FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Provoz gRPC backendu v cloudovém prostředí
Student:	Bc. Radka Bodnárová
Vedoucí:	Ing. Josef Gattermayer, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Technologie gRPC je vážný kandidát na nahrazení RESTových API, avšak celý ekosystém je ještě ve stádiu vývoje, takže je zde z pohledu orchestrace a instrumentace spousty zajímavých nedořešených témat. Cílem práce je prozkoumat aktuální stav a pomocí proof-of-concept implementací, navrhnout infrastrukturu pro imaginární službu s desítkami milionů uživatelů.

Pokyny:
- Navrhněte architekturu imaginární služby s několika backendovými micro services.
- Pro zvolenou architekturu vyberte 2 rozdílné technologie cloudové infrastruktury, které budete porovnávat.
- Implementujte PoC několika micro services.
- Implementuje infrastrukturu (as a code).
- Implementujte infrastrukturu pro deployment nových verzí aplikace pomocí blue-green deployment nebo canary deployment.
- Výsledné řešení změřte pod zátěží simulující miliony uživatelů. Dokažte, že ani při deploymentu nové verze aplikace nedojde k žádnému výpadku.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 22. října 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práca

Provoz gRPC backendu v cloudovém prostředí

Bc. Radka Bodnárová

Katedra softwarového inženýrství
Vedúci práce: Ing. Josef Gattermayer, Ph.D.

3. mája 2022

Poďakovanie

V prvom rade by som sa chcela poďakovať Ing. Josefovi Gattermayerovi, Ph.D. za vedenie tejto tejto práce. Ďalej poďakovanie patrí Ing. Martinovi Beránkovi za cenné rady a pripomienky. Veľká vďaka patrí tiež mojej rodine a priateľovi za podporu počas celého štúdia.

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, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 3. mája 2022 . . . . . .

Odkaz na túto prácu

Bodnárová, Radka. *Provoz gRPC backendu v cloudovém prostředí.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Táto diplomová práca sa zaoberá technológiou gRPC pre tvorbu webových aplikácií a ich prevádzkovanie v cloude. Implementovaná bola aplikácia demonštrujúca použitie tejto technológie. Práca ďalej predstavuje a porovnáva možnosti prevádzkovania tejto aplikácie pre 3 vybrané technológie cloud infraštruktúry. Nechýba serverless, Kubernetes a service mesh. Práca sa tiež venuje nasadzovaniu nových verzií do cloudu. Implementovaná bola infraštruktúra pre nasadzovanie nových verzií aplikácie pomocou stratégie canary a riešenie bolo otestované záťažovými testami.

Klíčová slova gRPC, gRPC-Web, cloud, Google Cloud, Kubernetes, service mesh, canary

Abstract

The master's thesis deals with gRPC technology for creating web applications and running them in the cloud. An application demonstrating the use of this technology is implemented. The thesis further presents and compares the possibilities of running this application for three selected cloud infrastructure technologies including serverless, Kubernetes and service mesh. The

thesis also discusses deploying new versions to the cloud. The infrastructure for deploying new versions using canary deployment is implemented and the solution is tested with load tests.

Keywords gRPC, gRPC-Web, cloud, Google Cloud, Kubernetes, service mesh, canary

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Mikroslužby predstavujú architektonický vzor, ktorý je charakteristický najmä rozdelením aplikácie na menšie komponenty, ktoré je možné nezávisle vyvíjať, nasadzovať a škálovať. Cieľom je vytvárať voľne previazané služby, pričom jedným z najdôležitejších aspektov vývoja je zaistiť komunikáciu medzi jednotlivými komponentami.

Pomerne nový trend vo svete komunikácie mikroslužieb je **gRPC**. Technológia bola predstavená v roku 2015 a stále viac sa stáva populárnou alternatívou ku tradične používanému komunikačnému vzoru REST. Viacero organizácií vrátane veľkých technologických spoločností ako je Netflix, Spotify, Dropbox, či Cisco si už gRPC osvojili.[1]

gRPC je tiež súčasťou ekosystému Cloud Native Computing Foundation (CNCF), ktorý ponúka cloud-native riešenia. Podla CNCF umožňujú cloud native technológie organizáciám vytvárať a prevádzkovať škálovateľné aplikácie v moderných dynamických prostrediach, ako sú verejné, súkromné a hybridné cloudy. Príkladom tohto prístupu sú technológie ako kontajnery, service meshes, mikroslužby, nemenná infraštruktúra a deklaratívne API. Tieto techniky umožňujú voľne previazané systémy, ktoré sú odolné, spravovateľné a dajú sa monitorovať. V kombinácii s robustnou automatizáciou umožňujú vývojárom vykonávať zmeny s veľkým dopadom často a predvídateľne a to s minimálnou námahou.[2]

Motivácia

gRPC je výkonný framework postavený na HTTP/2, protocol buffers a je podporovaný väčšinou moderných programovacích jazykov. Kvôli obmedzeným možnostiam webového prehliadača nie je možné použiť gRPC priamo v prehliadači. Z toho dôvodu existuje **gRPC-Web**, ktorý poskytuje (limitovanú) podporu gRPC v prehliadači.

Práca sa zameriava na využitie technológie gRPC pre webové aplikácie a ich následnú prevádzku v cloude. Predstavuje filozofiu fungovania a spôsob nasadenia takejto aplikácie pre 3 rôzne technológie cloudovej infraštruktúry.

Jednou z diskutovaných technológii bude platforma **Google Cloud Run**, ako zástupca serverless computingu. Podľa posledného prieskumu *State of Cloud Native Development Report* od CNCF zaznamenala služba Google Cloud Run v období Q1 2020 - Q1 2021 najvyšší nárast používania spomedzi serverless technológií. V rámci prieskumu bolo tiež zistené, že počet vývojárov používajúcich Kubernetes sa za 12 mesiacov zvýšil o 67%.[3] Druhou zvolenou cloud technológiou pre túto prácu je **Kubernetes**.

Nakoniec sa bližšie pozrieme na to, čo znamená využitie technológie **service mesh** pre gRPC-Web aplikácie. Podľa [4] je koncept service mesh niečo ako *švajčiarsky nožík pre moderný software, ktorý rieši najzložitejšie problémy distribuovaných aplikácií založených na mikroslužbách.* Poslednou diskutovanou technológiou bude **Kubernetes s využitím Anthos service mesh**.

Ciele práce

Jedným z cieľom teoretickej časti práce je oboznámiť čitateľa s technológiou gRPC a jej využitie pre tvorbu webových aplikácií. Ďalej predstaviť vybrané technológie cloud infraštruktúry medzi ktoré patrí Google Cloud Run, Google Kubernetes Engine a Anthos Service Mesh.

Praktická časť pozostáva z návrhu a implementácie aplikácie demonštrujúcej použitie gRPC a gRPC-Web. Cieľom je implementovanú aplikáciu nasadiť na vybrané technológie cloud infraštruktúry, predstaviť spôsob nasadenia a zhodnotiť riešenie z pohľadu:

- možnosti monitorovania
- nákladov na prevádzku
- údržby riešenia

Ďalším cieľom je preskúmať možnosti realizácie nasadenia stretégiou `canary`. Nakoniec implementované nasadenie stratégiou `canary` otestovať záťažovými testami a zhodnotiť stabilitu riešenia.

Štruktúra práce

Prvá kapitola je venovaná technológii gRPC. Popisuje základné princípy a spôsoby komunikácie. Predstavená je tiež technológia gRPC-Web a jej rozdiel oproti gRPC. V druhej kapitole uvádzame základne pojmy týkajúce sa kontajnerizácie, technológií Kubernetes a service mesh. Diskutované sú tiež rôzne stratégie nasadenia. Tretia kapitola predstavuje vybrané technológie cloud infraštruktúry v prostredí Google Cloud Platform. Štvrtá kapitola sa zaoberá

návrhom a implementáciou PoC aplikácie. V piatej kapitole je predstavený spôsob prevádzkovania tejto aplikácie v cloude. Šiesta kapitola je venovaná nasadeniu stratégiou canary pre všetky diskutované platformy. Posledná kapitola popisuje možnosti záťažového testovania gRPC aplikácií spolu s výsledkami testovania pri nasadzovaní stratégiou `canary`.

Kapitola 1

gRPC

gRPC je open source framework vyvinutý spoločnosťou Google, ktorý je založený na princípe vzdialeného volania procedúr (RPC, Remote Procedure Call). Aj keď by sa mohlo zdať, že písmeno „*g*“ v slove gRPC odkazuje na Google, pravdou je, že jeho význam sa s každou verziou mení.[1]

RPC je technika pre vytváranie distribuovaných aplikácií založených na princípe klient-server. Je postavená na rozšírení princípu lokálneho volania procedúr tak, že volaná procedúra nemusí existovať v rovnakom adresnom priestore ako volajúca procedúra. Oba procesy môžu byť v rovnakom systéme alebo v rôznych systémoch prepojených sieťou. [5]

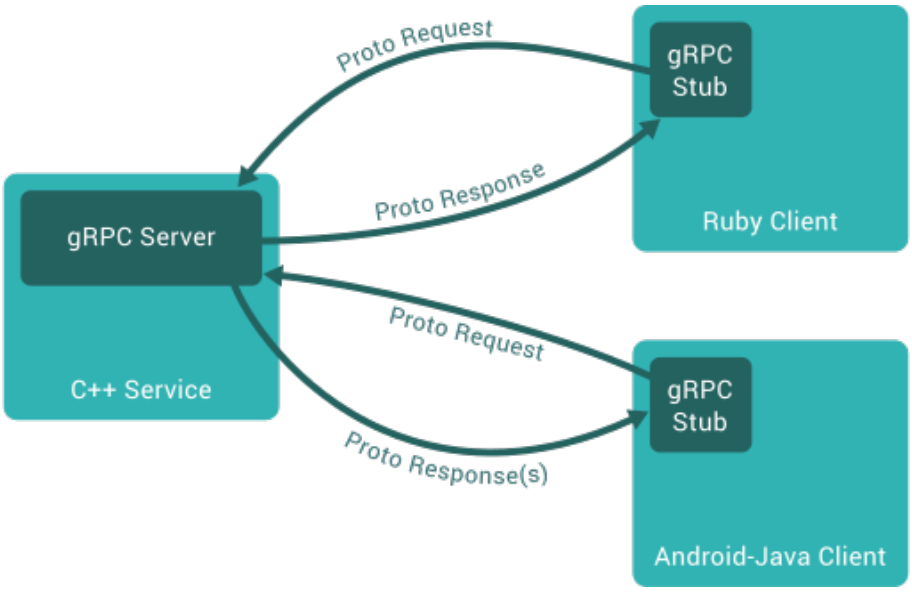
Rovnako ako v mnohých RPC systémoch, gRPC je postavený na myšlienke definovania služieb, ktoré pozostávajú z RPC metód, pričom sa špecifikujú parametre a návratové typy metód. Serverová aplikácia, na ktorej beží gRPC server toto rozhranie implementuje a spracováva požiadavky klientských aplikácii. Klientská aplikácia obsahuje tzv. client stub, čo predstavuje kus kódu, ktorý poskytuje rovnaké metódy ako server.[6] Na základe toho môže klientská aplikácia zavolať vzdialenú procedúru vyvolaním (lokálného) client stub. A to sa dá urobiť aj vtedy, ak sú server a klient napísané v rôznych jazykoch.

gRPC je navrhnutý pre prácu s rôznymi jazykmi a podporuje prístup contract-first, čo znamená, že pri vývoji sa najprv definuje rozhranie služieb. Na základe pevne definovaného rozhrania je možné vygenerovať kód pre klientskú i serverovú aplikáciu. Kód je možné vygenerovať pre ktorýkoľvek gRPC podporovaný programovací jazyk, čo umožňuje značne urýchliť vývoj aplikácií. Použitie automaticky generovaného kódu tiež zaisťuje, že klient odošle požiadavku, ktorú server očakáva, vrátane formátu odosielaných dát, čím sa predchádza problémom s interoperabilitou. V prostredí, kde na vývojí služieb pracujú nezávislé tímy, je toto obzvlášť užitočné.

Ako znázorňuje obrázok 1.1, integrácia napr. android mobilného klienta so serverom napísaným v C++ nie je žiaden problém. Služby medzi sebou

[1] `https://github.com/grpc/grpc/blob/master/doc/g_stands_for.md`

komunikujú pomocou mechanizmu protocol buffers, ktorý bude bližšie predstavený v sekcii 1.2.



Obr. 1.1: Integrácia gRPC služieb[6]

1.1 Princíp komunikácie

gPRC využíva ako prenosový protokol HTTP/2, čo prináša radu výhod oproti bežnému HTTP/1.x a robí z gRPC robustný a vysoko výkonný protokol.

V rámci tradičného HTTP/1.x protokolu nie je možné posielať niekoľko paralelných požiadaviek v rámci jedného TCP spojenia. Ak chce klient vykonávať viacero paralelných požiadaviek, je nútený využiť viacero TCP spojení. Cieľom protokolu HTTP/2 je byť v tomto smere efektívnejší. Prináša funkciu zvanú **multiplexing**, čo znamená, že klient a server môžu paralelne posielať viacero požiadaviek a odpovedí cez jediné TCP spojenie. V rámci jedného TCP spojenia tak môže existovať viacero obojsmerných tokov bajtov, ktoré sa nazývajú streamy. Streamy sú do značnej miery na sebe nezávislé, takže zablokovaná alebo zastavená požiadavka alebo odpoveď nezabráni chod ostatných streamov.[7]

Multiplexing, rovnako ako aj ostatné funkcie a optimalizácie HTTP/2 protokolu sú možné vďaka vlastnosti nazývanej **binary framing**. Tá umožňuje klientovi a serveru rozdeliť HTTP správu do nezávislých blokov – frames. Frame je najmenšia komunikačná jednotka, ktorá prenáša špecifický typ údajov – napríklad HTTP hlavičky, dáta atď.[8]

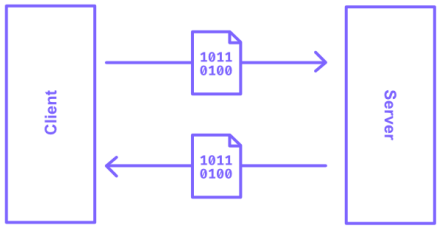
Ďalšou výkonnou vlastnosťou HTTP/2 je **kompresia hlavičiek** s využitím algoritmu HPACK. Tento algoritmus kóduje metadáta hlavičky pomocou Huffmanovho kódovania, čím sa značne zmenší ich veľkosť. Podľa Cloudflare umožňuje HPACK znížiť ingress traffic až o 53%.[9]

Hlavným prínosom gRPC sú streamy, čo ponúka veľkú škálu použitia. Napríklad spracovávanie väčšieho objemu dát už v priebehu čítania streamu, kde nie je nutné čakať na celú dávku, alebo dlhodobé streamy, kde klient môže získavať aktualizácie dát.

1.2 Typy RPC v gRPC

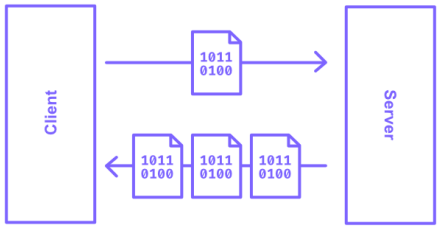
V rámci gRPC komunikácie rozlišujeme 4 typy RPC:

Unary – Najjednoduchší typ RPC je unary. Klient pošle jednu správu na server a server vráti jednu odpoveď.



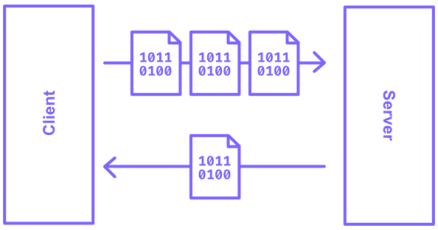
Obr. 1.2: Unary RPC[10]

Server streaming – Klient pošle na server jednu správu a server odpovedá sekvenciou správ. Táto sekvencia sa nazýva stream.



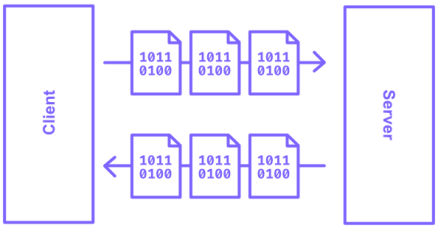
Obr. 1.3: Server streaming RPC[10]

Client streaming – Client streaming funguje na opačnom princípe ako server streaming. Klient zapisuje sekvenciu správ do streamu, z ktorého server správy prijíma. Po doručení všetkých správ server vráti jednu odpoveď.



Obr. 1.4: Client streaming RPC[10]

Bi-directional streaming – Tento typ RPC umožňuje jak klientovi, tak serveru posielať sekvencie správ. Komunikácia musí byť iniciovaná zo strany klienta a následne prebieha v dvoch nezávislých streamoch. Obaja, klient i server sa môžu rozhodnúť v akom poradí budú vykonávať zápis a čítanie správ, záleží teda na implementácií aplikačnej logiky. Poradie správ v oboch streamoch je vždy zachované.



