UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

PREHOD Z ARHITEKTURE MONOLITOV NA ARHITEKTURO MIKROSTORITEV

Ljubljana, februar 2022

BLAŽ ČUKELJ

IZJAVA O AVTORSTVU

Podpisani Blaž Čukelj, študent Ekonomske fakultete Univerze v Ljubljani, avtor predloženega dela z naslovom Prehod z arhitekture monolitov na arhitekturo mikrostoritev, pripravljenega v sodelovanju s svetovalcem red. prof. dr. Tomažem Turkom

I Z J A V L J A M

1. da sem predloženo delo pripravil samostojno;
2. da je tiskana oblika predloženega dela istovetna njegovi elektronski obliki;
3. da je besedilo predloženega dela jezikovno korektno in tehnično pripravljeno v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani, kar pomeni, da sem poskrbel, da so dela in mnenja drugih avtorjev oziroma avtoric, ki jih uporabljam oziroma navajam v besedilu, citirana oziroma povzeta v skladu z Navodili za izdelavo zaključnih nalog Ekonomske fakultete Univerze v Ljubljani;
4. da se zavedam, da je plagiatorstvo – predstavljanje tujih del (v pisni ali grafični obliki) kot mojih lastnih – kaznivo po Kazenskem zakoniku Republike Slovenije;
5. da se zavedam posledic, ki bi jih na osnovi predloženega dela dokazano plagiatorstvo lahko predstavljalo za moj status na Ekonomski fakulteti Univerze v Ljubljani v skladu z relevantnim pravilnikom;
6. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v predloženem delu in jih v njem jasno označil;
7. da sem pri pripravi predloženega dela ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;
8. da soglašam, da se elektronska oblika predloženega dela uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
9. da na Univerzo v Ljubljani neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve predloženega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja predloženega dela na voljo javnosti na svetovnem spletu preko Repozitorija Univerze v Ljubljani;
10. da hkrati z objavo predloženega dela dovoljujem objavo svojih osebnih podatkov, ki so navedeni v njem in v tej izjavi.

V Ljubljani, dne ____________________ Podpis študenta: __________________

KAZALO

UVOD **1**
1 APLIKACIJSKA ARHITEKTURA **2**
1.1 Opredelitev in značilnosti aplikacijske arhitekture **4**
1.2 Vrste aplikacijskih arhitektur **5**
2 MONOLITNA ARHITEKTURA **6**
2.1 Arhitektura monolitov **8**
2.2 Prednosti in slabosti monolitne arhitekture **10**
3 MIKROSTORITVE **12**
3.1 Mikrostoritvena arhitektura **13**
3.1.1 Glavne karakteristike mikrostoritev 14
3.1.2 Delovanje mikrostoritvene arhitekture 15
3.1.3 Ogrodja mikrostoritev 16
3.1.4 Docker in zabojniška tehnologija 17
3.1.5 Upravljanje mikrostoritev 18
3.2 Kdaj uporabiti mikrostoritve **19**
3.3 Prednosti in slabosti uporabe mikrostoritev **21**
3.3.1 Tehnološka neodvisnost 22
3.3.2 Odpornost na napake 23
3.3.3 Razširjanje 23
3.3.4 Organizacijska uskladitev 24
3.3.5 Slabosti mikrostoritvene arhitekture 25
4 PRIMERJAVA MONOLITOV IN MIKROSTORITEV **25**
4.1 Razvoj **25**
4.2 Razširjanje **26**
4.3. Testiranje **27**
5 PREHOD NA MIKROSTORITVE **27**
5.1 Tipi/vrste prehodov na mikrostoritve **28**
5.2 Razlogi za prehod **29**
5.3 Razgradnja monolitov v mikrostoritve **30**
5.3.1 Razgradnja glede na tok podatkov 31
5.3.2 Razgradnja glede na domeno aplikacije 33
5.4. Postopek prehoda na mikrostoritve **34**
5.5 Migracijske strategije **35**

5.5.1 Postopek postopne menjave komponent ... 36
5.6 Primer uspešnega prehoda na arhitekturo mikrostoritev ... 38
6 EMPIRIČNA RAZISKAVA STANJA PROGRAMSKE ARHITEKTURE V SLOVENSKIH PODJETJIH ... 39
6.1 Raziskovalna vprašanja ... 40
6.2 Zasnova vprašalnika ... 41
6.3 Rezultati in ugotovitve raziskave ... 41
6.4 Testiranje in analiza hipotez ... 58
7 DISKUSIJA O UGOTOVITVAH EMPIRIČNE RAZISKAVE ... 63
SKLEP ... 64
LITERATURA IN VIRI ... 66

KAZALO TABEL

Tabela 1: Raziskovalna vprašanja magistrskega dela ... 40
Tabela 2: Vloga v podjetju ... 42
Tabela 3: Število profesionalnih let izkušenj na IT-področju ... 43
Tabela 4: Razmišljanje o uporabi mikrostoritvene arhitekture glede na število let profesionalnih izkušenj ... 47
Tabela 5: Število ustvarjenih mikrostoritev glede na pristop grajenja mikrostoritvene programske opreme ... 50
Tabela 6: Osnovna statistika možnosti hitrega vzdrževanja programske opreme ... 59
Tabela 7: Osnovna statistika učinkovitosti glede na razširljivost programske opreme ... 59
Tabela 8: Osnovna statistika glede arhitekturne kompleksnosti ... 60
Tabela 9: Osnovna statistika glede uporabe oblačnih ponudnikov storitev za nameščanje programske opreme ... 61
Tabela 10: Osnovna statistika naklonjenosti do zabojniške tehnologije ... 62
Tabela 11: Osnovna statistika uporabe različnih programskih jezikov ... 63

KAZALO SLIK

Slika 1: Vpetost aplikacijske arhitekture v življenjski cikel programske opreme ... 3
Slika 2: Vzorec uporabe začetne arhitekture za aplikacije ... 7
Slika 3: Trinivojska arhitektura ... 8
Slika 4: primer monolitne arhitekture spletne trgovine ... 9
Slika 5: Primer mikrostoritvene aplikacije ... 16
Slika 6: Orkestracija in koreografija storitev ... 19
Slika 7: Kompleksnost in produktivnost glede na arhitekturni slog ... 21

Slika 8: Heterogenost mikrostoritvene arhitekture 22
Slika 9: Razširjanje monolitne in mikrostoritvene arhitekture 24
Slika 10: Pregled razširjanja arhitektur 26
Slika 11: Proces razgradnje monolitne programske opreme glede na tok podatkov 32
Slika 12: Omejeni kontekst 34
Slika 13: Primer postopka postopne menjave komponent 38
Slika 14: Anketiranci glede na velikost podjetja, v katerem delujejo 42
Slika 15: Prikaz po oblikah in načinu razvoja programske arhitekture 43
Slika 16: Zadovoljstvo z obstoječo programsko opremo (monolitna arhitektura) 44
Slika 17: Slabosti monolitne programske opreme 45
Slika 18: Razlogi za uporabo mikrostoritvene arhitekture 48
Slika 19: Prikaz uporabe zabojniške tehnologije glede na pristop k grajenju mikrostoritvene programske opreme 49
Slika 20 Dejavniki, ki so vplivali na izbor mikrostoritvene arhitekture 51
Slika 21: vpliv težav na razvoj mikrostoritvene programske opreme 53
Slika 22: Uporaba programskih jezikov glede na obliko programske arhitekture 54
Slika 23: Uporaba strežnikov glede na obliko programske arhitekture 56
Slika 24: Zaznavanje prednosti glede na različno obliko programske arhitekture 58
Slika 25: Uporaba zabojniške arhitekture 62

SEZNAM KRATIC

API - (angl. application programing interface); aplikacijski programski vmesnik
CPU - (angl. central processing unit); procesor
CSS - (angl. cascading style sheets); slogi kaskadnih podlog
HTML - (angl. hyper text markup language); jezik za označevanje nadbesedila
IKT - informacijsko-komunikacijska tehnologija
IP - (angl. internet protocol); internetni protokol
IT - informacijska tehnologija
DEVOPS (angl.software development and IT operations); razvoj in IT operacije
SOA (angl. service-oriented architecture); storitveno usmerjena arhitektura

UVOD

Informacijska tehnologija ima v zadnjih desetletjih velik vpliv na poslovanje podjetij in drugih organizacij. Zaradi njenega razvoja so se celovito spremenili poslovni modeli, kar je privedlo do novih poslovnih priložnosti. Ker je informacijska tehnologija eden ključnih dejavnikov poslovanja, je njena izbira zelo pomembna. Informacijska tehnologija ima dolgoročen vpliv na poslovanje, zato ji podjetja in organizacije namenjajo vse več pozornosti.

Eden glavnih dejavnikov informacijske tehnologije, ki mu podjetja in organizacije posvečajo vse več pozornosti, je izbira aplikacijske arhitekture. Aplikacijska arhitektura, ki igra zelo pomembno vlogo v poslovnih sistemih, opisuje načrtovanje in vedenje aplikacij (kako elementi znotraj aplikacije komunicirajo med seboj in z ostalimi uporabniki) na podlagi poslovnih in funkcionalnih zahtev. Osredotočena je na interakcijo med aplikacijo in njenimi uporabniki. V zadnjih nekaj letih je prišlo pri aplikacijski arhitekturi do večjega razvoja, pojavili so se novi koncepti načrtovanja aplikacij, ki slonijo na decentralizaciji.

Podjetja in druge organizacije skušajo najti načine za nemoten in hiter razvoj aplikacij ter hitro dodajanje novih funkcionalnosti k obstoječi aplikaciji. Eden najbolj odmevnih trendov aplikacijske arhitekture je mikrostoritvena aplikacijska arhitektura. Ta arhitektura se je pojavila v želji po pospeševanju ciklov izdaje posodobitev, da bi nove funkcionalnosti čim hitreje prispele do uporabnikov.

Mikrostoritvena arhitektura je relativno nov pojem, ki se je prvič pojavil pred nekaj leti. Ta arhitekturni slog je podoben storitveno usmerjeni arhitekturi (angl. service-oriented architecture), ki obstaja že dlje časa. Kljub temu, da sta si zelo podobna, med njima obstajajo pomembne razlike. Glavni cilj mikrostoritev je, da aplikacijo razdelimo na več različnih storitev, pri čemer vsaka opravlja svojo nalogo. Pomembna lastnost pri tem je, da posamezne storitve komunicirajo med seboj in si izmenjujejo podatke. Podjetja in druge organizacije se za to arhitekturo odločajo zaradi različnih koristi. Te koristi so hitra odzivnost, neodvisnost od ene tehnologije, lažje razumevanje itd.

Zaradi omenjenih koristi in trendov se vse več podjetij in organizacij odloča za uporabo mikrostoritvene arhitekture. To arhitekturo uporabijo bodisi pri gradnji novih aplikacij ali pa želijo mikrostoritveno arhitekturo prenesti na obstoječe aplikacije. Ker je prehod na to arhitekturo dolgoročen in tvegan proces, je potrebno dobro predhodno načrtovanje. V magistrskem delu bom zato podrobneje raziskal prehod s tradicionalne centralizirane monolitne arhitekture na arhitekturo mikrostoritev in raziskal, kakšni so trenutni trendi aplikacijske arhitekture v slovenskem okolju.

Namen magistrskega dela je podrobneje predstaviti in raziskati področje prehoda iz arhitekture monolitov na arhitekturo mikrostoritev, saj gre za relativno nov koncept, za katerega je na voljo malo literature, podjetja pa se za prehod odločajo vse pogosteje.

Da bi dosegel namen magistrskega dela, so opredeljeni naslednji cilji:

- predstaviti področje prehoda iz arhitekture monolitov na arhitekturo mikrostoritev;
- s pomočjo dosedanjih raziskav prikazati dejavnike uspešnih prehodov;
- oceniti stanje programske arhitekture v slovenskih podjetjih in ugotoviti, katera arhitektura prevladuje in kakšni so razlogi za to.

V prvem sklopu magistrskega dela bom na osnovi metode zbiranja sekundarnih podatkov zbral podatke s področja aplikacijske arhitekture monolitov in mikrostoritev. Na podlagi teh podatkov bom preučil dejavnike, zaradi katerih se podjetja in ostale organizacije odločajo za prehod, in dejavnike, ki vplivajo na uspešnost prehoda. Zbrane informacije bom uporabil kot podlago za poznejši raziskovalni del, kjer bom ustvaril anketni vprašalnik za oceno stanja aplikacijske arhitekture v slovenskih podjetjih in organizacijah.

Raziskovalni del magistrskega dela bo temeljil na anketnem vprašalniku. Anketni vprašalnik bo imel več sklopov z zaprtim in odprtim tipom vprašanj. V vzorec bodo zajeti predvsem zaposleni v informatiki v naključno izbranih slovenskih podjetjih iz različnih panog. Na podlagi analize anketnega vprašalnika bom poizkusil ugotoviti, katera oblika aplikacijske arhitekture prevladuje v Sloveniji in kakšni so razlogi za to, poleg tega bom poizkusil priti do ugotovitev, kakšne so koristi prehoda in kakšni so morebitni razlogi za obstanek na monolitni arhitekturi.

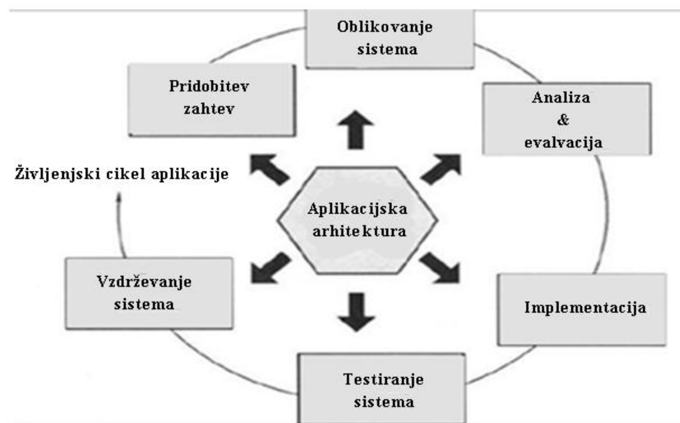
Pridobljeni podatki bodo v nadaljevanju statistično analizirani. Ključni izsledki raziskave bodo prikazani v magistrskem delu.

1 APLIKACIJSKA ARHITEKTURA

Na spletu je veliko definicij, ki opisujejo aplikacijsko arhitekturo. Ena izmed definicij, ki jo v svoji knjigi opisuje Thakur (2008), se glasi: »Aplikacijska arhitektura je abstrakcija ali pogled na sistem na visoki ravni. Osredotoča se na vidike sistema, ki so najbolj koristni pri doseganju glavnih ciljev, kot so zanesljivost , razširljivost in spremenljivost. Arhitektura pojasnjuje, kako se lotiti doseganja teh ciljev.« Če to definicijo pretvorimo v enostavnejšo obliko, potem aplikacijska arhitektura predstavlja načrt za gradnjo aplikacij, ki ga organizacija uporablja, da bi dosegla zastavljene cilje (Thakur, 2008).

Gre za eno izmed najpomembnejših faz v aplikacijskem razvoju. Kot opisujeta Tang in Lau (2014), je aplikacijska arhitektura temelj aplikacije, njenih komponent in povezav, kar zagotavlja njeno zasnovo za razvoj. Vpeta je v vse elemente življenjskega cikla programske opreme od postavljanja poslovnih zahtev, do oblikovanja in implementacije celotne aplikacije, kot je prikazano na sliki 1.

Slika 1: Vpetost aplikacijske arhitekture v življenjski cikel programske opreme



Prirejeno po Qin, Zheng & Xing (2008).

Ker aplikacije postajajo vse bolj kompleksne in vključujejo vse več deležnikov, je široko sprejeto, da se aplikacijsko arhitekturo izdela skozi proces ponovitev (iteracij), preden je aplikacija razvita. Med vsako ponovitvijo se arhitektura izpopolni, tako da upošteva dodatne pomisleke in zahteve. Ponovitev je v tem primeru zelo splošna oblika, ki se lahko nanaša na različne nivoje abstrakcije v arhitekturi. V zgodnji fazi je na primer eden glavnih pomislekov izpolnjevanje zahtev programske opreme, v poznejših fazah pa se po navadi uporablja merljiva sredstva, s katerimi se zagotovi, da bodo različne komponente strojne in programske opreme medsebojno sodelovale. Pomembno je, da programska oprema izpolni določene funkcionalne in nefunkcionalne zahteve, med katerimi je na primer zmogljivost aplikacije. Ob vsaki ponovitvi se sprejmejo nove odločitve o arhitekturnem oblikovanju. Te odločitve vplivajo na poznejše odločitve in izbiro primerne arhitekture.

Kot opisujeta Tang in Lau (2014), so odločitve o arhitekturnem oblikovanju zelo pomembne, zato morajo biti prenesene na vse udeležence aplikacije, sicer se lahko pojavijo napake, kot so:

- vplivi odločitev niso nujno znani tistim, ki sprejemajo odločitve na drugih delih sistema;
- cilji, zahteve in pomisleki glede oblikovanja niso znani ali niso sporočeni ljudem, ki jih morajo vedeti;
- odločevalec se ne zaveda posledic svojih odločitev, ko navede cilj ali zahtevo;
- oblikovalci sprejemajo pristranske odločitve.

V poslovnem svetu podjetja in organizacije uporabljajo poslovne aplikacije, ki obravnavajo veliko količino podatkov, poleg tega pa vsebujejo veliko poslovnih pravil in deležnikov, zato je pomembno, da ustvarijo dobro aplikacijsko arhitekturo in s tem zagotovijo čim večjo funkcionalnost in agilnost. Na izbiro aplikacijske arhitekture vpliva več dejavnikov (Weinreich & Groher, 2016):

- tehnični dejavniki,
- organizacijski dejavniki,
- organizacijska struktura,
- deležniki aplikacije.

1.1 Opredelitev in značilnosti aplikacijske arhitekture

Ker se poslovne aplikacije hitro razvijajo, je zanesljiva aplikacijska arhitektura ključnega pomena. Aplikacije se morajo konstantno posodabljati zaradi spremembe zahtev organizacije ali zaradi okolja, v katerem organizacija deluje. Ta proces je imenovan razvoj/evolucija programske opreme in je ključnega pomena za inženirstvo. Če lahko sistematično uredimo evolucijo programske opreme, lahko zmanjšamo stroške vzdrževanja programske opreme in povečamo možnost ponovne uporabe programske opreme, kar podaljšuje življenjski cikel aplikacije. Če želimo ustvariti dobro evolucijo programske opreme, je za to treba dobro zastaviti aplikacijsko arhitekturo (Lu, Zeng & Xie, 2020).

Aplikacijsko arhitekturo se lahko opredeli kot načrt, po katerem je poslovna aplikacija sestavljena, da lahko izpolnjuje poslovne in uporabniške potrebe, poleg tega pa dobro zastavljena aplikacijska arhitektura pomaga zagotoviti, da so že razvite aplikacije prilagodljive in zanesljive. Povedano drugače, aplikacijska arhitektura zagotavlja načrt aplikacije, ki zagotavlja abstrakcijo za upravljanje kompleksnosti ter vzpostavlja komunikacijske in koordinacijske mehanizme med komponentami aplikacije.

Aplikacijska arhitektura definira, kako aplikacije komunicirajo z vmesno programsko opremo, podatkovnimi bazami in ostalimi zunanjimi aplikacijami. V večini primerov aplikacijska arhitektura sledi splošno sprejetim načelom oblikovanja aplikacij. Uveljavilo se je veliko splošnih aplikacijskih arhitektur. Med najbolj znane arhitekturne sloge se uvršča storitveno usmerjena arhitektura (angl. service-oriented architecture). Ta arhitektura je nastala v devetdesetih letih prejšnjega stoletja, pozneje pa je vodila v več različnih arhitektur, med njimi je trenutno najbolj odmeven mikrostoritveni arhitekturni slog (Weinreich & Groher, 2016).

Aplikacijska arhitektura mora biti določena na podlagi poslovnih in funkcionalnih zahtev, kar vključuje definiranje interakcij med aplikacijskimi paketi, bazami podatkov in vmesnimi sistemi v smislu funkcionalne pokritosti. Pomaga prepoznati morebitne težave pri integraciji in morebitnih luknjah v funkcionalni pokritosti. Na podlagi aplikacijske arhitekture lahko sestavimo migracijski načrt za sisteme, ki so na koncu življenjskega cikla ali imajo visoka tehnološka tveganja (Thakur, 2008).

Vsaka aplikacijska arhitektura je sestavljena iz komponent, povezav in omejitev. Komponente aplikacijske arhitekture so elementi, ki skupaj tvorijo arhitekturo. Aplikacijska arhitektura je po navadi razdeljena v podsisteme, ki so lahko razdeljeni v module, moduli pa v razrede (v objektnem programiranju), kar sestavlja skupek komponent. Arhitekturne povezave so

abstraktne logične povezave med arhitekturnimi komponentami, ki narekujejo, kako posamezne komponente med seboj sodelujejo. Te povezave vsebujejo podatke o odvisnosti komponent, sestavi in agregaciji. Omejitve pa zagotavljajo pogoje in omejitve za povezave med komponentami in povezujejo arhitekturo z sistemskimi zahtevami.

Pri odločanju o tem, katero aplikacijsko arhitekturo uporabiti za novo aplikacijo ali celovito posodobitev trenutne aplikacije, je treba začeti z določanjem strateških ciljev. Strateški cilji pomagajo pri strateških usmeritvah. Strateške usmeritve zagotavljajo strukturo za vsakodnevne odločitve, ki pa sledijo dolgoročnim načrtom in ustvarjajo smer, v katero naj bi se podjetje ali organizacija v prihodnosti gibalo (Harrison, 2013). Po določitvi strateških ciljev je treba oblikovati in izbrati aplikacijsko arhitekturo, ki bo podpirala izbrane strateške cilje. Pri izboru je treba upoštevati številne dejavnike – od tega, kako pogosto naj bi se izdajalo posodobitve, da bi zadovoljili operativne potrebe uporabnikom, do zahtevanih funkcionalnosti s strani poslovnih ciljev in razvojnih potreb (Red Hat, Inc., 2020).

1.2 Vrste aplikacijskih arhitektur

Podjetja in organizacije lahko glede na svoje potrebe in poslovne cilje izbirajo med različnimi aplikacijskimi arhitekturami. V splošnem se lahko aplikacijske arhitekture razdeli v tri glavne skupine, ki se jih še naprej loči na različne podvrste, glede na to, za kakšne vrste aplikacij gre, na katerih platformah in napravah bodo delovale aplikacije itd. Glavne tri aplikacijske arhitekture so: monolitna arhitektura, storitveno usmerjena arhitektura in mikrostoritvena arhitektura. Vsaka izmed teh arhitektur ima svoje prednosti in slabosti. Cilj podjetja ali organizacije je, da oceni, katera arhitektura bi se najbolj prilegala njihovim potrebam, ciljem in strateškim usmeritvam.

Koncept monolitne arhitekture leži v skupini komponent, ki so potrebne za delovanje programske opreme. Te komponente so med seboj tesno povezane in praktično neločljive. Vsi elementi v celoti delujejo v eni aplikaciji in na enem strežniku. Ker vse poteka na enem strežniku in ker so komponente močno povezane, je ta arhitektura ob napakah šibka, saj lahko napaka na eni komponenti ogrozi celoten sistem. Kljub temu, da gre za eno najstarejših oblik aplikacijske arhitekture, to še ne pomeni, da je zastarela. Monolitna arhitektura v določenih primerih še vedno odlično deluje: določena velika podjetja, kot je Etsy, ki se ukvarja s spletno prodajo, kljub novim, bolj priljubljenim arhitekturam še vedno ostajajo na monolitni arhitekturi. V splošnem pa naj bi bila ta arhitektura primernejša za manjša podjetja in organizacije, velikokrat še v ustanovitveni stopnji, kjer se izdelek/storitev šele gradi, zato naj bi bila najbolj primerna za zagonska podjetja in manjše organizacije (RubyGarage, 2019).

Druga arhitektura je storitveno usmerjena arhitektura, ki je arhitekturni slog, ki se nanaša na aplikacije, ki so sestavljene iz diskretnih in ohlapno povezanih programskih agentov in ki izvajajo željeno funkcijo. Ta arhitektura ima dve glavni vlogi, in sicer »ponudnik storitev« in »uporabnik storitev«. V storitveno usmerjeni arhitekturi je aplikacijo mogoče oblikovati in

zgraditi tako, da so njeni moduli integrirani in se jih lahko enostavno ponovno uporabi (Laskey & Laskey 2009).

Storitveno usmerjena arhitektura je primernejša za kompleksne poslovne aplikacije, kot jih na primer uporabljajo banke. Bančne sisteme je zelo težko razdrobiti v mikrostoritve, monolitna arhitektura pa tudi ni priporočljiva za bančne sisteme, saj lahko en del sistema poškoduje celotno aplikacijo, npr. v primeru napake v določenem modulu. Zaradi tega je v takih primerih storitveno usmerjena arhitektura ena od boljših izbir (Laskey & Laskey 2009).

Zadnja izmed najpogosteje uporabljenih programskih arhitektur je mikrostoritvena arhitektura, ki ena od najnovejših arhitektur, ki se je razvila v zadnjem desetletju. Glavni element te arhitekture so mikrostoritve. Mikrostoritve so majhne, neodvisne komponente, ki imajo omejen obseg delovanja. Vsaka mikrostoritev se osredotoča na točno določeno nalogo. Med seboj so povezane z lahkimi mehanizmi, kot je npr. RESTful, in skupaj tvorijo celotno aplikacijo oziroma sistem (Chan, 2017).

Primerna je za vse vrste sistemov, najbolj pa za obsežne in kompleksne sisteme, saj v primerjavi z ostalimi arhitekturami omogoča hitro vzdrževanje. Poleg lahkega vzdrževanja pa je tudi odporna na napake, ki se lahko zgodijo ob izvajanju. Napake pri tej arhitekturi ne vplivajo na celotno aplikacijo, ampak samo na določen nabor povezanih storitev. Zaradi številnih prednosti to arhitekturo uporablja vse več svetovno znanih podjetij, med njimi sta tudi Netflix in Amazon, vse večjo priljubljenost pa pridobiva tudi pri manjših podjetjih.

2 MONOLITNA ARHITEKTURA

Beseda monolit izvira iz antične Grčije, kjer je opisovala velik kamen. Danes se ta beseda uporablja širše v različnih domenah, vendar ideja v ozadju ostaja enaka. Beseda monolit v programskem svetu pomeni programsko nedeljivo enoto, ki združuje vse elemente, potrebne za nemoteno delovanje aplikacije, ki so med seboj tesno povezani (Chan, 2017).

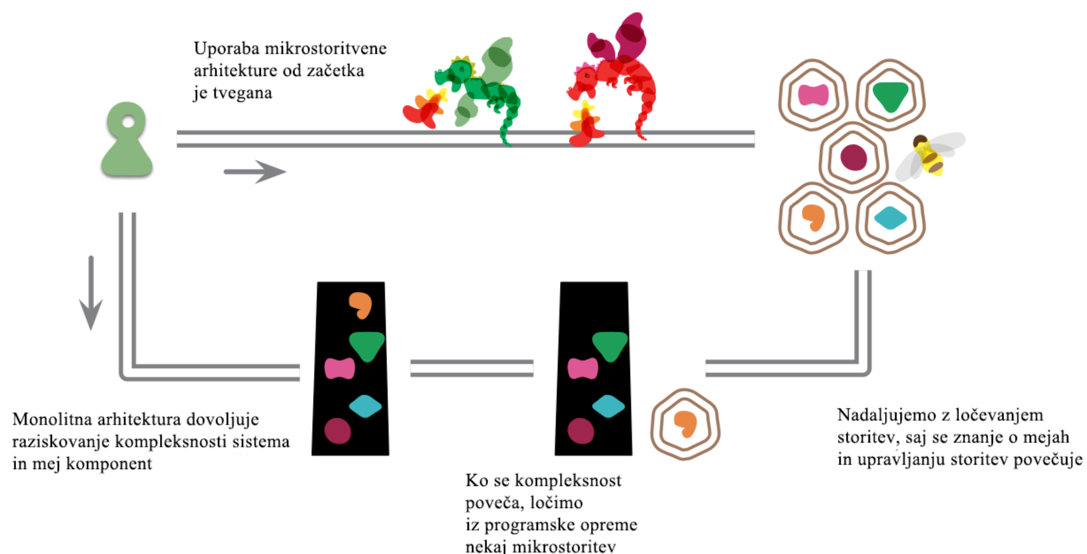
Monolitna aplikacijska arhitektura opisuje enoslojno aplikacijo, pri kateri se uporabniški vmesnik in podatki nahajajo v eni aplikaciji in na eni platformi oziroma strežniku. Monolitna aplikacija je sama po sebi samostojna in neodvisna od ostalih aplikacij ali storitev. Filozofija pri takih vrstah aplikacij je, da aplikacija ni odgovorna samo za določeno nalogo, ampak lahko izvaja vsak korak, ki je potreben za dokončanje določene funkcije.

Monolitne aplikacije lahko dandanes vidimo skoraj povsod. Res je, da gre za starejšo aplikacijsko arhitekturo, vendar številne organizacije še vedno prisegajo nanjo. Velikokrat se na primer uporabljajo pri aplikacijah za osebne finance, saj uporabniku pomagajo izvršiti celotno opravilo in na ta način zagotavljajo tudi varnost glede zaupnih podatkov, saj vsi podatki ostanejo znotraj aplikacije. Monolitno arhitekturo velikokrat uporabljajo zagonska podjetja.

Kot pravi Fowler (2015), je v začetnih fazah aplikacije monolitna arhitektura enostavnejša od ostalih arhitektur, saj dovoljuje lažje raziskovanje kompleksnosti in iskanje meja komponent.

Fowler (2015) pravi, da imajo zagonska podjetja možnost, da se pri novi aplikaciji odločijo med mikrostoritveno in monolitno arhitekturo. Po njegovih ugotovitvah je razvoj nove aplikacije na temeljih mikrostoritvene arhitekture bolj tvegana izbira, saj je kompleksnejša in zahteva razumevanje različnih programskih jezikov in pristopov. Zaradi tega se podjetja in organizacije, ki se že v samem začetku razvoja odločajo za mikrostoritveno arhitekturo, v številnih primerih soočajo z neuspešnostjo. Fowler (2015) zato predlaga, da zagonska podjetja začnejo svoje aplikacije razvijati na monolitni arhitekturi, saj jim ta omogoča raziskovanje komponent, ki jih aplikacija potrebuje, da bi podjetje doseglo zastavljene cilje. Ker pa sočasno z velikostjo aplikacije raste tudi kompleksnost aplikacije, postane aplikacija sčasoma težko obvladljiva in jo je z monolitno arhitekturo težje vzdrževati. V tem primeru je potreben strateški premislek podjetja o prehodu na mikrostoritveno arhitekturo, kar je prikazano na sliki 2.

Slika 2: Vzorec uporabe začetne arhitekture za aplikacije



Prirejeno po Fowler (2015).

Ena od glavnih slabosti, zaradi katere imajo organizacije odpor do monolitne arhitekture, je modularnost. Monolitna arhitektura je zasnovana skoraj brez modularnosti, ki pa je v programskem svetu zaželjena. Modularnost dovoljuje upravljanje kompleksnosti aplikacije, tako da se aplikacijo razčleni na več manjših obvladljivih modulov. Modularnost se lahko pri monolitni arhitekturi največkrat zasledi v programski kodi, kjer lahko razvijalci celotno aplikacijo razdelijo na različne funkcije, razrede in metode, ki rešujejo specifičen problem in zagotavljajo funkcionalnost aplikacije. Če se primerja modularnost monolitov in mikrostoritev, je modularnost slednjih znatno višja, kar omogoča lažje vzdrževanje in razvijanje (Rengaiah, 2014).

2.1 Arhitektura monolitov

Monolitna arhitektura je v primerjavi z ostalimi aplikacijskimi arhitekturami preprost pristop, vendar ima svoje omejitve, predvsem kadar gre za velike in kompleksne aplikacije. Zaradi tega se monolitna arhitektura največkrat uporablja pri manjših aplikacijah. Manjše aplikacije je na tej arhitekturi relativno lahko vzdrževati in posodabljati, vendar pa se aplikacija skozi čas razvija, zaradi česar postane kompleksna, izdajni cikli posodobitev pa vedno daljši in dražji. Zakaj so velike monolitne aplikacije tako zapletene, je možno razbrati že iz samih lastnosti te arhitekture.

Monolitne poslovne aplikacije so pogosto zgrajene iz treh nivojev (angl. three-tier application architecture). V določenih primerih so lahko zgrajene tudi iz enega samega ali dveh nivojev, vendar pa v splošnem prevladujejo trinivojske aplikacije. Trinivojske aplikacije so sestavljene iz nivoja odjemalca, poslovnega nivoja in podatkovnega nivoja. Na nivoju odjemalca se nahaja uporabniški vmesnik, ki skrbi za nemoteno interakcijo uporabnika z aplikacijo. Na tem nivoju so največkrat uporabljene spletne tehnologije, kot so html, javascript in css. Poslovni nivo z ostalima dvema nivojema komunicira preko API ali podobnih klicev. V tem nivoju je zbrana vsa poslovna logika, ki je potrebna za pravilno delovanje sistema. Tukaj se največkrat uporabljajo programski jeziki, kot so Java, C in Python. Zadnji je podatkovni nivo, kjer se nahajajo vsi podatki, potrebni za nemoteno delovanje aplikacije. Ta nivo je izjemno pomemben in mora biti vedno na voljo poslovnemu nivoju za nemoteno delovanje celotne aplikacije. Na sliki 3 je prikazana trinivojska monolitna arhitektura in povezave med posameznimi nivoji. Uporabniki preko računalnika ali katerekoli druge naprave z aplikacijo komunicirajo preko uporabniškega vmesnika, ki je povezan s poslovnim nivojem aplikacije. Poslovna logika pa glede na zahteve uporabnika in morebitne omejitve komunicira z bazo podatkov, kjer se urejajo in zapisujejo podatki (Haq, 2018).

Slika 3: Trinivojska arhitektura

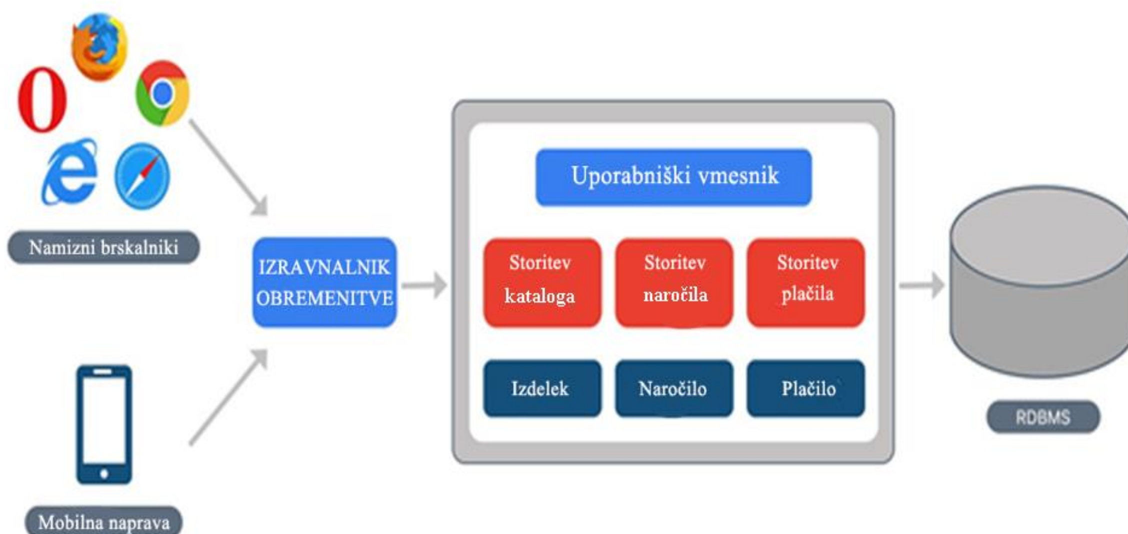
Prirejeno po Rehman & Ashraf (2019).

Glavna razlika med monolitno in mikrostoritveno arhitekturo je na poslovnem nivoju aplikacije, kjer je shranjena celotna poslovna logika. V primeru monolitne aplikacije je ta logika

zapisana v eni skupni nedeljivi in močno povezani enoti, v primeru mikrostoritvene aplikacije pa je logika porazdeljena na različne storitve, ki so med seboj ohlapno povezane in skupaj tvorijo celotno aplikacijo. Ker je monolitna aplikacija sestavljena iz nedeljivih in močno povezanih elementov, se lahko zgodi, da v primeru napake na enem elementu ta napaka v času izvajanja onemogoči izvajanje celotne aplikacije. V takih primerih izpada je potreben ponovni zagon, dokler pa aplikacija ni ponovno zagnana, je njena uporaba onemogočena (Haq, 2018).

Delovanje trinivojske arhitekture na monolitni arhitekturi lahko ponazorimo z primerom delovanja spletne trgovine. Na levem delu se nahaja nivo odjemalca, kjer lahko uporabniki preko različnih tehnologij, kot sta na primer računalnik ali mobilni telefon, dostopajo do spletne trgovine preko internetnih brskalnikov ali mobilne aplikacije. Uporabniki komunicirajo s poslovnim nivojem aplikacije preko API-jev. Ker se velikokrat zgodi, da ima podjetje ali organizacija, zaradi večje zmogljivosti delovanja nameščenih več enakih primerkov/instanc aplikacije, je potrebno razporediti zahteve uporabnika na različne aplikacijske strežnike, na katerih je nameščena aplikacija. Za razporejanje zahtev se uporablja izravnalnik obremenitve (angl. load balancer). Izravnalnik obremenitve skrbi za uravnovešanje obremenitve med instancami aplikacije. V osrednjem delu se nahaja poslovni nivo. Ta nivo je pri monolitni arhitekturi zgrajen iz nedeljive enote. Znotraj tega nivoja se nahajajo različne storitve oziroma elementi z različnimi nalogami. Te storitve vsebujejo poslovno logiko, ki skrbi, da aplikacija deluje. Če vzamemo primer plačila, le ta vsebuje vso potrebno logiko, potrebno za izvedbo plačila kupca. Čisto na desni strani monolitne arhitekture pa imamo še podatkovni nivo, ki hrani podatke, potrebne za delovanje aplikacije (Haq, 2018). Ta nivo aplikacije komunicira s poslovnim nivojem, kot je prikazano na sliki 4.

Slika 4: primer monolitne arhitekture spletne trgovine



Prirejeno po Haq (2018).

2.2 Prednosti in slabosti monolitne arhitekture

Monolitna arhitektura je arhitekturni slog, ki ima na eni strani veliko prednosti, vendar pa se je treba zavedati tudi njenih slabosti. Te so v začetnih fazah morda težje vidne, vendar pa so lahko njihove posledice zelo velike. Zato je pomembno, da podjetje ali organizacija skrbno pripravi načrt in razišče, katera aplikacijska arhitektura je najprimernejša za doseganje poslovnih ciljev.

V literaturi je moč najti pet glavnih prednosti monolitne arhitekture (Gnatyk, 2018):

- enostaven razvoj,
- enostavno testiranje,
- hitro horizontalno razširjanje,
- enostavno vzdrževanje
- enostavno nameščanje.

Vse zgoraj naštete prednosti monolitne arhitekture veljajo predvsem, kadar je sistem oziroma aplikacija relativno majhna in ne kompleksna v smislu samih funkcionalnosti aplikacije (kaj omogoča aplikacija), berljivosti razvojne kode in števila uporabnikov ter razvijalcev, ki aplikacijo uporabljajo oz. razvijajo. V splošnem velja, da se z velikostjo aplikacije manjšajo koristi monolitne arhitekture, seveda pa je to velikokrat odvisno tudi od samega primera aplikacije (Gnatyk, 2018).

Monolitna arhitektura je, kot že omenjeno, primerna predvsem, kadar si podjetje ali organizacija lasti manjšo aplikacijo. Glavni vzrok za to je, da je monolitna aplikacija po navadi zgrajena kot samostojen projekt v integriranem razvojnem okolju (angl. integrated development environment). Samostojen projekt omogoča, da so vsi elementi aplikacije razviti na enem mestu, kar dovoljuje lažje usklajevanje med razvijalci in hiter prenos informacij med njimi. Tudi vzdrževanje celotne baze kode na enem mestu in nameščanje aplikacije na eno mesto ima veliko prednosti. Za vzdrževanje aplikacije je treba vzdrževati samo eno shrambo, razvijalci pa morajo biti zmožni hitrega iskanja vseh funkcionalnosti v eni mapi. Ker je monolitna aplikacija zgrajena samostojno, ima to vpliv tudi na namestitev, saj je treba vzdrževati samo en test in en namestitveni cevovod, s čimer se je moč izogniti visokim stroškom namestitve (Karwatka, 2020).

Ena od prednosti monolitne arhitekture je enostavno horizontalno razširjanje, ki je danes pogosteje uporabljeno kot vertikalno. Opredeljeno je kot nameščanje dodatnih enot enake aplikacije (primerkov aplikacije) na različne strežnike. Horizontalno razširjanje monolitne arhitekture je v primerjavi z mikrostoritveno arhitekturo lažje, saj se aplikacijo preprosto namesti na različne strežnike in doda izravnalec obremenitve, ki poskrbi, da sta oba strežnika enako izrabljena, v primeru mikrostoritvene arhitekture pa je treba določiti še povezave med posameznimi storitvami, kar povečuje kompleksnost (Messina, Rizzo, Storniolo & Tripiciano, 2016).

Slabosti monolitov so (Karwatka, 2020):

- nefleksibilnost pri uporabi različnih tehnologij,
- slaba distribucija razvoja,
- delovanje kot ena enota,
- počasen razvoj,
- slaba optimizacija virov.

Programski jeziki in ostala programska oprema se hitro razvijajo in ponujajo številne možnosti za različne primere uporabe. Zaradi tehnološke raznolikosti se lahko podjetje ali organizacija v času razvoja nove aplikacije odloči za najprimernejšo tehnologijo za doseganje njenih ciljev. Ker se tehnologija hitro spreminja, funkcionalnost aplikacije pa se sčasoma razširi, obstaja možnost da tehnologija, s katero se je razvoj aplikacije začel, ne zadošča več za doseganje nadaljnjih ciljev, kar lahko zavira rast in razvoj podjetja ali organizacije. V primeru monolitne arhitekture v takem primeru sledi t. i. »tehnološka zaklenjenost«, saj je podjetje ali organizacija ujeta v tehnologiji, s katero se je razvoj začel. Tehnološka zaklenjenost onemogoča fleksibilno izbiro tehnologije za določeno funkcionalnost, lahko pa tudi za celotno aplikacijo. Če bi želeli preiti na drugo tehnologijo, bi pri monolitni arhitekturi to obsegalo prehod celotne aplikacije na drugo tehnologijo, kar pa pomeni visoke stroške in dolgotrajen prehod. Podjetja in organizacije se zaradi tega velikokrat ne odločijo za prehod na novo tehnologijo, temveč ostanejo na obstoječi, kar ima posledice za učinkovitost in zmogljivost. Na dolgi rok v večini primerov sledi tudi povečanje oportunitetnih stroškov zaradi izbire obstoječe tehnologije (Karwatka, 2020).

Distribucija razvoja pri monolitni arhitekturi zahteva veliko koordinacije. V kolikor na isti aplikaciji deluje več različnih razvojnih skupin, te težko delujejo simultano, saj celotna aplikacija sloni na eni bazi razvojne kode. Paziti je treba tudi na vse odvisnosti posameznih modulov in razredov monolitne aplikacije, kadar ena razvojna skupina razvija novo funkcionalnost, saj lahko sprememba enega modula ali knjižnice poškoduje drug del aplikacije, kar onemogoča njeno uporabo. Zaradi tega je nujno, da informacije o spremembah potujejo do vseh udeležencev pri razvoju aplikacije, da lahko razvojne skupine prilagodijo povezane module glede na spremembe. Dodajanje funkcionalnosti in vzdrževanje aplikacije zato zahteva dobro koordinacijo in informiranost razvojnih skupin, ki pa jo je težko doseči, kadar je aplikacija velika. Zaradi tega se porabi veliko časa za poznejše reševanje konfliktov ob posodabljanju aplikacije (Gnatyk, 2018).

Monolitna arhitektura je v splošnem slabo odporna na napake, ki se lahko zgodijo v aplikaciji (npr. hrošči). Hrošč, ki se pojavi v aplikaciji, lahko ogrozi delovanje celotne aplikacije, saj se lahko ta v celoti zruši. Aplikacijo, zaustavljeno zaradi hrošča, je v tem primeru treba znova v celoti zagnati, kar lahko predvsem pri velikih monolitnih aplikacijah traja kar nekaj časa. Slabo odpornost monolitne arhitekture gre pripisati veliki povezanosti komponent, ki delujejo kot celota, kar ogroža pravilno delovanje celotne aplikacije v primeru napak. Ta problem se lahko

vsaj deloma reši z dodajanjem več primerkov aplikacije na strežnik. Ko določen primerek naleti na težavo, ostali primerki še vedno delujejo. Rešitev dodajanja primerkov zaradi morebitnih napak pa ni optimalna, kar se tiče izkoriščenosti virov kot tudi stroškov, ki pri tem nastanejo. Viri pri monolitnih aplikacijah v večini primerov niso optimizirani. Po navadi je določena funkcionalnost oziroma storitev aplikacije bolj podvržena obremenitvam kot druga. Zaradi tega prihaja do ozkih grl. Ozka grla ne zmanjšujejo samo zmogljivost določenega elementa, ampak zmanjšujejo zmogljivost celotne aplikacije. V kolikor se podjetje ali organizacija odloči, da bo ozko grlo odpravilo, ima na voljo dve možnosti: vertikalno ali horizontalno razširjanje. Ker ima vertikalno razširjanje omejitve v sposobnosti komponent, se v večini primerov uporablja horizontalno razširjanje. Pri horizontalnem razširjanju monolitne aplikacije je treba razširiti aplikacijo kot celoto, kar pomeni, da je treba razširiti vse module in funkcionalnosti. Ker so razširjeni vsi moduli aplikacije, viri ostanejo neizkoriščeni, saj določeni elementi niso podvrženi obremenitvam, kljub temu pa so nameščeni na več strežnikov (But, 2019).

3 MIKROSTORITVE

Organizacijska agilnost, možnost fleksibilnega prilagajanja in pripravljenost na spremembe v hitro spreminjajočem se okolju so postali ključni dejavniki za doseganje trajne konkurenčne prednosti. Informacijska komunikacijska tehnologija (IKT) je ključni omogočevalec omenjenih karakteristik ne samo v tehnološkem sektorju, pač pa tudi v drugih panogah (npr. AirBnB, CocaCola). Veliko organizacijskih zmožnosti in funkcij je podkrepljenih z velikimi monolitnimi sistemi. Kot omenjeno v prejšnjem poglavju, monoliti ne ponujajo neodvisnega izvrševanja svojih modulov. Taki sistemi postanejo čez čas kompleksni in težji za spreminjanje, kar lahko povzroči, da se upočasnita organizacijski razvoj in sposobnost hitrega odziva na spremembe iz okolja, kot so npr. spremembe obnašanja konkurentov in potrošnikov. Da bi se izognili negativnim učinkom velikega monolitnega sistema, je Fowler pred nekaj leti predlagal novo rešitev pri grajenju organizacijskih sistemov in aplikacij, imenovano mikrostoritvena arhitektura (Baškarada, Nguyen & Koronios, 2018).

Mikrostoritvena arhitektura (angl. microservice architecture) je relativno nov pristop pri gradnji aplikacij. Prvi koncept je Peter Rodgers leta 2005 med konferenco spletnih storitev predstavil pod imenom *Micro-Web-Services*. Rodgerjevo delo je nastalo že leta 1999 skupaj z raziskovalnim projektom pri Hewlett Packard Labs, katerega cilj je bil ustvariti obsežne zapletene programske sisteme, ki so odporni na spremembe in so manj krhki. Leta 2011 so strokovnjaki aplikacijske arhitekture na delavnici v Benetkah za nadgrajeni koncept Rodgersove arhitekture uporabili termin mikrostoritve, s katerim so opisali arhitekturni slog, ki so ga preučevali mnogi izmed njih (Sumerge, 2019).

Pri mikrostoritvah gre za arhitekturni slog in pristop k drugačnemu razvoju aplikacij, ki stremi k zadovoljevanju modernega poslovnega povpraševanja. To povpraševanje se hitro spreminja, zato je treba biti fleksibilen in pripravljen na nove spremembe. V kolikor se je podjetje ali organizacija zmožno tehnično prilagajati povpraševanju, bo imelo tudi konkurenčne prednosti

v primerjavi z ostalimi konkurenti (Rajesh, 2016). Ker aplikacije postajajo vse bolj kompleksne, je treba veliko virov nameniti prav upravljanju s kompleksnostjo aplikacij. Ta je odvisna od znanja razvijalcev, izbire tehnologije in razporeditve virov.
Današnji hiter proces digitalizacije zahteva fleksibilnost, da se organizacije lahko prilagodijo na hitro spreminjajoče se poslovne zahteve in novonastale poslovne priložnosti, ki se porajajo v poslovnem okolju. Zato morajo biti nove lastnosti oziroma funkcionalnosti organizacijskih aplikacij vse hitreje razvite in dane na trg notranjim oziroma zunanjim uporabnikom. Organizacije v svojem poslovanju uporabljajo oziroma razmišljajo o uporabi različnih vrst informacijske tehnologije: od mobilnih aplikacij, socialnih medijev, oblačnih storitev do interneta stvari. Zaradi tega se sprejema tehnologijo kot enega od ključnih dejavnikov, ki omogoča povečati prihodke, konkurenčno prednost in navsezadnje število kupcev. Da bi se organizacije čim bolje prilagodile na spremembe, ki jih prinaša informacijska tehnologija, se veliko tehnoloških in netehnoloških podjetij odloča za uvajanje mikrostoritvene aplikacijske arhitekture (Bogner & Zimmermann, 2016).

V zadnjih nekaj letih je mikrostoritvena arhitektura pridobila vse več pozornosti pri večjih in tudi manjših podjetjih. Mikrostoritveno arhitekturo uporabljajo številna znana podjetja, kot so Amazon, Netflix, Spotify in Twitter (Soldani, Tamburri & Van Den Heuvel, 2018). Dandanes vsa podjetja agresivno tekmujejo med seboj, da bi ohranila konkurenčno prednost. Ker se potrebe potrošnikov hitro spreminjajo, je treba nenehno izboljševati kakovost izdelkov in storitev, da bi ohranili potrošnike in ustvarili dobiček. Nivo produktivnosti se lahko zviša z izboljševanjem internih procesov, zato se je treba osredotočiti na racionalizacijo delovnih procesov in uvedbo novih metod, ki omogočajo ustvarjanje dobička. Ena glavnih nalog v današnjem poslovnem svetu je avtomatizacija notranjih procesov podjetja v najkrajšem možnem času. Hitrejšo avtomatizacijo procesov pa omogoča mikrostoritvena arhitektura napram monolitni (Patel, 2018).

3.1 Mikrostoritvena arhitektura

Ker so mikrostoritve relativno nov pojem pri aplikacijski arhitekturi, še vedno ni sprejete enotne definicije, ki bi opisovala mikrostoritveno arhitekturo. Na spletu in v člankih je možno najti veliko definicij, ki so si med seboj podobne. Tako ena izmed definicij po Fowlerju (2014), enega od začetnikov mikrostoritvene arhitekture, mikrostoritve definira kot arhitekturni slog za razvoj aplikacij; kot zbirko majhnih storitev, ki za dosego končnega cilja komunicirajo med seboj. Vsaka posamezna storitev deluje kot svoj proces in komunicira z drugimi storitvami z lahkimi protokoli (Fowler, 2014). Valderas, Torres in Pelechano (2020) arhitekturo mikrostoritev opišejo kot arhitekturni slog, kjer so aplikacije razdeljene v majhne neodvisne gradnike imenovane mikrostoritve. Vsaka mikrostoritev je osredotočena samo na eno poslovno zmožnost. Posamezna mikrostoritev je vzdrževana in nameščena neodvisno od drugih mikrostoritev.

3.1.1 Glavne karakteristike mikrostoritev

V literaturi je moč najti pet glavnih karakteristik mikrostoritev (Fowler, 2014):

- oblikovane so okoli poslovnih zmogljivosti, vsaka mikrostoritev vsebuje logiko samo ene poslovne zmogljivosti, zaradi česar gre pri mikrostoritvah za relativno majhne elemente;
- vzdrževati ni treba celotne aplikacije, ampak se lahko vzdržuje vsako posamezno storitev posebej, kar omogoča neodvisen razvoj;
- so nevtralne do programskega jezika, kar pomeni, da se lahko izbira med več različnimi jeziki, saj lahko določena storitev deluje bolje na enem programskem jeziku kot na drugem;
- ni treba razumeti implementacije drugih storitev znotraj aplikacije;
- decentralizirano upravljanje.

Neodvisen razvoj posamezne mikrostoritve zagotavlja hiter razvoj, namestitev in vzdrževanje. Pri razvoju mikrostoritvene aplikacije se skoraj nikoli ne srečuje z ozkimi grli pri razvoju, ker ni treba čakati drugih razvojnih ekip, ki razvijajo ostale mikrostoritve iste aplikacije, saj posamezna mikrostoritev deluje neodvisno od ostalih. Ker deluje neodvisno od ostalih mikrostoritev, se jo lahko po končanem razvoju tudi testira neodvisno od drugih mikrostoritev. Neodvisnost mikrostoritev omogoča tudi neodvisno razširjanje (angl. scaling) preobremenjenih mikrostoritev, kar pri monolitni arhitekturi ni mogoče; tam je zaradi preobremenjenega določenega dela aplikacije treba razširjati celotno aplikacijo, kar povzroča dodatne stroške (Lenarduzzi, Lomio, Saarimaki & Taibi, 2020).

Posamezna mikrostoritev je po definiciji zelo majhna in se osredotoča na točno določeno nalogo oziroma poslovno sposobnost, ki jo mora odlično opravljati. Vsaka mikrostoritev naj bi po podatkih IBM (IBM Cloud Education, 2019) imela le nekaj tisoč vrstic kode. Majhnost mikrostoritev omogoča, da se oblikuje več manjših razvojnih skupin, ki si lastijo eno ali več mikrostoritev. Z oblikovanjem teh skupin je omogočeno lažje upravljanje in porazdelitev odgovornosti, kar omogoča večjo učinkovitost in pregled nad razvojem (Lenarduzzi, Lomio, Saarimaki & Taibi, 2020). Ker so mikrostoritve majhni elementi, jih je možno hitro posodobiti in učinkovito vzdrževati, kar vodi h krajšim ciklom izdaje posodobitev (IBM Cloud Education, 2019).

Veliko organizacij pri razvoju aplikacij še vedno vztraja pri uporabi minimalnega števila programskih jezikov, bodisi zaradi uporabe monolitne arhitekture bodisi zaradi drugih organizacijskih ali tehničnih razlogov, kot je na primer pomanjkanje strokovnega znanja zaposlenih. Mikrostoritve nasprotno od monolitov omogočajo, da je vsaka mikrostoritev razvita v poljubnem programskem jeziku. Možnost izbire programskega jezika za posamezno mikrostoritev omogoča razvijalcem, da mikrostoritev razvijejo v tistem programskem jeziku, za katerega menijo, da je za dano nalogo najprimernejši, oziroma v tistem jeziku, s katerim imajo največ izkušenj. Kljub temu, da je raznolikost programskih jezikov znotraj ene aplikacije dobrodošla, pa to zahteva večje upravljanje programskih jezikov. Da bi bilo upravljanje

programskih jezikov in razvojne kode, napisane v različnih jezikih, lažje, je treba ob razvoju definirati primere uporabe in podatkovne strukture ter izdelati podrobno dokumentacijo (Luthra, brez datuma).

Po Juriču (2017) je ena izmed pomembnih lastnosti mikrostoritev ta, da podjetju, ki prične razvijati novo programsko rešitev, ki temelji na mikrostoriteveni arhitekturi, ni treba razvijati celotne aplikacije od začetka. V kolikor je podjetje razvilo že katerokoli drugo mikrostoritveno aplikacijo, lahko posamezne mikrostoritve pri novi rešitvi uporabi ponovno. Na ta način je treba razviti le še manjkajoče funkcionalnosti (nove storitve), ki še niso bile razvite. Ponovna uporaba razvojne kode je v sodobnem pristopu razvoja aplikacij izjemno dobrodošla, saj omogoča hitrejši razvoj, poleg tega pa tudi znižuje stroške (Ashan, 2019).

Kot že omenjeno, se lahko vsako posamezno mikrostoritev razvije in namesti neodvisno od drugih mikrostoritev. Prav tako pa se lahko vsako mikrostoritev razvija v različnem programskem jeziku ter uporabi različno ogrodje in podatkovne mehanizme. Organizacija ima tako prosto pot pri izbiri tehnologije, ki jo bo uporabila za razvoj programske rešitve. Ker so mikrostoritve majhne enote, ki tvorijo celotno aplikacijo, se jih lahko enostavno nadgradi ali pa v celoti zamenja z novo mikrostoritvijo, razvito v drugem programskem jeziku, v kolikor razvijalci menijo, da je drug programski jezik primernejši. Zaradi večje fleksibilnosti pri uporabi različnih tehnologij se zmanjša možnosti za tehnološki zaklep, ki se velikokrat zgodi pri monolitni arhitekturi. Zaradi neodvisnosti pri uporabi programskih jezikov se velikokrat uvaja ohlapno standardizacijo/napotke pri izboru tehnologije, ki se jo lahko uporabi pri razvoju mikrostoritev. Ta pravila po navadi določajo, katere programske jezike, ogrodja in tipe podatkovnih strežnikov lahko razvijalci pri razvoju rešitve uporabljajo. Z uporabo standardizacije želijo organizacije zmanjšati negativne vplive razpršene tehnologije. Razpršeno tehnologijo je težje obvladovati in se ji pozneje prilagajati, saj je pri njej treba zagotoviti dovolj virov za upravljanje specifičnih tehnologij (Bogner & Zimmermann, 2016).

3.1.2 Delovanje mikrostoritvene arhitekture

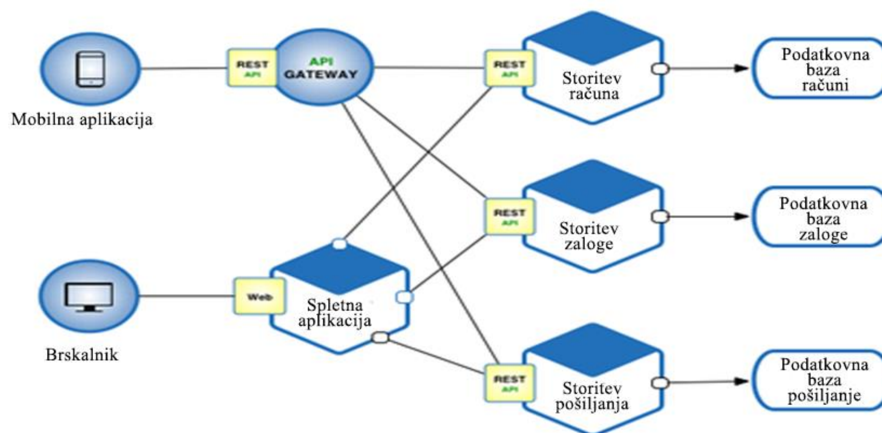
Kot že opisano, mikrostoritveno programsko opremo sestavlja zbirka neodvisno nameščenih storitev, ki so organizirane znotraj poslovnih zmožnosti podjetja ali organizacije z decentraliziranim upravljanjem in neodvisnim programskim jezikom. Glavna ideja mikrostoritvene arhitekture je v majhnosti storitev, ki jih je lažje razviti, upravljati in vzdrževati (Bogner & Zimmermann, 2016).

Pri uporabi mikrostoritev se funkcionalnost programske opreme loči v več neodvisnih storitev, ki so odgovorne za izvajanje natančno določenih samostojnih nalog. Te storitve med seboj komunicirajo preko preprostih in lahkih programskih vmesnikov API (angl. application programming interface). S komuniciranjem storitev se vzpostavi povezava, ki omogoča, da lahko aplikacija deluje kot celota (Patel, 2018). Preprost primer storitev in komunikacij med njimi, je prikazan na sliki 5, kjer aplikacija sestoji iz API prehoda, več različnih storitev,

podatkovnih baz, mobilne aplikacije, spletne trgovine in API-klicev. API-prehod je odgovoren za usmerjanje zahtev in služi kot enotna vstopna točka do aplikacije preko zunanjih virov (npr. mobilnih naprav). Pomembna lastnost storitev je, da delujejo kot neodvisne enote, ki so lahko nameščene na različnih strežnikih na različnih delih sveta. Vsaka storitev je lahko razvita z različnim programskim jezikom od ostalih storitev, kar omogoča fleksibilnost pri razvoju celotne aplikacije (Richardson, brez datuma).

Kadar želi uporabnik dostopati do določene funkcionalnosti aplikacije, se to izvede z API-klicem, ki je pozneje z API prehodom usmerjen do pravilne storitve. Nato storitev izvede svojo nalogo. Med izvedbo naloge lahko delujoča storitev pokliče tudi drugo storitev, da pridobi podatke, ki jih potrebuje za dokončanje naloge. Storitve lahko dostopajo do podatkovnih baz. Pri mikrostoritveni arhitekturi naj bi vsaka storitev imela svojo bazo podatkov, do katere druge storitve ne morejo dostopati, kar zagotavlja večjo varnost. Do podatkov podatkovne baze druge storitve lahko ena storitev dostopa le z API-klicem storitve. Ko storitev dokonča nalogo, pošlje podatke nazaj do uporabnika (Richardson, brez datuma).

Slika 5: Primer mikrostoritvene aplikacije



Prirejeno po Richardson (brez datuma).

3.1.3 Ogrodja mikrostoritev

Za gradnjo aplikacije v mikrostoritvenem arhitekturnem slogu je na spletu na voljo veliko ogrodij za različne programske jezike. Najbolj znana ogrodja so (Kurmi, 2020):

- Spring Boot with Spring Cloud,
- Eclipse Vert.X Microservices framework,
- Oracle Helidon Microservices framework,
- GoMicro,
- Molecular,
- Django,

- Flask.

Vsako izmed naštetih ogrodij je primerno za določen programski jezik. Med najpogosteje uporabljenimi so ogrodja, primerna za programske jezike Java, Python, Node.js itd. Najbolj razširjeno mikrostoritveno ogrodje je Spring Boot, ki temelji na razširjenem programskem jeziku Java (Kurmi, 2020).

Spring Boot je odprtokodno ogrodje, ki ga je razvil Rod Johnson, trenutno pa ga vzdržuje podjetje Pivotal. Gre za ogromen nabor orodij za poenostavitev razvoja aplikacije. Ponuja enostavno ustvarjanje samostojnih namiznih, spletnih in organizacijskih aplikacij, kot tudi dosleden programski model za različne vrste tehnologij, bodisi za dostop do podatkov bodisi do infrastrukture za sporočanje. Spring Boot razvijalcem omogoča hitro in učinkovito ustvarjanje samostojnih aplikacij oziroma storitev, ki temeljijo na programskem jeziku Java (Soni, Ganeshan & Rajesh, 2017).

3.1.4 Docker in zabojniška tehnologija

Zabojniška tehnologija je postala eden od glavnih trendov pri razvoju programske opreme kot alternativa virtualizaciji. Eno izmed bolj priljubljenih orodij zabojniške tehnolije je Docker. Docker sta leta 2010 na poletnem startup inkubatorju razvila Sebastien Pahl in Solomon Hykes. Tehnologija Docker razvijalcem omogoča enostavno gradnjo razširljivih in obvladljivih mikrostoritev na različnih operacijskih sistemih (Linux, Windows itd.) (Sviatoslav & Sergey, 2020).

Razvijalci imajo možnost mikrostoritve namestiti na navidezni (virtualni) računalnik ali na zabojniško tehnologijo. Navidezni računalniki so bili v začetku predstavljeni kot tehnika za optimizacijo računalniških virov. En sam strežnik lahko poganja več navideznih računalnikov in ustvari vsako instanco aplikacije kot ločen navidezni računalnik. S tem razvijalci zagotovijo stabilno okolje za posamezen primerek aplikacije. Na žalost pa razvijalci pri tej tehniki hitro naletijo na težave z učinkovitostjo. Neučinkovitost se pokaže predvsem takrat, ko je aplikacijo treba razširiti na več instanc, saj navidezni računalniki porabijo veliko virov, na primer procesorsko moč in pomnilnik strežnika (Sviatoslav & Sergey, 2020).

Navidezni računalnik je računalniška datoteka, imenovana slika (image), ki se obnaša kot pravi računalnik. Z drugimi besedami: gre za ustvarjanje računalnika znotraj računalnika. Ta deluje v načinu okna kot večina ostalih programov in končnemu uporabniku ponuja enako izkušnjo, kot bi jo imeli na gostiteljskem operacijskem sistemu. Navidezni računalnik je zaprt od ostalega sistema, kar pomeni, da programska oprema navideznega računalnika ne more uiti ali posegati v sam računalnik. Vsak navidezni računalnik zagotavlja svojo virtualno strojno opremo, vključujoč CPU, pomnilnik, trde diske itd (Microsoft Azure, brez datuma).

Druga možnost, s katero lahko razvijalci nameščajo in zaženejo programsko opremo ali mikrostoritve, pa je zabojniška tehnologija oz. Docker. Docker je odprtokodni mehanizem za zabojništvo, ki avtomatizira pakiranje, pošiljanje in uvajanje katerihkoli programskih aplikacij, ki so predstavljeni kot »lahki, prenosni in samozadostni vsebniki«, ki se lahko izvajajo kjerkoli. Pri zabojnikih Docker gre za skupek programske opreme, ki vsebuje vse potrebno za samostojen zagon storitev. Vsak nameščeni zabojnik Docker je popolnoma ločen in izoliran od drugih zabojnikov (Raj, Chelladhurai & Singh, 2015).

Osnovna ideja Dockerja je zapakirati aplikacijo z vsemi odvisnosti (binarne datoteke, knjižnice, konfiguracijske datoteke, skripte itd.) v eno standardizirano enoto za razvoj in uvajanje programske opreme. Dockerjevi zabojniki del programske opreme zavijejo v celoten datotečni sistem, ki vsebuje vse za zagon aplikacije (kodo, izvajalno okolje, sistemska orodja, sistemske knjižnice itd.), kar zagotavlja, da bo aplikacija vedno delovala na enak način, ne glede na to, v katerem okolju bo nameščena (Jaroslaw, 2016).

Na spletu se lahko najde številne prednosti zabojniške tehnologije. Glavne prednosti so (Jaroslaw, 2016):

- hitrost in majhnost,
- ponovljivi in prenosni gradniki,
- agilna infrastruktura,
- modularnost in razširjanje,
- stroškovna učinkovitost.

3.1.5 Upravljanje mikrostoritev

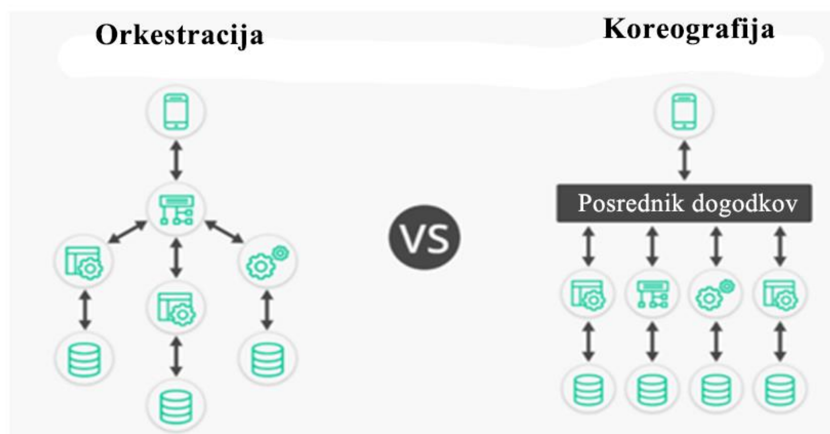
Kljub številnim prednostim, ki jih prinaša mikrostoritveni arhitekturni slog v primerjavi z monolitnim, se pojavljajo tudi izzivi. Eden od takih izzivov je povezan s sodelovanjem med posameznimi storitvami znotraj aplikacije. Ker se poslovni procesi in zahteve povečujejo, se povečuje tudi splošna kompleksnost poslovnih aplikacij. Mikrostoritve v mikrostoritveni arhitekturi se lahko nahajajo na različnih lokacijah (IP-naslovih) oziroma na različnih strežnikih, zato mora storitev, ki želi komunicirati z drugo storitvijo, vedeti, katera ta storitev je in kje se nahaja. Za upravljanje in komunikacijo med storitvami se uporabljata dva različna pristopa. Podjetja lahko izbirajo med orkestracijo in koreografijo. Pri orkestraciji gre za centraliziran proces, ki usklajuje vrsto pozivov (klicev) in storitev, medtem ko koreografija predstavlja decentralizirano usklajevanje storitev (Monteiro, Maia, Rocha & Mendonca, 2020).

Kot pravi Bonham (2017), je orkestracija tradicionalni pristop za upravljanje povezav med različnimi storitvami. Pri tem pristopu imamo običajno enega »orkestratorja«, ki skrbi za upravljanje vseh interakcij med storitvami. Ta običajno sledi vzorcu zahteve in odgovora. Za primer lahko vzamemo, da imamo tri storitve, ki morajo biti klicane v določenem vrstnem redu. Orkestrator izvede klice do vsake storitve in nato čaka odgovor posamezne storitve, preden

pokliče naslednjo storitev. Prednost takega upravljanja storitev je zagotavljanje dobrega načina za nadzor pretoka aplikacije pri sinhroni obdelavi. Sinhrona obdelava pomeni, da posamezna operacija blokira celoten proces, dokler ena operacija ni dokončana. Orkestrirano upravljanje storitev pa ima tudi slabosti, saj ustvarja odvisnosti med storitvami; če denimo ne deluje storitev A, potem storitvi B in C (če so povezane) ne bosta nikoli poklicani. Poleg tega imamo pri tem pristopu enega orkestratorja za vse zahteve, kar pomeni, da se celoten proces ustavi, če nastane napaka pri orkestraciji (Bonham, 2017).

Ker se želi pri mikrostoritvah izogniti odvisnostim na način, da lahko vsaka storitev deluje neodvisno, je poleg orkestracije na voljo tudi koreografija. Koreografija rešuje glavni problem orkestriranega pristopa. Pri koreografiji vsaka storitev izvaja svoje postopke neodvisno in ne potrebuje nobenih navodil orkestratorja. Gre za decentraliziran pristop oddajanja podatkov, znan kot dogodki. Storitve, ki se zanimajo za te dogodke, jih bodo uporabile in izvedle postopek. Storitve pri koreografiji vedo, na katere dogodke se morajo odzvati in na kakšen način to storiti. Za razliko od orkestracije gre za bolj asinhroni pristop (Singh, 2021). Pri koreografiji je osrednji element posrednik dogodkov (angl. event broker). Posrednik dogodkov skrbi za izmenjavo sporočil med storitvami, kadar se nekaj zgodi. Pri koreografiji je storitev opravila svojo nalogo, ko pošlje sporočilo naprej. Vse ostalo se dogaja v asinhronem načinu, brez čakanja na odgovor ali skrbi, kaj se bo zgodilo naslednje (Schabowsky, 2019). Na sliki 6 je prikazan orkestriran in koreografski način upravljanja mikrostoritev.

Slika 6: Orkestracija in koreografija storitev



Prirejeno po Schabowsky (2019).

3.2 Kdaj uporabiti mikrostoritve

Organizacije uporabljajo mikrostoritve za različne vrste poslovnih nalog, ki morajo biti izvršene. Določene aplikacije so na začetku razvite kot monoliti, saj so kot take relativno majhne, kar predstavlja enostaven razvoj in večjo usmerjenost na samo aplikacijo in ne na arhitekturo. Ker pa skozi čas aplikacija zraste in postane kompleksnejša jo je težje vzdrževati

na monolitni arhitekturi. Zaradi težkega vzdrževanja in drugih dejavnikov, kot je na primer težje razširjanje aplikacije, se podjetja in organizacije odločajo za prehod na mikrostoritveno arhitekturo. Obstajajo pa določene splošne predpostavke, kdaj je namesto monolitnega arhitekturnega sloga smiselno uporabiti mikrostoritve.

V splošnem se podjetja in organizacije odločajo za mikrostoritveno arhitekturo, kadar je aplikacija, sposobna opravljati veliko različnih nalog na različnih področjih. Tukaj naj bi šlo predvsem za logiko, da je lažje upravljati z vsako nalogo posebej (mikrostoritvena aplikacija) kot z eno posamezno neodvisno enoto, ki je zapisana nekje v celotni kodi programa (monolitna aplikacija). Pri monolitni aplikaciji se lahko določena funkcionalnost navezuje na različne dele programa, kar lahko negativno vpliva na razvoj, saj se pri upravljanju in vzdrževanju aplikacije lahko pojavijo različne napake, ker en element v monolitni arhitekturi vpliva na drugega (kot hierarhija). To pomeni, da se je pri spremembi enega dela kode treba prepričati, da bo program v celoti zopet normalno deloval (Patel, 2018).

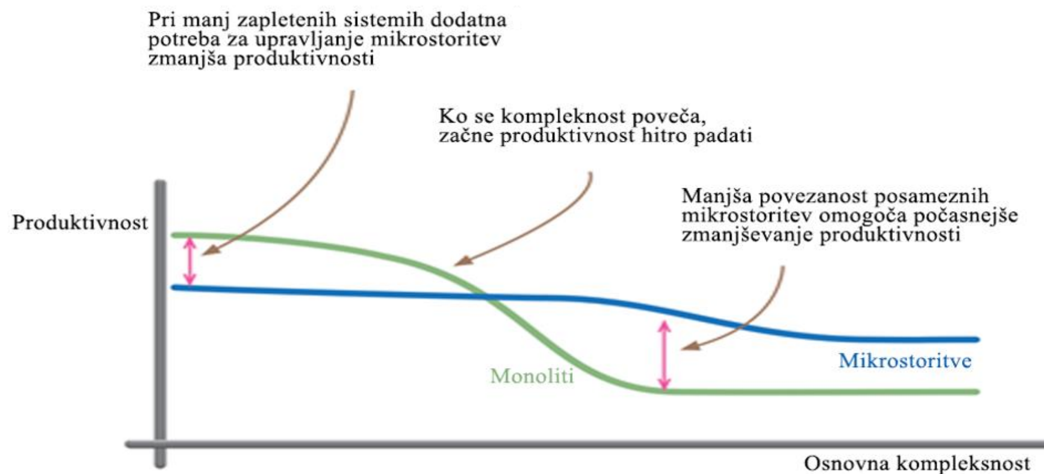
Mikrostoritve se velikokrat uporabljajo tudi takrat, kadar organizacije razpolagajo z več različnimi aplikacijami, ki so jim določene naloge skupne. V tem primeru je razvijalcem omogočena ponovna uporaba razvojne kode obstoječe aplikacije, namesto da bi podobno razvojno kodo ponovno pisali pri drugi aplikaciji. Največkrat prihaja do podvajanja nalog aplikacij pri avtorizacijah uporabnikov, možnosti iskanja v aplikaciji itd. Ker razvijalcem pri ponovni uporabi razvojne kode te ni treba ponovno pisati, se to pri organizaciji opazi kot prihranek v času in ostalih sredstvih, sama aplikacija pa omogoča razvijalcem lažji pregled nad njenim delovanjem (Fernando, 2018).

Kadar organizacija ima ali razvija aplikacijo, za katero se pričakuje, da jo bo treba velikokrat vzdrževati, posodabljati in dodajati nove funkcionalnosti, je smiselno, da razvijalci te organizacije premislijo o uporabi mikrostoritvene arhitekture. Pri mikrostoritveni arhitekturi organizacije opažajo boljše pogoje za testiranje novih funkcionalnosti, kar omogoča hitrejši razvoj in večjo fleksibilnost pri spreminjanju. Zaradi mikrostoritvene arhitekture se velikokrat spremeni tudi način posodabljanja programske opreme. Namesto tradicionalnih velikih posodobitev v daljšem obdobju razvijalci z mikrostoritveno arhitekturo uvajajo manjše posodobitve programske opreme na dnevni ravni (Fachat, 2019).

Mikrostoritveni arhitekturni pristop je v veliki meri odgovor na oteženo ravnanje z velikimi kompleksnimi monolitnimi sistemi. Pri uporabi mikrostoritev je treba delovati na avtomatiziranem uvajanju, spremljanju ter odpravljanju napak in ostalih stvari, ki jih prinaša distribuiran sistem. Za obvladovanje omenjenih značilnosti distribuiranega sistema obstajajo različni načini, vendar so za to potrebni dodatni napori, zato je eno od glavnih vodil to, da naj bi se mikrostoritvena arhitektura uporabila takrat, kadar ima organizacija sistem, ki je prezapleten za upravljanje na monolitni arhitekturi (Karwatka in drugi, brez datuma).

Kadar je kompleksnost logike aplikacije majhna, je produktivnost večja na strani monolitne arhitekture zaradi lažjega vzdrževanja ene same baze razvojne kode. Pri mikrostoritvah je pri majhni kompleksnosti logike aplikacije potrebno upoštevati še kompleksnost arhitekture zaradi povezovanja komponent (storitev) med seboj, kar znatno vpliva na produktivnost. Kadar se kompleksnost logike poveča, se produktivnost monolitne arhitekture v primerjavi z mikrostoritveno arhitekturo znatno zmanjša, kot je razvidno iz slike 7.

Slika 7: Kompleksnost in produktivnost glede na arhitekturni slog



Prirejeno po Karwatka in drugi (brez datuma).

Produktivnost se pri mikrostoritveni arhitekturi zmanjša zaradi večjega števila storitev, ki jih je treba povezovati, in veliko manj zaradi same razvojne kode, saj ta ostaja v majhnih storitvah, kar omogoča lažje vzdrževanje, upravljanje in testiranje. Pri monolitni arhitekturi pa se poleg kompleksnosti zaradi ene same baze razvojne kode poveča tudi kompleksnost zaradi povezanih enot in zaradi testiranja, saj je pri vsakem vzdrževanju in dodajanju funkcionalnosti treba testirati celotno programsko opremo, medtem ko se pri mikrostoritvah lahko testira posamezno storitev (Karwatka in drugi, brez datuma).

3.3 Prednosti in slabosti uporabe mikrostoritev

Aplikacije. razvite na mikrostoritveni arhitekturi, imajo v primerjavi z monolitnimi aplikacijami veliko različnih prednosti. Nekatere koristi mikrostoritev so vidne že takoj po implementaciji, nekatere pa se lahko opazi šele na dolgi rok. Kljub velikim koristim pa imajo mikrostoritve tudi nekaj slabosti, ki so vidne predvsem pri manjših aplikacijah, oziroma kadar aplikacije na dolgi rok nima smisla razširjati ali ji dodajati novih funkcionalnosti. V literaturi najdemo številne prednosti mikrostoritev, kot so (Jeremy, 2020):

- tehnološka neodvisnost,
- odpornost na napake,

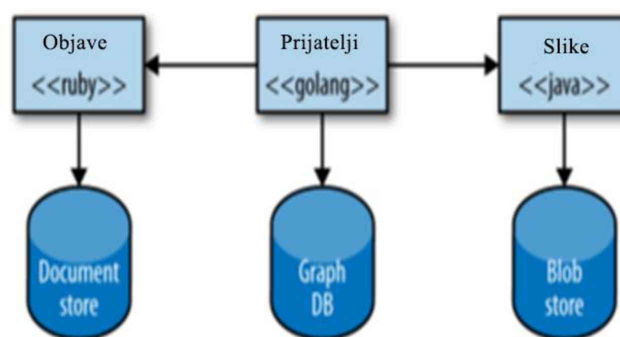
- razširjanje,
- enostavno uvajanje,
- organizacijska uskladitev,
- združljivost/kompatibilnost in
- optimizacija za nadomestljivost.

3.3.1 Tehnološka neodvisnost

Ena večjih prednosti mikrostoritvene arhitekture v primerjavi z monolitno je v tehnološki neodvisnosti. Ker so mikrostoritve sestavljene iz več sodelujočih storitev, se lahko razvijalci odločijo za uporabo različnih tehnologij v vsaki storitvi. Tehnološka neodvisnost omogoča, da razvijalci uporabijo primerno orodje za katerokoli nalogo, namesto da bi imeli možnost izbora med standardiziranimi univerzalnimi pristopi.

V kolikor določen del sistema potrebuje izboljšano zmogljivost, se lahko razvijalci zaradi tehnološke neodvisnosti mikrostoritev odločijo za uporabo različnega nabora tehnologij. Z različnim naborom tehnologije se lahko doseže željeni ali celo višji nivo zmogljivosti. Razvijalci se lahko prosto odločajo tudi o tem, kako bodo podatki shranjeni za različne dele sistema/aplikacije. Družabna omrežja lahko denimo shranjujejo interakcije uporabnikov v grafično orientirano bazo podatkov, objave uporabnikov pa so lahko shranjene v dokumentno orientirani shrambi podatkov, kar omogoča heterogeno arhitekturo. Heterogenost arhitekture je prikazana na sliki 8, kjer je za eno aplikacijo uporabljenih več različnih vrst tehnologij (Ayay, 2020).

Slika 8: Heterogenost mikrostoritvene arhitekture



Prirejeno po Newman (2015).

Tehnološka heterogenost dovoljuje razvijalcem dodajanje novih tehnologij tudi v času razvoja ali vzdrževanja sistema oziroma storitev na učinkovit način. Zaradi tega podjetja skozi čas niso odvisna le od določene tehnologije, ki se jo uporablja pri aplikaciji, ampak lahko, ko pride do

zastaranja tehnologije, aplikacijo preprosto posodobijo z novejšo oziroma primernejšo tehnologijo.

Veliko podjetij in organizacij je mnenja, da tehnološka heterogenost pripomore k hitrejši absorpciji novih tehnologij. Vendar pa obstaja možnost, da zaradi prevelike uporabe različnih tehnologij pride do velike kompleksnosti celotnega sistema, kar prinaša slabosti zaradi daljšega časa vzdrževanja takih sistemov in večjih možnosti za napake, poleg tega pa je ob večjih fluktuacijah težko najti razvijalce, ki obvladajo točno določene tehnologije, s katerimi so bile storitve razvite. Zaradi tega se številne organizacije odločajo o omejitvah predvsem pri izbiri programskega jezika za razvijanje aplikacij. Te omejitve so podane v smislu števila programskih jezikov, naboru željenih programskih jezikov itd. (Newman, 2015).

3.3.2 Odpornost na napake

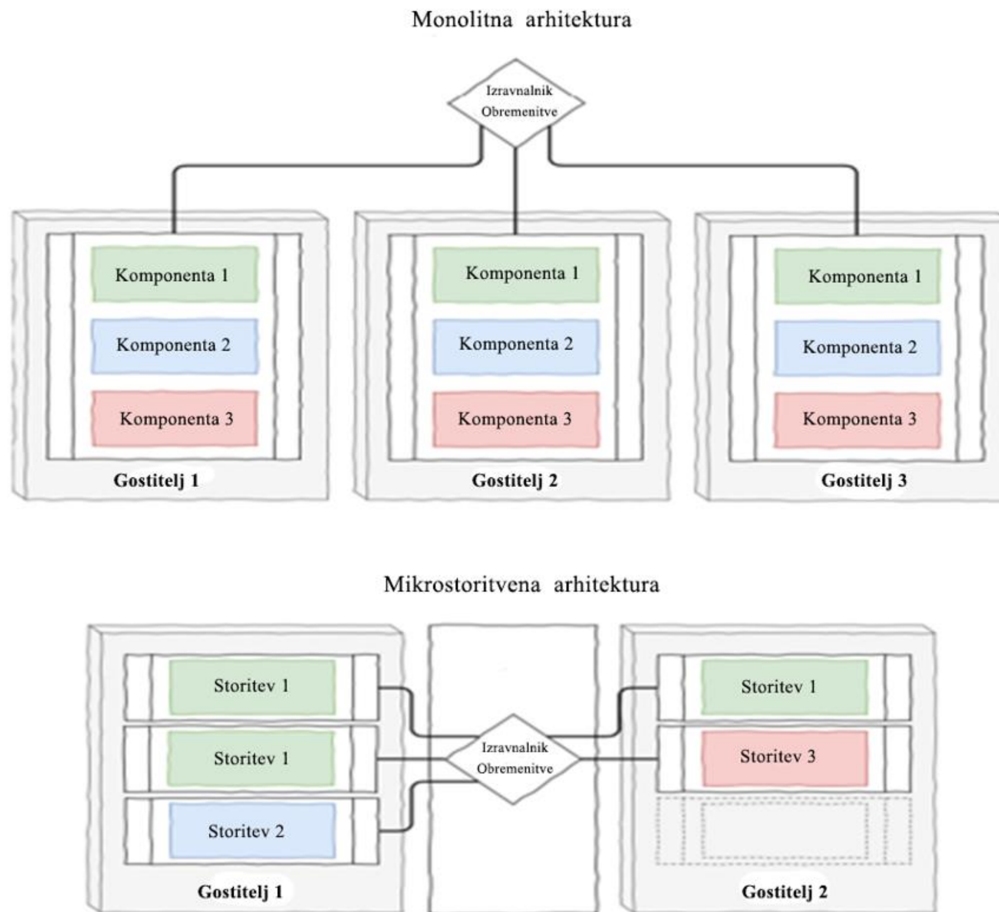
Ključni koncept odpornosti na napake pri mikrostoritveni arhitekturi je t. i. pregrada: če določena komponenta sistema odpove, se odpoved nanaša le na to in povezane komponente, preostale komponente pa lahko ne glede na odpovedano komponento nemoteno delujejo. V programskem inženirstvu se to imenuje izolacija problema. Pri mikrostoritvah storitvene meje postanejo pregrade. Pri monolitni arhitekturi pa je problem v tem, da celotna aplikacija preneha delovati, če odpove določena storitev. Pri monolitni aplikaciji se ta problem rešuje z zagonom več enakih aplikacij na več strežnikih/računalnikih, da se zmanjša možnost motenega delovanja sistema. Z mikrostoritveno arhitekturo se lahko zgradi aplikacijo, ki obvladuje popolno okvaro storitev in ustrezno poslabša funkcionalnost aplikacije do ponovne vzpostavitve oz. zagona storitve (Jeremy, 2020).

3.3.3 Razširjanje

Ena od večjih stroškovnih prednosti IT infrastrukture je lažje razširjanje mikrostoritvene arhitekture v primerjavi z monolitno. Pri monolitnih aplikacijah je v primeru preobremenjenosti določene komponente potrebno razširiti celotno aplikacijo z vsemi njenimi moduli. Mikrostoritvena arhitektura omogoča razširjanje samo tistih storitev, ki potrebujejo razširjanje zaradi pomanjkanja zmogljivosti, medtem ko ostale storitve aplikacije ostanejo na obstoječi strojni opremi, kot to prikazuje slika 9 (Dragoni in drugi, 2017).

Na sliki 9 so prikazane tri storitve oziroma komponente. Vsaka izmed njih opravlja določeno funkcionalnost. V zgornjem delu slike je prikazano delovanje na monolitne arhitekturi, v spodnjem delu na mikrostoritveni. Iz slike lahko razberemo, da je storitev/komponenta 1 najverjetneje ozko grlo celotne aplikacije. V primeru monolitne arhitekture bi bilo potrebno zaradi ozkega grla namestiti tri enake aplikacije na tri različne strežnike. V primeru mikrostoritvene arhitekture pa bi se lahko na strežnik namestilo več enakih storitev, ki predstavljajo ozko grlo. S tem bi se porabilo manj virov iz naslova števila strežnikov in porabe ostalih zmogljivosti (Dragoni in drugi, 2017).

Slika 9: Razširjanje monolitne in mikrostoritvene arhitekture



Prirejeno po Lanese in drugi (2017).

3.3.4 Organizacijska uskladitev

Pri monolitni arhitekturi se pojavlja velik problem, ki se nanaša na velike skupine in velike baze programske kode. Ta problem pa se lahko še poslabša, kadar imamo pri monolitni arhitekturi porazdeljene skupine. Mikrostoritve pri organizacijski uskladitvi dovoljujejo, da se arhitekturo prilagodi glede na organizacijo, kar pripomore k minimiziranju števila zaposlenih, ki delajo na določeni bazi programske kode oziroma storitvi. Manjše število zaposlenih, ki delujejo na določeni programski kodi, pa vodi k večji produktivnosti in manjšim stroškom (Jeremy, 2020).

Poleg opisanih koristi glede organizacijske uskladitve mikrostoritve dovoljujejo, da se porazdeli lastništvo storitev med razvojnimi skupinami, kar omogoča, da zaposleni delajo na točno določenih storitvah. To povečuje produktivnost in minimizira število zaposlenih v posameznih razvojnih skupinah. Majhnost skupin pa dovoljuje lažjo komunikacijo in koordinacijo tako med skupinami kot tudi znotraj njih.

3.3.5 Slabosti mikrostoritvene arhitekture

Poleg številnih že omenjenih prednosti imajo mikrostoritve tudi nekaj slabosti. Te slabosti so (Wittmer, 2020):

- kompleksen razvoj in testiranje,
- zahteve po kulturnih spremembah podjetja,
- vzdrževanje številnih podatkovnih baz,
- drag začetni razvoj.

Integracija številnih storitev v mikrostoritveni arhitekturi zahteva znanje medprocesnih komunikacij, kar so po navadi API-ji. Slabo razvite ali konfigurirane storitve imajo lahko težave pri vzpostavljanju komunikacije s preostalim sistemom, kar pomeni daljši celotni čas izvajanja. Kljub temu, da je pri mikrostoritveni arhitekturi lažje izvesti nadgrajevanje in razhroščevanje posameznih storitev zaradi nepovezane arhitekture, pa v nepovezani arhitekturi leži tudi razlog, ki razvijalcem otežuje razvoj in testiranje celotnega sistema, zgrajenega iz več storitev. Testiranje individualne storitve je enostavno, vendar pa ker je sistem zgrajen iz več storitev, nameščenih na različnih strežnikih, otežuje testiranje celotnega sistema. Tudi sama komunikacija med storitvami je lahko zelo kompleksna, saj lahko aplikacija vsebuje tudi več sto različnih storitev, ki morajo med seboj varno komunicirati (Diguer, 2020).

Poleg samih tehnoloških izzivov, ki jih prinaša mikrostoritvena arhitektura, se pojavljajo tudi organizacijski izzivi, saj mikrostoritvena arhitektura zahteva tudi kulturne spremembe v primeru prehoda. Še preden se prične proces migracije, je treba doseči zrelo agilno in DevOps kulturo. Že sama ideja mikrostoritev, kjer vsaka mikrostoritev deluje kot neodvisna enota, omogoča delovanje majhnih skupin na manjših projektih, ki delujejo neodvisno, kar je popolnoma drugače kot pri monolitni arhitekturi. Zaradi potrebnih sprememb je potrebno zelo dolgo, da se razvijalci privadijo na novo organizacijsko kulturo (Wittmer, 2020).

4 PRIMERJAVA MONOLITOV IN MIKROSTORITEV

Monolitna in mikrostoritvena arhitektura imata vsaka svoja svoje prednosti in slabosti. Nobena ni boljša od druge, saj je njuna uporabnost odvisna od posameznega primera aplikacije. Monolitna arhitektura je še vedno standarden način začetka grajenja aplikacij. Kljub temu, da je mikrostoritvena arhitektura v zadnjih nekaj letih pridobila vse več podpore in popularnosti, je aplikacijo v splošnem še vedno bolje začeti graditi z monolitno arhitekturo.

4.1 Razvoj

Razvoj popolnoma nove aplikacije je vsekakor lažje začeti z monolitno arhitekturo. Ker pa se funkcionalnost aplikacije čez čas veča, se povečuje tudi razvojna koda monolita. Zaradi velike razvojne kode pri monolitnih aplikacijah narastejo tudi problemi, povezani z monolitno arhitekturo. Mikrostoritvena arhitektura na drugi strani lažje obravnava večje aplikacije v

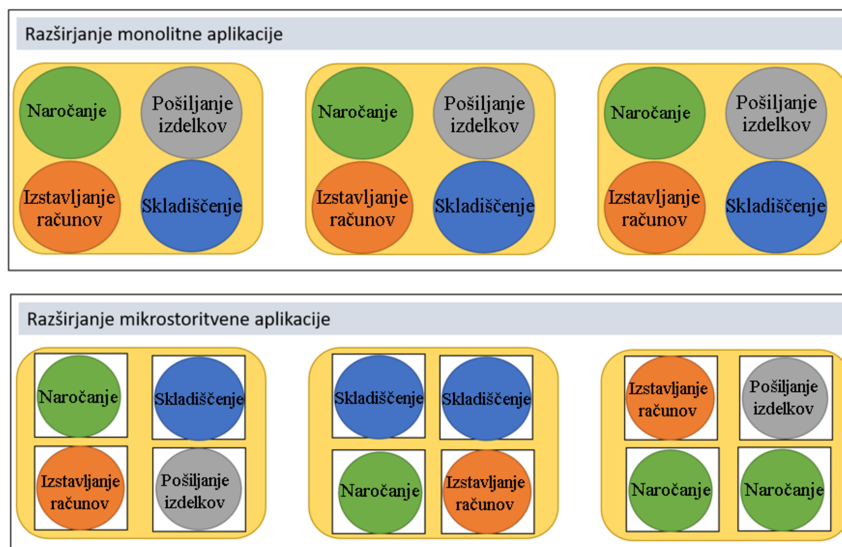
smislu same velikosti razvojne kode in tudi kompleksnosti. Razvoj novih funkcionalnosti je po navadi lažji z manjšo razvojno kodo, večja modularnost pa pripomore k temu, da razvijalci hitreje namestijo zahtevane spremembe. Zaradi manjše modularnosti monolitne arhitekture z veliko bazo razvojne kode prihaja do težav pri razvoju novih funkcionalnosti, saj ta razvoj velikokrat postane dolgotrajen, kar vpliva na stroške in konkurenčnost podjetja ali organizacije (Richardson, brez datuma).

4.2 Razširjanje

Tako monolitno, kot tudi mikrostoritveno arhitekturo se lahko razširja v obe smeri: horizontalno in vertikalno. Glavna razlika pri razširjanju teh dveh arhitektur je, da mikrostoritve dovoljujejo bolj drobnozrnato razširjanje v primerjavi z monolitno arhitekturo (Dragoni in drugi, 2017).

Mikrostoritvena arhitektura omogoča razširjanje glede na funkcionalne enote (storitve), pri monolitni arhitekturi pa je treba podvojiti celotno aplikacijo za nove instance/primerke. Zaradi tega se lahko v primeru mikrostoritvene aplikacije storitve z veliko obremenitvijo razširja neodvisno ne glede na celoten sistem. Na sliki 10 je prikazano horizontalno razširjanje monolitne in mikrostoritvene aplikacije. Zgornji del slike prikazuje razširjanje monolitne aplikacije, kjer je vsak del aplikacije razširjen skupaj z drugimi. Če je v aplikaciji na primer določen modul, ki zahteva več razširjanja (npr. naročanje), potem je treba namestiti celotno monolitno aplikacijo, da se lahko razširi ta modul. Na spodnjem delu slike pa je prikazano razširjanje mikrostoritvene arhitekture z enakimi elementi aplikacije kot pri monolitni arhitekturi. Razlika glede na monolitno arhitekturo je ta, da je z mikrostoritveno arhitekturo možno namestiti različno število storitev, ki so lahko enake ali pa popolnoma različne (Dragoni in drugi, 2017).

Slika 10: Pregled razširjanja arhitektur



Prirejeno po Ayay (2020).

4.3. Testiranje

Strategije testiranja se med monolitno in mikrostoritveno arhitekturo ne razlikujejo bistveno. Največ pozornosti je pri mikrostoritveni arhitekturi treba nameniti hitro delujočim avtomatiziranim testiranjem, kot je npr. »unit test«. Da bi se čim bolje izkoristilo mikrostoritveno arhitekturo, je nujno stremeti k nenehnemu uvajanju novih različic aplikacije. Nenehno uvajanje novih različic pa pomeni, da mora biti vsaka sprememba aplikacije hitro in temeljito pregledana in potrjena z učinkovitimi avtomatiziranimi testi. Za razliko od mikrostoritvenih aplikacij so pri monolitnih aplikacijah izdajni cikli daljši, kar omogoča razvijalcem več časa za dolgotrajna testiranja in tudi ročna preizkušanja. Pri mikrostoritvah je zaradi krajših izdajnih ciklov potrebna večja pokritost z avtomatiziranimi testiranji (Cser, 2018).

Ustvarjanje avtomatiziranih testov je pri mikrostoritvah lažje, saj ima vsaka posamezna storitev samo eno odgovornost. Zaradi tega je razvijalcem razumljivejše, kaj je namen posamezne storitve. Znotraj posamezne storitve je treba slediti dobrim standardom programiranja, da se lahko jasno opredeli odgovornost vsake metode in se ohrani storitvena majhnost, kar olajšuje testiranje. Na drugi strani pa pri monolitnih aplikacijah obstaja tveganje, da skozi čas aplikacija zraste v smislu števila razredov in metod aplikacije, kar znatno otežuje njihovo testiranje, poleg tega pa se pri tem porabi več časa in sredstev v primerjavi z mikrostoritvenimi aplikacijami (Cser, 2018).

5 PREHOD NA MIKROSTORITVE

Organizacijska agilnost ter zmožnost fleksibilnega odgovora na spremembe v okolju z hitro spremembo proizvodov in storitev sta postali ključnega pomena pri ohranjanju konkurenčnosti podjetja. V današnjem času veliko podjetij in organizacij za svojo poslovno programsko opremo še vedno uporablja monolitno arhitekturo. Zaradi dodajanja novih funkcionalnosti monolitne aplikacije skozi čas rastejo ter zato postanejo velike in kompleksne. Zaradi narave monolitne arhitekture je take sisteme težko vzdrževati, nekatere monolitne aplikacije postanejo v poznejših fazah celo okamenele za dodajanje novih posodobitev, kar močno vpliva na konkurenčnost in razvoj. Posledica tega je, da zaradi takih sistemov nastanejo veliki stroški in se je težje prilagajati spremembam, čemur sledi tudi organizacijska neučinkovitost. Zaradi tega se vse več podjetij in organizacij odloča za prehod svojih aplikacij iz monolitne arhitekture na arhitekturo mikrostoritev. Proces prehoda na mikrostoritveno arhitekturo je obsežen in dolgotrajen proces, ki zahteva skrbno planiranje in temeljite premisleke o posledicah migracije in njegovih izzivih (Baškarada, Nguyen & Koronios, 2018).

Podjetja in organizacije se za prehod na mikrostoritveno arhitekturo odločajo zaradi različnih razlogov. Najpogostejši razlogi za prehod so: boljši pogoji za vzdrževanje, možnost krajših izdajnih ciklov posodobitev, hitrejše razširjanje in visoka dostopnost. Kljub mnogim prednostim, ki jih ponuja mikrostoritvena arhitektura, pa ta prinaša tudi veliko pasti, zato je

potrebna velika pazljivost pri odločitvi za prehod, saj je celoten migracijski proces dolgotrajen in prinaša veliko stroškov, ki so lahko tudi skriti. Poleg samih stroškov pa je treba pridobiti veliko novega znanja, poleg predelave same arhitekture in aplikacije pa lahko sledijo celo organizacijska prestrukturiranja (Fritzsch, Bogner, Wagner & Zimmermann, 2019).

Ker je prehod na mikrostoritveno arhitekturo v zadnjih nekaj letih zelo pogosta tema v akademskem in tudi poslovnem svetu, se je na spletu pojavilo veliko principov, metod, tehnologij in orodij, ki omogočajo enostavnejši prehod. Vse te metode in orodja ponujajo različne koristi, zato je pomembno, da se arhitekti odločijo za pravo, saj bo to pozneje vplivalo na celotno poslovanje.

5.1 Tipi/vrste prehodov na mikrostoritve

Podjetja in organizacije se v večini primerov odločajo med tremi različnimi načini migracije iz monolitne v mikrostoritveno arhitekturo:

- sočasno vzporedno delovanje na obeh arhitekturah,
- postopen prehod posameznih funkcij in
- »big bang« prehod iz monolitne na mikrostoriveno arhitekturo.

Na izbiro načina prehoda vpliva več stvari. Med največkrat uporabljenimi razlogi za izbor določenega načina so stanje trenutnega sistema, ki temelji na monolitni arhitekturi, sredstva in viri ter izkušenosti zaposlenih na IT-področju glede mikrostoritvene arhitekture.

Pri vrsti prehoda na mikrostoritveno arhitekturo, ki sloni na sočasnem delovanju obeh arhitektur, se zagotavlja sočasno delovanje tako mikrostoritvene kot tudi monolitne programske opreme. Pri sočasnem delovanju razvijalci razvijejo mikrostoritveno aplikacijo, nato pa se za določen čas izvede paralelno delovanje aplikacije na obeh arhitekturah. Cilj tega pristopa je zmanjševanje tveganja za morebitni izpad v primeru napak na mikrostoritveni arhitekturi. V primeru napake se ta popravi, med izvajanjem popravkov pa aplikacija na monolitni arhitekturi deluje nemoteno. Slabost tega pristopa je večja poraba sredstev in časa, saj je treba usklajevati delovanje obeh sistemov (kar se zgodi v monolitni programski opremi, se mora zgoditi tudi v mikrostoritveni) (Megargel & Shankararaman, 2020).

V primeru postopnega prehoda iz monolitne na mikrostoritveno aplikacijsko arhitekturo razvijalci posamezne funkcije monolitne programske opreme postopoma nadomeščajo z mikrostoritveno programsko opremo. Ta proces ponazarja postopek postopne menjave komponent (angl. strangler fig pattern), ki je podrobneje opisan v poglavju 5.5.1. Prednost postopnega prehoda je v tem, da za razliko od sočasnega delovanja v primeru postopnega prehoda ni treba skrbeti za konsistenčnost podatkov in funkcij na obeh programskih opremah, saj je določena funkcija izvedena samo enkrat na eni arhitekturi, odvisno od tega, ali ta funkcija/storitev že deluje na mikrostoritveni ali deluje še na monolitni arhitekturi (do prehoda

storitve v tem primeru še ni prišlo, bo pa prišlo v prihodnosti). Slabost tega pristopa je v kompleksnosti, saj je treba vzpostavljati povezave med monolitno in mikrostoritveno programsko opremo, kar zahteva znanje in druge vire (Li, Ma & Lu, 2020).

Zadnja vrsta značilnih prehodov na mikrostoritve, ki je pogosto uporabljena je t. i. big bang prehod (prehod v smislu velikega poka). Big bang prehod se izvrši tako, da v določenem trenutku mikrostoritvena programska oprema v celoti nadomesti monolitno, ki v tem primeru nemudoma preneha z delovanjem. Pri tej vrsta prehoda je velika prednost v prihrankih virov, saj ni treba vzdrževati dveh programskih oprem, ampak samo eno. Slabost pa se največkrat odraža, kadar pride do napak mikrostoritvene programske opreme, saj lahko te napake ogrozijo celotno delovanje aplikacije (Lu, Glatz & Peuser, 2019).

5.2 Razlogi za prehod

Razlogi za selitev z monolitne na mikrostoritveno arhitekturi so različni. Kot že omenjeno, sta med glavnimi razlogi za prehod večja produktivnost in agilnost. Mikrostoritve so v primerjavi z monolitnimi aplikacijami manjše in manj kompleksne v smislu posamezne funkcionalnosti, zato jih razvijalci lažje posodabljajo in dodajajo nove funkcionalnosti. Naslednja izmed najpogostejših razlogov sta pričakovana izboljšana kvaliteta programske opreme in olajšanje vzdrževanja (Lenarduzzi, Lomio, Saarimaki & Taibi, 2020).

V raziskavi, ki so jo izvedli Taibi, Lenarduzzi in Pahl (2017), med glavnimi gonilniki za prehod iz monolitov na mikrostoritve navajajo vzdrževanje, razširljivost, delegacijo skupinskih odgovornosti, DevOps podporo, odpornost do napak, enostavno eksperimentiranje s tehnologijo in tudi popularnost tega početja (Taibi, Lenarduzzi & Pahl, 2017). Poleg omenjenih gonilnikov Li, Ma in Lu (2020) v svojem delu kot dejavnike na podlagi, katerih se podjetja in organizacije odločajo za prehod na mikrostoritve, opisujejo še nadzor in varnost.

Modularna arhitektura mikrostoritev dovoljuje zmanjševanje kompleksnosti monolitnih sistemov. Ker se sistem pri mikrostoritvah razčleni na neodvisne in samostojne storitve, to omogoča razvijalskim skupinam, da ustvarjajo spremembe in jih testirajo neodvisno od drugih storitev in drugih razvijalcev, kar poenostavlja porazdeljeni razvoj. Poleg tega pa manjše mikrostoritve prispevajo k boljšemu pregledu in razumevanju kode, kar vodi k izboljševanju vzdrževanja sistema (Taibi, Lenarduzzi & Pahl, 2017).

Kot razlog za prehod na mikrostoritve Taibi, Lenarduzzi in Pahl (2017) opisujejo tudi razširjanje. Razširjanje pomeni možnost dodajanja novih virov za obravnavo različnega števila zahtev, ki prihajajo do aplikacije. V splošnem se razširjanje loči na vertikalno in horizontalno. Zmožnost vertikalne razširljivosti pomeni zmožnost dodajanja novih virov fizičnim enotam, na katerih je nameščena aplikacija. Fizične enote so na primer spomin računalnika, procesorska moč itd. Horizontalna razširljivost pa pomeni zmožnost dodajanja virov kot fizičnih enot, na primer novih strežnikov, na katerih je nameščena aplikacija (Li, Ma & Lu, 2020). Razširjanje

mikrostoritev je enostavnejše kot razširjanje monolitov. V kolikor želijo razvijalci razširiti monolitni sistem, to zahteva velike naložbe v smislu strojne opreme in velikokrat tudi prilagoditve kode. Če pride do ozkega grla pri določeni komponenti aplikacije, se lahko za njegovo odpravo posodobi strojna oprema, na kateri deluje aplikacija, lahko pa se izda več primerkov ene monolitne aplikacije, ki je izvršena med strežniki in upravljana z izravnalnikom obremenitve (angl. load balancer). Če vzamemo mikrostoritve, pa je lahko vsaka posamezna mikrostoritev nameščena na različnem strežniku z različnimi nivoji zmogljivosti in je lahko napisana v zanjo najprimernejšem jeziku (Taibi, Lenarduzzi & Pahl, 2017).

Podjetja in organizacije se za prehod na mikrostoritve odločajo tudi zaradi boljše možnosti za delegiranje skupinskih odgovornosti. Ker mikrostoritve nimajo eksternih (zunanjih) odvisnosti, jih lahko različne razvijalske skupine razvijajo samostojno, kar zmanjšuje obremenitve komunikacije znotraj razvojnih skupin kot tudi med njimi, kar zmanjšuje tudi potrebe po usklajevanju. Vsaka razvijalska skupina si lasti svojo kodo in je odgovorna samo za razvoj storitve, ki si jo lasti, in ne celotne aplikacije. Mikrostoritvena arhitektura omogoča porazdelitev jasnih in neodvisnih odgovornosti med razvojne skupine ter razdelitev velikih projektnih skupin, ki bi nastale pri monolitni arhitekturi, na več majhnih, kar povečuje učinkovitost. Poleg tega pa je treba zaradi neodvisnosti od izbire tehnologije in drugačne interne strukture koordinirati samo najvišje tehnične odločitve, medtem ko se za ostale odločitve odloči posamezna razvojna skupina (Taibi, Lenarduzzi & Pahl, 2017).

5.3 Razgradnja monolitov v mikrostoritve

Pri prehodu z monolitne na mikrostoritveno arhitekturo se organizacije srečujejo s številnimi izzivi, med katerimi je eden največjih ta, kako monolitno aplikacijo učinkovito razgraditi v paket majhnih mikrostoritev. Pri tem procesu je najbolj kritična naloga določiti primerne particije monolitnega sistema, saj lahko mikrostoritvena arhitektura z različnimi granulacijami različno vpliva na kakovost sistema (npr. učinkovitost in možnost testiranja).

Razgradnja monolitnega sistema v sistem mikrostoritev je zelo podobna razgradnji storitveno usmerjene arhitekture (SOA) z dvema velikima razlikama. Prva razlika je ta, da so storitve v SOA grobo-, pri mikrostoritvah pa drobnozrnate. Druga razlika pa je ta, da se SOA osredotoča na izbiro optimalno sestavljenih storitev, ki jo razvijalci izberejo med vsemi možnimi storitvenimi kombinacijami glede na kvalitetne zahteve. V tem primeru gre za proces od spodaj navzgor (angl. bottom-up), pri mikrostoritveni arhitekturi pa se najprej uporabi proces razgradnje od zgoraj navzdol (angl. top-down), nato pa integracijo od spodaj navzgor.

Organizacije se razgrajevanja monolitne arhitekture lotevajo na različne načine. Načini razgradnje monolitov so predvsem ročni ter velikokrat vključujejo intuicijo in znanje strokovnjakov, ki se ukvarjajo z razgradnjo. Ker je razgradnja velikokrat prepuščena intuiciji strokovnjakov, to pomeni, da je njena kakovost v veliki meri pogojena z arhitektovimi ali razvijalskimi izkušnjami in znanji. Napačna izbira razgradnje monolitnega sistema je lahko

izjemno draga, saj lahko povzroči slabše delovanje celotnega sistema, prav tako pa to lahko vpliva na poznejše vzdrževanje in ne omogoča celotnega izkoristka mikrostoritvene arhitekture. Zato je pomembno, da je monolitni sistem razgrajen z dobro postavljenimi mejami posameznih storitev, kar omogoča, da so te ohlapno povezane. Ohlapno povezane storitve povečujejo izkoristek mikrostoritvene arhitekture in produktivnost razvojnih skupin. Da bi naslovili omenjene težave in izzive, so se v zadnjih nekaj letih razvili številni pristopi k razgrajevanju monolitnega sistema. Ti pristopi ne temeljijo zgolj na intuiciji razvijalcev, ampak tudi na različnih postopkih in pravilih. Nekateri od njih temeljijo na ročni, drugi pa na polavtomatski ali avtomatski razgradnji. Najbolj pogosto uporabljeni pristopi razgradnje monolitnega sistema so (Li in drugi, 2019):

- glede na poslovno zmožnost (angl. business capability),
- domensko usmerjen pristop (angl. domain-driven approach),
- razgradnja glede na glagol ali samostalnik,
- razgradnja glede na tok podatkov (angl. dataflow-driven approach).

Eden od najbolj znanih pristopov razgradnje, ki so ga uporabile tudi številne organizacije, je razgradnja glede na tok podatkov. Ta je enostavna za razumevanje, poleg tega pa ponuja razgradnjo na različne granulacije mikrostoritev. Različna granulacija mikrostoritev pri razgradnji glede na tok podatkov omogoča, da imajo razvijalci že v enem samem procesu razgradnje več različnih primerov granulacij, kar jim omogoča poznejše testiranje in izbiro najprimernejšega števila storitev. Pristop razgradnje glede na tok podatkov omenjajo tudi Li in drugi (2019), ki ga opišejo kot enega trenutno boljših pristopov k razgradnji monolitnega sistema, saj poleg razgradnje na različne granulacije omogoča tudi polavtomatski pristop razgradnje.

5.3.1 Razgradnja glede na tok podatkov

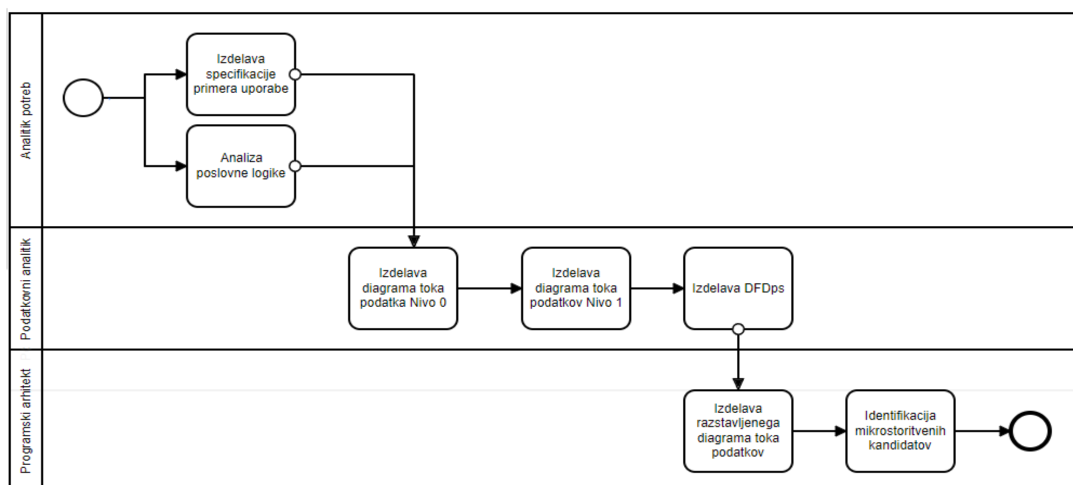
Razgradnja monolitne aplikacije glede na tok podatkov je metoda, ki se osredotoča na razgradnjo glede na poslovno logiko in tok informacij. Osrednji element te metode je diagram toka podatkov, na podlagi katerega se določi potencialne kandidate za mikrostoritve. Prednost te metode je, da za razliko od drugih dopušča možnost razgradnje monolitnega sistema na različne granulacije, tako da se izdela bolj in manj podrobne diagrame toka podatkov. Na ta način lahko razvijalci in arhitekti izberejo primerno granulacijo za razvoj posameznih mikrostoritev. Izbira primerne granulacije mikrostoritev igra ključno vlogo pri zmogljivosti aplikacije, saj ni nujno, da bo večja razdrobljenost tudi boljša. Gre za iskanje ravnotežja med razdrobljenostjo storitev, kar omogoča lažje prilagajanje aplikacije, in med zmogljivostjo aplikacije (Li in drugi, 2019).

Kot že omenjeno, je glavni element razgradnje glede na tok podatkov diagram toka podatkov. Ta je sestavljen iz štirih glavnih elementov, ki ponazarjajo, kako določena programska oprema deluje. Ti elementi so: proces, hramba podatkov, tok podatkov in zunanje entitete. Omenjeni

elementi se med seboj povezujejo, da se ustvari celoten proces, ki ponazori, kako se gibljejo podatki in kako deluje celotna aplikacija. Proces razgradnje sestavljajo trije različni diagrami toka podatkov. Ti se razlikujejo glede na granulacijo (podrobnosti) toka celotne aplikacije. Prvi diagram toka podatkov je imenovan kontekstni diagram in prikazuje programsko opremo na najvišjem nivoju abstrakcije s temeljnimi procesi, ki so potrebni za delovanje. Drugi diagram toka podatkov je detajlnejši od prvega. V tem koraku je vsak temeljni proces razgrajen še globlje – bolj drobnozrnato. Ta diagram se imenuje diagram nivo – 0. Zadnji diagram, ustvarjen s to razgradnjo, je imenovan nivo – 1. Ta diagram ponazarja detajlni potek procesov v diagramu na nivoju 0 in je najbolj drobnozrnati od vseh treh diagramov (Li in drugi, 2019).

Proces razgradnje glede na tok podatkov je sestavljen iz štirih glavnih korakov. V prvem koraku se analizira poslovno logiko s primeri uporabe in intervjuji uporabnikov programske opreme. Ta korak po navadi izvedejo analitiki, ki pozneje opredelijo še ključne elemente aplikacije (podatkovne shrambe, procese itd.). Na podlagi izdelane analize je naslednji korak ustvarjanje diagrama toka podatkov na nivoju 0. Ta diagram prikazuje grobo predstavo o tem, kako trenutni sistem deluje. Na podlagi tega diagrama se ustvari še diagram toka podatkov – nivo 1, ki predstavlja natančen prikaz delovanja sistema z vsemi procesi, tokovi podatkov, shrambami podatkov in zunanjimi entitetami. Nato se iz diagrama toka podatkov – nivo 1 izvzame vse zunanje entitete in toke podatkov, tako da na njem ostanejo samo še shrambe podatkov in procesi (Li in drugi, 2019). V tretjem koraku se iz zadnjega ustvarjenega diagrama izloči imena procesov in hramb podatkov (primer: proces, imenovan »pridobi status naročila«, se preimenuje v P1). Cilj preimenovanja je preoblikovanje pridobljenih podatkov na način, ki je primeren za računalniško identifikacijo kandidatov za posamezne mikrostoritve glede na odvisnosti procesov in hramb podatkov. Na podlagi preoblikovanih podatkov algoritem glede na povezave med hrambami podatkov in procesi določi primerne kandidate za mikrostoritve (Li in drugi, 2019). Celoten proces razgradnje monolitne programske opreme glede na tok podatkov, je prikazan na sliki 11.

Slika 11: Proces razgradnje monolitne programske opreme glede na tok podatkov



Prirejeno po Li in drugi (2019).

5.3.2 Razgradnja glede na domeno aplikacije

Moč mikrostoritev izhaja iz jasne opredelitve njihove odgovornosti in določitve meja med njimi. Domenska razgradnja je pogosta tehnika razgradnje monolitnega sistema, ki jo uporabljajo mnoga podjetja in organizacije ter ki pomaga določati jasne meje med mikrostoritvami. To tehniko je najbolje opisal Evans (2003) v svoji knjigi, kjer opisuje koncept domenske razgradnje kot zbirko principov, načel in vzorcev, kako oblikovati mikrostoritve. Glavno vodilo te razgradnje je, da se razvijalci in arhitekti pri preoblikovanju ne osredotočajo na tehnologijo, ampak na poslovno dejavnost, ki se jo želi z mikrostoritvijo podpreti (domeno). Domena pri razgradnji predstavlja, kaj podjetje ali organizacija v širšem smislu počne z monolitno aplikacijo, oziroma kateri problem z njo rešuje (Evans, 2003).

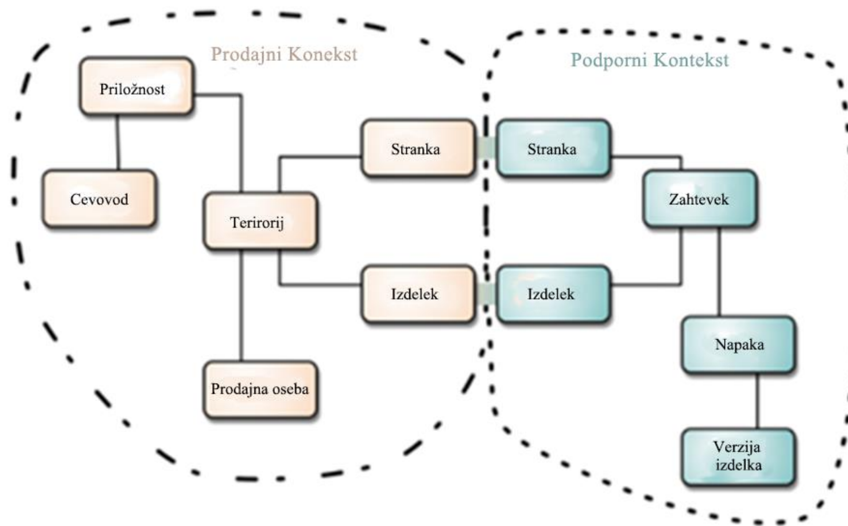
Pri domenski tehniki razgradnje se uporablja več različnih tehnik za oblikovanje novih mikrostoritev. Ena od teh tehnik je razgradnja domene na poddomene, ki predstavljajo funkcionalne, oziroma poslovne enote znotraj domene. Domena je tako sestavljena iz več poddomen, sama domenska razgradnja pa je sestavljena iz treh delov (Evans, 2003):

- vseprisoten jezik,
- strateško oblikovanje,
- taktično oblikovanje.

Za izdelavo modelov domenske tehnike razgradnje se uporablja vseprisoten jezik. Njegov namen je dobro razumevanje modela s strani vseh udeležencev (razvijalcev in uporabnikov), kaj določena stvar v modelu pomeni. Ta jezik je oblikovan v omejenem kontekstu (angl. bounded context). V tem kontekstu so opredeljeni izrazi in koncepti poslovne domene, zaradi česar ne bi smelo prihajati do dvoumnosti. Primer omejenega konteksta je prikazan na sliki 12, kjer ima na primer stranka v prodajnem kontekstu drugačen pomen kot v podpornem kontekstu (Fowler & Lewis, 2014).

Glavni steber domenske razgradnje je strateško oblikovanje. Glavni cilj tega oblikovanja je definiranje omejenih kontekstov in kontekstnega zemljevida skupaj s celotno projektno skupino, ki je sestavljena iz strokovnjakov za posamezna področja in tehnične skupine. Na sliki 12 sta prikazana dva omejena konteksta, in sicer prodajni in podporni kontekst. Omejeni kontekst predstavlja posamezno enoto/objekt v poslovnem procesu (npr. prodajni kontekst ali podporni kontekst). V celotnem modelu se nahaja več omejenih kontekstov. Omejeni konteksti niso vedno izolirani drug od drugega, ampak lahko med seboj komunicirajo. Interakcijo med posameznimi konteksti se opiše s kontekstnim zemljevidom (Fowler & Lewis, 2014).

Slika 12: Omejeni kontekst



Prirejeno po Fowler (2014).

5.4. Postopek prehoda na mikrostoritve

Podjetja in organizacije se prehoda iz monolitne arhitekture na arhitekturo mikrostoritev lotevajo na različne načine. Frye (2020), predlaga osem osnovnih korakov, ki naj bi se jim za učinkovito migracijo sledilo pri prehodu na mikrostoritve (Frye, 2020):

- prepoznavanje logičnih komponent,
- predelava in sproščanje komponent,
- prepoznava odvisnosti komponent,
- prepoznavanje skupine komponent,
- gradnja API za oddaljeni uporabniški vmesnik,
- migracija skupin komponent v makrostoritve (ločevanje skupin komponent v ločene projekte),
- migracija makrostoritev v mikrostoritve,
- ponavljanje zadnjih dveh postopkov, dokler proces ni zaključen.

Po Fryeu (2020) naj bi se migracija začela z identifikacijo logičnih komponent. Logične komponente opisuje kot logične nabore podatkovnih objektov in dejanj, ki jih sistem izvede nad temi objekti. Razvijalci morajo na začetku identificirati, katere podatke obstoječi sistem uporablja pri izvajanju funkcij in katera dejanja so potrebna, da se podatke preoblikuje v željene izhode. Po identifikaciji vseh možnih logičnih komponent sledi predelava logičnih komponent, kar vključuje spajanje podvojenih funkcij, pregled manjkajočih vrednosti v podatkih in identifikacija outlierjev za zmanjševanje kompleksnosti sistema na novi arhitekturi (logičnih komponent in funkcij, ki se jih ne uporablja) (Frye, 2020).

Ko so logične komponente identificirane in spremenjene, je treba identificirati odvisnosti med njimi. To aktivnost razvijalci običajno izvedejo z uporabo statične analize izvorne kode, tako da se poišče klice med različnimi knjižicami in podatkovnimi tipi. S statično analizo se pregleda le izvorno kodo, ki se je ne zažene. Razvijalci pa se lahko odločijo tudi za t. i. dinamično analizo, ki se izvede z različnimi orodji, ki lahko analizirajo vzorce uporabe pri aplikaciji med njeno izvedbo (Frye, 2020).

Po identifikaciji odvisnosti Frye (2020) predlaga grupacijo komponent v skupine, ki se jih lahko transformira v makrostoritve, ki lahko delujejo kot samostojna enota. Ko so komponente združene v makrostoritve, se prične gradnja API-jev makrostoritev za uporabniški vmesnik. Uporabniški vmesnik je edini način komunikacije med sistemom in uporabniki, zato je pomembno, da je ta vmesnik prilagodljiv in dobro načrtovan, da se lahko prepreči morebitne težave pri poznejšem uvajanju sprememb. Uporabniški vmesnik naj bi omogočal dodajanje novih predmetov, atributov in dejanj, zato sta oblikovanje in implementacija glavni ključ do uspeha pri migraciji na mikrostoritve. Uporabniški vmesnik mora biti sposoben obravnavati vse primere dostopa do podatkov, ki so podprti v aplikaciji, ki bo uporabljala API. Spremembe API-ja, ki prekinjajo povratno združljivost, morajo biti redke in se jih mora načrtovati vnaprej. Pri spremembah API-jev bi uporabniški vmesnik moral omogočati enostavno dodajanje novih podatkovnih objektov in funkcij, poleg tega pa naj bi se pri spremembah ne spreminjalo oblike obstoječih izhodov ali pričakovanih vhodov, oziroma naj bi bilo tega čim manj (Frye, 2020).

Po opredelitvi API-jev sledi migracija skupin komponent oziroma makrostoritev, opredeljenih v prejšnjih točkah, v ločene projekte, za katere skrbijo razvojne skupine. Glavni razlog, zakaj se mikrostoritev ne ustvarja takoj, je njihova kompleksnost. Zaradi prepletene logike monolitov Frye (2020) priporoča razgradnjo najprej na makrostoritve, ki omogočajo bolj zapletene interakcije, in nato razdiranje na mikrostoritve. Glavni cilj tega koraka je tako premakniti skupino komponent v ločene projekte (Frye, 2020).

Po razstavitvi na ločene projekte se začne migracija makrostoritev k mikrostoritvam. Gre za proces razdelitve na manjše storitve, vsaka storitev pa opravlja samo eno funkcionalnost. Vsaka storitev komunicira s svojo bazo podatkov, zato je v tem procesu potrebno grajenje ločenih podatkovnih baz. Ko se ustvari mikrostoritve, se prične uvajanje in testiranje preoblikovane aplikacije (Frye, 2020).

5.5 Migracijske strategije

Podjetja in organizacije se odločajo za različne migracijske pristope. Migracijske strategije so odvisne od usposobljenosti razvojnih skupin, organizacijske strukture, karakteristik dosedanjega sistema itd. Fritzsch, Bogner, Wagner in Zimmermann (2019) v svoji raziskavi navajajo največkrat uporabljene migracijske strategije, ki se uporabljajo pri prehodu na mikrostoritveno arhitekturo:

- prepis obstoječe aplikacije,
- uporaba postopka postopne menjave komponent,
- vzporedno delovanje,
- popolnoma nov sistem (Greenfield),
- razširitev obstoječe aplikacije.

Podjetja ali organizacije se glede na svoje potrebe odločajo med različnimi migracijskimi strategijami. Pri prehodu lahko uporabijo kombinacijo več strategij. Kot v svojem raziskovanju ugotovijo Fritzsch, Bogner, Wagner in Zimmermann (2019), je prevladujoča migracijska strategija prepis obstoječe aplikacije skupaj z razširitvijo funkcionalnosti. Ta strategija dveh pristopov se zdi smiselna, saj je eden od glavnih vzrokov za prehod na mikrostoritve pomanjkanje funkcionalnosti pri starem sistemu, kar je posledica pomanjkanja možnosti za vzdrževanje velikega monolitnega sistema. Prav zaradi tega se podjetja in organizacije največkrat odločajo za prehod na mikrostoritve (Fritzsch, Bogner, Wagner & Zimmermann, 2019).

5.5.1 Postopek postopne menjave komponent

Migracija iz monolitne na mikrostoritveno arhitekturo je lahko zelo težaven proces. Pri migraciji je poleg spremembe razvojne kode treba razdeliti tudi podatkovno bazo tako, da ima vsaka posamezna storitev dostop do svoje podatkovne baze. Za reševanje problema z migracijo so se razvili različni načini. Eden izmed njih je postopek postopne menjave komponent, ki je rešitev za migracijski proces, ki so ga v razvojni proces uvedla tudi številna znana podjetja, kot sta Google in IBM (Brown, 2017).

Postopek postopne menjave komponent je leta 2004 razvil Martin Fowler, ki je, kot že omenjeno, tudi eden od začetnikov mikrostoritvene arhitekture. Fowler je navdih za ta postopek dobil z opazovanjem narave. Bolj natančno: inspiracijo je dobil pri opazovanju smogonske fige v deževnem gozdu. Seme te fige raste in se spušča iz gostiteljskega drevesa, okoli katerega se ovija, dokler se ne ukorenini v tleh. Skozi dolga leta zrastejo v fantastične oblike, medtem pa gostiteljevo drevo ovijajo in dušijo, dokler to ne umre. To se dogaja tudi pri uporabi postopka postopne menjave komponent, vendar z manjšimi modifikacijami. Storitve mikrostoritvene arhitekture počasi ubijajo funkcionalnosti monolitne programske opreme tako, da se funkcionalnosti na monolitni arhitekturi vedno bolj zmanjšujejo, nadomešča pa se jih z mikrostoriveno programsko opremo (Brown, 2017).

Postopek postopne menjave komponent je pristop, ki namesto ene velike posodobitve na mikrostoritveno arhitekturo omogoča postopen razvoj migracijskega procesa. Brown (2017) prikazuje njegovo prednost v primerjavi z migracijo velikega poka predvsem v smislu hitrejše migracije. Pri migraciji velikega poka je treba celotno razvojno kodo najprej posodobiti, preden se prične uporabljati mikrostoritveno arhitekturo. Pri postopni menjavi komponent pa se funkcionalnosti postopoma premešča na mikrostoritveno arhitekturo, tako da določene

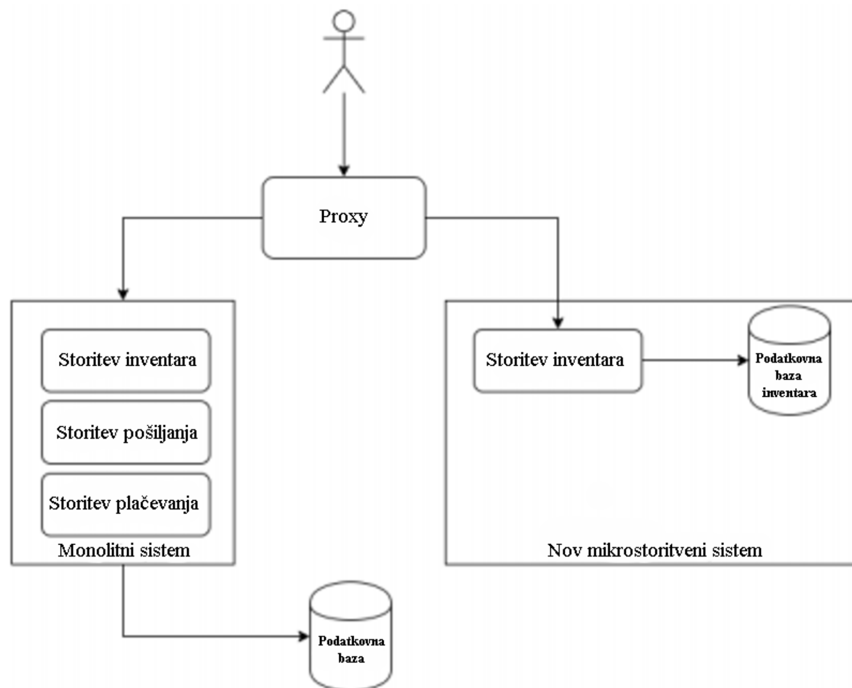
funkcionalnosti delujejo na monolitni, druge pa že na mikrostoritveni arhitekturi. Še ena pomembna prednost je v tem, da lahko celotna programska oprema deluje, ne da bi uporabniki opazili spremembe, v kolikor je proces pravilno izpeljan. Poleg tega pa ta postopek omogoča vzporedno razvijanje novih funkcionalnosti na mikrostoritveni arhitekturi, če je to potrebno (Brown, 2018).

Pri odločitvi za to vrsto prehoda na mikrostoritveno arhitekturo Brown (2017) opozarja na nekaj pomanjkljivosti. Ker je migracijski proces zahteven, so razvijalci velikokrat v skušnjavi in prenesejo celotno razvojno kodo na mikrostoritveno arhitekturo, ne da bi kopirali tudi bazo podatkov za posamezno storitev, kar pa v prihodnosti lahko povzroča neskladnost podatkov pri delovanju programske opreme. Poleg tega veliko podjetij in organizacij po izboru postopne menjave komponent za migracijski proces prične z uvajanjem številnih novih funkcionalnosti v obstoječo programsko opremo, kar lahko podaljša časovni okvir migracije, istočasno pa se lahko zmanjša motivacija razvijalcev in ostalih zaposlenih za dokončanje migracijskega procesa (Brown, 2017).

Podjetja in organizacije pri uporabi omenjenega postopka največkrat pričnejo z razdelitvijo monolitne programske opreme na domene, ki predstavljajo posamezno storitev z omejenim kontekstom. V naslednjem koraku lahko razvijalci v monolitni programski opremi uporabijo obstoječi izravnalnik obremenitve ali pa ustvarijo proxy, ki služi za preusmerjanje prometa. Ta korak je pomemben, saj se z njim ustvari most za prenos monolitnih funkcij na mikrostoritveno arhitekturo. Z vsako novo izdelano in nameščeno mikrostoritvijo proxy preusmeri promet iz obstoječe monolitne funkcije na mikrostoritev. Ko so vse storitve/funkcije monolitnega sistema premeščene na mikrostoritveno arhitekturo, je migracijski proces zaključen (Brown, 2017).

Na sliki 13 je prikazan migracijski proces z uporabo postopka postopne menjave komponent. Na levi strani slike je prikazan monolitni del programske opreme, na desni strani pa mikrostoritveni del programske opreme. Na začetku migracijskega procesa deluje celotna programska oprema na monolitni arhitekturi. Skozi čas razvijalci preoblikujejo posamezne elemente monolitne arhitekture v storitve mikrostoritvene arhitekture. Te storitve so prikazane na desni strani slike. Na sredini zgornjega dela slike je prikazan proxy, v določenih primerih gre tudi za izravnalnik obremenitva. Izravnalnik obremenitve skrbi za preusmerjanje prometa/zahtev bodisi k mikrostoritveni programski opremi, bodisi k monolitni, odvisno od tega kje je zahtevana funkcionalnost razvita. Elementov iz levega dela slike (monolitne programske opreme) je skozi čas vse manj, saj razvijalci te elemente postopoma razvijajo na mikrostoritveni arhitekturi. Ko je storitev razvita na mikrostoritveni arhitekturi, ni več potrebe po delovanju tega elementa na monolitni arhitekturi. Pri prehodu elementa na mikrostoritveno arhitekturo, le ta največkrat dobi lastno podatkovno bazo, do katere ima samo ona dostop. Proces migracije je končan, kadar na levi strani (monolitni arhitekturi), ni več nobenega elementa, kar pomeni, da so vsi elementi uspešno premeščeni na mikrostoritveno programsko arhitekturo. (Brown, 2017)

Slika 13: Primer postopka postopne menjave komponent



Prirejeno po Brown (2017).

5.6 Primer uspešnega prehoda na arhitekturo mikrostoritev

Ko se govori o uspešnih prehodih na mikrostoritve, je zagotovo vredno omeniti prehod iz monolitne na mikrostoritveno arhitekturo pri ameriškem podjetju za pretočne storitve Netflix. Netflix je trenutno eden največjih ponudnikov spletnih medijskih vsebin na svetu. Podjetje je uspešno migriralo s tradicionalne monolitne arhitekture na oblačno mikrostoritveno arhitekturo. Prehod je bil dolgoročen proces in je v celoti trajal dve leti (Rud, 2019).

Po besedah oblačnega arhitekta Netflixa se je migracijski proces pričel že leta 2009. Ko je Netflix razglasil prehod na mikrostoritveno arhitekturo, je bilo podjetje deležno številnih kritik, saj skoraj nihče ni verjel, da je taka migracija možna. Glavni razlog za prehod v oblak in mikrostoritveno arhitekturo je bilo drastično povečevanje podatkov in uporabniških informacij, ki jih je bilo težko shranjevati v takratnih podatkovnih centrih, kar je povzročalo številne probleme, ki jih je bilo treba sproti reševati. Rešitev je Netflix dosegel s pomočjo Amazonovih spletnih storitev (angl. Amazon Web Service), ki zagotavljajo veliko virov s podatkovnimi centri, ki so na voljo, kadar jih podjetje potrebuje, in ki zagotavljajo varnost (Rud, 2019).

Med selitvijo v oblak je podjetju Netflix svojo monolitno aplikacijo uspelo razdeliti na stotine majhnih ohlapno povezanih storitev. Za primer: danes Netflix razpolaga z več kot tisoč različnimi mikrostoritvami, ki opravljajo vsaka svojo nalogo. Postopek migracije se je začel pri funkcionalnostih, ki se neposredno ne nanašajo na stranke podjetja. Bolj natančno: proces migracije so pričeli s funkcionalnostjo kodiranja filmov. Leta 2010 pa je podjetje pričelo z

migracijo še preostalih delov programske opreme v oblak: na primer registracija računa, izbira filmov in ostalih storitev. Decembra leta 2011 je podjetje svoje celotno poslovanje uspešno migriralo v oblak in na mikrostoritveno arhitekturo (Rud, 2019).

Podjetje se je med celotnim procesom migracije soočalo z vrsto težav. Treba je bilo vzdrževati in poganjati oblačne strežnike kot tudi lokalne strežnike v podjetju, da se je zagotovilo nemoteno delovanje. Migracija v oblak je pomenila tudi replikacijo podatkov iz lokalnih podatkovnih centrov v oblačne podatkovne centre, pri čemer se je podjetje srečevalo z ogromno količino podatkov. Poleg omenjenih zapletov so se pri podjetju srečevali tako z zakasnitvijo povezav (latenco) kot tudi s povečanjem obremenitve, napakami storitev in drugimi težavami, povezanimi z zmogljivostjo. Kljub mnogim težavam je podjetje uspešno prestalo celoten proces migracije. Migracija k mikrostoritveni arhitekturi je pripomogla, da lahko podjetje neodvisno razširja posamezne storitve, namesto da razširja celotno monolitno aplikacijo, kar je pripomoglo k znižanju stroškov, saj podjetje plačuje samo za stroške virov, ki jih dejansko potrebuje, in ne za tiste, ki jih ne. Poleg razširjanja so pri podjetju opazili tudi hitrejše delovanje in večjo agilnost razvoja, saj so inženirji podjetja pridobili možnost neodvisnega razvoja, testiranja in nameščanja storitev. To je dovoljevalo oblikovanje več kot tridesetih različnih skupin, ki lahko delujejo na različnih delih sistema oziroma storitvah brez potrebe po čakanju ostalih, da dokončajo nalogo na ostalih storitvah, kar je pripomoglo k hitrejšemu razvoju (Rud, 2019).

6 EMPIRIČNA RAZISKAVA STANJA PROGRAMSKE ARHITEKTURE V SLOVENSKIH PODJETJIH

Empirični del magistrskega dela je kvantitativna raziskava stanja programske arhitekture v slovenskih podjetjih. Kot znanstveno metodo raziskovanja sem izvedel spletno anketo. Anketni vprašalnik služi kot pomoč pri preverjanju teoretičnih izsledkov dela. Cilj takih raziskav je analiza različnih spremenljivk, kar omogoča posploševanje na širšo populacijo. To je smiselno pri pripravi empirične raziskave tega magistrskega dela, saj sem lahko na tak način preveril točnost teoretično opredeljenih lastnosti uporabe mikrostoritvene in monolitne arhitekture pri podjetjih in organizacijah in poizkusil razumeti, kateri so dejavniki, ki pripomorejo k odločitvi za prehod na mikrostoritveno arhitekturo.

Namen empirične raziskave je raziskati stanje uporabe programske arhitekture slovenskih podjetij. Slovenskim podjetjem želim podati vpogled v ugotovitve raziskave, ki jim bodo lahko pomagale pri njihovem odločanju o uporabi novega koncepta programske arhitekture, imenovanega mikrostoritvena programska arhitektura. Spoznanja raziskave nimajo koristi samo za podjetja, ki razmišljajo o uporabi te arhitekture, pač pa tudi za vsa ostala podjetja in organizacije, ki koncepta mikrostoritvene programske arhitekture še ne poznajo.

Raziskavo o uporabi programske arhitekture sem izvedel jeseni 2021. Raziskavo sem izvedel s pomočjo spletnega orodja 1ka, ki omogoča podporo pri vseh korakih spletnega anketiranja. V

sodelovanje pri izpolnjevanju ankete sem povabil 3082 slovenskih podjetij. Pri izboru podjetij sem upošteval merilo, da mora podjetje delovati v Sloveniji in imeti najmanj 20 zaposlenih, razen v primeru, ko je glavna dejavnost, s katero se podjetje ukvarja, informacijska tehnologija. Za iskanje podjetij in njihovih elektronskih poštnih naslovov sem uporabil poslovni asistent bizi.si, ki nudi finančne in kontaktne podatke za slovenska podjetja. Anketa je bila odprta dva meseca, v tem obdobju pa sem pridobil vzorec 376 podjetij. Anketo je popolnoma rešilo 227 podjetij/anketirancev, kar predstavlja 7,36-odstotni odziv.

6.1 Raziskovalna vprašanja

V magistrskem delu sem si zastavil raziskovalna vprašanja, na katera sem želel pridobiti odgovore. Raziskovalna vprašanja so predstavljena v tabeli 1. Raziskovalna vprašanja od št. 6 do vključno št. 10 so osnova za preverjanje hipotez v nadaljevanju analize pridobljenih podatkov, ki jih podrobneje opišem v poglavju 6.4, Testiranje in analiza hipotez.

Tabela 1: Raziskovalna vprašanja magistrskega dela

RV 1	Katere slabosti občutijo podjetja, ki uporabljajo monolitno programsko arhitekturo?
RV 2	Kateri so glavni razlogi za uporabo mikrostoritvene arhitekture pri podjetjih, ki uporabljajo mikrostoritve od začetka?
RV 3	Kateri dejavniki vplivajo na izbor mikrostoritvene arhitekture?
RV 4	S katerimi težavami se srečujejo podjetja pri razvoju mikrostoritvene arhitekture? Ali se te težave razlikujejo glede na to, ali podjetje gradi mikrostoritve od začetka ali pa se je izvedel prehod iz monolitne arhitekture?
RV 5	Kateri so glavni programski jeziki, uporabljeni pri grajenju programske opreme, glede na programsko arhitekturo? Ali se razlikujejo glede na obliko arhitekture?
RV 6	Katere so značilnosti mikrostoritvene programske arhitekture, ki prevladujejo v primerjavi z monolitno programsko arhitekturo?
RV 7	Katera programska arhitektura prevladuje pri slovenskih podjetjih?
RV 8	Ali podjetja, ki uporabljajo mikrostoritveno programsko arhitekturo, pogosteje posegajo po ponudnikih oblačnih storitev za nameščanje programske opreme v oblak?
RV 9	Ali so podjetja, ki uporabljajo mikrostoritveno arhitekturo od začetka, bolj naklonjena zabojniški tehnologiji, kot podjetja, ki so izvedla prehod na mikrostoritveno arhitekturo?
RV 10	Katera podjetja glede na izbor programske arhitekture uporabljajo več različnih programskih jezikov?

Vir: lastno delo.

6.2 Zasnova vprašalnika

Anketni vprašalnik je oblikovan na podlagi preučevane literature. V vprašalniku sem zajel glavne teme, ki sem jih obravnaval v predhodnih poglavjih magistrskega dela. Anketni vprašalnik je v grobem sestavljen iz dveh delov. V prvem delu so zajeta osnovna vprašanja o respondentih in podjetjih, v katerih so zaposleni. Drugi del zajema vprašanja o arhitekturi programske opreme. Ta del anketnega vprašalnika je sestavljen iz treh različnih poddelov. Vsak izmed anketirancev izpolni samo en poddel – odvisno od tega, katero vrsto programske arhitekture ima podjetje, v katerem deluje.

Glavni poddeli drugega dela anketnega vprašalnika so:

- podjetje je programsko opremo na mikrostoritveni arhitekturi oblikovalo od začetka,
- podjetje je izvedlo prehod na mikrostoritveno programsko arhitekturo,
- podjetje nima mikrostoritvene programske arhitekture

Določena vprašanja so v vseh poddelih enaka, tako da sem lahko naredil primerjalno analizo glede na različno vrsto arhitekture programske opreme, določena vprašanja pa zadevajo samo določen tip arhitekture, saj jih zaradi specifičnosti arhitekturnih oblik nisem mogel aplicirati na ostale. Predviden čas reševanja anketnega vprašalnika je 10 minut. Vprašalnik je v večini sestavljen iz vprašanj zaprtega tipa, nekaj pa jih je odprtega tipa in ta zahtevajo kratek odgovor. Podatke, ki sem jih pridobil z anketnim vprašalnikom, sem podrobneje analiziral v programu Microsoft Excel z deskriptivno statistiko. Zbrane podatke sem prikazal s pomočjo grafičnih in tabelaričnih prikazov.

6.3 Rezultati in ugotovitve raziskave

V tem poglavju bom predstavil zbrane rezultate anketnega vprašalnika. Rezultati so predstavljeni z opisno statistično analizo. V naslednjem poglavju se bom dotaknil tudi testiranja hipotez, s katerimi bom poizkušal dobiti odgovore na zastavljena raziskovalna vprašanja.

Na anketni vprašalnik je skupaj odgovorilo 376 respondentov, od tega jih je 227 v celoti izpolnilo anketo, preostalih 149 respondentov pa je anketo končalo delno. Skupno je bilo poslanih 3082 vabil k reševanju spletne ankete, torej je pri v celoti izpolnjenih anketah prišlo do 7,36%-odziva. Razlog za manjši odziv je moč pripisati temu, da veliko manjših podjetij nima svojega internega oddelka za informacijsko tehnologijo, ampak zunanje izvajalce. Poleg tega je kar nekaj elektronskih naslovov na spletnem portalu bizi.si neveljavnih, kar je v mojem primeru obsegalo približno 250 elektronskih naslovov. Da bi dobil celotno sliko stanja programske arhitekture v Sloveniji, je bil anketni vprašalnik poslan podjetjem v različnih regijah Slovenije. Na anketni vprašalnik so respondenti lahko odgovarjali od 3. 7. 2021 do vključno 4. 9. 2021.

V tabeli 2 so prikazane vloge anketirancev v podjetju. V raziskavi je glede na rezultate ankete sodelovalo največ vodij projektov (92, oz. 25,77 odstotkov vseh anketirancev), sledijo programski razvijalci in glavni programski razvijalci, ki predstavljajo skupaj 33,05 odstotka vseh anketirancev. Sladijo še programski arhitekti in direktorji (oziroma katerekoli vodstvene vloge, npr. direktor podjetja, logistike, tehnologije itd.). Na koncu tabele so združeni vsi ostali odgovori, ki niso sodili v nobeno izmed glavnih skupin vlog v podjetju.

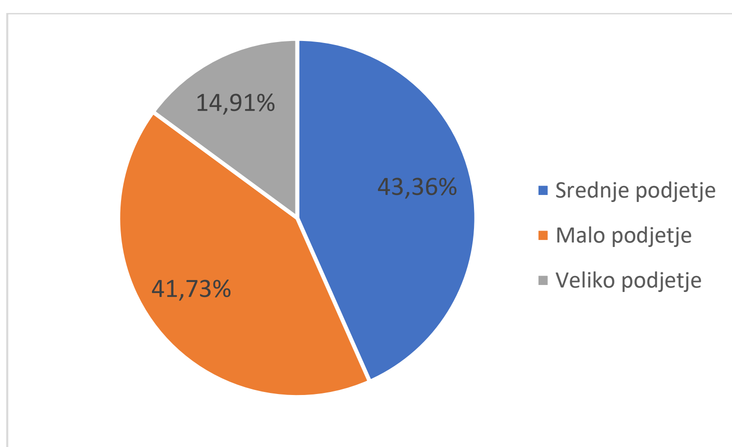
Tabela 2: Vloga v podjetju

Vloga v podjetju	**Frekvenca**	**Relativna frekvenca**
Vodja projektov	92	25,77 %
Programski razvijalec	78	21,85 %
Glavni programski razvijalec	40	11,20 %
Programski arhitekt	39	10,92 %
Direktor	19	5,32 %
Sistemski administrator	16	4,48 %
Načrtovalec aplikacij	15	4,20 %
IT-podpora	15	3,64 %
Ostalo	45	12,62 %
Skupaj	**357**	**100,00 %**

Vir: lastno delo.

Na anketno vprašanje v kako velikem podjetju delujejo anketiranci, so bili na voljo trije različni odgovori (malo, srednje in veliko podjetje). Glede na rezultate je največ anketirancev zaposlenih v srednje velikih podjetjih (43,36 % vseh anketirancev), sledijo zaposleni v malih podjetjih, kjer jih deluje 41,73 % vseh anketirancev. V velikih podjetjih deluje vsega le 14,91 % vseh anketirancev. Rezultati so prikazani na sliki 14.

Slika 14: Anketiranci glede na velikost podjetja, v katerem delujejo



Vir: lastno delo.

Glede na anketno vprašanje, koliko let profesionalnih izkušenj imajo anketiranci na področju informacijske tehnologije, prevladujejo anketiranci z več kot deset let profesionalnih izkušenj, kar sovpada z odgovori na anketno vprašanje o vlogi anketirancev v podjetju, saj jih največ deluje v vlogi vodij projektov, za kar potrebujejo izkušnje. Sledijo jim anketiranci z od 4 do vključno 9 let profesionalnih izkušenj (22,4 % vseh anketirancev) in od 1 do vključno 3 let profesionalnih izkušenj (9,84 % vseh anketirancev). Najmanjši delež (4,64 % vseh anketirancev predstavljajo anketiranci z manj kot enim letom profesionalnih izkušenj. Podatki so prikazani v tabeli 3.

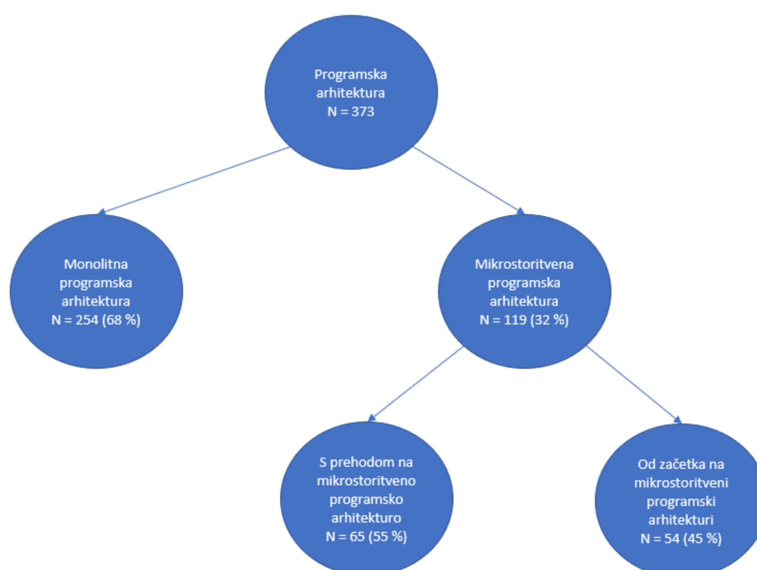
Tabela 3: Število profesionalnih let izkušenj na IT-področju

Število let izkušenj	**Frekvenca**	**Relativna frekvenca**
Do enega leta	17	4,64 %
Od 1 do vključno 3 let	36	9,84 %
Od 4 do vključno 9 let	82	22,40 %
Več kot 10 let	231	63,12 %
Skupaj	**366**	**100,00 %**

Vir: lastno delo.

Na anketno vprašanje, ali imajo anketiranci v podjetju, v katerem delujejo, programsko opremo, ki je bila razvita s pomočjo mikrostoritvene programske arhitekture, je 119 anketirancev odgovorilo pritrdilno, preostalih 254 anketirancev pa v podjetju, v katerem delujejo, nima programske opreme, zgrajene na mikrostoritveni arhitekturi, kar pomeni, da večina podjetij v Sloveniji uporablja monolitno arhitekturo, kar prikazuje tudi slika 15.

Slika 15: Prikaz po oblikah in načinu razvoja programske arhitekture

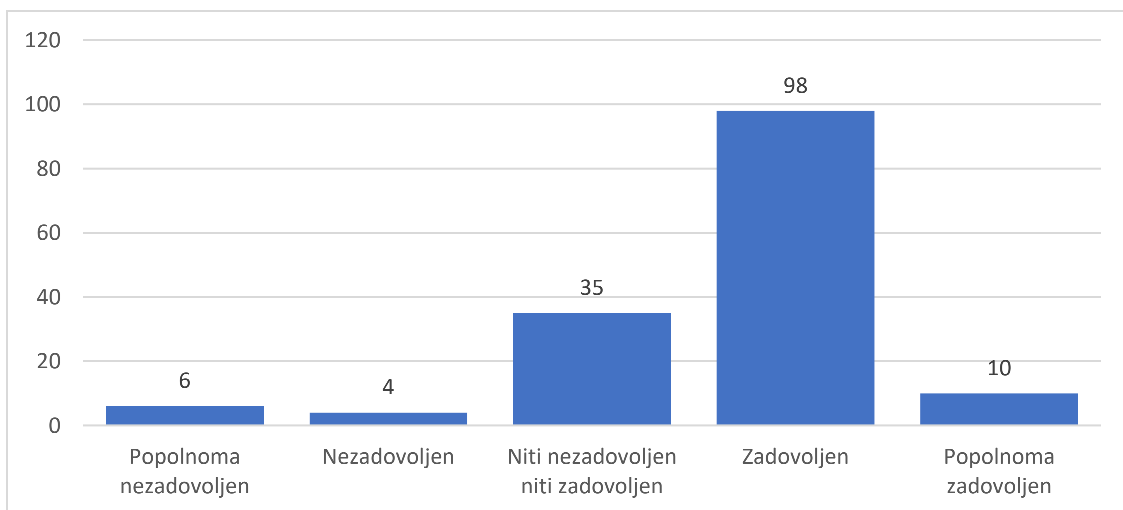


Vir: lastno delo.

Od vseh anketirancev, ki imajo v podjetju razvito mikrostoritveno programsko upremo, jih je 54 odgovorilo, da je bila programska oprema že od samega začetka razvita na tej arhitekturi. Preostalih 65 anketirancev pa je odgovorilo, da so se v podjetju odločili za prehod iz obstoječe programske arhitekture na mikrostoritveno.

Na vprašanje, kako so anketiranci, ki v podjetju nimajo razvite programske opreme na mikrostoritveni arhitekturi, zadovoljni z delovanjem trenutne programske opreme, je skupno odgovorilo 153 anketirancev. Anketno vprašanje je bilo postavljeno v obliki 5-stopenjske Likertove lestvice za merjenje stališč. Večina anketirancev je zadovoljstvo z obstoječo programsko opremo ocenilo s 4 – zadovoljni (skupaj 98 anketirancev oz. 64 % vseh anketirancev, ki v podjetju še niso uporabljali mikrostoritvene arhitekture). Sledijo jim anketiranci, ki so zadovoljstvo obstoječe programske opreme ocenili z niti nezadovoljen niti zadovoljen (35 anketirancev). V povprečju so anketiranci ocenili zadovoljstvo z obstoječo programsko opremo oceno 3,7, kar nakazuje, da so anketiranci, ki uporabljajo monolitno arhitekturo, z njo relativno zadovoljni. Zbrani odgovori so prikazani na sliki 16.

Slika 16: Zadovoljstvo z obstoječo programsko opremo (monolitna arhitektura)



Vir: lastno delo.

Kljub dobri povprečni oceni pri vprašanju o zadovoljstvu s programsko opremo anketirancev, ki v podjetju nimajo razvite programske opreme na mikrostoritveni arhitekturi, so ti našteli tudi nekaj slabosti. Anketiranci so pri vprašanju o slabostih obstoječe programske opreme izbirali med različnimi vnaprej pripravljenimi odgovori. Izbrali so lahko eno ali več slabosti, v kolikor pa katera od slabosti, ki so jo želeli izpostaviti, ni bila zapisana, so jo lahko tudi dodatno zapisali. Anketiranci so kot največjo slabost obstoječe programske opreme v podjetju označili počasen razvoj novih funkcionalnosti, s čimer se je strinjalo 76 anketirancev oziroma 50 % vseh anketirancev, ki v podjetju ne uporabljajo mikrostoritvene arhitekture. Počasen razvoj novih funkcionalnosti sovpada z navedbami v teoretičnem delu magistrskega dela, kjer različni avtorji opisujejo visoko kompleksnost monolitnih sistemov zaradi vse večje medsebojne odvisnosti komponent, poleg tega pa je vsa programska koda zapisana kot celota, kar pomeni zahtevno

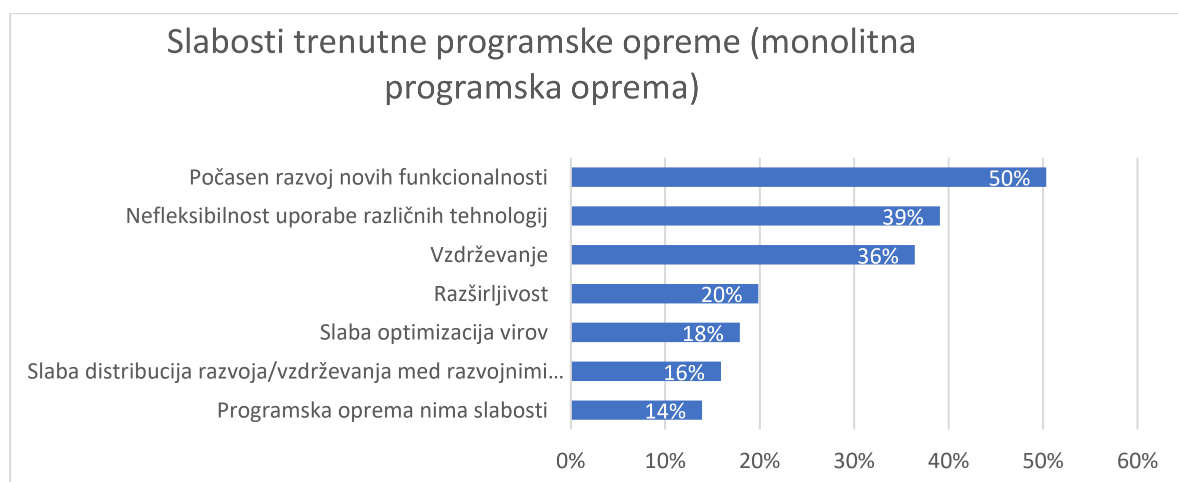
urejanje in posodabljanje programske kode za namen dodajanja novih funkcionalnosti. Kot drugo največjo slabost anketiranci ocenjujejo pomanjkanje fleksibilnosti pri uporabi različnih tehnologij. Pomanjkanje fleksibilnosti obstoječe programske opreme je skupaj izbralo 59 anketirancev, kar predstavlja 39 % vseh anketirancev. Tudi ta slabost sovpada z teoretičnim delom, kjer nekateri avtorji opisujejo hibo monolitnih sistemov, ki ne dovoljujejo dodajanja novih tehnologij, oziroma je dodajanje oteženo in zahteva dodatne napore. Naslednja slabost obstoječe programske opreme po mnenju anketirancev je njeno vzdrževanje. Za to slabost se je opredelilo 36 % vseh anketirancev. Sledi še oteženo razširjanje programske opreme, s čimer se je strinjalo 20 % vseh anketirancev, slaba optimizacija virov z 18 % in slaba distribucija razvoja med razvojnimi skupinami s 16 %. Med različnimi možnostmi izbora so se anketiranci lahko odločili tudi za izjavo, da trenutna programska oprema v podjetju nima slabosti. S to trditvijo se je strinjalo 21 anketirancev, oziroma 14 % vseh anketirancev. Odgovori so prikazani na sliki 17.

Poleg vnaprej pripravljenega nabora slabosti obstoječe programske opreme so anketiranci navedli še:

- slab pretok informacij,
- povečevanje stroškov vzdrževanja,
- nefleksibilnost,
- velika poraba časa za vzdrževanje, zaradi česar preostane manj časa za razvoj novih aplikacij in funkcionalnosti.

Ker me je pri prvem raziskovalnem vprašanju zanimalo, katere so glavne slabosti monolitne arhitekture, sem s tem anketnim vprašanjem ugotovil, da med največje tri slabosti pri slovenskih podjetjih sodijo počasen razvoj novih funkcionalnosti (50 %), nefleksibilnost uporabe različnih tehnologij (39 %) in vzdrževanje (36 %).

Slika 17: Slabosti monolitne programske opreme



Vir: lastno delo.

Ker slabosti obstoječe programske opreme močno vplivajo na poslovanje in delo podjetij, me je zanimalo še, kako na poslovanje vplivajo slabosti, ki so jih navedli anketiranci, ki v podjetju ne uporabljajo mikrostoritvene arhitekture. Anketiranci so med najpogostejšimi vplivi na poslovanje, ki jih povzročijo slabosti trenutne programske opreme, izpostavili:

- daljše roke za implementacijo potrebnih prilagoditev programske opreme glede na potrebe notranjih delovnih procesov podjetja, kar vodi do motenega in upočasnjenega dela preostalih oddelkov v podjetju,
- upočasnjeno delovanje trenutnih sistemov,
- slabo klimo v podjetju.

Zaradi različnih negativnih vplivov monolitne programske opreme me je zanimalo tudi, ali so anketiranci, ki v podjetju ne uporabljajo mikrostoritvene arhitekture, že kdaj razmišljali o prehodu obstoječe arhitekture na mikrostoritveno arhitekturo. Anketiranci so na vprašanje o prehodu odgovarjali z vnaprej pripravljenimi odgovori, pri čemer so lahko izbrali enega izmed štirih možnih odgovorov. Odgovore na anketno vprašanje sem še dodatno razdelil glede na število profesionalnih let delovnih izkušenj anketiranca. Skupno je 75 (51 %) vseh anketirancev, ki ne uporabljajo mikrostoritvene programske arhitekture, odgovorilo, da trenutna programska arhitektura zadostuje njihovim potrebam. Delež anketirancev, ki menijo, da trenutna programska arhitektura zadostuje njihovim potrebam, je največji v skupini anketirancev z več kot 10 leti profesionalnih izkušenj na področju informacijske tehnologije (57 % vseh anketirancev iz te skupine). 30 anketirancev (20 %) iz vseh starostnih skupin je na vprašanje o prehodu odgovorilo, da so za to arhitekturo slišali prvič. Največji delež anketirancev (50 %), ki so za mikrostoritveno programsko arhitekturo slišali prvič, izhaja iz skupine anketirancev, ki imajo do enega leta profesionalnih delovnih izkušenj na področju informacijske tehnologije, sledi skupina od enega do vključno treh let profesionalnih izkušenj (30 %), delež pri ostalih dveh skupinah pa je znatno nižji. 24 vseh anketirancev iz vseh starostnih skupin (16 %) je odgovorilo, da v podjetju v prihodnosti načrtujejo uporabo mikrostoritvene arhitekture, preostalih 19 (13 % vseh anketirancev) pa je odgovorilo, da so razmišljali o njeni uporabi, vendar v podjetju nimajo dovolj znanja. Zanimivo je, da je najpogostejši odgovor v skupini anketirancev z več kot 10 leti profesionalnih izkušenj ta, da trenutna programska arhitektura zadostuje potrebam podjetja. Razlog za tako visok delež bi lahko pripisali nenaklonjenosti starejših zaposlenih do novih tehnologij, saj je za to treba vložiti dodatne napore za pridobivanje dodatnih izkušenj. Odgovori anketirancev so prikazani v tabeli 4.

Tabela 4: Razmišljanje o uporabi mikrostoritvene arhitekture glede na število let profesionalnih izkušenj

1. nivo: Število let profesionalnih izkušenj 2. nivo: Ali ste v podjetju razmišljali o uporabi mikrostoritvene arhitekture?	Število odgovorov	% odgovorov
Do enega leta	**6**	**4 %**
Za mikrostoritveno arhitekturo smo slišali prvič	3	50 %
Razmišljali smo in načrtujemo uporabo te arhitekture	1	17 %
Razmišljali smo, vendar nimamo dovolj znanja	2	33 %
Od 1 do vključno 3 let	**10**	**7 %**
Nismo razmišljali – trenutna arhitektura zadostuje potrebam	4	40 %
Za mikrostoritveno arhitekturo smo slišali prvič	3	30 %
Razmišljali smo in načrtujemo uporabo te arhitekture	1	10 %
Razmišljali smo, vendar nimamo dovolj znanja	2	20 %
Od 4 do vključno 9 let	**32**	**22 %**
Nismo razmišljali – trenutna arhitektura zadostuje potrebam	14	44 %
Za mikrostoritveno arhitekturo smo slišali prvič	4	13 %
Razmišljali smo in načrtujemo uporabo te arhitekture	10	31 %
Razmišljali smo, vendar nimamo dovolj znanja	4	13 %
Več kot 10 let	**99**	**67 %**
Nismo razmišljali – trenutna arhitektura zadostuje potrebam	56	57 %
Za mikrostoritveno arhitekturo smo slišali prvič	20	20 %
Razmišljali smo in načrtujemo uporabo te arhitekture	12	12 %
Razmišljali smo, vendar nimamo dovolj znanja	11	11 %
Skupaj	**147**	**100 %**

Vir: lastno delo.

Pri anketirancih, ki uporabljajo mikrostoritveno programsko arhitekuro od začetka (brez prehoda), me je zanimalo, kateri so glavni razlogi za njeno uporabo. Pri vprašanju so anketiranci lahko izbrali med različnimi pripravljenimi odgovori, če pa željenega odgovora ni bilo na seznamu, so lahko zapisali svojega. Anketiranci so lahko izbrali več različnih odgovorov. Na vprašanje je skupaj odgovorilo 24 anketirancev. Kot glavni vzrok uporabe mikrostoritvene arhitekture so anketiranci izbrali hiter razvoj posameznih funkcionalnosti; ta odgovor je izbralo kar 21 anketirancev, ki uporabljajo mikrostoritveno arhitekturo od začetka (88 %). Kot so navedli tudi številni avtorji, je eden od glavnih razlogov, zakaj se je razvila mikrostoritvena arhitektura, ohlapna povezanost storitev med seboj in samostojnost samih storitev, kar omogoča hiter razvoj novih funkcionalnosti in tudi njihovo testiranje. Kot drugi najpogostejši razlog za uporabo mikrostoritvene programske arhitekture so anketiranci izbrali razširljivost, za katero se je odločilo 16 anketirancev oz. 67 % vseh anketirancev, ki uporabljajo mikrostoritveno arhitekturo od začetka. Sledita še možnost uporabe več različnih programskih jezikov, ki jo je izbralo 11 anketirancev, in enostavno testiranje, ki ga je izbralo 7 anketirancev. Na zadnjem mestu razlogov za uporabo mikrostoritvene arhitekture je odpornost na napake. S tem anketnim vprašanjem sem poizkušal dobiti odgovor na drugo raziskovalno vprašanje, kjer me je zanimalo, kateri so glavni razlogi za uporabo mikrostoritvene programske arhitekture pri podjetjih, ki to arhitekturo uporabljajo od začetka. Ugotovil sem, da izstopata dva razloga, ki

sta bila izbrana pri več kot polovici anketirancev: hiter razvoj posameznih funkcionalnosti in zmožnost razširjanja programske opreme. Rezultati so prikazani na sliki 18.

Slika 18: Razlogi za uporabo mikrostoritvene arhitekture

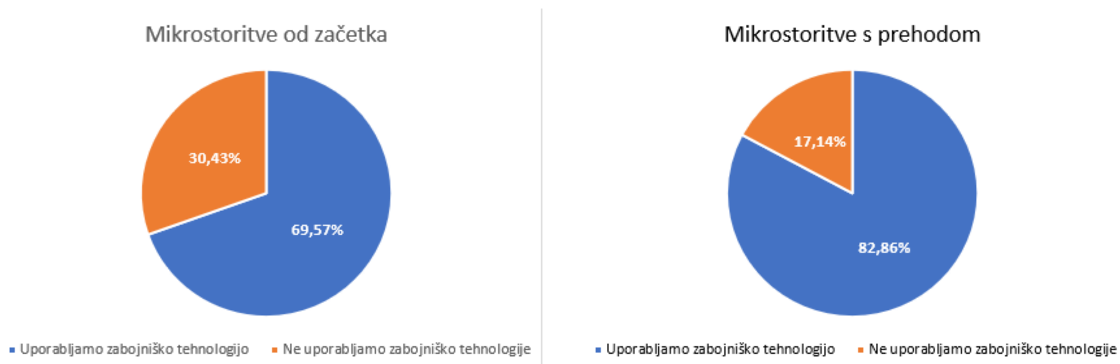
Vir: lastno delo.

V teoretičnem delu magistrskega dela sem navedel različna ogrodja za gradnjo mikrostoritvene programske opreme, ki jih navajajo avtorji. Ker je na voljo širok nabor ogrodij, me je zanimalo, katera ogrodja za grajenje mikrostoritev prevladujejo pri slovenskih podjetjih, ki uporabljajo mikrostoritveno arhitekturo. Anketiranci so imeli na voljo vnaprej pripravljene odgovore, lahko pa so zapisali tudi svoje. Na anketno vprašanje je odgovorilo 47 anketirancev, ki uporabljajo mikrostoritveno arhitekturo (od začetka in s prehodom). Ugotovil sem, da slovenska podjetja uporabljajo veliko različnih ogrodij, saj jih je bilo naštetih več kot 20. Podjetja, v katerih delujejo anketiranci, najpogosteje uporabljajo ogrodje SpringBoot, ki ga je izbralo skupno 14 od 47 anketirancev (30 % vseh anketirancev, ki uporabljajo mikrostoritveno arhitekturo). Ta ugotovitev sovpada z navedbami Kurmi (2020), da je to trenutno prevladujoče ogrodje za grajenje mikrostoritev, ki temelji na programskem jeziku Java. Med drugimi priljubljenimi ogrodji za grajenje mikrostoritvene arhitekture sta še Django in Eclipse Vert.X.

V teoretičnem delu sem ugotovil, da avtorji pri uporabi mikrostoritvene arhitekture opažajo trend uporabe zabojniške tehnologije, zaradi česar me je zanimalo, ali je ta trend prisoten tudi pri slovenskih podjetjih. Da bi to ugotovil, sem postavil anketno vprašanje, pri katerem sem anketirance, ki uporabljajo mikrostoritveno arhitekturo, vprašal o uporabi zabojniške tehnologije. Odgovore sem razdelil v dve skupini: na anketirance, ki v podjetju uporabljajo mikrostoritveno arhitekturo od začetka, in na anketirance, ki so izvedli prehod na to arhitekturo. Na vprašanje je skupno odgovorilo 58 anketirancev, od katerih jih 23 uporablja mikrostoritveno arhitekturo od začetka, preostalih 35 pa je izvedlo prehod nanjo. Ugotovil sem, da zabojniško tehnologijo uporablja 78 % vseh anketirancev. Delež je nekoliko večji pri anketirancih, ki

uporabljajo mikrostoritveno programsko opremo s prehodom, saj zabojniško tehnologijo uporablja 29 anketirancev (82,86 %). Pri podjetjih, ki uporabljajo mikrostoritve od začetka, ta delež znaša 69,57 % (16 anketirancev), kar je približno 13 odstotnih točk manj od podjetij, ki so izvedla prehod. Odgovori so prikazani na sliki 19.

Slika 19: Prikaz uporabe zabojniške tehnologije glede na pristop k grajenju mikrostoritvene programske opreme



Vir: lastno delo.

Anketirancem, ki v podjetju uporabljajo mikrostoritveno arhitekturo, sem postavil tudi vprašanje o tem, koliko mikrostoritev je bilo ustvarjenih pri gradnji celotne programske opreme. Anketiranci so imeli možnost izbirati med vnaprej pripravljenimi odgovori. Odgovore sem razdelil v dve skupini: na anketirance, ki mikrostoritve uporabljajo od začetka, in na tiste, ki so izvedli prehod na to arhitekturo. Ugotovil sem, da sta si skupini po številu ustvarjenih mikrostoritev zelo podobni.

Na anketno vprašanje je skupno odgovorilo 58 anketirancev, od tega jih 24 deluje v podjetju, kjer mikrostoritveno programsko opremo uporabljajo od začetka, 34 pa jih je izvedlo prehod na nanjo. Največ (24) anketirancev je odgovorilo, da so v podjetju pri gradnji mikrostoritvene programske opreme ustvarili do 10 mikrostoritev, kar predstavlja skupno 41 % vseh anketirancev. Pri grajenju mikrostoritev od začetka je bilo takih odgovorov 10 od 24 oziroma skupaj 42 % anketirancev, pri mikrostoritvah s prehodom pa 14 od 34 oziroma 41 % anketirancev. Naslednji najpogostejši odgovor anketirancev je bil, da so pri gradnji uporabili od 11 do vključno 50 mikrostoritev, Ta odgovor je skupno izbralo 23 anketirancev oziroma 40 % vseh anketirancev. Veliko manjši delež anketirancev pa deluje v podjetju, kjer so pri gradnji mikrostoritvene programske opreme zgradili med 51 do vključno 100 mikrostoritev, in takih, ki so jih zgradili 100 ali več. Skupno deluje v podjetjih, kjer so zgradili več kot 50 mikrostoritev, 11 anketirancev oziroma 19 % vseh anketirancev. Največ razlik med skupinami obstaja pri gradnji od 11 do 50 mikrostoritev in od 51 do 100 mikrostoritev. V skupini anketirancev, ki so v podjetju gradili mikrostoritve od začetka, jih je 8 odgovorilo, da so zgradili med 11 in 50 mikrostoritev, kar predstavlja 33 % anketirancev v tej skupini. Pri skupini anketirancev, ki so gradili mikrostoritve s prehodom, pa je ta odgovor izbralo 15 anketirancev oziroma 44 % anketirancev v tej skupini. V splošnem lahko sklepamo, da se slovenska podjetja,

ki uporabljajo mikrostoritveno arhitekturo, v večini poslužujejo gradnje do 50 mikrostoritev in gre v tem primeru verjetno za manjše programske opreme, ali pa so posamezne mikrostoritve bolj grobozrnate. Podrobni rezultati so prikazani v tabeli 5.

Tabela 5: Število ustvarjenih mikrostoritev glede na pristop grajenja mikrostoritvene programske opreme

	Mikrostoritve od začetka		**Mikrostoritve s prehodom**	
Število ustvarjenih mikrostoritev	Frekvenca	Relativna frekvenca	Frekvenca	Relativna frekvenca
Do 10	10	42 %	14	41 %
Od 11 do 50	8	33 %	15	44 %
Od 51 do 100	4	17 %	2	6 %
Več kot 100	2	8 %	3	9 %
Skupaj	24	100 %	34	100 %

Vir: lastno delo.

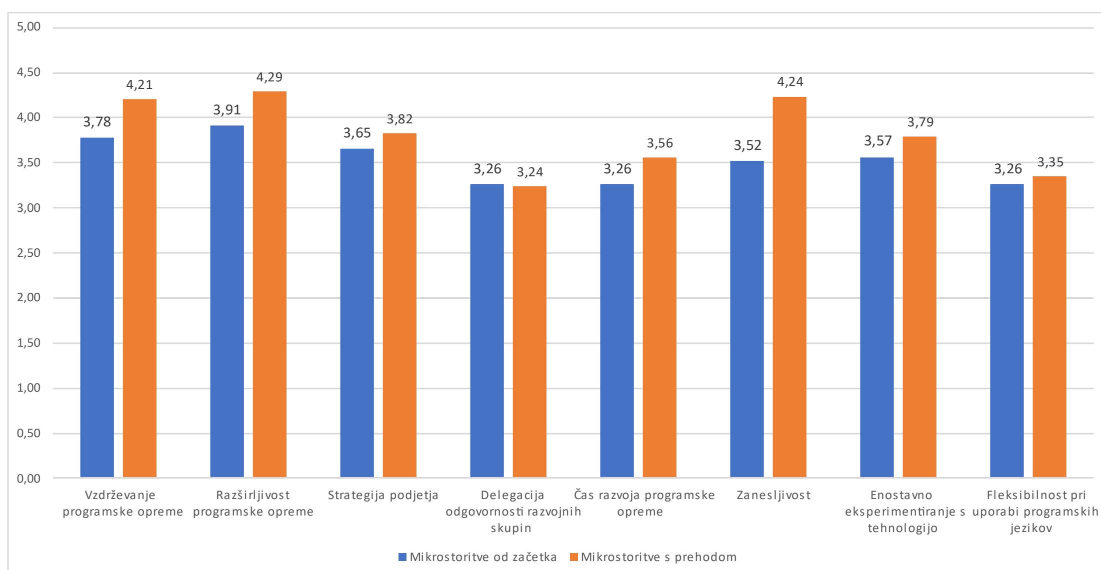
Naslednje vprašanje, ki sem ga postavil anketirancem, ki uporabljajo mikrostoritveno arhitekturo, je bilo, kako so različni dejavniki vplivali na izbor mikrostoritvene arhitekture. Zanimalo me je, ali obstajajo razlike med anketiranci, ki uporabljajo mikrostoritveno arhitekturo od začetka, in med tistimi, ki so izvedli prehod nanjo. Anketiranci so imeli na voljo 8 različnih dejavnikov, ki so jih ocenili na 5-stopenjski lestvici. Na vprašanje je skupaj odgovorilo 57 anketirancev. 23 anketirancev uporablja programsko opremo, ki je bila zgrajena na mikrostoritveni arhitekturi od začetka, preostalih 34 anketirancev pa je izvedlo prehod na mikrostoritveno arhitekturo. Na sliki 20 so prikazani rezultati odgovorov s povprečnimi ocenami, kako so posamezni dejavniki vplivali na izbor mikrostoritvene arhitekture. Anketiranci so imeli na voljo naslednje dejavnike:

- Vzdrževanje programske opreme: anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,78; anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, pa so pomembnost tega dejavnika ocenili s povprečno oceno 4,21.
- Razširljivost programske opreme: anketiranci, ki uporabljajo mikrostoritve od začetka, so ta dejavnik ocenili s povprečno oceno 3,91. Anketiranci, pri katerih je bil izveden prehod na mikrostoritveno arhitekturo, pa so pomembnost razširljivosti ocenili s povprečno oceno 4,29. Razširljivost programske opreme je dosegla najboljšo povprečno oceno med naštetimi dejavniki pri obeh vrstah grajenja mikrostoritev, kar pomeni, da razširljivost programske opreme igra pomembno vlogo pri odločanju za uporabo mikrostoritvene arhitekture.
- Strategija podjetja: anketiranci, ki uporabljajo mikrostoritve od začetka, so temu dejavniku namenili povprečno oceno 3,65. Anketiranci, pri katerih je bil izveden prehod na mikrostoritveno arhitekturo, pa so pomembnost skladnosti s strategijo podjetja ocenili s povprečno oceno 3,82.

- Delegacija odgovornosti razvojnih skupin: anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,26, anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, pa so pomembnost dejavnika ocenili s povprečno oceno 3,24. Glede na pridobljene rezultate je bil ta dejavnik med vsemi naštetimi dejavniki pri obeh skupinah anketirancev povprečno ocenjen najslabše.
- Čas razvoja programske opreme: anketiranci, ki uporabljajo mikrostoritve od začetka, so ta dejavnik ocenili s povprečno oceno 3,26. Anketiranci, pri katerih je bil izveden prehod, pa so pomembnost hitrosti razvoja programske opreme ocenili s povprečno oceno 3,56. Čas razvoja programske ocene je bil v povprečju eden od slabše ocenjenih dejavnikov, kar pomeni, da ta dejavnik ni bistveno vplival na izbor mikrostoritvene arhitekture.
- Zanesljivost: anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,53; anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, pa so pomembnost zanesljivosti ocenili s povprečno oceno 4,24, kar predstavlja zelo velik razkorak med tema dvema skupinama.
- Enostavno eksperimentiranje s tehnologijo: anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,57. Anketiranci, pri katerih je bil izveden prehod na mikrostoritveno arhitekturo, pa so pomembnost enostavnega eksperimentiranja s tehnologijo ocenili s povprečno oceno 3,79.
- Fleksibilnost pri uporabi programskih jezikov: anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,26; anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, pa so fleksibilnost ocenili s povprečno oceno 3,53.

Kot je razvidno iz slike 20, so imeli pri obeh pristopih najslabšo povprečno oceno naslednji dejavniki: delegacija odgovornosti razvojnih skupin, čas razvoja programske opreme in fleksibilnost pri uporabi programskih jezikov.

Slika 20 Dejavniki, ki so vplivali na izbor mikrostoritvene arhitekture



Vir: lastno delo.

Pri tretjem raziskovalnem vprašanju sem ugotovil, da so na izbor mikrostoritvene arhitekture pri podjetjih, ki so izvedla prehod, najbolj vplivali naslednji dejavniki: vzdrževanje programske opreme (4,21), razširljivost programske opreme (4,29) in zanesljivost (4,24). Pri podjetjih, ki uporabljajo mikrostoritve od začetka, pa so na to najbolj vplivali dejavniki: vzdrževanje programske opreme (3,78), razširljivost programske opreme (3,91) in strategija podjetja (3,65).

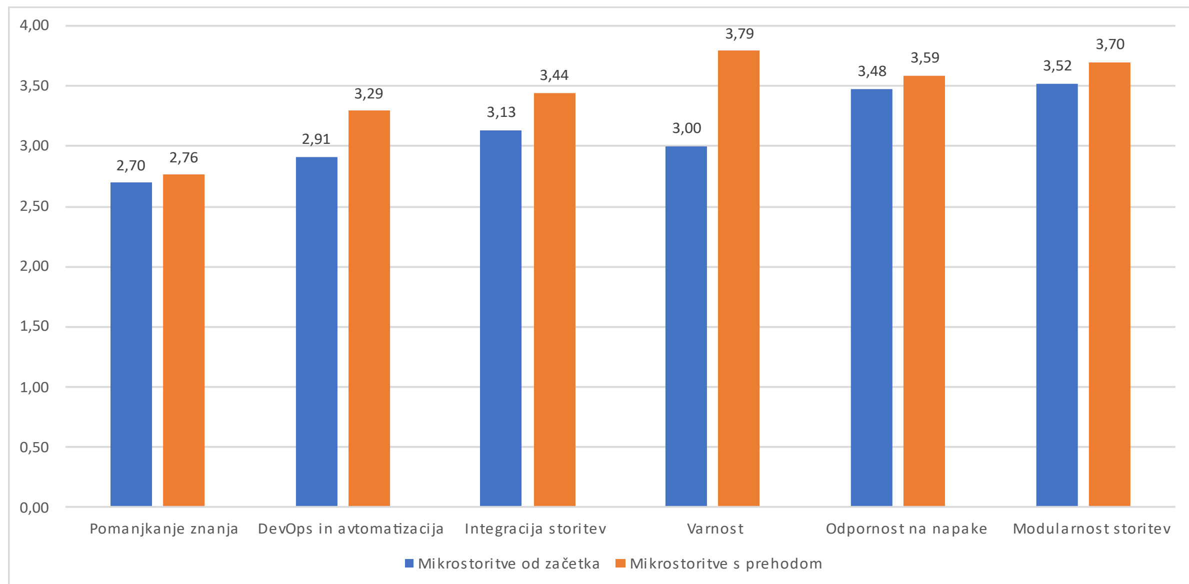
V nadaljevanju anketnega vprašalnika me je zanimalo, kateri so glavni dejavniki, ki so predstavljali največje izzive pri razvoju mikrostoritvene programske opreme. Anketiranci so imeli na voljo šest izzivov oziroma težav, ki se najpogosteje pojavljajo v literaturi. Na anketno vprašanje je odgovorilo 57 anketirancev; od tega jih je 23 anketirancev uporabljalo programsko opremo, ki je bila zgrajena na mikrostoritveni arhitekturi od začetka, ostali anketiranci pa so izvedli prehod nanjo. Anketiranci so odgovarjali s 5-stopenjsko lestvico, pri čemer je 1 pomenila, da izziv/težava ni imela popolnoma nič vpliva na razvoj mikrostoritvene programske opreme, ocena 5 pa je pomenila zelo velik vpliv. Anketiranci so ocenjevali naslednje težave:

- Pomankanje znanja: predstavljalo je najmanj težav pri razvoju mikrostoritvene programske opreme pri obeh pristopih razvoja, tako pri mikrostoritvah od začetka kot pri mikrostoritvah s prehodom. Anketiranci, ki uporabljajo mikrostoritveno programsko opremo od začetka, so izziv pomankanja znanja povprečno ocenili z 2,70, anketiranci s prehodom pa z 2,76.
- DevOps in avtomatizacija: anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 2,91; anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, pa so podali povprečno oceno 3,29. Vez med razvojem in operacijami podjetja ni predstavljala znatnih težav pri razvoju mikrostoritvene programske opreme, saj je iz slike razvidno, da je pri obeh pristopih takoj za pomankanjem znanja predstavljala najmanj izzivov.
- Integracija storitev: pri anketirancih, ki uporabljajo mikrostoritve od začetka, je bila povprečno ocenjena s 3,13. Anketiranci, ki so izvedli prehod, so integracijo storitev ocenili s povprečno oceno 3,44.
- Varnost: največ izzivov je varnost predstavljala anketirancem, ki so izvedli prehod na mikrostoritveno programsko opremo, saj so jo povprečno ocenili s 3,79. Na drugi strani pa so jo anketiranci, ki uporabljajo mikrostoritve od začetka, povprečno ocenili le s 3,00, kar pomeni, da se povprečna ocena razlikuje za kar 0,79.
- Odpornost na napake: anketiranci, ki uporabljajo mikrostoritve od začetka, so odpornost na napake povprečno ocenili s 3,48; anketiranci, pri katerih je bil opravljen prehod na mikrostoritveno programsko opremo, pa so odpornost na napake povprečno ocenili s 3,59.
- Modularnost storitev: največji izziv je modularnost predstavljala anketirancem, ki uporabljajo mikrostoritve od začetka, saj so podali povprečno oceno kar 3,52. Ravno tako so se z izzivi srečali tudi anketiranci, ki so opravili prehod, saj so modularnost storitev povprečno ocenili 3,70.

Rezultate sem prikazal na sliki 21, kjer so anketiranci, ki uporabljajo mikrostoritve od začetka, vse naštete izzive, povezane z gradnjo mikrostoritev, v povprečju ocenili z nižjo povprečno oceno kot anketiranci, ki v podjetju uporabljajo mikrostoritve s prehodom. Iz tega je možno

sklepati, da so ti anketiranci imeli manj težav pri gradnji programske opreme na mikrostoritveni arhitekturi. Anketirancem, ki uporabljajo mikrostoritve od začetka, je največ težav predstavljala modularnost storitev; anketirancem, ki uporabljajo mikrostoritve, zgrajene s prehodom, pa je največ težav predstavljala varnost programske opreme.

Slika 21: vpliv težav na razvoj mikrostoritvene programske opreme



Vir: lastno delo.

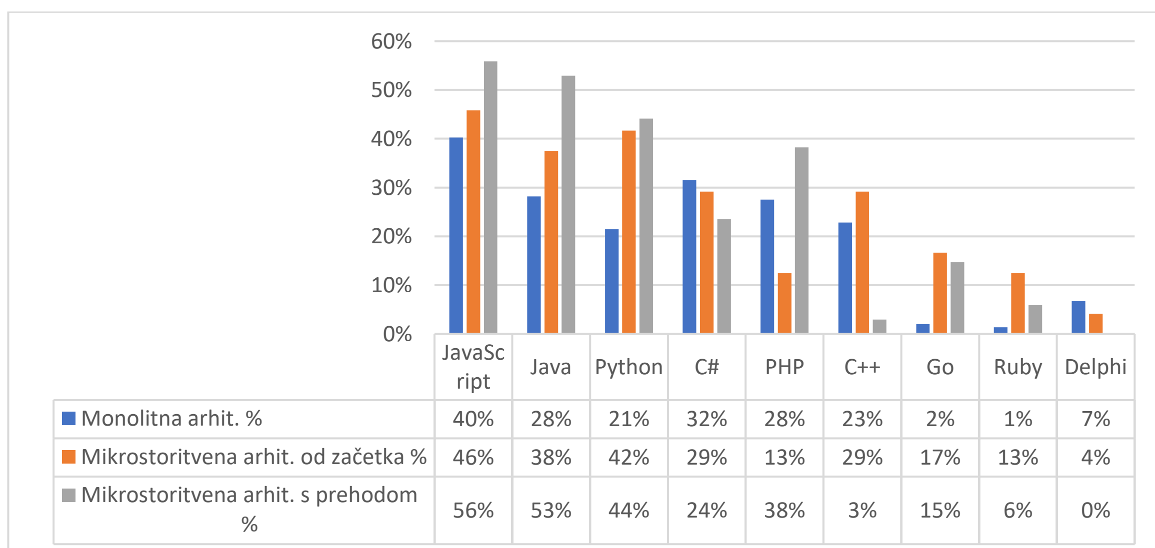
Pri četrtem raziskovalnem vprašanju sem ugotovil, da so se podjetja, ki so izvedla prehod na mikrostoritveno arhitekturo, srečala z naslednjimi težavami: varnost (3,79), odpornost na napake (3,59) in modularnost storitev (3,70). Pri podjetjih, ki uporabljajo mikrostoritve od začetka, pa so se pojavljale naslednje težave: integracija storitev (3,13), odpornost na napake (3,48) in modularnost storitev (3,52). Bistvene razlike med mikrostoritvami s prehodom in mikrostoritvami od začetka se pojavijo le pri varnosti.

Naslednje anketno vprašanje je bilo povezano z uporabo različnih programskih jezikov glede na obliko programske arhitekture. Na to vprašanje so odgovarjali anketiranci z vseh treh oblik programske arhitekture. Nanj je bilo mogoče odgovoriti z več različnimi odgovori, saj je lahko za posamezno programsko opremo uporabljenih več programskih jezikov. Anketiranci so imeli za odgovore na voljo 8 najpogostejših programskih jezikov, ki se pojavljajo v literaturi, in možnost, da manjkajočega tudi dopišejo. Na vprašanje, katere programske jezike anketiranci uporabljajo v podjetju pri razvoju programske opreme, je odgovorilo skupno 207 anketirancev. Od skupno 207 anketirancev 149 anketirancev uporablja monolitno programsko arhitekturo, 24 anketirancev uporablja programsko opremo, ki je bila od začetka razvita na mikrostoritveni arhitekturi, preostalih 34 anketirancev pa uporablja programsko opremo, pri kateri je bil izveden prehod na mikrostoritveno arhitekturo.

Med rezultati lahko razberemo da je programski jezik JavaScript najpogostejši jezik pri vseh oblikah programske arhitekture. Največji delež anketirancev, ki uporabljajo ta programski jezik, predstavljajo anketiranci, ki uporabljajo mikrostoritveno arhitekturo s prehodom. Njihov delež v njihovi skupini znaša kar 56 % oziroma 19 od skupno 34 anketirancev. Delež anketirancev z monolitno programsko arhitekturo, ki uporablja ta programski jezik, pa je 40 % oziroma 60 anketirancev, ki uporabljajo to programsko opremo. Med priljubljenimi programskimi jeziki za gradnjo programske opreme sta tudi Java in Python. V vseh teh treh programskih jezikih se lahko z slike razbere vzorec, kjer najmanjši delež uporabe predstavljajo anketiranci, ki uporabljajo monolitno arhitekturo, nato sledijo anketiranci, ki uporabljajo mikrostoritveno arhitekturo od začetka, največji delež pa predstavljajo anketiranci, ki uporabljajo mikrostoritveno arhitekturo s prehodom.

Med najmanj priljubljene programske jezike, uporabljene pri programski opremi, sodijo Go, Ruby in Delphi. Slednjega so anketiranci vpisali kot dopolnitev predlaganega seznama programskih jezikov. Zanimivo je, da Ruby in Go prevladujeta pri programski opremi, grajeni na mikrostoritveni arhitekturi, medtem ko ju na monolitni arhitekturi uporablja zelo malo anketirancev. Nekaj anketirancev uporablja tudi programski jezik Delphi, med katerimi je največ takih, ki uporabljajo monolitno programsko arhitekturo (10 anketirancev oziroma 7 % vseh anketirancev), medtem ko tega programskega jezika ne uporablja noben anketiranec, ki uporablja mikrostoritveno arhitekturo s prehodom. Rezultati so prikazani na sliki 22.

Slika 22: Uporaba programskih jezikov glede na obliko programske arhitekture



	JavaScript	Java	Python	C#	PHP	C++	Go	Ruby	Delphi
Monolitna arhit. %	40%	28%	21%	32%	28%	23%	2%	1%	7%
Mikrostoritvena arhit. od začetka %	46%	38%	42%	29%	13%	29%	17%	13%	4%
Mikrostoritvena arhit. s prehodom %	56%	53%	44%	24%	38%	3%	15%	6%	0%

Vir: lastno delo.

Pri petem raziskovalnem vprašanju me je zanimalo, kateri so glavni programski jeziki, ki jih podjetja uporabljajo pri grajenju programske opreme. Ugotovil sem, da je pri vseh pristopih grajenja programske opreme najpogosteje uporabljen programski jezik JavaScript. Ostali pogosto uporabljeni programski jeziki so še Java, Python in C#. Z analizo ugotavljam, da se izbor programskega jezika bistveno ne razlikuje glede na obliko programske arhitekture.

Naslednje anketno vprašanje je bilo, na katerem strežniku imajo anketiranci nameščeno programsko opremo. Na to anketno vprašanje so imeli anketiranci na voljo več možnih vnaprej pripravljenih odgovorov. Izbrali so lahko več različnih odgovorov, saj so lahko posamezne funkcionalnosti programske opreme nameščene na različnih strežnikih. Na to anketno vprašanje je odgovorilo 208 anketirancev, od teh jih 149 uporablja monolitno arhitekturo, 24 mikrostoritveno arhitekturo od začetka in 35 mikrostoritveno arhitekturo s prehodom. Tako kot pri prejšnjem vprašanju je bila tudi tu možnost, da anketiranci podajo dodatne ponudnike strežnikov (ki jih ni na seznamu), na katerih imajo nameščeno programsko opremo.

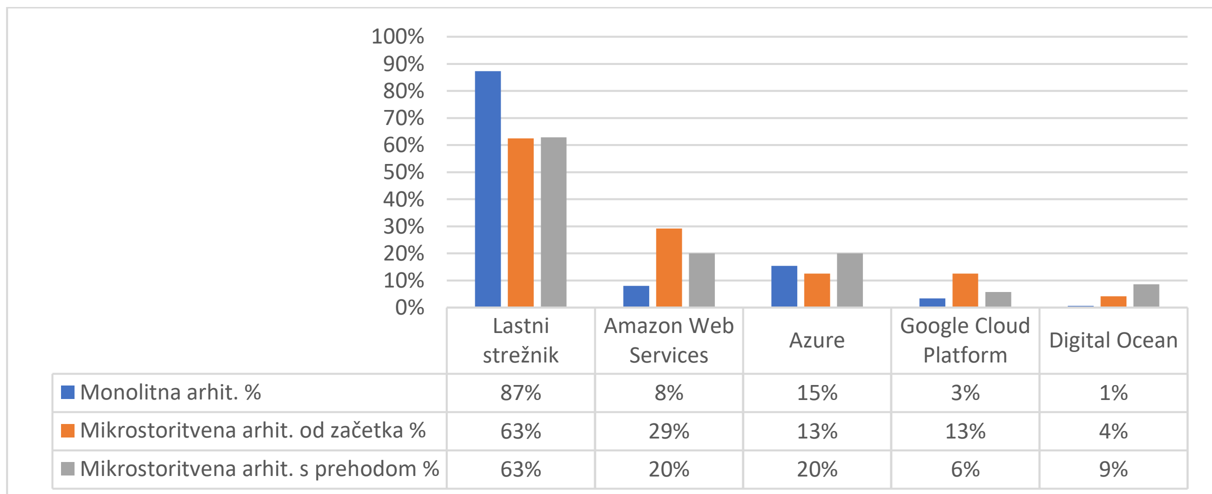
Največ anketirancev iz vseh treh skupin oblik programske arhitekture uporablja lasten strežnik. Delež anketirancev, ki uporabljajo lastni strežnik, je največji pri anketirancih, ki v podjetju uporabljajo monolitno arhitekturo, pri čemer lastni strežnik uporablja kar 87 % vseh anketirancev iz te skupine. Pri anketirancih, ki uporabljajo mikrostoritveno programsko arhitekturo, jih lastni strežnik uporablja 63 % ne glede na to, ali je programska oprema grajena na mikrostoritveni arhitekturi od začetka ali pa se je zgodil prehod. Ta delež je še vedno zelo visok glede na to, da se pri mikrostoritveni arhitekturi stremi k razpršitvi posameznih storitev na različne strežnike.

Med ponudniki oblačnih storitev je pri mikrostoritveni arhitekturi od začetka najbolj priljubljen ponudnik Amazon Web Services, ki ga uporablja 29 % anketirancev iz te skupine in 20 % anketirancev, ki uporabljajo mikrostoritveno programsko opremo, pri kateri se je zgodil prehod. Amazon Web Services uživa manjšo podporo pri podjetjih, kjer uporabljajo monolitno programsko arhitekturo, saj ga uporablja le 8 % anketirancev.

Med ponudniki spletnih storitev je pri anketirancih, ki uporabljajo monolitno programsko arhitekturo, najbolj priljubljen ponudnik Azure, ki ga uporablja 15 % podjetij, v katerih delujejo anketiranci. Podoben delež predstavljajo tudi anketiranci, ki uporabljajo mikrostoritveno arhitekturo od začetka (13 %), nekaj več pa na spletnega ponudnika Azure prisega tistih anketirancev, ki uporabljajo mikrostoritveno arhitekturo s prehodom (20 %).

Pri uporabi oblačnih storitev lahko sklepamo, da prevladujejo podjetja s programsko opremo, zgrajeno na mikrostoritveni arhitekturi, kar sovpada z navedbami avtorjev, ki sem jih opisal v teoretičnem delu. Iz rezultatov ankete lahko tudi sklepamo, da pri uporabi lastnih strežnikov za namestitev programske opreme prevladujejo podjetja, ki uporabljajo monolitno arhitekturo. Rezultati so predstavljeni na sliki 23.

Slika 23: Uporaba strežnikov glede na obliko programske arhitekture



	Lastni strežnik	Amazon Web Services	Azure	Google Cloud Platform	Digital Ocean
Monolitna arhit. %	87%	8%	15%	3%	1%
Mikrostoritvena arhit. od začetka %	63%	29%	13%	13%	4%
Mikrostoritvena arhit. s prehodom %	63%	20%	20%	6%	9%

Vir: lastno delo.

Naslednje vprašanje se je glasilo: kako so anketiranci zaznali različne prednosti programske opreme glede na vrsto programske arhitekture, ki jo v podjetju uporabljajo. Na vprašanje so odgovarjali vsi anketiranci, ki uporabljajo monolitno programsko opremo in mikrostoritve od začetka, ter anketiranci, ki so izvedli prehod na mikrostoritveno programsko opremo. Anketiranci so ocenjevali sedem različnih prednosti, ki se pogosto pojavljajo v literaturi. Prednosti so ocenjevali s 5-stopenjsko lestvico, na kateri ocena 1 pomeni, da korist sploh ni zaznana, ocena 5 pa, da je korist zelo zaznana. Na to anketno vprašanje je odgovorilo 23 anketirancev, ki v podjetju uporabljajo programsko opremo, zgrajeno na mikrostoritveni arhitekturi od začetka; 34 anketirancev, ki je izvedlo prehod na mikrostoritveno arhitekturo; in 146 anketirancev, ki v podjetju uporabljajo monolitno arhitekturo. Skupno je pri tem anketnem vprašanju sodelovalo 203 anketirancev. Anketiranci so ocenjevali naslednje posamezne prednosti:

- Možnost hitrega vzdrževanja programske opreme: anketiranci, ki uporabljajo monolitno programsko arhitekturo, so podali povprečno oceno 3,55; anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,74; medtem ko so anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, možnost hitrega vzdrževanja programske opreme povprečno ocenili s 4,03.
- Razširljivost programske opreme: pri anketirancih, ki uporabljajo monolitno arhitekturo, je bila povprečno ocenjena s 3,46; anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,78. Anketiranci, ki so izvedli prehod, so razširljivost programske ocene ocenili s povprečno oceno 4,24.
- Majhna arhitekturna kompleksnost: anketiranci, ki uporabljajo monolitno arhitekturo, so majhno arhitekturno kompleksnost povprečno ocenili s 3,19; anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,39; anketiranci, ki so opravili prehod, pa so jo povprečno ocenili z 3,06.

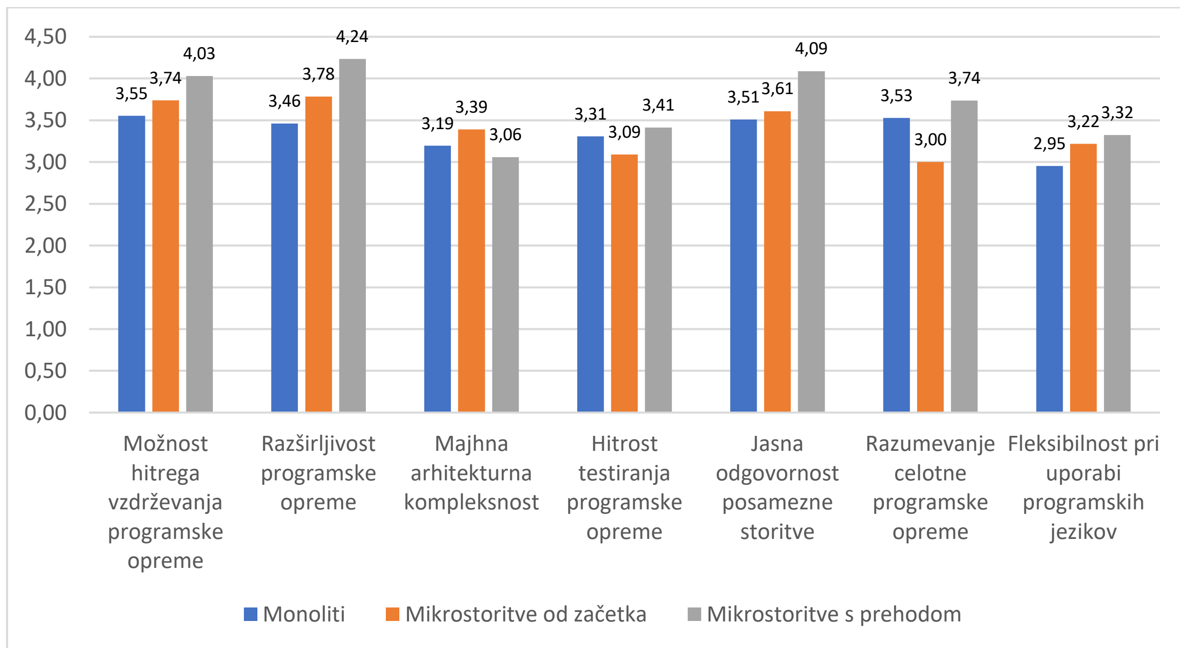
- Hitrost testiranja programske opreme: pri anketirancih, ki uporabljajo monolitno arhitekturo, je bila povprečno ocenjena s 3,31. Anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,09; anketiranci, ki so opravili prehod, pa so hitrost testiranja podatkov povprečno ocenili z 3,41.
- Jasna odgovornost posamezne storitve: anketiranci, ki uporabljajo monolitno arhitekturo, so podali povprečno oceno 3,51; anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,61; medtem ko so anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, jasno odgovornost posamezne storitve povprečno ocenili z 4,09.
- Razumevanje celotne programske opreme: pri anketirancih, ki uporabljajo monolitno arhitekturo, je bila povprečno ocenjena s 3,53; anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,00. Anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo so podali povprečno oceno 3,74.
- Fleksibilnost pri uporabi programske opreme: anketiranci, ki uporabljajo monolitno arhitekturo, so fleksibilnost pri uporabi programskih jezikov povprečno ocenili z 2,95, iz česar je razvidno, da korist ni zaznana; anketiranci, ki uporabljajo mikrostoritve od začetka, so podali povprečno oceno 3,22; anketiranci, ki so opravili prehod na mikrostoritveno arhitekturo, pa so jo povprečno ocenili s 3,32.

S slike 24 je razvidno, da so določene lastnosti/prednosti programske arhitekture opaznejše pri nekaterih arhitekturnih slogih, določene pa pri drugih. Podjetjem v Sloveniji, ki uporabljajo mikrostoritveno programsko arhitekturo, je tako skupno, da največ prednosti pripisujejo razširljivosti programske opreme, saj je ta med anketiranci dosegla najboljšo povprečno oceno pri obeh pristopih gradnje mikrostoritvene programske opreme.

Na drugi strani je pri anketirancih, ki uporabljajo monolitno programsko arhitekturo, najboljšo povprečno oceno dosegla lastnost možnost hitrega vzdrževanja programske opreme, vendar pa je ta povprečna ocena slabša od povprečne ocene pri obeh pristopih grajenja programske opreme na mikrostoritveni arhitekturi. Od grajenja programske opreme na mikrostoritveni arhitekturi od začetka je povprečna ocena manjša za 0,19, od prehoda na mikrostoritveno arhitekturo pa za kar 0,48.

Z najslabšo oceno so anketiranci, ki v podjetju uporabljajo monolitno programsko arhitekturo, ocenili fleksibilnost pri uporabi programskih jezikov, in sicer s povprečno oceno 2,95. Ta slabost monolitne programske opreme je omenjena tudi v teoretičnem delu, saj lahko poleg ostalih težav zaradi nefleksibilnosti pride do zastaranja tehnologije, nezmožnosti dodajanja novih funkcionalnosti in izdajanja posodobitev programske opreme. Glede na literaturo je ta pomanjkljivost monolitne arhitekture ena od večjih prednosti mikrostoritvene arhitekture, vendar pa so to prednost anketiranci, ki uporabljajo mikrostoritveno programsko arhitekturo od začetka, ocenili s povprečno oceno le 3,22; anketiranci, ki so izvedli prehod na mikrostoritveno arhitekturo, pa s 3,32.

Slika 24: Zaznavanje prednosti glede na različno obliko programske arhitekture



Vir: lastno delo.

6.4 Testiranje in analiza hipotez

Da bi čim bolje preučil stanje programske arhitekture pri slovenskih podjetjih, sem glede na postavljena raziskovalna vprašanja (od 6 do 10) oblikoval hipoteze, ki sem jih na podlagi rezultatov spletne ankete potrdil ali ovrgel. Pri tem je potrebno upoštevati, da se hipoteze navezujejo na podjetja, ki imajo več kot 20 zaposlenih, oziroma manj kot 20, če se podjetje primarno ukvarja z informacijsko tehnologijo in se ne navezujejo na podjetja, z manj kot 20 zaposlenih kjer njihova primarna dejavnost ni informacijska tehnologija.

Hipoteza 1a: Podjetja, ki uporabljajo mikrostoritveno arhitekturo, zaznavajo hitrejše vzdrževanje programske opreme kot podjetja, ki uporabljajo monolitno programsko opremo.

Hipotezo sem preverjal z analizo anketnega vprašanja o tem, kako anketiranci zaznavajo prednost obstoječe programske opreme glede možnosti hitrega vzdrževanja programske opreme. Anketiranci so na to vprašanje odgovarjali na Likertovi lestvici od 1 do 5. S testom dveh aritmetičnih sredin neodvisnih vzorcev z neznano populacijsko varianco sem preveril, ali je aritmetična sredina ocene hitrega vzdrževanja programske opreme pri anketirancih, ki v podjetju uporabljajo monolitno arhitekturo, manjša od anketirancev, ki v podjetju uporabljajo mikrostoritveno arhitekturo. Pri tem testu sem najprej združil ocene anketirancev, ki v podjetju uporabljajo mikrostoritveno arhitekturo od začetka, in anketirancev, ki v podjetju uporabljajo mikrostoritveno arhitekturo s prehodom, saj so bila vprašanja za vsak tip arhitekture postavljena ločeno. Na ta način sem dobil skupek ocen anketirancev, ki uporabljajo mikrostoritveno arhitekturo. Osnovna statistika je prikazana v tabeli 6.

Tabela 6: Osnovna statistika možnosti hitrega vzdrževanja programske opreme

	Monolitna arhitektura	**Mikrostoritvena arhitektura**
Aritmetična sredina	3,55	3,91
Standardni odklon	1,06	1,02
Velikost	146	57

Vir: lastno delo.

Pri preverjanju hipoteze sem postavil ničelno in alternativno hipotezo. Ta hipoteza velja tudi za hipotezi 1b in 1c:

- H_0: $\mu_{mo} \geq \mu_{ms}$
- H_1: $\mu_{mo} < \mu_{ms}$

Hipoteza je bila preverjena z enostranskim testom dveh aritmetičnih sredin neodvisnih vzorcev. Izračunana p-vrednost testa je 0,014. Ker je p-vrednost manjša od $\alpha = 0,05$ $(0,014 < 0,05)$ **zavrnem ničelno hipotezo**, da je aritmetična sredina možnosti hitrega vzdrževanja programske opreme pri podjetjih, ki uporabljajo monolitno programsko opremo, večja ali enaka aritmetični sredini možnosti hitrega vzdrževanja programske opreme pri podjetjih, ki uporabljajo mikrostoritveno arhitekturo.

Hipoteza 1b: Podjetja, ki uporabljajo mikrostoritveno programsko arhitekturo, zaznavajo boljšo učinkovitost programske opreme glede razširljivosti programske opreme kot podjetja, ki uporabljajo monolitno programsko arhitekturo.

Hipoteza je bila preverjena z anketnim vprašanjem o tem, kako anketiranci zaznavajo prednost obstoječe programske opreme glede razširljivosti programske opreme. To vprašanje se je v anketi pojavilo večkrat glede na to, katero vrsto arhitekture anketiranci uporabljajo. Anketiranci so na vprašanje odgovarjali na Likertovi lestvici od 1 do 5. Statistika je prikazana v tabeli 7.

Tabela 7: Osnovna statistika učinkovitosti glede na razširljivost programske opreme

	Monolitna arhitektura	**Mikrostoritvena arhitektura**
Aritmetična sredina	3,46	4,05
Standardni odklon	1,06	0,91
Velikost	145	57

Vir: lastno delo.

Na anketno vprašanje je odgovorilo skupno 202 anketirancev, med njimi jih je v podjetju 145 uporabljalo monolitno in 57 mikrostoritveno arhitekturo. Povprečna ocena pri anketirancih, ki

v podjetju uporabljajo monolitno arhitekturo, je bila 3,46; pri anketirancih, ki uporabljajo mikrostoritveno programsko arhitekturo, pa 4,05.

Hipotezo H1b sem preveril s testom dveh aritmetičnih sredin neodvisnih vzorcev, pri čemer je rezultat t = -3,699, p = 0,000. Na osnovi tega rezultata lahko **zavrnem ničelno hipotezo**, da je aritmetična sredina glede razširljivosti monolitne programske arhitekture višja od aritmetične sredine razširljivosti mikrostoritvene programske arhitekture.

Hipoteza 1c: Podjetja, ki uporabljajo mikrostoritveno arhitekturo, zaznavajo manjšo arhitekturno kompleksnost kot podjetja, ki uporabljajo monolitno arhitekturo.

Hipoteza je bila preverjena z anketnim vprašanjem o tem, kako anketiranci zaznavajo prednost obstoječe programske opreme glede arhitekturne kompleksnosti. Tako kot pri hipotezi H1a so tudi pri tej združeni odgovori obeh tipov mikrostoritvene arhitekture (s prehodom in od začetka), da sem lahko dobil celotno stanje podjetij, ki uporabljajo to arhitekturo. Anketiranci so na to vprašanje odgovarjali na Likertovi lestvici od 1 do 5. V tabeli 8 je prikazana osnovna statistika vprašanja o arhitekturni kompleksnosti:

Tabela 8: Osnovna statistika glede arhitekturne kompleksnosti

	Monolitna arhitektura	**Mikrostoritvena arhitektura**
Aritmetična sredina	3,19	3,19
Standardni odklon	1,04	1,36
Velikost	144	57

Vir: lastno delo.

Hipotezo H1c sem preveril s testom dveh aritmetičnih sredin neodvisnih vzorcev, pri čemer je rezultat t-testa t = 0, p = 0,500, kar pomeni da na osnovi rezultata **ne morem zavrniti** ničelne hipoteze pri stopnji značilnosti $\alpha = 0,05$, da anketiranci, ki v podjetju uporabljajo monolitno arhitekturo, zaznavajo boljšo ali enako učinkovitost glede arhitekturne kompleksnosti programske opreme.

Hipoteza 2: Večina podjetij v Sloveniji še vedno uporablja monolitno arhitekturo in ne mikrostoritvene.

Hipoteza 2 pravi, da večina podjetij v Sloveniji še vedno uporablja monolitno arhitekturo. S slike 15 je razvidno, da je od skupno 373 udeležencev ankete 254 udeležencev odgovorilo, da uporabljajo monolitno arhitekturo, kar predstavlja 68 % vseh anketirancev, preostalih 32 % pa uporablja mikrostoritveno arhitekturo.

Pri preverjanju hipoteze sem postavil ničelno in alternativno hipotezo:

- H_0: PI ≥ 0,5
- H_1: PI < 0,5

Hipotezo 2 sem preveril z binomskim testom, pri čemer je rezultat testa p = 0,999, kar pomeni, da na osnovi rezultata **zavrnem** ničelno hipotezo pri α = 0,05, da več ali enako število slovenskih podjetij uporablja mikrostoritveno in ne monolitno programsko arhitekturo.

Hipoteza 3: Podjetja, ki uporabljajo mikrostoritveno programsko arhitekturo, pogosteje posegajo po oblačnih ponudnikih storitev za namestitev programske opreme kot podjetja, ki uporabljajo monolitno programsko arhitekturo.

Hipotezo sem preveril z anketnim vprašanjem, na katerih strežnikih imajo anketiranci nameščeno programsko opremo. Na to vprašanje so odgovarjale vse tri skupine udeležencev. Med anketiranci je bilo 149 takih, ki uporabljajo monolitno programsko arhitekturo, in 59 takih, ki uporabljajo mikrostoritveno arhitekturo. Med anketiranci, ki uporabljajo monolitno arhitekturo, je 47 takih, ki uporabljajo oblačne ponudnike storitev, med anketiranci, ki uporabljajo mikrostoritveno programsko arhitekturo, pa jih je 31. Za anketirance, ki v podjetju posegajo po oblačnih ponudnikih storitev, sem štel vse tiste anketirance, ki so odgovorili, da uporabljajo vsaj enega ponudnika oblačnih storitev, četudi poleg tega uporabljajo tudi lastne strežnike. Osnovna statistika anketnega vprašanja je prikazana v tabeli 9.

Tabela 9: Osnovna statistika glede uporabe oblačnih ponudnikov storitev za nameščanje programske opreme

	Monolitna arhitektura	**Mikrostoritvena arhitektura**
Aritmetična sredina	0,31	0,53
Standardni odklon	0,47	0,50
Velikost	149	59

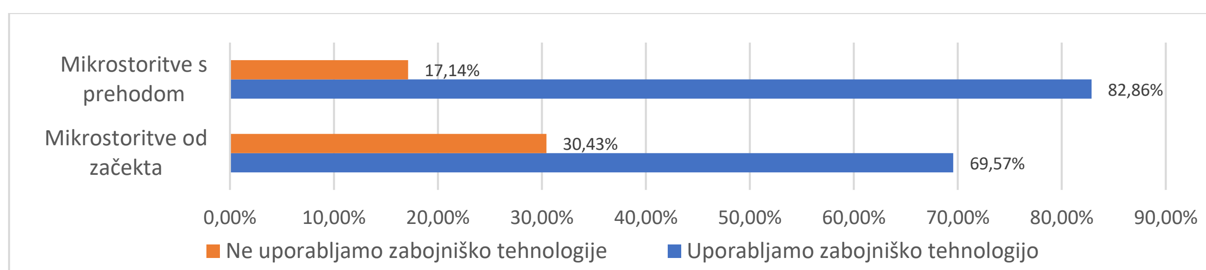
Vir: lastno delo.

Hipotezo sem preveril s testom dveh aritmetičnih sredin neodvisnih vzorcev, pri čemer je rezultat t-testa 2,871, p = 0,002. Na osnovi tega rezultata lahko **zavrnem ničelno hipotezo** pri α = 0,05, da uporabniki monolitne programske arhitekture pogosteje ali enako pogosto posegajo po ponudnikih oblačnih storitev kot uporabniki mikrostoritvene programske arhitekture, in sprejmem alternativno hipotezo, da uporabniki mikrostoritvene programske arhitekture pogosteje posegajo po ponudnikih oblačnih storitev kot uporabniki mikrostoritvene arhitekture.

Hipoteza 4: Podjetja, ki uporabljajo mikrostoritveno arhitekturo od začetka, pogosteje uporabljajo zabojniško tehnologijo kot podjetja, ki so izvedla prehod na mikrostoritveno arhitekturo.

Hipotezo sem preveril z anketnim vprašanjem o tem, ali so pri razvoju uporabili zabojniško tehnologijo, kot je npr. Docker. Na to anketno vprašanje so odgovarjali samo anketiranci, ki imajo v podjetju zgrajeno programsko opremo na mikrostoritveni arhitekturi od začetka, in anketiranci, ki so bili v podjetju deležni prehoda na to arhitekturo. Na anketno vprašanje je skupno odgovorilo 58 anketirancev, od tega je 23 takih, ki v podjetju uporabljajo mikrostoritveno arhitekturo od začetka, in 35 iz podjetja, kjer se je zgodil prehod na mikrostoritveno arhitekturo. Na sliki 25 je prikazan delež uporabe zabojniške tehnologije glede na vrsto prehoda mikrostoritvene arhitekture (s prehodom ali od začetka), v tabeli 10 pa osnovna statistika.

Slika 25: Uporaba zabojniške arhitekture



Vir: lastno delo.

Hipotezo sem preveril s testom dveh aritmetičnih sredin neodvisnih vzorcev. Rezultat t-testa je 1,16, $p = 0{,}874$, pri čemer pri stopnji značilnosti $\alpha = 0{,}05$, **ne morem zavrniti** ničelne hipoteze, da podjetja, ki uporabljajo mikrostoritveno arhitekturo s prehodom, pogosteje ali enako pogosto uporabljajo zabojniško tehnologijo kot podjetja, ki uporabljajo mikrostoritveno arhitekturo od začetka.

Tabela 10: Osnovna statistika naklonjenosti do zabojniške tehnologije

	Mikrostoritve od začetka	**Mikrostoritve s prehodom**
Aritmetična sredina	0,70	0,83
Standardni odklon	0,47	0,38
Velikost	23	35

Vir: lastno delo.

Hipoteza 5: Podjetja, ki uporabljajo mikrostoritveno programsko opremo, uporabljajo več programskih jezikov kot podjetja, ki uporabljajo monolitno arhitekturo.

Hipotezo sem preverjal z anketnim vprašanjem o tem, katere programske jezike anketiranci uporabljajo pri trenutni programski opremi. Na vprašanje so odgovarjale vse tri skupine anketirancev. Na vprašanje je veljavno odgovorilo skupaj 199 anketirancev, od tega jih je 141 uporabljalo monolitno programsko arhitekturo, preostalih 58 pa mikrostoritveno. V tabeli 11 je

prikazana osnovna statistika odgovorov glede števila programskih jezikov, ki jih anketiranci uporabljajo v podjetju.

Tabela 11: Osnovna statistika uporabe različnih programskih jezikov

	Monolitna arhitektura	**Mikrostoritvena arhitektura**
Aritmetična sredina	2,19	2,52
Standardni odklon	1,15	1,25
Velikost	141	58

Vir: lastno delo.

Hipotezo sem preveril s testom dveh aritmetičnih sredin neodvisnih vzorcev. Rezultat t-testa je 1,793, p = 0,037. Na podlagi tega rezultata lahko pri stopnji značilnosti α = 0,05 **zavrnem** ničelno hipotezo, da podjetja, ki uporabljajo monolitno programsko arhitekturo, uporabljajo enako ali večje število programskih jezikov kot podjetja, ki uporabljajo mikrostoritveno arhitekturo, in sprejmem alternativno hipotezo, da podjetja, ki uporabljajo mikrostoritveno arhitekturo uporabljajo več programskih jezikov kot podjetja, ki uporabljajo monolitno programsko arhitekturo.

7 DISKUSIJA O UGOTOVITVAH EMPIRIČNE RAZISKAVE

Ključne ugotovitve magistrskega dela nakazujejo, da je stanje v slovenskih podjetjih glede uporabe programske arhitekture zelo podobno tistemu v literaturi, vendar pa obstaja tudi nekaj razlik. Vzrok razhajanj morda leži v tem, da je večina raziskovanj na temo programske arhitekture omejena samo na določeno domeno raziskovanja ali pa samo na določena podjetja v določenih panogah delovanja.

Z raziskovanjem sem potrdil, da večina anketirancev v slovenskih podjetjih še vedno uporablja monolitno programsko arhitekturo, saj ta v veliki meri zadostuje trenutnim potrebam podjetja, kar nekaj anketirancev pa sploh še ni slišalo za mikrostoritveno arhitekturo. To bi lahko nakazovalo na pomanjkanje izobraževanja v slovenskih podjetjih. Nad samimi rezultati uporabe monolitne arhitekture sicer nisem bil presenečen, saj je, kakor pravi Fowler (2015), monolitna programska oprema še dandanes vidna skoraj povsod, saj jo še vedno uporablja veliko podjetij in drugih organizacij, pa čeprav spada med eno od starejših oblik aplikacijske arhitekture.

Ugotavljal sem tudi, kateri so glavni vzroki, zaradi katerih se slovenska podjetja odločajo za uporabo mikrostoritvene arhitekture. Ugotovil sem, da določeni vzroki prevladujejo nad ostalimi. V veliki meri prevladujejo boljša možnost vzdrževanja programske opreme, razširljivost programske opreme in zanesljivost programske opreme. V študiji, ki so jo pripravili Fritzsch, Bogner, Wagner in Zimmermann (2019), pa so ugotovili, da so glavni vzroki za uporabo te programske arhitekture možnost vzdrževanja programske opreme, razširljivost in

zadovoljevanje funkcijskih zahtev podjetja. Zanimivo je, da slovenska podjetja, ki so se odločila za prehod iz monolitne na mikrostoritveno programsko opremo, kot enega glavnih vzrokov navajajo zanesljivost (na drugem mestu po pomembnosti), medtem ko je ta vzrok pri Fritzsch, Bogner, Wagner in Zimmermann (2019) na zadnjem mestu med vzroki za uporabo mikrostoritvene arhitekture.

Do podobnih ugotovitev sem prišel tudi pri uporabi programskih jezikov pri mikrostoritveni programski opremi pri slovenskih podjetjih, saj izstopa 5 programskih jezikov, ki jih podjetja uporabljajo pri razvoju aplikacij. Ti programski jeziki so: Javascript, Java, Python, C# in PHP. Podobne rezultate so pridobili tudi v raziskavi, ki so jo izvedli Viggiato, Terra, Rocha, Valente in Figueiredo (2018). Glede na njihove in moje rezultate izstopa le programski jezik Python, ki je v moji raziskavi tretji najpogosteje uporabljan programski jezik pri gradnji mikrostoritvene programske opreme, pri Viggiato, Terra, Rocha, Valente in Figueiredo (2018), pa ta jezik sploh ni omenjen kot eden od ključnih programskih jezikov za grajenje mikrostoritvene arhitekture.

Glede na to, da obstaja zelo malo raziskav na področju programske arhitekture pri slovenskih podjetjih, rezultatov te raziskave na žalost nisem mogel primerjati na takem nivoju. Kljub temu, da sem v to raziskavo vključil podjetja različnih velikosti po številu zaposlenih in s celotnega območja Slovenije, je vzorec podjetij, ki uporabljajo mikrostoritveno programsko arhitekturo, majhen, kar predstavlja omejitev raziskave pri posploševanju na celotno področje. Ker so rezultati te raziskave izključno kvantitativne narave z uporabo anketnega vprašalnika, lahko rezultati magistrskega dela služijo kot povod za prihodnja poglobljena raziskovanja v smeri pridobivanja kvalitativnih podatkov (npr. poglobljeni intervjuji), na podlagi katerih bi lahko še bolje razumeli stanje programske arhitekture v slovenskih podjetjih ter ugotovili, zakaj še vedno prevladuje monolitna arhitektura in kakšni so razlogi za to.

SKLEP

Mikrostoritvena programska arhitektura je relativno nov pojem v svetu grajenja programske opreme, ki pa postaja vse bolj priljubljena med različnimi podjetji. Ta programska arhitektura je nastala kot odgovor na težave, prisotne pri monolitni programski arhitekturi, saj mora biti programska oprema v današnjem svetu hitro prilagodljiva na odzive iz okolja, v katerem podjetje deluje.

V magistrskem delu sem se osredotočil na pregled celotne slike programske arhitekture v Sloveniji. V teoretičnem delu sem analiziral monolitno in mikrostoritveno programsko arhitekturo, pozneje pa tudi prehod s prve na drugo arhitekturo. Analiziral sem bistvene prednosti in slabosti obeh arhitektur, predstavil specifike in na kratko opisal, kako je vsaka arhitektura oblikovana, oziroma v čem sta si različni. Po predstavitvi obeh arhitektur sem se osredotočil na analizo teorije prehoda na mikrostoritveno arhitekturo. Opisal sem različne tehnike oziroma pristope pri prehodu na mikrostoritveno arhitekturo in opisal različne tehnike razgradnje obstoječe monolitne programske opreme. V naslednjem delu sem pridobljeno

teoretsko znanje združil z empiričnim delom, kjer me je zanimalo, kakšno je stanje programske arhitekture v Sloveniji.

V empiričnem delu me je predvsem zanimalo, kakšen delež slovenskih podjetij uporablja mikrostoritveno programsko opremo, oziroma vsaj razmišlja o njeni uporabi pri svojem poslovanju. Zanimalo me je tudi, kakšni so razlogi oziroma dejavniki, na podlagi katerih se podjetja odločijo za uporabo mikrostoritvene arhitekture. Za namen raziskave sem udeležence v anketi razdelil v tri skupine: udeleženci, ki uporabljajo monolitno programsko arhitekturo; udeleženci, ki uporabljajo programsko opremo, ki je bila že od začetka zgrajena na mikrostoritveni arhitekturi; in udeleženci, ki uporabljajo mikrostoritveno programsko opremo, ki je bila deležna prehoda z monolitne programske arhitekture. Na podlagi odgovorov anketirancev sem primerjal, kako so zadovoljni z različnimi lastnostmi trenutne programske opreme. Zanimalo me je tudi, ali obstajajo razlike pri uporabi lastnih strežnikov in strežnikov v oblaku glede na različen tip programske opreme, tj. ali se pri določenem tipu programske arhitekture bolj uporablja lastni strežnik in pri določenem bolj strežniki v oblaku.

Na podlagi testiranja hipotez in izdelanih analiz sem prišel do ugotovitev, da v Sloveniji še vedno prevladuje monolitna programska arhitektura. Vzroki za to so različni. Veliko anketirancev je mnenja, da jim trenutna monolitna arhitektura zadostuje, veliko pa je takih, ki so za to arhitekturo slišali prvič. Nekaj je tudi takih, ki trenutno uporabljajo monolitno arhitekturo, vendar v prihodnosti načrtujejo uporabo mikrostoritvene arhitekture. Podjetja, ki uporabljajo monolitno arhitekturo, imajo največ težav s počasnim razvojem novih funkcionalnosti. Poleg počasnega razvoja so podjetja velikokrat zaznala nefleksibilnost pri uporabi različnih tehnologij v svoji programski opremi bodisi s strani programskih jezikov ali nameščanja programske opreme.

Podjetja, ki so se odločila za uporabo mikrostoritvene arhitekture imajo različne razloge za njeno uporabo. Najpogostejši razlog za uporabo sovpada s slabostjo monolitne programske opreme, in sicer je to hiter razvoj novih funkcionalnosti, kar je razlog pri kar 88 odstotkih anketirancev, ki uporabljajo mikrostoritveno arhitekturo. Med pogostimi razlogi za uporabo te arhitekture pa sta tudi možnost boljše razširljivosti programske opreme in možnost uporabe več programskih jezikov.

S testiranjem hipotez sem ugotovil, da podjetja statistično značilno zaznavajo boljšo učinkovitost mikrostoritvene programske opreme pri hitrem vzdrževanju programske opreme in boljše možnosti razširjanja programske opreme; ne moremo pa trditi, da podjetja, ki uporabljajo mikrostoritveno arhitekturo, občutijo manjšo arhitekturno kompleksnost, saj to pri statističnem testu ni bilo značilno. S statističnim testom sem tudi ugotovil, da podjetja, ki uporabljajo mikrostoritveno arhitekturo, pogosteje uporabljajo oblačne storitve za nameščanje programske opreme kot podjetja, ki uporabljajo monolitno arhitekturo. Poleg pogostejše uporabe oblačnih ponudnikov storitev podjetja, ki uporabljajo mikrostoritveno arhitekturo, pogosteje posegajo po več različnih programskih jezikih v posamezni programski opremi.

LITERATURA IN VIRI

1. Ashan, F. (2019). *The Dilemma of code reuse in microservices* [objava na blogu]. Pridobljeno 11. aprila 2021 iz https://blog.bitsrc.io/the-dilemma-of-code-reuse-in-microservices-a925ff2b9981
2. Ayay, K. (2020). *Aspect-Oriented Modeling of Technology Heterogeneity in Microservice Architecture* [objava na blogu]. Pridobljeno 18. aprila 2021 iz https://www.devopsschool.com/blog/aspect-oriented-modeling-of-technology-heterogeneity-in-microservice-architecture/
3. Baškarada, S., Nguyen, V. & Koronios, A. (2018). Architecting Microservices: Practical Opportunities and Challenges. *Journal of Computer information Systems, 60*(5), 1-9.
4. Bogner, J. & Zimmermann, A. (2016). Towards Integrating Microservices with adaptable enterprise architecture. *IEEE 20th International Enterprise Distributed Object Computing Workshop* (str. 158-162). Dunaj: IEEE.
5. Bonham, A. (2017, 23. januar). *Microservice – When to React Vs. Orchestrate* [objava na blogu]. Pridobljeno 25. decembra 2020 iz https://medium.com/capital-one-tech/microservices-when-to-react-vs-orchestrate-c6b18308a14c#:~:text=Orchestration%20is%20the%20traditional%20way,%2DOriented%20Architecture%20(SOA).&text=For%20example%2C%20if%20three%20services,response%20before%20calling%20the%20next
6. Brown, K. (2017, 13. februar). IBM. *Apply the Strangler Application Pattern to microservices applications*. Pridobljeno 11. maja 2021 iz https://developer.ibm.com/technologies/microservices/articles/cl-strangler-application-pattern-microservices-apps-trs/
7. But, C. (2019, 23. maj) *Scaling a Monolith – Vertical Scaling & Horizontal Scaling simply defned* [objava na blogu]. Pridobljeno 11. aprila 2021 iz https://colin-but.medium.com/scaling-a-monolith-vertical-scaling-horizontal-scaling-simply-defined-4337c8a07326
8. Chan, M. (2017, 13. december). *Microservices vs. Monoliths: What's the Right Architecture for your Software?* [objava na blogu]. Pridobljeno 29. februarja 2021 iz https://www.thorntech.com/2017/12/microservices-vs-monoliths-whats-right-architecture-software/
9. Cser, T. (2018, 23. julij). *How to Test Microservices* [objava na blogu]. Pridobljeno 18. aprila 2021 iz https://www.functionize.com/blog/how-to-test-microservices/
10. Diguer, S. (2020, 1. junij). *Microservices Advantages and Disadvantages: Everything You Need to Know* [objava na blogu]. Pridobljeno 1. decembra 2020 iz https://solace.com/blog/microservices-advantages-and-disadvantages/
11. Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2017). Microservices: yesterday, today and tomorrow. V Mazzara M. & Meyer B. (ur.), *Present and Ulterior Software Engineering* (str. 195-216). Cham: Springer.
12. Evans, E. (2003). *Domain-Driven Desing: Tackling Complexity in the Heart of Software* (1. izd.). ZDA: Pearson Education.

13. Fachat, A. (2019, 30. januar). IBM. *Challenges and benefits of the microservice architectural style, Part 1*. Pridobljeno 20. decembra 2020 iz https://developer.ibm.com/depmodels/microservices/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-1/
14. Fernando, A. (2018, 18. januar). *Can We Reuse Code Between Microservices* [objava na blogu]. Pridobljeno 18. aprila 2021 iz https://codeburst.io/can-we-reuse-code-between-microservices-508fa4c1d06d
15. Fowler, M. (2014, 15. januar). *BoundedContext*. Pridobljeno 20. aprila 2021 iz https://martinfowler.com/bliki/BoundedContext.html
16. Fowler, M. (2015, 3. junij). *MonolithFirst*. Pridobljeno 6. aprila 2021 iz https://martinfowler.com/bliki/MonolithFirst.html
17. Fowler, M. & Lewis, J. (2014, 25. marec). *Microservices a definition of the new architectural term*. Pridobljeno 11. aprila 2021 iz https://martinfowler.com/articles/microservices.html
18. Fritzsch, J., Bogner, J., Wagner, S. & Zimmermann, A. (2019). Microservices Migration in Industry: Intentions, Strategies, and Challenges. *IEEE International Conference on Software Maintenance and Evolution* (str. 481–490). Cleveland: IEEE.
19. Frye, B. (2020, 28. september). *8 Steps for Migrating Existing Applications to Microservices* [objava na blogu]. Pridobljeno 28. novembra 2021 iz https://insights.sei.cmu.edu/sei_blog/2020/09/8-steps-for-migrating-existing-applications-to-microservices.html
20. Gnatyk, R. (2018, 3. oktober). *Microservices vs. Monolith: which architecture is the best choice for your business?* Pridobljeno 9. aprila 2021 iz https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/
21. Haq, S. (2018, 2. maj). *Introduction to Monolithic Architecture and MicroServices Architecture* [objava na blogu]. Pridobljeno 29. februarja 2021 iz https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63
22. Harrison, R. (2013). *Strategic Thinking in 3D: A Guide for National security, foreign policy and business professionals* (1. izd.). Dulles: Potomac Books
23. IBM Cloud Education. (2019). *Microservices*. Pridobljeno 20. decembra 2020 iz https://www.ibm.com/cloud/learn/microservices
24. Jaroslaw, K. (2016). *Developing with Docker* (1. izd.). Birmingham: Packt Publishing.
25. Jeremy, H. (2020). *7 Key Benefits of Microservices* [objava na blogu]. Pridobljeno 18. aprila 2021 iz https://blog.dreamfactory.com/7-key-benefits-of-microservices/
26. Jurič, M. (2017, 13. oktober). *Zakaj je prehod na arhitekturo mikrostoritev, API-je in DevOps tako pomemben in kaj prinaša*. Pridobljeno 9. marca 2021 iz https://www.src.si/revija/zakaj-je-prehod-na-arhitekturo-mikrostoritev-api-je-in-devops-tako-pomemben-in-kaj-prinasa/
27. Karwatka, P. (2020, 14. januar). *Monolithich architecture vs microservices* [objava na blogu]. Pridobljeno 11. aprila 2021 iz https://divante.com/blog/monolithic-architecture-vs-microservices/

28. Karwatka, P., Gil, M., Grabowski, M., Graf, A., Jedrzejewski, P., Kurzeja, M., Orfin, A. & Picho, B. (brez datuma). *Microservices Architecture for eCommerce*. Wroclaw: Divante.
29. Kurmi, A. (2020, 23. januar). *Top 10 Microservices frameworks for 2020* [objava na blogu]. Pridobljeno 26. junija 2020 iz https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2
30. Lanese, I., Dragoni, N., Mazara, M., Safina, L, Larsen, S. & Mustafin, R. (2017, februar). Microservices: How To Make Your Application Scale. *Perspectives of System informatics 11th International Andrei P. Ershov Informatics Conference*. Moskva, Rusija.
31. Laskey, K. B. & Laskey, K. (2009, julij). Service oriented architecture. *Wiley Interdisciplinary Reviews: Computational Statistics, 1*(1), 101-105.
32. Lenarduzzi, V., Lomio, F., Saarimaki, N. & Taibi, D. (2020). Does migrating a monolithic system to miroservices decrease the technical debt? *Journal of Systems and Software*, *169*, 1-16.
33. Li, C. Y., Ma, S. P. & Lu, T. W. (2020). Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System. *2020 International Computer Symposium* (str. 519–524). Taiwan: IEEE.
34. Li, S., Zhang, H., Jia, Z., Li, Z., Zhang, C., Li, J., Gao, Q., Ge, J. & Shan, Z. (2019, november). A dataflow-driven approach to identifying microservices from monolithic applications. *The Journal of Systems and Software 157*, 1-16.
35. Lu, C. Z., Zeng, G. S. & Xie, Y. J. (2020). Biograph specification of software architecture and evolution analysis in mobile computing environment. *Future Generation Computer Systems, 108*, 662-676.
36. Lu, N., Glatz, G. & Peuser, D. (2019). Moving mountains – practical approaches for moving monolithic applications to Microservices. *International Conference on Microservices*. Dortmund.
37. Luthra, R. (brez datuma). *Development of a Language – Independent Microservice Architecture*. Pridobljeno 22. decembra 2020 iz https://cloudsek.com/development-of-a-language-independent-microservice-architecture/
38. Megargel, A. & Shankararaman, V. (2020). Migrating from monoliths to cloud-based microservices: A banking industry example. *Software Engineering in the Era of Cloud Computing*, 85–108. Cham: Springer.
39. Messina, A., Rizzo, R., Storniolo, P. & Tripiciano, M. (2016). The Database-is-the-Service Pattern for Microservice Architectures. *Conference: International Conference on information technology in bio.and medical infromatics* (str. 223-233). Cham: Springer.
40. Microsoft Azure. (brez datuma). *What is a virtual machine?* Pridobljeno 26. decembra 2020 iz https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/#overview
41. Monteiro, D., Maia, P. H. M., Rocha, L. S. & Mendonca, N. C. (2020). Building orchestrated microservice systems using declerative business processes. *Service Oriented Computing and applications*, *14*(4), 243–268.
42. Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems* (1. izd.). Newton: O'Reilly Media.

43. Patel, Y. (2018, 12. april). *Microservice Architecture Benefits And Its Business Value* [objava na blogu]. Pridobljeno 9. marca 2021 iz https://www.thesunflowerlab.com/blog/microservice-architecture-benefits-business-value/
44. Qin, Z., Zheng, X. & Xing, J. (2008). *Software architecture* (1. izd.). Berlin Heidelberg: Springer.
45. Raj, P., Chelladhurai, J. S. & Singh, V. (2015). *Learning Docker*. Birmingham: Packt Publishing Ltd.
46. Rajesh, R. V. (2016). *Spring Microservices*. Birmingham: Packt Publishing Ltd.
47. Red Hat, Inc. (2020, 9. marec). *What is an application architecture?*. Pridobljeno 6. junija 2020 iz https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture
48. Rehman, K. & Ashraf, M. (2019). A Comparative Analysis of Distributed and Parallel Computing. *VFAST Transactions on Software Engineering*, 2018, 60-67.
49. Rengaiah, P. (2014, 24. februar). *On Modular Architectures* [objava na blogu]. Pridobljeno 27. maja 2021 iz https://medium.com/on-software-architecture/on-modular-architectures-53ec61f88ff4
50. Richardson, C. (brez datuma). *What are microservices?*. Pridobljeno 11. aprila 2021 iz https://microservices.io/index.html
51. RubyGarage. (2019, 18. april). *Best Architecture for an MVP: Monolith, SOA, Microservices or serverless* [objava na blogu]. Pridobljeno 4. aprila 2021 iz https://rubygarage.org/blog/monolith-soa-microservices-serverless
52. Rud, A. (2019, 24. julij). *Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience* [objava na blogu]. Pridobljeno 3. maja 2021 iz https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/
53. Schabowsky, J. (2019, 26. november). *Microservices Choreography vs. Orchestration: The Benefits of Choreography* [objava na blogu]. Pridobljeno 20. aprila 2021 iz https://solace.com/blog/microservices-choreography-vs-orchestration/
54. Singh, N. (2021, 6. avgust). *Microservices with Orchestration and Choreography* [objava na blogu]. Pridobljeno 27. septembra iz https://medium.com/walmartglobaltech/microservices-with-orchestration-and-choreography-d09941b1b4e6
55. Soldani, J., Tamburri, D. A. & Van Den Heuvel, W. (2018). The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, *146*, 215–232.
56. Soni, R. K., Ganeshan, A. & Rajesh, R.V. (2017). *Spring: Developing Java Applications for the Enterprise*. Birmingham: Packt Publishing Ltd.
57. Sumerge (2019, 15. julij). *What is Microservices Architecture*. Pridobljeno 26. junija 2020 iz https://sumerge.com/what-is-microservices-architecture/
58. Sviatoslav, A. & Sergey, C. (2020, 9. januar). *Advantages of using Docker for microservices* [objava na blogu]. Pridobljeno 26. decembra 2020 iz https://rubygarage.org/blog/advantages-of-using-docker-for-microservices

59. Taibi, D., Lenarduzzi, V. & Pahl, C (2017). Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing, 4*(5), 22-32.
60. Tang, A. & Lau, M. F. (2014). Software architecture review by association. *Journal of Systems and Software, 88*, 87–101.
61. Thakur, V. (2008). *ASP.NET 3.5 Application architecture and design*. Birmingham: Packt publishing Ltd.
62. Valderas, P., Torres, V. & Pelechano, V. (2020). A Microservice Composition Approach based on the Choreography of BPMN fragments. *Information and Software Technology, 127*, 106370.
63. Viggiato, M., Terra, R., Rocha, H., Valente, M. & Figueiredo, E. (2018). Microservices in Practice: A Survey Study. *VI Workshop on Software Visualization, Evolution and Maintenance*, 1-8.
64. Weinreich, R. & Groher, I. (2016, december). Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review. *Information and Software Technology 80*, 265-286.
65. Wittmer, P. (2020). *MICROSERVICES DISADVANTAGES & ADVANTAGES*. Pridobljeno 20. februarja 2021 iz https://www.3pillarglobal.com/insights/disadvantages-of-a-microservices-architecture/