IEC 25010[2] definuje bezpečnost následovně:

> Stupeň, v jakém produkt nebo systém chrání informace a data tak, aby osoby nebo jiné produkty či systémy měly k datům přístup v míře odpovídající jejich typům a úrovním oprávnění.

V nejjednodušší formě se můžeme zaměřit na tři hlavní charakteristiky:

- Důvěrnost - data nebo služby jsou chráněny před neautorizovaným přístupem.
- Integrita - data nebo služby nejsou měněna bez autorizace.
- Dostupnost - systém bude dostupný pro oprávněné použití.

Deployability

Deployability popisuje náročnost, s jakou probíhá nasazení systému do produkčního prostředí. Tento atribut je v dnešní době důležitější než dříve, vzhledem k častému vydávání nových verzí a vzhledem k podpoře ze strany

nástrojů. Dobře zvládnuté nasazování a vydávání verzí nám umožňuje rychle řešit chyby a dodávat nové funkcionality, což nám může poskytnout výhodu nad konkurencí. Pokud zajdeme do extrému, můžeme novou verzi systému vydávat a nasazovat prakticky kdykoliv a několikrát denně.

Zde je samozřejmě důležitý kontext. Pro bankovní software budou platit jistá omezení, která pro e-commerce platit nebudou a mohou mít vliv na tento atribut.

Scalability (škálovatelnost)

Pro tuto práci použijeme následující definici škálovatelnosti[4]:

> Schopnost systému zvládat vyšší zátěž a udržet výkonnost v akceptovatelných mezích.

V rámci softwarových systémů škálovatelnost nejčastěji dělíme na dvě:

- vertikální,
- horizontální.

Vertikální škálovatelnost je přidávání nebo ubírání systémových zdrojů (CPU, RAM, atd.) na jednom uzlu. Vertikální škálovatelnosti se také říká "scaling up/down". Horizotnílní škálovatelnost (také "scaling out/in") je přidávání či ubírání dalších uzlů.

Existuje více možností, jak škálovatelnost dělit, například *Scale Cube*[1], která krom klonování uzlů (horizontální škálování výše) zmiňuje také funkční dekompozici a data partitioning.

Maintainability (udržovatelnost)

ISO/IEC 25010[2] definuje udržovatelnost následovně:

> Stupeň účinnosti a efektivity, s jakou lze výrobek nebo systém upravit, aby se zlepšil, opravil nebo přizpůsobil změnám prostředí a požadavků.

Jedním z atributů zařazených pod udržovatelnost v ISO/IEC 25010 je modifikovatelnost. Na tu se v této části zaměříme více.

Standard definuje modifikovatelnost následovně:

> Stupeň, do kterého lze výrobek nebo systém účinně a efektivně upravit, aniž by se objevily vady nebo se zhoršila kvalita stávajícího výrobku.

Pro tento atribut je důležité zvážit jaké části systému se budou měnit, jak často se bude měnit a kolik bude změna stát.

Cena změny je důležitým faktorem pro stakeholdery. Jako architekti můžeme zavést do systému mechanismy, které podporují modifikatelnost (např. generování kódu klienta webových služeb z WSDL). U mechanismu můžeme vzít v potaz dva druhy ceny:

[1] https://microservices.io/articles/scalecube.html

- cena zavedení mechanismu (dále IC),
- cena modifikace za použití mechanismu (dále UC).

Dále si zaveďme MC jako cenu za provedení modifikace bez mechanismu a N jako počet (odhad) podobných změn prováděných pomocí mechanismu. Poté se můžeme na základě nerovnice níže[3] rozhodnout, zda mechanismus zavádět či nezavádět.

$$N * MC \leq IC + (N * UC)$$

Robustness (robustnost) a fault-tolerance

Fault-tolerance a robustnost jsou velice podobné atributy, které spolu blízce souvisí, avšak existuje mezi nimi rozdíl. Fault-tolerance samotná znamená vyhnout se selhání služeb v případě chyb. Robustnost je speciální případ fault-tolerance, který se vztahuje k externím chybám (tj. chybám způsobených vnějším působením)[1]. Chybou pro případ robustnosti může být například nevalidní vstup od uživatele.

Vzhledem k tomu, že robustnost je specializací fault-tolerance, můžeme říci, že fault-tolerance implikuje robustnost (ale ne naopak). Fault-tolerance souvisí také s dostupností. Pokud náš systém v případě chyb selže, není dostupný a snižuje dostupnost systému.

5.1.3 Recoverability

Spolu s fault-tolerance a robustností souvisí také obnovitelnost. ISO/IEC 25010[2] definuje obnovitelnost následovně:

> Míra, do jaké může produkt nebo systém v případě přerušení nebo selhání obnovit přímo dotčená data a obnovit požadovaný stav systému.

Pro účely této práce nebude uvažována jen obnova dat, ale obnova či náhrada selhaných komponent.

5.1.4 Observability

Observability definujeme[6] v softwarových systémech následovně:

> Schopnost shromažďovat údaje o provádění programu, vnitřních stavech modulů a komunikaci mezi komponentami.

Pro dosažení těchto cílů je potřeba zavedení logování, trasování požadavků a monitoring systému a jednotlivých komponent.

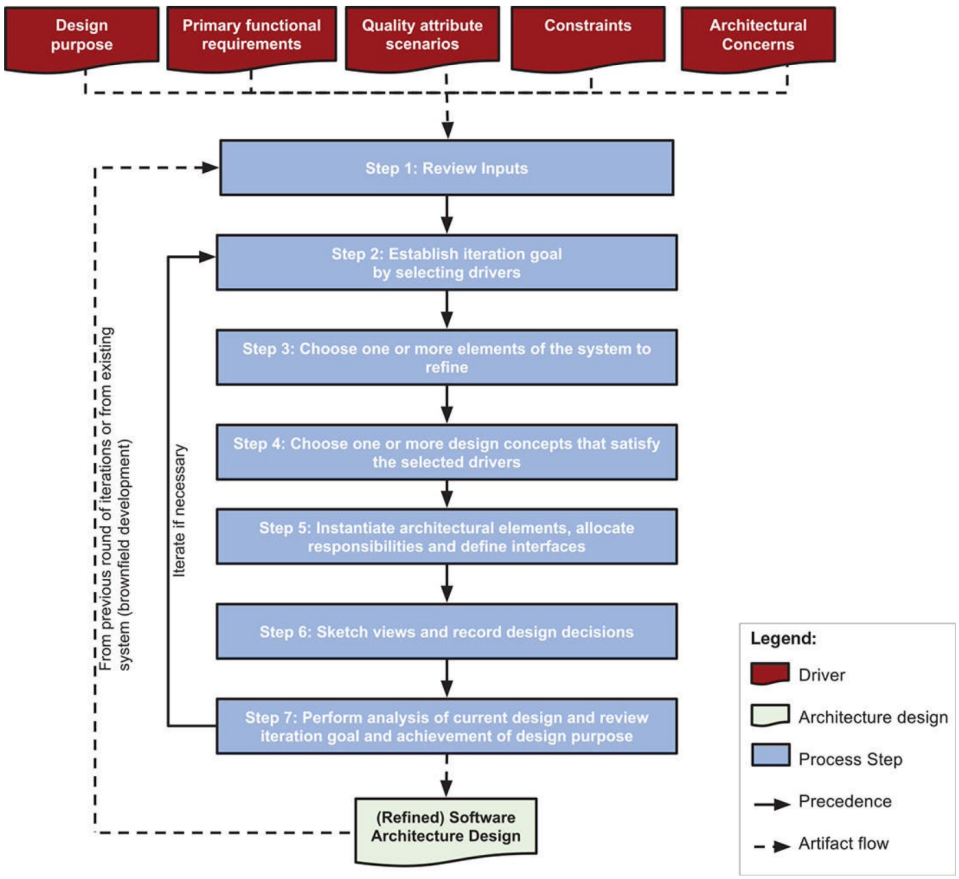
5.2 Metodika návrhu

Pro návrh architektury bude v této práci použita metodika *Attribute-driven design 3.0*. Metodika byla zvolena z důvodu její vazby na scénáře atributů kvality, se kterými pracuje, a proto je hodící se metodikou pro tuto práci. Dalším přínosem této metodiky je její výstup. Pokud jsou všechny iterace dokumentovány, na konci máme k dispozici dokumentaci celého procesu návrhu, a to včetně rozhodnutí, která byla v jednotlivých iteracích učiněna a z jakých důvodů.

Návrh architektury v této metodice probíhá v iteracích. Každá iterace je rozdělena na podčásti, které jsou detailněji popsány v následující podkapitole.

5.2.1 Popis jedné iterace ADD

Schéma jedné iterace ADD je popsáno na obrázku 5.2.



Obrázek 5.2: Iterace metodiky ADD[5]

Krok 1: Revize a validate vstupů

V prvník kroku revidujeme a validujeme vstupy. Zde je důležité ověřit, že máme vše, co je potřebné pro návrh a že je vše korektní. V tomto kroku nás zajímá následující:

- máme scénáře atributy kvality a jsou seřazeny dle priority,
- známe omezení, která jsou na architekturu kladena,
- pokud se nejedná o první iteraci, máme k dispozici všechny výstupy z předchozích iterací,
- pokud se jedná o první iterace a zároveň o refactoring již existující architektury, pak máme k dispozici analýzu a stav stávající architektury.

Tento krok je jedním z nejdůležitějších. Rozhodnutí, která uděláme v dalších krocích, jsou závislá na vstupech a pokud vstupy nejsou správné, nemohou být správná ani další rozhodnutí na nich postavená.

Krok 2: Vyběr cíle iterace

V každé iteraci se soustředíme na dosažení nějakého cíle. Typicky se snažíme uspokojit vybrané cíle z požadavků na vstupu (například use-case nebo scénář atributu kvality).

Krok 3: Výběr jedné nebo více částí systému k úpravě

V tomto kroku produkujeme architektonické struktury. Zde typicky provádíme některé z následujících činností:

- rozpad elementů na více elementů s větší mírou detailu a užším zaměřením,
- kombinace menších elementů do větších celků s menší mírou detailu a větším zaměřením,
- vylepšení elementů z předchozí iterace.

Pokud vyvíjíme na zelené louce, můžeme v první iteraci jako první element k rozpadu použít systém jako takový a vytvořit první elementy. V dalších iteracích, nebo v případě refactoringu již existujícího systému, začínáme s již existujícími elementy. Elementy, které zde vybereme k úpravě, jsou ty, které uspokojují některé z cílů iterace.

Ač jsou kroky *Výběr cíle iterace* a *Výběr jedné nebo více částí systému k úpravě* uvedeny v tomto pořadí, někdy je možné pořadí těchto dvou kroků prohodit.

Krok 4: Výběr designových konceptů, které uspokojují vybrané požadavky

Zde vybíráme designové koncepty, které nám pomohou uspokojit požadavky, které jsme si dali jako cíl současné iterace. Zde uvažujeme různé alternativy řešení a vybíráme, která z nich použijeme. Ač je popis kroku krátký, jedná se o ten nejsložitější.

Příkladem může být výběr referenční architektury, se kterou začínáme (event driven, vrstevnatá, atd.) nebo výběr konkrétních technologií (např. relační databáze vs. dokumentová NoSQL databáze).

Krok 5: Vytvoření elementů, rozdělení zodpovědnosti a definice rozhraní

V tomto kroku vytváříme elementy, které jsou součástí designových konceptů z předchozího kroku. Následně můžeme říci jakou mají elementy zodpovědnost a jak spolu, a se zbytkem systému, komunikují.

Krok 6: Architektonické pohledy a zaznamenání rozhodnutí

V tomtu kroku provádíme revizi a úpravu našich architektonických pohledů na základě rozhodnutí z předchozích kroků. Je důležité poznamenat, že v této fázi nemusí jít o plně formální verzi pohledů, může se jednat jen o skicy, nástiny (toto platí obzvláště pro počáteční iterace). Dále v tomto kroku zaznamenáváme důležitá rozhodnutí.

Krok 7: Analýza současného stavu designu a dosažení cílů iterace

V tomto kroku analyzujeme náš nově upravený design a validujeme, zda a do jaké míry naplnil cíle iterace. V ideálním případě máme pro tento krok k dispozici další osobu, která nebyla součástí předchozích kroků.

5.3 Požadavky

5.3.1 Scénáře atributů kvality

Popis struktury

V rámci této práce bude mít každý scénář následující atributy:

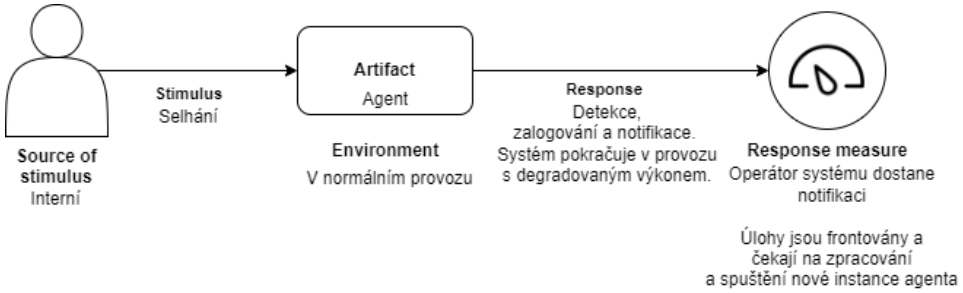
- *ID* - jednoznačná identifikace, která umožní odkazování na atribut.
- *Atribut kvality* - o jaký atribut kvality se jedná.
- *Byznysová priorita* - priorita s ohledem na vliv na úspěšnost systému. Určena stakeholdery.
- *Technický risk* - Výše technického risku spojená se scénářem. Určuje architekt.

- *Scénář* - popis scénáře notací definovanou v kapitole 5.1.1 Scénář atributů kvality.

Pro *Byznysová priorita* a *Technický risk* bude použita stupnice LOW, MEDIUM a HIGH. Na základě ohodnocení daných dimenzí budou následně scénáře prioritizovány.

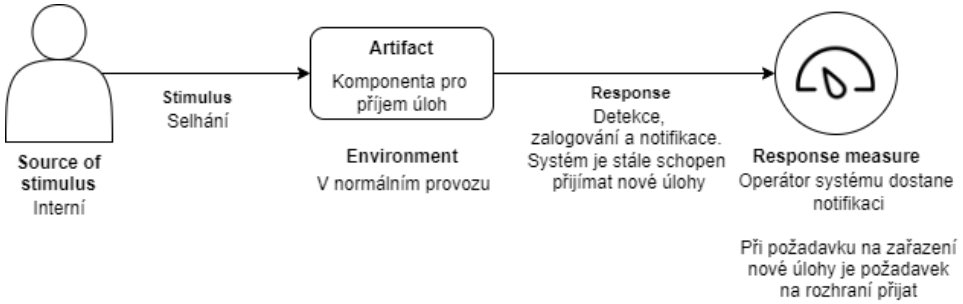
QA-1

ID	QA-1
Atribut kvality	Availability
Byznysová priorita	MEDIUM
Technický risk	MEDIUM
Scénář	



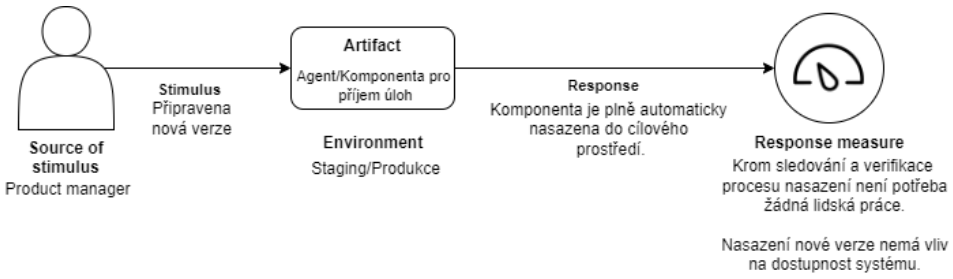
QA-2

ID	QA-2
Atribut kvality	Availability
Byznysová priorita	MEDIUM
Technický risk	MEDIUM
Scénář	



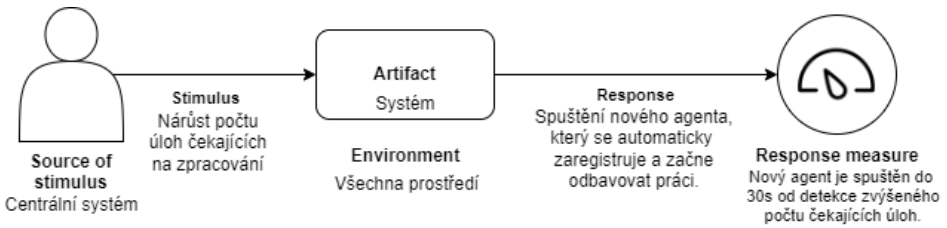
QA-3

ID	QA-3
Atribut kvality	Deployability
Byznysová priorita	HIGH
Technický risk	MEDIUM
Scénář	



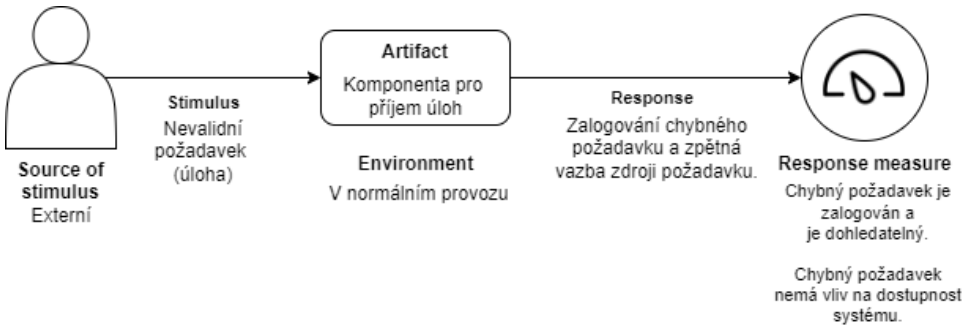
QA-4

ID	QA-4
Atribut kvality	Scalability
Byznysová priorita	MEDIUM
Technický risk	HIGH
Scénář	



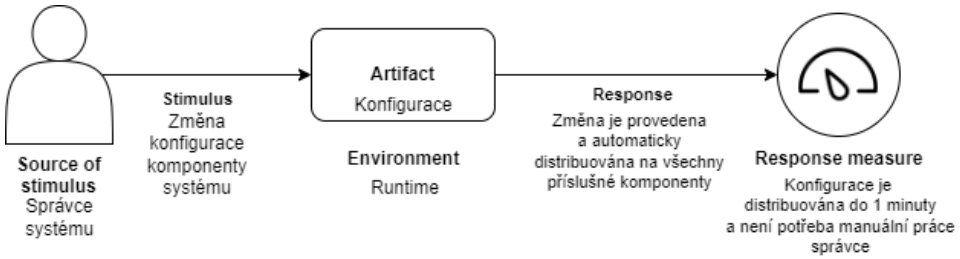
QA-5

ID	QA-5
Atribut kvality	Robustness
Byznysová priorita	HIGH
Technický risk	LOW
Scénář	



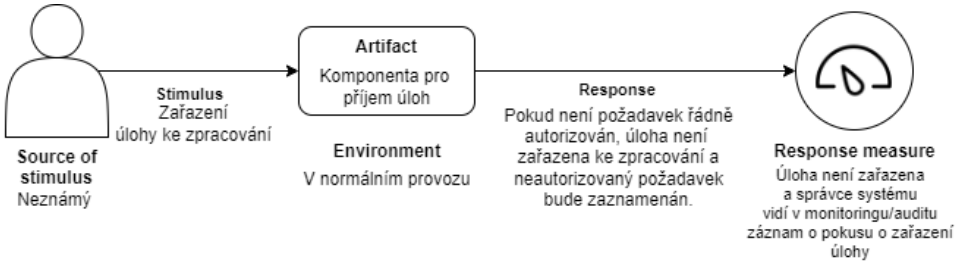
QA-6

ID	QA-6
Atribut kvality	Modifiability
Byznysová priorita	LOW
Technický risk	MEDIUM
Scénář	



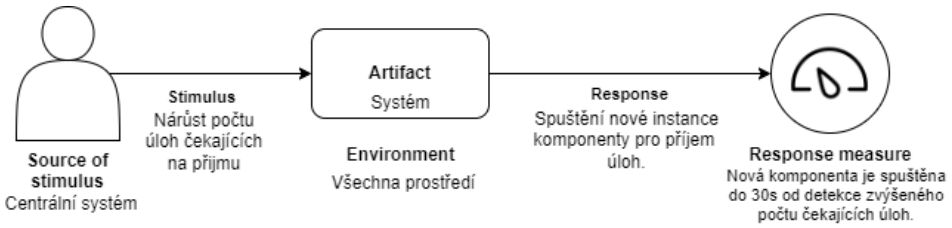
QA-7

ID	QA-7
Atribut kvality	Security
Byznysová priorita	HIGH
Technický risk	LOW
Scénář	



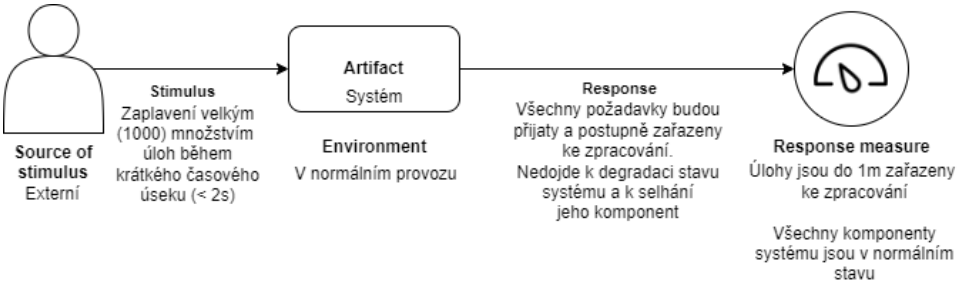
QA-8

ID	QA-8
Atribut kvality	Scalability
Byznysová priorita	LOW
Technický risk	MEDIUM
Scénář	



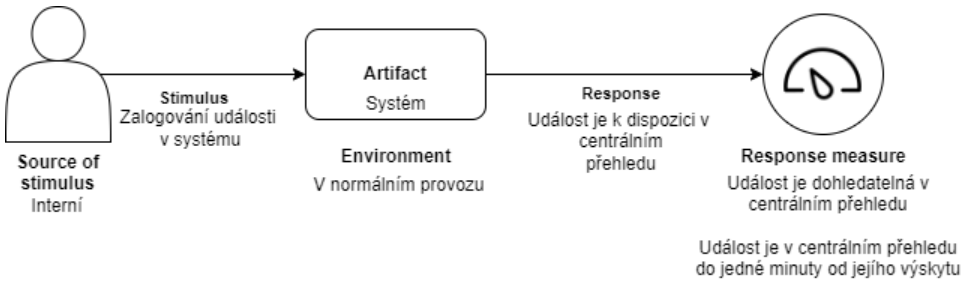
QA-9

ID	QA-9
Atribut kvality	Scalability
Byznysová priorita	MEDIUM
Technický risk	HIGH
Scénář	



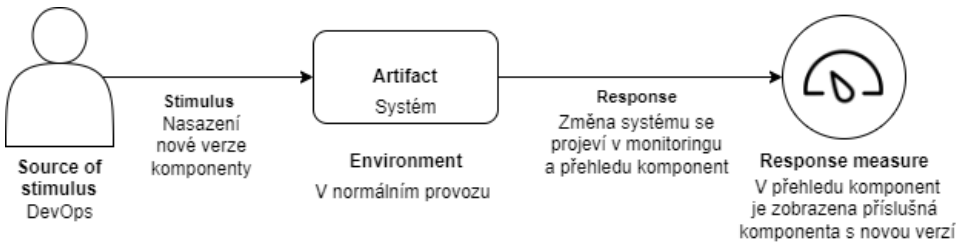
QA-10

ID	QA-10
Atribut kvality	Observability
Byznysová priorita	HIGH
Technický risk	MEDIUM
Scénář	



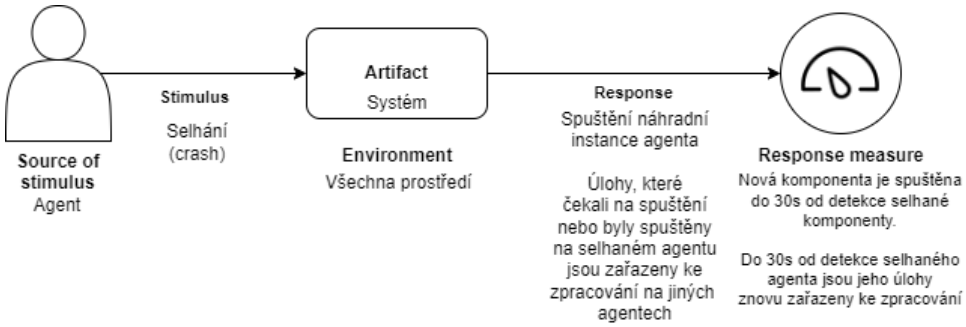
QA-11

ID	QA-11
Atribut kvality	Observability
Byznysová priorita	HIGH
Technický risk	HIGH
Scénář	



QA-12

ID	QA-12
Atribut kvality	Recoverability
Byznysová priorita	HIGH
Technický risk	HIGH
Scénář	



5.3.2 Prioritizace scénářů

Prioritizace je provedena pomocí tzv. matice priorit. Na jedné ose máme *Byznysovou prioritu* (dále jen BP), na druhé *Technický risk* (dále jen TR). Hodnotami jsou ID (jen číselná část, pro přehlednost) jednotlivých scénářů. Následně se můžeme soustředit primárně na scénáře (H, H), následně (M,H)/(H,M) a tak dále, v závislosti na čase, který na návrh máme.

$TR \backslash BP$	L	M	H
L			$5, 7$
M	$6, 8$	$1, 2$	$3, 10$
H		$4, 9$	$11, 12$

Tabulka 5.3: Matice priorit

5.4 Proces návrhu

V této části je popsán proces návrhu nové architektury za pomocí metodiky ADD popsané v části 5.2.

5.4.1 Krok 1: Revize a validate vstupů

Z části Požadavky známe případy užití, scénáře atributů kvality a jejich prioritizaci. Dále také z kapitoly Zadání známe další omezující podmínky. Z kapitoly Analýza současného řešení známe kontext, zkušenosti aktuálního týmu a prostředí, v jakém je/bude systém vyvíjen a provozován. Máme tedy všechny podklady, které jsou potřebné pro další kroky návrhu.

5.4.2 Iterace 1: Celková struktura systému

Krok 2: Výběr cíle iterace

Jedná se o první iteraci návrhu systému "na zelené louce". Zde mějme na paměti hlavně následující atributy kvality (a jejich scénáře):

- dostupnost,
- škálovatelnost.

Krok 3: Výběr jedné nebo více částí systému k úpravě

Jelikož se jedná o první iteraci návrhu "na zelené louce", je částí k úpravě systém jako celek. V této iteraci budeme systém upravovat formou dekompozice na menší elementy.

Krok 4: Výběr designových konceptů, které uspokojují vybrané požadavky

Rozhodnutí	Odůvodnění
Využití architektonického stylu Microservices	Mezi základní koncepty microservisní architektury patří tvorba samostatně nasaditelných komponent. Tento princip umožňuje zároveň spouštění více instancí komponent nezávisle na sobě, což podporuje dostupnost a škálovatelnost.

Tabulka 5.4: Návrhová rozhodnutí v první iteraci

Alternativa	Důvod nevybrání
Vrstevnatá architektura	Nízká granularita škálovatelnosti, velká šance, že vývoj sklouzne k monolitickému designu.
SOA	Cílem je návrh architektury jedné aplikace, ne celé podnikové architektury.
Event driven	Nejedná se o aplikaci, kde by se měnil stav entit a aplikace by na tyto podněty měla reagovat. Smyslem aplikace je vyřizování požadavků "klientského systému" a stavy, které uchovává, jsou čistě režijní.

Tabulka 5.5: Alternativní rozhodnutí v první iteraci

Krok 5: Vytvoření elementů, rozdělení zodpovědnosti a definice rozhraní

Rozhodnutí	Odůvodnění
Vytvoření služby Příjem požadavků a odeslání výsledků (Message box)	Tato služba bude přijímat požadavky od "klientského systému". Její úlohou je interní buffering, uchovávání procesních informací a řízení zpracování (například deduplikace či zajištění běhu jen jedné instance úlohy). Služba může být škálovaná samostatně po ose Z na ScaleCube.

Vytvoření služby Agent	Tato služba má na starosti vykonávání samotné úlohy. Její rozhraní má na vstupu instanci úlohy a na výstupu její výsledek. Agent bude služba s nejvyšším využitím škálování. Její instance se budou spouštět dle potřeby a zátěže systému (počet úloh ve službě pro příjem a rychlost jejich odbavování) s možným minimálním množstvím instancí.
Vytvoření služby Agent Controller	Tato služba má na starosti škálování agentů. Služba bude sledovat metriky zátěže a rychlosti zpracování úloh z jednotlivých služeb pro příjem a v případě potřeby bude spouštět nové instance služby Agent pro daný typ úlohy.
Vytvoření služby Configuration	Služba pro načítání konfigurace. Jednotlivé typy úloh mají v rámci systému konfiguraci, kterou se řídí při zpracování. Tato služba poskytuje tuto konfiguraci, kterou je následně možné připojit ke zprávě na vstupu pro agenta.
Použití messaging komponenty pro komunikaci	Synchronní komunikace mezi službami je blokující, hůře škálovatelná a zvyšuje závislost mezi službami. Asynchronní komunikace pomocí messaging komponent je neblokující, snazší na škálování a provázání služeb je nižší. Implementace asynchronní komunikace je o něco náročnější, ale v našem případě jsou její přínosy vyšší, než zvýšená náročnost. Messaging bude použit pro komunikaci mezi "klientským systémem" a službou pro příjem (zde je důležité neblokování) a dále mezi službou pro příjem a agentem.

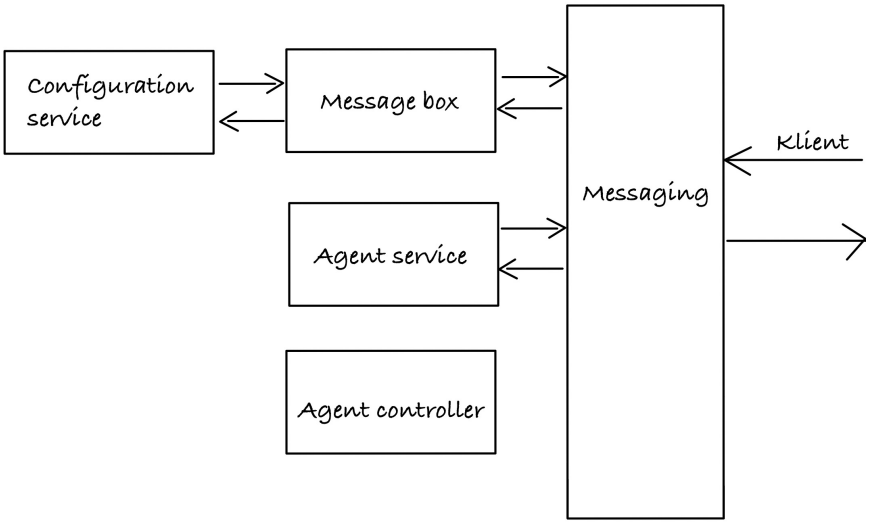
Tabulka 5.6: Krok 5 v první iteraci

Krok 6: Architektonické pohledy a zaznamenání rozhodnutí

Výstupem první iterace je tento high-level náčrt diagramu komponent a jejich komunikace.

Krok 7: Analýza současného stavu designu a dosažení cílů iterace

Výsledky aktuální iterace jsou uvedeny v tabulce 5.7. Tabulka reprezentuje tzv. kanban board, kde jsou jednotlivé požadavky, míra jejich naplnění a, pokud došlo k jejich přesunu, důvod přesunu.



Obrázek 5.3: Výsledek první iterace, použití microservices architektury

Nenaplněné	Částečně naplněné	Úplně naplněné	Designové rozhodnutí
QA-1			
QA-2			
	QA-3		Rozhodnutí zvolit microservices architekturu podporuje automatizaci nasazení aplikace.
	QA-4		Rozhodnutí zvolit microservices architekturu umožňuje agenta škálovat nezávisle. Vytvoření služby Agent Controller pomůže automatizaci této činnosti.
QA-5			
		QA-6	Služba Configuration bude mít na starosti sledování změn v konfiguraci a jejich předání na potřebné komponenty.
QA-7			
	QA-8		Rozhodnutí zvolit microservices architekturu umožňuje komponentu škálovat nezávisle.

	QA-9	Rozhodnutí využít messaging pro komunikaci zabraňuje přetížení a selhání služeb.
QA-10		
QA-11		
	QA-12	Vytvoření služby Agent Controller zajišťuje detekci neodbavování zpráv a spouští novou instanci. Služba Message Box, která ví, které úlohy nebyly spuštěny, případně na jakém agentu byly spuštěny, má možnost znovu spustit jejich zpracování.

Tabulka 5.7: Naplnění cílů iterace 1

5.4.3 Iterace 2: Observability

Krok 2: Výběr cíle iterace

V druhé iteraci je cílem doplnit strukturu systému z první iterace o možnosti monitorování a sběru logů tak, aby bylo možné centrálně monitorovat systém. V této iteraci se soutředíme primárně na atribut *Observability*, ovšem monitoring a logování souvisí částečně i s robustností a dostupností (je nutné mít monitoring, aby bylo možné detekovat stav systému a podniknout další kroky).

Krok 3: Výběr jedné nebo více částí systému k úpravě

V této iteraci je částí systému k úpravě systém jako celek. Dále můžeme pracovat s existujícími komponentami a jít do vyšší míry detailu a doplnit je o elementy týkající se monitoringu a sběru logů.

Krok 4: Výběr designových konceptů, které uspokojují vybrané požadavky

Rozhodnutí	**Odůvodnění**
Použití vzoru Aplikační metriky	Tento vzor řeší náš požadavek na centralizovaný přehled systému. Výhodou je separace funkcionality sběru, správy a vizualizace metrik do samostatných komponent (je možné použít existující technologie) a není nutné tuto logiku implementovat v každé z našich komponent.
Použití vzoru Agregace logů	Tento vzor nám umožní vidět logy ze všech komponent na jednom místě. Stejně tak můžeme následně logy procházet, vidět celý kontext a je možné na základě jejich obsahu i notifikovat správce systému.
Použití vzoru Distributed tracing	Při využití vzoru Agregace logů nám může pomoci využití vzoru Distributed tracing. To nám umožní sledování konkrétních úloh v čase a sledovat jejich zpracování.
Použití vzoru Health check API	Využití vzoru Health check API nám umožní sledovat stav aplikací a případné upozornění správců v případě, že je s instancemi něco v nepořádku (například nedostatek zdrojů). I tyto údaje je možné předávat jako metriky.
Použití vzoru Service registry	Potřebujeme mít možnost zjistit jaké služby běží, kde běží a jak s nimi komunikovat.

Tabulka 5.8: Návrhová rozhodnutí v druhé iteraci

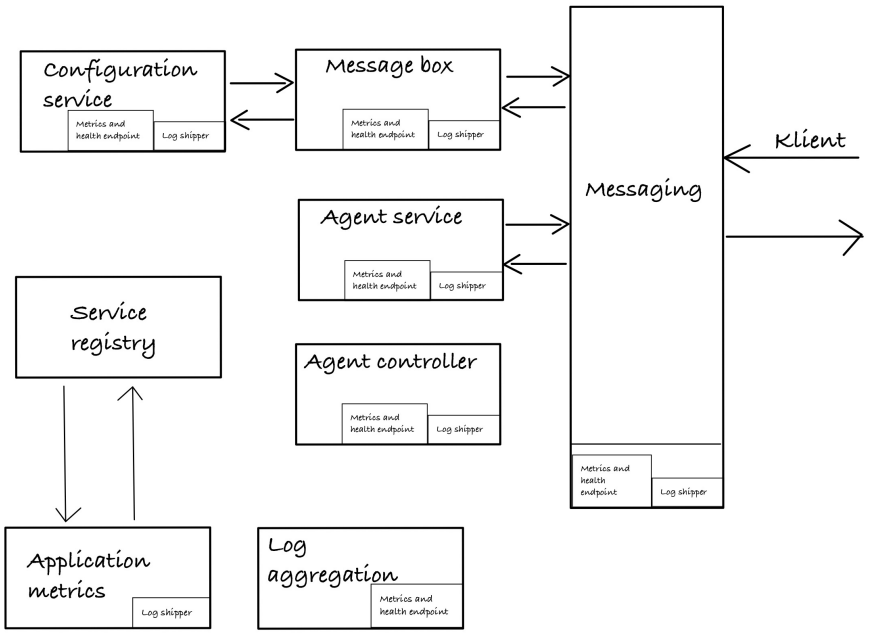
Krok 5: Vytvoření elementů, rozdělení zodpovědnosti a definice rozhraní

Rozhodnutí	Odůvodnění
Použití PULL komunikace pro získání metrik	Ač je možné v rámci messagingu implementovat i PUSH metodu posílání metrik, PULL má výhody pro vývoj, kdy je možné se na metriky podívat i manuálně. Navíc je možné periody sbírání metrik měnit na straně koletoru metrik a není potřeba upravovat jejich poskytovatele.
Přidání tzv. log shipper do každé komponenty	V rámci komponent je nutné jejich logy do agregátory nějakým způsobem odesílat.
Přidání endpointů pro sběr metrik a health check	V rámci použití vzorů aplikačních metrik a health check API je nutné přidat endpointy poskytující potřebné informace.
Přidání komponenty pro sběr a zpracování logů	Logy přijaté z komponent budou uloženy a zpracovány na jednom místě.
Přidání komponenty pro sběr a zpracování aplikačních metrik	Komponenta bude mít na starosti sběr metrik, jejich následné uložení, zpracování a vizualizaci.
Přidání ID úlohy do každé zprávy logu z každé komponenty	ID úlohy je její univerzální identifikátor, který má k dispozici každá komponenta úlohu zpracovávající. Je ji tak možné použít jako ID pro Distributed tracing
Přidání komponenty pro Service registry	Service registry bude samostatná komponenta, kterou budou ostatní používat.

Tabulka 5.9: Krok 5 v druhé iteraci

Krok 6: Architektonické pohledy a zaznamenání rozhodnutí

V tomto nákresu jsou pro přehlednost vynechány komunikace mezi službou pro aplikační metriky a endpointy na jednotlivých komponentách a stejně tak mezi komponentou pro agregaci logů a log shipper elementy.



Obrázek 5.4: Výsledek druhé iterace, zavedení komponent a vzorů pro sledování stavu a chodu systému

Krok 7: Analýza současného stavu designu a dosažení cílů iterace

Nenaplněné	Částečně naplněné	Úplně naplněné	Designové rozhodnutí
	QA-1		Díky agregaci logů, jejich analýze a sběru metrik/health check API je možné notifikovat operátora systému.
	QA-2		Díky agregaci logů, jejich analýze a sběru metrik/health check API je možné notifikovat operátora systému.
	QA-4		Díky sběru metrik je možné detekovat nárůst, či stagnaci počtu čekajících úloh.
	QA-5		Díky agregaci logů je nevalidní požadavek dohledatelný v centrálním přehledu logů.
	QA-7		Díky agregaci logů je nevalidní požadavek dohledatelný v centrálním přehledu logů.

QA-8	Díky sběru metrik je možné situaci detekovat.
QA-10	Využití agregace logů kompletně naplňuje tento scénář, ovšem je potřeba zvolit technologii, která bude použita. Z tohoto důvodu zůstává scénář v této kategorii.
QA-11	Využití aplikačních metrik kompletně naplňuje tento scénář, ovšem je potřeba zvolit technologii, která bude použita. Z tohoto důvodu zůstává scénář v této kategorii.
QA-12	Díky sběru metrik a agregaci logů je možné situaci detekovat.

Tabulka 5.10: Naplnění cílů iterace 2

5.4.4 Iterace 3: Funkcionalita

Krok 2: Výběr cíle iterace

Cílem třetí iterace je návrh samotných komponent tak, aby splňovaly svou stanovenou roli a aby jejich návrh podporoval naplnění scénářů atributů kvality.

Krok 3: Výběr jedné nebo více částí systému k úpravě

Elementy, na které se budeme soustředit v této iteraci, jsou Messaging a samotné služby Message Box, Agent a Agent controller. Ty je nutné navrhnout tak, aby podporovaly nezávislé fungování a tím i škálovatelnost a dostupnost.

Krok 4: Výběr designových konceptů, které uspokojují vybrané požadavky

Rozhodnutí	**Odůvodnění**
Použití vzoru Database per service	Tento vzor umožňuje nižší provázanost mezi službami, stejně tak mohou používat typ databáze, který je pro ně nejvhodnější. Použití vzoru také podporuje dostupnost, kdy by pád sdílené databáze omezil na funkčnosti více komponent.
Použití vzoru Sidecar pro Agenta	Vzor Sidecar nám umožní extrahovat funkcionalitu, která se netýká agenta samotného do samostatné komponenty. Tím bude kód agenta nezávislý na zbytku systému a může se jednat jen o skript zpracovávající vstup a produkující výstup. Využití tohoto vzoru také umožní vývoj agentů bez nutnosti znát implementaci zbytku systému.
Odbavování zpráv úloh z messaging komponenty bez prodlení	Úlohy z messaging komponenty chceme odebírat co nejrychleji a poskytnout klientskému systému zpětnou vazbu o jejich přijetí.
Umožnit komponentě Message Box ukládání režijních informací	Jelikož tato komponenta pracuje jako orchestrátor zpracování úlohy a musí zajistit splnění různých podmínek (například na souběh úloh), je nutné umožnit komponentě uchovávat perzistentně režijní informace.
Použití vzoru Externalized configuration	Služby budou mít svou konfiguraci externě uloženou a načítat ji při startu.
Využití mechanismu routování pro messaging	Vstupní zprávy budou směrovány na příslušné instance komponenty Message Box pomocí routování.

Použití taktiky Heartbeat	Služby Agent budou posílat periodické zprávy komponentě Message Box. Ta tak bude vědět o stavu komponent a bude moci reagovat na její selhání navazující logikou.

Tabulka 5.11: Návrhová rozhodnutí ve třetí iteraci

Alternativa	Důvod nevybrání
Sdílená databáze	Některé služby by mohly používat druh databáze, který se jim nehodí. Dále dochází ke svázání služeb jak ve vývoji, tak i v rámci běhu. Výhodou tohoto vzoru je, že je jednodušší řešit například zámky, ovšem v tomto systému služby data nesdílejí a databáze slouží jen pro uchování režijních dat.
Ponechání zpráv úloh na messaging komponentě	Toto řešení, ač jednodušší na implementaci, má řadu nevýhod. Odloženým odbavením zpráv nemůžeme klientský systém informovat o přijetí požadavku, což není vhodné, pokud by na takové potvrzení čekal. Dále také některé message broker technologie jsou navržené primárně na odesílání zpráv a ne na jejich delší ukládání. To by mohlo nastat v případě, kdy například běží služba Message Box, ovšem neběžel by zrovna žádný agent zpracovávající daný typ úlohy.

Tabulka 5.12: Alternativní rozhodnutí ve třetí iteraci

Krok 5: Vytvoření elementů, rozdělení zodpovědnosti a definice rozhraní

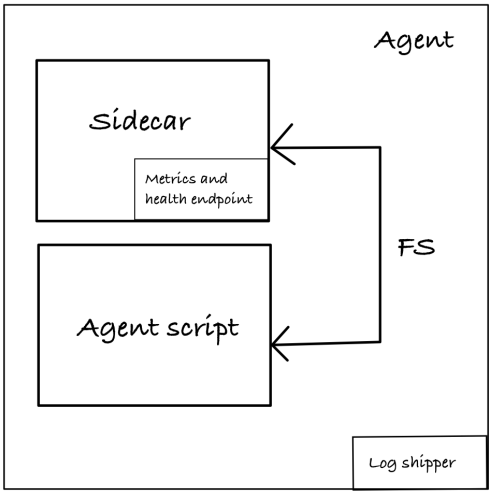
Rozhodnutí	Odůvodnění
Rozdělení komponenty Agent na Sidecar a skript	Sidecar bude v tomto případě zajišťovat komunikaci s okolním světem (tj. zbytek systému) a skript bude výkonným kódem provádějícím byznys logiku agenta. Sidecar a skript spolu budou komunikovat přes lokální souborový systém, kdy Sidecar uloží vstup na souborový systém a spustí skript agenta s cestou k zadání a cestou pro výstup. Agent pak produkuje výstupy do složky, kterou mu poskytne Sidecar na vstupu.

Přidání lokální relační databáze každé instanci služby Message Box	Služba Message Box bude mít k dispozici lokální relační databázi pro ukládání režijních dat.
Přidání interní fronty každé instanci služby Message Box	Služba Message Box bude mít k dispozici perzistentní interní frontu, do které bude moci ukládat zprávy ihned po jejich přijetí z messagingu. Z této fronty budou následně úlohy odbavovány. Fronta bude monitorována a její monitoring bude klíčový pro škálování služby agent.
Použití routování mezi messagingem a Message Boxem na základě typu úlohy a případných parametrů úlohy	Routování bude založené na typu úlohy a na jejich parametrech. To umožní škálování služby Message Box po ose Z na *Scaling cube*. Parametry mohou a nemusí být brány v potaz, záleží na počtu úloh daného typu a je možno to později jednoduše změnit.
Použití proměnných prostředí pro externí konfiguraci	Veškerá konfigurace (přístupové údaje, url, message box, na který se má agent napojit, atd.) budou předávány službám pomocí proměnných prostředí.
Odesílání zprávy heartbeat z Agenta na Message Box jednou za pět sekund. Komponentu označíme za selhanou, pokud nepošle heartbeat do deseti sekund	V rámci požadavků chceme detekovat selhání do 30 sekund. Deset sekund nám poskytne dostatečnou rezervu pro navazující režii.

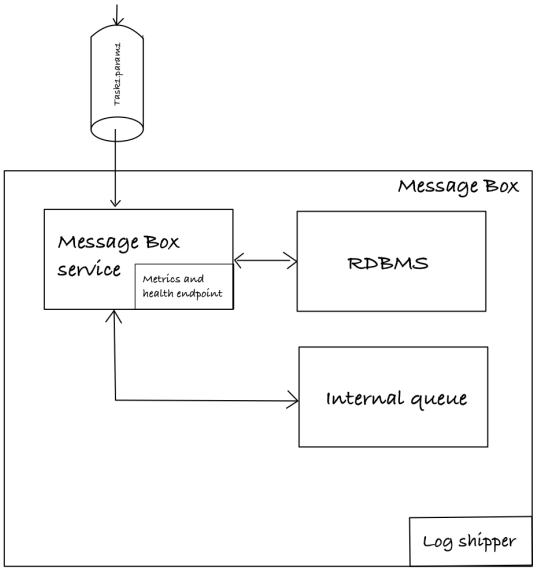
Tabulka 5.13: Krok 5 ve třetí iteraci

Krok 6: Architektonické pohledy a zaznamenání rozhodnutí

V této iteraci budou pro přehlednost pohledy na jednotlivé komponenty uvedeny samostatně. Na obrázku 5.5 je uveden pohled na komponentu Agent, na 5.6 na komponentu Message Box.



Obrázek 5.5: Pohled na komponentu Agent na konci iterace 3



Obrázek 5.6: Pohled na komponentu Message Box na konci iterace 3

Krok 7: Analýza současného stavu designu a dosažení cílů iterace

Nenaplněné	Částečně napl-nění	Úplně napl-nění	Designové rozhodnutí

QA-4		Monitoringem interních front jednotlivých instancí služby Message Box je možné tuto situaci detekovat a následně na ni reagovat spuštěním nových agentů.
	QA-8	Odbavováním zpráv bez prodlení a jejich umístění do interní fronty automaticky řeší tento požadavek.
	QA-9	Použití messaging komponenty pro příjem zajišťuje to, že systém nebude přehlcen a nedojde k selhání komponent. Úlohy budou postupně odbaveny a umístěny na interní fronty Message Boxu a následně předány na zpracování agentovi.
QA-12		Díky interní relační databázi pro uchovávání režijních dat má Message Box přehled o tom jaká práce byla spuštěna na jakých agentech (pro jeho daný typ úlohy a parametry) a může na informaci o pádu agenta následně reagovat.

Tabulka 5.14: Naplnění cílů iterace 3

5.4.5 Iterace 4: Výběr technologií

Krok 2: Výběr cíle iterace

Cílem této iterace je zvolení technologií pro jednotlivé komponenty systému. Zvolené technologie budou podporovat naplnění požadavků na systém a pokud možno budou využity aktuální zkušenosti týmu.

Krok 3: Výběr jedné nebo více částí systému k úpravě

V této iteraci budeme pracovat se všemi komponentami (a jejich vnitřními elementy), tj. Message Box, Agent, Agent Controller, Messaging, Configuration Service a podpůrnými službami pro aplikační metriky a agregaci logů.

Krok 4: Výběr designových konceptů, které uspokojují vybrané požadavky

Rozhodnutí	Odůvodnění
Výběr relační databáze pro Message Box	Jelikož se jedná o databázi pro uložení režijních dat, uvažujeme primárně analytické dotazy a ne tak vysoký objem dat.
Výběr message broker technologie pro messaging	V rámci messagingu budou použity dva styly komunikace - Pub/Sub a Point-Point. Message broker technologie je navržena přímo pro tyto způsoby komunikace.
Výběr distribuovaného search engine pro indexaci a vyhledávání v agregovaném logu	Pro vyhledávání je vhodné použít full-text search engine. Z důvodů většího objemu dat a nároků na dostupnost by měl být ideálně distribuovaný.
Výběr řešení pro ukládání časových řad metrik a jejich následnou analýzu a upozornění	Metriky data v čase a proto je vhodné využití řešení používající databázi pro časové řady.
Výběr řešení pro agregaci a filtraci logů z shipper elementů	Je vhodné vybrat řešení, které umožní agregaci, filtraci, případnou transformaci logů a jejich následné odeslání do vyhledávacího engine.
Výběr řešení pro vizualizaci metrik	Vyhledávání a práce s metrikami přímo v řešení pro jejich kolekci a uchování je často možné, ovšem ne velmi praktické. Ideální je volba další komponenty, která umí s těmito daty pracovat, vizualizovat je a poskytnout přehled o celém systému.

Použití jazyka Java a frameworku na něm postavených pro služby a Sidecar u agenta	Java je nezávislá na platformě, pro naše potřeby dostatečně rychlá, pohodlná na vývoj a aktuální tým s ní má zkušenosti.
Použití jazyka Python pro skripty Agentů	Aktuální tým má s jazykem Python zkušenosti a někteří agenti jsou již dnes psáni v jazyce Python (jejich kód tak bude možné znovupoužít).

Tabulka 5.15: Návrhová rozhodnutí ve čtvrté iteraci

Alternativa	**Důvod nevybrání**
Výběr NoSQL databáze pro Message Box	Nejedná se o ukládání Big Dat a je potřeba provádět spíše analytické dotazy.
Použití sdílené databáze pro messaging	Je možné namodelovat funkcionalitu messagingu pomocí databázových systémů. Ty na to ovšem nejsou navrženy a jejich použití pro fronty je spojeno často s výkonnostními problémy.
Použití databáze pro indexaci a vyhledávání v logu	Implementace této funkčnosti v rámci relačních i NoSQL databází je možný, ovšem implementace není primitivní a tyto nástroje na to nejsou primárně určeny.

Tabulka 5.16: Alternativní rozhodnutí ve čtvrté iteraci

Krok 5: Vytvoření elementů, rozdělení zodpovědnosti a definice rozhraní

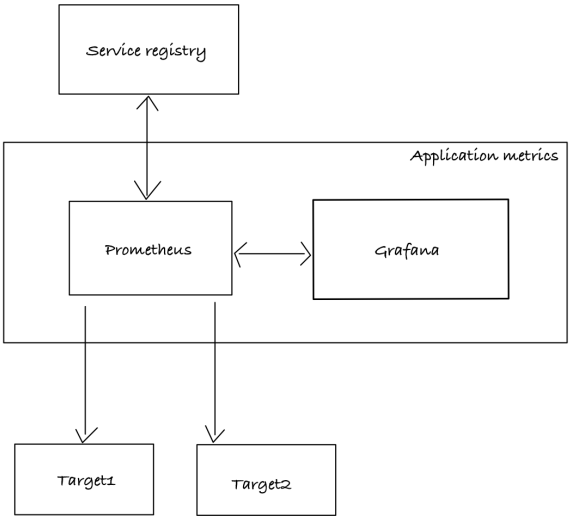
Rozhodnutí	**Odůvodnění**
Použití PostgreSQL jako relační databáze pro Message Box	PostgreSQL je rozšířená SQL open source databáze. Pro potřeby systému plně dostačující.
Použití RabbitMQ jako message broker technologie	RabbitMQ je navržena přesně pro naše použití. Alternativou by mohlo být použití Apache Kafka, ovšem ta je lepší pro události a persistentní append-only log. RabbitMQ je také použit ve stávajícím systému a tak má aktuální tým zkušenosti s jeho použitím a provozem.

Použití ELK (+beats) stack pro agregaci, filtraci a indexaci logů	ELK stack je nejrozšířenějším řešením pro tyto účely a poskytuje navíc i řešení pro vizualizaci.
Použití Prometheus a Grafany pro aplikační metriky	Jedná se o často používané řešení pro aplikační metriky a jejich vizualizaci. Alternativní TICK hůře škáluje a prometheus je navržený pro sběr metrik pomocí metody PULL, zatímco TICK spíše PUSH (metoda PULL byla zvolena v předchozích iteracích).
Použití frameworku Spring Boot pro služby Message Box, Agent Controller a Configuration Service	Tyto služby nemají nároky na rychlý start či nízkou spotřebu paměti. Tým má s tímto frameworkem také zkušenosti, které jsou schopni do nového vývoje přenést.
Použití frameworku Micronaut pro Agent Sidecar	Tato služba by měla být schopna rychlého startu a nemít vysoké nároky na paměť, protože bude spouštěna s každou instancí Agenta, kterých může být velké množství. Micronaut je preferovaný před frameworkem Quarkus, protože má blíže k frameworku Spring Boot.
Použití knihoven pro aplikační metriky pro frameworky Spring Boot a Micronaut	Pro ulehčení implementace je možné použít již připravené knihovny, které umí poskytovat metriky v požadovaném formátu.

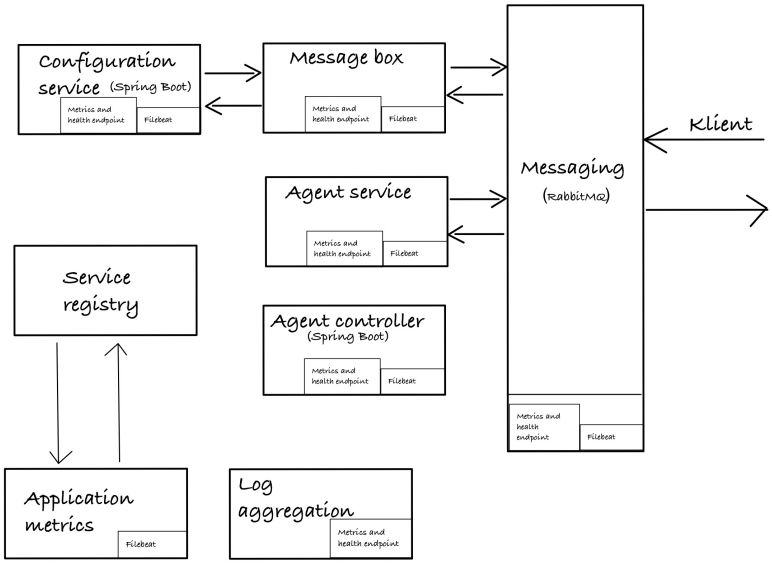
Tabulka 5.17: Krok 5 ve čtvrté iteraci

Krok 6: Architektonické pohledy a zaznamenání rozhodnutí

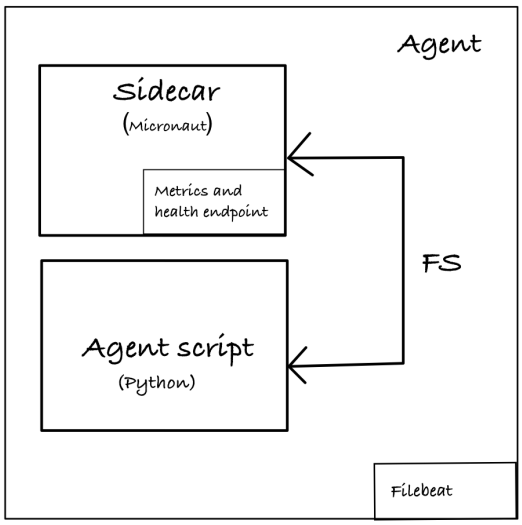
Následující pohledy jsou pohledy z iterací 2 a 3 doplněné o vybrané technologie. Obrázek 5.7 ukazuje použití technologií vybraných pro sběr a vizualizaci aplikačních metrik. Obrázek 5.8 ukazuje technologie použité pro komunikaci a pro elementy bez většího rozpadu. Obrázek 5.11 ukazuje použití ELK stacku v systému pro agregaci logů a obrázky 5.10 a 5.9 ukazují technologie použité pro služby Agent a Message Box.



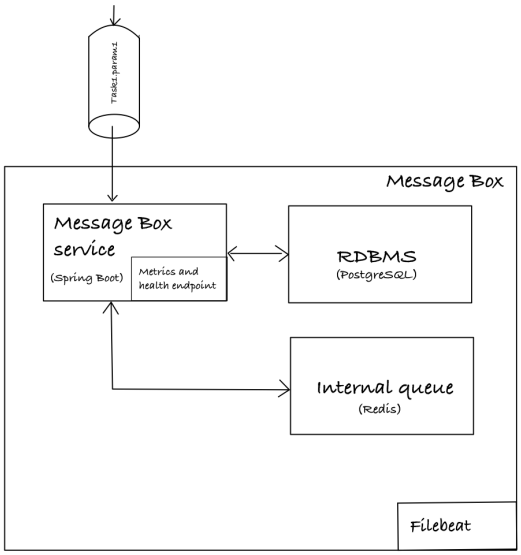
Obrázek 5.7: Technologie vybrané v rámci iterace 4 pro aplikační metriky



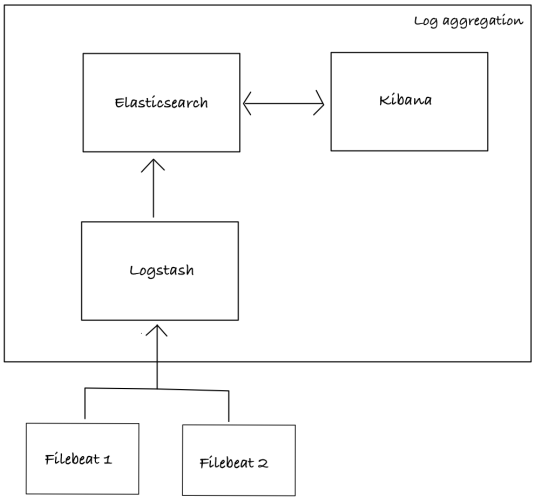
Obrázek 5.8: Technologie vybrané v rámci iterace 4 pro komunikaci a jednoelementové komponenty



Obrázek 5.9: Technologie vybrané v rámci iterace 4 pro elementy služby agenta



Obrázek 5.10: Technologie vybrané v rámci iterace 4 pro elementy Message Box služby



Obrázek 5.11: Technologie vybrané v rámci iterace 4 pro agregaci logů

Krok 7: Analýza současného stavu designu a dosažení cílů iterace

Nenaplněné	Částečně naplněné	Úplně naplněné	Designové rozhodnutí
		QA-2	Technologie pro agregaci logů a notifikaci je vybraná, stálý příjem bude poskytovat messaging komponenta.
		QA-10	Agregace logů a zvolené technologie (ELK stack) plně naplňují tento scénář.
		QA-11	Aplikační metriky a zvolené technologie (Prometheus+Grafana) plně naplňují tento scénář.

Tabulka 5.18: Naplnění cílů iterace 2

5.4.6 Iterace 5: Distribuce a nasazení

Krok 2: Výběr cíle iterace

Cílem této iterace je výběr technologií pro distribuci a nasazení systému.

Krok 3: Výběr jedné nebo více částí systému k úpravě

V této iteraci se soustředíme na všechny komponenty systému, které je nutné distribuovat a provozovat. Jedná se o jednotlivé služby plnící samotnou funkčnost systému i o podpůrné služby.

Krok 4: Výběr designových konceptů, které uspokojují vybrané požadavky

Rozhodnutí	Odůvodnění
Využití kontejnerizace pro distribuci artefaktů	Využití kontejnerizace nám umožní artefakty zabalit spolu s jejich závislostmi do jednoho balíčku, tzv. kontejneru. To umožňuje konzistentní běh na různých infrastrukturách a odstínění od běhového prostředí.
Využití nástrojů pro orchestraci kontejnerů	Nástroje pro orchestraci kontejnerů umožní automatizaci práce s nimi. Zajišťují mimo jiné nasazování, škálování, životní cyklus či automatické restartování selhaných komponent.

Tabulka 5.19: Návrhová rozhodnutí v páté iteraci

Krok 5: Vytvoření elementů, rozdělení zodpovědnosti a definice rozhraní

Rozhodnutí	Odůvodnění
Využití technologie Docker	Jedná se o nejrozšířenější technologii pro kontejnerizaci.
Docker image jako výstup sestavení všech artefaktů	Jelikož bude použita technologie docker, je třeba ,aby výstupy byly ve formě docker image.
Docker image služby Agenta bude složen ze dvou vrstev produkovaných týmem	Rozdělení na dvě vrstvy, kde jedna bude obsahovat Sidecar a druhá bude stát nad ní a přidávat konkrétní skript, nám dá možnost sestavovat tyto části nezávisle a přidá další odstínění vývojářů skriptu od zbytku systému.

Použití kubernetes jako nástroje pro orchestraci	Jedná se o rozšířené řešení s bohatými zdroji. Výhodou je dostupnost managed řešení v cloudu, v případě, že by tým nechtěl nadále provozovat cluster sám, může bez větších obtíží provést migraci k nějakému z větších poskytovatelů cloudových služeb.
Použití nástroje Helm	Helm je správce balíčků a šablonovací engine pro Kubernetes. Ulehčuje práci s kubernetes a usnadňuje instalaci komponent třetích stran.
Škálování služby Agenta pomocí kubernetes Jobů	Krom možnosti mít fixní množství instancí agenta pro daný typ úlohy můžeme dále škálovat agenty pomocí spouštění pomocných instancí jako kubernetes Job. Alternativním řešením by bylo použití automatického škálování poskytovaného přímo kubernetes, to ovšem nebere to, zda agent zrovna vykonává nějakou práci či ne. Pro Joby bude nutné nastavit time-to-live, protože ve výchozím nastavení zůstávají Pody po dokončení Jobu existovat kvůli například sběru logů.
Použití kubernetes service discovery	Kubernetes nabízí nativní service discovery, která je pro naše potřeby plně dostačující. Navazující technologie jako Promehteus umí s kubernetes SD komunikovat out-of-the-box.
Použití kubernetes API v rámci služby Agent Controller	Pokud chceme škálovat agenty formou kubernetes Jobu, je třeba integrace na kubernetes API, přes které budou Joby spouštěny.
Použití kubernetes secrets pro uložení klíčů a certifikátů	Požadavky v rámci systému budou podepisovány pomocí privátních klíčů. Stejně tak požadavky od centrálního systému a odpovědi budou podepisovány. Kubernetes secrets jsou vhodným nástrojem pro uložení těchto klíčů a jejich sdílení mezi komponentami v clusteru.
Využití RabbitMQ operátoru pro kubernetes	Jedná se o existující řešení pro správu a nasazení RabbitMQ clusteru na kubernetes.
Využití kube-prometheus-stack helm chartu	Tento balíček zajistí nasazení a provoz Prometheus+Grafany a přidává monitoring celého kubernetes cluseteru a prostředky pro definici monitoringu aplikací pomocí kubernetes entit.

Využití helm chartu pro ELK stack	Pro každou z komponent ELK stacku existuje helm chart, kterým je možné je nasadit. Dále pak bude použit helm chart pro Filebeat, který jej do clusteru nasadí jako tzv. *DaemonSet*, což zajistí běh jednoho Podu na každém worker node, kde bude sbírat logy ze standardního výstupu spuštěných kontejnerů.

Tabulka 5.20: Krok 5 v páté iteraci

Krok 6: Architektonické pohledy a zaznamenání rozhodnutí

V této iteraci není potřeba obnovovat samotné pohledy, na struktuře systému se nic nezměnilo.

Krok 7: Analýza současného stavu designu a dosažení cílů iterace

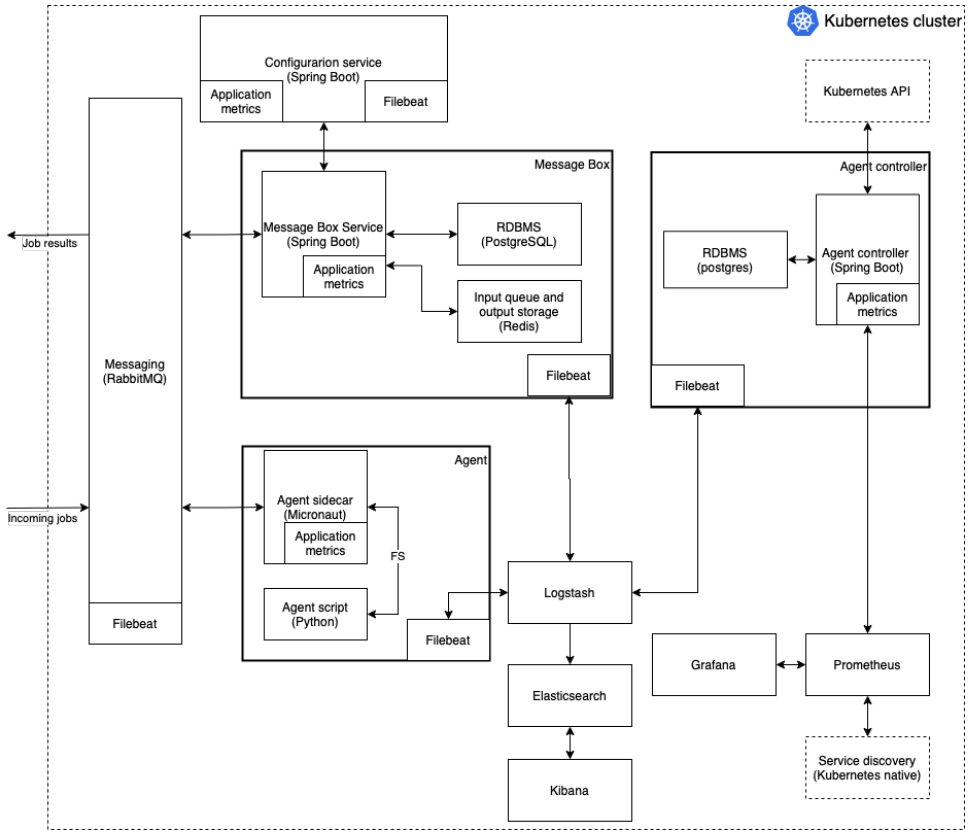
Nenaplněné	Částečně naplněné	Úplně naplněné	Designové rozhodnutí
		QA-1	Po selhání Agenta bude zajištěno spuštění nové instance buďto kubernetes (pokud se jednalo a stálou instanci) či Agent controllerem při detekci stagnace fronty.
		QA-3	Díky kombinaci technologií docker, kubernetes a helm je možné takto aplikace nasazovat. Jelikož služby Agent i Message Box jsou za messaging rozhraním, z pohledu klienta nedochází ke snížení dostupnosti.
		QA-4	Detekce situace probíhá sledováním aplikačních metrik. Následné spouštění nové komponenty proběhne službou Agent Controller pomocí Kubernetes API.
		QA-5	V případě selhání komponenty kubernetes zajistí její opětovný start.
		QA-7	Autorizaci požadavku je možné ověřit jeho podpisem.

QA-12	V případě selhání komponenty kubernetes zajistí její opětovný start. Pokud se jednalo o instanci Agenta spuštěnou jako Job, bude restart zajištěn také kubernetes, případně bude spuštěna nová instance pomocí Agent Controlleru.

Tabulka 5.21: Naplnění cílů iterace 5

5.4.7 Finální diagram nové architektury

Na obrázku 5.12 je zobrazena finální podoba nové architektury tak, jak byla navržena v předchozích iteracích metody ADD. Pro přehlednost jsou vynechány spojení mezi komponentou Prometheus a jednotlivými elementy, které monitoruje.



Obrázek 5.12: Nová architektura

5.4.8 Návrh procesu vývoje, nasazení a provozu

V této části bude popsán nástin doporučeného procesu vývoje, nasazení a provozu. Nejedná se o technický a detailní popis, spíše o sadu doporučení a možností spojených s novou architekturou.

Vývoj

Vývoj komponent systému bude verzován pomocí nástroje Git a Gitlab. Oba dva nástroje používá vývojový tým již nyní a mají s ním tak praktické zkušenosti. Každá služba bude mít svůj repozitář a své CI/CD pipelines. Pro službu Agent Service budou použity repozitáře dva - jeden pro Sidecar komponentu a jeden se skripty pro jednotlivé úlohy. Jako model pro vývoj

s verzovacím systémem Git bude použit Trunk Based Development[38]. Ten je jednodušší než alternativní GitFlow model a dostačující pro náš systém.

Pro většinu vývojových činností není potřeba mít lokální Kubernetes cluster, výjimku tvoří jen vývoj služby Agent Controller, která komunikuje přímo s Kubernetes API. Pro zbylé služby postačí lokální vývojové nástroje (JDK, Python, IntelliJ) a Docker, případně Docker-Compose, pro spuštění dalších komponent (například RabbitMQ).

Tvorba artefaktů

Tvorba artefaktů bude probíhat v rámci CI pipeline v nástroji GitLab. Každá služba bude mít ve svém repozitáři Dockerfile pro sestavení docker image. V rámci běhu pipeline budou spuštěny jednotkové testy, integrační testy a statická analýza kódu (například nástroj SonarQube[2]).

Speciálním případem zde je tvorba artefaktů jednotlivých agentů. Pro každou úlohu vznikne jeden docker image (tj. pro každou bude v repozitáři se skripty existovat Dockerfile). Tento docker image bude vycházet z docker image sestaveného pro Agent Sidecar a bude ho doplňovat o konkrétní skript.

Každý docker image bude umístěn do interního docker image repozitáře.

Nasazení

Jelikož bude celý systém provozován na Kubernetes a bude používat balíčkovací nástroj Helm, bude existovat repozitář, který bude obsahovat veškeré Helm charts a Kubernetes yaml soubory, které budou potřeba k nasazení a správě systému. Systém pak bude nasazován pomocí nástroje Helm a definovaných charts, v případě potřeby více low-level zásahů pak přímo pomocí Kubernetes klienta.

Pro každé provozní prostředí bude existovat samostatný repozitář (všechny budou mít stejnou strukturu) obsahující všechny potřebné definice. Alternativní možností je jeden sdílený repozitář a použití větví. Možnost "repozitář per prostředí" je zde zvolena pro jednodušší údržbu a z důvodu, že by se zde nejednalo o standardní použití větví pro prostředí, kdy se změny z jednoho zpropagují časem do ostatních, ale zde se může jednat o vyloženě rozdílné konfigurace.

Provoz

Architektura bude provozována na aktuální infrastruktuře vlastněné týmem, která projde transformací na Kubernetes cluster. Jedná se tedy o nasazení typu on-premise. Případná migrace do cloudu je možná, ovšem zde je potřeba zvážit náklady, které se u většiny poskytovatelů počítají dle prostředků a výpočetního času.

Díky použití Docker a Kubernetes je vývoj od výsledné infrastruktury, na které cluster bude spuštěn, odstíněn a případná migrace způsobí minimální zátěž pro tým.

[2] https://www.sonarqube.org

Kapitola 6

Prototyp

V této kapitole bude popsáno otestování požadavků architektury (ve formě scénářů) pomocí připraveného prototypu. Prototyp používá technologie zvolené v rámci návrhu.

Prototyp je připraven pro lokální testování na lokálním Kubernetes clusteru. Nejedná se o produkčně nastavené podpůrné služby a komponenty (tj. prototyp neřeší perzistentní uložení dat apod.).

Postup pro lokální spuštění prototypu je popsán v příloze A.Pro otestování prototypu je potřeba základní znalost ovládání Kubernetes clusteru pomocí `kubectl` a základní znalost linuxové příkazové řádky.

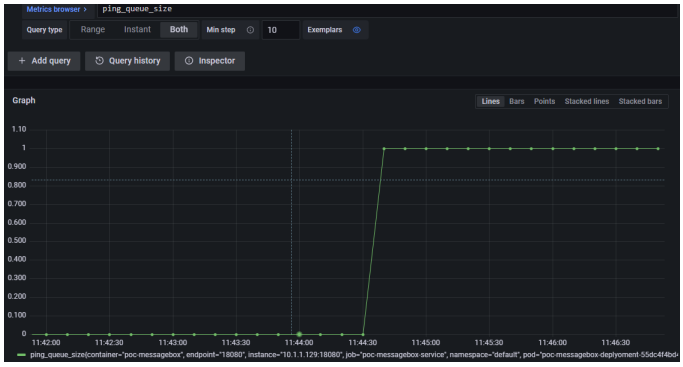
6.1 QA-1

Pro tento scénář si připravíme prototyp tak, že běží instance všech služeb. Následně odebereme z clusteru službu Agent. Služba sice implementuje endpoint `/crash` pro simulaci selhání, ovšem pro tento scénář není vhodný (Kubernetes by službu restartovalo). Pomocí RabbitMQ pošleme nové úlohy. Na obrázku 6.1 vidíme nárůst čekajících úloh. Na obrázku 6.2 vidíme log o ukončení služby v centrálním přehledu logů. Při nastavení upozornění ve službě Kibana je pak možné odesílat notifikace při takovém logu.

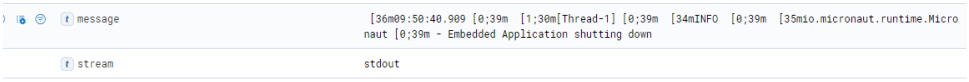
Níže vidíme také výstup příkazu `kubectl get pods -w`, kde vidíme vypnutí a odebrání služby Agent.

```
NAME                              READY   STATUS
ping-agent-deplyoment-f6cc7bd     1/1     Running

ping-agent-deplyoment-f6cc7bd     1/1     Terminating
ping-agent-deplyoment-f6cc7bd     0/1     Terminating
```



Obrázek 6.1: Metrika pro počet čekajících úloh ve službě Message Box



Obrázek 6.2: Záznam o vypnutí aplikace v centrálním přehledu logů

6.2 QA-2

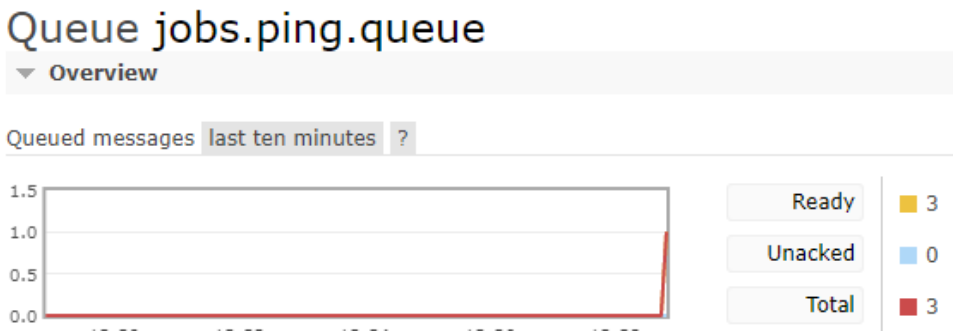
Pro tento scénář nám stačí, aby v rámci prototypu byla spuštěna služba Message Box (mohou být spuštěny i další). Selhání můžeme simulovat podobně jako na službě Agent pomocí endpointu `/crash`, ovšem Kubernetes by zajistilo restart služby. Proto službu odebereme z clusteru úplně. Následně odešleme pomocí RabbitMQ nové úlohy. Ty budou z pohleda klienta systémem přijaty.

Níže vidíme také výstup příkazu `kubectl get pods -w`, kde vidíme vypnutí a odebrání služby Message Box.

```
NAME                          READY   STATUS
poc-messagebox-deplyoment     3/3     Running

poc-messagebox-deplyoment     3/3     Terminating
poc-messagebox-deplyoment     0/3     Terminating
poc-messagebox-deplyoment     0/3     Terminating
poc-messagebox-deplyoment     0/3     Terminating
```

Na obrázku 6.3 vidíme počet úloh čekajících na rozhraní. Na obrázku 6.3 vidíme log o ukončení služby v centrálním přehledu logů. Při nastavení upozornění ve službě Kibana je pak možné odesílat notifikace při takovém logu.



Obrázek 6.3: Počet přijatých zpráv, které čekají na rozhraní na vyzvednutí

# log.offset	2,814
t message	1:signal-handler (1652268763) Received SIGTERM scheduling shutdown...
t stream	stdout

Obrázek 6.4: Záznam o vypnutí aplikace v centrálním přehledu logů

6.3 QA-3

Tento scénář můžeme v rámci prototypu otestovat bez použití technologie Helm. Vytvoříme si nový Docker image pro službu Message Box s jiným jménem (v praxi by byl použit tag pro verzování). Následně upravíme název Docker image v definici Kubernetes zdroje Deployment pro tuto službu a provedeme příkaz `kubectl apply -f messagebox.yaml`. Tento příkaz provede nejdříve vypnutí původní služby a nasadí novou.

Níže vidíme výstup příkazu `kubectl apply -f messagebox.yaml` po změně názvu image a můžeme vidět, že se změnil Deployment.

```
deployment.apps/poc-messagebox-deplyoment configured
service/poc-messagebox-service unchanged
servicemonitor/test-app-service-monitor unchanged
```

Níže vidíme výstup příkazu `kubectl get pods -w`, který byl spuštěn před aplikací nového yaml souboru. Vidíme zde, že byla nejdříve nasazena nová verze aplikace a následně pak vypnuta verze původní. Vše plně automaticky.

```
NAME                                READY   STATUS
poc-messagebox-deplyoment-55...     3/3     Running

poc-messagebox-deplyoment-5f...     0/3     Pending
poc-messagebox-deplyoment-5f...     0/3     Pending
```

```
poc-messagebox-deployoment-5f... 0/3     Containe...
poc-messagebox-deployoment-5f... 3/3     Running
poc-messagebox-deployoment-55... 3/3     Terminating
poc-messagebox-deployoment-55... 0/3     Terminating
poc-messagebox-deployoment-55... 0/3     Terminating
poc-messagebox-deployoment-55... 0/3     Terminating
poc-messagebox-deployoment-5f... 2/3     Error
poc-messagebox-deployoment-5f... 3/3     Running
```

Pomocí příkazu `kubectl describe pod nový_pod` je možné zvalidovat použití nového image.

Díky použití nepřímé komunikace a messagingu nemá tato operace vliv na dostupnost a z pohledu klienta je systém stále dostupný.

V rámci produkčního nasazení by pro tyto potřeby byl použit Helm a příkaz `helm upgrade`. Ten by udělal render nové šablony yaml souboru s novou verzí a tu poté také aplikoval. Dále by tento proces byl automatizován v rámci CI/CD pipeline.

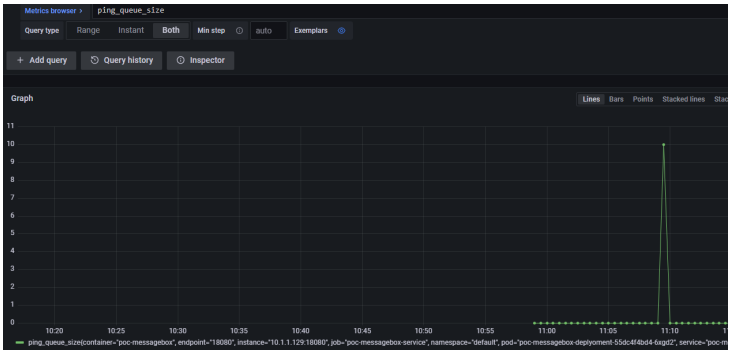
6.4 QA-4

Pro tento scénář si připravíme prototyp tak, že není spuštěna žádná instance agenta, která by odbavovala práci. V clusteru máme nasazenu jednu instanci služby Message Box a službu Agent Controller. Následně odešleme na testovací RabbitMQ vyšší počet požadavků, které nebudou odbavovány. Služba Agent Controller pomocí sledování metrik ve službě Prometheus detekuje nárůst a spustí instanci přislušného agenta.

Níže vidíme výstup příkazu `kubectl get jobs -w`, který ukazuje spuštění a dokončení Jobu po odbavení celé fronty. Perioda kontroly metriky ve službě Agent Controller je nastavena na 10 sekund, Job je tedy spuštěn nejpozději 10 sekund od detekce nárůstu úloh.

```
NAME                   COMPLETIONS   DURATION   AGE
support-ping-agent-1   0/1                      0s
support-ping-agent-1   0/1           0s         0s
support-ping-agent-1   1/1           35s        35s
```

Následně na obrázku 6.5 vidíme postupný nárůst a pokles počtů úloh ke zpracování ve frontě ve službě Grafana.



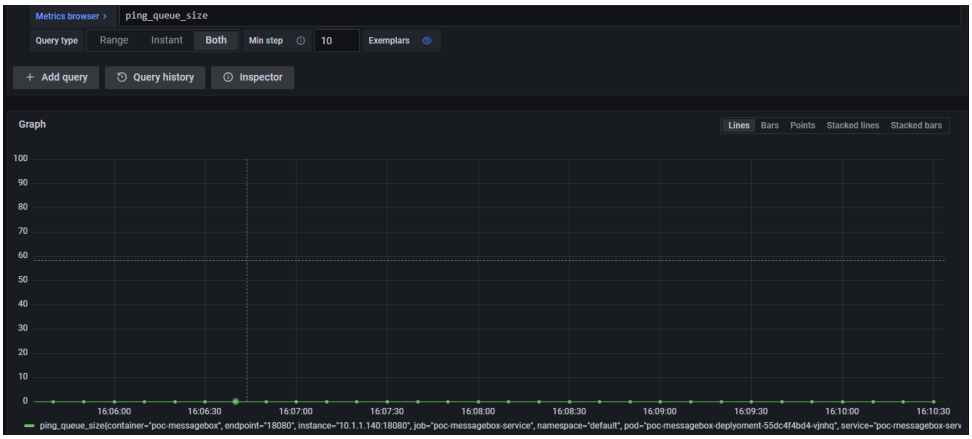
Obrázek 6.5: Metrika pro počet čekajících úloh ve službě Message Box

6.5 QA-5

Pro tento scénář si připravíme prototyp do stavu, kdy jsou spuštěny všechny služby. Na rozhraní systému pošleme následující úlohu:

```
{
    "jobID": "testId1",
    "jobType": "invalid",
    "parameters": { "address": "fel.cvut.cz" }
}
```

Na obrázku 6.7 vidime, že chybný požadavek je dostupný v centrálním přehledu logů. Zároveň požadavek nebyl zařazen ke zpracování, jak ukazuje obrázek 6.6.



Obrázek 6.6: Počet zpráv v interní frontě

t message	2022-05-11 14:09:35.771 ERROR 1 --- [ntContainer#0-1] c.s.c.f.d.m.r.IncomingJobs Receiver : Invalid request {"jobID": "testId1","jobType": "invalid","parameters": {"address":"fel.cvut.cz"}}
t stream	stdout

Obrázek 6.7: Záznam o nevalidním požadavku v centrálním přehledu logů

6.6 QA-6

Pro tento scénář si připravíme prototyp do stavu, kdy jsou spuštěny služby Message Box a Configuration service. Pomocí příkazu

```
kubectl port-forward
service/configuration-service-service 18080
```

si vytvoříme prostup na službu přes localhost. Následně použijeme následující příkaz pro simulaci nahrání nové konfigurace:

```
echo $(date +"%T") && curl localhost:18080/
configuration/ping
-H 'Content-type: application/json' --data '{"field":
 "new_value"}'
```

Na výstupu uvidíme zopakovanou odeslanou konfiguraci a čas odeslání. Výstup příkazu je níže.

```
17:36:39
{"field": "new_value"}
```

Následně v logu služby Message Box 6.8 uvidíme záznam o stažení nové konfigurace a jejím uložení. Log vidíme v čase 15:36:42.444 (UTC, tedy 17:36:41.444), tedy v žádaném limitu jedné minuty.

```
2022-05-14 15:36:42.444  INFO 1 --- [pool-4-thread-1] c.f.d.m.ConfigurationRefresher$ConfigJob :
 Received new configuration {"field": "new_value"}
```

Obrázek 6.8: Log služby Message Box

6.7 QA-7

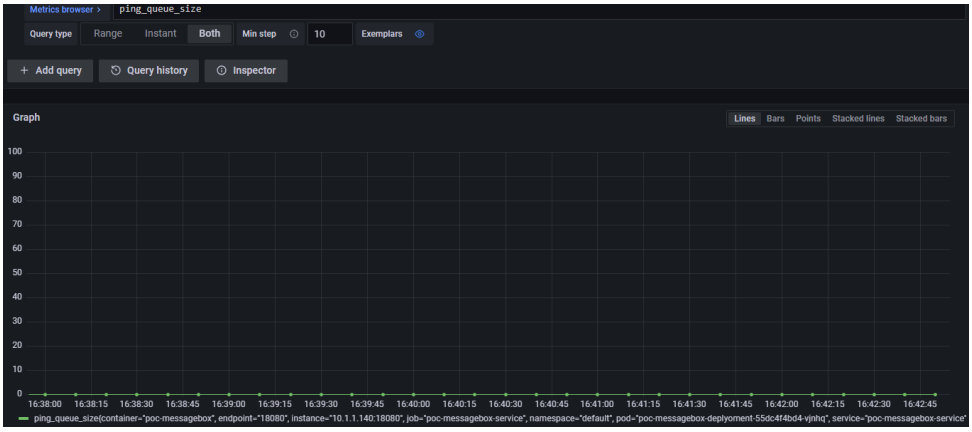
Scénář QA-7 je ověřen stejným postupem jako scénář QA-5, jen za použití jiné zprávy:

```
{
    "jobID": "testId1",
```

```
            "jobType": "unauthorized",
            "parameters": { "address": "fel.cvut.cz" }
        }
```

V rámci prototypu není implementováno podepisování zpráv a ověření podpisu, funkčnost je simulována typem úlohy `unauthorized` v obsahu zprávy.

Na obrázku 6.10 vidime, že neautorizovaný požadavek je dostupný v centrálním přehledu logů. Zároveň požadavek nebyl zařazen ke zpracování, jak ukazuje obrázek 6.9.



Obrázek 6.9: Počet zpráv v interní frontě

t message	2022-05-11 14:42:17.675 ERROR 1 --- [ntContainer#0-1] c.s.c.f.d.m.r.IncomingJobsReceiver : Unauthorized request {"jobID": "testId1","jobType": "unauthorized","parameters": {"address":"fel.cvut.cz"}}
t stream	stdout

Obrázek 6.10: Záznam o nevalidním požadavku v centrálním přehledu logů

6.8 QA-8

Tento scénář není v rámci prototypu ověřován. Situace je ošetřena návrhem systému a použitím interní fronty.

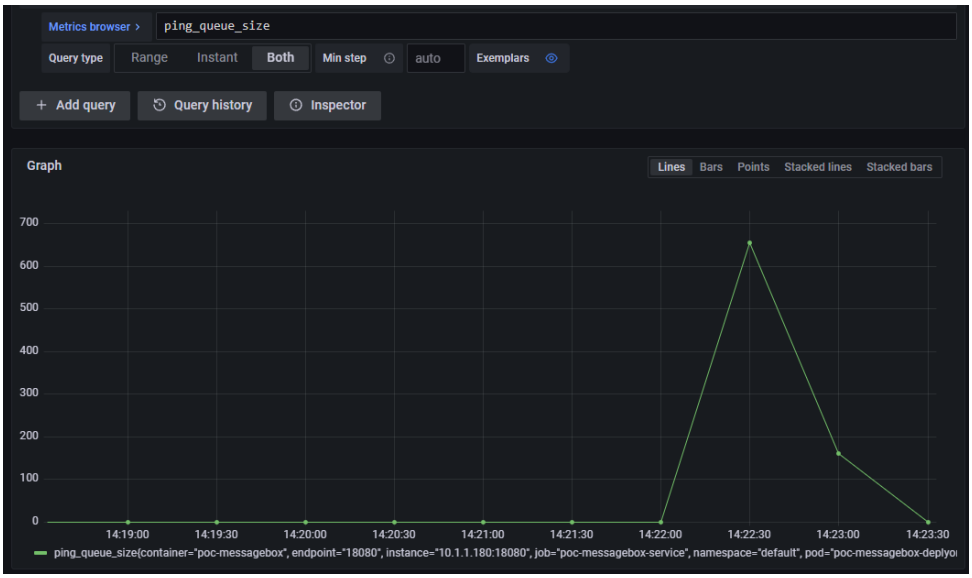
6.9 QA-9

Pro tento scénář máme v rámci prototypu nasazené služby Message Box a Agent. Spustíme příkaz `kubectl get pods -w`, zde můžeme sledovat, zda žádná z komponent neselže. Následně použijeme připravený Kubernetes Job pro tento scénář. Ten v jednoduché smyčce odešle 1000 úloh na RabbitMQ.

```
NAME                                READY   STATUS
ping-agent-deplyoment-7584ccd..     1/1     Running
poc-messagebox-deplyoment-55d..     3/3     Running
rabbit-mq-test-cluster-server       1/1     Running

qa-9-spam-job--1-glgdz              0/1     Pending
qa-9-spam-job--1-glgdz              0/1     Pending
qa-9-spam-job--1-glgdz              0/1     Creating
qa-9-spam-job--1-glgdz              1/1     Running
qa-9-spam-job--1-glgdz              0/1     Completed
```

Z výstupu sledování stavu podů vidíme vytvoření podu pro spuštění jobu a že žádná z komponent systému neselhala. Ve službě Grafana můžeme sledovat metriku velikosti interní fronty. Na obrázku 6.11 vidíme, že zprávy byly přijaty a přesunuty na interní fronty do 30 sekund (metrika je zde o něco nižší, protože již došlo ke zpracování některých úloh). Do jedné minuty a 30 sekund byly zpracovány všechny úlohy.



Obrázek 6.11: Výsledek dotazu na metriku s velikostí interní fronty

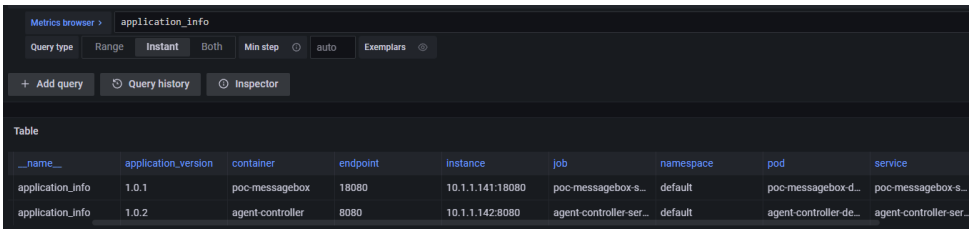
6.10 QA-10

Tento scénář je ověřen již předchozími scénáři, kde je ukázáno, že se události v logu objevují v centrálním přehledu. Filebeat běžící na každém worker uzlu kontroluje nové události s frekvencí určenou parametrem `scan_frequency`, ten je ve výchozím nastavení nastaven na 10 sekund.

6.11 QA-11

Pro tento scénář máme v rámci prototypu nasazené služby Message Box a Agent controller. Obě tyto služby poskytují metriku s informací o verzi aplikace. Metriku je následně možné sledovat ve službě Grafana a případně vytvořit přehledný dashboard s verzemi služeb.

Na obrázku 6.12 je výsledek dotazu na metriku s verzemi služeb.



Obrázek 6.12: Výsledek dotazu na metriku s informacemi o verzích služeb

6.12 QA-12

Pro tento scénář si připravíme prototyp tak, že budeme mít nasazenu instanci služby Message Box a Agent. Pomocí RabbitMQ odešleme úlohu, pro kterou prototyp nevrací žádnou odpověď.

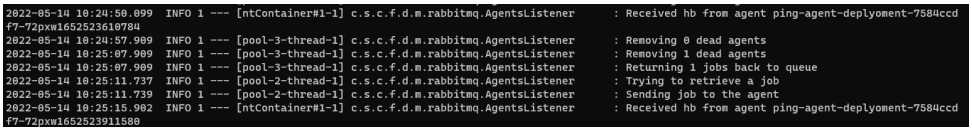
```
{
    "jobID": "testId1",
    "jobType": "swallow",
    "parameters": { "address": "fel.cvut.cz" }
}
```

Spustíme příkaz `kubectl get pods -w` a provedeme volání endpointu `/crash` na službě Agent.

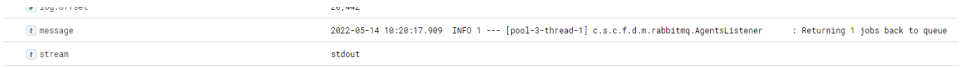
```
NAME                              READY STATUS    RESTARTS
ping-agent-deplyoment-7584c... 1/1   Running  1(4m42s ago
)

ping-agent-deplyoment-7584c... 0/1   Error    1(4m49s ago
)
ping-agent-deplyoment-7584c... 0/1   CrashL.. 1(14s ago)
ping-agent-deplyoment-7584c... 1/1   Running  2(14s ago)
```

Jak vidíme ve výpisu příkazu, pod s příslušnou službou byl restartován. Na obrázku 6.13 je výpis logů služby, na kterém je vidět, že úloha byla znovu zařazena do interní fronty a vyzvednuta novou instancí služby Agent. Na obrázku 6.14 vidíme, že log je dostupný i v centrálním přehledu logů.



Obrázek 6.13: Log služby Message Box



Obrázek 6.14: Záznam v centrálním přehledu logů

Kapitola 7

Vyhodnocení nové architektury

Architekturu budeme vyhodnocovat na základě naplnění námi připravených scénářů. Částečné hodnocení můžeme vidět již na konci návrhu, kdy díky použité metodice ADD a přehledu na konci každé iterace víme, do jaké míry máme které scénáře naplněny. Díky prototypu můžeme tyto scénáře také prakticky vyhodnotit a díky měřitelnosti jejich naplnění (response measure) můžeme říci, zda jsou splněny.

7.1 Naplnění definovaných scénářů

V následující tabulce je přehled scénářů. Ke každému je uvedeno, zda byl naplněn v rámci návrhu, zda dosáhl požadovaných výsledků v rámci prototypu a případný komentář.

Scénář	**Naplněno v návrhu**	**Verifikace prototypem**	**Komentář**
QA-1	Ano	Ano	V rámci scénáře nebylo nastaveno odeslání notifikací ze služby Kibana.
QA-2	Ano	Ano	V rámci scénáře nebylo nastaveno odeslání notifikací ze služby Kibana.
QA-3	Ano	Ano	
QA-4	Ano	Ano	
QA-5	Ano	Ano	
QA-6	Ano	Ano	
QA-7	Ano	Ano	
QA-8	Ano	-	
QA-9	Ano	Ano	
QA-10	Ano	Ano	
QA-11	Ano	Ano	
QA-12	Ano	Ano	

Tabulka 7.1: Naplnění scénářů

Kapitola 8

Závěr

Cílem práce byla analýza stávající architektury, sběr požadavků na novou architekturu, včetně jejich ohodnocení z byznysového a technického pohledu, a samotný návrh architektury nového systému dle požadavků.

Nejprve byla provedena analýza stávající architektury. Architektura byla následně popsána a to včetně jejich aktuálních nedostatků. Analýza aktuálního řešení nám pomohla pochopit jaký je aktuální stav a jaké jsou aktuální zkušenosti týmu s různými technikami vývoje a technologiemi.

Následně proběhla rešerše architektonických stylů, vzorů, taktik a technologií, které by bylo možné použít. Díky této rešerši bylo možné v následném návrhu zvolit vhodné styly, vzory, taktiky a technologie, a bylo možné je porovnat.

Pro samotný návrh architektury byla zvolena metodika ADD využívající atributy kvality a jejich scénáře. Nejdříve byl představen koncept scénářů a jednotlivé, pro tuto práci relevantní, atributy kvality a nakonec samotná metodika. Následovalo vytvoření samotných scénářů, které byly založeny na požadavcích ze zadání a konzultovány se zadavatelem. Ten následně provedl jejich byznysovou prioritizaci. Následovala technická prioritizace scénářů tak, aby bylo možné se nejdříve soustředit na náročnější a byznysově důležitější scénáře.

Samotný návrh proběhl v pěti iteracích použité metodiky. Ty postupně zvyšovaly detail pohledu na systém a postupně naplňovaly scénáře atributů kvality. Tento postup vytvořil dokumentaci, která může být použita při dalším rozvoji architektury, či při diskuzích nad zvolenými řešeními. Poslední částí návrhu byl popis navrhovaných procesů pro vývoj, nasazení a provoz navrženého systému.

Nakonec byl realizován prototyp, který umožnil praktické otestování naplnění měřitelných požadavků na scénáře. Zároveň ukazuje použití technologií a vzorů použitých při návrhu systémů. Scénáře byly postupně na prototypu testovány a ke každému byl zdokumentován výstup z jeho testování a zda byl naplněn či nikoliv.

Architektura byla nakonec vyhodnocena na základě výstupů ze samotného návrhu a z testování prototypem. Architektura splňuje všechny požadavky uvedené v zadání a díky použité metodice je poskytnuta dokumentace, která implementátorům poskytne vhled a důvody pro zvolená řešení.

Literatura

[1] AVIZIENIS, A., J.-C. LAPRIE, B. RANDELL a C. LANDWEHR, 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* [online]. **1**(1), 11-33 [cit. 2022-01-08]. ISSN 1545-5971. Dostupné z: doi:10.1109/TDSC.2004.2

[2] ISO 25000. *ISO 25000 Portal* [online]. [cit. 2022-01-08]. Dostupné z: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[3] BASS, Len, Paul CLEMENTS a Rick KAZMAN, 2021. *Software architecture in practice*. 4th ed. Boston: Addison-Wesley. ISBN 978-0136886099.

[4] PILLAI, Anand Balachandran, 2017. *Software architecture with Python: design and architect highly scalable, robust, clean, and high performance applications in Python*. Birmingham: Packt. ISBN 978-1786468529.

[5] CERVANTES, Humberto a Rick KAZMAN, 2016. *Designing Software Architectures: A Practical Approach*. Boston: Addison-Wesley. ISBN 0134390784.

[6] FELLOWS, Geoff, 1998. High-Performance Client/Server: A Guide to Building and Managing Robust Distributed Systems. *Internet Research*. **8**(5). ISSN 1066-2243. Dostupné z: doi:10.1108/intr.1998.17208eaf.007

[7] RICHARDS, Mark, 2015. *Software Architecture Patterns* [online]. United States of Americe: O'Reilly Media [cit. 2022-03-02]. ISBN 9781491924242. Dostupné z: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/

[8] FOWLER, Martin a James LEWIS. *Microservices: a definition of this new architectural term* [online]. 25. 3. 2014 [cit. 2022-03-02]. Dostupné z: https://martinfowler.com/articles/microservices.html

[9] ERL, Thomas, 2009. *SOA: servisně orientovaná architektura : kompletní průvodce*. Brno: Computer Press. Programování (Computer Press). ISBN 978-80-251-1886-3.

[10] CERNY, Tomas, Michael J. DONAHOO a Jiri PECHANEC, 2017. Disambiguation and Comparison of SOA, Microservices and Self-Contained

Systems. *Proceedings of the International Conference on Research in Adaptive and Convergent Systems* [online]. New York, NY, USA: ACM, 2017-09-20, 228-235 [cit. 2022-03-04]. ISBN 9781450350273. Dostupné z: doi:10.1145/3129676.3129682

[11] CURBERA, Francisco, Donald FERGUSON, Martin NALLY a Marcia L. STOCKTON, 2007. Toward a Programming Model for Service-Oriented Computing. *Service-Oriented Computing — ICSOC 2007* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, 33-47 [cit. 2022-03-04]. Lecture Notes in Computer Science. ISBN 978-3-540-74973-8. Dostupné z: doi:10.1007/11596141_4

[12] MICHELSON, Brenda. *Event-Driven Architecture Overview: Event-Driven SOA Is Just Part of the EDA Story* [online]. **2006**, 2 [cit. 2022-03-04]. Dostupné z: doi:10.1571/bda2-2-06cc

[13] DOBBELAERE, Philippe a Kyumars Sheykh ESMAILI, 2017. Kafka versus RabbitMQ. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* [online]. New York, NY, USA: ACM, 2017-06-08, 227-238 [cit. 2022-04-13]. ISBN 9781450350655. Dostupné z: doi:10.1145/3093742.3093908

[14] AYANOGLU, Emrah, [2015]. *Mastering RabbitMQ: master the art of developing message-based applications with RabbitMQ*. Mumbai: [PACKT Publishing] open source. ISBN 978-178-3981-526.

[15] SAX, Matthias J., 2018. Apache Kafka. *Encyclopedia of Big Data Technologies* [online]. Cham: Springer International Publishing, 2018-02-10, 1-8 [cit. 2022-04-13]. ISBN 978-3-319-63962-8. Dostupné z: doi:10.1007/978-3-319-63962-8_196-1

[16] NARKHEDE, Neha, Gwen SHAPIRA a Todd PALINO, 2017. *Kafka: the definitive guide : real-time data and stream processing at scale*. Sebastopol: O'Reilly. ISBN 978-1491936160.

[17] RabbitMQ: AMQP 0-9-1 Model explained. *RabbitMQ* [online]. [cit. 2022-04-13]. Dostupné z: https://www.rabbitmq.com/tutorials/amqp-concepts.html

[18] KJERRUMGAARD, David, 2014. *Apache Pulsar in action*. Manning. ISBN 978-1617296888.

[19] *Consul Documentation* [online]. [cit. 2022-04-14]. Dostupné z: https://www.consul.io/docs

[20] *ZooKeeper documentation* [online]. [cit. 2022-04-16]. Dostupné z: https://zookeeper.apache.org/doc/current/index.html

[21] *Apache Curator* [online]. [cit. 2022-04-16]. Dostupné z: https://curator.apache.org/

[22] *Kubernetes documentation: Service* [online]. [cit. 2022-04-16]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/

[23] *Netflix Eureka Github Wiki: Eureka at a glance* [online]. [cit. 2022-04-16]. Dostupné z: https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

[24] *Grafana Documentation: Grafana basics* [online]. [cit. 2022-04-16]. Dostupné z: https://grafana.com/docs/grafana/latest/basics/

[25] *Prometheus documentation: Overview* [online]. [cit. 2022-04-16]. Dostupné z: https://prometheus.io/docs/introduction/overview/

[26] NAQVI, Syeda Noor Zehra a Sofia YFANTIDOU, 2017. *Time Series Databases and InfluxDB*. Bruxelles. Dostupné také z: https://cs.ulb.ac.be/public/_media/teaching/influxdb_2017.pdf. Universite libre de Bruxelles. Vedoucí práce Dr. Esteban Zimanyi.

[27] CHHAJED, Saurabh, 2015. *Learning ELK Stack*. United Kingdom: Packt Publishing. ISBN 9781785886706.

[28] *The complete Guide to the ELK Stack* [online], 2020. [cit. 2022-04-16]. Dostupné z: https://logz.io/learn/complete-guide-elk-stack/

[29] *Docker documentation* [online]. [cit. 2022-04-17]. Dostupné z: https://docs.docker.com/get-started/overview/

[30] *Kubernetes documentation* [online]. [cit. 2022-04-17]. Dostupné z: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[31] *Spring Boot Documentation* [online]. [cit. 2022-04-18]. Dostupné z: https://spring.io/projects/spring-boot

[32] *Micronaut Documentation* [online]. [cit. 2022-04-18]. Dostupné z: https://docs.micronaut.io/latest/guide/

[33] *Quarkus website* [online]. [cit. 2022-04-18]. Dostupné z: https://quarkus.io/

[34] *PostgreSQL website* [online]. [cit. 2022-04-20]. Dostupné z: https://www.postgresql.org/

[35] KLEPPMANN, Martin, 2017. *Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems*. Boston: O'Reilly Media. ISBN 14-493-7332-1.

[36] *Redis Documentation* [online]. [cit. 2022-04-20]. Dostupné z: https://redis.io/docs

[37] *Helm documentation* [online]. [cit. 2022-05-01]. Dostupné z: https://helm.sh/docs/

[38] *Trunk Based Development* [online]. [cit. 2022-05-03]. Dostupné z: https://trunkbaseddevelopment.com

[39] *Microservices.io* [online]. [cit. 2022-05-04]. Dostupné z: https://microservices.io

[40] *The Manager Workers Pattern* [online], 2004. UVK - Universitaetsverlag Konstanz [cit. 2022-05-04]. Dostupné z: https://lya.fciencias.unam.mx/jloa/publicaciones/A4.final.pdf

[41] *Manager-Worker Communication Patterns* [online]. [cit. 2022-05-04]. Dostupné z: https://levelup.gitconnected.com/manager-worker-communication-patterns-c3580b9db5db

[42] *The Twelve-Factor App* [online]. [cit. 2022-05-07]. Dostupné z: https://12factor.net/

Příloha A

Lokální spuštění prototypu

Pro lokální spuštění budeme potřebovat obsah přílohy. Ten je dostupný také online v repozitáři `https://github.com/strazovan/diploma-thesis-poc`. Dále je potřeba lokální instalace Docker Desktop, na něm povolený Kubernetes cluster a nainstalovaný nástroj Helm.

Prvním krokem je vytvoření Docker image pro jednotlivé služby. To je provedeno pomocí následujících příkazů:

```
helm upgrade --install ingress-nginx ingress-nginx --repo
 https://kubernetes.github.io/ingress-nginx
cd deployment
kubectl apply -f local-storage-class.yaml
cd ..
cd agent/sidecar/agent-sidecar/
docker build -t agent-sidecar .
cd ../../ping/
docker build -t ping-agent .
cd ../../messagebox/
docker build -t system-messagebox .
cd ../agentcontroller/
docker build -t agent-controller .
cd ../configurationservice/
docker build -t configuration-service .
cd ../qa-9-spam/
docker build -t qa-9-spam .
```

Nyní máme připraveny Docker image pro všechny naše služby a pro testovací skript pro jeden ze scénářů. Další částí je samotné nasazení komponent, konfigurace clusteru a další nastavení. Pro nasazení podpůrných služeb a messagingu použijeme následující sekvenci příkazů ve složce `deployment`:

```
helm install -f prometheus-grafana/prometheus-values.yaml
 prometheus-grafana-test prometheus-community/kube-
prometheus-stack
helm install filebeat elk/filebeat/
helm install logstash elk/logstash/
```

```
helm install elasticsearch elk/elastic/
helm install kibana elk/kibana/
helm install monitored-rabbitmq rabbitmq/
kubectl apply -f rabbitmq/instance.yaml
kubectl create clusterrolebinding defaultbinding2 --
clusterrole=edit --serviceaccount=default:default
```

Následně je potřeba přihlásit se do RabbitMQ GUI a vytvořit nového uživatele pro služby. Pomocí příkazu

```
kubectl get secret rabbit-mq-test-cluster-default-user -
oyaml
```

získáme výchozí údaje pro přihlášení. Ty jsou zakódované v base64 a je potřeba provést jejich dekódování. Pomocí příkazu

```
kubectl port-forward service/rabbit-mq-test-cluster 5672
```

si vytvoříme prostup na GUI. Po přihlášení v sekci administrace vytvoříme uživatele se jménem `test-user` a heslem `test-password`. Přiřadíme mu virtualní host `/` a roli administrátora. Následně můžeme provést nasazení jednotlivých komponent systému následujícími příkazy:

```
kubectl apply -f agent-controller.yaml
kubectl apply -f configuration-service.yaml
kubectl apply -f messagebox.yaml
kubectl apply -f ping-agent.yaml
```