**Česká zemědělská univerzita v Praze**

**Provozně ekonomická fakulta**

**Katedra informačních technologií**

# Diplomová práce

## Komparace architektury mikroslužeb a Serverless

**Jan Businský**

# ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

# ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Jan Businský

Systémové inženýrství a informatika
Informatika

Název práce

**Komparace architektury mikroslužeb a Serverless**

Název anglicky

**Microservices and Serverless architecture comparation**

---

**Cíle práce**

Diplomová práce je tematicky zaměřena na problematiku mikroslužeb a serverless řešení. Hlavním cílem práce je komparace architektury mikroslužeb a Serverless pro specifické situace. Dílčí cíle práce jsou:
- vypracování přehledu možností mikroslužeb,
- vypracování přehledu možností serverless řešení,
- definování specifických situací řešených pomocí mikroslužeb a serverless řešení.

**Metodika**

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Vlastní práce spočívá na základě definovaných specifických situací v objektivní komparaci architektury mikroslužeb a Serverless řešení. Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry diplomové práce.

**Doporučený rozsah práce**

60 – 80 stran textu

**Klíčová slova**

Mikroslužby, serverless, FaaS, IaaS, cloud computing, PaaS, containerization

---

**Doporučené zdroje informací**

HUMBLE, Jez a David FARLEY, 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston: Addison-Wesley Professional. ISBN 9780321601919.

LEWIS, James, 2014. Microservices. MartinFowler.com [online]. Chicago: martinFowler.com [cit. 2021-6-20]. Dostupné z: https://martinfowler.com/articles/microservices.html

NEWMAN, Sam, 2015. Building Microservices: Designing Fine-Grained Systems. Newton: O'Reilly Media. ISBN 978-1491950357.

ROBERTS, Mike, 2018. Serverless Architectures. MartinFowler.com [online]. Chicago: martinFowler.com [cit. 2021-6-20]. Dostupné z: https://martinfowler.com/articles/serverless.html

---

**Předběžný termín obhajoby**

2022/23 ZS – PEF

**Vedoucí práce**

Ing. Pavel Šimek, Ph.D.

**Garantující pracoviště**

Katedra informačních technologií

Elektronicky schváleno dne 9. 8. 2021

**doc. Ing. Jiří Vaněk, Ph.D.**

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2021

**Ing. Martin Pelikán, Ph.D.**

Děkan

V Praze dne 28. 03. 2023

**Čestné prohlášení**

Prohlašuji, že svou diplomovou práci "Komparace architektury mikroslužeb a Serverless" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2023 _____

**Poděkování**

Rád bych touto cestou poděkoval doc. Ing. Pavlu Šimkovi Ph.D. za podnětné rady, připomínky a trpělivost při vedení mé závěrečné práce. Rád bych také poděkoval své rodině, všem přátelům a kolegům, kteří mě při vytváření této práce podpořili a bez jejichž podpory by nebylo možné práci dokončit.

# Komparace architektury mikroslužeb a Serverless


## Abstrakt

Rychlý vývoj paradigmat vývoje a nasazení softwaru dal vzniknout inovativním přístupům, jako je architektura mikroslužeb a serverless. Tato práce představuje komplexní srovnání těchto dvou architektonických stylů se zaměřením na jejich pokroky, výzvy a příležitosti v moderním vývoji softwaru. Hlavním cílem je poskytnout vhled do praktických důsledků přijetí a optimalizace těchto technologií v různých softwarových projektech.

Diplomová práce začíná podrobným přehledem architektury mikroslužeb a serverless computingu, nastíněním jejich historického kontextu, principů návrhu a typických případů použití. Následuje podrobné srovnání, které zkoumá silné a slabé stránky jednotlivých přístupů, pokud jde o škálovatelnost, výkon, složitost vývoje a cenu.

Kromě toho tato práce identifikuje nové trendy, výzvy a příležitosti v oblasti distribuovaných výpočtů a nabízí náhled na možný budoucí pokrok a oblasti výzkumu. Poskytnutím analýzy architektury mikroslužeb a serverless computingu si tato práce klade za cíl přispět k pochopení a přijetí těchto nejmodernějších technologií ve stále se vyvíjejícím prostředí vývoje softwaru.



**Klíčová slova:** mikroslužby, mikroservisy, serverless, softwarová architektura, FaaS, komparace, PaaS, IaaS, cloud, cloud computing

# Microservices and Serverless architecture comparation

**Abstract**

The rapid evolution of software development and deployment paradigms has given rise to innovative approaches such as microservices architecture and serverless. This paper presents a comprehensive comparison of these two architectural styles, focusing on their advances, challenges, and opportunities in modern software development. The main goal is to provide insight into the practical implications of adopting and optimizing these technologies in various software projects.

The thesis begins with a detailed overview of microservices architecture and serverless computing, outlining their historical context, design principles and typical use cases. This is followed by a detailed comparison that examines the strengths and weaknesses of each approach in terms of scalability, performance, development complexity, and cost.

In addition, this paper identifies emerging trends, challenges, and opportunities in distributed computing and offers insights into potential future advances and research areas. By providing an analysis of microservices architecture and serverless computing, this thesis aims to contribute to the understanding and adoption of these state-of-the-art technologies in the ever-evolving software development landscape.

**Keywords**: microservices, serverless, software architecture, FaaS, comparation, PaaS, IaaS, cloud, cloud computing

# Obsah

# 1 Úvod

V posledních letech došlo ke změně paradigmatu ve způsobu navrhování, vývoje a nasazování softwarových systémů. Nástup distribuovaných a modulárních výpočetních architektur, jako jsou mikroslužby a serverless, otevřel cestu ke škálovatelnějším, flexibilnějším a efektivnějším softwarovým řešením. Cílem této práce je poskytnout komplexní srovnání architektury mikroslužeb a serverless a prozkoumat jejich pokrok, výzvy a příležitosti při vývoji moderního softwaru.

Architektura mikroslužeb je přístup k vývoji softwaru, který strukturuje aplikaci jako soubor volně provázaných služeb. Každá služba je zodpovědná za konkrétní funkci a může být nezávisle vyvíjena, nasazována a škálována. Tento architektonický styl si získal značnou pozornost díky své schopnosti podporovat agilitu, odolnost a snadnou údržbu komplexních softwarových systémů.

Naproti tomu serverless je model provádění cloud computingu, kde jsou infrastruktura a její správa abstrahovány od vývojářů. Umožňuje automatické škálování zdrojů na základě poptávky, což vývojářům umožňuje soustředit se na psaní kódu bez starostí o základní infrastrukturu. Serverless se široce rozšířil pro svou nákladovou efektivitu, škálovatelnost a zjednodušený proces vývoje.

Tato práce nejprve poskytne ucelený přehled architektury mikroslužeb a serverless, včetně jejich historického kontextu, principů návrhu a běžných případů použití. Následně se pustí do podrobného srovnání těchto dvou architektonických stylů a prozkoumá jejich silné a slabé stránky z různých hledisek, jako je škálovatelnost, výkonnost, složitost vývoje a udržovatelnost.

V neposlední řadě se práce bude zabývat novými trendy, výzvami a příležitostmi v oblasti distribuovaných aplikací a nabídne vhled do možných budoucích pokroků a oblastí výzkumu. Poskytnutím důkladné analýzy architektury mikroslužeb a serverless chce tato práce přispět k pochopení a přijetí těchto špičkových technologií ve stále se vyvíjejícím prostředí vývoje softwaru.

# 2 Cíl práce a metodika

## 2.1 Cíl práce

Diplomová práce je tematicky zaměřena na problematiku mikroslužeb a serverless řešení. Hlavním cílem práce je komparace architektury mikroslužeb a serverless pro specifické situace.

Dílčí cíle práce jsou:

- Vypracování přehledu možností mikroslužeb
- Vypracování přehledu možností serverless řešení
- definování specifických situací řešených pomocí mikroslužeb a serverless řešení.

## 2.2 Metodika

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Vlastní práce spočívá na základě definovaných specifických situací v objektivní komparaci architektury mikroslužeb a serverless řešení.

Prostřednictvím vícekriteriální analýzy variant bude vybrán poskytovatel cloudových služeb, který následně bude podroben cenové analýze vybraných služeb a zátěžovému testu. Bude vytvořena modelová aplikace, na jejímž příkladu budou vybrány specifické modelové příklady, které budou podrobeny komparaci. Komparace bude provedena vícekriteriální analýzou variant s využitím výsledků cenové analýzy, zátěžového testu a dalších kritérií.

Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry diplomové práce.

# 3 Teoretická východiska

## 3.1 Softwarová architektura

Softwarové systémy jsou konstruovány s cílem uspokojit obchodní potřeby dané společnosti. Architektura je mostem mezi těmito potřebami a výsledným systémem. Zatímco cesta od abstraktních cílů ke konkrétnímu systému může být komplexní, dobrá zpráva je, že nové softwarové architektury mohou být navrženy, analyzovány, zdokumentovány a implementovány pomocí známých technik. Tyto techniky umožňují dosáhnout stanovených obchodních cílů a komplexitu lze zkrotit tím, že ji rozložíme na menší, lépe sledovatelné části. (1 str. 3)

Existuje mnoho definicí softwarové architektury, které lze jednoduše dohledat webovým vyhledavačem, ale nejlépe odpovídá tato: *Softwarová architektura systému je množina struktur potřebných k uvažování o systému, které zahrnují softwarové prvky, vztahy mezi nimi a vlastnosti obou.* (1 str. 4)

Z definice vyplývá, že struktura je množina prvků, které jsou drženy pohromadě za pomoci vztahů mezi nimi. Softwarové systémy jsou složeny z mnoha struktur a žádnou samotnou strukturu nelze považovat za architekturu. Existují tři kategorie architektonických struktur, které hrají významnou roli při návrhu, dokumentaci a analýze architektur. (1 str. 4)

První z nich jsou moduly. Některé struktury rozdělí systém do implementačních jednotek. Modulům je přiřazena konkrétní výpočetní odpovědnost a jsou také základem pro přiřazení pracovních úkolů a odpovědností jednotlivým vývojovým týmům. Struktura, která zachycuje tento rozklad, je druhem modulové struktury, jinými slovy struktury modulového rozkladu. Dalším druhem modulové struktury je výstup objektově orientované analýzy a diagramů návrhových tříd. Modulové struktury jsou statické struktury, které se zaměřují na způsob rozdělení funkcionalit systému a přiřazení vývojovým týmům u těchto systémů. (1 str. 4)

Druhou kategorií jsou struktury dynamické. Zaměřují se na způsob interakce jednotlivých prvků mezi sebou za běhu systému. Za předpokladu že systém má být postaven jako množina služeb, pak služby interagují s infrastrukturou. Tyto synchronizační a interakční vztahy mezi nimi mají za následek další druh struktury často používaný k popisu

systému. Tyto služby se skládají z (zkompilovaných) programů v různých implementačních jednotkách, někdy jsou také nazývány jako komponenty. (1 str. 5)

Poslední kategorií je struktura popisující mapovaní ze softwarových struktur na organizační, vývojová, instalační a spouštěcí prostředí systému u těchto systémů. Moduly jsou například přiřazeny týmům k vývoji a přiřazeny k místům ve struktuře souborů pro implementaci, integraci a testování. Komponenty jsou nasazeny na hardware za účelem spuštění. Tato mapování se nazývají alokační struktury. (1 str. 5)

Přestože software zahrnuje nekonečné množství struktur, ne všechny jsou architektonické. Například sada řádků zdrojového kódu, které obsahují písmeno „z“, seřazené podle rostoucí délky od nejkratší k nejdelší, je softwarová struktura. Ale není to moc zajímavé, ani architektonické. Struktura je architektonická, pokud podporuje uvažování o systému a jeho vlastnostech. Úvaha by se měla týkat atributu systému, který je důležitý pro nějakou zainteresovanou stranu. Mezi ně patří funkčnost dosažená systémem, dostupnost systému tváří v tvář poruchám, obtížnost provádění konkrétních změn v systému, schopnost systému reagovat na požadavky uživatelů a mnoho dalších. (1 str. 5)

Architektura je tedy především abstrakcí systému, která vybírá určité detaily a jiné potlačuje. Ve všech moderních systémech spolu prvky interagují pomocí rozhraní, která rozdělují podrobnosti o prvku na veřejné a soukromé části. Architektura se zabývá veřejnou stránkou tohoto rozdělení; soukromé detaily prvků, detaily, které se týkají výhradně interní implementace, nejsou architektonické. Kromě rozhraní nám však architektonická abstrakce umožňuje podívat se na systém z hlediska jeho prvků, jak jsou uspořádány, jak interagují, jak jsou složeny, jaké jsou jejich vlastnosti, které podporují naše systémové uvažování, a tak dále. Tato abstrakce je nezbytná pro zkrocení složitosti systému, se kterým se prostě nemůžeme a nechceme neustále vypořádat. (1 str. 6)

### 3.1.1 Modulové struktury

- **Struktura dekompozice**. Jednotky v této struktuře jsou moduly, které jsou vzájemně propojeny vztahem "je podmodulem jiného modulu". Tento vztah ukazuje, jak jsou moduly rekurzivně rozkládány na menší moduly, dokud nejsou moduly dostatečně malé, aby je bylo možné snadno pochopit. Moduly v této struktuře představují společný výchozí bod pro návrh, protože architekt vyjmenovává, co budou jednotky softwaru muset dělat, a každou položku

přiřadí modulu pro následný (podrobnější) návrh a případnou implementaci. Moduly mají často spojené produkty (jako jsou specifikace rozhraní, kód, testovací plány atd.). Struktura rozkladu určuje do značné míry modifikovatelnost systému tím, že zajišťuje, že pravděpodobné změny jsou lokalizovány. To znamená, že změny spadají do působnosti nanejvýš několika (nejlépe malých) modulů. Tato struktura se často používá jako základ pro organizaci vývojového projektu, včetně struktury dokumentace a integračních a testovacích plánů projektu. Jednotky v této struktuře mívají názvy, které jsou specifické pro organizaci, například „segment“ nebo „subsystém“. (1 str. 12)

- **Strukturu použití.** V této důležité, ale přehlížené struktuře jsou zde jednotky také moduly, možná třídy. Jednotky jsou propojeny relací „use“, specializovanou formou závislosti. Jednotka softwaru používá jinou, pokud správnost první vyžaduje přítomnost správně fungující verze (na rozdíl od útržku) druhé. Struktura použití se používá k návrhu systémů, které lze rozšířit o přidání funkčnosti nebo ze kterých lze extrahovat užitečné funkční podmnožiny. Schopnost snadno vytvořit podmnožinu systému umožňuje postupný vývoj. (1 str. 12)
- **Struktura vrstev**. Moduly v této struktuře se nazývají vrstvy. Vrstva je abstraktní „virtuální stroj“, který poskytuje soudržnou sadu služeb prostřednictvím spravovaného rozhraní. Vrstvy mohou používat další vrstvy přísně řízeným způsobem; v přísně vrstvených systémech může vrstva používat pouze vrstvu bezprostředně pod ní. Tato struktura se používá k naplnění systému přenositelností, schopností změnit základní výpočetní platformu. (1 str. 12)
- **Struktura tříd.** Modulové jednotky v této struktuře se nazývají třídy. Relace mezi třídami je že dědí od rodičovské třídy nebo je instancí dané třídy. Tento pohled podporuje úvahy o kolekcích podobného chování nebo schopností (např. tříd, od kterých jiné třídy dědí) a parametrizovaných rozdílech. Struktura třídy umožňuje uvažovat o opětovném použití a postupném přidávání funkcí. Pokud existuje nějaká dokumentace pro projekt, který

prošel objektově orientovanou analýzou a procesem návrhu, je to obvykle tato struktura. (1 str. 12)

- **Datový model.** Datový model popisuje statickou informační strukturu jako datové entity a jejich vztahy. Například v bankovním systému budou entity obvykle zahrnovat účet, zákazníka a půjčku. Účet má několik atributů, jako je číslo účtu, typ (spořící nebo běžný), stav a aktuální zůstatek. Vztah může diktovat, že jeden zákazník může mít jeden nebo 1 žádný účet a jeden účet je spojen s jedním nebo dvěma zákazníky. (1 str. 13)

### 3.1.2 Komponentové struktury

Struktury komponent ukazují běhový pohled na systém. V těchto strukturách byly všechny výše popsané moduly zkompilovány do spustitelných forem. Všechny struktury komponent jsou tedy ortogonální ke strukturám založeným na modulech a zabývají se dynamickými aspekty běžícího systému. Vztah ve všech strukturách komponent je připojení, které ukazuje, jak jsou komponenty spojeny dohromady. Užitečné komponentové struktury zahrnují následující: (1 str. 13)

- **Struktura služeb**. Jednotky jsou zde služby, které vzájemně spolupracují prostřednictvím mechanismů koordinace služeb, jako je SOAP (dnes také REST a GraphQL). Struktura služeb je důležitou strukturou, která pomáhá navrhnout systém složený z komponent, které mohly být vyvinuty anonymně a nezávisle na sobě. (1 str. 13)
- **Struktura souběžnosti**. Tato struktura umožňuje určit příležitosti pro paralelismus a místa, kde může docházet ke sporům o zdroje. Jednotky jsou komponenty a konektory jsou jejich komunikační mechanismy. Komponenty jsou uspořádány do logických vláken; logické vlákno je posloupnost výpočtů, které lze později v procesu návrhu přidělit samostatnému fyzickému vláknu. Struktura souběžnosti se používá na začátku procesu návrhu k identifikaci požadavků na řízení problémů spojených se souběžným prováděním. (1 str. 13)

### 3.1.3 Alokační struktury

Alokační struktury definují, jak se prvky z komponentové nebo modulové struktury mapují na věci, které nejsou software: typicky hardware, týmy a systémy souborů. Mezi užitečné alokační struktury patří: (1 str. 14)

- **Struktura nasazení**. Struktura nasazení ukazuje, jak je software přiřazen k hardwarovým prvkům zpracování a komunikace. Prvky jsou softwarové prvky, hardwarové entity a komunikační cesty. Relace jsou alokovány, což ukazuje, na kterých fyzických jednotkách se softwarové prvky nacházejí, a migrují do, pokud je alokace dynamická. Tuto strukturu lze použít k posouzení výkonu, integrity dat, zabezpečení a dostupnosti. Zvláštní zájem je o distribuované a paralelní systémy. (1 str. 14)
- **Implementační struktura.** Tato struktura ukazuje, jak jsou softwarové prvky (obvykle moduly) mapovány do struktury souborů v prostředí vývoje, integrace nebo konfigurace systému. To je zásadní pro řízení vývojových aktivit a procesů budování. (1 str. 14)
- **Struktura zadání práce.** Tato struktura přiděluje odpovědnost za implementaci a integraci 1uzlů týmům, které ji budou provádět. Tím, že součástí architektury je struktura zadání práce, je jasné, že rozhodnutí o tom, kdo práci udělá, má architektonické i manažerské důsledky. Architekt bude znát odborné znalosti potřebné pro každý tým. U velkých distribuovaných vývojových projektů s více zdroji je struktura pracovních úkolů také prostředkem pro vyvolání jednotek funkčních společných funkcí a jejich přiřazení k jednomu týmu, místo aby je zaváděl každý, kdo je potřebuje. Tato struktura bude také určovat hlavní komunikační cesty mezi týmy: pravidelné telekonference, wiki, e-mailové seznamy a tak dále. (1 str. 14)

## 3.2 Cloud computing

Cloud computing je ve zjednodušené formě poskytování počítačových služeb přes internet. Včetně serverů, úložišť, databází, sítí, softwaru, analytiky a inteligence. S cílem nabídnout rychlejší inovace, flexibilní zdroje a úspory z rozsahu. Obvykle se platí pouze za cloudové služby nebo zdroje, které se používají či spotřebují. To pomáhá snížit provozní náklady, provozovat infrastrukturu efektivněji a škálovat podle změn a potřeb. (2)

Všechny cloudy nejsou stejné a ani jeden typ cloud computingu není vhodný pro každé řešení. Proto se dělí na několik typů. Nejprve je potřeba určit typ nasazení cloudu nebo architektury cloud computingu, na kterém budou navrhované cloudové služby implementovány. Existují tři různé způsoby nasazení cloudových služeb ve veřejném cloudu, privátním cloudu nebo hybridním cloudu. (3 str. 54)

1. **Veřejný cloud** – Veřejné cloudy vlastní a provozují poskytovatelé cloudových služeb třetí strany, kteří dodávají své výpočetní zdroje, jako jsou servery a úložiště, přes internet. Microsoft Azure je příkladem veřejného cloudu. V případě veřejného cloudu je veškerý hardware, software a další podpůrná infrastruktura vlastněna a spravována poskytovatelem cloudu. K těmto službám přistupujete a svůj účet spravujete pomocí webového prohlížeče. (4)
2. **Privátní cloud** – Soukromý cloud označuje zdroje cloud computingu používané výhradně jednou firmou nebo organizací. Privátní cloud může být fyzicky umístěn v datovém centru společnosti na místě. Některé společnosti také platí poskytovatelům služeb třetích stran za hostování jejich privátního cloudu. Privátní cloud je takový, ve kterém jsou služby a infrastruktura udržovány v privátní síti. (4)
3. **Hybridní cloud** – Hybridní cloudy kombinují veřejné a privátní cloudy, spojené dohromady technologií, která umožňuje sdílení dat a aplikací mezi nimi. Tím, že umožňuje přesun dat a aplikací mezi soukromými a veřejnými cloudy, poskytuje hybridní cloud vaší firmě větší flexibilitu, více možností nasazení a pomáhá optimalizovat vaši stávající infrastrukturu, zabezpečení a dodržování předpisů. (4)

Základní charakteristiky cloud computingu:

1. **Samoobsluha na vyžádání** – Spotřebitel si může jednostranně zajistit výpočetní schopnosti, např. čas na serveru a síťové úložiště, automaticky podle potřeby, aniž by bylo nutné, aby se o ně staral člověk. interakce s jednotlivými poskytovateli služeb. (5 str. 2)
2. **Všudypřítomný přístup k síti** – Prostředky jsou k dispozici v síti a přístup k nim je možný prostřednictvím standardních mechanismů, které podporují používání heterogenních tenkých nebo tlustých klientských platforem (např, mobilní telefony, tablety, notebooky a pracovní stanice). (5 str. 2)
3. **Sdílení zdrojů –** Výpočetní zdroje poskytovatele jsou sdruženy tak, aby sloužily více spotřebitelům pomocí modelu více nájemců, přičemž různé fyzické a virtuální zdroje jsou dynamicky přidělovány a přerozdělovány podle poptávky spotřebitelů. Existuje zde pocit nezávislosti na umístění, protože zákazník obecně nemá kontrolu nad přesným umístěním poskytovaných zdrojů ani o něm nemá žádné znalosti, ale může být schopen určit umístění na vyšší úrovni abstrakce (např. země, stát nebo datové centrum). (5 str. 2)
4. **Elasticita** – Prostředky lze pružně poskytovat a uvolňovat, v některých případech automaticky, aby se rychle rozšiřovaly a zvyšovaly podle poptávky. Spotřebiteli se často zdá, že schopnosti, které jsou k dispozici pro poskytování, jsou neomezené a lze si je kdykoli přivlastnit v libovolném množství. (5 str. 2)
5. **Metriky** – Cloudové systémy automaticky řídí a optimalizují využívání zdrojů pomocí funkce měření na určité úrovni abstrakce odpovídající typu služby (např. úložiště, zpracování, šířka pásma a aktivní uživatelské účty). Využívání zdrojů lze monitorovat, kontrolovat a vykazovat, což zajišťuje transparentnost pro poskytovatele i spotřebitele využívané služby. (5 str. 2)

Mezi nejvýznamnější poskytovatele cloudových computingu se řadí (6):

- Amazon Web Services
- Google Cloud Platform

- Microsoft Azure
- Alibaba Cloud
- IBM Cloud
- Salesforce
- Tencent Cloud
- Oracle

Zde je stručný přehled jednotlivých platforem:

Amazon Web Services, založená v roce 2006, je komplexní a široce rozšířená cloudová platforma nabízející více než 200 plně funkčních služeb z datových center po celém světě. AWS poskytuje různé služby, jako je výpočetní výkon, úložiště a databáze, které pomáhají firmám škálovat a růst. Miliony zákazníků, včetně začínajících firem, podniků a organizací veřejného sektoru, spoléhají na AWS v oblasti infrastruktury. (7)

Google Cloud Platform, spuštěná v roce 2011, je sada služeb cloud computingu, které běží na stejné infrastruktuře, jakou Google interně používá pro své produkty pro koncové uživatele, jako je Google Search a YouTube. GCP nabízí širokou škálu služeb, včetně výpočetní techniky, úložišť, datové analýzy a strojového učení, které jsou určeny pro podniky všech velikostí. (8)

Microsoft Azure, dříve známý jako Windows Azure, je služba cloud computingu vytvořená společností Microsoft v roce 2010. Nabízí širokou škálu cloudových služeb, včetně výpočetních, analytických, úložných a síťových. Azure poskytuje podnikům flexibilitu při vytváření, nasazování a správě aplikací v globální síti pomocí preferovaných nástrojů a frameworků. (9)

Alibaba Cloud, založený v roce 2009, je odnoží společnosti Alibaba Group, která se zabývá cloud computingem. Nabízí komplexní sadu globálních služeb cloud computingu, včetně elastických výpočtů, ukládání dat, správy databází a zpracování velkých objemů dat. Alibaba Cloud má silné zastoupení v asijsko-pacifickém regionu a je určen pro podniky všech velikostí. (10)

IBM Cloud, spuštěný v roce 2011, je služba cloud computingu poskytovaná společností IBM, která nabízí soubor služeb zahrnující infrastrukturu jako službu (IaaS), platformu jako službu (PaaS) a software jako službu (SaaS). IBM Cloud je určen

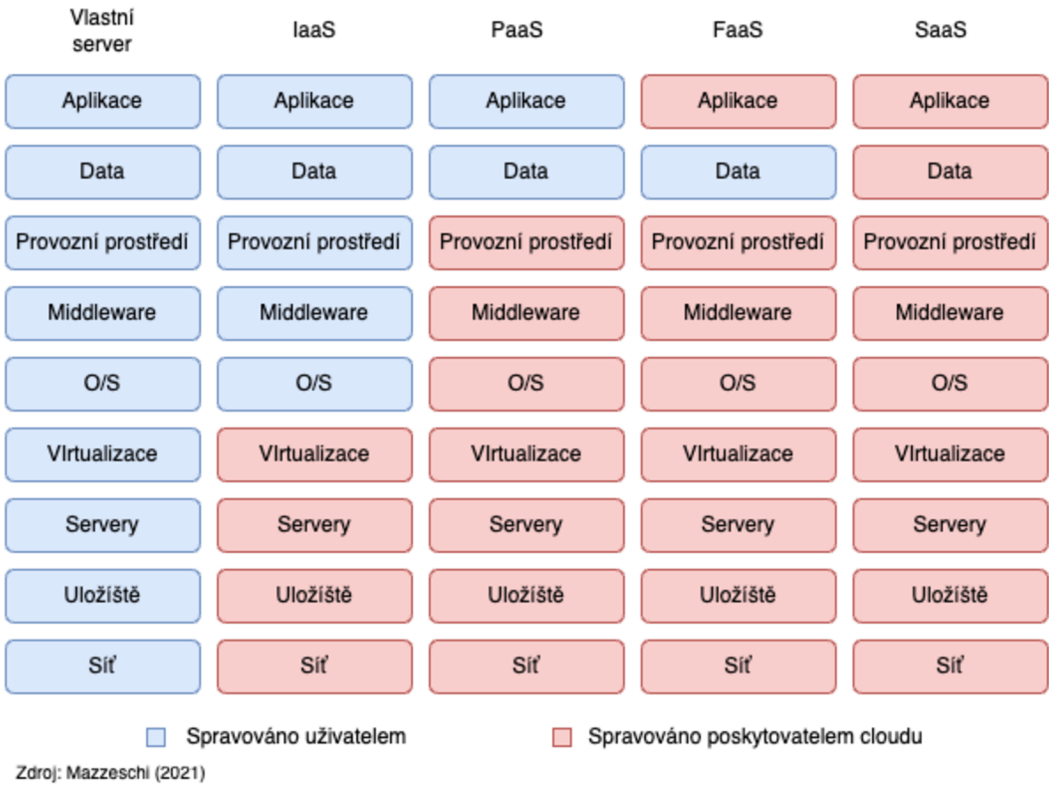
pro podniky, které vyžadují robustní, flexibilní a bezpečná cloudová řešení se zaměřením na data, analytiku a kognitivní funkce. (11)

Společnost Salesforce, založená v roce 1999, je cloudová platforma pro řízení vztahů se zákazníky (CRM), která pomáhá firmám navázat kontakt se zákazníky, partnery a zaměstnanci. Salesforce nabízí různé cloudové aplikace pro prodej, služby, marketing a další oblasti, které firmám umožňují efektivně řídit vztahy se zákazníky a obchodní procesy. (12)

Tencent Cloud, spuštěný v roce 2013, je divize cloud computingu čínského nadnárodního technologického konglomerátu Tencent. Nabízí rozsáhlou škálu cloudových služeb, včetně výpočetní techniky, úložišť, sítí a poskytování obsahu, určených především pro podniky a organizace v Číně a asijsko-pacifickém regionu. (13)

Oracle Cloud, spuštěný v roce 2012, je služba cloud computingu nabízená společností Oracle Corporation. Poskytuje různé služby, včetně softwaru jako služby (SaaS), platformy jako služby (PaaS), infrastruktury jako služby (IaaS) a dat jako služby (DaaS). Oracle Cloud je navržen tak, aby pomáhal organizacím zavádět inovace, zefektivňovat podnikové procesy a snižovat složitost IT. (14)

### 3.2.1 Modely Cloudových služeb




*Obrázek 1 Modely cloudových služeb*

Modely služeb cloud computingu nabízejí standardizovaný způsob kategorizace cloudových služeb na základě úrovně jejich infrastruktury. Existují čtyři hlavní modely služeb, z nichž každý udává počet úrovní infrastruktury, které se uživatel rozhodne pronajmout a spravovat. Model „Vlastní server“ na obrázku 1 se nepovažuje za model služby, protože se týká situace, kdy společnost uživatele kontroluje a spravuje veškerou svou infrastrukturu a nevyžaduje cloudové služby. Pochopením strukturálních rozdílů mezi modely služeb mohou uživatelé zvolit vhodný model služeb, který bude vyhovovat jejich specifickým potřebám. (15)

**Software as a Service (SaaS)**

V tomto případě je koncový uživatel spotřebitelem. Uživatel využívá různé aplikace, které fungují v cloudovém prostředí. Tyto aplikace zahrnují e-mail, kalendáře, streamování videa a aplikace pro spolupráci v reálném čase. Spotřebitel nemá na starosti ani neovládá základní infrastrukturu cloudu, jako jsou sítě, servery, operační systémy či úložiště,

ani jednotlivé funkce aplikací. Jedinou možnou výjimkou mohou být omezená uživatelská nastavení pro konfiguraci aplikací. (16)

**Platform as a Service (PaaS)**

V tomto případě jsou zákazníky vývojáři nebo správci systémů. Platforma nabízí širokou škálu služeb, z nichž si zákazník může vybírat. Mezi tyto služby patří různé možnosti databází, vyvažování zátěže, dostupnost a vývojová prostředí. Zákazník nasazuje aplikace v cloudovém prostředí pomocí programovacích jazyků a nástrojů, které poskytovatel podporuje. Zákazník nespravuje ani neřídí základní infrastrukturu cloudu, jako jsou sítě, servery, operační systémy nebo úložiště, avšak má kontrolu nad nasazenými aplikacemi a případně nad konfiguracemi prostředí hostujících aplikace. Některé úrovně kvalitativních atributů, jako jsou doba provozuschopnosti, doba odezvy, zabezpečení či doba opravy chyb, mohou být stanoveny ve smlouvách o úrovni služeb (SLA). (17)

**Infrastructure as a Service (IaaS)**

Spotřebitelem je v tomto případě vývojář nebo správce systému. Schopnost poskytovaná spotřebiteli je poskytovat zpracování, úložiště, sítě a další základní výpočetní zdroje, kde je spotřebitel schopen nasadit a provozovat libovolný software, který může zahrnovat operační systémy a aplikace. Spotřebitel se může například rozhodnout vytvořit instanci virtuálního počítače a poskytnout mu nějakou konkrétní verzi Linuxu. Zákazník nespravuje ani neřídí základní cloudovou infrastrukturu, ale má kontrolu nad operačními systémy, úložištěm, nasazenými aplikacemi a případně omezenou kontrolu nad vybranými síťovými komponentami (např. hostitelskými firewally). Opět platí, že SLA se často používají ke specifikaci klíčových atributů kvality. (18)

**Function as a Service (FaaS)**

FaaS představuje způsob implementace výpočetního prostředí bez serveru, v němž vývojáři vytvářejí aplikační logiku, jež se poté spouští v linuxových kontejnerech plně spravovaných platformou. Serverless přístup abstrahuje infrastrukturní části, jako je správa nebo zabezpečení serverů a přidělování zdrojů, od vývojářů a svěřuje je platformě, což umožňuje vývojářům soustředit se na psaní kódu a dodávání funkcí. Funkce představuje část softwaru provozující aplikační logiku v operačním systému. Aplikace mohou sestávat z mnoha funkcí. Využití modelu FaaS je jedním ze způsobů, jak vytvořit aplikaci

s architekturou serverless. Avšak s rostoucí popularitou paradigmatu serverless hledají vývojáři řešení, která podporují tvorbu serverless mikroslužeb a serverless kontejnerů. (19)

### 3.2.2 Cloudové nativní aplikace

Cloudové nativní aplikace představují kolekci malých, nezávislých a volně propojených služeb. Jsou navrženy tak, aby poskytovaly dobře uznávanou obchodní hodnotu, například schopnost rychle začleňovat uživatelskou zpětnou vazbu pro neustálé zlepšování. Stručně řečeno, vývoj cloudových aplikací urychluje tvorbu nových aplikací, optimalizaci stávajících a jejich propojení. Cílem je dodávat aplikace, které uživatelé požadují, v tempu, které je potřeba. (20)

Aplikace "cloudově nativní" jsou speciálně navrženy tak, aby poskytovaly konzistentní vývoj a automatizovanou správu napříč soukromými, veřejnými a hybridními cloudy. Organizace využívají cloud computing pro zvýšení škálovatelnosti a dostupnosti aplikací. Těchto výhod je dosaženo prostřednictvím samoobslužného poskytování zdrojů, zdrojů na vyžádání a automatizace životního cyklu aplikace od vývoje až po nasazení v produkčním prostředí. (20)

Cloud-native development je přístup k rychlému vytváření a aktualizací aplikací při současném zlepšování kvality a snižování rizik. Konkrétněji je to způsob, jak vytvářet a provozovat škálovatelné a odolné aplikace kdekoli, ať už ve veřejných, soukromých nebo hybridních cloudech. (20)

Adopce kontejnerů podporuje tyto postupy tím, že nabízí ideální jednotku pro nasazení aplikací a samostatné prováděcí prostředí. Díky DevOps a kontejnerům lze snadněji vydávat a aktualizovat aplikace jako kolekci volně propojených služeb, jako jsou mikroslužby. Místo čekání a plánování na jedno velké vydání. (20)

Cloudově nativní vývoj se zaměřuje na modularitu architektury, volné propojení a nezávislost jednotlivých služeb. Každá mikroslužba implementuje obchodní hodnotu. Běží ve svém vlastním procesu a komunikuje prostřednictvím aplikačních programovacích rozhraní (API) nebo zasíláním zpráv. Tato komunikace může být řízena prostřednictvím vrstvy service mesh. (20)

Není nutné však vždy začínat s mikroslužbami, aby se urychlilo doručování aplikací jako součást cloudových nativních aplikací. Mnoho organizací může stále optimalizovat své

starší aplikace pomocí pragmatické architektury založené na službách. Tato optimalizace je podporována pracovními postupy DevOps, jako je kontinuální integrace a kontinuální nasazení (CI/CD), plně automatizované operace nasazení a standardizovaná vývojová prostředí. (20)

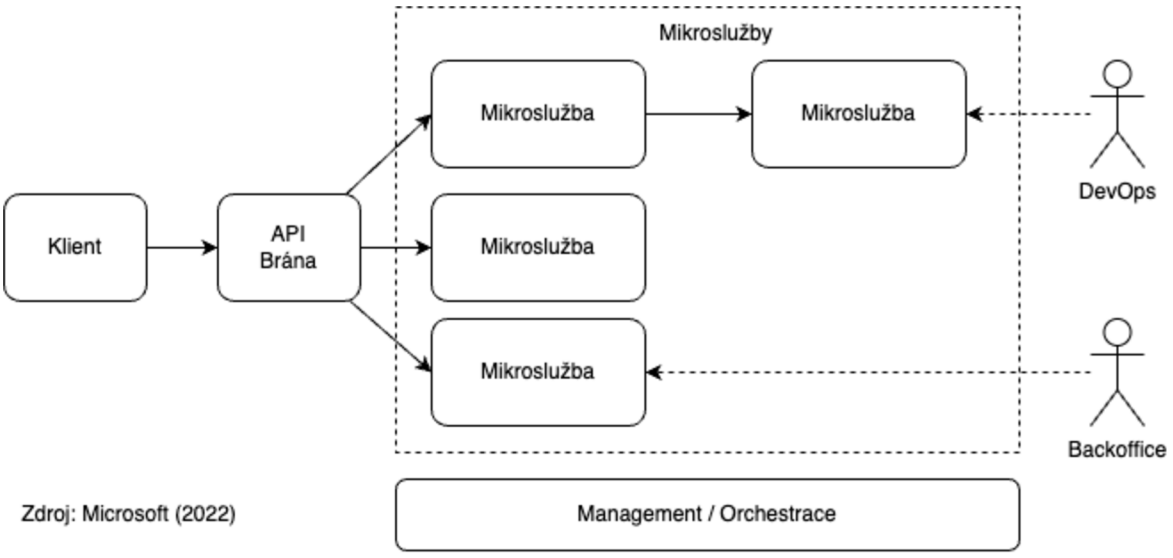
## 3.3 Mikroslužby

Architektura mikroslužeb (také označovaná jako mikroservis) označuje architektonický styl pro vývoj aplikací. Mikroslužby umožňují rozdělit velkou aplikaci na menší nezávislé části, přičemž každá část má svou vlastní oblast odpovědnosti. Aby aplikace, která je založená na mikroslužbách, obsloužila požadavek jednoho uživatele, může zavolat mnoho interních mikroslužeb, aby sestavila svou odpověď. (21)

Architektura mikroslužeb je definována:

*Architektura mikroslužeb je typ architektury aplikace, kde je aplikace vyvinuta jako kolekce služeb. Poskytuje rámec pro nezávislý vývoj, nasazení a údržbu diagramů architektury mikroslužeb.* (21)

V rámci architektury mikroslužeb je každá mikroslužba jedinou službou vytvořenou tak, aby vyhovovala funkci aplikace a zpracovávala jednotlivé úkoly, jak to znázorněné na obrázku 2. Každá mikroslužba komunikuje s ostatními službami prostřednictvím jednoduchých rozhraní a vykonává obchodní logiku. (21)




*Obrázek 2 Mikroslužby*

Výhody mikroslužeb (22):

- **Škálovatelnost** – Mikroslužby umožňují jednoduché škálování jednotlivých komponent, což poskytuje lepší kontrolu nad zdroji potřebnými pro jednotlivé části aplikace.
- **Odolnost** – Díky nezávislému fungování mikroslužby zůstává systém stabilní i při selhání jedné z nich, což zvyšuje celkovou spolehlivost systému.
- **Flexibilita** – Mikroslužby lze vyvíjet a nasazovat samostatně, což zrychluje vývojové cykly a usnadňuje údržbu.
- **Technologická rozmanitost** – Mikroslužby mohou být vytvořeny s využitím různých technologií, což je výhodné pro organizace, které chtějí rychle přijímat nové technologie nebo pro každou úlohu používat nejvhodnější nástroj.
- **Lepší udržitelnost** – U mikroslužby je jednodušší provádět změny v jednotlivých komponentách bez vlivu na celý systém, což snižuje náklady spojené s údržbou.

Nevýhody mikroslužeb (22):

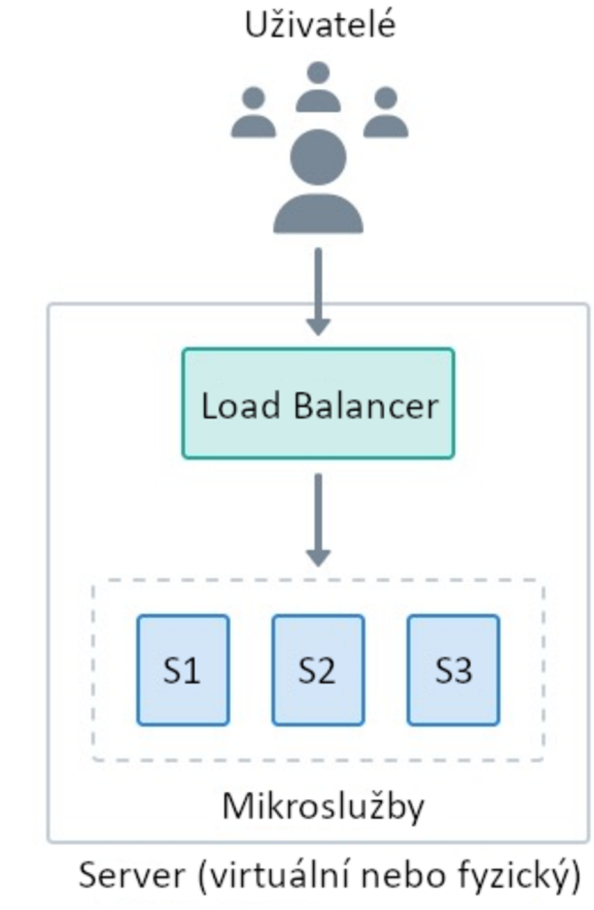
- **Složitost** – Mikroslužba mohou zvýšit složitost systému, protože vyžadují koordinaci mezi více službami a další infrastrukturu pro řízení komunikace mezi nimi.
- **Odladění** – Ladění systému založeného na mikroslužby může být náročnější, protože zahrnuje sledování požadavků napříč více službami.
- **Zpoždění** – Komunikace mezi mikroslužbami může zvýšit latenci systému a potenciálně snížit jeho celkový výkon.
- **Zvýšené náklady** – Vývoj, nasazení a údržba mikroslužeb mohou být dražší než u tradiční monolitické architektury.

- **Testování** – Testování mikroslužeb může být složitější, protože je třeba testovat každou službu zvlášť a může být nutné i další testování interakcí mezi službami.

Celkově by rozhodnutí o použití mikroslužeb mělo vycházet z konkrétních potřeb a omezení vytvářené aplikace. Mikroslužby mohou nabídnout řadu výhod, ale přinášejí s sebou také řadu výzev a kompromisů, které je třeba pečlivě zvážit. (22)

### 3.3.1 Nasazení mikroslužeb

Nejjednodušším způsobem, jak spustit aplikaci s mikroslužbami je spustit více procesů na jednom počítači. Každá služba naslouchá na jiném portu a komunikuje přes zpětné rozhraní. Tento způsob je znázorněn na obrázku 3. (23)




Zdroj: Fernandez (2022)

*Obrázek 3 Server pro spuštění služeb*

Tento jednoduchý přístup má některé jasné výhody. Jednoduchý na provoz, není zde žádná režie, protože se jedná pouze o procesy běžící na serveru. Jedná se i o skvělý způsob, jak si vyzkoušet mikroslužby bez nutnosti učit se, jak je tomu u jiných nástrojů. Snadné řešení problémů, vše je na stejném místě, takže nalezení problému nebo návrat k funkční konfiguraci v případě potíží je velmi jednoduchý. (23)

Tento přístup funguje nejlépe u malých aplikací s pouze několika mikroslužbami. Po překročení tohoto limitu je nedostatečný, protože nelze škálovat, jakmile se vyčerpají prostředky serveru je konec. Pokud server spadne, aplikace spadne s ním. (23)

Nasazení mikroslužeb je křehké. Je zapotřebí vlastních skriptů pro nasazení a monitorování, aby byla zajištěna správná instalace a běh služeb. Žádné omezení zdrojů. To znamená, že jakýkoli proces mikroslužba může spotřebovat libovolné množství procesoru nebo paměti, což může vést k vyhladovění ostatních služeb a k degradaci aplikace.

Je to jedna možnost škálování tohoto řešení a to horizontálně. To znamená stejnou konfiguraci aplikace spustit na více serverech.

Všechny zmíněné nedostatky lze zmírnit pomocí kontejnerů. Kontejnery jsou balíčky, které obsahují vše, co program potřebuje ke svému běhu. Obraz kontejneru je samostatná jednotka, kterou lze spustit na libovolném serveru, aniž by bylo nutné nejprve instalovat jakékoli závislosti nebo nástroje (kromě samotného běhového prostředí kontejneru). (23)

#### 3.3.1.1 Kontejnery

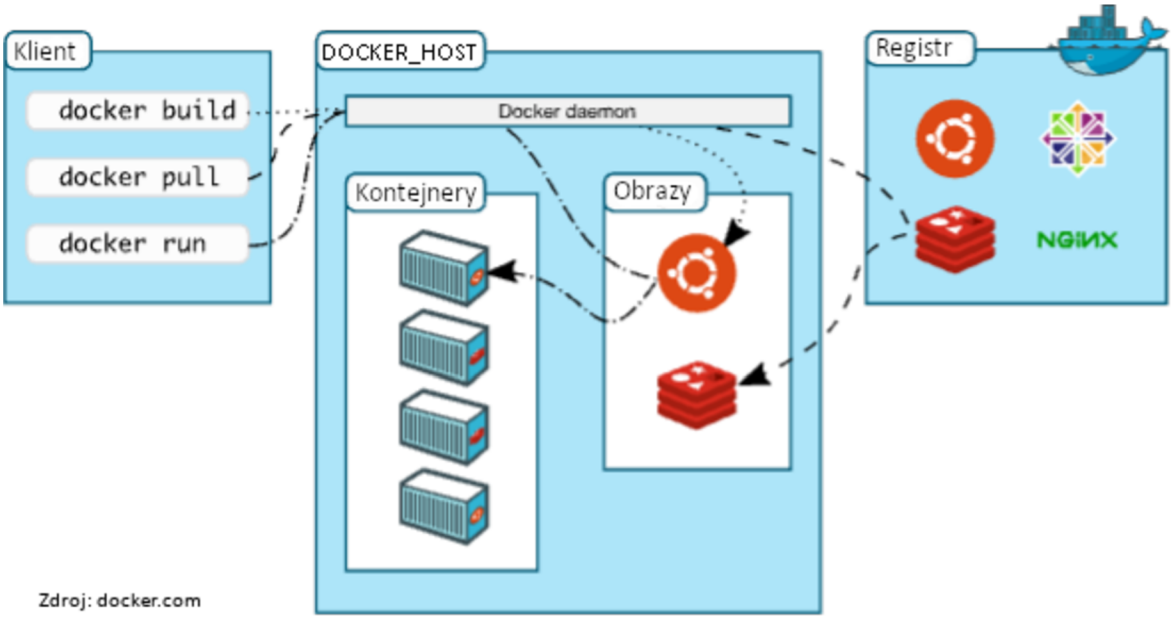
Kontejnery jsou balíčky softwaru, které obsahují všechny nezbytné prvky pro běh v jakémkoli prostředí. Tímto způsobem kontejnery virtualizují operační systém a běží kdekoli, od soukromého datového centra po veřejný cloud nebo dokonce na vývojářském osobním notebooku. (23)

Pro vytvoření kontejneru je nejprve potřeba připravit obraz. To je šablona pouze pro čtení s pokyny pro vytvoření kontejneru. Obraz je často založen na jiném obrazu s nějakým dalším přizpůsobením. Je možné například vytvořit bitovou kopii, která je založena na bitové kopii linuxového operačního systému Ubuntu, ale nainstaluje webový server Apache a vlastní aplikaci, stejně jako podrobnosti o konfiguraci potřebné ke spuštění

aplikace. Buďto je potřeba vytvořit image nebo použít některou z veřejně dostupných v registru. (24)

Nejznámější platformou pro kontejnerizaci je Docker. Docker nabízí mnoho obrazů ve svém registru, které je možné použít pro potřeby daného řešení. Nebo je možné vytvořit vlastní skrze konfigurační soubor s názvem Dockerfile. No hostitelském stroji musí běžet Docker daemon, který stará o běh kontejnerů, obrazů, virtuálních sítí a uložišť. Ovládání probíhá skrze Docker klienta, jak je znázorněno na obrázku 4. (24)

Pokud je potřeba k běhu aplikace rozběhnout jiné aplikace, bez kterých daná aplikace nemůže fungovat, je vhodné k tomu použít klienta Docker Compose. Ten umožňuje spustit set kontejnerů v dané pořadí, dle určených závislostí. Například databází pak aplikací databázového schématu, a nakonec samotnou aplikaci. (24)




*Obrázek 4 Architektura Dockeru*

Kontejnery jsou vhodným příkladem architektury mikroslužeb, protože umožňují soustředit se na vývoj služeb bez obav ze závislostí. Moderní cloudové nativní aplikace jsou obvykle vytvářeny jako mikroslužby pomocí kontejnerů.

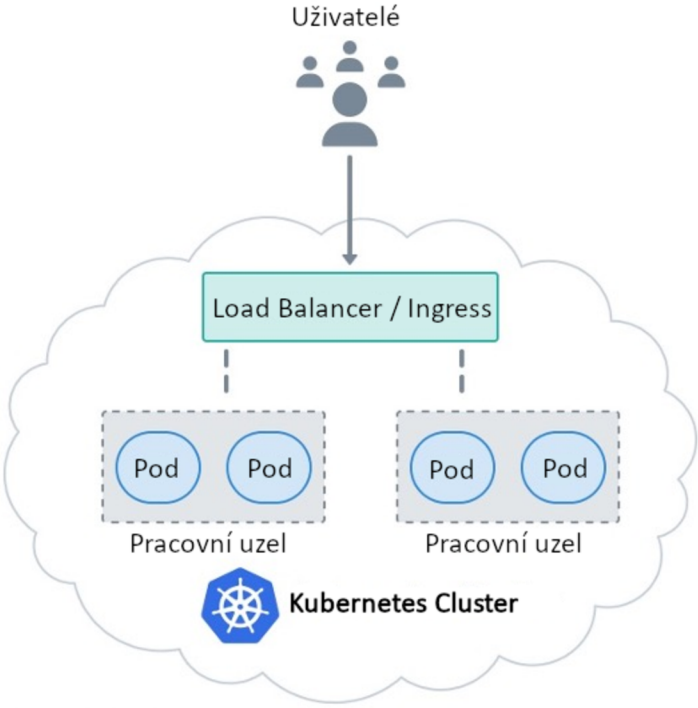
### 3.3.2 Orchestrace

Orchestrace mikroslužeb označuje proces koordinace a řízení interakce a komunikace mezi různými mikroslužbami v distribuovaném systému. Zahrnuje definování

vzájemné interakce mikroslužeb, řízení toku dat mezi nimi a zajištění toho, aby každá služba vykonávala přidělené úkoly efektivně a spolehlivě. (23)

Cílem orchestrace mikroslužeb je zjednodušit správu distribuovaného systému tím, že poskytne centralizovaný způsob řízení chování jednotlivých služeb. Zahrnuje použití nástrojů a technologií, jako jsou registry služeb, vyrovnávače zátěže a brány API, ke správě různých aspektů systému, například vyhledávání služeb, směrování a monitorování. (23)

Orchestrace mikroslužeb lze dosáhnout různými přístupy, například pomocí centralizovaného orchestračního nástroje viz obrázek 5 nebo decentralizovaného přístupu, kdy každá mikroslužba řídí své vlastní interakce s ostatními službami. Mezi oblíbené orchestrační nástroje patří Kubernetes, Docker Swarm a Apache Mesos. Tyto nástroje poskytují funkce, jako je automatické škálování, zjišťování služeb, vyrovnávání zátěže a odolnost proti chybám, což usnadňuje správu a škálování aplikací mikroslužeb. (23)




Zdroj: Fernandez (2022)

*Obrázek 5 Orchestrace mikroslužeb*

## 3.4 Serverless

Serverless představuje způsob, jak vytvářet a provozovat aplikace a služby bez nutnosti spravovat infrastrukturu. To znamená, že poskytovatel cloudu je zodpovědný

za provádění kódu, správu a škálování podkladových serverů a účtuje pouze přesné množství spotřebovaného výpočetního času. (25)

Serverless architektura umožňuje vývojářům soustředit se na návrh softwaru a kód namísto infrastruktury. Škálovatelnost a vysoká dostupnost jsou snazší k dosažení, a cena bývá často spravedlivější, neboť se platí pouze za spotřebované zdroje. Díky serverless architektuře lze potenciálně snížit složitost systému minimalizací počtu vrstev a množství potřebného kódu. (26 str. 57)

Úlohy jako konfigurace a správa serveru, opravy a údržba, jsou zajišťovány poskytovatelem, což šetří čas i peníze. Pokud nejsou konkrétní požadavky na správu nebo úpravu výpočetních zdrojů, pak nechat dodavatele o ně pečovat je skvělé řešení. Vývojáři jsou zodpovědní pouze za svůj vlastní kód, zatímco provozní a administrativní úkoly jsou ponechány v rukou poskytovatele cloudových řešení. (26 str. 57)

Bezstavovost a škálovatelnost výpočtu lze využít pro řešení problémů, které profitují z paralelního zpracování. Backendy pro CRUD aplikace, e-commerce, back-office systémy, komplexní webové aplikace a všechny druhy mobilních a desktopových softwarů lze rychle sestavit pomocí serverless architektur. Úkoly, které dříve zabíraly týdny, lze provést ve dnech nebo hodinách, pokud je zvolena správná kombinace technologií. Serverless přístup může být začínající projekty mimořádně efektivní, když chtějí inovovat a rychle postupovat. (26 str. 57)

Tradiční serverová architektura vyžaduje servery, které nemusí nutně běžet po celou dobu na plný výkon. Škálování, i u automatizovaných systémů, zahrnuje přidání nových serverů, které jsou často zbytečné, dokud nedojde k dočasnému nárůstu provozu nebo nových údajů. Na druhou stranu, serverless systémy jsou mnohem flexibilnější, pokud jde o škálování a jsou nákladově efektivní, zvláště když jsou špičkové zátěže nerovnoměrné nebo neočekávané. (26 str. 57)

Vývojáři nemusí nutně používat serverless architekturu k nahrazení celého backendu, pokud to nechtějí nebo nemohou udělat. Místo toho mohou použít například funkce k řešení specifických problémů, zejména pokud tyto problémy profitují z paralelizace. Stojí za zmínku, že serverless systémy lze škálovat snadněji než tradiční systémy. (26 str. 58)

### 3.4.1 Container as a service

V předchozí kapitole byly popsány možné způsoby nasazení mikroslužeb, které vycházeli k nasezeni na vlastních serverech. Tyto způsoby je možné nahradit serverless řešeními od poskytovatelů cloudových služeb. Jedná se o služby typu PaaS neboli Platform as a Service. Jedná se o nabídky kontejnerů jako služby (Container as a Service), jako jsou AWS Fargate a Heroku, umožňují provozovat kontejnerové aplikace bez nutnosti zabývat se servery. Stačí vytvořit obraz kontejneru a nasměrovat jej na poskytovatele cloudu, který se postará o zbytek. Zajistí virtuální stroje a stáhne, spustí a monitoruje obrazy. Tyto spravované služby obvykle obsahují vestavěný load balancer, což je o jednu starost méně. (23)

Takováto služba má pár nesporných výhod, a to že není třeba udržovat ani opravovat servery. Nasazení služeb je mnohem snazší, stačí sestavit obraz kontejneru a říct službě, aby jej použila. Obvykle je součástí služby i automatické škálování, kdy poskytovatel cloudu může poskytnout větší kapacitu při prudkém nárůstu poptávky nebo zastavit všechny kontejnery, když není žádný provoz. (23)

Jsou tu ale i nevýhody, které je potřeba zmínit. Hlavní nevýhodou je závislost na poskytovateli cloudových služeb. Odchod od spravované služby je vždy náročný, protože většinu infrastruktury poskytuje a řídí poskytoval cloudu. Tyto služby mají stanovené limity procesoru a paměti, kterým se nelze vyhnout. Omezená je i možnost kontroly nad prostředím. Pokud poskytovatel nenabízí potřebnou funkcionalitu ke kontrola není moc možností, jak to řešit. (23)

Nasazení pomocí kontejnerů je vhodné pro malé až středně velké aplikace mikroslužeb. Poskytovatelé mají stanovené limity pro maximální počet takto spuštěných aplikací. (23)

Pro rozsáhlá nasazení, kdy už je zapotřebí využít služeb orchestrace s pomocí nástroje jako je například Kubernetes i zde je možné využít služeb cloudových poskytovatelů, kteří poskytují orchestrátory jako službu. (23)

Nabídka kontejnerů jako služby od významných poskytovatelů cloudových služeb:

- AWS Fargate (27)
- GCP Cloud Run (28)

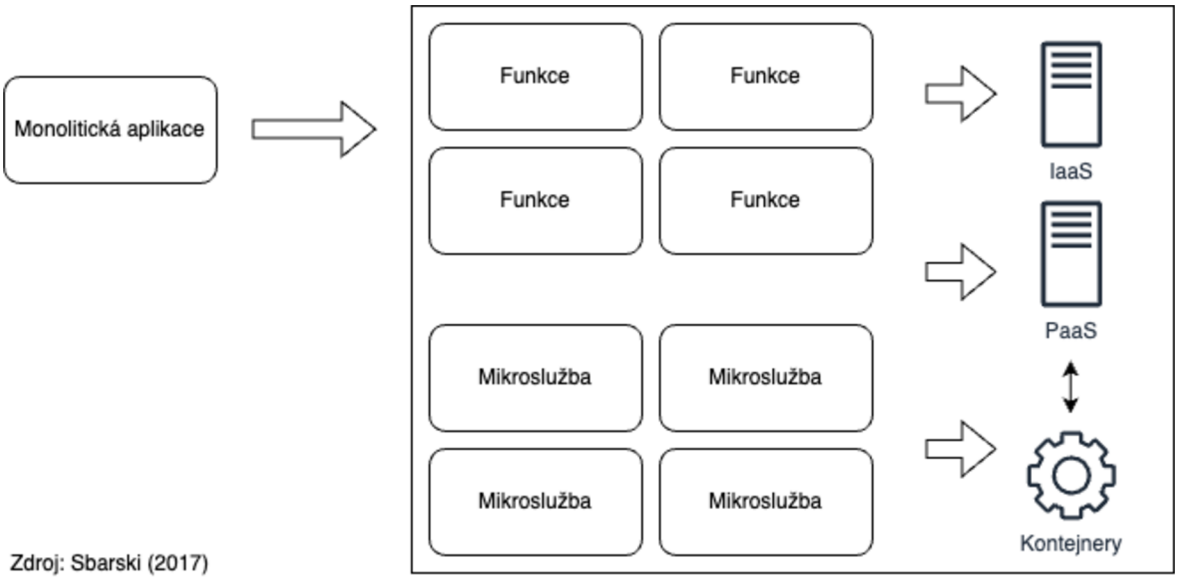
- Microsoft Azure Container Instances (29)
- IBM Cloud Engine (30)
- Heroku dynos (Salesforce) (31)
- Alibaba Elastic Container Instance (32)
- OCI Container Instances (33)

### 3.4.2 Function as a Service

Serverless funkce se odchylují od všeho, co bylo dosud zmíněno. Místo serverů, procesů nebo kontejnerů je používán cloud k jednoduchému spouštění kódu na vyžádání. Serverless produkty jako AWS Lambda, Microsoft Azure Functions a Google Cloud Functions se starají o všechny detaily infrastruktury potřebné pro škálovatelné a vysoce dostupné služby. (23)

Serverless architektury jsou přirozeným rozšířením myšlenek SOA. V serverless architekturách je veškerý vlastní kód napsán a spuštěn jako izolovaný, nezávislý, a často jako granulární funkce, které jsou spouštěny v bez stavové výpočetní službě. Vývojáři mohou napsat funkce, které provedou téměř jakýkoli běžné úlohy, jako je čtení a zápis do datového zdroje, volání jiných funkcí, a provádění výpočtů. Ve složitějších případech mohou vývojáři nastavit více propracované pipeline a orchestrovat volání více funkcí. Mohou nastat scénáře, kdy je stále potřeba, aby něco dělal server. Tyto případy jsou však okrajové a mohou být velmi vzácné. Vývojáři by se měli vyhnout interakci se serverem, pokud je to možné. (26 str. 48)

Serverless architektura není řešením typu všechno nebo nic. V případě existující monolitické aplikace, pak tento monolit může být rozložen na jednotlivé komponenty, které mohou využít široké spektrum nabízených služeb, tak aby daná technologie co nejlépe vyhověla potřeba dané komponenty. Z toho může vzniknout aplikace s hybridní architekturou, která využívá FaaS, PaaS, IaaS a služby třetích stran viz obrázek 6. (26 str. 56)




Zdroj: Sbarski (2017)

*Obrázek 6 Serverless architektura*

Shrnutí výhod a nevýhod serverless funkcí:

- Serverless není vhodné řešení za všech okolností. Nemusí být vhodné pro aplikace nebo software citlivé na latenci se specifickými smlouvami o úrovni služeb (SLA). Závislost na poskytovateli cloudových služeb může být problémem pro podnikové a vládní klienty, a decentralizace služeb může být výzvou. (26 str. 57)
- Serverless funkce běží ve veřejném cloudu, takže kritické aplikace by neměly být na nich postaveny. Bankovní systém provádějící velkoobjemové transakce nebo systém podpory života pacienta vyžaduje vyšší úroveň výkonu a spolehlivosti, než může poskytnout veřejný cloud. Organizace mohou využívat vyhrazený hardware nebo provozovat privátní či hybridní cloudy s vlastním výpočetním výkonem, které mohou splňovat požadavky na provozuschopnost a spolehlivost. V takovém případě by tyto architektury mohly být použity. (26 str. 57)
- Efektivita získaná tím, že se poskytovatel stará funkce platformy a škálování přicházejí na úkor možnosti přizpůsobení operační systému nebo ladění základní instance. K dispozici je konfigurace RAM přidělené funkci a časové limity změn, ale to je v základu vše. Podobně se různé služby třetích stran budou lišit úrovně přizpůsobení a flexibility. (26 str. 57)

- Závislost na poskytovateli. Pokud se vývojář rozhodne používat API třetích stran a služeb, existuje šance, že by architektura mohla být silná spojené s používanou platformou. Důsledky závislosti na poskytovateli a riziko používání služeb třetích stran – včetně životaschopnosti společnosti, suverenity dat a soukromí, náklady, podpora, dokumentace a dostupná sada funkcí – je potřeba důkladně zvážit a promyslet. (26 str. 57)
- Přechod od monolitického přístupu k více decentralizovanému přístupu bez serveru automaticky nesnižuje ani složitost základního systému. Distribuovaná povaha řešení může představovat své vlastní problémy, protože potřeba provádět vzdálená volání spíše než volání v průběhu procesu a potřeba řešit selhání a latence v síti. (26 str. 57)

Nabídka funkcí jako služby od významných poskytovatelů cloudových služeb:

- AWS Lambda (34)
- Microsoft Azure Functions (35)
- GCP Cloud Functions (36)
- IBM Cloud Functions (37)
- Alibaba Function Compute (38)
- Tencent Serverless Cloud Function (39)
- OCI Cloud Functions (40)

### 3.4.3 Netflix použití FaaS v praxi

Společnost Netflix k provozu své streamovací služby využívá cloudových služeb od poskytovatele Amazon Web Services. Již po několik let využívá i Function as a Service. Zde je několik příkladu užití přímo od společnosti Netflix. (41)

Vydavatelé denně nahrávají do Netflixu tisíce souborů a každý kus těchto souborů musí být zakódován a roztříděn, než je nakonec zpřístupněn uživatelům. Jakmile se soubory nahrají do uložiště S3, Amazon spustí událost, která zavolá funkci AWS Lambda, která video rozdělí na pětiminutové kousky, jež se zakódují do 60 různých paralelních

streamů, které Netflix potřebuje. Jakmile se zpracuje poslední část videa, dojde k jejich agregaci a nasazení pomocí řady pravidel a událostí. (42)

Dalším způsobem, jak Netflix využívá AWS Lambda, je jejich zálohovací systém. Vzhledem k tomu, že se denně mění a upravují tisíce souborů, lambdy kontrolují, zda je třeba soubory zálohovat, kontrolují platnost a integritu souborů, a pokud něco selže, mohou se vrátit ke zdroji problému a znovu spustit proces. (42)

V oblasti bezpečnosti má Netflix tisíce procesů, které neustále zastavují a spouštějí instance, a pomocí lambd ověřují, že každá instance je sestavena a nakonfigurována v souladu s pravidly a předpisy systému. Lambda se také používá k vytváření výstrah a vypínání v případě neoprávněného přístupu. (42)

# 4 Vlastní práce

Na začátku každého projektu je nápad, který se postupně rozvíjí. V případě aplikace je po nápadu potřeba navrhnou funkcionality, které by mohl budoucí uživatel aplikace ocenit a které mohou ušetřit cenný čas uživatele. Když je základní koncepce hotová je možné na jejím základě začít navrhovat architekturu aplikace. Pro tuto práci byla vytvořena modelová aplikace pro fakturaci, vedení účetnictví a zjednodušení podání daňového přiznání. Cílovou skupinou této aplikace jsou malí a střední podnikatelé. Aplikace bude určena pro mobilní zařízení.

Hlavní funkce aplikace jsou:

- Vystavovaní faktur
- Zjednodušené účetnictví
- Automatické párovaní plateb a faktur
- Příprava daňového přiznání

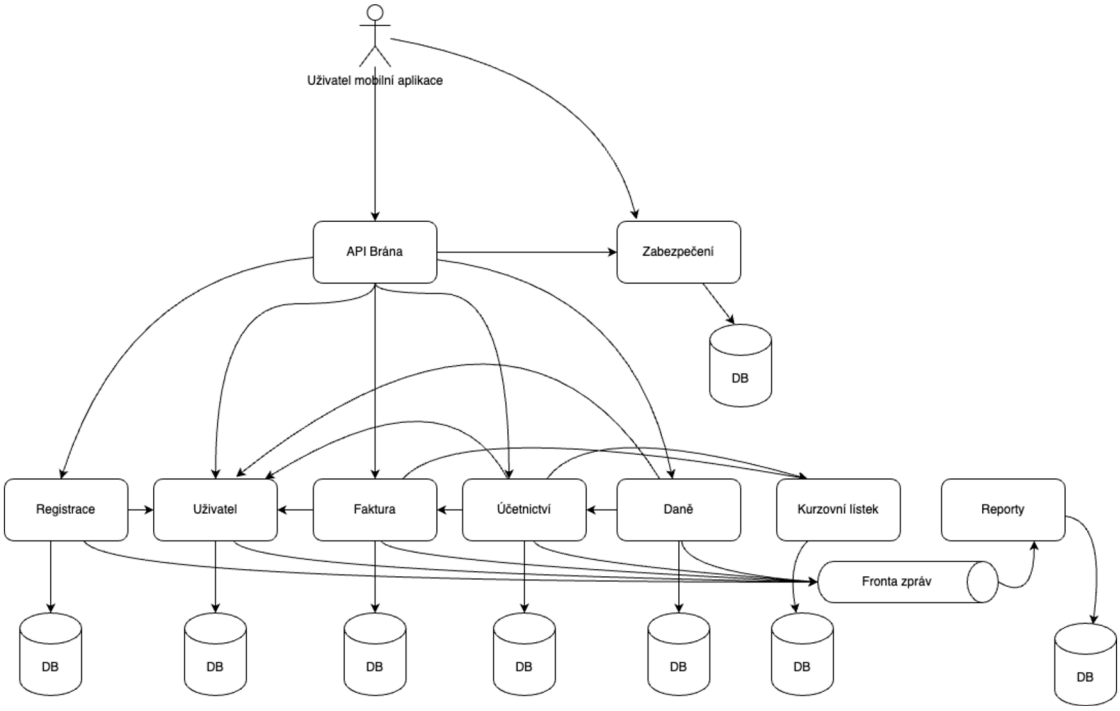
Aplikace by mohla oslovit až 1 milion uživatelů. Pro modelový příklad jsou připravena i data na základě kterých bude postavena architektura aplikace.

Akvizice nových uživatelů bude 8000 uživatelů za měsíc. Kdy pouze 20 % registrací je plně dokončených a uživatel se dostal do aplikace. To znamená 40000 zahájených registrací za měsíc. Aplikace má potenciál získat až 50 % trhu, což je 500 000 uživatelů. Tyto údaje je potřeba brát na zřetel při návrhu architektury, aby aplikace byla dostatečně robustní pro plánovaný počet uživatelů a zároveň provozně efektivní, aby náklady na provoz aplikace byly co nejnižší.

V praxi by vývoj modelové aplikace zahrnoval vývoj mobilní aplikace a serverové aplikace. Komunikace mezi nimi by probíhala skrze vystavené API. Pro provoz je serverové části je potřeba infrastruktura, na které aplikace poběží a s ní i spojené další služby. Cloudové aplikace využívají služeb poskytovatelů cloudových řešení, kteří nabízí vše potřebné pro provoz aplikace.

## 4.1 Návrh architektury modelové aplikace

Na základě technických požadavků a business analýzy je možné se pustit do tvorby diagramu architektury. Na obrázku 7 je schéma architektury na vysoké úrovni (High Level). Je výchozím bodem pro pochopení architektury systému a v průběhu projektu může být dále rozpracováno na konkrétnější komponenty a interakce. Diagram byl vytvořen v podobě architektury mikroslužeb, kde každá služba zastupuje konkrétní část produktu, který je cílem vývoje.




*Obrázek 7 High Level diagram architektury modelové aplikace*

Diagram architektury na vysoké úrovni poskytuje přehled komponent a interakcí systému a zobrazuje hlavní komponenty a jejich vztahy na vysoké úrovni abstrakce. Obvykle obsahuje hlavní součásti systému, jejich interakce a tok dat mezi nimi. Diagram se často používá jako komunikační nástroj, který pomáhá zúčastněným stranám pochopit celkovou strukturu a návrh systému. Lze jej také použít jako základ pro další podrobné návrhové a implementační práce.

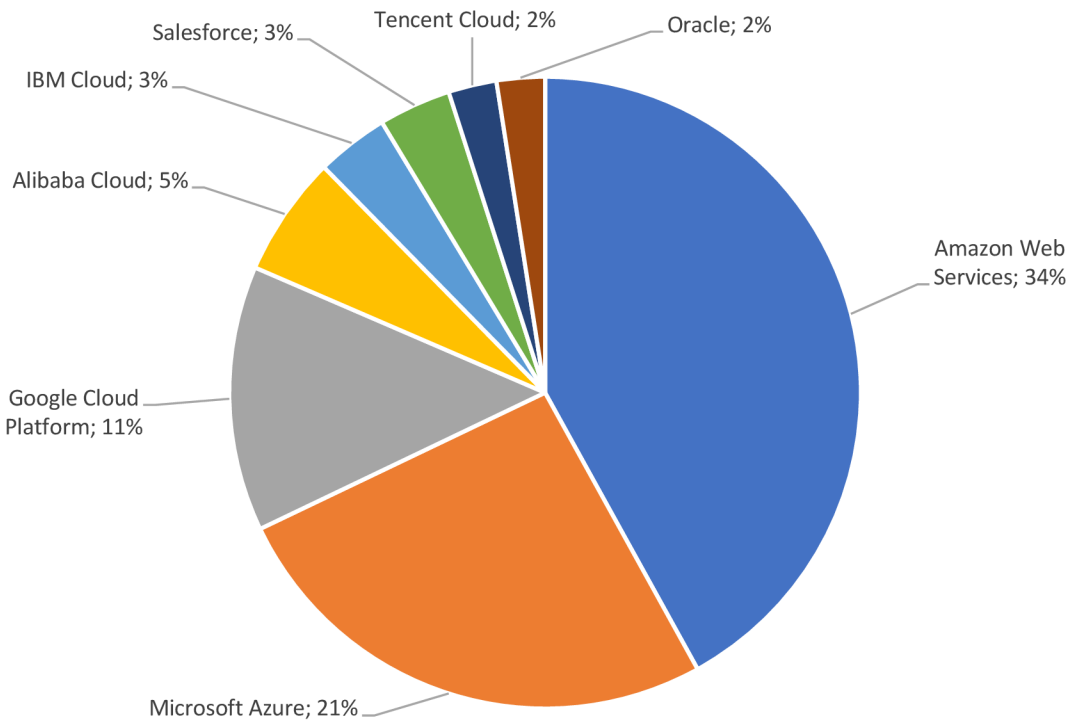
Z diagramu je možné vidět uživatelské rozhraní, kterým je mobilní aplikace, které komunikuje s backendem skrze bránu. Brána zorchestruje volání na mikroslužba. Každá mikroslužba má vlastní databázi a komunikuje s ostatními mikroslužbami.

Pro zabezpečení komunikace je mikroslužba „Zabezpečení“ se kterou mobilní aplikace komunikuje napřímo. Dále je na diagramu možné vidět frontu, která přijímá zprávy z mikroslužeb, které následně konzumuje mikroslužba „Reporty“.

## 4.2 Výběr poskytovatele cloudu

Nyní je na řadě výběr poskytovatele cloudových služeb, u kterého by byla modelová aplikace provozována. Místo toho budou tyto služby použity k testování a experimentům.

Při výběru poskytovatele cloudu pro provozování serverless řešení je důležité zvážit faktory, jako je cenový model poskytovatele, výkon a škálovatelnost, zabezpečení a dodržování předpisů a dostupnost příslušných nástrojů a služeb. Důležité je také zhodnotit dosavadní výsledky a pověst poskytovatele na trhu a úroveň podpory, kterou svým zákazníkům nabízí. Nejlepší poskytovatel cloudu pro provozování Serverless řešení bude nakonec záviset na konkrétních potřebách a požadavcích.




*Graf 1 Rozdělení trhu mezi hlavním poskytovateli cloudu Q3 2022 (6)*

Z grafu 1 je patrné, že největšími poskytovateli na trhu cloud computing jsou společnosti Amazon se značkou Amazon Web Services, Microsoft se značkou Azure a nakonec společnost Google s Google Cloud Platformou. Ostatní poskytovatelé jako IBM

nebo Oracle se řadí také mezi velké poskytovatel, ale jejich portfolio služeb je určené spíše pro korporátní sféru.

AWS (Amazon Web Services), GCP (Google Cloud Platform) a Azure (Microsoft Azure) se dají považovat rozhodně za tři nejoblíbenější platformy cloud computingu. Každá z těchto platforem nabízí řadu služeb a funkcí, ale existují některé zásadní rozdíly, které mohou ovlivnit rozhodnutí při výběru poskytovatele cloudových služeb.

Volba mezi AWS, GCP a Azure nakonec závisí na konkrétních potřebách a požadavcích. Před rozhodnutím se vyplatí zhodnotit ceny, funkce a možnosti podpory jednotlivých platforem.

Výběr bude postaven na několik kritériích, kterými jsou:

- Cena
- Nabídka služeb
- Regiony
- Komunita
- Podíl na trhu

### 4.2.1 Cena

Ceny cloudových služeb jsou často složité a mohou se lišit v závislosti na řadě faktorů, jako je využití, region, typy instancí, úložiště a další faktory. Každý poskytovatel cloudu má jiné cenové modely a nabízí různé služby s různou cenovou strukturou. Všichni poskytovatelé nabízí cenový model pay-as-you-go, kdy zákazníci platí pouze za to, co používají, a to za hodinu nebo za sekundu. Tento model je nejvíce vypovídající v obecné rovině. Kalkulátory cloudových poskytovatelů sice nabízí aplikace jiných cenových modelů, ale jejich nabídky se odvíjí od specifického užívání jejich služeb, což v modelovém případě není možné určit.

K obecnému cenovému porovnání bylo zvoleny ceny největší a nejmenší instance virtuálního stroje. Každý poskytovatel nabízí nejmenší instanci, které může ale i nemusí dostačovat pro běh dané aplikace, ale je to rozhodně startovní bod, kde se začíná jak s výkonem, tak i s cenou. Níže se dostat nejde. S rostoucími požadavky na zdroje je možné dosáhnout až na maximální možný typ instance, kdy naopak více zdrojů už poskytovatel nenabízí.

| Velikost instance | AWS | Azure | GCP |
|---|---|---|---|
| Nejmenší | "t4g.nano", který má 2 vCPU, 0,5 GB paměti a stojí 0,0033 USD za hodinu (43) | "B1s", který má 1 vCPU, 1 GB paměti a stojí 0,018 USD za hodinu (44) | "f1-micro", který má 1 vCPU, 0,6 GB paměti a stojí 0,0078 USD za hodinu (45) |

*Tabulka 1 Ceny instancí poskytovatelů cloudových služeb*

Z tabulky 1 byly stanoveny cenové intervaly, jejichž středová hodnota budou použita pro kritérium ceny ve vícekriteriální hodnocení variant.

AWS: 0,0033-11,952 střed=5,97435 USD

Azure: 0,018-30,773 střed=15,2965 USD

GCP: 0,0078-8,316 střed=4,1541 USD

### 4.2.2 Nabídka služeb

Amazon Web Services, Microsoft Azure i Google Cloud Platform nabízí přes 200 různých služeb v široké škále kategorií, včetně výpočetní techniky, úložišť, databází, analytiky, strojového učení, sítí, zabezpečení a dalších. Všechny společnost do svých platforem pravidelně přidává nové služby a funkce a také aktualizuje stávající služby s cílem zlepšit jejich funkčnost a výkon.

K porovnání byly zvoleny pouze služby cloud computingu, které jsou vhodné pro běh mikroslužeb a serverless funkcí. Tyto služby jsou v tabulce 2. Data byl získána z výpisu nabídky produktů AWS (46), Azure (47) a GCP (48).

| Služba | AWS | Azure | GCP |
|---|---|---|---|
| VM | EC2 | Azure Virtual Machine | Google Compute Engine |
| PaaS | Elastic Beanstalk | App Service | Google App Engine |
| Kontejnery | Elastic Container Service, Elastic Kubernetes Service, AWS Fargate, Elastic Container Registry | Azure Kubernetes Service, Azure Container Registry, Azure Container Instances | Google Kubernetes Engine, Container Registry, Cloud Run |
| FaaS | Lambda | Azure Functions | Cloud Functions |

*Tabulka 2 Nabídka služeb poskytovatelé cloudových služeb*

### 4.2.3 Regiony

Regiony jsou fyzická místa po celém světě, kde jsou umístěna datová centra pro hostování a provozování svých cloudových výpočetních služeb. Regiony jsou navzájem izolované a jsou navrženy tak, aby byly na sobě zcela nezávislé, což zákazníkům umožňuje provozovat aplikace a ukládat data v konkrétních regionech, které splňují jejich požadavky na dodržování předpisů a latenci.

Každý region se navíc skládá z několika zón dostupnosti, což jsou samostatná datová centra s vlastním napájením, sítí a konektivitou. Zóny dostupnosti v rámci jednoho regionu jsou propojeny linkami s nízkou latencí, což umožňuje vysoce dostupnou architekturu odolnou proti chybám.

AWS má 31 regionů (49), Azure 63 (50) a GCP 35 (51).

### 4.2.4 Komunita

Celkově lze říci, že komunita kolem technologie může být mocnou silou pro učení, podporu, inovace a vliv a může přispět k úspěchu technologie a jejích uživatelů. Je vždy dobrým znamením, když kolem dané technologie existuje silná a početná komunita, neboť je pak snazší čerpat a sdílet znalosti a v případě problémů najít možnosti řešení. V rámci této komunity mohou vznikat nové nápady, které posouvají možnosti technologie dále.

Jak početné jsou komunity kolem zvolených cloudových poskytovatelů bylo zjištěno na základě výsledků z dotazníkového šetření webu StackOverflow (49), který se řadí mezi přední weby pro vývojáře softwaru. Cílem průzkumu je shromáždit údaje o současném stavu odvětví vývoje softwaru, včetně trendů, platů, spokojenosti s prací, programovacích jazyků, nástrojů a technologií.

Průzkum je jedním z největších a nejobsáhlejších průzkumů mezi vývojáři softwaru, kterého se účastní desítky tisíc vývojářů z celého světa. Průzkum se obvykle provádí online a obsahuje řadu otázek týkajících se programovacích jazyků, vývojových postupů, platů, pracovních zkušeností a dalších.

První místo v kategorii nejpopulárnější cloudová platforma obsadilo AWS s 51,01 %, druhé místo Azure s 28,72 % a na třetím místě GCP s 26,81 %.

### 4.2.5 Výběr poskytovatele

Aby byl výběr co nejvíce objektivní bude použita Saatyho metoda pro vícekriteriální hodnocení variant. Pro zvolená kritéria byly stanoveny váhy a následně vypočteny výsledné váhy jednotlivých kritérií.

Z grafu 1 byla odečtena data o podílu na trhu jednotlivých zvolených poskytovatelů cloudových služeb. Společnost Amazon má podíl na trhu 34 %, Microsoft 21 % a Google 11 %. V tabulce 3 byly určeny hodnoty kritérií a vypočteny váhy. V tabulce 4 jsou stanoveny hodnoty těchto kritérií.

| **Kritéria** | **Cena** | **Nabídka služeb** | **Podíl na trhu** | **Komunita** | **Regiony** | **$G_i$** | **$v_i$** |
|---|---|---|---|---|---|---|---|
| Cena | 1,0000 | 2,0000 | 2,0000 | 2,0000 | 3,0000 | 1,8882 | 0,3347 |
| Nabídka služeb | 0,5000 | 1,0000 | 2,0000 | 2,0000 | 3,0000 | 1,4310 | 0,2537 |
| Podíl na trhu | 0,5000 | 0,5000 | 1,0000 | 2,0000 | 3,0000 | 1,0845 | 0,1923 |
| Komunita | 0,5000 | 0,5000 | 0,5000 | 1,0000 | 3,0000 | 0,8219 | 0,1457 |
| Regiony | 0,3333 | 0,3333 | 0,3333 | 0,3333 | 1,0000 | 0,4152 | 0,0736 |

*Tabulka 3 Saatyho metoda – určení vah*

| | **Cena** | **Nabídka služeb** | **Podíl na trhu** | **Komunita** | **Regiony** |
|---|---|---|---|---|---|
| Váhy | 0,3347 | 0,2537 | 0,0736 | 0,1457 | 0,1923 |
| AWS | 5,97 | 7 | 31 | 51,01 | 34 |
| Azure | 15,30 | 6 | 63 | 28,72 | 21 |
| GCP | 4,15 | 6 | 35 | 26,68 | 11 |
| | min | max | max | max | max |

*Tabulka 4 Výběr poskytovatele cloudu*

| **Výsledek** | |
|---|---|
| Amazon Web Services | **0,36582376** |
| Microsoft Azure | 0,28295839 |
| Google Cloud Platform | 0,35121784 |

*Tabulka 5 Výsledek výběru poskytovatel cloudu*

Z výsledků v tabulce 5 je zřejmé, že na základě kritérií je nevhodnějším poskytovatelem cloudových služeb Amazon Web Services.

## 4.3 Amazon Web Services

Pro nasazení aplikace byl zvolen poskytovatel cloudových řešení Amazon Web Services. K dispozici je hned několik technologií vhodných k hostování aplikace. V této kapitole se technologie pro hostování mikroslužeb a serverless analyzují.

Cílem analýzy je odhalit možnosti a limity těchto technologií, aby bylo možné vybrat co nevhodnější řešení pro hostovaní modelové aplikace. Pro hostování aplikace s architekturou mikroslužeb s využitím dalších služeb v AWS, je hned několik produktů. Jsou to Elastic Beanstalk, Elastic Container Service a Elastic Kubernetes Service. Ze serverless to jsou produkty AWS Lambda a AWS Fargate. (46)

### 4.3.1 AWS Elastic Beanstalk

AWS Elastic Beanstalk (53) je platforma jako služba (PaaS), která zjednodušuje nasazení a správu webových aplikací v cloudu AWS. Celkově lze říci, že Elastic Beanstalk abstrahuje základní infrastrukturu, což umožňuje soustředit se na samotný vývoj aplikace a její nasazení, zatímco AWS se stará o správu základní infrastruktury.

Elastic Beanstalk automaticky vytváří prostředí pro aplikaci, které zahrnuje zdroje potřebné pro běh aplikace, jako jsou webové servery, aplikační servery a databáze. Konfiguraci prostředí je možné na základě požadavků aplikace přizpůsobit, například počet instancí, typy instancí a load balancery. Škálování je zajištěno automaticky na základě vzorců provozu aplikace, tak aby aplikace zvládla zátěž.

AWS Elastic Beanstalk má různé kvóty nebo limity, které upravují využívání prostředků a služeb. Tyto kvóty se mohou lišit v závislosti na regionu AWS a typu prostředí Elastic Beanstalk. Pro potřeby modelové aplikace jsou kvóty natolik vysoké, že není riziko, že by došlo k problému nasazení. Například maximální počet aplikací v regionu je 75 a maximální počet verzí aplikace je 1000.

Tato služba bude použita pro příklad nasazení mikroslužby s JAR souborem.

### 4.3.2 AWS Elastic Container Service

AWS Elastic Container Service (ECS) (54) je plně spravovaná služba pro orchestraci kontejnerů, která umožňuje spouštět a spravovat kontejnery Docker v clusteru instancí

Amazon EC2. ECS funguje tak, že abstrahuje od základní infrastruktury a poskytuje škálovatelný a spolehlivý způsob nasazování, správy a škálování kontejnerových aplikací.

Při použití ECS se nejprve definuje aplikace pomocí definice úlohy. Definice úlohy určuje podrobnosti, například Docker obraz, který se má použít, kolik procesoru a paměti se má přidělit a jaké síťové porty se mají vystavit. Po definování úlohy je možné ji spustit jako úlohu v jedné instanci EC2 nebo jako službu v clusteru instancí.

Pokud bude úloha spuštěna jako služba, ECS automaticky řídí škálování a umístění úloh na základě požadovaného počtu úloh a kapacity clusteru. Například, pokud úloha selže nebo dojde k výpadku instance, ECS automaticky změní rozvržení úloh na ostatních instancích v clusteru.

Služba AWS Elastic Container Service (ECS) má určité kvóty a limity, které jsou zavedeny za účelem zajištění stability a výkonu služby. Tyto kvóty a limity se mohou lišit v závislosti na regionu AWS a typu používaných prostředků. Pro potřeby modelové aplikace jsou kvóty natolik vysoké, že není riziko, že by došlo k problému nasazení. Například počet kontejnerů na cluster je 5000 a počet kontejnerů na definici úlohy je 10.

ECS bude použito k orchestraci a nasazení mikroslužeb přes Docker obrazy.

### 4.3.3 AWS Fargate

AWS Fargate (27) je serverless služba pro kontejnery, který usnadňuje provozování kontejnerů na Amazon Web Services bez nutnosti spravovat základní infrastrukturu. S AWS Fargate mohou uživatelé spouštět kontejnery ve škálovatelném, vysoce dostupném a bezpečném prostředí, aniž by museli zajišťovat, spravovat nebo škálovat clustery virtuálních strojů.

Spravuje základní infrastrukturu potřebnou ke spuštění kontejnerů a automaticky škáluje infrastrukturu podle poptávky. Uživatelé se tak mohou soustředit na vývoj svých aplikací, a ne na správu infrastruktury. Služba Fargate je integrována s dalšími službami AWS, jako je Elastic Container Service (ECS) a Elastic Kubernetes Service (EKS), a poskytuje tak bezproblémové nasazení kontejnerů.

Podporovány jsou kontejnery pro Linux i Windows a uživatelé platí pouze za zdroje, které jejich kontejnery spotřebují, což z něj činí nákladově efektivní řešení pro provoz kontejnerů v cloudu. Z technologií jsou podporovány Docker kontejnery, ale může nastat

situace, že některé obrazy kontejnerů nebudou kompatibilní pro nasazení. Omezení či kvóty pro AWS Fargate přímo nejsou, jsou nastavené orchestračním nástrojem, jako je Elastic Container Service (ECS) a Elastic Kubernetes Service (EKS)

Tato služba bude použita pro serverless nasazení Docker obrazů mikroslužeb do ECS.

### 4.3.4 AWS Lambda

AWS Lambda je serverless výpočetní služba, která uživatelům umožňuje spouštět kód bez nutnosti zajišťovat nebo spravovat infrastrukturu. Funguje tak, že spouští kód v reakci na události a automaticky spravuje výpočetní zdroje, které kód vyžaduje.

Když uživatel vytvoří funkci Lambda, nahraje svůj kód a zadá vstupní parametry funkce. Služba Lambda pak vytvoří prostředí pro spuštění kódu, přidělí potřebné prostředky a funkci provede. Služba škáluje zdroje nahoru nebo dolů v závislosti na počtu příchozích požadavků, takže uživatelé se nemusí starat o poskytování serverů nebo správu infrastruktury.

Lambdu lze použít ke spuštění kódu v reakci na nejrůznější události, jako jsou změny dat uložených v Amazon S3 nebo příchozí požadavky API. Uživatelé mohou spouštět funkce Lambda také pomocí dalších služeb AWS, jako jsou Amazon SNS, Amazon SQS a AWS CloudFormation.

Jelikož už o samotné spuštění kódu se stará služba AWS Lambda, přichází s tím řada omezení, které je potřeba vzít na vědomí při navrhování a nasazování serverless funkcí. Maximální doba provádění funkcí je 15 minut. Pokud se funkce vykoná déle, bude ukončena a bude zapotřebí upravit kód tak, aby byl efektivnější, nebo jej rozdělit na menší funkce. Při vytváření funkcí je přiděleno určité množství paměti, které nelze během provádění zvětšovat. Maximální velikost paměti pro funkci Lambda je v současné době 3 GB. Funkce mají přístup k malému množství místního úložiště (512 MB), které se po každém vyvolání odstraní. Pokud funkce potřebuje ukládat data mezi jednotlivými voláními, je pro tyto účely potřeba použít externí datové úložiště. Při prvním volání nové funkce nebo po určité době nečinnosti může dojít ke zpoždění k takzvanému studenému startu (cold start). Důvodem je, že podkladová infrastruktura zajišťuje pro funkci nový kontejner. To může mít za následek zvýšenou latenci při několika prvních voláních funkce. Velikost balíčku funkce Lambda

(včetně případných závislostí) je omezena na 50 MB při přímém nahrávání nebo na 250 MB při nahrávání přes Amazon S3. Funkce mohou přistupovat pouze k omezené sadě proměnných prostředí (do 4 KB), což nemusí být pro některé případy použití dostatečné. AWS Lambda má výchozí limit souběžnosti 1000 spuštění na region a pro jednotlivé funkce je výchozí nastavení 10 spuštění na funkci. Naráz může fungovat maximálně 100 funkcí dohromady při výchozí limitech.

Na Lambdě bude vyzkoušeno nasazení čistě kódu funkce.

## 4.4 Cenová analýza

Amazon Web Services nabízí širokou škálu služeb cloud computingu s flexibilními cenovými možnostmi, které vyhovují různým potřebám a rozpočtům. Základním principem cenové politiky AWS je zákazník platí pouze za ty zdroje, které spotřebuje. Nic se v tomto případě neplatí předem ani není nutné se k ničemu dlouhodobě zavazovat. Ceny se ovšem liší podle služby. Každá AWS služba má svou cenovou strukturu a ceny se mohou lišit v závislosti na faktorech, jako je množství použitého úložiště nebo přenosu dat, počet zpracovaných požadavků nebo požadovaná úroveň výkonu.

AWS nabízí několik cenových modelů, včetně cen na vyžádání, rezervovaných instancí a spotových instancí, které zákazníkům umožňují vybrat si cenový model, který nejlépe vyhovuje jejich potřebám. Také se ceny mohou lišit podle regionu, přičemž v různých geografických lokalitách se ceny za stejné služby liší. AWS poskytuje pro mnoho svých služeb bezplatné úrovně použití, které zákazníkům umožňují vyzkoušet a experimentovat se službami AWS bez jakýchkoli nákladů. Některé služby AWS mohou zahrnovat také další poplatky za funkce, jako je přenos dat, zálohování úložiště nebo technická podpora. Na nic z toho nebude brán zřetel v rámci cenové analýzy.

Ceny jednotlivých služeb budou získány z cenového kalkulátoru AWS. Je zapotřebí stanovit parametry pro všechny služby společné, aby výsledná data měla stejný základ. Zároveň parametrů, které je možné zadat je velké množství, takže druhotným cílem je počet parametrů, které se budou volit srazit na menší počet. Proto ty zbylé, budou ponechány ve výchozím nastavení nebo bude zvolena vlastní výchozí hodnota pro celou analýzu.

Zvolené výchozí hodnoty pro povinné parametry:

- Region Evropa Frankfurt (eu-central-1)
- Výpočet bez jednotek zdarma
- Architektura Arm
- Operační systém Linux
- Velikost uložiště bude ponecháno výchozí minimální hodnota
- Typ nájmu instancí sdílené
- Instance na vyžádání (On-Demand)
- Konstantní využití instance
- Provisioned concurrency, pouze během pracovních hodin. Vykoná až 80 % požadavků.

### 4.4.1 AWS Elastic Beanstalk

Služba Elastic Beanstalk je bezplatná. Zpoplatněny jsou však služby, které jsou použity pro běh aplikace jako virtuální stroje, databáze, firewall, API a jiné. Pro běh aplikace jsou využívány instance EC2 (Elastic Container).

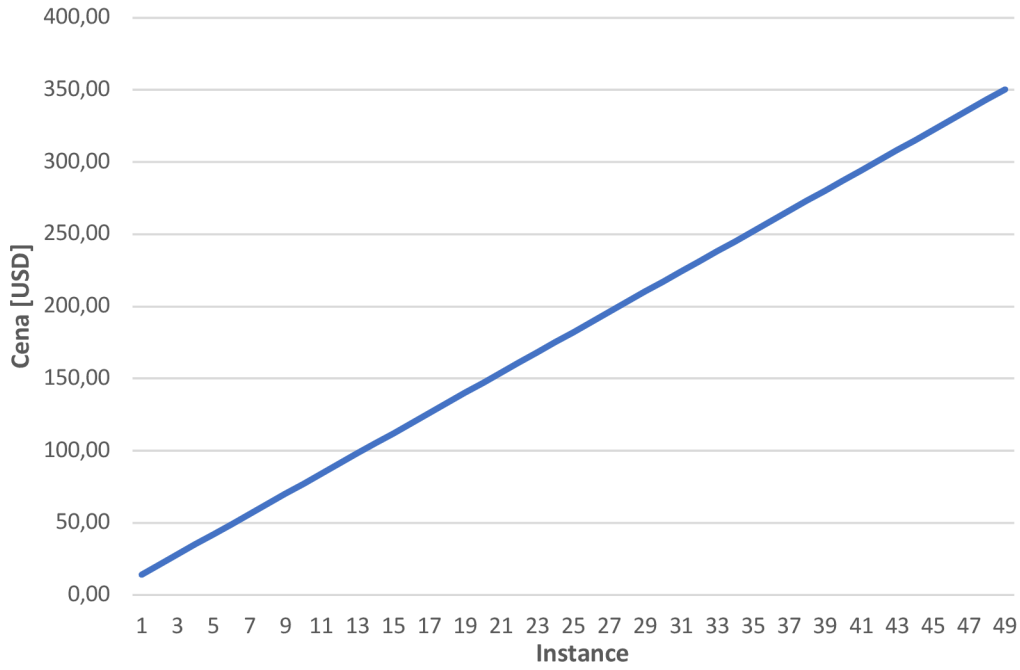
Parametry pro výběr instance byly zvoleny zmíněné výchozí a další byly zvoleny tyto:

- Počet vCPU 1
- Velikost alokované paměti 1 GB

Po vložení parametrů do AWS kalkulátoru byla doporučená instance typu t4g.micro (2vCPU a 1 GB alokované paměti). Výsledná cena byla stanovena vzorcem:

$$< Počet\ instancí > \times\ 0{,}0096\ \times\ 730 = Cena\ za\ měsíc \qquad (1)$$

Cena instance t4g.micro je 0,0096 USD za hodinu a AWS stanovil počet hodin za měsíc na 730.




*Graf 2 AWC EC2 cena za instanci*

Cena jedné instance EC2 za měsíc je 7,01 USD. Na grafu 2 je vidět lineárně zvyšující se cena se stoupajícím počtem instancí.

### 4.4.2 AWS Elastic Container Service

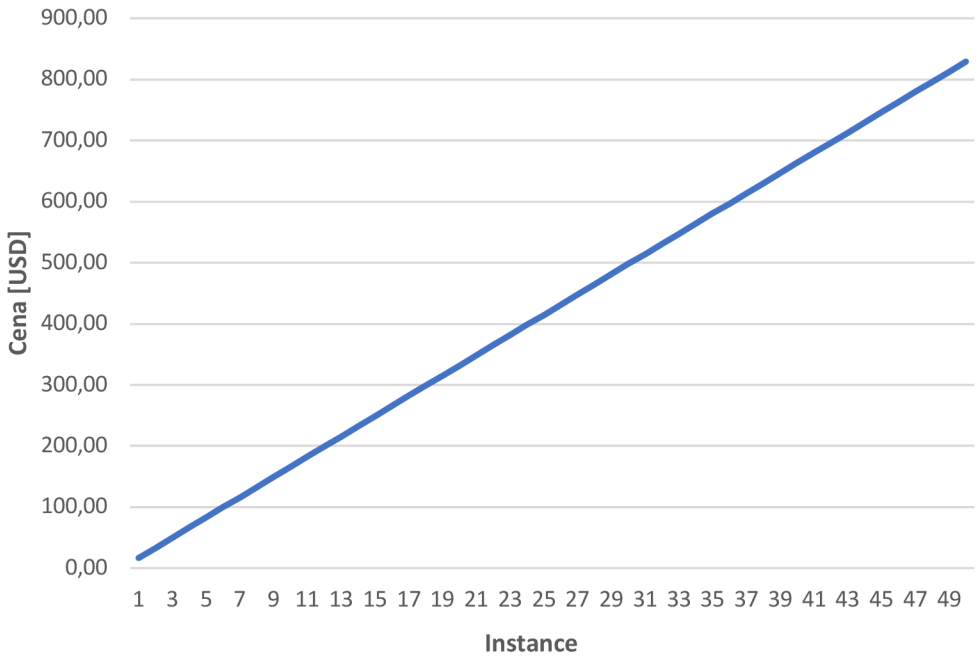
Služba Elastic Container Service je bezplatná. Platí se za spotřebované zdroje. Stejně jako u služby AWS Elastic Beanstalk se platí za EC2 instanci. V případě použití Docker kontejnerů, kdy některé mikroslužby nevyužijí plně zdroje EC2 instance je možné spustit na jedné instanci více kontejnerů a tím maximálně využít dostupně zdroje a snížit náklady.

V případě ceny za instanci EC2, platí stejné podmínky jako v předchozí kapitole. Je možné si vybrat z různých typů a velikostí instancí EC2, které mají různé cenové možnosti na základě faktorů, jako je výkon procesoru, paměti a sítě.

Pro potřeby modelové aplikace by byl použit stejný typ instance t4g.micro. Cena za potřebných 9 instancí je 63,07 USD za měsíc. Pro hladké fungování clusteru je potřeba zprovoznit i další služby jako load balancer s čímž vzrostou náklady. Jelikož ale ECS s EC2 umožňuje si vše nakonfigurovat dle potřeby. Je možné správnou konfigurací dosáhnout snížení nákladů. Je však zapotřebí znalost AWS a DevOps problematiky což může mít za následek zvýšení nákladů u lidských zdrojů, z důvodů vyšší kvalifikace nebo lidí odpovědných za infrastrukturu.

### 4.4.3 AWS Fargate

Ceny služby Fargate jsou založeny na množství vCPU a paměťových zdrojů, které jsou přiděleny kontejnerům běžícím ve službě. Platí se pouze za prostředky, které se spotřebují, bez počátečních nákladů nebo dlouhodobých závazků. S Fargate lze snadno škálovat zdroje kontejnerů nahoru nebo dolů podle potřeby, což umožňuje optimalizovat náklady.




*Graf 3 AWS Fargate cena za běh úlohy*

Na grafu 3 je vidět cena za daný počet běžících úloh. Pro výpočet byly použity definované výchozí parametry a pro vCPU byla zvolena hodnota 0,5CPU a pro RAM hodnota 1 GB. Ostatní parametry byly ponechány na výchozí hodnotách.

Hodnoty v grafu 3 byly vypočítány tímto způsobem:

$$< počet\ úloh > \times\ (730 \div 24) = počet\ úloh\ za\ měsíc \qquad (2)$$

$$< počet\ úloh\ za\ měsíc > \times\ 0{,}5\ \times\ 24 \times 0.03725 = cena\ za\ vCPU \qquad (3)$$

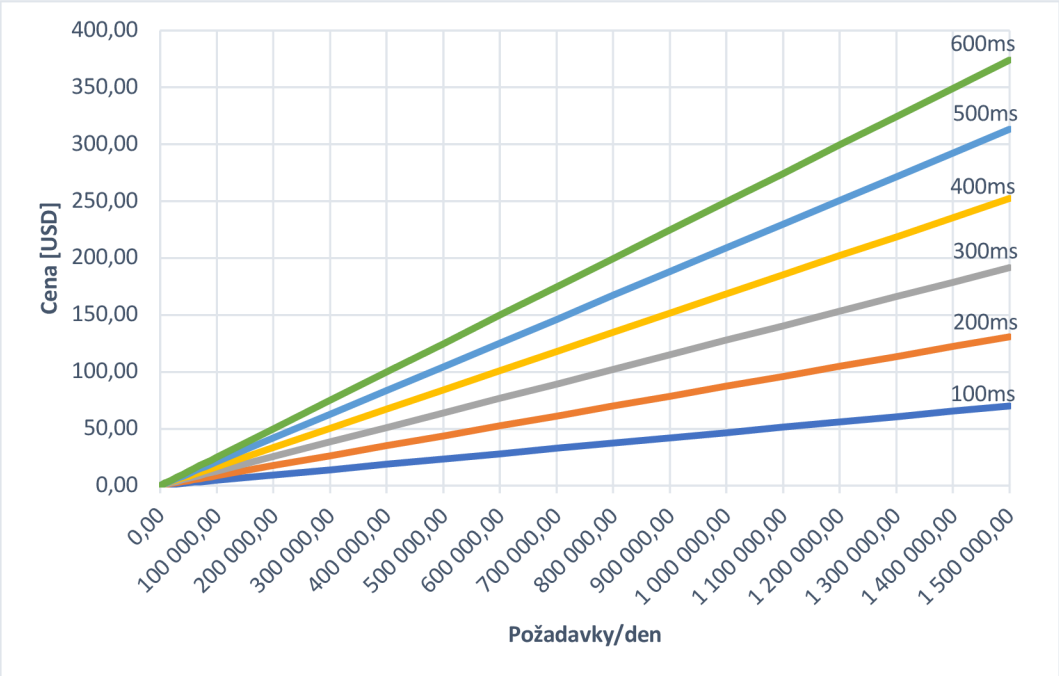
$$< počet\ úloh\ měsíc > \times\ 1\ \times\ 24\ \times 0{,}00409 = cena\ za\ RAM \qquad (4)$$

$$< cena\ za\ vCPU > + < cena\ za\ RAM > = celková\ cena\ za\ měsíc \quad (5)$$

Je však důležité zmínit, že to je konečná cena. Není zapotřebí platit další služby potřebné k běhu infrastruktury. O samotnou infrastrukturu se stará AWS tudíž není zapotřebí hlubší znalostí ani vynakládat větší výdaje na DevOps. Cena za jednu instanci je 16,58 USD.

### 4.4.4 AWS Lambda

Cena za službu AWS Lambda je založena na počtu požadavků na funkci a na množství výpočetního času použitého ke spuštění funkce. Pro výpočet ceny je potřeba určit čas spuštění funkce a velikost použité paměti. Počet vCPU se nezadává je stanoven na základě velikosti paměti. RAM byla nastavena na 1 GB. Ostatní parametry byly ponechány na výchozí hodnotách.




*Graf 4 AWS Lambda cena za požadavky*

Hodnoty v grafu 4 byly vypočítány tímto způsobem:

$$< počet\ požadavků > \times < délka\ spuštění > \times\ 0{,}001\ = celková\ doba\ spuštění \quad (6)$$

$$< velikost\ RAM > \times < celková\ doba\ spuštění\ > = celkvová\ doba\ využití\ RAM \quad (7)$$

$$< celkvová\ doba\ využití\ RAM > \times 0{,}0000166667 = cena\ za\ spuštění \quad (8)$$

$$< počet\ požadavků > \times 0{,}0000002 = cena\ za\ požadavky \quad (9)$$

$$< cena\ za\ spuštění > + < cena\ za\ požadavky > = celková\ cena\ za\ měsíc \quad (10)$$

Je důležité zmínit, že to je konečná cena. Není zapotřebí platit další služby potřebné k běhu infrastruktury. O samotnou infrastrukturu se stará AWS tudíž není zapotřebí hlubší znalostí ani vynakládat větší výdaje na DevOps. Pouze v případě funkce provisioned concurrency se k celkové ceně přičte poplatek vypočtený z následujícího vzorce.

$$< počet\ concurrency > \times < délka\ spuštění\ v\ concurrency > \times < RAM\ v\ concurrency > \times 0{,}0000033334 = poplatek\ za\ provisined\ concurrency \quad (11)$$

Měsíční cena za 100 000 požadavků denně při délce spuštění 200ms je 8,72 USD. Na grafu 4 je vidět, že při vyšší zátěži je cena velmi vysoká.

## 4.5 Zátěžový test

Zátěžový test je typ nefunkčního testu, který hodnotí, jak systém funguje při určitém pracovním zatížení nebo úrovni použití. Účelem zátěžového testu je určit maximální množství provozu nebo uživatelů, které systém zvládne, než začne docházet k poklesu výkonu nebo selhání.

Při zátěžovém testu se generuje simulovaná zátěž, která se aplikuje na testovaný systém. Tuto zátěž lze generovat pomocí různých nástrojů nebo technik, například simulací velkého počtu současně pracujících uživatelů nebo generováním velkého objemu požadavků. Odezva systému na tuto zátěž je průběžně měřena a analyzována, aby se zjistilo, jak dobře systém funguje při zátěži.

Testování zátěže je důležitou součástí zajištění toho, aby systém zvládl očekávaný objem provozu nebo používání, a může pomoci identifikovat úzká místa výkonu nebo jiné problémy, které mohou ovlivnit výkon systému. Často se používá v souvislosti s webovými

aplikacemi nebo jinými distribuovanými systémy, ale lze jej použít pro jakýkoli systém, u kterého se očekává, že bude zvládat značné množství zátěže.

### 4.5.1 Příprava testu

Pro test byla připravena jednoduchá aplikace simulující výkon aplikační logiky, jejíž provedení trvá 250ms. Jedná se o obecný průměr pokrývající čtení a zápis do databáze a provedení výpočtu.

Pro vytvoření testovacích aplikací byly vybrány tyto technologie:

- Programovací jazyk Kotlin verze 1.7.22
- Java verze 17
- Frameworku Spring Boot verze 3.0.4
- Frameworku Spring Cloud Function verze 4.0.1
- REST API

Byla vytvořena jednoduchá mikroslužba ve frameworku Spring Boot. Služba má v sobě dvě komponenty, a to restový kontrolér pro onboarding a služba s aplikační logikou pro onboarding.

```
@RestController
class OnboardingController(
    private val onboardingService: OnboardingService
) {

    @PostMapping("/register")
    fun register(
        @RequestBody request: Registration
    ): ResponseEntity<UUID> {
        val userId = onboardingService.register(request)
        return ResponseEntity.ok(userId)
    }

}

data class Registration(
    val name: String,
    val email: String,
    val password: String
)
```

*Obrázek 8 Zdrojový kód rest kontroléru*

Na obrázku 8 je zdrojový kód rest kontroléru s jedním koncovým bodem. Definována třída je anotovaná *@RestController* anotací, díky které se o veškerá příchozí volání postará framework Spring Boot. V třídě je deklarovaná funkce *register* s anotací *@PostMapping*. Do této anotace se zadá URI, které pak při spuštění aplikace framework vystaví ven. Z názvu anotace také vyplývá že použitá HTTP metoda je POST.

```
@Service
class OnboardingService(
    @Value("\${delay}") private val delayMillis: Long
) {

    fun register(registration: Registration): UUID {
        val userId = UUID.randomUUID()
        Thread.sleep(delayMillis)
        return userId
    }

}
```

*Obrázek 9 Zdrojový kód služby*

Na obrázku 9 je definice třídy *OnboardingService* s deklarací metody *register*. Doba, po kterou se vykonává aplikační logika funkce může být různá. V rámci jednoduché implementace, která umožní simulace různých případů, byl použita funkce *Thread.sleep()*. Ta po stanovenou dobu uspí vlákno, které vykonává dané volání funkce. Hodnota času, po kterou má být vlákno uspáno je v milisekundách a je nastavena v externí konfiguraci na obrázku 10. Konfigurace je uložena v souboru application.yaml.

```
server:
  port: 8080
spring:
  main:
    banner-mode: off
  application:
    name: onboarding

delay: 250
```

*Obrázek 10 Externí konfigurace mikroslužby*

Hodnotu je možné změnit přepsáním s pomocí proměnné prostředí z anglického environment variable. Díky této konfiguraci není nutné pro každou změnu hodnoty času vytvářet novou verzi aplikace. Stačí pouze nastavit proměnou prostředí v konfiguraci služby ve které bude aplikace nasazena.

Definice proměnné prostředí pro nastavení hodnoty času pro funkci *Thread.sleep()*

DELAY=250

Zdrojový kód, který bude spuštěn službou AWS Lambda bude prakticky stejný. Jediné, co bude jiné je rozhraní přes, které bude aplikace přijímat požadavky ke zpracování. Aplikace se bude chovat stejně, bude mít vystavený koncový bod, který bude možné volat. Framework Spring Cloud Function umožňuje vystavit více něž jeden koncový bod, ale služba AWS Lambda umožňuje vystavit do sítě pouze jeden koncový bod. Jsou způsoby, jak tento limit obejít, ale tím se tato práce nebude zabývat.

```
@SpringBootApplication
class CloudFunctionApplication(
    private val service: OnboardingService
) {

    @Bean
    open fun register(): (Registration) -> String {
        return {
            service.register(it).toString()
        }
    }
}

data class Registration(
    val name: String,
    val email: String,
    val password: String
)


fun main(args: Array<String>) {
    runApplication<CloudFunctionApplication>(*args)
}
```

*Obrázek 11 Zdrojový kód funkce*

Na obrázku 11, kde je zdrojový kód funkce, je možné vidět definici třídy aplikace s deklarovanou funkcí *register*, kde návratovým typem je funkce. Tato návratová funkce má jako vstupní parametr datovou třídu *Registration* a jako návratový typ textový řetězec. V těle funkce *register* je deklarován lambda výraz, který se má vykonat.

Framework Spring Cloud Function se při startu aplikace pokusí najít funkce, které mají být vystaveny ven. Pro případ, že by mohlo dojít k chybě nebo by mohla být vystavena funkce, které neměla být je vhodnější v externí konfiguraci nastavit název funkce, která má být vystavena jako koncový bod.

```
spring:
  main:
    banner-mode: off
  application:
    name: onboarding
  cloud:
    function:
      definition: register

delay: 250
```

*Obrázek 12 Externí konfigurace funkce*

Na obrázku 12 je konfigurace v souboru application.yml. Je zde nastavená proměnná *spring.cloud.function.definition,* do které je vložena hodnota *register*, což je název funkce, která má být zavolána. Tímto je funkce připravena a aplikace se po spuštění bude chovat stejně jako kdyby měla restový kontrolér. Zbytek zdrojového kódu je stejný.

### 4.5.2 Test

Lambda je produkt, u kterého přechází veškerá odpovědnost za infrastrukturu na poskytovatele cloudových služeb. S tím ale přichází i některá omezení. Z dokumentace vyplývá, že ve výchozím nastavení jedna funkce může být škálována až na 10 paralelně běžících procesů. Každý proces se může vykonávat různě dlouhou dobu. Čím je doba vykonávání delší, tím méně požadavků je při výchozím nastavení Lambda schopna odbavit. Cílem je pomoci zátěžového testu získat ucelený přehled, jak se Lambda chová pod zvýšenou zátěží a kolik požadavků v průměru je schopna odbavit.

K provedení samotného testu bude použit testovací nástroj Artillery. Jedná se o open-source nástroj pro testování zátěže, který se používá k simulaci velkého objemu provozu pro testování výkonu a škálovatelnosti webových aplikací, rozhraní API a dalších systémů. Umožňuje definovat a provádět komplexní testovací scénáře, které simulují reálné chování uživatelů, a poskytuje podrobné metriky a reporty, které pomáhají identifikovat úzká místa výkonu a další problémy.

Na obrázku 13 je ukázaná konfigurace testu. Test je rozdělen do tří částí:

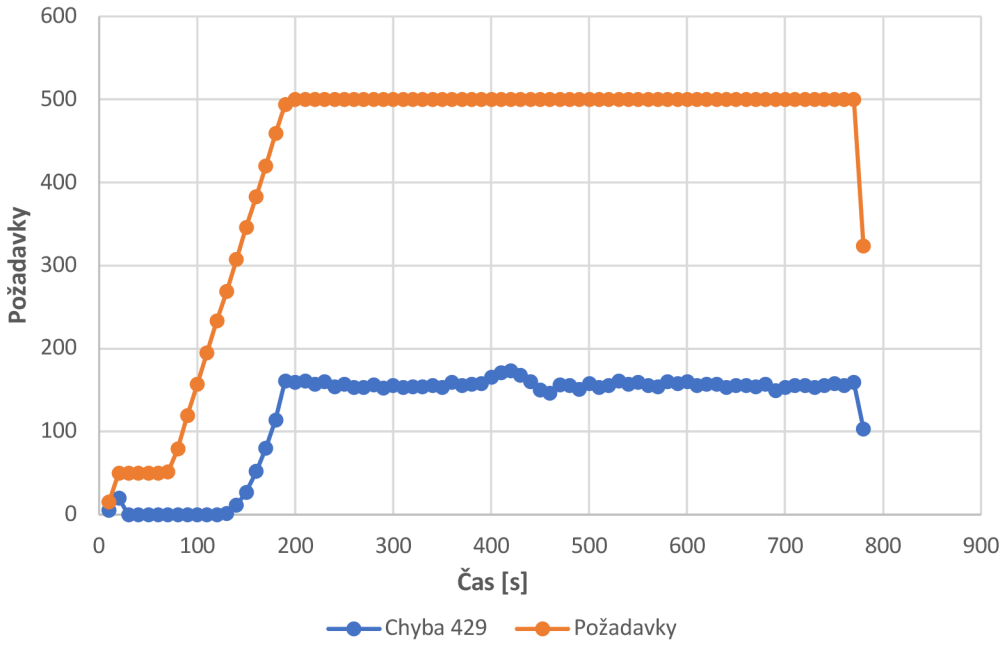
1. Zahřívání – kde se posílá 5 požadavků za vteřinu. Tato část trvá 1 minutu.
2. Náběh – požadavky začínají na 5 za vteřinu, ale v průběhu 2 minut se počet požadavků zvadne až na 50 za vteřinu.

3. Trvalá zátěž – Tato část trvá 10 minut a po celou dobu se posílá 50 požadavků za vteřinu.

```
config:
  target: "<URL>"
  phases:
    - duration: 60
      arrivalRate: 5
      name: Warm up
    - duration: 120
      arrivalRate: 5
      rampTo: 50
      name: Ramp up load
    - duration: 600
      arrivalRate: 50
      name: Sustained load
scenarios:
  - name: "Register"
    flow:
      - post:
          url: "/register"
          json:
            name: "{{ $randomString() }}"
            email: "{{ $randomString() }}"
            password: "{{ $randomString() }}"
```

*Obrázek 13 Konfigurace zátěžového testu*

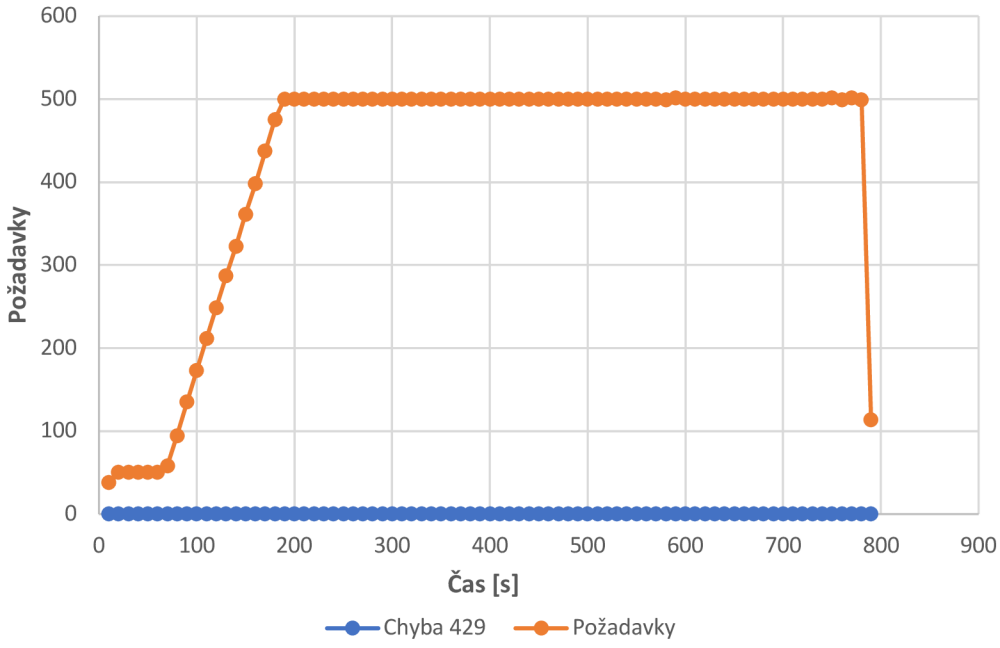
Byly provedeny 3 testy, každý na jiném prostředí.




*Graf 5 AWS Lambda zátěžový test*

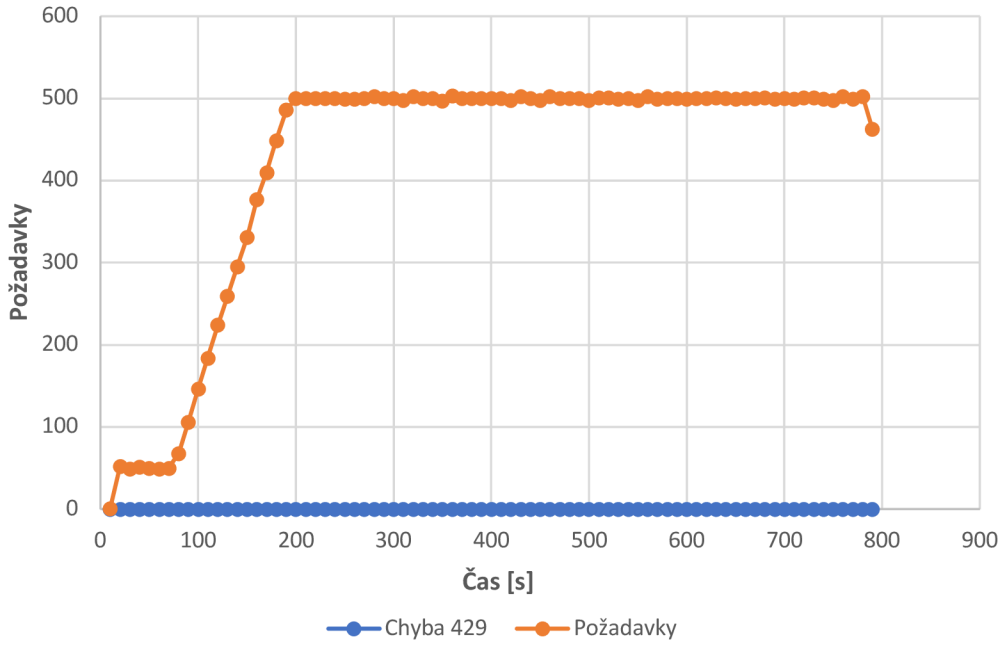
Výsledek zátěžového testu pro AWS Lambda je vidět na grafu 5. V Zahřívací části testu, kdy Lambda ještě nebyla nastartovaná, je možné vidět výskyt několik chyb. Kdy vlivem takzvaného studeného startu, tedy neběžící funkce došlo k zahození příchozích požadavků. Po spuštění funkce je počet chyb na nule. S rostoucím počtem požadavků stoupá i počet chyb a tím i zahozených požadavků. Každý bod v grafu znamená počet požadavků v 10vteřinovém intervalu. Od 269 požadavků, počet chyb narůstá a při 500 požadavcích, každých 10 vteřin, je počet chyb v rozmezí 150-160.

Z testu vyplývá, že při výchozím nastavení Lambda v AWS, kdy funkce může zpracovávat paralelně maximálně 10 požadavků a čas na vykonaní funkce je 250ms, je možné zpracovat při zachování spolehlivosti 34 požadavků za vteřinu. V případě delšího času zpracování bude počet možných požadavků za vteřinu nižší a naopak.




*Graf 6 AWS Fargate zátěžový test*

Na grafu 6 je výsledek zátěžového testu pro AWS Fargate. Za celou dobu běhu testu nedošlo k žádná chybě. Aplikace odbavila veškeré požadavky velmi rychle. Nedošlo ani k žádné vetší časové prodlevě mezi požadavky.




*Graf 7 AWS EC2 zátěžový test*

Na grafu 7 je výsledek zátěžového testu pro AWS EC2. Pro připomenutí EC2 je instance virtuálního stroje, které využívá jak služba Elastic Beanstalk, tak Elastic Container Service s EC2. Během testu nedošlo k žádná chybě, ale došlo k prodlevám během zpracování jednotlivých požadavků. V grafu je vidět že při zasílání 50 požadavků za vteřinu nastaly situace, kdy nebyly všechny požadavky odbaveny během vteřiny.

## 4.6 Specifické případy pro komparaci

Specifické případy užití budou postaveny na modelové aplikaci z kapitoly 4.1. Pro připomenutí se jedná o aplikaci, která má usnadnit pro malým a středním podnikatelům fakturaci a účetnictví. Architektura aplikace je postavena na mikroslužbách, které samy o sobě jsou skvělým způsobem, jak zlepšit výkon v případě zátěže pouze části aplikace. Záleží ovšem i na použitých technologiích, na kterých jsou aplikace nasazeny, aby bylo dosaženo co nejlepšího poměru cena výkon.

V následujících podkapitolách budou mikroslužby a serverless služby ve specifických situacích porovnány na základě stanovených kritérií, které co nejlépe umožní rozpoznat jaký způsob vývoje a nasazení aplikace je pro daný případ nejlepší.

Zvolená kritéria jsou:

- **Cena** – Cena je nejdůležitějším kritériem. Aby aplikace byla rentabilní je vždy potřeba udržet výdaje provoz na únosné úrovni nebo nejlépe se je pokusit ještě snížit.
- **Potřebná úroveň znalostí** – AWS služby vyžadují různé úrovně znalost, které jsou potřebné k jejich spuštění a následné konfiguraci. Tím mohou vzrůst náklady z důvodu zapojení dalších osob s potřebnou znalostí nebo získání potřebné kvalifikace.
- **Výkon** – Aplikace musí splňovat požadavky na výkon, aby zkušenost uživatelů s používáním aplikace byla co nejlepší.
- **Škálování** – V případě mikroslužeb se jedná o velmi důležité kritérium. V případě zvýšené zátěže je nezbytné aplikaci škálovat v místě kde je zátěž vysoká, aby nedošlo k přetížení a tím i nedostupnosti služeb. Některé služby provádí škálování automaticky s jednoduchou konfigurací, jiné naopak potřebují manuální přípravu skupiny instancí virtuálních strojů pro škálování.
- **Komplexita** – Provoz aplikace nebo i jednotlivých služeb může dosahovat různých úrovní složitosti. Kdy je pro vývoj, nasazení a provoz potřeba více znalostí různých technologií.
- **Podpora platformy** – Serverless služby jako produkt druhé strany obvykle nemohou hned nabídnout poslední verze některých technologií. Vývoj aplikace pak může narazit na nedostupnost potřebné technologie.

Váhy kritérií byly spočítány Saatyho metodou párového porovnání. Výpočet je v tabulce 6 a výsledné váhy v tabulce 7.

| Kritéria | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora | $G_i$ |
|---|---|---|---|---|---|---|---|
| Cena | 1,0000 | 2,0000 | 3,0000 | 4,0000 | 5,0000 | 7,0000 | 3,0717 |
| Další náklady | 0,5000 | 1,0000 | 2,0000 | 3,0000 | 4,0000 | 6,0000 | 2,0396 |
| Výkon | 0,3333 | 0,5000 | 1,0000 | 2,0000 | 3,0000 | 5,0000 | 1,3077 |
| Škálování | 0,2500 | 0,3333 | 0,5000 | 1,0000 | 3,0000 | 5,0000 | 0,9247 |
| Komplexita | 0,2000 | 0,2500 | 0,3333 | 0,3333 | 1,0000 | 3,0000 | 0,5054 |
| Podpora | 0,1429 | 0,1667 | 0,2000 | 0,2000 | 0,3333 | 1,0000 | 0,2612 |

*Tabulka 6 Výpočet vah kritérií*

| | Cena | Další náklady | Výkon | Škálování | Komplexita | Podpora |
|---|---|---|---|---|---|---|
| Váhy | 0,3787 | 0,2515 | 0,1612 | 0,1140 | 0,0623 | 0,0322 |

*Tabulka 7 Výsledné váhy kritérií*

### 4.6.1 Onboarding

Základem většiny aplikací je registrace uživatelů. Jedná se o část aplikace, kterou každý uživatel využije pouze jednou, aby se do aplikace dostal a mohl začít využívat nově získaných služeb. Z technického hlediska je na onboarding vyvíjena mnohem menší zátěž než na zbytek aplikace. V rámci modelové aplikace je odhadovaný počet zahájených registrací tady zaslaných požadavků blížící se hodnotě až 40000 požadavků za měsíc.

V reálném provozu je zátěž dosti nevyvážená. Marketingové akce mohou mít za následek zvýšený zájem o registraci a tím i zvýšenou zátěž na aplikaci. Ale i opačně může dojít k opadnutí zájmu a aplikace pro onboarding nemusí být zatížena vůbec.

Pro modelovou aplikaci je onboarding velice jednoduchý. Jde pouze o vytvoření účtu pro nového uživatele, tedy uložení uživatelského jména a hesla s unikátním identifikátorem, který bude použit napříč celou aplikací pro identifikaci dat daného uživatele.

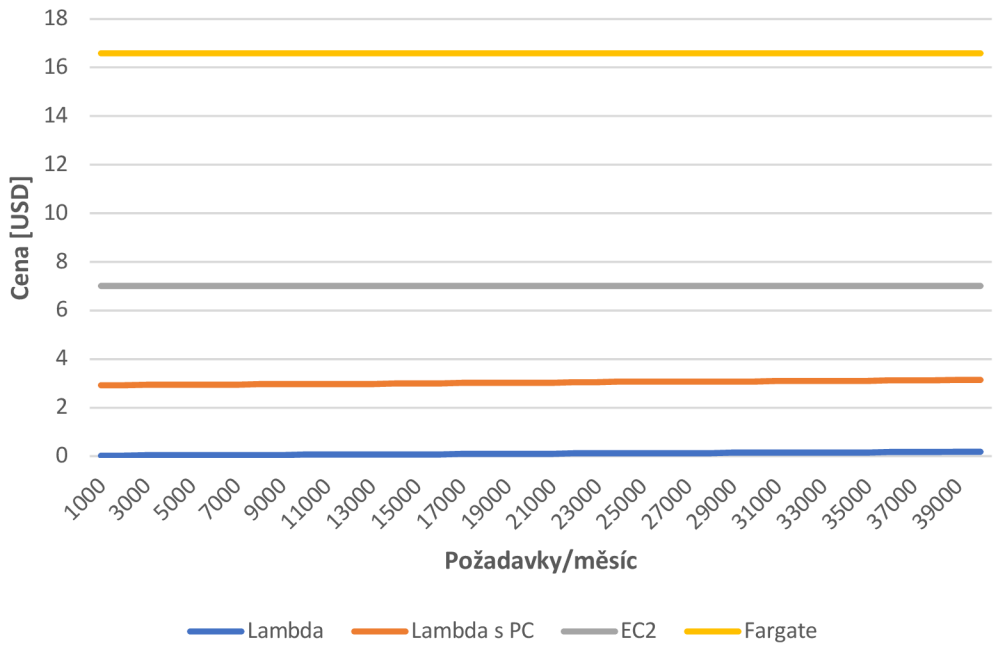
Byla provedena cenová analýza, kdy na základě odhadovaného zatížení aplikace, se porovnaly cenové náklady na provoz různých služeb od AWS. Výsledek analýzy je na grafu 5, kde je vidět že díky nízkému zatížení by se využitím služby AWS Lambda daly velice dobře snížit náklady na provoz celé aplikace. Stojí i za zvážení zajištění funkcí ve stavu hyper-ready s pomocí funkce provisioned concurrency. Díky této funkci je minimalizován problém se studeným startem kdy, první volání funkce trvá déle.

Parametry mikroslužby:

- 0,5 vCPU
- 1 GB RAM

Parametry funkce:

- 40 000 požadavků za měsíc
- Doba spuštění funkce 250ms
- 1 GB RAM




*Graf 8 Cenová analýza onboarding*

Na grafu 8 je vidět rozdíl v cenách při zadaných parametrech. Běh virtuálního stroje je cenově nákladnější než funkce. Varianty využití možností serverless služeb nebo postavení čistě na mikroslužbě byly porovnány vícekriteriální analýzou. Kritéria byla určena v tabulce 8.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 0,19 | Úroveň znalostí AWS služeb je nízká | 34,3456 | Škálování je provedeno automaticky. | Nasazení a konfigurace je jednoduchá | Nabídka programovacích jazyků je omezená včetně verzí. |
| Lambda s PC | 3,14 | Úroveň znalostí AWS služeb je nízká | 34,3456 | Škálování je provedeno automaticky. | Nasazení a konfigurace je jednoduchá | Nabídka programovacích jazyků je omezená včetně verzí. |
| ECS EC2 | 7,01 | Úroveň znalostí AWS služeb je vysoká | 49,935 | Pro škálování je potřeba vytvořit skupinu instancí EC2 | Vysoká, konfigurace prostředí se dělá kompletně manuálně | Spouští se Docker obraz aplikace |
| ECS Fargate | 16,59 | Úroveň znalostí AWS služeb je střední | 50 | Škálování je provedeno automaticky dle konfigurace | Střední, nasazení je jednoduchém, ale vyžaduje ruční přípravu prostředí | Spouští se Docker obraz aplikace |
| Beanstalk EC2 | 7,01 | Úroveň znalostí AWS služeb je vyšší | 49,935 | Škálování je provedeno na základě konfigurace automaticky, omezeně. | Nižší, vytvoření probíhá automaticky, ale další konfigurace a údržba je manuální | Spouští se balíček aplikace. |
| | min | min | max | max | min | max |

*Tabulka 8 VAV Onboarding*

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 10 | 10 | 1 | 10 | 8 | 7 |
| Lambda s PC | 4 | 10 | 1 | 10 | 8 | 7 |
| ECS EC2 | 5 | 5 | 9 | 6 | 1 | 10 |
| ECS Fargate | 1 | 8 | 10 | 9 | 7 | 10 |
| Beanstalk EC2 | 5 | 7 | 9 | 3 | 4 | 9 |
| | max | max | max | max | max | max |

*Tabulka 9 VAV Onboarding bodové ohodnocení*

V tabulce 9 byla všechna kritéria ohodnocena body a převedena z minimalizačních do maximalizačních. V tabulce 10 je výpočet vícekriteriální analýzy variant.

| | Cena | Další náklady | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Váhy | 0,3787 | 0,2515 | 0,1612 | 0,1140 | 0,0623 | 0,0322 |
| Lambda | 3,78743198 | 2,51489801 | 0,16123524 | 1,1401053 | 0,49853586 | 0,22542976 |
| Lambda s PC | 1,51497279 | 2,51489801 | 0,16123524 | 1,1401053 | 0,49853586 | 0,22542976 |
| ECS EC2 | 1,89371599 | 1,257449 | 1,45111714 | 0,68406318 | 0,06231698 | 0,32204251 |
| ECS Fargate | 0,3787432 | 2,01191841 | 1,61235237 | 1,02609477 | 0,43621888 | 0,32204251 |
| Beanstalk EC2 | 1,89371599 | 1,76042861 | 1,45111714 | 0,34203159 | 0,24926793 | 0,28983826 |

*Tabulka 10 VAV Onboarding výpočet*

| Výsledek | |
|---|---|
| Lambda | **8,32763614** |
| Lambda s PC | 6,05517696 |
| ECS EC2 | 5,6707048 |
| ECS Fargate | 5,78737014 |
| Beanstalk EC2 | 5,98639951 |

*Tabulka 11 VAV Onboarding výsledek*

V tomto specifickém případě na základě kritérií, dle výsledku v tabulce 11, je vítězná varianta AWS Lambda.

### 4.6.2 Zpracování souborů

V rámci modelové aplikace je možné nahrát soubor s transakcemi, které se následně spárují s vydanými fakturami. Pro tuto funkcionalitu bude zapotřebí služba, která načte daný soubor a každou transakci pošle do fronty zpráv, aby si další mikroslužby transakci zpracovali transakci a provedli párování.

Možností, jak tento případ užití je více. Jedním z nich je, že po zavolání koncového bodu se vrátí URL pro nahrání souboru do uložiště S3, které je nabízené jako služba od AWS. Po nahrání souboru S3 pošle zprávu a přidání nového souboru a spustí se zpracování. Samotné zpracování může trvat různě dlouhou dobu, takže v případě modelové situace zpracování větších souborů se bude počítat s dobou trvání 2 vteřiny.

V modelovém případě, kdy je očekávaných až 500000 uživatelů může být v průměru nahráno 1,5 souboru na uživatele. To znamená až 750 000 požadavků za měsíc. Mnoho uživatelů, ale tuto funkci vůbec nemusí využít, což by ukázal až reálný provoz aplikace. Je tedy možné, že funkcionalita může být ve většině času nevyužita a stojí zde za zvážení, zda nevyužít některou z optimalizací, kdy by se služba škálovala dle potřeby s tím, že může být i ve stavu kdy neběží, čímž by se optimalizovali náklady.

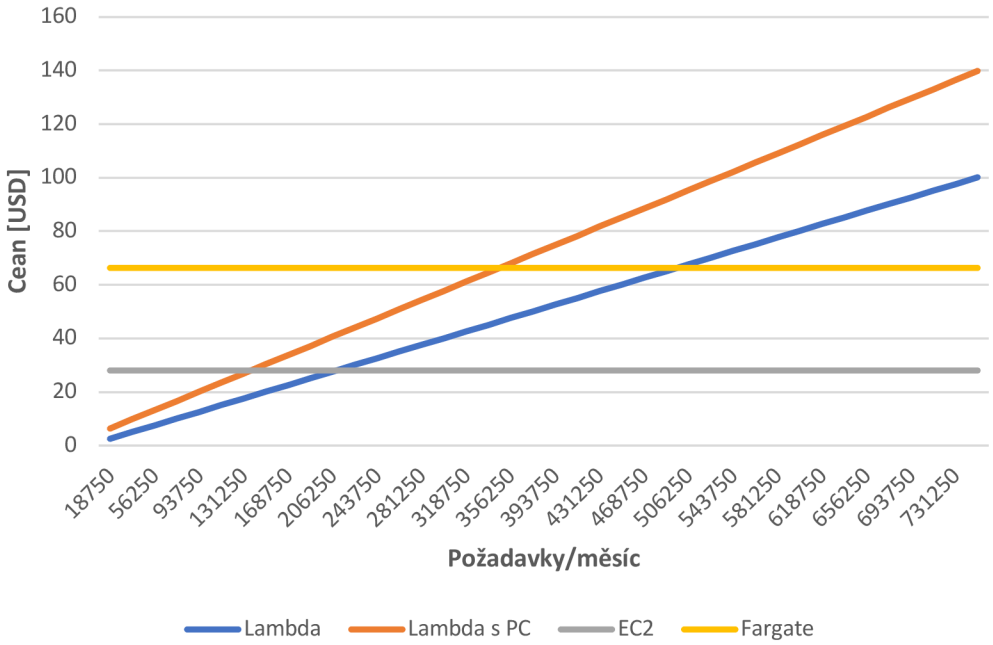
Byla provedena cenová analýza, kdy na základě odhadovaného zatížení aplikace, se porovnaly cenové náklady na provoz různých služeb od AWS. Výsledek analýzy je na grafu 5, kde je vidět že z důvodu dlouhého času spuštění služba AWS Lambda není schopna konkurovat trvale běžící instanci virtuálního stroje.

Parametry mikroslužby:

- 1vCPU
- 4 GB RAM

Parametry funkce:

- 750 000 požadavků za měsíc
- Doba spuštění funkce 2000ms
- 4 GB RAM




*Graf 9 Náklady na službu pro zpracování souborů*

Na grafu 9 je vidět rozdíl v cenách při zadaných parametrech. Výpočetní náročnost má za následek, že cena za spuštění funkce s rostoucím počtem požadavků překonává cenu za instanci virtuálního stroje. Všechny varianty byly porovnány vícekriteriální analýzou. Kritéria byla stanovena v tabulce 12.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 100,16 | Úroveň znalostí AWS služeb je nízká + S3 | 34,3456 | Škálování je provedeno automaticky. | Nízká, zprovoznění je jednoduché a rychlé | Nabídka programovacích jazyků je omezená včetně verzí. |
| Lambda s PC | 139,88 | Úroveň znalostí AWS služeb je nízká + S3 | 34,3456 | Škálování je provedeno automaticky. | Nízká, zprovoznění je jednoduché a rychlé | Nabídka programovacích jazyků je omezená včetně verzí. |
| ECS EC2 | 28,03 | Úroveň znalostí AWS služeb je vysoká + S3 | 49,935 | Pro škálování je potřeba vytvořit skupinu instancí EC2 | Vše je potřeba udělat ručně, spuštění i konfigurace jsou složité | Spouští se Docker obraz aplikace |
| ECS Fargate | 66,33 | Úroveň znalostí AWS služeb je střední + S3 | 50 | Škálování je provedeno automaticky dle konfigurace | Střední, nasazení je jednoduchém, ale je potřeba vytvořit cluster | Spouští se Docker obraz aplikace |
| Beanstalk EC2 | 28,03 | Úroveň znalostí AWS služeb je vyšší + S3 | 49,935 | Škálování je provedeno na základě konfigurace automaticky, omezeně. | Nižší, nasazení je jednoduché, další konfigurace je složitější | Spouští se balíček aplikace. |
| | min | min | max | max | min | max |

*Tabulka 12 VAV hodnoty zpracování souborů*

V tomto specifickém případě přibyla potřeba nastavení uložiště S3, což zvyšuje nároky na znalosti a prohlubuje komplexitu. Kdy je zapotřebí nastavit pro jednotlivé služby nastavit práva přístupu a propojení skrze síť. U některých služeb to je jednodušší než u těch druhých.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 4 | 8 | 1 | 10 | 8 | 7 |
| Lambda s PC | 1 | 8 | 1 | 10 | 8 | 7 |
| ECS EC2 | 10 | 3 | 9 | 6 | 1 | 10 |
| ECS Fargate | 6 | 7 | 10 | 9 | 7 | 10 |
| Beanstalk EC2 | 10 | 5 | 9 | 3 | 4 | 9 |
| | max | max | max | max | max | max |

*Tabulka 13 VAV bodové ohodnocení kritérií*

V tabulce 13 byla všechna kritéria ohodnocena body a převedena z minimalizačních do maximalizačních. Výpočet vícekriteriální analýzy variant je v tabulce 14.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Váhy | 0,3787 | 0,2515 | 0,1612 | 0,1140 | 0,0623 | 0,0322 |
| Lambda | 1,51497279 | 2,01191841 | 0,16123524 | 1,1401053 | 0,49853586 | 0,22542976 |
| Lambda s PC | 0,3787432 | 2,01191841 | 0,16123524 | 1,1401053 | 0,49853586 | 0,22542976 |
| ECS EC2 | 3,78743198 | 0,7544694 | 1,45111714 | 0,68406318 | 0,06231698 | 0,32204251 |
| ECS Fargate | 2,27245919 | 1,76042861 | 1,61235237 | 1,02609477 | 0,43621888 | 0,32204251 |
| Beanstalk EC2 | 3,78743198 | 1,257449 | 1,45111714 | 0,34203159 | 0,24926793 | 0,28983826 |

*Tabulka 14 VAV zpracování souboru výpočet*

| Výsledek | |
|---|---|
| Lambda | 5,55219735 |
| Lambda s PC | 4,41596776 |
| ECS EC2 | 7,06144119 |
| ECS Fargate | **7,42959633** |
| Beanstalk EC2 | 7,3771359 |

*Tabulka 15 VAV zpracování souborů výsledek*

V tomto specifickém případě na základě kritérií, dle výsledků v tabulce 15, je vítězná varianta Elastic Container Service s AWS Fargate.

### 4.6.3 Reporty

Při provozu aplikace je dobré vědět co uživatel dělá a jak jsou funkce aplikace využívány. Tato data mohou být následně analyzována pro potřeby vývoje, obchodu, marketingu apod.

Data jsou sbírána průběžně a aplikace, které je za to zodpovědná stojí spíše stranou, protože se nepodílí přímo na funkcích, které jsou určeny zákazníkům. Sběr dat tady není závislý na rychlé odezvě, spíše na tom, aby se data cestou neztratila a byly zpracovány všechny události z aplikace.

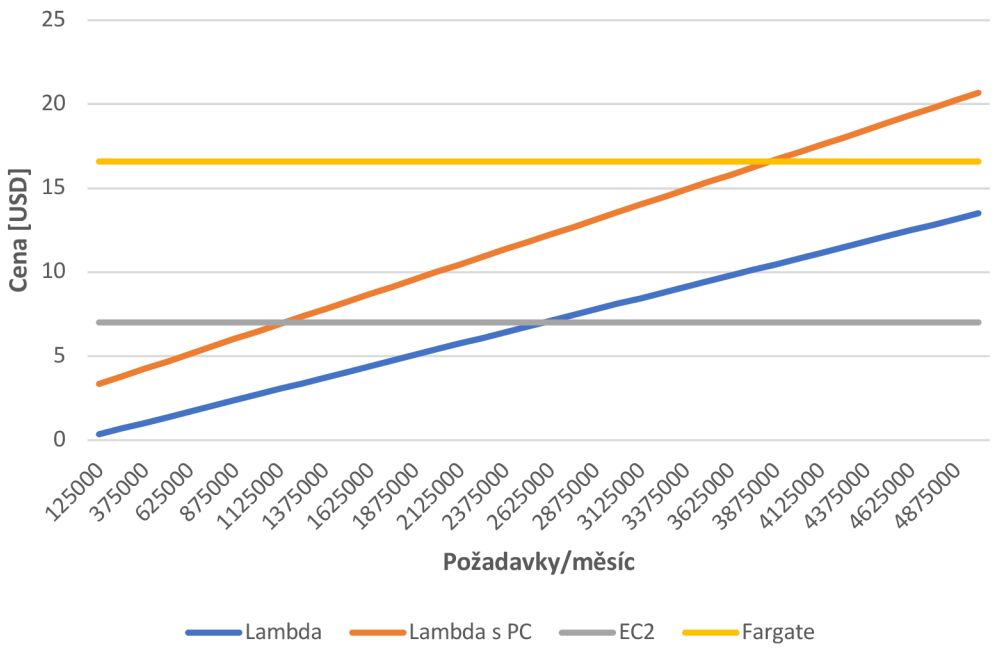
V modelovém příkladu aplikace se při každé akci uživatele odešle zpráva do fronty, kde čeká na zpracování. Služba zodpovědná za tato data zprávy z fronty přečte a data následně zpracuje. Data jsou také uložena do databáze. Dále mohou být dála zpracovávána dle potřeb reportů.

Z modelového přikladu vyplývá, že během měsíce je uživateli vytvořen provoz o zhruba 5.000.000 požadavcích. V průměru 50 požadavků za měsíc na uživatele. Z každé akce, kterou uživatel provede je vytvořena zpráva, která je zaznamenaná pro potřeby reportingu.

Byla provedena cenová analýza, kdy na základě odhadovaného zatížení aplikace, se porovnaly cenové náklady na provoz různých služeb od AWS. Výsledek analýzy je na grafu 7, kde je vidět že z důvodu vysokého počtu požadavků ke zpracování přestává být služba AWS Lambda cenově výhodnější oproti trvale běžící instanci virtuálního stroje.

Parametry mikroslužby:
- 0,5 vCPU
- 1 GB RAM

Parametry funkce:
- 5 000 000 požadavků za měsíc
- Doba spuštění funkce 150ms
- 1 GB RAM




*Graf 10 Cenová analýza reporty*

Na grafu 10 je vidět rozdíl v cenách při zadaných parametrech. Výpočetní náročnost má za následek, že cena za spuštění funkce s rostoucím počtem požadavků překonává cenu za instanci virtuálního stroje. Všechny varianty byly porovnány vícekriteriální analýzou. Kritéria byla stanovena v tabulce 16.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 13,51 | Úroveň znalostí AWS služeb je nízká + Fronta zpráv | 34,3456 | Škálování je provedeno automaticky. | Nízká, zprovoznění je jednoduché a rychlé | Nabídka programovacích jazyků je omezená včetně verzí. |
| Lambda s PC | 20,68 | Úroveň znalostí AWS služeb je nízká + Fronta zpráv | 34,3456 | Škálování je provedeno automaticky. | Nízká, zprovoznění je jednoduché a rychlé | Nabídka programovacích jazyků je omezená včetně verzí. |
| ECS EC2 | 7,01 | Úroveň znalostí AWS služeb je vysoká + Fronta zpráv | 49,935 | Pro škálování je potřeba vytvořit skupinu instancí EC2 | Vše je potřeba udělat ručně, spuštění i konfigurace jsou složité | Spouští se Docker obraz aplikace |
| ECS Fargate | 16,59 | Úroveň znalostí AWS služeb je střední + Fronta zpráv | 50 | Škálování je provedeno automaticky dle konfigurace | Střední, nasazení je jednoduchém, ale je potřeba vytvořit cluster | Spouští se Docker obraz aplikace |
| Beanstalk EC2 | 7,01 | Úroveň znalostí AWS služeb je nízká + Fronta zpráv | 34,3456 | Škálování je provedeno automaticky. | Nízká, zprovoznění je jednoduché a rychlé | Nabídka programovacích jazyků je omezená včetně verzí. |
| | min | min | max | max | min | max |

*Tabulka 16 VAV hodnoty reporty*

V tomto specifickém případě přibyla potřeba nastavení fronty zpráv což zvyšuje nároky na znalosti a prohlubuje komplexitu. Kdy je zapotřebí nastavit pro jednotlivé služby nastavit práva přístupu a propojení skrze síť. U některých služeb to je jednodušší než u těch druhých.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 5 | 7 | 1 | 10 | 7 | 7 |
| Lambda s PC | 1 | 7 | 1 | 10 | 7 | 7 |
| ECS EC2 | 10 | 1 | 9 | 6 | 1 | 10 |
| ECS Fargate | 4 | 6 | 10 | 9 | 6 | 10 |
| Beanstalk EC2 | 10 | 3 | 9 | 3 | 3 | 9 |
| | max | max | max | max | max | max |

*Tabulka 17 VAV bodové ohodnocené kritérií*

V tabulce 17 byla všechna kritéria ohodnocena body a převedena z minimalizačních do maximalizačních. Výpočet vícekriteriální analýzy variant je v tabulce 18.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Váhy | 0,3787 | 0,2515 | 0,1612 | 0,1140 | 0,0623 | 0,0322 |
| Lambda | 1,89371599 | 1,76042861 | 0,16123524 | 1,1401053 | 0,43621888 | 0,22542976 |
| Lambda s PC | 0,3787432 | 1,76042861 | 0,16123524 | 1,1401053 | 0,43621888 | 0,22542976 |
| ECS EC2 | 3,78743198 | 0,2514898 | 1,45111714 | 0,68406318 | 0,06231698 | 0,32204251 |
| ECS Fargate | 1,51497279 | 1,50893881 | 1,61235237 | 1,02609477 | 0,3739019 | 0,32204251 |
| Beanstalk EC2 | 3,78743198 | 0,7544694 | 1,45111714 | 0,34203159 | 0,18695095 | 0,28983826 |

*Tabulka 18 VAV výpočet reporty*

| Výsledek | |
|---|---|
| Lambda | 5,61713377 |
| Lambda s PC | 4,10216098 |
| ECS EC2 | 6,55846159 |
| ECS Fargate | 6,35830315 |
| Beanstalk EC2 | **6,81183932** |

*Tabulka 19 VAV výsledek reporty*

Dle výsledků v tabulce 19 je v tomto specifickém případě na základě kritérií vítězná varianta AWS Elastic Beanstalk.

### 4.6.4 Data třetích stran

Jsou případy, kdy aplikace pro svou aplikační logiku, potřebuje data třetí strany. Mohou to být data, která se mění nahodile jednou a čas nebo v pravidelných intervalech. V případech, kdy časový interval je jednou denně nebo i delší, je teoreticky možné, že by se dalo změnou architektury nebo optimalizací technologií dosáhnout úspory v nákladech a zvýšení efektivity aplikace.

V modelové aplikaci je zapotřebí služba, která získá kurzovní lístek z národní banky v zemi, kde je aplikace nabízena uživatelům. Pro konverzi měn mezi domácí a zahraniční měnou v případě, že uživatel vystavuje faktury v cizí měně.

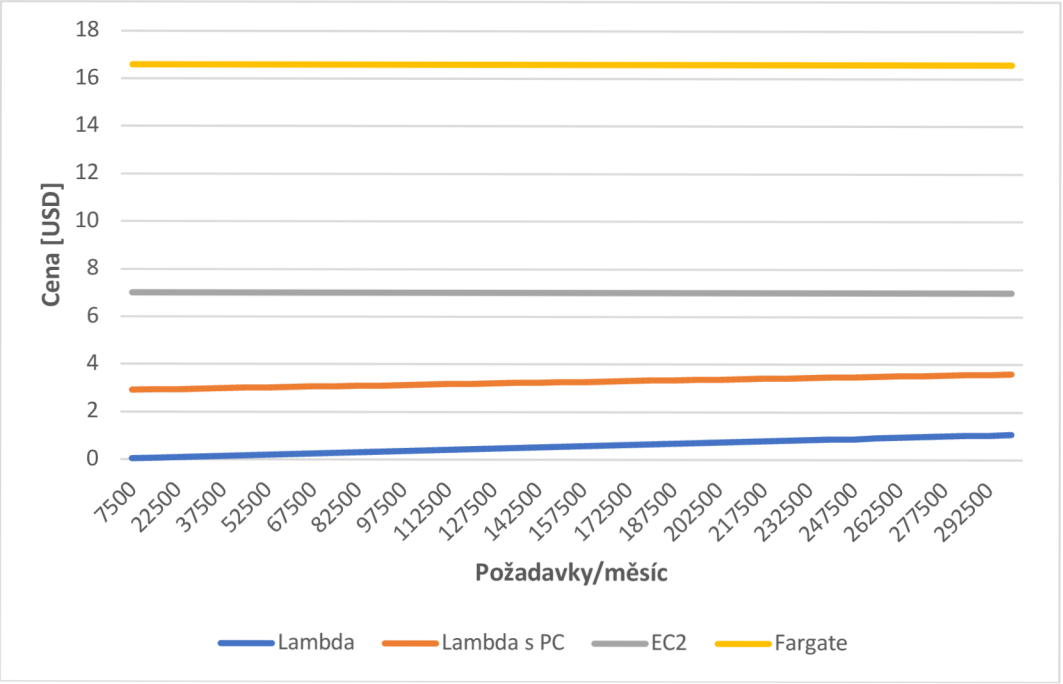
Modelová služba musí stáhnout data od třetí strany jednou denně. Tedy kurzovní lístek pro daný den. Služba se plánovaně spustí jednou denně pro stažení dat. Získána data jsou uložena do databáze. Vystavené API bude je jeden koncový bod, ze kterého budou ostatní služby získávat směnné kurzy. Příliš mnoho požadavků na vracení kurzů se neočekává. Ve většině času bude služba nevyužita.

Modelového aplikace očekává celkem až 500 000 uživatelů. Každý uživatel v průměru vystaví 3 faktury měsíčně. 20 % faktur je v cizí měně. Zhruba může přijít 300 000 požadavků na měnový kurz.

Byla provedena cenová analýza, kdy na základě odhadovaného zatížení aplikace, se porovnaly cenové náklady na provoz různých služeb od AWS. Výsledek analýzy je na grafu 11.

Parametry mikroslužby:
- 0,5 vCPU
- 1 GB RAM

Parametry funkce:
- 300 000 požadavků za měsíc
- Doba spuštění funkce 200ms
- 1 GB RAM




*Graf 11 Cenová analýza data třetích stran*

Při zadaných parametrech vychází cena za spuštění 1 funkce na 0,14 dolaru za měsíc. Při takto nízké ceně je možné si připlatit za službu provisioned concurrency, kdy by bylo možné během pracovních hodin držet funkci v pohotovosti, ve stavu hyper-ready. Tím minimalizovat studené starty kdy funkce nabíhá při prvním spuštěním pomaleji z důvodů spouštění prostředí pro běh funkce. Varianty využití možností serverless služeb nebo postavení čistě na mikroslužbě byly porovnány vícekriteriální analýzou. Hodnoty kritérií byly stanoveny v tabulce 20.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 1,07 | Úroveň znalostí AWS služeb je nízká | 34,3456 | Škálování je provedeno automaticky. | Nasazení a konfigurace je jednoduchá | Nabídka programovacích jazyků je omezená včetně verzí. |
| Lambda s PC | 3,61 | Úroveň znalostí AWS služeb je nízká | 34,3456 | Škálování je provedeno automaticky. | Nasazení a konfigurace je jednoduchá | Nabídka programovacích jazyků je omezená včetně verzí. |
| ECS EC2 | 7,01 | Úroveň znalostí AWS služeb je vysoká | 49,935 | Pro škálování je potřeba vytvořit skupinu instancí EC2 | Vysoká, konfigurace prostředí se dělá kompletně manuálně | Spouští se Docker obraz aplikace |
| ECS Fargate | 16,59 | Úroveň znalostí AWS služeb je střední | 50 | Škálování je provedeno automaticky dle konfigurace | Střední, nasazení je jednoduchém, ale vyžaduje ruční přípravu prostředí | Spouští se Docker obraz aplikace |
| Beanstalk EC2 | 7,01 | Úroveň znalostí AWS služeb je vyšší | 49,935 | Škálování je provedeno na základě konfigurace automaticky, omezeně. | Nižší, vytvoření probíhá automaticky, ale další konfigurace a údržba je manuální | Spouští se balíček aplikace. |
| | min | min | max | max | min | max |

*Tabulka 20 VAV hodnoty data třetích stran*

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Lambda | 10 | 10 | 1 | 10 | 8 | 7 |
| Lambda s PC | 8 | 10 | 1 | 10 | 8 | 7 |
| ECS EC2 | 6 | 5 | 9 | 6 | 1 | 10 |
| ECS Fargate | 1 | 8 | 10 | 9 | 7 | 10 |
| Beanstalk EC2 | 6 | 8 | 9 | 3 | 4 | 9 |
| | max | max | max | max | max | max |

*Tabulka 21 VAV bodové ohodnocení kritérií*

V tabulce 18 byla všechna kritéria ohodnocena body a převedena z minimalizačních do maximalizačních. Výpočet vícekriteriální analýzy variant je v tabulce 22.

| | Cena | Úroveň znalostí | Výkon | Škálování | Komplexita | Podpora platformy |
|---|---|---|---|---|---|---|
| Váhy | 0,3787 | 0,2515 | 0,1612 | 0,1140 | 0,0623 | 0,0322 |
| Lambda | 3,78743198 | 2,51489801 | 0,16123524 | 1,1401053 | 0,49853586 | 0,22542976 |
| Lambda s PC | 3,02994558 | 2,51489801 | 0,16123524 | 1,1401053 | 0,49853586 | 0,22542976 |
| ECS EC2 | 2,27245919 | 1,257449 | 1,45111714 | 0,68406318 | 0,06231698 | 0,32204251 |
| ECS Fargate | 0,3787432 | 2,01191841 | 1,61235237 | 1,02609477 | 0,43621888 | 0,32204251 |
| Beanstalk EC2 | 2,27245919 | 1,76042861 | 1,45111714 | 0,34203159 | 0,24926793 | 0,28983826 |

*Tabulka 22 VAV výpočet bodové ohodnocení kritérií*

| Výsledek | |
|---|---|
| Lambda | **8,32763614** |
| Lambda s PC | 7,57014975 |
| ECS EC2 | 6,049448 |
| ECS Fargate | 5,78737014 |
| Beanstalk EC2 | 6,36514271 |

*Tabulka 23 VAV výsledek data třetích stran*

V tabulce 23 je s nejvyšší hodnotou varianta AWS Lambda. V tomto specifickém případě na základě kritérií je nejvhodnější variantou k použití.

# 5 Výsledky a diskuse

## 5.1 Onboarding

| Výsledek VAV | |
|---|---|
| Lambda | **8,33** |
| Lambda s PC | 6,06 |
| ECS EC2 | 5,67 |
| ECS Fargate | 5,79 |
| Beanstalk EC2 | 5,99 |

*Tabulka 24 Výsledek VAV onboarding*

Výsledkem vícekriteriální analýzy variant pro případ onboardingu je použití serverless řešení AWS Lambda. Dle výsledků v tabulce 24. Dalším možným řešením je AWS Lambda s provisioned concurrency, kterou je vhodné použít v případě, že studené starty služby AWS Lambda by způsobovaly problém s dostupností a měly by za následek zhoršení akvizice nových uživatelů.

Velmi záleží na tom, jak je akvizice uživatelů aplikace navržena. Jelikož je akvizice uživatelů velmi zjednodušená, v tomto konkrétním případě se lze vydat i cestou kdy se doména akvizice uživatelů spojí doménou uživatele. Tedy že onboarding by byl součástí mikroslužby pro správu uživatelských dat.

## 5.2 Zpracování souborů

| Výsledek VAV | |
|---|---|
| Lambda | 5,55 |
| Lambda s PC | 4,42 |
| ECS EC2 | 7,06 |
| ECS Fargate | **7,43** |
| Beanstalk EC2 | 7,38 |

*Tabulka 25 Výsledek VAV zpracování souborů*

Pro modelový příklad zpracování souborů, kdy jsou nahrané soubory zpracovány a data uložena do databáze včetně předání dat skrze frontu zpráv do dalších služeb, je nejvhodnější použít serverless službu AWS Fargate typu container as a service. Dle výsledků v tabulce 25.

I v případě, že je cluster aplikací postavený na instancích typu EC2, je na místě zvážit tento výsledek a pro nejlepší poměr cena/výkon postavit architekturu na hybridním modelu

mikroslužeb se serverless řešením. Služba Elastic Container Service umožňuje provozovat instance Fargate i EC2 zároveň.

## 5.3 Reporty

| Výsledek VAV | |
|---|---|
| Lambda | 5,62 |
| Lambda s PC | 4,10 |
| ECS EC2 | 6,56 |
| ECS Fargate | 6,36 |
| Beanstalk EC2 | **6,81** |

*Tabulka 26 Výsledek VAV reporty*

Pro zpracování dat pro reporty bylo vícekriteriální analýzou variant určeno jako nevhodnější řešení typu platform as a service, služba Elastic Beanstalk. Z hlediska modelové aplikace, u služby, která nemá na starost funkcionalitu potřebnou pro uživatele, je možné použít toto řešení pro zpracování dat z fronty zpráv. Dle výsledků v tabulce 26.

Ovšem stojí za zmínění, že v případě funkčního clusteru mikroslužeb je možné využít již existujícího prostředí a tuto službu přidat do clusteru mezi ostatní mikroslužby. Ostatně k tomu navádí druhý nejlepší výsledek z vícekriteriální analýzy variant. Elastic Container Service s typem instance EC2.

## 5.4 Data třetích stran

| Výsledek VAV | |
|---|---|
| Lambda | **8,33** |
| Lambda s PC | 7,57 |
| ECS EC2 | 6,05 |
| ECS Fargate | 5,79 |
| Beanstalk EC2 | 6,37 |

*Tabulka 27 Výsledek VAV data třetích stran*

Pro získávání dat třetích stran, dle výsledků v tabulce 27, vyšlo jako nejvhodnější použití serverless řešení v podobě funkce se služnou AWS Lambda. Opět je na místě zvážení za cenou vyšších nákladů použití funkce provisioned concurrency. Kdy je prostředí pro spuštění funkce drženo ve stavu hyper-ready, čímž je snížena latence v důsledku studeného startu na minimum.

Oproti předchozím případům je zde použití serverless řešení v architektuře na místě, a to z důvodu, kdy je získání potřeba za delší časový úsek (24 hodin) a zároveň je potřebná dostupnost služby v případě volání. Mikroslužba v tomto případě by v poměru cena výkon nebyla příliš efektivní z důvodu nečinnosti a nelze to řešit ani plánovaným spouštěním mikroslužby z důvodu potřeby volání služby pro získání dat.

# 6 Závěr

Architektura mikroslužeb je v dnešní době velmi populární. Nabízí mnoho pozitiv, které velmi usnadňují vývoj, nasazení a provoz. Má ovšem i některé negativní stránky. I když se jedná o jednu aplikaci, která může být navržena a postavena jako monolit, ve výsledku se jedná o mnoho malých samostatných aplikací, které je potřeba navrhnout, vyvinout, nasadit a udržovat. S tím roste potřeba na kvalitní management infrastruktury. Zde přichází na řadu serverless. Poskytovatelé cloudových služeb jdou naproti vývojářům a architektům nabídkou služeb, které se o infrastrukturu a další potřebné služby postarají za ně. Je však potřeba přizpůsobit architekturu vyvíjeného softwaru přizpůsobit nabízeným službám.

Serverless nabízí nové možnosti vývoje cloudových aplikací a rozhodně ulehčení v některých oblastech tvorby softwaru. Výraznou nevýhodou ovšem je vznikající závislost na poskytovateli dané služby. V některých případech, kdy daný poskytovatel v některém z regionů neposkytuje konkrétní služby může nastat problém s expanzí aplikace na nové trhy.

Pro splnění hlavního cíle práce bylo potřeba zvolit specifické případy pro komparaci architektury mikroslužeb a serverless. Během teoretického výzkumu bylo zjištěno, že serverless má určité omezení, které je potřeba implikovat do výběru specifických případů. Serverless funkce se rozhodně nehodí pro služby s konstantním zatížením, a to z důvodu ceny a výkonu. Naopak se hodí pro jednodušší úlohy, kde není zatížení natolik vysoké. Serverless kontejnery se nehodí v případě kdy je zapotřebí rozšířená konfigurace prostředí nebo v případě udržení nízkých provozních nákladů i za cenu delší tvorby prostředí.

Z výsledků komparace je patrné, že velmi záleží na všech detailech a okolnostech daného případu, který porovnáván. Každá změna v architektuře nebo návrhu logiky může mít vliv na výsledek komparace. Porovnávané modelové případy ukazují vhodné případy užití, kdy je díky hybridní architektuře možné dosáhnout optimalizace nákladů a v některých případech i výkonu díky škálování.

Práce by se dala rozšířit o porovnání s daty z reálného provozu. Modelová aplikace by byla naprogramována a vybrané případy by mohly být srovnány na základě reálných dat výkonu, provozu a cen.

# 7 Seznam použitých zdrojů


1. Bass, Len, Clements, Paul a Kazman, Rick. *Software Architecture in Practice: Third Edition.* New Jersey : Addison-Wesley Professional, 2012. 9780321815736.

2. What is cloud computing. *Microsoft Azure.* [Online] Microsoft, 2023. [Citace: 13. 1 2023.] https://azure.microsoft.com/cs-cz/resources/cloud-computing-dictionary/what-is-cloud-computing/.

3. Marinescu, Dan. *Cloud Computing: Theory and Practice.* Waltham : Elsevier, 2022. 978-0323852777.

4. What is cloud computing? *IBM.* [Online] 2022. [Citace: 14. 1 2023.] https://www.ibm.com/topics/cloud-computing.

5. Mell, Peter. The NIST Definition of Cloud Computing. *Computer Security Resource Center.* [Online] 28. 9 2011. [Citace: 14. 1 2023.] https://csrc.nist.gov/publications/detail/sp/800-145/final. SP 800-145.

6. Richter, Felix. Amazon, Microsoft & Google Dominate Cloud Market. *Statista.* [Online] 23. 12 2022. [Citace: 15. 1 2023.] https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

7. What is AWS. *Amazon Web Services.* [Online] AWS, 2022. [Citace: 10. 1 2023.] https://aws.amazon.com/what-is-aws/.

8. Cloud Computing Services. *Google Cloud.* [Online] Google, 2022. [Citace: 10. 1 2023.] https://cloud.google.com/.

9. What is Azure? *Microsoft Cloud Services.* [Online] 2022. [Citace: 15. 1 2023.] https://azure.microsoft.com/en-us/overview/what-is-azure/.

10. Alibaba Cloud About. *Alibaba Cloud.* [Online] 2022. [Citace: 15. 1 2023.] https://www.alibabacloud.com/about.

11. IBM Cloud. *IBM.* [Online] 2022. [Citace: 15. 1 2023.] https://www.ibm.com/cloud.

12. What is Salesforce? - What does Salesforce do? *Salesforce.com.* [Online] 2022. [Citace: 15. 1 2023.] https://www.salesforce.com/what-is-salesforce/.

13. Why Tencent Cloud ? . *Tencent Cloud.* [Online] 2022. [Citace: 15. 1 2023.] https://intl.cloud.tencent.com/.

14. Oracle Cloud Infrastructure. *Oracle.* [Online] 2022. [Citace: 15. 1 2023.] https://www.oracle.com/cloud/.

15. Mazzeschi, Michelangiolo. What Are Cloud IAAS, PAAS, SAAS, FAAS, And Why We Use Them. *Medium.* [Online] 19. 4 2021. [Citace: 15. 1 2023.] https://pub.towardsai.net/what-are-cloud-iaas-paas-saas-faas-and-why-we-use-them-8af979dad141.

16. What is SaaS – software-as-a-service? *IBM.* [Online] 2022. [Citace: 15. 1 2023.] https://www.ibm.com/topics/saas.

17. What is Platform-as-a-Service (PaaS)? *IBM.* [Online] 2022. [Citace: 15. 1 2023.] https://www.ibm.com/topics/paas.

18. What is IaaS? *Red Hat.* [Online] 22. 6 2022. [Citace: 15. 1 2023.] https://www.redhat.com/en/topics/cloud-computing/what-is-iaas.

19. What is Function-as-a-Service (FaaS)? *Red Hat.* [Online] 3. 1 2020. [Citace: 15. 1 2023.] https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas.

20. Understanding cloud-native applications. *Red Hat.* [Online] 10. 5 2022. [Citace: 16. 1 2023.] https://www.redhat.com/en/topics/cloud-native-apps.

21. What is Microservices Architecture? *Google Cloud.* [Online] [Citace: 19. 1 2023.] https://cloud.google.com/learn/what-is-microservices-architecture.

22. Microservices architecture design. *Microsoft.* [Online] [Citace: 19. 1 2023.] https://learn.microsoft.com/en-us/azure/architecture/microservices/.

23. Fernandez, Tomas. 5 Options for Deploying Microservices. *Semaphore.* [Online] 8. 9 2022. [Citace: 20. 1 2023.] https://semaphoreci.com/blog/deploy-microservices.

24. Docker Get Started. *Docker Docs.* [Online] 2023. [Citace: 21. 1 2023.] https://docs.docker.com/get-started/.

25. What is serverless? *Red Hat.* [Online] 11. 5 2022. [Citace: 21. 1 2023.] https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless.

26. Sbarski, Peter. *Serverless Architecture on AWS.* Shelter Island : Manning Publications Co., 2017. 9781617293825.

27. AWS Fargate. *AWS Fargate.* [Online] AWS, 2023. [Citace: 23. 1 2023.] https://aws.amazon.com/fargate/.

28. Cloud Run. *Google Cloud.* [Online] Google, 2023. [Citace: 21. 1 2023.] https://cloud.google.com/run.

29. Container Instances. *Microsoft Azure.* [Online] Microsoft, 2023. [Citace: 21. 1 2023.] https://azure.microsoft.com/en-us/products/container-instances/.

30. IBM Cloud Code Engine. *IBM.* [Online] IBM, 2023. [Citace: 21. 1 2023.] https://www.ibm.com/cloud/code-engine.

31. Heroku dynos. *Heroku.* [Online] Salesforce, 2023. [Citace: 21. 1 2023.] https://www.heroku.com/dynos.

32. Elastic Container Instance. *Alibaba Cloud.* [Online] Alibaba, 2023. [Citace: 21. 1 2023.] https://www.alibabacloud.com/product/elastic-container-instance.

33. Container Instances. *Oracle.* [Online] Oracle, 2023. [Citace: 21. 1 2023.] https://www.oracle.com/cloud/cloud-native/container-instances/.

34. AWS Lambda. *Amazon Web Services.* [Online] Amazon Web Services, 2023. [Citace: 21. 1 2023.] https://aws.amazon.com/lambda/.

35. Azure Functions. *Microsoft Azure.* [Online] Microsoft, 2023. [Citace: 21. 1 2023.] https://azure.microsoft.com/en-us/products/functions/.

36. Cloud Functions. *Google Cloud.* [Online] Google, 2023. [Citace: 21. 1 2023.] https://cloud.google.com/functions.

37. IBM Cloud Functions. *IBM Cloud.* [Online] IBM, 2023. [Citace: 21. 1 2023.] https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-getting-started.

38. Function Compute. *Alibaba Cloud.* [Online] Alibaba, 2023. [Citace: 21. 1 2023.] https://www.alibabacloud.com/product/function-compute.

39. Serverless Cloud Function. *Tencent Cloud.* [Online] Tencent, 2023. [Citace: 21. 1 2023.] https://www.tencentcloud.com/products/scf.

40. Cloud Functions. *Oracle Cloud Infrastructure .* [Online] Oracle, 2023. [Citace: 21. 1 2023.] https://www.oracle.com/cloud/cloud-native/functions/.

41. Netflix & AWS Lambda Case Study. *Amazon Web Services.* [Online] Amazon Web Services, 2014. [Citace: 22. 1 2023.] https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/.

42. Retter, Mariliis. Serverless Case Study - Netflix. *Dashbird.* [Online] DashVird, 30. 7 2020. [Citace: 23. 1 2023.] https://dashbird.io/blog/serverless-case-study-netflix/.

43. AWS Pricing Calculator. *Amazon Web Services.* [Online] Amazon Web Services, 2023. [Citace: 28. 1 2023.] https://calculator.aws/.

44. Pricing calculator. *Microsoft Azure.* [Online] Microsoft, 2023. [Citace: 28. 1 2023.] https://azure.microsoft.com/en-us/pricing/calculator/.

45. Google Cloud Pricing Calculator. *Google Cloud.* [Online] Google, 28. 1 2023. [Citace: 28. 1 2023.] https://cloud.google.com/products/calculator.

46. AWS Cloud Products. *Amazon Web Services.* [Online] Amazon Web Services, 2023. [Citace: 28. 1 2023.] https://aws.amazon.com/products/.

47. Azure Products. *Microsoft Azure.* [Online] Microsoft, 2023. [Citace: 28. 1 2023.] https://azure.microsoft.com/en-us/products/.

48. Google Cloud products. *Google Cloud.* [Online] Google, 2023. [Citace: 28. 1 2023.] https://cloud.google.com/products.

49. Regions and Availability Zones. *Amazon Web Services.* [Online] Amazon Web Services, 2023. [Citace: 29. 1 2023.] https://aws.amazon.com/about-aws/global-infrastructure/regions_az/.

50. Azure geographies. *Microsoft Azure.* [Online] Microsoft, 2023. [Citace: 29. 1 2023.] https://azure.microsoft.com/en-us/explore/global-infrastructure/geographies/.

51. Regions and zones. *Google Cloud.* [Online] Google, 2023. [Citace: 29. 1 2023.] https://cloud.google.com/compute/docs/regions-zones.

52. 2022 Developer Survey. *StackOverflow.* [Online] StackOverflow, 2022. [Citace: 28. 1 2023.] https://survey.stackoverflow.co/2022/.

53. AWS Elastic Beanstalk Documentation. *Amazon Web Services.* [Online] Amazon Web Services, 2023. [Citace: 29. 1 2023.] https://docs.aws.amazon.com/elastic-beanstalk/index.html.

54. Amazon Elastic Container Service Documentation. *Amazon Web Services.* [Online] Amazon Web Services, 2023. [Citace: 30. 1 2023.] https://docs.aws.amazon.com/ecs/.

# 8 Seznam obrázků, tabulek, grafů a zkratek

## 8.1 Seznam obrázků



## 8.2 Seznam tabulek

## 8.3 Seznam grafů



## 8.4 Seznam použitých zkratek

DevOps – Development Operations
vCPU – virtual CPU – Central Processing Unit
RAM – Random Access Memory
ECS – Elastic Container Service
EC2 – Elastic Container 2
URI - Uniform Resource Identifier
CI – Continuous Integration
CD - Continuous Deployment
AWS – Amazon Web Services
GCP – Google Cloud Platform
OCI – Oracle Cloud Infrastructure