Obr. 1.5: Bi-directional streaming RPC[10]

1.3 Protocol buffers

gRPC sa najčastejšie využíva spolu s mechanizmom nazývaným protocol buffers. Protocol buffers bol interne vyvinutý spoločnosťou Google a neskôr sprístupnený ako open source. Je definovaný ako jazykovo a platformovo nezávislý,

rozšíriteľný mechanizmus na serializáciu štruktúrovaných dát.[11] Využívanie protocol buffers pre posielanie správ umožňuje ich rýchlejší prenos po sieti, keďže správy sú serializované do binárnej podoby, a tak je ich veľkosť menšia ako v prípade formátov XML alebo JSON. Z toho dôvodu je formát protocol buffers efektívny a to najmä pre veľké dátové prenosy. Na rozdiel od formátov JSON a XML sa protocol buffers využíva aj ako jazyk definície rozhrania (z angl. interface definition language), nie len ako formát na výmenu správ. Súbor vo formáte *.proto* predstavuje popis rozhrania, na základe ktorého je možné s pomocou *protoc*[12] kompilátoru vygenerovať pre zvolený jazyk triedy a metódy potrebné pre implementáciu rozhrania. Výpis kódu 1 predstavuje jednoduchú definíciu rozhrania v jazyku protocol buffers. Každý atribút v definícii správy obsahuje jedinečné číslo, ktoré slúži na identifikáciu atribútov v binárnej správe. Protocol buffers je silne typovaný jazyk – atribúty v správe obsahujú okrem názvu aj typ, v tomto prípade je to *string*. V aktuálnej verzii proto3 sú všetky atribúty považované za voliteľné. Ak atribút odoslanej správy neobsahuje hodnotu, dosadí sa predvolená hodnota pre tento atribút. V prípade typu string je to prázdny reťazec.

```
// definícia gRPC služby
service Greeter {
  // metóda typu Unary
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// definícia požiadavky
message HelloRequest {
  string name = 1;
}

// definícia odpovede
message HelloReply {
  string message = 1;
}
```

Výpis kódu 1: Príklad definície rozhrania v protocol buffers formáte

1.4 gRPC-Web

V súčasnosti nie je možné implementovať špecifikáciu HTTP/2 gRPC vo webovom prehliadači, pretože prehliadače neposkytujú dostatočné API pre obsluhu takýchto požiadaviek. Nie je napríklad možné vynútiť použitie HTTP/2

a ak by aj bolo, surový HTTP/2 frame nie je v prehliadači dostupný. [13] Inak povedané, protokol gRPC vyžaduje pre fungovanie určitú kontrolu nad HTTP/2 frames, no žiaden moderný prehliadač k nim nemá prístup. Z tohoto dôvodu bol vytvorený projekt gRPC-Web. Aktuálne existujú 2 implementácie gRPC-Web určené pre webový prehliadač:

- **Google gRPC-Web client**[14] – považovaný za dostatočne stabilný pre produkčné použitie
- **Improbable gRPC-Web client**[15] – má status *alpha*, používa sa pre podmnožinu single-page aplikácií Improbable v produkcii

Obe implementácie zavádzajú pomerne odlišný protokol v porovnaní s protokolom gRPC cez HTTP/2. Tabuľka 1.1 zobrazuje prehľad podporovaných RPC metód pre obe implementácie. Ako je možné vidieť aktuálne, ani jedna implementácia nepodporuje všetky 4 typy gRPC metód, ale je obmedzená na *unary RPC* a *server streaming RPC*.

Pričom rozdiel medzi dvoma variantami v implementácií od Googlu je:

- grpcwebtext
  - prenášaná správa je kódovaná v base64
  - Content-type: application/grpc-web-text
- grpcweb
  - prenášaná správa je v binárnom protobuf formáte
  - Content-type: application/grpc-web+proto

Na pozadí klient využíva fetch API v prípade implementácie od Improbable, alebo XMLHttpRequest v prípade implementácie od Google.

Objekt XMLHttpRequest (XHR) sa používa na interakciu prehliadača so servermi. Umožňuje načítať údaje z adresy URL bez toho, aby bolo nutné vykonať úplnu obnovu stránky. To umožňuje aktualizovať len časť webovej stránky bez narušenia činnosti používateľa.[16] Fetch API poskytuje JavaScript rozhranie pre prístup a manipuláciu častí protokolu HTTP, ako sú požiadavky a odpovede. Jedná sa o novšiu alternatívu k XHR.[17]

Implementácia	Unary	Server streaming	Client/bi-directional stream
Improbable	áno	áno	nie
Google (grpcwebtext)	áno	áno	nie
Google (grpcweb)	áno	nie	nie

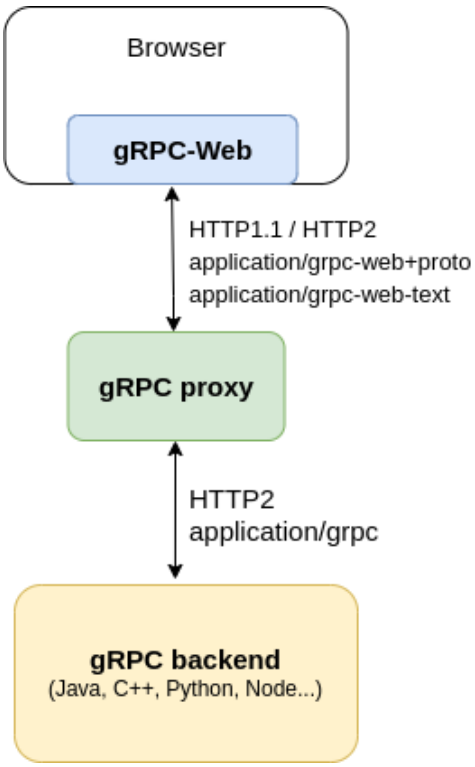
Tabuľka 1.1: Prehľad podporovaných gRPC metód v rámci gRPC-Web implementácií

Komunikácia medzi gRPC-Web aplikáciou a gRPC aplikáciou nie je možná napriamo. Základná myšlienka fungovania gRPC komunikácie medzi klientom, ktorý beží vo webovom prehliadači, a gRPC serverom spočíva vo využití prostredníka (proxy). Tento model je znázornený na obrázku 1.6. Úlohou proxy je konvertovať požiadavky typu gRPC-Web od klienta a posielať ich cez gRPC protokol na server.

Komunikácia medzi klientom a proxy môže prebiehať cez HTTP/1.1 alebo HTTP/2. Použitie HTTP/1.1 predstavuje nasledujúce nevýhody:

- Obmedzenie na počet streamov – počet súbežných streamov je obmedzený maximálnym počtom HTTP spojení, ktoré prehliadač povoľuje. Väčšina moderných prehliadačov umožňuje maximálne šesť spojení na jednu doménu.[18]
- Absencia binárnej optimalizácie

Použitie HTTP/2 pri komunikácií cez gRPC-Web medzi klientom a proxy je možné dosiahnuť zapnutím TLS.[19]



Obr. 1.6: Diagram komunikácie medzi gRPC backend službou a gRPC-Web klientom

Kapitola 2

Kontajnerizácia a kubernetes

2.1 Obrazy a kontajnery

Kontajnery sú odľahčené balíčky, ktoré obsahujú zdrojový kód softwaru vrátane závislostí, ako je konkrétna verzia behového prostredia (z angl. runtime environment) programovacieho jazyka a knižnice, ktoré sú potrebné na spustenie softvérových služieb.[20] Návod na zostavenie kontajneru sa nazýva obraz (image). Obraz predstavuje statický súbor, ktorý sa používa na vytváranie kontajnera a spúšťanie kódu v ňom.

2.2 Kubernetes

Kubernetes je platforma, používaná na kontrolované nasadzovanie a škálovanie aplikácií založených na kontajneroch, pričom osobitný dôraz sa kladie na definovanie spoločného jazyka medzi vývojom a prevádzkou. Poskytuje framework, ktorý spája koncepty vývoja a prevádzky do spoločného základu: popis zdrojov vo formáte YAML (alebo JSON).[21]

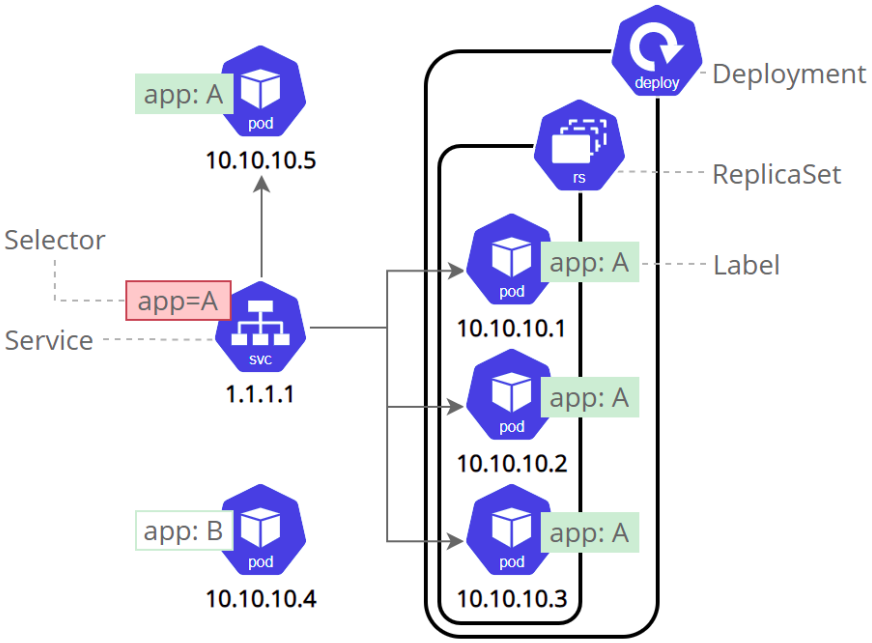
Cieľom práce nie je kompletne predstaviť technológiu Kubernetes, preto budú stručne predstavené len základné stavebné bloky spomínané v texte práce. Tieto stavebné bloky sú zobrazené tiež na obrázku 2.1. Kubernetes klaster pozostáva zo súboru pracovných strojov, nazývaných uzly, na ktorých bežia kontajnerové aplikácie. Každý klaster má aspoň jeden pracovný uzol. Pracovné uzly hostia Pody, na ktorých beží aplikácia.[22]

Pod je najmenšou nasaditeľnou jednotkou, ktorú je možné spravovať v Kubernetes. Skladá sa z jedného alebo viacerých kontajnerov, ktoré bežia na tom istom uzle.

ReplicaSet zabezpečuje, že je v danom čase spustený určitý počet replík Podov.

Deployment slúži k aktualizácií objektov ReplicaSet a Pod pomocou deklaratívneho prístupu. Vyjadruje požadovaný stav, na základe ktorého Deployment controller zabezpečuje replikáciu, zavedenie a automatickú opravu v prípade zlyhania Podu. Pody sú vytvárané a ničené tak, aby zodpovedali tomuto požadovanému stavu.[22]

Service predstavuje abstraktný spôsob, ako vystaviť aplikáciu, ktorá beží na súbore Podov ako sieťovú službu. Služba môže byť typu ClusterIP, vtedy je prístupná len aplikáciám v rámci klastra. Pre externé vystavenie služby mimo klastra slúžia služby typu NodePort alebo LoadBalancer.



Obr. 2.1: Základné stavebné bloky v Kubernetes [22]

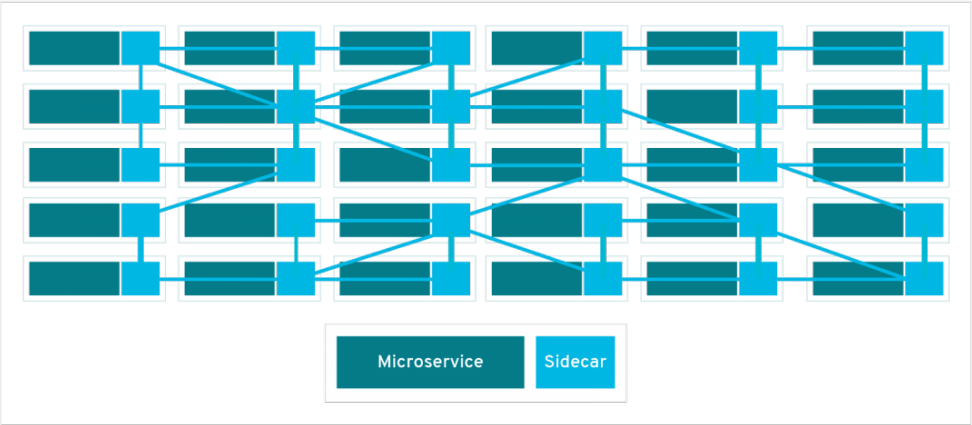
2.3 Service mesh

Service mesh je špecializovaná vrstva infraštruktúry, ktorá spravuje komunikáciu medzi internými mikroslužbami a komponentami. Umožňuje transparentne pridávať vyspelé funkcie, ako je monitorovanie, správa prevádzky a zabezpečenie, bez nutnosti modifikovať kód. Termín „service mesh“ označuje jednak typ softvéru, ktorý sa využíva na implementáciu tohto vzoru, ako aj bezpečnostnú alebo sieťovú doménu, ktorá sa vytvorí pri používaní tohto softvéru.[23]

Komunikácia medzi službami v service mesh je typicky riadená pomocou takzvaných sidecar proxies. Sidecar proxy je nasadená vedľa každej inštancie kontajneru služby a abstrahuje logiku zameranú na komunikáciu tejto mikroslužby. Všetky prichádzajúce i odchádzajúce požiadavky služby sú zachytávané touto proxy. Tento vzor je znázornený na obrázku 2.2. Medzi hlavné funkcie, ktoré sa bežne vyskytujú v service mesh implementáciách patria:

- dynamické riadenie smerovania prevádzky medzi službami,
- monitorovanie prevádzky a sledovanie komunikačných tokov,
- zabezpečenie komunikácie medzi službami,
- podpora odolnosti služieb pomocou vzorov ako circuit breaking, timeouts, retries [24].

Bez service mesh by musela byť táto logika naprogramovaná v každej mikroslužbe zvlášť. S každou novou pridanou službou je prostredie komunikácie komplikovanejšie a zvyšuje sa počet bodov možného zlyhania.[25] Service Mesh umožňuje riadiť túto komplexnú komunikáciu a zjednodušuje jej diagnostiku.



Obr. 2.2: Sidecar proxy sa nachádza vedľa mikroslužby a smeruje požiadavky na ostatné proxy.[25]

Architektúra service mesh pozostáva z dvoch častí:

Data plane označuje sieť, ktorá je tvorená dvojicami mikroslužba a jej sidecar proxy.

Control plane časť sa stará o správu služieb v rámci data plane. Poskytuje API pre konfiguráciu, manipuláciu a pozorovanie celej siete.

2.4 Stratégie nasadenia softvéru

Continuous delivery je schopnosť bezpečne, rýchlo a udržateľným spôsobom dodávať zmeny aplikácie – počítajúc do toho novú funkciu, zmenu konfigurácie, opravy chýb – do produkcie alebo do rúk používateľov.[26] Jedným z cieľov continuous delivery je nasadiť aplikáciu bez viditeľného vplyvu na koncového používateľa.

Kubernetes poskytuje dve základne stratégie[27], pomocou ktorých je možné nahradiť pri nasadení starú verziu novou:

- **Recreate** – všetky pody starej verzie sú zničené predtým než sa vytvoria pody s novou verziou. Výsledkom je nedostupnosť aplikácie medzi vypnutím a reštartom.
- **RollingUpdate** – pody s novou verziou sa postupne pridávajú, pody so starou verziou sa postupne ukončujú.

Ani jedna stratégia však neposkytuje dostatočnú kontrolu nad nasadením. Najmä ak sa pri nasadení novej verzie objavia problémy, automatické vrátenie sa k starej verzii nie je podporované.

V tejto sekcii budu priblížené 2 stratégie, pomocou ktorých je možné dosiahnuť nasadenie bez odstávky a znížiť riziko pri uvedení novej verzie do produkcie. Jedná sa o **Blue-green deployment** a **Canary deployment**.

2.4.1 Blue-green deployment

Stratégia blue-green spočíva v prevádzkovaní dvoch identických prostredí, kde jedno je označované ako *green* a druhé ako *blue*. Na presnom označení nezáleží, dôležitý je fakt, že v jednom momente sú produkčné požiadavky odosielané len na jedno prostredie. Nová verzia služby je najprv nasadená na prostredie, označené napr. *green*, zatiaľ čo stará beží na produkčnom prostredí *blue*. Po tom, ako prebehne testovanie nových zmien na prostredí *green*, nastáva premigrovanie celej prevádzky z prostedia *blue* na *green* bez citeľnej zmeny pre používateľov. V prípade nečakaných problémov je možné jednoducho premigrovať prevádzku späť na predchádzajúcu verziu na prostredí *blue*. Ak všetko prebehne v poriadku a prostredie *green* je dostatočne stabilné, pri ďalšom nasadení využijeme prostredie *blue* pre testovanie novej verzie. Prepnutie prevádzky z *green* na *blue* nastáva rovnakým spôsobom ako prepnutie z *blue* na *green*. Táto stratégia je vhodná najmä pre nasadenie kontajnerov v cloudovej infraštruktúre a menej vhodná pre nasadenia v dátových centrách.[28]

2.4.2 Canary deployment

Stratégia canary predstavuje rozšírenie stratégie blue-green. Zatiaľ čo v rámci stratégie blue-green je celá prevádzka premigrovaná naraz, v prípade canary

sa toto deje postupne. Nová verzia služby je nasadená do produkcie, pričom je stará verzia stále aktívna. Časť prichádzajúcich požiadaviek je presmerovaná na novú verziu, typicky na základe percenta požiadaviek. Ak všetko prebieha v poriadku, zvyšuje sa percento požiadaviek obsluhovaných novou verziou až kým nedosiahne 100%. Týmto spôsobom je možné odtestovať novú verziu na malej podmnožine používateľov v produkcii pred sprístupnením všetkým používateľom. Keďže uvedenie do produkcie nastáva kontrolovane, je tento proces o niečo menej rizikový ako v prípade stratégie blue-green.

2.5 gRPC a bezvýpadkové nasadzovanie

V klasickej HTTP komunikácií, kde je komunikácia striktne v režime požiadavka-odpoveď, je veľmi jednoduché výpadok vyriešiť na strane proxy serveru, vtedy sa požiadavka zopakuje na dalšom funkčnom serveri. V prípade streamov je problém komplikovanejší. Pred prijatím opatrení je nutné sa zamyslieť nad povahou dát v streame. Základným kritériom je požiadavka na konzistenciu dát.

Základné príklady dát v streame:

1. Dáta sú platné len v čase svojho vzniku a sú prezentované používateľovi napríklad v mobilnej aplikácií. Ak aktualizujeme napríklad periodicky hodnotu kurzu meny pre účely živého zobrazenia v mobilnej aplikácií, je výpadok jednotiek hodnôt v streame zanedbateľný a nie je potrebné ho riešiť.

2. Dáta generované systémom v určitom čase, pričom je nutné spracovať každý záznam. V tomto prípade nie je možné o žiadne dáta prísť.

3. Dáta, kde na základe podmienok chceme prečítať všetky odpovedajúce dátové vety. Tu opäť požadujeme, aby dáta boli konzistentné.

Proxy všeobecne poskytuje len možnosť opakovania požiadavky v reakcii na konkretné stavy odpovede. Toto riešenie bude dostatočne fungovať v prípade metód typu `unary` a v prípade dát popísaných v bode 1. V prípade dát v bode 3 by sme dostali duplicitné dáta (stream by začal od začiatku) a v bode 2 by záležalo na implementácií serveru, ale je možné, že by sme mali rovnaký problém.

Pri konfigurácii opakovania je tiež dôležitým faktorom idempotencia požiadavky. Opakovanie je bezpečné len v prípade idempotentných požiadaviek, teda takých, u ktorých má viacnásobné vykonanie rovnaký efekt, ako keď je požiadavka vykonaná iba raz. Typickým príkladom požiadavky, ktorá nie je idempotentná je taká, ktorej vykonanie spôsobí vytvorenie nového zdroja na serveri.

V prípade, keď konzistencia dát nie je nutná, je možné výpadok ošetriť na strane klienta služby, kde si klient sám zaistí zopakovanie volania, prípadne na strane proxy.

V ostatných prípadoch nie je možné realizovať zopakovanie požiadavky iba na strane proxy a je potrebné to riešiť implementačne.

Príklad 1: čítame dlhšiu dátovu vetu, kde využívame stream z dôvodu možnosti spracovania jednotlivých záznamov už v priebehu čítania. Aby klient dokázal prekonať výpadok, je potrebné naviazať na posledný prečítaný záznam. To vyžaduje implementáciu na strane servera i klienta. Tejto funkcionality nie je možné dosiahnúť konfiguračne na strane proxy. Bolo by samozrejme možné požiadavku zopakovať a začať od začiatku, ale klientská aplikácia by musela vedieť spracovávať duplicitné záznamy. Je na zváženie, či by sa v takomto prípade nejednalo o zbytočnú utilizáciu prostriedkov.

Príklad 2: Dlhodobý stream s požiadavkou na spracovanie všetkých dát. Tu sa jedná o veľmi komplexný problém, ktorý bude musieť byť riešený ako na strane klienta, tak serveru. Zopakovanie požiadavky pomocou proxy by mohol byť iba doplnkom k implementácii.

Ak pracujeme so streamami, je potrebné mať na pamäti možnosť výpadku streamu už v čase návrhu aplikácie. Je potrebné zhodnotit povahu dát a vybrať správny prístup.

Kapitola 3

Google Cloud Platform

Google Cloud Platform (GCP) je platforma od Googlu poskytujúca cloud služby. Podľa [3] je po Amazon Web Services (AWS) druhý najobľúbenejší poskytovateľ cloudových služieb medzi Kubernetes používateľmi.

3.1 Google Container registry

Pre nasadenie kontajnerových aplikácií do cloudu je potrebné vytvorenie obrazu Docker kontajneru. Tieto obrazy sa následne ukladajú do registrov, ktoré môžu byť verejné alebo privátne. Google poskytuje službu **Container registry**, privátny register, pomocou ktorého je môžné jednoducho ukladať, spravovať a zabezpečovať obrazy Docker kontajnerov a následne ich sťahovať do iného systému, ako je napríklad Kubernetes klaster.

3.2 Cloud Run

Serverless je cloud-native vývojový model, ktorý umožňuje vývojárom vytvárať a spúšťať aplikácie bez toho, aby museli spravovať servery. Napriek tomu, že výraz serverless v preklade znamená „bezserverový“, neznamená to absenciu servera. Kód stále beží na serveroch, ale tie sú abstrahované od vývoja aplikácií.[29] Najčastejšie je pojem serverless spájaný s distribučným modelom Function as a Service (FaaS), častokrát sa tieto pojmy aj zamieňajú. Serverless nemusí vždy znamenať FaaS a príkladom toho je aj platforma Cloud Run.

Cloud Run je platforma, ktorá kombinuje kontajnery a serverless prístup a bola vytvorená s cieľom preklenúť medzeru medzi týmito dvoma spôsobmi distribúcie aplikácií. Umožňuje spúštanie kontajnerov serverless spôsobom a je postavená na platforme Knative.

Knative je open-source framework, ktorý operuje nad Kubernetes. Poskytuje API a behové prostredie pre vytváranie, nasadzovanie a správu serverless aplikácií.

> *„If Kubernetes is an electrical grid, then Knative is its light switch.“*
> [30] (Kelsey Hightower, Google Cloud Platform)

Cieľom Cloud Run je zjednodušenie procesu vývoja a nasadzovania aplikácií do cloudu. Vyznačuje sa nasledujúcimi charakteristikami:

- **podporuje gRPC** – Cloud Run je určený pre spúšťanie bezstavových kontajnerov, ktoré sú riadené udalosťami. Môže sa jednať o požiadavku typu HTTPS, gRPC alebo o Pub/Sub[2] správu.
- **automatické škálovanie** – nasadená služba sa automaticky škáluje podľa počtu prichádzajúcich požiadaviek bez toho, aby bolo potrebné konfigurovať alebo spravovať plnohodnotný Kubernetes klaster. Ak neprichádzajú žiadne požiadavky, služba je automaticky škálovaná na nulu, t.j. nevyužíva žiadne zdroje.[31]
- „**platíš len za to, čo využiješ**“ – automatické škálovanie na nulu umožňuje platiť len za konkrétne využívanie zdrojov a vyhnúť sa plateniu v prípade nečinného hardvéru.
- **flexibilita** – vďaka použitiu kontajnerov dokáže Cloud Run pracovať s akýmkoľvek programovacím jazykom či knižnicami
- **automaticky vygenerovaný názov domény + TLS** - každá služba má automaticky priradený názov domény (ktorý je možné zmeniť) spolu s TLS certifikátom.
- **integrované monitorovanie a logovanie**

3.2.1 Cloud Run model

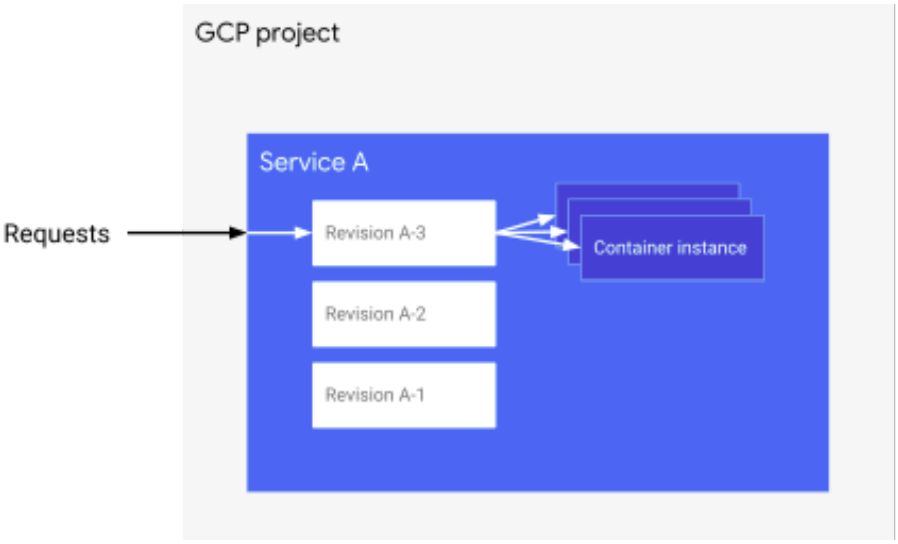
Služba je základným prvkom platformy Cloud Run. Každá služba vystavuje jedinečný endpoint a automaticky škáluje infraštruktúru tak, aby zvládala obsluhovať prichádzajúce požiadavky.[32] Služba môže obsahovať niekoľko rôznych revízií – aplikačných verzií.

Revízia sa skladá z konkrétneho obrazu kontajnera spolu s nastaveniami, ako sú premenné prostredia, či limity využívanej pamäte alebo CPU. Každé nasadenie do služby vytvára novú revíziu, ktorá je immutable. Každá zmena konfigurácie, napr. zmena premennej prostredia vedie k vytvoreniu novej revízie.[32]

Revízia, ktorá prijíma požiadavky sa automaticky škáluje na počet **inštancií kontajnera** potrebných na spracovávanie požiadaviek. Jedna inštancia

[2] `https://cloud.google.com/pubsub/docs/overview`

kontajnera môže prijímať viacero požiadaviek súčasne. Pomocou nastavenia hodnoty *concurrency* je možné nastaviť maximálny počet požiadaviek, ktoré môžu byť paralelne odoslané danej inštancii kontajnera.[32]



Obr. 3.1: Cloud Run model[32]

3.2.2 Škálovanie

Automatické škálovanie a šetrenie zdrojov v rámci serverless sprevádza problém nazývaný **studený štart** (Cold start problem). Ak služba nespracováva požiadavky, je po určitej dobe automaticky škálovaná na 0, čo znamená, že sa všetky inštancie kontajnerov vypnú a zničia z dôvodu šetrenia zdrojov. Tak to ostáva až do momentu, keď služba neprijíme ďalšiu požiadavku. V dôsledku toho je doba spracovania prvej takejto požiadavky predĺžená o proces inicializácie a spúšťania kontajneru.

V prípade Cloud Run je maximálna doba nečinnosti, po ktorej budú inštancie zničené, 15 minút.[33] Pokiaľ by sme sa chceli vyhnúť problému so studeným štartom, je možné nastaviť minimálny počet inštancií, ktoré budú „zahrievané“ – inštancie aj v prípade nečinnosti ostávajú zapnuté a pripravené spracovávať požiadavky.

3.2.3 Riadenie prevádzky (Traffic management)

Knative API umožňuje kontrolu prevádzky naprieč jednotlivými revíziami služby. Vďaka tomu je možné v Cloud Run špecifikovať, ktorá revízia bude prijímať požiadavky a v akom objeme. Prevádzku je možné rozdeliť medzi

niekoľko revízií definovaním percentuálneho objemu spracovávaných požiadaviek pre každú revíziu. Pomocou jednoduchého premigrovania časti prevádzky z jednej revízie na druhú je možné implementovať procesy ako postupné zavádzanie novej verzie, či prípadne vrátenie sa k predchádzajúcej verzii.

3.2.4 Nasadzovanie

Nasadenie aplikácie do platformy Cloud Run je jednoduchý proces, ktorý pozostáva z dvoch krokov:

- Príprava docker obrazu a jeho nahratie do `Container registry`
- Nasadenie kontajneru do Cloud Run

Nasadenie kontajneru do Cloud Run je možné priamo v cloud konzole alebo s využitím CLI nástroja *gcloud*[3]. Gcloud umožňuje jediným príkazom vytvoriť službu a nasadiť existujúci obraz kontajneru uložený v Container registry. Konfiguráciu služby je možné špecifikovať pomocou argumentov príkazu alebo priložiť vo forme *yaml* špecifikácie.

3.2.5 Cenotvorba

Ako už bolo spomínané, Cloud Run umožňuje platiť len za čas, keď sú zdroje reálne využívané. Cena sa počíta na základe využitia procesoru vo vCPU/s a pamäte v GB/s s presnosťou na 100ms. K cene sa ďalej pripočítava počet spracovaných požiadaviek a objem odchádzajúcich požiadaviek.

Každá inštancia služby má štandardne priradený 1 vCPU, čo predstavuje jeden fyzický procesor. Túto hodnotu je možné navýšiť i znížiť, avšak zníženie znamená určité obmedzenia. Jedným z obmedzení je napríklad to, že hodnota *concurrency* musí byť nastavená na 1.[34] V takomto prípade môže jedna inštancia kontajnera spracovať v jeden moment maximálne jednu požiadavku.

Každý mesiac máme k dispozícií určitý objem požiadaviek, vCPU a GB sekúnd zdarma. Tento objem ako aj sadzba po prekročení sa líšia od toho, či sa rozhodneme platiť len za čas ked sú zdroje reálne využívané alebo požadujeme trvalú alokáciu CPU, t.j. aby boli inštancie „zahrievané“. Presné sadzby je možné dohľadať v cenníku[4] alebo využiť pre výpočet kalkulačku[5].

3.3 Google Kubernetes Engine

Google Kubernetes Engine (GKE) predstavuje spravovanú Kubernetes platformu od spoločnosti Google. Ide o spravovanú službu v tom zmysle, že Google spravuje virtuálne stroje a počiatočné nastavenie.

[3] `https://cloud.google.com/sdk/gcloud/reference/run`
[4] `https://cloud.google.com/run/pricing`
[5] `https://cloud.google.com/products/calculator`

V GKE sa klaster skladá minimálne z jednej control plane časti a viacerých pracovných strojov nazývaných uzly. Control plane spravuje pracovné uzly a pody v klastri.

GKE poskytuje dva režimy konfigurácie klastra:

- **Standard** – tento režim nám umožňuje spravovať základnú infraštruktúru klastrov (napr. konfigurácie uzlov).
- **Autopilot** – Google za nás spravuje celú základnú infraštruktúru klastrov, zaisťuje optimalizáciu klastra a skupín uzlov. Automaticky pridáva a odoberá uzly na základe požadovaných zdrojov aplikacií, čím sa podobá funkcii Cloud Run.

3.3.1 Cenotvorba

Poplatky za GKE režime Standard závisia od počtu uzlov a typov použitých virtuálných strojov v nich. K dispozící je široký výber virtuálných strojov s rôznou kapacitou vCPU a RAM. Zdroje výpočtového výkonu sa účtujú za každú sekundu používania, až kým nie sú uzly zmazané.

Ďalším z poplatkov je poplatok za správu klastra vo výške 0,10 USD za hodinu a vzťahuje sa na všetky klastre GKE bez ohľadu na režim prevádzky, veľkosť klastra alebo topológiu. GCP ponúka „free tier“ s kreditom, ktorý pri využití jedného klastra dokáže pokryť tieto náklady.

Medzi ďalšie poplatky môže patriť poplatok za úložisko a poplatok za Load Balancer, ktorý sa odvíja od počtu smerovacích pravidiel a objemu prichádzajúcich dát.

3.4 Anthos service mesh

Anthos Service Mesh využíva API a základné komponenty open source service mesh platformy Istio.[6] Istio je aktuálne najpopulárnejšia implementácia service mesh pre Kubernetes.[35] So službou Anthos Service Mesh získame testovanú a podporovanú distribúciu Istio spravovanú spoločnosťou Google.

Istio používa rozšírenú verziu open source proxy Envoy na sprostredkovanie prichádzajúcej a odchádzajúcej prevádzky pre všetky služby v rámci service mesh.

Envoy umožňuje väčšinu pokročilých funkcií, ktoré Istio ponúka vrátane riadenia prevádzky, terminácie TLS a pozorovateľnosti.

Okrem toho, že je Envoy používaná ako proxy pre komunikáciu medzi jednotlivými službami, je v rámci Istio využívaná tiež ako Load Balancer a slúži na smerovanie prevádzky z prostredia mimo service mesh k službám, ktoré bežia v service mesh.

[6] `https://istio.io`

3.4.1 Pozorovateľnosť

Anthos Service Mesh poskytuje prehľad o stave a výkonnosti služieb, pomocou automatickej integrácie so službami Cloud Monitoring a Cloud Logging. Každá sidecar proxy, ktorá je súčasťou podu využíva Cloud Monitoring API a Cloud Logging API pre reportovanie telemetrických údajov. Tie sú následne automaticky odosielané a dostupné na stránkach Anthos Service Mesh v Cloud konzoli.[36]

3.4.2 Riadenie prevádzky (Traffic management)

Jednou zo základných funkcií Istia je riadenie prevádzky. Medzi kľúčové prvky pre riadenie prevádzky patria:

Gateway predstavuje konfiguráciu pre Ingress gateway, ktorá slúži ako Load Balancer na vstupe do service mesh. Umožňuje nastaviť funkcie vrstvy L4-L6 load balanceru, takže napríklad zoznam portov, ktoré by mali byť vystavené, typ protokolu, konfiguráciu TLS.

Virtual Service zachytáva a smeruje prevádzku na Kubernetes služby podľa definovaných pravidiel. Umožňuje smerovať prevádzku na rôzne služby na základe konkrétnych endpointov. V rámci `Virtual Service` je tiež možné nakonfigurovať rozdelenie prevádzky medzi viaceré verzie služby, buď podľa váh (vhodné pri nasadení stratégiou canary) alebo na základe HTTP hlavičiek.

Destination Rule definuje pravidlá, ktoré sa aplikujú na prevádzku po tom, čo je presmerovaná na Kubernetes službu cez Virtual Service. Destination rule sa používa napr. na identifikáciu viacerých verzií služby, ktoré sa nazývajú podmnožiny.

3.4.3 Bezpečnosť

Anthos Service Mesh (ASM) poskytuje TLS certifikáty pre všetky služby, ktoré sú súčasťou service mesh. Od verzie ASM 1.5 je automatické vzájomné TLS (auto mTLS) predvolene zapnuté. Pri použití funkcie auto mTLS klientská sidecar proxy automaticky zistí, či server obsahuje sidecar. Ak server sidecar obsahuje, klientská sidecar s ním komunikuje cez mTLS. Ak server sidecar neobsahuje, komunikácia medzi klientom a serverom je nešifrovaná.[37]

3.4.4 Cenotvorba

Služba Anthos Service Mesh je k dispozícii ako súčasť systému Anthos. Cena za službu Anthos je založená na počte vCPU v klastri. Existujú dve možnosti stanovenia ceny – vo forme „pay-as-you-go“, kde platíme za každú využitú hodinu vCPU, alebo vo forme mesačného predplatného. Mesačné predplatné

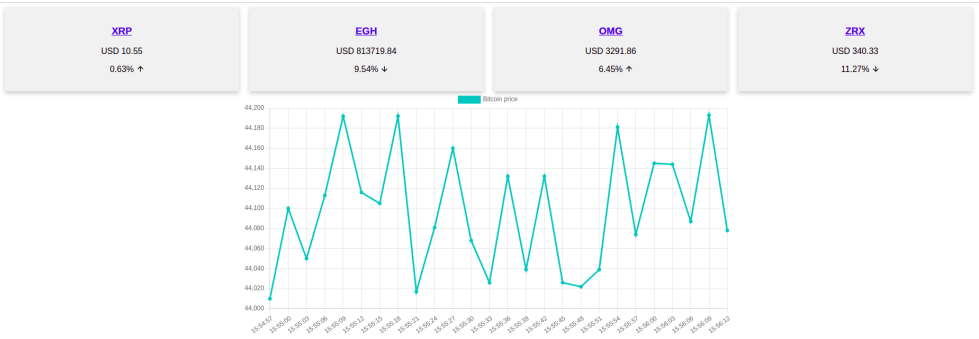
činí 6 USD/vCPU, pri platbe za každú hodinu po celý mesiac sa dostávame na cenu 8 USD/vCPU. Tieto náklady sa vzťahujú len na softvérovú vrstvu Anthos a nezahŕňajú základnú infraštruktúru v cloude.

