VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Baigiamasis bakalauro darbas

Mikroservisų sistemos vertinimas

(Microservice system evaluation)

Atliko: 4 kurso 2 grupės studentas
Džiugas Serbenta (Parašas)

Darbo vadovas: Partn. Doc.
Liutauras Ričkus (Parašas)

Recenzentas: Doc. Dr.
Antanas Mitašiūnas (Parašas)

Vilnius
2019

TURINYS

Įvadas

Šiais laikais mikroservisų architektūra yra populiarus būdas kurti įvairias sistemas, kadangi remiantis šia architektūra sukurtos sistemos turi įvairių privalumų, lyginant su monolitine architektūra – įprastuoju sistemų kūrimo būdu. Kai kuriems projektams šios architektūros pritaikymas yra praktiškai būtinas, kadangi jos suteikiami privalumai suteikia jiems labai didelės naudos, be kurios projekto vykdymas kainuotų daug kartų daugiau pastangų, laiko ir pinigų, o, kadangi dažnai projektas, kuris, įvertinus, per daug kainuoja yra tiesiog nevykdomas, gali būti taip, kad daug projektų, kurie galėtų būti sėkmingai vykdomi remiantis mikroservisų architektūra, įprastais programavimo metodais yra tiesiog nekuriami. Taigi šis programų kūrimo būdas tikrai yra vertingas, nes jis leidžia vykdyti tokius projektus, kurių be šios architektūros tiesiog nebūtų logiška vykdyti.

Kuriant sistemą remiantis mikroservisų architektūra, sistema yra išskaidoma į atskiras dalis, kurios, veikdamos kartu, atlieka sistemos reikalavimuose apibrėžtas užduotis. Šis apibrėžimas yra labai paprastas, tačiau gerai atspindi šio sistemų kūrimo būdo esmę. Iš tiesų, vienareikšmiškai teisingo ir pilno apibrėžimo mikroservisų architektūrai nėra, kadangi yra daug mikroservisų architektūros implementavimo būdų. Tačiau visus šiuos būdus vienija tai, kad sistemos veikimas, diegimas, kūrimas, ar visi šie dalykai yra kažkaip pagerinami, palengvinami.

Tačiau, kaip ir visa kita programų sistemų kūrime, mikroservisų architektūra tikrai nėra stebuklingas sprendimas, tinkantis visoms sistemoms. Jos pritaikymas turi daug sunkumų, kurie yra nesutinkami projektus kuriant kitomis architektūromis. Dažnai yra taip, kad logiškiausia projektą yra kurti kitokia architektūra, tarkim monolitine architektūra – ji yra dažnai lyginama su mikroservisų architektūra ir yra laikoma jos priešinga – mikroservisų architektūroje projektas yra išskaidomas į dalis, o monolitinėje architektūroje projektas yra kuriamas kaip vientisas daiktas – iš to ir kilo šių architektūrų pavadinimai.

Be to, kad kuriant projektą galima pritaikyti skirtingas architektūras, jeigu vis dėl to pasirenkama pritaikyti mikroservisų architektūrą, ją taip pat galima pritaikyti įvairiais būdais. Norint, kad kuriama sistema būtų kuo galima labiau efektyvi, reikia mikroservisų architektūrą jai pritaikyti pačiu efektyviausiu būdu. Tačiau ši užduotis nėra triviali, kadangi yra sunku įvertinti, kokiu būdu mikroservisų architektūrą projektui pritaikyti būtų geriausia, bei kiek sistema nukentėtų, jeigu jai būtų pritaikyta kitokia mikroservisų architektūros rūšis (atskyrimo būdas).

Šio darbo tikslas yra nustatyti, kiek skirtingas mikroservisų architektūros pritaikymas gali įtakoti sistemos kokybę. Tam tikslui bus įvykdyti šie uždaviniai:

- Pateikta teorija apie mikroservisų architektūrą ir jos privalumus, bei trūkumus. Pateiktas mikroservisų architektūros apibrėžimas. Taip pat pateiktas monolitinės architektūros apibrėžimas ir nusakyta, kokias monolitinės architektūros problemas sprendžia mikroservisų architektūra.
- Pateikti mikroservisų architektūros pritaikymo būdai.
- Išskirti sistemų vertinimo kriterijai.
- Sukurtos dvi panašios mikroservisų architektūra paremtos sistemos, besiskiriančios mikroservisų architektūros pritaikymo būdais.
- Atlikti bandymai, remiantis išskirtais kriterijais, abi sukurtos sistemos įvertintos.
- Apibendrinti bandymų rezultatai ir gautos išvados.

1. Mikroservisų architektūros pritaikymas

1.1. Mikroservisų architektūros apibrėžimai

Norint sužinoti, kaip mikroservisų architektūros pritaikymas įtakoja sistemą ir jos kūrimą įvairiais atvejais, reikia suprasti, kas yra mikroservisų architektūra, kuo ji skiriasi nuo įprastų sistemų kūrimo būdų, kaip ją galima implementuoti, bei kuo skiriasi kiekvienas implementavimo būdas. Tom Huston apibrėžia mikroservisų architektūrą kaip tokį būdą kurti programų sistemoms, kuriuo remiantis stengiamasi kurti vienos funkcijos modulius su gerai apibrėžtomis sąsajomis ir operacijomis[Hus19]. Jis pabrėžia mikroservisų naudą lyginant su monolitine architektūra.

1.2. Monolitinė architektūra

Tom Huston apibrėžia monolitinę architektūrą kaip tokią, kuri yra kuriama kaip vientisas, autonominis vienetas[Hus19]. Robert Annett pateikia šiek tiek platesnį šios architektūros apibrėžimą: jis sako, kad monolitas – tai toks architektūros stilius, arba programų sistemų kūrimo būdas, kuriuo remiantis sukurtos sistemos atitinka bent vieną iš šių trijų kategorijų[Ann14]:

- Modulio monolitą
- Diegimo monolitą
- Veikimo monolitą

Modulio monolito kategoriją atitinka tokios sistemos, kurias sudarantis kodas egzistuoja vienoje kodo bazėje – tai yra, kai kodas yra kompiliuojamas, galutinis jo kompiliavimo rezultatas yra vienas. Tai nebūtinai reiškia, kad tas rezultatas yra vienas failas – tai tiesiog reiškia, kad kompiliavimo meto kompiliatorius „mato“, arba kompiliuoja tokį kodą, kuris yra vientisas. Kad sistemos kodas atitiktų šią kategoriją, nėra būtina, kad visas kodas egzistuotų viename faile – kodas gali būti struktūrizuotas įprastai, naudojant įvairias struktūras, klases, paketus, bibliotekas ir panašiai. Tačiau visa tai kas yra kode galiausiai kompiliuojama kaip vienas daiktas. Lyginant su mikroservisų architektūra, joje gali būti taip, kad atskiras sistemos dalis reikia kompiliuoti atskirai, arba kurių kompiliavimo rezultatų yra daug – dažnai tiek, kiek yra tokios sistemos dalių.

Diegimo monolito kategoriją atitinka tokios sistemos, kurios yra diegiamos „visos iškart“. Tai reiškia, kad, kai sistema yra paruošta diegti, visose vietose, kur sistema yra diegiama, yra diegiama ta pati tos sistemos versija. Ši kategorija skiriasi nuo modulio monolito kategorijos tuo, kad ji neatsižvelgia į monolito kategorijos kriterijus – nesvarbu, ar sistema kompiliuojama į daug daiktų, ar ne. Svarbiausia yra tai, ar įmanoma sistemą diegti skirtingu metu, bei diegti skirtingas

jos versijas skirtingose vietose. Jeigu įmanoma, tai tokia sistema neatitinka diegimo monolito kategorijos. Tai reiškia, kad kai šią kategoriją atitinkančią sistemą reikia atnaujinti, tai reikia atlikti visą diegimo procesą iš naujo, nors gali būti, kad buvo atnaujinta tik maža sistemos dalis – reikia sustabdyti visą sistemą visose vietose, vėl įkelti naują sistemos versiją ir vėl paleisti. Tokios sistemos, kurios šios kategorijos neatitinka, galėtų būti diegiamos kiek kitaip, pavyzdžiui, atnaujinus vieną sistemos dalį, galima būtų per nauja diegti tik tą sistemos dalį atskirai, o ne visą sistemą kartu.

Vykdymo monolito kategoriją atitinka tokios sistemos, kurių apibrėžtas užduotis vykdymo metu atliks tik viena aplikacija, arba procesas. Šią kategoriją taip pat gali atitikti ir sistemos, kurios naudojasi kažkokiais išoriniais ištekliais, todėl tokios sistemos nebūtinai turi atitikti ir modulio monolito kategoriją. Taigi šios dvi kategorijos yra nepriklausomos. Lyginant su diegimo monolito kategorija, vykdymo monolitas dažniausiai bus ir diegimo monolitas. Vienas atvejis, kai vykdymo monolito kategoriją atitinkanti sistema neatitinka diegimo monolito kategorijos yra tada, jeigu diegiant sistemą, ji yra diegiama skirtinguose regionuose, skirtingiems vartotojams, arba diegiama skirtingu metu.

1.3. Mikroservisų architektūros skirtumai nuo monolitinės architektūros

Sistemos, kurios neatitinka bent vienos iš trijų minėtų kategorijų greičiausiai nėra monolitinės, o yra paremtos kažkokia kita architektūra. Žinoma, šios kategorijos yra sukurtos taip, kad būtų pabrėžtas skirtumas tarp monolitinės architektūros ir mikroservisų architektūros, todėl šiuo atveju greičiausiai tokios sistemos bus paremtos mikroservisų architektūra. Tačiau norint tikrai nuspręsti, ar sistema yra sukurta remiantis mikroservisų architektūra, reikia nustatyti, ar ji atitinka mikroservisų architektūros kriterijus.

