Univerzita Pardubice

Fakulta ekonomicko-správní

Moderní softwarové architektury

Bakalářská práce

2023 Martin Mudroch

Univerzita Pardubice
Fakulta ekonomicko-správní
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Martin Mudroch**
Osobní číslo:	**E20143**
Studijní program:	**B0688A140004 Informatika a systémové inženýrství**
Specializace:	**Informatika ve veřejné správě**
Téma práce:	**Moderní softwarové architektury**
Zadávající katedra:	**Ústav systémového inženýrství a informatiky**

Zásady pro vypracování

Cílem práce je namodelovat vybrané moderní softwarové architektury. Součástí práce bude srovnání navržených modelů a posouzení předností daných architektur.
Osnova:
- Koncept SW architektury a jeho historie.
- SW a systémové architektury v kontextu UML, SOA a datových skladů.
- Moderní SW architektury a jejich utilizace.
- Srovnání architektur z hlediska jejich výhod a nevýhod.
- Formulace závěrů a doporučení.

Rozsah pracovní zprávy: **cca. 35 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

BOOCH, G., J. RUMBAUGH a I. JACOBSON. *Unified modeling language user guide*. Massachusetts: Addison-Wesley, 1999. ISBN 0-201-57168-4.
VAISMAN, Alejandro a Esteban ZIMÁNYI. *Data warehouse systems: design and implementation*. Heidelberg: Springer, [2014]. Data-centric systems and applications. ISBN 3642546544.
EELES, Peter a Peter CRIPPS. *Architektura softwaru*. Brno: Computer Press, 2011. ISBN 9788025130360.
FORD, Neal a Mark RICHARDS. *Fundamentals of Software Architecture*. O'Reilly Media, Inc, USA, 2020. ISBN 1492043451.
J. BRAUDE, Eric a Michael E. BERNSTEIN. *Software Engineering: Modern Approaches*. Second Edition. Waveland Press, 2016. ISBN 978-1478632306.

Vedoucí bakalářské práce: **Ing. et Ing. Martin Lněnička, PhD.**
Ústav systémového inženýrství a informatiky

Datum zadání bakalářské práce: **1. září 2022**
Termín odevzdání bakalářské práce: **30. dubna 2023**

L.S.

prof. Ing. Jan Stejskal, Ph.D.
děkan

RNDr. Ing. Oldřich Horák, Ph.D.
vedoucí ústavu

V Pardubicích dne 1. září 2022

Prohlášení:

Prohlašuji:

Práci s názvem *Moderní softwarové architektury* jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 25. 8. 2023

Martin Mudroch v. r.

Poděkování

Rád bych využil této příležitosti k vyjádření svého upřímného poděkování panu doktorovi Martinovi Lněničkovi za jeho cenné vedení, podporu a odborné rady v průběhu mé práce. Jeho vedení a znalosti v oblasti softwarové architektury byly neocenitelné pro mé hlubší porozumění této problematice. Díky jeho podnětným připomínkám a konstruktivní zpětné vazbě jsem byl rozhodně schopen dosáhnout vyšší úrovně kvality výsledné práce. Jeho trpělivost a ochota naslouchat byly klíčovými faktory, které mi umožnily úspěšně dokončit tento projekt. Velice si vážím jeho profesionálního přístupu a vstřícnosti. Děkuji za jeho vedení a věnovaný čas, který mi umožnil dosáhnout tohoto cíle.

ANOTACE

Tato práce zkoumá základy softwarové architektury jako takové. Dále probírá moderní softwarové architektury včetně EDA, MVC, MVVM, mikroslužeb a serverless. Analyzuje jejich výhody, nevýhody a aplikovatelnost s důrazem na porovnání v kontextu veřejné správy.

KLÍČOVÁ SLOVA

Moderní softwarové architektury, EDA, MVC, MVVM, mikroslužby, serveless architektura, veřejná správa, model

TITLE

Modern software architectures

ANNOTATION

This thesis explores the fundamentals of software architecture itself. Then explains basics of modern software architectures, including EDA, MVC, MVVM, microservices, and serverless. It analyses their advantages, disadvantages, and applicability, with an emphasis on comparisons within context of public administration.

KEYWORDS

Modern software architectures, EDA, MVC, MVVM, microservices, serverless architecture, public administration, model

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

24/7	24 hodin denně, 7 dní v týdnu
AHP	Analytický hierarchický proces/Analytic Hierarchy Process
API	Application Programming Interface
BI	Business Intelligence
CAP teorém	Konzistence (C), Dostupnost (A), Tolerance rozdělení (P)
CR	Consitency Ratio
EDA	Event-Driven Architecture/ Událostmi řízená architektura

FaaS	Function as a Service
GDPR	General Data Protection Regulation
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
MVC	Model-view-controller
MVVM	Model–view–viewmodel
SOA	Service Oriented Architecture
SWOT	Strengths, Weaknesses, Opportunities, Threats
UML	Unified Modeling Language
YAGNI	You aren't gonna need it
WAF	Web application firewall

ÚVOD

Bez správné komunikace a koordinace mezi jednotlivými komponentami v softwaru nemusí výsledný produkt správně fungovat. Softwarová architektura je tedy jedním z důležitých bodů ve vývoji softwaru. Její podíly na vývoji softwaru jsou navrhování rozhodnutí související s celkovou strukturou systému a jeho komponentami. Zároveň však i jejich komunikací a vztahem mezi nimi. Moderní softwarová architektura tedy nabízí mnoho výhod. Dnes je to hlavní základ pro všechny softwarové aplikace a systémy. Při její tvorbě je potřeba pro každou její část upřednostnit jiné podstatné vlastnosti řešení (jako je např. rychlost, spolehlivost, nízkou cenu apod.) a díky tomu se i požadavky na architekturu jednotlivých softwarových systémů často odlišují. Proto je důležité vybrat vhodnou softwarovou architekturu, která se hodí právě do dané situace.

Navrhnout dobrou architekturu není jednoduchá činnost, pokud by totiž architektura nebyla navržena dobře, může vést k zbytečným problémům jako špatná adaptabilita k inovacím, kvůli které je zapotřebí opakovaného kódování a kvůli tomu nakonec i drahé údržbě. Naopak dobře promyšlená architektura může snížit náklady na vývoj a údržbu, zejména pokud se jedná o bezpečnost dat, ale hlavně může vydržet po celou dobu životního cyklu softwarové aplikace.

Vývoj, který v posledních letech probíhá čím dál rychleji, klade mnohem větší časovou náročnost na seberozvoj v tomto oboru. Dříve softwarová architektura byla často vnímána jenom jako plán, který definoval strukturu a rozvržení systému. Nicméně v dnešní době, kdy se už zmiňovaný vývoj neustále zrychluje, se pohled na softwarovou architekturu trochu změnil. Softwaroví architekti se musí každou chvíli vyrovnávat s novými složitostmi a hledat optimální řešení. Moderní architektura se tím pádem dnes spíše zaměřuje na principy, které umožňují právě tu neustálou adaptaci, rychlé inovace a snadnou škálovatelnost.

Cílem této práce je namodelovat vybrané moderní softwarové architektury ve smyslu srovnání navržených modelů a posouzení předností daných architektur. Za účelem splnění tohoto cíle jsou na základě studia dostupné literatury identifikovány moderní softwarové architektury, popsány jejich výhody a nevýhody, zvolena kritérií pro jejich porovnání, navržen rozhodovací model, vybraná optimální softwarová architektura, která je nakonec namodelována pro zvolený příklad využití architektury v praxi, tzn. ve veřejné správě.

První část práce se zaměřuje na koncept softwarové architektury a jeho historii. Cílem kapitoly je blíže představit vývoj v této oblasti, který je důležitý pro pochopení požadavků, které jsou kladeny na moderní softwarové architektury a jejich návrh. Kromě přehledu požadavků a představení postupů pro výběr vhodné architektury jsou v této kapitole také zmíněny konkrétní metody pro porovnání a výběr softwarové architektury, a především vymezení softwarových architektur v kontextu veřejné správy.

Další kapitola se zaměřuje na vymezení a pochopení souvisejících pojmů, které jsou klíčové pro moderní softwarové architektury. Nachází se zde softwarová architektura v kontextu UML (Unified Modeling Language), SOA (Service-Oriented Architecture) a datových skladů, to pomáhá mnohem lépe identifikovat vzájemné vztahy a integraci těchto architektur.

Ve třetí kapitole této práce jsou představeny konkrétní moderní softwarové architektury, tedy EDA (Event-Driven Architecture), MVC (Model-View-Controller), MVVM (Model-View-ViewModel), mikroslužby a serverless architektura. Každá z těchto architektur je podrobněji popsána.

Klíčová kapitola práce se zabývá porovnáním těchto typů softwarových architektur pomocí vybraných kritérií a s pomocí metody vícekriteriálního rozhodování představuje modely, které jsou navržené pro vybraný příklad tak, aby byly posouzení přednosti daných architektur pro daný příklad a vybrané nejvhodnější alternativa. Ta je následně namodelována a jsou popsány jednotlivé prvky dané architektury.

V poslední kapitole jsou následně zformulovány závěry a doporučení pro využívání daných architektur s ohledem na jejich výhody a nevýhody pro potřeby veřejné správy.

1 KONCEPT SOFTWAROVÉ ARCHITEKTURY A JEHO HISTORIE

1.1 VYMEZENÍ KONCEPTU A SOUVISEJÍCÍ POJMY

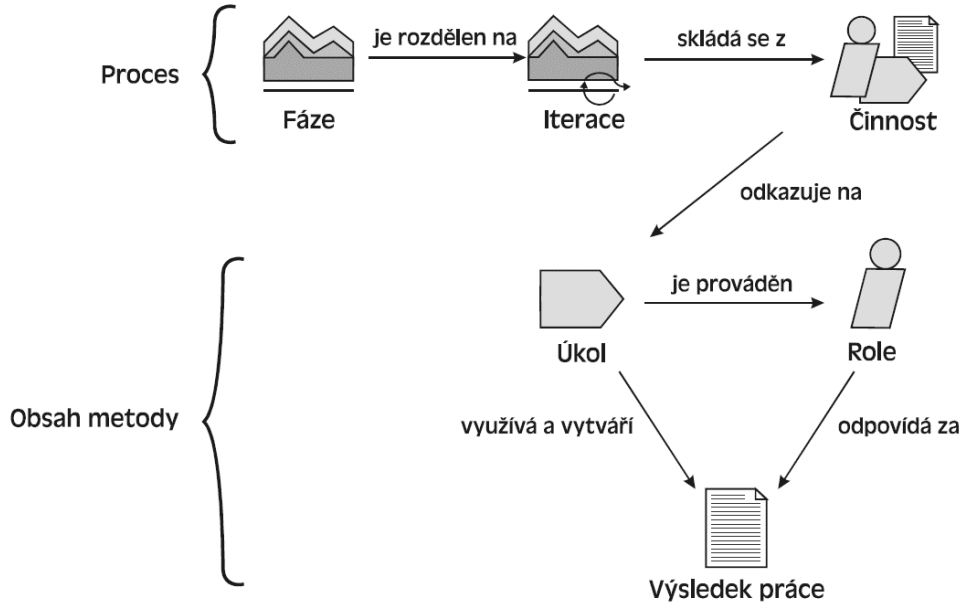
Definice Richards a Ford (2020) říká, že se softwarová architektura skládá ze struktury systému v kombinaci s charakteristikami architektury, které musí systém podporovat, architektonickými rozhodnutími, a nakonec principy návrhu. Podle autorů Braude a Bernstein (2016) softwarová architektura určuje, jak je daný software rozdělen do subsystémů nebo modulů a softwarových rozhraní mezi nimi. Brown (2016) pak tvrdí, že je to cokoli a vše, co souvisí s významnými prvky softwarového systému, od struktury a základů kódu až po úspěšné nasazení tohoto kódu do reálného prostředí.

Softwarovou architekturu lze tedy definovat jako proces, který je prováděný při návrhu softwaru. Hlavním cílem je na základě definovaných kroků a postupů vytvořit softwarový systém, který bude splňovat požadavky zákazníka. Za tímto účelem softwarová architektura sdružuje různé elementy, objekty, aplikace, systémy atd. Jejich konkrétní struktura a vazby mezi nimi vychází ze zvolené softwarové architektury. Zároveň je nutné zdůraznit, že kromě struktury je zde důležité brát v úvahu i chování, ale rovněž také funkčnost, výkon, odolnost, opětovné použití, srozumitelnost, ekonomická a technologická omezení atd. Jedná se tedy o snahu dosáhnout optimálního návrhu softwarového systému z hlediska jeho struktury, chování, vlastností a vazeb na vnější prostředí, tzn. jejich požadavky a omezení (Booch et al., 1999; Braude a Bernstein, 2016; Brown, 2016; Richards a Ford, 2020).

Pro každou softwarovou architekturu jsou zásadní tyto koncepty (Eeles a Cripps, 2011):

- role: kdo,
- výsledky práce: co,
- úkoly: jak,
- fáze, iterace a činnosti: kdy.

Pro všechny tyto koncepty by pak měly existovat i návody, šablony, příklady a postupy. Obecně pak mluvíme o metodě, která vše toto shrnuje, viz Obrázek 1 (Eeles a Cripps, 2011).



Obrázek 1. Klíčové koncepty metody a jejich vztahy. Zdroj: Eeles a Cripps (2011).

Jedním z klíčových vstupů, bez kterého by nebylo možné realizovat další fáze, jsou lidé a role, které mají v rámci softwarového projektu, resp. organizace, protože všechny role i mimo daný projekt se navzájem ovlivňují. Každá role má zodpovědnost za výsledek nebo výsledky práce, které jsou prováděny pomocí zadaných úkolů (Braude a Bernstein, 2016; Eeles a Cripps, 2011). V kontextu softwarové architektury je pak primární role architekta, který řeší analýzu a návrh architektury. Tato role se zpravidla nazývá softwarový architekt, když související role jsou vedoucí architekt, aplikační architekt, architekt infrastruktury, a datový architekt. Tyto role se někdy překrývají, případně splývají do jedné role, když vždy záleží na velikosti projektu (Eeles a Cripps, 2011; Richards a Ford, 2020).

Softwarový architekt je někdo, kdo rozumí kódu, ale zároveň má velké zkušenosti s tím, co a kolik toho přesně je k dosažení cíle potřeba skutečně udělat. Cílem softwarového architekta je zajistit, aby softwarový systém byl dobře navržený, pružný, snadno udržovatelný a splňoval požadavky zákazníka. Softwarový architekt se zaměřuje na technické aspekty softwarového systému a návrh jeho struktury (Brown, 2016). Konkrétní požadavky na tuto roli lze nalézt např. v Richards a Ford (2020).

Pro každou softwarovou architekturu je důležité i architektonické myšlení, kde jde o pochopení rozdílu mezi architekturou a designem a znalost toho, jak spolupracovat s vývojovými týmy, aby architektura fungovala. Zároveň je nutné mít širokou škálu technických znalostí a pochopit, analyzovat a sladit kompromisy mezi různými řešeními, technologiemi, a požadavky dalších rolí v projektu (Eeles a Cripps, 2011; Richards a Ford, 2020).

1.2 HISTORIE SOFTWAROVÝCH ARCHITEKTUR

1.2.1 Obecný pohled na historii SW architektur

V posledním desetiletí došlo k výraznému rozšíření rozsahu softwarové architektury, která nyní zahrnuje stále širší a širší oblasti zodpovědnosti a perspektiv. Dříve byl vztah mezi architekturou a provozem často založen na formálních smlouvách, což často vedlo k nadměrné byrokracii. Firmy často krát upřednostnili outsourcování provozu svých systémů třetím stranám, které měly za úkol poskytovat služby na základě smluvně dohodnutých parametrů, jako je dostupnost, škálovatelnost, odezva a další architektonické charakteristiky. Dnes jsou architektury, jako například mikroslužby, mnohem pružnější a integrují prvky, které dříve byly vyhrazeny pouze pro provozní část (Erl, 2016; Richards a Ford, 2020).

Architektura softwaru je neustále se vyvíjejícím a proměnlivým konceptem v důsledku rychle se rozvíjejícího vývoje softwaru a souvisejících požadavků. Definice, které platí dnes proto mohou být za pár let zastaralé. Softwarová architektura se zabývá rozhodnutími o základních strukturálních prvcích. Moderní architektonické styly jako mikroslužby umožňují postupné a méně nákladné úpravy. Je důležité si uvědomit, že softwarová architektura je dynamická a mění se spolu s vývojem softwaru (Braude a Bernstein, 2016; Erl, 2016).

V minulosti, kdy se začínal vyvíjet software, byla infrastruktura pro provoz aplikací velmi drahá. Lidé se snažili co nejefektivněji využít sdílené zdroje, jako jsou operační systémy, aplikační servery a databázové servery. To vedlo k vytvoření architektur, které sdílely tyto zdroje a centralizovaly funkcionalitu. Dnes jsou již moderní softwarové architektury, jako jsou mikroslužby a další, navrženy tak, aby zvládly rostoucí požadavky moderních aplikací na škálovatelnost, agilitu, bezpečnost a flexibilitu. K tomuto výrazně přispěl rozvoj cloud computingu, když architektury jako mikroslužby a serverless využívají cloudové zdroje pro škálování a řízení aplikací (Rajan, 2018; Richards, 2022).

Velké množství informací o softwarové architektuře má tedy velmi malý historický význam, který se týká hlavně provozu a údržby již vytvořených softwarových systémů. Mnoho zkratek a odkazů na rozsáhlé informační zdroje a znalosti, představují zastaralé nebo neúspěšné pokusy. Některá řešení, která byla před pár lety používána, již nemusí fungovat optimálně, protože se kontext a požadavky změnily. Dnes je proto důležité se zaměřit na moderní softwarové architektury a popřemýšlet nad některými dřívějšími předpoklady, protože se objevily nové technologie a postupy, které mohou nabítdnout lepší řešení (Richards, 2022; Richards a Ford, 2020).

1.2.2 Důležité body v historii těchto architektur

Jeden z důležitých bodů v historii softwarové architektury byl vznik architektonického stylu klient-server, který se začal prosazovat v 80. letech. Ten rozděluje aplikace na dvě části: serverovou část, která poskytuje služby a data, a klientovou část, která komunikuje se serverem a zobrazuje uživatelské rozhraní. Tato architektura umožnila decentralizaci a distribuci funkcionality a zlepšila škálovatelnost systémů (Braude a Bernstein, 2016).

Dalším významným bodem v historii byl vznik architektury třívrstvého modelu (n-tiers), který se stal populárním v 90. letech. Ta rozděluje aplikaci na tři vrstvy: prezentační vrstvu, logickou vrstvu a datovou vrstvu. Díky tomu může oddělit prezentaci, logiku a datový přístup a poskytuje větší flexibilitu při úpravách a rozšiřování systému (Mary a Rodrigues, 2012; Vaisman a Zimányi, 2014)

S rozvojem internetu a webových aplikací se prosadila architektura orientovaná na služby SOA. Ta je založena na principu poskytování funkcí jako samostatných služeb, které mohou být kombinovány a využívány v různých aplikacích a procesech. Umožňuje větší modularitu, znovupoužitelnost a integraci systémů (MuleSoft, 2013).

V posledních letech se stává stále populárnější architektura mikroslužeb. Mikroslužby jsou malé, nezávislé a samostatné služby, které spolupracují a komunikují pomocí lehkého protokolu. Tento přístup umožňuje vyšší škálovatelnost, rychlé nasazení změn a flexibilitu při vývoji a provozu aplikací (Erl, 2016; Richards, 2022).

1.3 POŽADAVKY NA SOFTWAROVOU ARCHITEKTURU

Jednou z klíčových dovedností architekta je schopnost pohlížet na architekturu z mnoha různých hledisek. Každé z těchto hledisek samostatně nemusí být úplně relevantní, ale jejich kombinací získáváme nadhled nad celkem. Tato hlediska zahrnují principy, standardy, pravidla a postupy, které jsou velmi důležité pro úspěch projektu (Brown, 2016).

Požadavky jsou velmi důležité, jelikož určují potřebné funkce systému a definují rozsah systému a jeho vlastnosti. Kdyby nebyly zadány, projekt by prakticky nikdy nebyl dokončen, jelikož by se daly přidávat nové a nové funkce. Daly by se rozdělit na tři základní směry, kterými se abstrakce ubírá (Brown, 2016; Eeles a Cripps, 2011; Richards a Ford, 2020). Funkcionální požadavky potřebných požadavků určují, co systém musí dělat, nefunkční požadavky určují, jak dobře to systém musí dělat, a omezení stanovují rámec, v němž je systém navrhován a vyvíjen (Ameller, 2014).

Funkcionální požadavky – Před tím, než se všichni pustí do práce při vytvoření nového softwaru, je potřeba zjistit kontext práce. Definují specifické funkce, úkoly nebo operace, které software musí vykonávat, aby splňoval očekávání uživatele a podnikového prostředí. Jsou to požadavky, které určují, co software musí dělat, a často se vyjadřují jako scénáře nebo use-case. Funkcionální požadavky jsou zaměřeny na to, co má být provedeno, a jak má být interakce s uživateli a systémem realizována. např. co všechno má obsahovat (možnost vyhledávat, psaní recenzí, …).

Nefunkcionální požadavky – Zaměřují se na kvalitu a charakteristiky softwarového systému. Tyto požadavky popisují očekávání v oblasti výkonu, spolehlivosti, bezpečnosti, škálovatelnosti, udržovatelnosti a dalších aspektů. Nefunkcionální požadavky mohou mít zásadní dopad na architekturu systému a jsou klíčové pro zajištění dobré uživatelské zkušenosti a provozuschopnosti softwaru. Jinak řečeno, jak by se měl systém chovat. Jedná se o požadavky typu „Systém má být schopný obsloužit tisíce uživatelů“, „Systém má fungovat 24/7“, nebo „Systém má mít krátkou odezvu“.

Specifikovaná omezení – definují podmínky a parametry, kterými musí softwarová architektura dodržovat. Omezení mohou vycházet z vnějších faktorů, jako jsou požadavky na použité technologie, omezení hardware nebo požadavky na kompatibilitu s existujícími systémy. Omezení mohou také vyplývat z podnikových směrnic, zákonů nebo standardů, které musí systém dodržovat (např. GDPR).

Kdyby byla práce zadána pouze na požadavcích, tak by bylo velmi vysoké riziko, že by konečný zákazník s finální prací nebyl spokojen. Vývojáři, kteří pracují na požadavcích zákazníka se snaží je doladit a vylepšit. Může se tedy stát, že na požadavku, který pro zákazníka nebyl tak důležitý se pracovalo důkladněji (a tím i déle), než bylo potřeba. Naopak požadavek, který je pro zákazníka klíčový a zákazník očekával, že tato funkce bude udělaná co nejlépe, je realizována jen průměrně nebo, i podprůměrně, protože jeho realizace nebyla pro vývojáře zábavná a zajímavá. Je tedy třeba si s koncovým zákazníkem pořádně projednat a ujasnit priorit. Zde je ideální cestou agilní vývoj, kdy je vyvíjený produkt i v rozpracovaném stavu pravidelně předváděn zákazníkovi a jsou okamžitě zapracovávány jeho připomínky a podněty (Ameller, 2014).

Když je již stanoveno, jaké jsou požadavky na projekt, je možné začít s výběrem a poté návrhem architektury.

1.4 VÝBĚR VHODNÉ SOFTWAROVÉ ARCHITEKTURY

Při výběru vhodné architektury je potřeba se zeptat na otázku, zda a jak se bude náš načrtnutý model rozšiřovat? Pokud ano, co přesně se bude rozšiřovat (např. množství dat, počet požadavků, …). Na tyto otázky je možné odpovídat dvěma způsoby. Lze využít buď horizontálního, nebo vertikálního škálování. Další otázkou by mohlo být, jak udělat systém rychlejší? Jeden z velmi často využívaných principů je využívání metody „cache“ (přednačtení, nebo předzpracování opakovaně používaných dat).

1.4.1 Horizontální škálování

Horizontální škálování, také známý jako scale-out, se zaměřuje na přidání dalších instancí (kopií) systému pro, zvýšení celkové kapacity. Při horizontálním škálování jsou nové instance nasazeny na více fyzických serverů nebo ve virtualizovaných prostředích. Systém je tak rozdělen mezi více samostatných instancí, které mohou pracovat paralelně a sdílet zátěž. Horizontální škálování se často využívá v distribuovaných prostředích a cloudových infrastrukturách (Blinowski et al., 2022).

Je ovšem potřeba dávat pozor, aby se nečelilo CAP teorému – pro distribuovaný datový sklad není možné poskytovat více než dvě záruky z těchto tří – konzistence, dostupnost a odolnost k přerušení (Gilbert a Lynch, 2012). Aby se přešlo tomuto problému je potřeba určit, zdali bude použita architektura bezestavová (stateless), nebo stavová (stateful).

Stavové škálování je přístup, který zachovává stav systému i při jeho škálování. To znamená, že všechny instance systému sdílejí společný stav a jsou schopny se vzájemně ovlivňovat. Ve stavově škálovaném systému se tedy musí zajistit synchronizace a sdílení stavových informací mezi všemi instancemi. Stavové škálování se často používá v systémech, které vyžadují udržování stavu, například databázové systémy nebo transakční aplikace (BasuMallick, 2023).

Bezstavové škálování je naopak při škálování neuchovává stav systému. Každá instance systému je nezávislá a nemá informace o stavu jiných instancí. Každá požadavek je zpracován odděleně a nezávisle na předchozích požadavcích. Je často používán v architekturách mikroslužeb nebo webových aplikacích, které jsou navrženy tak, aby byly bezstavové a vysoce škálovatelné. Tento přístup je jednodušší, protože není nutné řešit synchronizaci stavových informací. Nelze jej však využít vždy (BasuMallick, 2023).

1.4.2 Vertikální škálování

Vertikální škálování, známé také jako scale-up, se zaměřuje na zvýšení výkonu a kapacity jedné instance systému. Při vertikálním škálování se systém upgraduje na vyšší výkonný hardware, jako jsou serverové stroje s vyššími parametry (např. více procesory, více paměti). Tím se zvýší daná kapacita a výkon jediné instance systému (Blinowski et al., 2022).

1.4.3 Cache

Cache (mezipaměť) je komponenta, která uchovává data proto, aby v budoucnu mohla být stejná data použita znovu a tím pádem systém fungoval rychleji (Smith, 1982). Nicméně vytvořit systém, tak aby pracoval s cache tak, aby to pro daný systém bylo opravdu výhodné, může být poměrně složitá záležitost. Při nesprávném propojení se například může stát, že uložená cache ještě nedošla ke správnému požadavku, ale požadavek se již vykonává. V tuto situaci by cache pouze zabírala nadbytečné místo a systém by neurychlila (Richards, 2022; Richards a Ford, 2020).

1.5 METODY POROVNÁNÍ A VÝBĚRU SOFTWAROVÉ ARCHITEKTURY

Dnes je velké množství softwarových architektur a není vždy jasné, která se pro danou situaci hodí, proto se tato kapitola zabývá dvěma metodami, které mohou pomoci při rozhodování o výběru vhodné softwarové architektury.

1.5.1 SWOT analýza

SWOT analýza je nástroj pro strategickou analýzu, který se používá k posouzení silných stránek (Strengths), slabých stránek (Weaknesses), příležitostí (Opportunities) a hrozeb (Threats). V oblasti softwarové architektury je to např. nástroj pro analyzování vývoje a provozu dané softwarové aplikace. Zde jsou popsány jednotlivé prvky SWOT analýzy (Namugenyi et al., 2019):

Silné stránky (Strengths): jedná se o interní faktory, které popisují pozitivní aspekty softwarové architektury. To může zahrnovat efektivní využití návrhových vzorů, vysokou výkonnost, snadnou škálovatelnost nebo dobrou modularitu systému.

Slabé stránky (Weaknesses): interní faktory, které zahrnují technické nedostatky jako, složitost, která komplikuje údržbu, nedostatečnou dokumentaci nebo problémy s bezpečností a zabezpečením.

Příležitosti (Opportunities): jedná se o externí faktory, které by mohly pozitivně ovlivnit architekturu. To zahrnuje např. nové technologie, trendy v odvětví, potřeby uživatelů, které mohou být naplněny a vytvořením nových funkcí.

Hrozby (Threats): externí faktory, které by mohly negativně ovlivnit architekturu. To zahrnuje např. konkurenci, změny v požadavcích uživatelů, bezpečnostní rizika nebo nestabilitu třetích stran, na kterých je architektura závislá.

1.5.2 Analytický hierarchický proces

Analytický hierarchický proces (Analytic Hierarchy Process, AHP) slouží v rozhodování k optimálnímu určení toho, jakou alternativu zvolit. Jedná se o metodu, pomocí které lze určit softwarovou architekturu (alternativu) ke konkrétním definovaným požadavkům (kritériím), tzn., že je to metoda vícekriteriálního rozhodování. AHP má několik kroků, jsou jimi (De Fsm Russo et al., 2015; Saaty, 1988):

Definice problému: Jasně stanoví rozhodovací problém a také stanoví kritéria a alternativy, které mezi sebou budou porovnávány.

Vytvoření hierarchie: Kritéria a alternativy se uspořádají do hierarchické struktury.

Sestavení matice porovnání párů: Pro každou úroveň hierarchie se tvoří matice porovnání, ve které se porovnávají důležitosti jednotlivých prvků v rámci dané úrovně (alternativy a kritéria). Porovnání se provádí pomocí škály, obvykle od 1 do 9, kde číslo reprezentuje důležitost každého prvku (1 je stejná důležitost, 9 je absolutní důležitost).

Určení vah: Na základě matice porovnání se vypočítají váhy pro jednotlivé prvky hierarchie. Tyto váhy vyjadřují důležitost prvků v rámci celého problému.

Konzistence matice: Pro kontrolu konzistence porovnání se vypočítá tzv. poměrem konzistence (CR – consitency ratio). Pokud CR vychází méně než 10 %, tak je matice brána jako konzistentní.

Vážený součet: Slouží k vypočítání celkové váhy kritéria. Z váženého součtu lze poté zjistit, která z možných alternativ je optimální.

Aplikace a dokumentace: Pokud zjištěný výsledek vychází lépe jak momentální stav, tak je možné ho aplikovat a zdokumentovat pro případné budoucí účely.

1.6 ARCHITEKTURA VE VEŘEJNÉ SPRÁVĚ

Softwarová architektura ve veřejné správě dnes hraje důležitou roli především v rozhodování o nasazení a využívání cloud computingu, tzn. centralizované architektuře pro poskytování softwarových aplikací a souvisejících služeb (podpora) různým organizacím veřejné správy. Cloud computing tedy přináší veřejné správě mnoho výhod jako je snížení nákladů, zvýšení flexibility, škálovatelnost nebo snadnější přístup k nastavování a dodržování bezpečnostních opatření. Rozhodování o tom, zda a který typ nasazení cloud computingu využít, je však složité a vyžaduje zvážení různých faktorů (Ahn et al., 2015; Sallehudin et al., 2020).

Vláda České republiky schválila koncepci eGovernment cloudu, která stanovuje podmínky pro využívání cloudových služeb státními orgány. V rámci této koncepce budou poskytovány cloudové služby jak komerčními poskytovateli, tak státním poskytovatelem, za splnění předem stanovených bezpečnostních kritérií. Důvodem, proč stát potřebuje cloud computing, je snaha minimalizovat náklady a zefektivnit provoz informačních systémů. Budování a provoz vlastních serverových úložišť by bylo finančně náročné a složité. Cloudové služby umožňují státu nakupovat potřebné kapacity u specializovaných firem, čímž využívá přednosti cloud computingu. Státní data v cloudu budou rozdělena podle citlivosti. Nejcitlivější data zůstanou pod správou státu a budou umístěna v rámci státní části eGovernment cloudu, kterou provozuje státem vybraný poskytovatel. Ta budou muset zůstat na území České republiky. Méně citlivá data, jako jsou e-maily, budou moci být umístěna v komerčním cloudu, avšak i zde bude klíčové dodržovat bezpečnostní opatření a legislativu (Národní úřad pro kybernetickou a informační bezpečnost, 2023).

Zabezpečení dat v cloudu je klíčové a odpovědnost za bezpečnost dat nese především úřad, který data schraňuje. V rámci Úřadu a Ministerstva vnitra vznikají pravidla, která stanoví podmínky pro cloudové řešení a bezpečnostní požadavky. Bude existovat katalog prověřených dodavatelů cloudových technologií, kteří budou muset splnit bezpečnostní standardy. Vyhlášky o vstupních kritériích, bezpečnostních pravidlech a bezpečnostních úrovních budou stanovit konkrétní pravidla pro využívání cloudových služeb ve veřejné správě. Poskytovatelé musí splnit požadavky a orgány veřejné moci budou mít povinnost dodržovat bezpečnostní pravidla. Proces ex ante a ex post kontroly zajistí bezpečnost dat a minimalizuje rizika (Národní úřad pro kybernetickou a informační bezpečnost, 2023).

2 VYMEZENÍ SOFTWAROVÝCH A SYSTÉMOVÝCH ARCHITEKTUR V RŮZNÝCH KONTEXTECH

Tato kapitola se věnuje problematice softwarových architektur a jejich vztahu k několika klíčovým konceptům a technologiím, které v současnosti ovlivňují vývoj softwarových systémů, konkrétně UML, SOA a datové sklady. Obecně UML se zaměřuje na modelování softwarových systémů, SOA na návrh a organizaci služeb a datové sklady na shromažďování a analýzu dat.

2.1 VYMEZENÍ V KONTEXTU UML

UML je standardizovaný jazyk a notace používaná pro vizualizaci, návrh a dokumentaci softwarových a jiných systémů. Byl vyvinut jako společný jazyk pro modelování softwarových projektů a podporuje vizualizaci různých aspektů systému pomocí různých typů diagramů. UML zahrnuje mnoho diagramů, které lze použít k různým účelům. Zde jsou některé z typů diagramů v UML (Booch et al., 1999):

Use case diagramy: popisují interakce mezi uživateli (aktéry) a systémem prostřednictvím use casů, které reprezentují scénáře chování systému z pohledu uživatelů.

Class diagram: používají se k reprezentaci statické struktury systému. Zobrazují třídy a jejich vztahy, atributy a metody tříd a umožňují modelovat dědičnost, asociace, agregaci a kompozici.

Activity diagram: popisují chování systému jako posloupnost aktivit nebo kroků. Jsou často používány k modelování procesů, toků řízení nebo algoritmů.

Sequence diagram: ilustrují interakce mezi různými objekty nebo třídami v časovém sledu. Ukazují posloupnost zpráv, které jsou posílány mezi objekty a umožňují vizualizaci dynamiky systému.

UML poskytuje jednotný způsob modelování a umožňuje lepší komunikaci mezi členy týmu a zainteresovanými stranami během životního cyklu softwarového projektu. Diagramy v UML jsou často používány jak při analýze a návrhu systému, tak i při dokumentaci a údržbě hotového softwaru (Booch et al., 1999). Celkově UML zvyšuje efektivitu celého vývojového procesu a zlepšuje porozumění a řízení projektu (López-Sanz et al., 2008).

2.2 VYMEZENÍ V KONTEXTU SOA

Servisně orientovaná architektura (SOA) je metodou vývoje softwaru, která využívá softwarové komponenty nazývané služby k vytváření podnikových aplikací. Každá služba poskytuje konkrétní podnikovou funkci a služby mohou mezi sebou bezproblémově komunikovat přes různé platformy a programovací jazyky. Hlavním cílem SOA je podpora opakovaného využití služeb v různých systémech a možnost kombinovat nezávislé služby pro řešení složitých úkolů (Papazoglou a van den Heuvel, 2007).

Architektura se skládá z pěti vrstev. Vrstva uživatelského rozhraní umožňuje aplikaci pracovat s grafickým uživatelským rozhraním (GUI) pro koncové uživatele, kteří přistupují k aplikacím. Vrstva obchodních procesů pracuje s případy použití z pohledu podnikání v rámci aplikace. Vrstva služeb pracuje se službami pro celou firmu ve skladu služeb. Vrstva komponent služeb je využívána k vytváření služeb, jako jsou funkční a technické knihovny. Vrstva provozních systémů obsahuje datový model (Erl, 2016).

Služby jsou spojovány pomocí komunikačních protokolů, službových registrů, službových sběrnic a dalších technik, které umožňují identifikaci, nalezení, komunikaci a spolupráci mezi službami v rámci SOA. Tato architektura je často používaná ve veřejné správě, např. registrační systémy, správa dokumentů, elektronické zdravotnické záznamy apod. (Erl, 2016; Papazoglou a van den Heuvel, 2007).

2.3 VYMEZENÍ V KONTEXTU DATOVÝCH SKLADŮ

Datový sklad (Data Warehouse) je digitální úložný systém, který propojuje a sjednocuje rozsáhlé množství dat z různorodých zdrojů. Využívá se ho převážně pro to, aby podniky byly konkurenceschopné. Jeho cílem je poskytovat informace pro obchodní inteligenci, reporting, analýzy a podporovat dodržování regulací, aby organizace mohly efektivně využívat svá data a rozhodovat se na základě datových poznatků. Pokud systém zanikne, zaniknou i tato data, tzn., že neslouží k jejich zálohování (Vaisman a Zimányi, 2014).

Architektura datových skladů se skládá ze tří úrovní, tedy nejvyšší, střední a spodní úroveň. Nejvyšší úroveň tvoří front-end klient, který prezentuje výsledky prostřednictvím nástrojů pro reporting a analýzu dat. Střední úroveň zahrnuje analytický engine, který slouží k přístupu a analýze dat. Spodní úrovní architektury je databázový server, kde jsou data načtena a uložena. Data, ke kterým se často přistupuje, jsou ukládána na rychlém uložišti, naopak data, ke kterým se přistupuje zřídka, je možné uložit do pomalejšího rozsáhlejšího uložiště. Pokud datový sklad

zjistí, že nějaká data jsou používaná častěji, tak zajistí, že tato data, jsou přesunuta do rychlého úložiště, aby byla optimalizována rychlost dotazů (Vaisman a Zimányi, 2014).

2.4 PROPOJENÍ UML, SOA A DATOVÝCH SKLADŮ

Moderní softwarová architektura využívající kombinaci UML, SOA a datových skladů je založena na několika klíčových krocích. Jedná se o identifikaci a modelování, poté koordinace a interakce mezi službami, následovně datové sklady jako takové, dále integrace a správa, také řízení a monitorování.

Prvním krokem je identifikace a modelování jednotlivých služeb pomocí UML diagramů, jako je Use Case diagram, Class diagram a diagram aktivit. Tyto diagramy umožňují přesně specifikovat funkcionalitu a interakce mezi službami. Dalším důležitým aspektem této architektury je koordinace a interakce mezi službami. UML sekvenční diagramy se využívají k vizualizaci a popisu toku dat a komunikace mezi jednotlivými komponentami. Tím se zajišťuje efektivní a koordinovaná spolupráce mezi službami v rámci celého systému.

Datové sklady mají také klíčovou roli v této architektuře. Class diagramy sloužící k modelování struktury datových skladů a jejich fyzického nasazení. Diagramy zobrazují entitní třídy, vztahy mezi nimi a také, jak se služby připojují k těmto datovým skladům.

Integrace a správa dat jsou dalšími důležitým aspektem této architektury. UML diagramy se využívají k modelování mechanismů pro propojení a správu datových zdrojů. Diagramy umožňují přesun dat mezi různými systémy a jejich transformaci do formátu, který je vhodný pro datové sklady. Díky tomu je zajištěna konzistence a kvalita dat, která jsou důležitá pro správnou funkci celého systému. Řízení a monitorování jsou nedílnou součástí této architektury. UML diagramy slouží k návrhu mechanismů pro sledování výkonu, správu chyb a logování událostí. Tím je zajištěna celková správnost a efektivita systému.

Při propojení těchto systémů vytváří softwarová architektura strukturu a základy pro návrh, implementaci a správu pokročilých softwarových systémů. UML diagramy slouží k vizualizaci a dokumentaci architektury, SOA poskytuje metodu pro vytváření a propojování služeb, a nakonec datové sklady slouží k centralizovanému ukládání a správě dat (Brown, 2016; López-Sanz et al., 2008; Richards a Ford, 2020; Vaisman a Zimányi, 2014).

3 MODERNÍ SOFTWAROVÉ ARCHITEKTURY A JEJICH UTILIZACE

Tato část se soustředí na typy jednotlivých architektur, jejich vysvětlení, jak fungují a kdy se využívají. Konkrétní architektury mohou být používány v různých kontextech a technologiích a jejich použití vždy omezené pouze na jedno odvětví. Volba architektury závisí na konkrétních požadavcích, technologiích a preferencích vývojářů.

3.1 EDA ARCHITEKTURA

Událostmi řízená architektura (EDA) řídí chování aplikace na základě událostí, jako jsou změny dat nebo uživatelské akce. Model se skládá z tří komponent – poskytovatel, přesměrovatel, příjemce (Richards, 2022).

Poskytovatel je komponenta nebo systém, který generuje události. Může jít o různé zdroje, například senzory, uživatelské interakce, externí systémy apod. Poskytovatel vytváří a publikuje události do systému. Tyto události jsou často asynchronní a mohou obsahovat informace o stavu, změně nebo akci, která se stala.

Přesměrovatel slouží jako prostředník mezi poskytovatelem a příjemcem událostí. Jeho úkolem je přijímat události od poskytovatelů a přesměrovávat je k příslušným příjemcům. Má na starosti distribuci událostí a zajištění, aby byly doručeny těm příjemcům, kteří jsou pro ně relevantní. Může také poskytovat další funkce, jako je řízení fronty událostí, filtraci nebo směrování založené na určitých pravidlech.

Příjemce je komponenta, která se přihlašuje k příjmu určitých událostí. Specifikuje své zájmy a preference ohledně typů událostí, které chce přijímat. Po zjištění, že přijímá informaci, dostane relevantní události od přesměrovatele a provede s nimi požadované akce. Příjemce může být implementován jako samostatná aplikace, služba nebo modul v rámci většího systému. Je zodpovědný za zpracování přijatých událostí a případnou odezvu na ně.

Obecně, poskytovatel vytváří eventy, které budou přesměrovány skrze přesměrovatele, k příjemci.

Tato architektura se využívá v systémech zpracovávajících velké objemy dat, komplexních workflow, nebo v systémech reagujících na události. Může se uplatnit například ve veřejné správě, v oblasti finančních služeb nebo u systémů monitorování (Richards, 2022).

3.2 MVC ARCHITEKTURA

MVC architekturu popsat složkami View, Controller a Model. Uživatel používá část View, má nějaké prostředí, ve kterém se dokáže orientovat a s ním pracuje. Pokud něco vyžádá/chce (Request), tak tím aktivuje část Controller a Model. View pošle data Modelu (např. neuloženou část dokumentu). Controller podá informaci Modelu, aby na požadavku pracoval (např. ulož soubor) a zároveň zaúkoluje Model, aby podal části View (uživateli) vědět, že požadavek byl zpracován. Z uživatelského pohledu tedy většinou uživatel v části View pouze vidí, že mu Model poslal zprávu (např. po uložení souboru se zobrazí zpráva „soubor byl úspěšně uložen“ nebo je tato skutečnost indikována jinak např. skrytím ikony rozpracovaného dokumentu „tužka“) (Deacon, 2009).

Popis jednotlivých komponent:

Model – Obsahuje veškerá data a logiku prace s nimi.

View – Ukazuje data uživateli a poskytuje mu takové prostředí, se kterým s nimi dokáže pracovat.

Controller – Vytváří komunikaci mezi View a Modelem. Proto potřebuje zpracovávat uživatelské vstupy, provádět logiku aplikace a upravovat data v modelu.

Tento typ architektury se často používá v mnoha webových frameworcích a technologiích pro vývoj webových aplikací (Deacon, 2009).

3.3 MVVM ARCHITEKTURA

Velmi podobné jako MVC architektura, rozdílem je, že místo controlleru, se zde nachází View-Model komponenta. Rozebrání jednotlivých komponent:

Model: Stejně jako v případě MVC, reprezentuje data a logiku aplikace.

View: Zde je prezentováno uživatelské rozhraní, které vizualizuje data zobrazená uživateli.

ViewModel: ViewModel je prostředník mezi modelem a pohledem. Obsahuje logiku pro zpracování uživatelských akcí a transformaci dat z modelu do podoby, kterou je snadné zobrazit. ViewModel také umožňuje dvousměrnou vazbu mezi pohledem a modelem.

Často se využívá při vývoji desktopových nebo mobilních aplikací s uživatelským rozhraním (Stonis, 2022).

3.4 MIKROSLUŽBY

Mikroslužby (microservices) jsou architektonickým přístupem k vývoji softwarových systémů, který staví na principu rozdělení aplikace do menších, nezávislých a samostatně nasaditelných komponent nazývaných mikroslužby. Každá mikroslužba je navržena tak, aby plnila specifickou funkčnost a mohla být vyvíjena, nasazována a škálována nezávisle na ostatních mikroslužbách. Pro lepší pochopení je důležité si představit pár základních vlastností mikroslužeb (Erl, 2016; Richards a Ford, 2020).

Oddělená zodpovědnost: Každá mikroslužba je zodpovědná za konkrétní funkčnost nebo oblast systému. To umožňuje týmům pracovat nezávisle na sobě a rychle reagovat na požadavky a změny.

Nezávislé nasazování: Mikroslužby lze nasazovat nezávisle na sobě. To znamená, že změny nebo opravy v jedné mikroslužbě nemají vliv na ostatní mikroslužby, což usnadňuje nasazování nových funkcí a oprav.

Škálovatelnost: Každou mikroslužbu lze škálovat nezávisle na ostatních. To znamená, že je možné přidat nebo odstranit instance jednotlivých mikroslužeb podle potřeby a efektivně využívat zdroje.

Flexibilita technologií: Mikroslužby umožňují používat různé technologie a jazyky pro každou mikroslužbu, což umožňuje využít nejvhodnější nástroje pro konkrétní úlohu.

Škálování týmů: Díky oddělené zodpovědnosti za jednotlivé mikroslužby je možné rozdělit týmy podle funkcionalit a umožnit jim pracovat nezávisle. To zlepšuje spolupráci, agilitu aefektivitu vývojových týmů.

Odpovědnost za životní cyklus: Každá mikroslužba má svůj vlastní životní cyklus, včetně vývoje, nasazování, provozu a údržby. Tím se zvyšuje zodpovědnost týmů za jejich vlastní služby.

Často je využívaná při vývoji a nasazování rozsáhlých, komplexních aplikací. Organizace, které mají složité podnikové systémy, často využívají mikroslužby pro oddělení funkcionalit do samostatných, nezávislých služeb (Stonis, 2022).

3.5 SERVERLESS ARCHITEKTURA

Serverless architektura je moderní přístup k návrhu a vývoji softwarových aplikací, který se zaměřuje na minimalizaci operací a správy infrastruktury a umožňuje vývojářům se soustředit

především na psaní kódu a logiku aplikace. Tento kód a logika aplikace na serveru však stále běží, nicméně úlohy související se správou a zřizováním infrastruktury nejsou pro vývojáře viditelné. Vývojáři se poté mohou soustředit přímo na kód a logiku, díky tomu se značně ulehčí práce a tím pádem i urychlí práce. Jedná se tedy o architekturu, kde přenesu velkou část úkolů na poskytovatele třetí strany, proto abych je nemusel řešit. Nejčastěji se používá u cloudových aplikací, webové mobilní aplikace (API služby) a poslední dobou čím dál více populární chatboti (Rajan, 2018; Stonis, 2022).

Function as a service (FaaS) – jedná se o druh serverless architektury, kde vývojáři píší kód aplikace jako soubor oddělených funkcí. Každá funkce vykonává konkrétní úlohu po spuštění událostí, jako je příchozí e-mail nebo HTTP požadavek. Po standardním testování vývojáři nasadí své funkce spolu se spouštěči do účtu poskytovatele cloudu. Když je funkce vyvolána, poskytovatel cloudu buď provede funkci na běžícím serveru, nebo, pokud momentálně žádný server neběží, spustí nový server pro provedení funkce. Tento proces provádění je abstrahován od pohledu vývojářů, kteří se soustředí na psaní a nasazování kódu aplikace (Rajan, 2018; Stonis, 2022).

4 SROVNÁNÍ ARCHITEKTUR Z HLEDISKA JEJICH VÝHOD A NEVÝHOD

V současné digitální éře, kdy se technologie neustále rozvíjejí, je výběr vhodné softwarové architektury pro konkrétní projekt zásadním rozhodnutím. Každá architektura přináší své vlastní výhody a omezení, a proto je klíčové pečlivě zvážit, která z nich nejlépe vyhovuje specifickým potřebám a cílům projektu. Tato kapitola se zaměřuje na porovnání softwarových architektur s cílem navrhnout řešení konkrétních příkladů pro využití ve veřejné správě (konkrétně pro Krajský úřad).

Jako srovnávané alternativy jsou použity výše popsané moderní softwarové architektury, tzn. EDA, MVC, MVVM, mikroslužby a serverless architektura. Pro jejich porovnání a následnému výběru vhodné architektury jsou využity dvě metody – SWOT analýza a AHP model.

SWOT analýza pomůže identifikovat silné stránky, slabé stránky, příležitosti a hrozby každé architektury v kontextu specifických potřeb a požadavků kraje. Díky tomuto přístupu bude snadnější pochopit, jaké benefity a omezení přináší jednotlivé architektury. AHP model pak poskytne kvantitativní nástroj pro porovnání a vážení klíčových faktorů, které jsou pro danou situaci nejdůležitější. Tímto způsobem lze objektivněji určit, která architektura nejlépe splňuje požadavky na **škálovatelnost, implementaci, komplexnost, závislost/nezávislost, znovupoužitelnost a řízení dat,** a také jak architektury zohledňující předpokládaný nízký počet uživatelů, který je v tomto případě aplikací, využívaných jen interními zaměstnanci krajského úřadu, ale i pro další úzce specializované aplikace typický.

4.1 DEFINOVÁNÍ KRITÉRIÍ PRO POROVNÁNÍ

Na základě studia literatury a dalších zdrojů informací byla zvolena následující kritéria pro porovnání vybraných softwarových architektur: **škálovatelnost**, **implementace**, **komplexnost**, **závislost/nezávislost**, **znovupoužitelnost** a **řízení dat**. Každé z těchto kritérií přináší konkrétní pohled na zhodnocení vybraných architektur a pomáhá lépe porozumět jejich vlastnostem a přínosům.

Škálovatelnost je klíčovým kritériem, protože chceme zvolit architekturu, která umožní snadno a efektivně přizpůsobovat mé aplikace vzrůstajícím požadavkům a zátěži. Podstatou tedy je, aby architektura umožňovala horizontální nebo vertikální škálovatelnost a zajistila tak plynulý provoz a výkon systému.

Implementace je dalším důležitým kritériem, který je nutné zohlednit. Architektura by měla být snadno implementovatelná a umožnit rychlý vývoj aplikací. Důležité tedy je, aby architektura poskytovala strukturovaný přístup a podporovala modulární design, to totiž usnadňuje vývojářům práci a zkracuje čas potřebný k nasazení nových funkcionalit.

Komplexnost je dalším kritériem, který je nutné uvažovat. Řeší totiž, jak složité je navrhovat, spravovat a udržovat danou architekturu. Je vhodné najít rovnováhu mezi dostatečnou flexibilitou a jednoduchostí. Architektura by měla být dostatečně robustní a škálovatelná, aniž by zbytečně zvyšovala složitost vývoje a údržby.

Závislost/nezávislost je následujícím kritériem. Softwarová architektura by měla minimalizovat závislosti mezi různými částmi systému. Podstatou je, aby jednotlivé služby nebo komponenty mohly pracovat nezávisle na sobě, to usnadňuje jejich nasazení, škálování a údržbu.

Znovupoužitelnost je ve veřejné správě taky důležitá, tzn., že podporuje znovupoužitelnost komponent a modulů. Z důvodu, aby bylo možné jednotlivé části architektury použít opakovaně v různých kontextech, šetří čas, úsilí a zvyšuje efektivitu vývoje.

Posledním kritériem je **řízení dat**. To, co je důležité, je, jak daná architektura spravuje a řídí datové toky mezi jednotlivými komponentami. Hledáme architekturu, která umožňuje efektivní správu dat, synchronizaci a sdílení informací v rámci systému.

4.2 VÝHODY A NEVÝHODY JEDNOTLIVÝCH ARCHITEKTUR

Popis výhod a nevýhod opět vychází ze studia zdrojů použitých v této práci.

4.2.1 Výhody a nevýhody EDA

Výhody: Škálovatelnost: EDA umožňuje efektivní škálování systému, protože zpracování událostí lze snadno distribuovat a paralelizovat. Implementace: Navrhování aplikací v EDA je flexibilní a umožňuje snadnou integraci nových komponent nebo služeb. Řízení dat: Událostmi řízená architektura usnadňuje synchronizaci dat mezi různými komponentami a zajišťuje konzistenci systému.

Nevýhody: Komplexnost: Správa a sledování událostí může být složitá, zejména v případě rozsáhlých a distribuovaných systémů. Závislost/nezávislost: Komponenty v EDA jsou

vzájemně provázány prostřednictvím událostí, což může vést k větší závislosti a složitosti mezi nimi.

4.2.2 Výhody a nevýhody MVC

Výhody: Škálovatelnost: MVC umožňuje snadnou škálovatelnost aplikace díky oddělení logiky, dat a prezentace. Implementace: Poskytuje jasnou a strukturovanou metodiku pro implementaci aplikací. Znovupoužitelnost: Kontrolery a modely lze znovu použít v různých částech aplikace, což zvyšuje efektivitu vývoje.

Nevýhody: Komplexnost: MVC může být náročné na správu a udržování v případě velkých a komplexních aplikací. Závislost/nezávislost: Prezentační vrstva (view) a logika (controller) jsou pevně propojeny, což může omezovat nezávislost komponent.

4.2.3 Výhody a nevýhody MVVM

Výhody: Škálovatelnost: Díky oddělení prezentace od logiky je možné snadno rozšiřovat a přizpůsobovat uživatelské rozhraní. Implementace: Poskytuje přehlednou strukturu a jednoduchou implementaci aplikací, zejména v kombinaci s moderními frameworky. Znovupoužitelnost: ViewModely lze opakovaně použít v různých částech aplikace, což usnadňuje vývoj a údržbu. Řízení dat: MVVM poskytuje mechanismy pro efektivní řízení datových toků mezi modelem, view a view modelem.

Nevýhody: Komplexnost: Koncept MVVM vyžaduje určitou úroveň porozumění a znalostí, což může být pro některé vývojáře náročné. Závislost/nezávislost: ViewModely jsou často úzce propojeny s konkrétním view, což může omezovat jejich nezávislost.

4.2.4 Výhody a nevýhody mikroslužeb

Výhody: Škálovatelnost: Mikroslužby umožňují nezávislé škálování jednotlivých služeb, což zajišťuje flexibilitu a výkon v prostředí s proměnlivým provozem. Implementace: Díky modularitě a nezávislosti jednotlivých služeb je možné rychle nasazovat nové funkcionality a aktualizace. Závislost/nezávislost: Mikroslužby jsou navrženy tak, aby byly nezávislé na sobě, což umožňuje vyšší odolnost systému a snadnou výměnu či aktualizaci jednotlivých služeb.

Nevýhody: Komplexnost: Správa a koordinace velkého počtu mikroslužeb může být náročná a vyžadovat sofistikovanou infrastrukturu. Řízení dat: Udržování konzistence dat mezi mikroslužbami vyžaduje řešení jako event sourcing nebo centralizovaná správa dat.

4.2.5 Výhody a nevýhody serverless architektury

Výhody: Škálovatelnost: Serverless architektura umožňuje automatické škálování aplikace na základě požadavků, což eliminuje nutnost správy infrastruktury. Implementace: Vývoj aplikací v prostředí serverless je rychlý a efektivní, protože se můžete zaměřit přímo na vývoj funkcionality bez nutnosti řešit infrastrukturní detaily. Závislost/nezávislost: Jednotlivé funkce v serverless jsou nezávislé na sobě, což umožňuje snadnou výměnu a nasazení nových funkcí.

Nevýhody: Komplexnost: Serverless vyžaduje dodatečné úsilí při správě a řízení závislostí mezi jednotlivými funkcemi. Řízení dat: Správa stavu a komunikace mezi funkcemi v serverless může být náročná, zejména při zpracování většího objemu dat.

4.3 POROVNÁNÍ ARCHITEKTUR PRO KONKRÉTNÍ PŘÍPAD UŽITÍ

4.3.1 Porovnání architektur pro konkrétní příklad

S ohledem na aktuální potřeby veřejné správy je definován následující případ využití v kontextu moderních softwarových architektur.

- Webová aplikace pro kraj na dotace s nízkým počtem uživatelů s tím, že kraj chce vše řešit interně.

Jak je patrné z Tabulky 1, tak EDA se ukázala jako dobrá volba pro aplikaci s očekávaně nízkým počtem uživatelů a dynamicky se měnícími požadavky, protože umožňuje snadné škálování a integraci nových komponent.

Tabulka 1: SWOT analýza pro EDA. Zdroj: vlastní.

Silné stránky (S)	Slabé stránky (W)	Příležitosti (O)	Hrozby (T)
Škálovatelnost: Distribuované a paralelní zpracování událostí umožňuje snadné škálování aplikace s rostoucím počtem uživatelů.	**Komplexnost:** Správa a sledování událostí může být složitá, zejména u rozsáhlých a distribuovaných systémů.	**Flexibilita implementace:** Možnost implementovat nové komponenty nebo služby v reálném čase a bez významných zásahů do stávající infrastruktury.	**Správa událostí:** Náročnost správy událostí a jejich synchronizace může vést k chybám a nekonzistenci dat.
Implementace: Navrhování aplikací v EDA je flexibilní a umožňuje snadnou integraci nových	**Závislost/nezávislost:** Komponenty v EDA jsou vzájemně provázány prostřednictvím	**Zvýšená flexibilita:** Možnost využití různých technologií a nástrojů pro implementaci	**Náročnost správy událostí:** Správa a sledování událostí může vyžadovat

Silné stránky (S)	Slabé stránky (W)	Příležitosti (O)	Hrozby (T)
komponent nebo služeb.	událostí, což může vést k větší závislosti a složitosti mezi nimi.	asynchronních komponent.	sofistikované nástroje a infrastrukturu.
Řízení dat: Událostmi řízená architektura usnadňuje synchronizaci dat mezi různými komponentami a zajišťuje konzistenci systému.		**Rozšiřitelnost:** Možnost snadného přidávání nových funkcionalit do systému prostřednictvím nových událostí.	**Zvýšená náročnost na testování:** Asynchronní povaha architektury může ztížit testování a ladění.

Na základě analýzy v Tabulce 2 lze říci, že MVC nabízí strukturovaný a jasný přístup k implementaci aplikací, což může usnadnit vývoj.

Tabulka 2: SWOT analýza pro MVC. Zdroj: vlastní.

Silné stránky (S)	Slabé stránky (W)	Příležitosti (O)	Hrozby (T)
Implementace: Poskytuje strukturovaný a jasný přístup k implementaci aplikací.	**Komplexnost implementace:** Větší složitost při správě a udržování v případě velkých a komplexních aplikací.	**Strukturovaný přístup:** Možnost jednoduchého a rychlého přidávání nových komponent nebo funkcionalit.	**Náročnost udržování:** Nárůst nákladů a času na údržbu a správu aplikace v průběhu času.
Znovupoužitelnost: Kontrolery a modely mohou být znovu použity v různých částech aplikace.	**Závislost mezi komponentami:** Pevné propojení mezi prezentační vrstvou (view) a logikou (controller) může omezovat nezávislost komponent.	**Možnost škálování:** Možnost snadného škálování aplikace pro různé uživatele.	**Omezená modularita:** může ovlivnit schopnost aplikace přizpůsobit se novým požadavkům.
Řízení dat: Snadná synchronizace dat mezi jednotlivými komponentami (model, view, controller).		**Jednoduchost implementace:** Jednoduchý design a implementace, což usnadňuje vývojářům práci a zkracuje čas potřebný k nasazení nových funkcionalit.	**Omezená flexibilita:** může bránit rychlému přizpůsobení se novým požadavkům a trendům.

Jak je patrné z Tabulky 3, tak MVVM se zdá být vhodný pro aplikaci s oddělením prezentace od logiky, což umožňuje snadnou úpravu uživatelského rozhraní.

Tabulka 3: SWOT analýza pro MVVM. Zdroj: vlastní.

Silné stránky (S)	Slabé stránky (W)	Příležitosti (O)	Hrozby (T)
Škálovatelnost: možnost snadno rozšiřovat a přizpůsobovat uživatelské rozhraní. (Díky oddělení prezentace od logiky)	**Komplexnost implementace:** tato architektura vyžaduje určitou úroveň porozumění a znalostí	**Oddělení prezentace od logiky:** Možnost snadného škálování aplikace při zvyšujícím se počtu uživatelů.	**Náročnost učení:** Nárůst nákladů na školení a vzdělávání vývojářů, aby správně implementovali a udržovali MVVM.
Implementace: Přehledná struktura a jednoduchá implementace aplikací, zejména v kombinaci s moderními frameworky.	**Závislost mezi komponentami:** ViewModely často úzce propojeny s konkrétním view, což může omezovat jejich nezávislost.	**Znovupoužitelnost:** Příležitost znovupoužití ViewModelů v různých částech aplikace, což usnadňuje vývoj a údržbu.	**Omezená podpora technologií:** Omezená podpora některých technologií nebo programovacích jazyků může omezovat možnosti implementace.
Řízení dat: Poskytuje mechanismy pro efektivní řízení datových toků mezi modelem, view a view-modelem.		**Řízení datových toků:** Možnost centralizovaného řízení datových toků pro lepší správu a synchronizaci.	**Omezení pro malé aplikace:** Pro malé aplikace může být MVVM nadměrně komplexní a nepřinést dostatečný užitek.

Z Tabulky 4 lze vyčíst, že mikroslužby poskytují vysokou flexibilitu a nezávislost jednotlivých služeb, což usnadňuje rychlý vývoj a aktualizace.

Tabulka 4: SWOT analýza pro mikroslužby. Zdroj: vlastní.

Silné stránky (S)	Slabé stránky (W)	Příležitosti (O)	Hrozby (T)
Implementace: Díky modularitě a nezávislosti jednotlivých služeb je možné rychle nasazovat nové funkcionality a aktualizace.	**Komplexnost implementace:** Správa a koordinace velkého počtu mikroslužeb může být náročná a vyžadovat sofistikovanou infrastrukturu.	**Nezávislé škálování:** U jednotlivých služeb umožňuje flexibilitu a výkon v prostředí s proměnlivým provozem.	**Nárůst nákladů na infrastrukturu:** S narůstajícím počtem mikroslužeb mohou náklady na infrastrukturu dramaticky vzrůst.
Škálovatelnost: Možnost nezávislého škálování jednotlivých služeb zvyšuje odolnost a výkon systému.	**Závislost mezi službami:** Existují závislosti a komunikace mezi mikroslužbami, které by mohli ovlivnit	**Flexibilita vývoje:** nasazování nových funkcionalit do mikroslužeb.	**Komplexnost správy:** Nárůst komplexity správy a monitorování velkého počtu mikroslužeb.

Silné stránky (S)	Slabé stránky (W)	Příležitosti (O)	Hrozby (T)
	nezávislost jednotlivých částí architektury.		
Závislost/nezávislost: Jednotlivé mikroslužby jsou navrženy tak, aby byly nezávislé na sobě, což umožňuje vyšší odolnost systému a snadnou výměnu nebo aktualizaci služeb.		**Znovupoužitelnost:** Možnost znovupoužití jednotlivých mikroslužeb v různých kontextech.	**Nárůst nákladů na testování:** S narůstajícím počtem mikroslužeb se zvyšují nároky na testování a validaci.

Z Tabulky 5 si lze povšimnout, že Serverless architektura nabízí snadné škálování aplikace a rychlý vývoj nových funkcionalit, avšak vyžaduje dodatečnou pozornost při správě závislostí a nákladech na infrastrukturu.

Tabulka 5: SWOT analýza pro serverless architekturu. Zdroj: vlastní.

Silné stránky (S)	Slabé stránky (W)	Příležitosti (O)	Hrozby (T)
Škálovatelnost: umožňuje automatické škálování aplikace na základě požadavků, a to eliminuje nutnost správy infrastruktury.	**Komplexnost implementace:** Dodatečné úsilí při správě a řízení závislostí mezi jednotlivými funkcemi.	**Automatické škálování:** Možnost automatického škálování aplikace dle momentální zátěže.	**Komplexnost správy:** Složitost správy a koordinace velkého množství funkcí v rámci serverless architektury.
Implementace: Vývoj aplikací je rychlý a efektivní, protože umožňuje vývojářům se zaměřit přímo na vývoj funkcionality bez nutnosti řešit infrastrukturní detaily.	**Závislost mezi funkcemi:** Správa stavu a komunikace mezi funkcemi může být náročná, zejména při zpracování většího objemu dat.	**Rychlý vývoj aplikací:** Příležitost rychlého vývoje a nasazení nových funkcionalit bez zásahu do infrastruktury.	**Omezená podpora technologií:** Omezená podpora některých technologií nebo programovacích jazyků může omezovat možnosti implementace.
Závislost/nezávislost: Jednotlivé funkce jsou nezávislé na sobě, a to umožňuje snadnou výměnu a nasazení nových funkcí.		**Nezávislé funkce:** Možnost nezávislého nasazení a spouštění funkcí.	**Bezpečnostní rizika:** Možná bezpečnostní rizika související s používáním třetích stran pro správu funkcí.

4.3.2 Rozhodovací model

4.3.2.1 Kriteriální tabulka

Na základě SWOT analýz provedených v předchozí kapitole byla vytvořena kriteriální tabulka, viz Tabulka 6. Kritéria 1.1 až 1.6 odpovídají kritériím z kapitoly 4.1, tzn. **škálovatelnost**, **implementace**, **komplexnost**, **závislost/nezávislost**, **znovupoužitelnost** a **řízení dat**. Pro hodnocení kritérií v tabulce pro jednotlivé alternativy je použita vlastní škála, která je vytvořená na základě struktury SWOT analýzy, tzn., že 1 = nejhorší až 4 = nejlepší.

Tabulka 6: Kriteriální tabulka pro příklad užití. Zdroj: vlastní.

Kriteriální tabulka						
Alternativy	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**
	MAX	**MAX**	**MAX**	**MAX**	**MAX**	**MAX**
EDA	3	3	2	2	3	2
MVC	2	4	1	3	2	3
MVVM	2	4	3	3	3	3
Mikroslužby	4	4	4	3	2	2
Serverless	4	4	4	2	2	3

Kritérium škálovatelnosti (1.1) se ukazuje, že architektury mikroslužeb a serverless dosahují nejvyššího ohodnocení, s hodnotou 4. Tyto architektury umožňují snadné a efektivní škálování v souladu s měnícími se požadavky. Zbývají EDA, MVC a MVVM, ty získávají nižší hodnocení (EDA 3; MVC 2; MVVM 2), protože jejich škálovatelnost je méně flexibilní v porovnání s mikroslužbami a serverless.

U implementace (1.2), MVC, MVVM a mikroslužby získávají nadprůměrné hodnocení (3), kvůli tomu, že tyto architektury nabízejí dobrou strukturovanost a relativně snadnou implementaci. Naopak potom EDA a serverless mají nižší hodnocení (2), protože implementace může být složitější kvůli asynchronní povaze EDA a specifickým výzvám u serverless architektury.

Co se týče závislost/nezávislosti (1.3), architektury mikroslužeb a serverless obdržely nejvyšší hodnocení (4), protože jsou navrženy tak, aby byly co nejvíce nezávislé. Dále MVC a MVVM obdržely nižší hodnocení (MVC 1; MVVM 3), protože mají silnější propojení mezi různými

komponentami. EDA byla ohodnocena středně (2), protože sice propojuje komponenty pomocí událostí, ale mohou vznikat složitější závislosti.

V oblasti znovupoužitelnosti (1.4) byly MVC, MVVM a mikroslužby ohodnoceny nadprůměrně (3), protože nabízejí nějakou úroveň znovupoužitelnosti komponent. EDA a serverless mají nižší hodnocení (2), protože mohou být méně vhodné pro opakované využití komponent.

Pokud jde o řízení dat (1.5), EDA získává nadprůměrné hodnocení (3), protože její událostmi řízená povaha umožňuje efektivní správu dat mezi komponentami. MVVM je hodnoceno nadprůměrně (3), protože má mechanismy pro řízení dat mezi modelem, view a view modelem. Mikroslužby a serverless mají nižší hodnocení (2), protože vyžadují složitější způsoby správy dat mezi službami nebo funkcemi.

Co se týče komplexnosti (1.6), MVC, MVVM a serverless architektura získávají hodnocení (2), což naznačuje, že tyto architektury nabízejí relativně umírněnou úroveň komplexnosti. Naopak EDA a mikroslužby obdržely nadrpůměrné hodnocení (3), na základě toho, že EDA může být složitější kvůli asynchronní povaze a specifickým výzvám serverless architektury a mikroslužby mohou být náročné na správu a koordinaci více služeb.

4.3.2.2 Podstatnost kritérií

Zde je popsáno pořadí, v jakém jsou kritéria pro definovaný příklad důležitá, kde 1) znamená nejdůležitější a 6) z vybraných kritérií nejméně důležité.

1) Implementace: Kvalitní implementace zajišťuje, že aplikace bude splňovat požadavky a bude spolehlivá.

2) Komplexnost: Zohlednění komplexity je důležité. Ačkoliv se očekává malý počet uživatelů, zbytečně složitá architektura nebo implementace by mohla způsobit problémy v údržbě a rozvoji aplikace.

3) Závislost-Nezávislost: Vzhledem k malému počtu uživatelů může být lehčí řešit některé závislosti a integrace interně. Nicméně je stále vhodné zohlednit možnosti budoucího rozšíření nebo změn.

4) Znovupoužitelnost: Je důležitá pro rychlejší vývoj a údržbu, ale pro malý kraj nemusí mít nejvyšší prioritou.

5) Řízení dat: I přes nízký počet uživatelů je důležité zajištění datové integrity a spolehlivost správy dat pro efektivní provoz aplikace.

6) Škálovatelnost: S ohledem na malý počet uživatelů, který je omezen krajem a interní povahu procesu může škálovatelnost být méně důležitá.

4.3.2.3 Sestavení matice pro porovnání párů kritérií

Pro vytvoření AHP modelu byl použit online nástroj AHP Priority Calculator vytvořený Klaus D. Goepel a dostupný na https://bpmsg.com/ahp/ahp-calc.php (Goepel, 2022).

Cat		Priority	Rank	(+)	(-)
1	Škálovatelnost	3.4%	6	1.3%	1.3%
2	Implementace	40.9%	1	13.3%	13.3%
3	Závislost-nezávislost	15.2%	3	3.6%	3.6%
4	Znovupoužitelnost	9.7%	4	3.3%	3.3%
5	Řízení dat	4.9%	5	1.5%	1.5%
6	Komplexnost	25.8%	2	6.7%	6.7%

	1	2	3	4	5	6
1	1	0.14	0.20	0.25	0.50	0.17
2	7.00	1	4.00	5.00	6.00	2.00
3	5.00	0.25	1	2.00	4.00	0.50
4	4.00	0.20	0.50	1	3.00	0.25
5	2.00	0.17	0.25	0.33	1	0.20
6	6.00	0.50	2.00	4.00	5.00	1

Number of comparisons = 15
Consistency Ratio CR = 3.8%

Principal eigen value = 6.241
Eigenvector solution: 5 iterations, delta = 6.6E-8

Obrázek 2: Porovnání kritérií navzájem. Zdroj: vlastní.

Na základě porovnání kritérií na Obrázku 2 analýza ukazuje přijatelnou hodnotu CR 3,8 %, která je nižší než hranice 10 %. Nejvýznamnějším kritériem je "Implementace", která představuje pevný základ pro spolehlivý provoz aplikace. „Komplexnost“, „Závislost-nezávislost“ a „Znovupoužitelnost“ následují a představují další podstatná kritéria pro tento konkrétní příklad. Zbylá kritéria „Řízení dat“ a „Škálovatelnost“ jsou umístěni níže, navíc jsou ještě sníženy o další stupeň z důvodu, že aplikace je mířena pro nízký počet uživatelů a tato aplikace bude omezena počtem obyvatel v kraji. Výsledek tedy odpovídá záměrům aplikace.

4.3.2.4 Porovnání architektur (alternativ) v kontextu kritérií

Škálovatelnost (1.1) – mikroslužby a serverless architektura jsou v rámci tohoto kritéria na stejné úrovni, zatímco EDA se blíží k této úrovni, a MVC a MVVM jsou na nižší prioritní úrovni, viz Obrázek 3.

Cat		Priority	Rank	(+)	(-)
1	EDA	15.3%	3	3.5%	3.5%
2	MVC	6.4%	4	0.8%	0.8%
3	MVVM	6.4%	4	0.8%	0.8%
4	Mikroslužby	35.9%	1	5.7%	5.7%
5	Serverless	35.9%	1	5.7%	5.7%

	1	2	3	4	5
1	1	3.00	3.00	0.33	0.33
2	0.33	1	1.00	0.20	0.20
3	0.33	1.00	1	0.20	0.20
4	3.00	5.00	5.00	1	1.00
5	3.00	5.00	5.00	1.00	1

Number of comparisons = 10
Consistency Ratio CR = 1.2%

Principal eigen value = 5.056
Eigenvector solution: 4 iterations, delta = 3.2E-9

Obrázek 3: Porovnání alternativ pro kritérium 1.1. Zdroj: vlastní.

Implementace (1.2) – Srovnání pomocí matice rozhodování ukazuje, že v oblasti implementace jsou MVC, MVVM, mikroslužby a serverless architektury na stejné úrovni a všechny získaly prioritní hodnocení. Na druhé straně má EDA mírně nižší prioritu, což může být důsledkem její asynchronní povahy a složitějšího charakteru implementace, viz Obrázek 4.

Cat		Priority	Rank	(+)	(-)
1	EDA	7.7%	5	0.0%	0.0%
2	MVC	23.1%	1	0.0%	0.0%
3	MVVM	23.1%	1	0.0%	0.0%
4	Mikroslužby	23.1%	1	0.0%	0.0%
5	Serverless	23.1%	1	0.0%	0.0%

	1	2	3	4	5
1	1	0.33	0.33	0.33	0.33
2	3.00	1	1.00	1.00	1.00
3	3.00	1.00	1	1.00	1.00
4	3.00	1.00	1.00	1	1.00
5	3.00	1.00	1.00	1.00	1

Number of comparisons = 10
Consistency Ratio CR = 0.0%

Principal eigen value = 5.000
Eigenvector solution: 1 iterations, delta = 0.0E+0

Obrázek 4: Porovnání alternativ pro kritérium 1.2. Zdroj: vlastní.

Závislost/nezávislost (1.3) – mikroslužby a serverless architektury mají podobný vliv na závislost a nezávislost, zatímco EDA má nižší prioritu kvůli potenciálně složitějším závislostem. MVC a MVVM jsou střední v tomto ohledu, viz Obrázek 5.

Cat		Priority	Rank	(+)	(-)
1	EDA	7.6%	4	2.3%	2.3%
2	MVC	3.9%	5	1.1%	1.1%
3	MVVM	16.1%	3	4.6%	4.6%
4	Mikroslužby	36.2%	1	7.3%	7.3%
5	Serverless	36.2%	1	7.3%	7.3%

	1	2	3	4	5
1	1	3.00	0.33	0.20	0.20
2	0.33	1	0.20	0.14	0.14
3	3.00	5.00	1	0.33	0.33
4	5.00	7.00	3.00	1	1.00
5	5.00	7.00	3.00	1.00	1

Number of comparisons = 10
Consistency Ratio CR = 3.0%

Principal eigen value = 5.136
Eigenvector solution: 4 iterations, delta = 5.6E-8

Obrázek 5: Porovnání alternativ pro kritérium 1.3. Zdroj: vlastní.

Znovupoužitelnost (1.4) - MVC, MVVM a mikroslužby nabízejí větší potenciál pro znovupoužitelnost komponent ve srovnání s EDA a serverless architekturou, viz Obrázek 6.

Cat		Priority	Rank	(+)	(-)
1	EDA	9.1%	4	0.0%	0.0%
2	MVC	27.3%	1	0.0%	0.0%
3	MVVM	27.3%	1	0.0%	0.0%
4	Mikroslužby	27.3%	1	0.0%	0.0%
5	Serverless	9.1%	4	0.0%	0.0%

	1	2	3	4	5
1	1	0.33	0.33	0.33	1.00
2	3.00	1	1.00	1.00	3.00
3	3.00	1.00	1	1.00	3.00
4	3.00	1.00	1.00	1	3.00
5	1.00	0.33	0.33	0.33	1

Number of comparisons = 10
Consistency Ratio CR = 0.0%

Principal eigen value = 5.000
Eigenvector solution: 1 iterations, delta = 0.0E+0

Obrázek 6: Porovnání alternativ pro kritérium 1.4. Zdroj: vlastní.

Řízení dat (1.5) - EDA a MVVM mají větší schopnost řídit data ve srovnání s ostatními architekturami. MVC, Mikroslužby a Serverless architektury jsou v této oblasti na srovnatelné úrovni, ale nemají tolik důrazu na řízení dat jako EDA a MVVM, viz Obrázek 7.

Cat		Priority	Rank	(+)	(-)
1	EDA	33.3%	1	0.0%	0.0%
2	MVC	11.1%	3	0.0%	0.0%
3	MVVM	33.3%	1	0.0%	0.0%
4	Mikroslužby	11.1%	3	0.0%	0.0%
5	Serverless	11.1%	3	0.0%	0.0%

	1	2	3	4	5
1	1	3.00	1.00	3.00	3.00
2	0.33	1	0.33	1.00	1.00
3	1.00	3.00	1	3.00	3.00
4	0.33	1.00	0.33	1	1.00
5	0.33	1.00	0.33	1.00	1

Number of comparisons = 10
Consistency Ratio CR = 0.0%

Principal eigen value = 5.000
Eigenvector solution: 1 iterations, delta = 0.0E+0

Obrázek 7: Porovnání alternativ pro kritérium 1.5. Zdroj: vlastní.

Komplexnost (1.6) – MVC, MVVM a Serverless architektura zde dosáhli nejlepších výsledků. Naopak oproti nim Mikroslužby a EDA dosáhli horších výsledků, viz Obrázek 8.

Cat		Priority	Rank	(+)	(-)
1	EDA	9.1%	4	0.0%	0.0%
2	MVC	27.3%	1	0.0%	0.0%
3	MVVM	27.3%	1	0.0%	0.0%
4	Mikroslužby	9.1%	4	0.0%	0.0%
5	Serverless	27.3%	1	0.0%	0.0%

	1	2	3	4	5
1	1	0.33	0.33	1.00	0.33
2	3.00	1	1.00	3.00	1.00
3	3.00	1.00	1	3.00	1.00
4	1.00	0.33	0.33	1	0.33
5	3.00	1.00	1.00	3.00	1

Number of comparisons = 10
Consistency Ratio CR = 0.0%

Principal eigen value = 5.000
Eigenvector solution: 1 iterations, delta = 0.0E+0

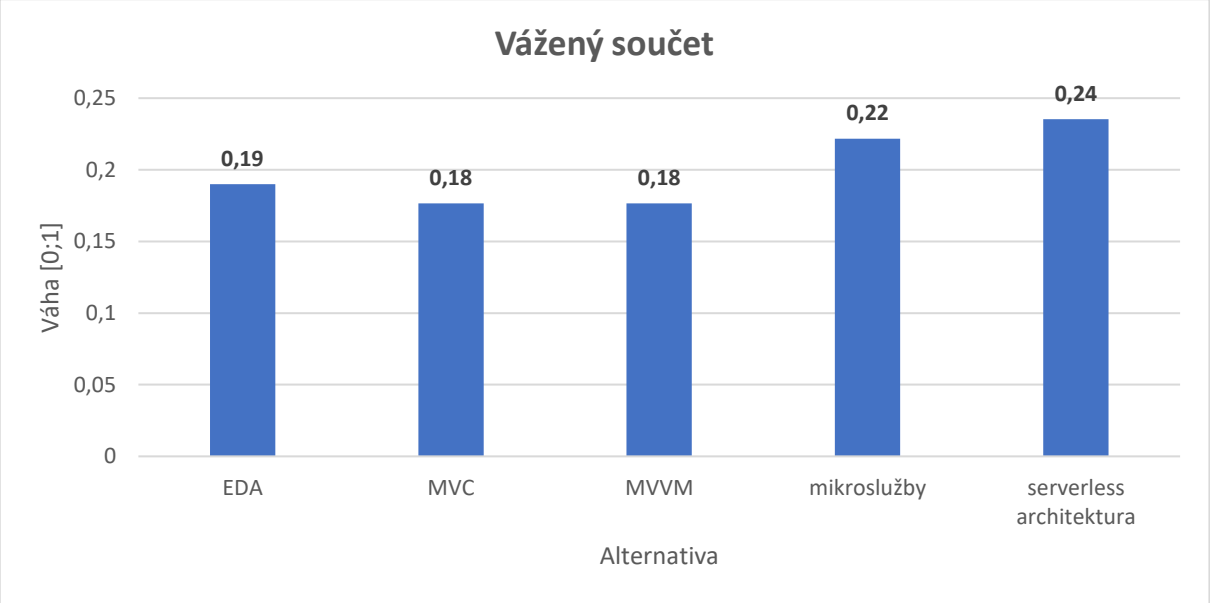
Obrázek 8: Porovnání alternativ pro kritérium 1.6. Zdroj: vlastní.

4.3.2.5 Výběr optimální alternativy pomocí váženého součtu

Pro výše definovaný příklad, tzn. vytvořit webovou aplikaci pro poskytování dotací na úrovni kraje s omezeným počtem uživatelů a s důrazem na interní řešení, byl proveden komplexní proces analýzy a srovnání různých softwarových architektur. Tento proces zahrnoval výhody a nevýhody architektur. Díky nim pak bylo možné vytvořit různé SWOT analýzy, kde byly identifikovány klíčové silné a slabé stránky, příležitosti a hrozby pro každou architekturu.

Na základě SWOT analýz byla vytvořena kriteriální tabulka, která reflektovala důležité faktory a požadavky jako jsou škálovatelnost, implementace, závislost/nezávislost, znovupoužitelnost, řízení dat a komplexnost. Váhy pro jednotlivá kritéria byly stanoveny s ohledem na jejich relativní důležitost pro příklad.

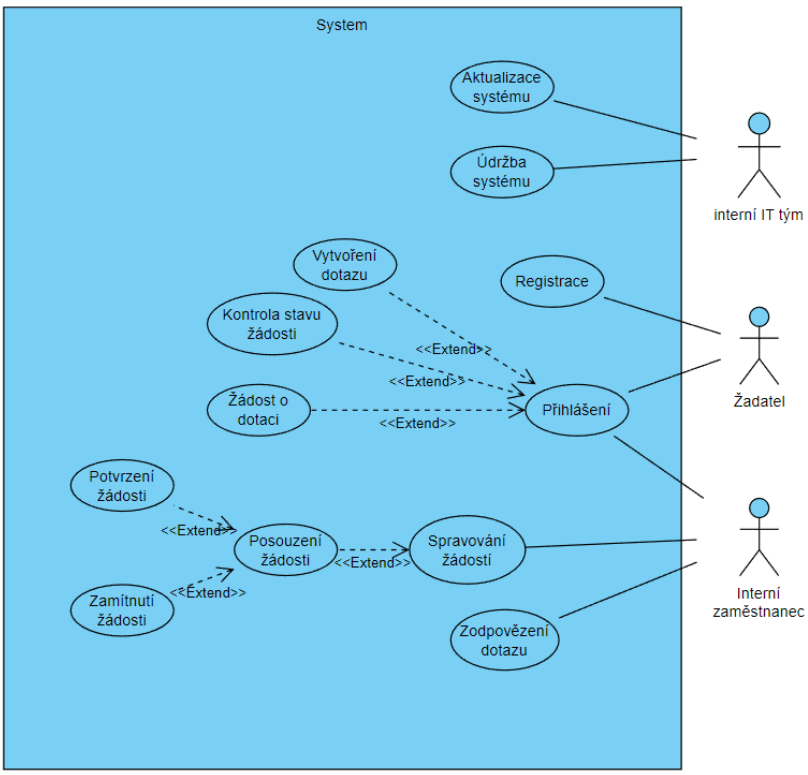
Následně byl použit AHP model k vyhodnocení a porovnání jednotlivých architektur. Výsledný vážený součet ukázal, že mezi vybranými architekturami "mikroslužby" a "serverless architektura" dosahují nejvyššího skóre, to tedy ukazuje, že by měli nejlépe vyhovovat potřebám pro tento konkrétní příklad. Serverless architektura dosáhla nejvyššího skóre, byla by tedy pro tento konkrétní příklad vybrána, viz Obrázek 9.



Obrázek 9: Vážený součet pro porovnávané alternativy. Zdroj: vlastní.

4.4 MODEL OPTIMÁLNÍ MODERNÍ SOFTWAROVÉ ARCHITEKTURY

V předchozí kapitole byla zvolena jako optimální alternativa serverless architektura, a proto je v této kapitole stručně představen její model pro definovaný příklad. S ohledem na význam UML při modelování softwarových architektur je na Obrázku 10 zachycen Use Case diagram, který zachycuje jednotlivé role v systému a jejich případy užití, tzn. interakce se softwarovou aplikací. Model na Obrázku 10 byl vytvořen ve webové aplikaci Visual Paradigm – Online Productivity Suite, která je dostupná na adrese https://online.visual-paradigm.com/. Model obsahuje tři role – interní IT tým (správce/admin), interní zaměstnanec a žadatel. Tento model slouží jako vstup do dalšího kroku.



Obrázek 10: UML Use Case ke konkrétnímu případu. Zdroj: vlastní.

S ohledem na cíl práce je na Obrázku 11 zobrazen pouze konceptuální model, který zachycuje klíčové komponenty softwarové architektury a jejich vzájemné vztahy. Model na Obrázku 11 byl vytvořen ve webové aplikaci Lucid visual collaboration suite, která je dostupná na adrese https://lucid.app/. Jako prostředí pro serverless architekturu byl zvolen Amazon cloud.

4.4.1 Proces Front-end

Uživatel si otevře webovou stránku, tím se nachází v prostředí Amplify, kde zadá/vybere to, co potřebuje z menu (např. žádost o dotaci). Request uživatele se pošle do Web Application Firewall, která zkontroluje, zda je request pro aplikaci bezpečný. Poté API odešle příkaz o tom, že se může spustit jakákoliv funkce spojená s tím, co uživatel zrovna potřebuje. Díky tomu se spustí část s databázemi a z ní, nebo do ní, se pošlou informace ohledně žádosti requestu. Prostředí Amplify se přizpůsobuje na základě oprávnění přihlášeného uživatele. Oprávnění uživatelů jsou rozdělena pomocí funkce Amazon Cognito, která funguje samostatně.

4.4.2 Proces Back-end

Z databáze (DynamoDB), přecházejí data do Amazon Elastic MapReduce, kde se analyticky zpracují do uložiště. Poté uložiště může posílat data zpět do query prvku (DynamoDB).

4.4.3 Jednotlivé prvky v architektuře

Amazon Cognito – spravuje oprávnění uživatelů.

Amazon Web Services Lambda – při jakémkoliv eventu se spustí kód – (např. když uživatel chce spustit formulář a klikne na něj v menu, spustí se formulář).

Amazon Simple Storage Service – uložiště (složka – pro konkrétní účel).

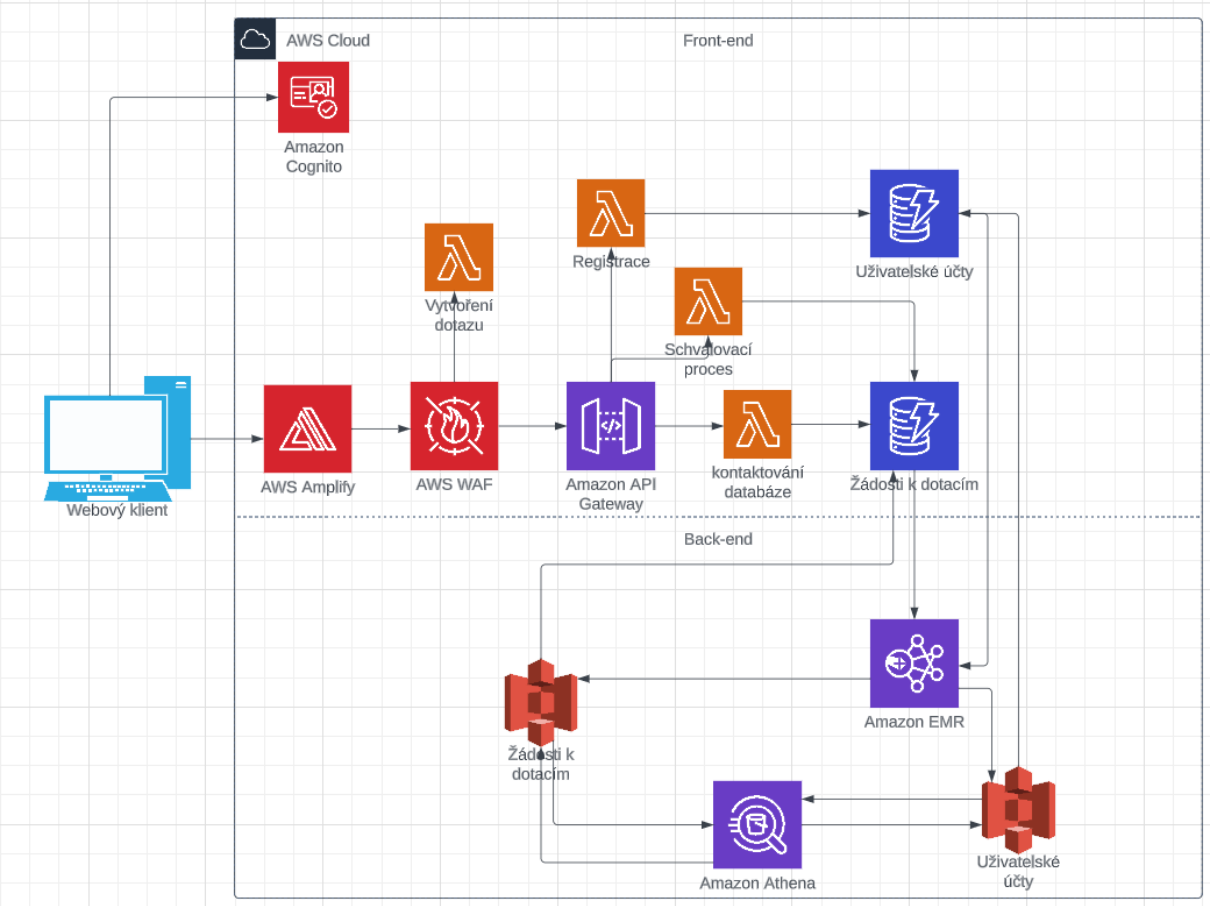
Amazon Web Services amplify – rozhraní pro uživatele.

Amazon Elastic MapReduce – analyticky zpracovává data.

Amazon Athena – vyhledávač – dotazování do databáze.

Amazon Web Services Web Application Firewall – kontrola komunikace, filtruje (ochrana) to co jde na cloud a co nejde.

API – komunikační rozhraní mezi klientem a příjemcem (službou).



Obrázek 11: Ukázková serverless architektura ke konkrétnímu příkladu. Zdroj: vlastní.

5 FORMULACE ZÁVĚRŮ A DOPORUČENÍ

V této kapitole jsou výše navržené modely vyhodnoceny a prodiskutovány závěry a doporučení, které posuzují přednosti daných architektur pro vybraný příklad.

Z analýzy vyplynulo, že každá architektura má své výhody a omezení, které mohou být relevantní v závislosti na konkrétních cílech a potřebách. Proces výběru optimální architektury je komplexní a může zahrnovat spoustu faktorů a pokud je nějaký pominut, může to mít velké důsledky pro výsledný softwarový produkt.

Při výběru architektury pro konkrétní příklad je tedy potřeba brát v potaz, že příklad byl zadán tak, aby ukázal základní myšlenkový proces softwarového architekta. Nicméně skutečný příklad by zahrnoval mnohem více podmínek a požadavků na architekturu, které by udělaly celý proces mnohem komplexnější a mohl by tak i změnit výsledky ve váženém součtu. Na výsledcích je vidět i to, že MVC a MVVM architektury si jsou dosti podobné a jelikož příklad nebyl z důvodu komplexnosti zadán s více podmínkami, tak jejich rozdíl nakonec není ve váženým součtu vidět, avšak v konkrétních porovnání v kritériích rozdíl vidět je.

Z navrženého modelu je patrné, že architektura funguje pouze na základním principu, avšak splňuje nejzákladnější nároky serverless architektury, takže pro názornou ukázku, jak by takový model mohl vypadat, se ukazuje jako vhodný příklad.

Pro vytváření softwarové architektury v podmínkách veřejné správy vyplývají ze závěrů této práce následující doporučení. Je vhodné se nejdříve podívat na to, co je konkrétně potřeba zjistit – funkcionální a nefunkcionální požadavky. Ty je potřeba případně omezit specifikovanými omezeními (jako jsou například finanční zdroje nebo zákonné požadavky), ale také, aby stále naplňovaly důvody, resp. cíle vytváření softwarové architektury. Dále je výhodné zjistit pomocí nějaké rozhodovací metody, jaká softwarová architektura je vhodná, proces modelu si pomocí dalšího nástroje nějak navrhnout (např. UML) a poté vytvořit danou softwarovou architekturu. Jak bylo také zjištěno, tak pro veřejnou správu hraje klíčovou roli cloud computing, který jí přináší mnoho výhod.

ZÁVĚR

Cílem práce bylo namodelovat vybrané moderní softwarové architektury ve smyslu srovnání navržených modelů a posouzení předností daných architektur. Za tímto účelem byly definovány jednotlivé kroky, které byly v předchozích kapitolách této práce úspěšně splněny.

V úvodních dvou kapitolách je popsáno, co vlastně takový pojem softwarové architektury je, na co se u ní zaměřit a co při její tvorbě pomáhá. Zároveň popisuje základní historii tohoto pojmu a poukazuje tak na pohled toho, že se svět velmi rychle vyvíjí a architektury, které jsou moderní dnes, nemusí být moderní i v dalších letech. Třetí kapitola následně popisuje moderní architektury, které lze používat ve veřejné správě. Ve čtvrté kapitole se poté získané informace používají k definování konkrétního příkladu a jeho řešení ve smyslu vytvoření rozhodovacích a dalších modelů sloužících pro pochopení dané problematiky. Poslední, pátá kapitola, obsahuje formulaci závěrů a doporučení se zaměřením na veřejnou správu.

Hlavní přínosy lze nalézt právě v úvodních kapitolách, kde je možné se dozvědět, jaký je myšlenkový pochod při tvorbě softwarové architektury a poté je možné se na jeden takový podívat ve čtvrté kapitole. Závěrem je, že kvalitně navrženou architekturu je možné udržovat moderní ještě několik let, protože se díky tomu do ní dají jednoduše implementovat právě moderní architektury i s tím, že je na ni dostatečně připravena.

Na základě zjištění z této práce lze doporučit, že implementace moderní softwarové architektury by měla být prováděna s ohledem na konkrétní požadavky projektu a tím pádem i na prostředí, ve kterém bude systém provozován. V dnešní době je potřeba neustálého vzdělávání a sledování trendů v oblasti softwarové architektury. Rychlý vývoj technologií vyžaduje pružnost a otevřenost k novým modelům. Pravidelná školení, nebo častý seberozvoj jsou velkou pomocí pro trvalý úspěch při navrhování a implementaci moderních softwarových systémů.

Celkově to lze shrnout tak, že správná volba a aplikace moderní softwarové architektury přináší mnoho výhod a zajišťuje konkurenceschopnost a udržitelnost softwarových projektů v dnešním rychlém technologickém prostředí.

POUŽITÉ ZDROJE

AHN, Jaesuk, et al. Open cloud architecture for public sector: Requirements and architecture. In: *2015 International Conference on Platform Technology and Service*. IEEE, 2015, p. 43-44.

AMELLER, David. *Non-Functional Requirements as drivers of Software Architecture Design*. Barcelona, 2014. Disertační práce. Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics.

BASUMALLICK, Chiradeep. *Stateful vs. Stateless: Understanding the Key Differences* [online]. 2023. [cit. 5.8.2023]. Dostupné z: https://www.spiceworks.com/tech/cloud/articles/stateful-vs-stateless/

BLINOWSKI, Grzegorz, OJDOWSKA, Anna a Adam PRZYBYŁEK. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 2022, 10: 20357-20374.

BOOCH, Grady, James RUMBAUGH a Ivar JACOBSON. *The Unified Modeling Language User Guide*. Massachusetts: Addison-Wesley Longman Inc., 1999. ISBN 0-201-57168-4.

BRAUDE, Eric J. a Michael E. BERNSTEIN. *Software Engineering: Modern Approaches*. Long Grove: Waveland Press, Inc., 2016. ISBN 978-1-4786-3230-6.

BROWN, Simon. *Software Architecture for Developers*. Lean Publishing, © 2012 - 2016.

DE FSM RUSSO, Rosaria; CAMANHO, Roberto. Criteria in AHP: A systematic review of literature. *Procedia Computer Science*, 2015, 55: 1123-1132.

DEACON, John. *Model-View-Controller (MVC) Architecture* [online]. 2009. [cit. 5.8.2023]. Dostupné z: https://shorturl.at/pruH1

EELES, Peter a Peter CRIPPS. *Architektura softwaru*. Brno: Computer Press, 2011. ISBN 978-80-251-3036-0.

ERL, Thomas. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Prentice Hall, 2016. ISBN 978-0-13-385858-7.

GILBERT, Seth a Nancy LYNCH. Perspectives on the CAP Theorem. *Computer*, 2012, 45.2: 30-36.

GOEPEL, Klaus D. *AHP Priority Calculator* [online]. 2022. [cit. 15.8.2023]. Dostupné z: https://bpmsg.com/ahp/ahp-calc.php

LÓPEZ-SANZ, Marcos, et al. *Modelling of Service-Oriented Architectures with UML*. Electronic Notes in Theoretical Computer Science, 2008, 194.4: 23-37.

MARY, S. Roselin a Paul RODRIGUES. Software architecture-evolution and evaluation. *International Journal of Advanced Computer Science and Applications*, 2012, 3.8: 82-88.

MULESOFT. *Evolution of SOA: How organizations develop agile SOA to rein in IT costs and secure competitive advantage* [online]. 2013. [cit. 5.8.2023]. Dostupné z: https://immagic.com/eLibrary/ARCHIVES/GENERAL/MULES_US/SOA-whitepaper.pdf

NAMUGENYI, Christine, NIMMAGADDA, Shastri L. a Torsten REINERS. Design of a SWOT analysis model and its evaluation in diverse digital business ecosystem contexts. *Procedia Computer Science*, 2019, 159: 1145-1154.

NÁRODNÍ ÚŘAD PRO KYBERNETICKOU A INFORMAČNÍ BEZPEČNOST. *FAQ* [online]. 2023. [cit. 5.8.2023]. Dostupné z: https://www.nukib.cz/cs/kyberneticka-bezpecnost/regulace-a-kontrola/faq/

PAPAZOGLOU, Mike P. a Willem-Jan VAN DEN HEUVEL. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 2007, 16: 389-415.

RAJAN, R. Arokia Paul. Serverless architecture-a revolution in cloud computing. In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. IEEE, 2018. p. 88-93.

RICHARDS, Mark. *Software Architecture Patterns*. Sebastopol: O'Reilly Media, Inc., 2022. ISBN 978-1-098-13427-3.

RICHARDS, Mark a Neal FORD. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol: O'Reilly Media, Inc., 2020. ISBN 978-1-492-04345-4.

SAATY, Thomas L. What is the analytic hierarchy process?. In: *Mathematical Models for Decision Support. NATO ASI Series, vol 48*. Springer, Berlin, Heidelberg, 1988, p. 109-121.

SALLEHUDIN, Hasimi, et al. Performance and key factors of cloud computing implementation in the public sector. *International Journal of Business and Society*, 2020, 21.1: 134-152.

SMITH, Alan Jay. Cache memories. *ACM Computing Surveys (CSUR)*, 1982, 14.3: 473-530.

STONIS, Michael. *Enterprise Application Patterns Using .NET MAUI*. Redmond, Washington: Microsoft Developer Division, .NET, and Visual Studio product teams, 2022.

VAISMAN, Alejandro a Esteban ZIMÁNYI. *Data Warehouse Systems: Design and Implementation*. Heidelberg: Springer, 2014. ISBN 978-3-642-54655-6.