Kapitola 4

Implementácia aplikácie

Jedným z cieľom práce je implementovať aplikáciu, ktorá demonštruje použitie technológie gRPC-Web. Aplikácia pozostáva z dvoch mikroslužieb – backend servera a frontend webového klienta. Požiadavkou bolo, aby aplikácia implementovala obidve typy metód, ktoré gRPC-Web podporuje – *unary* a *server streaming*.

Aplikácia simuluje „dashboard“, ktorý zobrazuje hodnoty kryptomien. Ukážka aplikácie je na obrázku 4. Na zobrazenom grafe umožňuje používateľovi sledovať vývoj ceny vybranej kryptomeny v reálnom čase. Graf je tvorený na základe streamovaných dát zo serveru – jedná sa teda o požiadavku typu *server streaming*. Kartičky predstavujú jednoduché požiadavky typu *unary*. Pre jednoduchosť sú všetky zobrazované dáta náhodne generované.



Obr. 4.1: PoC aplikácie

4.1 API

API pozostáva zo služby s názvom PriceService, ktorá obsahuje 2 RPC metódy:

- metóda `getPrice` predstavuje unary RPC, prijíma správu typu PriceRequest a vracia správu typu PriceReply, ktorá obsahuje hodnotu a časovú značku.
- metóda `getPriceStream` predstavuje server streaming RPC, prijíma správu typu *PriceStreamRequest* a vracia stream správ typu *PriceReply*.

```
service PriceService {
  // unary RPC
  rpc getPrice(PriceRequest) returns (PriceReply);
  // server streaming RPC
  rpc getPriceStream(PriceStreamRequest)
        returns (stream PriceReply);
}

message PriceRequest {
  string name = 1;
}

message PriceStreamRequest {
  string name = 1;
}

message PriceReply {
  double value = 1;
  int64 timestamp = 2;
}
```

Výpis kódu 2: Definícia rozhrania

4.2 Frontend

Frontend pozostáva z jednoduchej single page aplikácie pre zobrazovanie dát. Okrem `unary` požiadaviek umožňuje konzumovať stream dát, na základe ktorého aktualizuje graf. Aplikácia bola napísaná v jazyku JavaScript a využíva knižnicu gRPC-Web[38] pre komunikáciu s backend serverom (cez proxy).

Výpis kódu 3 predstavuje implementáciu metódy `getPriceStream`. Objekty `PriceServiceClient` a `PriceStreamRequest` sú vygenerované na základe `proto` definície API. Implementovaná bola aj jednoduchá *retry* logika, ktorá v prípade stratenia spojenia s backendom zopakuje požiadavku a obnoví stream. Podľa gRPC dokumentácie [39] by mal klient zopakovať požiadavku, keď server vráti kód odpovedi 14 (UNAVAILABLE).

Pre vykresľovanie grafu bola použitá knižnica Chart.js[40]. Ako server, kde pobeží frontend časť, bol zvolený HTTP server Nginx[41].

```
var client = new PriceServiceClient(url);

const streamPrice = async (currency, maxAttempts, delay) => {

    var streamRequest = new PriceStreamRequest();
    streamRequest.setName(currency);

    const execute = async (attempt) => {
        var stream = client.getPriceStream(streamRequest, {});
        stream.on('data', (response) => {
            var time = getLocalTime(response.getTimestamp())
            updateChart(response.getValue(), time)
        });
        stream.on('error', (err) => {
            if (err.code === 14 && (attempt <= maxAttempts)) {
                const nextAttempt = attempt + 1
                return wait(() => execute(nextAttempt), delay)
            } else {
                throw err
            }
        });
    }
    return execute(1)
}
```

Výpis kódu 3: Implementácia metódy `getPriceStream`

4.3 Proxy

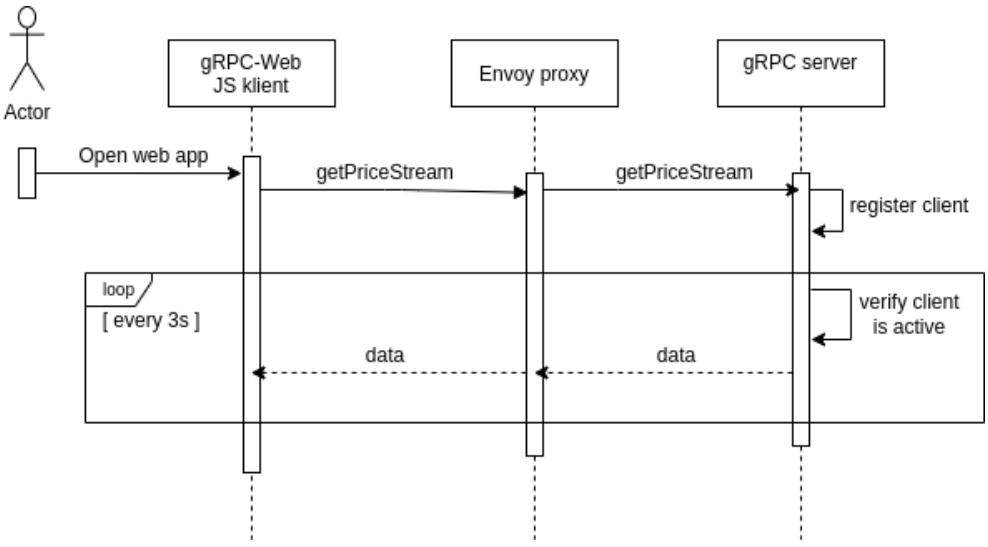
Pre sprevádzkovanie komunikácie medzi backend serverom a frontend klientom bola použitá proxy Envoy [42]. Podľa oficiálnej gRPC-Web dokumentácie [14] je práve Envoy doporučenou proxy, keďže poskytuje vstavanú podporu pre gRPC-Web. Preklad medzi gRPC a gRPC-Web požiadavkami je možné zapnúť pomocou rozšírenia gRPC Web filter[43].

4.4 Backend server

Backend server poskytuje klientskej aplikácií dáta pomocou 2 spomínaných metód – unary metóda `getPrice` a server streaming metóda `getPriceStream`.

Metóda `getPriceStream` bola implementovaná ako dlhotrvajúce spojenie. Keď používateľ navštívi webovú aplikáciu, na klientovi sa zavolá metóda `getPriceStream`, čím sa prihlási na odber dát. Dáta sú klientovi odosielané až do momentu, kým spojenie nie je ukončené zo strany klienta. Štandardne to funguje tak, že ak je spojenie ukončené na strane klienta, považuje sa za polouzavreté. Na strane serveru sa spojenie automaticky neukončí a server sa naďalej snaží zapisovať do streamu, aj keď je klient už odpojený. Po čase by tak mohlo dôjsť k zahlteniu serveru neaktívnymi klientskými pripojeniami.

Z toho dôvodu bola metóda implementovaná tak, že backend server si udržuje zoznam aktívnych klientov, ktorým v pravidelných intervaloch odosiela dáta. Pred odoslaním dát server skontroluje, či je klient aktívny. Ak nie, vyradí ho zo zoznamu aktívnych klientov. Tento proces je znázornený na sekvenčnom diagrame na obrázku 4.2.



Obr. 4.2: Sekvenčný diagram streamovania dát

4.4.1 Použité technológie

Pre implementáciu bol zvolený jazyk **Java** a framework **Spring Boot**[7]. Výber jazyka bol na základe preferencií a skúseností autorky. Pre implementáciu tejto časti by bol rovnako vhodný ktorýkoľvek gRPC podporovaný jazyk.

Framework Spring Boot umožňuje aplikovať prístup k vývoju aplikácií známy ako Rapid application development a je dnes štandard pre vývoj mikroslužieb v jazyku Java.[44] Umožňuje vytvárať samostatné aplikácie a zjednodušiť proces vývoja tým, že poskytuje širokú škálu knižníc nazývaných „starters“. Jednou z týchto knižníc, ktorá bola použitá je aj knižnica pre vývoj gRPC aplikácií **gRPC-Spring-Boot-Starter**[8].

gRPC-Spring-Boot-Starter automaticky konfiguruje a spúšťa vstavaný gRPC server, dokáže vyhľadať všetky servisné triedy na základe anotácie `@GrpcService` a priradiť tieto gRPC služby k serveru. Umožňuje taktiež integráciu so Spring-boot actuator[9] pre zber metrík o prichádzajúcich a odchádzajúcich požiadavkách a ich vystavenie.

Pre zostavenie aplikácie a správu závislostí bol použitý nástroj **Maven**[10]. Pomocou pluginu **Maven Protocol Buffers Plugin**[11] je možné na základe **.proto** súborov vygenerovať potrebné Java objekty automaticky pri zostavení aplikácie.

[7] `https://spring.io/projects/spring-boot`

[8] `https://yidongnan.github.io/grpc-spring-boot-starter`

[9] `https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator`

[10] `https://maven.apache.org/`

[11] `https://www.xolstice.org/protobuf-maven-plugin`

Kapitola 5

Prevádzkovanie aplikácie v cloude

5.1 Príprava docker obrazov

V tejto sekcii bude popísaný postup vytvorenia docker obrazov. Dockerfile pre frontend aj backend časť obsahuje zostavenie aplikácie vrátane vygenerovania kódu z definície *proto*. Pre dosiahnutie menšej veľkosti výsledného obrazu bol použitý **multi-stage build**[45], ktorý umožňuje rozdeliť Dockerfile do viacerých fáz (stages), kde každá môže využiť iný základný obraz. Zostavenie aplikácie prebieha v dočasnom obraze a do výsledného obrazu sú nakopírované len potrebné zostavené artefakty.

5.1.1 Frontend

Dockerfile pre frontend pozostáva z týchto dvoch fáz:

- **Fáza 1:** využíva obraz *grpcweb/prereqs*, ktorý obsahuje potrebné závislosti a vykonáva generovanie kódu pomocou kompilátoru *protoc* a zostavenie aplikácie pomocou *webpack*.
- **Fáza 2:** využíva Nginx obraz pre spustenie webového serveru, na ktorom beží frontend

5.1.2 Backend

Dockerfile pre backend službu pozostáva z týchto dvoch fáz:

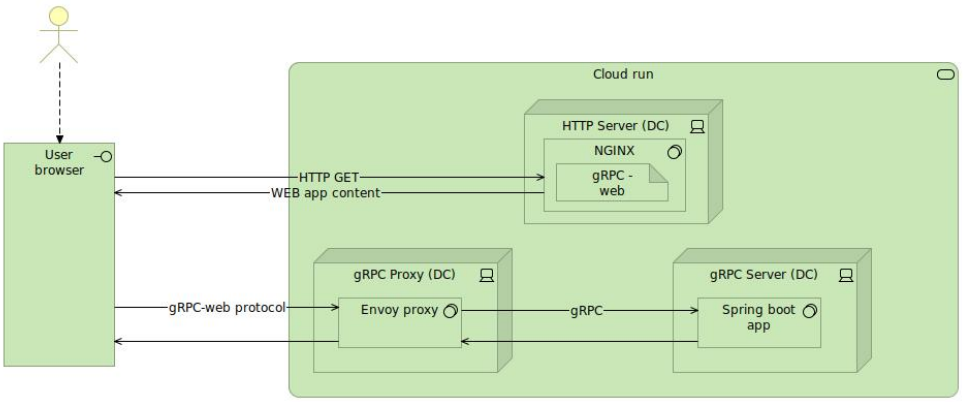
- **Fáza 1:** využíva základny obraz *maven:3.6.3-adoptopenjdk-11*, ktorý obsahuje Java Development Kit (JDK) a nástroj maven pre zostavenie aplikácie. Triedy sú z definície *proto* generované počas zostavenia aplikácie.

- **Fáza 2:** do odľahčeného obrazu, ktorý už obsahuje iba behové prostredie Java Runtime Environment je nakopírovaný výsledný *jar* archív.

5.2 Cloud Run

5.2.1 Nasadenie

Aplikácia bola nasadená na platformu Cloud Run v podobe 3 služieb, ako je zobrazené na obrázku 5.1.

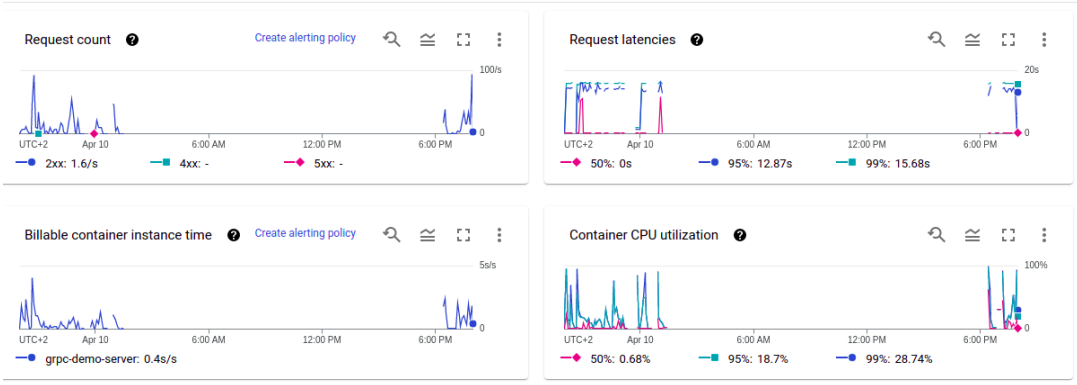


Obr. 5.1: Cloud run deployment diagram

Pre správne fungovanie gRPC serveru je potrebné povoliť `end-to-end HTTP/2` spojenie do kontajnera buď priamo v cloud konzoli alebo pri použití nástroja `gcloud` pridať prepínač `--use-http2`. Ďalším dôležitým nastavením je `request timeout`, ktorý určuje čas, za ktorý sa musí vrátiť odpoveď. Štandardne je táto hodnota nastavená na 5 minút, čo znamená, že stream bude po tejto dobe ukončený. Ak požadujeme dlhodobé streamy, je možné tento čas predĺžiť na maximálne 60 minút.

5.2.2 Monitorovanie

Cloud Run poskytuje zber niekoľkých základných metrík služieb automaticky. Metriky sú dostupné v nástroji Cloud monitoring, prípadne na predkonfigurovanom dashboarde každej služby, ako zobrazuje obrázok 5.2.



Obr. 5.2: Cloud Run dashboard s metrikami

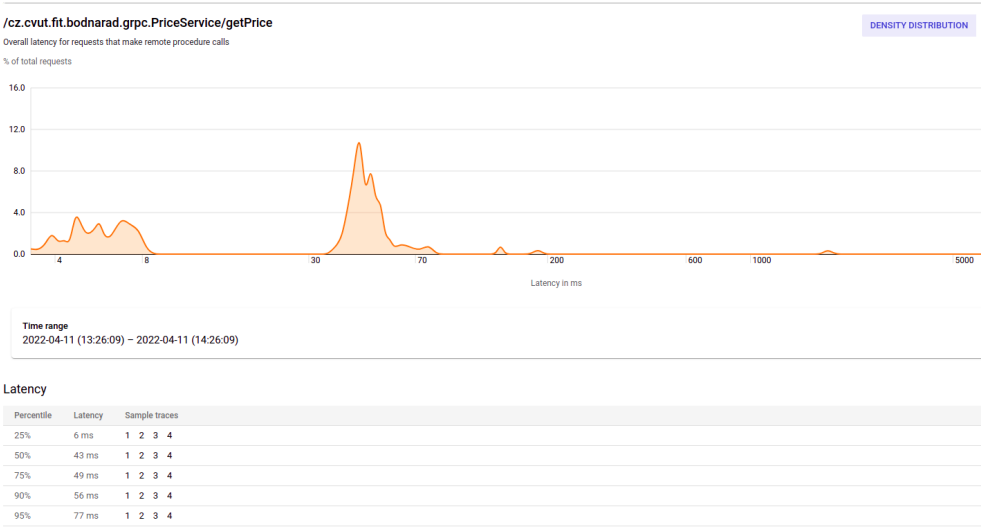
Z metrík zameraných na infraštruktúru sú napr. dostupné metriky udávajúce využívanie zdrojov CPU/RAM, latenciu pri štartovaní kontajneru, či počet využívaných inštancií kontajnerov. Okrem toho poskytuje Cloud Run metriky týkajúce sa spracovávaných požiadaviek:

- **počet prichádzajúcich požiadaviek (request count)** – pomocou tejto metriky je možné sledovať vyťaženosť systému napr. v podobe počtu požiadaviek za sekundu. Graf zobrazujúci počet požiadaviek za sekundu sa nachádza tiež na dashboarde u každej služby. Počet požiadaviek je ďalej možné filtrovať podľa kódu odpovedi. Pre HTTP požiadavky je táto možnosť dostačujúca pre sledovanie chybovosti systému. V prípade gRPC služby sú všetky odpovede označené kódom 200 i v prípade, že služba vráti gRPC chybový kód. Nie je tak možné rozlíšiť úspešné gRPC požiadavky od neúspešných.

- **latencia požiadaviek (request latencies)** – metrika zobrazuje celkovú latenciu požiadaviek nezávisle na tom, či sa jedná o unarne požiadavky alebo streamy. Keďže v prípade streamu závisí dĺžka vykonania požiadavky na dĺžke trvania streamu, táto metrika nezodpovedá skutočnej reakčnej dobe systému.

Pre zistenie latencie gRPC požiadaviek typu `unary`, je možné použiť službu Cloud Trace[12]. Tá poskytuje reporty pre analýzu latencie, v rámci ktorých je možné filtrovať požiadavky podľa URI. Graf na obrázku 5.3 zobrazuje distribúciu latencie pre požiadavky typu `unary`, vrátane rozdelenia na percentily 25, 50, 75, 90 a 95.

[12] `https://cloud.google.com/trace`



Obr. 5.3: Graf latencie v reporte z Cloud Trace

5.2.3 Údržba riešenia

Prevádzkovanie aplikácií na plne spravovanej platforme Cloud Run nevyžaduje správu infraštruktúry, typy serverov, zdrojov, či operačného systému. Verzia prostredia a všetky aktualizácie sú v réžií dátového centra. Stačí nám nasadiť kontajnerovú aplikáciu a o zvyšok sa nemusíme starať, platforma za nás taktiež zaistí automatické škálovanie. V prípade predstavenej gRPC-Web aplikácie je nutná len prípadná aktualizácia Envoy proxy.

5.2.4 Náklady na prevádzku

U riešenia prevádzkovaného na platforme Cloud Run je možné využitie modelu, kde platíme za zdroje, len keď sú využívané, alebo modelu kde sú inštancie kontajnerov trvale zapnuté. Ceny boli vypočítané na základe predpokladanej návštevnosti 1 milión používateľov denne, čo odpovedá asi 5 miliónom API požiadaviek.

Tabuľka 5.1 predstavuje mesačnú kalkuláciu s predpokladanou návštevnosťou milión používateľov denne v režime, keď platíme za zdroje, len keď sú využívané. Celková cena takto činí **257,23 EUR**.

Pre porovnanie sa pozrieme na kalkuláciu v režime kde sú inštancie kontajnerov trvale zapnuté. Tá je zobrazená v tabuľke 5.2. Predpokladáme, že pri maximálnej vyťaženosti budú využité 2 inštancie servera a proxy. Vďaka tomu, že v druhom prípade odpadá povinnosť platiť za počet vykonaných požiadaviek, sa dostávame skoro na 40% pôvodnej ceny. Pri takejto návštevnosti sa už oplatí využiť trvalé zapnuté inštancie kontajnerov hlavne u služieb,

ktoré poskytujú API, teda **gRPC server** a **gRPC proxy**. V prípade služby **gRPC web** môžeme kľudne zostať u prvého modelu, keďže v tomto prípade naštartovanie kontajneru nezaberie skoro žiaden čas.

Výsledná cena pri predpokladanej návštevnosti milión používateľov denne s maximálne 2000 súbežnými API požiadavkami činí **115,15 EUR**.

	gRPC server	gRPC proxy	gRPC web
CPU	1 vCPU	1 vCPU	1 vCPU
RAM	2 GiB	0.5 GiB	0.5 GiB
Počet požiadaviek	150 mil	150 mil.	30 mil.
Max. concurrency	1000	1000	1000
Cena	128,89 EUR	118,26 EUR	10,08 EUR

Tabuľka 5.1: Mesačná kalkulácia nákladov na prevádzku na platforme Cloud Run v režime, keď platíme za zdroje, len keď sú využívané.

	gRPC server	gRPC proxy	gRPC web
CPU	1 vCPU	1 vCPU	1 vCPU
RAM	2 GiB	0.5 GiB	0.5 GiB
Počet inštancií pri max. vyťaženosti	2	2	1
Cena	56,79 EUR	48,28 EUR	21,32 EUR

Tabuľka 5.2: Mesačná kalkulácia nákladov na prevádzku na platforme Cloud Run v režime, kde sú inštancie kontrajnerov trvale zapnuté.

5.3 Google Kubernetes Engine

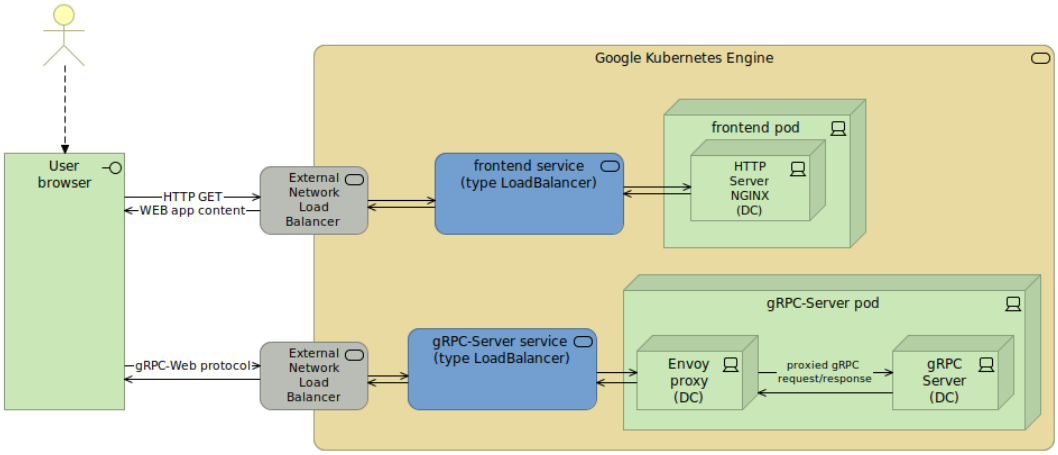
5.3.1 Nasadenie

Aplikácia bola nasadená na platformu GKE v podobe 2 služieb, ako zobrazuje diagram na obrázku 5.4. V tomto prípade nie je Envoy proxy ako samostatná služba, ale bola nasadená ako ďalší kontajner do podu spolu s backend serverom podľa vzoru ambasádor[46].

Cieľom vzoru ambasádor je skryť zložitosť primárneho kontajnera a poskytnúť jednotné rozhranie, prostredníctvom ktorého môže primárny kontajner pristupovať k službám mimo Podu. Kontajnery sú spúšťané paralelne a ambasádor kontajner tak zdieľa s aplikáciou kompletný životný cyklus.

Pre vystavenie služieb do internetu bol použitý typ služieb `LoadBalancer`, ktorý zaistí vytvorenie Network Load Balanceru[13], a vygeneruje IP adresu, ktorá bude presmerovávať všetku prevádzku na službu.

[13] `https://cloud.google.com/load-balancing/docs/network`



Obr. 5.4: GKE deployment diagram

5.3.2 Monitorovanie

Pre monitorovanie gRPC aplikácií v prostredí Kubernetes môžeme využiť to, že všetky požiadavky smerujúce na gRPC službu prechádzajú cez Envoy proxy. Envoy vedie podrobné štatistiky o prichádzajúcich/odchádzajúcich požiadavkách, ktoré umožňuje vystaviť vo viacerých formátoch. Jedným z týchto formátov je aj formát `prometheus`.

Prometheus[14] je jeden z najvyužívanejších nástrojov pre monitorovanie aplikácií v prostredí Kubernetes. Jedná sa o open source nástroj, ktorý umožňuje zber a ukladanie metrík, ktoré služba vystavuje. Pomocou nástroja Prometheus môžeme v pravidelných intervaloch získavať metriky, ktoré poskytuje Envoy proxy.

Metriky vystavuje Envoy cez endpoint administratívneho rozhrania, konkrétny formát pre Prometheus je dostupný na ceste `/stats/prometheus`. Aj keď Envoy zhromažďuje a vystavuje niektoré metriky automaticky, v prípade gRPC metrík je potrebné explicitne povoliť zber dát o gRPC požiadavkách pridaním špeciálneho filtra[47] do Envoy konfigurácie. Štandardne sú metriky zbierané na úrovni služby, bez rozlišovania metód služby. Kód 4 predstavuje konfiguráciu pre zber metrík na úrovni jednotlivých metód spolu so zaznamenávaním doby spracovania požiadaviek.

[14] `https://prometheus.io/`

```
http_filters:
- name: envoy.filters.http.grpc_stats
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.filters.http.grpc_stats.v3.FilterConfig
     stats_for_all_methods: true
     enable_upstream_stats: true
```

Výpis kódu 4: Použitie gRPC stats filtra

Filter nam umožní sledovať metriky ako:

- **počet vykonaných požiadaviek** – dostupné sú metriky pre počet všetkých vykonaných požiadaviek, počet úspešných a počet neúspešných požiadaviek
- **počet prijatých/odoslaných správ v streame**
- **čas spracovania požiadavky**

Namerané dáta je možné spracovávať a agregovať pomocou výkonného dopytovacieho jazyka PromQL. Metriky je možné zoskupiť podľa názvu gRPC metódy a sledovať počet volaní jednotlivých metód, či dĺžku spracovania v prípade metód typu `unary`. Prometheus je možné integrovať s Cloud Monitoring, kde je možné nazbierané metriky zobrazovať a pozorovať.

5.3.3 Údržba riešenia

Prevádzkovanie aplikácie v prostredí Kubernetes vyžaduje jeho pravidelnú aktualizáciu. Kubernetes Open Source Software (OSS) vydáva 3 krát do roka novú minor verziu. Google poskytuje celkovo 14 mesiacov podpory pre každú minor verziu po jej sprístupnení.[48] Tím GKE vykonáva automatické aktualizácie control plane časti klastra. Na základe toho je potrebné vykonávať aktualizácie uzlov. V prípade režimu Autopilot sa toto deje automaticky pri aktualizácií control plane a nie je možné to zmeniť. V prípade režimu Standard je možné aktualizácie spravovať alebo povoliť automatické aktualizácie.

Podobne ako pre platformu Cloud Run, aj v tomto prípade je potrebná správa a prípadne aktualizácia Envoy proxy.

5.3.4 Náklady na prevádzku

Podobne ako pre platformu Cloud Run boli vypočítané náklady na prevádzku s predpokladanou návštevnosťou milión používateľov denne. Opäť využijeme dve inštancie serveru, kde jedna inštancia zvládne obslúžiť asi 1000 súbežných požiadaviek. Potrebné odhadované zdroje sú v tabuľke 5.3. Rozdiel oproti platforme Cloud Run je tá, že nemusíme počítať s minimálnou požiadavkou na zdroje v podobe 1 vCPU. Na základe toho boli pre klaster zvolené 2 uzly s typom stroja `n1-standard-2`, ktorý obsahuje 2 vCPUs a 7,5 GB RAM. Pre každý GCP účet je prvý klaster zdarma, takže poplatok za správu môžeme vynechať.

Celková mesačná kalkulácia predstavuje:

- **Uzly (2) x n1-standard-2:** 112,56 EUR (2x 56,28)
- **Network load balancing:** 19,80 EUR
- **Celkom:** 132,36 EUR

	gRPC server	Envoy proxy	frontend	Celkovo
CPU	1 vCPU	0.8 vCPU	0.3 vCPU	3,9 vCPU
RAM	2 GB	0.6 GB	0.4 GB	5.6 GB RAM
Počet replík	2	2	1	

Tabuľka 5.3: Odhadované zdroje pre prevádzku na platforme GKE.

5.4 Google Kubernetes Engine s Anthos service mesh

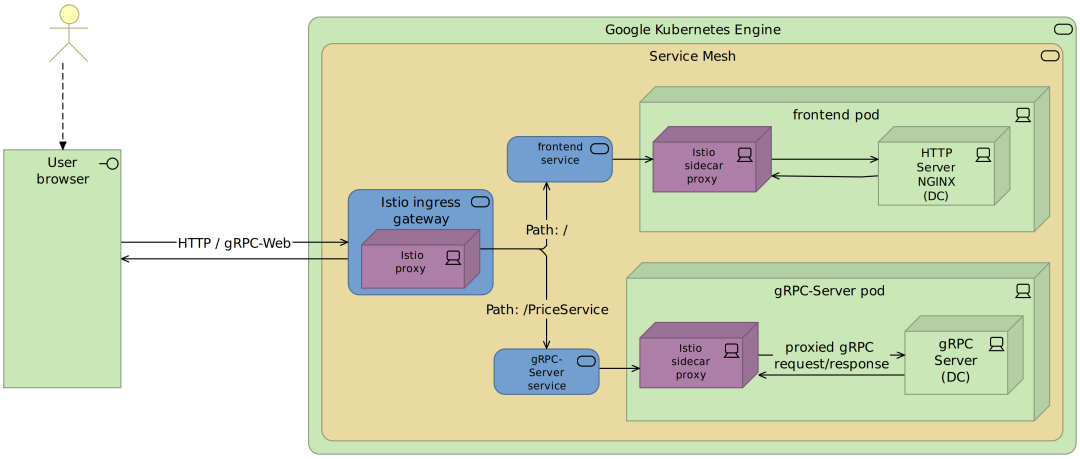
5.4.1 Nasadenie

Motivácia pre použitie Istio, prípadne Anthos Service Mesh je hlavne tá, že ako sidecar proxy využíva **Envoy**. Keďže Envoy poskytuje podporu pre **gRPC-Web**, odpadá nutnosť spravovať vlastnú proxy ako ďalší kontajner, či službu vedľa gRPC serveru. Control plane za nás totiž automaticky nakonfiguruje Envoy proxy spolu s gRPC-Web filtrom. Pre aplikovanie gRPC-Web filtra je potrebné v definícii Kubernetes služby pomenovať port začínajúc na `grpc-web`, ako je v ukážke kódu 5.

Diagram nasadenia je zobrazený na obrázku 5.5. Webový klient odosiela požiadavky na Istio ingress gateway, ktorá filtruje požiadavky na základe URI. Jedná sa o ďalšiu Envoy proxy, ktorá slúži ako Load Balancer pre prichádzajúcu prevádzku. Požiadavky sú z gateway smerované na sidecar proxy danej služby a v prípade gRPC serveru prekladané na gRPC protokol.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: grpc-server
spec:
  type: ClusterIP
  selector:
    app: grpc-server
  ports:
  - name: grpc-web-api
    protocol: TCP
    port: 8080
    targetPort: 9090
```

Výpis kódu 5: Port gRPC-server služby



Obr. 5.5: Diagram nasadenia na Google Kubernetes Engine s Anthos Service Mesh

5.4.1.1 Sprevádzkovanie service mesh

Pred nasadením by sme sa mali uistiť, že náš GKE klaster beží na podporovanej verzii a obsahuje dostatočný výpočetný výkon. Anthos Service Mesh vyžaduje minimálne 8 vCPU.[49] Ak sa jedná o inštaláciu na privátny klaster, je nutné otvoriť port `15017` používaný webhookom, ktorý spravuje sidecar proxies.

Anthos service mesh je možné nainštalovať pomocou CLI nástroja **asmcli**[15]. Pri použití prepínača `--enable_all` za nás nástroj povolí potrebné role, Google APIs a nakonfiguruje kluster pre použitie so service mesh. Posledným dôležitým krokom je povoliť automatic sidecar injection. Na Kubernetes namespace, v ktorom nám bežia služby stačí nastaviť label `istio.io/rev=$REVISION`, kde `REVISION` označuje nasadenú verziu ASM[49]. Po tomto bude každý novovytvorený pod obsahovať sidecar proxy (existujúce pody je treba reštartovať).

5.4.1.2 Ingress gateway a smerovanie prevádzky

Ingress gateway je vstupný bod pre prevádzku prichádzajúcu do klastra. Ingress gateway je potrebné nasadiť ako ďalšiu Kubernetes aplikáciu, ktorá je vystavená cez Kubernetes službu typu `LoadBalancer`. Ako správny postup je z bezpečnostných dôvodov podľa Istio dokumentácie[50] odporúčané nasadiť gateway do iného namespace než toho, kde beží control plane. Ingress gateway je Envoy proxy, ktorá je konfigurovateľná pomocou zdrojov `Gateway`, `VirtualService` a `DestinationRule`. Pri aplikácií týchto zdrojov, Kubernetes API server vytvorí udalosť, na základe ktorej Istio control plane aplikuje novú konfiguráciu na Envoy.

Kód 6 predstavuje konfiguráciu Gateway, na základe ktorej Ingress gateway akceptuje prevádzku prichádzajúcu na host `grpc-demo.com` a port `443` a vynucuje TLS pripojenie. Konfigurácia Gateway je k podu Ingress gateway priradená na základe `selectoru`. Atribút `credentialName` odkazuje na Kubernetes secret, kde je uložený certifikát.

Zdroj `VirtualService` dáva Ingress gateway informácie, ako smerovať požiadavky, ktoré boli povolené do klastra. K jednej Ingress gateway je možné pripojiť viacero `VirtualServices`, ktoré odkazujú na Kubernetes služby kam je podľa pravidiel odosielaná prevádzka.

[15] `https://cloud.google.com/service-mesh/docs/unified-install/asmcli-overview`

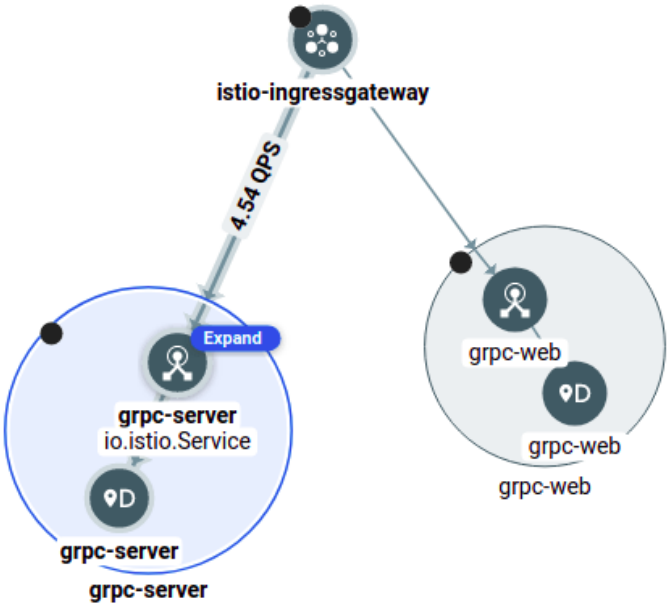
```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: grpc-gateway
spec:
  selector:
    istio: ingressgateway # use istio ingress gateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: istio-ingressgateway-certs
    hosts:
    - grpc-demo.com
```

Výpis kódu 6: Definícia zdroju Gateway

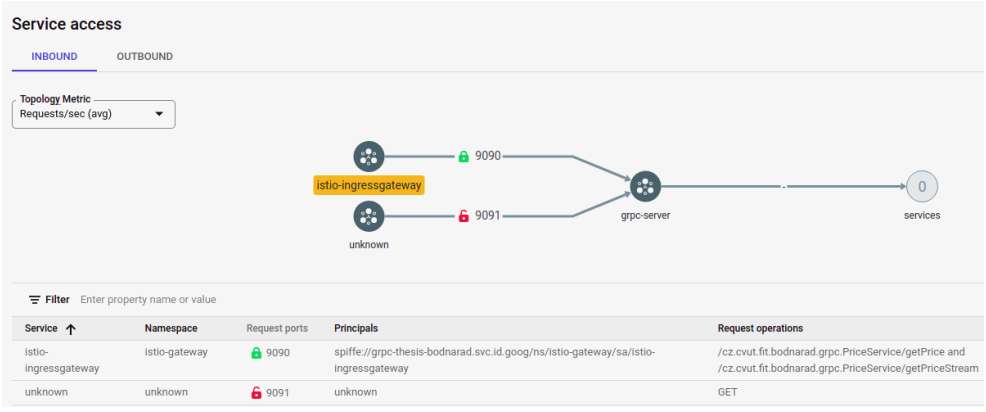
5.4.2 Monitorovanie

Jednou z najväčších výhod použitia service mesh je rozšírenie možnosti pozorovania systému. Anthos Service Mesh automaticky detekuje, ktoré služby medzi sebou komunikujú a umožňuje vizualizáciu vzťahov medzi službami. Obrázok 5.6 zobrazuje topológiu service mesh siete implementovanej aplikácie. V tomto prípade je topológia jednoduchá, no so zvyšovaním počtu mikroslužieb je detailný prehľad vzťahov veľmi praktickou funkciou. Na zobrazenej topológii je možné taktiež sledovať relatívny objem požiadaviek, ktoré každá služba dostáva vrátane informácie odkiaľ požiadavky prichádzajú.

Vďaka detailnému prehľadu, ktorý je zobrazený na obrázku 5.7 máme k dispozícii kompletný zoznam všetkých služieb, ktoré sa pripájajú k vybranej službe alebo zoznam služieb, ku ktorým sa vybraná služba pripája. Zobrazené sú tiež porty, cez ktoré služba komunikuje, aké požiadavky prichádzajú od ktorej služby a či je komunikácia medzi službami zabezpečená alebo nie. ASM automaticky poskytuje TLS certifikáty pre všetky služby, ktoré sú súčasťou service mesh.



Obr. 5.6: Prehľad service mesh topológie



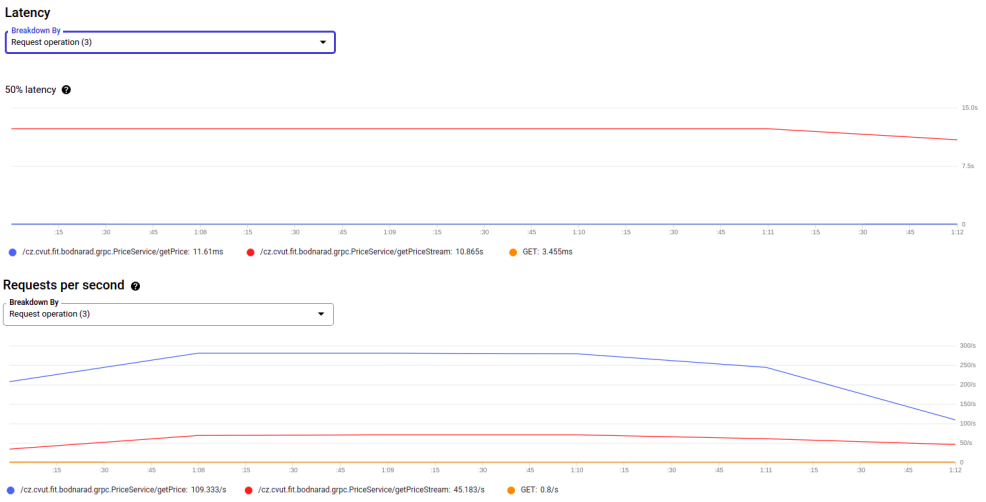
Obr. 5.7: Diagram pripojených služieb

Keďže Anthos Service Mesh zhromažďuje a agreguje údaje o každej požiadavke a odpovedi, nie je nutná implementácia alebo nastavenie zberu metrík na strane aplikácie. Okrem automatického zberu metrík poskytuje pre každú

službu predkonfigurovaný dashboard s grafmi zobrazujúcimi všetky požadované informácie pre sledovanie stavu aplikácie:

- počet požiadaviek za sekundu,
- miera chybových odpovedí,
- reakčný čas odpovedí (latencia),
- veľkosť prichádzajúcich požiadaviek a odpovedí,
- využitie CPU, pamäte.

Metriky je možné taktiež rozpadnúť a pozorovať podľa toho aká služba vykonávala požiadavku, aký bol status odpovede, použitý protokol (v prípade, že služba podporuje HTTP(S) i gRPC), či ktorá metóda služby bola zavolaná. Na obrázku 5.8 sú grafy zobrazujúce latenciu a počet prichádzajúcich požiadaviek rozdelené podľa volanej metódy. Týmto je možné odlíšiť `unary` požiadavky od streamov, ktoré nám v prípade sledovania celkovej hodnoty latencie dávajú falošný obraz o reakčnej dobe služby.



Obr. 5.8: Grafy zobrazujúce latenciu a počet prichádzajúcich požiadaviek

Anthos Service Mesh ponúka tiež možnosť sledovať výkon služieb pomocou definovania Service Level Indicators (SLIs)[16] a Service Level Objectives

[16]SLI je definovaná kvantitatívna metrika určitého aspektu úrovne poskytovaných služieb.[51]

(SLOs)[17] pre jednotlivé služby. Ukazovatele SLI je možné nastaviť na základe vopred definovaných metrík, ako je dostupnosť (z angl. availability, počet úspešných požiadaviek za určité časové obdobie) alebo latencia (koľko odpovedí bolo rovnakých alebo rýchlejších ako minimálna požadovaná latencia). Taktiež je možné využiť vlastné metriky. V tomto prípade sme použili vlastnú metriku, ktorá je na obrázku 5.9 pre sledovanie latencie čisto pre požiadavky typu `unary`, u ktorých požadujeme rýchlosť odpovede do 200 ms, namiesto sledovania celkovej latencie spolu so stream požiadavkami.

Na základe týchto SLO je možné nastaviť alerty. Ide o veľmi užitočnú funkciu, najmä v prípade aplikácií, ktoré sú citlivé na latenciu alebo musia dodržiavať určité nefunkčné požiadavky.

Obr. 5.9: Nastavenie SLO a SLI v Anthos Service Mesh

5.4.3 Údržba riešenia

Anthos Service Mesh je možné prevádzkovať 2 spôsobmi:

- **managed control plane** – jedná sa o plne spravovanú službu od Googlu. Google sa stará o správu control plane časti, rieši aktualizácie verzií, zabezpečenie a škálovanie. Control plane komponenty sú automaticky škálované na nulu, ak data plane neobsahuje žiadne komponenty.

[17] SLO je cieľová hodnota alebo rozsah hodnôt pre úroveň služby, ktorá sa meria pomocou SLI.[51]

- **in-cluster control plane** – v prípade tejto varianty je inštalácia a následna aktualizácia control plane časti plne v našej réžií. Aktualizácia sa rovnako ako inštalácia vykonáva pomocou spomínaného nástroja `asmcli`.

5.4.4 Náklady na prevádzku

Požiadavkou pre správne fungovanie Anthos Service Mesh je, aby klaster obsahoval aspoň 8 vCPUs. Opäť použijeme typ stroja `n1-standard-2`, ktorý obsahuje 2 vCPUs a 7,5 GB RAM. Klaster bude pozostávať zo 4 uzlov. Tieto zdroje budú dostatočné pre služby spojené so service mesh i s našou aplikáciou tak, aby zvládala obslúžiť milión používateľov denne a maximálne 2000 súbežných požiadaviek. Celková mesačná kalkulácia predstavuje:

- **Uzly (4) x n1-standard-2:** 225,12 EUR (4 x 56,28)
- **Network load balancing:** 19,80 EUR
- **Anthos predplatné (8 vCPUs):** 43.2 EUR (8 x 5,40)
- **Celkom:** 377,28 EUR

5.5 Zhrnutie

Monitorovanie

Z pohľadu monitorovania poskytuje najviac možností pre pozorovanie systému využitie **Anthos Service Mesh (ASM)**, kde máme detailný prehľad o komunikácií medzi mikroslužbami. Metriky sú automaticky zbierané pre všetky služby, ktoré sú súčasťou service mesh, bez nutnosti čokoľvek nastavovať. Na stránkach Anthos Service Mesh sú detailne zobrazené metriky pre sledovanie 3 zo 4 signálov známych ako *„golden signals of monitoring“*[51]: latency, traffic a errors.

Na platforme **GKE** bol predstavený zber metrík z Envoy proxy pomocou nástroja Prometheus. Tieto metriky poskytujú dostatočný prehľad o gRPC požiadavkách, vrátane gRPC status kódov a možnosti rozlíšiť požiadavky podľa volanej metódy. Umožňuje sledovať latenciu, počet prichádzajúcich požiadaviek aj mieru neúspešných požiadaviek. Oproti riešeniu s využitím ASM je nutná vlastná správa Prometheus serveru.

Platforma **Cloud Run** poskytuje automatický zber metrík. Umožňuje sledovať latenciu a počet prichádzajúcich požiadaviek. Metriky neposkytujú informácie o gRPC status kódoch ako v prípade predchádzajúcich dvoch riešení. Keďže má každá požiadavka status odpovedi 200, nie je možné určiť reálnu mieru neúspešných požiadaviek.

Údržba riešenia

Z pohľadu údržby riešenia a prípadných aktualizácií poskytuje Google pre každé z 3 riešení možnosť automatickej správy aktualizácií. V prípade Cloud Run nemáme možnosť výberu a všetky aktualizácie sú v réžií datového centra. Na platforme GKE máme možnosť vlastnej správy verzie Kubernetes alebo sa prihlásiť k automatickým aktualizáciám. Podobne je to aj s ASM, kde aktualizácie control plane časti môžeme vykonávať sami alebo využiť spravovanú službu.

V prípade prevádzkovania na platforme Cloud Run a GKE je navyše nutné spravovať vlastnú Envoy proxy. V prípade použitia Anthos Service Mesh táto starosť odpadá.

Náklady na prevádzku

U každého riešenia bola vypočítaná kalkulácia pri predpokladanej záťaži milión používateľov denne, a tak aby aplikácia dokázala obslúžiť 2000 súbežných požiadaviek. Zhrnutie cien pre všetky platformy:

- **Cloud Run**: 115,15 EUR
- **GKE**: 132,36 EUR
- **GKE s Anthos Service Mesh**: 377,28 EUR

Kapitola 6

Nasadenie stratégiou canary

6.1 Cloud Run Release Manager

Pre nasadenie na platformu Cloud Run stratégiou canary bol použitý nástroj Cloud Run Release Manager[52]. Jedná sa o open source nástroj umožňujúci plne automatizované nasadenie stratégiou canary. Využíva natívnu funkciu Cloud Run pre postupné presúvanie prevádzky zo starej verzie služby na novú verziu. Počas celého procesu nasadenia v pravidelných intervaloch získava metriky a kontroluje zdravie služby.

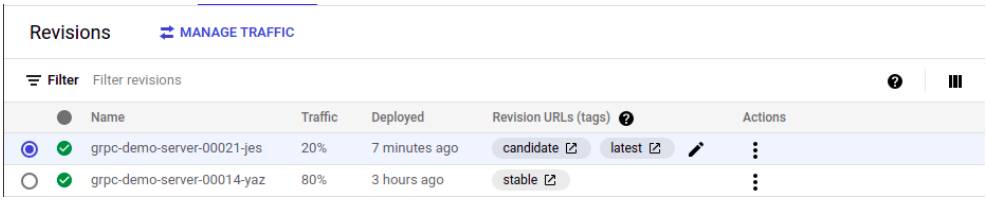
Cloud Run Release Manager je potrebné nasadiť ako ďalšiu Cloud Run službu, ktorá je spúšťaná na základe vytvoreného cron jobu v Cloud scheduler[18]. Nejedná sa teda o zabudovanú funkciu Cloud Run ani o produkt dostupný priamo v GCP. Scenár pre nasadenie sa definuje pomocou argumentov pri nasadení služby Cloud Run Release Manager.

Celý proces nasadenia je možné sledovať priamo v cloud konzole v zozname revízií služby. Pri nasadení novej verzie stačí nastaviť na revíziu špeciálny štítok `rollout-strategy=gradual` a nasadiť službu s možnosťou `--no-traffic`, čo znamená, že žiadne požiadavky nebudú na novú verziu smerované. Tento krok je zobrazený na obrázku 6.1.

Obr. 6.1: Zoznam revízií po nasadení na platformu Cloud Run

[18] `https://cloud.google.com/scheduler`

Release manager na základe štítku detekuje, že bola nová verzia nasadená a označí ju ako *candidate*. Následne začne postupne presmerovávať prevádzku z revízie *stable* na revíziu *candidate*, tak ako je možné vidieť na obrázku 6.2. Po tom, ako bola úspešne presmerovaná celá prevádzka na novú verziu, je nová verzia označená ako *stable*.



Obr. 6.2: Prebiehajúci proces nasadenia stratégiou canary na platforme Cloud Run

Výhodou tohoto nástroja je jeho jednoduchosť, vrátane procesu sprevádzkovania a taktiež to, že dokáže spravovať nasadenia pre viaceré Cloud Run služby v rámci projektu. Poskytuje dostatočné možnosti konfigurácie pomocou argumentov, ktoré je možné špecifikovať pri nasadení nástroja. Nevýhodou je to, že sa jedná o experimentálny nástroj a môžeme naraziť na problémy pri použití v produkcii.

6.2 Argo Rollouts

Nasadenie na platformu GKE (bez i s Anthos Service Mesh) bolo implementované s využitím nástroja **Argo Rollouts**[19], ktorý je súčasťou open source projektu Argo. Jedná sa o Kubernetes controller určený pre postupné nasadzovanie softvéru stratégiou *blue-green* alebo *canary*.

Princíp použitia spočíva v nahradení zdroja Kubernetes Deployment špeciálnym zdrojom Rollout[20]. YAML definícia pre Rollout je veľmi podobná definícii pre Deployment, obsahuje navyše niekoľko atribútov pre špecifikáciu stratégie nasadenia. Podobne ako Deployment, aj Rollout vytvára replicaset.

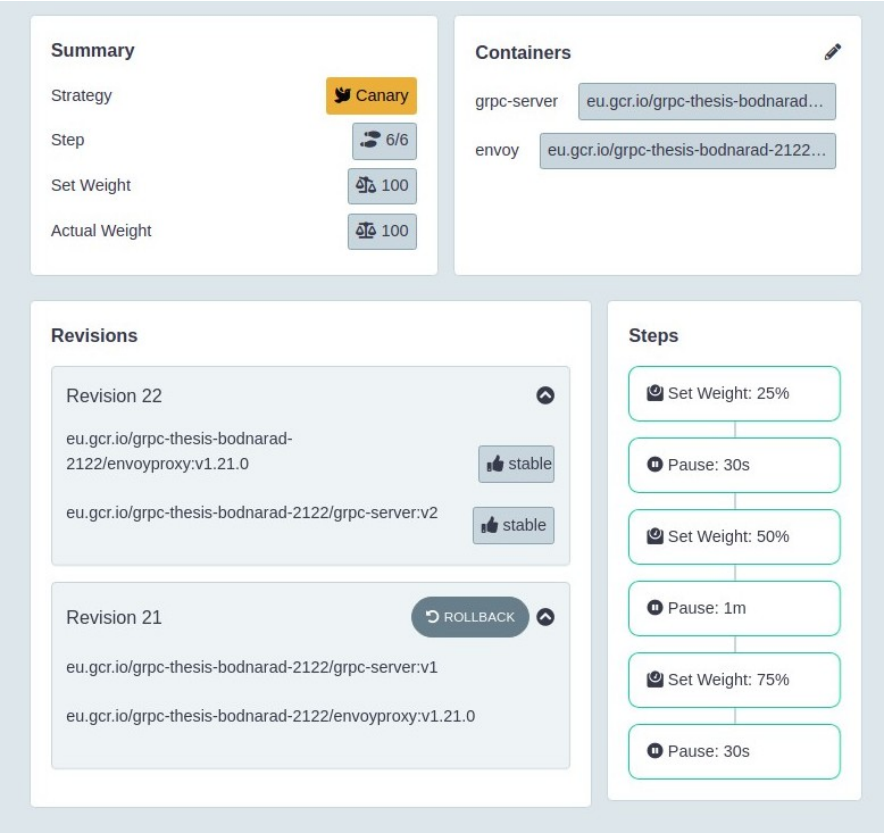
Kód 7 zobrazuje časť YAML špecifikácie pre Rollout, ktorá definuje canary nasadenie. Scenár nasadenia pozostáva zo 4 krokov definovaných v `steps`. Po prvom kroku, keď je časť prevádzky presmerovaná na novú verziu aplikácie, je na pozadí spustená analýza. O analýze bližšie pojednáva sekcia 6.2.1. Neúspešné/chybné vykonanie analýzy spôsobí, že sa proces nasadenia preruší a vykoná sa takzvaný *rollback* – vrátenie sa k starej verzii aplikácie. Týmto spôsobom je možné dosiahnúť postupné, kontrolované nasadenie novej verzie.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: grpc-server
  labels:
    app: grpc-server
spec:
  strategy:
    canary:
      analysis:
        templates:
        - templateName: success-rate
        startingStep: 2
        args:
        - name: service-name
          value: grpc-server-canary.default.svc.cluster.local
      steps:
      - setWeight: 20
      - pause:
          duration: "5m"
      - setWeight: 50
      - pause:
          duration: "5m"
```

Výpis kódu 7: Časť definície pre Rollout

[19] https://argoproj.github.io/argo-rollouts/
[20] https://argoproj.github.io/argo-rollouts/features/specification/

Argo Rollouts poskytuje plugin pre kubectl, pomocou ktorého je možné nasadenie spustiť a sledovať jeho proces priamo v príkazovom riadku alebo s využitím uživateľského rozhrania zobrazeného na obrázku 6.3.



Obr. 6.3: Argo rollout UI

6.2.1 Smerovanie prevádzky počas nasadenia

Pri použití Argo rollouts spolu s Anthos Service Mesh je využívaný zdroj *Virtual Service*, ktorý definuje pravidlá smerovania prevádzky. V definícií *Virtual Service*, viď. kód 8 je odkaz na 2 služby – `grpc-server`, ktorá odkazuje na stabilnú verziu služby a `grpc-server-canary`, ktorá odkazuje na verziu služby, ktorá sa bude nasadzovať. Na začiatku je celá prevádzka smerovaná na službu `grpc-server`. Argo Rollouts priebežne počas nasadenia upravuje hodnoty váh vo *Virtual Service* na základe definovaných krokov v špecifikácii Rollout. Ak máme napríklad krok s príkazom `setWeight: 20`, nastaví sa váha u canary služby na 20, čím sa presmeruje 20% prevádzky.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: grpc-vs
spec:
  hosts:
  - "*"
  gateways:
  - grpc-gateway
  http:
  - name: primary
    match:
     - uri:
        prefix: /cz.cvut.fit.bodnarad.grpc.PriceService
    route:
     - destination:
         host: grpc-server
         port:
           number: 8080
       weight: 100
     - destination:
         host: grpc-server-canary
         port:
           number: 8080
       weight: 0
```

Výpis kódu 8: Definícia Virtual Service pre Argo Rollouts

Nasadenie na GKE stratégiou canary je možné implemenovať aj bez použitia Anthos Service Mesh a jeho funkcie smerovania prevádzky. Nevýhoda tohoto prístupu je, že prevádzka bude rozdelená úmerne v závislosti na počte replík. Ak máme napríklad 5 replík, môžeme zvyšovať váhu len v násobkách 20, takže rozdelenie medzi canary a stabilnú verziu bude v pomeroch 20:80 (1/5 podov), 40:60 (2/5 podov), 60:40 (3/5 podov), 80:20 (4/5 podov). V prípade 4 replik zvyšujeme váhy v násobkách 25. Ak by sme chceli zvyšovať prevádzku v jednotkách percent, dôsledkom bude vysoká spotreba zdrojov.

6.2.2 Analýza

Analýza počas canary nasadenia typicky pozostáva z pravidelného kontrolovania určitých metrík. Argo Rollouts poskytuje možnosť definovať spôsob vykonania analýzy pomocou rôznych CRDs[21]. Jedným z týchto CRDs je *AnalysisTemplate*, ktorý obsahuje definíciu kontrolovanej metriky, intervalu kontroly

[21]Custom Resource Definition (CRD) je spôsob, ako rozšíriť Kubernetes API rozhranie o objekty, ktoré nie sú priamo pokryté v Kubernetes. CRD je podobne ako základné Kubernetes zdroje definovaný pomocou YAML.

a prahových hodnôt, na základe ktorých sa rozhoduje o úspešnosti, prípadne neúspešnosti analýzy.

V rámci práce bola využitá integrácia s nástrojom Prometheus. V prípade Anthos Service Mesh je potrebné pri inštalácií service mesh zmeniť propagáciu metrík do nástroja Prometheus namiesto Cloud Monitoring.

Definícia analýzy v kóde 9 zisťuje v 30 sekundovom intervale pomocou PromQL dopytu stav canary služby. Dopyt vráti pomer počtu úspešných gRPC požiadaviek k celkovému počtu gRPC požiadaviek za poslednú minútu. Prahová hodnota pre úspešnosť dopytu je vyjadrená pomocou atribútu `successCondition`, maximálny počet neúspešných dopytov pre vyhodnotenie analýzy ako neúspešnej vyjadruje atribút `failureLimit`. Použitá metrika `istio_requests_total` je automaticky dostupná pri použití jak Anthos Service Mesh, tak i v prípade Istio.

```
apiVersion: argoproj.io/v1alpha1
kind: AnalysisTemplate
metadata:
  name: success-rate
spec:
  args:
  - name: service-name
  metrics:
  - name: success-rate
    successCondition: result[0] >= 0.95
    interval: 30s
    failureLimit: 1
    provider:
      prometheus:
        address: http://prometheus.istio-system.svc.cluster.local:9090
        query: |
          sum(irate(
            istio_requests_total{reporter="source",
            destination_service=~"{{args.service-name}}",
            grpc_response_status!~"2|14", request_protocol=~"grpc"}[1m]
          )) /
          sum(irate(
            istio_requests_total{reporter="source",
            destination_service=~"{{args.service-name}}",
            request_protocol=~"grpc"}[1m]
          ))
```

Výpis kódu 9: Definícia AnalysisTemplate pre Argo Rollouts

Pri riešení na GKE bez Anthos Service Mesh bola implementácia analýzy veľmi podobná. Keďže v tomto prípade nie je dostupná metrika, ktorú poskytuje Istio, bola využitá metrika od Envoy. Kód 10 predstavuje PromQL dopyt, ktorý vráti pomer počtu úspešných gRPC požiadaviek k celkovému po-

čtu gRPC požiadaviek za poslednú minútu na základe metrík, ktoré poskytuje Envoy. Nevýhodou v tomto prípade je, že nie je možné rozoznať, či sa jedná o požiadavky na novú verziu alebo na starú verziu služby ako v prípade Istio metrík v predchádzajúcom príklade.

```
query: |
   sum(irate(
      envoy_cluster_grpc_success[1m]
   )) /
   sum(irate(
      envoy_cluster_grpc_total[1m]
   ))
```

Výpis kódu 10: PromQL dopyt s využitím metrík z Envoy proxy.

Kapitola 7

Testovanie

Cieľom testovania bolo overiť stabilitu riešenia pri nasadzovaní novej verzie stratégiou canary a spôsobom predstaveným v kapitole 6. Každé riešenie bolo najprv otestované bez procesu nasadenia. Následne bol test s rovnakou záťažou spustený počas nasadzovania novej verzie.

7.1 Výber testovacieho nástroja

Pri výbere nástroja bola zohľadňovaná najmä možnosť tvorby testov formou kódu a podpora testovania gRPC metód typu *unary* a *server streaming*. Výber prebiehal z troch nástrojov, ktoré poskytujú testovanie gRPC aplikácií:

- **K6**[22] – K6 je výkonný testovací nástroj, ktorý umožňuje písanie testov v jazyku JavaScript. Aktuálne nepodporuje streamy iba gRPC metódy typu *unary*.
- **ghz**[23] – Jedná sa o jednoduchý CLI nástroj. Tvorba testov spočíva v spustení nástroja s rôznymi parametrami testu formou argumentov príkazového riadku. Podporuje všetky typy gRPC metód.
- **Gatling**[24] – Nástroj postavený na jazyku Scala a frameworku Netty[25]. Umožňuje písanie testov v jazyku Scala alebo Java, pričom používa jazyk DSL[26], vďaka čomu sú testy jednoducho čitateľné. Výhodou oproti *ghz* je rozšírená funkcionalita a možnosť definovania scenára v rámci jedného testu. S použitím pluginu **Gatling-gRPC**[27] je možné testovať gRPC aplikácie. Plugin podporuje všetky typy gRPC metód. Gatling taktiež

[22] `https://k6.io/`
[23] `https://ghz.sh/`
[24] `https://gatling.io/`
[25] `https://netty.io/`
[26] Domain Specific Language (DSL) definuje jazyk špecifický pre aplikačnú doménu.
[27] `https://github.com/phiSgr/gatling-grpc`

disponuje funkciou zobrazenia výsledkov testov s podrobnými metrikami prehľadne na generovanom reporte.

Ako najvhodnejší nástroj pre testovanie sa z týchto troch ukázal nástroj **Gatling**.

7.1.1 Definícia používateľa a scenára

Scenár je reprezentácia toho, čo sa skutočne deje, keď používatelia navštívia našu aplikáciu. Bežný používateľ navštívi aplikáciu s cieľom zistiť aktuálne hodnoty kryptomien, prípadne sa chvíľu zdrží pre pozorovanie vývoja hodnoty na grafe. Pre účely testu budeme predpokladať, že používateľ strávi pozorovaním 15 sekúnd. Použitý testovací scenár pozostáva zo 4 požiadaviek typu *unary* a jednej požiadavky typu *server streaming* v dobe trvania 15 sekúnd. Medzi vykonanim každej požiadavky je vložená náhodne dlhá pauza v dobe trvania 500 až 2000 ms.

Testovací nástroj neumožňuje dostatočne otestovať prípad, kedy je stream ukončený zo strany klienta, pretože v takomto prípade nie je možné zvalidovať, že bola požiadavka úspešná. Požiadavka je považovaná za úspešnú, pokiaľ je stream ukončený zo strany servera, čo znamená, že server zašle všetky správy a status odpovedi je 0 (OK). Z toho dôvodu bol do implementácie pridaný testovací mód, ktorý upravuje funkciu streamovania tak, že namiesto nekonečného streamu je možné zaslať konečný počet odpovedí na základe zaslaného čísla v požiadavke.

7.1.2 Konfigurácia testovacieho nástroja

Nástroj Gatling poskytuje kontrolu nad vkladaním používateľov do testu. Vkladanie môže prebiehať na základe dvoch modelov – **uzavretý systém** a **otvorený systém**.

Uzavretý systém je systém, v ktorom je obmedzený počet súbežných používateľov. Nový scenár sa spustí vždy po dokončení predchádzajúceho scenára. Rýchlosť príchodu používateľov nemôžeme ovplyvniť, vyplýva z rýchlosti spracovávania požiadaviek.

Otvorený systém je systém, ktorý nemá kontrolu nad počtom súbežných používateľov. Používatelia sú pridávaní do systému na základe definovanej rýchlosti (napr. 20 uživateľov za sekundu). Po tom, čo vykonajú svoj scenár, sú ukončení. To je hlavný rozdiel oproti uzavretému systému – ak používateľ v otvorenom systéme ukončí svoj scenár, nenastáva spustenie nového. Scenár je spustený len s príchodom nového používateľa.

Vo všeobecnosti sa webové stránky riadia modelom otvoreného systému. V otvorenom systéme to funguje tak, že ak sa systém začne spomaľovať, zvýšia

sa časy odozvy, zatiaľ čo noví používatelia budú prichádzať do systému v stále rovnakej rýchlosti. Dôsledkom je, že sa môže výrazne nahromadiť počet používateľov.

Pre záťažový test bol zvolený model **uzavretého systému** s konfiguráciou:

- `constantConcurrentUsers(50).during(1200)` – Na začiatku je do simulácie vložených naraz 50 používateľov, ďalší používatelia sú vložení až keď niektorí z tých 50 ukončia test. Simulácia trvá 20 minút.

V prípade použitia modelu otvoreného systému, kde sa každú sekundu pridáva konštantný počet používateľov a kde čas spracovania jedného používateľa je niečo vyše 15 sekúnd (z dôvodu streamu), by sa za 15 sekúnd vytvoril pätnásťnásobok požadovaného počtu používateľov. Pri vkládaní 50 používateľov by celkový počet používateľov bol 750, čo už nie je v možnostiach použitého hardvéru.

7.2 Špecifikácia platformy pre spúšťanie testov

Špecifikácia platformy, na ktorej boli spúšťané záťažové testy.

Procesor	Intel® Core™ i7-8565U CPU @ 1.80GHz × 8
Operačný systém	Ubuntu 18.04.6 64-bit
Pamäť RAM	24 GB
Bandwidth	94 Mbps download, 10 Mbps upload

Tabuľka 7.1: Špecifikácia platformy

7.3 Výsledky

7.3.1 Cloud Run

Pri nasadení na platformu Cloud Run neboli namerané žiadne výpadky. Výsledky ukazujú, že všetky vykonané požiadavky skončili úspešne. Na obrázku 7.1 sú štatistiky zobrazujúce počet vykonaných požiadaviek a v prípade `unary` požiadaviek aj dobu ich trvania.

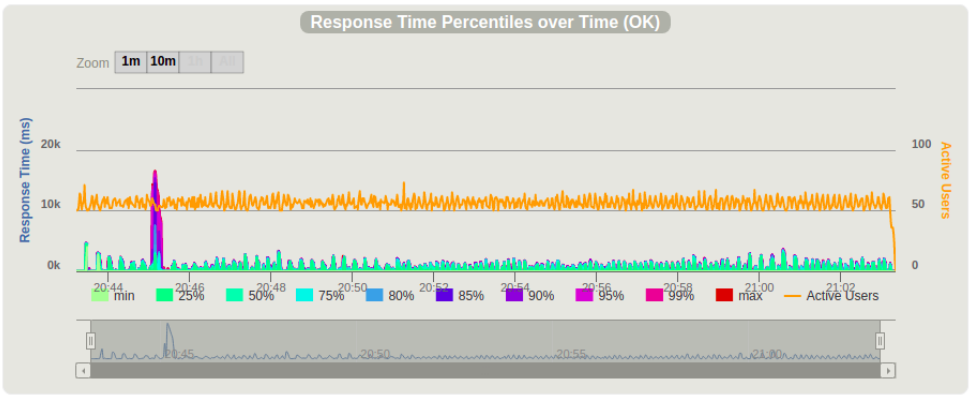
Hodnoty pre percentily 95 a 99 ukazujú, že doba trvania niektorých požiadaviek bola vyššia než by sa pre takúto jednoduchú aplikáciu predpokladalo. Tu sa nejedná o dôsledok nasadenia, podobné hodnoty boli namerané aj pri teste bez nasadenia. Rozdiel v teste pri nasadeni bol ale v tom, že zaznamenané boli aj extrémne hodnoty času odpovedí ako napríklad 16544 ms. Zaujímavé je, preto pozrieť sa na graf rozdelenia časov odpovedí počas vykonávania

testu, kde je možné pozorovať minimálny a maximálny čas požiadavky v každom okamihu testu. Na grafe na obrázku 7.2 vidíme, že tieto extrémne časy sa vyskytovali len v krátkom časovom období na začiatku. Pravdepodobne sa jednalo o prípad, keď boli požiadavky odosielané na novú verziu služby, pričom inštancia kontajnera ešte štartovala.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	37989	37989	0	0%	31.396	0	45	382	1557	2452	16544	335	664
Unary request	30460	30460	0	0%	25.174	35	52	602	1665	2596	16544	418	717
Server stream	7529	7529	0	0%	6.222	0	0	0	0	0	0	0	0

Obr. 7.1: Štatistiky vykonaných požiadaviek na platforme Cloud Run



Obr. 7.2: Graf rozdelenia časov odpovedí na platforme Cloud Run

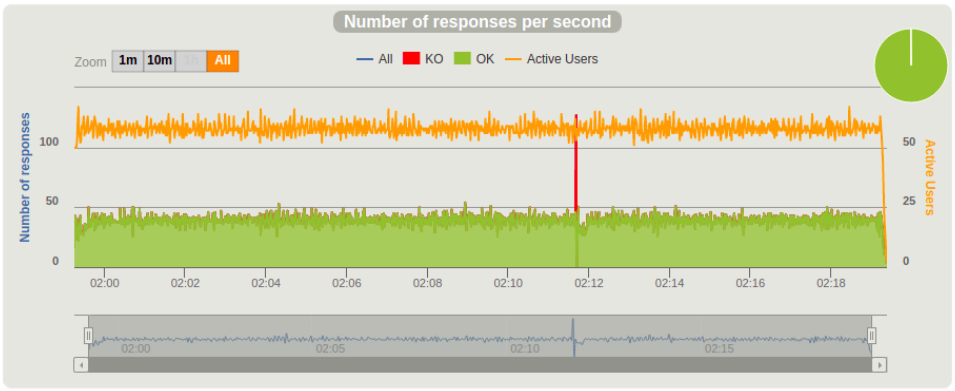
7.3.2 GKE

Pri nasadení na platformu GKE bez Anthos Service Mesh, bol zaznamenaný drobný výpadok. Šlo najmä o neúspešné požiadavky typu stream, ako ukazuje tabuľka na obrázku 7.3. Na grafe na obrázku 7.4 zobrazujúcom počet odpovedí v každom okamihu testu sú tieto neúspešné požiadavky označené červenou farbou. Výpadok trval celkom 2 sekundy a vyskytoval sa vždy na konci scenára nasadenia. Ako zobrazuje graf rozdelenia časov odpovedí na obrázku 7.5, v tomto čase bola aj dlhšia odozva pre požiadavky, ktoré neskončili chybou.

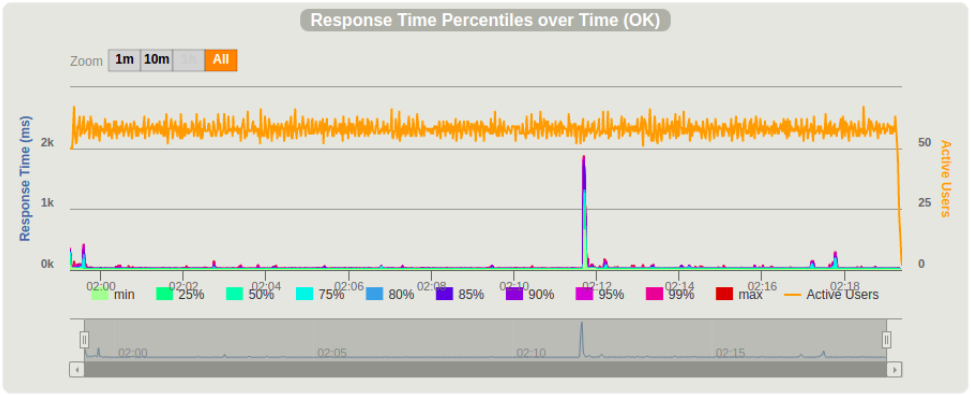
STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	47438	47320	118	0%	39.237	0	18	20	23	43	1868	17	38
Unary request	38028	38024	4	0%	31.454	13	19	20	24	53	1868	21	42
Server stream	9410	9296	114	1%	7.783	0	0	0	0	0	0	0	0

Obr. 7.3: Štatistiky vykonaných požiadaviek na platforme GKE



Obr. 7.4: Graf rozdelenia počtu odpovedí na platforme GKE



Obr. 7.5: Graf rozdelenia časov odpovedí na platforme GKE

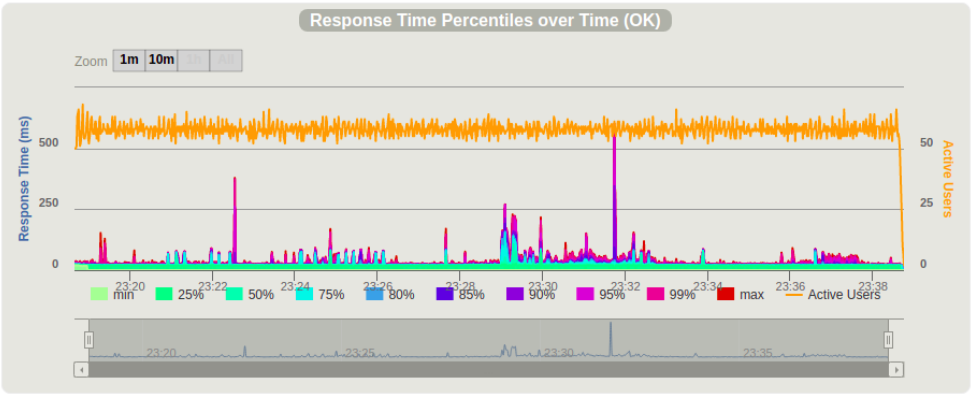
7.3.3 GKE s Anthos Service Mesh

Nasadenie na platformu GKE s Anthos Service Mesh sa aj po opakovanom testovaní ukázalo dostatočne stabilné. Všetky vykonané požiadavky boli úspešné a rýchlosť odozvy u 99% požiadaviek bola do 74 ms, ako ukazujú štatistiky na obrázku 7.7. Časy odpovedí boli zrovnateľné s výsledkami pri testovaní na platforme GKE bez Anthos Service Mesh.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	47251	47251	0	0%	39.115	0	20	22	31	74	602	18	15
Unary request	37876	37876	0	0%	31.354	13	20	22	34	77	602	23	14
Server stream	9375	9375	0	0%	7.761	0	0	0	0	0	0	0	0

Obr. 7.6: Štatistiky vykonaných požiadaviek na platforme GKE s Anthos Service Mesh



Obr. 7.7: Graf rozdelenia časov odpovedí na platforme GKE s Anthos Service Mesh

7.4 Zhrnutie

Testovanie ukázalo, že všetky tri riešenia sú dostatočne stabilné pre obsluhu požiadaviek pri záťaži 50 súbežných používateľov. Počas nasadzovania boli namerané výpadky len na platforme GKE. Jednalo sa o krátky výpadok najmä streamov, ktorý dokáže ošetriť zopakovanie požiadavky na klientovi. Na platforme Cloud Run a platforme GKE s Anthos Service Mesh prebiehalo nasadenie bez výpadkov.

Test nebolo možné vykonať pre rádovo väčší počet používateľov z dôvodu obmedzenia použitého hardvéru. Počet používateľov bol zvolený ako maximálny možný, pri ktorom neboli výsledky ovplyvnené nedostatočnými zdrojmi testového klienta. Zvažované boli aj možnosti spúšťania testov v cloude, ktoré nástroj Gatling ponúka. V rámci cenovo dostupných možností sa nedostaneme na väčší počet používateľov než v prípade lokálneho prostredia. Maximálný počet virtuálných používateľov, ktorý Gatling v cenníku udáva, je 100 000, pričom táto varianta stojí 4000 dolárov na mesiac. Podobne sa pohybujú ceny aj v prípade iných testovacích nástrojov, ktoré ponúkajú spúšťanie testov v cloude. Spúšťanie testov s miliónom virtuálných používateľov je skutočne náročné a pre tieto prípady ponúka ako Gatling, tak spomínaný nástroj K6 individuálne plány pre firmy.

Ďalej je nutné poznamenať, že testovaný bol prípad, kedy stream trvá 15 sekúnd. Test nedokazuje, že v prípade dlhšieho trvania streamu by bolo možné dosiahnúť nasadenie bez výpadku. V prípade dlhodobého streamu, ako bolo pôvodne navrhnuté, by určite došlo po čase k prerušeniu streamu pri nasadení. Tento problém ale dostatočne vyrieši implementovaná logika kde sa požiadavka zopakuje na klientovi a stream sa obnoví bez toho, aby používateľ niečo poznal. Možnosti riešenia tohto problému boli hlbšie predstavené v sekcii 2.4.2.

Aj keď Envoy proxy podporuje zopakovanie požiadavky na základe stavu odpovede, túto logiku sa v rámci práce nepodarilo dostať do funkčného stavu. gRPC využíva `HTTP trailer`[28] pre posielanie status kódov (grpc-status). Podľa Envoy dokumentácie[53] aktuálne nie je podporované zopakovanie požiadavky na základe status kódu obsiahnutého v `trailer`:

„*gRPC retry logika je v súčasnosti podporovaná len pre status kódy gRPC posielané v hlavičkách odpovedí. gRPC status kódy poslané v trailers nespustia retry logiku.*“

Ak by fungovalo zopakovanie požiadavky na strane proxy, bolo by tiež nutné aby bola proxy prevádzkovana mimo podu, v ktorom beží aplikácia.

[28] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Trailer`

Záver

Cieľom tejto diplomovej práce bolo preskúmať súčasný stav technológie gRPC, jej použitie pre tvorbu webových aplikácií a ich následnú prevádzku v cloude.

Práca prináša odpovede na niekoľko tém, kde niektoré priamo súvisia s gRPC a niektoré sú všeobecne platné pri prevádzkovaní aplikácií v cloude. V prvom rade práca oboznamuje čitateľa o špecifikách gRPC a špecifikách vývoja webových aplikácií s využitím gRPC-Web. Vzhľadom k povahe diplomovej práce nebola implementácia hlavným cieľom, ale aj tak ukazuje základy implementácie gRPC v jazyku Java s využitím Spring boot.

Práca popisuje možnosti nasadzovania nových verzií do cloudu, bližšie boli rozobrané možnosti nasadenia stratégiou blue-green a canary a ich možnosti oproti nasadeniu rolling update.

Práca sa venuje tiež cloud technológiam v prostredí Google Cloud Platform a sústredí sa na možnosti prevádzky kontajnerizovaných aplikácií. Jednotlivé spôsoby sú zhodnotené z pohľadu prevádzky, údržby, ceny a monitorovania. Informácie je možné použiť ako pre gRPC aplikácie, tak i pre bežnú aplikáciu.

Aj keď záťažové testovanie bolo len doplnkom pri overení bezvýpadkového nasadenia, môže slúžiť ako úvod do problematiky záťažového testovania gRPC aplikácií.

Zhodnotenie

V teoretickej časti práce bola predstavená technológia gRPC, gRPC-Web a jej limitácie. Priblížené boli metódy nasadenia stratégiou canary a blue-green. Diskutované bolo tiež bezvýpadkové nasadzovanie gRPC aplikácií, problémy v prípade použitia streamov a ich možné riešenia v závislosti na povahe dát.

V rámci práce bola implementovaná aplikácia demonštrujúca použitie gRPC a gRPC-Web. Aplikácia bola nasadená na platformu Google Cloud Run, Google Kubernetes Engine a Google Kubernetes Engine s Anthos Service Mesh.

V kapitole 5 boli predstavené možnosti nasadenia a prevádzky pre tieto 3 technológie cloud infraštruktúry. V rámci každej technológie boli diskutované možnosti monitorovania, čo zahŕňa údržba riešenia a aké sú náklady na prevádzku. Najjednoduchším a najlacnejším riešením je prevádzkovanie na platforme Cloud Run. Platforma Google Kubernetes Engine s využitím Anthos Service Mesh poskytuje najviac možností z pohľadu monitorovania. Jedná sa zároveň o najdrahšiu možnosť.

Kapitola 6 popisuje implementáciu nasadenia stratégiou canary, kde pre platformu Cloud Run bol využitý nástroj Cloud Run Release Manager. Pre platformu Google Kubernetes Engine bol použitý nástroj Argo Rollouts a porovnané bolo použitie s Anthos Service Mesh a bez. Následne bol tento spôsob nasadenia otestovaný záťažovými testami, ktorými sa zaoberá kapitola 7. Na platforme Cloud Run a Google Kubernetes Engine s Anthos Service Mesh sa podarilo dosiahnúť bezvýpadkové nasadenie. Na platforme Google Kubernetes Engine bez service mesh boli pozorované výpadky.

Bibliografia

1. *gRPC - Who's using this and why?* [online]. [cit. 2022-02-28]. Dostupné z : https://grpc.io/docs/what-is-grpc/faq/%5C#whosusing-this-and-why.
2. *CNCF Cloud Native Definition v1.0* [online]. [cit. 2022-02-27]. Dostupné z : https://github.com/cncf/toc/blob/main/DEFINITION.md.
3. JACK WITKOWSKI, Konstantinos Korakitis. *The state of cloud native development*. 2021-12. Dostupné tiež z: https://www.cncf.io/wp-content/uploads/2021/12/Q1-2021-State-of-Cloud-Native-development-FINAL.pdf.
4. *The Role of Service Mesh as a Cloud-Native Enabler is Building Fast* [online]. [cit. 2022-03-05]. Dostupné z : https://go.451research.com/2019-mi-service-mesh-cloud-native-enabler.html.
5. *Remote Procedure Calls (RPC)* [online]. [cit. 2022-02-20]. Dostupné z : https://users.cs.cf.ac.uk/Dave.Marshall/C/node33.html.
6. *Introduction to gRPC*. 2021. Dostupné tiež z: https://grpc.io/docs/what-is-grpc/introduction/.
7. BELSHE, Mike; PEON, Roberto; THOMSON, Martin. *Hypertext transfer protocol version 2 (HTTP/2)*. RFC 7540, 2015.
8. GRIGORIK, Ilya. *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, Inc., 2013.
9. KRASNOV, Vlad. *HPACK: the silent killer (feature) of HTTP/2* [online]. [cit. 2022-02-15]. Dostupné z : https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/.
10. ARONSSON, Simon. *Performance testing gRPC services*. 2020. Dostupné tiež z: https://k6.io/blog/performance-testing-grpc-services/.

11. INDRASIRI, Kasun; KURUPPU, Danesh. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. Ö'Reilly Media, Inc.", 2020.

12. *Protocol Buffers - Google's data interchange formath*. Dostupné tiež z: https://github.com/protocolbuffers/protobuf.

13. BRANDHORST, Johan. *The state of gRPC in the browser*. 2019. Dostupné tiež z: https://grpc.io/blog/state-of-grpc-web.

14. *gRPC-Web*. Dostupné tiež z: https://github.com/grpc/grpc-web.

15. LTD, Improbable Worlds. *gRPC-Web: Typed Frontend Development*. Dostupné tiež z: https://github.com/improbable-eng/grpc-web.

16. *XMLHttpRequest* [online]. [cit. 2022-03-10]. Dostupné z : https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

17. *Fetch API* [online]. [cit. 2022-03-10]. Dostupné z : https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

18. *Browser connection limitations*. Dostupné tiež z: https://docs.pushtechnology.com/cloud/latest/manual/html/designguide/solution/support/connection_limitations.html.

19. ZHBANKOV, Denis. *gRPC-Web via HTTP2*. Dostupné tiež z: https://medium.com/%5C%5C@denis.zhbankov/grpc-web-via-http2-b05c8c8f9e6.

20. *What are containers?* [online]. [cit. 2022-02-28]. Dostupné z : https://cloud.google.com/learn/what-are-containers.

21. GONZALEZ, David. *Implementing Modern DevOps: Enabling IT organizations to deliver faster and smarter*. Packt Publishing Ltd, 2017.

22. Using a Service to Expose Your App. In: *Kubernetes Documentation* [online]. [B.r.] [cit. 2022-04-15]. Dostupné z : https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/.

23. *The Istio service mesh* [online]. [cit. 2022-02-28]. Dostupné z : https://istio.io/latest/about/service-mesh/.

24. CALCOTE, Lee; JACKSON, Nick; BOUWER, Paul. *Service Mesh Patterns*. O'Reilly Media, 2022.

25. *What's a service mesh?* [online]. 2018. [cit. 2022-02-28]. Dostupné z : https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh.

26. HUMBLE, Jez. *What is Continuous Delivery?* [online]. [cit. 2022-03-28]. Dostupné z : https://continuousdelivery.com/.

27. Deployments - strategy. In: *Kubernetes Documentation* [online]. [B.r.] [cit. 2022-04-04]. Dostupné z : https://kubernetes.io/docs/concepts/workloads/controllers/deployment/%5C#strategy.

28. RUDRABHATLA, Chaitanya K. Comparison of zero downtime based deployment techniques in public cloud infrastructure. In: *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE, 2020, s. 1082–1086.

29. *What is serverless?* [online]. [cit. 2022-02-28]. Dostupné z : `https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless`.

30. *Knative* [online]. [cit. 2022-03-06]. Dostupné z : `https://knative.dev/docs/about/testimonials/`.

31. CHRISTOPH BUSSLER, Amina Mansour. *Google Kubernetes Engine vs Cloud Run: Which should you use?* [online]. 2019-11. [cit. 2022-02-28]. Dostupné z : `https://cloud.google.com/blog/products/containers-kubernetes/when-to-use-google-kubernetes-engine-vs-cloud-run-for-containers`.

32. *Cloud Run - Resource model* [online]. [B.r.]. [cit. 2022-03-01]. Dostupné z : `https://cloud.google.com/run/docs/resource-model`.

33. GOOGLE, Inc. About container instance autoscaling. In: *Cloud Run Documentation* [online]. [B.r.] [cit. 2022-03-14]. Dostupné z : `https://cloud.google.com/run/docs/about-instance-autoscaling%5C#idle-instance`.

34. GOOGLE, Inc. CPU limits. In: *Cloud Run Documentation* [online]. [B.r.] [cit. 2022-03-14]. Dostupné z : `https://cloud.google.com/run/docs/configuring/cpu`.

35. *The comparison of Service Mesh and how it can be a game-changer for enterprises* [online]. [B.r.]. [cit. 2022-03-01]. Dostupné z : `https://cldcvr.com/news-and-media/blog/the-comparison-of-service-mesh-and-how-it-can-be-a-game-changer-for-enterprises/`.

36. GOOGLE, Inc. About container instance autoscaling. In: *Anthos service mesh documentation* [online]. 2022 [cit. 2022-03-27]. Dostupné z : `https://cloud.google.com/service-mesh/docs/observability-overview`.

37. GOOGLE, Inc. Anthos Service Mesh by example: mTLS. In: *Service Mesh Documentation* [online]. [B.r.] [cit. 2022-03-14]. Dostupné z : `https://cloud.google.com/service-mesh/docs/by-example/mtls`.

38. *gRPC-Web* [soft.]. [B.r.]. Dostupné tiež z: `https://www.npmjs.com/package/grpc-web`.

39. Status codes and their use in gRPC. In: *GRPC Core Documentation* [online]. 2022 [cit. 2022-04-04]. Dostupné z : `https://grpc.github.io/grpc/core/md_doc_statuscodes.html`.

40. *Chart.js* [soft.]. [B.r.]. Dostupné tiež z: `https://www.chartjs.org/`.

41. *Nginx* [soft.]. [B.r.]. Dostupné tiež z: `https://nginx.org/`.

42. *Envoy proxy* [soft.]. [B.r.]. Dostupné tiež z: https://www.envoyproxy.io/.

43. Filters - gRPC Web. In: *Envoy Documentation* [online]. [B.r.] [cit. 2022-04-04]. Dostupné z : https://www.envoyproxy.io/docs/envoy/latest/api-v3/extensions/filters/http/grpc_web/v3/grpc_web.proto%5C#envoy-v3-api-msg-extensions-filters-http-grpc-web-v3-grpcweb.

44. LTD, Maruti TechLabs Pvt. *Top 16 Rapid Application Development Tools in 2021* [online]. 2022. [cit. 2022-04-03]. Dostupné z : https://marutitech.com/rapid-application-development-tools/%5C#4_Spring_Boot.

45. DOCKER, Inc. Use multi-stage builds. In: *Docker Documentation* [online]. [B.r.] [cit. 2022-03-25]. Dostupné z : https://docs.docker.com/develop/develop-images/multistage-build/.

46. IBRYAM, Bilgin; HUß, Roland. *Kubernetes Patterns*. dpunkt, 2020.

47. gRPC Statistics. In: *Envoy Documentation* [online]. [B.r.] [cit. 2022-04-10]. Dostupné z : https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/grpc_stats_filter.

48. GOOGLE, Inc. GKE versioning and support. In: *GKE Documentation* [online]. [B.r.] [cit. 2022-04-25]. Dostupné z : https://cloud.google.com/kubernetes-engine/versioning.

49. GOOGLE, Inc. Anthos prerequisites. In: *Anthos service mesh Documentation* [online]. [B.r.] [cit. 2022-03-29]. Dostupné z : https://cloud.google.com/service-mesh/docs/unified-install/anthos-service-mesh-prerequisites.

50. ISTIO. Installing Gateways. In: *Istio Documentation* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z : https://istio.io/latest/docs/setup/additional-setup/gateway/.

51. BEYER, Betsy; JONES, Chris; PETOFF, Jennifer; MURPHY, Niall Richard. *Site reliability engineering: How Google runs production systems*. O'Reilly Media, Inc., 2016.

52. *Cloud Run Release Manager* [soft.]. [B.r.]. Dostupné tiež z: https://github.com/GoogleCloudPlatform/cloud-run-release-manager.

53. Router. In: *Envoy Documentation* [online]. [B.r.] [cit. 2022-04-24]. Dostupné z : https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/router_filter.

Dodatok **A**

Zoznam použitých skratiek

HTTP Hypertext Transfer Protocol

XML Extensible markup language

REST REpresentational State Transfer.

API Application Programming Interface.

JS JavaScript.

GCP Google Cloud Platform.

GKE Google Kubernetes Engine.

ASM Anthos Service Mesh.

vCPU virtual Centralized Processing Unit

CLI command-line interface

DSL Domain Specific Language

PoC Proof Of Concept

Dodatok **B**

Obsah priloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `app` ..... zdrojové kódy aplikácie
  - `infrastructure` .... Terraform repozitár pre vytvorenie infraštruktúry
  - `resources` ..... konfigurácia Envoy proxy a Kubernetes objektov
  - `tests` ..... zdrojové kódy testov s výsledkami
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX