FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Multimodální navigace a její nasazení v škálovatelné architektuře
Student:	Bc. Jan Sokol
Vedoucí:	Ing. Ondřej Guth, Ph.D.
Studijní program:	Informatika
Studijní obor:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2021/22

Pokyny pro vypracování

Prozkoumejte problematiku plánovačů cest pro multimodální navigaci, která využívá různé typy dopravních prostředků (např. veřejná doprava, taxi služby, půjčovny kol).
Prozkoumejte algoritmy pro hledání nejkratší cesty v navigačních modelech závislých na čase (např. veřejná doprava) i v modelech časově nezávislých (např. individuální automobilová doprava). Navrhněte propojení těchto modelů a architekturu systému pro plánování cesty pomocí různých typů dopravních prostředků dle zadaných optimalizačních kritérií.
Proveďte analýzu (sepište požadavky) a návrh multimodálního plánovače. Tento plánovač realizujte jako webovou službu. Předpokladem je, že plánovač využije další poskytovatele dat.

Navrhněte nasazení plánovače jako zabezpečené webové služby s využitím moderních kontejnerových technologií.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 2. listopadu 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Multimodální navigace a její nasazení v škálovatelné architektuře

Bc. Jan Sokol

Katedra počítačových systémů
Vedoucí práce: Ing. Ondřej Guth Ph.D.

7. ledna 2021

Poděkování

Rád bych poděkoval Ing. Ondřej Guth Ph.D. za jeho cenné rady, podporu a připomínky při vedení této práce, které mně velmi pomohly. Dále bych chtěl poděkovat své rodině, bez které bych studium nejspíše nezvládl a mým přátelům i kolegům z práce za veškeré jejich konzultace a informace.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. ledna 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Sokol, Jan. *0.0.0.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá návrhem plánovače cest v geoprostorových grafech, konkrétně s omezením na cesty ve městě. Plánovač nabízí různé způsoby dopravy a v určitých kombinacích také využívá vícera možností prostředků najednou. Plánovací služba je navržena dle principů tzv. mikroslužeb (microservices). K plánovací službě je přístup navržen pomocí REST API. Technologií jako Docker a Kubernetes je využito k nasazení aplikace do distribuovaného a škálovaného systému v druhé části práce. Při nasazení do Kubernetes je brán důraz na zabezpečení aplikace. Část práce je věnována možnostem škálování aplikace v distribuovaném systému Kubernetes. Je brán důraz na vysokou dostupnost aplikace jak při běžném provozu, tak při častém nasazování.

Klíčová slova

navigace, plánování cest, multimodální plánovač, hledání nejkratších cest, otp, osrm, dijskra, a star, contracted hierarchies, distribuovaný systém, vysoká dostupnost, mikroslužba, kubernetes, eks, docker, kontejnerizace, api gateway, oauth2

Abstract

This thesis deals with design of a trip planner in geospatial graphs, with a limitation of routes in cities. Route planner offers various means of transport. Multiple ways of transport are combined into one single trip when certain combinations are used. Route planning service is designed using principles of so called microservices. Access to the planning results is designed using REST API. Technologies Docker and Kubernetes will be used to deploy the route planner into distributed and scallable system in the second part of the thesis. While deploying the service in an distributed system an emphasis is taken on security of the whole architecture. Part of the thesis is dedicated to the application scalability. Importance is put on high availability of the application, both in usual day to day business and also while deploying route planner microservices.

Keywords

navigation, routing, multimodal, shortest path, otp, osrm, dijskra, a star, contracted hierarchies, distributed system, high availability, scalable service, microservice, kubernetes, eks, containerization, docker, api gateway, oauth2

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V současné době je plánování cest velmi důležité a na důležitosti stále více přibývá. A s tím, jak dopravní síť začíná být čím dál složitější a naše pohyblivost po městě začíná být čím dál více důležitější, tak také stoupá potřeba po efektivním a rychlém plánovači. Plánovač je obsažen ve většině současných chytrých mobilních telefonů a také většina leteckých či drážních společností poskytuje nějaký způsob naplánování cesty s jejich dopravními prostředky.

Současné plánovací aplikace mají až na výjimky jedno společné omezení – to je, že plánují jen ve svém vlastním způsobu dopravy. Když využijeme aplikaci hromadné dopravy, plánovač nám ukáže pouze tramvaje, metro a jiné prostředky MHD. Podobně je to s GPS navigací, kde můžeme hledat cestu jen v silniční síti.

Díky tomu, že současnost poskytuje ve velkých městech velké množství způsobů dopravy, vidíme, že jedna z možností je tyto způsoby dopravy kombinovat. Toto je něco, co často není tolik využíváno – už jen proto, že takovýto plán obsahující více možností je složitý na složení, alespoň manuálně. Pro zkombinování více způsobů dopravy je třeba vytvořit pokročilý plánovač, který spočítá více typů dopravy. Takový plánovač se nazývá multomodální.

Tato práce má tedy za jeden z cílů toto. Vybrat počáteční lokaci, konečnou lokaci, společně s časem odjezdu (pro tuto práci bude brán pouze případ aktuálního času) a volitelně způsoby dopravy (ku příkladu sdílené městské prostředky, jako kola, skůtry, koloběžky, taxi, hromadná doprava) a daná aplikace vrátí seznam tras blízkých optimální trase (v této práci je brána nejrychlejší cesta jako optimální).

Dalším stěžejním cílem je nasazení výše zmíněné aplikace do kontejnerizovaného, distribuvaného a škálovatelného prostředí. Spouštění programů v kontejnerech se v poslední době těší veliké oblibě, většinou z důvodu potřeby vysoké dostupnosti aplikace. Takové aplikace většinou přichází s určitými potřebami – měly by být zabezpečené a škálovatelné. V současnosti k takovým požadavkům přichází určitá řešení.

Jedním z řešení k nasazení takové aplikace je využití cloudových řešení, oproti využití nasazení přímo do konvenčních serverů, či virtuálních serverů. I přes to, že cloud je zajímavá alternativa, nenabízí vysokou dostupnost automaticky. Tedy pro to, aby aplikace byla vysoce dostupná, měla by být k takovým požadavkům navržena už od začátku (taková aplikace se nazývá *cloud-native*). Takové prostředí musí být také orchestováno, k čemu existují určité nástroje, kterým se tato práce bude také věnovat.

Práce představí kontejner s výše zmíněnou aplikací, spolu s tím orchestrační software, kterým se aplikace bude spravovat. Dále představí problémy při nasazování vysoce dostupné aplikace do orchestrovaného, kontejnerizovaného prostředí.

Struktura a cíle jsou více do detailu popsány v kapitole níže.

Kapitola 1

Struktura a cíle práce

Diplomová práce je rozdělena do dvou hlavních bodů. Prvním je navrhnout a popsat aplikaci či systém, který bude plánovat cesty v geoprostorových grafech. Konkrétně je omezení na cesty ve městech, a to s použitím různých metod dopravy. Se současným rozvojem sdílených dopravních prostředků ve městech bude služba využívat jak veřejné hromadné dopravy, tak i těchto sdílených vozidel. druhou částí je návrh a popis nasazení plánovací aplikace (mikroslužby) do distribuovaného systému.

1.1 Plánovač cest

První částí je plánovač cest ve městech. Výstupem navrhovaného plánovače budou cesty s použitím jednoho typu prostředku, ale také s jejich kombinacemi (takové kombinace, které dávají smysl – tento výběr bude také v práci diskutován). Ke správnému návrhu a pochopení problematiky hledání cest budou popsány dva základní modely – model závislý a model nezávislý na čase. Na nich budou popsány algoritmy hledající nejkratší cesty. K následnému návrhu bude použit software publikovaný pod otevřenou licencí. U tohoto software práce popíše algoritmy, dle kterých jsou cesty v grafech hledány a na kterých je software stavěn.

1.2 Nasazení mikroslužby v distribuovaném systému

Druhou částí je navržení nasazení aplikace do distribuovaného systému Kubernetes. Bude diskutováno, proč Kubernetes byl vybrán a jeho součásti budou popsány. V nasazení má být brán důraz na několik faktorů. Jedním z ním je bezpečnost aplikace běžící v otevřeném internetu. Tedy je důležité mít komunikaci s aplikací (ve veřejných *subnet*ech) řešenou šifrovaně. Důležitá je též vysoká dostupnost aplikace, budou tedy popsány mechanismy vysoké dostup-

nosti v distribuovaném systému, a to i při opakovaném nasazování. Budou diskutovány způsoby nasazení, mezi ně patřící *Blue/Green deployment*, *Carnary releases*, *Rolling updates* atp. Nasazování bude řešeno automatizovaně spolu s popisem navržené *CI/CD pipeline* (automatizovaných nasazení aplikace).

Kapitola 2

Multimodální plánovač

2.1 Základy teorie grafů

Pro pochopení problému hledání cest je prvně třeba definovat jednotlivé stavební bloky. Práce tedy v kapitole níže popíše relevantní definice z teorie grafů.

2.1.1 Graf

Definice 2.1. Graf

Graf (jednoduchý neorientovaný graf) je uspořádaná dvojice $G = (V, E)$, kde V je množina vrcholů a E je množina hran — množina vybraných dvouprvkových podmnožin množiny vrcholů[1].

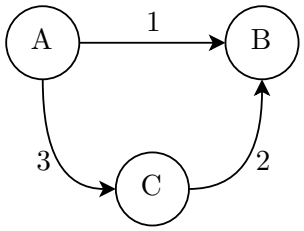
Definice 2.2. Hrana, vrchol

Hranu mezi vrcholy u a v označujeme jako $\{u, v\}$.

Vrcholy spojené hranou nazýváme vrcholy sousední. Značkou $V(G)$ označujeme množinu vrcholů grafu G, množinu hran označujeme jako $E(G)$[1].

Definice 2.3. Orientovaný graf

Orientovaný graf je uspořádaná dvojice $D = (V, E)$, kde $E \subseteq V \times V$[1].



Obrázek 2.1: Příklad orientovaného grafu s ohodnocenými hranami

Všechny dále zmíněné grafy budou orientované, tj. orientace hrany je důležitá.

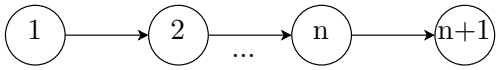
2.1.2 Ohodnocení hran

Tím hlavním rozdílem mezi časově závislým a časově nezávislým grafem je právě způsob ohodnocení hran. U časově nezávislého modelu nám stačí přiřadit hraně konstatní hodnotu. U časově závislého je ohodnocení hran v ruzné denní časy různé.

2.1.3 Cesta

Definice 2.4. Cesta
Podgrafu $H \subseteq G$, který je isomorfní nějaké cestě, říkáme cesta v G.

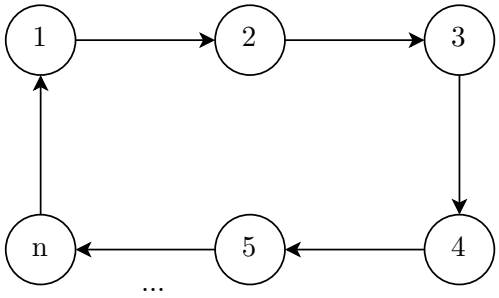
Jinak řečeno, cesta P je sekvence uzlů tak, že pro každý $1 \leq i < k$ platí podmínka $(v_i, v_{i+1}) \in E$.



Obrázek 2.2: Cesta délky n

Definice 2.5. Délka cesty
Délka cesty je součet jejich ohocnocení hran podél cesty.

Definice 2.6. Kružnice
Kružnice délky n má $n \geq 3$ vrcholů spojených do jednoho cyklu n hranami.



Obrázek 2.3: Kružnice délky n

2.2 Modely pro plánování cest

Podkapitoly níže představí časově závislé a časově nezávislé modely, nutné k pochopení plánování cest v těchto modelech.

2.2.1 Časově nezávislý model

Způsoby dopravy, kde se nemusíme řídit žádným jízdím řádem, či předem určenými zastávkami, přiřazujeme k časově nezávislému modelu. Mezi tyto způsoby dopravy řadíme například jízdní kola, taxi, chůzi, služby sdílení aut (*carsharing*) atp. K plánování cest v takovémto modelu můžeme využít konvenčních algoritmů. Určitě jsou rozdíly mezi různými dopravními prostředky, například pro cyklistické kolo je třeba nastavit jinou cestovní rychlost, než na auto.

Silniční síť je modelována jako časově nezávislý model, tj. jako orientovaný graf $G = (V, E)$, kde V je množina uzlů a E je množina hran spojující uzly. Křižovatka je reprezentována jako uzel $s \in V$ a silnice mezi křižovatkami je reprezentována jako hrana $(s, t) \in E$, kde $s \neq t$. Ke každé hraně je přiřazena váhová funkce $l(s, t)$, vracící nenulovou hodnotu a korespondující času, za který úsek s daným prostředkem můžeme ujet. Vzdálenost $dist(s, t)$ je rovna součtu všech vzdáleností v cestě mezi uzly s a t[2].

2.2.2 Časově závislý model

V minulé kapitole jsme diskutovali, jak řešit plánování cesty v silničním (časově nezávislém) modelu. V této kapitole budeme zkoumat model časově závislý, tedy pro veřejnou dopravu. Zde se budeme dotýkat pouze prostředků jako autobus, tramvaj, vlak – tedy těch, které zavisí na nějakém předem určeném jízdním řádu. Z toho vychází název „časově závislý“.

Jízdní řád může být modelován dvěmi základními přístupy, a to jako časově závislý (*time-dependent* model) a časově rozšířený (*time-expanded*) model. V obou případech je model zobrazen jako orientovaný graf $G = (V, E)$. Oba modely mají své výhody a nevýhody, pro zjednodušení zde budeme hovořit pouze o časově závislém modelu.

Časově závislý model byl prvně prezentován v *Brodal and Jacob* (2004)[6]. Model je v učitých místech podobný s časově nezávislým modelem. V tomto modelu uzly $s \in V$ korespondují se zastávkami hromadné dopravy a hrana $(s, t) \in E, s \neq t$ existuje v případě, že dopravní prostředek má spoj ze stanice s do stanice t a nikde mezitím nezastavuje. Hlavním rozdílem oproti časově nezáviskému modelu je to, že hrana existuje pouze v určité časy a tedy čas cesty závisí na čase, kdy jsme dorazili do počátečního uzlu. Tato informace je zakódována jako funkce doby cesty mezi uzly s a t.

2.2.3 HexSpace indexovací systém

„Gridové“ systémy (*Grid systems*) jsou nástoje důležité pro analýzu velkých *datasetů* prostorových dat a pro rozdělení oblastí planety do identifikovatelných buněk dle mřížky. Akademický termín pro takový nástroj je diskrétní globální síťový systém (*discrete global grid system*) [78]. Je to tedy diskrétní

systém, který rozděluje svět na diskrétní buňky – ke každé pozici na světě je přidružen identifikátor buňky.

„Grid“ systém **H3**[77] je známým nástrojem v této oblasti, byl navržen společností Uber, která nástroj využívá pro plánování taxi jízd. Společnost Uber tento nástroj veřejně vydala jako open source projekt a je široce používán. V odstavcích níže popíši tento nástroj a jeho využití v moji konkrétní implementaci.

H3

Jednoduše řečeno, mřížka v **H3** je šestihranný objekt (hexagon) a který lze znovu rekurzivně rozdělit na menší hexagony v mřížce. Hexagonní systém je vhodnější pro modelování a prostorové transformace, protože sousední objekty jsou v tomto systému stejně vzdáleni (na rozdíl od tvarů jako jsou trojúhelníky nebo čtverce).

H3 také obsahuje řadu analytických nástrojů, jako například funkce pro převod souřadnic a geoprostorové indexování.

Konkrétní implementace v plánovací aplikaci

V mojí konkrétní implementaci budu takovýto indexovací systém reprezentovat 2 hlavními objekty. Těmi jsou

1. Hexagon objekt, který obsahuje informace, jako své `hexagon_id`, geolokaci bodu uprostřed hexagonu, které vozidla obsahuje a rozlišení (velikost) hexagonu.
2. HexSpace objekt, který je wrapper nad **H3** knihovnou a obsahuje všechny hexagony.

2.3 Hledání nejkratší cesty

V této kapitole představíme algoritmy k plánování cest.

2.3.1 Problém nejkratší cesty

Prvně je důležité formálně popsat problém nejkratší cesty.

Definice 2.7. Nechť $G = (V, E)$ je vážený, orientovaný nebo neorientovaný graf.

Váha cesty $P =< v_0, v_1, v_2, \cdots, v_k >$ je $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Nejkratší cesta $\delta(u, v)$ z u do v má váhu

$$\delta(u, v) = \begin{cases} \min\{w(P) : P \text{ je cesta z } u \text{ do } v\} & \text{pokud cesta existuje} \\ \infty & \text{jinak} \end{cases}$$

[84]

Mezi vlastnosti nejkratších cest patří

- Části cest z nejkratších cest jsou také nejkratší cesty: Pokud $P =< u = v_0, v_1, v_2, \cdots, v_k = v >$ je nejkratší cesta z u do v, pak pro $i < k$ $P' =< u = v_0, v_1, v_2, \cdots, v_i >$ je nejkratší cesta z u do v_i
- Neexistuje nejkratší cesta, pokud graf má cyklus s negativní vahou. [1]

Existují rozdílné varianty problému nejkratších cest. Níže letmě zmíním ty varianty, které jsou pro práci relevantní.

- *Nejkratší cesta mezi dvojicí uzlů*: Nalezení nejkratší cesty z u do v (1:1)
- *Nejkratší cesta z daného uzlu grafu do všech ostatních uzlů grafu – Single source shortest path (SSSP)*: Nalezení nejkratší cesty z s od všech uzlů $v \in V$ (1:N)
- *Nejkratší cesta mezi všemi dvojicemi uzlů grafu – All pair shortest path (APSP)*: Nalezení nejkratší cesty z u do v pro všechny $u, v \in V$ (N:N)

Každá z těchto variant problému nejkratší cesty lze vyřešit pomocí Dijskrova algoritmu (někdy opakovaným spuštěním, např. u APSP). Dijskrův algoritmus bude popsán v kapitole níže.

2.3.2 Algoritmy k hledání nejkratší cesty

V následující podkapitole popíši algoritmy k hledání nejkratší cesty.

2.3.2.1 Dijskrův algoritmus

Práce nazvaná *A Note on Two Problems in Connexion with Graphs* byla publikována v žurnálu *Numerische Mathematik* v roce 1959. Bylo to právě v této práci, kde Edsger W. Dijkstra navrhl Dijkstrův algoritmus pro řešení různých variant problému nejkratších cest.

Princip Dijskrova algoritmu

V následujících krocích popíši, jak Dijskrův algoritmus funguje.

Krok 1 Prvně přiřaď $Node(A) = 0$ jako váhu počátečního uzlu a $w(x) = \infty$ všem ostatním uzlům, kde x jsou ostatní uzly.

[1] V práci pouze bereme v potaz grafy, které nemají hrany s negativními vahami. Je možné najít nejkratší cestu v grafu, který má hrany s negativními vahami (Ale ne negativními cykly), ale ne všechny algoritmy lze pro takový graf použít a obecně tyto algoritmy jsou pomalejší.

Krok 2 Hledej uzel x, který má nejmenší dočasnou váhu $w(x)$. Zastav algoritmus, pokud $w(x) = \infty$ nebo už nejsou žádné dočasné uzly. Uzel x je teď uznačen jako trvalý a aktuální uzel, což znamená, žr x a $w(x)$ se již nezmění.

Krok 3 Pro každý přilehlý uzel k x, označený y, pokud je y stále dočasný, aplikuj: pokud $w(x)+Wxy < w(y)$, pak $w(y)$ je nastaven na $w(x)+Wxy$, kde W je váha přilehlého uzlu. Přiřaď y, aby měl nadřazený uzel x

Krok 4 Opakuj Krok 2 dokud není nalezena nejkratší cesta.

Hlavním využitím Dijskrova algoritmu je pro hledání cest v silničních sítích (tak, jak se jim věnuje tato práce). Dalším využitím je *Open Shortest Path First* (*OSPF*) algoritums, díky kterému se směruje v síti Internet.

Implementace Dijkstrova algoritmu

Níže přidávám konkrétní implementaci Dijskrova algoritmu, využitého v plánovači z této práce, viz 2.4.3.1.

```
def shortest_path(graph, start_vertex, goal_node):
    distances, paths = dijkstra(graph, start_vertex)
    route = [goal_node]
    while goal_node != start_vertex:
        route.append(paths[goal_node])
        goal_node = paths[goal_node]
    route.reverse()
    return route
```

2.3.2.2 A star

Algoritmu A star je využito v open source backendu *Open Trip Planner*, proto stojí za to si zde představit jeho myšlenku.

A star je algoritmus využívaný k nalezení optimálních cest v kladně ohodnocených grafech. Prvně byl představen Peterem Hartem, Nilsem Nilssonem a Bertramem Raphaelem v [130]. A star využívá hladový přístup pro nalezení optimální cesty. Optimální cestou je nejrychlejší (případně nejkratší, nejlevnější) v závislosti na hodnotách vah u hran v grafu.

Využívá funkci $f(x)$, která ohodnocuje uzly pro určení pořadí, v jakém mají být uzly procházeny. Funkce se skládá ze dvou dalších funkcí, $f(x) = g(x)+h(x)$, kde $g(x)$ je funkce vzdálenosti mezi počátečním uzlem a aktuálním uzlem, $h(x)$ je heuristická funkce. Funkce $h(x)$ je odhad délky cesty z x do cílového stavu, kde

- $h(g) = 0$ pro cílový stav,

- $h(x)0$ pro každý stav.

Více do detailu je algoritmus popsán v [130] a [131].

2.3.2.3 Contraction Hierarchies

Algoritmu Contraction Hierarchies je využito v open source backendu *Open Source Routing Machine*, proto stojí za to si zde představit jeho myšlenku.

Contraction hierarchies (CH) [133] je dvoufázová technika, míněná ke zrychlení výpočtů k nalezení nejkratších cest.

V první **„preprocessing“ fázi** heuristicky seřadíme uzly podle důležitosti a provedeme kontrakci uzlů od nejméně důležitého k nejvíce důležitý. Intuitivně, uzly, které jsou obsaženy ve velké části nejkratších cest, považujeme za důležité (například dálnice) a ty, které nejsou tolik obsaženy v nejkratších cestách, považujeme za méně důležité (například okresky). [132]

Kontrakce uzlu v vypadá následovně

- dočasně odstraníme uzel v z grafu,
- přidáme hrany mezi sousedy uzlu v, abychom zachovali mezi nimi vzdálenosti (v grafu bez v).

Nová cesta je nutná pouze v případě, že je to nejkratší cesta mezi danými body (což může být ověřeno například Dijskrou).

Hledací fáze provádí dvousměrnou Dijskru (stejný algoritmus jako v 2.3.2.1, jen je prováděn z počátečního uzlu a konečného uzlu zároveň [135], algoritmus končí, když se oba běhy v některém uzlu potkají) na upraveném grafu z první fáze.

Více do detailu je algoritmus popsán v [132], [133] a [134] (obsahuje vizualizaci).

2.3.3 Multimodální plánování cest

Již bylo prozkoumáno mnoho způsobů, jak přistoupit k plánování tras tam, kde kombinujeme více modelů najednou.

Já se budu více do hloubky věnovat přávě metodě *Transit Node Routing*, dle které je postaven právě algoritmus plánovače níže.

2.3.3.1 Transit Node plánování

Transit Node Routing, neboli „plánování tras s přestupními stanicemi“, je algoritmus k nalezení nejkratší cesty v grafu, který pro zrychlení běhu předvypočítává časté přestupní stanice a také předvypočítává trasy mezi nimi. Algoritmus byl představen Hannou Bast a Peterem Sandersem v [5].

Algoritmus se vyznačuje statickým přístupem, kde musíme předvypočítat vzdálenosti mezi důležitými uzly v grafu. Dynamický přístup zatím nebyl publikován.

Doprava na delší vzdálenost většinou obsahuje cestování po podmnožině dopravní sítě, jako například dálníce místo okresek, či metro místo tramvaje. Na tuto „podsíť“ můžeme vstoupit jen na několika řídce distribuovaných uzlech. Když vedle sebe porovnáme několik cest na dlouhou vzdálenost, většinou obsahují stejný malý počet nástupních a výstupních přestupních stanic. Tato myšlenka platí pouze pro cesty na delší vzdálenost. Když cestujeme na kratší vzdálenost, přestupní stanice nebudou využity a použijeme lokálnější cesty.

