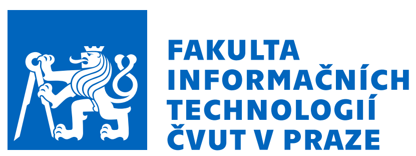


Zadání diplomové práce

Název:	TapiX - přepracování architektury pro dosažení vysokého výkonu
Student:	Bc. Beňadik Štrba
Vedoucí:	Mgr. Jan Gargulák
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

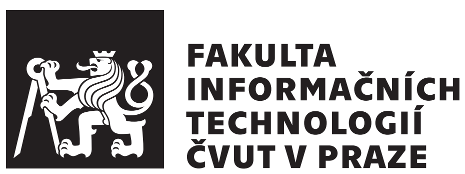
TapiX je služba, ktorá poskytuje API pre obohacovanie platobných dát. Ďalšie informácie na https://tapix.io.

Služba beží v prostredí Amazon Web Services. Pri počiatočnom návrhu služby sa nepočítalo s tak veľkým počtom requestov, ktoré musí dnes toto API obslúžiť. Kvôli tomu zažívame nečakané spomalenia až výpadky.

Je potrebné prepracovať architektúru riešenia, aby služba zvládala niekoľko násobný počet requestov, ktorý obsluhuje dnes, bez výrazného spomalenia služby.

- Zoznámte sa so súčasnou architektúrou služby TapiX
- Analyzujte a otestujte súčasné riešenie
- Identifikujte najslabšie miesta
- Analyzujte nájdené slabiny a vyhodnotte ich vplyv na výkonnosť, popíšte zložitosť, výhody a riziká ich opravy
- Identifikujte najkritickejšie slabiny a pre každú navrhnite a implementujte riešenie
  - Otestujte každú implementáciu a diskutujte jej pozitívne vlastnosti na výkon
  - Diskutujte náklady vzniknuté nasadením riešenia na AWS

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 28. března 2023 v Praze.



Diplomová práca

TapiX - prepracovanie architektúry pre dosiahnutie vysokého výkonu

Bc. Beňadik Štrba

Katedra webového inžinierstva
Vedúci práce: Mgr. Jan Gargulák

8. januára 2024

Poďakovanie

Chcel by som poďakovať predovšetkým vedúcemu práce Mgr. Janovi Gargulákovi za ochotu, odborné rady a čas, ktorý mi venoval pri vedení tejto diplomovej práce. Ďalej chcem poďakovať aj mojej rodine a priateľom za podporu.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. Ďalej prehlasujem, že som s ČVUT uzavrel dohodu, na základe ktorej sa ČVUT vzdalo práva na uzavrenie licenčnej zmluvy o používaní tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona. Táto skutočnosť nemá vplyv na ust. § 47b zákona č. 111/1998 Sb. o vysokých školách.

V Prahe 8. januára 2024 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2024 Beňadik Štrba. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu

Štrba, Beňadik. *TapiX - prepracovanie architektúry pre dosiahnutie vysokého výkonu.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Abstrakt

Práca sa zaoberá prepracovaním softwarovej architektúry existujúcej služby TapiX, ktorá poskytuje API na obohacovanie platobných transakcií o validné dáta. Prepracovanie architektúry je potrebné k zvýšeniu výkonu služby a k možnosti jednoducho službu škálovať. Na základe detailnej analýzy boli identifikované slabiny a nedostatky súčasného riešenia. Pri návrhu novej architektúry boli reflektované najkritickejšie slabiny. Nová architektúra zahŕňa použitie vhodnejšieho databázového systému a vytvorenie samostatnej mikroslužby pre asynchrónne spracovanie časti požiadavku.

Táto architektúra bola nasadená v AWS a testovanie potvrdilo jednoduchú možnosť škálovania služby. Záťažové testovanie ukázalo, že prepracovaná architektúra zvláda obslúžiť niekoľko krát viac požiadaviek s rýchlejšou odozvou, než tá pôvodná. Nová architektúra sa používa už na produkčnom prostredí, a benefity, ktoré priniesla, odpovedajú výsledkom testovania.

Kľúčové slová API, škálovateľnosť, softwarová architektúra, AWS, kartové transakcie, MongoDB

Abstract

The thesis focuses on the reengineering of the software architecture of the existing TapiX service, which provides an API for enriching payment transactions with valid data. The architectural reengineering is essential to enhance the performance of the service and facilitate its scalable deployment. Through a detailed analysis, weaknesses and deficiencies in the current solution were identified. The design of the new architecture takes into account the most critical weaknesses identified during the analysis. The new architecture includes the adoption of a more suitable database system and the creation of a standalone microservice for the asynchronous processing of a portion of the request.

This architecture was deployed on AWS, and testing confirmed that scaling of the service can be achieved with minimal effort. Performance testing demonstrated that the new architecture can handle several times more requests with faster response times, than the original one. The new architecture is already in use in the production environment, and the benefits it has brought align with the results of the testing.

Keywords API, scalability, software architecture, AWS, card transactions, MongoDB

Obsah

Úvod **1**

1 Predstavenie služby **3**

1.1 Vymedzenie základných pojmov . . . 3
1.1.1 Platobné kartové transakcie . . . 3
1.1.2 HTTP a HTTPS . . . 4
1.1.3 API . . . 4
1.1.4 REST API . . . 4
1.1.5 OAuth 2.0 . . . 5
1.2 Čo je to TapiX? . . . 5
1.3 TapiX API . . . 6
1.3.1 Shops . . . 6
1.3.2 Merchants . . . 8
1.3.3 Invalidations . . . 8

2 Analýza **11**

2.1 Technológie súčasnej architektúry . . . 11
2.1.1 Spring Boot . . . 11
2.1.2 Amazon Aurora . . . 12
2.1.3 AWS Elastic Beanstalk . . . 12
2.1.4 Amazon ElastiCache for Redis . . . 12
2.1.5 Amazon Kinesis Data Streams . . . 12
2.1.6 AWS Lambda . . . 13
2.1.7 Amazon S3 . . . 13
2.1.8 Amazon CloudWatch . . . 13
2.2 Súčasná architektúra . . . 13
2.2.1 Interné instancie . . . 14
2.3 Analýza kľúčovej funkcionality . . . 14
2.3.1 Vyhľadávanie shopov k transakciám . . . 14
2.3.2 Graf pravidiel . . . 16
2.3.3 Virtuálne shopy . . . 16
2.3.4 Aktualizácia dát . . . 17
2.3.4.1 Prvá fáza aktualizácie . . . 17
2.3.4.2 Druhá fáza aktualizácie . . . 18

2.4 Analýza slabín 18
2.4.1 Vyhľadávanie v pravidlách 18
2.4.2 Správa virtuálnych shopov 20
2.4.3 Správa unsolved požiadaviek 21
2.4.4 Získavanie invalidácií na produkčnej databáze 21
2.4.5 Historické invalidácie 22
2.4.6 Potenciálne slabiny 23
2.4.6.1 Wildcard pravidlá 23
2.4.6.2 Log požiadaviek 23
2.4.7 Nedokonalosť návrhu pôvodného riešenia 23
2.5 Najkritickejšie slabiny 23
2.6 Analýza riešení kritických slabín 24
2.6.1 Vyhľadávanie v pravidlách 24
2.6.2 Správa virtuálnych shopov a invalidácie 29
2.7 Analýza požiadaviek 30
2.7.1 Funkčné požiadavky 30
2.7.2 Nefunkčné požiadavky 31
2.8 Prípady použitia 32

3 Návrh **35**
3.1 MongoDB Sharded Cluster 35
3.1.1 Návrh kolekcie pravidiel 37
3.1.1.1 Výber shard kľúča 37
3.1.2 Vyhľadávanie v pravidlách 39
3.1.3 Infraštruktúra clustra 40
3.1.4 Integrácia MongoDB do aplikácie 41
3.1.5 Aktualizácia kolekcie dát v MongoDB 43
3.2 Mikroslužba pre správu virtuálnych shopov 44
3.2.1 Konzumácia virtuálnych shopov 44
3.2.2 Invalidácia virtuálnych shopov 46
3.2.2.1 Čiastočná invalidácia virtuálnych shopov . . . 47
3.2.2.2 Návrh API 49
3.2.3 Databázový model 49

4 Implementácia **53**
4.1 Vyhľadávanie pravidiel v MongoDB 53
4.2 MongoDB Sharded Cluster 55
4.2.1 Vytvorenie Kubernetes clustra v EKS 55
4.2.2 Vytvorenie MongoDB Sharded Clustra 55
4.3 Odolnosť a škálovateľnosť riešenia 56
4.4 TapixVsUpdater mikroslužba 57
4.4.1 Správa virtuálnych shopov 57
4.4.2 Proces invalidácie 60
4.4.3 Nasadenie mikroslužby a databázy v EKS 62

5 Testovanie **65**
5.1 Testovanie vyhľadávania pravidiel v MongoDB 65
5.2 Testovanie výkonnosti MongoDB Sharded Clustra 65
5.3 Testovanie funkčnosti mikroslužby 71

5.4 Testovanie výkonu aplikácie so streamovaným spracovaním virtuálnych shopov 71
5.5 Zhodnotenie výsledkov 74

6 Vzniknuté náklady **77**
6.1 Odhad nákladov 77

7 Možné vylepšenia **79**

Záver **81**

Bibliografia **83**

A Zoznam použitých skratiek **87**

B Slovník pojmov **89**

C Obsah príloh **91**

Zoznam obrázkov

1.1 Diagram použitia TapiX API 9

2.1 TapiX vysoko-úrovňová architektúra 15
2.2 Diagram aktualizácie dát 19
2.3 Aurora Performance Insights, záťaž writer instancie 21
2.4 Odozva API počas aktualizácie 22

3.1 Obecná architektúra MongoDB sharded clustra [32] 36
3.2 MongoDB Sharded Cluster v EKS 42
3.3 Diagram tried pre komponentu vyhľadávania v pravidlách 43
3.4 Diagram aktivít procesu vytvorenia a spracovania virtuálneho shopu 45
3.5 Vysoko úrovňová architektúra správy virtuálnych shopov 46
3.6 Sekvenčný diagram aktualizácie dát 48
3.7 Databázový model pre schému public 50
3.8 Databázový model pre schému importovaných dát 51

4.1 Vymazávanie platných položiek z Redis cache pre nedostatok miesta 62

5.1 Porovnanie priemernej doby odpovede aplikácie, pri záťaži 1 instancie skriptu 67
5.2 Porovnanie priemernej doby odpovede databázy, pri záťaži 1 instancie skriptu 67
5.3 Porovnanie počtu spracovaných požiadaviek, pri záťaži 1 instancie skriptu 68
5.4 Porovnanie priemernej doby odpovede aplikácie, pri záťaži 3 instancií skriptu 68
5.5 Porovnanie priemernej doby odpovede databázy, pri záťaži 3 instancií skriptu 69
5.6 Porovnanie počtu spracovaných požiadaviek, pri záťaži 3 instancií skriptu 69
5.7 Porovnanie priemernej doby odpovede aplikácie, pri záťaži 5 instancií skriptu 70
5.8 Porovnanie priemernej doby odpovede databázy, pri záťaži 5 instancií skriptu 70

5.9 Porovnanie počtu spracovaných požiadaviek, pri záťaži 5 instancií skriptu 71
5.10 Zaťaženie databázy počas testovania pred a po nasadení nového riešenia 73
5.11 Porovnanie priemernej doby odpovede aplikácie, pri záťaži 5 instancií skriptu 74
5.12 Porovnanie počtu spracovaných požiadaviek, pri záťaži 5 instancií skriptu 74

Zoznam tabuliek

2.1 Tabuľka pravidiel 14
2.2 Pokrytie funkčných požiadaviek 33

3.1 Tabuľka podielu null hodnôt pre polia pravidiel 37

Zoznam výpisov kódu

2.1 SQL vyhľadávanie v pravidlách 16
3.1 MongoDB vyhľadávanie v pravidlách 39
4.1 Metóda v triede BaseMongoDao pre získanie zoznamu výsledkov 53
4.2 Metódy v TxRuleMongoDao pre vytvorenie dotazu na získanie pravidiel 54
4.3 Statefulset konfigurácia pre jeden shard 58
4.4 Funkcia pre konzumáciu dát z Kinesis 60
4.5 Funkcia pre konzumáciu dát z Kinesis 61
5.1 Trieda poskytujúca PostgreSQL kontajner pre testy 72

Úvod

S nástupom digitálnej éry získavajú bezhotovostné platby, najmä prostredníctvom platobných kariet, stále väčšiu obľubu. Dáta o platobných transakciách sú potom dôležité z hľadiska užívateľov, napríklad v internetovom bankovníctve, ale aj z hľadiska ďalšej analýzy, napríklad pre získanie hlbších znalostí o klientoch. Platobné dáta neboli navrhnuté na ďalšie spracovanie a analýzu. Preto vznikla služba TapiX, ktorá poskytuje API pre obohatenie týchto platobných dát o konkrétne informácie týkajúce sa obchodníka, obchodu, či kategórie platby.

S rastúcim počtom klientov služby rastú aj nároky na jej výkon. Súčasná architektúra služby nepočítala s tak vysokými požiadavkami na výkon, preto sa táto diplomová práca zameriava na čiastočné prepracovanie architektúry služby. Nová architektúra bude reflektovať nedostatky súčasného riešenia. Jej súčasťou bude odstránenie najkritickejších slabín, schopnosť služby jednoducho škálovať a reagovať na rastúce požiadavky na výkon.

Zapálenie pre optimalizáciu aplikácií a zvedavosť z výsledkov testovania výkonu pri použití nových technológií a postupov bolo hlavnou motiváciou pri výbere tejto témy. Zároveň sa podľa môjho názoru jedná o prospešnú službu s veľkým potenciálom.

Práca je rozdelená do siedmych kapitol. Prvá kapitola predstavuje samotnú službu a problém, ktorý daná služba rieši. Zároveň vysvetľuje niektoré pojmy a termíny používané ďalej v práci. Druhá kapitola sa zaoberá analýzou súčasnej architektúry služby, popisom používaných technológií a detailnejším vysvetlením ako služba funguje. Súčasťou je aj analýza slabín aplikácie a ich riešenie. Kapitola končí analýzou funkčných a nefunkčných požiadaviek. Tretia kapitola popisuje návrh novej architektúry, ktorá bude riešiť nájdené kritické slabiny a nedostatky. Štvrtá kapitola popisuje proces implementácie a nasadenia upravenej aplikácie. Piata kapitola sa zameriava na testovanie služby z hľadiska funkčnosti aj výkonu a diskutuje dosiahnuté výsledky. V šiestej kapitole sú odhadnuté náklady za novo vzniknutú infraštruktúru vrámci AWS. Posledná kapitola popisuje možné rozšírenia a automatizácie týkajúce sa novej architektúry.

Ciele práce

Cieľom tejto práce je vytvoriť návrh a implementáciu novej architektúry služby TapiX, ktorá odstráni najkritickejšie nedostatky a bude zvládať rastúci počet požiadaviek na službu. Zároveň bude jednoducho škálovateľná.

K dosiahnutiu tohto cieľa bude nutná analýza súčastného návrhu aplikácie a identifikácia najkritickejších slabín. Na základe nájdených slabín sa popíše analýza možných riešení pre najkritickejšie miesta a táto analýza bude vstupom pre návrh novej architektúry. Ďalším bodom bude vytvoriť softwarový návrh služby s dôrazom na škálovateľnosť a odstránenie nájdených nedostatkov, podľa ktorého bude implementovaná prepracovaná architektúra. Nasledovať bude nasadenie a testovanie pôvodného, ako aj prepracovaného riešenia a porovnanie ich výkonu. V neposlednom rade si táto práca nesie za cieľ vyhodnotenie dosiahnutých výsledkov testovania a pretože bude novo vzniknutá infraštruktúra aplikácie spustená v AWS, tak bude popísaný aj odhad vzniknutých nákladov u tohto cloud poskytovateľa.

Kapitola 1

Predstavenie služby

Táto kapitola si nesie za cieľ predstavenie služby TapiX a domény okolo ktorej je táto služba postavená. Ďalej tu zaznie definícia základných konceptov, spolu s motiváciou k tejto práci.

1.1 Vymedzenie základných pojmov

1.1.1 Platobné kartové transakcie

Kartová transakcia je štandardizovaný platobný proces, iniciovaní držiteľmi platobných kariet, prostredníctvom čítačiek kariet na bankomatoch, termináloch na miestach predaja (*point of sale terminals*) a online platobných bránach. Medzinárodný štandard pre komunikáciu pri platobných transakciách za použitia platobnej karty je ISO 8583 [1].

Transakčné dáta obsahujú informácie odvodené z platobnej karty, platobného terminálu, samotnej transakcie a ďalšie potrebné meta informácie. ISO 8583 správa je tvorená z [1]:

- **MTI** (*Message type indicator*) – je štvorciferné číselné pole a označuje postupne podľa cifier ISO 8583 verziu, účel správy, spôsob „prúdenia“ správy v systéme a pôvod správy
- **bitovej mapy** – naznačuje, či sú ďalšie dátové prvky prítomné v správe, je tvorená z primárnej a voliteľnej sekundárnej bitovej mapy
- **dátových prvkov** – predstavujú samostatné polia, slúžiace pre informácie z transakcie, prítomné sú polia so stanoveným významom ako aj všeobecné polia, systémovo závislé polia, či polia závislé na krajine

Počet dátových prvkov sa pohybuje v súvislosti s ISO verziou v rozmedzí od 128 do 192. Každý prvok má svoju pozíciu [2, 3]. Z pohľadu služby TapiX sú najvýznamnejšie dátové prvky nachádzajúce sa v poli na pozícií:

- 43 *card acceptor name/location (1–23 street address, 24–36 city, 37–38 state, 39–40 country)* – zložený prvok, ktorý sa skladá z nasledujúcich častí:
  - 1–23 *street address* (`merchantDescription`) – v tejto časti sa väčšinou nachádza meno alebo adresa obchodníka

  - 24–36 *city* (`city`) – mesto
  - 37–38 *state* (`state`) – štát
  - 39–40 *country* (`country`) – krajina

- 42 *card acceptor identification code* (`merchantId`) – identifikuje obchodníka
- 41 *card acceptor terminal identification* (`posId`) – taktiež aj *point of sale id*, je unikátny kód identifikujúci platobný terminál u obchodníka
- 18 *merchant category code* (`mcc`) – slúži na klasifikáciu podniku podľa druhu tovarov alebo služieb, ktoré poskytuje

1.1.2 HTTP a HTTPS

Hypertext Transfer Protocol [4] je bezstavový protokol, slúžiaci pre výmenu dát na internete. Jedná sa o klient-server protokol, požiadavky sú posielané na server, server tieto požiadavky spracuje a posiela späť odpoveď na klienta. Súčasťou HTTP odpovede je aj stavový kód, ktorý naznačuje, či bol požiadavok úspešne dokončený alebo nie. Jedným z prvkov požiadavku je aj HTTP metóda, tá definuje operáciu, ktorú chce klient vykonať. Medzi základné HTTP metódy patrí GET, POST, PUT, DELETE.

HTTP nie je šifrovaný a pre zabezpečenú komunikáciu existuje šifrovaná verzia protokolu nazývaná HTTPS, ktorá používa TLS na šifrovanie celej komunikácie medzi klientom a serverom [5].

1.1.3 API

Klientské programy používajú *application programming interface* [6] na komunikáciu s webovými službami. API vystavuje množinu dát a funkcií na uľahčenie komunikácie medzi aplikáciami a umožňuje výmenu informácií medzi nimi. *Interface* môže byť chápaný ako kontrakt medzi dvoma aplikáciami komunikujúcich medzi sebou posielaním požiadaviek a odpovedí. Existujú rôzne typy API [7]

- SOAP API
- RPC API
- Websocket API
- REST API

Každý typ má svoje charakteristické vlastnosti.

1.1.4 REST API

Representational State Transfer je často aplikovaný štýl architektúry pri návrhu API moderných webových služieb. Webové API, ktoré dodržuje architektonický štýl REST sa nazýva REST API. REST používa stavebné bloky protokolu HTTP a definuje množinu zdrojov identifikovaných unikátnymi URI. Môže byť implementovaný ľubovoľnou sieťovou technológiou, no štandardne sa používa

HTTP. Pre prístup k definovaným zdrojom sa používajú HTTP metódy. Centrálnym princípom štýlu REST je koncept HATEOAS. Namiesto definovania zoznamu vecí, ktoré môže klient robiť v statickom dokumente, vyžaduje od klienta, aby dynamicky objavoval funkcionalitu počas používania API. To je zaistené tým, že v odpovediach od serveru sa nachádzajú ďalšie URI, ktoré môže klient použiť [8].

1.1.5 OAuth 2.0

Protokol pre autorizáciu. Umožňuje užívateľom udeliť obmedzený prístup ku chráneným prostriedkom. Klient požiada o prístup k prostriedkom, ktoré riadi vlastník prostriedkov a hostuje server prostriedkov. Tento server vydáva tokeny, ktoré schváli vlastník prostriedkov a klient ich používa pre prístup k chráneným prostriedkom vlastníka. OAuth 2.0 priamo súvisí s *OpenID Connect (OIDC)*. *OIDC* je vrstva zabezpečujúca overenie a autorizáciu, založená na OAuth 2.0 [9].

1.2 Čo je to TapiX?

TapiX je služba ktorá ponúka REST API pre obohatenie platobných dát v reálnom čase. Na vstupe služba dostane nespracované dáta, ktorých popis sa nachádza v prvej časti tejto kapitoly 1.1.1. Výstupom je obohatená platobná transakcia, poskytujúca konkrétne informácie o obchodníkovi a obchode, v ktorom bola transakcia vykonaná. Obohatenie zahŕňa napríklad presný názov obchodníka, logo obchodníka, miesto a GPS súradnice nákupu, kategóriu zakúpeného tovaru, uhlíkovú stopu transakcie a mnoho ďalších informácií o platbe [10].

Prečo je obohacovanie platobných dát potrebné?

Platobné dáta sú typicky v nespracovanej a neštrukturalizovanej forme s nepresnými hodnotami o obchodníkovi, či obchode. Kvalite týchto platobných dát neprospieva ani fakt, že jednotlivé dátové prvky platobnej transakcie nemusia byť ani vyplnené. Ďalší faktor znižujúci kvalitu takýchto dát je, že tieto dáta neboli navrhnuté na ďalšie spracovanie a analýzu. Svedčí tomu napríklad dátový prvok MCC 1.1.1, ktorý slúži na klasifikáciu podniku, ale pri porovnaní so sofistikovanejším spôsobom kategorizácie je iba 63% šanca, že transakcia bude kategorizovaná správne, ak sa pre kategorizáciu použije iba MCC. Je to spôsobené tým, že niektoré MCC zachytávajú iné aspekty transakcie, než je potrebné zachytiť pri kategorizácií. Napríklad pod MCC *Miscellaneous & Specialty Retail Stores* spadajú kategórie ako cestovanie, šport, či móda [11, 12].

Najväčší prínos obohacovania transakcií spočíva v analýze dát po obohatení, zvýšenej personalizácií a segmentácií klientov. Ďalšie využitie spočíva v zlepšení užívateľského zážitku klientov, napr. v internetovom bankovníctve. To vďaka nahradeniu nespracovaných a nepresných platobných dát, za obohatené a vyčistené dáta, ako je napríklad meno a logo obchodníka.

1.3 TapiX API

Ako už bolo spomínané v predchádzajúcej časti, TapiX ponúka REST API pre získanie detailných informácií o kartových transakciách alebo bankových prevodoch.

API je prístupné iba cez HTTPS. Všetky API endpointy sú zabezpečené a pre prístup k nim je doporučené používať OAuth 2.0.

Terminológia

Definície základných pojmov [13], ktoré budú ďalej v texte používané v súvislosti s API.

- **Merchant** Obchodník, majiteľ jedného alebo viacerých obchodov. Napríklad Tesco, Shell, Amazon.
- **Shop** Obchod, pre obchodníka s viacerými obchodmi tento termín označuje konkrétny obchod, kde bol uskutočnený nákup.
- **Card transaction** Kartová transakcia, je prevod peňazí, ktorý sa deje pri presune peňazí pomocou debetnej alebo kreditnej karty.
- **Bank transfer** Bankový prevod, je prevod peňazí, ktorý sa deje pri prevádzaní peňazí medzi bankovými účtami. Rôzne krajiny ako napríklad Veľká Británia, Rakúsko a Česká republika používajú rôzne štandardy bankových prevodov.
- **Handle** Identifikátor, slúži na rozdelenie transakcií alebo prevodov, ktoré sa vzťahujú na jeden obchod do menších skupín. Každá skupina je identifikovaná pomocou vlastného handle. Využíva sa najmä pre invalidáciu aktualizovaných dát.

TapiX ponúka niekoľko endpointov pre obohacovanie kartových transakcií a platobných prevodov (v rôznych štandardoch SEPA, BACS, CERTIS), či tzv. open dát. Táto práca sa zaoberá časťami služby, súvisiacimi s kartovými transakciami, pretože majorita požiadaviek prichádzajúcich na službu sú práve kartové transakcie. Z toho dôvodu nie sú predstavené všetky, ale iba relevantné endpointy.

1.3.1 Shops

GET `/shops/findByCardTransaction`

Slúži na obohatenie jednej transakcie. Najdôležitejšie parametre pre tento endpoint sú popísané v časti 1.1.1. Pre získanie správneho výsledku nie sú vždy nutné všetky parametre, ale čím viac relevantných parametrov je poskytnutých, tým presnejšie výsledky služba vracia. Parametre sú predané vrámci URL [14]. Príklad ako by mohol vyzerať požiadavok a odpoveď pre tento endpoint:

URL požiadavku s parametrami

```
/v5/shops/findByCardTransaction?
          posId=820014&
          merchantId=180520001&
          description=TESCO%20PRAHA%20ZLICIN%20PRAHA%20CZ&
          city=PRAHA&
          country=CZ
```

Kladná odpoveď, v prípade, že TapiX rozpozná transakciu

```
{
    "result": "found",
    "handle": "MmPdedgnvjXiRZnBJzJQKb",
    "shop": {
        "uid": "MmPdedgnvjXiRZnBJzJQKb"
    }
}
```

Záporná odpoveď, v prípade, že TapiX nerozpozná transakciu

```
{
    "result": "unsolved",
    "handle": "!wtecEsQvfZjwGIfOJLVcDd"
}
```

- `result` udáva výsledok vyhľadávania shopu pre transakciu:
  - `found` – shop bol nájdený
  - `unsolved` – shop nebol nájdený, ale môže byť doplnený v najbližšej dobe
- `handle` slúži ako identifikátor pre spojenie transakcie s výsledkom volania služby. Toto spojenie je potrebné, pretože dáta s ktorými služba pracuje sa medzičasom vylepšujú a teda `unsolved` transakcie môžu byť vyriešené a chybne priradené shopy transakciám opravené. Takéto transakcie je potom potrebné identifikovať práve na základe `handle` a invalidovať.
- `shop.uid` je unikátny identifikátor shopu, ktorý je potrebný pre získanie detailných informácií o shope. Taktiež sa používa aj pre invalidovanie shopov, ktoré boli aktualizované.

Diagram použitia TapiX API je na obrázku 1.1.

POST `shops/findByCardTransactionBatch`

Slúži na obohatenie niekoľkých transakcií poslaných v jedinom požiadavku. Poskytuje rovnakú funkcionalitu ako `/shops/findByCardTransaction`, líši sa len posielaním transakcií v tele požiadavku a v JSON formáte. Tento endpoint je doporučený pre produkčné prostredia, ktoré potrebujú obohacovať veľké množstvo transakcií, pretože znižuje zaťaženie infraštruktúry.

GET `shops/{uid}`

Vráti všetky dostupné informácie týkajúce sa shopu so zadaným `uid`. Súčasťou odpovede je aj merchant `uid`, ktorým je možné získať detailné informácie o merchantovi. Rovnako ako `handle` pri `/shops/findByCardTransaction` endpointe `uid` shopu je potrebné aj pre identifikáciu shopu, ktorý bol aktualizovaný a je potreba ho invalidovať.

1.3.2 Merchants

GET `merchants/{uid}`

Vráti všetky dostupné informácie týkajúce sa merchanta so zadaným `uid`. `Uid` sa opäť využíva pre invalidovanie aktualizovaných merchantov.

1.3.3 Invalidations

Invalidácie predstavujú mechanizmus ako udržiavať získané dáta aktuálne. Užívateľ služby si ukladá ku prevolaným transakciám `handle` a k získaným shopom a merchantom ich `uid`. Tieto dáta sa môžu časom aktualizovať. Typickým príkladom je pridanie adresy, gps súradníc, či ďalších informácií ku shopom, ktoré predtým týmito informáciami nedisponovali. Ďalším príkladom môže byť zmena priradenia shopu k transakcií, či vyriešenie `unsolved` transakcie. Invalidácie poskytujú práve také `handles` a `uids`, ktoré boli nejakým spôsobom aktualizované. Invalidácie je možné vnímať v rôznych leveloch, ktoré predstavujú hĺbku zmeny [15]. Tieto levely sú:

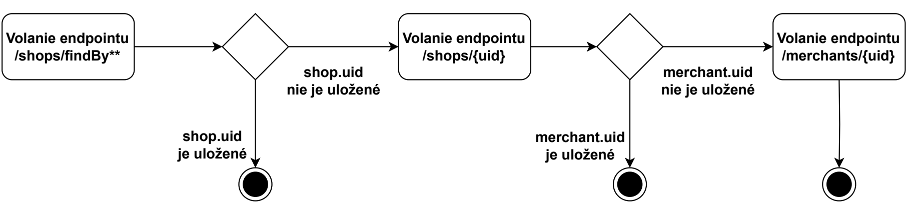
- `shallow` – vyjadruje, že aktualizácia dát iba pridala alebo zmenila informácie k existujúcemu shopu, či merchantovi
- `deep` – vyjadruje, že je potrebné premapovať transakciu k inému shopu, teda zmena `shop.uid` a `handle` pre transakciu
- `solved` – vyjadruje, že `unsolved` shop bol vyriešený

GET `/invalidations/item/range`

Endpoint slúži pre získanie rozsahu identifikátorov invalidovaných objektov v určenom časovom období pomocou parametrov typu *timestamp* s názvami `from` a `to`. Tento endpoint by nemal byť volaný pravidelne, ale len pre začiatočnú inicializáciu a získanie počiatočného `fromId`, ktoré bude použité v endpopinte `/invalidations`.

GET `/invalidations`

Na základe parametra `fromId`, získaného v `/invalidations/item/range`, vracia stránkovaný zoznam invalidovaných objektov a `lastItemId` pre pokračovanie a získanie všetkých invalidácií.



Obr. 1.1: Diagram použitia TapiX API

Kapitola **2**

Analýza

Obsahom tejto kapitoly je analýza súčasnej architektúry služby TapiX. Prvým krokom bude predstavenie technológií, ktoré služba aktuálne používa. Ďalej bude popísaná architektúra a budú identifikované najslabšie miesta z hľadiska škálovateľnosti a výkonu. Z identifikovaných slabín sa vyberú najkritickejšie, na základe ktorých bude rozobratá analýza ich riešenia. A nakoniec budú definované funkčné a nefunkčné požiadavky na prepracovanú architektúru a nové komponenty.

2.1 Technológie súčasnej architektúry

Služba TapiX a všetky komponenty, ktoré sú jej súčasťou bežia v prostredí cloudového poskytovateľa Amazon. Celá aplikácia je napísaná v jazyku Java a postavená na frameworku Spring Boot.

2.1.1 Spring Boot

Jedná sa o open-source framework pre tvorbu mikroslužieb a webových aplikácií v jazyku java. Spring Boot si nesie za cieľ čo najviac zjednodušiť vývoj v Spring frameworku, ktorého Spring Boot je nadstavbou. Snaží sa o minimalizáciu generického *boilerplate* kódu a tým pomáha vývojárom sa sústrediť na podstatnú logiku aplikácie [16].

Spring Boot prináša 4 zásadné vylepšenia:

1. *Automatic configuration* – zabezpečuje automatickú konfiguráciu pre podporu istých funkcionalít ako je napríklad prístup k databáze

2. *Starter dependencies* – zabezpečuje pridanie správnych knižníc a závislostí do projektu, bez potreby riešenia konkrétnych verzií a konfliktov

3. *The command-line interface* – umožňuje vývojárom písať čisto aplikačný kód, bez potreby tradičného zostavenia projektu

4. *The Actuator* – poskytuje prehľad o tom, čo sa deje v bežiacej Spring Boot aplikácií

2.1.2 Amazon Aurora

Aurora je plne spravovaný relačný databázový systém, kompatibilný s MySQL a PostgreSQL. Všetok kód, nástroje a aplikácie, ktoré sa používajú s MySQL a PostgreSQL môžu byť použité aj s Aurorou.

Táto služba vytvára databázový cluster, ktorý sa skladá z jednej alebo viacerých databázových instancií a cluster úložiska, ktoré spravuje dáta pre tieto instancie. Toto úložisko je rozložené do viacerých *Availability Zones* a každá z nich obsahuje kópiu dát. Vďaka tomu tento databázový systém zabezpečuje vysokú dostupnosť a jednoduchšiu škálovateľnosť, oproti napríklad obyčajnej PostgreSQL databáze. Cluster je tvorený dvoma typmi instancií:

- primárna – podporuje *read* a *write* operácie a vykonáva všetky modifikácie na úložisku
- replika – podporuje iba *read* operácie, je pripojená k rovnakému úložisku ako primárna instancia

Každý cluster disponuje jednou primárnou instanciou a maximálne 15 replika instanciami.

Aurora je súčasťou spravovanej databázovej služby *Amazon Relational Database Service (RDS)*. RDS je služba, ktorá uľahčuje nastavenie, spravovanie a škálovanie relačnej databázy v cloude [17].

2.1.3 AWS Elastic Beanstalk

Služba, ktorá ponúka rýchle nasadenie a správu aplikácie v cloude bez potreby učiť sa o infraštruktúre na ktorej aplikácia pobeží. Inými slovami znižuje komplexnosť správy aplikácie v cloude. Automaticky zabezpečuje alokáciu potrebných zdrojov, vyvažovanie záťaže, škálovanie a monitorovanie zdravia aplikácie. Po vytvorení prostredia pre aplikáciu, AWS vytvorí dodatočné zdroje pre jej chod ako sú napríklad: *load balancer* alebo *auto scaling group*, ktorá automaticky navyšuje alebo znižuje počet EC2 instancií, podľa záťaže aplikácie a jednu alebo viacero EC2 instancií. Každé prostredie disponuje CNAME (URL) ukazujúcu na *load balancer* [18].

2.1.4 Amazon ElastiCache for Redis

Redis je open source, *in-memory* úložisko dát, používaný, okrem iného, ako databáza, cache, streamovací systém. Poskytuje dátové štruktúry ako napríklad reťazce, listy, množiny, zoradené množiny. Má zabudovanú replikáciu pre podporu vysokej dostupnosti a núdzového prepnutia, rôzne úrovne perzistencie dát na disku, horizontálne škálovanie, či dokonca aj transakcie [19].

ElastiCache je služba poskytujúca jednoduché nastavenie, správu a škálovanie distribuovaného pamäťového úložiska v cloude. Základným stavebným blokom je cluster, ten predstavuje kolekciu jedného alebo viacerých cache uzlov, pričom na každom beží instancia Redis softwaru [20].

2.1.5 Amazon Kinesis Data Streams

Služba, ktorá zbiera a spracováva veľké streamy dát v reálnom čase. Stream dáta sú dáta, ktoré sú neustále generované viacerými zdrojmi a typicky sú po-

sielané v menších veľkostiach súčasne. Typický scenár použitia je, že producenti posielajú dáta priamo do streamu, ktoré potom konzument spracováva. Stream zabraňuje strate dát v prípade, že na strane konzumenta nastane chyba a stane sa nedostupným. Kinesis odbremeňuje užívateľov od potreby vytvárať a spravovať dáta *pipeline*. Ďalej ešte poskytuje dynamické škálovanie v závislosti na veľkosti prietoku dát [21].

2.1.6 AWS Lambda

Predstavuje výpočetnú službu, ktorá umožňuje spúšťať kód bez potreby zaisťovania a spravovania serverov. Lambda spúšťa kód na výpočetnej infraštruktúre s vysokou dostupnosťou a zaisťuje celú administráciu výpočetných zdrojov ako je napríklad údržba serverov, operačných systémov, zaisťovanie výpočetnej kapacity, či automatické škálovanie [22].

2.1.7 Amazon S3

Služba poskytujúca objektové úložisko garantujúce škálovateľnosť, dostupnosť dát, zabezpečenie a výkon. K dispozícií je niekoľko typov úložiska, pre rôzne účely zákazníkov [23].

2.1.8 Amazon CloudWatch

CloudWatch monitoruje v reálnom čase všetky služby a aplikácie, ktoré užívateľ spravuje na AWS. Zbiera logy a rôzne metriky aplikácií a služieb, vďaka ktorým užívateľ získa lepší pohľad na výkon aplikácie, vyťaženie infraštruktúry, či jednoduchšiu detekciu slabých a kritických miest. CloudWatch zbiera od väčšiny služieb základné metriky, ako je napríklad vyťaženie CPU u EC2, ale je možné monitorovať aj vlastné metriky. Všetky metriky je možné vynášať na grafy a vytvárať nad nimi alarmy, ktoré môžu posielať upozornenie, napríklad kvôli preťaženej službe, či spustiť automatické škálovanie [24].

2.2 Súčasná architektúra

Súčasná architektúra služby TapiX je znázornená na diagrame 2.1. Celá architektúra sa nachádza v súkromnej sieti a nie je teda dostupná z internetu. Okrem znázornených komponent sa v infraštruktúre vyskytuje ešte WAF (*web application firewall*), ktorý poskytuje bezpečný prístup z internetu k službe, a open source aplikácia Keycloack, ktorý zabezpečuje OAuth 2.0 autorizáciu.

Aplikácia TapiX beží v AWS službe ElasticBeanstalk, tá prináša automatické škálovanie, teda pridávanie ďalších instancií pri veľkom zaťažení a aplikačný load balancer ALB, ktorý distribuuje jednotlivé požiadavky medzi instancie. Ďalšou výhodou je bezvýpadkové nasadenie novej verzie aplikácie. To funguje tak, že nová verzia sa nasadí najskôr na jednu instanciu na ktorú ALB prestane posielať požiadavky a až keď je aplikácia spustená na tejto instancií a je zdravá, tak začne aktualizácia na ďalšej instancií.

Ďalšou a najviac zaťaženou časťou služby je databáza. Použitá je PostgreSQL databáza, konkrétne AWS služba Aurora pre PostgreSQL. Tá riadi automatické škálovanie a pridávanie *read* replík, ktoré sú vyhradené pre určité náročné SQL dotazy. Aurora zároveň poskytuje aj režim vysokej dostupnosti

databázy, tým že pri výpadku hlavnej *writer* intancie, je niektorá z *read* replík povýšená na novú *writer* instanciu.

Pre zníženie záťaže na Aurore je použitá Redis databáza, ktorá slúži ako cache pre výsledky hlavného vyhľadávacieho mechanizmu, ktorý vyhľadáva shopy k jednotlivým transakciám.

CloudWatch ukladá logy aplikácie a informácie o každom prichádzajúcom požiadavku pre ďalšiu analýzu a vylepšovanie služby. TapiX ukladá do súboru pre každý prichádzajúci požiadavok jeho parametre, odpoveď, čas za ktorý bol spracovaný a ďalšie potrebné informácie. Tento súbor je v reálnom čase streamovaný do služby CloudWatch. Odtiaľ sa následne tieto dáta streamujú do Kinesis. Dáta sú z Kinesis archivované do služby S3 pomocou služby Firehose, ktorá je v tomto prípade určená iba na prenos dát z Kinesis do S3. S3 zároveň slúži aj ako úložisko lóg merchantov, ktoré TapiX vracia ako jednu z niekoľkých informácií o merchantovi. Ďalšou komponentou je Lambda funkcia, ktorá je volaná vždy pri príchode nových dát do Kinesis. Kvôli optimalizácií sú dáta do Lambda funkcie posielané batchovo a táto funkcia následne vkladá tieto dáta do databázovej tabuľky pre ďalšie spracovanie. Pokiaľ by nastal výpadok interného DWH, kde sú dáta finálne uložené, alebo by nastala chyba v Lambda funkcii, tak sú dáta dostupné v Kinesis po dobu 7 dní a po oprave chyby sa pokračuje s batchom dát, ktoré sa nepodarilo uložiť. Kinesis tak zabraňuje strate dát, ku ktorým by mohlo dochádzať keby sa dáta posielali zo služby CloudWatch priamo do Lambda funkcie.

2.2.1 Interné instancie

Mimo infraštruktúru AWS sa ešte spravuje niekoľko interných TapiX instancií, ktoré sa používajú rovnako na obohacovanie platobných transakcií. Tieto instancie sa využívajú jednoúčelovo a nie sú na ne kladené vysoké nároky na výkon. Práve preto majú obmedzenú funkcionalitu oproti instancií bežiacej v AWS.

2.3 Analýza kľúčovej funkcionality

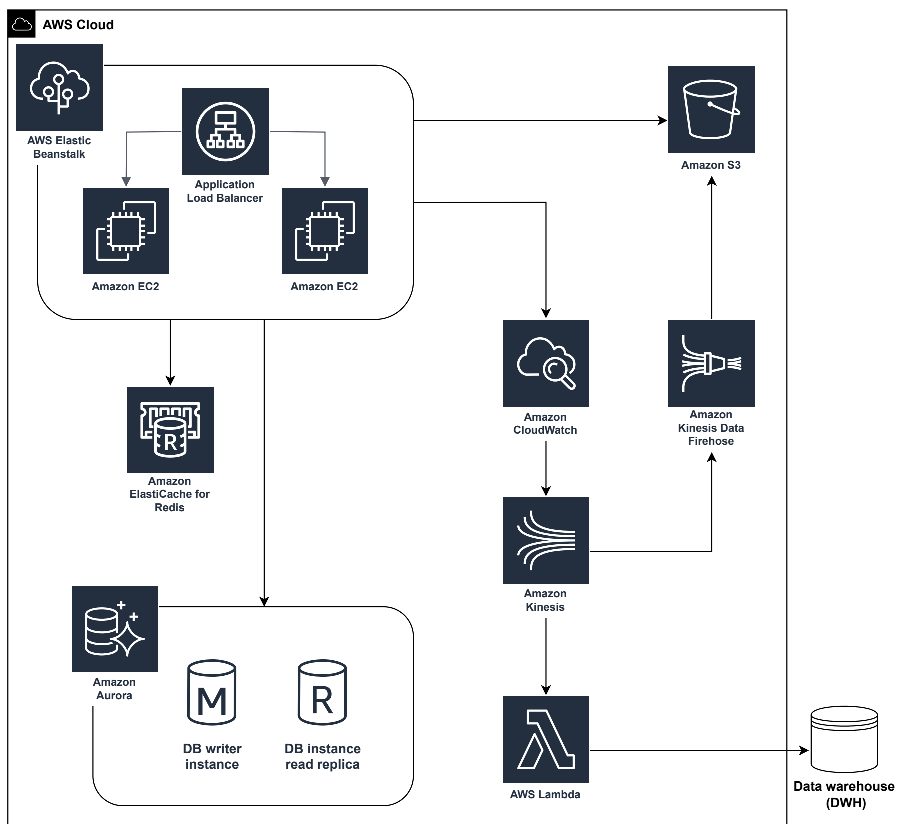
2.3.1 Vyhľadávanie shopov k transakciám

Najdôležitejšou funkcionalitou služby je vyhľadávanie shopov na základe vstupných parametrov transakcie. Na vyhľadávanie shopov ku transakciám slúži tabuľka pravidiel. Príklad ako také pravidlá môžu vyzerať je vidieť v tabuľke 2.1.

merchantId	**posId**	**merchantDescription**	**priority**	**shopId**
123456789	null	null	1	1
987654321	00110022	TESCO PRAHA	1	2
null	null	TST COFFEE*	4	3

Tabuľka 2.1: Tabuľka pravidiel

Pravidlá v tabuľke majú niekoľko stĺpcov, tie najdôležitejšie pomocou ktorých sa vyhľadáva sú popísané v 1.1.1 a k tomu sa používajú ešte 2 ďalšie `merchantDescriptionWild` a `zip`. Pre jednoduchosť majú neuvedené zvyšné stĺpce v tabuľke 2.1 hodnotu null. Každé pravidlo má svoju prioritu a dokopy



Obr. 2.1: TapiX vysoko-úrovňová architektúra

tvoria hierarchickú štruktúru, v ktorej sa vyhľadáva. Ku každému pravidlu je asociovaný výsledný shop. K jednému shopu môže existovať viacero pravidiel. Proces vyhľadávania má niekoľko krokov.

1. Vyhľadávanie prebieha v databázovej tabuľke pravidiel a iba v pravidlách, ktorých hodnota stĺpca `merchantDescriptionWild` je null a to takým spôsobom, že vstupné hodnoty požiadavku sú porovnávané s hodnotami v pravidlách. Pokiaľ je vstupná hodnota napríklad `merchantId` vyplnená, tak požiadavku môže odpovedať pravidlo, ktoré má v stĺpci `merchantId` rovnakú alebo null hodnotu. Naopak ak vstupná hodnota nie je vyplnená, požiadavku môže odpovedať iba pravidlo, ktorého hodnota v odpovedajúcom stĺpci je null. SQL dotaz pre nasledujúce vstupné hodnoty `merchnatId=123456789, posId=0000, country=CZ` by vyzeral ako v ukážke 2.1.

2. Vyhľadávanie prebieha v pamäti. Konkrétne v optimalizovanej, stromovej, dátovej štruktúre, ktorá sa inicializuje pravidlami s vyplneným stĺpcom `merchantDescriptionWild` pri spustení aplikácie. Hľadá sa teda vo wildcardových pravidlách podobne ako pri regulárnych výrazoch.

3. Vyberie sa výsledné pravidlo s najnižšou prioritou.

```
select id from tapix_rules
where merchant_description_wild is null and
(merchant_id = '123456789' or merchant_id is null) and
(pos_id = '0000' or pos_id is null) and
merchant_description is null and
(country = 'CZ' or country is null) and
mcc is null and
city is null and
zip is null
```

Výpis kódu 2.1: SQL vyhľadávanie v pravidlách

Ak by sa nenašlo žiadne pravidlo alebo nastal konflikt v prípade, že sa nájde viacero rôznych pravidiel s rovnakou prioritou ale rôznym shopom, tak je výsledok `unsolved` shop.

2.3.2 Graf pravidiel

Niektoré pravidlá, ktoré majú rôznu prioritu, sú vo vzťahu a tvoria orientovaný, acyklický graf. Tento graf existuje kvôli potrebám v procese invalidácií. V prípade že sa pridá nové pravidlo R2, ktoré rieši rovnaké transakcie, ako už existujúce pravidlo R1, ktoré má ale vyššiu prioritu, vytvorí sa medzi nimi hrana R1 $\rightarrow$ R2. R1 sa stane rodičovským resp. obecnejším pravidlom, konkrétnejšieho pravidla R2. Tento vzťah je potom potrebný pri invalidáciach, pretože v takomto prípade je nutné invalidovať všetky transakcie, ktoré boli vyriešené pravidlom R1. Informácia, že je nutné invalidovať všetky transakcie vyriešené pravidlom R1 je dostupná iba z grafu pravidiel.

2.3.3 Virtuálne shopy

Virtuálne shopy predstavujú ďalší dôležitý koncept. Riešia problém obrovského počtu transakcií, ktoré sa v určitých prípadoch musia invalidovať. Podľa množstva poskytnutých informácií o transakcií, vracia TapiX shopy, ktoré môžu byť konkrétnejšie a disponujúce adresou, GPS súradnicami a ďalšími informáciami. Pokiaľ nie je dostatok informácií k identifikácií konkrétneho shopu, tak v takom prípade môže TapiX vrátiť viac obecné shopy. Obecný shop je taký shop, ktorý identifikuje iba merchanta ako napríklad Tesco a už neposkytuje žiadne konkrétnejšie informácie. Na takýto obecný shop sa odkazuje veľké množstvo transakcií, pre ktoré sa nenašiel konkrétnejší shop alebo bolo poskytnutých málo vstupných informácií. Niektoré obecné shopy tvoria takzvané **big shopy**. Každý big shop má definovanú sadu relevantných stĺpcov pre daný shop z pravidiel, podľa ktorých sa transakcie agregujú do virtuálnych shopov. Pre každú transakciu, ktorá podľa nejakého pravidla vedie na big shop sa nájde alebo vytvorí virtuálny shop a `uid` tohoto virtuálneho shopu sa vráti ako `handle` v odpovedi na požiadavok. Virtuálne shopy sú teda automaticky vytvorené shopy, ktoré zoskupujú transakcie odkazujúce na rovnaký big shop do skupín podľa definovanej sady stĺpcov. Pre lepšie pochopenie slúži nasledujúca ukážka.

```
//Existujúce pravidlo, neuvedené polia sú null
    id=1 | merchantId=MID | shopId=10 | priority=10
//Big shop
    id=10 | tx_vs_base='merchant_id,pos_id'
//Transakcie na vstupu
    merchantId=MID | posId=PID1 | merchantDescription=DESC
    merchantId=MID | posId=PID1 | merchantDescription=DESC
    merchantId=MID | posId=PID2 | merchantDescription=DESC
//Vytvorené virtuálne shopy, neuvedené polia okrem uid sú null
    id=1 | merchantId=MID | posId=PID1 | shopId=10
    id=2 | merchantId=MID | posId=PID2 | shopId=10
```

Ako je možné vidieť, `tx_vs_base` u big shopu definuje polia, podľa ktorých sa agregujú transakcie do virtuálnych shopov a na hodnotách ostatných polí nezáleží. Virtuálne shopy znižujú počet invalidácií, pretože pri vzniku, zmene alebo zániku pravidla, ktoré odkazuje na big shop by sa museli invalidovať všetky transakcie, ktorým bol big shop priradený. Vďaka virtuálnym shopom sa nemusí invalidovať celý big shop, ale len jeho podmnožina a teda virtuálne shopy, ktoré sú ovplyvnené vytvoreným, zmeneným alebo zaniknutým pravidlom. Sada relevantných stĺpcov pre big shop je definovaná na základe pravidiel, ktoré vznikajú. Vďaka tomu je možné pri zmene takého pravidla zacieliť iba relevantné virtuálne shopy a invalidovať tak iba časť transakcií, ukazujúcich na big shop.

Virtuálne shopy a pravidlá zdieľajú istú množinu rovnakých polí, to sú napríklad: `merchant_id`, `pos_id`, `merchant_description` a ďalšie. Pri invalidácií sa virtuálne shopy vyhľadávajú na základe hodnôt v poliach pravidiel. Môže ale nastať situácia, kedy zmenené pravidlo, podľa ktorého by sa mali hľadať virtuálne shopy do invalidácie, má všetky polia zo sady relevantných polí big shopu null. V takom prípade nie je možné podľa tohoto pravidla dohľadať virtuálne shopy, ktoré by sa mali invalidovať. Preto sa k virtuálnym shopom ukladajú aj `id` pravidiel, ktoré k nim viedli, aby bola možná invalidácia aj týchto problémových pravidiel. Dvojice identifikátorov, teda `id` pravidla a `id` virtuálneho shopu sa ukladajú do samostatnej tabuľky a nazývajú sa linky virtuálnych shopov.

2.3.4 Aktualizácia dát

Vrámci TapiXu sa tvoria a ukladajú virtuálne shopy, invalidované záznamy, účty s prístupom k službe a ďalšie podobné tabuľky s dôležitými meta dátami a dátami potrebnými pre chod služby. Dáta, pomocou ktorých sa obohacujú transakcie, ako sú napríklad pravidlá, shopy, merchanti, kategórie a ďalšie, sa dostávajú do TapiXu formou pravidelných aktualizácií. Aktualizácia dát prebieha v dvoch fázach. Ale ešte predtým je dôležité zmieniť, že dáta, ktoré sa tvoria a sú spravované v TapiXe, sú v databázovej schéme *public* a dáta, ktoré sa do TapiXu dostávajú pri aktualizácií sú v schéme *data_timestamp*. *Timestamp* rastie pri každej aktualizácií a TapiX pri spustení použije vždy balíček dát zo schémy, ktorá má najvyšší *timestamp*. Celý proces aktualizácie je možné vidieť na diagrame 2.2.

2.3.4.1 Prvá fáza aktualizácie

1. Stiahne sa balíček dát určený k importu.

2. V databáze sa vytvorí nová schéma *update*.
3. V schéme *update* sa vytvoria všetky potrebné tabuľky.
4. Stiahnuté dáta sa naimportujú do tabuliek v schéme *update*.
5. Nad tabuľkami v schéme *update* sa vytvoria potrebné indexy.

2.3.4.2 Druhá fáza aktualizácie

V druhej fáze sa spustí aplikácia, ktorá prevedie invalidáciu dát. Začína sa `shallow` invalidáciou. Tá zahŕňa invalidáciu shopov a merchantov. Najskôr sa získa databázový rozdiel medzi dátami z aktuálnej schémy a novými dátami z *update* schémy. Tento rozdiel vzniká medzi relevantnými stĺpcami tabuliek určených k invalidácií. Databázový rozdiel sa robí obojsmerne pre získanie nových a aktualizovaných shopov, ale aj pre získanie tých odstránených. Z výsledkov rozdielu sa extrahujú `uid`, ktoré sú potom vložené do *invalidated* tabuľky. Z nej sa klientom vracajú invalidované záznamy. Rovnaký postup platí aj pre invalidáciu merchanta.

Pre `deep` invalidáciu sa zistí množina ovplyvnených pravidiel, zahŕňajúca nové, aktualizované a odstránené pravidlá. Táto množina pravidiel sa opäť získa pomocou databázového rozdielu tabuľky pravidiel a tabuľky grafu pravidiel. Do tejto množiny sú naviac pridané aj rodičovské pravidlá nových a aktualizovaných pravidiel. To je z dôvodu, že nové pravidlo, ktoré má rodičov, má nižšiu prioritu ako rodičovské pravidlá a teda časť transakcií, ktorá bola vyriešená rodičovskými pravidlami bude teraz vyriešená týmto novým pravidlom, ktoré napríklad odkazuje už na iný, konkrétnejší shop. Pomocou výsledných pravidiel sa nájdu všetky virtuálne shopy a ich `uid` sa objavia v množine invalidovaných `handles`. Pre problematické pravidlá, popísané v 2.3.3, pomocou ktorých nie je možné nájsť virtuálne shopy, sa virtuálne shopy hľadajú pomocou `id` pravidiel.

V poslednej časti invalidácií sa hľadajú vyriešené `unsolved` požiadavky. TapiX každý unikátny `unsolved` požiadavok uloží do tabuľky. Pomocou množiny ovplyvnených pravidiel získanej v predchádzajúcej časti sa získajú `unsolved` požiadavky a pre každý z nich sa potom pomocou nového balíčku dát pokúsi nájsť shop. Ak sa taký shop nájde, dostane sa `uid` požiadavku do `solved` invalidácií.

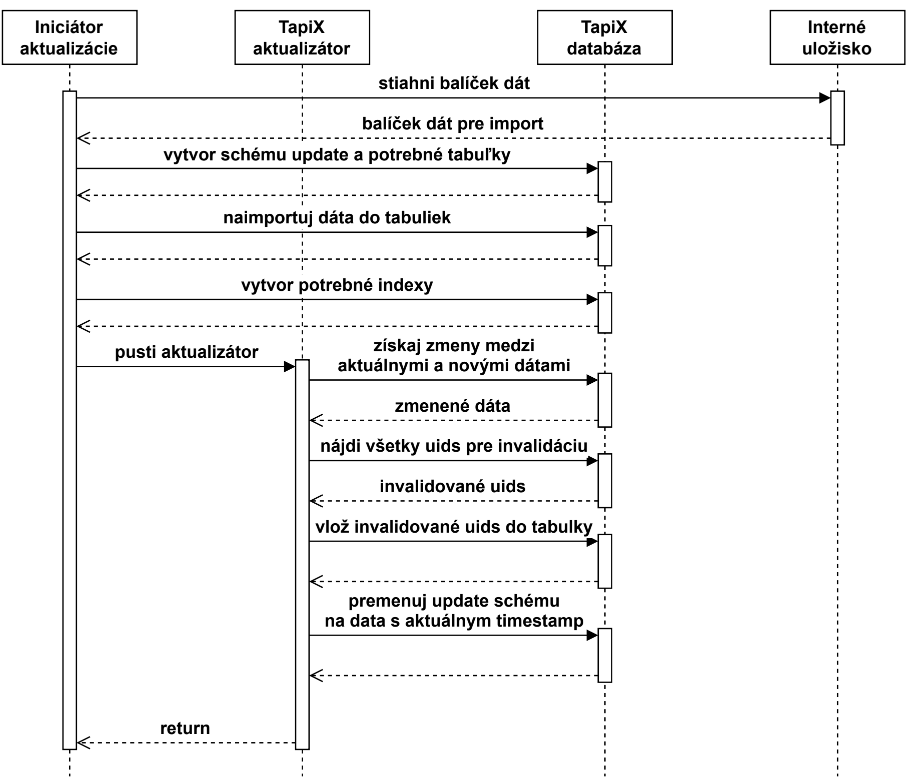
Po dokončení invalidácií všetkých dát sa schéma *update* s novými dátami premenuje na *data_timestamp* s aktuálnym *timestamp* a prevedie sa reštart aplikácie, aby začala používať najnovší dátový balíček. Zároveň sa reštartom invaliduje aj Redis cache, pretože kľúč pre prístup k dátam obsahuje ako prefix meno aktuálne používaného dátového balíčka, resp. databázovej schémy.

2.4 Analýza slabín

Identifikácia kritických a slabých miest aplikácie pozostávala najmä z analýzy zdrojového kódu a z analýzy monitoringu aplikácie.

2.4.1 Vyhľadávanie v pravidlách

Aktuálna implementácia vyhľadáva pravidlá v PostgreSQL databáze, konkrétne používa AWS službu Aurora. Tento spôsob je dostatočný pre 10 000 požiada-



Obr. 2.2: Diagram aktualizácie dát

viek za minútu, ktorý aplikácia bežne obsluhuje. V prípade, že sa ale počet požiadaviek za minútu niekoľko krát znásobí, služba sa výrazne spomalí, pretože databáza je úplne zaťažená.

Po diskusií s tímom, ktorý má na starosti vytváranie pravidiel je možné odhadovať že s pribúdajúcimi klientmi a rozširovaním služby do ďalších zemí, bude nielen počet požiadaviek na službu, ale aj počet pravidiel rýchlo rásť, odhadom na stovky miliónov pravidiel. Aurora síce podporuje pridávanie *read* replík, ktoré pomôžu rozložiť záťaž, ale ich počet je obmedzený. S rastúcim počtom pravidiel je toto riešenie nielen nákladné, ale zároveň nie je veľmi dobre škálovateľné a dlhodobo udržateľné.

Riešením tejto slabiny by bol napríklad výber vhodnejšieho databázového systému, pre vyhľadávanie v pravidlách, ktorý by umožňoval dlhodobé škálovanie pri rýchlom raste dát. Odstránenie tejto slabiny by prinieslo flexibilitu v prípade náhleho zvýšenia dát. Takéto škálovanie by nevyžadovalo žiadnu ďalšiu implementáciu na strane aplikácie. Jedná sa o najdôležitejšiu komponentu celej aplikácie, pretože sa používa pri každom požiadavku na obohatenie transakcie. Preto je ďalším dôležitým bodom pre takúto komponentu, aby bola schopná bežať v režime vysokej dostupnosti.

2.4.2 Správa virtuálnych shopov

Súčasná implementácia pracuje s virtuálnymi shopmi synchrónne. To znamená, že virtuálne shopy sa spravujú počas každého požiadavku, ktorý vyhľadáva shop k transakcií a výsledkom vyhľadávania je pravidlo, vedúce na big shop. V takom prípade sa počas každého požiadavku spraví niekoľko krokov, ktoré výrazne predlžujú dobu odpovede. Tieto kroky sú:

1. Aplikácia sa podľa vstupných parametrov požiadavku a relevantnej sady polí z big shopu pokúsi vyhľadať, či virtuálny shop už existuje. Ak existuje, vráti sa jeho `uid`.

2. Ak virtuálny shop neexistuje, vygeneruje sa nové `uid` a virtuálny shop sa vloží do tabuľky. Nad stĺpcom `uid` v tabuľke je vytvorený *unique* index, ak by teda `uid` už existovalo, vygeneruje sa nové a znova sa pokúsi vložiť virtuálny shop do databázy.

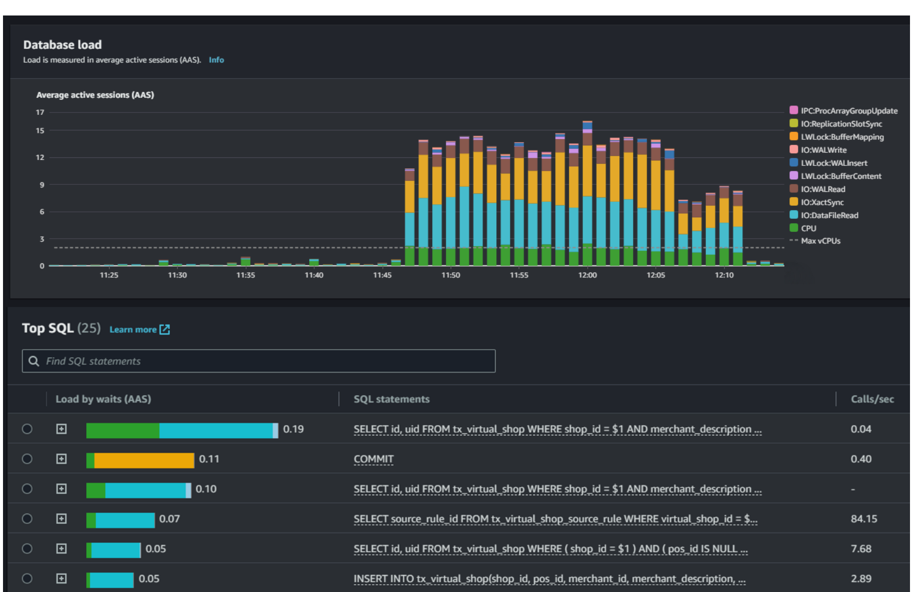
3. Vloží sa záznam `virtual_shop_id, rule_id` do tabuľky linkov, ktorý pre každý virtuálny shop uloží identifikátory pravidiel, ktoré na neho vedú. Tentop krok je dôležitý kvôli problematickým pravidlám 2.3.3 pri procese invalidácií.

Jedná sa o výraznú slabinu, pretože aplikácia by mohla zvládať aj vyšší počet požiadaviek, ale ak by tieto jednotlivé požiadavky obsahovali transakcie, ktoré by viedli na big shopy, tak by to spôsobilo preťaženie aplikácie, čo by malo dopad na všetkých klientov.

Pre príklad je uvedená reálna situácia, kedy klient začal prevolávať API takými transakciami, ktoré generovali príliš veľa virtuálnych shopov. To malo za následok výrazné spomalenie služby. *Performance Insights* je funkcia na ladenie a monitorovanie výkonu databázy. Na snímku obrazovky z AWS 2.3 je možné vidieť *Performance Insights* služby Aurora a konkrétne vizualizáciu záťaže writer instancie Aurora clustera. Na obrázku je vidieť niekoľko násobné zaťaženie databázovej instancie, práve počas tohoto intervalu klient extenzívne prevolával API. Prerušovaná čiara na grafe označuje maximálne vCPU, to znamená počet virtuálnych procesorov pre jednu instanciu databázy. Na jednom vCPU môže bežať v jeden časový okamih iba jeden proces. Ak je počet procesov väčší ako počet vCPU, procesy sa začnú ukladať do fronty, čo negatívne ovplyvňuje výkon [25]. Tyrkysová farba v grafe označuje čakanie na čítanie zo súboru, oranžová farba značí čakanie na zámok a zelená farba označuje vyťaženie CPU. Pod grafom sú vidieť SQL príkazy, ktoré najviac vyťažovali databázu v zobrazenom časovom intervale a všetky súvisia s hľadaním v tabuľke virtuálnych shopov, v tabuľke linkov alebo s vkladaním virtuálnych shopov do databázy.

Pre vyriešenie tejto slabiny, by bolo nutné implementovať odlišný mechanizmus agregácie transakcií odkazujúcich na big shopy, ktorý by nezaťažoval databázu. Ďalším spôsobom by mohlo byť asynchronné spracovanie virtuálnych shopov. Virtuálne shopy nie sú esenciálne pre správny chod API, sú potrebné až pri vytváraní invalidácií, preto nie je nutné ich spracovávať počas odpovede na požiadavok. Takéto riešenie by nebolo úplne triviálne implementovať.

Riziká spojené s vyriešením tejto slabiny by zahŕňali najmä chyby pri implementácií, ktoré by mohli spôsobiť stratu, resp. neuloženie niektorých virtuálnych shopov. Následkom toho by bolo, že pri aktualizácií dát, by zákazníci nemuseli dostať všetky aktualizované `uid` a používali by zastaralé dáta.



Obr. 2.3: Aurora Performance Insights, záťaž writer instancie

Najväčší prínos vo vyriešení tejto slabiny spočíva v navýšení výkonovej rezervy aplikácie. Súčasná implementácia nezvláda vyšší nápor požiadaviek, ktoré by boli špecifické tým, že by generovali virtuálne shopy. Po vyriešení tejto slabiny by výkon nebol obmedzovaný typom požiadaviek, ktoré služba spracováva.

2.4.3 Správa unsolved požiadaviek

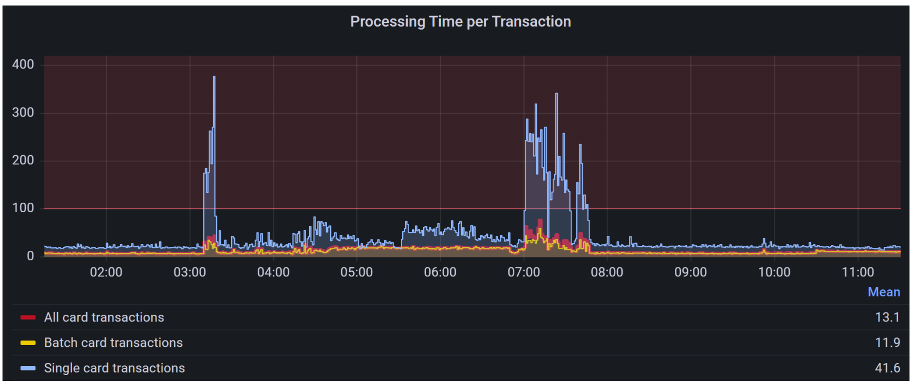
`Unsolved` požiadavky predstavujú podobnú slabinu ako virtuálne shopy. Pokiaľ sa pre transakciu v požiadavku nenájde žiaden shop, tak sa hľadá či už pre takúto transakciu neexistuje `unsolved` požiadavok. Ak existuje, vráti sa jeho `uid`, inak sa vytvorí nový `unsolved` požiadavok a vráti sa jeho `uid`. Táto slabina predstavuje jednoznačne menší problém, než virtuálne shopy, z toho dôvodu, že takýchto `unsolved` požiadaviek sa vytvorí niekoľko násobne menej, než virtuálnych shopov a teda nepredstavujú až takú záťaž pre databázu. Ďalším dôvodom je, že sa tieto požiadavky vytvárajú a vyhľadávajú len v jednej tabuľke narozdiel od virtuálnych shopov, ktoré vyhľadávajú a aktualizujú dáta v dvoch tabuľkách.

Riešenie tejto slabiny by bolo opäť asynchronné spracovávanie `unsolved` požiadaviek. Riziká a prínosy vyriešenia tejto slabiny sú rovnaké ako je popísané pri správe virtuálnych shopov. 2.4.2.

2.4.4 Získavanie invalidácií na produkčnej databáze

Súčasný proces aktualizácie dát služby vytvorí novú schému a tabuľky do ktorých naimportuje nové dáta. Po importe dát sa spustí *updater* aplikácia, ktorá začne vytvárať záznamy do *invalidated* tabuľky ako už bolo popísané v 2.3.4.2.

Problémom je vyhľadávanie invalidácií prebiehajúce nad produkčnou databázou aj keby nutne nemuselo. Získavaním invalidácií sa teda myslí proces, ktorý mimo iné úkony vyhľadáva v databáze. Tento proces výrazne zaťažuje databázu a má negatívny dopad na celkový výkon služby. Tento dopad je možné vidieť vo forme vyššieho času odpovede služby na grafe 2.4, ktorý sleduje priemernú dobu odpovede služby pre jeden požiadavok v milisekundách. O 3:00 sa spustila aktualizácia dát a skončila okolo 7:40. Riziko pri takomto spôsobe aktualizácií spočíva aj v tom, že spojenie vyššej záťaže generovanej zákazníkmi, spolu so záťažou vygenerovanou prebiehajúcou aktualizáciu, môže spôsobiť vysoké spomalenie až výpadok služby. Pritom počítanie invalidácií nie je vôbec nutné vykonávať na produkčnej databáze.



Obr. 2.4: Odozva API počas aktualizácie

Odstránenie tejto slabiny by vyžadovalo vytvorenie a správu novej mikroslužby, spolu s ďalšou databázou, kde by mohol prebiehať výpočet invalidácií. Odstránenie slabiny by znamenalo elimináciu rizika, spojeného s kombinovaním zvýšenej záťaže od zákazníkov a záťaže vyplývajúcej z aktuálnych aktualizácií. Takáto záťaž by mohla zapríčiniť vysoké spomalenie až výpadok služby. Vytvorenie novej mikroslužby, ktorá by sa špecializovala na výpočet invalidácií prináša väčšiu komplexitu do celej aplikácie a zvyšuje riziko chýb, ktoré môžu nastať pri komunikácií a výmene dát medzi službami.

2.4.5 Historické invalidácie

Súčasná implementácia ukladá všetky invalidácie do *invalidated* tabuľky. Tabuľka každou aktualizáciou rastie o milióny nových záznamov a žiadne sa z nej neodstraňujú. To spomaľuje endpointy popísané v sekcií 1.3.3, kvôli náročným SQL dotazom, ktoré pre tak obrovskú tabuľku zaťažujú databázu a trvajú niekedy aj niekoľko sekúnd. Pomalú odpoveď dostávajú iba niektorí klienti, ktorí neinvalidujú dáta pravidelne. Ich požiadavky potom na dlhý časový rozsah vytvárajú popisovanú záťaž.

Oprava tejto slabiny by bola veľmi náročná, pretože by si vyžadovala návrh nového spôsobu pracovania s invalidovanými záznamami. Možným riešením by bolo šikovné rozdelenie dát do viacero tabuliek, poprípade použitie vhodnejšej databázy na prácu s obrovskými dátami. Prínosom opravy tejto slabiny by

bolo spoľahlivejšie a rýchlejšie získanie invalidovaných záznamov zákazníkom. Rizikom je možné zrýchlenie rastu dát, pretože aktuálne je jeden invalidovaný záznam v tabuľke zdieľaný pre všetkých zákazníkov, to už pri nejakom konkrétnom rozdelení dát nemusí byť pravda.

2.4.6 Potenciálne slabiny

Slabiny, ktoré nepredstavujú problém v blízkej budúcnosti, ale predstavujú nedokonalosť súčasného riešenia, ktorá sa môže časom prejaviť.

2.4.6.1 Wildcard pravidlá

Wildcard pravidlá v ktorých sa vyhľadáva, sú celé uložené v pamäti RAM v stromovej štruktúre, optimalizovanej pre vyhľadávanie v takýchto pravidlách. Pri rastúcom počte wildcard pravidiel hrozí vyčerpanie celej pamäte RAM na jednotlivých EC2 instanciách, na ktorých aplikácia beží.

2.4.6.2 Log požiadaviek

Aktuálne sa pre každý prichádzajúci požiadavok na službu zapisujú základné informácie o tomto požiadavku do súboru. Tieto informácie sa dostanú až do Kinesis. Lambda funkcia potom tieto záznamy z Kinesis vyberá a vkladá ich do databáze. Potencionálna slabina pri tomto riešení je práve vkladanie výsledných dát do databázy. S pribúdajúcim počtom požiadaviek môže jednoducho naškálovať aj počet lambda funkcií, ktoré vkladajú dáta do databázy. Vkladanie je síce batchové no pri veľkom počte dát a lambda funkcií to už databáza nemusí výkonnostne zvládať.

2.4.7 Nedokonalosť návrhu pôvodného riešenia

Pri analýze aplikácie bola objavená nedokonalosť súvisiaca s aktualizáciou dát a invalidovaním virtuálnych shopov. Chyba nie je kritická, ale môže spôsobovať, že klienti nedostanú všetky invalidácie a budú používať zastaralé dáta.

Počas druhej fázy aktualizácie dát, teda po importe nových dát, prebieha získavanie invalidácií. Celý výpočet invalidácií prebieha v jednej transakcií a môže trvať v jedinečných prípadoch až niekoľko hodín. Počas aktualizácie služba používa stávajúci balíček dát, nový sa začne používať až po dokončení aktualizácie a reštarte aplikácie. Z toho dôvodu, ak by v priebehu aktualizácie vznikol virtuálny shop, ktorý by sa mal invalidovať vrámci prebiehajúcej aktualizácie, tak môže nastať situácia, že nebude invalidovaný, pretože fáza invalidovania virtuálnych shopov už mohla skončiť. To vedie k tomu, že daný virtuálny shop, ktorý mal byť invalidovaný vrámci prebiehajúcej aktualizácie, invalidovaný nebude a v nasledujúcich aktualizáciách nie je zaručené, že bude invalidovaný.

2.5 Najkritickejšie slabiny

Za najviac závažné a aktuálne boli vybrané tieto slabiny:

1. **Vyhľadávanie v pravidlách** Jedná sa o najdôležitejšiu komponentu aplikácie. Preťaženie tejto časti by malo dopad na väčšinu požiadaviek,

ktoré prichádzajú na službu, preto je kritické, aby práve táto komponenta aplikácie mohla byť rýchlo a jednoducho škálovateľná a vedela sa prispôsobiť buď narastajúcemu počtu požiadaviek na službu alebo veľkosti dát v novom balíčku pri aktualizácií.

2. **Správa virtuálnych shopov** Slabina spôsobuje nestabilitu aplikácie v závislosti na type dát, ktoré sú obsahom jednotlivých požiadaviek.

3. **Získavanie invalidácií na produkčnej databáze** Každá aktualizácia dát spôsobuje zbytočné spomalenie služby kvôli získavaniu invalidácií, ktoré sa deje na produkčnej databáze. V kombinácií s veľkým počtom požiadaviek prichádzajúcich na službu by mohlo dôjsť až k výpadku služby.

Tieto slabiny boli vybraté, pretože už teraz spôsobujú problémy službe. Odstránením práve týchto slabín sa docieli najväčší pozitívny vplyv na výkon.

2.6 Analýza riešení kritických slabín

V prvej časti tejto podkapitoly budú rozobraté možné riešenia vyhľadávania v pravidlách. Druhá časť sa zaoberá riešeniami ohľadom správy virtuálnych shopov. S virtuálnymi shopmi je spojené aj získavanie ich invalidácií na produkčnej databáze. Preto bude druhá a tretia slabina z 2.5 riešená spoločne.

2.6.1 Vyhľadávanie v pravidlách

Kritickou časťou je databázový *select*, ktorý vyhľadáva v tabuľke pravidiel. Jeho podoba je naznačená v 2.1. Nad touto tabuľkou je vytvorená množina rôznych databázových indexov. Databázový index je dátová štruktúra, slúžiaca pre zrýchlenie vyhľadávania v tabuľke. Prvotnou optimalizáciou by mohlo byť zefektívnenie týchto indexov. To znamená preskúmať pôvodné indexy a prípadne ich upraviť, aby sa používali čo najefektívnejšie. Za zmienku by stálo aj vyskúšať nahradiť hodnoty *null* v stĺpcoch nejakou hodnotou.

Za vyskúšanie by stál aj *table partitioning*, ten slúži na rozdelenie jednej veľkej tabuľky na menšie fyzické časti, tie sa dajú predstaviť ako osobitné tabuľky, ktoré ukladajú časť dát. *Partitioning* môže výrazne zvýšiť výkon dotazov, najmä keď sa väčšina často pristupovaných riadkov tabuľky nachádza v jednej alebo malom množstve partícií. Pri dotazoch alebo aktualizáciách, ktoré pristupujú k veľkej časti jednej partície, môže byť dotaz optimalizovaný použitím sekvenčného prehľadávania tejto partície namiesto použitia indexu, ktorý by vyžadoval náhodný prístup k dátam rozhádzaným po celej tabuľke [26].

Navrhnuté optimalizácie by mohli urýchliť súčasnú implementáciu a navýšiť aktuálny výkon služby, no tieto návrhy nepredstavujú ultimátne riešenie s jednoduchým a spoľahlivým škálovaním. Dlhodobo udržateľné riešenie by malo poskytovať horizontálne škálovanie a vysokú dostupnosť. Primárne sa nad tabuľkou pravidiel vykonávajú *read* operácie. Aplikácia dokonca nezapisuje do tabuľky pravidiel žiadne dáta, nové záznamy získava len prostredníctvom aktualizácií. Preto efektívne zapisovanie nemá pri výbere vhodného riešenia vysokú prioritu. Prioritou je hlavne efektívne čítanie dát.

Ďalším možným riešením tejto slabiny by mohla byť implementácia pamäťovej štruktúry stromového typu, ktorá by vyhľadávala v pravidlách v pamäti

RAM. To by ale znamenalo vysoké nároky na RAM pamäť jednotlivých inštancií a implementovaná by musela byť tiež distribúcia dát spolu s vysokou dostupnosťou. Preto ani vyhľadávanie v pamäti nepredstavuje najefektívnejšie riešenie a lepšie bude použiť existujúcu *NoSQL* databázu.

Kvôli výkonnostným nárokom kladených na výsledné riešenie bude databázový systém musieť podporovať distribúciu dát. Bude teda vhodné predstaviť s ňou súvisiacich niekoľko pojmov.

Sharding

Sharding je proces rozdeľovania databázy podľa dokumentov v dokumentovej databáze alebo podľa riadkov v relačnej databáze. Jednotlivé časti predstavujú shardy a tie sú uložené na samostatných serveroch. Pre navýšenie výkonu je možné škálovať vertikálne, no často to vyžaduje vyššie finančné prostriedky a čas. Pre sharding platí, že ďalšie servery môžu byť pridané do clustra na požiadanie a existujúce servery sú stále využívané. Pre implementáciu shardingu je dôležitý výber sharding kľúča a algoritmus rozdeľovania dát. Sharding kľúč je jeden alebo viacero polí v dokumentovej, či viacero stĺpcov v relačnej databáze, ktoré sa používajú pri zoskupovaní dokumentov do rôznych shardov. Algoritmus rozdeľovania dát používa tento kľúč ako vstup a určuje vhodný shard pre ich umiestnenie. Medzi základné algoritmy rozdelenia dát patrí [27]:

- range – zoskupuje susedné hodnoty a posiela ich na rovnaký shard, to je užitočné keď shard kľúč tvorí usporiadaná množina hodnôt, ako napríklad dátumy alebo čísla, v takom prípade by mohli byť dáta rozložené v shardoch napríklad po jednotlivých mesiacoch
- hash – používa hash funkciu, podľa ktorej sa rozdeľujú dáta rovnomerne do jednotlivých shardov

Replication

Replikácia je proces ukladania viacerých kópií rovnakých dát na samostatné servery. Takýto proces zabezpečuje vysokú dostupnosť a zvyšuje výkon pre *read* operácie. Medzi dva najrozšírenejšie modely replikácie patria [27]:

- **Master-slave** Master je server v clustri, ktorý prijíma požiadavky pre zápis aj čítanie. Je zodpovedný za replikáciu a kopírovanie aktualizovaných dát na ostatné servery v clustri. Ostatné servery spracovávajú požiadavky len pre čítanie dát. V prípade, že by nastala chyba na master serveri, ostatné slave servery iniciujú protokol na výber nového master servera spomedzi nich.
- **Masterless alebo Peer-to-peer** Prvý model nefunguje príliš dobre pri veľkom množstve požiadaviek pre zápis dát. Tento model dovoľuje všetkým serverom spracovávať požiadavky pre zápis aj čítanie dát. Servery v tomto modeli pracujú v skupinách. Teda keď príde požiadavok pre zápis na server, ten nedistribuuje svoje dáta všetkým ostatným serverom, ale len tým s ktorými je v skupine až sa dáta dostanú na všetky servery. Pri požiadavkách pre zápis je dôležité kvôli konzistencií, aby rovnaké požiadavky chodili na rovnaké servery. To zabezpečí výber vhodného kľúča pre distribúciu dát.

CAP teorém

Tvrdí, že v prípade distribuovaných databáz nie je možné dosiahnuť súčasne konzistenciu, dostupnosť a odolnosť voči prerušeniu [27, 28]:

- konzistenciu (*consistency*) – konzistentné kópie dát na jednotlivých serveroch, teda prevedenie všetkých operácií na serveroch musí prebiehať rovnako, akoby boli prevedené sekvenčne jeden po druhom na jednom serveri
- dostupnosť (*availability*) – každý úspešne prijatý požiadavok na funkčnom serveri musí vrátiť odpoveď
- odolnosť voči prerušeniu (*partition tolerance*) – pokiaľ nastane chyba v sieti, ktorá prepája servery, tak tie sú stále dostupné s konzistentnými dátami

Konzistencia a dostupnosť sú tradičné vlastnosti relačných databáz. Tie sa riadia vlastnosťami **ACID**:

- atomicita (*atomicity*)
- konzistencia (*consistency*)
- nezávislosť (*independence*)
- trvanlivosť (*durability*)

Dostupnosť a odolnosť voči prerušeniu sú vlastnosti, ktoré sú typické pre *NoSQL* databázy, riadia sa princípmi **BASE**.

- *basically available* – systém funguje aj pri výpadkoch niektorých častí
- *soft state* – dáta môžu byť nakoniec prepísané novšími dátami
- *eventual consistency* – na krátku dobu môže byť databáza v nekonzistentnom stave, no nakoniec bude systém v konzistentnom stave

NoSQL databázy

NoSQL je databázová technológia, ktorá ukladá dáta vo flexibilnej schéme a je jednoducho škálovateľná. Tento typ databáz bol vytvorený aby riešil obmedzenia relačných databázových systémov. Bežne je navrhnutý tak, aby využíval viacero serverov a preto je často používaný v distribuovanom prostredí. Existuje niekoľko typov distribuovaných *NoSQL* databázových systémov [27]:

- **Key-value** Využíva jednoduchý model umožňujúci ukladať a vyhľadávať hodnoty pomocou unikátneho kľúča.
- **Document** Podobné *key-value* databázam, ale s tým rozdielom, že hodnoty sú ukladané ako Dokumenty. Dokument je polo-štrukturovaná entita. Namiesto ukladania každého atribútu entity pod samostatný kľúč, dokumentové databázy ukladajú skupinu atribútov pod jeden kľúč. Výhodou je, že umožňujú získavať dokumenty nielen podľa kľúča, ale aj na základe hodnôt atribútov.

- **Column Family** Zdieľajú niekoľko pojmov s relačnými databázami ako je riadok a stĺpec. Stĺpec je základnou jednotkou úložiska a pozostáva z názvu a hodnoty, ktorá môže byť typu skalár, ale aj množina, zoznam, či mapa. Stĺpce je možné zoskupovať do kolekcií súvisiacich stĺpcov. Množina stĺpcov tvorí riadok a riadky môžu mať rovnaké alebo rozličné stĺpce. Nie je potrebná preddefinovaná fixná schéma. Nevýhodou je, že nepodporujú *join* tabuliek.
- **Graph** K uloženiu dát využívajú uzly a vzťahy medzi nimi alebo tiež vrcholy a hrany. Vrchol je objekt obsahujúci identifikátor a množinu atribútov. Hrana je spojenie medzi dvoma vrcholmi, ktoré obsahuje atribúty týkajúce sa vzťahu medzi dvoma vrcholmi.

Pretože celá infraštruktúra služby je v AWS, budú ďalej v texte predstavení kandidáti pre vhodnú *NoSQL* databázu najmä z portfólia AWS.

Amazon DynamoDB

Je plne spravovaná *key-value* databáza, ktorá zabezpečuje distribúciu dát, škálovanie aj replikáciu. Hlavnými komponentami tú tabuľky, položky a atribúty. Tabuľka je kolekcia položiek a položka je kolekcia atribútov. Používa unikátne primárne kľúče na identifikáciu položiek v tabuľke. Primárny kľúč je zložený buď z jedného atribútu a nazýva sa *partition key* alebo je zložený z dvoch atribútov, vtedy sa nazýva *partition key and sort key. Partition key* slúži ako vstup do hash funkcie na základe ktorej sa určuje partícia na ktorej bude položka uložená. *Sort key* je kľúč podľa ktorého budú položky zoradené na danej partícií. Pre získanie položiek podľa ďalších atribútov rôznych od primárneho kľúča, je možné vytvoriť nad tabuľkou indexy. Index je tvorený z *partition key* a *sort key*, teda maximálne dvoma atribútmi. Tieto indexy nie sú dostatočné pre vyhľadávanie v pravidlách a súčasný spôsob vyhľadávania, ako je naznačené v 2.1 nie je možné efektívne použiť nad touto databázou [29].

Amazon Neptune

Je plne spravovaná grafová databáza, ktorá uľahčuje prevádzku aplikácií pracujúcich s veľmi prepojenými dátami. Jadro databázy tvori vysoko výkonný systém, umožňujúci ukladať miliardy vzťahov a dotazovanie grafu s rýchlosťou odpovede v milisekundách. Neptune podporuje populárne jazyky na dotazovanie grafových databáz ako sú Gremlin, openCypher, SPARQL. Medzi kľúčové komponenty patrí primárna instancia, ktorá vykonáva všetky operácie týkajúce sa modifikácie dát, ďalej sú to repliky na čítanie dát, ktoré zvyšujú výkon a zabezpečujú vysokú dostupnosť a v neposlednom rade to je cluster úložisko, ktoré je tvorené kópiami dát v niekoľkých zónach, pre vysokú dostupnosť a spoľahlivosť. Pre využitie tejto služby by sa musel súčasný dátový model pravidiel previesť na graf a muselo by sa implementovať vyhľadávanie v takom grafe. Jedná sa hlavne o ďalšiu alternatívu, ktorá je ale aktuálne zbytočne náročná na implemnetáciu v porovnaní s ostatnými možnosťami [30].

Amazon Keyspaces (for Apache Cassandra)

Je škálovateľná, vysoko dostupná a spravovaná *column-family* databáza kompatibilná s Apache Cassandra. Architektúra takejto databázy používa *peer-to-*

peer model, v ktorom sú všetky uzly clustra rovnaké. Pri takejto architektúre neexistuje jediný bod zlyhania. Dáta sú distribuované a replikované medzi jednotlivé uzly. Škálovanie je veľmi priamočiare, jednoducho sa pridávajú alebo odoberajú ďalšie uzly. Pretože tu neexistuje jediný koordinujúci server, tak jednotlivé servery musia zabezpečovať:

- zdieľanie stavu uzla v clustri
- najaktuálnejšiu verziu dát na jednotlivých uzloch
- ukladanie dát určených pre zápis, keď je uzol, ktorý ich mal spracovať nedostupný

Cassandra pomocou *Gossip* protokolu zabezpečuje všetky tieto funkcie. Primárne kľúče slúžia aj ako *partition* kľúče pre distribúciu dát medzi uzlami. Cassandra podporuje primárne indexy, ktoré sú vytvorené nad identifikátorom riadka a sekundárne indexy vytvorené nad ľubovolným stĺpcom. Nie je tu však podpora zložených indexov nad viacerými stĺpcami [31].

MongoDB Sharded Cluster

MongoDB je dokumentová databáza. Záznam v MongoDB je dokument zložený z párov polí a hodnôt. Podporuje replikáciu dát pre zvýšenie výkonu a vysokú dostupnosť, ako aj horizontálne škálovanie pomocou sharding techniky. K distribúcií dát sa používa shard kľúč, ktorý je tvorený jedným alebo viacerými poliami dokumentov. Shard kľúč je možné zmeniť bez výpadku databázy. Shardované dáta sú ďalej delené na fragmenty, ktoré sú v prípade potreby migrované balancerom, medzi jednotlivé shardy pre čo najrovnomernejšie rozdelenie dát. Každý fragment disponuje dolným a horným rozsahom založenom na shard kľúči. V clustri môžu koexistovať shardované aj neshardované kolekcie. MongoDB podporuje sekundárne indexy a aj zložené sekundárne indexy nad viacerými poliami. Pre shard kľúč musí byť vždy vytvorený index. Ďalej databáza podporuje hash a range stratégie pre rozloženie dát medzi uzlami. Podporované sú dokonca aj transakcie [32].

Amazon DocumentDB (with MongoDB Compatibility)

MongoDB alternatíva v AWS je spoľahlivá, plne spravovaná dokumentová databáza kompatibilná s MongoDB. DocumentDB podporuje dva typy clustrov: *instance-based* cluster a *elastic* cluster. *Instance-based* cluster predstavuje databázu s jednou primárnou instanciou a podporou až pätnástich *read* replík. No zaujímavý je až druhý typ clustra, ktorý podporuje distribúciu dát a teda sharding. Ten je obdobou MongoDB Sharded Clustera. *Elastic* cluster zabezpečuje síce správu a jednoduché škálovanie clustra, no prichádza s veľkým počtom obmedzení. Medzi tie najzásadnejšie patria [33]:

- maximálny počet shardov je 32
- nie je možný sharding existujúcej kolekcie
- nie je možné zmeniť sharding kľúč
- nepodporuje sharding kľúč zložený z viacerých polí kolekcie

- nepodporuje veľké množstvo príkazov týkajúcich sa obecnej administrácie databázy, monitoringu správneho rozloženia dát v clustri, správy jednotlivých shardov a ich dát, donedávna nebol podporovaný dokonca ani *explain* príkaz na ladenie dotazov

Pre objektívnejšie zhodnotenie služby, bol cluster podrobený testovaniu. Testovaný bol bežný prípad použitia a teda import dát, vytvorenie indexov, dotazovanie nad dátami. Počas testovania sa narazilo na radu ďalších drobných obmedzení ako napríklad, že hodnoty shard kľúča nesmú obsahovať diakritiku. Pri vytváraní indexov a importe dát, cluster niekoľko krát odpovedal chybovou hláškou *MongoServerError: Internal server error.* Odpoveď clustra na jednotlivé dotazy bola veľmi pomalá, no kvôli absencií príkazu *explain* v čase testovania, nebolo možné zistiť z akej príčiny sú dotazy pomalé. Pridanie alebo odobranie shardu štandardne trvá jednotky minút, no v prípade *elastic* clustra operácia trvala jednotky hodín. Záver testovania je, že *elastic* cluster nie je vhodný pre takéto použitie s prihliadnutím aj na vysoké náklady, ktoré táto služba predstavuje.

Výsledné riešenie

Pri výbere vhodného databázového systému pripadali do úvahy MongoDB Sharded Cluster a Amazon Keyspaces (for Apache Cassandra). Sekundárne indexy v Cassandre nie sú súčasťou *partition* kľúča. To znamená, že Cassandra musí vyslať dotaz všetkým uzlom a čakať kým všetky odpovedia. Pri vytváraní sekundárnych indexov Cassandra vytvorí skrytú tabuľku v ktorej ukladá potrebné metadata indexu. Táto tabuľka nie je distribuovaná medzi všetky uzly, ale je uložená spolu so zdrojovými dátami na rovnakých uzloch. Takže pri dotazu, ktorý používa sekundárny index musí prečítať index metadáta z každého uzla a zozbierať výsledky. Pri veľkom počte uzlov to môže viesť ku zvýšenému prenosu dát a pomalšej odozve. Podobné obmedzenia má aj MongoDB Sharded Cluster. Dajú sa riešiť pridaním podmienky na *partition* kľúč do dotazu [34, 35]. Nakoniec bol vybratý MongoDB Sharded Cluster z niekoľkých dôvodov. MongoDB podporuje zmenu sharding kľúča bez výpadku služby, to v Cassandre nie je možné. Cassandra podporuje rozdeľovanie dát iba pomocou hash funkcie, čo môže byť v určitých prípadoch limitujúce. Pre architektúru Cassandri platí, že každý uzol je samostatný server a replikácia je zabezpečená ukladaním dát na niekoľko uzlov. Naproti tomu v MongoDB replikáciu dát zabezpečuje to, že každý shard je replikovaný medzi uzly s ktorými tvorí replica-set. Zlyhanie uzla v Cassandre spôsobí presun dát medzi uzlami v clustri s cieľom udržať nastavenie replikácie, čo môže mať negatívny dopad na výkon. Zlyhanie uzla v MongoDB nevyvoláva presun dát, pretože tie isté dáta sú uložené v ďalších replikách konkrétneho shardu. Cassandra teda obvykle poskytuje menej predvídateľný výkon než MongoDB počas zlyhania servera. To isté platí aj pri pridávaní a odoberaní uzla z clustra [36].

2.6.2 Správa virtuálnych shopov a invalidácie

Úzke hrdlo tejto slabiny je vyhľadávanie v tabuľke virtuálnych shopov počas spracovávania požiadavku. Spracovanie je ešte pomalšie pokiaľ sa virtuálny shop nenájde, v takom prípade sa počas požiadavku musí vytvoriť nový virtuálny shop a vložiť do tabuľky. Zároveň sa počas toho ešte vyhľadáva v tabuľke

linkov a vkladá do nej. Pri spracovaní požiadavku sa trávi zbytočne dlhú dobu pri správe virtuálnych shopov. Tie ale vôbec nie je nutné spracovávať počas odpovede na požiadavok. So správou virtuálnych shopov súvisí aj získavanie ich invalidovaných uids.

Prvoplánové riešenie by mohlo vyzerať tak, že namiesto správy virtuálnych shopov počas požiadavku by sa pre požiadavok vytvoril nový virtuálny shop spolu s linkami ak by nebol nájdený v cache. Následne by sa uložil do súboru a súčasne aj do cache pamäte. To by odstránilo celú réžiu súvisiacu s ich databázovým spracovaním. Tieto súbory by boli pravidelne spracovávané a celá správa virtuálnych shopov, teda vyhľadávanie v tabuľkách a vkladanie do nich by sa diala mimo spracovania požiadavku. Architektúra takéhoto riešenia by mohla pozostávať z AWS služieb S3, Elasticache a Lambda. Súbory s virtuálnymi shopmi by sa ukladali na S3 a Lambda funkcia by zabezpečovala ich vyhľadávanie a vkladanie do databáze. Elasticache by redukovala počet vzniknutých virtuálnych shopov. Takéto riešenie je jednoduché a dočasne by urýchlilo spracovanie požiadaviek. Na druhú stranu nie je dobre škálovateľné, pretože spracovanie virtuálnych shopov stále zaťažuje produkčnú databázu. Zároveň by sa virtuálne shopy stále invalidovali na produkčnej databáze, teda takéto riešenie nie je dlhodobo udržateľné.

Viac robustnejším a škálovateľnejším riešením je vytvorenie mikroslužby s vlastnou databázou, ktorá by správu virtuálnych shopov riešila kompletne na svojej strane. To znamená, že virtuálne shopy by spracovávala v osobitnej databáze. Invalidácie virtuálnych shopov by sa rovnako nepočítali už na produkčnej databáze, ale v osobitnej, ktorú by spravovala mikroslužba. Tá by súčasne riešila aj objavenú nedokonalosť v časti 2.4.7 a bola by dobre škálovateľná, flexibilná a teda by podporovala rozšírenie o ďalšiu funkcionalitu. To by zabezpečovalo aj rýchlejší vývoj. Práve takúto flexibilitu by bolo pri prvoplánovom návrhu náročné splniť. Nová mikroslužba je teda perspektívnejším riešením daného problému a bude riešiť druhú a zároveň aj časť tretej slabiny popísanej v 2.5.

2.7 Analýza požiadaviek

Táto sekcia popisuje požiadavky kladené na výsledné riešenie. Účel stanovenia požiadaviek je premeniť abstraktné *business* požiadavky na presnejší zoznam požiadaviek, ktorý nakoniec vedie k návrhu systému. Požiadavky môžu byť funkčné alebo nefunkčné. Funkčné požiadavky sa priamo vzťahujú k funkcionalite, ktorú systém musí vykonávať. Nefunkčné požiadavky sa vzťahujú k vlastnostiam, ktoré sú na systém kladené. Opisujú rôzne charakteristiky týkajúce sa systému ako sú napríklad rýchlosť, dostupnosť, spoľahlivosť. Nefunkčné požiadavky ovplyvňujú predovšetkým rozhodnutia, robené počas návrhu systému [37].

2.7.1 Funkčné požiadavky

F1: Aktualizácia dát v novej databáze

Nová databáza pre vyhľadávanie v pravidlách bude schopná pravidelných aktualizácií dát. Táto aktualizácia dát bude súčasťou aktuálneho procesu aktualizácie dát.

F2: Mikroslužba spracováva virtuálne shopy

Nová mikroslužba bude získavať vytvorené virtuálne shopy a ukladať ich do databáze.

F3: Mikroslužba invaliduje virtuálne shopy

Nová mikroslužba bude získavať invalidované `uids` pre uložené virtuálne shopy.

F4: Mikroslužba rieši nájdenú nedokonalosť

Vrámci analýzy slabín bola nájdená nedokonalosť pôvodného riešenia invalidácie virtuálnych shopov 2.4.7. Nová mikroslužba bude správne invalidovať aj virtuálne shopy vzniknuté počas aktualizácie.

F5: Mikroslužba poskytuje invalidované uid

Aplikácia pre aktualizáciu dát nebude zaťažovať produkčnú databázu získavaním `uids` virtuálnych shopov, ale bude ich získavať pomocou REST API z novej mikroslužby.

2.7.2 Nefunkčné požiadavky

NF1: Výkon

Nová architektúra musí zvládať spracovať minimálne 30000 požiadaviek za minútu s priemerným časom odpovede pre jeden požiadavok do 100 milisekúnd. Samotné databázové vyhľadávanie v pravidlách musí byť zvládnuté s priemernou rýchlosťou do 50 milisekúnd.

NF2: Škálovateľnosť

Nová architektúra musí byť jednoducho škálovateľná v zmysle rastúcej veľkosti dát a rastúceho počtu požiadaviek na službu. S pribúdaním nových zákazníkov a rozširovaním služby do nových zemí bude rásť, mimo iné, počet pravidiel pomocou ktorých služba vyhľadáva jednotlivé shopy k transakciám. Rovnako bude rásť aj celkový počet požiadaviek na službu. Služba musí teda jednoducho škálovať, aby dovoľovala zvyšovanie počtu pravidiel a požiadaviek.

NF3: Vysoká dostupnosť

Nová databáza pre vyhľadávanie v pravidlách musí byť spustená v režime vysokej dostupnosti.

NF4: Konfigurovateľnosť

Novo pridaná funkcionalita musí byť zapínateľná, teda služba sa bude dať nakonfigurovať tak, aby mohla používať súčasnú alebo novú funkcionalitu. Je to najmä z dôvodu interných instancií služby na ktoré nie je kladený až taký nárok na výkon. Tie budú aj naďalej používať pôvodnú architektúru.

NF5: Flexibilný dátový model

Nová databáza pre vyhľadávanie v pravidlách musí podporovať zmenu dátového modelu.

NF6: Infraštruktúra

Novo vzniknuté komponenty a súčasti aplikácie budú využívať služby AWS a bežať v AWS infraštruktúre.

NF7: Monitoring

Novo vzniknuté komponenty budú monitorované pomocou AWS služby CloudWatch.

NF8: Technológie

Novo vzniknuté aplikácie budú napísané v jazyku Java s použitím Spring Boot frameworku.

2.8 Prípady použitia

Prípady použitia komunikujú na vysokej úrovni, čo má systém robiť. Ich popisy sú založené na identifikovaných požiadavkách. Zachytávajú typickú interakciu systému so systémovými používateľmi, teda koncovými používateľmi alebo inými systémami [37].

- **UC1: Aktualizácia dát v novej databáze**
  Súčasťou pravidelných aktualizácií dát bude krok v ktorom sa aktualizujú dáta aj novej databázy pre vyhľadávanie pravidiel. Aktérom je pravidelný *job*.
- **UC2: Virtuálne shopy sa spracovávajú asynchrónne**
  Ak sa pri spracovaní požiadavku vytvorí virtuálny shop, tak bude asynchrónne spracovaný novou mikroslužbou. Aktérom je udalosť príchodu dát.
- **UC3: Mikroslužba invaliduje virtuálne shopy**
  Aplikácia pre aktualizáciu dát už neinvaliduje virtuálne shopy, ale namiesto toho komunikuje pomocou REST API s novou mikroslužbou. Posiela jej pokyn pre invalidáciu a potom získava invalidované `uids`. Aktérom je aplikácia pre aktualizáciu dát.
- **UC4: Mikroslužba invaliduje virtuálne shopy vzniknuté počas aktualizácie dát**
  Mikroslužba adresuje nájdenú nedokonalosť 2.4.7 a invaliduje aj virtuálne shopy vzniknuté počas aktualizácie dát. Aktérom je čas.

Pre kontrolu splnenia všetkých funkčných požiadaviek slúži tabuľka 2.2. Každý požiadavok musí byť pokrytý aspoň jedným prípadom použitia.

	UC1	**UC2**	**UC3**	**UC4**
F1	+			
F2		+		
F3			+	
F4				+
F5			+	

Tabuľka 2.2: Pokrytie funkčných požiadaviek

Kapitola 3

Návrh

Táto kapitola je rozdelená na dve časti. Prvá časť popisuje architektúru MongoDB Sharded Clustra a náslenú integráciu do aplikácie. Druhá časť sa zaoberá softwarovým návrhom mikroslužby zodpovednej za správu a invalidovanie virtuálnych shopov.

3.1 MongoDB Sharded Cluster

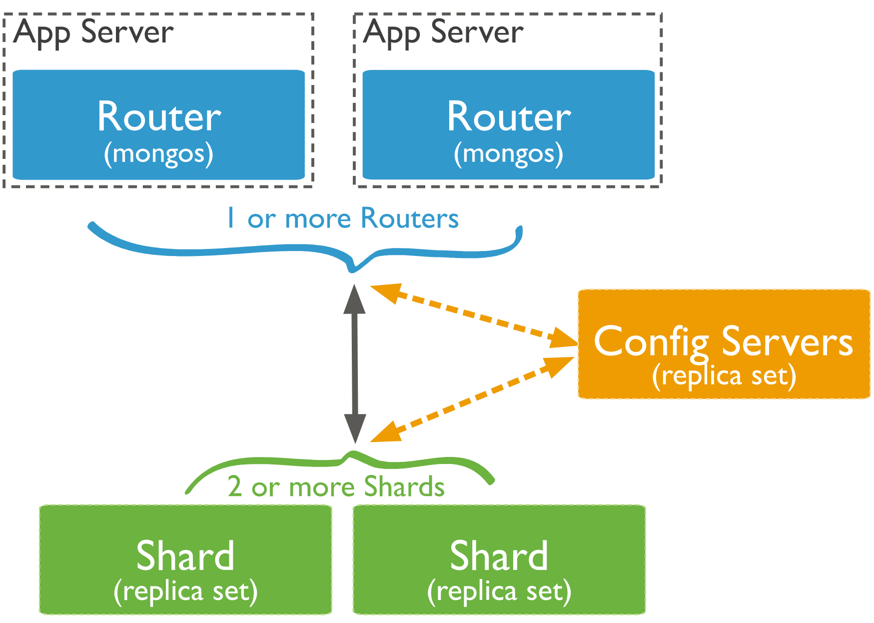
Dôležitou súčasťou clustra je aj replikácia. Replica-set je skupina mongod procesov, ktoré si udržiavajú rovnaký dátový stav. Mongod je primárny démon proces riadiaci MongoDB. Replica-set poskytuje redundanciu a vysokú dostupnosť. S viacerými kópiami dát na rôznych serveroch poskytuje replikácia určitú mieru odolnosti voči strate jedného databázového servera. Taktiež zvyšuje výkon pre *read* požiadavky, pretože klienti môžu posielať *read* požiadavky na rôzne servery. Jeden člen takejto množiny je primárny, ten prijíma všetky operácie pre zápis dát, aplikuje ich a zaznamenáva zmeny do operačného logu. Sekundárni členovia replikujú operačný log a aplikujú zmeny na svojich dátach. Ak primárny člen nie je dostupný, sekundárni členovia zvolia nového [32].

Architektúru clustra je možné vidieť na obrázku 3.1 a skladá sa z nasledujúcich komponent [32]:

- **Mongos** Smeruje dotazy a požiadavky pre zápis dát do jednotlivých shardov. Aby boli toho schopné, používajú dáta uložené na konfiguračných serveroch. Mongos poskytuje rozhranie medzi klientskými aplikáciami a clustrom. Pri posielaní dotazov do clustra, určí mongos zoznam shardov, ktoré musia prijať dotaz a nastaví kurzory na cieľové shardy. Pri odpovedi zlúči dáta z každého shardu a vráti výsledné dokumenty. Ak sa použije shard kľúč, mongos smeruje takéto požiadavky na konkrétny shard alebo skupinu shardov. Požiadavky, ktoré shard kľúč neobsahujú sú smerované na všetky shardy. Potom sa získajú jednotlivé odpovede, zlúčia sa získané dáta a vrátia sa výsledné dokumenty.

- **Shard** Obsahuje podmnožinu distribuovaných dát. Každá databáza v clustri má primárny shard, na ktorom sú mimo iné aj neshardované kolekcie. Mongos zvolí primárny shard pri vytváraní novej databázy. Ten je zvolený na základe najmenšieho obsahu dát.

- **Config severs** Ukladajú metadáta a konfiguračné nastavenia clustra. Metadáta odrážajú stav všetkých dát a komponent v clustri, zahŕňajú napríklad aj zoznam fragmentov pre každý shard a ich definované rozsahy. Fragment pozostáva z istého rozsahu shardovaných dát. Mongos dočasne ukladajú a pri potrebe aktualizujú dáta poskytované config servermi. Tie potom používajú na smerovanie požiadaviek na správne shardy. Štandardne sú nasadené ako replica-set. V prípade, že by nastal výpadok primárneho člena a nemohol by byť zvolený nový, metadáta uložené na týchto serveroch budú dostupné iba na čítanie. Vtedy je stále možné čítať a zapisovať dáta do clustra, ale nie je už možná migrácia alebo rozdeľovanie fragmentov. Dáta uložené na konfiguračnom serveri sú malé v porovnaní s dátami uloženými v clustri. Tieto servery predstavujú nízku záťaž z hľadiska výkonu.



Obr. 3.1: Obecná architektúra MongoDB sharded clustra [32]

MongoDB delí rozsah hodnôt alebo hashovaných hodnôt shard kľúča na neprekrývajúce sa rozsahy. Každý rozsah je spojený s fragmentom dát a MongoDB sa snaží rovnomerne tieto fragmenty rozdeliť medzi shardy. Výber shard kľúča priamo ovplyvňuje vytváranie a distribúciu fragmentov medzi shardy. Distribúcia dát potom ovplyvňuje efektivitu a výkonnosť operácií nad clustrom. Medzi najzákladnejšie parametre shard kĺúča patria kardinalita a frekvencia. Shard kľúč by mal mať vysokú kardinalitu, tá určuje maximálny počet fragmentov, ktorý môže byť vytvorený. Frekvencia shard kľúča udáva ako často sa nejaká hodnota shard kľúča vyskytuje v dátach. Ak väčšina dokumentov obsahuje iba podmnožinu možných hodnôt shard kľúča, potom fragmenty, ktoré

ukladajú dáta s týmito hodnotami môžu byť úzkym hrdlom clustra a horizontálne škálovanie nebude ďalej také efektívne.

3.1.1 Návrh kolekcie pravidiel

Kolekcia pravidiel bude mať podobnú štruktúru ako tabuľka pravidiel. Polia dokumentov v kolekcií budú:

- `id`
- `merchant_id`
- `pos_id`
- `merchant_description`
- `merchant_description_wild`
- `shop_id`
- `country`
- `mcc`
- `city`
- `zip`
- `priority`
- `created`

Kolekcia bude mať názov *tapix_rules_<timestamp>* a aplikácia po štarte zistí všetky dostupné kolekcie v clustri a vyberie tú, ktorá má najvyšší *timestamp*.

3.1.1.1 Výber shard kľúča

Z povahy dát by nebolo vhodné použiť range stratégiu pre distribúciu dát, preto bola použitá hash stratégia. *Hashed sharding* používa buď jedno pole dokumentu pre hashed index alebo zložený hashed index. MongoDB síce podporuje zložený hashed index, ale len jedno pole zo zloženého kľúča bude zahashované. Bolo testovaných niekoľko konfigurácií shard kľúča. Polia s najvyššou kardinalitou a najnižšou frekvenciou null hodnôt sú: `merchant_id`, `merchant_description` a `pos_id`. Presný podiel null hodnôt voči všetkým záznamom je možné vidieť v tabuľke 3.1. Podiel null hodnôt vo zvyšných polí je ešte vyšší.

	merchantId	**merchantDescription**	**posId**
podiel null hodnôt (%)	17,8	28,2	75,5

Tabuľka 3.1: Tabuľka podielu null hodnôt pre polia pravidiel

Ďalej sú popísané testované konfigurácie shard kľúča pre cluster s troma shardmi, ak je nejaké pole hashed, bude podčiarknuté. Výsledné rozloženie dát pre kľúč, ktorý je tvorený z polí vyzerá nasledovne:

- merchant_id, merchant_description, pos_id
  - shard 1 – 59,8 %
  - shard 2 – 19,9 %
  - shard 3 – 20,3 %
- merchant_id, merchant_description, pos_id
  - shard 1 – 76,7 %
  - shard 2 – 23,3 %
  - shard 3 – 0.0 %
- merchant_id, merchant_description
  - shard 1 – 59,8 %
  - shard 2 – 19,9 %
  - shard 3 – 20,3 %
- merchant_id, merchant_description
  - shard 1 – 59,1 %
  - shard 2 – 21,1 %
  - shard 3 – 19,8 %
- merchant_id, merchant_description
  - shard 1 – 68,6 %
  - shard 2 – 21,2 %
  - shard 3 – 10,2 %

Z nasledujúcich výsledkov je vidieť, že distribúcia dát je najviac závislá na hashed poli. Preto pre ďalšie testovanie bude vytvorené nové pole concat_md ktoré bude reťaziť merchant_id a merchant_description. Ak bude nejaké z týchto dvoch polí null, zreťazí sa ako prázdny reťazec. Toto pole ma ešte vyššiu kardinalitu a podiel null hodnôt voči všetkým záznamom len 0,5%. Rozloženie dát vyzerá následovne:

- shard 1 – 26,7 %
- shard 2 – 46,6 %
- shard 3 – 26,7 %

Pri pridaní pos_id a zreťazení všetkých troch polí vyzerá rozloženie následovne:

- shard 1 – 26,7 %
- shard 2 – 46,6 %
- shard 3 – 26,7 %

Ďalšie pole nemá na distribúciu dát už žiaden vplyv, preto nie je ani potrebné ho mať ako súčasť shard kľúča. Posledná optimalizácia bude filtrácia pravidiel. Pravidlá s vyplneným `merchant_description_wild` sa vyhľadávajú v pamäti, preto môžu byť odfiltrované a kolekcia pravidiel nebude ani potrebovať toto pole. Po odfiltrovaní takýchto pravidiel je rozloženie nasledujúce:

- shard 1 – 33,6 %
- shard 2 – 33,2 %
- shard 3 – 33,2 %

Finálny návrh kolekcie bude teda pozmenený tak, že z kolekcie sa odstráni pole `merchant_description_wild`, pravidlá ktoré toto pole nemajú null budú odfiltrované a nakoniec sa pridá pole `concat_md`, ktoré bude slúžiť ako shard kľúč.

3.1.2 Vyhľadávanie v pravidlách

V MongoDB sa bude vyhľadávať rovnakým spôsobom ako sa teraz vyhľadáva v PostgreSQL a teda pre SQL dotaz v príklade 2.1, bude MongoDB dotaz vyzerať ako je uvedené v ukážke 3.1. Dotazy boli otestované s operátorom `$in` aj s operátorom `$or`. Z hľadiska výkonu nebol medzi nimi pozorovaný žiadny rozdiel, ale pri porovnávaní rovnakého poľa s nejakou hodnotou je lepšie použiť operátor `$in` [32].

```
1   db.tapix_rules.find({
2     $and: [
3       { concat_md: { $in: ['123456789', ''] } },
4       { merchant_id: { $in: ['123456789', null] } },
5       { pos_id: { $in: ['0000', null] } },
6       { merchant_description: null },
7       { country: { $in: ['CZ', null] } },
8       { mcc: null },
9       { city: null },
10      { zip: null }
11    ]
12  })
```

Výpis kódu 3.1: MongoDB vyhľadávanie v pravidlách

Hodnoty, ktoré môže pole `concat_md` pri vyhľadávaní nadobúdať, sú kombinácie možných hodnôt polí `merchant_id` a `merchant_description` na vstupe. Pri vyhľadávaní pravidiel podľa vstupných hodnôt sa vyhľadáva pravidlo, ktoré v odpovedajúcom stĺpci obsahuje vstupnú hodnotu alebo null. Ak je vstupná hodnota null, odpovedajúce pravidlo musí mať v tomto stĺpci tiež hodnotu null. Pre vstup, ktorý by obsahoval `merchant_id` aj `merchant_description` by `concat_md` mohol vyzerať takto.

```
//hodnoty na vstupe
    merchantId=MID
    merchantDescription=DESC

//možné hodnoty concat_md
    concat_md=MIDDESC
    concat_md=MID
    concat_md=DESC
    concat_md=''
```

Pri vyhľadávaní pravidiel sa používa vstupná hodnota poľa alebo null, pre dve polia to znamená štyri možné kombinácie. V prípade, že by `merchant_id` aj `merchant_description` boli na vstupe null, tak `concat_md` môže nadobúdať jedine hodnotu prázdneho reťazca. Pretože pri reťazení sa null reťazí ako prázdny reťazec.

Takýto shard kľúč zabezpečuje, že bude môcť byť súčasťou každého dotazu a že bude cieliť na maximálne štyri shardy. Pretože shard kľúč bude mať v dotaze definované maximálne štyri hodnoty, tak cluster môže mať ľubovoľný počet shardov, ale pre jeden vyhľadávací dotaz sa použijú maximálne štyri shardy a nie je potrebný *broadcast* na všetky.

3.1.3 Infraštruktúra clustra

Existuje niekoľko možností, ako prevádzkovať MongoDB Sharded Cluster. Najjednoduchšou možnosťou by bolo použiť MongoDB Atlas, čo je spravovaná databázová služba, ktorá výrazne uľahčuje nasadenie a správu MongoDB. Táto služba je ale pomerne nákladná a nie je žiadúce využívať veľa služieb tretích strán. Ďalšou možnosťou by bolo vytvoriť tento cluster s použitím iba EC2 inštancií v AWS. Takéto riešenie by bolo ale veľmi náročné na správu a údržbu. Lepšou alternatívou by bolo využiť AWS služby ako *Elastic Container Service (ECS)* alebo *Elastic Kubernetes Service (EKS)*. ECS predstavuje spravovanú službu pre orchestráciu kontajnerov. EKS je spravovaný Kubernetes cluster.

Kubernetes

Je rozšíriteľná open-source platforma pre správu kontajnerizovaných služieb, ktorá umožňuje deklaratívnu konfiguráciu a automatizáciu. Prečo je Kubernetes potrebný? Kontajnery sú dobrým spôsobom ako zabaliť a spúšťať aplikácie. V produkčnom prostredí je potrebné spravovať kontajnery, v ktorých bežia aplikácie a zabezpečiť bezvýpadkovosť. Napríklad ak kontajner zlyhá, mal by byť spustený iný kontajner, ktorý ho nahradí. Práve toto zabezpečuje Kubernetes. Poskytuje platformu pre správu distribuovaných systémov s vysokou odolnosťou voči výpadkom. Medzi hlavné charakteristiky Kubernetes patrí jednoduché horizontálne škálovanie a teda pridávanie alebo odoberanie výkonu, manuálne aj automaticky. Ďalšou funkcionalitou je, že Kubernetes dokáže reštartovať, vymeniť alebo zastaviť kontajnery, ktoré zlyhali alebo neodpovedajú na užívateľom zvolené *health checks*. Je možné popísať novú konfiguráciu stavu nasadeného kontajnera a Kubernetes zariadi zmenu aktuálneho stavu na novo definovaný. Pre každý kontajner stačí zadefinovať koľko CPU a pamäti RAM potrebuje a Kubernetes tieto kontajnery nasadí na jednotlivé výpočetné uzly clustra tak, aby čo najefektívnejšie využil ich zdroje.

Kubernetes cluster sa skladá z množiny výpočetných strojov zvaných uzly. Tie slúžia pre beh podov. Pod je skupina jedného alebo viacerých kontajnerov so zdieľaným úložiskom, sieťovými prostriedkami a špecifikáciou akou sa spúšťajú kontajnery. Druhou zložkou je *control plane*, ten slúži na správu uzlov a podov v clustri [38].

Workload je aplikácia, ktorá beží v Kubernetes. Bez ohľadu na to, či je to samostatná komponenta alebo skupina komponent pracujúcich spolu v Kubernetes je spustená v sade podov. Namiesto spravovania jednotlivých podov samostatne je možné využiť *workload resources*, ktoré spravujú množinu podov a zabezpečujú, že v clustri beží správny počet podov správneho typu. Medzi *workload resources* patrí [38]:

- **Deployment** používa sa na spravovanie aplikácií bez stavu. Každý Pod v rámci Deploymentu je vzájomne vymeniteľný a môže byť nahradený ak je to potrebné.
- **StatefulSet** umožňuje spúšťať jeden alebo viacero súvisiacich podov, ktoré sledujú svoj stav. Narozdiel od Deploymentu udržuje pre každý z jeho podov trvalý identifikátor, tieto pody niesu vzájomne vymeniteľné. Ak je takýmto podom priradené uložisko, a pod by zlyhal, trvalý identifikátor umožňuje priradiť toto existujúce úložisko novému podu, ktorý nahradí ten zlyhaný.

PersistentVolume (PV) je uložný priestor v clustri. PV majú nezávislý životný cyklus od podov, ktoré ich využívajú. PersistentVolumeClaim (PVC) je požiadavok užívateľa na úložný priestor.

Service je abstrakcia pre vystavenie sieťových aplikácií, ktoré bežia ako jeden alebo viacero podov v clustri. Každý pod napríklad vrámci Deployment dostane vlastnú IP adresu. Pody môžu byť odstránené alebo nahradené za iné, a aby odpovedali žiadanému stavu clustra, nemusia byť teda dlhotrvajúce. Aby ostatné služby mohli pristupovať k aplikácií, ktorá beží v takýchto podoch a nemuseli riešiť jednotlivé IP adresy podov, používajú k prístupu Service.

Amazon Elastic Block Store (Amazon EBS) CSI ovládač spravuje životný cyklus úložiska pre EKS. EKS predstavuje robustnejšie riešenie, určené pre ľahšie škálovanie a správu než ECS.

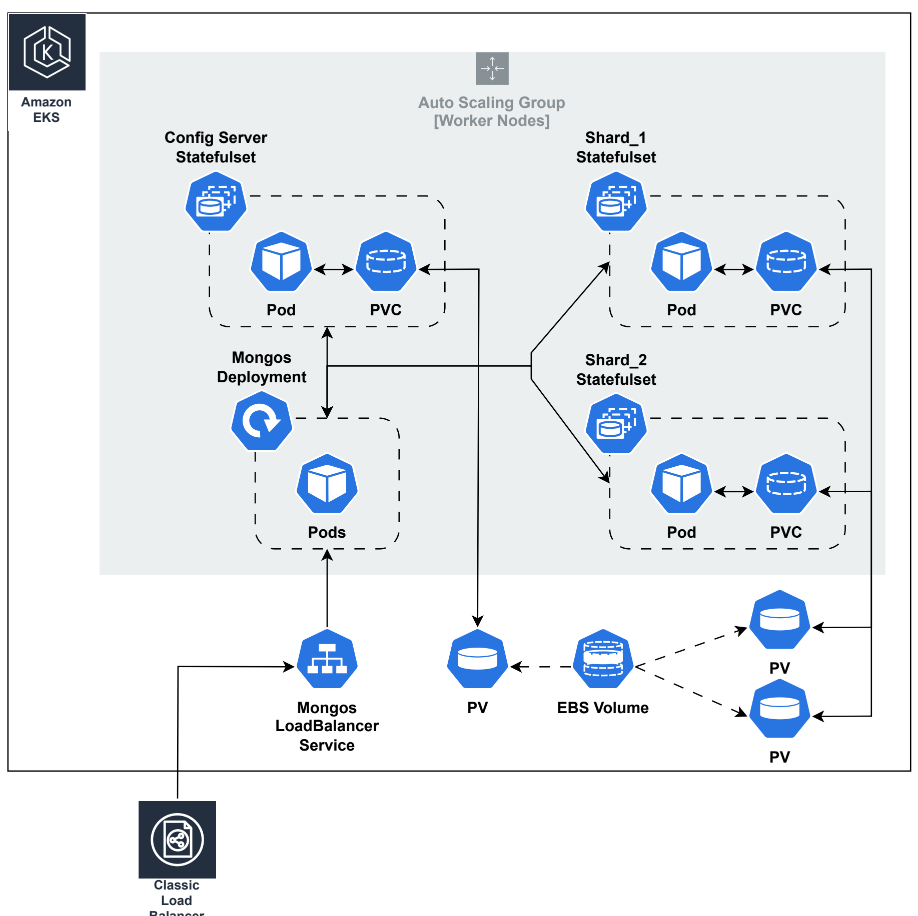
Návrh clustra s najdôležitejšími komponentami a dvoma shardmi je zachytený na diagrame 3.2. Deployment a Statefulset zdroje používajú replica-set s hodnotou 3, teda v každom zdroji bežia 3 pody s ich vlastnými PVC. Cluster bude v privátnej sieti a rovnako aj Classic Load Balancer, ktorý bude slúžiť pre prístup k Mongos.

3.1.4 Integrácia MongoDB do aplikácie

Pre integrovanie nového databázového systému do aplikácie je potrebné abstrahovať vrstvu spojenú s vyhľadávaním v tabuľke pravidiel. Návrh štruktúry kľúčových tried je možné vidieť na diagrame 3.3.

`TxSearchDao` Rozhranie, ktoré poskytuje metódy potrebné pre vyhľadávanie v pravidlách.

`TxRuleDao` Trieda implementuje vyhľadávanie pravidiel v PostgreSQL databáze. Jedná sa o pôvodné riešenie vyhľadávania.



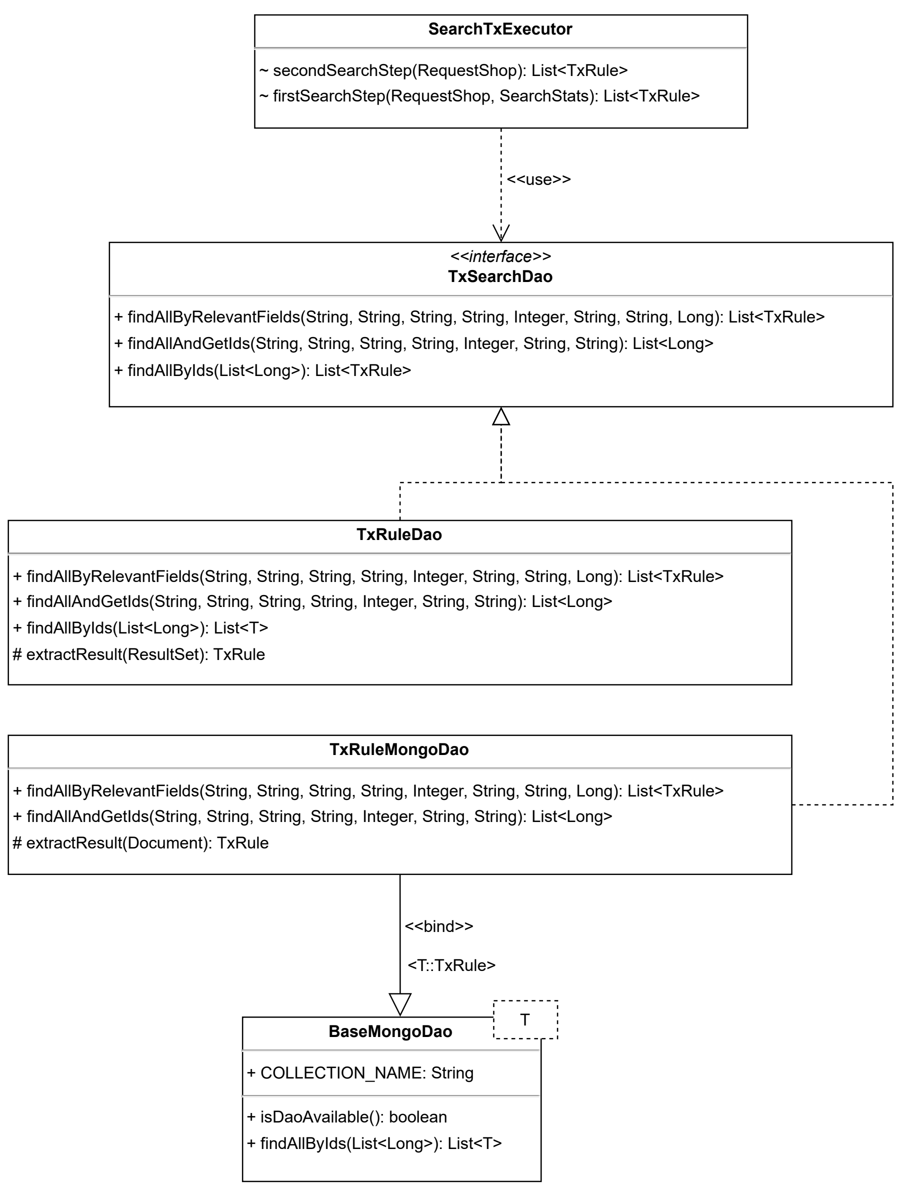
Obr. 3.2: MongoDB Sharded Cluster v EKS

`TxRuleMongoDao` Trieda implementuje vyhľadávanie pravidiel v MongoDB.

`BaseMongoDao` Trieda poskytuje základné metódy ako napríklad prístup ku kolekcií v MongoDB alebo pre serializáciu výsledkov dotazov. Trieda je generická a teda môže byť prepoužitá pre prístup k ľubovoľnej kolekcií v MongoDB.

`SearchTxExecutor` Trieda implementujúca logiku vyhľadávania transakčných pravidiel. K vyhľadávaniu používa implementáciu rozhrania, ktorú dostane k dispozícií.

Pre spätnú kompatibilitu je vyhľadávanie v MongoDB zapínateľná funkcionalita. Ak sa pri štarte aplikácie zistí, že MongoDB databáza nie je dostupná, automaticky sa použije vyhľadávanie v PostgreSQL.



Obr. 3.3: Diagram tried pre komponentu vyhľadávania v pravidlách

3.1.5 Aktualizácia kolekcie dát v MongoDB

V MongoDB clustri bude uložená kolekcia pravidiel, ktorá bude rovnako ako ostatné dáta v aplikácií vyžadovať pravidelné aktualizácie. Aktualizácia kolekcie bude integrovaná do súčasného procesu aktualizácie dát. Súčasný proces aktualizácie dát je popísaný v časti 2.3.4. Pridaná funkcionalita vytvorí novú shardovanú kolekciu s názvom *update*. Zo stiahnutého balíčka použije súbor pravidiel, ktorý ešte odfiltruje od pravidiel s vyplneným `merchant_description_wild`

a pridá nové pole `concat_md`. Tento súbor bude následne naimportovaný do novej kolekcie. Po úspešnom importe dát, vytvorí všetky potrebné indexy a ďalej sa začne s aktualizáciou dát v PostgreSQL. Pre elimináciu potenciálnych chýb je potrebné aby:

1. balíček dát bol stiahnutý práve jeden krát a použitý na aktualizáciu dát v MongoDB aj PostgreSQL

2. premenovanie novej dátovej schémy v PostgreSQL a aj novej kolekcie v MongoDB musí prebehnúť ako posledný krok úspešnej aktualizácie

Prvý bod zabraňuje prípadu, kedy by sa stiahol balíček určený k importu a bol by naimportovaný do MongoDB. Medzitým by bola nasadená nová verzia balíčka, ktorá by sa stiahla a použila pre import do PostgreSQL. To by mohlo znamenať nekompatibilitu a nekonzistenciu dát medzi MongoDB a PostgreSQL, čo môže viesť až k chybným odpovediam klientom alebo dokonca chybám na strane servera. Druhý bod zabraňuje prepnutiu aplikácie do nekonzistentného stavu. Keby počas aktualizácie nastala chyba v ľubovoľnom kroku, ale kolekcia alebo schéma by boli premenované na názov s najvyšším *timestamp*, tak po najbližšom štarte by aplikácia začala používať tieto chybné dáta. Proces aktualizácie aj s aktualizáciou kolekcie v MongoDB je možné vidieť na diagrame 3.6.

3.2 Mikroslužba pre správu virtuálnych shopov

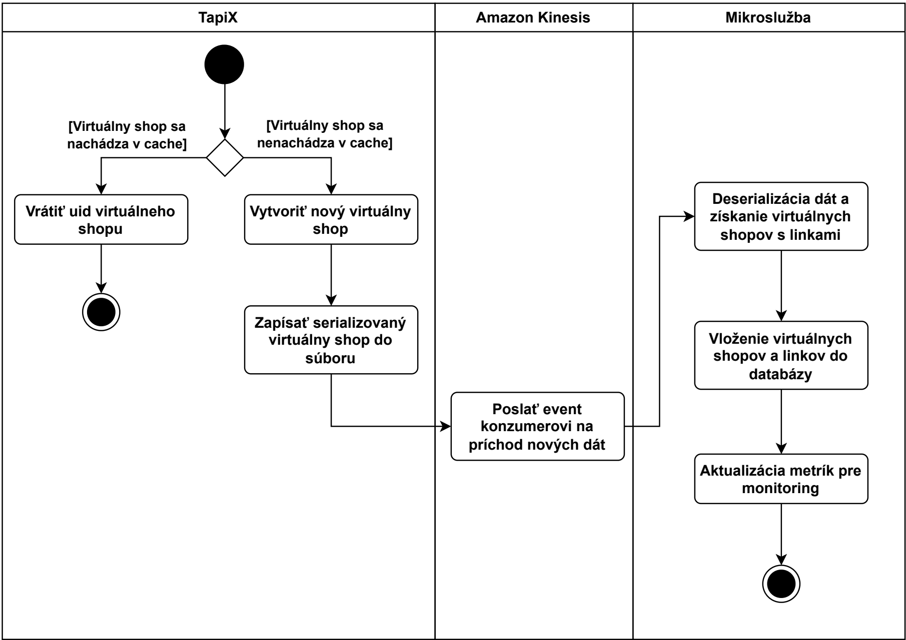
Mikroslužba s názvom TapixVsUpdater bude plniť dva účely. Prvým bude konzumácia virtuálnych shopov vytvorených v TapiX aplikácií. Druhým bude invalidácia týchto shopov. Dôležitým krokom bude aj správne vytváranie virtuálnych shopov na strane TapiX aplikácie.

3.2.1 Konzumácia virtuálnych shopov

Virtuálne shopy sa vytvoria v TapiX aplikácií, no namiesto ich vyhľadávania v databáze a vkladania do nej, spolu s linkami, sa virtuálne shopy a linky zapíšu do súboru v serializovanom tvare. Pri použití databázy sa nevytvárali duplicitné virtuálne shopy, pretože sa najskôr vyhľadávali v databáze. Duplicitné virtuálne shopy nepredstavujú problém, ale zvyšujú počet invalidovaných `uids`. Pre redukciu týchto duplicít bude použitá cache a prefix kľúča pre vložené dáta bude aktuálna verzia schémy dátového balíka, ktorý sa používa. To zaistí invalidáciu cache po aktualizácií dát. Súbor so serializovanými dátami bude potom prenesený do služby CloudWatch. Tieto virtuálne shopy by mohli byť prenášané zo služby CloudWatch, napríklad pomocou služby Lambda priamo do databáze novej mikroslužby. Takéto riešenie má niekoľko problémov. Prvým problémom je, že pri výpadku databázy alebo chybe Lambda funkcie budú virtuálne shopy chybne spracované a stratené. Druhým problémom by bola silná závislosť databázového modelu a Lambda funkcie.

Správnym riešením bude virtuálne shopy zo služby CloudWatch streamovať do služby Kinesis a mikroslužba bude tieto dáta konzumovať tiež priamo z Kinesis. Takáto architektúra zabezpečí istý level robustnosti riešenia a pri výpadku mikroslužby alebo jej databázy, budú virtuálne shopy uchované po istý

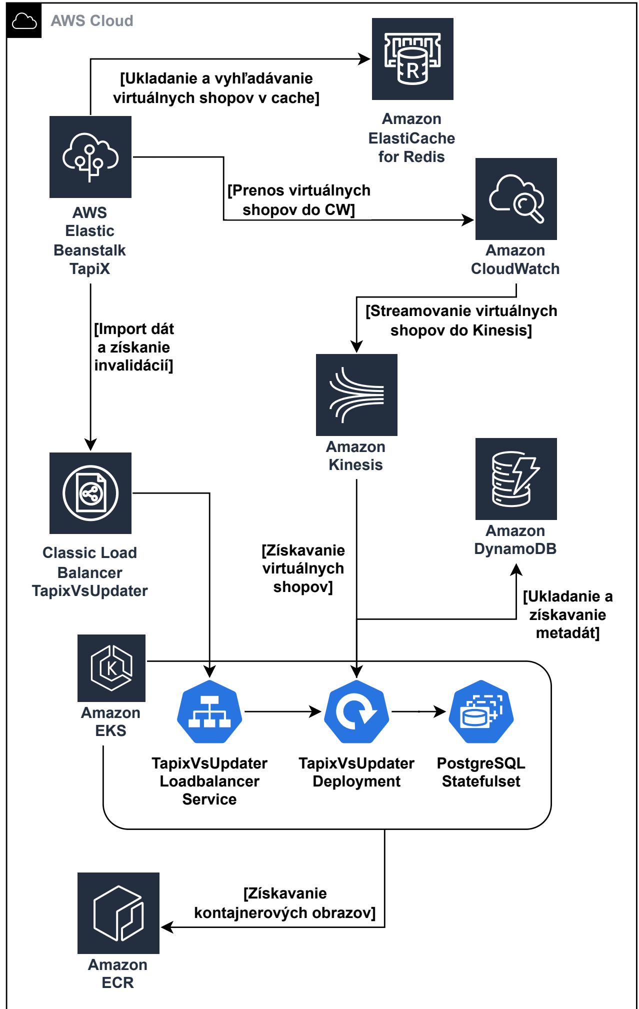
časový interval. Po vyriešení výpadku služby môže služba pokračovať v spracovávaní virtuálnych shopov tam kde prestala. Spracovanie virtuálneho shopu znamená deserializácia dát a vloženie do databáze. Ukladať virtuálne shopy do súboru, ktorý bude streamovaný do služby CloudWatch má tú výhodu, oproti priamemu zápisu do Kinesis, že priamy zápis môže zlyhať. Zlyhanie pri synchrónnom zápise do Kinesis bude znamenať obmedzenie TapiX služby a pri asynchrónnom môže hroziť strata dát. Zápisom virtuálnych shopov do súboru nehrozí strata dát, pretože virtuálne shopy budú uložené v službe CloudWatch, ani obmedzenie TapiX služby, pretože nebude priamo závislá na Kinesis. Potencionálna chyba, pri takomto prenose dát z CloudWatch do Kinesis, službu vôbec neovplyvní. Proces vytvárania a spracovania virtuálnych shopov je možné vidieť na diagrame 3.4.



Obr. 3.4: Diagram aktivít procesu vytvorenia a spracovania virtuálneho shopu

Mikroslužba bude vytvorená v jazyku Java, konkrétne Java 17 a s využitím frameworku SpringBoot. Ďalej bude použitá open source objektovo-relačná databáza PostgreSQL. Tieto technológie boli zvolené pretože sú vhodné pre implementáciu popisovanej mikroslužby. Ďalším argumentom je, že sú to aktuálne využívané technológie v spoločnosti, pre ktorú je práca určená, teda ostatní vývojári majú s nimi praktické skúsenosti a nebude pre nich problém rozširovať funkcionalitu vytvorenej mikroslužby. Ako cache pre ukladanie virtuálnych shopov sa použije Redis, konkrétne *ElastiCache for Redis*. Pre konzumácia dát z Kinesis existuje niekoľko riešení ako napríklad využiť AWS SDK, Kinesis Client Library alebo AWS Kinesis Binder pre Spring Cloud Stream. Najjednoduchšie a dostačujúce riešenie, ktoré vyžaduje čo najmenej integračného kódu je použiť AWS Kinesis Binder knižnicu pre framework Spring Cloud Stream, ktorý slúži na vytváranie škálovateľných *event-driven* mikroslužieb. AWS Kinesis Binder

knižnica používa DynamoDB pre ukladanie metadát ako napríklad pozícia čítania dát z Kinesis streamu. Mikroslužba spolu s databázou budú bežať v EKS clustri, rovnako ako MongoDB Sharded Cluster. Pre ukladanie nových verzií obrazov aplikácie, bude použitá služba Elastic Container Registry, čo je spravovaný register kontajnerových obrazov. Vysoko úrovňovú architektúru je možné vidieť na diagrame 3.5. Smer šípok v grafe znázorňuje volania služieb.



Obr. 3.5: Vysoko úrovňová architektúra správy virtuálnych shopov

3.2.2 Invalidácia virtuálnych shopov

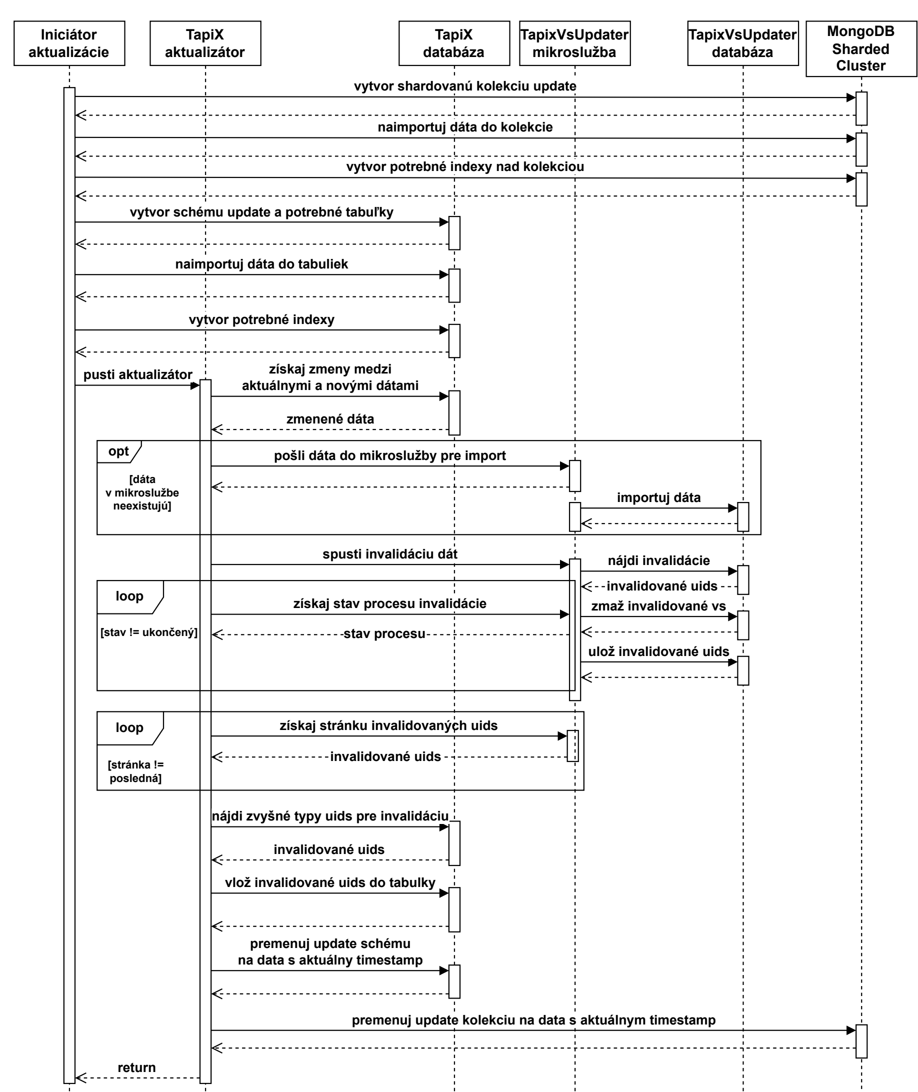
Vďaka ukladaniu virtuálnych shopov do databáze mikroslužby, je možné, aby mikroslužba získavala, teda vyhľadávala vo svojej databáze `uids` invalidovaných virtuálnych shopov. Pozitívny dopad to má na produkčnú databázu TapiX aplikácie, ktorá už nebude počas aktualizácie zaťažovaná získavaním `uids` invalidovaných virtuálnych shopov. Celý proces invalidácie na mikroslužbe pre-

bieha v transakcií a pri ľubovoľnej chybe je vyhodená výnimka, ktorá zabezpečí *rollback* transakcie a udržanie dát v konzistentnom stave. Všetky kroky aktualizácie aj s aktualizovaním MongoDB je možné vidieť na diagrame 3.6. Proces invalidácie virtuálnych shopov je popísaný v niekoľkých bodoch:

- Spustí sa proces aktualizácie dát.
- Vytvoria sa potrebné rozdiely dát, ktoré budú slúžiť na získanie `uids` invalidovaných entít.
- Po invalidáciach shopov, merchantov a pravidiel sa spustí externá invalidácia na mikroslužbe:
  - Skontroluje sa či TapixVsUpdater už nemá naimportovaný aktuálny rozdiel dát. Pokiaľ nie, tak sa vyfiltrujú z rozdielu relevantné dáta pre invalidáciu virtuálnych shopov a výsledok sa pošle do mikroslužby pomocou REST API.
  - Mikroslužba tieto dáta uloží do novej schémy.
  - Na mikroslužbu sa odošle požiadavok, ktorý spustí invalidáciu virtuálnych shopov. Tento požiadavok bude asynchronný.
  - Vyhľadávanie `uids` bude fungovať obdobne ako v pôvodnom riešení, virtuálne shopy budú vyhľadávané na základe zmenených pravidiel. Rozdielom bude, že pri tomto návrhu už virtuálne shopy po invalidácií nebudú potrebné, a preto budú zmazané spolu s ich linkami.
  - Nájdene `uids` sa uložia do tabuľky spolu s ďalšími metadátami a všetkým virtuálnym shopom sa nastaví príznak `processed` na hodnotu *true*.
  - Proces aktualizácie bude vykonávať *polling* na mikroslužbu ohľadom vývoja invalidácií, kým nedostane odpoveď, že invalidovanie bolo úspešne skončené.
  - Proces aktualizácie získa všetky invalidované `uids` z mikroslužby pomocou REST API a uloží ich do tabuľky. Invalidované `uids` sa ukladajú do tabuľky v mikroslužbe v ktorej stĺpec `id` disponuje rastúcou sekvenciou. Počas procesu invalidácie sa tieto `uids` získavajú batchovo pomocou `id`. V TapiXe sa potom bude ukladať informácia o poslednom získanom `id`.
  - Do TapiX databáze sa uloží záznam s metadátami o externej invalidácií a získanom rozsahu invalidovaných `uids`.

3.2.2.1 Čiastočná invalidácia virtuálnych shopov

Pre adresovanie nájdenej nedokonalosti v pôvodnom riešení 2.4.7 bude mikroslužba v určitých časových intervaloch vykonávať čiastočnú invalidáciu. Tá bude invalidovať virtuálne shopy, ktoré vznikli počas aktualizácie dát s použitím správneho rozdielu dát. Na začiatku čiastočnej invalidácie sa získajú také názvy databázových schém s uloženými rozdielmi dát, pre ktoré existujú virtuálne shopy, ktoré zatiaľ neboli invalidované. To znamená, také virtuálne shopy, ktoré majú nastavený príznak `processed` na hodnotu *false* a



Obr. 3.6: Sekvenčný diagram aktualizácie dát

ich `schema_version` odpovedá verzií, ktorú má schéma s importovaným rozdielom dát. Pre každú získanú schému, sa spustí čiastočný proces invalidácie. Tento proces je totožný s procesom invalidácie popísaným v 3.2.2 až na niekoľko rozdielov. Pri vyhľadávaní virtuálnych shopov sa do jednotlivých SQL dotazov pridajú dve podmienky. Stĺpec `processed` musí mať hodnotu *false* a `schema_version` musí odpovedať verzií schémy, pre ktorú prebieha čiastočná invalidácia. Invalidované `uids` sa uložia do tabuľky a budú získané pri ďalšom kompletnom procese invalidácie. To vďaka vlastnosti, že `uids` sa získavajú z tabuľky na základe rastúceho `id`. Na konci procesu sa `processed` nastaví na hodnotu *true* len pre odpovedajúcu množinu virtuálnych shopov a nie pre všetky.

3.2.2.2 Návrh API

Komunikácia medzi procesom aktualizácie a mikroslužbou bude sprostredkovaná pomocou REST API. Pre zabezpečenie API bude použitá *Basic auth*.

GET `/v1/diff/exists/` Endpoint vyžaduje parameter `vsSchemaVersion` a jeho úlohou je skontrolovať či v databáze existuje schéma s importovaným rozdielom dát, ktorej verzia by odpovedala parametru `vsSchemaVersion`. Odpoveďou je `200 Ok` pokiaľ schéma existuje, inak vracia `404 Not Found`.

POST `/v1/diff/import/` Endpoint prijíma rozdiel dát medzi určitými tabuľkami dvoch schém pre potreby invalidácie. Pre tieto dáta vytvorí novú schému, dáta do nej naimportuje a vloží záznam do `control` tabuľky o tomto importe. Odpoveďou je `204 No Content`

GET `/v1/invalidations/invalidate` Endpoint slúži na spustenie procesu invalidácie virtuálnych shopov. Spustenie je asynchronné, odpoveďou endpointu je `202 Accepted`.

GET `/v1/invalidations/progress` Endpoint slúži ná získanie postupu v invalidáciách. Odpoveďou je aktuálny stav prebiehajúceho procesu invalidácie a správa, ktorá upresňuje v akej fáze invalidácií sa mikroslužba nachádza.

GET `/v1/invalidations` Endpoint vyžaduje parametre `fromId` a `pageSize`. Endpoint vracia zoznam invalidovaných `uids` od `id` špecifikovanom v parametri `fromId`. Vrátený zoznam má stránkovanú podobu a veľkosť stránky je špecifikovaná parametrom `pageSize`. Odpoveďou je zoznam `uids`, `id` posledného `uid` pre ďalšie volanie a príznak, či sa jedná o poslednú stránku alebo nie.

3.2.3 Databázový model

Dáta aplikácie budú uložené v niekoľkých schémach. V schéme `public` sú uložené dáta, ktoré bude spravovať samotná aplikácia. Databázový model public schémy je vidieť na 3.7.

`tx_virtual_shop` Tabuľka slúži na ukladanie virtuálnych shopov vytvorených v TapiX aplikácií. Oproti pôvodnej tabuľke virtuálnych shopov sú v tejto tabuľke naviac stĺpce: `schema_version`, `processed` a `created`. Stĺpec `schema_version` označuje meno dát, resp. databázovej schémy, ktorú používal TapiX pri vytvorení virtuálneho shopu. Stĺpec `processed` označuje, či bol už virtuálny shop spracovaný počas invalidácie. To neznamená, že virtuálny shop musel byť invalidovaný, ale že sa nachádzal v množine dát v ktorej sa vyhľadávalo počas invalidácie. Stĺpec `created` označuje čas vzniku virtuálneho shopu.

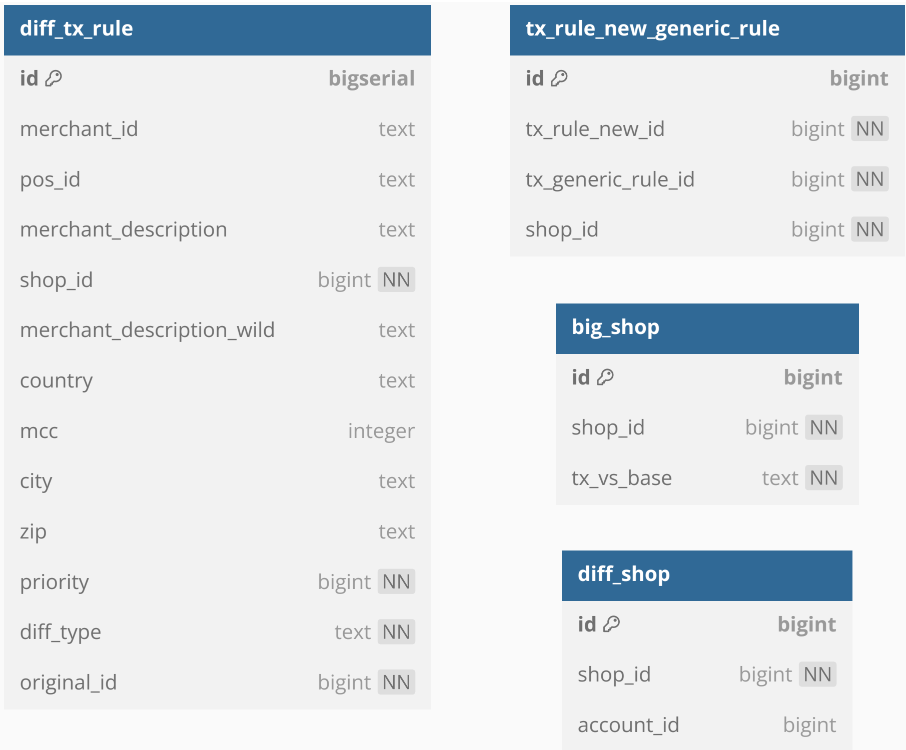
`tx_virtual_shop_source_rule` Tabuľka ukladá pre `id` virtuálneho shopu `ids` pravidiel, ktoré boli nájdené pre vstupné parametre transakcie.

`invalidated_virtual_shop` Tabuľka ukladá informácie o invalidovaných virtuálnych shopov a teda najmä invalidované `uids`. Nad stĺpcom `id` je vytvorená rastúca sekvencia.

Obr. 3.7: Databázový model pre schému public

`control` Tabuľka ukladá záznam o importe rozdielu dát z TapiX aplikácie do mikroslužby. Pri aktualizácií dát sa bude využívať mikroslužba na získanie invalidovaných `uids` virtuálnych shopov. Aby mikroslužba mohla nájsť tieto invalidované `uids`, bude potrebovať rozdiel medzi súčasnými a novými dátami. Tento rozdiel získa počas aktualizácie z TapiX aplikácie a všetky potrebné dáta uloží do samostatnej schémy. Záznam o získaní týchto dát bude uložený práve v `control` tabuľke. V stĺpci s názvom `schema_diff_version` je uložené meno schémy do ktorej sa dáta importovali. Stĺpec `vs_schema_version` ukladá zasa meno schémy virtuálnych shopov, ktoré majú byť invalidované oproti tomuto rozdielu dát.

Dáta, ktoré predstavujú rozdiel niektorých tabuliek súčasnej schémy a novej schémy vzniknutej počas aktualizácie sa dostanú do mikroslužby importom z TapiX aplikácie a uložia sa do novej schémy. Databázový model tejto schémy je znázornený na 3.8.



Obr. 3.8: Databázový model pre schému importovaných dát

`diff_tx_rule` Tabuľka ukladá pravidlá, ktoré vzišli z rozdielu medzi tabuľkami pravidiel počas aktualizácie dát. Stĺpec `diff_type` rozlišuje, či je pravidlo z rozdielu nové, upravené alebo či je zmazané, teda staré.

`tx_rule_new_generic_rule` Tabuľka ukladá pre nové alebo aktualizované pravidlá z rozdielu pravidiel ešte `id` rodičovského pravidla a `shop_id` shopu na ktorý vedie rodičovské pravidlo. To je z dôvodu, že časť transakcií vyriešená rodičovskými pravidlami, môže byť vyriešená novým alebo aktualizovaným pravidlom.

`diff_shop` Tabuľka ukladá `id` shopov, ktoré vzišli z rozdielu tabuliek shopov. Popri `id` shopu ukladá aj `account_id`, ktorý ak obsahuje hodnotu null označuje, že invalidácie sú určené pre všetkých klientov. Naopak ak hodnota nie je null, takéto invalidácie sú určené len pre špecifického klienta s daným identifikátorom.

`big_shop` Tabuľka ukladá všetky shopy, ktoré sú označené ako big shopy.

Vrámci TapiX aplikácie vznikla tabuľka `external_invalidation`, tá slúži na ukladanie stavu získaných invalidovaných `uids` z mikroslužby. Najdôležitejšie sú stĺpce `from_id` a `to_id`, ktoré vyjadrujú rozsah identifikátorov invalidovaných `uids` získaných počas aktualizácie. Keďže nad týmito identifikátormi je

vytvorená rastúca sekvencia, pri ďalšej aktualizácií sa pre získanie ďalších `uids` použije posledné `to_id` + 1.

Kapitola 4

Implementácia

Táto kapitola popisuje proces implementácie vyhľadávania pravidiel v MongoDB, vytvorenie Kubernetes clustra pomocou AWS EKS služby a vytvorenie MongoDB Sharded Clustra v Kubernetes. Ďalej sú rozobraté scenáre ako je napríklad škálovanie a zlyhanie komponenty. V poslednej časti je popísaný proces implementácie TapixVsUpdater mikroslužby a jej nasadenie.

4.1 Vyhľadávanie pravidiel v MongoDB

Aby TapiX aplikácia podporovala prácu s MongoDB boli do projektu pridané potrebné závislosti. V aplikácií bola vytvorená DAO vrstva pre prácu s MongoDB dátovým zdrojom. Implementovaná bola abstraktná trieda, ktorá obsahuje meno kolekcie ku ktorej bude pristupovať a `MongoCollection` objekt pre prístup k nej. Táto trieda poskytuje základnú funkcionalitu pre volanie jednotlivých dotazov. Jednu z hlavných metód je možné vidieť v ukážke 4.1. `BaseMongoDao` je dedená triedami pre prístup ku konkrétnym kolekciám. Tie

```
1  public <R> List<R> queryForList(
2                     Function<MongoCollection<Document>,
3                     FindIterable<Document>> collectionFunction,
4                     Function<Document, R> extractResult) {
5
6      FindIterable<Document> docs = collectionFunction.apply(collection);
7      List<R> result = new ArrayList<>();
8      for (Document doc : docs)
9          result.add(extractResult.apply(doc));
10     return result;
11 }
```

Výpis kódu 4.1: Metóda v triede BaseMongoDao pre získanie zoznamu výsledkov

implementujú abstraktné metódy ako napríklad `extractResult(...)` pre serializáciu dát do instancií tried. Nad kolekciou pravidiel boli vytvorené indexy po vzoru súčasného riešenia. Pre overenie funkčnosti týchto indexov boli vy-

skúšané všetky typy dotazov, teda dotazy, kde pre každé pole je buď definovaná hodnota alebo null. Tieto dotazy je možné vidieť v priloženom súbore `mongo_queries_test.json` a doba odpovede pre každý z nich bola v rádoch jednotkách milisekúnd. Vytváranie dotazu pre vyhľadávanie v kolekcií pravidiel je vidieť v ukážke 4.2. Dôležité bolo pri vytváraní MongoDB klienta nastaviť parameter *readPreference* na hodnotu *nearest*. Predvolená hodnota je *primary* a znamená, že všetky operácie používajú iba primárnu repliku. Pre rozloženie záťaže je vhodné použiť *nearest*, to zaistí, že pre jednotlivé dotazy sa vyberá náhodne, vhodný člen replica-setu na základe latencie. Po tomto nastavení nie je zaťažená jediná primárna replika, ale záťaž je rozložená medzi všetkých členov. Používanie MongoDB pre vyhľadávanie v pravidlách je zapínateľná funkciona-

```
1  List<TxRule> findAllByRelevantFields(
2                                  String posId, String merchantId,
3                                  String description, String country,
4                                  Integer mcc, String city,
5                                  String zip, Long gtPriority) {
6
7      Map<String, Object> paramMap = initParamMap(posId, merchantId,
8                                              description, country,
9                                              mcc, city, zip);
10
11     Bson filter = buildCNFFindInRulesFilter(paramMap);
12     return queryForList(collection -> collection.find(filter),
13                         this::extractResult);
14 }
15
16 Bson buildCNFFindInRulesFilter(Map<String,Object> nameToValue) {
17     List<Bson> equalOrNull = new ArrayList<>();
18     String nullField = null;
19     for (Iterator<String> keyIt =
20          nameToValue.keySet().iterator(); keyIt.hasNext(); ) {
21
22         String name = keyIt.next();
23         Object value = nameToValue.get(name);
24         if (value != null)
25             equalOrNull.add(Filters.in(name, value, nullField));
26         else
27             equalOrNull.add(Filters.eq(name, nullField));
28     }
29     equalOrNull.add(createFilterForShardKey(nameToValue));
30     return Filters.and(equalOrNull);
31 }
```

Výpis kódu 4.2: Metódy v TxRuleMongoDao pre vytvorenie dotazu na získanie pravidiel

lita. Pri štarte aplikácie, konkrétne pri inicializácií jednotlivých komponent, ak nie je definovaný *hostname* pre pripojenie k MongoDB, tak sa pre vyhľadávanie v pravidlách použije PostgreSQL. Naopak ak je definovaný, aplikácia sa najskôr pokúsi pripojiť k databáze a vykonať *ping* príkaz pre kontrolu funkč-

nosti databázy. Pokiaľ je príkaz úspešný MongoDB bude použité, v opačnom prípade sa použije PostgreSQL.

4.2 MongoDB Sharded Cluster

4.2.1 Vytvorenie Kubernetes clustra v EKS

Pre nasadenie MongoDB Sharded Clustra bolo potrebné vytvoriť Kubernetes cluster s použitím AWS EKS služby. Ten je možné vytvoriť pomocou webovej aplikácie, príkazovej riadky alebo pomocou nástroja `eksctl`. V prvom kroku sa vytvorí cluster, kde sa definuje region, verzia Kubernetes a *subnets*. V ďalšom kroku sa vytvorí *nodegroup*, ktorá definuje uzly, na ktorých budú jednotlivé služby spustené. Pri *nodegroup* je možné definovať typ EC2 instancie, región, počet uzlov a ďalšie vlastnosti. Po vytvorení clustra je potrebné pridať Amazon EBS CSI doplnok, ktorý spravuje životný cyklus úložiska pre Kubernetes. Aby bolo možné monitorovať jednotlivé služby bežiace v Kubernetes je potrebné v clustri spustiť službu Fluent Bit. Tá zabezpečí odosielanie logov a metrík do služby CloudWatch. Aktuálna infraštruktúra clustra pozostáva z troch uzlov typu m5.large. Pri výbere typu instancií uzlov bolo dôležité vyhnúť sa takzvaným *burstable* instanciám. Tie sú menej nákladné, no ponúkajú základný CPU výkon napr. 30%. Ak instancia využíva menej ako tento základný výkon nabíjajú sa kredity, ktoré môžu byť neskôr spotrebované, keď instancia potrebuje využiť viac ako základný výkon. Ak instancia potrebuje využívať vyšší ako základný výkon a nemá dostatok kreditov, obnáša to ďalšie náklady naviac. Uzly v Kubernetes sú neustále zaťažené nad hranicou základného výkonu a teda takéto instancie nedávajú zmysel pre tento prípad použitia.

4.2.2 Vytvorenie MongoDB Sharded Clustra

Pre vytvorenie clustra je potrebné [32]:

1. Vytvoriť config server replica-set. To znamená spustiť jednotlivé mongod procesy s prepínačmi `--configsvr`, `--replSet`, `--bind_ip`.

2. Pripojiť sa na jeden z config serverov pomocou mongosh nástroja.

3. Inicializovať replica-set, inicializácia prebehne spustením nasledujúceho príkazu:

```
rs.initiate({
    _id: "myReplSet",
    configsvr: true,
    members: [
      { _id : 0, host : "cfg1.example.net:27019" },
      { _id : 1, host : "cfg2.example.net:27019" },
      { _id : 2, host : "cfg3.example.net:27019" }
    ]})
```

4. Vytvoriť shard replica-set. Opäť spustiť jednotlivé mongod procesy. Tento krát s prepínačmi `--shardsvr`, `--replSet`, `--bind_ip`.

5. Pripojiť sa znova pomocou mongosh na jeden shard a inicializovať replica-set.

```
rs.initiate({
    _id : "myReplSet",
    members: [
      { _id : 0, host : "s1-mongo1.example.net:27018" },
      { _id : 1, host : "s1-mongo2.example.net:27018" },
      { _id : 2, host : "s1-mongo3.example.net:27018" }
    ]})
```

6. Spustiť mongos proces s prepínačmi `--bind_ip` a `--shardsvr` pri ktorom sa uvedú hostname config serverov.

7. Pripojiť sa na mongos a pomocou nasledujúceho príkazu pridať shardy do clustra.

```
sh.addShard( "myReplSet/s1-mongo1.example.net:27018,
             s1-mongo2.example.net:27018,
             s1-mongo3.example.net:27018")
```

Pre jednotlivé komponenty clustra boli v Kubernetes vytvorené *workloads*. Pre config servery bol vytvorený Statefulset s vlastným úložiskom, pre mongos servery bol vytvorený Deployment a pre shardy bol vytvorený rovnako Statefulset, kde každá replika shardu disponuje vlastným perzistentným úložiskom. Shardy budú vytvárať na infraštruktúre najvyššiu záťaž, preto bolo definované `podAntiAffinity` pravidlo medzi replikami jedného shardu aj medzi replikami rôznych shardov. To zabezpečuje, že plánovač, ktorý rozhoduje o tom na aké uzly budú pody umiestnené, bude preferovať rozloženia, kde niesu pody shardov na rovnakom výpočetnom uzle. Konfigurácia Statefulsetu pre jeden shard je vidieť v ukážke 4.3. Pre kompaktnosť ukážky sú z nej vynechané definície `readinessProbe` a `livenessProbe`.

4.3 Odolnosť a škálovateľnosť riešenia

Aktuálne je každá komponenta MongoDB clustra nasadená v replica-sete s troma replikami. Keďže Kubernetes cluster disponuje troma uzlami, každá replika ľubovoľnej komponenty môže byť spustená na jednom z uzlov. To zabezpečuje vysokú dostupnosť služby. Pre dnešné nároky na výkon služby bol zvolený počet shardov stanovený na 2. Vďaka `podAntiAffinity` nastaveniu pri shard komponentách je dôležité, že jednotlivé repliky shardov sú rozdelené medzi jednotlivé uzly. Teda na každom uzle bude spustená jedna replika z prvého shardu a jedna replika z druhého shardu. Keďže jednotlivé dotazy sú rozposielané na všetky repliky, tak pri potrebe čo najrýchlejšie navýšiť výkon služby, stačí pridať do Kubernetes ďalšie uzly a previesť postupný reštart replík shardov. Nastavenie `podAntiAffinity` zabezpečí, že niektoré repliky budú nasadené aj medzi pridané uzly.

Nelimitujúci typ škálovania je pridanie ďalšieho shardu do MongoDB clustra. Bol otestovaný scenár s pridaním ďalšieho shardu. Pridanie je možné uskutočniť bez výpadku služby a hneď po pridaní shardu je spustené balancovanie dát a presun na novo pridaný shard. Po pridaní druhého shardu k existujúcemu, ktorý obsahoval dva milióny dokumentov môžme vidieť výslednú distribúciu pomocou `getShardDistribution()` príkazu.


```
[{
    "Shard shard1rs at shard1rs/192.168.129.198:50001": {
      "data": "638.06MiB",
      "docs": 2000000,
      "chunks": 2,
      "estimated data per chunk": "319.03MiB",
      "estimated docs per chunk": 1000000
    },
    "Shard shard2rs at shard2rs/192.168.129.198:50004": {
      "data": "288.8MiB",
      "docs": 904149,
      "chunks": 9,
      "estimated data per chunk": "32.08MiB",
      "estimated docs per chunk": 100461
    },
    "Totals": {
      "data": "926.87MiB",
      "docs": 2904149,
      "chunks": "092",
      "Shard shard2rs": ["31.15 % data", "31.13 % docs in cluster",
                        "334B avg obj size on shard"],
      "Shard shard1rs": ["68.84 % data", "68.86 % docs in cluster",
                        "334B avg obj size on shard"]
    }}]
```


Ako je možné vidieť distribúcia nie je veľmi rovnomerná. Rovnako je možné si všimnúť, že celkový počet dokumentov narástol na hodnotu 2904149. To je z toho dôvodu, že pri presúvaní dát sa na prvom sharde dokumenty len označia ako zmazané a vzniknú takzvané *orphaned* dokumenty. Tie budú zmazané v blízkej budúcnosti a distribúcia dát bude rovnomerná [39].

Ďalej bol otestovaný scenár so zlyhaním primárnej alebo sekundárnej repliky komponenty. Zlyhanie repliky bolo simulované pomocou príkazu `kubectl exec pod-name -- kill 1`, ktorý ukončí bežiaci proces v pode. Počas toho boli na službu posielané požiadavky a neboli zaznamenané žiadne výpadky.

Rovnakým spôsobom prebiehalo aj otestovanie scenára pri zlyhaní celého uzla. Ten bol manuálne terminovaný a počas toho neboli zaznamenané žiadne výpadky služby.

4.4 TapixVsUpdater mikroslužba

4.4.1 Správa virtuálnych shopov

Použitie tejto mikroslužby a streamovanie virtuálnych shopov do Kinesis namiesto ukladania do databáze musí byť zapínateľnou funkcionalitou, pretože interné instancie TapiX aplikácie nebudú túto funkcionalitu používať. Preto bolo

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongodbshard1stateful
spec:
  selector:
    matchLabels:
      role: mongoshard1
  serviceName: mongodbshard1service
  replicas: 3
  template:
    metadata:
      labels:
        role: mongoshard1
        replicaset: mongoshards1
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 100
              podAffinityTerm:
                labelSelector:
                  matchExpressions:
                    - key: replicaset
                      operator: In
                      values:
                        - mongoshards1
                topologyKey: kubernetes.io/hostname
            - weight: 50
              podAffinityTerm:
                labelSelector:
                  matchExpressions:
                    - key: replicaset
                      operator: In
                      values:
                        - mongoshards2
                topologyKey: kubernetes.io/hostname
      terminationGracePeriodSeconds: 10
      containers:
        - name: mongodshard1container
          image: mongo:6.0.9
          command:
            - "mongod"
            - "--shardsvr"
            - "--replSet"
            - "mongoreplicaset1shard1"
            - "--bind_ip_all"
            - "--port"
            - "27017"
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongodbshard1pvc
              mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: mongodbshard1pvc
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: gp2
        resources:
          requests:
            storage: 20Gi
```

Výpis kódu 4.3: Statefulset konfigurácia pre jeden shard

implementované spoločné rozhranie, ktoré implementujú oba prístupy správy virtuálnych shopov. Streamovanie virtuálnych shopov do kinesis začína tým,

že vytvorené virtuálne shopy sa zapíšu do súboru. Zápis do súboru zabezpečuje osobitný logger definovaný v konfiguračného súboru `logback-spring.xml`, pomocou ktorého sa zapisujú virtuálne shopy do súboru vo formáte `json`. Streamovanie vlastných súborov do služby CloudWatch nie je priamo podporované službou Beanstalk. Preto pre prenesenie týchto súborov je potrebné vytvoriť adresár `.ebextensions`, v ktorom je možné definovať konfiguračné súbory zabezpečujúce nastavenie alebo upravenie prostredia a AWS zdrojov, ktoré obsahuje. V tomto prípade bude nainštalovaný CloudWatch Logs agent na EC2 instancie, kde je Beanstalkom spustená aplikácia. A v neposlednom rade sú nastavené súbory, ktoré má agent do CloudWatch preniesť. Zo služby CloudWatch sa dáta do Kinesis prenášajú nastavením *subscription* filtra pre konkrétne dáta. Aby bolo možné tento *subscription* vytvoriť je nutné vytvoriť IAM rolu, ktorá udelí CloudWatch oprávnenie vkladať dáta do Kinesis streamu.

O konzumáciu dát z Kinesis streamu sa stará knižnica AWS Kinesis Binder, tá predstavuje implementáciu *binder* komponenty, ktorá podporuje prácu s AWS Kinesis. Komunikácia využíva *publish-subscribe* model, pomocou ktorého sú aplikácie prepojené prostredníctvom zdieľaných témat. Dáta zo streamu môžu byť konzumované viacerými aplikáciami. To zabezpečujú takzvané *consumer groups*, ktoré sú identifikované špecifickým menom. V skupine s rovnakým menom môže existovať viacero konzumentov. Všetky rôzne skupiny, majúce nastavený *subscription* nejakej destinácie dostanú kópiu zverejnených dát. Ale iba jediný člen každej skupiny dáta dostane. Viacero instancií v jednej skupine zaručuje vysokú dostupnosť, pretože ak nejaká instancia v skupine zlyhá, ďalšia začne spracovávať dáta od posledného kontrolného bodu pri ktorom instancia zlyhala. Pri škálovaní je možné nastaviť konfiguračné parametre `instanceCount` a `instanceIndex` pre instancie konzumentov rovnakej skupiny. To zaručí že dáta sú rovnomerne distribuované medzi konzumentov v tejto jednej skupine. Pre súčasné potreby je dostačujúca jediná instancia TapixVsUpdater konzumenta. Nie je potrebné ani viacero instancií v rôznych skupinách pre vysokú dostupnosť, pretože dáta v Kinesis sú uložené po dobu 7 dní. Teda výpadok služby by musel byť dlhší ako 7 dní, aby došlo k strate dát, čo je veľmi nepravdepodobné.

Konfigurácia integrácie s Kinesis pre TapixVsUpdater vyzerá nasledovne, pre kompaktnosť bola predpona `spring.cloud.stream` v niektorých prípadoch vynechaná:

```
# nazov Kinesis streamu
...bindings.consumeTxVirtualShops-in-0.destination=tapix_virtual_shops
# nazov consumer skupiny
...bindings.consumeTxVirtualShops-in-0.group=tapix_vs_updater
...bindings.consumeTxVirtualShops-in-0.content-type=application/json
...bindings.consumeTxVirtualShops-in-0.consumer.batch-mode=true

# stream uz bol vytvoreny, nie je ziaduce aby sa vytvaral automaticky
spring.cloud.stream.kinesis.binder.autoCreateStream=false
# Konfiguracia poctu konzumentov
spring.cloud.stream.instanceCount=1
# Konfiguracia indexu konzumenta
spring.cloud.stream.instanceIndex=0
```

Stačí, že aplikácia obsahuje závislosti na module spring cloud stream spolu

s *binder* závislosťami a vďaka automatickej konfigurácií stačí definovať Bean entitu typu `Supplier`, `Function` alebo `Consumer` a tá bude automaticky registrovaná ako konzument dát a spúšťaná pri príchode nových dát. Funkciu slúžiacu v mikroslužbe TapixVsUpdater ako konzument dát je možné vidieť v ukážke 4.4. Prichádzajúce dáta je potrebné pred spracovaním dekomprimovať. Metriky ktoré sa sledujú sú prichádzajúci počet virtuálnych shopov a čas strávený spracovaním jedného batch požiadavku. Tieto metriky sú potom pomocou `CloudWatchAsyncClient` odosielané do služby CloudWatch. Odtiaľ sa potom tieto metriky vizualizujú v nástroji Grafana.

```
1  @Bean
2  public Consumer<Message<byte[]>> consumeTxVirtualShops() {
3      return txVirtualShops -> {
4          consumeTxVirtualShopFromCW(txVirtualShops.getPayload());
5      };
6  }
7
8  public void consumeTxVirtualShopFromCW(byte[] payload) {
9      long startTime = System.currentTimeMillis();
10
11     Optional<byte[]> decompressedPayload =
12                               StreamUtils.decompressGzip(payload);
13
14     String jsonRecord = new String(decompressedPayload.orElseThrow());
15
16     Optional<Map<String, Object>> jsonMap =
17                           StreamUtils.jsonStringToMap(jsonRecord);
18
19     List<String> virtualShopsAsJson =
20               retrieveVirtualShopsDataAsJsons(jsonMap.orElseThrow());
21
22     List<TxVirtualShopDTO> streamedVirtualShops =
23                        virtualShopsAsJson.stream().map(vsJson -> {
24         Optional<TxVirtualShopDTO> vs =
25         StreamUtils.jsonStringToObject(vsJson, TxVirtualShopDTO.class);
26         vs.orElseThrow().setProcessed(false);
27         return vs.get();
28     }).toList();
29
30     insertTxVirtualShopsAndLinks(streamedVirtualShops);
31     updateMetricsAndLog(streamedVirtualShops,
32                         System.currentTimeMillis() - startTime);
33 }
```

Výpis kódu 4.4: Funkcia pre konzumáciu dát z Kinesis

4.4.2 Proces invalidácie

Proces invalidácie v mikroslužbe začína vytvorením balíčka v TapiX aplikácií. Pre komunikáciu s TapixVsUpdater bola použitá trieda `RestTemplate`.

Celkový proces invalidácie funguje podobne ako v pôvodnom riešení. Proces prešiel istým refaktoringom a líši sa od pôvodného riešenia tým, že invalidované virtuálne shopy a ich linky sú počas invalidácie mazané. Počas *polling* fázy sa vracia stav spolu so správou o aktuálnom postupe procesu. Táto správa je priebežne aktualizovaná, aby odrážala čo najpresnejšie postup, v ktorom sa proces nachádza.

Proces čiastočnej invalidácie

Rozdiel čiastočnej invalidácie oproti kompletnej je vo vyhľadávaní virtuálnych shopov, ktoré majú byť invalidované. Čiastočná invalidácia používa rovnakú triedu ako proces invalidácie, ale implementuje vlastné metódy, ktoré vyhľadávajú virtuálne shopy. Pri čiastočnej invalidácií sa k vyhľadávaniu virtuálnych shopov pridávajú podmienky na verziu schémy a príznaku, či už bol virtuálny shop niekedy spracovaný. Cieľom čiastočnej invalidácie je invalidovať virtuálne shopy, ktoré vznikli počas procesu invalidácie, oproti správnemu rozdielu dát. Čiastočná invalidácia je spúšťaná automaticky každých 5 hodín. Za spúšťanie je zodpovedná metóda, ktorú je možné vidieť v ukážke 4.5. Tá najskôr vyhľadá všetky verzie schém ku ktorým existuje virtuálny shop s príznakom `processed` nastaveným na *false*. K týmto verziám vyhľadá mená všetkých schém, v ktorých je uložený rozdiel dát. Čiastočná invalidácia je potom spustená pre každý takýto rozdiel dát a vždy sa invalidujú len virtuálne shopy s odpovedajúcou verziou schémy. Aby sa zabránilo duplicitným invalidáciam bol použitý `ReentrantLock`, ktorý zabraňuje spusteniu čiastočnej invalidácie, keď bol spustený proces invalidácie všetkých virtuálnych shopov.

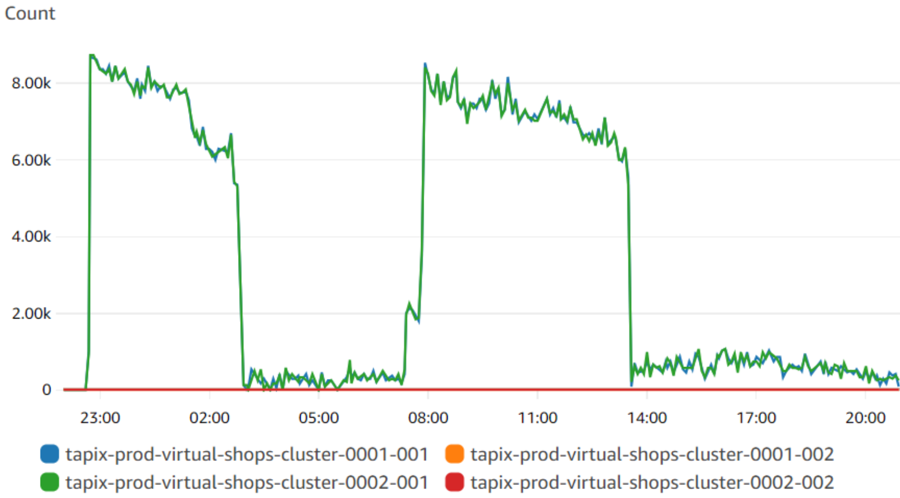
```
1  @Scheduled(cron = "${scheduled.partialInvCron:}", zone = TIME_ZONE)
2  public void scheduledPartialInvalidation() {
3      log.info("Scheduled partial invalidation started...");
4      List<String> vsSchemaVersions =
5      txVirtualShopDao.findVsSchemaVersionsWithUnprocessedVirtualShops();
6
7      List<String> schemaNames =
8      controlDao.findSchemaDiffNameByVsSchemaVersions(vsSchemaVersions);
9
10     schemaNames
11     .forEach(virtualShopInvalidator::partialVirtualShopsInvalidation);
12
13     log.info("Scheduled partial invalidation finished...");
14 }
```

Výpis kódu 4.5: Funkcia pre konzumáciu dát z Kinesis

Podobne ako je spúšťaná aj čiastočná invalidácia, je spúšťané mazanie schém s rozdielom dát. Zmazané sú schémy s takou verziou, ku ktorým neexistuje virtuálny shop, ktorý by mal nastavený príznak `processed` na hodnotu *false*.

4.4.3 Nasadenie mikroslužby a databázy v EKS

Pred nasadením aplikácie s databázou bolo potrebné vytvoriť jednotlivé služby vrámci AWS. Ako prvý bol vytvorený Kinesis stream. Aby aplikácia v službe Beanstalk mohla streamovať vlastné súbory do služby CloudWatch, potrebuje *CloudWatchLogsFullAccess* oprávnenie. Kinesis stream bude obsahovať jeden shard, čo bude dostatočné pre odhadovaný počet vznikajúcich virtuálnych shopov. Dĺžka počas ktorej bude Kinesis uchovávať dáta je nastavená na 7 dní. Aby mohli byť virtuálne shopy posielané zo služby CloudWatch do Kinesis bude nutné vytvoriť rolu s potrebnými oprávneniami, ktorú bude potom CloudWatch používať pre vkladanie dát do Kinesis. Ako druhá služba bol vytvorený Redis cluster. Na základe odhadovaného počtu vznikajúcich virtuálnych shopov, boli zvolené dve instancie typu *t2.micro*, ktorých veľkosť pamäte je 500MB. Pre obe instancie boli ešte vytvorené repliky pre zabezpečenie vysokej dostupnosti. Po nasadení mikroslužby a celkového riešenia na produkciu bolo zrejmé z grafu *Evictions*, ktorý je možné vidieť na obrázku 4.1, že veľkosť cache pamäte nie je dostatočná. Graf ukazuje počet položiek v pamäti, ktorým ešte nevypršala platnosť, ale boli vymazané z pamäti, aby sa uvoľnilo miesto pre zápis nových položiek. Na grafe je vidieť, že pri vyššej záťaži je cache nedostatočne veľká, preto boli instancie vymenené za typ *t2.small*, ktorý už je dostatočný.



Obr. 4.1: Vymazávanie platných položiek z Redis cache pre nedostatok miesta

Keďže už pri implementácií MongoDB Sharded Clustra bol vytvorený Kubernetes cluster, bude v ňom nasadená aj Mikroslužba TapixVsUpdater spolu s PotgreSQL databázou. Pretože aplikácia TapiX a súvisiace komponenty smerujú k riešeniu, že všetky jej komponenty budú nasadené v Kubernetes. Databáza bude *workload* typu Statefulset a aplikácia typu Deployment. Pre nastavenie premenných prostredia boli použité Kubernetes objekty ConfigMap a Secret. TapixVsUpdater pristupuje k AWS službe Kinesis a keď pod potrebuje pristúpiť k nejakej AWS službe, jedna z možností je použiť *service account* vytvorený v Kubernetes. *Service account* poskytuje identitu pre procesy bežiace

v pode. Preto bolo potrebné nakonfigurovať *service account* so správnou rolou a právami pre prístup ku Kinesis streamu. Pody využívajúce tento *service account* máju potom právo vykonávať definované operácie ohľadom nakonfigurovaného streamu.

Nasadzovanie novej verzie aplikácie do Kubernetes bolo automatizované s použitím nástroja Jenkins. Aby bolo možné využívať AWS príkazovú riadku, je nutné nainštalovať AWS CLI. Rovnako pre ovládanie Kubernetes clustra je potrebné nainštalovať nástroj `kubectl`. Aby nebola potrebná inštalácia všetkých zmienených nástrojov na servery spúšťajúce nasadenie, bol využitý kontajner `guitarrapc/docker-awscli-kubectl`, ktorý má mimo iné nástroje nainštalované aj AWS CLI a `kubectl`. Pre prihlásenie do účtu AWS je ku kontajneru pripojený `.aws` adresár, ktorý obsahuje súbor `credentials` s prístupovými údajmi k AWS účtu. Nasadenie prebieha v niekoľkých krokoch:

1. Spustia sa testy aplikácie, ak sú úspešné, pokračuje sa v ďalšom kroku.
2. Zostaví sa projekt z požadovanej revízie.
3. Nová verzia obrazu TapixVsUpdater sa nasadí do ECR.

```
# run container
docker run --rm -v ./.aws:/root/.aws \
guitarrapc/docker-awscli-kubectl \

# login to AWS ECR
aws ecr get-login-password --region eu-central-1 | \
docker login -u AWS --password-stdin <ecr-url>

# build image
docker build -t tapix-vs-updater -f docker/Dockerfile .

# pushing new version to ecr
docker tag tapix-vs-updater <ecr-url>/tapix-vs-updater:${TAG}
docker push <ecr-url>/tapix-vs-updater:${TAG}
```

4. Nová verzia aplikácie sa nasadí do Kubernetes clustra v EKS. Aplikujú sa prípadné zmeny v Deployment konfigurácií a prevedie sa reštart Deploymentu. Keďže je v konfigurácií nastavený parameter `imagePullPolicy: Always`, tak každý reštart spôsobí, že sa stiahne najaktuálnejšia verzia obrazu aplikácie.

```
# include volume dep with deployment.yaml configuration
docker run --rm -v ~/.aws:/root/.aws -v ./dep:/opt/dep \
guitarrapc/docker-awscli-kubectl /bin/bash -c \

# set correct cluster for kubectl command
"aws eks update-kubeconfig --region eu-central-1 \
--name <eks-cluster-name>; \

# apply newest version of configuration file
kubectl apply -f /opt/dep/deployment.yaml; \
```

```
# restart deployment
kubectl rollout restart deployment tapixvsupdater"
```

Kapitola 5

Testovanie

Kapitola sa bude zaoberať testovaním funkčnosti vzniknutého kódu a testovaním výkonu prepracovanej architektúry. V prvej časti sa kapitola zaoberá testovaním výkonu TapiX aplikácie s použitím MongoDB v kontraste s pôvodným riešením. Druhá časť sa venuje porovnaniu výkonu aplikácie s databázovým spracovaním virtuálnych shopov verzus s asynchronným spracovaním. V závere kapitoly bude zhodnotenie dosiahnutých výsledkov a splnenie požiadavkov.

5.1 Testovanie vyhľadávania pravidiel v MongoDB

Pre testovanie správnosti vyhľadávania v pravidlách boli použité unit a integračné testy. Unit testy slúžia pre overenie funkčnosti malých častí zdrojového kódu, ktoré môžu byť logicky izolované v systéme. Integračné testy na druhú stranu testujú jednotlivé komponenty kombinované spolu. Cieľom je odhaliť potenciálne problémy v interakciách medzi jednotlivými časťami. Počas implementácie bol kód neustále testovaný pomocou Jenkins jobu, ktorý automaticky pre každý *commit* spúšťal testy.

Pre vytvorenie integračných databázových testov vyhľadávania pravidiel v MongoDB bol použitý framework Testcontainers. Ten narozdiel od rôznych vstavaných riešení ponúka úplnú službu s ktorou test potrebuje interagovať a spustí ju v samostatnom kontajneri. V tomto prípade to bude kontajner v ktorom bude spustená MongoDB databáza. Samostatný kontajner sa bude vytvárať pre každú test triedu a nie pre každý test aby sa ušetril zbytočný čas strávený vytvorením kontajnera. Keďže všetky testy vrámci jednej triedy budú používať rovnaký kontajner, tak pred začiatkom každého testu bude MongoDB databáza znova inicializovaná potrebnými dátami, aby každý test mohol používať ľubovoľné operácie nad databázou. Naviac boli prepoužité existujúce testy transakčných endpointov, ktoré testujú celkovú funkčnosť aplikácie. Pri ich prepoužití sa používa pri vyhľadávaní pravidiel MongoDB namiesto H2 databázy.

5.2 Testovanie výkonnosti MongoDB Sharded Clustra

Pri testovaní boli sledované nasledujúce metriky:

- priemerná doba odpovede aplikácie na požiadavok
- priemerná doba vyhľadania pravidiel pre daný požiadavok v databáze
- počet spracovaných požiadaviek za časový interval jedna minúta

Testovanie prebiehalo na duplikovanom produkčnom prostredí, teda aplikácia bola spustená v službe Beanstalk s rovnakou konfiguráciou ako na produkcií a bol vytvorený nový Amazon Aurora cluster z produkčnej zálohy. Ako testovacie požiadavky bolo použitých niekoľko stoviek tisíc rôznych požiadaviek, ktoré boli niekedy odoslané na produkčnú aplikáciu. Keďže požiadavky sa po istom čase začnú opakovať a test sa zameriava hlavne na výkon vyhľadávania v pravidlách, tak počas testovania nebude použitá Redis cache pre ukladanie výsledkov vyhľadávania pravidiel. Testovaná bola len implementácia vyhľadávania v MongoDB clustri, virtuálne shopy sa počas testu spracovávali rovnako ako v pôvodnom riešení, teda s použitím databázy. Vďaka opakujúcim sa požiadavkám boli virtuálne shopy už vytvorené a preto táto slabina neovplyvňovala výkon aplikácie počas testu. MongoDB cluster, ktorý bol podrobený testovaniu bol nasadený v Kubernetes. Cluster disponuje dvoma shardmi, aby bola otestovaná konfigurácia, s ktorou sa bude začínať. Podrobnejšie otestované boli aj ďalšie konfigurácie s jedným, troma a piatimi shardmi. No pri aktuálnej veľkosti dát sa neprejavilo výrazne zrýchlenie a benefity pri použití väčšieho počtu shardov.

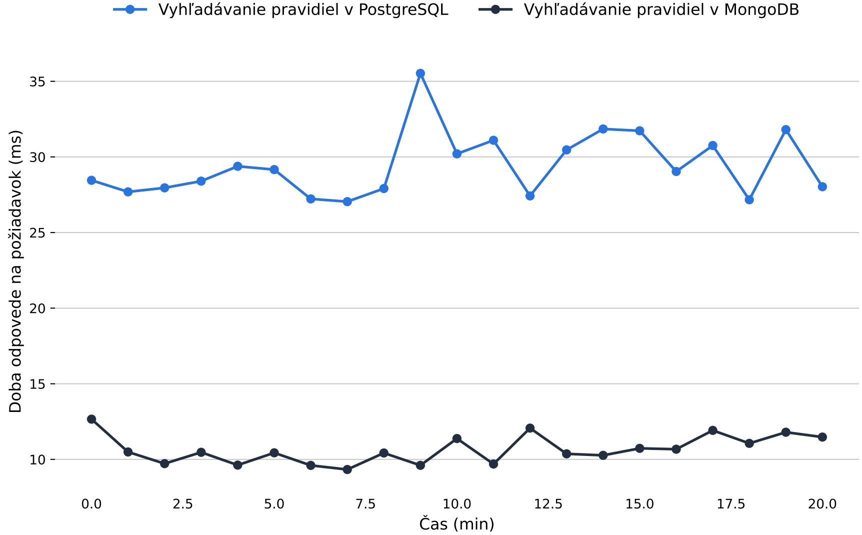
Pri testovaní bol použitý python skript, ktorý odosiela zmienené požiadavky na TapiX aplikáciu vo viacerých vláknach. Tento skript bol spúšťaný paralelne v niekoľkých instanciách z rôznych serverov pre vygenerovanie čo najvyššej záťaže. Počas testovania sa postupne navyšoval počet požiadaviek, ktoré boli posielané na službu. Počet posielaných požiadaviek závisel na počte instancií skriptu, ktoré odosielali na službu požiadavky. Testovalo sa postupne pre jednu, tri a päť instancií záťažového skriptu. Merané dáta boli získané z logov aplikácie pomocou služby *CloudWatch Logs Insights*, vďaka ktorej je možné extrahovať dáta z logov. Pre každý požiadavok na službu sa zalogujú potrebné informácie a požadované hodnoty, ako je napríklad doba vyhľadania pravidiel v databáze. Výsledok nasledujúceho dotazu sú priemerné časy odpovede databázy po jednej minúte extrahované z logov aplikácie.

```
fields @timestamp, @message
| filter @message like /Search time/
| parse @message "Search time: (*ms)" as @rTime
| stats avg(@rTime) by bin(1m)
```

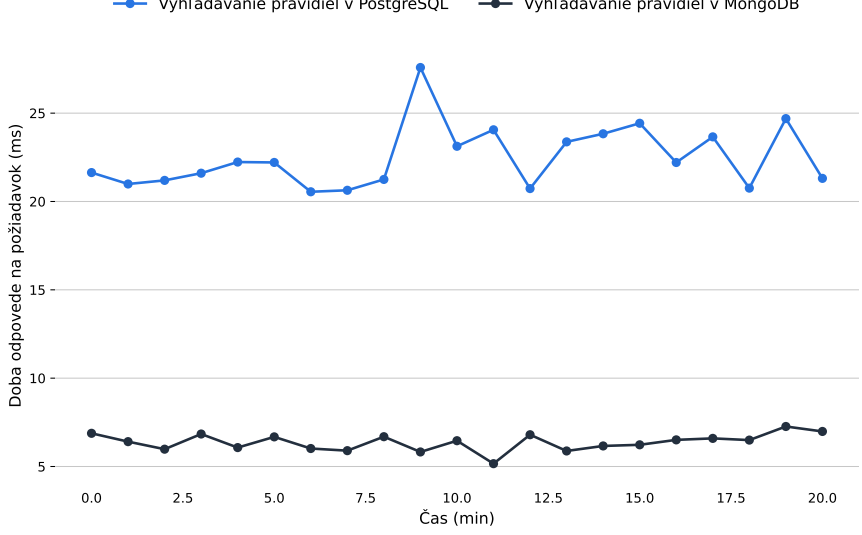
Na záver bol testovaný aj prípad, kedy bol počet pravidiel niekoľkonásobne umelo navýšený. Hodnoty polí pravidiel boli náhodne generované reťazce a pomer null hodnôt v poliach odpovedal pomeru v reálnych dátach. Výsledky tohto testovania ale neboli relevantné a boli skreslené, pretože hodnoty v pravidlách boli až príliš unikátne a neodzrkadľovali možné hodnoty v reálnom svete.

V nasledujúcich grafoch je možné vidieť výsledky testovania pôvodného riešenia, porovnané s výsledkami nového riešenia. Pri porovnaní grafov priemernej doby odpovede aplikácie a priemernej doby odpovede databázy je očividné, že aplikácia trávi väčšinu času odpovede na požiadavok vyhľadávaním v databáze. Použitím MongoDB Sharded clustra pre vyhľadávanie v pravidlách sa podarilo

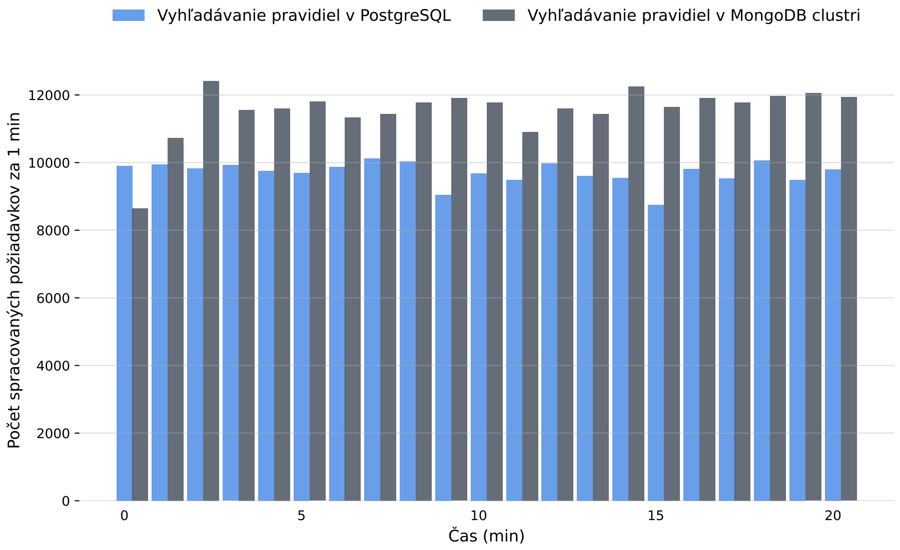
pri vyššej záťaži služby viac ako zdvojnásobiť počet spracovaných požiadaviek za minutú. Rovnako je vidieť, že priemerná doba odpovede na požiadavok je niekoľko násobne rýchlejšia pri použití MongoDB Sharded Clustra ako PostgreSQL.



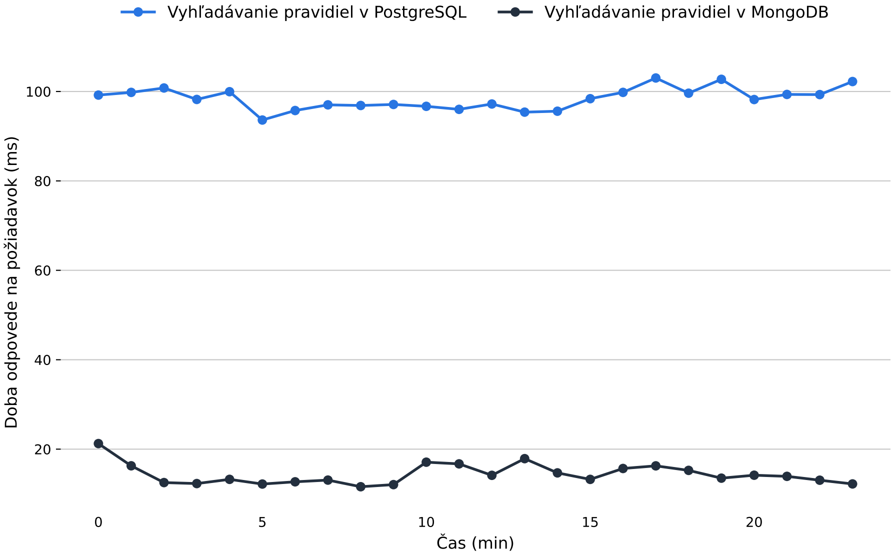
Obr. 5.1: Porovnanie priemernej doby odpovede aplikácie, pri záťaži 1 instancie skriptu



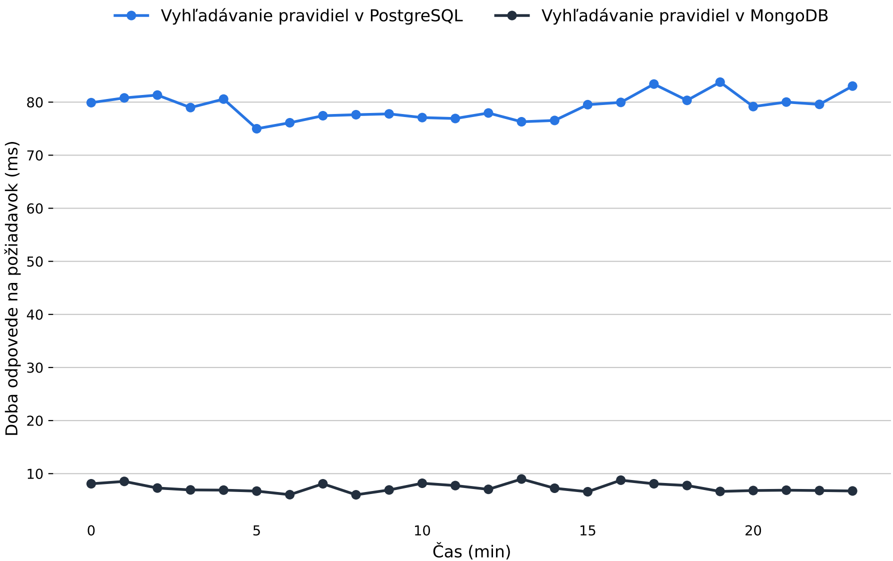
Obr. 5.2: Porovnanie priemernej doby odpovede databázy, pri záťaži 1 instancie skriptu



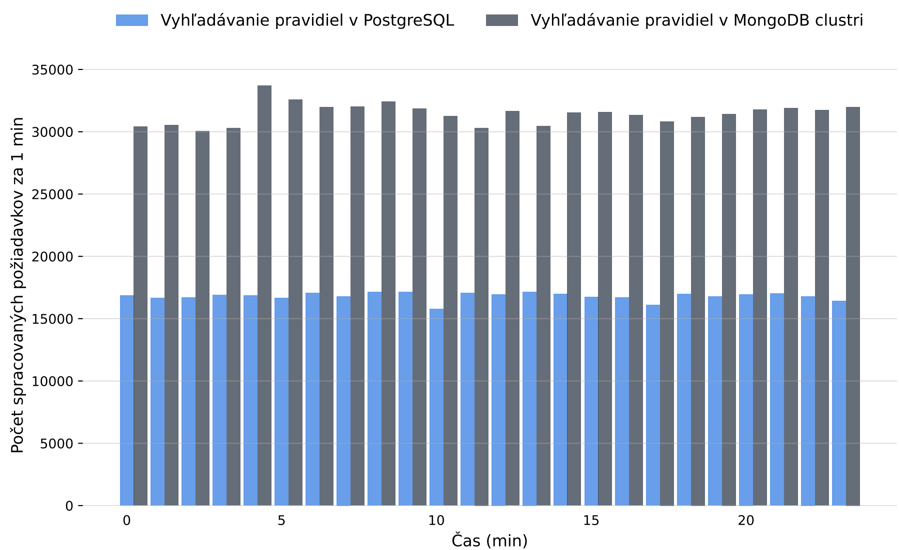
Obr. 5.3: Porovnanie počtu spracovaných požiadaviek, pri záťaži 1 instancie skriptu



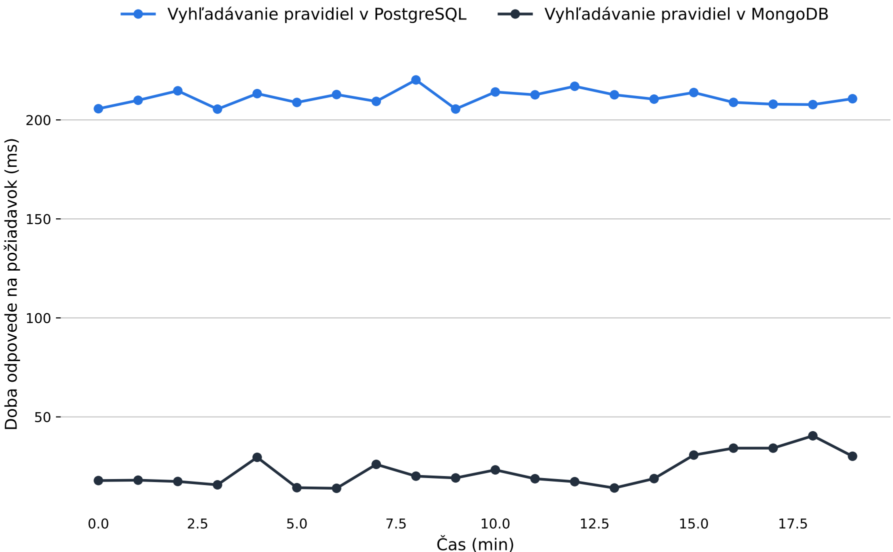
Obr. 5.4: Porovnanie priemernej doby odpovede aplikácie, pri záťaži 3 instancií skriptu



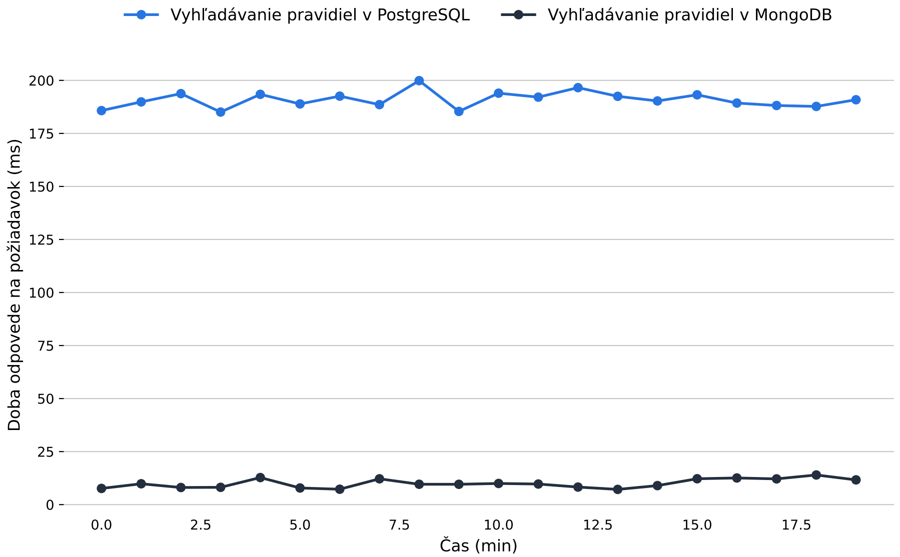
Obr. 5.5: Porovnanie priemernej doby odpovede databázy, pri záťaži 3 instancií skriptu



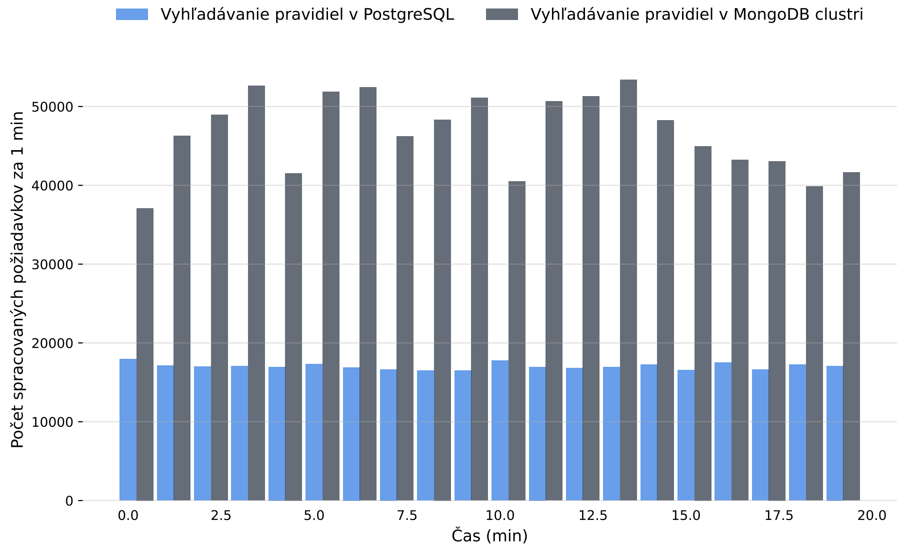
Obr. 5.6: Porovnanie počtu spracovaných požiadaviek, pri záťaži 3 instancií skriptu



Obr. 5.7: Porovnanie priemernej doby odpovede aplikácie, pri záťaži 5 instancií skriptu



Obr. 5.8: Porovnanie priemernej doby odpovede databázy, pri záťaži 5 instancií skriptu



Obr. 5.9: Porovnanie počtu spracovaných požiadaviek, pri záťaži 5 instancií skriptu

5.3 Testovanie funkčnosti mikroslužby

Správna funkčnosť mikroslužby bola otestovaná pomocou unit a integračných testov. Mikroslužba rovnako využíva framework Testcontainers. Každý test, ktorý používa databázu, dedí triedu `TestUsingPostgresqlDBContainer` 5.1. Tá zabezpečuje vytvorenie samostatného kontajnera pre každú test triedu a re-inicializáciu databázy pred každým testom. Aby bolo možné spustiť ďalší kontajner popri bežiacom, ktorý ešte nestihol skončiť, je potrebné, aby každý kontajner použil iný port kam vystaví bežiacu službu. Testcontainers vystaví službu na ľubovoľný voľný port. Jeho hodnotu je možné získať pomocou metódy `getFirstMappedPort()`. Metóda `registerPgProperties(...)` sa postará o správne nastavenie premenných v súbore `application.properties` ešte predtým ako je vytvorený `Datasource`. Ten je potom správne nastavený na databázu v bežiacom kontajneri. Inicializácia databázy pred začiatkom každého testu vytvorí potrebné tabuľky a skopíruje do nich dáta z `csv` súborov.

Nové integračné testy boli vytvorené aj v TapiX aplikácií na otestovanie funkčnosti práce s novou cache pamäťou pri vytváraní virtuálnych shopov. Pre cache sa vytvára Redis kontajner za pomoci Testcontainers. Ďalej boli upravené a pridané testy na proces invalidácie s invalidovaním virtuálnych shopov pomocou externej mikroslužby.

5.4 Testovanie výkonu aplikácie so streamovaným spracovaním virtuálnych shopov

Testovanie malo za cieľ porovnať výkon TapiX aplikácie s databázovým a streamovaným spracovaním virtuálnych shopov pre najhorší možný prípad, kedy

```
@Testcontainers
public abstract class TestUsingPostgresqlDBContainer {
    @Autowired
    private DataSource dataSource;
    @Value("${TEST_SET_UP_PUB_SCHEMA_SQL_FILENAME}")
    private String pubSQL;
    @Value("${TEST_SET_UP_DIFF_SCHEMA_SQL_FILENAME}")
    private String diffSQL;
    @Value("${database.schema.diff}")
    protected String DEFAULT_SCHEMA_NAME;
    @Container
    public static PostgreSQLContainer<?> psql =
    new PostgreSQLContainer<>("postgres:14")
            //mount init data for tests
            .withClasspathResourceMapping("test_data",
                                          "/tmp/sql",
                                          BindMode.READ_ONLY);
    /**
     * Sets up application properties before context is created.
     * @param registry
     */
    @DynamicPropertySource
    static void registerPgProperties(DynamicPropertyRegistry registry){
        registry.add("database.hostname", () -> psql.getHost());
        registry.add("database.port", () -> psql.getFirstMappedPort());
        registry.add("database.username", () -> psql.getUsername());
        registry.add("database.password", () -> psql.getPassword());
        registry.add("database.name", () -> psql.getDatabaseName());
    }
    /**
     * Initializes test database before each test.
     */
    @BeforeEach
    public void setUpDatabase() {
        Resource preparePublicSchema = new ClassPathResource(pubSQL);
        Resource prepareDiffSchema = new ClassPathResource(diffSQL);
        ResourceDatabasePopulator databasePopulator =
                                 new ResourceDatabasePopulator();
        databasePopulator.addScripts(pubSQL, diffSQL);
        databasePopulator.execute(dataSource);
    }
}
```

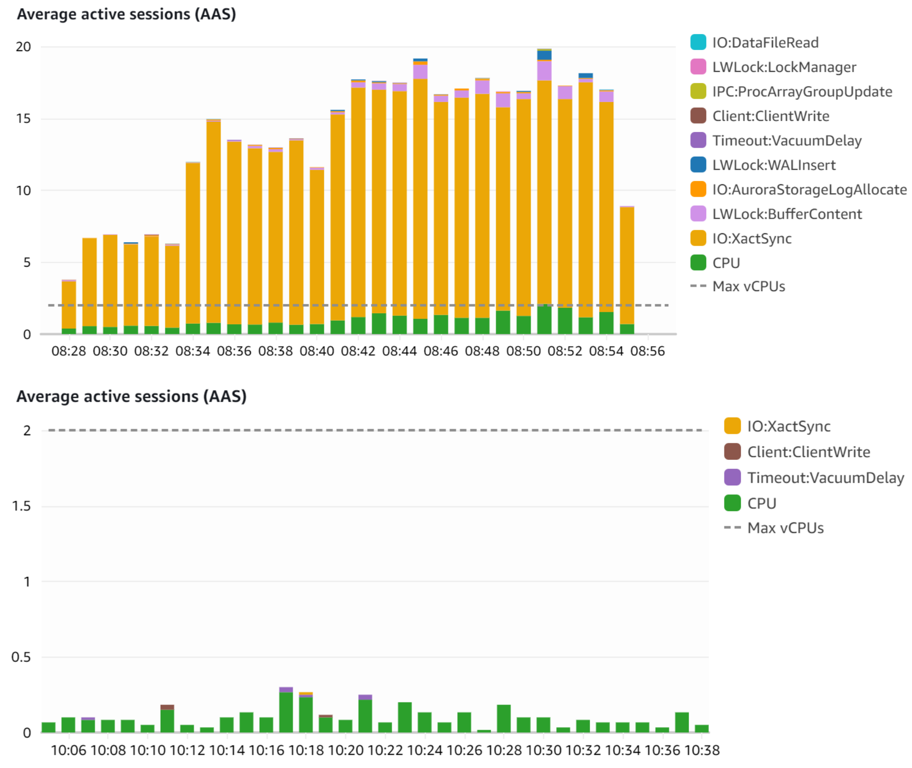
Výpis kódu 5.1: Trieda poskytujúca PostgreSQL kontajner pre testy

je pre každý požiadavok na službu vytvorený virtuálny shop. Pri testovaní TapiX aplikácia používala pre vyhľadávanie pravidiel MongoDB namiesto PostgreSQL, aby výkon PostgreSQL databázy nebol ovplyvnený vyhľadávaním v pravidlách. Požiadavky posielané na službu boli upravené, aby bol každý požiadavok unikátny a bol počas jeho spracovania vytvorený virtuálny shop. V takom prípade cache nedáva zmysel a preto cache pre výsledky vyhľadávania v

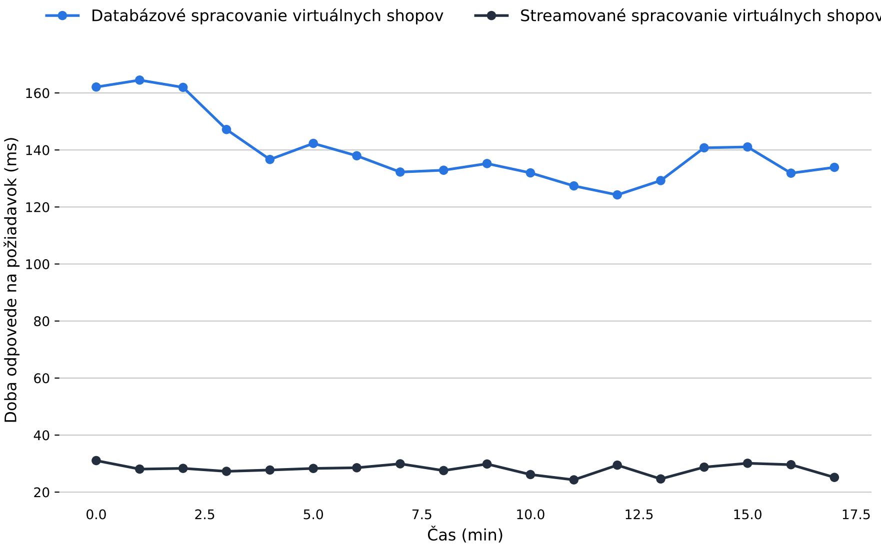
pravidlách aj pre virtuálne shopy nebola používaná. Sledované metriky počas testovania boli:

- priemerná doba odpovede aplikácie na požiadavok
- počet spracovaných požiadaviek za časový interval jedna minúta

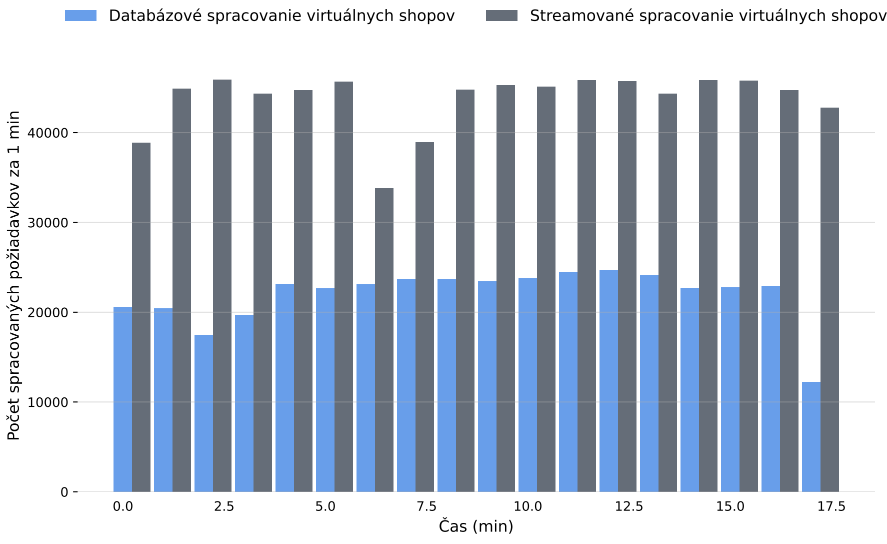
Požiadavky boli posielané rovnakým skriptom ako pri testovaní predchádzajúcom testovaní. V grafoch 5.11 a 5.12 je možné vidieť výsledky testovania pre paralelne bežiacich 5 inštancií skriptu. Upravená architektúra v ktorej sa virtuálne shopy spracovávajú asynchrónne a streamujú sa cez Kinesis do externej mikroslužby dokáže spracovať viac ako dvojnásobok požiadaviek za minútu, ktoré vytvárajú virtuálne shopy a rovnako aj doba odpovede na požiadavok je niekoľko násobne nižšia oproti pôvodnému riešeniu. Na grafoch 5.10 z *RDS Performance Insights* je vidieť zaťaženie databázy. Prvý graf zobrazuje zaťaženie pri databázovom spracovávaní virtuálnych shopov. Je na ňom vidieť zaťažené CPU a `IO:XactSync`, to nastáva pri čakaní na potvrdenie databázovej transakcie. Druhý graf zobrazuje zaťaženie databázy pri streamovanom spracovávaní virtuálnych shopov, pri ktorom je vidieť, že databáza je omnoho menej zaťažená.



Obr. 5.10: Zaťaženie databázy počas testovania pred a po nasadení nového riešenia



Obr. 5.11: Porovnanie priemernej doby odpovede aplikácie, pri záťaži 5 instancií skriptu



Obr. 5.12: Porovnanie počtu spracovaných požiadaviek, pri záťaži 5 instancií skriptu

5.5 Zhodnotenie výsledkov

Výsledky testovania potvrdili očakávania na zvýšenie výkonu aplikácie po nasadení novej architektúry. Boli splnené všetky funkčné aj nefunkčné požiadavky

kladené na výsledné riešenie. Zhrnutie splnenia nefunkčných požiadaviek:

- Služba je schopná obslúžiť niekoľko násobne viac požiadaviek s priemerným časom odozvy do 50 milisekúnd.
- Je možné jednoduché škálovanie vyhľadávania v pravidlách a aj spracovania virtuálnych shopov bez zásahov do architektúry.
- Jednotlivé komponenty MongoDB clustra sú spustené v replica-sete a teda databáza beží v režime vysokej dostupnosti.
- Nová funkcionalita je zapínateľná pomocou premenných prostredia.
- MongoDB podporuje zmenu dátového modelu.
- Všetky nové komponenty sú nasadené v Kubernetes s využitím AWS služby EKS.
- Logy a metriky všetkých nových komponent sú zhromažďované v službe CloudWatch, ktorá zabezpečuje monitoring služieb.
- TapixVsUpdater aplikácia je napísaná v jazyku Java s využitím Spring Boot frameworku.

KAPITOLA 6

Vzniknuté náklady

Kapitola sa zaoberá odhadom pre náklady, ktoré vznikli pri vytvorení novej infraštruktúry a používaním ďalších služieb na AWS.

6.1 Odhad nákladov

Náklady na vytvorené prostriedky vrámci AWS budú odhadnuté pomocou služby *AWS Pricing Calculator*[1], ktorá pre jednotlivé služby, na základe použitia, spočíta odhadované náklady. Pre získanie informácií o spotrebovaných zdrojoch jednotlivých služieb a ďalších informácií o nákladoch bude použitá AWS služba *Billing and Cost Management*. Nasleduje výpis jednotlivých použitých zdrojov a odhad nákladov pre každý z nich. Náklady sú odhadované pre produkčné prostredie.

- **Elastic Kubernetes Service** Náklady pre jeden cluster sú 0,10 USD za hodinu. Výsledná cena za mesiac, teda 730 hodín je 73 USD.
- **EC2** Kubernetes cluster používa ako worker uzly instancie typu *m5.large*. *On-demand* cena týchto instancií je 0,115 USD za hodinu behu instancie. Cluster disponuje tromi takými instanciami. Výsledná cena instancií za mesiac, teda 730 hodín je 251,85 USD.
- **Elastic Block Store** Všetky úložiska v Kubernetes clustri sú typu *gp2*. Pre tento typ 1 GB pamäte na mesiac stojí 0,119 USD. Samotné EC2 instancie disponujú úložiskom o velikosti 100GB. MongoDB cluster používa úložisko o veľkosti 5GB pre config servery a 20GB úložisko pre shardy. Každá replika config servera a shardu disponuje vlastným úložiskom. V Kubernetes beží ešte PostgreSQL databáza s úložiskom o veľkosti 1000GB. Dohromady sa jedná o 1375GB. Výsledná cena za mesiac, teda 730 hodín je 163,63 USD.
- **Elastic Load Balancing** Pre vystavenie mongos serverov z Kubernetes clustra a TapixVsUpdater mikroslužby bol použitý load balancer typu *classic*, jeden pre každú službu. Pre vystavenie PostgreSQL databázy bol použitý load balancer typu *network*. Cena za *classic* load balancer je 0,03

[1] https://calculator.aws/

USD za hodinu a 0,008 USD za jeden spracovaný GB dát. Vďaka Cost Management bolo možné zistiť, že za mesiac bolo spracovaných 190GB dát. Hodinová cena *network* load balancera je 0,027 USD. Odhadovaná cena všetkých load balancerov za mesiac, teda 730 hodín je 66,56 USD.

- **ElastiCache** Bola zvolená cache typu *t2.small*. Využíva sa cluster s dvoma shardmi a dvoma replikami, teda dohromady sa používajú štyri *t2.small* instancie. Cena za hodinu pre jednu instanciu je 0,038 USD. Odhadovaná cena za mesiac, teda 730 hodín je 110,96 USD.

- **DynamoDB** Veľkosť uložených tabuliek je menšia než 1KB, preto bude cena pozostávať len z počtu požiadaviek pre čítanie a zápis. Cena za milión požiadaviek pre čítanie je 0,305 USD a milión požiadaviek pre zápis stojí 1,525 USD. Mesačne sa prevedie zhruba 250 000 požiadaviek pre čítanie a 2 000 000 požiadaviek pre zápis. Odhadovaná cena za mesiac, je 3,09 USD.

- **Elastic Container Registry** V ECR sa ukladá obraz aplikácie TapixVsUpdater s tagom pre produkčné prostredie. Obraz zaberá 360MB pamäte a cena za 1GB na mesiac je 0,1 USD. Odhadovaná cena za mesiac, je 0,04 USD.

- **Kinesis Data Streams** Dve najviac nákladné položky sú cena za jeden shard na hodinu a cena za predĺženú dobu uloženiu dát. Cena za 1 shard na hodinu je 0,018 USD. Cena za predĺženú dobu uloženia dát na hodinu je 0,024 USD. Ďalej sa platí 0,0175 USD za milión požiadaviek pre vloženie dát. Horný odhad pre službu je 100 miliónov vložení za mesiac. Odhadovaná cena za mesiac, teda 730 hodín je 32,41 USD.

- **CloudWatch** V službe sa ukladajú metriky pre monitoring všetkých ostatných služieb ako je napríklad vyťaženie CPU, obsadená pamäť, počet udalostí. Cena jednej metriky je 0,30 USD. Horný odhad na počet všetkých metrík je 150. Spoplatnené je potom množstvo dát ktoré sú prijímané službou CloudWatch a to čiastkou 0,63 USD za jeden GB. Ďalej sa platí za dlhodobé uloženie logov v službe. Cena je 0,0324 za 1 GB na mesiac. Logy sa ale ukladajú komprimované a ich veľkosť po komprimácií sa odhaduje na 15% pôvodnej veľkosti. Horný odhad na veľkosť prijatých dát za mesiac je 90 GB. To zahŕňa logy všetkých vytvorených služieb a virtuálne shopy. Odhadovaná cena za mesiac je 102,14 USD.

Mesačný odhad výslednej ceny všetkých využitých služieb na AWS je 803,68 USD. Táto čiastka predstavuje rozumne vysoké náklady prihliadajúc na benefity, ktoré nové riešenie ponúka. Keďže sa v novom riešení používa pre vyhľadávanie v pravidiel MongoDB, tak sa značná časť záťaže Aurora služby presunula. To má za následok výrazné zníženie nákladov za Aurorua a výsledná cena s novým riešením nebude preto až o toľko vyššia.

S narastajúcou záťažou vyvíjanou na službu bude rásť aj cena pre potreby škálovania. Pri škálovaní budú zvýšenú cenu tvoriť náklady na ďalšie EC2 instancie do Kubernetes clustra, vyššie náklady služby CloudWatch a pri potrebe škálovať spracovanie virtuálnych shopov aj náklady na ďalší stream v službe Kinesis.

Kapitola 7

Možné vylepšenia

Posledná kapitola sa zaoberá vylepšeniami, ktoré môžu znížiť náklady riešenia v AWS, automatizovať manuálne procesy a škálovanie, či zabezpečiť ešte vyššiu robustnosť a vysokú dostupnosť služby.

Automatické pridávanie ďalších shardov

Súčasné riešenie podporuje manuálne pridávanie ďalších shardov pri horizontálnom škálovaní. Pri aktualizácií dát aplikácie by bolo možné zistiť počet nových pravidiel, ktoré budú importované do MongoDB. Na základe počtu by sa automaticky určil počet shardov, ktoré budú použité. V Prvom kroku by sa nasadili novo pridané shardy v replica-sete v Kubernetes a v druhom kroku by sa spustil import dát.

Automatické škálovanie Kubernetes clustra

Súčasné riešenie podporuje manuálne škálovanie Kubernetes clustra, respektíve pridávanie nových worker uzlov, ktoré by navyšovali výpočetnú kapacitu. Pridávanie nových uzlov do clustra by mohlo byť automatické na základe vyťaženosti CPU a pamäte. Automatické škálovanie by znížilo riziko preťaženia služby pri neočakávanom množstve požiadaviek.

Automatické nasadenie infraštruktúry

Bolo by užitočné vytvorenú infraštruktúru na AWS popísať pomocou nástroja ako je napríklad Terraform pre prípad, že by bolo potreba infraštruktúru znova nasadiť, alebo vytvoriť ďalšie prostredie.

Failover vyhľadávania v pravidlách

Súčasné riešenie kvôli spätnej kompatibilite pre interné instancie podporuje vyhľadávanie pravidiel v MongoDB aj v PostgreSQL. Pri výpadku MongoDB Sharded clustra by mohla aplikácia spoznať, že sa jedná o výpadok a začať používať PostgreSQL pre vyhľadávanie v pravidlách. PostgreSQL by používala po dobu výpadku MongoDB clustra.

Zjednotiť load balancre v Kubernetes

Aktuálne je pre každú vystavenú službu v Kubernetes, teda pre mongos, TapixVsUpdater a PostgreSQL samostatný load balancer, ktorý zabezpečuje prístup k službe. Namiesto jednotlivých load balancerov by mohol pre interné služby existovať len jeden a poprípade jeden záložný, ktorý by na základe portu rozposielal požiadavky na odpovedajúce služby. To by zaistilo jednoduchšiu správu komponent v Kubernetes a nižšie náklady.

Záver

Cieľom práce bolo vytvoriť návrh novej architektúry služby TapiX, ktorá odstráni najkritickejšie slabiny pôvodného riešenia a bude jednoducho škálovateľná s rastúcim počtom požiadaviek.

Na začiatok bola nutná analýza súčasného riešenia, počas ktorej boli identifikované dokázateľne slabé miesta a nedostatky pôvodného riešenia. Medzi najkritickejšiu slabinu patrila databáza, ktorá nebola schopná jednoducho škálovať so zvyšujúcou sa záťažou vyvíjanou na službu. S ňou bolo spojených niekoľko nedostatkov, ktoré boli reflektované pri návrhu novej architektúry. Pri jej tvorbe bol použitý ďalší databázový systém, ktorý vďaka svojim vlastnostiam dovoľuje jednoduché škálovanie s rastúcou veľkosťou dát, aj rastúcim počtom požiadaviek na systém. Ďalej bolo jej súčasťou vytvorenie samostatnej mikroslužby pre asynchrónne spracovanie časti požiadavky, ktorá sa pôvodne spracovávala počas odpovede. Architektúra a vybraté technológie spĺňajú požiadavky na jednoduché škálovanie.

Po implementácií, ktorá bola nasadená na testovacie prostredie, boli otestované jednotlivé scenáre škálovania a robustnosti riešenia. Tie potvrdili, že nové riešenie je jednoducho škálovateľné a zabezpečuje vysokú dostupnosť služby. Počas záťažového testovania, zameraného na výkon, bolo simulované produkčné zaťaženie, ktorému bola podrobená pôvodná aj nová architektúra služby. Testovanie prebiehalo v niekoľkých fázach podľa množstva záťaže kladenej na službu. Zatiaľ čo pri najnižšej záťaži bolo pôvodné aj nové riešenie porovnateľné, pri vyššom zaťažení bolo jasne viditeľné, že nová architektúra disponuje niekoľko krát vyšším výkonom, než pôvodná. Výsledky testovania sa potvrdili aj po nasadení služby na produkciu, kde je možné pozorovať, že služba má rýchlejšiu odozvu a výrazne vyššiu kapacitu na počet spracovaných požiadaviek.

V poslednej časti bol vypočítaný odhad nákladov na novo vytvorenú infraštruktúru v AWS. Náklady predstavujú rozumnú čiastku rozpočtu služby a budú akceptovateľné aj pri zvýšení záťaže vyvíjanej na službu a potrebnom škálovaní. Nová architektúra služby je nasadená a využíva sa produkčne.

Bibliografia

1. *ISO 8583* [online]. 2023. [cit. 2023-09-13]. Dostupné z : https://en.wikipedia.org/wiki/ISO_8583.
2. ASSOCIATION FRANÇAISE DE NORMALISATION. *Financial transaction card originated messages — Interchange message specifications.* 3. vyd. Association française de normalisation, 2023.
3. *ISO8583 flows, fields meaning and values* [online]. [cit. 2023-09-13]. Dostupné z : https://www.admfactory.com/iso8583-flows-data-elements-meaning-and-values/#field42.
4. *An overview of HTTP* [online]. [cit. 2023-09-23]. Dostupné z : https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.
5. *HTTPS* [online]. [cit. 2023-09-23]. Dostupné z : https://developer.mozilla.org/en-US/docs/Glossary/HTTPS.
6. H., Massé Mark. In: *REST API design rulebook.* Sebastopol, CA: O'Reilly, 2012, s. 5–5. ISBN 978-1-449-31050-9.
7. *What Is An API (Application Programming Interface)?* [Online]. [cit. 2023-09-15]. Dostupné z : https://aws.amazon.com/what-is/api/.
8. JACOBSON, Daniel; WOODS, Dan; BRAIL, Greg. In: *APIs: a strategy guide.* Sebastopol, CA: O'Reilly, 2012, s. 60–61. ISBN 978-1-449-30892-6.
9. *Ověřování OAuth 2.0 s ID Microsoft Entra* [online]. 2023. [cit. 2023-09-24]. Dostupné z : https://learn.microsoft.com/cs-cz/azure/active-directory/architecture/auth-oauth2.
10. *Features for truly insightful data* [online]. 2023. [cit. 2023-09-14]. Dostupné z : https://tapix.io/features.
11. *Data Enrichment & Analytics from Payment Data: The Key to Optimal Banking App Functionality* [online]. 2023. [cit. 2023-09-14]. Dostupné z : https://tapix.io/resources/industry-insights/data-enrichment-analytics-payment-data-optimal-banking-app-functionality.
12. *Why MCC codes do not help* [online]. 2023. [cit. 2023-09-14]. Dostupné z : https://www.tapix.io/resources/post/why-mcc-codes-do-not-help-much-with-payment-categorization.

13. *Getting started with TapiX* [online]. 2023. [cit. 2023-09-19]. Dostupné z : https://developers.tapix.io/guides/getting-started-with-tapix.
14. *Enrich card payment data* [online]. 2023. [cit. 2023-09-21]. Dostupné z : https://developers.tapix.io/guides/enrich-card-payment-data.
15. *Keep data up-to-date with invalidations* [online]. 2023. [cit. 2023-09-23]. Dostupné z : https://developers.tapix.io/guides/keeping-data-up-to-date#introduction.
16. WALLS, Craig. In: *Spring Boot in action*. Shelter Island: Manning, 2016, s. 2–7. ISBN 9781617292545.
17. *What is Amazon Aurora?* [Online]. 2023. [cit. 2023-09-25]. Dostupné z : https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html.
18. *What is AWS Elastic Beanstalk?* [Online]. 2023. [cit. 2023-09-25]. Dostupné z : https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html.
19. *Introduction to Redis* [online]. 2023. [cit. 2023-09-25]. Dostupné z : https://redis.io/docs/about/.
20. *What is Amazon ElastiCache for Redis?* [Online]. 2023. [cit. 2023-09-25]. Dostupné z : https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html.
21. *What Is Amazon Kinesis Data Streams?* [Online]. 2023. [cit. 2023-09-26]. Dostupné z : https://docs.aws.amazon.com/streams/latest/dev/introduction.html.
22. *What is AWS Lambda?* [Online]. 2023. [cit. 2023-09-28]. Dostupné z : https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.
23. *What is AWS Lambda?* [Online]. 2023. [cit. 2023-09-30]. Dostupné z : https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html.
24. *What is Amazon CloudWatch?* [Online]. 2023. [cit. 2023-09-30]. Dostupné z : https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html.
25. *Maximum CPU* [online]. 2023. [cit. 2023-10-09]. Dostupné z : https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PerfInsights.Overview.MaxCPU.html.
26. *Table Partitioning* [online]. 2023. [cit. 2023-10-20]. Dostupné z : https://www.postgresql.org/docs/current/ddl-partitioning.html.
27. SULLIVAN, Dan. *NoSQL dor MEre Mortals*. 1. vyd. Addison-Wesley Professional, 2015. ISBN 978-0-13-402321-2.
28. SVOBODA, Martina. Basic Principles. In: *Pokročilé databázové systémy (NI-PDB), Přednáška č. 5* [online]. 2020–2021 [cit. 2020-11-10]. Dostupné z : https://www.ksi.mff.cuni.cz/~svoboda/courses/201-MIE-PDB/lectures/MIEPDB16-Lecture-08-Principles.pdf. [Súbor prístupný po prihlásení do siete ČVUT – kópia súboru uložená na priloženej SD karte].

29. *What is DynamoDB?* [Online]. 2023. [cit. 2023-10-23]. Dostupné z : https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

30. *What Is Amazon Neptune?* [Online]. 2023. [cit. 2023-10-23]. Dostupné z : https://docs.aws.amazon.com/neptune/latest/userguide/intro.html.

31. *What is Amazon Keyspaces (for Apache Cassandra)?* [Online]. 2023. [cit. 2023-10-26]. Dostupné z : https://docs.aws.amazon.com/keyspaces/latest/devguide/what-is-keyspaces.html.

32. *What is MongoDB?* [Online]. 2023. [cit. 2023-10-27]. Dostupné z : https://www.mongodb.com/docs/manual/.

33. *Using Amazon DocumentDB elastic clusters* [online]. 2023. [cit. 2023-10-26]. Dostupné z : https://docs.aws.amazon.com/documentdb/latest/developerguide/docdb-using-elastic-clusters.html.

34. ARDELEAN, Viktor. Damn Cool Algorithms, Part 1: BK-Trees. In: [online]. 2022 [cit. 2023-10-25]. Dostupné z : https://www.baeldung.com/cassandra-secondary-indexes.

35. *Cassandra vs MongoDB Comparison* [online]. 2023. [cit. 2023-10-25]. Dostupné z : https://www.mongodb.com/compare/cassandra-vs-mongodb.

36. How would you compare MySQL sharding vs Cassandra vs MongoDB? In: *Quora* [online]. 2012 [cit. 2023-10-25]. Dostupné z : https://www.quora.com/How-would-you-compare-MySQL-sharding-vs-Cassandra-vs-MongoDB.

37. DENNIS, Alan; WIXOM, Barbara Haley; TEGARDEN, David Paul. *Systems analysis and design.* 5. vyd. John Wiley a Sons, 2015.

38. *Why you need Kubernetes and what it can do* [online]. 2023. [cit. 2023-11-03]. Dostupné z : https://kubernetes.io/docs/concepts/overview/.

39. *Sharding increases count and size of collection, what to do?* [Online]. 2023. [cit. 2023-11-16]. Dostupné z : https://www.mongodb.com/community/forums/t/sharding-increases-count-and-size-of-collection-what-to-do/209996.

Dodatok **A**

Zoznam použitých skratiek

ALB Application Load Balancer

API Application Programming Interface

AWS Amazon Web Services

CNAME Canonical Name Record

DAO Data Access Object

DWH Data Warehouse

EC2 Elastic Compute Cloud

ECS Elastic Container Service

EKS Elastic Kubernetes Service

HATEOAS Hypermedia as the Engine of Application State

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

OIDC OpenID Connect

PV Persistent Volumes

PVC Persistent Volume Claim

RAM Random Access Memory

RDS Relational Database Service

REST Representational State Transfer

RPC Remote Procedure Call

S3 Simple Storage Service

SOAP Simple Object Access Protocol

TLS Transport Layer Security

URI Uniform Resource Identifier

URL Uniform Resource Locator

vCPU Virtual CPU

WAF Web Application Firewall

Dodatok B

Slovník pojmov

auto scaling group – umožňuje automaticky riadiť skupiny virtuálnych serverov za účelom automatického škálovania a správy

availability zones – zóny skladajúce sa z jedného alebo viacerých dátových centier, sú umiestnené v samostatných zariadeniach

batch dát – skupina dát spracovávaná spoločne

broadcast – správa určená pre všetky shardy

cache – pamäť pre dočasné uloženie často používaných dát, za účelom zrýchlenia prístupu

cluster – skupina serverov a iných zdrojov, ktoré fungujú ako jeden systém

commit – git príkaz, ktorý vytvorí snímku git repozitára

dáta pipeline – séria krokov, ktoré prepravujú, transformujú a ukladajú dáta

endpoint – bod vstupu v komunikačnom kanáli pri komunikácii dvoch systémov

framework – sada vývojárskych nástrojov, ktorá pomáha programátorom pracovať rýchlejšie a efektívnejšie

health-check – sledovanie a vyhodnocovanie stavu aplikácie s cieľom zabezpečiť ich správnu funkčnosť

iam rola – sada oprávnení, ktorá definuje akcie a zdroje ku ktorým má entita prístup

join tabuliek – databázová operácia, ktorá spája dáta z rôznych tabuliek

kardinalita – veľkosť, počet prvkov nejakej množiny

load-balancer – zariadenie, ktoré umožňuje distribuovať pracovnú záťaž medzi viaceré servery

mongodb-fragment – súvislý rozsah hodnôt shard kľúča vrámci konkrétneho shardu

mongosh – prostredie pre interakciu s MongoDB

nodegroup – skupina virtuálnych serverov

polling – proces pri ktorom sa periodicky overuje stav nejakej udalosti

rollback – operácia, ktorá vráti všetky zmeny vykonané v rámci danej transakcie

subnets – rozsah IP adries v rámci definovaného VPC

wildcard – znak, ktorý slúži ako zástupný znak pre jeden, či viac iných znakov

Dodatok C

Obsah príloh

```
readme.txt...............................stručný popis obsahu SD karty
src
  TapiX ......................................... zdrojové kódy Tapix
  TapixVsUpdater .......... zdrojové kódy mikroslužby TapixVsUpdater
thesis..........................zdrojová forma práce vo formáte LaTeX
text ........................................................text práce
  thesis.pdf..............................text práce vo formáte PDF
materials.........................ďalšie materiály využité v texte práce
  MIEPDB16-Lecture-08-Principles.pdf ........... citovaná prednáška
  mongo_queries_test.json ......... možné typy dotazov do MongoDB
```