Yra problema – mikroservisų architektūra neturi bendrai priimto apibrėžimo, kadangi egzistuoja daug jos versijų, kurios visos šiek tiek skiriasi viena nuo kitos. Chris Richardson mikroservisų architektūra pagrįstas sistemas apibrėžia kaip tokias, kurios:

- Yra sudarytos iš atskirų dalių
- Patogiai testuojamos ir palaikomos
- Viena nuo kitos menkai priklausomos
- Gali būti diegiamos atskirai
- Gali būti kuriamos mažų komandų

Bene svarbiausias požymis, kad sistema yra pagrįsta mikroservisų architektūra yra tai, kad ji yra sudaryta iš atskirų dalių. Veikdamos kartu šios dalys turi atlikti visas tokios sistemos reikalavimuose apibrėžtas užduotis. Toks yra paprasčiausias mikroservisų architektūros apibrėžimas. Dažnai įvairiuose šaltiniuose minima, kad atskiri mikroservisai privalo būti tvirtai apibrėžti ir būti atsakingi tik už vieną dalyką – tuo mikroservisų architektūra labiausiai ir skiriasi nuo monolitinės architektūros, kuria remiantis sistemą sudaro tik viena dalis, kuri yra atsakinga už absoliučiai viską, ką visa sistema privalo atlikti pagal reikalavimus.

1.4. Mikroservisų architektūros privalumai

Mikroservisų architektūra turi daug privalumų, kurių nesuteikia monolitinė architektūra. Anna Monus išskiria tokius šios architektūros pranašumus[Mon18]:

- Kiekvieną sistemos dalį galima kurti ir diegti atskirai, naudojant skirtingas komandas, programavimo kalbas, bei įrankius. Tai leidžia kiekvieno serviso komandoms būti mažiau priklausomoms viena nuo kitos, o tai joms suteikia daugiau lankstumo ir leidžia darbą vykdyti daug greičiau ir efektyviau.

- Sistemos, paremtos mikroservisų architektūra, yra geriau plečiamos. Norint plėsti tokias sistemas, pakanka plėsti tik tas sistemos dalis, kurių veikimas yra nepakankamas įvairiais aspektais. Norint plėsti sistemą, kuri yra sukurta remiantis monolitine architektūra, reikia plėsti visą sistemą.

- Atskiros sistemos dalys yra atsparesnės klaidoms. Kadangi atskiri servisai yra daug mažesni, negu visa sistema, juos yra patogiau palaikyti ir testuoti. Taip pat pasidaro aiškiau, kaip sistemoje gali įvykti klaidos ir į tai atsižvelgti kuriant servisą.

Chris Richardson išskiria dar vieną mikroservisų architektūros privalumą – sistema netampa taip stipriai priklausoma nuo kažkokios technologijos[Ric18]. Priklausomybė nuo kažkokios technologijos nėra geras dalykas dėl to, kad, jeigu ta technologija tampa nebepalaikoma, reikia perdaryti tas sistemos dalis, kurios ir naudojasi tos technologijos suteikiamomis paslaugomis. Žinoma, vis vien naudojamasi kažkokiomis išorinėmis technologijomis, kad palengvinti darbą, todėl dažniausiai nebūna taip, kad sistema yra visiškai nepriklausoma nuo nieko. Tačiau, kai yra daug mikroservisų, kurie visi yra kuriami atskirai (tačiau atitinkantys sistemos architektūrą), didžiausia dalis, kuri gali tapti priklausoma nuo kažkokios technologijos yra vienas mikroservisas.

1.5. Mikroservisų architektūros trūkumai

Vis dėl to, nors mikroservisų architektūra turi daug privalumų, ji taip pat turi ir daug trūkumų, antraip ji būtų vienintelis realus pasirinkimas kuriant bet kokią sistemą. Chris Richardson išskiria tokius mikroservisų architektūros trūkumus[Ric18]:

- Kūrimo sudėtingumas – išskirstytą sistemą dažniausiai yra sunkiau sukurti, nei monolitinę. Kuriant mikroservisais paremtą sistemą reikia apgalvoti daug daugiau dalykų. Programavimo aplinkos yra pritaikytos kurti monolitines aplikacijas, testavimas tampa sudėtingesnis, nes reikia testuoti kelis servisus vienu metu, reikia implementuoti komunikaciją tarp atskirų sistemos dalių, dažnai reikia glaudaus bendradarbiavimo tarp komandų, norint užtikrinti teisingą sistemos vystymą. Dėl to pradinė kūrimo stadija trunka ilgiau.

- Diegimo sudėtingumas – nors diegti atskirus servisus yra greičiau, nei monolitą, visą sistemą įdiegti tampa sunkiau, dėl didelio atskirų dalių skaičiaus – kai monolitinėje architektūroje užtenka įdiegti vieną (nors ir komplikuotą) bloką, mikroservisus diegti reikia atskirai. Taip pat tampa sudėtinga tokią sistemą ir valdyti, dėl sudėtingų sistemos dalių ryšių.

- Didesnis atminties naudojimas – mikroservisų architektūros sistemoje veikia daug izoliuotų procesų, kurių visuma įgyvendina apibrėžtas sistemos funkcijas. Kad izoliuoti šiuos procesus, juos visus reikia paleisti naudojant skirtingas virtualias (arba realias) aplinkas. Taigi, atminties sunaudojimas išauga dėl servisų izoliavimo būtinumo.

1.6. Mikroservisų architektūros pritaikymas

Žinant, kad mikroservisų architektūra turi nemažai trūkumų, pasidaro aišku, kad ją pritaikyti reikia tokiu būdu, kuris minimizuoja jos trūkumų sukeltus nuostolius projektui, bei maksimizuotų visus jos privalumus. Tai padaryti nėra trivialu, kadangi egzistuoja daug šios architektūros pritaikymo būdų. Kiekvienas būdas gali būti geriausias kažkokiai sistemai – visa tai priklauso nuo pačios sistemos ir jos poreikių.

Visų pirma, reikia išsiaiškinti, ar išvis yra prasminga sistemą kurti mikroservisų pagrindu. Dažnai, kai kuriama tik pirmoji programos versija, nėra žinoma, ar jai bus aktualios tos problemos, kurias sprendžia mikroservisų architektūra. Jei ne, tai ją taikyti nėra prasmės, nes tai tik apsunkins sistemos kūrimo procesą – sistemą taps sudėtinga kurti, sudėtinga diegti ir valdyti,

ji naudos daugiau atminties, galbūt net daugiau procesoriaus pajėgumo. Šio darbo autorius pabrėžia kelis atvejus, kai mikroservisų architektūrą projektui taikyti gali būti prasminga:

- Jeigu manoma, kad sistemos vartotojų skaičius, arba užduočių skaičius sistemos veikimo laikotarpyje augs, ypač, jeigu toks augimas gali būti eksponentinis.

- Jeigu planuojama, kad su projektu bus intensyviai dirbama po pradinės jo versijos sukūrimo. Jeigu bus intensyviai su sistema toliau dirbama ilgą laiko tarpą, diegiamos naujos funkcijos.

- Jeigu yra didelės projekto apimtys. Mikroservisų architektūra duoda daugiausiai naudos stambiems didelių įmonių projektams, todėl, jeigu programa yra kuriama vieno žmogaus, mikroservisų architektūra greičiausiai nėra geras pasirinkimas.

Jeigu vis dėl to yra nusprendžiama naudoti mikroservisų architektūrą, reikia apgalvoti, kokie mikroservisai sudarys kuriamą sistemą. Šio darbo autorius išskiria tokius sistemos dalių atskyrimo būdus:

- Atskyrimas pagal verslo funkcijas. Remiantis šia logika, mikroservisai yra atskiriami pagal tai, kaip veikia verslas, tarkime, jeigu turime prekybos tinklalapį, už pirkėjus gali būti atsakingas kažkoks pirkėjų mikroservisas, o už užsakymus – užsakymų mikroservisas. Galiausiai, projekto struktūra atitiks pačio verslo struktūrą.

- Atskyrimas pagal domenus, arba pagal duomenų sklidimo analizę. Šis atskyrimo būdas taip pat yra susijęs su verslo funkcijomis ir projekto struktūra atitinka verslo struktūrą.

- Atskyrimas pagal panaudojimą, arba veiksmus. Remdamiesi šiuo būdu, atskiriame servisus pagal jų aktyvias funkcijas, pavyzdžiui, elektroninio pašto projekte galima atskirti laiško išsiuntimo mikroservisą, kuris būtų atsakingas už galutinį laiškų išsiuntimą.

- Atskyrimas pagal resursus. Toks atskyrimo būdas apibrėžia servisus pagal tai, už kokius resursus ir su jais susijusius veiksmus jie yra atsakingi, tarkime, kurdami elektroninio pašto projektą, galėtume išskirti vartotojo paskyros mikroservisą, kuris apibendrintų vartotojo paskyros resursus ir veiksmus su jais.

Tada reikia apibrėžti, kaip atskiros sistemos dalys turėtų bendrauti. Šio darbo autorius išskiria tokius bendravimo principus:

- Agregacijos principas. Remiantis šiuo principu, egzistuoja vienas procesas, kuris kviečia ir naudoja visus kitus, o kiti procesai tarpusavyje visiškai nebendrauja – tik per

agregatorių. Jie gali turėti atskiras duomenų bazes vidinėms reikšmėms išsaugoti, tačiau skirtingų procesų duomenų bazės negali susisiekti – taip būtų sulaužytas agregacijos principas.

- Nuorodų principas. Šis principas panašus į pirmąjį, tačiau nėra tikros agregacijos – pirminis procesas veikia kaip delegatorius, kuris turi nuorodas į visus servisus. Kaip ir agregacijos principe, servisai tarpusavyje nebendrauja. Nors tikros agregacijos nėra, delegatorius vis dėl to gali atlikti kažkokius veiksmus su duomenimis prieš pateikdamas juos kviečiančiam servisui.

- Grandinės principas. Skirtingai nuo ankstesnių principų, remiantis šiuo principu sudaryta sistema veikia priešingai – visi servisai bendrauja tarpusavyje ir pirminiam procesui pasiekiamas tik vienas servisas. Jeigu kuriamas šiuo principu paremtas projektas, reikia atsižvelgti, kad grandinė nebūtų per ilga, nes kuo ilgesnė grandinė, tuo ilgiau pirminis procesas turi laukti atsakymo.

- Šakų principas. Apjungia agregatoriaus ir grandinės principus. Pirminis procesas gali kviesti tam tikrą skaičių kitų procesų, kurie yra grandinės – jie gali kviesti tik po vieną procesą, kurie gali kviesti po vieną ir taip toliau. Remiantis šiuo principu galima išvengti grandinės principo bėdos, kai dėl per didelio grandinės ilgio programos darbas lėtėja.

- Bendrų duomenų principas. Pagal šį principą, keli servisai gali dalintis duomenų baze. Taip galima paprasčiau pakeisti monolitine architektūra pagrįstą aplikaciją į mikroservisų architektūra pagrįstą, nes išvengiama duomenų pertekliaus ir netolygumų. Šį principą prasminga naudoti tuo atveju, jeigu tie servisai kažkaip susiję ir atskirų duomenų naudojimas stipriai pakenktų programos efektyvumui, arba atsirastų galimybė įsivelti klaidoms.

- Nesinchroniško bendravimo principas. Remiantis šiuo principu, kai kurie mikroservisai gali kviesti kitus servisus asinchroniškai. Taip servisai gali nesiblokuoti, kai yra naudojami, ir tai padeda kuriant sudėtingų struktūrų projektus. Taip galima sukurti sklandžią vartotojo sąsają, kuri veiktų sklandžiai ir greitai. Taip pat šis principas apsaugo projektą nuo galimų gedimų, jeigu dėl kažkokios priežasties kažkuri kita projekto dalis nustotų veikti, kadangi ji tiesiog nepateiktų neveikiančios dalies duomenų, o kiti duomenys toliau būtų pateikiami. Jeigu būtų bendraujama sinchroniškai, tokia neveikianti dalis galėtų sutrikdyti viso projekto veiklą.

- Mišrusis principas. Remiantis šiuo principu, servisai gali remtis keliais aukščiau išvardintais principais, siekiant įgyvendinti sudėtingesnį funkcionalumą.

Kadangi mikroservisų architektūroje sistema yra išskirstyta, norint pasiekti sistemos reikalavimuose apibrėžtą funkcionalumą, kažkur projekto struktūroje reikia vykdyti agregaciją, kuri apibendrintų informaciją ir pateiktų ją pirminiui, arba pirminiams sistemos procesams. Visi principai yra taikomi pagal projekto poreikius: jeigu kažkoks servisas yra atsakingas už kitų servisų duomenis, tai gal būt jis galėtų būti agregatorius, arba nuoroda; jeigu iš bendresnės sistemos dalies išskiriami kažkokie smulkesni servisai, gali būti, kad verta pritaikyti grandinės, arba šakų principą. Šio darbo autorius pabrėžia, kad bendrų duomenų principo derėtų vengti, kadangi jį naudojant atskiros sistemos dalys tampa labiau priklausomos viena nuo kitos, ko mikroservisų architektūra siekia ir išvengti. Jį geriausia būtų naudoti tuo atveju, kai yra pereinama iš monolitinės architektūros į mikroservisų architektūrą ir yra sunku atskirti duomenis. Nesinchroniško bendravimo principą verta įgyvendinti, jeigu yra norima išvengti neapibrėžto sistemos veikimo atidėjimo. Jeigu kažkurių mikroservisų duomenys yra svarbesni už kitus, verta įgyvendinti nesinchroniško bendravimo principą, kad nebūtų situacijos, kai laukiama ne tiek svarbių duomenų, nors vartotojui gali reikėti tik svarbiųjų.

2. Mikroservisų sistemos vertinimas

2.1. Vertinimo metodikos

Norint nustatyti, ar mikroservisų architektūra paremta sistema yra implementuota gerai, ją reikia vertinti atsižvelgiant į daug skirtingų veiksnių ir aplinkybių. Turint omenyje tai, kad kiekvienai sistemai mikroservisų architektūra gali būti pritaikoma skirtingai, galima sakyti, kad taip pat kiekviena sistema gali būti ir vertinama skirtingai, kadangi visoms sistemoms mikroservisų architektūra naudą suteikia skirtingai.

Tačiau kad ir kaip skirtųsi įvairių sistemų poreikiai, galima išskirti keletą metrikų, kurios gali būti taikomos beveik visoms mikroservisų architektūra pagrįstoms sistemoms. Šio darbo autorius išskiria tokius sistemos vertinimo būdus:

- Sistemos dydžio vertinimas – naudojantis įvairiomis metrikomis (Funkcinių taškų, kodo eilučių kiekio ir t.t.) galima įvertinti sistemos dydį. Norint, kad sistemą kurti būtų kuo lengviau, užtruktų mažiau laiko, bei kainuotų mažiau pinigų, dažniausiai yra geriausia mintis pasirinkti kuo mažesnę sistemą, kuri išpildo visus sistemos poreikiuose apibrėžtus reikalavimus.

- Sistemos efektyvumo vertinimas – tai vienas universaliausių bet kokios sistemos vertinimo būdų. Įvertinti, kaip greitai, kiek daug ir kaip gerai sistema atlieka jai skirtas užduotis galima, pavyzdžiui, testuojant įvairias jos dalis, arba visą sistemą kartu. Taip pat sistemos efektyvumą galima įvertinti ir netiesiogiai, žinant vien tik pačios sistemos architektūrą.

Be šių vertinimo būdų galima išskirti ir daug kitų būtent mikroservisų architektūrai vertinti skirtų vertinimo metodų, kurie nustato kaip gerai yra išnaudojami mikroservisų architektūros suteikiami privalumai, bei minimizuojami trūkumai, tačiau nėra tiksliai išmatuojami. Dažniausiai šie vertinimo kriterijai yra susiję su pačia sistemos architektūra (tai yra mikroservisų išsidėstymu, išskyrimu, apibrėžimu ir t.t.).

Paprastas būdas išskirti vertinimo kriterijus yra nustatymas, ar sistema turi dažnai mikroservisų architektūroje sutinkamų problemų. Tai nustačius galima iš dalies įvertinti, kaip gerai mikroservisų architektūra yra pritaikyta ir kaip gerai gali pasireikšti jos privalumai. Taip pat, žinoma, verta išskirti ir teigiamus sistemos aspektus, tačiau tokius rasti daug sunkiau, kadangi, gerai taikant mikroservisų architektūrą, dažniausiai ir tikimasi, kad sistema išnaudos visus šios architektūros suteikiamus privalumus. Taigi kol kas bus išskirta keletas dažnai mikroservisų architektūroje sutinkamų problemų, kurios gali pakenkti sistemos darbui ir vystymui. Mark Richards išskiria 7 tokias problemas[Ric16]:

2.1.1. Duomenų pirmumo migracijos metu problema

Ši problema pasireiškia, kai jau egzistuojanti monolitinė sistema yra skaidoma į atskirus mikroservisus ir yra norima išlaikyti sistemoje jau esamus duomenis. Ši problema pasireiškia tuo, kad duomenų migracijos metu dažnai susiduriama su klaidomis, be to mikroservisų architektūros pritaikymo procese gali atsitikti taip, kad servisus reikia sujungti, arba išskirstyti vėl, todėl duomenų migraciją reikia atlikti daug kartų. Mark Richards siūlo šią problemą spręsti duomenų migracijas atidedant iki vėlesnių sistemos vystymo etapų, kai tampa aišku (arba aiškiau), kad sistemos mikroservisų kiekis, kuris dirba su esamais duomenimis nesikeis.

2.1.2. „Timeout“ problema

Ši problema yra susijusi su *Timeout* funkcijos naudojimu servisų tarpusavio bendravime. Jeigu yra nustatomas komunikacijos nutraukimo laikas, yra nežinoma, kokios būsenos bus sistema, kadangi nėra sulaukiamas atsakas į žinutę, kuri galėjo pakeisti kažkokios duomenų bazės duomenis. Tuomet sistema gali bandyti vėl siųsti žinutę, kuri vėl pakeis duomenų bazę ir sistemoje įvyks klaida – bus neteisingai įrašyti duomenys duomenų bazėje. Geras būdas spręsti šią problemą yra pritaikyti „Circuit Breaker“ šabloną, kuris, įvykus kažkokiai klaidai ir servisui

neatsakant į užklausas, užtikrina, kad į tą servisą besikreipiantys kiti servisai žinotų, ar buvo įvykdytos jų užklausos.

2.1.3. Kodo dalijimosi problema.

Mikroservisų architektūroje norima, kad atskiri mikroservisai būtų kuo labiau atskirti vienas nuo kito. Tačiau dažnai pasitaiko, kad kažkokio funkcionalumo reikia keliems mikroservisams. Tai gali būti tokie dalykai, kaip apsaugos sistemos, įvairūs formatavimo algoritmai. Šioje vietoje galima susidurti su sunkumais, jeigu toks kodas, kurio reikia keliems servisams yra bendras tarp servisų – atlikus pakeitimus, kodas pasikeičia visų kitų mikroservisų kode. Tačiau toks kodo dalijimasis turi daug problemų – tampa sunku testuoti, bet kokie pakeitimai gali turėti nežinomų pasekmių kiekvienam bendru kodu besinaudojančiam mikroservisui. Yra keletas būdų spręsti šią problemą, paprasčiausias iš kurių yra tiesiog kodo kartojimas. Žinoma, toks sprendimas turi savų problemų, tačiau jis išvengia būtent šios problemos.

2.1.4. Per stambių arba per smulkių servisų atskyrimo problema

Ši problema paprasčiausiai nusako, kad kai kurie sistemos mikroservisai nėra teisingai atskirti ir yra arba per stambūs, arba per smulkūs. Servisų dydžiai įtakoja daug sistemos aspektų – efektyvumą, patikimumą, vystymą, testavimą, bei diegimą. Mark Richards pabrėžia, kad yra gana paprasta padaryti šią klaidą ir mikroservisus sukurti per smulkius. Vienas indikatorius, kad sistema yra išskirstyta per smulkiai yra tai, kad duomenų bazių operacijos pasidaro sudėtingos, palaikyti duomenų vientisumą pasidaro sudėtinga. Yra siūlomas sprendimas – nežinant, kokio dydžio mikroservisus išskirti, juos iš pradžių išskirti stambesnius, o po to vertinti, ar buvo teisingai pasielgta ir juos smulkinti, jeigu pasidaro aišku, kad jie yra per stambūs.

2.1.5. Nepagrįsto mikroservisų architektūros pritaikymo problema

Tai tokia problema, kai mikroservisų architektūra yra pritaikoma projektui neatsižvelgiant į sistemos reikmes, apimtį, kuriančios organizacijos išteklius ir kitus veiksnius. Ši problema pasireiškia visais anksčiau minėtais aspektais – efektyvumu, kūrimo, bei diegimo sudėtingumu, atminties naudojimu ir kitais mikroservisų architektūrai būdingais trūkumais. Jeigu nėra aišku, ar mikroservisų architektūra yra tinkamas sprendimas projektui, nebus aišku, ar verta dėl šios architektūros privalumų projektui sukelti problemų dėl jos trūkumų. Ši problema pasireiškia, jeigu sistemos planavimo metu nėra aiškiai įvertinamos mikroservisų architektūros pritaikymo pasekmės, arba jos yra įvertinamos neteisingai. Jos sprendimas yra paprastas – tereikia vėl įvertinti visus mikroservisų architektūros privalumus ir trūkumus ir nustatyti, ar verta keisti sistemą. Tačiau šią problemą reikia taisyti kuo anksčiau, antraip gali būti daug iššvaistyto darbo, kurio bus atsisakyta keičiant architektūras.

2.1.6. Statiško kontrakto problema

Ši problema yra susijusi su tuo, kaip kiekvienas mikroservisas sistemoje apibrėžia bendravimą su savimi. Šis apibrėžimas yra vadinamas kontraktu ir jis gali būti problemų šaltiniu, jeigu nėra apibrėžtas kažkoks kontrakto versijavimas. Jeigu kontraktas yra apibrėžiamas statiškai (be versijavimo) tampa sunku daryti jo pakeitimus, nes nuo to kontrakto gali priklausyti ne tik vienas mikroservisas, bet ir visi servisai, kurie juo naudojasi. Mikroservisų architektūroje priimta, kad atskiri mikroservisai turi būti kuo mažiau priklausomi vienas nuo kito, siekiant išvengti tokių problemų, kurios apsunkina pakeitimus, testavimą ir daugiau dalykų dėl to, kad yra kažkokios kitos sistemos dalys, kurios yra priklausomos nuo keičiamos sistemos dalies ir neaišku, kokios bus tokio pakeitimo pasekmės. Kalbant apie kontraktus, vis vien reikia, kad mikroservisai žinotų, kaip reikia bendrauti vienas su kitu, todėl šis priklausomumas yra būtinas. Kontraktų versijavimas minimizuoja šio priklausomumo pasekmes, nes jis leidžia atlikti pakeitimus tik atskiruose mikroservisuose, leidžia jiems palaikyti senos versijos kontraktus.

2.1.7. Bendravimo neįvertinimo problema

Ši problema pasireiškia sistemos efektyvumo sumažėjimu dėl to, kad dėl neįvertinus pasirinkto bendravimo tipo, kuris blogai atitinka atskiros sistemos dalies reikmes, pasirinkimo, bendravimas tarp mikroservisų tampa lėtesnis, negu galėtų būti. Dažniausias pasirinkimas bendravimui tarp mikroservisų yra REST protokolas. Tačiau yra ir daug kitų bendravimo protokolų – JMS, AMQP, MSMQ ir t. t. Sprendžiant šią problemą reikia atsižvelgti į kiekvieno bendravimo protokolo privalumus ir trūkumus, lyginant su vienas kitu, pavyzdžiui, REST protokolas yra plačiai žinomas ir taikomas, todėl juo bendrauti yra logiška tokiam servisui, kuris turi bendrauti su trečių asmenų sukurtomis programomis (tarkim kaip API), o AMQP protokolas turi asinchroninio bendravimo galimybių, todėl yra tinkamas, kai stengiamasi išvengti problemų, susijusių su *timeout* – komunikacijos nutraukimu dėl nenumatyto operacijos laiko padidėjimo.

2.2. Objektyvus ir subjektyvus sistemų vertinimas

Yra ir be galo daug problemų, kurios gali įtakoti mikroservisų architektūra pagrįstos sistemos veikimą, plėtojimą, bei kitas savybes. Tačiau šios problemos yra pakankamai dažnai sutinkamos [Ric16], kad, šio darbo autoriaus nuomone, jų išskyrimas yra pakankamas gana geros vertinimo sistemos sukūrimui. Įvertinant, kiek iš minėtų problemų turi sistema, galima spręsti, kaip gerai ji pritaiko mikroservisų architektūrą, bei kaip gerai tokia sistema yra sukurta apskritai (ypač remiantis „paprastaisiais“ įvertinimais – sistemos efektyvumu ir dydžiu). Vienintelis tokio sistemos vertinimo trūkumas yra, žinoma, kad daugelis šių dalykų yra subjektyvūs sistemos architektūros vertinimai. Kadangi egzistuoja daug mikroservisų architektūros apibrėžimų ir implementacijos būdų, objektyviai apibrėžti tai, kas sistemą daro prastesnę už kitas, yra labai

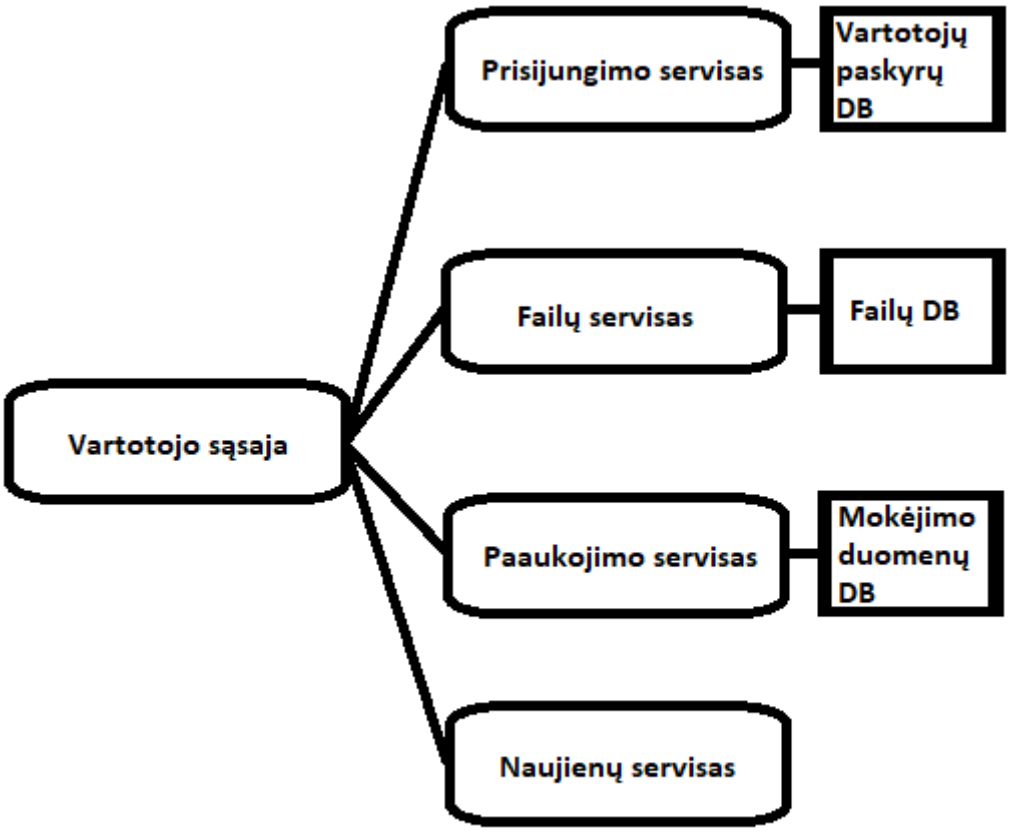
sunku ir dažniausiai tai būna vien tik tiksliai apibrėžiami vertinimai (efektyvumas ir dydis). Tačiau, autoriaus nuomone, vien šių įvertinimų nepakanka, nes tai tėra „paviršutiniškas“ vertinimas – nėra gilinamasi į sistemos struktūrą, kuri nulemia nemažą dalį sistemos aspektų, ne vien tik greitį ir dydį. Ši struktūra nulemia visus mikroservisų architektūros suteikiamus privalumus, tokius kaip plečiamumas, testavimo palengvinimas ir t. t. Sistemos architektūros kūrimas reikalauja ir kūrybingumo, todėl subjektyvus sistemos vertinimas nėra blogybė. Priešingai, šio darbo autoriaus nuomone, kiekviena komanda turi skirtingus gebėjimus, reikmes, todėl sistemą ir reikėtų vertinti atsižvelgiant į visus ją kuriančios organizacijos ypatumus. Tiesa, tai reiškia, kad išvardinti virtinimo kriterijai nebūtinai yra tinkami visais atvejais. Tačiau, šio darbo autoriaus nuomone, šios problemos gana stipriai įtakoja didžiąją dalį mikroservisų architektūra paremtų sistemų, todėl vertinti sistemos kokybę pagal jų buvimą ir nebuvimą atrodo logiška. Taip pat galima būtų išskirti ir daugiau problemų, tačiau šios problemos atspindi daugelį problemų, kurios atsitinka mikroservisų architektūros pritaikymo metu, todėl, šio darbo autoriaus nuomone, toks vertinimas yra pakankamas.

3. Realių sistemų vertinimas ir lyginimas

3.1. Pirmoji tinklalapio sistema

Siekiant pademonstruoti vertinimo sistemą, buvo sukurta paprasta mikroservisų sistema, susidedanti iš 5 dalių: Vartotojo sąsajos (tinklalapio programos), prisijungimo serviso, failų saugojimo serviso ir paaukojimo serviso. Pagal pavadinimus aišku, kad šie servisai yra atskirti pagal panaudojimo principą – visi jie yra atsakingi už kažkokį veiksmą. Vartotojo sąsaja šiuo atveju veikia kaip agregatorius, kuris visų servisų duomenis sukaupia ir atvaizduoja vartotojui. Tai yra Web aplikacija, taigi sistemą galima pasiekti per naršyklę. Pati sistema yra minimalistinė, sukurta vertinimo sistemos išbandymui. Ji buvo sukurta C# kalba, vartotojo sąsajai naudojant ASP.NET Core, o kitiems servisams – .NET Core. Bendravimas tarp servisų implementuotas tiesiogiai, naudojantis TCP protokolu. Sistemos veikimas vyksta taip: vartotojas, įvedęs sistemos adresą į naršyklės langą, mato prisijungimo langą, kuriame galima ir užsiregistruoti. Prisijungimo lange taip pat yra ir naujienų tekstas, kuris yra nustatomas naujienų servise. Šis tekstas yra saugomas vartotojo sąsajos servise, bet kas kažkiek laiko yra atnaujinamas, nusiunčiant užklausą į naujienų servisą. Prisijungiant ir registruojantis, sistema bando prisijungti prie prisijungimo serviso ir jo paklausti, ar toks vartotojas egzistuoja. Prisijungimo servisas, jeigu užklausa yra prisijungimo ir vartotojas egzistuoja, arba jeigu užklausa yra registruotis ir toks vartotojas neegzistuoja, grąžina teigiamą atsakymą. Tada vartotojo sąsaja vartotoją nukreipia į pagrindinį langą. Prisijungęs vartotojas turi galimybę pasirinkti įkelti failus, peržiūrėti

jau įkeltus failus, arba paaukoti projektui. Pasirinkęs įkelti failus, vartotojas nukreipiamas į langą, kuriame jis pasirenka failus ir tai patvirtina. Tada sistema bando susisiekti su failų servisu. Failų servisas iš vartotojo sąsajos failus gauna po vieną ir įkelia juos į duomenų bazę, apibrėžęs juos kaip vartotojo, kuris yra prisijungęs failus. Tada vartotojas yra nukreipiamas į įkeltų failų puslapį, kuriame turi būti atvaizduojamas sąrašas to vartotojo įkeltų failų. Šį sąrašą sistema taip pat gauna iš failų serviso, nurodydama tam tikro vartotojo failų sąrašo užklausą. Vartotojas pagrindiniame meniu taip pat gali pasirinkti paaukoti projektui. Tai padarius, vartotojas nukreipiamas į langą, kur prašoma įvesti mokėjimo duomenis. Dėl paprastumo, sistemą iš tikrųjų tų duomenų netikrina ir nepriima aukos, tik išsaugo juos duomenų bazėje. Duomenys į paaukojimo servisą nukeliauja per užklausas iš vartotojo sąsajos kontrolierių. Sistemos struktūra pavaizduota schemoje 1.



Schema 1 – pirmosios sistemos architektūros schema

3.2. Pirmosios sistemos vertinimas

3.2.1. Bendri įverčiai

Iš pradžių buvo įvertintas sistemos efektyvumas. Buvo ištestuoti visi servisai, išskyrus naujienų servisą, kuris veikia pasyviai visų kitų testų metu. Buvo atlikti tokie testai:

1. Pradinio puslapio įkėlimas ir prisijungimas

2. Pradinio puslapio įkėlimas, prisijungimas ir failų sąrašo peržiūrėjimas
3. Pradinio puslapio įkėlimas, prisijungimas ir paaukojimas

Kiekvienas testas buvo vykdomas 10 minučių, naudojant po 10 virtualių vartotojų.

Buvo gauti tokie pirmojo testo rezultatai:

- Vidutinis testo atsako greitis buvo 213 ms
- Nebuvo nė vieno testo, kuris grąžino klaidą
- Iš viso testas įvykdytas 3531 kartą
- Vidutinis užklausų greitis – 6 užklausos per sekundę
- Didžiausia užfiksuota atsako trukmė – 515 ms

Antrojo testo rezultatai buvo tokie:

- Vidutinis testo atsako greitis buvo 180 ms
- Nebuvo nė vieno testo, kuris grąžino klaidą
- Iš viso testas įvykdytas 3055 kartą
- Vidutinis užklausų greitis – 4 užklausos per sekundę
- Didžiausia užfiksuota atsako trukmė – 433 ms

Trečiojo testo rezultatai buvo tokie:

- Vidutinis testo atsako greitis buvo 169 ms
- Nebuvo nė vieno testo, kuris grąžino klaidą
- Iš viso testas įvykdytas 1873 kartą
- Vidutinis užklausų greitis – 4 užklausos per sekundę
- Didžiausia užfiksuota atsako trukmė – 287 ms

Sistemos kodą sudaro 3354 kodo eilutės.

3.2.2. Dažnųjų klaidų paieška

Be šių testų, reikia patikrinti, ar sistema neturi ankstesniame skyriuje išskirtų architektūros klaidų.

Iš pradžių bus patikrinta, ar sistema turi duomenų pirmumo migracijos problemą. Kadangi ši sistema yra sukurta iš naudojantis mikroservisų architektūra nuo pat pradžių, o ne pakeista į mikroservisų architektūrą iš kitos architektūros, nėra jokių duomenų, kuriuos reikia migruoti. Tai reiškia, kad ši sistema tokios problemos neturi.

Toliau bus patikrinta, ar sistema turi *timeout* problemą. Ši sistema turi šią problemą, tačiau tai gali atsitikti tik tuo atveju, jeigu įvyksta klaida siunčiant paskutinį patvirtinimo pranešimą failo persiuntimo procese. Šansas pasiekti šią situaciją yra mažas, kadangi pačio failo siuntimas daug daugiau kartų naudoja siuntimo ir skaitymo operacijų, todėl yra mažai tikėtina, kad, jeigu yra problemų su failų servisu, arba su tinklu, šios problemos nebus pasireiškusios anksčiau failų siuntimo operacijos metu. Tačiau šios tikimybės vis vien negalima atmesti. Ši problema taip pat gali pasireikšti ir paaukojimo mikroservisui, nes jis taip pat valdo duomenų bazę. Šiuo atveju ji gali pasireikšti lygiai taip pat – jeigu nėra sulaukiama patvirtinimo žinutės, kad duomenys gauti sėkmingai. Šią problemą galima būtų išspręsti pasinaudojant *circuit-breaker* šablonu, kuris užtikrintų, kad duomenų bazėse duomenys nebūtų patalpinti keletą kartų.

Kodo dalijimasis sistemoje egzistuoja kaip *string* tipo kintamųjų formatavimas, bei kelios klasės, praplečiančios standartinių klasių funkcionalumą. Šioje sistemoje toks kodas yra tiesiog kopijuojamas – nėra bendros kodo bazės, kurią pakeitus, pasikeičia visuose servisuose esantis kodas, todėl šios problemos ši sistema išvengia.

Patikrinti, ar sistema turi per smulkių, arba per smulkių mikroservisų šiuo atveju yra gana lengva, kadangi pati sistema yra gana paprasta. Visos duomenų bazių operacijos yra labai paprastos, todėl atrodo, kad nėra per stambių mikroservisų. Nusakyti, ar yra per smulkių mikroservisų yra kiek sunkiau, tačiau, kadangi sistema yra sukurta pagal apibrėžtą procesą, naudojantis atskyrimu pagal panaudojimo principą, bei dėl to, kad nėra perteklinio bendravimo tarp mikroservisų vykdant paprastas sistemos operacijas, galima sakyti, kad mikroservisai nėra išskaidyti per smulkiai.

Mikroservisų architektūros pritaikymas šiai sistemai yra pagrįstas, kadangi ši sistema ir buvo sukurta mikroservisų architektūros vertinimui. Jeigu ignoruotume tą aspektą, ši sistema neturėtų jokios priežasties būti pagrįsta mikroservisais – ją kuria vienas žmogus – šio darbo autorius, jos funkcijos yra paprastos, sistemos nėra planuojama plėsti. Mikroservisų architektūros trūkumai šiuo atveju tikrai padaro daugiau žalos, nei naudos duoda privalumai. Tačiau jeigu neatmetame,

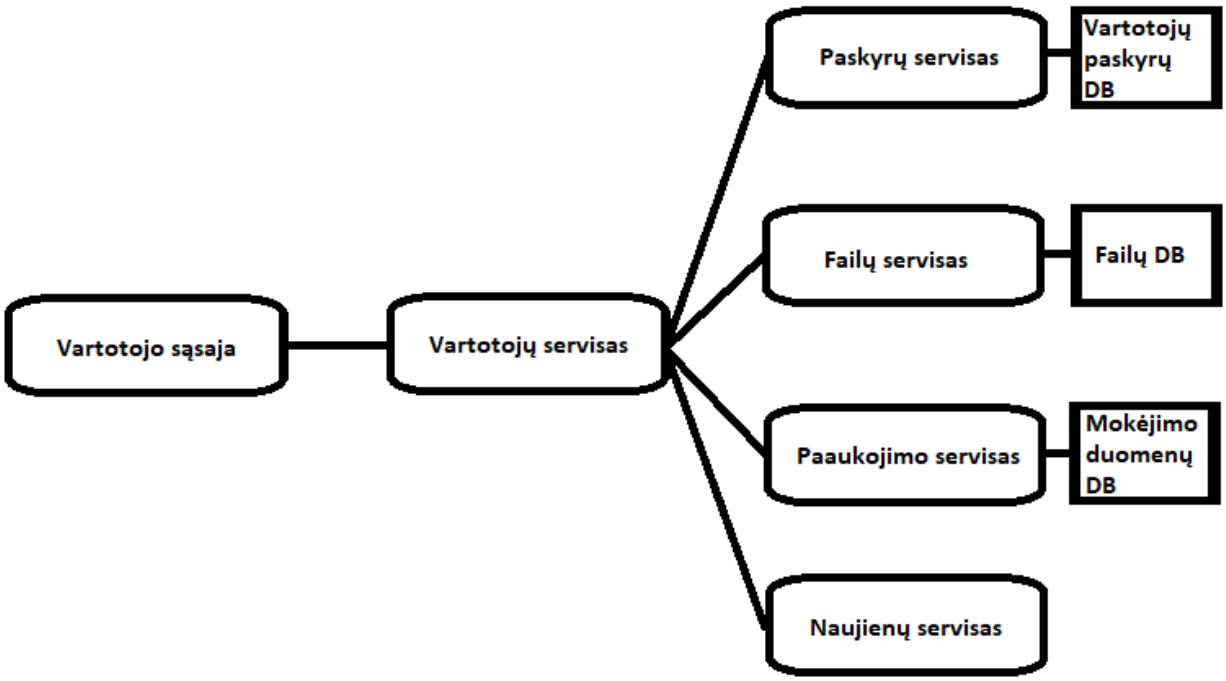
kad šis bakalaurinis darbas yra pakankama priežastis šios sistemos sukūrimui naudojantis mikroservisų architektūra, tai galima sakyti, kad šios architektūros pritaikymas šiuo atveju yra pagrįstas.

Šioje sistemoje kontraktas yra apibrėžtas statiškai – nėra versijavimo. Tai reiškia, kad kontrakto pakeitimus daryti yra sunku, kadangi nėra įmanoma palaikyti senų kontraktų versijų, kol nėra pilnai palaikomi naujieji kontraktai.

Bendravimas tarp servisų šioje sistemoje yra pasiekiamas tiesiogiai per TCP protokolą. Nėra naudojamas nei REST, nei kiti populiarūs mikroservisų bendravimo protokolai, todėl tai reiškia, kad ateityje būtų sunku plėsti šią sistemą, nes bendravimas tarp servisų yra sunkiai suprantamas. Tačiau, kadangi neplanuojama, kad su šia sistema ateityje bus dirbama, ar ji plečiama, tai nėra aktualu. Deja, kiti bendravimo būdai nebuvo testuoti, todėl nėra visiškai aišku, ar kiti bendravimo būdai būtų efektyvesni. Todėl galima sakyti, kad bendravimo būdai nėra įvertinti pakankamai gerai.

3.3. Antroji tinklalapio sistema

Taip pat buvo sukurta ir antra tinklalapio sistema. Ši sistema yra gana panaši į pirmąją sistemą, tačiau ji susideda iš 6 dalių: tinklalapio programos, vartotojų serviso, naujienų serviso, paskyrų serviso, failų serviso ir paaukojimo serviso. Šiuo atveju sistema yra išskirta pagal resursus – tinklalapio programa agreguoja du mikroservisus – naujienų ir vartotojų. Vartotojų servisas agreguoja likusius servisus, nes visiems likusiems servisams reikia vartotojo duomenų, norint užbaigti operacijas. Kaip ir pirmojoje sistemoje, ši sistema buvo parašyta naudojant C# kalbą, tinklalapio programa parašyta naudojant ASP.NET Core, o kiti mikroservisai – .NET Core. Kaip ir pirmojoje sistemoje, bendravimas tarp mikroservisų yra implementuotas tiesiogiai TCP protokolu. Prisijungus prie sistemos tinklalapio, viskas vizualiai atrodo lygiai taip pat, kaip ir pirmojoje sistemoje – pradiniame lange yra prisijungimo ir prisiregistravimo laukai, bei iš naujienų serviso gaunamas tekstas. Ši sistema skiriasi nuo pirmosios tuo, kad yra naujas mikroservisas – vartotojų servisas, kuris agreguoja prisijungimo, failų ir paaukojimo servisus. Tai reiškia, kad, norėdama pasiekti šiuos servisus, tinklalapio programa privalo kreiptis ne tiesiai į juos, o į vartotojų servisą, kuris priima visų trijų agreguojamų servisų užklausas ir jas nukreipia į atitinkamus servisus. Visais kitais atvejais sistema veikia taip pat, kaip ir pirmoji – tik operacijos, reikalaujančios funkcionalumo iš paskyrų, failų ir paaukojimo servisų privalo būti atliekamos per vartotojų servisą. Sistemos struktūra pavaizduota schemoje 2.



Schema 2 – antrosios sistemos architektūros schema

3.4. Antrosios sistemos vertinimas

3.4.1. Bendri įverčiai

Iš pradžių bus įvertintas sistemos efektyvumas. Kaip ir pirmosios sistemos atveju, buvo ištestuoti visi servisai, išskyrus naujienų servisą, kuris veikia pasyviai visų kitų testų metu. Vėlgi, buvo atlikti tokie testai:

1. Pradinio puslapio įkėlimas ir prisijungimas
2. Pradinio puslapio įkėlimas, prisijungimas ir failų sąrašo peržiūrėjimas
3. Pradinio puslapio įkėlimas, prisijungimas ir paaukojimas

Kiekvienas testas buvo vykdomas 10 minučių, naudojant po 10 virtualių vartotojų.

Buvo gauti tokie pirmojo testo rezultatai:

- Vidutinis testo atsako greitis buvo 788 ms
- Testavimo metu vieną kartą buvo grąžinta klaida
- Iš viso testas įvykdytas 2546 kartą
- Vidutinis užklausų greitis – 5 užklausos per sekundę

- Didžiausia užfiksuota atsako trukmė – 1.6 s

Antrojo testo rezultatai buvo tokie:

- Vidutinis testo atsako greitis buvo 275 ms
- Nebuvo nė vieno testo, kuris grąžino klaidą
- Iš viso testas įvykdytas 2202 kartą
- Vidutinis užklausų greitis – 4 užklausos per sekundę
- Didžiausia užfiksuota atsako trukmė – 446 ms

Trečiojo testo rezultatai buvo tokie:

- Vidutinis testo atsako greitis buvo 444 ms
- Nebuvo nė vieno testo, kuris grąžino klaidą
- Iš viso testas įvykdytas 1723 kartą
- Vidutinis užklausų greitis – 3 užklausos per sekundę
- Didžiausia užfiksuota atsako trukmė – 1.19 s

Sistemos kodą sudaro 3928 kodo eilutės.

3.4.2. Dažnųjų klaidų paieška

Iš pradžių bus patikrinta, ar sistema turi duomenų pirmumo migracijos problemą. Ši sistema, kaip ir pirmoji testuojama sistema, yra kuriama remiantis mikroservisų architektūra nuo pat pradžių, o ne pereinant iš kitos architektūros. Dėl to nėra jokių duomenų, kuriuos galima būtų migruoti – migracijos problema dėl to ir negali pasireikšti.

Toliau bus patikrinta, ar sistema turi *timeout* problemą. Kaip ir pirmoji sistema, ši sistema šią problemą taip pat turi. Agregatorius esantis prieš failų ir paaukojimo servisus šiuo atveju mažai ką pakeičia – vis vien gali atsitikti tas pats – dėl nepersiųsto patvirtinimo pranešimo į failų, bei paaukojimų duomenų bazes gali įsiterpti daugiau duomenų, negu iš tikrųjų reikia.

Kodo dalijimasis šioje sistemoje, kaip ir pirmojoje, egzistuoja kaip *string* tipo kintamųjų formatavimas, bei kelios klasės, praplečiančios standartinių klasių funkcionalumą. Šioje sistemoje toks kodas yra tiesiog kopijuojamas – nėra bendros kodo bazės, kurią pakeitus,

pasikeičia visuose servisuose esantis kodas, todėl šios problemos ši sistema, kaip ir pirmoji sistema, išvengia.

Toliau bus įvertintas sistemos mikroservisų stambumas. Ši sistema yra išskaidyta labai panašiai, kaip ir pirmoji. Vėl gi, galima sakyti, kad, kadangi visos duomenų bazių operacijos yra gana paprastos, per stambių mikroservisų greičiausiai nėra (bent jau tokių, kurie dirba su duomenų bazėmis). Nustatyti, ar šioje sistemoje yra per smulkių mikroservisų yra kiek kebliau, nes nėra aišku, ar būtent vartotojų mikroserviso išskyrimas yra pagrįstas, kadangi, kaip paaiškėjo iš sistemos bandymų, vien dėl tokio serviso įterpimo į šią architektūrą nukentėjo sistemos efektyvumas.

Mikroservisų architektūros pritaikymas šiai sistemai yra pagrįstas, kadangi ši sistema ir buvo sukurta mikroservisų architektūros vertinimui. Jeigu ignoruotume tą aspektą, ši sistema neturėtų jokios priežasties būti pagrįsta mikroservisais – ją kuria vienas žmogus – šio darbo autorius, jos funkcijos yra paprastos, sistemos nėra planuojama plėsti. Mikroservisų architektūros trūkumai šiuo atveju tikrai padaro daugiau žalos, nei naudos duoda privalumai. Tačiau jeigu neatmetame, kad šis bakalaurinis darbas yra pakankama priežastis šios sistemos sukūrimui naudojantis mikroservisų architektūra, tai galima sakyti, kad šios architektūros pritaikymas šiuo atveju yra pagrįstas.

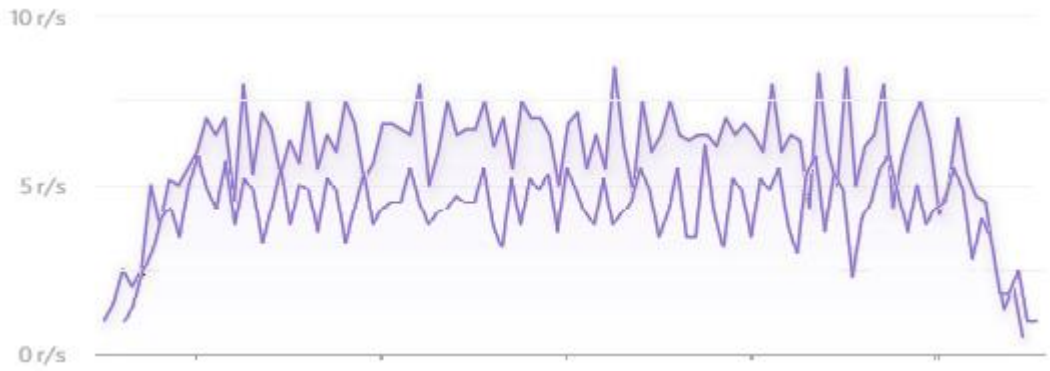
Šioje sistemoje kontraktas yra apibrėžtas statiškai – nėra versijavimo. Tai reiškia, kad kontrakto pakeitimus daryti yra sunku, kadangi nėra įmanoma palaikyti senų kontraktų versijų, kol nėra pilnai palaikomi naujieji kontraktai.

Bendravimas tarp servisų šioje sistemoje yra pasiekiamas tiesiogiai per TCP protokolą. Nėra naudojamas nei REST, nei kiti populiarūs mikroservisų bendravimo protokolai, todėl tai reiškia, kad ateityje būtų sunku plėsti šią sistemą, nes bendravimas tarp servisų yra sunkiai suprantamas. Tačiau, kadangi neplanuojama, kad su šia sistema ateityje bus dirbama, ar ji plečiama, tai nėra aktualu. Deja, kiti bendravimo būdai nebuvo testuoti, todėl nėra visiškai aišku, ar kiti bendravimo būdai būtų efektyvesni. Todėl galima sakyti, kad bendravimo būdai nėra įvertinti pakankamai gerai.

3.5. Pirmosios ir antrosios sistemų vertinimo rezultatų apibendrinimas

Lyginant šių dviejų sistemų testavimo rezultatus galima pamatyti, kad antroji sistema veikia šiek tiek lėčiau, bei turi didesnį atsako laiką – sistemų testų duomenys yra atvaizduoti grafike 3. Tai reiškia, kad dėl vartotojų serviso išskyrimo nukentėjo sistemos efektyvumas. Šio serviso iškyrimą galima būtų pateisinti tuo atveju, jeigu tuo būtų palengvinamas sistemos vystymas. Tai gali būti tiesa – išskyrus tokį agregatorių nebereikia, norint atlikti skirtingas operacijas, kreiptis į

skirtingus servisus – tinklalapio sistema, norėdama pasiekti kažkokį agreguojamą funkcionalumą, gali kreiptis į agregatorių, kuris pateikia užklausą reikiamam servisui. Taip supaprastėja, kaip yra pasiekiami agreguojami servisai, pasidaro paprasčiau įterpti daugiau servisų, kurie naudojasi agreguotaisiais servisais. Tačiau, kadangi ši sistema toliau nebus plėtojama, nėra svarbu, kad sistemos vidinės dalys būtų lengvai pasiekiamos, todėl šio serviso išskyrimas nėra pagrįstas ir galima sakyti, kad šiuo aspektu pirmoji sistema yra geresnė. Galima į tai pažiūrėti ir iš kito kampo – ši sistema yra išskirstyta pagal resursus – resursai, susiję su vartotojais, yra agreguojami vartotojų serviso. Nustatyti, ar verta sistemą išskaidyti pagal resursus, galima įvertinus, ar sistema turi daug sudėtingų resursų, kuriuos yra sunku valdyti. Jeigu taip, galima būtų skaidyti pagal resursus, tikintis, kad tai padarys sistemos kūrimo procesą sklandesnį ir dėl to sistema taps efektyvesnė. Tačiau šiuo atveju, sistema neturi tiek daug funkcijų. Tai reiškia, kad paprastesnis išskaidymas greičiausiai šiuo atveju yra geresnis, o tai yra pirmoji sistema, kurioje viskas yra tiesiog agreguojama tinklalapio mikroservise. Be to, matome, kad pirmoji sistema taip pat turi mažiau kodo eilučių, o tai greičiausiai reiškia, kad ji yra paprastesnė. Taigi šiuo atveju vartotojų serviso iškyrimas ne tik sumažina sistemos efektyvumą, bet ir apsunkina jos kūrimą. Tačiau verta paminėti tai, kad antroji sistema yra šiek tiek lengviau plečiama, jeigu sistemos poreikiai kada nors pasikeistų (tarkime, jeigu sistemą būtų pasirinkta plėsti ir ją pritaikyti kitoms reikmėms, nesusijusioms su šiuo bakalauriniu darbu).



Schema 3 – Užklausų kiekis per sekundę testų metu. Apatinė kreivė – antrosios sistemos.

Žiūrint į kitus sistemų vertinimo rezultatus, matyti, kad jos praktiškai nesiskiria. Iš tiesų, to ir galima buvo tikėtis, kadangi buvo apibrėžta, kad šios dvi sistemos skirsis tik architektūros implementacija, kitais aspektais sistemų kokybė ir turėjo išeiti panaši.

Abi sistemos turi „*timeout*“, statiško kontrakto, bei bendravimo neįvertinimo problemas. Žinant, kad nepagrįstai sistemą sukūrus jos mikroservisus išskiriant pagal resursus gautas

mažesnis sistemos efektyvumas, galima daryti išvadą, kad pataisius ir šias problemas, sistema turėtų pagerėti kažkokiu aspektu: arba efektyvumu, arba mikroservisų architektūros privalumais, tokiais kaip plečiamumas, lengvesnis testavimas ir panašiai.

Rezultatai ir išvados

Šio darbo tikslas buvo nustatyti, kiek skirtingas mikroservisų architektūros pritaikymas gali įtakoti sistemos kokybę.

Tam padaryti iš pradžių buvo pateiktas mikroservisų architektūros apibrėžimas. Po to buvo pateikta keletas mikroservisų pritaikymo būdų. Tada buvo, remiantis šaltiniais, buvo sudaryta paprasta mikroservisų architektūra paremtų sistemų vertinimo sistema. Šią sistemą sudarė dvi dalys: įprastų sistemų vertinimai, bei mikroservisų architektūros sistemų vertinimai. Įprastų sistemų vertinimus sudarė:

- Sistemos efektyvumas
- Sistemos kodo eilučių skaičius

Mikroservisų architektūros sistemų vertinimus sudarė problemos, kurių pasireiškimas nusako, ar sistema yra geros kokybės, ar ne. Tos problemos buvo:

- Duomenų pirmumo migracijos problema
- „*Timeout*“ problema
- Kodo dalijimosi problema
- Per stambių arba per smulkių servisų atskyrimo problema
- Nepagrįsto mikroservisų architektūros pritaikymo problema
- Statiško kontrakto problema
- Bendravimo neįvertinimo problema

Tada buvo sukurtos dvi mikroservisų architektūra paremto sistemos. Pirmoji sistema buvo sukurta servisus išskirstant pagal panaudojimą, o antroji – pagal resursus. Kiekviena sistema aprašyta. Pirmąją sistemą sudarė 5 servisai, o antrąją – 6. Abi sistemos buvo implementuotos. Joms buvo sukurti trys testai:

- Pradinio puslapio įkėlimas ir prisijungimas
- Pradinio puslapio įkėlimas, prisijungimas ir failų sąrašo peržiūrėjimas
- Pradinio puslapio įkėlimas, prisijungimas ir paaukojimas

Abi sistemos buvo ištestuotos visais trimis testais. Taip pat buvo įvertinta, ar sukurtos sistemos turi anksčiau išskirtas problemas. Nustatyta, kad abi sistemos turi „*timeout*“ problemą, statiško kontrakto problemą, bei bendravimo neįvertinimo problemą.

Baigus testus ir vertinimus, taip pat atsižvelgiant ir į abiejų sistemų kodo eilučių skaičių, buvo nustatyta, kad antroji sistema yra prastesnė už pirmąją, nes visų trijų testų metu jos rezultatai buvo prastesni už pirmosios, o tuo tarpu kodo eilučių skaičius yra 574 didesnis. Nustatyta, kad testų metu atsako greitis buvo nuo 95 ms iki 575 ms lėtesnis, bei, dviejuose testuose, užklausų greitis buvo mažesnis viena užklausa per sekundę. Nustatyta, kad to priežastis buvo tai, kad, atsižvelgiant į sistemos reikmes, sistemą yra geriau išskirstyti pagal panaudojimą, kadangi duomenų logika nėra pakankamai sudėtinga, kad būtų reikalingas išskyrimas pagal resursus.

Galutinė išvada yra, kad kiekvienas mikroservisų architektūros pritaikymo būdas gali turėti didelę įtaką sistemos kokybei įvairiais aspektais, tokiais kaip: dydis, efektyvumas, plečiamumas, atsparumas klaidoms, bei kitiems mikroservisų architektūros privalumams ir trūkumams.

ŠALTINIAI

[Hus19] Tom Huston „What is Microservices?“. Prieiga per internetą: https://smartbear.com/solutions/microservices/. [žiūrėta 2019-05-19]

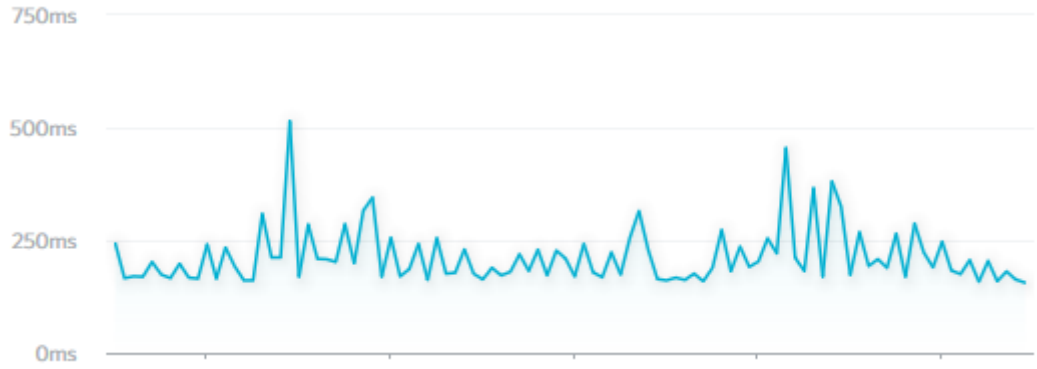
[Ann14] Robert Annett „What is a Monolith“. Prieiga per internetą: http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html. [žiūrėta 2019-05-19]

[Ric18] Chris Richardson „Pattern: Microservice Architecture“. Prieiga per internetą: https://microservices.io/patterns/microservices.html. [žiūrėta 2019-05-20]

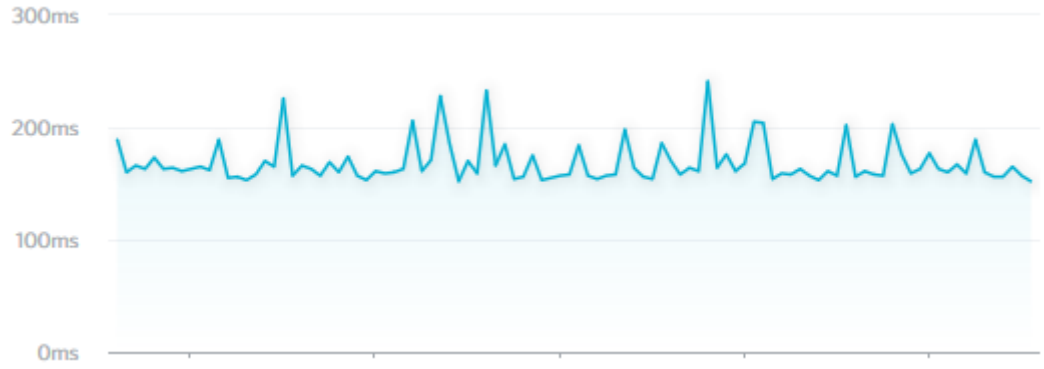
[Mon18] Anna Monus „What are microservices? The pros, cons, and how they work“. Prieiga per internetą: https://raygun.com/blog/what-are-microservices/. [žiūrėta 2019-05-21]

[Ric16] Mark Richards „Microservices antipatterns and pitfalls“. Prieiga per internetą: https://www.oreilly.com/ideas/microservices-antipatterns-and-pitfalls. [žiūrėta 2019-05-27]

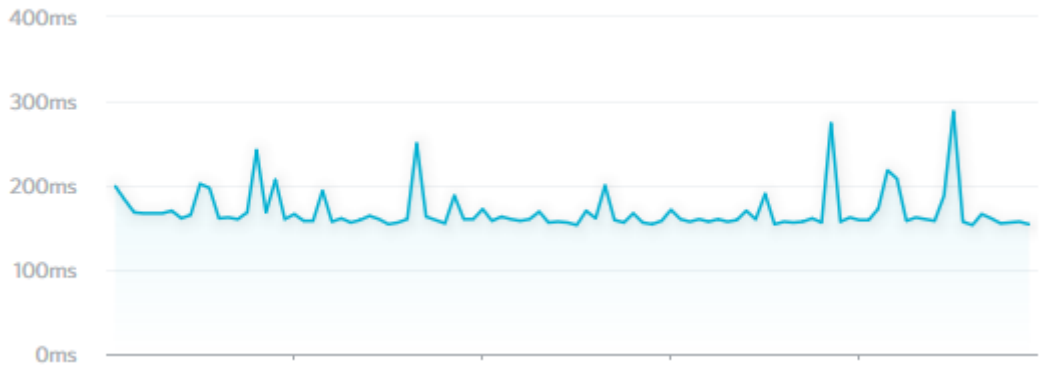
1 PRIEDAS



Schema 4 – Pirmosios sistemos pirmojo testo atsako greičio grafikas

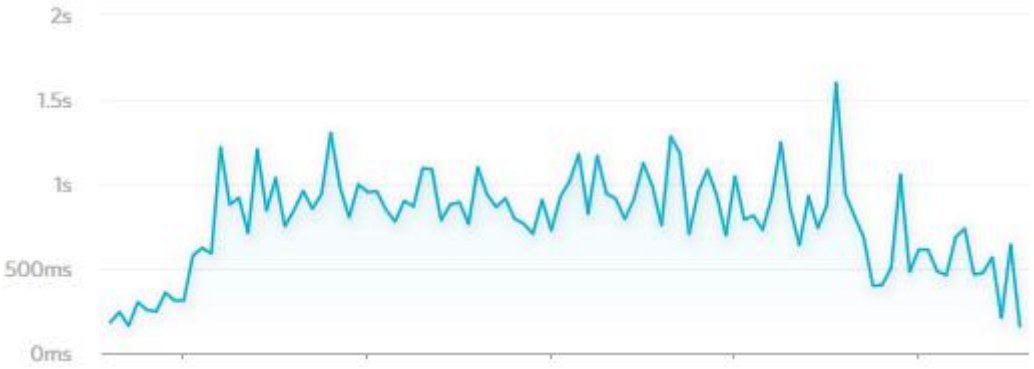


Schema 5 – Pirmosios sistemos antrojo testo atsako greičio grafikas

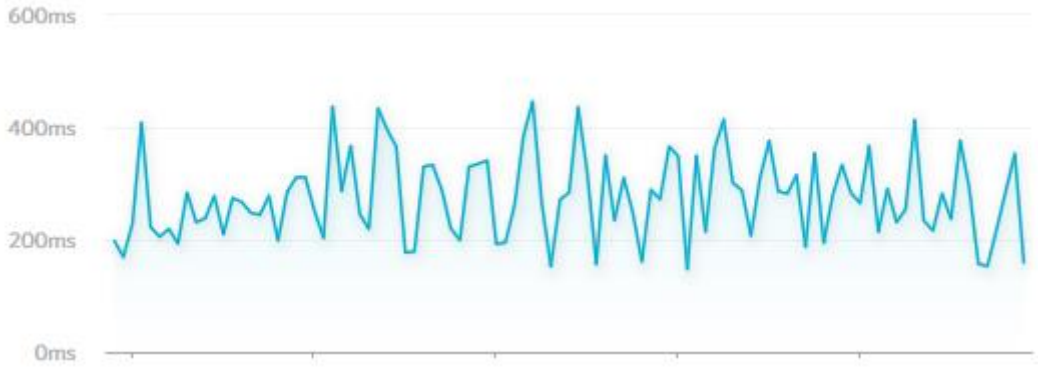


Schema 6 – Pirmosios sistemos trečiojo testo atsako greičio grafikas

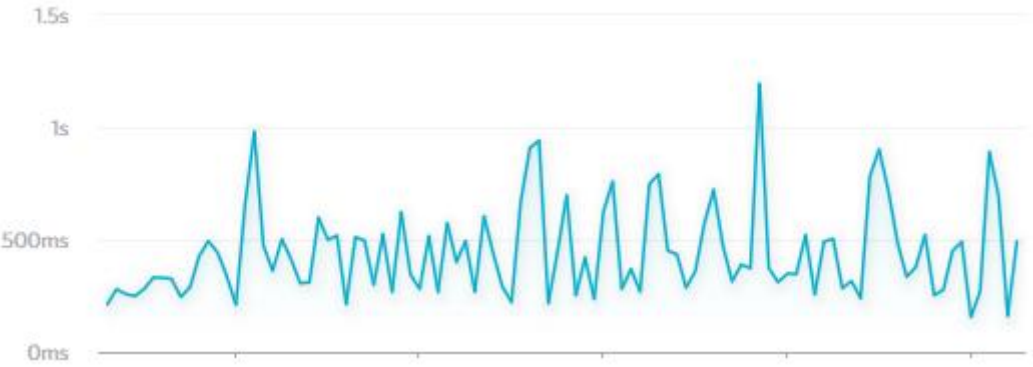
2 PRIEDAS



Schema 7 – Antrosios sistemos pirmojo testo atsako greičio grafikas



Schema 8 – Antrosios sistemos antrojo testo atsako greičio grafikas



Schema 9 – Antrosios sistemos trečiojo testo atsako greičio grafikas