Díky tomu, že počet přestupních stanic je malý (v porovnání s celkovým počtem všech stanic), můžeme předvypočítat trasy mezi nimi a ty uložit. Když potom počítáme nejkratší cestu, pouze cesty z počátečního bodu do přestupní stanice a z jiné přestupní stanice do konečného bodu musí být vypočítány.

Postup algoritmu

Spíše než algoritmem je *Transit Node Routing* tzv. framework. Kroky jsou dle [5] následující

- Začneme s výběrem přestupních stanic $T \subseteq V$, jako podmnožinou všech uzlů V.
- Pro každou hranu $v \in V$ vybereme ze všech přestupních stanic „dopředné“ (nástupní) přestupní stanice $\overrightarrow{A}(v) \subseteq T$ a „konečné“ (výstupní) přestupní stanice $\overleftarrow{A}(v) \subseteq T$.
- Spočítáme a uložíme párové vzdálenosti mezi mezi přestupními stanicemi D_T a vzdálenosti mezi uzly v a jejich přiřazenými „nástupními“ přestupními stanicemi d_A jsou spočítány a uloženy.
- Vzdálenost mezi dvěma uzly je spočítána jako $d(s,t) = \min_{u \in \overrightarrow{A}(s), v \in \overleftarrow{A}(t)} d_A(s,u) + D_T(u,v) + d_A(v,t)$.

Jak je plánovač založen na tomto algoritmu je popsáno níže.

2.4 Návrh plánovače

Následující část práce popíše aktuální řešení multimodálního plánování, co již jsou na trhu a jaké funkce splňují. Dále prozkoume open source plánovače Open Trip Planner a Open Source Routing Machine, které budou využity k plánovači v této práci. Následovně bude popsán návrh multimodálního plánovače a jeho kontejnerizace.

2.4.1 Aktuální řešení na trhu

V této kapitole jsou popsány již existující řešení na trhu.

2.4.1.1 Multimodální navigace v Google Mapách

V roce 2019 přidal Google do své služby Mapy možnost vyhledávání tras s více prostředky najednou[51]. Každopádně tato funkce je značně omezená a dostupná pouze pro nějaká města.

Omezením je, že jedinými kombinacemi jsou:

- Plánování dle „první míle“ – první část, tedy od počátečního bodu k nějaké vzdálené přestupní stanici je cesta realizována pomocí taxislužby (podporované jsou aktuálně pouze Uber, Lyft a Bolt) a zbytek trasy je realizován pomocí hromadné dopravy.

- Plánování dle „poslední míle“ – vetšina trasy je realizována pomocí hromadné dopravy a zbytek (tedy posledních několik km) je realizován pomocí taxislužby.

Další nevýhodou je vysoká cena za využití *Google Maps API*. 1000 požadavků na vyhledání cest vychází na 10 USD[52].

2.4.1.2 Multimodální plánovač AnyRoute

AnyRoute je projekt od společnosti Umotional s.r.o., která se nazývá jako „spin-off z ČVUT“. Sami projekt AnyRoute popisují jako „Plánovač je výsledkem několikaletého firemního vývoje navazující na předcházející akademický výzkum. Využívá technik umělé inteligence k tomu, aby navrhnul vhodné door-to-door trasy veškerými způsoby osobní dopravy vyskytujícími se v moderních městech, včetně jejich kombinací (např. Park+Ride, Bike+Ride, MHD+sdílené kolo nebo MHD+taxi). Naplánované trasy mohou být kromě času optimalizovány na řadu dalších kritérií (např. cena, komfort, emise nebo zdravotní dopady).“ [54] Prezentaci projektu lze nalézt zde [55].

Aplikace je veřejně dostupná jako demo pro Prahu, veřejně dostupné API jsem nenalezl.

2.4.1.3 Multimodální plánovač Trafi

Významným poskytovatelem *Mobility as a Service* (*MaaS*, mobilita jako služba) je litevská společnost Trafi [56]. Žádné veřejné demo neposkytují, ale nabízí své služby společnostem/aplikacím jako např. Jelbi [57](multimodální plánovač pro Berlín) nebo yumuv [58](multimodální plánovač pro Švýcarsko). Služba obsahuje multimodální plánovač, platební systém, informace o hromadné dopravě, analytiku a management uživatelů.

2.4.1.4 Multimodální plánovač OpenTripPlanner

Open source řešením, které nabízí multimodální plánování tras je Open Trip Planner[50]. Plánovač sice nenabízí všechny nutné funkce a konfigurace, je ale možné na něm stavět. Více do hloubky se mu bude věnovat následující kapitola.

2.4.2 Výběr již vytvořených plánovačů s otevřeným kódem

Jak již bylo zmíněno, ani jedno z aktuálně nabízených řešení na trhu nenabízí vše. Aktuálním problémem je nestandardizovaný formát dat pro sdílené prostředky (od služeb sdílených mobilit). Většinou je nutné data získat od poskytovatelů osobně pod nějakým NDA. Stavět celý plánovač by mohlo být časově velmi náročné (a ve výsledku by jeho rychlost nemusela být ohromující), proto jsem se rozhodl využít již vytvořených řešení s otevřeným kódem. Každý z nich má nějaké nevýhody – Open Source Routing Machine nepodporuje plánování s hromadnou dopravou, ale je rychlý, na druhou stranu Open Trip Planner plánování s hromadnou dopravou podporuje, ale má horší podporu v silničních grafech a oproti OSRM je mnohem pomalejší. Plánovač v této práci proto tyto open source řešení kombinuje.

2.4.2.1 Open Source Routing Machine

Open Source Routing Machine je open source plánovač, který je designovaný pro využití s daty z Open Street Map.

Narozdíl od ostatních plánovačů (i Open Trip Planner popsaný dále) využívá pro hledání nejkratších cest algoritmus contraction hierarchies (popsaný v kapitole 2.3.2.3) namísto A star algoritmu (popsaného v kapitole 2.3.2.2). Díky tomu na plánovací dotazy odpovídá velmi rychle (většinou pod 1ms)[65]. Mezi hlavní výhody patří

- velmi rychlé plánování,
- flexibilní profily dopravních prostředků,
- poměrně jednoduché vložení dat o dopravním provozu, výškových úrovní, atp.

OSRM je napsaný v C++ a vydaný pod BSD licencí[66].

Mapový podklad

Open Source Routing Machine (*OSRM*) vyžaduje mapový podklad z Open Street Map ve formátu `PBF` (tzv. *Protocolbuffer Binary Format*)[59]. Open Street Map využívá několik různých formátů souborů obsahující mapové podklady – ať už zmíněný `PBF`, nebo `PBF`, či `CSV`. Manuál k OSM formátům je dostupný z [60].

`PBF` soubory obsahující maový podklad je možné stáhnout z [61], kde si můžeme vybrat ať už celou planetu, nebo jen část, kterou potřebujeme (v našem případě tedy pouze podklad pro Prahu). Aktualizované podklady pro Českou republiku poskytuje například VUT Brno na [62]. Je tedy třeba mapu „oříznout“, což je popsáno v [63].

Způsob využití OSRM

Z OSRM jsou využity dvě hlavní funkce, API endpoint pro plánování cest z bodu A do bodu B a endpoint pro hledání časových matic.

- Základní službou, která je v v OSRM využita, je služba pro hledání cest [83]. Služba nalezne nejrychlejší cestu mezi body zadanými vstupními parametry. Je možné zvolit různé nepovinné parametry, jako přidání nalezených alternativních cest, případně verbosita informací o plánované cestě.
- OSRM již nabízí nabízí službu pro hledání časových matic [82]. Vstupem jsou seznamy počátečních a konečných bodů (ve formátu GPS souřadnic). Výstupem služby je matice nejkratších časů, během kterých je možné uskutečnít cestu mezi všemi páry počátečních a konečných bodů.

2.4.2.2 Open Trip Planner

Open Trip Planner (OTP)[50] je multimodální plánovač cest, který se dle oficiální dokumentace zaměřuje na cesty s využitím hromadné dopravy v kombinaci s jízdou na kole, chůzí, či mobility službami jako sdílení kol (tato funkce je ale poměrně omezená a není ideálně dokumentovaná). Serverová část OTP je schopna běžet na jakékoli platformě, na které běží Java virtual machine (tedy Linux, Mac nebo Windows). OTP nabízí REST a GraphQL API, ke kterým projekt nabízí i různé frontendy. Staví svoji reprezentaci dopravní sítě na otevřených datech v otevřených formátech, jako jsou GTFS a podklady map OpenStreetMap. Nabízí upozornění a změny tras v reálném čase dle výluk na dopravní síti.

V roce 2020 byla vydána verze 2.0, která je ve fázi RC (*Release Candidate*). Kvůli dlouhodobějšímu vývoji plánovače z této práce je využito starší verze OTP, tj. verze 1.3. *Open Trip Planner* je vydán pod licencí LGPL [70], tedy dílo pod LGPL lze linkovat (v případě knihovny užívat) programem, která nemá licenci (L)GPL, a který může být jak svobodný software, tak software proprietární[71].

GTFS Data

Využívaným formátem obsahující jízdní řády městské hromadné dopravy je *General Transit Feed Specification* (*GTFS*)[46]. GTFS má v sobě zakódován

relevantní informace jízdních řádů, jako například geografické lokace míst, přes která linka projíždí, časy příjezdů a odjezdů ze stanic, či informace o stanicích. Dopravci mohou publikovat své jízdní řády jako GTFS soubory, které mohou vývojáři a aplikace dále využívat. Například město Praha nabízí tato data veřejně dostupně na svých stránkách `opendata.praha.eu` [69].

Pro poskytnutí aktualizovaných informací, jako zpožděné odjezdy či příjezdy mohou být GTFS data dále rozšířena pomocí *GTFS Realtime extension*. S využitím *GTFS Realtime extension* mohou být oznámeny události jako výluky či nehody na tratích. V této práci se budeme zabývat pouze GTFS daty bez rozšíření.

Díky integraci Open Trip Planneru s GTFS daty je právě této služby využíváno k plánování tras pomocí městské hromadné dopravy.

Mapový podklad

Mapový podklad je stejný jako u OSRM, tedy `.osm.pbf` formát. Jak mapový podklad sehnat je popsáno v kapitole 2.4.2.1 a jak „oříznout“ je popsáno v [63].

Způsob využití OTP

Z *Open Trip Planner*u je využito již existujícího plánovacího endpointu, který je popsaný v [64]. API specifikace je dostupná v [47].

V základě *Open Trip Planner* nenabízí API endpoint pro vytvoření časové matice (Oproti němu OSRM časovou matici již nabízí; časová matice je využita například v **Kroku 2a.5** v kapitole 2.4.3.1). Proto musel být tento endpoint do aplikace doprogramován. Endpoint je koncipován stejně, jako v OSRM službě, popsaný v 2.4.2.1.

Vytvořený endpoint je dostupný na URI `routers/<routerId>/vector` a je implementován následovně (zjednodušeno pro čitelnost)

```
Router router = otpServer.getRouter(routerId);

// Parse destinations sent as params
ArrayList<GenericLocation> destinationPlaces =
    decodeDestinations(encodedDestinations);

// Build generic RoutingRequest
RoutingRequest request = super.buildRequest();

// Generate SPT
request.batch = true;
request.setRoutingContext(router.graph);
request.worstTime = request.dateTime + 100000;
ShortestPathTree spt = new AStar().getShortestPathTree(request);
```

2.4.3 Možnosti plánovače

V následující kapitole jsou popsány kombinace dopravních prostředků, které budou v plánovači navrhovaném touto prací podporovány.

2.4.3.1 Singlemodální s využitím sdílených prostředků

Pro plánování cest s hromadnou dopravou je využito aplikace *Open Source Routing Machine* zmíněného v 2.4.2.1.

API endpoint

Pro přístup k plánovači je využito REST API. Restful endpoint vypadá takto

```
GET /v1/routing/<city>/<vehicle_type>?end_lat=<float>&end_lng=<float>
&start_lat=<float>&start_lng=<float>
```

kde

- `<city>` je město, pro který má být plán vytvořen,
- `<vehicle_type>` je dopravní prostředek, pro který má být plán vytvořen. Může být jeden z `{kickscooter,bike,carshare,scooter}`,
- `<services>` je seznam společností sdílených mobilit, s jejichž prostředky májí být cesty plánovány,
- `<start_lat>` je zeměpisná šířka počátečního bodu,
- `<start_lng>` je zeměpisná délka počátečního bodu,
- `<end_lat>` je zeměpisná šířka konečného bodu,
- `<end_lng>` je zeměpisná délka konečného bodu.

Algoritmus

Algoritmus k nalezení cesty se sdílenými prostředky je následnovný

Krok 1 Z požadavku od klienta převezmeme potřebné parametry. Mezi teto parametry patří

- zeměpisné délky a šířky počátečního a konečného bodu,
- výběr města (toto je z důvodu, že mapový podklad je omezen vždy na jedno město hlavně kvůli šetření paměti),
- výběr společností sdílené mobility. Pokud parametr není zadán, jsou automaticky vybrány všechny možné služby pro daný typ mobility a pro dané město.

Krok 2a konkurentně vedle sebe vytvoříme n různých procesů (mohou být i vlákna, ale výpočty plánů tras pro odlišné poskytovatele mobilit mezi sebou žádná data v průběhu výpočtu nesdílí), kde n je počet společností v parametru `<services>`. Dockované a nedockované prostředky mají postup lehce odlišný. Jeden způsob je pro plánování cest s dockovanými prostředky (se stanicemi, ze kterých lze sdílený prostředek vyzvednout a do kterého lze vrátit a pro prostředky, které je možné vrátit pouze ve vyznačených zónách), druhý pro nedockované prostředky. V každém z procesů budeme hledat trasy, obsahující daný sdílený prostředek. Pro dockované vozidla sdílených mobilit

Krok 2a.1 vybereme všechna sdílená vozidla v okruhu od počátečního bodu, kde poloměr okruhu je dán konfigurací aplikace,

Krok 2a.2 Vytvoříme HexSpace graf[2],

Krok 2a.3 do HexSpace grafu vložíme 2 uzly – počáteční a konečný bod cesty a pro ně vytvoříme Hexagon objekt,

Krok 2a.4 z docků a krajních míst servisních zón vytvoříme hrany v HexSpace [3] (pokud již neexistuje Hexagon, tak ho vytvoříme),

Krok 2a.5 asynchronně spočítáme váhy jednotlivých etap tras (tyto váhy jsou v pozdějším **Kroku 2.a7** přiřazeny hranám v HexSpace grafu). Jako váhu etapy (ohodnocení hrany) považujeme čas, který je třeba na etapu. Pomocí OSRM backendu spočítáme

- Time matrix[4] z počátečního bodu do bodů s vozidly pomocí chůze (*one to many*),
- Time matrix z bodů s vozidly do bodů s docky (*many to many*),
- Time matrix z bodů s docky do konečného bodu (*many to one*).

Krok 2a.6 V HexSpace grafu přidáme uzly pro každé vybrané sdílené vozidlo, plus hranu z hranu z počátečního uzlu do uzlů s vozidly (s přidělenou váhou vytvořenou pomocí OSRM v bodu výše; a módem chůze),

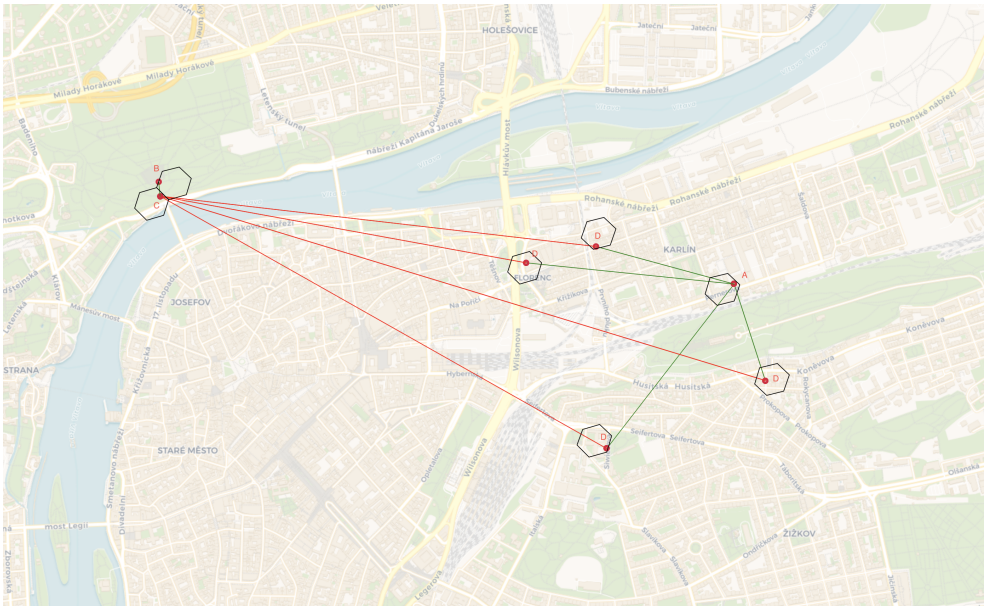
Krok 2a.7 V HexSpace grafu přidáme uzly pro vybrané stanice vozidel (případně mezní body zón), plus hrany

- z uzlu vozidla do uzlu stanice (resp. mezního bodu zóny), s přiděleným ohodnocením hrany (jak je zmíněno v **Kroku 2a.5**, ohodnocení je pro jednoduchost čas, za který se urazí etapa) a módem daného sdíleného prostředku,
- z uzlu docku (resp. mezního bodu zóny) do konečného uzlu, s přidělenou váhou a módem chůze,

[2] HexSpace je definován v předchozí kapitole 2.2.3.

[3] Výběr relevantních docků je popsán v 2.4.4.2 a výběr bodů ze servisních zón je popsán v 2.4.4.1.

[4] Endpoint OSRM s časovou maticí je popsán v 2.4.2.1



Obrázek 2.4: Singlemodální graf pro plán cesty z bodu A do bodu B, pomocí sdíleného prostředku. Uzly D označují místa, kde se nachází sdílené prostředky a jsou připraveny k použití. Uzel C označuje nejbližší místo k bodu B, kde lze sdílený prostředek vrátit. Zelené hrany jsou realizovány pomocí chůze, červené pomocí sdíleného prostředku. K vizualizaci grafu je využita knihovna Folium. [98]

Krok 2a.8 pomocí **Dijskrova algoritmu** [5] nalezneme v HexSpace grafu **nejkratší cestu**,

Krok 2a.9 vrátíme nejlepší trasu nalezenou pomocí Dijskrova algoritmu v kroku **Krok 2a.8**.

Krok 2b Pro nedockované vozidla sdílených mobilit

Krok 2b.1 vybereme všechna vozidla v okruhu od počátečního bodu, kde poloměr je dán konfigurací aplikace,

Krok 2b.2 pro všechna vybraná vozidla se pokusíme nalézt 2 cesty:

- cestu chůzí z počátečního bodu ke sdíleným vozidlům (tzv. one to many popsaný v kapitolách výše),
- cestu na vozidle z aktuálního místa vozidla do cílového bodu cesty.

Tyto dvě etapy celkové cesty spojíme do sebe a vypočítáme celkovou délku, dobu a cenu trasy. Cena je vypočítána dle základního tarifu dané služby mobilit.

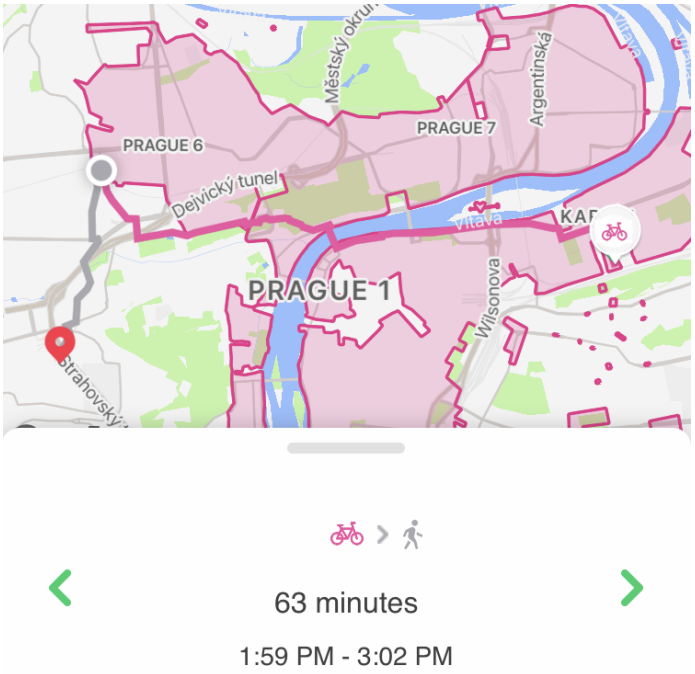
[5] Dijskrův algoritmus je obecně definován kapitole 2.3.2.1, spolu s konkrétní implementací.

Krok 2b.3 Plány tras jsou seřazeny (dle doby cesty) a pro každou společnost je vybrána nejrychlejší trasa, která je poté vrácena.

Krok 3 K trasám spočítáme detaily (jako detail se počítá cena, polyline, čas a adresa počátku a konce, atp) všech etap trasy tak, že

- pro etapy, kde způsob dopravy je sdílený dopravní prostředek nebo chůze, využijeme OSRM backendu,
- pro etapy, kde způsob dopravy je veřejná doprava, využijeme OTP backendu.

Krok 4 Vrátíme nejrychlejší cestu pro každou sdílenou mobilitu.



Obrázek 2.5: Konečná cesta ze singlemodálního plánovače s využitím sdílených prostředků. Červený bod je konečný.

2.4.3.2 Singlemodální s využitím hromadné dopravy

Pro plánování cest s hromadnou dopravou je využito aplikace Open Trip Planner zmíněného výše. Plánování cest tímto způsobem je poměrně přímočaré. Je třeba mít dostupný OTP backend, ideálně nasazený ve stejném clusteru jako plánovací aplikace.

API endpoint

Pro přístup k plánovači je využito stejně jako u ostatních způsobů REST API. Restful endpoint vypadá takto

```
GET /v1/routing/<city>/public_transport?end_lat=<float>&end_lng=<float>
&start_lat=<float>&start_lng=<float>
```

kde

- `<city>` je město, pro který má být plán vytvořen,
- `<start_lat>` je zeměpisná šířka počátečního bodu,
- `<start_lng>` je zeměpisná délka počátečního bodu,
- `<end_lat>` je zeměpisná šířka konečného bodu,
- `<end_lng>` je zeměpisná délka konečného bodu.

Algoritmus

Plán cesty s hromadnou dopravou se zkládá z těchto kroků:

Krok 1 Z požadavku od klienta převezmeme potřebné parametry. Mezi teto parametry patří

- zeměpisné délky a šířky počátečního a konečného bodu,
- výběr města (toto je z důvodu, že mapový podklad je omezen vždy na jedno město hlavně kvůli šetření paměti),
- výběr módů dopravy (mezi ně patří tramvaj, autobus, chůze, lanovka, atp.),
- a další dodatečné parametry k vyhledání nejvhodnější cesty. Mezi tyto dodatečné parametry patří maximální délka chůze, maximální počet přestupů a minimální čas přestupu.

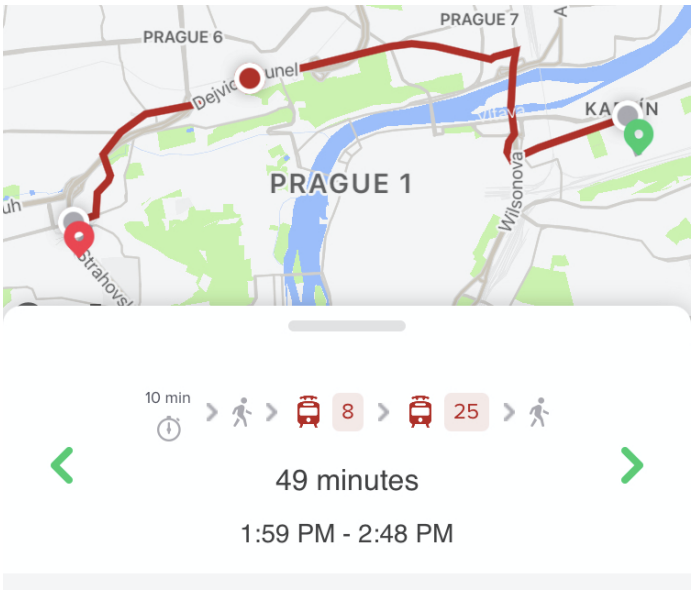
Krok 2 Z požadavku od klienta vytvoříme požadavek pro Open Trip Planner. K požadavku přidáme dodatečné parametry z předchozího kroku, které OTP podporuje. Požadavek poté odešleme na Open Trip Planner backend.

Krok 3 Z aplikace Open Trip Planner získáme seznam cest, pokud nějaké existují.

Krok 4 Jednotlivé cesty dekódujeme a ke každé si ukládáme celkový čas cesty, a to jak k celé trase, tak i k jejím etapám (částím).

Krok 5 Podíváme se, zda je danou část (etapu) cesty možné cestovat více různými prostředky. Pokud ano, přidáme alternativní prostředek jako proměnnou k etapě cesty.

Krok 6 Vybereme n nejlepších cest dle kritéria (nejkratšího času).



Obrázek 2.6: Konečná cesta ze singlemodálního plánovače s využitím hromadné dopravy

2.4.3.3 Multimodální s využitím hromadné dopravy a sdílených kol

Pro multimodální cesty jsou aktuálně vybrány dvě kombinace – jednou je pro *problém první míle*, kde v počáteční, kratší etapě trasy využíváme prostředku sdílené mobility a na zbytek vuyžíváme hromadné dopravy. Druhou kombinací je *problém poslední míle*, kde na většinu trasy využijeme hromadnou dopravu a na konečný zbytek sdílený prostředek.

Algoritmus níže je inspirován algoritmem *Transit Node Routing*, popsaným v kapitole 2.3.3.1. V současné době je algoritmus zjednodušen, takže trasy mezi přestupními stanicemi nejsou ukládány v mezipaměti, ale jsou pro každou novou trasu počítány znovu. Výběr přestupních stanic ale platí stejně.

API endpoint

Pro přístup k plánovači je využito stejně jako u singlemodálního plánovače – pomocí REST API. Restful endpoint vypadá takto

```
GET /v1/routing/<city>/multimodal?end_lat=<float>&end_lng=<float>
&start_lat=<float>&start_lng=<float>&services=service_list
```

kde

- `<city>` je město, pro který má být plán vytvořen,
- `<service_list>` je seznam poskytovatelů sdíledných prostředků, které budou do HexSpace grafu přidány,
- `<start_lat>` je zeměpisná šířka počátečního bodu,
- `<start_lng>` je zeměpisná délka počátečního bodu,
- `<end_lat>` je zeměpisná šířka konečného bodu,
- `<end_lng>` je zeměpisná délka konečného bodu.

Algoritmus

Algoritmus je pro čitelnost rozdělen do dvou částí, a to dle problému první míle a dle problému poslední míle.

Plánování dle problému první míle

Plán cesty s hromadnou dopravou a sdílenými prostředky (kolo, koloběžka, skútr, atp) se skládá z těchto kroků:

Krok 1 Z požadavku od klienta převezmeme potřebné parametry. Mezi teto parametry patří

- zeměpisné délky a šířky počátečního a konečného bodu,
- výběr města (toto je z důvodu, že mapový podklad je omezen vždy na jedno město hlavně kvůli šetření paměti),
- výběr společností sdílené mobility. Pokud parametr není zadán, jsou automaticky vybrány všechny možné služby pro daný typ mobility a pro dané město.

Krok 2 vytvoříme HexSpace graf [6],

Krok 3 do HexSpace grafu vložíme 2 uzly – počáteční a konečný bod cesty a pro ně vytvoříme Hexagon objekt,

Krok 4 vybereme vozidla v okolí počátečního bodu[7] a přidáme je jako uzly do HexSpace grafu,

[6] HexSpace je definován v předchozí kapitole 2.2.3.

[7] Vozidla v okolí bodu jsou vybírána pomocí Harvesinova vzorce, který je popsán v kapitole 2.4.4.2.

Krok 5 vybereme přestupní stanice v okolí počátečního bodu[8] a přidáme je jako uzly do HexSpace grafu,

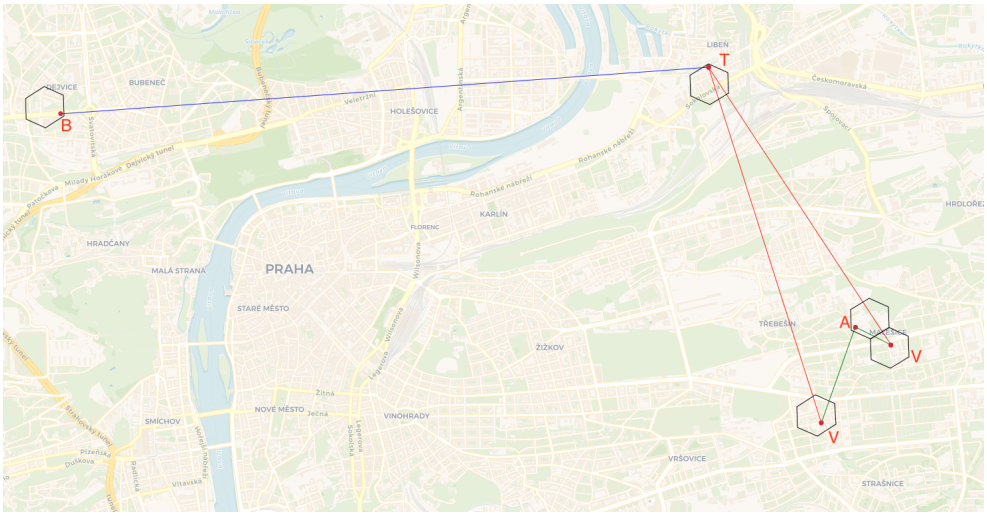
Krok 6 asynchronně spočítáme váhy jednotlivých etap tras. (tyto váhy jsou v pozdějším **Kroku 7** přiřazeny jako ohodnocení hranám v HexSpace grafu). Jako váhu etapy (ohodnocení hrany) považujeme čas, který je třeba na etapu. Pomocí OSRM a OTP backendů spočítáme

- Time matrix[9] z počátečního bodu do bodů s vozidly pomocí chůze (*one to many*),
- Time matrix z bodů s vozidly do bodů s přestupními stanicemi (*many to many*),
- Time matrix[10] z přestupních stanic do konečného bodu (*many to one*).

Krok 7 Do HexSpace grafu přidáme

- hrany z počátečního uzlu do uzlů se sdílenými vozidly vybranými v předchozím kroku (s přidělenou váhou vytvořenou pomocí OSRM v bodu výše; a módem chůze),
- hrany z uzlů se sdílenými vozidly do uzlů s přestupními stanicemi s přidělenou váhou a módem daného sdíleného prostředku,
- hrany z uzlů se přestupními stanicemi do konečného uzlu, s přidělenou váhou a módem hromadné dopravy,

[8] Výběr přestupních stanic je popsán v kapitole 2.4.4.3.
[9] Endpoint OSRM s časovou maticí je popsán v 2.4.2.1.
[10] Endpoint OTP s časovou maticí je popsán v 2.4.2.2.



Obrázek 2.7: Multimodální graf pro plán cesty z bodu A do bodu B, metodou první míle. Černé hexagony T obsahují přestupní stanice (Označeny jako červené body). Body V obsahují lokace se sdílenými prostředky. Modré hrany označují zjednodučenou cestu pomocí hromadné dopravy, červené hrany označují cesty pomocí sdíleného prostředku. K vizualizaci grafu je využita knihovna Folium. [98]

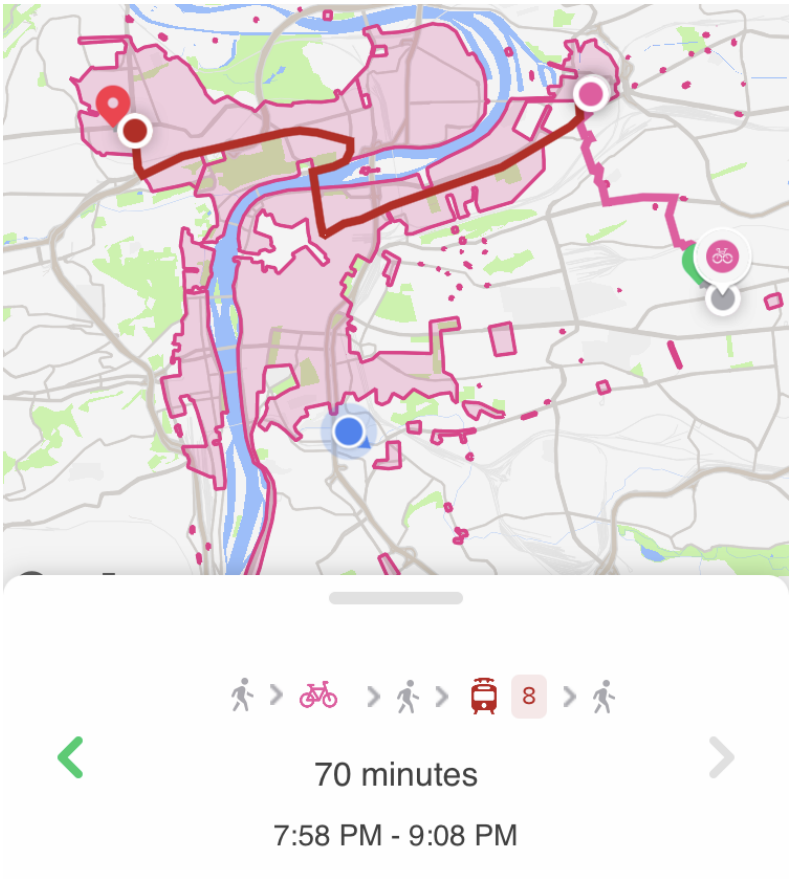
Krok 8 pomocí **Dijskrova algoritmu** [11] nalezneme v HexSpace grafu **nejkratší cestu**,

Krok 9 vrátíme nejlepší trasu nalezenou pomocí Dijskrova algoritmu v kroku **Krok 8**.

Krok 10 K trase (trasám) z předchozích kroků konkurentně spočítáme detaily všech etap trasy tak, že

- pro etapy, kde způsob dopravy je sdílený dopravní prostředek nebo chůze, využijeme OSRM backendu,
- pro etapy, kde způsob dopravy je veřejná doprava, využijeme OTP backendu.

[11] Dijskrův algoritmus je obecně definován kapitole 2.3.2.1, spolu s konkrétní implementací.



Obrázek 2.8: Konečná cesta z multimodálního plánovače s využitím hromadné dopravy, dle problému první míle.

Plánování dle problému poslední míle

Krok 1 Z požadavku od klienta převezmeme potřebné parametry. Mezi teto parametry patří

- zeměpisné délky a šířky počátečního a konečného bodu,
- výběr města (toto je z důvodu, že mapový podklad je omezen vždy na jedno město hlavně kvůli šetření paměti),
- výběr společností sdílené mobility. Pokud parametr není zadán, jsou automaticky vybrány všechny možné služby pro daný typ mobility a pro dané město.

Krok 2 vytvoříme HexSpace graf [12].

[12] HexSpace je definován v předchozí kapitole 2.2.3.

Krok 3 do HexSpace grafu vložíme 2 uzly – počáteční a konečný bod cesty a pro ně vytvoříme Hexagon objekt,

Krok 4 Zjistíme, zda konečný bod je v zóně, kde vozidlo lze vrátit.

- Pokud není, algoritmus ukončíme a vrátíme prázný seznam cest.
- Pokud ano, vybereme vozidla v okolí konečného bodu[13] a přidáme je jako uzly do HexSpace grafu,

Krok 5 vybereme přestupní stanice v okolí konečného bodu[14] a přidáme je jako uzly do HexSpace grafu,

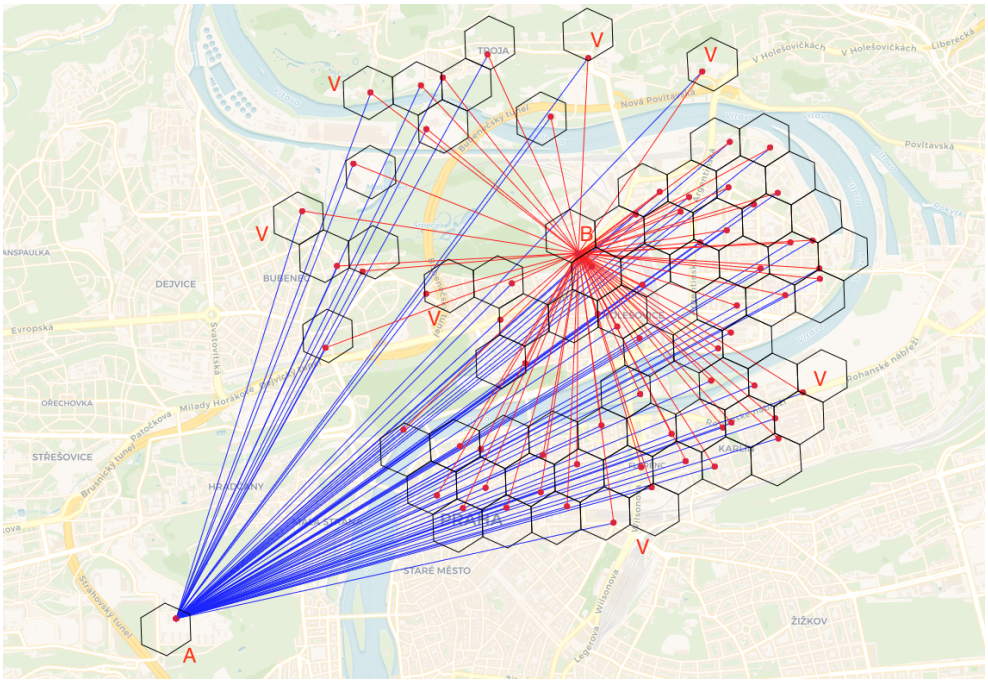
Krok 6 asynchronně spočítáme váhy jednotlivých etap tras. (tyto váhy jsou v pozdějším **Kroku 7** přiřazeny jako ohodnocení hranám v HexSpace grafu). Jako váhu etapy (ohodnocení hrany) považujeme čas, který je třeba na etapu. Pomocí OSRM a OTP backendů spočítáme

- Time matrix[15] z počátečního bodu do bodů se sdílenými prostředky (*one to many*).
- Time matrix[16] z bodů s vozidly do konečného bodu pomocí módu sdíleného prostředku (*many to one*),

Krok 7 Do HexSpace grafu přidáme

- hrany z počátečního uzlu do uzlů se sdílenými vozidly vybranými v předchozím kroku (s přidělenou váhou vytvořenou pomocí OTP v bodu výše; a módem hromadné dopravy),
- hrany z uzlů se sdílenými vozidly do konečného uzlu s přidělenou váhou a módem daného sdíleného prostředku,

[13] Vozidla v okolí bodu jsou vybírána pomocí Harvesinova vzorce, který je popsán v kapitole 2.4.4.2.
[14] Výběr přestupních stanic je popsán v kapitole 2.4.4.3.
[15] Endpoint OTP s časovou maticí je popsán v 2.4.2.2
[16] Endpoint OSRM s časovou maticí je popsán v 2.4.2.1



Obrázek 2.9: Multimodální graf pro plán cesty z bodu A do bodu B, metodou poslední míle. Černé hexagony T obsahují přestupní stanice (Označeny jako červené body). Body V obsahují lokace se sdílenými prostředky. Modré hrany označují zjednodučenou cestu pomocí hromadné dopravy, červené hrany označují cesty pomocí sdíleného prostředku. K vizualizaci grafu je využita knihovna Folium. [98]

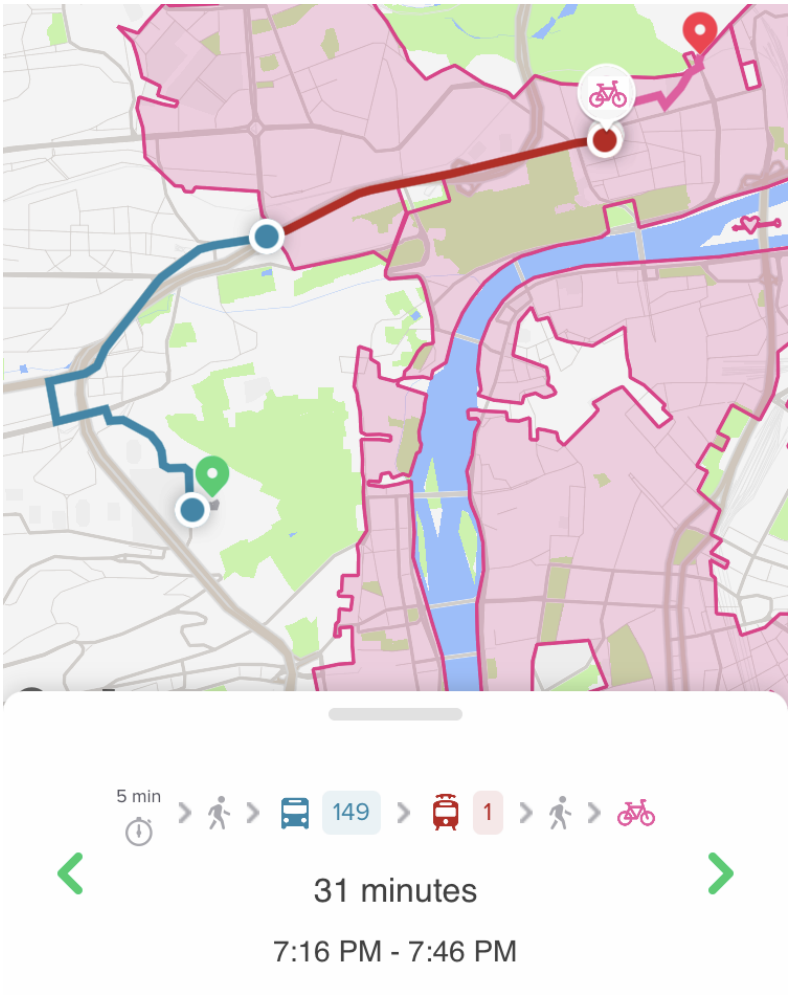
Krok 8 pomocí **Dijskrova algoritmu** [17] nalezneme v HexSpace grafu **nejkratší cestu**,

Krok 9 vrátíme nejlepší trasu nalezenou pomocí Dijskrova algoritmu v kroku **Krok 8**.

Krok 10 K trase (trasám) z předchozích kroků konkurentně spočítáme detaily všech etap trasy tak, že

- pro etapy, kde způsob dopravy je sdílený dopravní prostředek nebo chůze, využijeme OSRM backendu,
- pro etapy, kde způsob dopravy je veřejná doprava, využijeme OTP backendu.

[17] Dijskrův algoritmus je obecně definován kapitole 2.3.2.1, spolu s konkrétní implementací.



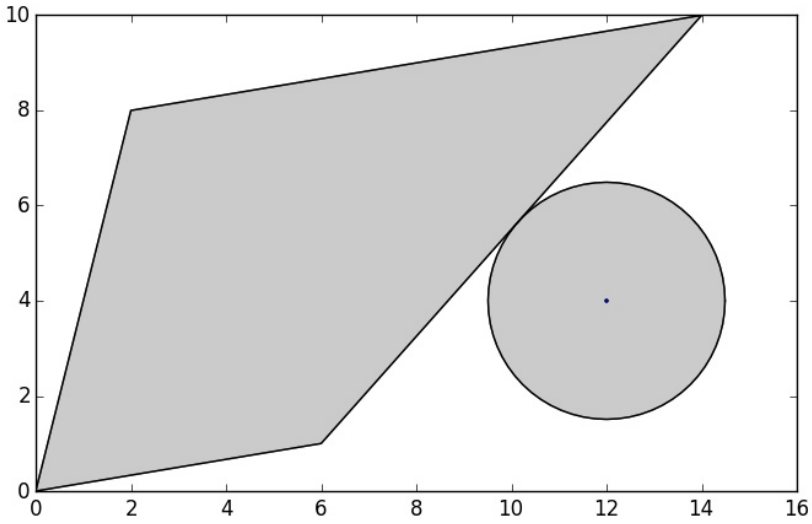
Obrázek 2.10: Konečná cesta z multimodálního plánovače s využitím hromadné dopravy, dle problému poslední míle.

2.4.4 Pomocné funkce plánovače

Podkapitola níže popisuje pomocné funkce, které jsou využity k plánování tras.

2.4.4.1 Výběr bodů ze servisních zón

K manipulaci s prostorovými daty je k výběru nejbližšího bodu v servisní zóně využito knihovny shapely [79]. V tomto případě máme nějaký konkrétní bod (buď cílový bod cesty u singlemodálních cest, nebo cílový krok etapy u multimodálních cest) a nějaký polygon, značící servisní zónu, kam sdílený prostředek můžeme vrátit.



Obrázek 2.11: Servisní zóna a bod mimo zónu

Diagram 2.11 zobrazuje tyto dva objekty. K nalezení nejbližšího bodu v servisní zóně je využito funkce `nearest_points(geom1, geom2)`, kde `geom1` a `geom2` jsou nějaké geometrické objekty.

2.4.4.2 Výběr relevantních docků

Někteří poskytovatelé sdílené mobility vyžadují, aby jejich vozidla byla vracena ve stojanech. Následující odstavec popisuje, jak relevantní stojany vybrat. Výběr stojanů je poměrně jednoduchý, tedy v okruhu (s předem daným poloměrem) od předem daného bodu zájmu (například cílový bod cesty).

Použitý algoritmus je naivní, tedy, že cyklem procházíme přes všechny stojany, a pro každý stojan spočítáme vzdálenost od bodu zájmu dle Harvesinova vzorce. Pokud je vzdálenost stojanu od bodu menší, než předem daný poloměr, stojan vybereme pro pozdější výpočty.

Pomocí Harvesinova vzorce[81] je možné spočítat vzdálenost dvou bodů jako vzdálenost dvou bodů po kulové ploše. Harvesinova metoda obsahuje nepřesnost, protože bere jako model kouli (skutečný tvar Země se od tvaru koule mírně liší, což ale na kratších vzdálenostech, jako cesty po městě, je zanedbatelný problém).

2.4.4.3 Výběr přestupních stanic

Multimodální cesty jsou realizovány pomocí takzvaných *tranzitních* (přestupních) stanic. Jako *tranzitní* stanici můžeme považovat takovou stanici, která je v určité vzdálenosti od bodu zájmu (tato vzdálenost je dána konfigurací).

V současné době jsou přestupní stanice hledány naivně, tedy cyklem procházíme přes všechny stanice a pro každou stanici spočítáme vzdálenost od bodu zájmu pomocí Harvesinova vzorce[81]. [18]

- V případě, že předpoklad platí, stanice je vybrána do dalšího kroku.

2.5 Kontejnerizace plánovače

V následující kapitole popíši vytvoření kontejneru pro plánovací aplikaci. Následovné nasazení kontejneru do Kubernetes je popsáno v kapitole **??**.

Jelikož je aplikace psána v Pythonu pomocí frameworku FastAPI[68], bude využita jako podkladová vrstva kontejneru obraz `debian:buster`[128], resp. na něm postavený obraz `tiangolo/uvicorn-gunicorn-fastapi:python3.8`[129].

Dockerfile vypadá následovně

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.8

RUN apt-get -y update
# Copy app to remote container
COPY . /app

# Set working directory
WORKDIR /app

RUN python3 -m pip install -r requirements.txt
EXPOSE 6004
ENV PYTHONPATH=/app

ENTRYPOINT ["sh", "docker-entrypoint.sh"]
```

Soubor `docker-entrypoint.sh` obsahuje

```
#!/bin/bash
set -e
uvicorn planner.main:app --host 0.0.0.0 --port 6004 --log-level debug
```

[18] Myšlenka Harvesinova vzorce je popsána v kapitole 2.4.4.2.

Kapitola 3

Architektura plánovače

V následující kapitole popíši architekturu plánovače z předchozí části práce spolu s následovným nasazením zmíněného plánovače. Prvně budou popsány koncepty, které budou využity, jako např. mikroslužby, virtualizace, atp. Následně bude popsán systém Kubernetes, do kterého bude aplikace nasazena a bude vybrána nejvhodnější distribuce Kubernetes. Posledně budou popsány všechny předpisy a nastavení, která jsou nutná pro úspěšné nasazení plánovací aplikace.

3.0.1 Koncept mikroslužeb

Architektura mikroslužeb je způsob návrhu aplikace, při kterém míříme k vytvoření skupiny malých, lehkých a vzájemně nezávislých služeb. Každá ze služeb běží ve svém procesu nezávislém na ostatních[18]. Všechny služby mezi sebou komunikují na daném mechanizmu (protokolu).

Opačným přístupem je **monolitická architektura**. U monolitického přístupu je celá aplikační logika sepsána jako společný *codebase*[17]. Toto přináší některé nevýhody – v případě, že programátor chce změnit jakkoli malou část kódu, je vždy třeba zkompilovat, sestavit a nasadit celou aplikaci. Tato vlastnost monolitické architektury může způsobit komplikace při vývoji aplikace, a to obvzlášť v případě, že aplikace je velká a na aplikaci souběžně pracuje více vývojářů.

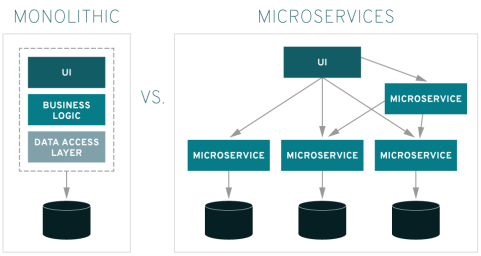
I proto jsou automatická nasazení (*CI/CD*) mnohem jednodužší v architektuře mikroslužeb[19]. V takovém případě není nutno kompilovat, či sestavovat celou aplikaci, ale pouze danou mikroslužbu, kterou měníme. Jedním z důsledků je i to, že se můžeme jednodušeji orientovat v kódu a také jednodušeji nacházíme a opravujeme *bugy*.

Díky tomu můžeme vyvíjet a následně nasazovat do produkčního prostředí nové verze aplikace nepoměrně rychleji. Proto je také možno nasazovat menší změny a to s rychlejší frekvencí.

Dalším problémem monolitické aplikace může být škálování – a to proto, že celá aplikace musí být škálovatelná. U mikroslužeb, kde každá komponenta, či služba je izolovaná (a pokud je služba navržena jako škálovatelná, pak je samostatně škálovatelná), můžeme v případě potřeby naškálovat jen ty části, u kterých je to aktuálně z důvodu vysokého zatížení nutné. Tím ušetříme za HW zdroje u služeb, které aktuálně vysoké vytížení nemají. Takový proces oddělení se nazývá *decoupling* [19].

Mezi výhody také patří to, že můžeme jednotlivé instance mikroslužeb rozdělit a distribuovat na několik fyzických strojů, či dokonce datacenter. Pro to, aby služby, co běží na různých serverech, spolu komunikovali, je nutné, aby byl implementovaný *Service Discovery* protokol [20].

V kontrastu k mikroslužbám musí být monolitická architektura nasazena ve stejném prostředí a to buď na jednom stroji, připadně rozdělena pomocí HA funkcionality (loadbalanceru).

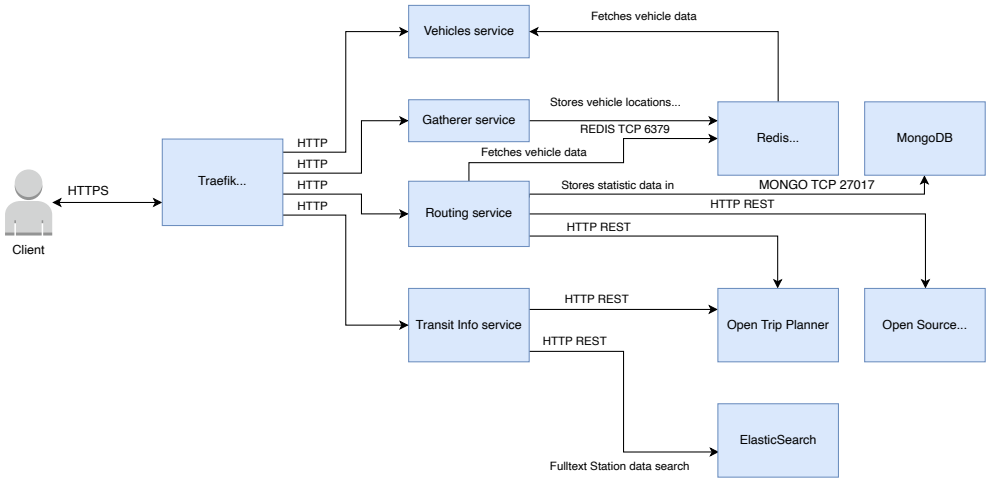


Obrázek 3.1: Monolitická architektura vs. architektura mikroslužeb. Diagram převzat z [21].

Diagram výše zobrazuje hlavní rozdíly mezi monolitickou a mikroslužbovou architekturou. I přes zjevné výhody architektury založené na mikroslužbách, je nutné podotknout i její nevýhody. Jedním z nich je interní komunikace napříč mezi službami. Ty mohou být implementovány buď jako *REST*, *GraphQl API*, či jako *RPC* komunikace. Díky tomu by měla být brána v zřetel odezva mezi službami. Další věc je, že taková to komunikace vyžaduje stabilní a bezpečnou síť. Nevýhodou také může být nustnost orchestrace, která přichází s mikroslužbami. Manuální nasazování může být u mikroslužeb nepřehledné a může vést k lidským chybám, jak je dále rozvedeno v kapitole 3.1.7.1. V ekosystému mikroslužeb je také nutné si vytvořit testovací prostředí, protože kvůli komplexitě návrhu přichází ztížený vývoj. Také je třeba vytvářet API pro každou službu a API dokumentovat.

Také s tím, jak aplikace založená na mikroslužbách začíná narůstat, se začíná objevovat potřeba pro kvalitní *monitoring* a *tracing*.

Diagram 3.2 ukazuje rozdělení služeb nutných k plánování tras. Jednotlivé služby jsou blíže popsány v kapitole 3.1.7.2.



Obrázek 3.2: Rozvržení služeb nutných k plánování tras

3.1 Distribuované výpočetní systémy

V sekci níže nejdříve popíši koncepty nutné k pochopení distribuovaného systému, do kterého bude plánovač nasazen. Prvně budou ukázány koncepty, které jsou využity, jako virtualizace, následně kontejnerová virtualizace a její implementace pomocí technologie *Docker*. Dále bude popsán orchestrátor kontejnerů, *Kubernetes* a výběr jeho distribuce.

3.1.1 Virtualizace

Virtualizace je koncept zavedený již dekády[22]. Koncept odkazuje na vytvoření virtuálního objektu (zdroje), jako je např. operační systém, hardware, úložiště, či síť, který emuluje objekt na reálné vrstvě.

Tyto emulované a virtualizované systémy mohou být nastavovány, udržovány a replikovány mnohem jednodužšeji (a na požádání). Další věcí je, že virtualizací jsou zdroje přiřazovány uživatelům (či aplikacím) dle jejich reálných potřeb a díky tomu mohou být zdroje mnohem lépe využity. Takový přístup pomáhá jak ke snížení nákladů, tak také k šetření životního prostředí.

Poskytovatelé cloudových služeb, jako *Amazon AWS*[23], či *Microsoft Azure*, ale i mnoho dalších využívají virtualizace v jejich datacentrech a následně zdroje nabízejí jako službu (*IaaS – Infrastructure as a service*).

Virtualizaci můžeme najít jak na serverové části, tak na té klientské. V této práci budou zmíněny jen ty na serverové části.

Dvě hlavní virtualizační techniky, tedy virtualizace založená na hypervisoru a na kontejnerech jsou popsány níže.

3.1.2 Hypervizor virtualizace

Virtualizace založená na hypervizorech umožnuje spouštění plných virtuálních strojů (*Virtual machines*) na hypervisoru. Hypervisorem zde myslíme takovou součást, která řídí přístup virtualizovaných počítačů k hardwaru hostitelského počítače, řídí jejich běh a zároveň je od sebe odděluje[24]. Takové virtuální stroje se skládají z plného operačního systému, obsahujícího kernel, aplikaci a všechny závislosti[7].

Hypervisory se dle Robert P. Goldberga dále dělí do dvou základních částí, Hypervisor typu 1 a typu 2[24]:

- **Hypervisor typu 1** (také jako Baremetal Hypervisor) běží přímo na hardwaru daného počítače. Mezi ně patří například *Xen*, nebo *VMWare ESXi*[25].
- **Hypervisory typu 2** (také jako hostovaný hypervisor) běží jako další vrstva nad operačním systémem. Mezi hostované hypervisory patří *VirtualBox*, *QEMU*, či *VMWare Workstation*.

3.1.3 Kontejnerová virtualizace

Virtualizace založená na kontejnerech (také nazývaná jako virtualizace na úrovni operačního systému, případně *kontejnerizace*) je odlehčená alternativa k hypervisorům. Tento způsob *kontejnerizace* využívá funkcí kernelu ze serveru, na kterém virtualizace běží a to tím, že vytváří instance oddělených *„user-spaců“* (ve kterých běží skupiny procesů), nazývaných kontejnery.

Kontejner se tváří z pohledu uvnitř běžícího procesu jako plnohodnotný operační systém. Reálně je to ale oddělený jmenný prostor uvnitř hostujícího operačního systému, se kterým sdílí zdroje (sdílí společný kernel, tedy žádný hypervisor není potřeba). A tedy kontejnery nemají svůj virtualizované hardware (jako např. výše zmíněná hypervizor virtualizace).

Díky tomu, že kontejnery *neemulují* žádný hardware, nepotřebují čas pro nastartování operačního systému. Proto nabízí velmi rychlé startovací časy (v milisekundách)[26]. Kontejner do sebe zabaluje všechny závislosti, které může potřebovat – jako například knihovny, binární soubory, či jiné konfigurace potřebné k nastavení operačního systému nebo aplikace.

Virtualizace založená na kontejnerech může být implementována na jakémkoli operačním systému, každopádně populární techniky, jako třeba Docker jsou založeny na funkcích v Linux kernelu[27].

3.1.3.1 Mechanismy kontejnerizace

Kontejnery jsou většinou vytvořeny pomocí následujících funkcí v jádře operačního systému: *kernel namespace* (jmenný prostor v jádru) a *cgroups* (control groups)[28].

Tyto funkce se zaměřují na vytváření skupin procesů, které jsou od sebe odděleny (*kernel namespaces*) a vynucují jim limity na zdroje (*control groups*).

Control grupy (*cgroups*) jsou využívány na vynucení limitů na hardware zdroje, jako např. počet procesorů, procesorové využití, přiřazení paměti, atp. Taková omezení mohou být přiřazeny buď jednomu procesu, či množině procesů[29]. *Cgroups* mohou být využity k zajištění, aby jeden kontejner nezahltil systém využitím všech jeho zdrojů.

Pravidla jsou organizována ve stromové struktuře, jsou děděny a volitelně vrstveny.

Cgroups mohou být vnímány jako vylepšení nad *ulimity*/*rlimity*. Nastavují se pomocí speciálního virtuálního souboru připojeném v cestě `/sys/fs/cgroup` a mohou být kdykoliv měněny.

Mezi hlavní skupiny *cgroup* patří *CPU*, *memory*, *BLKIO*, *devices*, *network* nebo *freezer*. V případech, kdy by některé z cgrup mohly být špatně špatně nastaveny, by mohla být taková chyba využita k „útěku“ z kontejneru ven (tzv. *priviledge escalation*)[31].

Právě Docker virtualizaci se se bude věnovat další podkapitola.

3.1.4 Docker

V minulé části jsem popsal obecně kontejnerovou virtualizaci, zde popíši reálnou implementaci. *Docker* přidává abstraktní vrstvu nad koncepty zmíněnými výše. *Docker* je platforma na vývoj, distribuci a nasazování aplikací[30].

Docker se zkládá z těchto funkcí:

- **Docker Engine** (jádro Docker ekosystému)
- **Docker Compose** (definice celé infrastrukury pomocí jednoho souboru, nenabízí detailní konfiguraci a proto nebude využit)
- **Docker Swarm** (orchestrace kontejnerů na HA klastrů, v práci nebude využito)
- **Docker Registry** (úložiště Docker obrazů)
- **Universal Control Plane** (management kontejnerů a klastrů v byznys prostředí, nebude využito)
- **Docker Secrets** (management hesel ve Swarmu)
- **Docker Content Trust** (ukládání a validace „značek“ (*tagů*) u Docker obrazů)

Relevantní komponenty popíši níže více detailně.

3.1.4.1 Docker Engine

Docker Engine je jádro Docker ekosystému, založený na client-server architektuře, která má 3 hlavní komponenty – Docker démon, REST API poskytované Docker démonem a CLI klient (příkaz `docker`).

Docker Daemon

Docker démon běží na stroji (jako `root`[19]) a je zodpovědný za naslouchání na REST API, odkud zpracovává požadavky od Docker klientů. Spravuje také Docker objekty pro kontejnery, obrazy instancí, sítě, diskové objekty.

Docker Client

Docker klient je využíván pro komunikaci s API na démonu. Toto je primární způsob komunikace s démonem.

V této práci je *Docker* využit jako *container-runtime* plánovací aplikace a poté je nasazen do *Kubernetes*, se kterým *Docker* komunikuje pomocí **Container Runtime Interface**[32].

3.1.4.2 Docker image a Docker kontejner

Docker obraz[20] je soubor vytvořený z jeho definice či šablony, zvaného Dockerfile.

3.1.5 Frameworky pro orchestraci kontejnerů

V současné době patří mezi nejpopulárnější frameworky pro orchestraci kontejnerů *Docker Swarm*[35], *Kubernetes* a *Apache Mesos*[36]. Mezi jejich nejčastěji zmiňované výhody patří vysoká dostupnost (*HA*) při nasazování do homogenních prostředí v datacentrech. Z důvodu nejvyšší podpory *Kubernetes* u poskytovatelů cloudových služeb a také v současnosti největší komunitě se budeme v příštích kapitolách věnovat právě jemu a konkurenční technologie vynecháme. Detailnější porovnání orchestračních nástrojů lze nalézt například v [34].

3.1.6 Kubernetes

Kapitola níže uvede komponenty *Kubernetes* pro potřeby nasazení plánovací aplikace. Každopádně pro kompletní a více detailní informace doporučuji oficiální dokumentaci, dostupnou v [37].

Jak definuje oficiální dokumentace [37], *Kubernetes* je orchestrační systém pro kontejnery, navržený pro nasazení, škálování, řízení a kompozici aplikačních

[19] Root je privilegovaný uživatel na *Unix-like* operačních systémech
[20] Docker obraz – Docker image

konternerů napříč klustery serverů (s využitím v produkčním prostředí). Je to robustní systém pro řízení kontejnerů, který nabízí virtuální abstrakční vrstvu nad poskytovatelem cloudových služeb a je velmi užitečný pro nasazování a udržování škálovatelných a distribuovaných systémů. Další velkou výhodou je, že pomáhá uživatelům konzistentně nasazovat aplikace na platformy od různých aplikačních poskytovatelů.

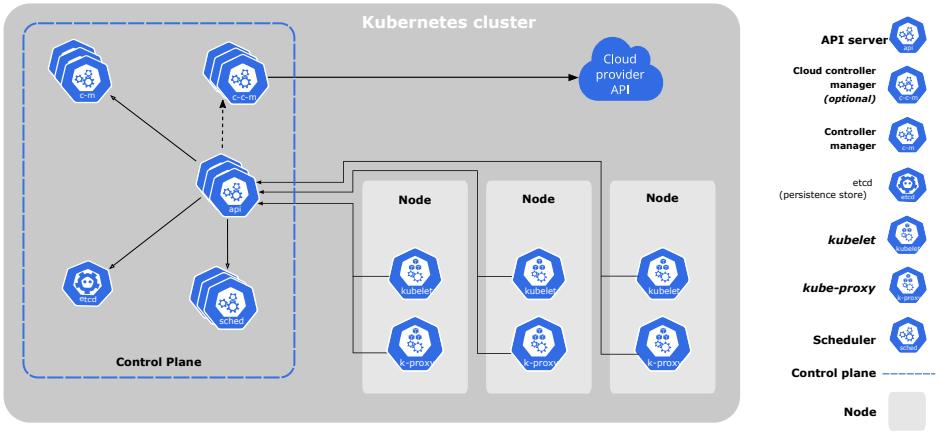
Kubernetes je třetí generace služby pro koordinaci kontejnerů od Googlu, představen na **Google Developer Forum** v červnu 2014. *Kubernetes* použil mnoho nápadů z předchozí technologie od Googlu – *Borgu*. Původně byly interní služby a aplikace v Googlu spouštěny právě pomocí *Borgu*, každopádně později se o podobnou technologii začli zajímat i mimo korporaci. Toto motivovalo Google k vývoji právě *Kubernetu*.

3.1.6.1 Architektura Kubernetes

Kubernetes uvádí aplikačně orientovanou architekturu [21] [38] s motivací:

- Co nejvíce zabstraktnit HW stroj a operační systém od aplikace a nasazení,
- razí pravidlo jedoho procesu na jeden kontejner. Proto pokud řídíme a ovládáme proces, ovládáme i aplikaci. Proto přesouvá *Kubernetes API* z infrastrukturně orientovaného na aplikačne orientovaný a zlepšuje „vhled“ (*observability*) na aplikace a jejich nasazení.

Kubernetes je složen ze dvou základních prvků – **Master** uzlů (také nazývané **Control Plane**) a **worker** uzlů (nazývané **Node**). Diagram 3.3 ukazuje, jaké komponenty jsou obsaženy v Control Plane a worker uzlech. Funkce komponent jsou rozvedeny v tabulkách 3.1 a 3.2.



Obrázek 3.3: Komponenty v Kubernetes architektuře. Diagram převzat z [139].

[21] AOI – Application-Oriented Infrastructure

Tabulka 3.1 popisuje komponenty, ze kterých se skládá Kubernetes Control Plane.

Komponenta	Popis
kube-apiserver	vystavuje API a frontend pro Kubernetes Control Plane
kube-scheduler	monitoruje nově vytvořené `Pody` a vybírá nody, na kterých by měly běžet
kube-controller-manager	V cyklu kontroluje a sleduje sdílený stav klastru (pomocí API serveru) a vytváří změny, kterými se snaží změnit aktuální stav na *desired state*
etcd	konzistentní a HA *key-value* úložiště pro interní data v Kubernetes

Tabulka 3.1: Komponenty v Kubernetes Control Plane

Tabulka 3.1 popisuje komponenty, ze kterých se skládá Kubernetes Worker uzel.

Komponenta	Popis
kubelet	Agent běžící na každém uzlu v klastru. Zajišťuje, že kontejnery běží v `Podu`.
kube-proxy	Umožňuje abstrakci z `Podu` na `Service` pomocí síťových pravidel (iptables) na OS uzlu, nastavuje *forwarding* spojení
container-runtime	software, který zajištuje, aby kontejnery běžely – podporuje několik runtimů: **Docker**, **rkt**, **runc** nebo jakýkoli jiný dle *Open Container Initiative* (*OCI*) runtime specifikace

Tabulka 3.2: Komponenty na Kubernetes uzlu

Předpisy v Kubernetes jsou perzistentní entity, které Kubernetes využívá k reprezentaci stavu klastru [140]. Konkrétně mohou popisovat:

- které kontejnerizované aplikace běží (a na jakých uzlech běží),
- zdroje dostupné těmto aplikacím,
- pravidla, dle kterých mají tyto aplikace se chovat. Mezi ně patří pravidla jak restartovat, nasazovat nové verze, nebo jak nastavit u aplikace vysokou dostupnost (*HA*).

Kubernetes předpis je záznam o úmyslu k vykonání nějaké činnosti. Když vytvoříme předpis, Kubernetes se pokusí o nastavení klastru do stavu, který

předpis definuje. Vytvořením předpisu definujeme, jak má klastr vypadat (tj. desired state).

Typ předpisu	Popis
Pod	`Pod` je základní stavební blok Kubernetes – Nejmenší a nejjednodužší jednotka v Kubernetes objekt modelu.
Service	`Service` je abstrakce definující množinu podů a pravidla, podle kterých k nim je možné přistupovat - někdy nazývaná micro-service.
ReplicaSet	Replikační kontrolér. `ReplicaSet` zajišťuje, že v určitý čas běží v clusteru určitý počet replik Podu.
Deployment	`Deployment` controller poskytuje deklarativní aktualizace pro Pods a ReplicaSets. V Deployment objektu popíšeme desired state, a Deployment controller změní aktuální stav na desired state.
StatefulSets	`StatefulSet` je API object pro workloady, využívaný pro managování stateful aplikací.
DaemonSet	`DaemonSet` zajišťuje aby všechny (nebo nějaké) nody běžely kopii podu. S tím, jak jsou nody přidány do clusteru, Pody jsou přidány s tím. Jak jsou nody odebrány z clusteru, Pody jsou sebrány garbage collecotrem.
Job	`Job` vytváří jeden nebo více podů a zajišťuje, že určitý počet z nich se úspěšně ukončí..
CronJob	`Cron Job` managuje Joby a spouští je dle nějakého časového pravidla.

Tabulka 3.3: Předpisy v Kubernetes

3.1.6.2 Vlastní nasazení Kubernetes vs Kubernetes as a Service

Kubernetes je poměrně složitý systém na nasazení a také na následnou údržbu. Jeho nastavení pro běh aplikace v produčním prostředí může být velice zdlouhavé a časově náročné. V případě bezpečnostních chyb v linuxovém jádře je třeba aktualizovat uzly (*nodes*), na kterých běží *master* i *worker* služby. Při běhu Kubernetes „on-premise“ (tj. na vlastním železe) je třeba takové změny vykonávat ručně.

Poskytovatelé cloudových služeb nabízí *Kubernetes* distribuce u kterých úkoly, jako například aktualizace verze Kubernetes, aktualizace operačního systému či například monitoring jsou již automatizované. Takové řešení se nazývá „Managed Kubernetes“, či „Kubernetes as a Service“ (*Kubernetes jako služba*).

Kvůli důvodům zmíněným výše bylo rozhodnuto, že bude využito možností managovaného přístupu k orchestrátoru. V následných kapitolách popíši, které

hlavní možnosti v roce 2020 existují a následně výběru nejvhodnější distribuci.

3.1.6.3 Výběr managované služby

Kapitola níže popíše nejvíce populární aktuálně dostupné managované distrubuce *Kubernetes*. Poté bude nastíněn její výběr. V současné době všichni největší poskytovatelé cloudových služeb nabízí managovaný *Kubernetes*. Některé jsou více integrované do jejich cloud platformy, některé méně. Cloud Native Computing Foundation (CNCF) vytváří seznam obsahující více než 70 certifikovaných Kubernetes distribucí a platforem [126]. Aby dosáhli konzistentnosti mezi platformami, zaměřují se na tři hlavní body:

- Konzistentnost – schopnost uživatele/admininstrátora konzistentně komunikovat s instalací Kubernetes,
- Časté aktualizace – je požadováno od poskytovatelů distribucí aktualizovat dostupné verze (alespoň jednouu ročně),
- Potvrditelnost – jakýkoli uživatel musí mít možnost potvrdit správnost pomocí nástroje Sonobuoy [127].

Z CNCF seznamu byly vybrány distribuce hlavních cloudových poskytovatelů, ukázané níže.

GKE (Google Kubernetes Engine)[75] je služba nabízená na *GCP (Google Cloud Platform)*. Díky tomu, že je poskytován od Googlu, nabízí nejvyšší propojenost mezi GCP cloudem a Kubernetes. Skoro všechny nastavení je možné vykonat z webového dashboardu. GKE nabízí podporu pro service mesh pomocí Istio, které se u jíných poskytovatelů musí nastavovat. Jednou z hlavních výhod u GKE je podpora aktualizací uzlů a Control Plane, které jsou dostupné většinou hned po vydání nové verze vanilla Kubernetes (u ostatních poskytovatelů čekání může být delší).

EKS (Elastic Kubernetes Service)[76] je služba od *AWS (Amazon Web Services)*. EKS nabízí velké propojení s AWS službami, což s sebou nese výhody, i nevýhody. Výhodou je, že užití EKS je poté téměř bez operačních úkonů a běží bez větších zákroků. Nevýhodou je tzv. „vendor-lockin“, tj. vysoká komplexita při přechodu z jedné cloudové platformy na druhou. Nevýhodou může být, že není možné vše nastavit přes AWS konzoli (*AWS Management Console*, webové rozhraní AWS). V případě, že ale definujeme infrastrukturu jako kód (přes Terraform, jak je popsáno níže) toto není velkou nevýhodou.

AKS (Azure Kubernetes Service)[101] je managované Kubernetes řešení od *Microsoft*u, dostupné od roku 2018. AKS je schopný běžet jak na Azure veřejném cloudu, tak *on-premise*, což pomáhá při vysoké dostupnosti u kritických aplikací. AKS může být výhodné v případech, kdy vyžadujeme bezproblémovou integraci s ostatními nástroji od Microsoftu, jako jsou Visual Studio, nebo Active Directory.

Popis funkcionality	EKS	AKS
Aktuálně podporovaná Kubernetes verze	1.17 (default) 1.16 1.15 1.14	1.19 (preview) 1.18 1.17 (default) 1.16
Počet podporovaných minor verzí	>=3 + 1 deprekovaná	3
Původní GA release datum	Červen 2018	Červen 2018
V souladu s CNCF Kubernetes	Ano	Ano
Poslední CNCF-certified verze	1.17	1.18
Upgrade proces Control plane	Inicializováno uživatelem Uživatel musí nmanuálně aktualizovat systémové služby, které běží na uzlech (e.g., kube-proxy, coredns, AWS VPC CNI)	Inicializováno uživatelem
Uprade proces nodů	Nemanagované node grupy: vše inicializované a managované uživatelem Managované node grupy: inicializované uživatelem; EKS „drainuje“ a nahradí za nody s novou verzí	inicializované uživatelem; AKS „drainuje“ a nahradí za nody s novou verzí
OS nodů	**Linux:** **Amazon Linux 2 (default); Ubuntu (partner AMI)** **Windows:** **Windows Server 2019**	**Linux:** **Ubuntu** **Windows:** **Windows Server 2019**
Runtime kontejnerů	Docker (default)	Docker (default) containerd
HA možnosti na Control plane	Control plane je nasazen napříč několik AZ (default)	Control plane komponeny jsou nasazeny napříč několik zón, které jsou definovány adminem
SLA na Control plane	99.95%	99.95% (SLA s finanční zárukou) 99.9% (bez fin. záruky)
SLA s finanční zárukou	Ano	Ano
Cena	0.10/hodina (USD) za cluster + standardní ceny za EC2 instance a jiné AWS zdroje	Pay-as-you-go: Standardní ceny za virt. instance a jiné zdroje
Control plane: sběr logů	Volitelné Default: Vypnuto Logy jsou posílány do AWS CloudWatch	Volitelné Default: Vypnuto Logy jsou posílány do Azure Monitor
Performance metriky kontejnerů	Volitelné Default: Vypnuto Logy jsou posílány do AWS CloudWatch Container Insights	Volitelné Default: Vypnuto Logy jsou posílány do Azure Monitor
Monitoring zdraví nodů	Žádná Kubernetes-aware podpora; pokud uzel přestane odpovídat, AWS ASG uzel nahradí	Auto repair dostupný. Node status monitoring dostupný. Používá autoscaling rules na přesun workloadů.

Tabulka 3.4: Porovnání managovaných distribucí Kubernetes a „vanilla“ Kubernetes, 1. část. Data z [124], [122], [123], [125].

Popis funkcionality	**GKE**	Kubernetes
Aktuálně podporovaná Kubernetes verze	1.17 1.16 1.15 (default) 1.14	1.19 1.18 1.17
# podporovaných minor verzí	4	3
Původní GA release datum	Srpen 2015	Červenec 2015 (Kubernetes 1.0)
V souladu s CNCF Kubernetes	Ano	Ano
Poslední *CNCF-certified* verze	1.17	-
Upgrade proces Control plane	Automaticky upgradované během maintenance window, může být inicializováno uživatelem	-
Uprade proces nodů	Automaticky upgradované (default) během maintenance window, může být inicializováno uživatelem; GKE „drainuje“ a nahradí za nody s novou verzí	-
OS nodů	**Linux:** **Container-Optimized OS (COS) (default), Ubuntu** **Windows:** **Windows Server 2019** **Windows Server version 1909**	
Runtime kontejnerů	Docker (default) containerd gVisor	**Linux:** **Docker** **Containerd** **Cri-o** **rktlet** **any runtime that implements the Kubernetes CRI (Container Runtime Interface)** **Windows:** **Docker EE-basic 18.09**
HA možnosti na Control plane	**Zonal clustery: 1 control plane** **Regional clustery: 3 Kubernetes control planes kvórum**	Podporováno
SLA na Control plane	Zonal clustery: 99.5% Regional clustery: 99.95%	-
SLA s finanční zárukou	Yes	-
Cena	$0.10/hodina (USD) za cluster + standardní ceny za GCE instance a jiné GCP zdroje	-
Control plane: sběr logů	Volitelné Default: Vypnuto Logy jsou posílány do Stackdriver	-
Performance metriky kontejnerů	Volitelné Default: Vypnuto Logy jsou posílány do Stackdriver	-
Monitoring zdraví nodů	Node auto-repair zapnut (default)	-

Tabulka 3.5: Porovnání managovaných distribucí Kubernetes a „vanilla“ Kubernetes, 2. část. Data z [124], [122], [123], [125].

Tabulky 3.4 a 3.5 porovnávají výše zmíněné distribuce. Pro porovnání je přidán „vanilla“ Kubernetes. Data jsou aktuální k listopadu 2020.

3.1.6.4 Výběr Kubernetes distribuce

Všechny výše zmíněné distribuce jsou implementované na vysoce kvalitní úrovni, nabízí SLA a nové verze Kubernetes jsou poměrně často vydávané. Velkou výhodou GKE oproti konkurenci jsou aktualizace, které jsou plně automatizované. Naopak u AKS a EKS je potřebná alespoň nějaká interakce člověka. Další výhodou u GKE je podporovaná škála operačních systémů na uzlech a škála runtime kontejnerů (*container runtimes*), mnou preferovaný Docker je ale podporovaný všemi. Ve výsledku tedy hlavním rozhodujícím faktorem je familiarita s danou cloud platformou.

Nakonec jsem vybral právě EKS díky mým osobním největším znalostem právě s AWS platformou, protože studium a správné nastavení ostatních platforem může zabrat nezanedbatelnou dobu.

Návrh a nastavení clusteru bude popsán v praktické části práce.

3.1.7 Distribuovaná architektura

V kapitole níže prvně popíši koncept softwarově definované architektury, dále představím nástroj na její implementaci (Terraform). Následně bude představen návrh architektury v AWS cloudu a EKS klastru.

3.1.7.1 Softwarově definovaná infrastruktura

S příchodem cloudu a virtualizace jako takové přišel i nespočet nových nástrojů a platforem, díky kterým začne vznikat portfolio systémů, o které se většinou musíme starat. V důsledku komplexnosti portfolia pomocných programů tedy i krok k více softwarově definované infrastruktuře. Softwarově definovaná architektura (většinou nazývaná jako *Infrastructure as Code*, zkráceně *IaC*) je pokus o využití maximálního potenciálu aktuální IT infrastruktury. Kief Morris [44] popisuje *IaC* následovně

> Infrastruktura jako kód je přístup k administraci IT infrastruktury v době cloudu, s využitím mikroslužeb, automatického nasazování a založeném na praktikách ze softwarového inženýrství.

Pro replikovatelnost řešení této práce budeme používat právě *IaC* pro popis nasazované infrastruktury.

Výhody IaC

Využití Softwarově definované architektury má několik hlávních výhod, které jsou popsané níže.

- **Jednoduše a rychle reprodukovatelné systémy**

  S použitím *IaC* mohou administrátoři rychle nasadit a nastavit celou infrastrukturu spuštěním jednoho jednoduchého příkazu, či skriptu.

  IaC skripty popisují všechny nutné kroky pro vytvoření požadovaného systému, jako například velikost instance, nainstalovaný software dané verze, nastavení firewallu, atp.

- **„Jednorázové“ systémy**

  IaC přeměnila původní, statický přístup k systémům na dynamický, kde již neměníme staré instance daného zdroje, ale pro jednoduchost vytvoříme nový. Po ujištění, že nový zdroj pracuje korektně, nahradí starou instanci. Díky tomu je možné programy přesouvat z jednoho serveru na druhý bez velkých potíží. To pomáhá také s aktualizacemi potřebných knihoven nebo operačního systému. Změna způsobu myšlení je proto nutná v případech, kdy potřebujeme dynamicky škálovat systémy a nemůžeme se spolehnout na hardware, na kterém aplikace běží. Obecně jsou tyto dva způsoby administrování architektury nazývány jako *cattle* („dobytek“, tj. nový způsob, kde neřešíme konkrétní instance) versus *pets* („mazlíčci“, tj. starý způsob, kde se o každou instanci staráme jednotlivě). [73]

- **Konzistence konfigurace**

  Lidská práce způsobuje a vždy způsobovala problémy při konzistenci konfigurace. A to i v případech, kdy jsou následovány procesy a postupy. Manuální úpravy vytvoří menší či větší odchylky od původního zdrojového kódu, což zvyšuje pravděpodobnost složitosti a časové náročnosti při opravě softwarové chyby.

  IaC plně standardizuje konfiguraci infrastruktury a díky tomu nenechává moc prostoru lidské chybě.

- **„Sebe se dokumentující“ systémy**

  Obvzáště v menších a dynamicky vyvíjených projektech se bojujeme s problémem, že dokumentace není užitečná, nebo nepřesně popisuje daný problém. S tím, jak dokumentaci či program upravujeme, tím je pravděpodobnější nekonzistentnost mezi dokumentací a kódem programu/infrastruktury. Další problémem je, že různí lidé různě zapisují dokumentaci a jejich vysvětlení problému nemusí být hned čtenáři zřejmé. I díky tomu většina dokumentací přesně nereprezentuje to, co daný problém znamená. *IaC* toto řeší tím, že definici a dokumentaci generuje přímo z kódu a právě kvůli tomu máme informace vždy aktuální. K tomu je třeba dopsat jen malé části dokumentace k dovysvětlení hůře pochopitelných částí.

- **Verzování všeho**

  S tím, jak máme definici infrastruktury v kódu, se nám otvírá možnost použít verzovací systém pro sledování změn, a případně vrácení se ke starší verze infrastruktury, v případě, že se objevily nějaké problémy.

  Verzovací systémy [22] nám nabízí zobrazení změn, obsahující všechny implementované změny, důvod, a osobu, která změny vytvořila. Taková funkce přidává užitečnost v případě oprav kódu, protože se jednouduše můžeme obrátit na člověka, který danou část změnil.

Terraform

V našem případě využijeme k popisu infrastruktury program Terraform. Terraform je nástroj na automatizaci infrastruktury od společnosti HashiCorp a je napsán v jazyce Go. Hlavní výhodou Terraformu je, že je multiplatformní, tj. podporuje mnoho různých cílů [23], mezi němi jsou cloudoví poskytovatelé, jako AWS, GCP, Azure, ale i bare metal možnosti nasazení, jako KVM (libvirt) [45].

Terraform popisuje infrastrukturu pomocí konfiguračních souborů, jež jsou psány v jazyce HCL. *Hashicorp Configuration Language* (HCL) je doménově specifický jazyk vyvinutý firmou HashiCorp.

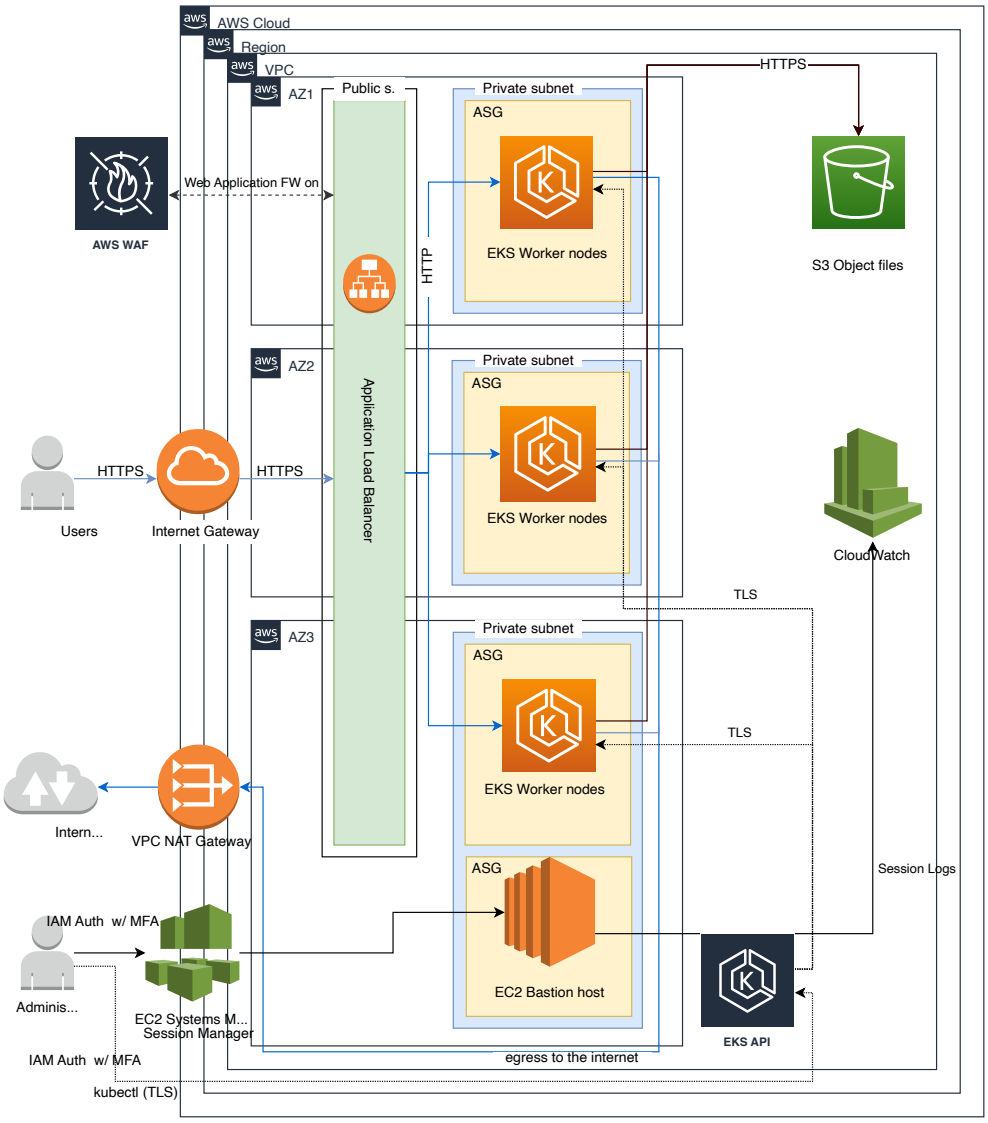
3.1.7.2 Návrh architektury

Diagram 3.4 zobrazuje rozložení služeb, které jsou využity k nasazení plánovače. Využité služby jsou

- **Amazon S3 bucket**, využitý pro ukládání objektových souborů. Mezi objektové soubory patří např. mapové podklady v `.pbf` formátu, či jiné objekty vytvořené v *preprocessing* fázi,
- *Autoscaling group* obsahující **bastion host**, díky kterému je možné se připojit do privátního subnetu obsahující Kubernetes uzly, případně jiné služby uvnitř VPC. Přístup k bastionu je řešen pomocí HTTPS pollingu přes EC2 Systems Manager,
- **NAT Gateway**, pomocí které mohou Kubernetes uzly komunikovat s Internetem,
- **Application Load Balancer** [24] s přiřazenou IP adresou, přes který Kubernetes služby komunikují s klientem,
- AWS Web Application Firewall (**WAF**), který je přiřazen k ALB a má nastavena OWASP pravidla,

[22] VCS – Version Control System
[23] v Terraform terminologii zvané backendy
[24] Nastavení ALB je diskutováno v kapitole 3.1.9.2.

- Autoscaling group obsahující **Kubernetes uzly**, nastavené napříč třemi AZ
- **CloudWatch log group**, obsahující logy z Fluentd.



Obrázek 3.4: Návrh architektury v AWS cloudu

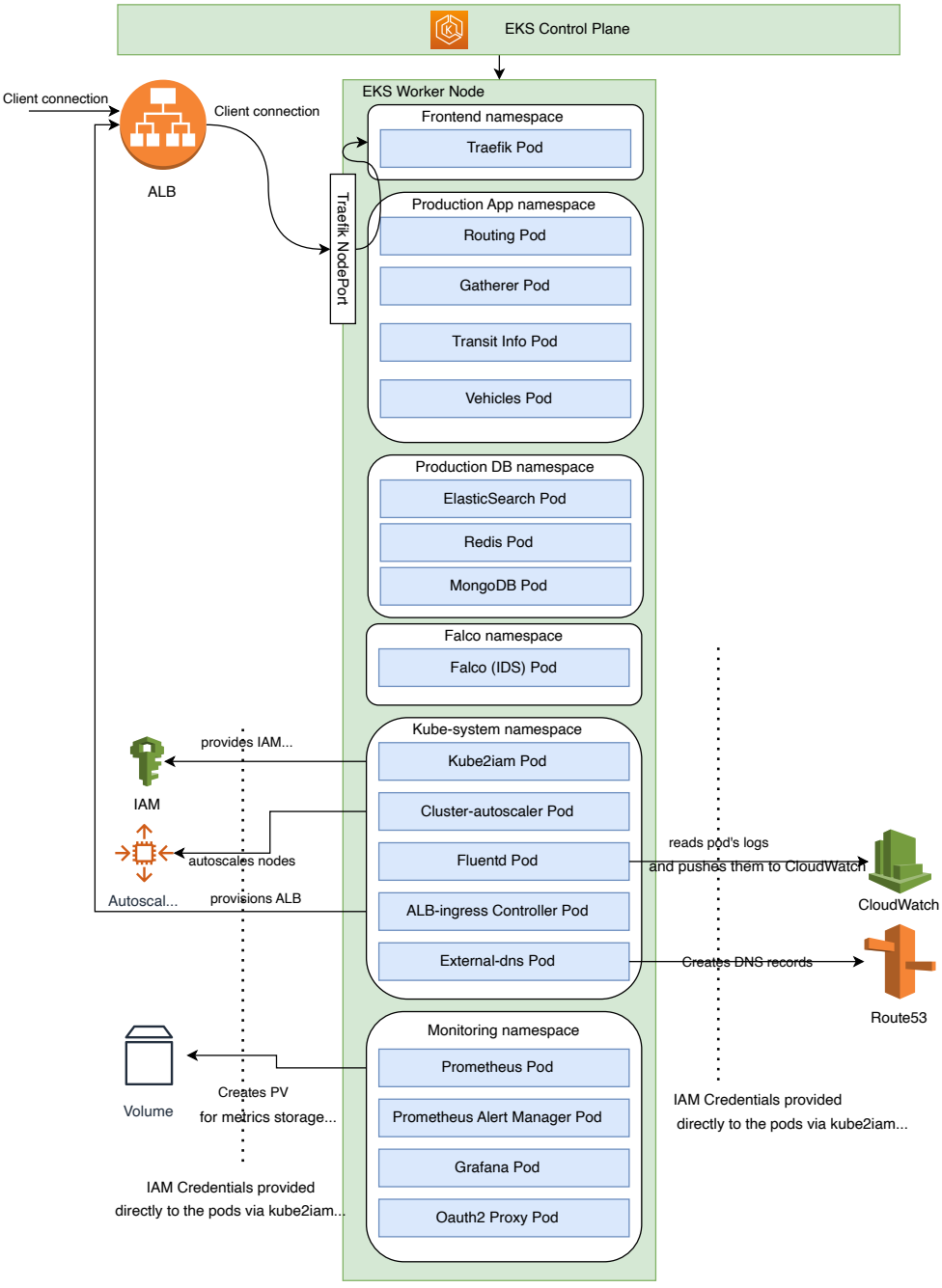
3.1.7.3 Návrh Kubernetes clusteru

Diagram 3.5 zobrazuje více do detailu návrh Kubernetes clusteru, resp. vytvořené *namespacy*, služby (předpis `Service`), resp. `Pody`.

Mezi *namespacy* jsou

- `kube-system` namespace, obsahující
  - **Kube2IAM** `DaemonSet`, popsaný v kapitole 3.1.10.1,
  - **Cluster Autoscaler** `Service`, popsaný v kapitole 3.4.1.2,
  - **Fluentd** `Service`, který sbírá logy z aplikačních kontejnerů a následně je přeposílá do služby CloudWatch. Pro délku práce již tato služba není zmiňována,
  - **ALB Ingress Controller** `Service`, popsaný v kapitole 3.1.9.2,
  - **External-dns** `Service`, popsaný v kapitole 3.1.9.3,
- `monitoring` namespace, obsahující **Prometheus Operator**, tedy Prometheus, Grafanu, Alert Manager a Oauth2 proxy pro autentizaci a autorizaci federovanou na *Google Oauth2*. Pro zkrácení práce není tato část diskutována. Jak je zobrazeno na diagramu, Prometheus si vytváří fyzické *volume* v EBS, kam ukládá sebrané metriky.
- `Falco` namespace, obsahující **Falco IDS** [136] (*Intrusion Detection System*). Z Falca jsou využity standardní pravidla pro Kubernetes[138], obsahující kontrolu neočekávaných nově vytvořených procesů, či nově vytvořených TCP spojení. Dále pravidla pro kontrolu integrity souborů[137]. Události jsou jako logy zaslány pomocí Fluentd do AWS služby CloudWatch, odkud mohou být procházeny například pomocí služby AWS Athena. Z důvodu délky této práce již služba není dále diskutována.
- `frontend` namespace, obsahující **Traefik**. Konfigurace a nasazení je diskutována v kapitole 3.1.8.1.
- `database` namespace, obsahující databázové komponenty nutné k běhu plánovače a dalších komponent. Mezi nimi jsou
  - **ElasticSearch** nasazený pomocí Helm chartu z [150], obsahující informace o zastávkách hromadné dopravy a adresách nutných pro rychlé a full textové vyhledávání na klientské aplikaci. Klientsky přístupný endpoint na *full textovým* vyhledáváním poskytuje služba *Transit Info*, jak ukazuje diagram 3.2,
  - **Redis** nasazený pomocí Helm chartu z [148], obsahující lokace a metadata sdílených prostředků, které jsou aktuálně k dispozici,
  - **Mongodb** nasazené pomocí Helm chartu z [149], obsahující analytická data o výpůjčkách, plánech tras a lokacích prostředků,
- `application` namespace, obsahující jednotlivé komponenty aplikace. Mezi nimi jsou

– **Routing** služba, které se věnuje celá první část práce,



Obrázek 3.5: Rozvržení podů a služeb v Kubernetes clusteru

– **Gatherer** služba, sbírající informace o sdílených prostředcích (a zónách) a ukládá je do Redisu,

- **Transit Info** služba, poskytující informace o stanicích hromadné dopravy (tedy informace, jako odjezdy a příjezdy na stanice, atp.) Applikace sbírá informace z *Open Trip Planner*u popsaném v kapitole 2.4.2.2. Z důvodu délky této práce již služba není dále diskutována.
- **Vehicles** služba, pracující jako interface nad informacemi o sdílených prostředcích v Redisu.

3.1.8 API brána

Když stavíme aplikaci jako množinu mikroslužeb, je nutné se rozhodnout, jak klienti aplikace budou komunikovat s jednotlivými mikroslužbami. S monolitickou aplikací máme pouze jeden (většinou replikovaný a load-balancovaný) endpoint. V architektuře mikroslužeb každá má služba množinu svých endpointů. V kapitole níže představíme, jak tento přístup ovlivňuje komunikaci mezi aplikací a klientem, dále představíme návrhový vzor zvaný API brána.

3.1.8.1 Výběr API brány

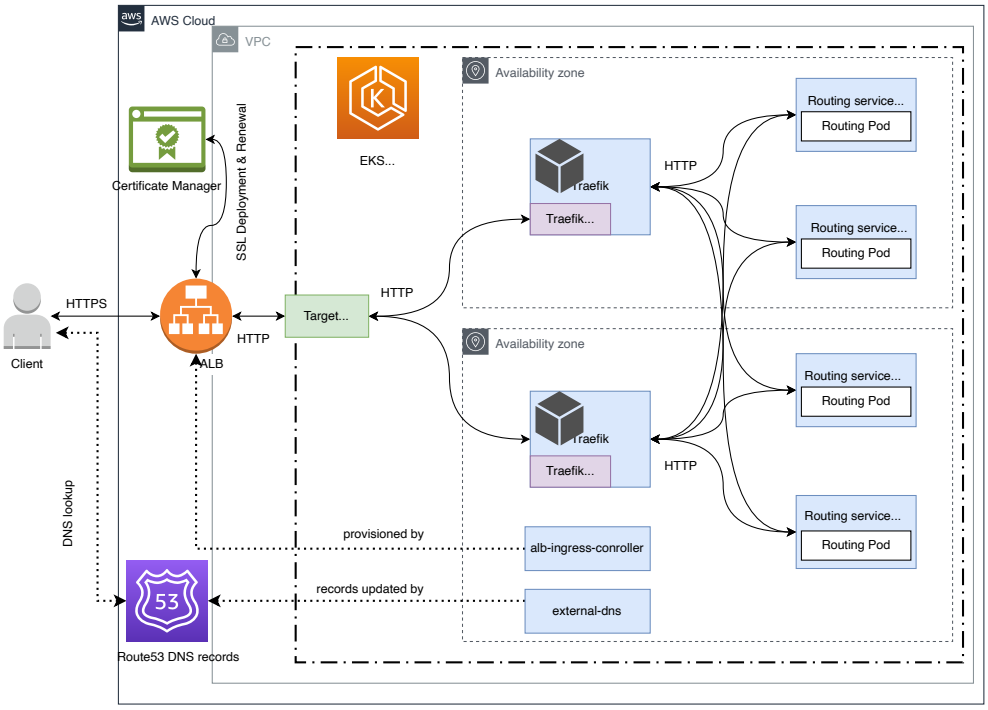
V *cloud-native* světě existuje mnoho řešení API brány. Pěkné porovnání jednotlivých řešení můžeme nalézt například v [112]. Dle CNCF průzkumu [113] vychází, že mezi nejpopulárnější patří *nginx* a a *haproxy*. Bohužel tyto aplikace mnou žádanou funkci, a to routování požadavků od klienta podle *query parametru*[25]. Po následné analýze nejlépe vycházela aplikace Traefik, které se budou věnovat následující odstavce.

Traefik

Traefik je open source load balancer a reverzní proxy. Jak bylo zmíněno výše, Traefik byl vybrán z důvodu vysoké rozšířitelnosti a konfigurovatelnosti. Navíc se vyznačuje tím, že byl stavěn přímo do *cloud-native*, narozdíl například od nginxu, či haproxy.

Na diagramu 3.6 je viditelná komunikace mezi klientskou aplikací a `Pody` v Kubernetes, jak prochází přes ALB, Traefik a nativní Kubernetes službu. Dále je na diagramu viditelný dynamický způsob nastavení DNS záznamů pomocí External-DNS a přiřazení ALB k Traefiku pomocí *ALB Ingress Controlleru*.

[25] Query řetězec je část URL ve formátu ?query_var=val.



Obrázek 3.6: Diagram komunikace klienta se službou

Co se mi nejvíce líbí na Traefiku, je jeho rozšiřitelnost, jak je možné vidět na diagramu v [114]. Požadavek od klienta přichází na *entrypoint* (který může být HTTP, Let's Encrypt self-signed/importovaný certifikát na HTTPS nebo TCP), směruje požadavek na jeden (nebo více) *middlewarů* (zde mohou být již vytvořené [114], nebo si ho můžete vytvořit sami [115]) a konečně je požadavek směrován na jednu z dle pravidel daných Kubernetes služeb.

Helm je využit k nasazení Traefiku a dalších služeb. Prvně, Traefik je nasazen z oficiálního Helm chartu[116]. Vedle standadní konfigurace, vytvoříme soubor `values.yaml`, obsahující

```
service:
  annotations: {}
  type: NodePort
```

Takto Kubernetes alokuje statický port z předem daného rozsahu a každý worker uzel v Kubernetes bude poslouchat na tomto portu, přes který bude jako proxy přeposílat požadavky na Traefik službu. Dále v práci je ukázáno, jak je ALB napojen na tento port.

Vedle standardního Traefik chartu, vytvoříme `Ingress` předpis, obsahující

```
---
# Source: traefik/templates/ingress.yaml
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik
  annotations:
    alb.ingress.kubernetes.io/actions.ssl-redirect:
      '{"Type": "redirect", "RedirectConfig":
      { "Protocol": "HTTPS", "Port": "443", "StatusCode": "HTTP_301"}}'
    alb.ingress.kubernetes.io/backend-protocol: HTTP
    alb.ingress.kubernetes.io/certificate-arn: <cert_arn1>,<cert_arn2>
    alb.ingress.kubernetes.io/healthcheck-path: /ping
    alb.ingress.kubernetes.io/healthcheck-port: "<port>"
    alb.ingress.kubernetes.io/healthcheck-protocol: HTTP
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}, {"HTTPS":443}]'
    alb.ingress.kubernetes.io/scheme: internet-facing
        alb.ingress.kubernetes.io/security-groups:
      <sg_group_id_where_alb_will_reside>
    kubernetes.io/ingress.class: alb
    external-dns.alpha.kubernetes.io/hostname: "domain1.net,domain2.com"
  labels:
    app: traefik
spec:
  rules:
    - http:
        paths:
          # HTTP to HTTPS redirect entry
          - path: /*
            backend:
              serviceName: ssl-redirect
              servicePort: use-annotation
          - path: "/*"
            backend:
              serviceName: traefik
              servicePort: 443
```

Jak je možné vidět, `Ingress` předpis obsahuje několik typů anotací. Anotace startující s `alb.ingress.kubernetes.io` a `kubernetes.io/ingress.class` jsou využity k nastavení *AWS LoadBalancer Controlleru*, `external-dns.alpha.kubernetes.io` jsou pro External-DNS, oboje více detailně popíšu níže. Pojmenování anotací by mělo být poměrně zřejmé z názvů, ale pro lepší pochopení je třeba konzultovat dokumentaci – pro AWS LoadBalancer Controller [117] a External-DNS [120].

Ingress předpis můžeme buď přidat do Helm chartu, nebo nasadit přímo pomocí `kubectl apply`.

3.1.8.2 Vytvoření loadbalanceru

Jak dokumentace[117] popisuje, AWS Load Balancer Controller je kontrolér pomáhající nastavovat Elastic Load Balancery v Kubernetes klastru. Kontrolér sleduje `Ingress` eventy a v případě, že nějaký `Ingress` předpis splňuje dané požadavky, pokusí se vytvořit AWS zdroje (ELB a jiné potřebné zdroje). My jej využijeme k nasazení ALB pro Traefik, jak ukazuje diagram 3.6.

ALB Ingress Controller je také nasazen pomocí Helmu, z EKS Helm repozitáře[118].

Vytvoříme soubor `values.yaml` obsahující

```
    # Select the region where the EKS cluster will reside
awsRegion: "eu-central-1"

# Select role which AWS LoadBalancer Controller will assume
podAnnotations:
  iam.amazonaws.com/role: "kube2iam_prod-cluster/aws-
       loadbalancer-controller"

clusterName: prod-cluster

autoDiscoverAwsVpcID: true
```

Role `kube2iam_prod-cluster/aws-loadbalancer-controller` potřebuje potřebná oprávnění[119] k vytvoření a přenastavení AWS loadbalancerů, plus další oprávnění. Roli je možné vytvořit v AWS konzoli, pomocí cli nebo Terraformu.

Konečně, nasadíme AWS Load Balancer Controller:

```
    helm install aws-loadbalancer-controller eks/aws-load-balancer-controller
 --values=values.yaml -n kube-system
```

3.1.8.3 Přiřazení DNS záznamu

Nastavovat DNS záznamy ručně nedává smysl, protože nové Kubernetes služby registrujeme a deregistrujeme poměrně často. *External-DNS* nám pomáhá s automatizací tohoto úkolu – můžeme nastavovat DNS záznamy dynamicky pomocí anotací u Kubernetes předpisů, v přístupu, který není závislý na žádném DNS poskytovateli (*DNS provider-agnostic way*)[120].

Nasadíme External-DNS z bitnami repozitáře[121]. Prvně vytvoříme soubor `values.yaml` obsahující

```
aws:
  # Select the region where the EKS cluster will reside
  region: "eu-central-1"
```

```
  preferCNAME: false

## ref: https://github.com/kubernetes-sigs/external-dns/blob/master/docs/proposal/reg
## bug: External-dns creates a TXT record with stored config,
## but External-dns creates CNAME (Cannonical) record instead of CNAME;
## Zone cannot contain CNAME and other record with the same principal)
registry: "noop"

podAnnotations:
  iam.amazonaws.com/role: "kube2iam_prod-cluster/external-dns"
```

Role `kube2iam_prod-cluster/external-dns` potřebuje být vytvořena a mít oprávněnní k vytváření a přenastavování záznamů v Route53. External-DNS je nasazen pomocí příkazu

```
helm install external-dns bitnami/external-dns
    --values=staging/external-dns/chart-values.yaml -n kube-system
```

3.1.9 Nastavení sítě

Diagram níže popisuje rozvržení sítě po plánovací aplikaci v cloudové platformě AWS. Základní oddělovací částí v AWS je VPC (*Virtual Private Cloud*)[39], umožňující vytvoření logicky izolované části AWS cloudu do které se nasazují AWS zdroje dle nastavení virtuální sítě.

Pro plánovací aplikaci využíváme jednoho AWS regionu, který se obvykle skládá ze tří dostupnostních zón (AZ, *Availability zones*)[40]. Každá dostupnostní zóna je jedno či více datacenter v jednom regionu, datacentra jsou navíc stavěna tak, aby každá z nich byla vystavěna jiným živelným rizikám. Tedy pro příklad, pokud je jedna dostupnostní zóna náchylná povodním, druhá AZ bude stavěna na kopci. Podobně to funguje s energetickým mixem či síťovým připojením.

Jak je zřejmé z diagramu, v našem případě vytváříme v každé AZ jeden soukromý a jeden veřejný subnet. Co subnety obsahují je zobrazeno na diagramu 3.4.

Obrázek 3.7: Rozvržení subnetů využitých pro aplikaci v jednom AWS regionu

3.1.10 Zabezpečení přístupu ke Kubernetes clusteru

V kapitole níže je diskutován princip nejméně privilegovaného, dle kterého se řídí administrátorské přístupy ke Kubernetes klastru. Dále je ukázán nástroj *kube2iam*, pomocí kterého jsou propojeny AWS IAM přístupy ke Kubernetes RBAC rolím.

3.1.10.1 Princip nejméně privilegovaného

Již v roce 1975 Saltzer a Schroeder poznamenali, že každý program a každý uživatel nějakého systému by měli operovat s využitím nejmenší množiny privilegií, která jsou nutná k vykonání nějakého úkonu[85]. Striktní následování principu nejméně privilegovaného předcházíme bezpečnostním problémům, jako *privilege creep* („privilegovaný slídil“, [87]). Bohužel porušení tohoto principu jsou poměrně častá[86], kde mezi hlavní důvody patří neznalost zabezpečitelnosti, otevřená privilegia pro budoucí použití, či nezabezpečené počáteční nastavení.

V AWS je využito principu nejméně privilegovaného při vytváření IAM uživatelů a rolí[93]. Tedy je nutné určit, jaké minimální oprávnění uživatel vyžaduje a vytvořit taková pravidla, aby byl schopen vykonat pouze takové úkony. Stejným principem budeme přiřazovat přístupy ke zdrojím v Kubernetes clusteru, kde využijeme RBAC[94].

Standardním procesem je přiřazení minimální množiny oprávnění a následně přidávat další potřebná. Tento přístup je bezpečnější, než přiřazení počátečních pravidel, která jsou moc volná a následně se je snažit utahovat.

Přiřazení Kubernetes RBAC rolí k IAM účtům

Ke Kubernetes clusteru je třeba administrativní přístup, ať už při hledání různých chyb, či pro monitoring systému. Pro zjednodušení administrace účtů je třeba mít *single source of truth* (*SSOT*), kde jsou účty uloženy. Jako úložiště je přirozeně v AWS ekosystému vybráno IAM – a účty v Kubernetes jsou na ně pomocí RBAC namapovány. Mapování uživatelů je detailně popsáno v [88].

Role jsou namapovány pomocí předpisu `ConfigMap` aws-auth v namespace `kube-system`, vypadající následovně

```
map_roles = [
  {
    rolearn  = "arn:aws:iam::{{AWS_ACCOUNT_NUMBER}}:role/tf_master"
    username = "tf_master:{{SessionName}}"
    groups   = ["system:masters"]
  },
  {
    rolearn  = "arn:aws:iam::{{AWS_ACCOUNT_NUMBER}}:role/admin"
    username = "admin:{{SessionName}}"
    groups   = ["admin"]
  },
  {
    rolearn  = "arn:aws:iam::{{AWS_ACCOUNT_NUMBER}}:role/dev"
    username = "dev:{{SessionName}}"
    groups   = ["dev"]
  },
```

```
  {
    rolearn  = "arn:aws:iam::{{AWS_ACCOUNT_NUMBER}}:role/ci_user"
    username = "ci_user:{{SessionName}}"
    groups   = ["ci_user"]
  },
]
```

Proměnnou `AWS_ACCOUNT_NUMBER` je třeba zaměnit za ID AWS účtu, ve kterém bude Kubernetes klastr nasazen.

Následně je nutné vytvořit předpis `Role` a `RoleBinding` pro role vytvořená pouze pro jeden namespace, resp. `ClusterRole` a `ClusterRoleBinding` pro role, které nejsou omezeny pouze na jeden namespace. Například role pro CI uživatele vypadá následovně

```
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
    namespace: application
    name: ci_user
rules:
- apiGroups: ["", "extensions", "apps", "autoscaling", "networking.k8s.io",
                "traefik.containo.us", "batch"]
    resources: ["deployments", "replicasets", "pods", "configmaps",
                "secrets", "serviceaccounts", "services",
                "horizontalpodautoscalers", "ingresses",
                "ingressroutes", "cronjobs"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

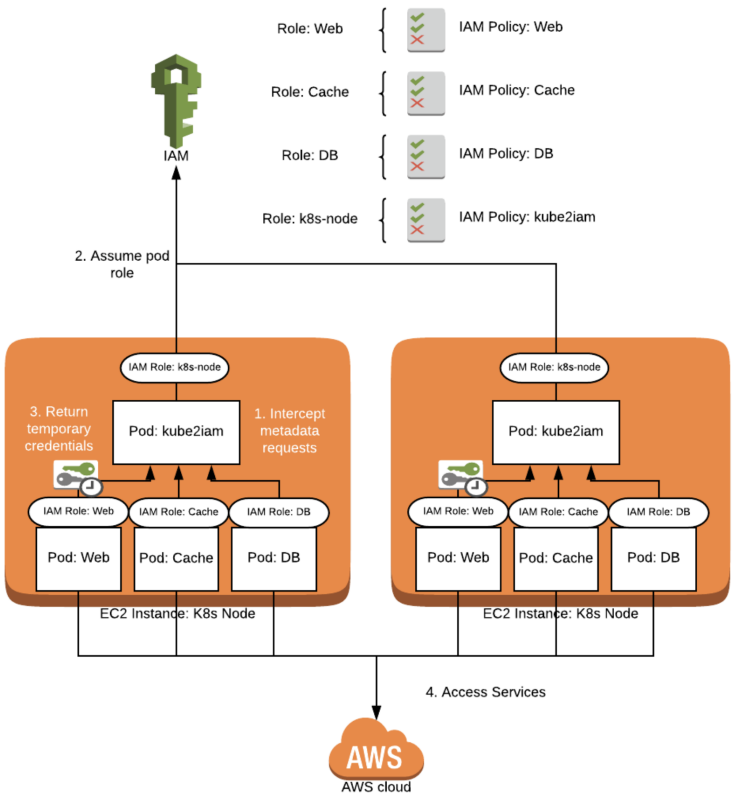
a jeho navázání

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
    name: ci_user
    namespace: application
subjects:
- kind: Group
    name: ci_user
    apiGroup: rbac.authorization.k8s.io
roleRef:
    kind: Role
    name: ci_user
```

```
apiGroup: rbac.authorization.k8s.io
```

Přiřazení IAM rolí k podům v Kubernetes

K přiřazení IAM rolí k podům v Kubernetes je využito open source nástroje *Kube2IAM* [92]. *Kube2IAM* byl první kandidát na vyřešení problému přiřazení IAM rolí ke Kubernetes zdrojům. Nástroj je nasazen na každý uzel Kubernetes jako **DaemonSet**. `DaemonSet` běží v privilegovaném bezpečnostním módu, takže může vytvářet `iptables` pravidla, kterými odchytává požadavky na *EC2 Metadata*, běžící na adrese `169.254.169.254`. Kube2IAM pody tedy na každém Kubernetes uzlu odposlouchávají požadavky na *IAM API*. Místo toho, aby autentizovali napřímo, *assumují* ("osvojují si") role, které jsou k podům přiřazeny pomocí anotací. Následovně odpoví s dočasnými přístupovými údaji z *assumované* role. Žádné změny nejsou nutné a aplikace může běžet tak, jak byla navrhnuta pro čistý Kubernetes.



Obrázek 3.8: Přiřazení IAM práv, komunikace mezi podem a IAM API. Diagram převzat z [99].

Existuje několik alternativ ke Kube2IAM. Jedním z nich je *Kiam* [91], který je silně inspirovaný právě Kube2IAM. Kiam jako projekt je ještě více bleeding-edge, a jako projekt je o něco méně aktivní. Kiam řeší škálovací problémy a problémy se zabezpečením, které byly obsaženy v Kube2IAM v době vytvoření Kiam. Řeší to následovně – rozděluje svoji činnost na serverovou část a agent část. Agent běží jako `DaemonSet` a zachytává komunikaci s EC2 metadatami, podobně jako u Kube2IAM. Rozdíl je ten, že klient nekomunikuje přímo s IAM, ale tuto práci přenechává na svoji server část. Blogový článek od vývojáře Kiam problematiku popisuje více do detailu. [89]

Další alternativou je přiřazení IAM rolí přímo na uzly Kubernetes workerů. Toto má jednu hlavní bezpečnostní nevýhodu – každý pod, který na daném uzlu beží, je schopen vykonávat úkony, které mu IAM role povoluje. Díku tomu tato možnost byla hned zavrhnuta.

Poslední časou využívanou variantou je využití federace autentizace pomocí OICD (*OpenID Connect*). [90] Této metody nebylo využito kvůli vendor lock-inu daného řešení.

3.2 Automatizace infrastruktury

Pro zvýšení důvěry ve správnost a kvalitu softwaru, Fowler et al. [17] doporučuje co nejvíce automatizovat opakované úkoly, jako například spouštění testů, vytváření kontejnerů, či nasazování. Dle jejich názoru by měl *CD (continuous deployment)* tvořit nasazení doslovně nudným. Také Newman [18] nazývá virtualizaci jako klíčovým faktorem k automatizaci infrastruktury, a to proto, že napomáhají k automatickému vytvoření a škálování virtuálních strojů pro vývojové či produkční prostředí.

3.2.1 Continuous Integration & Continuous Deployment

Podkapitolky níže popíšou koncepty Continuous integration, Continuous delivery (dohromady také zvané jako CI/CD), dále ukážou současnou nabídku takových služeb. Dále bude jedna ze služeb vybrána a její technologie bude diskutována.

3.2.1.1 Continuous integration

Cílem *Continuous integrace (CI)* [41] je propojení práce developera (jako například kompilace aplikace, sestavení kontejneru, spuštění testů) a kódu aplikace. Tato integrace by se měla dít s vyšší frekvencí, např. nekolikrát denně, či po každém commitu do repozitáře s kódem.

Částí CI procesu může být i kontrola kódu (*code review*) od kolegů developerů.

Výstupem CI procesu je připravená verze aplikace (např. tedy binární soubor nebo kontejner), který se nazývá artefakt (*artifact*).

Častá integrace a kvalitní testové pokrytí pomáhá s důvěrou v kód na straně jak developerů, tak i na straně serverových administrátorů.

3.2.1.2 Continuous delivery

Continuous delivery (CD) je v podstatě automatizace nasazení aplikací a řadí se za *Continuous integration*. CD tedy popisuje možností nasazení artefaktů z CI do různých aplikačních prostředí, jako je např. produkce.[42]

Každý tým má různé potřeby při stavění takových CD. Někde je třeba více testovacích (*develop*) prostředí, jiné týmy vyžadují mnoho *staging* (prostředí identické produkčnímu ale bez reálné zátěže) prostředí. Při CD vytvoření takového prostředí může být plně automatizováno, případně se může jednat o několik (potvrzujících) kliknutí.

3.2.2 Návrh CI/CD pro plánovací službu

Vzhledem na fakt, že infrastruktura je postavena na AWS, vybíral jsem z následujících CI/CD řešení

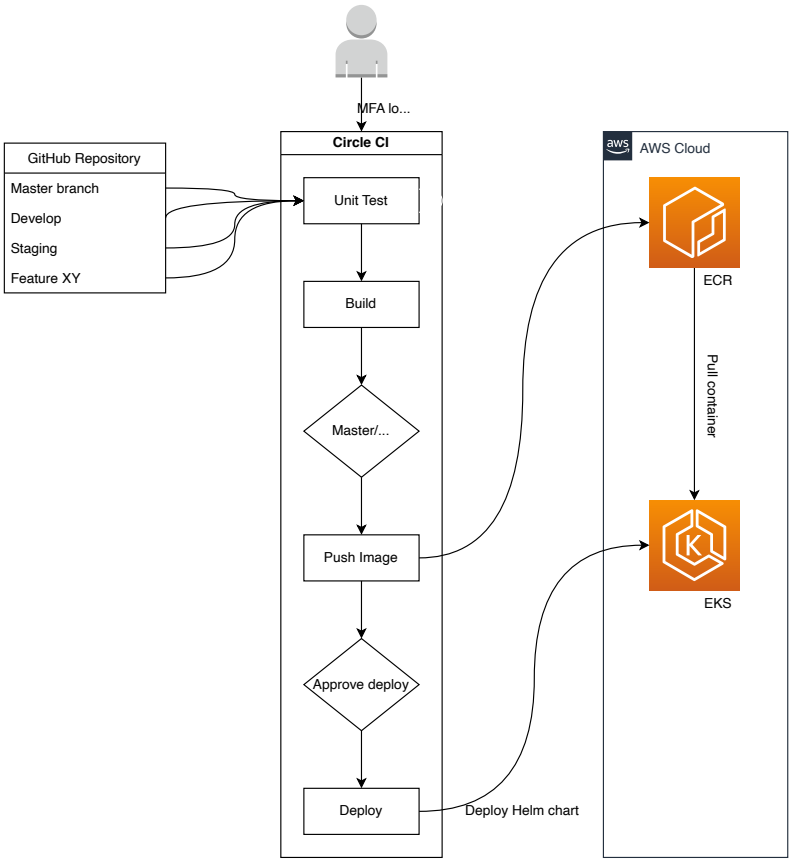
- Gitlab CI[151] je intuitivním řešením, plně integrovaným s Gitlab repozitáři. Software je nabízen jako SaaS (Software as a Service). Pro soukromé repozitáře zdarma poskytují 400 CI minut měsíčně, za 4 USD nabízí 2000 volných minut (měsíčně). Konfigurace je nastavována pomocí `.gitlab-ci.yaml` souboru v kořeni repozitáře.

- Circle CI [152]je také software nabízený jako SaaS. Nabízí zdarma 2500 minut týdně, kde při využití 1 vCPU a 2gb RAM vychází minuta na 5 kreditů. Za 15 USD měsíčně nabízí 25 000 kreditů. Pipeline je konfigurována pomocí `yaml` souboru.

- AWS nabízí službu CodePipeline [153], který nabízí jednu aktivní pipeline zdarma měsíčně. Službu je nutné propojit se službou CodeDeploy[155] na nasazení a CodeBuild[154] na build fáze. Tato nabídka je poměrně neintuitivní a nenabízí širokou rozšířitelnost.

Z porovnávaných řešení nejlépe vychází Gitlab CI a Circle CI. Díky tomu, že repozitář je uložen na službě GitHub, bylo vybráno jako CI řešení služba Circle CI. Diagram 3.9 ukazuje CI/CD proces pro plánovací službu z této práce.

Kód aplikace je uložen v git repozitáři na serveru GitHub. Po zapsání změn do repozitáře se spustí krok s unit testy a pokud proběhne úspěšně, z aplikace se začne vytvářet kontejner. V případě, že změny jsou v *master*, *staging*, či *develop* branchi, tak obraz je pushnut do kontejner repozitáře (ECR). CD proces poté čeká na vstup od uživatele, který svým schválením může aplikaci nasadit do respektivních prostředí.



Obrázek 3.9: Diagram v CI pipeline

3.2.3 Docker registry pro ukládání obrazů kontejnerů

Jak ukazuje diagram 3.9, obrazy kontejneru s plánovací aplikací je nutné uložit do repozitáře. Vzhledem na fakt, že infrastruktura je postavena v AWS cloudu, vychází 3 řešení

- Docker Registry [141] je řešení přímo od společnosti Docker. V základním plánu zdarma ale nenabízí soukromé repozitáře (ty jsou pouze v pro plánu za 5 USD [142]).
- Gitlab nabízí službu Container Registry [143], který má soukromé repozitáře zdarma až do velikosti 10GB. Po přesažení limitu se začnou mazat staré obrazy.
- AWS nabízí službu Elastic Container Registry (ECR)[144], který nabízí soukromé repozitáře zdarma až do velikosti 500MB. Nad limit se platí 0.10 USD za každý GB, plus za přenos dat, viz [145]. ECR také nabízí

funkci statického scanování Docker obrazů, ke kterému je využit nástroj Clair [147].

Z výše zmíněných možností bylo využito ECR, hlavně díky již využitému AWS ekosystému a také kvůli zmiňovanému již vestavěnému scanování obrazů.

3.3 Nasazení aplikace

Kapitola níže popisuje praktické nasazení aplikace do distribuovaného prostředí Kubernetes. Ukáže, které Kubernetes zdroje jsou potřebné a jak je možné proces automatizovat s nástrojem Helm.

3.3.1 Helm

Standardním způsobem definování zdrojů v Kubernetes clusteru je vytvoření konfiguračního souboru pro každý zdroj. V případě, kdy spouštíme aplikaci s různým nastavením (jako produkční a testovací prostředí, atp.), bychom museli jednotlivé konfigurační soubory duplikovat. Tím míříme k redundantnímu kódu, kterému se chceme vyhnout (pokud dodržujeme *DRY* [26] pravidlo).

K nasazení je využito nástroje Helm [43].

Helm pomáhá s vytvořením dynamických konfiguračních souborů s použitím proměnných. Helm používá *Go Template Engine*, kterým generuje Kubernetes zdroje s využitím předem definovaných proměnných. Právě kvůli tomu můžeme vytvořit pouze jeden *resource template* (šablonu Kubernetes předpisu), ze kterého se vygenerují lehce odlišné zdroje pro každé prostředí.

Využijeme Helmu verze 3, který všechny změny vypočítává na klientské části (na počítači u vývojáře), což je změna oproti verzi druhé, kde bylo třeba nasazovat *Tiller* (server část *Helmu v2*) do Kubernetes.

3.3.2 Kubernetes objekty

Jak bylo zmíněno v teoretické části práce, Kubernetes byl navrhnut jako *desired state model* a ten je možno definovat několika typy předpisů, mezi které patří například `YAML` a `JSON`. My využijeme `YAML` formátu. Tento předpis bude verzovaný a generovaný pomocí Helmu zmíněného výše. konfigurace by se dala také nazvat jako spustitelná dokumentace, ze které můžeme kdykoliv obnovit nastavení Kubernetes objektů.

Aplikace obsahuje tyto Kubernetes objekty (tj. vygenerované ze šablon z Helmu):

[26] DRY – Do not repeat yourself

Deployment

```
# Source: multimodal-planner/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: multimodal-planner-prod
    labels:
    helm.sh/chart: multimodal-planner-0.1.0
    app.kubernetes.io/name: multimodal-planner
    app.kubernetes.io/instance: multimodal-planner-prod
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
    selector:
    matchLabels:
        app.kubernetes.io/name: multimodal-planner
        app.kubernetes.io/instance: multimodal-planner-prod
    template:
    metadata:
        annotations:
        labels:
            app.kubernetes.io/name: multimodal-planner
            app.kubernetes.io/instance: multimodal-planner-prod
    spec:
        serviceAccountName: multimodal-planner-prod
        securityContext:
        {}
        containers:
        - name: "multimodal-planner"
            securityContext:
            {}
            image: "<account_id>.dkr.ecr.eu-central-1.amazonaws.com/
                multimodal-planner:production"
            imagePullPolicy: Always
            ports:
            - name: http
                containerPort: 80
                protocol: TCP
            livenessProbe:
            httpGet:
                path: /healthz
                port: http
            readinessProbe:
```

```
            httpGet:
                path: /readiness
                port: http
            resources:
            requests:
                cpu: 100m
                memory: 256Mi
---
```

Jak je vidět ve výstupu výše, je nutné, aby aplikace měla vytvořeny endpointy `/healthz` a `/readiness`, na které se bude Kubernetes pomocí *probe* připojovat a testovat, zda je `Pod` živý.

Service

```
# Source: multimodal-planner/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: multimodal-planner-prod
  labels:
    helm.sh/chart: multimodal-planner-0.1.0
    app.kubernetes.io/name: multimodal-planner
    app.kubernetes.io/instance: multimodal-planner-prod
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: multimodal-planner
    app.kubernetes.io/instance: multimodal-planner-prod
```

Předpis výše ukazuje vystavení služby `multimodal-planner-prod` na portu `80`.

Ingress

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
```

```
metadata:
    name: multimodal-planner-prod
    annotations:
        external-dns.alpha.kubernetes.io/hostname: "www.planner.net"
        external-dns.alpha.kubernetes.io/target: "planner.net"
spec:
    rules:
        - host: "planner.net"
```

Jak je definováno v kapitole s External-DNS, prázdný `Ingress` předpis je nutný kvůli vytvoření DNS záznamu. Tento konkrétní předpis vytvoří `CNAME www.planner.net -> planner.net`.

IngressRoute

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: multimodal-planner-prod-api-tls
  annotations:
    helm.sh/hook: "post-install"
  labels:

    helm.sh/chart: multimodal-planner-0.1.0
    app.kubernetes.io/name: multimodal-planner
    app.kubernetes.io/instance: multimodal-planner-prod
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  entryPoints:
    - websecure
  routes:
  - match: "Host(`planner.net`) || Host(`www.planner.net`)"
    kind: Rule
    services:
      - name: "multimodal-planner-prod"
        port: 80
```

Předpisem `IngressRoute` Traefik nakonfiguruje reverse proxy pravidlo pro všechny požadavky obsahující `planner.net` a `www.planner.net` v Host hlavičce a namíří na `multimodal-planner-prod` Kubernetes službu.

ServiceAccount

```
# Source: multimodal-planner/templates/serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: multimodal-planner-prod
  labels:
    helm.sh/chart: multimodal-planner-0.1.0
    app.kubernetes.io/name: multimodal-planner
    app.kubernetes.io/instance: multimodal-planner-prod
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
---
```

Předpis `ServiceAccount` vytvoří servisní účet, ve kterém poběží tento `Deployment`.

HorizontalPodAutoscaler

Předpis pro plánovač je popsaný v kapitole 3.4.1.1.

3.3.3 Rolling Update nasazení

Díky konfiguracím ukázaným výše a také Helmu je možné aplikaci nasazovat jako Rolling Update. Toho je docíleno pomocí příkazu

```
helm upgrade --install --atomic --namespace "application"
   "multimodal-planner-prod" "multimodal-planner-prod"
   --values "values/multimodal-planner-prod.yaml"
```

Díky tomu, že máme pomocí předpisu `HorizontalPodAutoscaler` nastaven minimální počet replik na 2 a maximální na 5, v případě, že aktuálně máme 2 repliky, Helm prvně přidá dvě nové repliky a pokud jsou funkční (tj. `/healthz` a `/readiness` endpointy vrací `200 OK`), odstraní dvě původní. V případě, že se něco pokazilo, parametr `--atomic` zajistí, že se aplikace vrátí do původního stavu. Toto celé proběhne bez výpadku.

Pokud by Kubernetes chyby na aplikaci nezachytil, je možné aplikaci vrátit do staré verze také manuálně, pomocí příkazu

```
helm rollback multimodal-planner-prod 1
```

kde `1` je číslo revize, která byla funkční.

3.4 Škálování aplikace

Jednou ze silných stránek Kubernetes, jakožto orchestrátora kontejnerů, leží ve schopnosti „managovat“ dynamická prostředí a reagovat na dynamické změny. Jedním z příkladů je nativní schopnost provést efektivní škálování zdrojů, ať už Kubernetes uzlů, nebo `Pod`ů. Každopádně Kubernetes nepodporuje pouze jeden škálovač, či jen jeden automaticky škálovací přístup. V kapitole níže popíši formy škálování, které Kubernetes nabízí.

Vertikální vs. Horizontální škálování

Horizontální škálováním je myšleno přidání dalších strojů/uzlů do poolu zdrojů (klastru). Takové škálování se také nazývá škálování do šířky (*scaling out*). Naopak škálování přidáním více výkonu (ať už CPU, RAM, atp.) na existujícím stroji se nazývá škálování vertikální (nebo také škálování do výšky, *scaling up*). Níže popíši rozdíly a využití jednotlivých škálování více do hloubky.

Vertikálního škálování je v Kubernetes možné dosáhnout pomocí nástroje **Vertical Pod Autoscaler**. VPA upravuje CPU a paměťové rezervace pro `Pod`y. Tato práce se mu nevěnuje, ale více je možné zjistit např. zde[156].

3.4.1 Horizontální škálování

Kubernetes nabízí dvě hlavní škálování do šířky, a to škálování `Pod`ů a škálování uzlů Kubernetes. Oba způsoby budou popsány jak fungují a jak jsou využity.

3.4.1.1 Horizontální škálování Podů

Podkapitola níže popíše horizontální škálování `Pod`ů, ke kterému je využito nástroje *Horizontal Pod Autoscaler*. HPA je ideální pro škálování bezstavových aplikací (takových, jako je plánovač popsaný v první části práce), každopádně může být využit i pro škálování aplikací stavových (v Kubernetes tzv. `StatefulSet`ů). Využití HPA v kombinaci se škálováním klusteru může pomoci ke snížení útrat za infrastrukturu tím, že na vytížení zdrojů můžeme rychle zareagovat snížením, či zvýšením uzlů dle potřeby.

Horizontal Pod Autoscaler

Horizontal Pod Autoscaler byl prvně představen v Kubernetes v1.1[100]. První verze HPA škálovala aplikaci dle zjištěných hodnot CPU utilizace a využití paměti. Od verze Kubernetes 1.6 byla přidána možnost využití Custom Metrics API, která nabízí HPA využití vlastních metrik pomocí REST API. Od verze Kubernetes 1.7 agregační vrstva API serveru umožnuje applikacím 3. stran rozšířit Kubernetes API pomocí registrace sebe samotných jako API Add-ony. Všimněme si, že tento koncept API agregace je podobný `Custom ResourceDefinition`, ale nabízí flexibilnější možnost implementace.

U aplikace s konfigurovaným HPA nástroj monitoruje Pody aplikace, u kterých se snaží zjistit, zda potřebují zvýšit/snížit počet replik Podů.

Dle [100], Horizontal Pod Autoscaler (HPA) v standardní konfiguraci dynamicky upravuje počet replik Podů v Deploymentu dle zjištěných hodnot CPU utilizace.

Níže přikládám předpis plánovací aplikace pro využití HPA.

```
# Source: multimodal-planner/templates/hpa.yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: multimodal-planner-prod-multimodal-planner
  labels:
    helm.sh/chart: multimodal-planner-0.1.0
    app.kubernetes.io/name: multimodal-planner
    app.kubernetes.io/instance: multimodal-planner-prod
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: multimodal-planner-prod-multimodal-planner
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 80
---
```

V předpisu výše si můžeme všimnout konfigurace minReplicas a maxReplicas. Díky nim můžeme nastavit, pod (a nad) kolik replik se nikdy Deployment nesmí dostat.

HPA škálovací algoritmus

Dle [100], implementace automaticky škálovacího algoritmu v HPA funguje následovně.

- Implementováno jako kontrolní cyklus - standadně 30s, ale nastavení je konfigurovatelné.
- Periodicky se dotazuje Podů a sbírá jejich CPU utilizaci.

- Porovná aritmetický průměr CPU utilizace daného `Podu` s nakonfigurovaným cílem.

- Přidá, či odebere repliky Podu (pokud je třeba dosáhnout konfigurovaného cíle), dle podmínek:

  - $MinReplicas \leq Replicas \leq MaxReplicas$
  - $CPUUtilization(C)$ = poslední CPU utilizace `Podu` (průměr za poslední minutu)
  - CPU vyžádané `Podem` (`spec.containers[].resources.requests.cpu`)
  - $TargetNumOfPods = ceil(\frac{sum(CurrentPodsCPUUtilization)}{TargetCPUUtilizationPercentage(T)})$

$$TargetNoPods = \lceil (\sum_{n}^{n=1} C_n)/T \rceil$$

- Škálování nahoru (přidání replik) nastává pouze ve chvíli, kdy neproběhla úprava replik v posledních 3 minutách. Toto je zdůvodu dočasné fluktuace CPU při startu/ukončení `Podů`.

- Škálování dolu (odebrání replik) nastává pouze ve chvíli, kdy neproběhla úprava replik v posledních 5 minutách. Toto je zdůvodu dočasné fluktuace CPU při startu/ukončení `Podů`.

- Jakékoli škálování proběhne pouze ve chvíli, kdy:

$$\frac{avg(CurrentPodsConsumption)}{TargetCPUUtilizationPercentage}$$

  spadne `Pod` 0.9 nebo se zvýší nad 1.1 (10% tolerance).

[100] říká, že výše zmíněný algoritmus má dvě hlavní výhody

- HPA funguje „konzervativně“ – tedy, že navýšení počtu `Podů` v případě vysoké utilizace je rychlé, ale se snížením počtu `Podů` HPA tolik nespěchá.

- HPA se snaží vyhnout tzv. *trashování*, tedy předchází rychlé exekuci rozhodnutí, díky kterým by mohl vzniknout vzájemný konflikt (v situacích, kdy utilizace není stabilní).

Požadavky pro nastavení HPA

Standardně HPA běží jako součást `kube-controller-manager` démonu. Může managovat pouze ty `Pody`, které byly vytvořeny *replication controllerem*, tedy `Deploymenty`, `ReplicaSety`, `StatefulSety`.

HPA vyžaduje zdroj metrik, které bude využívat. Pro škálování dle CPU utilizace závisí na `metrics-serveru`. Škálování založené na vlastním, či externím zdroji metrik vyžaduje nasazení služby, který implementuje `custom.metrics.k8s.io` nebo `external.metrics.k8s.io` API.

Pro *workloady* škálované pomocí standardní CPU metriky (v našem případě) musí mít `Pody` nastaveny CPU *resource limity*, které jsou nastaveny v `Deployment` specifikaci. Objekt s kontejnerem v cestě `spec.template.spec.containers` by tedy měl obsahovat

```
resources:
requests:
  cpu: 100m
  memory: 256Mi
```

3.4.1.2 Horizontální škálování uzlů clusteru

Zatímco HPA škáluje počet `Podů` běžících v klastru, Cluster Autoscaler může změnit počet uzlů v klastru.

Dynamicky škálované nody, které se snaží přesně pokrýt aktuální utilizaci klastru mohou šetřit cenu za infrastrukturu.

Cluster Autoscaler

Cluster Autoscaler[157] prochází v cyklu dvěmi hlavními úkoly – sledováním, zda existují neschedulovatelné `Pody` a vypočítáváním, zda může konsolidovat již vytvořené Pody na menší počet Kubernetes uzlů.

Autoscaler sleduje kastr, zda existují `Pody`, které není možné přiřadit k nějakému z existujících uzlů, protože klastr neobsahuje dostatěčné volné zdroje, nebo pokud *node affinity* pravidla (nebo *taint tolerace*) daného `Podu` neumožňují nikam ho přiřadit. Pokud má klastr `Pody` nepřiřaditelné (*unschedulable*) k žádnému uzlu, Autoscaler zkontroluje svoje managované node grupy a rozhodne se, do jaké grupy přidat uzel, kam by Kubernetes přiřadil `Pod`.

Autoscaler také skenuje uzly v node grupách, které managuje. Pokud uzel obsahuje `Pod`, který může být přesunut na nějaký z jiných dostupných uzlů v kastru, Autoscaler vytvoří *evict* událost a následně je odstraní z uzlu. Při rozhodování, zda `Pod` může být přesunut, autoscaler bere v potaz `Pod` prioritu a `PodDisruptionBudgets`.

Cluster Autoscaler je nasazen z oficiálního Helm chartu[158] s konfigurací

```
cloudProvider: aws
```

```
awsRegion: eu-central-1

rbac:
  create: true

autoDiscovery:
  clusterName: prod-cluster
  enabled: true

podAnnotations:
  iam.amazonaws.com/role: "kube2iam_prod-cluster/cluster-autoscaler"

extraArgs:
  v: 2
  stderrthreshold: info
  logtostderr: true
  scale-down-utilization-threshold: 0.3
```

Role `kube2iam_prod-cluster/cluster-autoscaler` potřebuje potřebná oprávnění[159] k vytvoření a odstranění EC2 instancí, plus další oprávnění. Roli je možné vytvořit v AWS konzoli, pomocí cli nebo Terraformu.

Závěr

Plánovač

V této práci jsem navrhl plánovač tras kombinující více způsobů dopravy (tzv. *multimodální plánovač*). Uvedl jsem nutné termíny a definice potřebné k pochopení problému.

V práci bylo porovnáno několik již existujících řešení na trhu. Jedním z nich je český AnyRoute, oproti kterému můj plánovač nabízí širokou rozšířitelnost a také větší množinu poskytovatelů sdílených prostředků.

Plánovač navržený v této práci podporuje několik různých kombinací prostředků – singlemodální s hromadnou dopravou, singlemodální s prostředkem sdílených mobilit a multimodální v kombinaci hromadné dopravy a prostředku sdílených mobilit (a to pro problémy první míle a poslední míle). Navržený plánovač využívá již vytvořené plánovače s otevřeným kódem, jmenovitě *Open Trip Planner* pro plánování cest hromadnou dopravou (díky své nativní integraci s *GTFS* daty) a *Open Source Routing Machine* pro plánování cest chůzí, na kole a autem (díky vysoké rozšířitelnosti *lua* profilami). Open-source plánovače jsou využity z důvodu omezení komplexnosti této práce a také díky vysoké spolehlivosti jejich výsledků. Jejich algoritmy jsou také velmi optimalizovány a vylepšeny různými zrychleními.

Algoritmus multimodálního plánovače z této práce byl inspirován *Transit Node Routing* algoritmem, který je popsán v teoretické části. Jsou vybírány přestupní stanice (tedy stanice, přes které prochází většina cest) a přes ně jsou směrovány cesty. Celý algoritmus je v práci popsán, práce také obsahuje grafy, nad kterým je Dijskrovým algoritmem nalezena nejkratší cesta. U každého algoritmu je také zobrazena pro ukázku konečná výsledná cesta.

Konečně je plánovač kontejnerizován, aby byl připraven pro distribuovanou infrastrukturu, popsanou v druhé části práce.

Architektura plánovače

V drůhé části práce jsem k plánovači navrhl architekturu. Prvně byly popsány koncepty jako mikroslužby, virtualizace, kontejnery, orchestrace kontejnerů a jejich reálné implementace. Představil jsem koncept softwarově definované architektury (*Infrastructure as Code*), poukázal na její výhody a nevýhody.

Pro orchestraci kontejnerů bylo vybráno systému Kubernetes, jehož funkce je popsána. Dále je diskutováno porovnání vlastního nasazení Kubernetes a tzv. managovaného Kubernetes. Dle kritérií popsaných v práci (potřeba malé potřeby obsluhy jak při výpadcích, tak při aktualizacích klastru, či OS na uzlech) byl vybraný managovaný Kubernetes, u kterých jsou představeny nabízené služby a následně jsou porovnány. Z porovnávaných služeb (EKS, GKE, AKS) byl vybrán EKS, a to z důvodu vysoké stability systému a také mé familiarity s AWS ekosystémem.

Architektura je představena ve dvou částech. První obsahuje služby využité v AWS cloudu (mimo Kubernetes klastr). Mezi služby patří administrátorské připojení do prostředí (*bastion host*), load balancer a Internet Gateway pro příchozí požadavky a VPC NAT Gateway pro odchozí komunikaci do internetu. Druhá část představuje služby běžící v Kubernetes klastru, rozdělené do jmenných prostorů.

Pro komunikaci mezi službami a klientskou aplikací je představen vzor *API gateway*. Jako brána je vybrána aplikace Traefik, která spolu s ostatními službami automaticky vytváří load balancery a automaticky vytváří DNS záznamy.

Konečně je představeno škálování v Kubernetes (horizontální škálování uzlů klastru pomocí aplikace Cluster Autoscaler i horizontální škálování replik aplikace pomocí Horizontal Pod Autoscaler).

Vylepšení na plánovací aplikaci

Dále popíši možná vylepšení, která plánovač v nejbližší době čekají. Jsou rozdělena na vylepšení na aplikaci a na vylepšení na infrastruktuře.

Ukládání tras mezi přestupními stanicemi do mezipaměti

Jak definuje algoritmus *Transit Node Routing* v kapitole 2.3.3.1, trasy mezi přestupními stanicemi by měly být ukládány do mezipaměti pro zrychlení následného vyhledávání. Toto v současné době není implementováno.

Tato akce by měla být poměrně elementární, díky tomu, že sdílená paměť redis již v klastru běží a jsou v ní ukládány lokace a metadata dopravních prostředků a také zóny, kde jsou provozovány prostředky od různých poskytovatelů vozidel.

Jde tedy pouze o implementaci uložení tras mezi přestupními stanicemi s určitou expirací dat. V případě, že data v cache nejsou, či jsou expirovaná, plánovač se dané služby (ať už OTP nebo OSRM) na etapu trasy znovu zeptá.

Návrh nejkratších cest s ohledem na dopravní zácpy

V současné době plánovač neobsahuje informace o tom, zda daná cesta je vytížena nebo ne. Backend OSRM podporuje vložení informací o dopravních zácpách [67] ve formátu `from_osm_id,to_osm_id,edge_speed_in_km_h` (formát `CSV`). Data se do OSRM vkládají v *preprocessing* fázi, tedy po vložení dat musí být kontejner znovu vytvořen.

Taková data jsou dostupná například od Googlu[106], nebo TomTomu[107] ve formátu strojově zpracovatelných dat, každopádně přístup k nim je vždy placený.

Přidání manipulační doby u nástupu/výstupu z prostředku

Aktuálně není počítáno s manipulační dobou při nástupu nebo výstupu ze sdíleného prostředku. Jsou prostředky (např. automobily), u kterých je třeba při nástupu zkontrolovat nastavit mnoho věcí (nastavení zrcátek, sedačky, navigace, atp.), při výstupu vozidlo nafotit a odhlásit se. Toto je třeba započítat do celkového času a potom s tímto celkovým časem při hledání nejkratší trasy počítat.

Vylepšení na architektuře

Service mesh

V současné době aplikace v Kubernetes klastru komunikují mezi sebou napřímo (přes svůj Kubernetes `Service` endpoint). To přináší několik nevýhod – služby mezi sebou komunikují na HTTP, takže data jsou na lokální síti nezašifrovaná, nad komunikací nemáme vizibilitu, atp.

Jedním z řešení je service mesh, která přidává abstrakční vrstvu pro komunikaci mezi službami. Většinou je service mesh implementována pomocí *sidecar proxy*, tak jako u Istio [111].

OIDC

V současné době je celé API zabezpečeno pouze *basicauth* autentizací a autorizace chybí. V dalších krocích tedy bude výběr jednoho z OIDC poskytovatelů, ať už AWS Cognito[109] nebo Okta[108], kteří nabízí OIDC jako „managovanou“ službu. Toto poskytne větší granularitu v přístupu k jednotlivým endpointům API (tedy například určitý zákazník bude mít přistup pouze k určitým endpointům). Toto je možné implementovat do API brány Traefik pomocí modulu ForwardAuth[110].

Literatura

[1] Základy Teorie Grafů pro (nejen) informatiky, Doc. RNDr. Petr Hliněný, Ph.D., Fakulta Informatiky Masarykova Univerzita, 2010

[2] Multi-Modal Route Planning, Thomas Pajor, Institut fur Theoretische Informatik, Universitat Karlsruhe (TH), 2009

[3] Route Planning in Transportation Networks, Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, Renato F. Werneck, Microsoft Research, 2014

[4] Modelling of preferences in multimodal routing algorithms, Santtu Saijets, Master's Thesis, Aalto University, 2018

[5] Fast Routing in Road Networks with Transit Nodes, Bast, H.; Funke, S.; Sanders, P.; Schultes, D., Science. 2015

[6] Time-dependent networks as models to achieve fast exact time-table queries, Gerth Stølting Brodal and Riko Jacob, Electronic Notes in Theoretical Computer Science, 2004

[7] Virtualization and containerization of application infrastructure: A comparison, M. J. Scheepers, , in 21st Twente Student Conference on IT, vol. 1, 2014

[8] Monitoring and managing iot applications in smart cities using kubernetes, S. Muralidharan, G. Song, and H. Ko, CLOUD COMPUTING 2019, 2019

[9] "Kubernetes as an availability manager for microservice applications, L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, 2019

[10] Microservices architecture enables devops: Migration to a cloud-native architecture, A. Balalaie, A. Heydarnoori, and P. Jamshidi, Ieee Software, 2016

[11] Leveraging microservices architecture by using docker technology, D. Jaramillo, D. V. Nguyen, and R. Smart, IEEE, 2016

[12] An introduction to docker for reproducible research, C. Boettiger, ACM SIGOPS Operating Systems Review, 2015

[13] Migrating to cloud-native architectures using microservices: an experience report, A. Balalaie, A. Heydarnoori, P. Jamshidi, in European Conference on Service-Oriented and Cloud Computing, Springer, 2015

[14] Elastisys, Setting up highly available kubernetes clusters, Online, Prosinec 2020. Dostupné z https://elastisys.com/wp-content/uploads/2018/01/kubernetes-ha-setup.pdf

[15] Abusing privileged and unprivileged linux containers, Jesse Hertz, Whitepaper, NCC Group, 2016

[16] Understanding and hardening linux containers, Aaron Grattafiori, Whitepaper, NCC Group, 2016

[17] Microservices, Martin Fowler and James Lewis, https://martinfowler.com/articles/microservices.html, 2014

[18] Building microservices: designing fine-grained systems, Sam Newman, O'Reilly Media, Inc., 2015

[19] Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, Sam Newman, O'Reilly Media, Inc., 2019

[20] Service Discovery in a Microservices Architecture - NGINX, Online, Prosinec 2020. Dostupné z https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

[21] Why you should choose the microservices architecture? - PRETIUS, Online, Prosinec 2020. Dostupné z https://pretius.com/why-you-should-choose-the-microservices-architecture/

[22] A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project, Graziano, Charles. "". Prosinec 2020

[23] AWS Nitro System, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/ec2/nitro/

[24] Architectural Principles for Virtual Computer Systems, GOLDBERG, Robert P, www.dtic.mil. Harvard University, 1973

[25] The Architecture of VMware ESXi, Online, Prosinec 2020. Dostupné z https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/ESXi_architecture.pdf

[26] Measuring Docker Performance: What a mess!!!,Vanessa Perciballi, Emiliano Casalicchio, Online, Prosinec 2020. Dostupné z http://www.diva-portal.org/smash/get/diva2:1107544/FULLTEXT01.pdf

[27] Introduction to Container Security,Understanding the isolation properties of Docker, Online, Prosinec 2020. Dostupné z https://www.docker.com/sites/default/files/WP_IntrotoContainerSecurity_08.19.2016.pdf

[28] Cgroups, containers and HTCondor, oh my, Online, Prosinec 2020. Dostupné z https://indico.cern.ch/event/733513/contributions/3118608/attachments/1711984/2760942/gthain-containers.pdf

[29] Docker: The Fundamentals, Gianluca Arbezzano, Online, Prosinec 2020. Dostupné z https://gianarb.it/downloads/the-fundamental.pdf

[30] Docker overview – Docker Documentation, Online, Prosinec 2020. Dostupné z https://docs.docker.com/get-started/overview/

[31] Understanding Docker container escapes – Trail of Bits Blog, Online, Prosinec 2020. Dostupné z https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/

[32] Introducing Container Runtime Interface (CRI) in Kubernetes – Kubernetes, Online, Prosinec 2020. Dostupné z https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/

[33] Posts – Docker Saigon, Online, Prosinec 2020. Dostupné z http://docker-saigon.github.io/post/Docker-Internals/

[34] Container Orchestration Comparison, Online, Prosinec 2020. Dostupné z https://awesomeopensource.com/project/GuillaumeRochat/container-orchestration-comparison

[35] Swarm mode key concepts Docker Documentation, Online, Prosinec 2020. Dostupné z https://docs.docker.com/engine/swarm/key-concepts/

[36] Apache Mesos - Documentation Home, Online, Prosinec 2020. Dostupné z http://mesos.apache.org/documentation/latest/

[37] Kubernetes Documentation, Online, Prosinec 2020. Dostupné z https://kubernetes.io/docs/home/

[38] Auto-scaling an optimisation algorithm using Docker and Kubernetes on the NeCTAR Research Cloud, San Kho Lin, UNIVERSITY OF MELBOURNE, 2018

[39] What is Amazon VPC? - Amazon Virtual Private Cloud, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

[40] Regions and Zones - Amazon Elastic Compute Cloud, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html#concepts-availability-zones

[41] Fowler, Martin. Continuous Integration, Martin Fowler, Prosinec 2020. Dostupné z https://martinfowler.com/articles/continuousIntegration.html

[42] What is Continuous Delivery? - Continuous Delivery, Humble, Jez., Online, Prosinec 2020. Dostupné z https://continuousdelivery.com/principles/

[43] Helm – Docs, Online, Prosinec 2020. Dostupné z https://helm.sh/docs/

[44] Infrastructure as Code: Managing Servers in the Cloud, Kief Morris, O'Reilly Media, 2016

[45] GitHub - dmacvicar/terraform-provider-libvirt: Terraform provider to provision infrastructure with Linux KVM using libvirt, Online, Prosinec 2020. Dostupné z https://github.com/dmacvicar/terraform-provider-libvirt

[46] GTFS Static Overview Static Transit, Google Developers. Prosinec 2020. Dostupné z https://developers.google.com/transit/gtfs

[47] OpenTripPlanner, Online, Prosinec 2020. Dostupné z http://dev.opentripplanner.org/apidoc/1.0.0/index.html

[48] Kubernetes: up and running: dive into the future of infrastructure, Hightower, Kelsey, Brendan Burns, and Joe Beda. . 1 ed. USA: O'Reilly Media, Inc, 2017

[49] An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In: International Conference on Architectural Support for Programming Languages and Operating Systems, Gan, Yu, Yanqi Zhang, and Dailun Cheng et al., ASPLOS. 2019

[50] OpenTripPlanner 2, Online, Prosinec 2020. Dostupné z http://docs.opentripplanner.org/en/dev-2.x/

[51] Travel your first and last mile with Google Maps, Online, Prosinec 2020. Dostupné z https://blog.google/products/maps/travel-your-first-and-last-mile-google-maps/

[52] Pricing Plans Google Maps Platform Google Cloud, Online, Prosinec 2020. Dostupné z https://cloud.google.com/maps-platform/pricing

[53] Umotional s.r.o., Online, Prosinec 2020. Dostupné z https://www.startupjobs.cz/startup/umotional-s-r-o

[54] AnyRoute: Plánovač tras nové generace pro veřejnou dopravu a mobilitu jako služba - URBIS SMART CITY FAIR - Veletrhy Brno, Online, Prosinec 2020. Dostupné z https://www.bvv.cz/urbis/zlaty-urbis/2019/prihlasene-exponaty/07-anyroute-planovac-tras-nove-generace-pro-verejn/

[55] AnyRoute: Next-Generation Routing Engine for Combined Mobility and MaaS 8211; Umotional, Online, Prosinec 2020. Dostupné z https://umotional.com/cs/anyroute/

[56] We envision cities without private cars – Trafi, Online, Prosinec 2020. Dostupné z https://www.trafi.com/company/

[57] Jelbi – Berlins Öffentliche und Sharing-Angebote in einer App., Online, Prosinec 2020. Dostupné z https://www.jelbi.de/

[58] Urbane Mobilität aus einer App – yumuv, Online, Prosinec 2020. Dostupné z https://yumuv.ch/en/about-us

[59] PBF Format - OpenStreetMap Wiki, Online, Prosinec 2020. Dostupné z https://wiki.openstreetmap.org/wiki/PBF_Format

[60] OSM File Formats Manual - osmcode, Online, Prosinec 2020. Dostupné z https://osmcode.org/file-formats-manual/

[61] Planet.osm - OpenStreetMap Wiki, Online, Prosinec 2020. Dostupné z https://wiki.openstreetmap.org/wiki/Planet.osm

[62] Index of /extracts, Online, Prosinec 2020. Dostupné z https://osm.fit.vutbr.cz/extracts/

[63] Preparing OSM Data - OpenTripPlanner 2, Online, Prosinec 2020. Dostupné z https://docs.opentripplanner.org/en/latest/Preparing-OSM/

[64] PlannerResource, Online, Prosinec 2020. Dostupné z http://dev.opentripplanner.org/apidoc/0.15.0/resource_PlannerResource.html

[65] Open Source Routing Machine - OpenStreetMap Wiki, Online, Prosinec 2020. Dostupné z https://wiki.openstreetmap.org/wiki/Open_Source_Routing_Machine

[66] osrm-backend/LICENSE.TXT at master - Project-OSRM/osrm-backend - GitHub, Online, Prosinec 2020. Dostupné z https://github.com/Project-OSRM/osrm-backend/blob/master/LICENSE.TXT

[67] Traffic - Project-OSRM/osrm-backend - GitHub, Online, Prosinec 2020. Dostupné z https://github.com/Project-OSRM/osrm-backend/wiki/Traffic

[68] FastAPI, Online, Prosinec 2020. Dostupné z https://fastapi.tiangolo.com/

[69] Datové sady - Opendata Praha, Online, Prosinec 2020. Dostupné z https://opendata.praha.eu/dataset?organization=dpp&res_format=GTFS

[70] LGPL 3.0, Online, Prosinec 2020. Dostupné z http://www.gnu.org/licenses/lgpl-3.0.en.html

[71] Why you shouldn't use the Lesser GPL for your next library, Online, Prosinec 2020. Dostupné z http://www.gnu.org/licenses/why-not-lgpl.html

[72] Project OSRM, Online, Prosinec 2020. Dostupné z http://project-osrm.org/

[73] Cattle vs Pets - DevOps Explained, Online, Prosinec 2020. Dostupné z https://www.hava.io/blog/cattle-vs-pets-devops-explained

[74] DigitalOcean, Online, Prosinec 2020. Dostupné z https://www.digitalocean.com/docs/kubernetes/

[75] Cluster architecture Kubernetes Engine Documentation Google Cloud, Online, Prosinec 2020. Dostupné z https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-architecture

[76] Amazon EKS clusters - Amazon EKS, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/eks/latest/userguide/clusters.html

[77] H3: Uber's Hexagonal Hierarchical Spatial Index, Online, Prosinec 2020. Dostupné z https://eng.uber.com/h3/

[78] Geodesic Discrete Global Grid Systems, Kevin Sahr, Denis White, and A. Jon Kimerling, Online, Prosinec 2020. Dostupné z http://webpages.sou.edu/~sahrk/sqspc/pubs/gdggs03.pdf

[79] The Shapely User Manual, Shapely 1.7.1 documentation, Online, Prosinec 2020. Dostupné z https://shapely.readthedocs.io/en/stable/manual.html

[80] The Shapely User Manual, Shapely 1.7.1 documentation, Online, Prosinec 2020. Dostupné z https://shapely.readthedocs.io/en/stable/manual.html#shapely.ops.nearest_points

[81] Měření vzdáleností a plochy pomocí gps, Diplomová práce, Bc. Jakub Konecký, 2009

[82] OSRM API Documentation, Online, Prosinec 2020. Dostupné z http://project-osrm.org/docs/v5.5.1/api/#table-service

[83] OSRM API Documentation, Online, Prosinec 2020. Dostupné z http://project-osrm.org/docs/v5.5.1/api/#route-service

[84] Algoritmy pro hledání nejkratších cest v neorientovaných grafech, Bakalářská práce Richard Benkovský, Masarykova univerzita Fakulta informatiky, 2007

[85] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems, 1975

[86] Gary McGraw and John Viega. Software security principles, Part 3: Controlling access - Least privilege and compartmentalization, Online, Prosinec 2020. Dostupné z http://www.ibm.com/developerworks/library/se-priv/index.html

[87] Security Threats: A Guide for Small and Medium Businessesm Whitepaper, GFI, 2009

[88] Managing users or IAM roles for your cluster - Amazon EKS, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/eks/latest/userguide/add-user-role.html

[89] Kiam: Iterating for Security and Reliability, Online, Prosinec 2020. Dostupné z https://medium.com/@pingles/kiam-iterating-for-security-and-reliability-5e793ab93ec3

[90] Technical overview - Amazon EKS, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-technical-overview.html

[91] GitHub - uswitch/kiam: Integrate AWS IAM with Kubernetes, Online, Prosinec 2020. Dostupné z https://github.com/uswitch/kiam

[92] GitHub - jtblin/kube2iam: kube2iam provides different AWS IAM roles for pods running on Kubernetes, Online, Prosinec 2020. Dostupné z https://github.com/jtblin/kube2iam

[93] What is IAM? - AWS Identity and Access Management, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

[94] Using RBAC Authorization – Kubernetes, Online, Prosinec 2020. Dostupné z https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[95] Using Bicycles for the First and Last Mile of a Commute, Mineta Transportation Institute, 2009

[96] Bike-sharing: History, Impacts, Models of Provision, and Future, DeMaio, Paul, Journal of Public Transportation, 2009

[97] Bikesharing in Europe, the Americas, and Asia: Past, Present, and Future, Shaheen, Susan; Guzman, S., and H. Zhang, Transportation Research Record: Journal of the Transportation Research Board, 2010

[98] Folium , Folium 0.11.0 documentation, Online, Prosinec 2020. Dostupné z https://python-visualization.github.io/folium/

[99] IAM Access in Kubernetes: Kube2iam vs. Kiam, Online, Prosinec 2020. Dostupné z https://www.bluematador.com/blog/iam-access-in-kubernetes-kube2iam-vs-kiam

[100] Kubernetes Developers. Kubernetes Documentation. Tech. rep. v1., Online, Prosinec 2020. Dostupné z https://kubernetes.io/docs/

[101] Microservices architecture on Azure Kubernetes Service (AKS) - Azure Architecture Center — Microsoft Docs, Online, Prosinec 2020. Dostupné z https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/containers/aks-microservices/aks-microservices

[102] GKE overview, Kubernetes Engine Documentation, Google Cloud, Online, Prosinec 2020. Dostupné z https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview

[103] Amazon EKS Architecture - Quick Start, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/quickstart/architecture/amazon-eks/

[104] EKS vs GKE vs AKS - Evaluating Kubernetes in the Cloud — StackRox, Online, Prosinec 2020. Dostupné z https://www.stackrox.com/post/2020/10/eks-vs-gke-vs-aks/

[105] From monolith to microservices, Alex Giamas, zalando's journey. InfoQ, 2016

[106] Traffic, Transit, and Bicycling Layers – Maps JavaScript API, Google, Online, Prosinec 2020. Dostupné z https://developers.google.com/maps/documentation/javascript/trafficlayer

[107] TomTom Traffic – TomTom, Online, Prosinec 2020. Dostupné z https://www.tomtom.com/cs_cz/drive/tomtom-traffic/

[108] Okta – The Identity Standard, Online, Prosinec 2020. Dostupné z https://www.okta.com/

[109] Features – Amazon Cognito – Amazon Web Services (AWS), Online, Prosinec 2020. Dostupné z https://aws.amazon.com/cognito/details/

[110] ForwardAuth - Traefik, Online, Prosinec 2020. Dostupné z https://doc.traefik.io/traefik/middlewares/forwardauth/

[111] Istio, Online, Prosinec 2020. Dostupné z https://istio.io/

[112] Kubernetes Ingress Controller Overview, Online, Prosinec 2020. Dostupné z https://medium.com/swlh/kubernetes-ingress-controller-overview-81abbaca19ec

[113] CNCF SURVEY, Deployments are getting larger as cloud native adoption becomes mainstream, Online, Prosinec 2020. Dostupné z https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf

[114] Overview - Traefik, Online, Prosinec 2020. Dostupné z https://doc.traefik.io/traefik/middlewares/overview/

[115] Plugin to Traefik: Create and Publish Your Own Middleware - YouTube, Online, Prosinec 2020. Dostupné z https://www.youtube.com/watch?v=hCYT1aKJORY

[116] GitHub - traefik/traefik-helm-chart: Traefik v2 helm chart, Online, Prosinec 2020. Dostupné z https://github.com/traefik/traefik-helm-chart#installing

[117] Annotations - AWS LoadBalancer Controller, Online, Prosinec 2020. Dostupné z https://kubernetes-sigs.github.io/aws-load-balancer-controller/guide/ingress/annotations/

[118] eks-charts/stable/aws-load-balancer-controller at master – aws/eks-charts – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/aws/eks-charts/tree/master/stable/aws-load-balancer-controller

[119] , Online, Prosinec 2020. Dostupné z https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingress-controller/master/docs/examples/iam-policy.json

[120] external-dns/aws.md at master – kubernetes-sigs/external-dns – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/kubernetes-sigs/external-dns/blob/master/docs/tutorials/aws.md#routing-policies

[121] external-dns helm chart – bitnami/external-dns – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/bitnami/charts/tree/master/bitnami/external-dns

[122] Amazon EKS Features – Managed Kubernetes Service – Amazon Web Services, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/eks/features/

[123] GKE overview – Kubernetes Engine Documentation Google Cloud, Online, Prosinec 2020. Dostupné z https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview

[124] Introduction to Azure Kubernetes Service - Azure Kubernetes Service – Microsoft Docs, Online, Prosinec 2020. Dostupné z https://docs.microsoft.com/en-us/azure/aks/intro-kubernetes

[125] EKS vs GKE vs AKS - Evaluating Kubernetes in the Cloud – StackRox, Online, Prosinec 2020. Dostupné z https://www.stackrox.com/post/2020/10/eks-vs-gke-vs-aks/

[126] Software conformance(Certified Kubernetes) — Cloud Native Computing Foundation Twitter WeChat YouTube Github Flickr LinkedIn Meetup Slack RSS, Online, Prosinec 2020. Dostupné z https://www.cncf.io/certification/software-conformance/

[127] GitHub - vmware-tanzu/sonobuoy: Sonobuoy is a diagnostic tool that makes it easier to understand the state of a Kubernetes cluster by running a set of Kubernetes conformance tests and other plugins in an accessible and non-destructive manner., Online, Prosinec 2020. Dostupné z https://github.com/vmware-tanzu/sonobuoy

[128] Docker Hub, Online, Prosinec 2020. Dostupné z https://hub.docker.com/_/debian

[129] GitHub - tiangolo/uvicorn-gunicorn-fastapi-docker: Docker image with Uvicorn managed by Gunicorn for high-performance FastAPI web applications in Python 3.6 and above with performance auto-tuning. Optionally with Alpine Linux., Online, Prosinec 2020. Dostupné z https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker

[130] A Formal Basis for the Heuristic Determination of Minimum Cost Paths, Nilsson, N. J.; Raphael, B., Transactions on Systems Science and Cybernetics SSC4. 1968

[131] Algoritmy pro pathfinding, Lukáš Bajer, MFF UK, Online, Prosinec 2020. Dostupné z https://artemis.ms.mff.cuni.cz/main/download/hagents/H-likeAgents4_Bajer060410.pdf

[132] Contraction Hierarchies briefly explained, Stefan Funke, Online, Prosinec 2020. Dostupné z https://fmi.uni-stuttgart.de/files/alg/teaching/s15/alg/CH.pdf

[133] Exact routing in large road networks using contraction hierarchies, Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter, Transportation Science, 2012

[134] Contraction Hierarchies path finding algorithm, illustrated using three.js, Online, Prosinec 2020. Dostupné z https://www.mjt.me.uk/posts/contraction-hierarchies/

[135] Nejkratší cesty, Online, Prosinec 2020. Dostupné z https://is.muni.cz/el/1433/jaro2016/IB002/um/IB002_2016_slajdyV.pdf

[136] The Falco Project, Online, Prosinec 2020. Dostupné z https://falco.org/

[137] File integrity monitoring Falco Rules – Cloud Native Security Hub, Online, Prosinec 2020. Dostupné z https://securityhub.dev/falco-rules/file-integrity-monitoring

[138] Kubernetes Falco Rules – Cloud Native Security Hub, Online, Prosinec 2020. Dostupné z https://securityhub.dev/falco-rules/kubernetes

[139] Kubernetes Components – Kubernetes, Online, Prosinec 2020. Dostupné z https://kubernetes.io/docs/concepts/overview/components/

[140] Understanding Kubernetes Objects – Kubernetes, Online, Prosinec 2020. Dostupné z https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[141] Repositories – Docker Documentation, Online, Prosinec 2020. Dostupné z https://docs.docker.com/docker-hub/repos/

[142] Docker Pricing – Subscriptions for Individuals – Teams, Online, Prosinec 2020. Dostupné z https://www.docker.com/pricing

[143] GitLab Container Registry – GitLab, Online, Prosinec 2020. Dostupné z https://docs.gitlab.com/ee/user/packages/container_registry/

[144] Amazon ECR – Docker Container Registry – Amazon Web Services, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/ecr/

[145] Amazon ECR Pricing – Docker Container Registry – Amazon Web Services, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/ecr/pricing/

[146] Image scanning - Amazon ECR, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-scanning.html

[147] GitHub - quay/clair: Vulnerability Static Analysis for Containers, Online, Prosinec 2020. Dostupné z https://github.com/quay/clair

[148] charts/bitnami/redis at master – bitnami/charts – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/bitnami/charts/tree/master/bitnami/redis

[149] charts/stable/mongodb at master – helm/charts – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/helm/charts/tree/master/stable/mongodb

[150] charts/stable/mongodb at master – helm/charts – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/elastic/helm-charts

[151] GitLab CI/CD – GitLab, Online, Prosinec 2020. Dostupné z https://docs.gitlab.com/ee/ci/

[152] Pricing and Plan Information - CircleCI, Online, Prosinec 2020. Dostupné z https://circleci.com/pricing/

[153] AWS CodePipeline Pricing – Amazon Web Services, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/codepipeline/pricing/?nc=sn&loc=3

[154] AWS CodeBuild Pricing – Amazon Web Services, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/codebuild/pricing/?nc=sn&loc=3

[155] AWS CodeDeploy Pricing – Amazon Web Services, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/codedeploy/pricing/

[156] Vertical Pod Autoscaler - Amazon EKS, Online, Prosinec 2020. Dostupné z https://docs.aws.amazon.com/eks/latest/userguide/vertical-pod-autoscaler.html

[157] autoscaler/cluster-autoscaler at master – kubernetes/autoscaler – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

[158] charts/stable/cluster-autoscaler at master – helm/charts – GitHub, Online, Prosinec 2020. Dostupné z https://github.com/helm/charts/tree/master/stable/cluster-autoscaler

[159] How can I set up Cluster Autoscaler on Amazon EKS?, Online, Prosinec 2020. Dostupné z https://aws.amazon.com/premiumsupport/knowledge-center/eks-cluster-autoscaler-setup/

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

XML Extensible markup language

CSV Comma-separated values

YAML YAML Ain't Markup Language

JSON JavaScript Object Notation

PBF Protocolbuffer Binary Format

API Application Programming Interface

CLI Command Line Interface

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

TCP Transmission Control Protocol

UDP User Datagram Protocol

TLS Transport Layer Security

OTP Open Trip Planner

OSRM Open Source Routing Machine

OSM Open Street Mao

GPS Global Positioning System

GTFS General Transit Feed Specification

MaaS Mobility as a Service

PaaS Platform as a Service

CPU Central Processing Unit

RAM Random Access Memory

BSD Berkeley Software Distribution

LGPL Lesser General Public License

RC Release Candidate

CD Continuous Deployment

CI Continuous Integration

CRI Container Runtime Interface

HA High Availability

GKE Google Kubernetes Engine

GCP Google Cloud Platform

EKS Elastic Kubernetes Service

AWS Amazon Web Services

ALB Application Load Balancer

VPC Virtual Private Cloud

EC2 Elastic Compute Cloud

IAM Identity and Access Management

RBAC Role-based access control

OIDC OpenID Connect

SSOT Single source of truth

HPA Horizontal Pod Autoscaler

IaC Infrastructure as Code

DNS Domain Name System

CNCF Cloud Native Computing Foundation

AZ Availability zone

Příloha B

Obsah přiloženého CD

- `README.md` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF