



Kauno technologijos universitetas

Informatikos fakultetas

Mikroservisų testų generavimas iš UML sekų diagramų

Magistro baigiamasis projektas

Projektą parengė

Tomas Vainoris

Projektui vadovavo

Doc. dr. Lina Čeponienė

Kaunas, 2026



Kauno technologijos universitetas

Informatikos fakultetas

Mikroservisų testų generavimas iš UML sekų diagramų

Magistro baigiamasis projektas

Veiklos skaitmeninimas ir sistemų architektūros (6211BX009)

Projektą parengė

Tomas Vainoris

Projektui vadovavo

Doc. dr. Lina Čėponienė

Projektą recenzavo

Prof. dr. Evaldas Vaičiukynas

Kaunas, 2026



Kauno technologijos universitetas

Informatikos fakultetas

Tomas Vainoris

Mikroservisų testų generavimas iš UML sekų diagramų

Akademinio sąžiningumo deklaracija

Patvirtinu, kad:

1. baigiamąjį projektą parengiau savarankiškai ir sąžiningai, nepažeisdamas (-a) kitų asmenų autoriaus ar kitų teisių, laikydamasis (-i) Lietuvos Respublikos autorių teisių ir gretutinių teisių įstatymo, Kauno technologijos universiteto (toliau – Universitetas) intelektualinės nuosavybės valdymo ir perdavimo nuostatų bei Universiteto akademinės etikos kodekse nustatytų etikos reikalavimų;
2. visi baigiamajame projekte pateikti duomenys ir tyrimų rezultatai yra teisingi ir gauti teisėtai, nei viena projekto dalis nėra plagijuota nuo spausdintinių ar elektroninių šaltinių, o visos baigiamojo projekto tekste pateiktos citatos ir nuorodos yra nurodytos literatūros sąrašė;
3. baigiamajame projekte tinkamai laikiausi asmens duomenų apsaugos reikalavimų, nenaudojau neskelbtinų ar konfidencialių duomenų be teisėto pagrindo, o jei juos naudoju, jie yra tinkamai nuasmeninti;
4. jei rengiant baigiamąjį projektą naudojausi dirbtinio intelekto (toliau – DI) ar kitais automatizuotais įrankiais, juos taikiau pagal Universitete nustatytą tvarką, nepažeisdamas (-a) akademinio sąžiningumo principų;
5. nesumokėjau ir nesu įsipareigojęs (-usi) mokėti jokių įstatymų nenumatytų piniginių sumų už baigiamąjį projektą ar jo dalis jokiam fiziniam ar juridiniam asmeniui;
6. suprantu, kad išaiškėjus akademinio nesąžiningumo ar kitų asmenų teisių pažeidimo faktui, man bus taikoma atsakomybė pagal Universitete nustatytą tvarką ir galiu būti pašalintas (-a) iš Universiteto; akademinio nesąžiningumo atvejis gali būti nagrinėjamas ir po studijų baigimo, inicijuojant kvalifikacinio laipsnio atšaukimo procedūrą.

Vainoris, Tomas. Mikroservisų testų generavimas iš UML sekų diagramų. Magistro baigiamasis projektas / projektui vadovavo doc. dr. Lina Čeponienė; Kauno technologijos universitetas, Informatikos fakultetas.

Studijų kryptis ir studijų kryptių grupė: Informacijos sistemos, Informatikos mokslai.

Reikšminiai žodžiai: mikroservisai, UML sekų diagramos, testų generavimas, BDD, Gherkin, komponentų testavimas.

Kaunas, 2026. 88 p.

Santrauka

Mikroservisų architektūra plačiai taikoma kuriant šiuolaikines programų sistemas, siekiant didesnio lankstumo, plečiamumo ir nepriklausomo komponentų vystymo. Tačiau mikroservisų testavimas kelia papildomų iššūkių – kiekvienas servisas komunikuoja su kitais per *HTTP* užklausas, todėl automatinį testų kūrimas reikalauja išsamios sąveikų analizės ir yra imlus laikui. Nors monolitinėms sistemoms egzistuoja įrankių, leidžiančių generuoti testus iš *UML* diagramų, mikroservisų architektūrai pritaikytų sprendimų, orientuotų į servisų tarpusavio komunikacijos testavimą, šiuo metu trūksta. Darbo tikslas – palengvinti mikroservisų testavimą, pasiūlant metodiką mikroservisų elgsenos testų generavimui iš *UML* diagramų.

Analizės metu nustatyta, kad mikroservisų architektūroje ypač svarbūs komponentų ir kontraktų testai, leidžiantys izoliuotai tikrinti mikroserviso elgseną *HTTP* komunikacijos lygmenyje. Išanalizavus egzistuojančius testų generavimo sprendimus nustatyta, kad nė vienas jų neužtikrina visų trijų funkcijų vienu metu: testų scenarijų generavimo, vizualizacijos bei mikroservisų komunikacijos lygio tikrinimo.

Remiantis analize, pasiūlyta metodika, kurioje *UML* sekų diagramos naudojamos *Gherkin* formato testų scenarijams generuoti. Sekų diagramose vaizduojamos tarp servisų atliekamos *HTTP* užklausos, o sugeneruoti scenarijai tikrina, ar testuojamas servisas atliks numatytas užklausas su korektiškais duomenimis. Metodika orientuota į *REST* stiliaus komunikaciją, apibrėžti devyni žingsnių šablonai bei sekų diagramų sudarymo taisyklės. Sukurtas įrankis realizuotas *C#* programavimo kalba, naudojant *.NET* karkasą. Įrankis integruojamas su *GitHub Copilot* įskiepiu per *MCP* protokolą, suteikiant galimybę generuoti testus tiesiogiai iš *Visual Studio Code* aplinkos. Testų vykdymui naudojama *Reqnroll* biblioteka.

Eksperimentinio tyrimo rezultatai parodė, kad sukurtas įrankis teisingai generuoja *Gherkin* scenarijus, apdoroja alternatyvos ir sąlygos fragmentus bei korektiškai formatuoja *HTTP* užklausų turinį. Sugeneruoti testai pasiekė vidutinį 92,6 % kodo padengimą ir 69,8 % mutacinio testavimo rezultatą. Ekspertų apklausoje 84,6 % specialistų nurodė, kad sugeneruoti testai yra naudingi, o 69,2 % taikytų metodiką praktikoje. Gauti rezultatai patvirtina, kad pasiūlyta metodika gali būti taikoma mikroservisų testavimo procesui palengvinti.

Vainoris, Tomas. *Microservice Test Generation from UML Sequence Diagrams*. Master's Final Degree Project / supervisor assoc. prof. Lina Čeponienė; Faculty of Informatics, Kaunas University of Technology.

Study field and study field group: Information Systems, Computing.

Keywords: Microservices, UML Sequence Diagrams, Test Generation, BDD, Gherkin, Component Testing.

Kaunas, 2026. 88 p.

Summary

Microservice architecture is widely used in the development of modern software systems in order to achieve greater flexibility, scalability, and independent component development. However, testing microservices introduces additional challenges—each service communicates with others via HTTP requests, therefore the creation of automated tests requires a detailed analysis of interactions and is time-consuming. Although tools exist for monolithic systems that generate tests from UML diagrams, there is currently a lack of solutions adapted to microservice architecture that focus on testing inter-service communication. The aim of the work is to facilitate microservice testing by proposing a methodology for generating behavioral tests for microservices from UML diagrams.

The analysis determined that component and contract testing is particularly important in microservice architecture, enabling isolated verification of a microservice's behavior at the HTTP communication level. After analyzing existing test generation solutions, it was found that none of them simultaneously provide all three functions: generation of test scenarios, visualization, and verification at the microservice communication level.

Based on the analysis, a methodology was proposed in which UML sequence diagrams are used to generate Gherkin-format test scenarios. Sequence diagrams represent HTTP requests exchanged between services, and the generated scenarios verify whether the system under test performed the expected requests with correct data. The methodology is oriented toward REST-style communication, defining nine step templates and rules for constructing sequence diagrams. The developed tool is implemented in the C# programming language using the .NET framework. The tool is integrated with the GitHub Copilot extension via the MCP protocol, enabling test generation directly within the Visual Studio Code environment. The Reqroll library is used for test execution.

The results of the experimental study showed that the developed tool correctly generates Gherkin scenarios, handles alternative and option fragments, and correctly formats HTTP request payloads. The generated tests achieved an average code coverage of 92.6% and a mutation testing score of 69.8%. In an expert survey, 84.6% of specialists indicated that the generated tests are useful, and 69.2% would apply the methodology in practice. The obtained results confirm that the proposed methodology can be applied to facilitate the microservice testing process.

Turinys

Lentelių sąrašas	8
Paveikslų sąrašas	9
Santrumpų ir terminų sąrašas	11
Įvadas.....	12
1. Probleminės srities analizė	14
1.1. Analizės tikslas.....	14
1.2. Tyrimo objektas, sritis ir problema.....	14
1.3. Mikroservisų testavimo naudojant automatinius testus analizė	14
1.3.1. Mikroservisų architektūra.....	14
1.3.2. Mikroservisų projektavimo šablonai	17
1.3.3. Transakcijų valdymas mikroservisų architektūroje.....	17
1.3.4. Mikroservisų testavimo strategijos.....	19
1.3.5. Nuolatinė integracija ir nuolatinis diegimas.....	21
1.3.6. <i>BDD</i> ir <i>Gherkin</i> karkasų taikymas testavime.....	22
1.3.7. Didžiųjų kalbos modelių taikymas testų generavime	23
1.3.8. Testų kokybės įvertinimo analizė.....	24
1.4. <i>UML</i> ir jos taikymas mikroservisų architektūros sistemoms modeliuoti	25
1.5. Mikroservisų testavimo dalyvių analizė	29
1.6. Esamų problemos sprendimo metodų analizė	30
1.6.1. Testų generavimo iš sekų diagramų metodas [51]	30
1.6.2. <i>SCOTEM</i> testų generavimo iš komunikacijos ir būsenų diagramų metodas [50]....	31
1.6.3. Testų generavimas naudojant <i>OpenApi</i> specifikaciją [49]	33
1.6.4. Grafu paremtas mikroservisų analizės įrankis [46]	33
1.6.5. <i>BDD</i> scenarijais grįstas sprendimas [47]	35
1.6.6. Testų generavimas naudojant <i>UML</i> būsenų diagramas ir <i>Cucumber</i> karkasą [48].	36
1.6.7. Testų generavimo ir vizualizavimo įrankių palyginimas	38
1.7. Siekiamo sprendimo apibrėžimas.....	40
1.8. Analizės išvados.....	40
2. Mikroservisų testų generavimo iš sekų diagramų reikalavimų specifikacija, formalizuotas aprašas	42
2.1. Reikalavimų specifikacija	42
2.2. Dalykinės srities modelis	43
2.3. Formalizuotas sprendimo aprašas	44
2.3.1. Testų generavimo logika	44
2.3.2. Testų generavime naudojami <i>Gherkin</i> žingsnių šablonai	48
2.3.3. Sekų ir klasių diagramų braižymo taisyklės	49
2.4. Reikalavimų apibendrinimas	55
3. Testų generavimo iš sekų diagramų įrankio projektas	56
3.1. Įrankio reikalavimų specifikacija	56
3.2. Įrankio projekto modelis	58
3.3. Įrankio projekto apibendrinimas	60
4. Testų generavimo iš sekų diagramų įrankio realizacija ir testavimas.....	61
4.1. Testų generavimo iš sekų diagramų įrankio realizacijos ir veikimo aprašas.....	61
4.2. Diegimo instrukcija	61

4.2.1. Įrankio diegimas	61
4.2.2. Testuojamo projekto konfigūravimas	63
4.2.3. Testų generatoriaus naudojimas	64
4.3. Testavimo modelis, duomenys, rezultatai	66
4.3.1. Testavimo duomenys	66
4.3.2. Vienetų testavimas	68
4.3.3. Integraciniai testai.....	69
5. Eksperimentinis testų generavimo iš sekų diagramų įrankio tyrimas	71
5.1. Eksperimento planas.....	71
5.2. Testų generavimo mikroservisų projektams eksperimentas.....	71
5.3. Testų generavimo mikroservisų projektams eksperimento rezultatai	73
5.4. Mikroservisų testų generavimo metodikos įvertinimo apklausa.....	77
5.5. Apklausos rezultatai	78
5.6. Testų generavimo iš sekų diagramų įrankio veikimo ir savybių analizė, kokybės kriterijų įvertinimas.....	80
5.7. Testų generavimo iš sekų diagramų metodikos taikymo rekomendacijos	81
Išvados.....	83
Literatūros sąrašas	84
Priedai	89
1 priedas	89
2 priedas	90
3 priedas	93
4 priedas	104

Lentelių sąrašas

1.1 lentelė. Egzistuojančių testų generavimo ir vizualizavimo įrankių sprendimų palyginimas	39
2.1 lentelė. Naudojamų šablonų generuojamuose scenarijuose sąrašas	49
2.2 lentelė. Sekų ir klasių diagramų braižymo taisyklės.....	51
5.1 lentelė. Testuojami galutiniai taškai (<i>angl. endpoint</i>)	73
5.2 lentelė. Statistika apie sekų diagramas	74
5.3 lentelė. Sugeneruotų testų statistika	76
5.4 lentelė. Apklauskos klausimai.....	78

Paveikslų sąrašas

1.1 pav. Mikroservisų architektūros pavyzdys [5].....	15
1.2 pav. Įvykiais paremtos komunikacijos principinė diagrama [10].....	16
1.3 pav. SAGA šablonas [16].....	18
1.4 pav. SAGA choreografijos šablonas [16].....	18
1.5 pav. SAGA orkestravimo šablonas [16].....	19
1.6 pav. Testavimo piramidė [17].....	20
1.7 pav. Kodo integravimo procesas [24].....	22
1.8 pav. Svetainės testavimo pavyzdiniai scenarijai	23
1.9 pav. UML diagramų hierarchija [39]	25
1.10 pav. Mikroservisų architektūros pavyzdinis vaizdavimas klasių diagrama [3]	27
1.11 pav. Mikroservisų, komunikuojančių asinchroniškai, komponentų diagrama [41]	28
1.12 pav. Komunikacijos tarp mikroservisų vaizdavimas naudojant sekų diagramą [42] ..	29
1.13 pav. Mikroservisų testavimo dalyviai	29
1.14 pav. Testų generavimo procesas naudojant sekų diagramas ir išplėstąjį baigtinį automatą [51]	31
1.15 pav. Testų generavimo procesas naudojant SCOTEM modelį [50].....	32
1.16 pav. Parametrų rinkinių generavimo pavyzdys [49].....	33
1.17 pav. GSMART grafo pavyzdys [46].....	34
1.18 pav. GSMART sukurto mikroservisų grafo pavyzdys [46]	35
1.19 pav. Bendrų automatinių testų architektūra [47].....	36
1.20 pav. Testų generavimo naudojant Skyfire procesas [48]	37
1.21 pav. Pavyzdinė būsenų diagrama, kuriai generuojami testai [48]	37
1.22 pav. Sugeneruoto scenarijaus pavyzdys [48].....	38
2.1 pav. Testų generatoriaus panaudojimo atvejų modelis	42
2.2 pav. Mikroservisų testų generavimo iš UML sekų diagramų proceso veiklos diagrama	43
2.3 pav. Testų generavimo iš sekų diagramų dalykinės srities modelis	44
2.4 pav. UML metaklasės naudojamos testų generavime [39].....	44
2.5 pav. Mikroservisų komunikacijos modelis	45
2.6 pav. Testų scenarijų modelis.....	46
2.7 pav. PA „Generuoti testų scenarijus servisui“ veiklos diagrama.....	47
2.8 pav. PA „Transformuoti diagramą į servisų komunikacijos modelį“ veiklos diagrama .	48
2.9 pav. Pavyzdinė mikroservisų klasių diagrama su sekų diagramos iškarpa	52
2.10 pav. Testavimo duomenų modeliavimas naudojant egzempliorių specifikaciją	52
2.11 pav. Pavyzdinė sekų diagrama	53
2.12 pav. Atsiskaitymo proceso pavyzdinis testų scenarijus	54
2.13 pav. Sekų diagrama su alternatyvos fragmentu	55
2.14 pav. Paskyros sukūrimo pavyzdiniai testų scenarijai	55
3.1 pav. Testų generatoriaus įrankio panaudojimo atvejai	56
3.2 pav. PA „Importuoti XMI formato modelį“ veiklos diagrama	57
3.3 pav. PA „Peržiūrėti sekų diagramų sąrašą“ veiklos diagrama	58
3.4 pav. PA „Generuoti testų scenarijus“ veiklos diagrama.....	58
3.5 pav. Įrankio paketų diagrama.....	59
3.6 pav. Testų generavimo įrankio diegimo diagrama.....	60

4.1 pav.	Testų generatoriaus diegimo komanda ir išvestis	61
4.2 pav.	Testų generatoriaus įrašymo tikrinimas	62
4.3 pav.	Failo <i>mcp.json</i> konfigūracijos pavyzdys	62
4.4 pav.	<i>MCP</i> serverio konfigūracija su patikrinimo užklausa	63
4.5 pav.	Bendrų žingsnių šablonų bibliotekos diegimas	63
4.6 pav.	Failo <i>reqnroll.json</i> konfigūracija	63
4.7 pav.	Bendrų žingsnių diegimas testavimo projekte	64
4.8 pav.	Modelio eksportavimo langas	64
4.9 pav.	Diagramų išvedimas naudojant <i>GitHub Copilot</i> įrankį kaip sąsają	65
4.10 pav.	Testų scenarijų generavimas naudojant <i>GitHub Copilot</i> įrankį kaip sąsają	65
4.11 pav.	Sugeneruotų testų vykdymo pavyzdys	66
4.12 pav.	Sekų diagrama skirta testuoti alternatyvos fragmento veikimą	67
4.13 pav.	Sugeneruoti testų scenarijai alternatyvos fragmento testavimo atveju	67
4.14 pav.	Sekų diagrama skirta testuoti grąžinamą atsakymo turinį	68
4.15 pav.	Sugeneruotas scenarijus su atsakymo turinio tikrinimu	68
4.16 pav.	Vienetų testų vykdymo rezultatas	69
4.17 pav.	Integracinių testų vykdymo rezultatas	70
5.1 pav.	Mikroservisų testavimo rezultatai	77
5.2 pav.	Respondentų patirtis dirbant su mikroservisų architektūra, <i>UML</i> , <i>BDD</i>	79
5.3 pav.	Testų generavimo iš sekų diagramų metodikos įvertinimo apklausos rezultatai ..	80
5.4 pav.	Užklauso argumento modeliavimas	81
5.5 pav.	Pavyzdinė egzempliorių specifikacijos ir sekų diagramų išdėstymo struktūra	82

Santrumpų ir terminų sąrašas

API (angl. Application Programming Interface) – programinės sąsajos rinkinys, kuris leidžia skirtingoms programoms ar sistemoms komunikuoti ir sąveikauti tarpusavyje;

HTTP (angl. Hypertext Transfer Protocol) – protokolas, skirtas pasiekti informaciją internete;

HTTPS (angl. Hypertext Transfer Protocol Secure) – interneto protokolo HTTP saugus variantas;

UML (angl. Unified Modeling Language) – standartizuota vizualizacijos kalba, skirta programų sistemų modeliavimui;

MCP (angl. Model Context Protocol) – protokolas, skirtas perduoti ir valdyti kontekstinę informaciją tarp dirbtinio intelekto modelių ir išorinių sistemų.

Ivadas

Darbo problematika ir aktualumas

Paskirstytosios sistemos, sukurtos remiantis mikroservisų architektūra, sudaro didelės dalies komercinių programinės įrangos sistemų pagrindą [1]. Šio architektūrinio stiliaus populiarumą lemia gebėjimas suteikti nepriklausomą servisų diegimą, plečiamumą ir skirtingų technologijų pasirinkimo laisvę [2]. Tačiau kartu su šiais privalumais mikroservisų architektūra sukuria sudėtingesnę testavimo aplinką: sistemą sudaro daugybė autonomiškų komponentų ir kiekvienas iš jų komunikuoja su kitais per tinklo sąsajas, todėl vienetų ar integracinių testų taikymas negarantuoja tarpusavio komunikacijos elgsenos patikrinimo.

UML diagramos gali būti naudojamos sistemos struktūrai ir elgsenai vizualizuoti: jos leidžia suprasti sistemos veikimą, net ir tiems, kurie nėra su ja susipažinę. *UML* klasių diagramos naudojamos *API* dokumentacijos specifikavimui, o sekų diagramos parodo, kaip mikroservisai sąveikauja tarpusavyje, galima vaizduoti klaidos atvejus, kaip sistema reaguoja, kai grąžinamas klaidos pranešimas iš kitos sistemos dalies [3]. Šių diagramų panaudojimas gali būti išplėstas už vizualizavimo ribų, taikant jas kaip testų generavimo pagrindą, nes jose esanti struktūrinė ir elgsenos informacija atspindi tarpusavio sąveikos logiką. Taigi, problemai siūlomas sprendimas yra testų generavimo metodika, kuri leistų automatiškai generuoti testus iš *UML* sekų diagramų, aprašančių mikroservisų tarpusavio komunikaciją.

Darbo tikslas ir uždaviniai

Tyrimo tikslas yra palengvinti mikroservisų testavimą, pasiūlant metodiką mikroservisų elgsenos testų generavimui iš *UML* diagramų. Testuojamo mikroserviso elgsena būtų tikrinama izoliuotai, kai kitų, su juo komunikujančių mikroservisų atsakymai yra simuliuojami. Šiam tikslui pasiekti išsikelti šie uždaviniai:

1. Išanalizuoti mikroservisų testavimo strategijas ir egzistuojančius testavimo įrankius.
2. Išanalizuoti *UML* ir jos taikymą sistemos funkcionalumo ir struktūros atvaizdavimui mikroservisų architektūroje.
3. Išanalizuoti testų generavimo galimybes iš *UML* diagramų ir sprendimus, skirtus kurti automatinius testus mikroservisams.
4. Pasiūlyti metodiką mikroservisų testams generuoti iš *UML* diagramų.
5. Realizuoti mikroservisų testų generavimo įrankį.
6. Eksperimentiškai ištirti sukurtos testų generavimo metodikos ir įrankio taikymą kuriant automatinius testus mikroservisams bei apibendrinti tyrimo rezultatus.

Darbo rezultatai ir jų svarba

Darbo rezultatas – metodika ir įrankis, leidžiantys automatiškai generuoti mikroservisų testų scenarijus iš *UML* sekų diagramų. Sukurtas įrankis integruotas su *GitHub Copilot* įskiepiu per *MCP* protokolą, todėl testų generavimas pasiekiamas tiesiogiai iš programavimo aplinkos. Eksperimentas parodė, kad sugeneruoti testai pasiekė vidutinį 92,6 % kodo padengimą ir 69,8 % mutacinio testavimo rezultata, o ekspertų apklausoje 69,2 % specialistų įvertino metodiką kaip naudingą. Tai rodo, kad sukurtas sprendimas gali praktiškai sumažinti rankinį darbą kuriant automatinius testus mikroservisams.

Darbo struktūra

Darbą sudaro penki pagrindiniai skyriai. Pirmajame skyriuje atliekama probleminės srities analizė: apžvelgiama mikroservisų architektūra, testavimo strategijos, *UML* diagramų taikymas ir egzistuojantys testų generavimo sprendimai. Antrajame skyriuje pateikiama testų generavimo metodikos reikalavimų specifikacija ir formalizuotas aprašas, apibrėžiami žingsnių šablonai ir sekų diagramų braižymo taisyklės. Trečiajame skyriuje aprašomas sukurto įrankio projektas – panaudojimo atvejai, komponentų ir diegimo diagramos. Ketvirtajame skyriuje pateikiamas įrankio realizacijos aprašas, diegimo instrukcija ir testavimo rezultatai. Penktajame skyriuje pateikiamas eksperimentinis tyrimas: aprašomas eksperimento planas, naudoti mikroservisų projektai ir testavimo duomenys, pateikiami kodo padengimo ir mutacinio testavimo rezultatai bei apklausos rezultatai.

Generatyvinio dirbtinio intelekto naudojimas

Rengiant baigiamąjį projektą buvo naudojami šie generatyvinio dirbtinio intelekto (GDI) įrankiai:

- ChatGPT (OpenAI, GPT-5.4-mini modelis) – naudotas teksto analizės, struktūrizavimo ir akademinio stiliaus formavimo pagalbai, rašybos ir gramatikos tikrinimui, teksto redagavimui bei perfrazavimui siekiant aiškesnės ir nuoseklesnės formuluotės;
- Claude (Anthropic, Sonnet 4.6 modelis) – taikytas prototipo programinės realizacijos etape, padedant generuoti ir tobulinti kodą, sprendžiant realizacijos problemas;
- Elicit – naudotas mokslinių straipsnių paieškai, literatūros apžvalgos rengimui ir šaltinių atrankai pagal tyrimo tematiką.

Šio projekto autorius nuodugniai įvertino, suprato ir pagal poreikį redagavo GDI sugeneruotą turinį bei prisiima dėl jo pilną atsakomybę.

Magistro baigiamojo projekto rezultatų publikavimas

Magistro baigiamojo projekto pagrindu buvo parengta mokslinė publikacija „Automated Test Generation for Microservices from UML Sequence Diagrams“ priimta į IVUS 2026 konferenciją ir pristatyta 2026 m. gegužės 12 d. (žr. 1 priedą).

1. Probleminės srities analizė

Šiame skyriuje pateikiama mikroservisų architektūros analizė, mikroservisų testavimo strategijų analizė, *UML* modeliavimo kalbos analizė, *UML* diagramų taikymo mikroservisų kūrimo procese analizė.

1.1. Analizės tikslas

Automatinių mikroservisų testų generavimo naudojant *UML* diagramas analizės tikslas – išnagrinėti testų generavimo galimybes iš *UML* diagramų, apžvelgti testavimo karkasus, naudojamus mikroservisams testuoti, ir įrankius kodo generavimui. Parinkti tinkamas *UML* diagramas ir testavimo karkasus automatiniam testams generuoti.

1.2. Tyrimo objektas, sritis ir problema

Šio darbo tiriamasis objektas – mikroservisų testavimas naudojant automatinius testus.

Mikroservisų architektūra yra vis plačiau naudojama, kaip alternatyva monolitinėms sistemoms. Kadangi sistema, sukurta pagal mikroservisų architektūrą, sudaro daugelis mažų servisų, komunikuojančių tarpusavyje, ryšiai tarp servisų ir vykstanti komunikacija gali būti sudėtinga. Todėl tampa sunku rašyti automatinius testus, kurie visiškai ištestuotų mikroservisų veikimą.

Tyrimo sritis – automatinių testų generavimo iš *UML* sekų diagramų įrankiai ir metodai mikroservisų testavime.

1.3. Mikroservisų testavimo naudojant automatinius testus analizė

Šiame poskyryje pateikiamas mikroservisų architektūros aprašymas ir pagrindiniai šios architektūros bruožai. Apžvelgta, kokiais komunikacijos architektūros stiliais komunikuojama tarp servisų. Taip pat apžvelgiama, kokio tipo testai ir strategijos yra naudojamos testuojant mikroservisus. Be to, pateikta, kokie testavimo karkasai gali būti naudojami automatinių testų kūrimui ir vykdymui. Galiausiai analizuojamos *UML* diagramos ir kaip jos naudojamos vaizduojant mikroservisų funkcionalumą ir struktūrą.

1.3.1. Mikroservisų architektūra

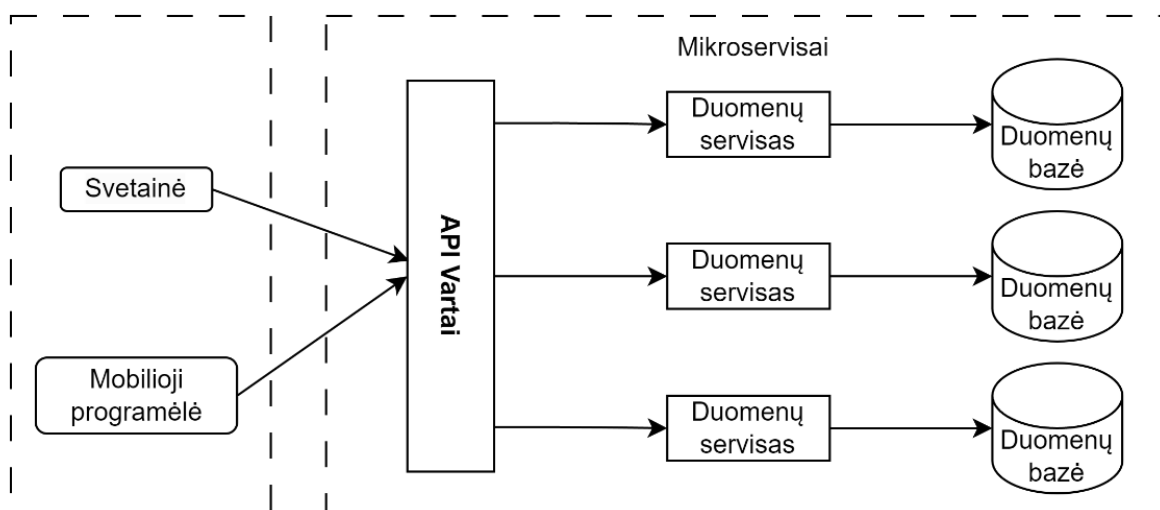
Mikroservisų architektūra – tai architektūrinis stilius, kuriuo siekiama sukurti autonomiškus servisus, atliekančius vieną aiškiai apibrėžtą funkciją [4]. Šie servisai gali būti lengvai diegiami atskirai, dėl mažo jų dydžio kūrimui pakanka vienos programuotojų ir testuotojų komandos. Be to, mikroservisai gali būti plėtojami naudojant skirtingas technologijas, suteikiant komandoms laisvę pasirinkti jiems tinkamas technologijas [4].

Mikroservisai turi ir trūkumų. Dėl padidėjusios komunikacijos tarp servisų gali padidėti atsakymų gaitės laikas ir naudojama daugiau kompiuterinių resursų [5]. Be to, galutinis atsakymas yra surenkamas iš kelių mikroservisų teikiamų duomenų. Todėl reikalingi servisai, kurie agreguoja kelių servisų atsakymus, tai didina sistemos kompleksumą. Taip pat atsiranda papildomi palaikymo ir diegimo darbai, nes reikia diegti ir koordinuoti kelis servisus, priešingai negu taikant monolitinę architektūrą, kai diegiamas tik vienas servisas [6]. Siekiant pasiekti mikroservisų nepriklausomumą, sistema turi būti atidžiai suprojektuota.

Jeigu projektuojant neatsižvelgiama į tai, kad skirtingi servisi atliktų konkrečią funkciją ir nesidalintų tomis pačiomis atsakomybėmis, mikroservisi gali tapti glaudžiai susiję dėl dažnos komunikacijos tarpusavyje. Be to, tokie mikroservisi negali būti atnaujinami visiškai nepriklausomai, nes jų diegimas reikalauja papildomos koordinacijos, taip sumažinant nepriklausomumo privalumą [4].

1.1 pav. pateikta, kaip gali atrodyti sistema, sukurta taikant mikroservisų architektūrą. Mikroservisus galima suskirstyti į 2 pagrindines grupes pagal paskirtį [4]:

- Duomenų servisas – atsakingas už duomenų saugojimą ir gavimą, todėl turi tik jam priklausančią duomenų bazę, kurią gali pasiekti tik šis servisas [4]. Kiti servisi negali tiesiogiai pasiekti duomenų bazės, jie privalo komunikuoti per serviso teikiamą sąsają. Toks sprendimas leidžia kurti nepriklausomus servisius, nes pokyčiai duomenų bazėje nedarys tiesioginės įtakos kitiems servisiams.
- *API vartų* (*angl. API gateway*) servisas – tai gali būti viešai pasiekiamas servisas, kuris priima užklausas iš programėlių, svetainių arba kitų sistemų, kurios naudojami sukurta sistema [4]. Jei šio serviso sąsaja yra viešai prieinama, ji turi būti gerai apsaugota naudojant pasirinktą autentifikacijos metodą. Dažniausiai šie servisi atlieka tarpininko vaidmenį ir tik perduoda užklausą vidiniams servisiams.



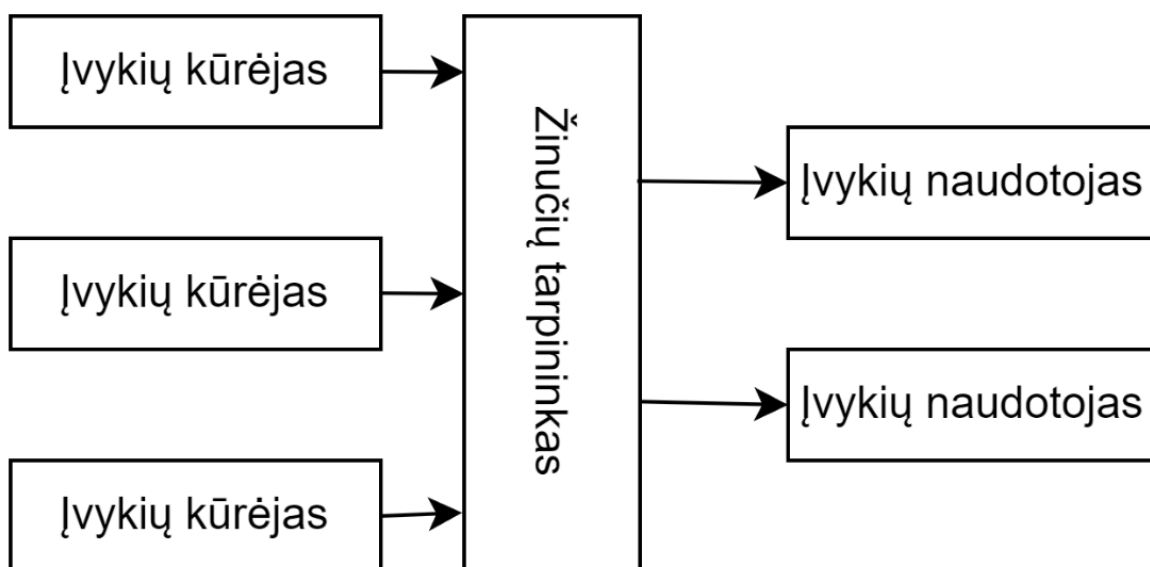
1.1 pav. Mikroservisų architektūros pavyzdys [5]

Mikroservisų architektūros esminis bruožas yra tai, kad servisi komunikuoja tinklo lygmenyje. Komunikacijai tarp servisų gali būti naudojamos skirtingos technologijos [7]. Sinchroninei komunikacijai naudojamas *REST* (*angl. Representational State Transfer*) architektūrinis stilius [8]. *REST API* aprašo, kokius resursus teikia ir kokias operacijas atlieka servisas. Pagrindiniai *REST* operacijų tipai [8]:

- *GET* – naudojamas duomenų gavimui. *GET* operacija turi nekeisti sistemos būsenos;
- *POST* – skirtas siųsti duomenis apdorojimui arba naujo resurso sukūrimui. *POST* metodas gali keisti sistemos būseną;
- *PUT* – skirtas sukurti arba atnaujinti resursą;
- *PATCH* – skirtas atnaujinti dalį resurso;
- *DELETE* – skirtas pašalinti nurodytą resursą.

Kaip alternatyva *REST* komunikacijos stiliui, taip pat naudojamas atvirojo kodo *gRPC* karkasas. Šis karkasas naudoja *HTTP/2* tinklo protokolą, siekiant didesnės greಿತaveikos [9]. Taip pat karkasas yra palaikomas daugelio populiariausių kalbų. *gRPC* karkasu servisų sąsajos yra apibrėžiamos naudojant *protobuf* failus, kurie nusako, kokie atributai sudaro užklausą [9].

Asinchroninei komunikacijai tarp mikroservisų gali būti naudojama įvykiais grindžiama (*angl. event-based*) komunikacija [10]. Kiekvienas įvykis turi duomenis, reikalingus operacijai atlikti. Servisas gali kurti arba priimti įvykius. Įvykiai nėra tiesiogiai perduodami tarp servisų; žinučių perdavimui naudojamas atskiras tarpinis komponentas, kuris užtikrina, kad žinutės bus perduotos reikalingiems servisams. 1.2 pav. pateikta įvykiais paremtos komunikacijos diagrama. Įvykius kuriantis servisas paskelbia žinutę į žinučių tarpininką, o žinutes priimantis servisas priima ir apdoroja žinutę. Esminis šio komunikacijos stiliaus bruožas yra tai, kad žinutės paskelbimas ir apdorojimas neturi vykti tuo pačiu metu. Įvykio kūrėjas nelaukia, kada žinutė bus priimta ir apdorota [10].



1.2 pav. Įvykiais paremtos komunikacijos principinė diagrama [10]

Sinchroninės ir asinchroninės komunikacijos tipai turi esminių skirtumų. Naudojant *REST* arba *gRPC* komunikacija vyksta sinchroniškai. Servisas, kuris atlieka užklausą, laukia atsakymo iš kito serviso. Jeigu atliekamos kelios užklausos, jų eiga yra nuosekli ir lengvai atsekama. Įvykiais paremta komunikacija kelia papildomų iššūkių automatinėms testų generavimui. Skirtingai nuo *REST*, šis komunikacijos stilius yra asinchroninis – servisas paskelbia žinutę ir nelaukia tiesioginio atsakymo, todėl vykdymo seka nėra deterministinė [10]. *UML* sekų diagramos yra orientuotos į nuoseklią, sinchroninę sąveiką, todėl jomis sunku tiksliai atvaizduoti asinchroninius srautus. Be to, testams reikalinga aiški užklausos ir atsakymo sąsaja. Dėl šių priežasčių siūloma metodika bus apribota sinchronine *REST* komunikacija.

1.3.2. Mikroservisų projektavimo šablonai

Praktikoje susiduriant su pasikartojančiais architektūriniais iššūkiais, buvo suformuoti mikroservisų projektavimo šablonai, apibrėžiantys patikrintus sprendimus dažniausiai pasitaikančioms problemoms spręsti paskirstytose sistemose: *API vartų*, agregatoriaus, *Strangler Fig*, *Anti-Corruption Layer* [11].

API vartai yra mikroservisų architektūros elementas, per kurį keliauja visos išorinės užklauskos ir kuris atsakingas už jų nukreipimą į vidinius servisus [12]. *API* vartų servisas supaprastina kliento sąveiką su sistema ir atlieka pasikartojančias funkcijas: užtikrina autentifikaciją, autorizaciją, apkrovos balansavimą ir užklauskų ribojimą. Be to, *API* vartai taikomi klientui pritaikytos serverinės dalies šablonui (*angl. Backend for Frontend*) [11]. Skirtingiems klientų tipams (pvz., žiniatinklio ar mobiliosioms aplikacijoms) kuriami atskiri *API* vartai, pritaikyti specifiniams funkciniais reikalavimams. Dar viena svarbi *API* vartų funkcija yra užklauskų agregavimas [13]. Vietoj to, kad klientas siųstų atskiras užklauskas kiekvienam servisui ir pats surinktų duomenis, agregatoriaus servisas atlieka visas reikalingas užklauskas lygiagrečiai arba nuosekliai ir grąžina apjungtą rezultatą [13]. Toks sprendimas sumažina kliento ir sistemos tarpusavio priklausomybę, slepia vidinę sistemos struktūrą ir mažina tinklo apkrovą.

Anti-Corruption Layer yra architektūrinis šablonas, skirtas apsaugoti sistemą nuo išorinių sistemų poveikio [11, 12]. Šis sluoksnis veikia kaip tarpinė abstrakcija, konvertuodamas išorinius duomenis ir sąsajas į vidiniam domeniui priimtina formą. Nors *Anti-Corruption Layer* principai yra universalūs ir gali būti taikomi bet kurioje sistemoje, kur reikia izoliuoti vidinį modelį nuo išorinių įtakų, mikroservisų architektūroje jis dažnai įgyvendinamas kaip savarankiškas mikroservisas.

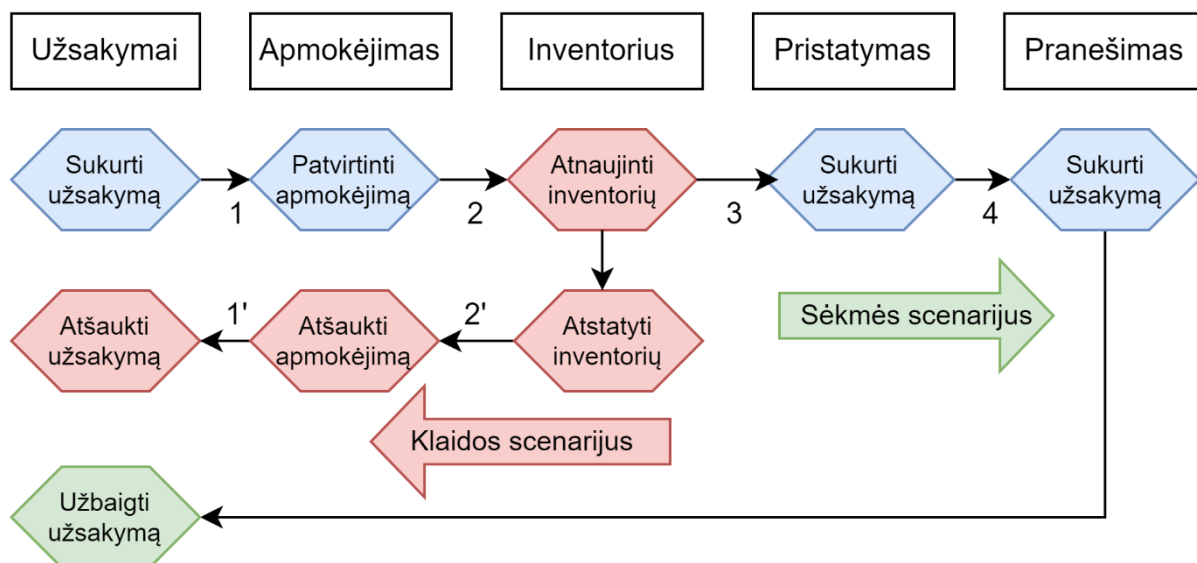
Strangler Fig šablonas naudojamas monolitinių sistemų migravimui į mikroservisų architektūrą. Įgyvendinant šį šabloną tarp kliento ir egzistuojančios sistemos paleidžiamas dedikuotas mikroservisas, kuris nekeičia užklauskų ir tiesiog veikia kaip tarpininkas [14]. Kai naujoje sistemoje dalis funkcionalumo yra paruošta, užklauskos yra nukreipiamos į naują sistemą. Taip nuosekliai vis didesnė užklauskų dalis yra siunčiama į naują sistemą, kol visas senos sistemos funkcionalumas yra išmigruotas. Toks metodas leidžia modernizuoti sistemą, palaikant sistemos veikimą ir sumažinant sistemos neveikimo riziką.

1.3.3. Transakcijų valdymas mikroservisų architektūroje

Monolitinėje architektūroje transakcijų įvykdymas nėra sudėtinga užduotis [15]. Kadangi naudojama viena duomenų bazė, galima užtikrinti, kad duomenų atnaujinimas nebus įvykdytas dalinai. Tačiau mikroservisai naudoja atskiras duomenų bazes, todėl nėra galimybės vienu metu atnaujinti susijusių duomenų, paskirstytų per skirtingas duomenų bazes. Todėl reikalingi būdai, kurie leistų užtikrinti, kad duomenų atnaujinimo užklauskos nebūtų įvykdytos tik dalinai.

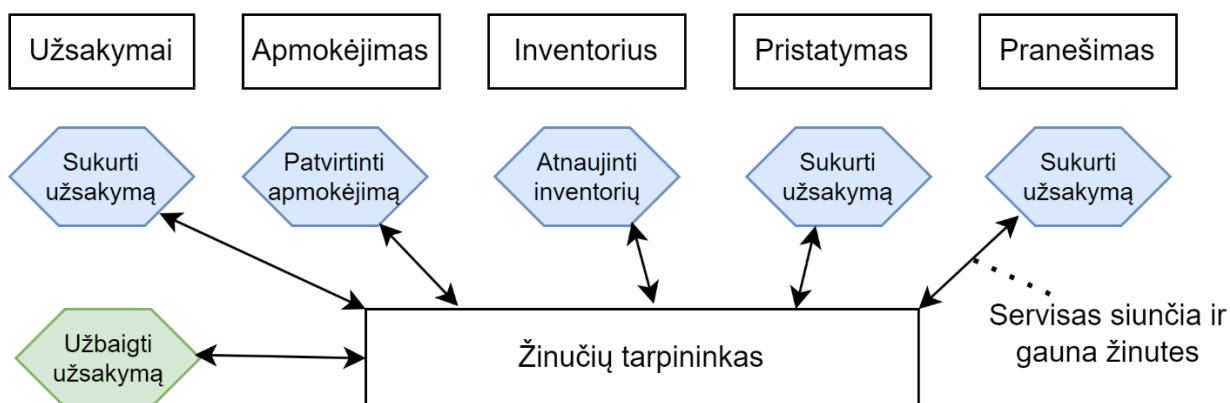
SAGA šablonas skirtas šiai problemai spręsti [15]. Šis šablonas siūlo, kad kiekviena operacija turėtų turėti priešingą operaciją. 1.3 pav. pateiktas *SAGA* šablono taikymo pavyzdys. Kiekviena operacija vykdoma viena po kitos nuosekliai. Jeigu vykdant operaciją įvyko klaida, tuomet įvykdomos atstatymo operacijos atgaline tvarka. Pavyzdžiui, jeigu

užsakant prekę buvo sukurtas užsakymas, tačiau atnaujinant inventorių įvyko klaida, tuomet apmokėjimas ir užsakymas yra atšaukiami.



1.3 pav. SAGA šablonas [16]

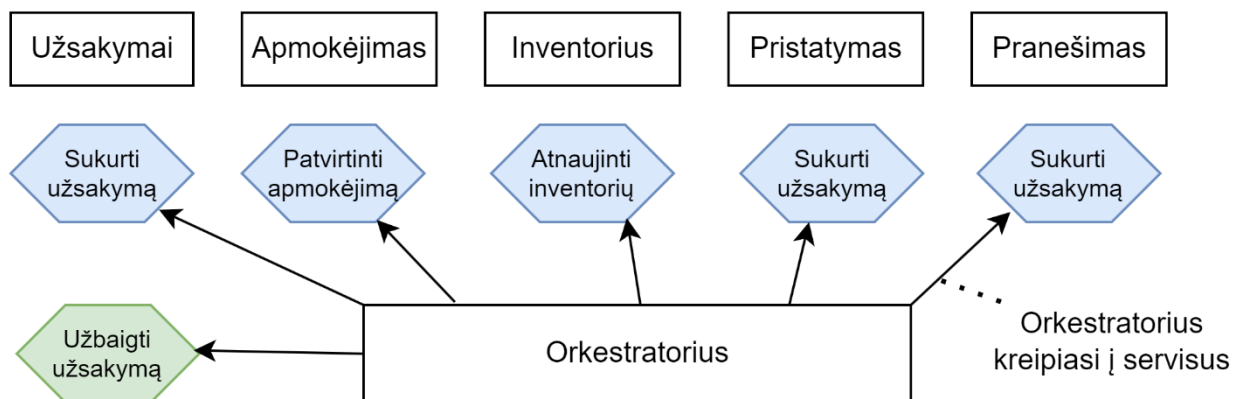
SAGA šablonas gali būti įgyvendintas 2 būdais: choreografijos ir orkestravimo [15]. Kai naudojamas choreografijos metodas, mikroservisai komunikuoja naudojant žinučių tarpininką tam, kad praneštų, kuris žingsnis buvo įvykdytas. 1.4 pav. pateikta principinė choreografijos metodo schema. Pavyzdžiui, kai sukuriamas užsakymas, užsakymų servisas siunčia įvykį į žinučių tarpininką, kad buvo sukurtas užsakymas. Tuomet apmokėjimų servisas gauna žinutę, kad buvo sukurtas užsakymas, ir naujam užsakymui patvirtina apmokėjimą.



1.4 pav. SAGA choreografijos šablonas [16]

Kai naudojamas orkestravimo metodas, reikalingas papildomas mikroservisas, kuris atsakingas už operacijų atlikimo valdymą. Šis servisas vadinamas orkestratoriumi. 1.5 pav. pateikta principinė orkestravimo metodo schema. Orkestratorius turi logiką, kurioje apibrėžiama, kokia tvarka vykdomos atskirų mikroservisų operacijos ir tikrina, kokį rezultatą

gražina mikroservisai [15]. Įvykus klaidai, orkestratorius kreipiasi į kitus mikroservisus, kad būtų atstatyta tokia sistemos būseną, kokia buvo prieš vykdant užklausą.



1.5 pav. SAGA orkestravimo šablonas [16]

Taikant choreografijos metodą, pasiekiamas didesnis sistemos atsparumas, tačiau šį metodą yra sunkiau įgyvendinti, nes kiekvienas servisas turi turėti logiką žinučių priėmimui ir siuntimui [16]. Orkestravimo metodą paprasčiau įgyvendinti, nes visa valdymo logika yra viename orkestravimo servise, tačiau naudojama sinchroninė komunikacija, todėl sistema gali būti neatspari laikiniams trikdžiams.

Naudojant mikroservisų architektūrą siekiama sukurti sistemą, kurioje galima greitai diegti pokyčius, nereikalaujant kiekvieną kartą iš naujo diegti visą sistemą. Šiam poreikiui įgyvendinti kuriami servisi, kurie turi atskiras duomenų bazines, tarpusavyje komunikuoja tinklo lygmenyje. Tačiau dėl to atsiranda papildomų iššūkių duomenų integralumui užtikrinti. Taigi, SAGA šablonas padeda užtikrinti duomenų integralumą, tačiau didina mikroservisų realizavimo ir testavimo sudėtingumą [15].

1.3.4. Mikroservisų testavimo strategijos

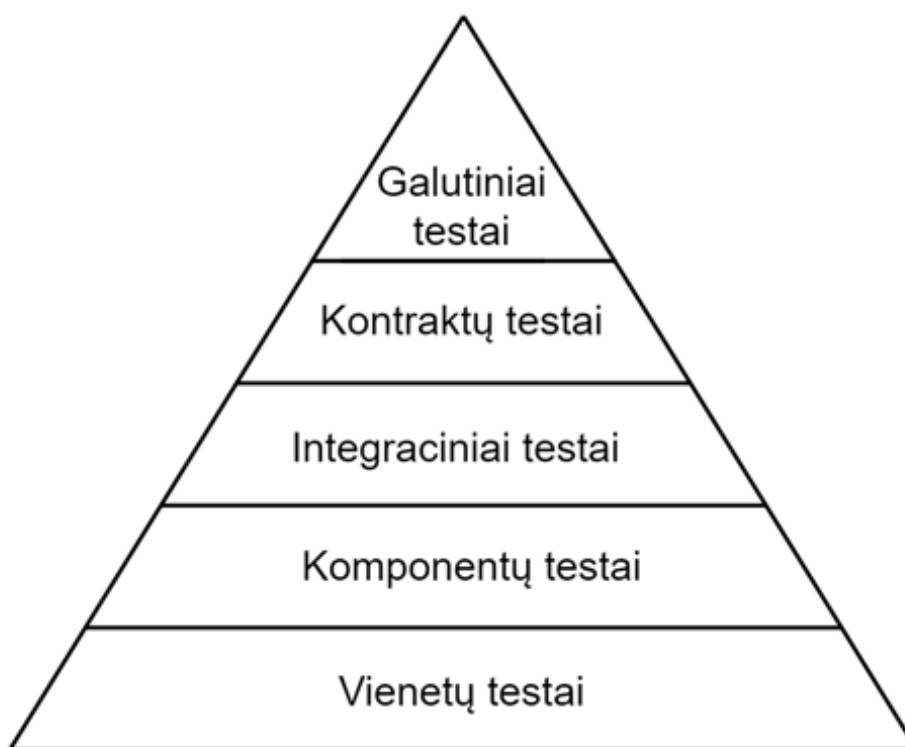
Testavimas yra svarbus etapas programinės įrangos kūrimo procese. Testavimo metu tikrinama, ar sistema atitinka funkcinis ir nefunkcinis sistemos reikalavimus, todėl testavimas padeda užtikrinti, kad buvo sukurta tai, ko reikalavo projekto užsakovas [17]. Sistemos testavimas rankiniu būdu gali užimti daug laiko, testavimas vyksta lėčiau ir sunku užtikrinti, kad testų scenarijai bus įvykdyti taip pat, kai testo vykdymas yra kartojamas kelis kartus. Todėl siekiama kuo didesnę sistemos dalį padengti automatiniais testais [17].

Kuriant testų scenarijus galima remtis tiek sistemos specifikacija, tiek pačia realizacija. Juodosios dėžės testavimas remiasi tik kuriamos sistemos specifikacija, neturint informacijos apie jos vidų [18]. Tokia testavimo technika tikrina, ar sistema atlieka numatytas funkcijas, kaip apibrėžta. Baltosios dėžės testavimas remiasi tik žiniomis apie sistemos realizaciją, kuriant testavimo scenarijus. Taikant baltosios dėžės techniką siekiama, kad būtų ištestuoti visi kode aprašomi scenarijai [18]. Taikant juodosios dėžės testavimą patikrinama, ar sistema atlieka specifikacijoje aprašytą funkcionalumą. Tačiau, jei specifikacija yra nepilna ir sistema turi papildomos logikos, taikant juodosios dėžės testavimo techniką, yra tikimybė praleisti testavimo scenarijus, kurie yra aprašyti

programiniame kode [18]. Taikant baltosios dėžės testavimą visi programiniame kode aprašomi scenarijai yra pastebimi, tačiau nepatikrinama, ar sistema atlieka visas specifikacijoje numatytas funkcijas. Taikant abi testavimo technikas galima patikrinti, ar sistema atitinka specifikaciją ir ar padengti visi kodo scenarijai.

Regresinis testavimas taikomas siekiant patikrinti, ar nauji kodo pakeitimai nesugadino egzistuojančio sistemos funkcionalumo [17]. Atliekant kodo pakeitimus ar pridėdant naują funkcionalumą išlieka tikimybė, kad kitoms sistemos dalims bus padaryta įtaka. Todėl svarbu, kad visi testai, kurie buvo sukurti prieš atliekant pakeitimus, būtų įvykdomi su sėkmingais rezultatais. Atliekant regresinį testavimą gali būti pasirenkama leisti visus testus, tačiau tai gali reikalauti daug laiko ir resursų [17]. Siekiant minimizuoti regresinio testavimo darbą svarbu parinkti testų scenarijus toms sistemos dalims, kurios galėjo būti įtakotos.

Sistemų testavimui naudojami kelių tipų automatiniai testai, kurie testuoja skirtingas sistemos dalis [17]. 1.6 pav. pateikta standartinė testavimo piramidė, kurioje vaizduojami skirtingų tipų testai. Ne išimtis ir mikroservisų architektūra, kuriai kaip ir kitoms sistemoms reikalingi kelių tipų testai [19]. Piramidės apačioje vaizduojamų tipų testus yra lengva kurti, nes jie testuoja mažą konkrečią sistemos dalį, todėl jų kiekis yra didžiausias. Piramidės viršuje esančių testų kūrimas užtrunka ilgiau, tačiau šie testai padengia didesnę sistemos dalį.



1.6 pav. Testavimo piramidė [17]

Automatinių testų tipai, naudojami testuojant mikroservisus [17]:

- Vienetų testai - testuoja atskiras izoliuotas sistemos dalis [20]. Vienas testas dažniausiai testuoja vieno metodo funkcionalumą. Jeigu testuojama klasė naudoja kitas klases, tai jos gali būti pakeičiamos netikromis klasės realizacijomis tam, kad

- būtų testuojamas tik pasirinktas metodas. Vienetų testai padeda užtikrinti, kad klasės teisingai atlieka reikalingas operacijas [17];
- Komponentų arba funkciniai testai - patikrina serviso, kaip izoliuotos sistemos veikimą. Naudojant šiuos testus siekiama patikrinti, ar vidinė realizacija veikia teisingai kaip visuma. Todėl siekiant testuoti servisą izoliuotai, susiję servisi pakeičiami netikrais servisi su iš anksto nustatytais atsakymais. Šiais testais siekiama patikrinti, ar servisas teisingai apdoroja duomenis ateinančius iš kitų servisų ir ar gražinamas tikėtinas atsakymas [17];
 - Integraciniai testai - naudojami patikrinti, ar galima pasiekti kitus servisius [21]. Pavyzdžiui, jeigu kito serviso prieigos taškai yra apsaugoti, tuomet su integraciniais testais galima patikrinti, ar testuojamo serviso *HTTP* klientas yra sukonfigūruotas teisingai, kad galėtų kreiptis į kito serviso galutinius taškus [17];
 - Kontraktų testai - naudojami tam, kad užtikrintų sėkmingą komunikaciją tarp servisų. Servisi turi naudoti vienodą kontraktą, kurį apibrėžia duomenis teikiantis servisas. Kadangi kiekvienas servisas yra nepriklausomas, kontrakto pakeitimas nėra iš karto pastebimas kaip klaida. Todėl atlikus kontrakto pakeitimą viename servise, galima sukurti nenumatytų problemų kituose servisuose [22];
 - Galutiniai testai (*angl. end-to-end tests*) - tikrina sistemos kaip visumos veikimą, todėl naudojami tikri servisi. Nors šie testai suteikia daugiausiai pasitikėjimo, tačiau juos gali būti sunku palaikyti [19]. Pavyzdžiui, jeigu testuose tikimasi konkrečių reikšmių, tuomet bet kokie menki pasikeitimai duomenų bazėje gali sugadinti testus [17].

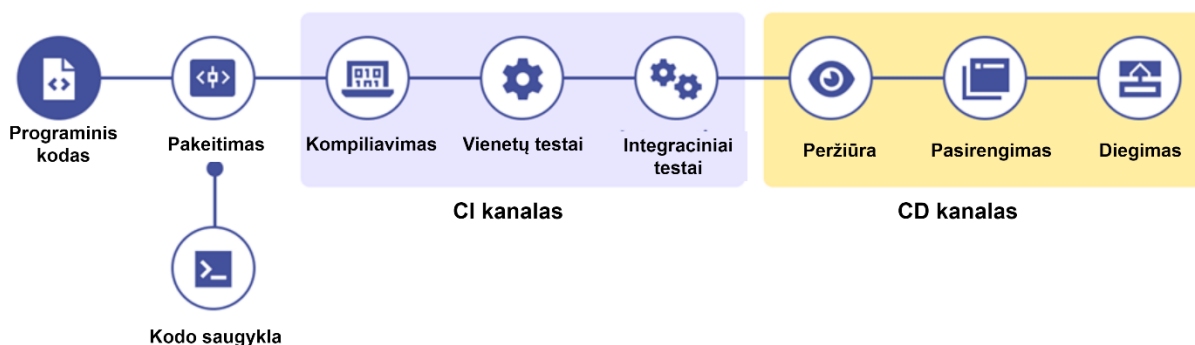
Ne visi testų tipai aktualūs sprendžiamai problemai. Vienetų testai nepriklauso nuo architektūros stiliaus ir jų poreikis nesikeičia. Taikant vienetų testus galima greitai patikrinti individualių klasių funkcionalumą [20]. Integraciniai testai skirti patikrinti, ar servisi yra tinkamai sukonfigūruoti [21]. Galutiniai testai tikrina, ar visi servisi kartu veikia teisingai. Galutiniai testai mikroservisų architektūroje nesiskiria nuo kitose architektūrose taikomų galutinių testų, nes tikrinama, ar įdiegta sistema gražina tikėtiną rezultatą [17]. Lyginant su monolitine architektūra, komponentų ir kontraktų testų poreikis yra didesnis mikroservisų architektūroje. Kadangi egzistuoja atskiri servisi, komunikuojantys tinklo lygmenyje, reikalingi testai, kurie tikrintų, kad servisi laikosi *API* sąsajos kontraktų. Taip pat reikalinga, kad serviso funkcionalumas būtų testuojamas izoliuotai naudojant komponentų testus. Jeigu vykdant testus testuojamo serviso atsakymą gali paveikti kiti servisi, tuomet pažeidžiamas nepriklausomumo principas. Taigi, taikant mikroservisų architektūrą atsiranda didesnis poreikis naudoti kontraktų ir komponentų testus, lyginant su monolitine architektūra, kad būtų galima izoliuotai ištestuoti mikroserviso elgseną.

1.3.5. Nuolatinė integracija ir nuolatinis diegimas

Kuriant mikroservisius didelis dėmesys skiriamas nuolatinės integracijos ir nuolatinio pristatymo procesams. Nuolatinė integracija apima nuolatinį kodo integravimą į bendrą kodo saugyklą naudojant *Git* įrankį. Nuolatinė integracija leidžia programuotojams gauti naujausius kodo pakeitimus ir dalintis savaisiais [23]. Kad būtų galima užtikrinti, kad kodo pakeitimai atitinka kokybės reikalavimus ir nesugadina egzistuojančios sistemos, reikalingi automatiniai sistemos kompiliavimai (*angl. automated builds*). Vykstant šiems procesams sistemos programinis kodas yra sukompiliuojamas, taip pat gali būti patikrinama, ar nėra stiliaus klaidų ir ar sistema gali būti paleista. Taip pat vykdomi automatiniai testai.

Automatiniai testai atlieka svarbią funkciją, jie užtikrina, kad kodo pakeitimai nesugadino egzistuojančių funkcijų [23]. Tai suteikia pasitikėjimo ir leidžia komandoms greitai integruoti pakeitimus, nereikalaujant rankinio testavimo kiekvieno pakeitimo metu. Kiekviena nauja sistemos versija yra unikali ir identifikuojama.

Nuolatinio pristatymo siekiama sudaryti tokias sąlygas, kad sistemos atnaujinimus būtų galima diegti lengvai ir greitai. Kad tai būtų galima įgyvendinti, reikalingas automatizuotas diegimo procesas. Integravus sistemos pakeitimus į kodo saugyklą ir įsitikinus, kad testai nerado klaidų, gali būti vykdomas diegimo procesas, kuris atliekamas naudojant sukurtus scenarijus ar įrankius [23]. 1.7 pav. pateiktoje diagramoje vaizduojamas apibendrintas integravimo ir diegimo procesas.



1.7 pav. Kodo integravimo procesas [24]

Automatiniai testai atlieka svarbų vaidmenį nuolatinio pristatymo ir nuolatinės integracijos procese. Testai padeda užtikrinti, kad sistemos funkcijos veikia tinkamai po kodo pakeitimų ir kad yra saugu diegti naują sistemos versiją [23]. Kuo anksčiau pastebimos klaidos, tuo lengviau jas ištaisyti, ir sutaupoma laiko, nes išvengiama neteisingos sistemos versijos diegimo. Vienetų, komponentų ir kontraktų testai gali būti vykdomi nuolatinės integracijos procese prieš sistemos diegimą. Galutinius testus galima paleisti tik įdiegus sistemą nuolatinio pristatymo proceso pabaigoje. Taigi, komponentų ir vienetų testai padeda anksti pastebėti klaidas nuolatinės integracijos procese, todėl padeda išlaikyti spartų mikroservisų kūrimo procesą.

1.3.6. BDD ir Gherkin karkasų taikymas testavime

Automatiniai testai gali būti kuriami įvairiais būdais ir populiariausios programavimo kalbos turi įvairių automatinių testų karkasų, kuriais galima kurti automatinius testus. Mikroservisų testavimui vis dažniau naudojamas BDD (angl. Behaviour Driven Development) metodas [25]. Ši metodika siekia pagerinti bendradarbiavimą tarp programuotojų, testuotojų ir verslo atstovų suteikiant karkasą sistemos funkcionalumui aprašyti [26]. Scenarijų aprašymui gali būti naudojama Gherkin kalba. Populiariausios programavimo kalbos turi karkasus, leidžiančius vykdyti Gherkin scenarijus, kai patys žingsniai realizuojami naudojant konkrečią programavimo kalbą [27]. Gherkin kalba aprašomi scenarijai susideda iš šių žingsnių [26]:

- Given – nusako testo kontekstą;
- When – nusako įvykį;
- Then – patikrina, ar įvyko numatytas rezultatas;

- *And, But* – jungia kelis to pačio tipo žingsnius.

BDD scenarijai yra aprašomi *feature* failuose [25]. Kiekvienas *feature* failas susideda iš vieno arba daugiau scenarijų. 1.8 pav. pateikiamas pavyzdinis *feature* failas su dviem pavyzdiniais scenarijais. Scenarijai aprašo tikėtiną sistemos funkcionalumą. Pirmajame scenarijuje aprašoma, jeigu projekto atnaujinimo metu įvedami teisingi duomenys, projektas yra atnaujinamas, o antrame scenarijuje aprašoma, kad jeigu įvedami neteisingi duomenys, projekto negalima atnaujinti. Vien tik scenarijų aprašymų nepakanka, reikalinga kiekvieną žingsnį įgyvendinti aprašant programinį kodą, kuris įvykdo reikiamus veiksmus ir patikrinimus. Tą patį žingsnį galima realizuoti tik vieną kartą, todėl tas pats žingsnis skirtinguose scenarijuose veiks taip pat [26].

Feature: Update project

Scenario: Update project with valid fields

Given I have logged in as project manager

And I have navigated to projects page

When I open edit project page

And I update all project fields and submit

Then project information should be updated

Scenario: Try to update project with invalid fields

Given I have logged in as project manager

And I have navigated to projects page

When I open edit project page

And I update all project fields with invalid characters

Then I should not be allowed to update project information

1.8 pav. Svetainės testavimo pavyzdiniai scenarijai

Testai parašyti taikant *BDD* gali reikalauti daugiau priežiūros, nes bet koks funkcinis ar *API* pasikeitimas gali paveikti daug scenarijų [25]. Šie testai testuoja atskiro serviso ar komponento veikimą. Net mažas logikos pokytis gali sugadinti testą. Tai ypač aktualu, jei sistema kuriama taikant *Agile* metodologiją, kurioje iteratyvus procesas gali įnešti daug pokyčių, tai lemia dažnus funkcionalumo pakeitimus, reikalaujančius atnaujinti testus [25]. Vienetų testai testuoja mažesnes kodo dalis, todėl sistemos pokyčiai daro įtaką tik tiesiogiai susijusiems testams [20]. Tačiau vienetų testai dažnai būna glaudžiai susiję su vidinių klasių struktūra ir simuliuojama priklausomų klasių elgsena. Dėl to vidinės struktūros pakeitimai priverčia atnaujinti ir pačius testus, net kai neatliekami logikos pakeitimai. Funkciniai testai, priešingai, tikrina tik galutinį rezultatą, todėl yra atsparesni kodo restruktūrizavimui.

1.3.7. Didžiųjų kalbos modelių taikymas testų generavime

Mikroservisų sistemų kūrimo ir testavimo procesuose pasitelkiami dirbtinio intelekto įrankiai, siekiant automatizuoti įvairių artefaktų generavimą, įskaitant testavimo scenarijus [28]. Didieji kalbos modeliai (*angl. Large Language Models, LLM*) geba analizuoti natūralios kalbos reikalavimus, techninę dokumentaciją bei egzistuojantį programinį kodą ir pagal tai automatiškai generuoti testų atvejus [29]. *LLM* gali būti taikomi tiek vienetinių, tiek integracinių ar galutinių testų kūrimui, taip sumažinant rankinio darbo poreikį ir pagreitinant testavimo procesą [30].

Be to, *LLM* gali būti integruojami į testavimo orkestravimo procesus, kur jie ne tik generuoja testus, bet ir interpretuoja gautus rezultatus, identifikuodami neatitikimus ar potencialias regresijas [29]. Tai leidžia aptikti integracijos problemas ir sumažinti rankinio testavimo poreikį. Taip pat *LLM* veikimas priklauso nuo turimos žinių bazės ir konteksto, todėl jų gebėjimas aptikti veiklos srities specifines klaidas gali būti ribotas. Nors *LLM* gali būti taikomi testų generavimui, jų taikymas turi tam tikrų trūkumų. Pirmiausia, *LLM* generuojami testai yra nedeterministiniai, todėl skirtingais generavimo atvejais, testai gali nesutapti, nors naudojamas tas pat įvesties kontekstas [31]. Taip pat *LLM* išvestis ir jos struktūra yra sunkiau kontroliuojama, todėl gali atsirasti perteklinių arba nenuoseklių scenarijų bei haliucinacijos, t. y. faktiškai neteisingos arba nepagrįstos informacijos generavimas [32]. Todėl efektyviausios testavimo strategijos dažnai derina tradicinius metodus su *LLM* generuotais testais, siekiant užtikrinti tiek pilnumą, tiek kokybę [29].

Integruotose programavimo aplinkose (*angl. Integrated Development Environment, IDE*), tokiose kaip *Visual Studio Code*, *JetBrains Rider*, *Visual Studio* yra integruoti įskiepai, kaip *GitHub Copilot*, skirti naudotis didžiais kalbos modeliais tiesiogiai per programavimo aplinką. Naudojant įskiepius galima užduoti klausimus arba naudoti agentą. Agentai gali veikti kaip interaktyvus pagalbininkas, gebantis generuoti kodo fragmentus, teikti pasiūlymus, analizuoti kontekstą [33]. Siekiant, kad tokie agentai galėtų sąveikauti su skirtingais projekto komponentais ir išoriniais įrankiais, naudojami standartizuoti komunikacijos mechanizmai. Vienas iš jų yra *MCP* (*angl. Model Context Protocol*) protokolas, skirtas perduoti informaciją tarp didžiųjų kalbos modelių, *IDE* aplinkos ir papildomų įrankių [34]. *MCP* leidžia agentams pasiekti projekto failus, vykdyti komandas ir naudoti resursus per vieną sąsają, taip sudarant sąlygas platesniam jų pritaikymui programavimo ir testavimo kontekste. *MCP* protokolas taip pat leidžia agentams *IDE* aplinkoje palaikyti ilgalaikį kontekstą tarp skirtingų sesijų ir komponentų, taip išplečiant agentų gebėjimus integruotis su įvairiomis programinės įrangos sistemomis ir automatizuoti veiksmus [35].

Taigi, dirbtinio intelekto įrankiai gali ženkliai pagreitinti testavimo procesą ir sumažinti rankinio darbo poreikį. Siekiant efektyviai išnaudoti šiuos įrankius testavimo kontekste, svarbu turėti standartizuotą jų komunikacijos ir orkestravimo mechanizmą. *MCP* protokolas leidžia agentams sąveikauti su išoriniais įrankiais ir sistemomis per standartizuotą sąsają, todėl gali būti taikomas automatizuojant testų generavimą bei vykdymą mikroservisų architektūroje.

1.3.8. Testų kokybės įvertinimo analizė

Testų kokybės įvertinimas leidžia nustatyti, ar sugeneruoti arba sukurti testai gali aptikti programinės įrangos defektus ir užtikrinti sistemos patikimumą. Aukštos kokybės testai turėtų ne tik padengti reikalavimus ir sistemos funkcionalumą, bet ir būti pakankamai jautrūs klaidoms, susijusioms su verslo logika, duomenų validacija ir komponentų tarpusavio sąveika. Šie principai taikomi įvairiose programinės įrangos architektūrose, taip pat ir mikroservisų architektūroje.

Vienas pagrindinių testų kokybės vertinimo kriterijų yra kodo padengimas (*angl. code coverage*)[36]. Funkcinis padengimas parodo, kokia sistemos funkcionalumo dalis yra

tikrinama testais, o kodo padengimas (pvz., eilučių, šakų ar sąlygų padengimas) leidžia įvertinti, kiek testai tikrina skirtingas programinio kodo vykdymo šakas. Tačiau didelis padengimo procentas nebūtinai garantuoja aukštą testų kokybę, nes testai gali neaptikti loginių klaidų ar netinkamų būsenos pakeitimų [37].

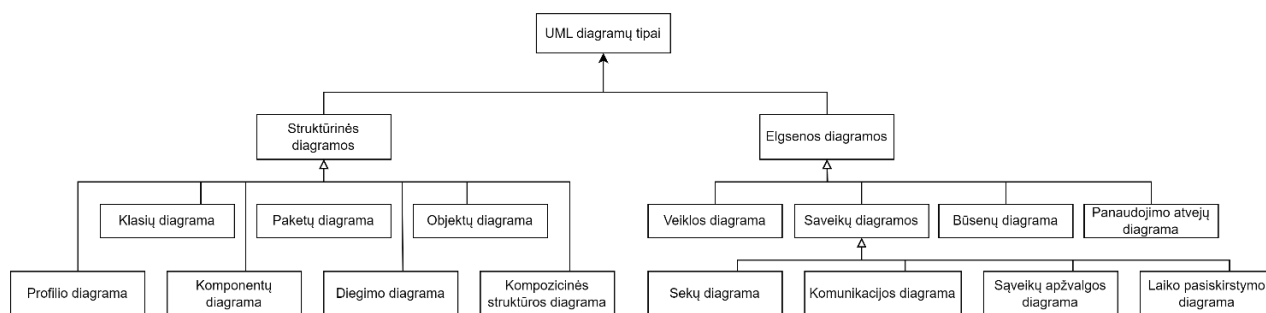
Kitas svarbus aspektas yra gebėjimas aptikti tikrus sistemos defektus. Dažnai šiam tikslui taikomas mutacinis testavimas, kurio metu į programinį kodą dirbtinai įterpiami smulkūs pakeitimai (mutacijos), o testų kokybė vertinama pagal tai, kiek defektų testai sugeba aptikti [38]. Aukštas mutacijų aptikimo rodiklis rodo, kad testai yra jautrūs net ir subtiliems elgsenos pokyčiams. Nors mutacinis testavimas leidžia patikimai įvertinti testų kokybę, tačiau reikalauja daug laiko ir pastangų [37][38].

Taip pat testai turi būti deterministiniai, pakartojami ir pateikiantys aiškias prielaidas bei laukiamus rezultatus [36]. Netikslūs arba neaiškiai suformuluoti testai gali sukelti klaidingus teigiamus arba klaidingus neigiamus rezultatus, taip sumažindami pasitikėjimą testavimo procesu.

Taigi, kodo padengimas ir mutacinis testavimas yra vienas kitą papildantys kokybės vertinimo metodai: pirmasis parodo, kokia kodo dalis yra vykdoma testų metu, o antrasis parodo, ar testai yra pakankamai jautrūs aptikti elgsenos pakeitimus. Abu šie metodai gali būti taikomi sugeneruotų testų kokybei įvertinti: kodo padengimas leistų nustatyti, kokią serviso kodo dalį padengia sugeneruoti testai, o mutacinis testavimas – ar jie geba aptikti dirbtinai įvestas klaidas.

1.4. UML ir jos taikymas mikroservisų architektūros sistemoms modeliuoti

UML (angl. *Unified Modeling Language*) yra modeliavimo ir specifikacijų kūrimo kalba, skirta specifiuoti, atvaizduoti ir konstruoti objektinių programų dokumentus [39]. UML suteikia galimybę atvaizduoti sistemos ar proceso veikimą ir struktūrą. 1.9 pav. pateikti pagrindiniai UML diagramų tipai: elgsenos ir struktūrinės diagramos. Elgsenos diagramos skirtos atvaizduoti, kokius veiksmus atlieka sistemos elementai, o struktūros diagramos vaizduoja, kaip sistemos elementai yra susiję vienas su kitu.



1.9 pav. UML diagramų hierarchija [39]

Struktūrinėms diagramoms priskiriami šie diagramų tipai [39]:

- klasių diagrama – vaizduoja sistemos klases, jų atributus, operacijas ir ryšius su kitomis klasėmis.

- paketų diagrama – vaizduoja, kaip elementai yra sugrupuoti. Paketų diagramos padeda struktūrizuoti sudėtingas klasių diagramas, suskirstant klases į paketus.
- komponentų diagrama – naudojama atvaizduoti, kaip sistemos komponentai yra susiję tarpusavyje, padeda atvaizduoti statinę sistemos realizaciją, nurodant, kokias sąsajas teikia ir naudoja kiekvienas komponentas.
- objektų diagrama – skirta atvaizduoti sistemą konkrečiu momentu, nurodant, kokie objektai tuo metu egzistuoja. Ši diagrama panaši į klasių diagramą. Kadangi vaizduojami objektai, o ne klasės, papildomai nurodomas sukurto objekto pavadinimas ir atributų reikšmės.
- kompozicinės struktūros diagrama – vaizduoja vidinę klasės struktūrą ir vidinių elementų ryšius.
- diegimo diagrama – vaizduoja sistemos diegimo konfigūraciją nurodant, kokie mazgai sudaro fizinę sistemą.

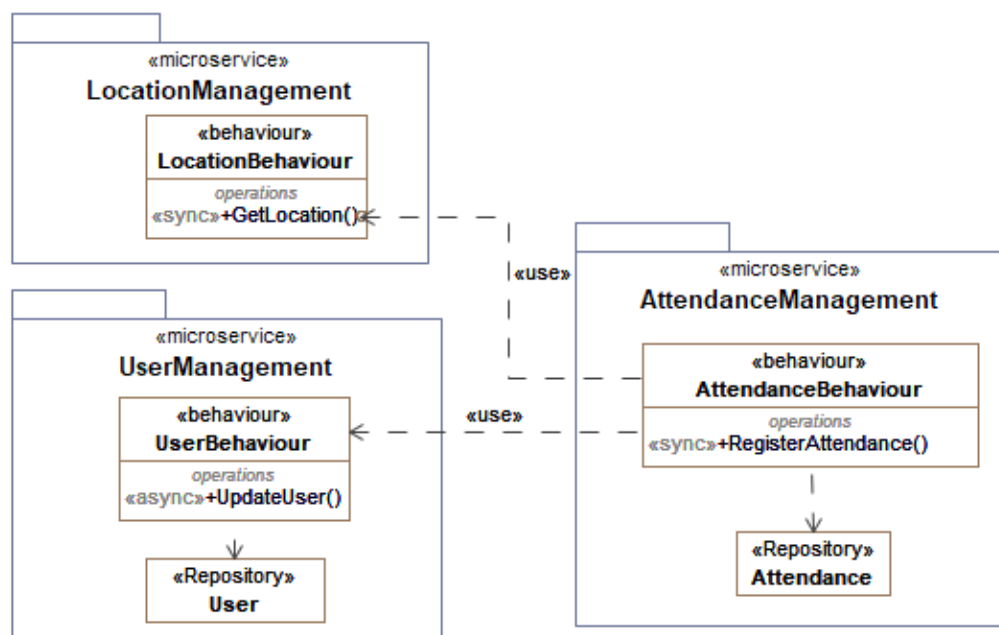
Elgsenos diagramoms priskiriami šie diagramų tipai [39]:

- sekų diagrama – vaizduoja elementų sąveiką tam tikru laiko momentu. Diagramoje nurodoma pranešimų seka, aktoriai, objektai ir jų gyvavimo trukmė.
- veiklos diagrama – vaizduoja veiksmų eigą, nurodant sistemos srautus ir veiklų sekas.
- būsenų diagrama – vaizduoja galimas objekto būsenas ir kokiomis sąlygomis pereinama nuo vienos būsenos prie kitos.
- panaudojimų atvejų diagrama – vaizduoja funkcinis sistemos reikalavimus, kuriuos gali atlikti sistemos aktoriai.
- laiko diagrama – vaizduoja objekto būsenų pokyčius laike, taip suteikiant galimybę matyti, kaip keitėsi objekto būsenos.
- sąveikų apžvalgos diagrama – vaizduoja, kaip skirtingi sistemos fragmentai gali būti inicializuoti skirtingais scenarijais. Diagrama panaši į veiklos diagramą, tačiau sąveikos fragmentai vaizduojami sekų diagramos forma.
- komunikacijos diagrama – panašiai kaip sekų diagrama, vaizduoja sąveiką tarp objektų. Ši diagrama savo struktūra panaši į objektų diagramą, papildomai vaizduojamos išskviečiamos operacijos tarp objektų.

Siekiant užtikrinti *UML* modelių suderinamumą tarp skirtingų įrankių, jie gali būti saugomi tam skirtu formatu. *XMI* (angl. *XML Metadata Interchange*) yra standartizuotas *XML* pagrindu sukurtas formatas, skirtas modelių duomenų apsikeitimui tarp skirtingų modeliavimo įrankių [40]. *UML* diagramų kontekste *XMI* naudojamas *UML* modelių struktūrai, elementams ir jų tarpusavio ryšiams aprašyti tekstiniu formatu. *UML* diagramos modeliavimo įrankiuose dažnai kuriamos grafiškai, tačiau jų vidinė reprezentacija gali būti saugoma *XMI* faile, kuriame aprašomi tokie elementai kaip klasės, objektai, pranešimai, asociacijos ar sekų diagramos sąveikos. Toks formatas leidžia *UML* modelius eksportuoti, importuoti ir automatiškai apdoroti skirtingose sistemose neprarandant modelio struktūros [39, 40].

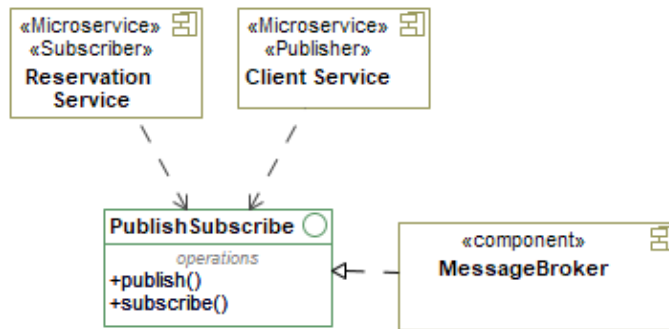
Nepriklausomai nuo naudojamos architektūros, *UML* diagramos gali būti naudojamos atvaizduoti sistemos funkcionalumą arba struktūrą pasirinktame abstrakcijos lygyje [3]. Naudojant *UML* diagramas, programų sistemų inžinieriai gali vizualizuoti ir patikrinti programinės įrangos dizainą prieš sistemos įgyvendinimą, kurio metu pokyčiai tampa

sudėtingi ir brangūs [3]. Taikant mikroservisų architektūrą svarbu tinkamai suprojektuoti sistemą, nes priešingu atveju sukurta sistema gali neturėti šios architektūros teikiamų privalumų [2]. Mikroservisų vidinei struktūrai modeliuoti galima naudoti klasių diagramas [3]. UML neturi numatytų stereotipų, kurie yra pritaikyti mikroservisų architektūrai. Galima susikurti naujus stereotipus, kurie nusakytų esybių funkcijas. 1.10 pav. pavaizduota pavyzdinė klasių diagrama, vaizduojanti galimą mikroservisų ir jų priklausomybių vaizdavimą. Kiekvienas atskiras mikroservisas yra išskiriamas kaip atskiras paketas su stereotipu *microservice*. Klasės, kurios turi bendrauti su kitais mikroservais išskiriamos *behaviour* stereotipu. Siekiant nurodyti, koks komunikacijos tipas naudojamas tarp mikroservisų, naudojami stereotipai *sync*, *async*, kurie nurodo, ar komunikacija vyksta sinchroniškai, ar asinchroniškai.



1.10 pav. Mikroservisų architektūros pavyzdinis vaizdavimas klasių diagrama [3]

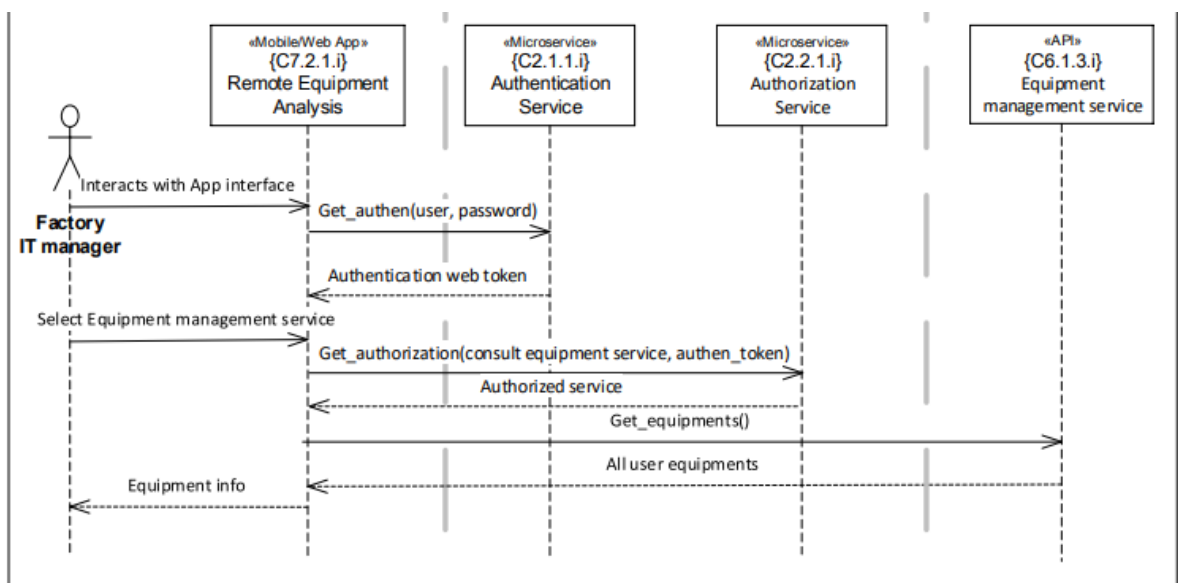
Jeigu sistema projektuojama aukštesniame abstrakcijos lygyje, klasių diagrama gali būti per daug detali. Todėl vietoj klasių diagramos gali būti naudojama komponentų diagrama, kai siekiama parodyti, kokie mikroservisai egzistuoja ir koku būdu jie komunikuoja [41]. 1.11 pav. pateikiama pavyzdinė komponentų diagrama, kurioje vaizduojami servisai, komunikuojantys per žinučių tarpininką. Taip pat, kaip ir naudojant klasių diagramą, galima sukurti stereotipus skirtingoms mikroservisų rolėms. Pavyzdžiui žinutes siunčiančiam servisui galima suteikti *publisher* stereotipą, o žinutes vartojančiam – *subscriber* [41].



1.11 pav. Mikroservisų, komunikuojančių asinchroniškai, komponentų diagrama [41]

Sistemos modelis gali būti naudojamas, generuojant sistemos programinį kodą [3, 39]. Mikroservisai dažnai turi panašią architektūrą [4], todėl galima generuoti pradinis mikroservisų projektų skeletus iš sukurtų *UML* diagramų. Toks sprendimas leidžia išlaikyti panašią vidinę struktūrą tarp mikroservisų ir programuotojai gali skirti daugiau laiko sistemos verslo logikos įgyvendinimui [41].

UML sekų diagramos naudojamos atvaizduoti, kaip skirtingi komponentai sąveikauja tarpusavyje. Sekų diagramose galima atvaizduoti komunikaciją tarp sistemos elementų skirtinguose abstrakcijos lygiuose. Taip pat galima vaizduoti, kokia tvarka sistemos elementai sąveikauja tarpusavyje ir kaip aktorius bendrauja su sistema, neatskleidžiant sistemos įgyvendinimo detalių. Ne išimtis ir mikroservisų architektūra, naudojant sekų diagramas, galima gauti vizualų supratimą apie tai, kaip skirtingi mikroservisai sąveikauja tarpusavyje, kokia tvarka siunčiamos užklauskos ir kaip komunikacijos eiga vyksta išskaidytoje sistemoje. Tai naudinga projektuojant mikroservisų architektūrą ir analizuojant sistemos elgseną [42]. 1.12 pav. vaizduojama, kaip naudotojas prisijungia prie sistemos, ir kokios užklauskos buvo atliktos kviečiant skirtingus mikroservisus.



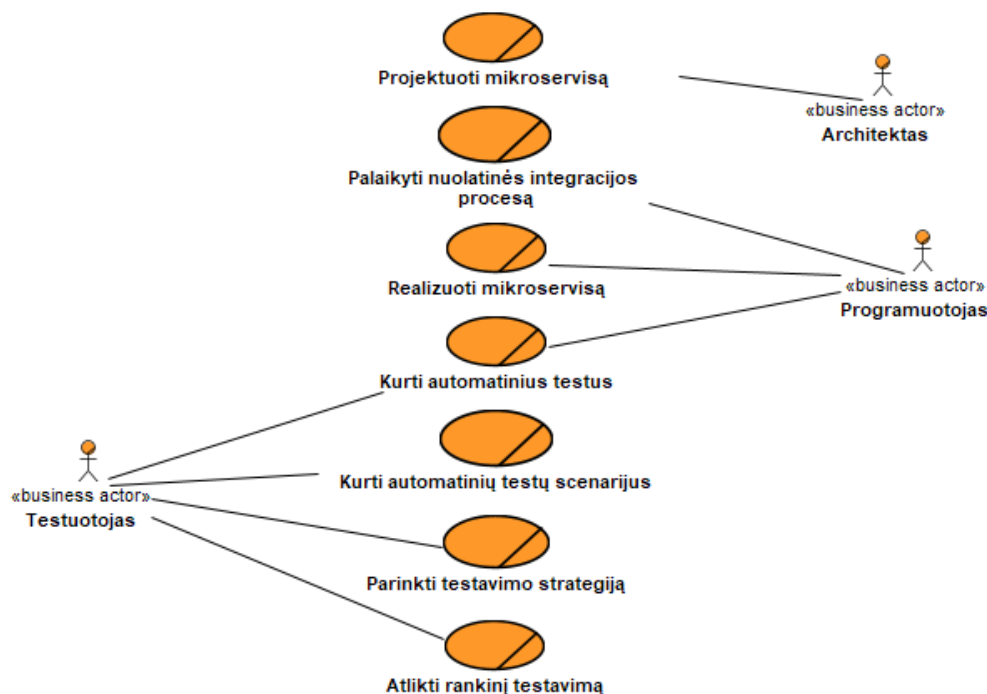
1.12 pav. Komunikacijos tarp mikroservisų vaizdavimas naudojant sekų diagramą [42]

Mikroservisų architektūrai modeliuoti yra naudojamos tik tam tikros *UML* diagramos. Iš struktūrinių diagramų naudojamos klasių ir komponentų diagramos. Klasių diagrama gali būti naudojama vaizduojant ne tik kokie servaisi egzistuoja, bet ir per kokias klases jie komunikuoja [3]. Komponentų diagramos naudojamos tik atvaizduoti, kokie servaisi egzistuoja ir kokias sąsajas jie teikia [41]. Iš elgsenos diagramų, sekų diagramos yra naudojamos atvaizduoti, kaip skirtingi servaisi komunikuoja tarpusavyje [42]. Šios *UML* diagramos gali suteikti papildomą informaciją generuojant testų scenarijus [43].

1.5. Mikroservisų testavimo dalyvių analizė

Mikroservisų architektūra reikalauja bendradarbiavimo tarp programuotojų, testuotojų ir architektų siekiant užtikrinti, kad mikroservaisi būtų patikimi ir veiktų teisingai. Ties mikroservisų testavimu dirbantys aktoriai ir jų atsakomybės pateiktos 1.13 pav.

Programuotojai yra atsakingi už mikroservisų kūrimą, kurie atlieka tam tikrą verslo funkciją [44]. Programuotojai turi turėti ne tik aukštus techninius įgūdžius, bet ir suprasti verslo logiką ir kliento poreikius. Jų užduotis yra rašyti aiškų, struktūrizuotą ir patikimą kodą bei užtikrinti, kad programa veiktų kaip numatyta. Kita vertus, testuotojai yra atsakingi už patikrinimą, ar programa atitinka numatytus reikalavimus ir veikia be klaidų [45]. Jie vykdo įvairius testavimo tipus, įskaitant funkcionalumo, našumo, saugumo ir integracijos testus. Testavimo scenarijus dažniausiai kuria ir vykdo testuotojai [45]. Automatinių testų įgyvendinimui reikalinga programuotojų pagalba tam, kad būtų realizuotas programinis kodas, atliekantis testų scenarijuose aprašomus veiksmus ir patikrinimus.



1.13 pav. Mikroservisų testavimo dalyviai

Bendradarbiavimas tarp programuotojų, testuotojų ir architektų yra svarbi programinės įrangos ir testavimo proceso dalis, ypač taikant mikroservisų architektūrą. Viena iš

priemonių, padedančių užtikrinti efektyvią komunikaciją, yra sekų diagramos, modeliuojančios mikroservisų sąveiką. Sekų diagramos dažniausiai taikomos projektavimo etape, tačiau jų taikymo sritis gali būti išplėsta ir į testavimo procesą. Tokios diagramos suteikia galimybę testuotojams tiksliau identifikuoti sistemos elgsenos scenarijus, nustatyti priklausomybes tarp komponentų. Be to, sekų diagramomis aprašyti scenarijai gali būti naudojami kaip pagrindas automatizuotiems testams kurti. Taip sekų diagramos tampa bendru komunikacijos įrankiu tarp architektų, programuotojų ir testuotojų, prisidedančiu prie testavimo proceso palengvinimo.

1.6. Esamų problemos sprendimo metodų analizė

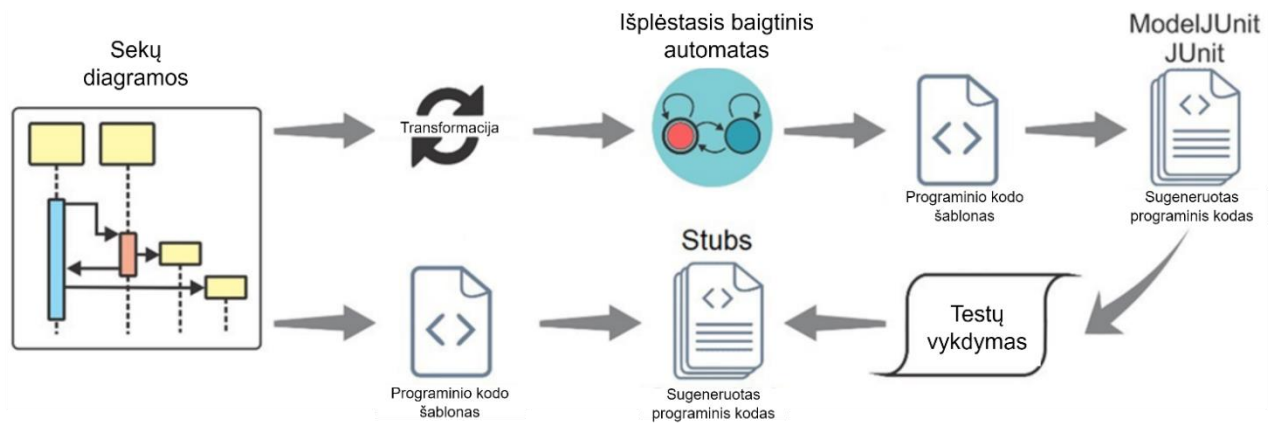
Testavimui palengvinti kuriami įvairūs sprendimai ir metodai. Analizuojant egzistuojančius sprendimus buvo apžvelgiami sprendimai, kurie padeda rašyti testus mikroservisams [46, 47], generuoti testus iš *UML* diagramų [48-51]. Pasirinkta analizuoti sprendimus, kurie nebūtinai pritaikyti mikroservisams, siekiant apžvelgti testų generavimo galimybes, kurias galėtų būti pritaikytos šiai architektūrai.

Testų generavimas iš *UML* sekų diagramų nėra nauja idėja. Jau egzistuoja sprendimų, kurie generuoja testus iš šių diagramų [43]. Sukurti sprendimai apima vienetų testų generavimą tiek individualioms klasėms, tiek atskiriems sistemos komponentams. Taip pat testų generavimui naudojamos *OpenApi* specifikacijos [52], kurios naudojamos galutiniams taškams testuoti. Testavimui palengvinti naudojami sprendimai, kurie negeneruoja testų, bet padeda vizualizuoti scenarijus ar dalintis testais tarp komandų [47].

1.6.1. Testų generavimo iš sekų diagramų metodas [51]

Vienas iš testų generavimo sprendimų naudoja išplėstinį baigtinį automatą kaip tarpininką tarp sekų diagramų ir testų programinio kodo [51]. Išplėstasis baigtinis automatas gali atvaizduoti duomenų srautus ir jų kontrolę. Šis sprendimas nėra specifiškai pritaikytas mikroservisų architektūrai, tačiau taip pat generuoja testus iš sekų diagramų, kaip darbe siūlomoje metodikoje.

1.14 pav. pateikta principinė testų generavimo schema. Sekų diagramos, kuriose vaizduojama detali sąveika tarp sistemos objektų ir aktorių, transformuojamos į baigtinių būsenų mašinas. Kadangi sekų diagramoje aprašomi veiksmai yra sunumeruoti ir turi nustatytą tvarką, sekų diagramą galima transformuoti į išplėstąjį baigtinį automatą [51]. Jeigu sekų diagramoje naudojamas ciklas ar sąlyga, tuomet šioms diagramos dalims reikalinga atitinkama transformacija. Sąlygos sakinyse baigtiniame automate atvaizduojamas kaip išsišakojimas, o ciklo sakinyse kaip būsenų ciklas. Testų scenarijų generavimui naudojama *ModelJUnit* biblioteka. Ši biblioteka, parašyta *Java* kalba, suteikia galimybę generuoti testus iš modelio ir vizualizuoti sukurtą modelį. *ModelJUnit* bibliotekos pagalba baigtinis automatas yra naudojamas testų scenarijų generavimui. Sugeneruoti testai yra vykdomi naudojant *JUnit* biblioteką.



1.14 pav. Testų generavimo procesas naudojant sekų diagramas ir išplėstąjį baigtinį automatą [51]

Taikant šį sprendimą, testuotojui nereikia pačiam rašyti programinio kodo, jis yra atsakingas už sekų diagramų kūrimą ir testų vykdymą, o testų generatorius automatiškai sukuria reikalingus testus iš sukurtų diagramų [51]. Vienas iš šio sprendimo trūkumų yra tai, kad jį galima naudoti tik su Java kalba sukurtomis sistemomis. Jeigu būtų poreikis testuoti sistemą, kuri parašyta kita kalba, reikėtų papildomo komponento, kuris būtų atsakingas už testų transformaciją į reikalingą programinę kalbą ar karkasą.

1.6.2. SCOTEM testų generavimo iš komunikacijos ir būsenų diagramų metodas [50]

Egzistuoja sprendimai, kurie generuoja testus ne tik iš sekų diagramų. Vienas iš jų yra metodas, naudojantis *SCOTEM* (*angl. State Collaboration Test Model*) modelį, kuris generuoja testų scenarijus naudodamas *UML* būsenų ir komunikacijos diagramas [50]. *SCOTEM* bando išspręsti integracijos tarp klasių testavimo problemą. Objektinė programavimo paradigma remiasi tuo, kad objektas turi būseną ir veiksmus, kurie keičia jo būseną. Dažniausiai atskirų klasių testavimas nėra sudėtingas, ypač kai metodai yra maži ir atlieka vieną konkrečią funkciją [50]. Tačiau, kai klasės yra integruojamos viena su kita, gali atsirasti sudėtingų scenarijų, kurie gali likti neištutuotais. Taip pat kai klasės sąveikauja viena su kita, gali atsirasti įvairių rūšių klaidų: sąsajos klaidos, konfliktuojantys ir trūkstami veiksmai. Siekiant užtikrinti, kad šios klaidos būtų atrastos, turi būti ištestuoti visi scenarijai, kam reikalinga gili analizė, apžvelgianti įvairius scenarijus esant skirtingoms objekto būsenoms. Šiai problemai spręsti buvo sukurtas *SCOTEM* metodas [50], naudojant būsenų ir komunikacijos diagramas siekiama sugeneruoti kuo daugiau skirtingų testų scenarijų, kurie ištestuotų sąveiką tarp objektų esant skirtingoms būsenoms.

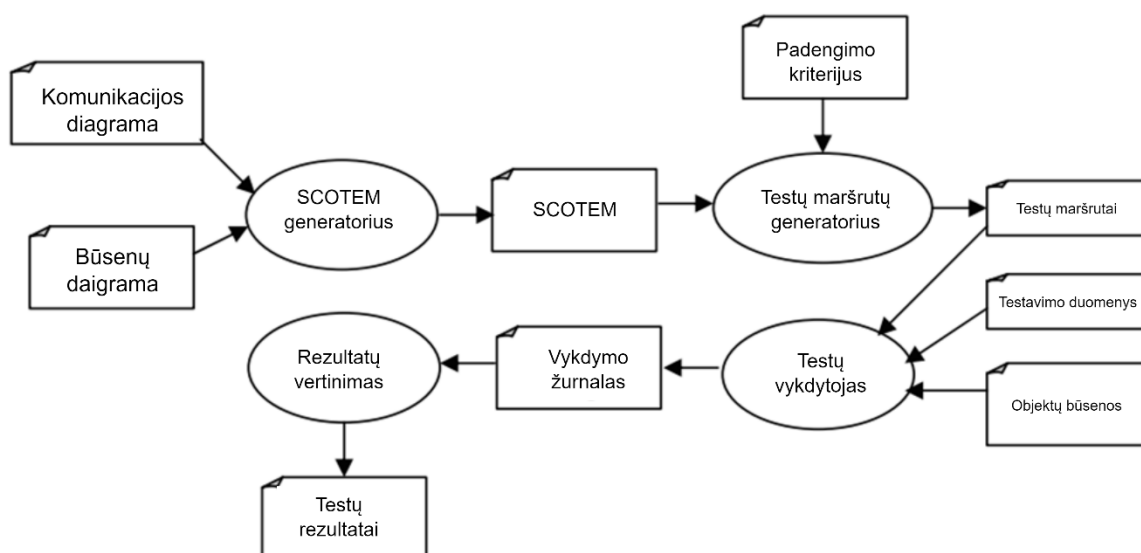
1.15 pav. pateikiami 4 pagrindiniai testų generavimo žingsniai. Pirmiausia vyksta *SCOTEM* modelio generavimas. Tai tarpinis testavimo modelis, kuris sukuriama iš *UML* komunikacijos diagramos ir susijusių būsenų diagramų. Šioje diagramoje, kaip ir komunikacijos diagramoje atvaizduojami perėjimai tarp objektų, tačiau modelis išsiskiria tuo, kad kartu su objektu saugoma objekto būsenos informacija.

Testų scenarijai generuojami iš *SCOTEM* remiasi pasirinktu kodo padengimo kriterijumi [50]. Nuo padengimo kriterijaus priklauso testų kiekis. Taikant vieno maršruto (*angl. single-path*) padengimo kriterijų, užtikrinama, kad visos sąveikos tarp objektų padengiamos bent kartą. Visų perėjimų (*angl. all-transition*) kriterijus užtikrina, kad visos objektų būsenos buvo

aplankytos. N-maršrutų (*angl. n-path*), kur *n* pasirenkamas skaičius, galintis būti nuo tokio dydžio, kuris užtikrina, kad buvo padengti visi būsenų perėjimai, iki tokio, kuris užtikrina, kad visi maršrutai buvo padengti. Taikant visų maršrutų padengimą (*angl. all-path*) užtikrinamas geriausias kodo padengimas, tačiau testų kiekis yra didžiausias, dėl ko gali sulėtėti testavimo procesas [50].

Kai testai yra sugeneruoti, jie yra vykdomi. Sprendimas generavo testus *Java* kalbai. Nors testų scenarijai yra generuojami automatiškai, reikalinga užpildyti testavimo duomenis rankiniu būdu [50]. Testų vykdymo metu sukuriama vykdomo žurnalas, kuriame yra užfiksuotos objekto būsenos prieš ir po kiekvienos žinutės vykdymo.

Surinkus testų vykdymo rezultatus, atliekamas rezultatų vertinimas. Objekto būsenos vykdymo žurnale lyginamos su tikėtomis objekto būsenomis testų maršrutuose, sukurtuose iš *SCOTEM* [50]. Jei po testo maršruto vykdymo objektas nėra reikiamoje būsenoje, tuomet atitinkamas testavimo atvejis laikomas nepavykusiu.



1.15 pav. Testų generavimo procesas naudojant *SCOTEM* modelį [50]

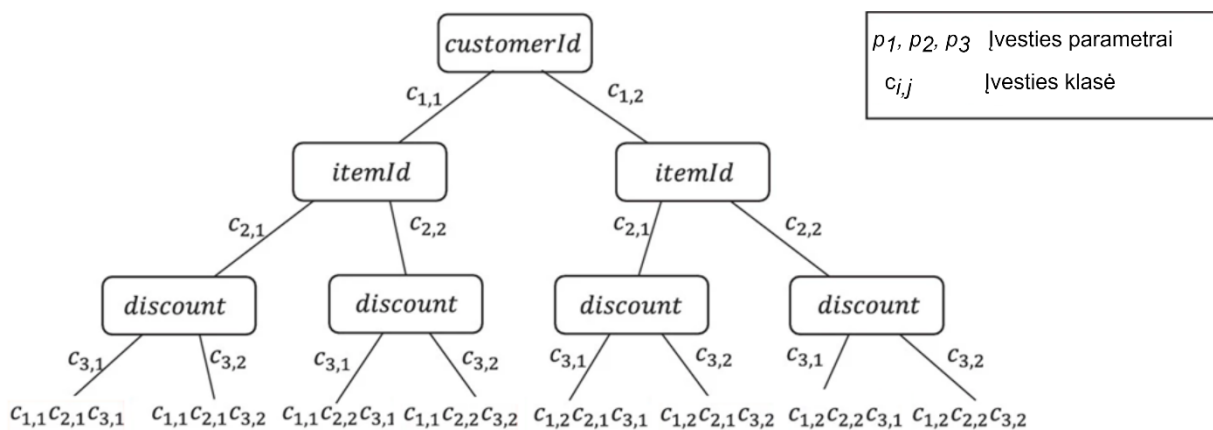
Vienas iš pagrindinių šio sprendimo pranašumų yra tai, kad galima sugeneruoti testų scenarijus ribinėms situacijoms [50]. Kadangi klasės metodai yra išskviečiami skirtingų būsenų atvejais, kyla pasitikėjimas, kad klasės kartu veikia teisingai.

Šio sprendimo trūkumas yra tai, kad reikalinga sukurti itin kruopščias komunikacijos ir būsenos diagramas. Bet kokie pokyčiai reikalautų atnaujinti ne tik programinį kodą, bet ir diagramas, kas sulėtintų sistemos kūrimo spartą. Taip pat sprendimas nėra orientuotas į mikroservisų architektūrą – netestuoja mikroservisų tarpusavio komunikacijos. Šis sprendimas netenka pranašumo, kai nenaudojami modeliai, turintys daug verslo logikos. Jeigu naudojamos būsenos neturinčios servisų klasės, kurios turi grynas funkcijas (*angl. pure functions*), tuomet nėra poreikio testuoti objektų sąveiką esant skirtingoms būsenoms.

1.6.3. Testų generavimas naudojant *OpenApi* specifikaciją [49]

Egzistuoja sprendimai, naudojantys mikroservisų *API* specifikacijas, tam, kad generuotų automatinius testus [52]. *OpenApi* specifikacijos aprašo, kokie atributai reikalingi užklausoms ir kokį formatą turi atitikti nurodyti atributai. Pavyzdžiui, *API* specifikacijoje galima nurodyti, kad atributo reikšmė gali būti tik sveikasis skaičius arba tekstas, kuris turi atitikti numatytą reguliarią išraišką. Naudojant šią informaciją galima sugeneruoti testus, kurie patikrina, ar mikroservisai laikosi nustatytų apribojimų ir ar yra gražinami klaidos pranešimai esant kitokioms situacijoms.

Vienas iš sprendimų, kuris taiko *OpenApi* specifikaciją generuojant testus, yra *UTEST* [49]. Šis metodas pirmiausia sugeneruoja klases kiekvienam užklausos parametru ir atributui. Pavyzdžiui, jeigu parametras yra sveikasis skaičius, gali būti sukurtos 2 klasės, kai skaičius yra teigiamas ir neigiamas. Naudojant sukurtas klases sugeneruojami visi galimi atributų rinkiniai, kaip pateikta 1.16 pav. Naudojant sugeneruotas kombinacijas atliekamas testavimas, siunčiant užklausą su viena iš atributų kombinacijų. Testas laikomas sėkmingu, jei buvo gražintas sėkmės pranešimas, kai buvo naudoti atributai, atitinkantys specifikaciją, ar klaidos pranešimas, kai buvo naudojamas neteisingo formato atributas.



1.16 pav. Parametrų rinkinių generavimo pavyzdys [49]

Taikant *UTEST* metodą galima ištestuoti, ar mikroservas atitinka specifikaciją. Taip pat galima atrasti, ar atliekamas teisingas duomenų patikrinimas [49]. Tačiau *OpenAPI* specifikacijos neturi verslo logikos aprašymo, todėl testai negali aptikti gedimų, susijusių su nuokrypiais nuo numatytosios *API* elgsenos. Be to, sugeneruoti testai negali aptikti gedimų, kai atsakymo būsenos kodas nesutampa su vidaus vykdymu, pavyzdžiui, kai gražinamas sėkmės pranešimas, nuslepiančias įvykusią vidinę klaidą [49].

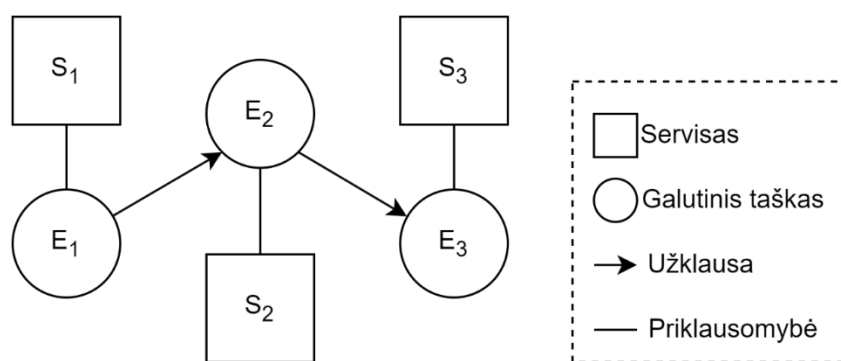
1.6.4. Grafu paremtas mikroservisų analizės įrankis [46]

Testų kūrimo problema sprendžiama kuriant ne tik įrankius, generuojančius testus, bet ir tokius, kurie padeda analizuoti sistemą testų kūrimo metu. Vienas iš tokių sprendimų yra grafais ir scenarijais paremtas mikroservisų analizės įrankis *GSMART* (angl. *Graph-based and Scenario-driven Microservice Analysis, Retrieval and Testing*) [46].

Šio įrankio kūrėjai apibrėžė 4 pagrindines šio įrankio funkcijas kartu su sprendžiamomis problemomis [46]:

- Priklausomybių tarp mikroservisų, taip pat tarp scenarijų ir paslaugų, valdymas ir vizualizavimas. Priklausomybių valdymas išskiriamas kaip esminis sistemos ir programinės įrangos projektavimo dalis [46]. Pagrindinis *GSMART* uždavinys yra surinkti užklausas tarp mikroservisų, siekiant sukurti ir vizualizuoti paslaugų priklausomybių grafus. Taip pat siekiama suteikti galimybę susieti sukurtus *BDD* scenarijus su atitinkamais mikroservisais.
- Ciklinių mikroservisų priklausomybių aptikimas. Ciklinės priklausomybės tarp mikroservisų kelia riziką sistemos vystymui ir priežiūrai, jos sukelia vykdymo klaidas ir padidina naujų mikroservisų versijų sudėtingumą [46]. Automatinis gebėjimas aptikti ciklines priklausomybes leistų sumažinti šias rizikas.
- Regresinių testų scenarijų parinkimas. Atliekant mikroservisų pakeitimus reikalinga atlikti regresinį testavimą [46]. Dažniausiai vykdomi visi testai siekiant įsitikinti, kad nebuvo sugadintas sistemos funkcionalumas. Tačiau dauguma mikroservisų nėra paveikti šiais pokyčiais, jeigu atliekama modifikacija viename servise [46]. Todėl su šiuo įrankiu siekiama parinkti pagal svarbą testavimo atvejus, siekiant sumažinti laiką ir kaštus, susijusius su regresiniu testavimu.
- Egzistuojančių mikroservisų automatinis atradimas. Siekiama, kad kuriant naujus mikroservisus ar pridėdant naujus galutinius taškus, nereikėtų rankiniu būdu įvesti šios informacijos [46]. Bandoma turėti veiksmingą mikroservisų gavimo metodą, kuris leistų visuomet turėti sistemos vaizdą, atspindintį dabartinę struktūrą.

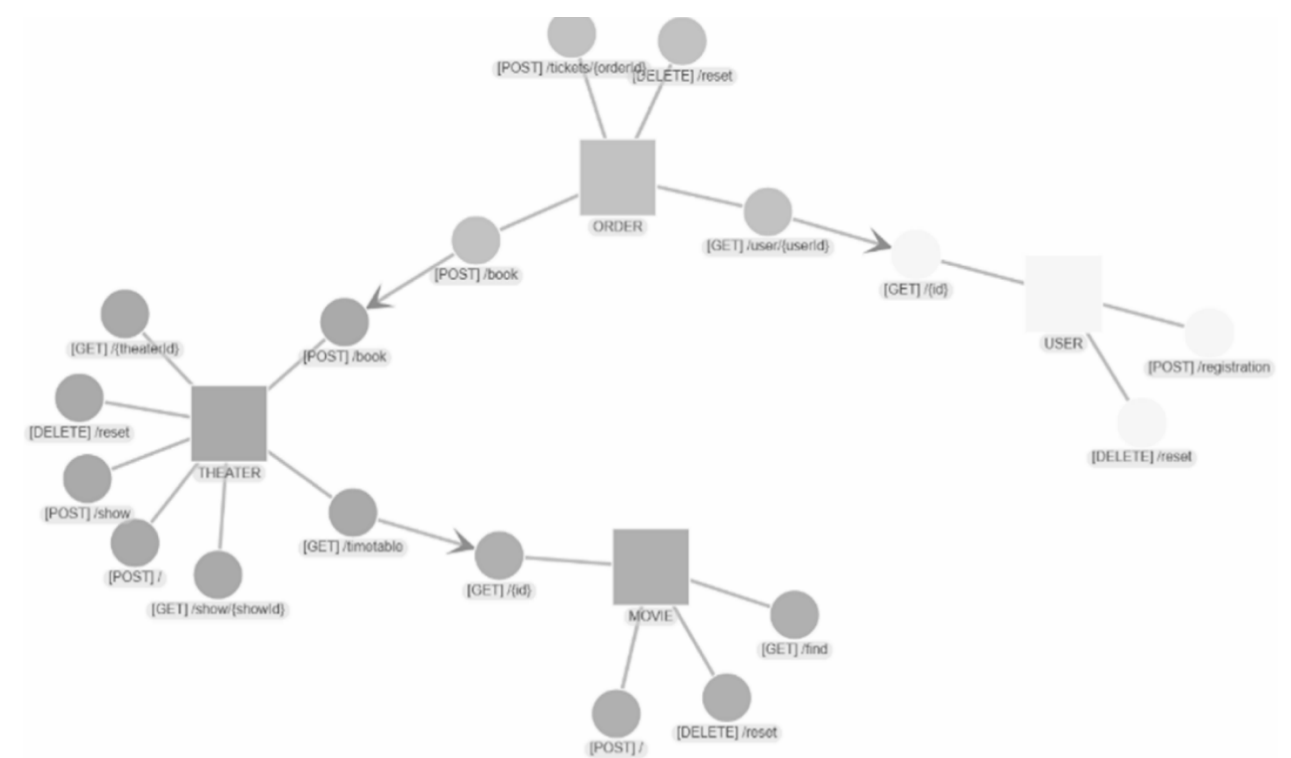
Šio sprendimo išeiga yra grafas, vaizduojantis mikroservisus ir jų ryšius. 1.17 pav. pateiktas pavyzdinis grafo fragmentas. Grafe galima matyti, kokius galutinius taškus teikia mikroservisai ir kaip mikroservisai yra susiję tarpusavyje. Grafe galima sekti individualias užklausas ir matyti, kur užklausa sustojo ir įvyko klaida [46]. Toks funkcionalumas padeda testuotojams derinti testus ir atrasti sistemos klaidas.



1.17 pav. *GSMART* grafo pavyzdys [46]

Sistemos grafo sudarymui gali būti naudojama *OpenApi* specifikacija ir pačių mikroservisų programinis kodas naudojant refleksiją [46]. Refleksija yra programavimo kalbos teikiamas funkcionalumas, kuris leidžia programos vykdymo metu gauti duomenis apie pačios programos struktūrą. Šį funkcionalumą palaiko dauguma programavimo kalbų kaip: *C#*, *Java*, *Python*. *GSMART* buvo pritaikytas *Spring Boot* karkasu parašytiems mikroservisams.

1.18 pav. pateiktas sudarytas grafas. Jame matoma, kad yra 4 servisi: užsakymų, naudotojų, teatrų ir filmų. Matoma, kokius galutinius taškus turi kiekvienas mikroservisas ir kuris galutinis taškas kviečia kitą mikroservisą.

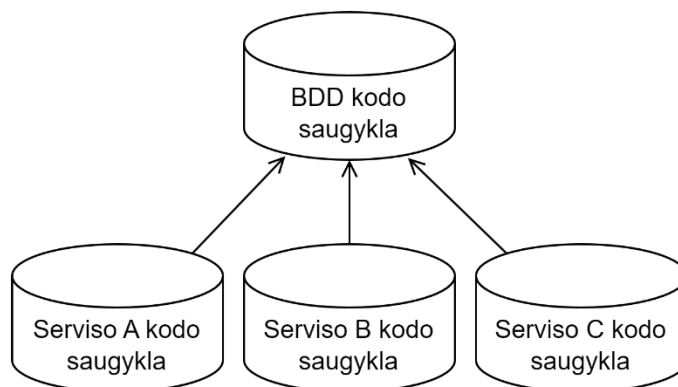


1.18 pav. GSMART sukurto mikroservisų grafo pavyzdys [46]

GSMART nėra įrankis skirtas generuoti testus, bet padeda analizuoti nesėkmingus testus, jų klaidų priežastis ir atrasti testų scenarijus, kurie galimai buvo paveikti kodo modifikacijų. Kaip ir kiti apžvelgti įrankiai, šis taip pat yra pritaikytas *Java* kalbai, todėl šio įrankio nebūtų galima naudoti su sistemomis, kurios naudoja kitas technologijas, be papildomų modifikacijų [46].

1.6.5. BDD scenarijais grįstas sprendimas [47]

Tame pačiame projekte esantys mikroservisi dažniausiai būna labai panašūs: turi panašų autentifikacijos ir autorizacijos mechanizmą, naudoja vienodą komunikacijos stilių, pavyzdžiui, *REST*. Šie panašumai gali leisti dar kartą panaudoti apibrėžtus žingsnius. Pavyzdžiui, žingsniai, skirti autentifikacijai, užklausos siuntimui ir atsakymo lyginimui, sutampa tarp mikroservisų, todėl juos galima panaudoti dar kartą keliuose servisuose. Siūlomas sprendimas, kuris pavaizduotas 1.19 pav., yra saugoti žingsnių šablonus vienoje kodo saugykloje [47]. Toks sprendimas sumažina kodo pasikartojimą, nes žingsnių aprašymai nesikartoja tarp servisų. Programuotojai gali fokusuotis ties žingsnių įgyvendinimu, o testuotojai – ties pačių testų scenarijų kūrimu. Taip pat produkto savininkas žino, kur gali atrasti testus, peržiūrėti funkcijų aprašymus, patikrinti, ar servisi atlieka numatytas funkcijas [47].

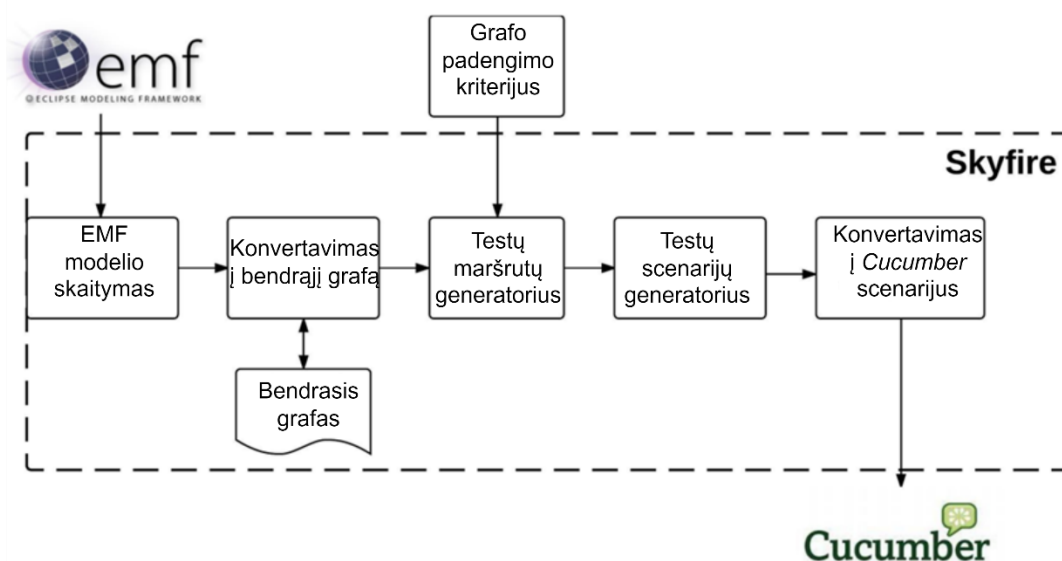


1.19 pav. Bendrų automatinių testų architektūra [47]

Šio sprendimo trūkumas yra tai, kad testų pakeitimai reikalauja atnaujinimų dviejose saugyklose: pirmiausia testų aprašymuose, vėliau kode. Siekiant to išvengti, galima laikytis hibridinio principo: bendrus žingsnius saugoti atskiroje saugykloje, o specifinius – kartu su testuojamo serviso kodu.

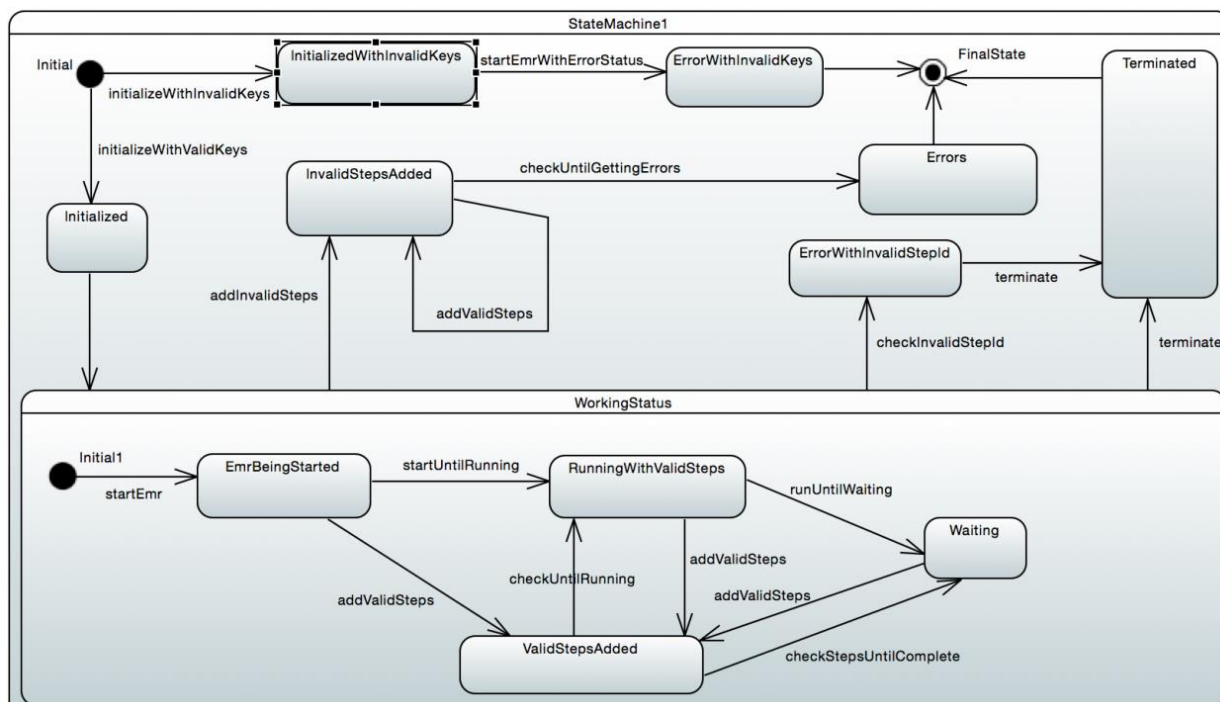
1.6.6. Testų generavimas naudojant *UML* būsenų diagramas ir *Cucumber* karkasą [48]

Skyfire yra įrankis, kuris skirtas generuoti *Cucumber* testų scenarijus iš *UML* būsenų diagramų [48]. Šio sprendimo principinė testų generavimo schema pateikta 1.20 pav. [48]. Šis sprendimas, panašiai kaip ir ankstesniuose skyriuose apžvelgti sprendimai, pirmiausia konvertuoja *UML* diagramą į pasirinktos struktūros grafą. *Skyfire* sprendimas konvertuoja būsenų diagramą į bendrąjį grafą. Iš bendrojo grafo sugeneruojami testų scenarijai, kurie konvertuojami į *Cucumber* karkaso testus.



1.20 pav. Testų generavimo naudojant *Skyfire* procesas [48]

Generuojant *Cucumber* testų scenarijus naudojami būsenų ir perėjimų pavadinimai. Perėjimai yra transformuojami į *Given* arba *When* žingsnius. *Then* žingsniai atspindi, kurioje būsenoje sistema turi būti. 1.21 pav. pateikta būsenų diagrama testuojamai sistemai, o 1.22 pav. pateiktas sugeneruotas testo scenarijus. Sugeneruotame scenarijuje nuosekliai aprašomi, kokie būsenų perėjimai įvyko, naudojant tokius pačius pavadinimus, kokie yra naudojami ir būsenų diagramoje. Būsenos tikrinimui naudojamas pakeistas būsenos pavadinimas. Pavyzdžiui, vietoj *RunningWithValidSteps* naudojamas *emrIsRunning* pavadinimas.



1.21 pav. Pavyzdinė būsenų diagrama, kuriai generuojami testai [48]

1.22 pav. scenarijaus aprašas yra sugeneruotas *Skyfire* įrankiu, tačiau žingsniai neturi realizacijos ir negali iš karto būti vykdomi. Reikalingas papildomas darbas tam, kad sukurti programinį kodą, kuris patikrintų būsenas ir įvykdytų perėjimus.

Given initializeWithValidKeys
When startEmr
Then emrCreationIsSuccess
When startUntilRunning
Then emrIsRunning
When addValidSteps
And checkStepsUntilComplete
Then stepsAreCompleteWaiting

1.22 pav. Sugeneruoto scenarijaus pavyzdys [48]

Skyfire siūlo metodą, kaip panaudoti *UML* būsenų diagramas testų scenarijų generavimui tam, kad geriau padengtų funkcionalumą automatiniais testais. Nors testai nėra generuojami konkrečiai kalbai, siūlomas sprendimas gali būti taikomas sistemoms, parašytoms kita programavimo kalba negu *Java*. Tokiu atveju reikalingas papildomas darbas žingsnių realizacijai.

1.6.7. Testų generavimo ir vizualizavimo įrankių palyginimas

Egzistuojančių sprendimų, kurie skirti sistemų testavimui, palyginimas pateiktas 1.1 lentelėje. Sprendimai buvo lyginami naudojant 10 kriterijų. Vienu iš kriterijų pasirinktas testo scenarijaus vizualizavimas. Testo scenarijus gali būti atvaizduotas *UML* diagramoje, baigtiniame automate ar grafe su testuojamais elementais, tačiau kai kurie sprendimai nepateikia jokie būdo scenarijui vizualizuoti. Testavimo scenarijaus vizualizavimas gali palengvinti scenarijaus analizę ir padėti suprasti, kokios sistemos dalys yra testuojamos [17]. Taip pat apžvelgiama, kokio abstrakcijos lygio elementams skirtas sprendimas. Mikroservisų architektūra išsiskiria tuo, kad sistemos dalys komunikuoja tinklo lygmenyje [4], todėl siekiama apžvelgti, kurie sprendimai orientuoti šiam sistemos lygiui. Sistema gali būti testuojama nuo žemo lygio kaip atskiri metodai ir klasės, aukštesnis lygis yra komunikacija tarp klasių ar to paties serviso modulių, aukščiausias nagrinėjamas lygis – komunikacija tarp atskirų servisų. Kad būtų palyginti ne tik, kuris lygmuo testuojamas, apžvelgiama ir kokio tipo testams skirti sprendimai. Apžvelgti įrankiai naudojami vienetų, kontraktų, komponentų ir galutiniams testams. Taip pat lyginama, kokia programavimo kalba įgyvendinti sprendimai. Sprendimų įgyvendinimui gali būti naudojama bet kuri programavimo kalba. Be to, lyginama, kokia kalba turi būti įgyvendinta testuojama sistema. Kadangi mikroservisai gali būti įgyvendinti naudojant skirtingas technologijas [4], ne visi sprendimai gali būti suderinami su mikroservisais, įgyvendintais skirtingomis technologijomis. Taip pat apžvelgiama, ar sprendimas naudojamas regresiniame testavime. Sukurtų sprendimų testai gali būti naudojami atliekant regresinį testavimą arba sprendimas gali būti skirtas padėti atrinkti testų scenarijus, kurie susiję su atliktu pokyčiu. Galiausiai lyginama, kokie duomenys yra reikalingi sprendimui. Įvedami duomenys gali būti tiek *UML* diagramos, *OpenApi* specifikacijos ar programinis kodas. Išvedami duomenys gali būti konkretaus karkaso testai, testuojamų elementų grafas arba atlikto testų vykdymo žurnalas.

1.1 lentelė. Egzistuojančių testų generavimo ir vizualizavimo įrankių sprendimų palyginimas

Kriterijus	Testų generavimo iš sekų diagramų metodas [51]	„SCOTEM“ [50]	„UTEST“ [49]	„GSMART“ [46]	Bendra BDD kodo saugykla [47]	„Skyfire“ [48]
Testo scenarijaus vizualizavimas	Sekų diagramose ir baigtiniame automate	„SCOTEM“ modelyje	-	Atvaizduojamas užklauskos kelias tarp mikroservisų	-	Būsenų diagramose
Nagrinėjamas abstrakcijos lygmuo	Komunikacija tarp klasių	Komunikacija tarp klasių ir jų būseną	Galutiniai taškai	Komunikacija tarp mikroservisų	Galutiniai taškai	Komunikacija tarp klasių ir jų būseną
Įgyvendinta su programavimo kalba	Java	Java	Java	Java	Nereikalauja konkrečios kalbos	Java
Programavimo kalba, kuria įgyvendinta testuojama sistema	Java	Java	Bet kokia kalba su įgyvendintu REST API	Java	Bet kokia kalba su įgyvendintu REST API	Sugeneruoti scenarijai įgyvendinami pasirinkta kalba
Naudojamas testų tipams	Vienetų	Vienetų	Kontraktų	Kontraktų ir komponentų	Galutiniams	Galutiniams
Siūlomas funkcionalumas	Testų scenarijų generavimas	Testų scenarijų generavimas	Testų scenarijų generavimas	Testų scenarijų atrinkimas ir vizualizavimas	Testų scenarijų perpanaudojimas tarp mikroservisų	Testų scenarijų generavimas
Testavimo technika	Baltosios dėžės	Baltosios dėžės	Juodosios dėžės	Baltosios dėžės	-	Baltosios dėžės
Naudojamas regresiniame testavime	-	-	-	Testų scenarijų atrinkimas	-	-
Naudojami duomenys	Sekų diagramos	Būsenų ir komunikacijos diagramos	OpenApi specifikacija	OpenApi specifikacija, programinis kodas	Rankiniu būdu sukurti testų scenarijai	Būsenų diagramos
Išvedami duomenys	JUnit karkaso vienetų testai	JUnit karkaso vienetų testai	Kontraktų testai skirti galutiniams taškams	Mikroservisų ir galutinių taškų grafai	-	Testų scenarijai parašyti Gherkin formatu

Atlikus sprendimų palyginimą pastebėta, kad daugelis apžvelgtų sprendimų yra parašyti *Java* kalba, tačiau ne visi reikalauja, kad ir testuojama sistema būtų realizuota šia kalba. *UTEST* sprendimas, nors ir yra parašytas *Java* kalba, tačiau gali būti naudojamas testuoti sistemas, parašytas kitomis programavimo kalbomis [49]. Kadangi *UTEST* įrankis testuoja galutinius taškus, todėl įrankis gali būti naudojamas tol, kol yra įgyvendintas *REST API*. Be to, *Skyfire* sprendimas irgi gali būti pritaikytas įvairioms programavimo kalboms, nes generuojami *Gherkin* formato testų scenarijai, kurie realizuojami pasirinkta kalba [48]. Siekiant išvengti pakartotinio darbo realizuojant testus, kartu galima naudoti bendro *BDD* kodo saugyklos metodą, kas leistų sumažinti žingsnių įgyvendinimui reikalingą laiką.

Iš apžvelgtų sprendimų, skirtų testų generavimui iš *UML* diagramų, tiek sprendimas, generuojantis testus iš sekų diagramų [51], tiek sprendimas, generuojantis testus iš komunikacijos ir būsenų diagramų [50] naudojamas generuoti vienetų testus, kuriais testuojamos klasės. *Skyfire* sprendimas yra skirtas generuoti galutinius testus, bet realizuojant žingsnius reikalinga tikrinti vykdymo metu sukurtų objektų būsenas. Todėl šio įrankio taip pat nebūtų galima naudoti tikrinti, ar mikroservisas atliko teisingas užklausas į kitus mikroservisus. Kiti sprendimai, kaip *UTEST*, skirti tik generuoti testus, bet nesuteikia vizualinės pagalbos juos analizuojant. Priešingai negu *GSMART*, kuris tik suteikia vizualinę pagalbą, pateikiant servisų ir prieigos taškų grafą, tačiau testų scenarijus turi kurti pats testuotojas. Todėl nė vienas iš lyginamų sprendimų nėra pritaikytas generuoti komponentų ir kontraktų testus, kuris taip pat galėtų vizualizuoti testuojamus scenarijus.

1.7. Siekiamo sprendimo apibrėžimas

Siekama sukurti metodą, kuris naudotų *UML* sekų diagramas automatinių testų generavimui. Sekų diagramoje būtų vaizduojamos tarp servisų atliekamos *HTTP* užklausos. Iš sekų diagramų būtų sugeneruojami testų scenarijai *Gherkin* kalba, kuriuose būtų aprašoma, ar mikroservisas atliko užklausas, nurodytas sekų diagramoje, ir ar užklausos naudoja teisingus duomenis. Tokiu būdu būtų patikrinamas mikroserviso funkcionalumas izoliuotame kontekste. Siūloma metodika apsiriboja tik *REST* stiliaus komunikacija ir nepalaiko įvykiais paremtos (*angl. event-driven*) komunikacijos. Planuojama testų scenarijų vykdymui naudoti *.NET* karkasą ir *Reqnroll* biblioteką.

1.8. Analizės išvados

1. Mikroservisų architektūros ir testavimo analizė parodė, kad taikant mikroservisų architektūrą reikalinga atlikti kontraktų ir komponentų testavimą, kad būtų ištestuota mikroservisų tarpusavio elgsena su kitais mikroservisais *HTTP* užklausų lygmenyje.
2. Išanalizavus *UML* taikymą mikroservisų architektūroje pastebėta, kad iš struktūrinių diagramų naudojamos klasių ir komponentų diagramos, aprašyti mikroserviso specifikacijai, o sekų diagramos naudojamos atvaizduoti, kaip mikroservisas integruojasi į kitas sistemas ir kaip jo teikiamas *API* gali būti naudojamas kliento, todėl sekų diagramos gali būti naudojamos testų generavimui.
3. Apžvelgus sprendimus, padedančius testuoti sistemas, pastebėta, kad yra sprendimų, kurie generuoja testus, vizualizuoja scenarijus ar testuoja komunikaciją tarp servisų.

Tačiau nė vienas iš apžvelgtų sprendimų neatlieka visų 3 funkcijų kartu, todėl yra poreikis kurti įrankį, kuris padėtų generuoti testus mikroservisams iš *UML* sekų diagramų.

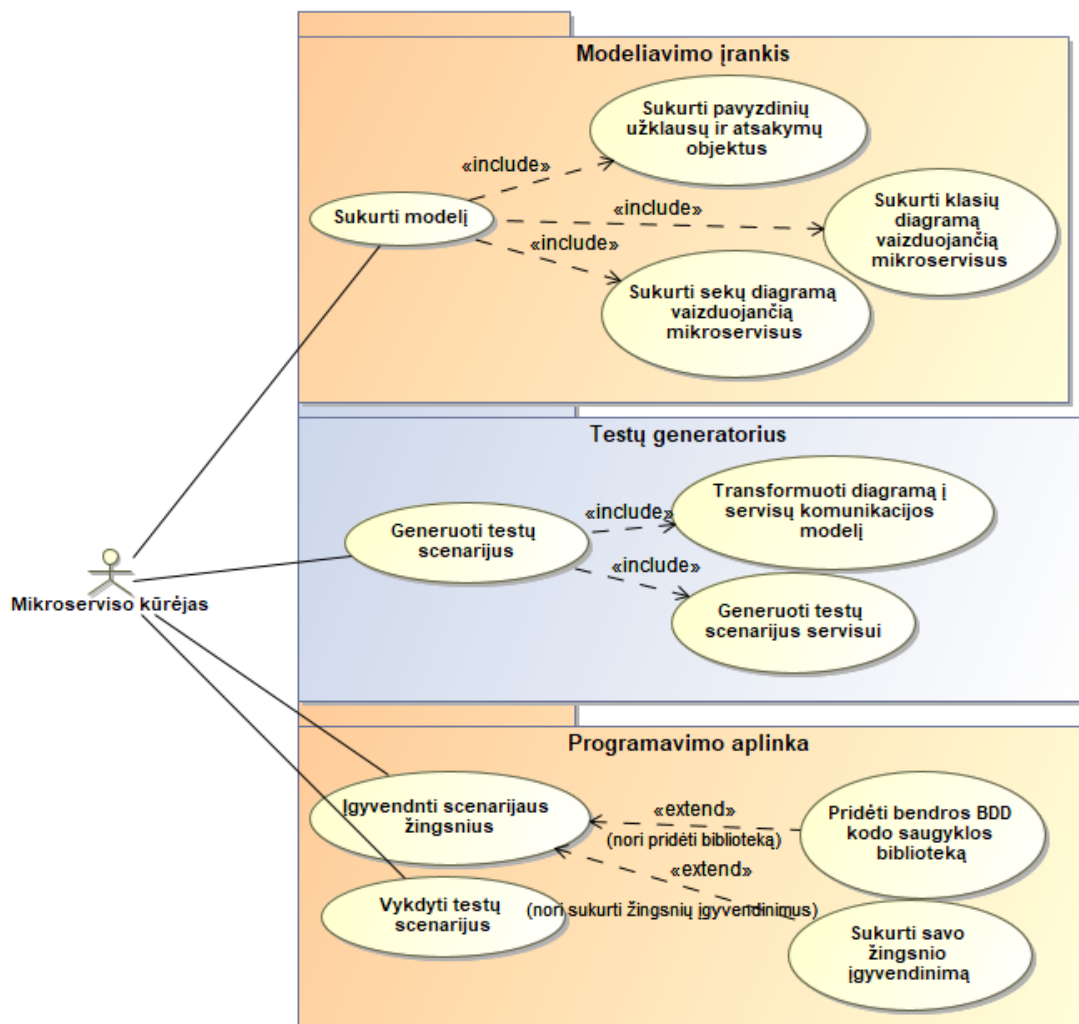
4. Apžvelgus didžiųjų kalbos modelių panaudojimą automatiniam testavimui buvo pastebėta, kad *MCP* serveriai leidžia naudoti pagalbinius įrankius integruotoje programavimo platformoje ir nereikalauja įsigilinti į technines naudojimo detales, taip palengvinant įrankių naudojimą.

2. Mikroservisų testų generavimo iš sekų diagramų reikalavimų specifikacija, formalizuotas aprašas

Šiame skyriuje aprašoma testų generavimo iš sekų diagramų reikalavimų modelis ir formalizuotas aprašas, nusakantis sekų diagramų kūrimo ir testų generavimo metodiką.

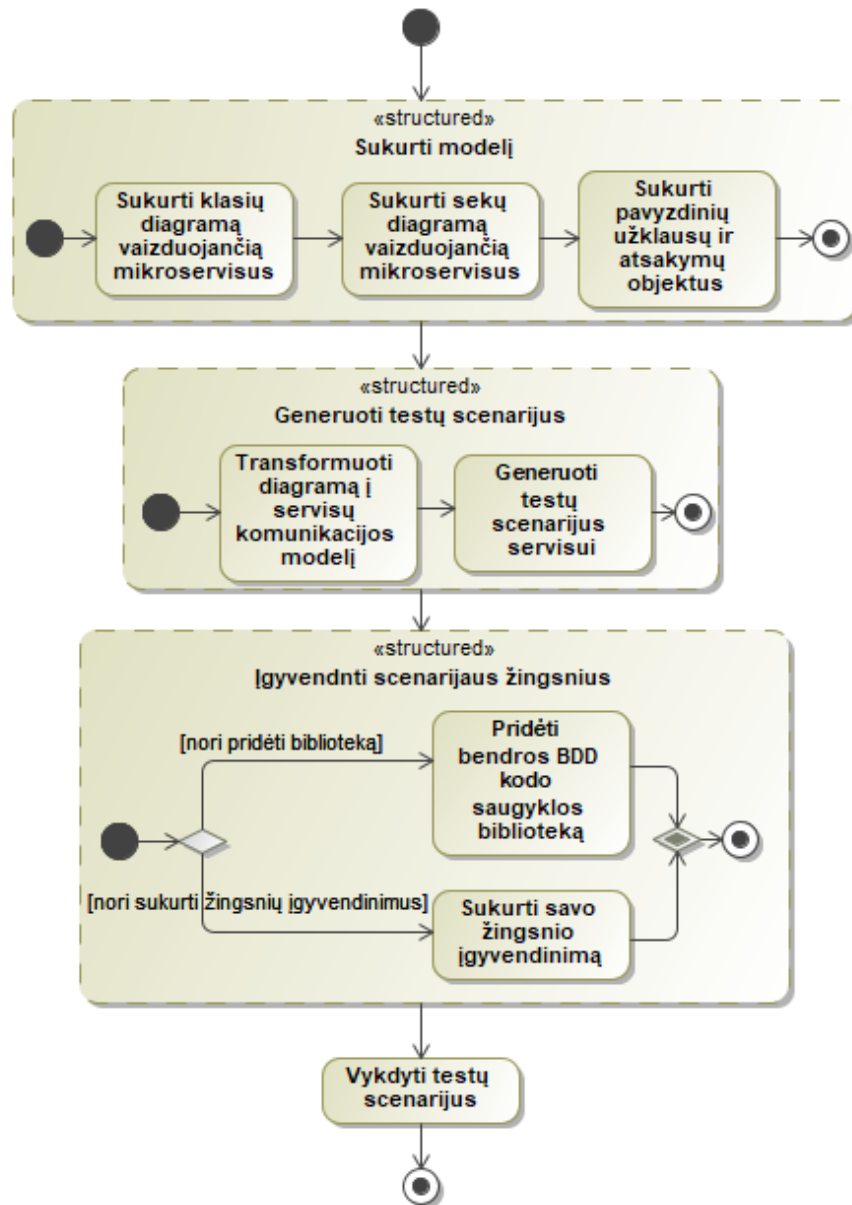
2.1. Reikalavimų specifikacija

Testų generavimo iš sekų diagramų metodikos panaudojimo atvejų modelis vaizduojamas 2.1 pav. Pagrindinis aktorius yra mikroserviso kūrėjas, kuris naudodamas modeliavimo įrankį sukuria *UML* modelį su mikroservisų klasių diagramomis ir sekų diagramomis. Gali būti naudojamas bet koks modeliavimo įrankis, kuris leidžia eksportuoti modelį *XMI* formatu. Generuojant testų scenarijus, sukurtas modelis yra naudojamas kaip įvesties duomenys. Sugeneruoti scenarijai yra įkeliami į mikroserviso projektą, o žingsniai yra realizuojami pačio naudotojo arba naudojama bendra *BDD* kodo saugykla.



2.1 pav. Testų generatoriaus panaudojimo atvejų modelis

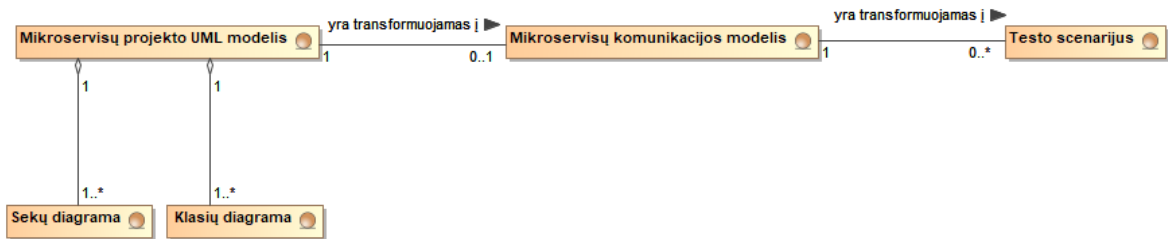
Mikroservisų testų generavimo iš *UML* sekų diagramų metodikos procesas pateikiamas 2.2 pav. veiklos diagramoje.



2.2 pav. Mikroservisų testų generavimo iš UML sekų diagramų proceso veiklos diagrama

2.2. Dalykinės srities modelis

Mikroservisų testų generavimo iš sekų diagramų metodikoje naudojamų dalykinės srities esybių modelis pateiktas 2.3 pav. Mikroservisų kūrėjas pirmiausia turi sukurti mikroservisų projekto UML modelį, kuriame turi būti bent viena klasių diagrama, modeliuojanti mikroservisų turimus galutinius taškus ir atsakymų modelius. Taip pat modelyje turi būti bent viena sekų diagrama, vaizduojanti, kaip mikroservisai komunikuoja tarpusavyje. Mikroservisų projekto UML modelis gali būti transformuojamas į mikroservisų komunikacijos modelį. Šis modelis skirtas atvaizduoti tik kaip mikroservisai komunikuoja tarpusavyje, išvengiant sekų diagramoje vaizduojamos perteklinės informacijos. Mikroservisų komunikacijos modelis gali būti transformuojamas į daug testų scenarijų, kuriais tikrinama sekų diagramoje vaizduojama logika. Dalykinės srities esybės yra detalizuojamos 2.4-2.6 pav.

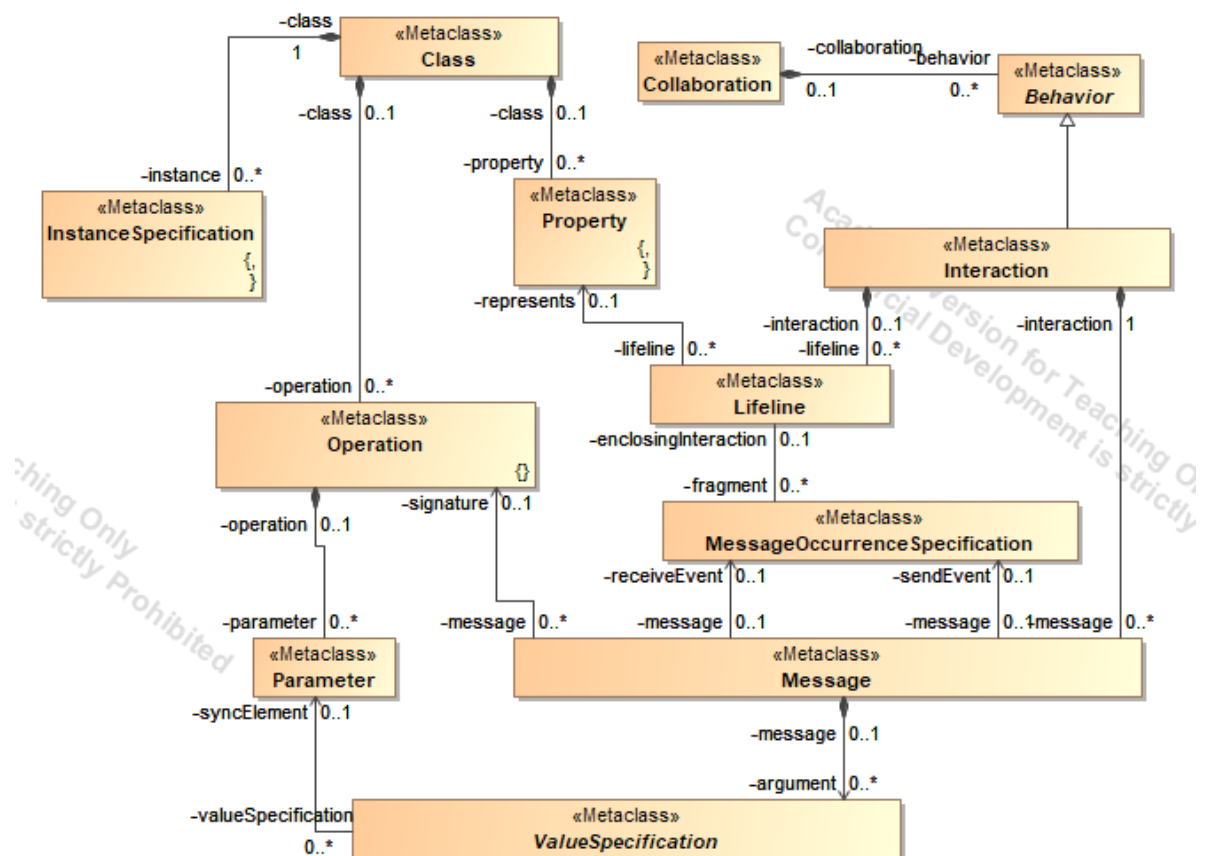


2.3 pav. Testų generavimo iš sekų diagramų dalykinės srities modelis

2.3. Formalizuotas sprendimo aprašas

2.3.1. Testų generavimo logika

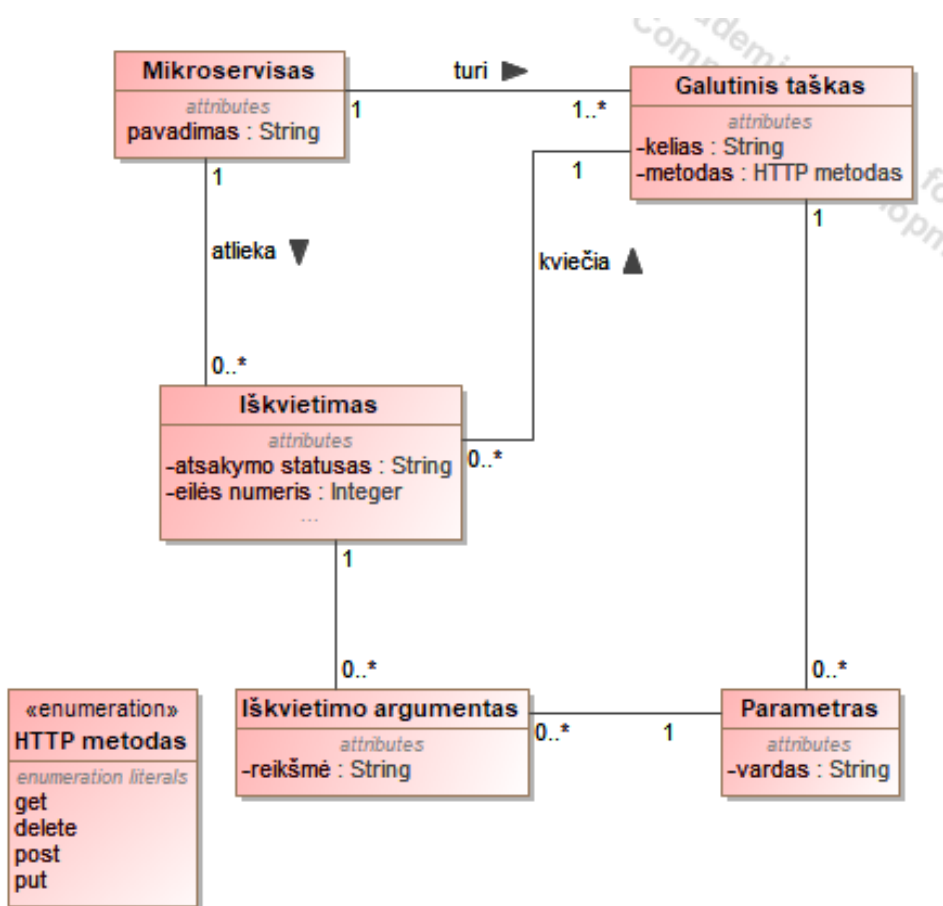
2.4 pav. pateiktos *UML* metamodelio klasės, kurios naudojamos testų generavime [39]. Į diagramą buvo įtrauktos klasės, kurios naudojamos sekų diagramoms vaizduoti. Taip pat įtrauktos operacijų ir parametrų klasės, nes jos sekų diagramoje naudojamos atvaizduoti su kokiais argumentais buvo atliekama *HTTP* užklausa.



2.4 pav. *UML* metaklasės naudojamos testų generavime [39]

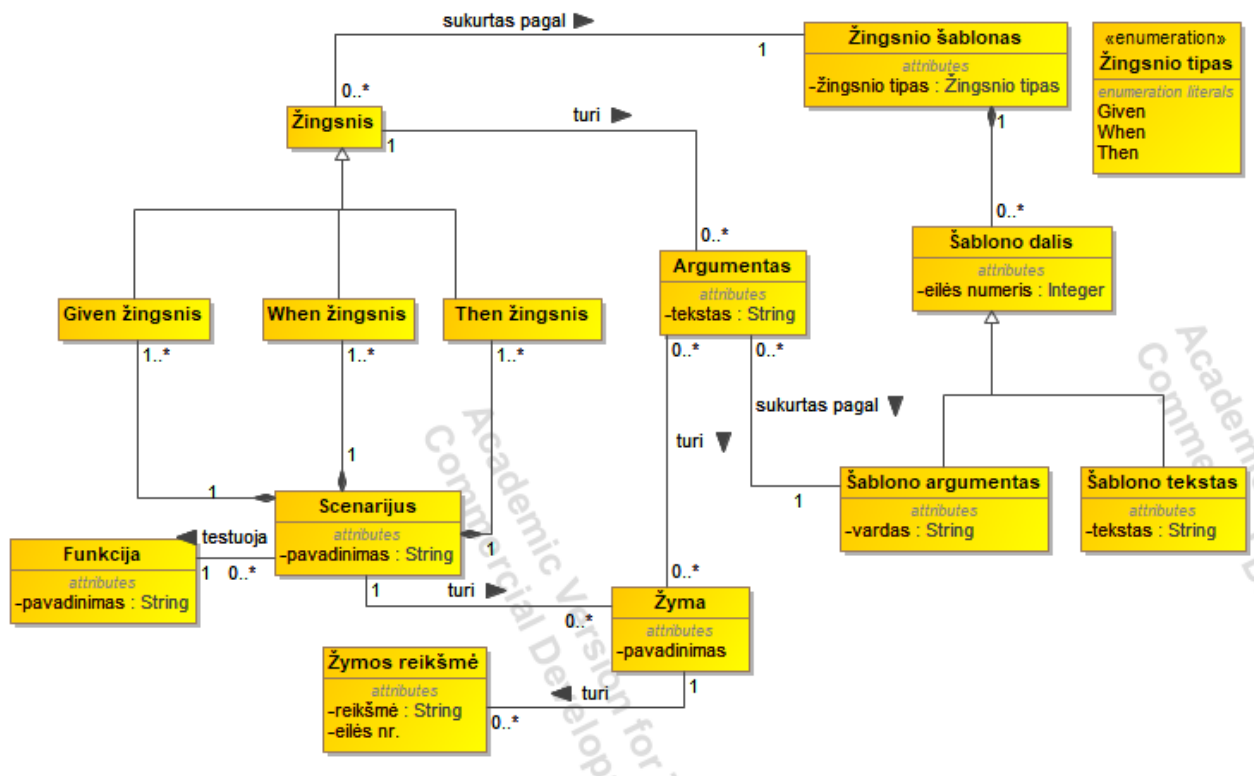
2.5 pav. pateiktos mikroservisų komunikacijos modelio klasės. Šis modelis vaizduoja, kokios klasės reikalingos vaizduoti komunikaciją tarp servisų. Naudojant testų generavimo įrankį *UML* metaklasės yra transformuojamos į supaprastintą modelį, kuris turi pagrindines klases, reikalingas komunikacijai vaizduoti grafo pavidalu: servisas, galutinis taškas, iškvietimas,

parametras ir iškvietimo argumentas. Kiekvienas servisas turi bent vieną galutinį tašką ir gali atlikti daug iškvietimų į kitų servisų galutinius taškus.



2.5 pav. Mikroservisų komunikacijos modelis

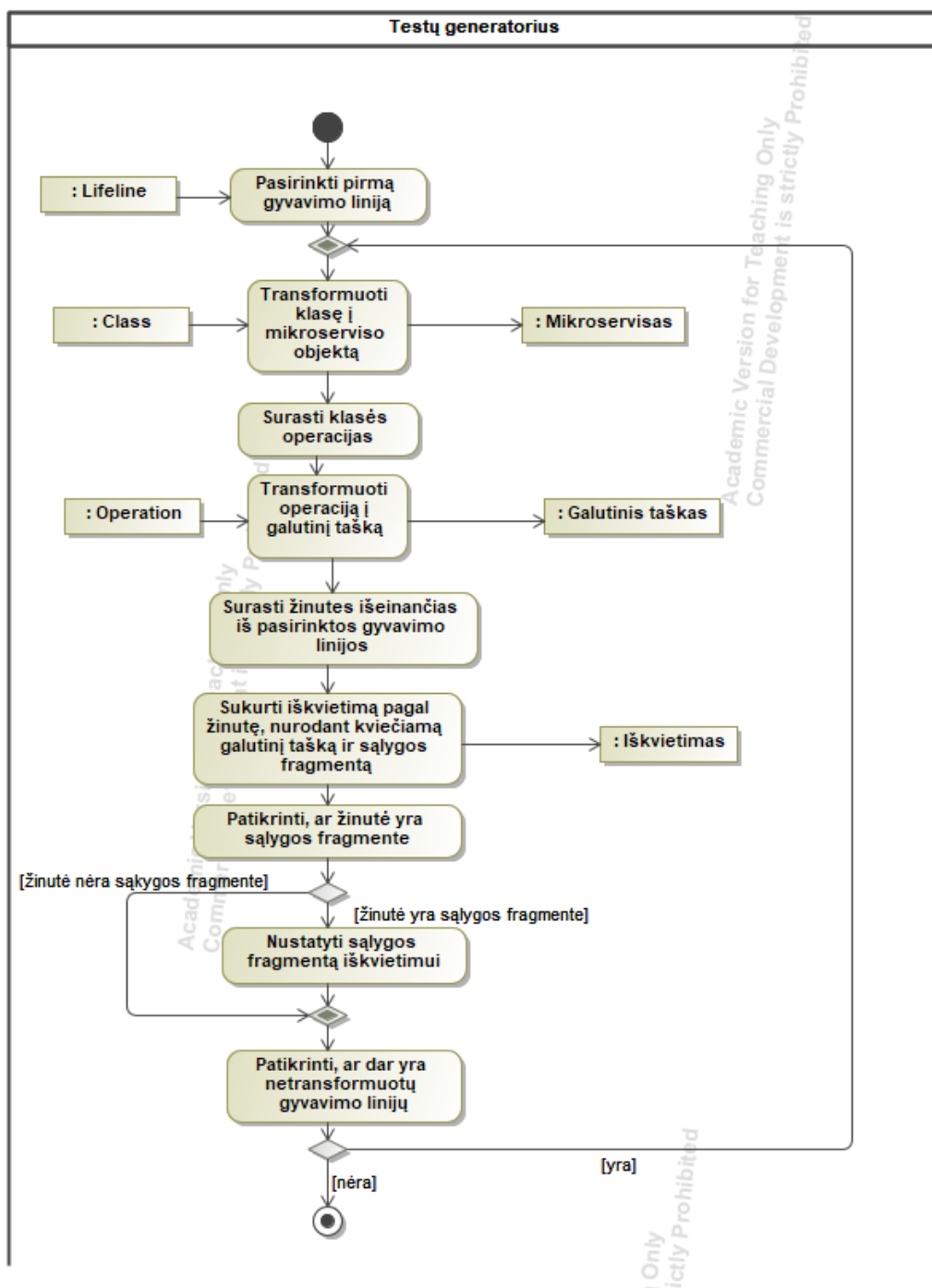
Naudojant servisų komunikacijos esybes generuojami *Gherkin* formato scenarijai, kurių elementai pateikti 2.6 pav. diagramoje. Scenarijai yra grupuojami į funkcijas ir kiekviena funkcija gali turėti daug scenarijų. Scenarijus turi turėti bent po vieną *Given*, *When*, *Then* žingsnį, o kiekvienas žingsnis yra sukurtas pagal žingsnio šabloną. Žingsnio šablonas yra sudarytas iš daug dalių, kurios gali būti šablono tekstas arba šablono argumentas. Taip pat kiekvienas argumentas gali turėti daug žymų ir ta pati žyma gali būti keliuose argumentuose. Žyma gali turėti kelias reikšmes, kad būtų galima kurti parametrizuotus testus.



2.6 pav. Testų scenarijų modelis

2.7 pav. pateikta testų generavimo servisui veiklos diagrama. Pradedant generuoti testų scenarijus servisui pirmiausia reikalinga surinkti visus iškvietimus, kurie buvo atlikti iš testuojamo serviso. Tai atlikus surandami visi alternatyvios sąlygos išsišakojimai, nes nuo jų kiekio priklauso, kiek scenarijų bus sugeneruota, jei nėra nė vieno alternatyvos fragmento, tuomet generuojamas tik vienas scenarijus. Scenarijaus generavimo metu svarbu atrasti, kokia užklausa turi būti vykdoma. Tam analizuojami galutinių taškų parametrai, kad nustatyti užklauskos argumentus. Taip pat tikrinamas vykdymo pagal sąlygą *opt* fragmentas: jei jis egzistuoja sekų diagramoje, tuomet žingsnyje naudojama žyma su sąlygoje nurodyta reikšme, taip sukuriant parametrizuotą testą. Nustačius užklauskos argumentus tikrinama, kur siunčiama užklausa. Kai užklausa siunčiama į testuojamą mikroservisą, tuomet kuriamas siunčiamos užklauskos žingsnis su tikėtinu atsakymu, o kai siunčiama į gretimą mikroservisą, tuomet kuriamas tikėtinai išsiųstos užklauskos į kitą mikroservisą žingsnis. Scenarijaus kūrimo pabaigoje žingsniai yra surikiuojami pagal tipą laikantis *Given*, *When*, *Then* tvarkos ir sukurti scenarijai yra išsaugomi.

mikroserviso esybe, turimos operacijos yra transformuojamos į galutinius taškus, o žinutės tarp gyvavimo linijų transformuojamos į iškvietimus.



2.8 pav. PA „Transformuoti diagramą į servisų komunikacijos modelį“ veiklos diagrama

2.3.2. Testų generavime naudojami *Gherkin* žingsnių šablonai

Mikroserviso *API* sąsajai testuoti reikalinga atlikti *HTTP* užklausas ir tikrinti grąžinamus atsakymus iš testuojamo mikroserviso, todėl reikalinga turėti sprendimą, leidžiantį kurti užklausas iš skirtingų dalių. *Gherkin* kalba leidžia turėti žingsnių šablonus, kas suteikia galimybę testų scenarijuose pakartotinai naudoti žingsnius su iš anksto apibrėžta elgsena. Todėl, kad iš sekų diagramų būtų galima generuoti *Gherkin* scenarijus, apibrėžti šablonai,

naudojami *API* sąsajos testavimo procese, kurie pateikti 2.1 lentelėje. *Given* tipo žingsniai formuoja siunčiamą užklausą ir nustato, kokius atsakymus grąžina kiti mikroservisai. Vienintelis *When* žingsnis išsiunčia suformuotą užklausą. *Then* tipo žingsniai skirti patikrinti, ar buvo grąžintas tikėtinas atsakymas ir ar buvo atliktos teisingos užklauskos.

2.1 lentelė. Naudojamų šablonų generuojamuose scenarijuose sąrašas

Žingsnis	Tipas	Šablono pavadinimas	Paaiškinimas
SD1	Given	I have an HTTP "{method}" "{path}" request	Suformuoja HTTP užklausą su nurodytu adresu.
SD2	Given	I have an HTTP "{method}" "{path}" request with body "{json}"	Suformuoja HTTP užklausą su nurodytu adresu ir užklauskos turiniu.
SD3	Given	service "{serviceName}" returns response with status code {statusCode}	Nustato, kad nurodytas servisas atsakys su pateiktu statuso kodu.
SD4	Given	service "{serviceName}" returns {statusCode} for "{method}" "{path}" with body "{json}"	Nustato, kad nurodytas servisas atsakys su pateiktu statuso kodu, kai gaus nurodytą užklausą su turiniu.
SD5	When	I send the request	Įvykdo suformuotą užklausą.
SD6	Then	the response status code should be {statusCode}	Patikrina, kad testuojamas mikroservisas grąžino nurodytą statuso kodą.
SD7	Then	the response body should match "{json}"	Patikrina, kad testuojamas mikroservisas grąžino nurodytą atsakymo turinį.
SD8	Then	service "{serviceName}" was called with HTTP "{method}" "{path}"	Patikrina, kad išorinio serviso adresas buvo iškvieštas.
SD9	Then	service "{serviceName}" was called with HTTP "{method}" "{path}" with body	Patikrina, kad išorinio serviso adresas buvo iškvieštas su užklauskos turiniu.

Iš viso sukurti 9 žingsnių šablonai. Žingsniai SD1-SD2 formuoja *HTTP* užklausą, siunčiamą į testuojamą servisą. Žingsniai SD3–SD4 aprašo, kokius atsakymus turi grąžinti gretimi mikroservisai. Kadangi testas vykdomas izoliuotai ir tikri gretimi servisirai nėra paleidžiami, jų atsakymai yra simuliuojami su iš anksto apibrėžtomis reikšmėmis, todėl šie žingsniai yra būtini siekiant aprašyti testuojamo serviso aplinką. Žingsnis SD5 inicijuoja suformuotos užklauskos siuntimą. Žingsniai SD6–SD7 tikrina testuojamo serviso grąžinamą atsakymą: statuso kodą ir atsakymo turinį. Žingsniai SD8–SD9 patikrina, ar testuojamas servisas atliko aprašytas užklauskas į gretimus servisirus, todėl simuliuojami gretimi servisirai registruoja visas gautas užklauskas, po testo vykdymo galima patikrinti, ar tikrai buvo išsiųstos tikėtinos užklauskos, ir taip įsitikinti, kad servisirai veikia teisingai.

2.3.3. Sekų ir klasių diagramų braižymo taisyklės

Sėkmingam testų generavimui iš sekų diagramų reikalinga laikytis apibrėžtų braižymo taisyklių. Šiame skyriuje aprašomos taisyklės, kurių privalu laikytis kuriant tiek klasių, tiek sekų diagramas, kad diagramų elementus būtų galima transformuoti į *Gherkin* formato testų scenarijus.

2.2 lentelėje pateikiamos sekų ir klasių diagramų braižymo taisyklės. Prieš kuriant sekų diagramas, būtina sukurti klasių diagramą, kurioje vaizduojami mikroservisai ir jų teikiami galutiniai taškai. Klasių diagrama suteikia struktūrinę bazę, iš kurios sekų diagrama naudoja operacijų aprašymus, todėl netinkamai sudaryta klasių diagrama tiesiogiai paveiks scenarijų teisingumą.

Kiekvienas mikroservisas klasių diagramoje vaizduojamas kaip atskira *UML* klasė, kurios pavadinimas atitinka mikroserviso pavadinimą (T1). Kiekvienas mikroserviso teikiamas *REST* galutinis taškas vaizduojamas kaip atskira klasės operacija. Operacijos pavadinimas formuojamas pagal konvenciją *{HttpMetodas}{EndpointKelias}*, pavyzdžiui, *PostLicense* arba *GetUserById* (T2). Tokia pavadinimų konvencija pasirinkta siekiant identifikuoti konkretų galutinio taško tipą, nereikalaujant papildomų *UML* stereotipų. Jei galutinis taškas turi *URL* parametru, jie nurodomi kaip operacijos parametrai ta pačia tvarka, kuria jie pasirodo galutinio taško adrese (T3).

Du parametru pavadinimai yra rezervuoti ir turi specialią semantinę reikšmę transformacijos metu. Parametras *body* žymi *HTTP* užklauso turinį, o parametras *response* – *HTTP* atsakymo turinį (T4). Abu parametrai yra neprivalomi ir naudojami tik tada, kai galutinis taškas atitinkamai priima arba gražina turinį. Jei galutinis taškas priima užklauso turinį arba gražina atsakymo turinį, kiekvienas toks turinys modeliuojamas kaip atskira kontrakto klasė – pavyzdžiui, *CreateLicenseRequest* arba *UserResponse*. Kontrakto klasės atributai atitinka *JSON* laukų pavadinimus ir jų duomenų tipus. Būtina nurodyti konkrečius primityvius duomenų tipus: *Boolean*, *String*, *Integer*, nes tai užtikrina teisingą serializaciją (T5). Kontrakto klasės naudojamos kaip operacijų *body* ir *response* parametru tipai. Konkretiems testavimo duomenims vaizduoti naudojamos egzempliorių specifikacijos (*angl. instance specifications*), kurios priskiriamos konkrečioms kontrakto klasėms ir naudojamos kaip argumentai sekų diagramų pranešimuose (T6).

Sekų diagrama vaizduoja konkretų mikroservisų sąveikos scenarijų. Kiekviena sekų diagrama turi turėti bent dvi gyvavimo linijas, o kiekvienai gyvavimo linijai turi būti priskirtas klasifikatorius – mikroserviso klasė iš anksčiau sudarytos klasių diagramos (T7). Išimtis taikoma pirmajai gyvavimo linijai: ji gali vaizduoti išorinį aktorį, kuris inicijuoja užklausą į testuojamą mikroservisą, tokiu atveju gyvavimo linijai nereikalinga priskirti klasifikatoriaus.

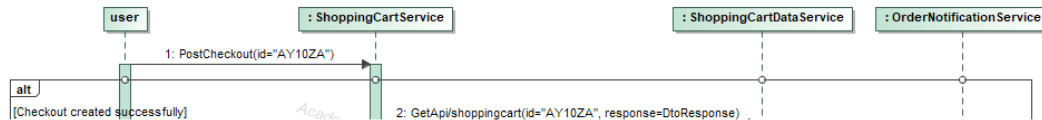
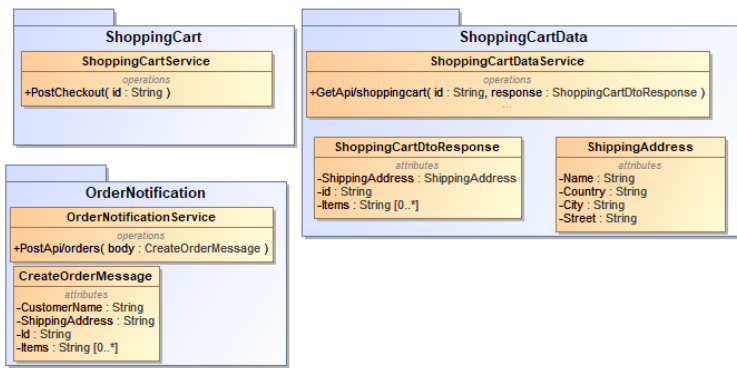
Kiekvienas pranešimas tarp gyvavimo linijų privalo nurodyti operaciją iš atitinkamos klasių diagramos (T8). Pranešimai, kuriems operacija nenurodyta, laikomi negaliojančiais ir negali būti transformuoti į scenarijaus žingsnius. Jei operacijos *body* parametras yra apibrėžtas, pranešime turi būti nurodytas atitinkamas argumentas – egzemplioriaus specifikacija arba konkreti reikšmė (T9). Jei pranešime argumentas nenurodomas, generavimo metu jo vietoje įterpiama žyma (*angl. placeholder*), kurią vėliau reikia pakeisti rankiniu būdu. Operacijos atsakymas gali būti nurodytas dviem būdais: kaip *response* parametro argumentas pranešime arba kaip *reply* pranešimo argumentas (T10). Jei atsakymas nurodytas abiem būdais, pirmenybė teikiama *reply* pranešime esančiam atsakymui. *Reply* pranešimai turi nurodyti *HTTP* statuso kodą sveikojo skaičiaus formatu – palaikomi visi standartiniai diapazonai nuo 2xx iki 5xx (T11).

Sekų diagramose skirtingiems vykdymo keliams vaizduoti gali būti naudojami alternatyvos arba sąlygos fragmentai (T12). Alternatyvos *alt* fragmentas naudojamas dviem ar daugiau skirtingų vykdymo scenarijų vaizdavimui, pavyzdžiui, sėkmės ir klaidos atvejais – kiekviena jo šaka transformuojama į atskirą sugeneruotą testavimo scenarijų. Vykdymo pagal sąlygą *opt* fragmentas naudojamas veiksmams, kurie vykdomi tik esant tam tikrai sąlygai. Ciklo *loop* fragmentas nėra palaikomas. 2.2 lentelėje pateiktos klasių ir sekų diagramų braižymo taisyklės.

2.2 lentelė. Sekų ir klasių diagramų braižymo taisyklės

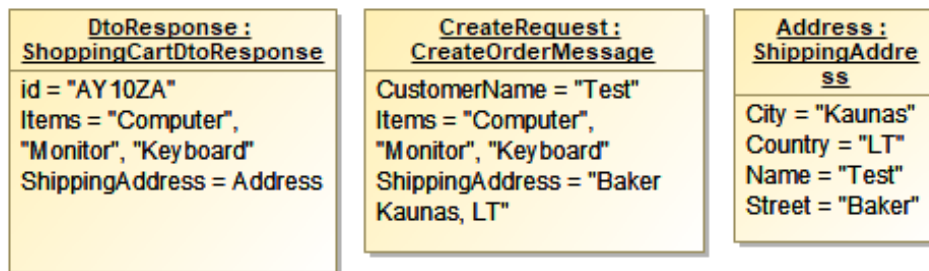
Taisyklė	Taisyklės aprašymas
T1	Kiekvienas mikroservisas klasių diagramoje vaizduojamas kaip atskira <i>UML</i> klasė, kurios pavadinimas atitinka mikroserviso pavadinimą.
T2	Kiekvienas REST galutinis taškas vaizduojamas kaip klasės operacija, pavadinta pagal konvenciją <i>{HttpMetodas}{EndpointKelias}</i> . Palaikomi metodų tipai: <i>GET, PUT, PATCH, POST, DELETE</i> .
T3	<i>URL</i> parametrai nurodomi kaip operacijos parametrai ta pačia tvarka, kuria jie pasirodo galutinio taško kelyje. Jeigu reikalinga naudoti kelis <i>URL</i> parametrus, tuomet paliekamas apatinio brūkšnio ženklas (<i>_</i>), pvz. <i>GetCompany/_/Users</i> .
T4	Parametras <i>body</i> žymi <i>HTTP</i> užklausos turinį, o <i>response</i> – <i>HTTP</i> atsakymo turinį. Abu pavadinimai yra rezervuoti.
T5	Jei galutinis taškas priima arba grąžina turinį, jis modeliuojamas kaip atskira kontrakto klasė su konkrečiais primityviaisiais duomenų tipais <i>Boolean, String, Integer</i> .
T6	Testavimo duomenys vaizduojami egzempliorių specifikacijomis, priskirtomis konkrečioms kontrakto klasėms, ir naudojami kaip argumentai sekų diagramų pranešimuose.
T7	Sekų diagrama turi turėti bent dvi gyvavimo linijas; kiekvienai gyvavimo linijai turi būti priskirtas servisas iš klasių diagramos. Pirmajai gyvavimo linijai leidžiama neturėti klasifikatoriaus – ji gali vaizduoti išorinį aktorį.
T8	Užklausoms vaizduoti naudojami sinchroniniai iškvietimo pranešimai. Ant kiekvieno pranešimo tarp gyvavimo linijų privalo būti nurodyta operacija iš klasių diagramos ir kiekvienas toks pranešimas privalo turėti atitinkamą <i>reply</i> pranešimą.
T9	Jeigu operacija turi užklausos turinį, jis turi būti paduotas naudojant <i>body</i> parametro vardą.
T10	Jeigu operacija grąžina atsakymą, jis turi būti nurodytas naudojant <i>response</i> parametro vardą arba <i>reply</i> žinutės argumentu. Jei naudojami abu, pirmenybė teikiama <i>reply</i> argumentui.
T11	<i>Reply</i> pranešimai turi nurodyti <i>HTTP</i> statuso kodą sveikojo skaičiaus formatu. Palaikomos reikšmės: 200–299, 300–399, 400–499, 500–599.
T12	Sekų diagramoje gali būti naudojami tik <i>alt</i> ir <i>opt</i> fragmentai. Kiekviena <i>alt</i> fragmento šaka transformuojama į atskirą sugeneruotą testavimo scenarijų. Ciklo <i>loop</i> fragmentas nėra palaikomas.

2.9 pav. pateiktas T1, T2, T3, T4 ir T7 taisyklių taikymo pavyzdys. Jame pateiktos mikroservisų klasės ir sekų diagramos fragmentas, vaizduojantis, kaip sukurtos klasės priskirtos gyvavimo linijoms. Taikant T1 taisyklę sukuriama klasė, vaizduojanti mikroservisus, ir taikant T7 taisyklę, sukurtos klasės yra priskiriamos prie gyvavimo linijų. Taikant T2 ir T3 taisykles sukurtos operacijos, vaizduojančios galutinius taškus. *GetApi/ShoppingCart* turi užklausos turinį, todėl remiantis T4 taisykle operacija turi *body* parametą.



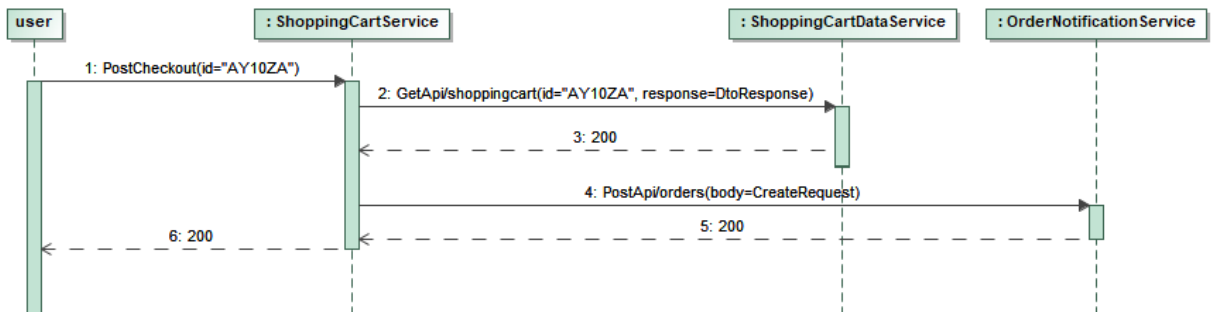
2.9 pav. Pavyzdinė mikroservisų klasių diagrama su sekų diagramos iškarpa

2.10 pav. pateiktas T6 taisyklės taikymas testavimo duomenų aprašymui naudojant egzempliorių specifikacijas. Šie egzemplioriai vėliau naudojami sekų diagramoje.



2.10 pav. Testavimo duomenų modeliavimas naudojant egzempliorių specifikaciją

2.11 pav. pateikta pavyzdinė sekų diagrama, vaizduojanti atsiskaitymo procesą. Šioje diagramoje pavaizduotas T8, T9, T10, T11 taisyklių taikymas. Remiantis T8 taisykle ant pranešimų nurodytos operacijos iš klasių diagramos. *PostApi/Orders* galutinio taško iškvietimui reikalingas užklauskos turinys, todėl pagal T9 taisyklę *body* parametru pateikiamas *CreateRequest* egzempliorius iš 2.10 pav. *GetApi/ShoppingCart* galutinis taškas grąžina atsakymo turinį, todėl pagal T10 taisyklę *response* parametru pateikiamas *DtoResponse* egzempliorius iš 2.10 pav. Pagal T11 taisyklę visi pranešimai nurodo *HTTP* statuso kodą.



2.11 pav. Pavyzdinė sekų diagrama

2.12 pav. pateiktas pavyzdinis scenarijus sukurtas pagal 2.11 pav. diagramą. Scenarijus buvo sukurtas remiantis taisyklėmis iš 2.1 lentelės. Pirmasis pranešimas, turintis priskirtą operaciją *PostCheckout*, kreipiasi į testuojamą servisą. Šis pranešimas neturi užklausos turinio, todėl užklausos formavimui naudojamas SD1 žingsnio šablonas. *ShoppingCartDataService* gražina atsakymą su turiniu, todėl naudojamas SD4 žingsnio šablonas, kad būtų simuliuojamas atsakymas iš serviso. Priešingai, *OrderNotificationService* mikroserviso galutinis taškas negražina atsakymo turinio, todėl naudojamas SD3 žingsnio šablonas. Kadangi testuojamas *ShoppingCartService* gražina 200 statuso kodą, todėl ir SD6 žingsnis nurodo tokį patį 200 statuso kodą. *ShoppingCartDataService* mikroserviso galutinis taškas *GetApi/ShoppingCart* neturi užklausos turinio, todėl užklausos iškvietimo patikrinimui naudojamas SD8 žingsnis. Priešingai, *OrderNotificationService* mikroserviso galutinis taškas turi užklausos turinį, todėl naudojamas SD9 žingsnio šablonas. Pateiktame scenarijuje taip pat matoma, kaip egzempliorių specifikacijos iš 2.10 pav. yra transformuotos į *JSON* formatą.

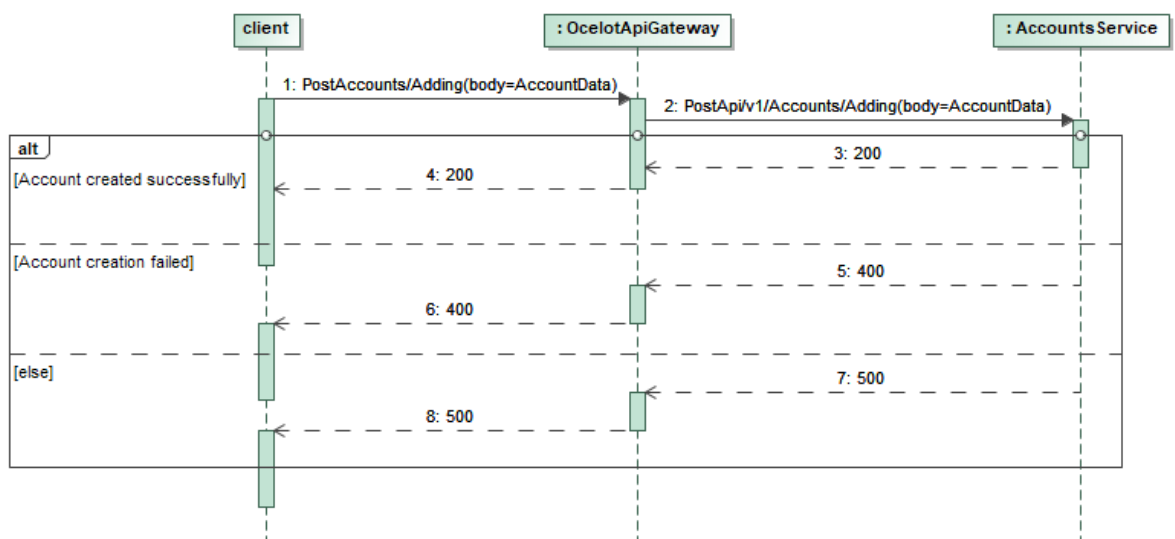
Scenario: Scenario 4

```
Given I have an HTTP "POST" "checkout/AY10ZA" request
And service "ShoppingCartDataService" returns 200 for "GET" "api/shoppingcart/AY10ZA" with body
***
{
  "shippingAddress": {
    "name": "Test",
    "country": "LT",
    "city": "Kaunas",
    "street": "Baker"
  },
  "id": "AY10ZA",
  "items": [
    "Computer",
    "Monitor",
    "Keyboard"
  ]
}
***

And service "OrderNotificationService" returns 200 for "POST" "api/orders"
When I send the request
Then the response status code should be 200
And service "ShoppingCartDataService" was called with "GET" "api/shoppingcart/AY10ZA"
And service "OrderNotificationService" was called with "POST" "api/orders" and body
***
{
  "customerName": "Test",
  "shippingAddress": "Baker Kaunas, LT",
  "items": [
    "Computer",
    "Monitor",
    "Keyboard"
  ]
}
***
```

2.12 pav. Atsiskaitymo proceso pavyzdinis testų scenarijus

2.13 pav. pateikta kitos sistemos sekų diagrama, kurioje taikoma T12 taisyklė skirtingų scenarijų atvaizdavimui naudojant alternatyvos fragmentą. Kiekvienas *alt* fragmentas vaizduoja, kaip testuojamas mikroservisas turi elgtis priklausomai nuo grąžinamo atsakymo statuso.



2.13 pav. Sekų diagrama su alternatyvos fragmentu

2.14 pav. pateikti 3 scenarijai, sukurti pagal 2.13 pav. pavaizduotą diagramą. Iš viso sukurti trys testų scenarijai, nes diagramoje buvo naudojami trys alternatyvos fragmentai. Kadangi į testuojamą servisą siunčiama užklausa su turiniu, todėl naudojamas SD2 žingsnio šablonas užklauskos su turiniu formavimui.

```
Scenario: Scenario 1
Given I have an HTTP "POST" "accounts/adding" request with body
---
{
  "customerId": "e77f1920-c7ac-4169-86af-0d6a941d8097",
  "accountId": "ae6ce3e2-d3eb-43bd-a8ec-8e4749ac5a2c",
  "amount": 10
}
---
And service "AccountsService" returns 500 for "POST" "api/v1/accounts/adding"
When I send the request
Then the response status code should be 500
And service "AccountsService" was called with "POST" "api/v1/accounts/adding" and body
---
{
  "customerId": "e77f1920-c7ac-4169-86af-0d6a941d8097",
  "accountId": "ae6ce3e2-d3eb-43bd-a8ec-8e4749ac5a2c",
  "amount": 10
}
---

Scenario: Scenario 2
Given I have an HTTP "POST" "accounts/adding" request with body
---
{
  "customerId": "e77f1920-c7ac-4169-86af-0d6a941d8097",
  "accountId": "ae6ce3e2-d3eb-43bd-a8ec-8e4749ac5a2c",
  "amount": 10
}
---
And service "AccountsService" returns 400 for "POST" "api/v1/accounts/adding"
When I send the request
Then the response status code should be 400
And service "AccountsService" was called with "POST" "api/v1/accounts/adding" and body
---
{
  "customerId": "e77f1920-c7ac-4169-86af-0d6a941d8097",
  "accountId": "ae6ce3e2-d3eb-43bd-a8ec-8e4749ac5a2c",
  "amount": 10
}
---

Scenario: Scenario 3
Given I have an HTTP "POST" "accounts/adding" request with body
---
{
  "customerId": "e77f1920-c7ac-4169-86af-0d6a941d8097",
  "accountId": "ae6ce3e2-d3eb-43bd-a8ec-8e4749ac5a2c",
  "amount": 10
}
---
And service "AccountsService" returns 200 for "POST" "api/v1/accounts/adding"
When I send the request
Then the response status code should be 200
And service "AccountsService" was called with "POST" "api/v1/accounts/adding" and body
---
{
  "customerId": "e77f1920-c7ac-4169-86af-0d6a941d8097",
  "accountId": "ae6ce3e2-d3eb-43bd-a8ec-8e4749ac5a2c",
  "amount": 10
}
---
```

2.14 pav. Paskyros sukūrimo pavyzdiniai testų scenarijai

2.4. Reikalavimų apibendrinimas

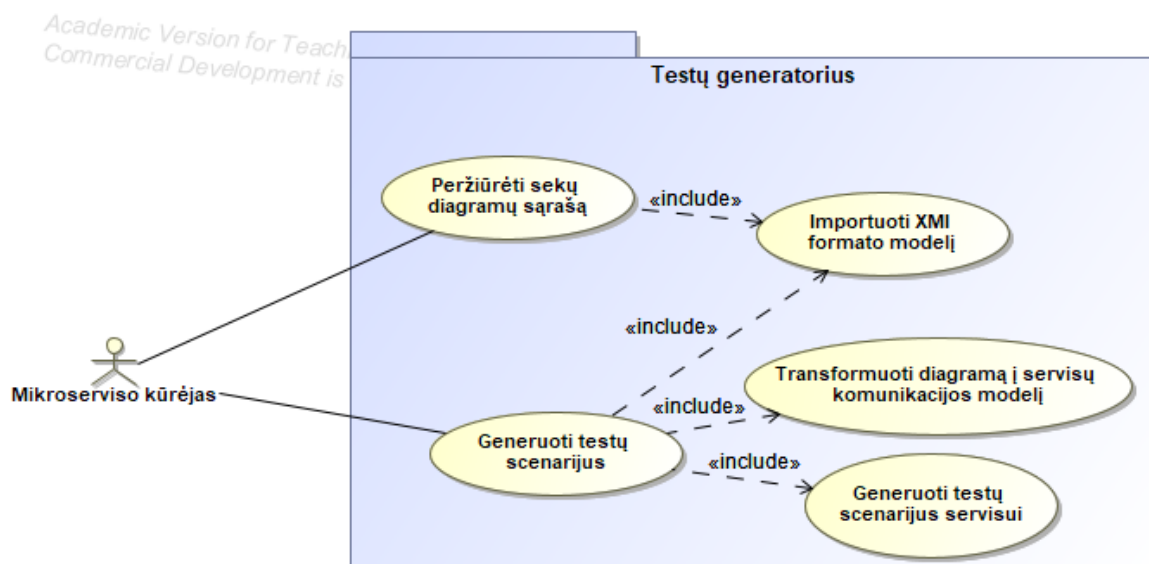
Mikroservisų testų generavimo iš sekų diagramų metodikoje išskirta, kad pagrindinis aktorius yra mikroservisų kūrėjas. Apibrėžta, kad generuojant scenarijus, pirmiausia reikia sukurti mikroservisų projekto *UML* modelį, kuriame būtų vaizduojami servisi ir komunikacija tarp jų naudojant sekų diagramas. Naudojant testų generavimo įrankį sugeneruojami testų scenarijai, kurie automatizuotai įkeliami į mikroservisų projektą. Apibrėžti 9 žingsnių šablonai, kuriuos naudojant būtų generuojami testų scenarijai pagal pranešimus siunčiamus tarp gyvavimo linijų.

3. Testų generavimo iš sekų diagramų įrankio projektas

Šiame skyriuje aprašomas testų generavimo iš sekų diagramų įrankio projektas.

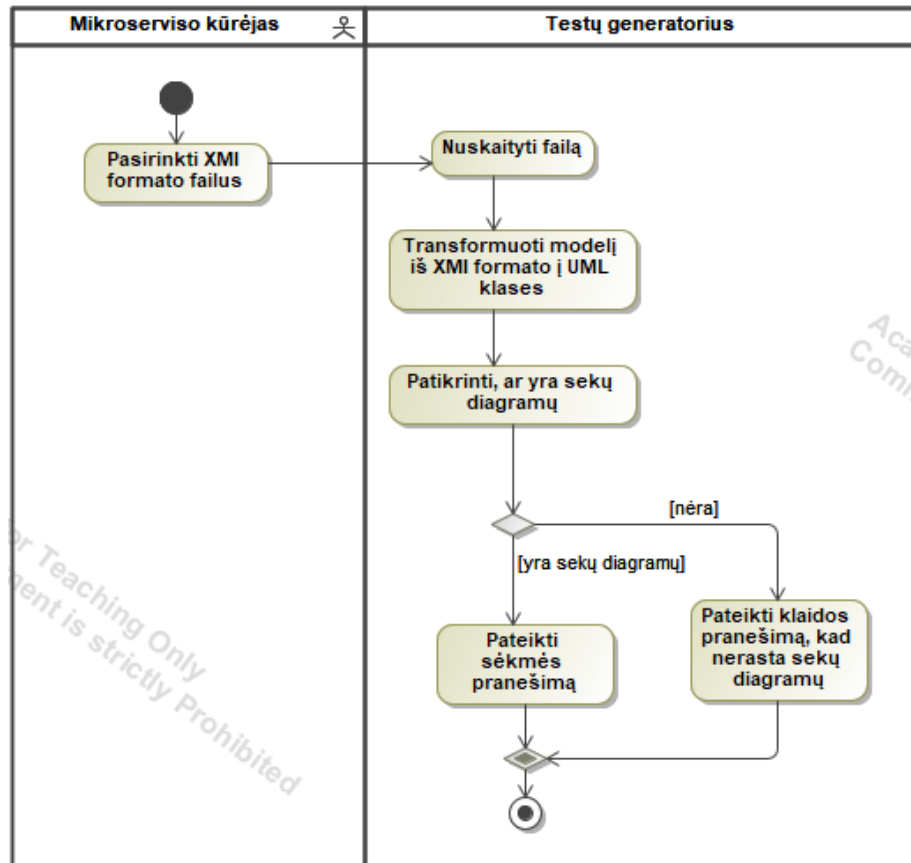
3.1. Įrankio reikalavimų specifikacija

Įrankio panaudojimo atvejų modelis vaizduojamas 3.1 pav. Naudotojas gali peržiūrėti sekų diagramų sąrašą ir generuoti testus iš įkeltų sekų diagramų.



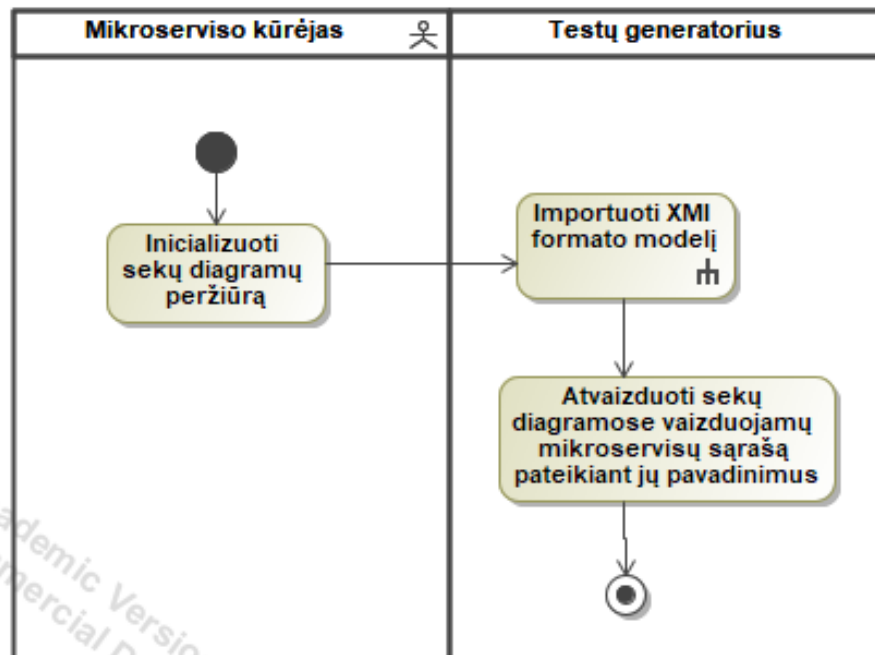
3.1 pav. Testų generatoriaus įrankio panaudojimo atvejai

Sekų diagramų importavimo procesas pateikiamas 3.2 pav. Naudotojas norėdamas importuoti diagramą turi pasirinkti *XMI* formato failą. Pasirinktas failas yra nuskaitytas, sukuriama *XMI* modelis, kuris transformuojamas į *UML* elementus. Jeigu tarp *UML* elementų nebuvo rasta sekų diagramų, apie tai informuojama klaidos pranešimu.



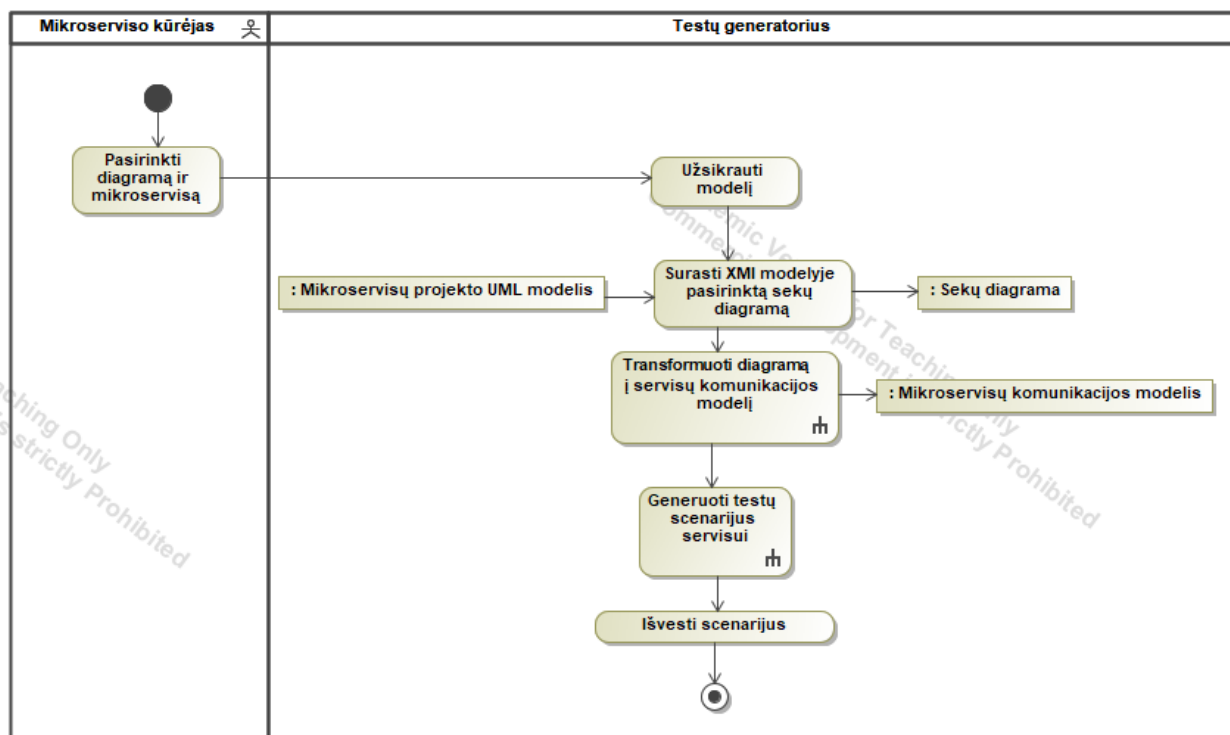
3.2 pav. PA „Importuoti XMI formato modelį“ veiklos diagrama

Sekų diagramų peržiūros procesas pateikiamas 3.3 pav. Naudotojas gali peržiūrėti sekų diagramų ir jose vaizduojamų mikroservisų pavadinimus.



3.3 pav. PA „Peržiūrėti sekų diagramų sąrašą“ veiklos diagrama

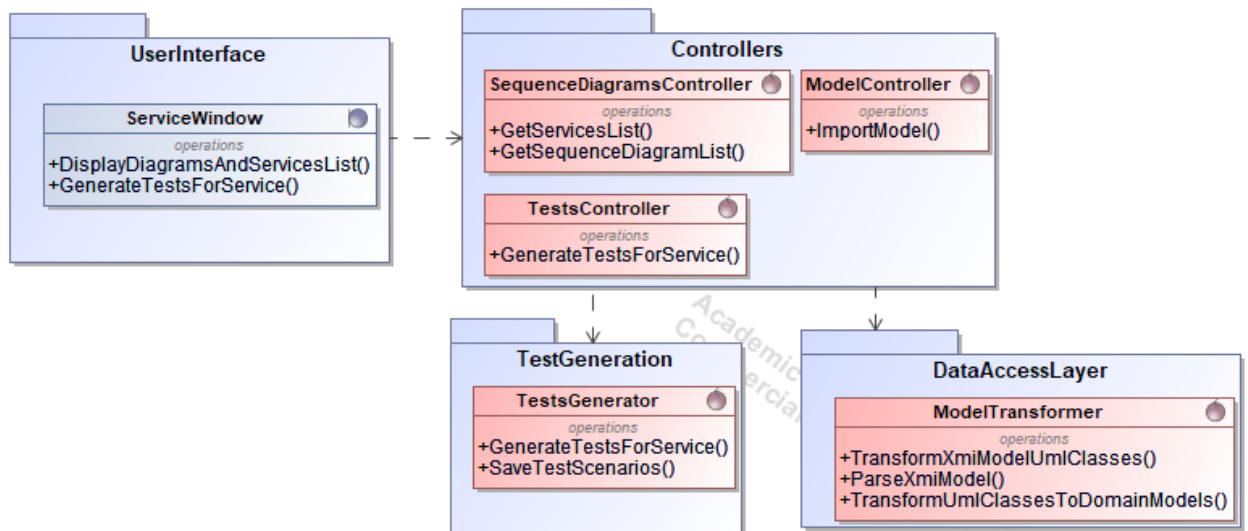
Testų generavimo iš sekų diagramų procesas pateikiamas 3.4 pav. Prieš generuojant scenarijus pirmiausia naudotojas pasirenka konkrečią diagramą ir servisą, kuriam generuojami scenarijai. Sugeneruoti scenarijai po generavimo eksportuojami į nurodytą aplanką.



3.4 pav. PA „Generuoti testų scenarijus“ veiklos diagrama

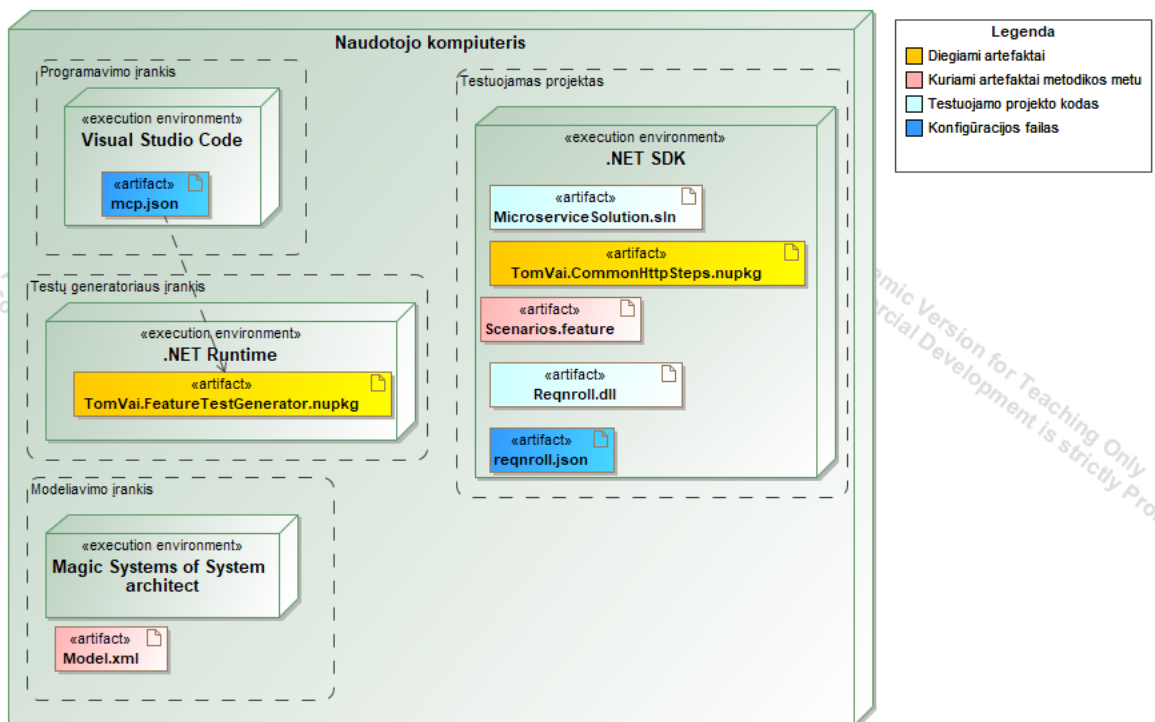
3.2. Įrankio projekto modelis

Įrankio projekto modelis yra suskaidytas į keturis pagrindinius paketus, kurie pateikti 3.5 pav. esančioje diagramoje. Siekiant laikytis švarios architektūros (*angl. Clean architecture*) principo, testų generavimo logika *TestGeneration* pakete nepriklauso nuo kitų paketų, t.y. neturi ryšių su gretimais paketais. Naudotojo sąsajos elementai talpinami atskirame *UserInterface* pakete, valdikliai, kurie teikia ir priima duomenis iš naudotojo sąsajos, yra patalpinti *Controllers* pakete. Modelio transformacijos ir duomenų saugyklos klasės laikomos *DataAccessLayer* pakete.



3.5 pav. Įrankio paketų diagrama

3.6 pav. pateikta testų generavimo įrankio diegimo diagrama. Testų generavimo įrankis diegiamas naudotojo kompiuteryje kaip atskira programa. Diegimo diagramoje taip pat pavaizduoti *Magic Systems of System Architect* modeliavimo įrankis ir mikroservisų projektas, vykdomas naudojant *.NET SDK*. Naudotojas gali pasirinkti kitą modeliavimo įrankį, kuris gali eksportuoti modelius *XMI* formatu, taip pat alternatyvią mikroservisų testų biblioteką, palaikančią *Gherkin* formato testų vykdymą.



3.6 pav. Testų generavimo įrankio diegimo diagrama

Nors visi įrankiai vaizduojami esantys tame pačiame naudotojo kompiuteryje, jie gali būti diegiami ir skirtinguose kompiuteriuose. Tai suteikia galimybę keliems asmenims bendradarbiauti atliekant modeliavimo, testų generavimo ir vykdymo užduotis.

3.3. Įrankio projekto apibendrinimas

Šiame skyriuje aprašytas testų generavimo įrankio projektas apima funkcinis reikalavimus, panaudojimo atvejų modelį ir architektūrinį sprendimą. Apibrėžti trys pagrindiniai panaudojimo atvejai: modelio importavimas, sekų diagramų peržiūra ir testų scenarijų generavimas. Projekto modelis suskaidytas į keturis paketus laikantis švarios architektūros principo, kas užtikrina generavimo logikos nepriklausomumą nuo naudotojo sąsajos ir duomenų prieigos sluoksnių. Diegimo diagrama parodo, kad įrankis gali veikti kartu su bet koku *XMI* formatą palaikančiu modeliavimo įrankiu ir bet koku *Gherkin* formatą palaikančiu testavimo karkasu.

4. Testų generavimo iš sekų diagramų įrankio realizacija ir testavimas

Šiame skyriuje pateikiami prototipui įgyvendinti naudojamos technologijos, diegimo instrukcija ir pavyzdinis naudojimo atvejis.

4.1. Testų generavimo iš sekų diagramų įrankio realizacijos ir veikimo aprašas

Testų generatorius¹ buvo sukurtas naudojant *C#* programavimo kalbą ir *.NET 10* karkasą [53]. Testų generavimo logika realizuota nenaudojant papildomų išorinių bibliotekų – remtasi tik standartinėmis sisteminėmis bibliotekomis, kurias teikia *.NET 10* karkasas. *MCP* serverio sąsajos palaikymui pasirinkta konsolės tipo aplikacija, kurioje naudojama *ModelContextProtocol .NET* biblioteka.

Bendrųjų testų žingsnių biblioteka taip pat realizuota naudojant *C#* programavimo kalbą. Testų vykdymui naudojama viena populiariausių testavimo bibliotekų – *xUnit*, o *BDD* žingsnių interpretavimui ir vykdymui pasitelkta *Reqnroll* biblioteka [54]. Užklausų simuliacijai naudojama *WireMock.Net* biblioteka [55]. Testų žingsnių *NuGet* paketas sukompiliuotas naudojant *.NET 10* karkasą, todėl projektas, kuriame jis integruojamas, taip pat turi naudoti ne žemesnę nei *.NET 10* versiją.

4.2. Diegimo instrukcija

Norint sėkmingai naudotis testų generatoriumi, būtina tinkamai įdiegti įrankį bei sukonfigūruoti testuojamą projektą. Šiame skyriuje pateikiama diegimo ir pradinės konfigūracijos instrukcija.

4.2.1. Įrankio diegimas

Įrankis buvo supakuotas kaip *NuGet* įrankis ir patalpintas viešojoje artefaktų saugykloje. Norint naudotis įrankiu, būtina įdiegti *.NET 10* vykdymo aplinką. Sėkmingai įdiegus vykdymo aplinką, įrankį galima įsidiegti naudojant 4.1 pav. pateiktą komandą.

```
dotnet tool install --global --version 0.1.0-beta Tomvai.FeatureTestGenerator
you can invoke the tool using the following command: feature-test-generator
tool 'tomvai.featuretestgenerator' (version '0.1.0-beta') was successfully installed.
```

4.1 pav. Testų generatoriaus diegimo komanda ir išvestis

Siekiant patikrinti, ar įrankis buvo sėkmingai įdiegtas, reikalinga iškviešti komandą, pateiktą 4.2 pav. Įvykdžius komandą neturėtų būti pateikiami klaidų pranešimai, o informacinis išvesties tekstas turėtų nurodyti, kad įrankis buvo sėkmingai paleistas.

¹ Testų generatoriaus prototipo kodo saugykla: <https://github.com/tomvai15/MicroserviceTestGenerator>

```
PS C:\Users\Tomas> feature-test-generator
info: ModelContextProtocol.Server.StdioServerTransport[857250842]
Server (stream) (FeatureTestGenerator.Mcp) transport reading messages.
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
Content root path: C:\Users\Tomas
```

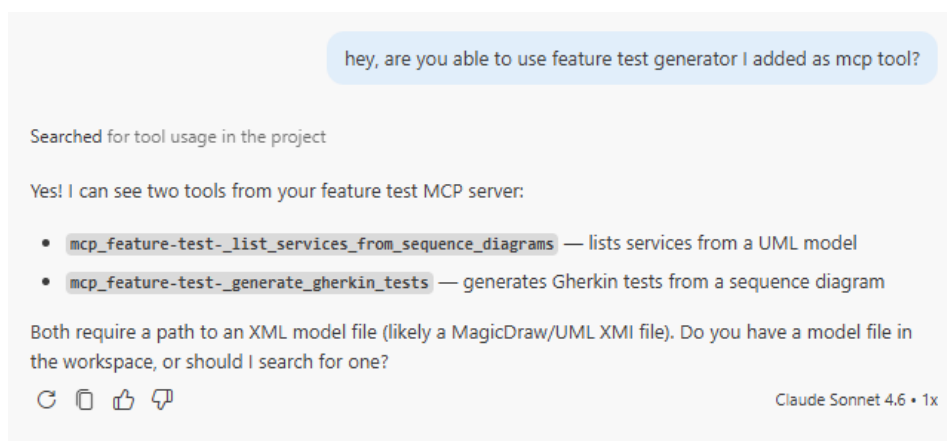
4.2 pav. Testų generatoriaus įrašymo tikrinimas

Kad įrankis veiktų kartu su *GitHub Copilot* įskiepiu *Visual Studio Code* aplinkoje, būtina testų generatoriaus įrankį pridėti kaip palaikomą *MCP* serverį. Paleidus *Visual Studio Code* programą ir paspaudus klavišų kombinaciją *Ctrl+P*, atidaromas paieškos langas. Šiame lange reikia surasti ir atidaryti *mcp.json* failą. Atidaryto failo serverių sekcijoje būtina pridėti *feature-test-generator* įrankį, kaip parodyta 4.3 pav.

```
1 {
2   "servers": {
3     "feature-test-mcp": {
4       "type": "stdio",
5       "command": "feature-test-generator"
6     }
7   },
8   "inputs": []
9 }
```

4.3 pav. Failo *mcp.json* konfigūracijos pavyzdys

Siekiant patikrinti, ar serveris yra sėkmingai pridėtas, galima įvykdyti užklausą, kuri tikrina įrankio pasiekiamumą 4.4 pav. Atliekant patikrą buvo naudojamas *Claude Sonnet 4.6* modelis.



4.4 pav. MCP serverio konfigūracija su patikrinimo užklausa

4.2.2. Testuojamo projekto konfigūravimas

Testų generatorius generuoja scenarijus naudodamas žingsnių šablonus. Žingsnių šablonai, skirti *.NET* karkasui, yra įgyvendinti atskiroje bibliotekoje ir pateikiami kaip *NuGet* paketas. Šis paketas sukompiliuotas naudojant *.NET 10* karkasą, todėl testų projektas, kuriame šis paketas naudojamas, privalo naudoti *.NET 10* arba naujesnę versiją. Kadangi testų projektas turi nuorodą į testuojamą projektą (pvz., *API* projektą), o ne atvirkščiai, testuojamas projektas gali naudoti žemesnę *.NET* karkaso versiją – tai neturi įtakos testų projekto suderinamumui. *NuGet* paketą galima įdiegti naudojant 4.5 pav. pateiktą komandą.

```
dotnet add package Tomvai.CommonHttpSteps --version 0.1.5-beta
```

4.5 pav. Bendrų žingsnių šablonų bibliotekos diegimas

Pridėjus biblioteką, reikalinga sukurti failą *reqnroll.json* ir jame nurodyti, kad žingsnių šablonai būtų naudojami iš pridėtos bibliotekos.

```
{  
  "$schema": "https://schemas.reqnroll.net/reqnroll-config-latest.json",  
  "bindingAssemblies": [  
    {  
      "assembly": "Tomvai.CommonHttpSteps"  
    }  
  ]  
}
```

4.6 pav. Failo *reqnroll.json* konfigūracija

Norint, kad bendri žingsniai veiktų su testuojamu mikroservisu, testavimo projektą reikia papildomai sukonfigūruoti, pridėdant testavimo komponentų registraciją, kaip pateikta 4.7 pav.

```

using ...
[assembly: CollectionBehavior(DisableTestParallelization = true)]
namespace FeatureTestFramework.Tests;

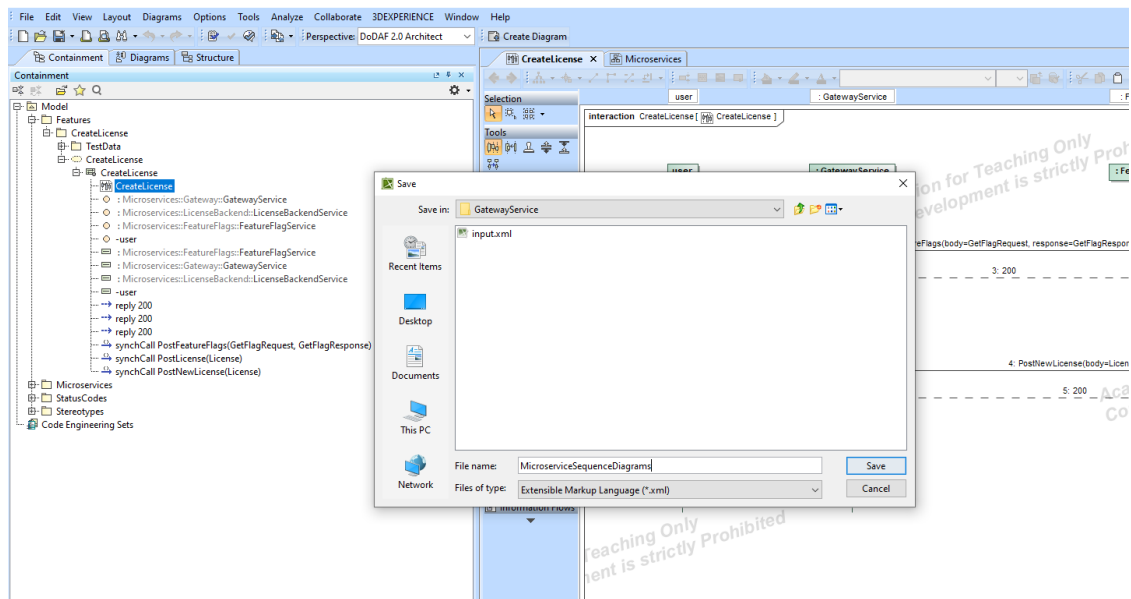
[Binding]
tomvai15 +1
public class TestStartup
{
    [BeforeTestRun(Order = TestRunOrder.InjectServices)]
    tomvai15 +1
    public static void RegisterServices()
    {
        GlobalTestStartup.SetupHttpsSteps<Program>();
    }
}

```

4.7 pav. Bendrų žingsnių diegimas testavimo projekte

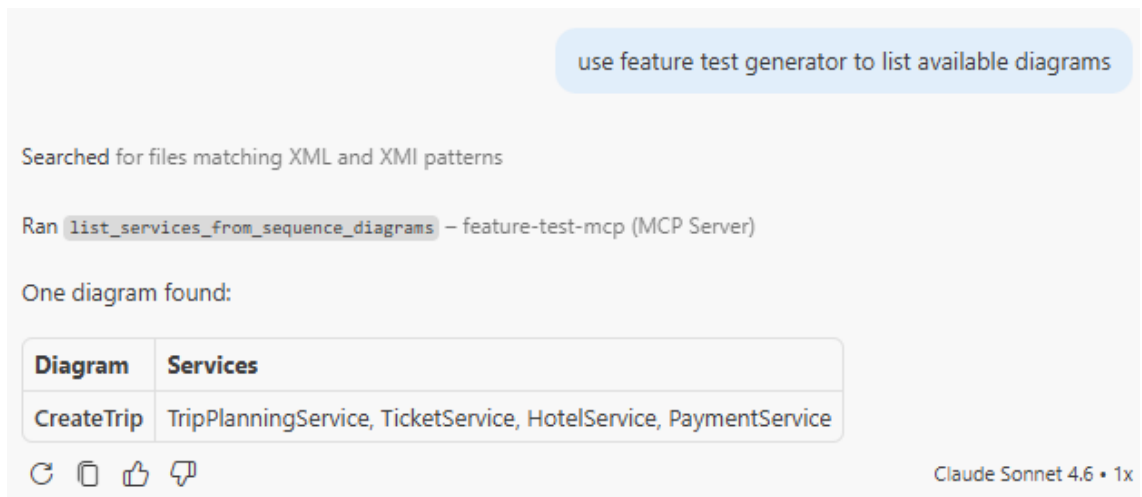
4.2.3. Testų generatoriaus naudojimas

Prieš generuojant testų scenarijus, reikalinga *UML* diagramų projektą išsaugoti *XML* failo formatu naudojant *Save Project As* parinktį, kaip pateikta 4.8 pav.



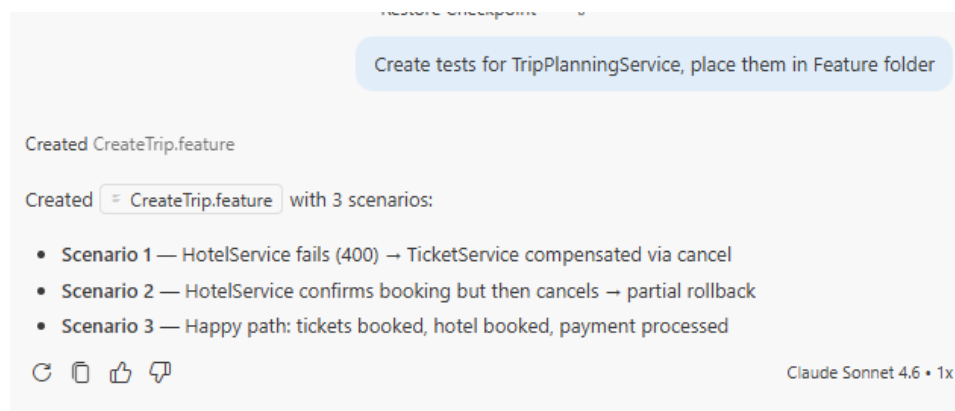
4.8 pav. Modelio eksportavimo langas

Išsaugojus diagramų failą, atidaromas testų projektas *Visual Studio Code* programoje ir atidaromas *GitHub Copilot* įrankis. Atliekama užklausa, kurioje prašoma išvesti palaikomas sekų diagramas naudojant testų generatoriaus įrankiu. Naudojama užklausa anglų kalba „Use feature test generator to list available diagrams“ kaip pateikta 4.9 pav.



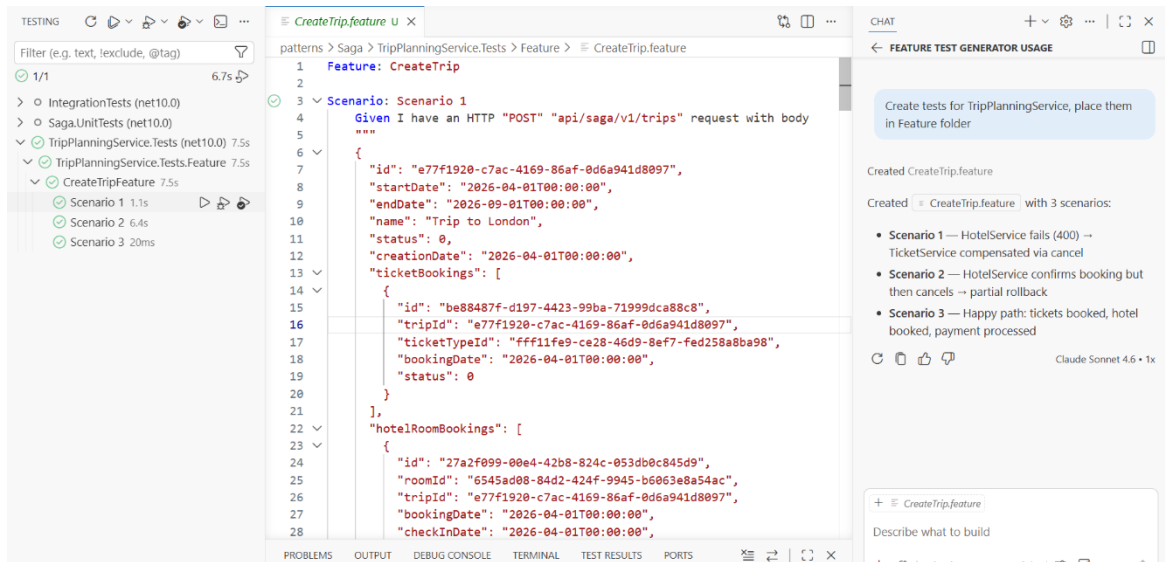
4.9 pav. Diagramų išvedimas naudojant *GitHub Copilot* įrankį kaip sąsają

Išvedus diagramų pavadinimus, galima paprašyti, kad mikroservisui iš diagramos būtų sugeneruoti scenarijai 4.10 pav.



4.10 pav. Testų scenarijų generavimas naudojant *GitHub Copilot* įrankį kaip sąsają

Sugeneruoti testų scenarijai buvo automatiškai pridėti į naują failą, kuris buvo sukurtas testų projekte. 4.11 pav. pateikta, kaip sugeneruoti scenarijai yra sėkmingai vykdomi.



4.11 pav. Sugeneruotų testų vykdymo pavyzdys

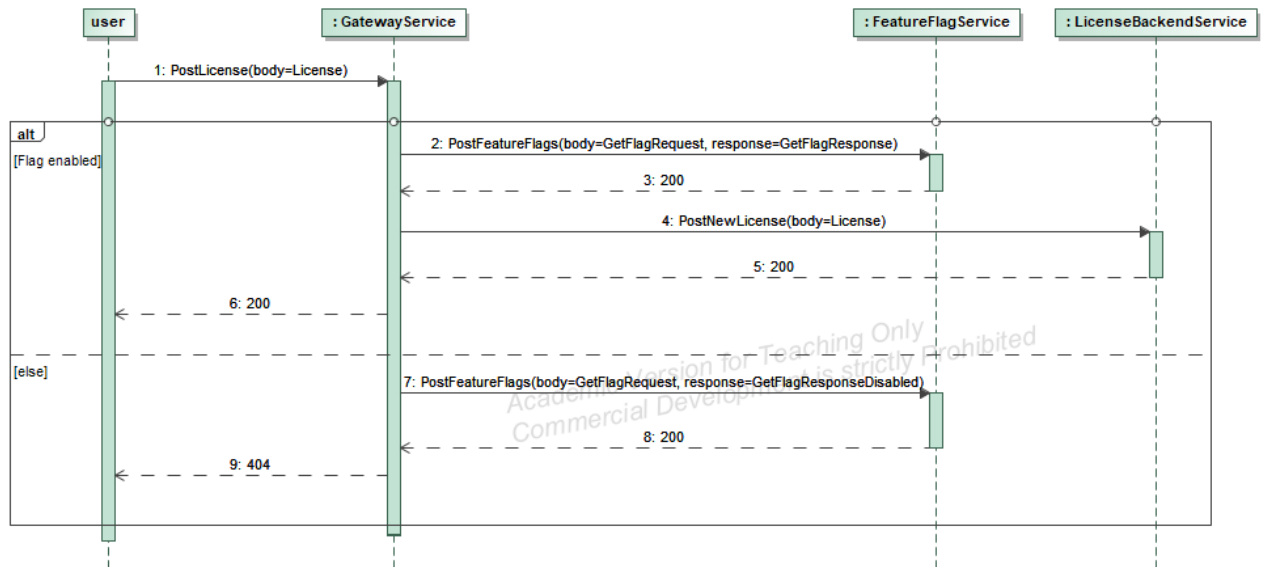
4.3. Testavimo modelis, duomenys, rezultatai

Siekiant užtikrinti testų generatoriaus kokybę, buvo naudojami skirtingų tipų testai: vienetų, integraciniai. Prototipo kokybei užtikrinti nuo pat realizacijos pradžios buvo taikoma testais grindžiamo programų kūrimo metodologija. Pirmiausia buvo kuriami aukšto abstrakcijos lygio testai, pagal kuriuos nuosekliai buvo atnaujinama realizacija, nuolat tikrinant testų rezultatus.

4.3.1. Testavimo duomenys

Testavimui iš viso buvo sukurta 10 diagramų, skirtų patikrinti skirtingus testų generatoriaus aspektus. Buvo sukurta viena diagrama, kuri tikrina patį paprasčiausią scenarijų, pavyzdžiui, kai servisas neatlieka jokių papildomų užklausų ir tik grąžina sėkmingą atsakymą. Siekiant patikrinti *alt* ir *opt* sąlygos fragmentų palaikymą, buvo sukurtos trys diagramos, kuriose naudojami šie fragmentai, kad būtų atlikta papildoma ar kitokia užklausa negu pagrindiniame scenarijuje. Teisingam užklauskos *BDD* žingsnių formatavimui buvo sukurtos trys sekų diagramos su užklauskos, atsakymo argumentais. trys sekų diagramos vaizdavo komunikaciją tarp servisų ir tikrina, ar teisingai generuojami žingsniai, tikrinantys užsiųstas užklauskas. Be sekų diagramų taip pat buvo sukurtos devynios klasių diagramos su klasių ir objektų elementais, kurie buvo naudojami sekų diagramose.

Žemiau pateikti du testavimui naudotų diagramų pavyzdžiai ir sugeneruoti scenarijai. 4.12 pav. pateiktas alternatyvos fragmento taikymas skirtingų scenarijų atvaizdavimui.



4.12 pav. Sekų diagrama skirta testuoti alternatyvos fragmento veikimą

4.13 pav. pateikti sugeneruoti scenarijai 4.12 pav. sekų diagramai. Buvo sėkmingai sugeneruoti du scenarijai, testuojantys mikroservisą, kai įjungtas ir išjungtas licencijos sukūrimo funkcionalumas.

```

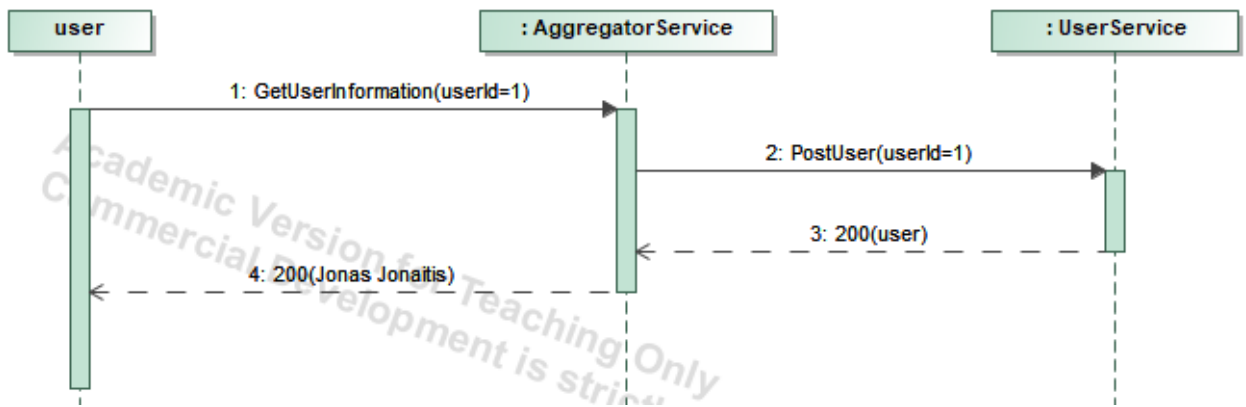
Feature: TestCases\8_GatewayServiceAltScenario-GatewayService

Scenario: Scenario 1
  Given I have an HTTP "POST" "license" request with body
  ...
  {
    "productType": "VideoEditor",
    "featureLevel": "Basic",
    "userId": "A1SASX"
  }
  ...
  And service "FeatureFlagService" returns 200 for "POST" "featureFlags" with body
  ...
  {
    "featureFlag": "create_licenses_enabled",
    "isEnabled": false
  }
  ...
  When I send the request
  Then the response status code should be 404
  And service "FeatureFlagService" was called with "POST" "featureFlags" and body
  ...
  {
    "featureFlag": "create_licenses_enabled"
  }
  ...

Scenario: Scenario 2
  Given I have an HTTP "POST" "license" request with body
  ...
  {
    "productType": "VideoEditor",
    "featureLevel": "Basic",
    "userId": "A1SASX"
  }
  ...
  And service "FeatureFlagService" returns 200 for "POST" "featureFlags" with body
  ...
  {
    "featureFlag": "create_licenses_enabled",
    "isEnabled": true
  }
  ...
  And service "LicenseBackendService" returns 200 for "POST" "newLicense"
  When I send the request
  Then the response status code should be 200
  And service "FeatureFlagService" was called with "POST" "featureFlags" and body
  ...
  {
    "featureFlag": "create_licenses_enabled"
  }
  ...
  And service "LicenseBackendService" was called with "POST" "newLicense" and body
  ...
  {
    "productType": "VideoEditor",
    "featureLevel": "Basic",
    "userId": "A1SASX"
  }
  ...
  
```

4.13 pav. Sugeneruoti testų scenarijai alternatyvos fragmento testavimo atveju

4.14 pav. pateikta sekų diagrama, kurioje testuojamas mikroservisas grąžina atsakymą su turiniu.



4.14 pav. Sekų diagrama skirta testuoti grąžinamą atsakymo turinį

4.15 pav. pateiktas sugeneruotas scenarijus, kuris tikrina grąžinamą atsakymo turinį.

```

Feature: TestCases\10_MultipleServices_ServiceWasCalledWithResponse-AggregatorService

Scenario: Scenario 1
  Given I have an HTTP "GET" "userinformation/1" request
  And service "UserService" returns 200 for "POST" "user/1" with body
  """
  {
    "name": "Jonas",
    "surname": "Jonaitis"
  }
  """
  When I send the request
  Then the response status code should be 200
  And the response body should match
  """
  {
    "fullName": "Jonas Jonaitis",
    "orderNumber": "AABBCC"
  }
  """
  And service "UserService" was called with "POST" "user/1"
  
```

4.15 pav. Sugeneruotas scenarijus su atsakymo turinio tikrinimu

Iš viso buvo sukurti 27 automatiniai testai, kurie tikrina scenarijų generavimo logiką, užklausų ir atsakymų formatavimą į JSON tipo tekstą. Automatiniam testavimui buvo naudojamas .NET 10 karkasas ir xUnit testavimo karkasas.

4.3.2. Vienetų testavimas

Vienetų testavimas buvo vykdomas siekiant įvertinti įrankio gebėjimą teisingai importuoti elementus iš failo bei užtikrinti korektišką XMI modelio elementų konvertavimą į UML modelio elementus. Kadangi testų generavimo procesas yra tiesiogiai priklausomas nuo sekų diagramų elementų, ypatingas dėmesys buvo skirtas konvertavimo mechanizmo patikimumui ir tikslumui.

Testavimo metu buvo analizuojamos šios funkcionalumo sritys: žingsnių *Given, When, Then* kūrimo ir formatavimo logika, scenarijų parametrų apdorojimas bei scenarijų išsaugojimo procesas. Be to, buvo tikrinamas *HTTP* operacijų identifikavimo mechanizmas bei užklausų ir atsakymų konvertavimas į *JSON* formatą. 4.16 pav. pateikti testų vykdymo rezultatai.

```

✓ [C#] FeatureTestGenerator.Domain.UnitTests (5 tests) Success
  ✓ {} FeatureTestGenerator.Domain.UnitTests (5 tests) Success
    > ✓ DomainSimpleTests (5 tests) Success
  ✓ [C#] FeatureTestGenerator.Infrastructure.UnitTests (10 tests) Success
    ✓ {} FeatureTestGenerator.Infrastructure.UnitTests (10 tests) Success
      > ✓ {} ModelParsing (3 tests) Success
      > ✓ {} ModelTransformation (1 test) Success
      > ✓ IIdentifiableExtensionsTests (2 tests) Success
      > ✓ PackageFindTests (2 tests) Success
      > ✓ SequenceDiagramConverterTests (2 tests) Success

```

4.16 pav. Vienetų testų vykdymo rezultatas

4.3.3. Integraciniai testai

Integraciniai testai tikrino, ar sistema gali apdoroti realius *XMI* modelius ir sugeneruoti atitinkamus *BDD* testų failus. Integraciniais testais buvo patikrinta visa testų generatoriaus eiga: modelio nuskaitymas, modelio konvertavimas į servisų modelį, scenarijų generavimas ir sugeneruoto rezultato palyginimas su numatytu.

Kiekvienam integracinio testo atvejui naudojamas atskiras katalogas su įvesties failu ir numatytu rezultatu. Kiekvienas integracinis atvejis organizuojamas katalogo principu:

- input.xml – *XMI* modelio failas.
- result.feature – tikėtino rezultato failas su *BDD* scenarijais.
- output.feature – sugeneruotas rezultatas (sukuriamas bandymo metu).

Bandymo rinkinyje įtraukti pavyzdžiai apima tiek paprastas sekų diagramas su viena *HTTP* užklausa, tiek sudėtingus kelių *HTTP* užklausų scenarijus. Taip pat naudojamos diagramos su alternatyvos ir sąlygos fragmentais. 4.17 pav. pateikti integracinių testų vykdymo rezultatai.

```

✓ Parse_Returns_ParsedModel (12 tests) Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\10_MultipleServices_ServiceWasCalledWith...", serviceName: "AggregatorService") Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\11_AllStatusCodes-AggregatorService", serviceName: "AggregatorService") Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\12_AllHttpMethodTypes-AggregatorService", serviceName: "AggregatorService") Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\1_InitialScenario", serviceName: null) Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\2_OptScenario", serviceName: null) Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\3_AltScenario", serviceName: null) Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\4_SingleService_Single_Request_WithBody", serviceName: null) Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\5_SingleService_Single_Request_WithPara"... , serviceName: null) Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\6_SingleService_Single_Request_WithResp"... , serviceName: null) Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\7_GatewayService-GatewayService", serviceName: "GatewayService") Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\8_GatewayServiceAltScenario-GatewayServ"... , serviceName: "GatewayService") Success
  ✓ Parse_Returns_ParsedModel(directory: "TestCases\\9_MultipleServices_ServiceWasCalledWith"... , serviceName: "AggregatorService") Success

```

4.17 pav. Integracinių testų vykdymo rezultatas

Testų apibūdinimai:

- *AltScenario* – naudojamas alternatyvos fragmentas, siekiant patikrinti, ar buvo sugeneruoti du scenarijai.
- *GatewayService* – tikrinama, ar teisingai generuojami scenarijai, tikrinantys iškvietas užklausas.
- *GatewayServiceAltScenario* – tikrinama, ar teisingai generuojami scenarijai, tikrinantys iškvietas užklausas, kai naudojamas alternatyvos fragmentas. Taip pat tikrinama, ar naudojamas teisingas atsakymo būsenos kodas.
- *InitialScenario* – tikrina scenarijų, kai servisas nekviečia kitų servisų.
- *MultipleServices_ServiceWasCalledWithBody* – viename kvietime yra užklausos turinys. Patikrinama, ar užklausų turiniai korektiškai generuojami.
- *MultipleServices_ServiceReturnedResponse* – tikrinama, ar teisingai generuojami žingsniai, nusakantys, kokius atsakymus grąžina servisi.
- *OptScenario* – tikrinama, ar sąlygos fragmentas egzistuoja viename iš sugeneruotų scenarijų.
- *SingleService_Single_Request_WithBody* – tikrinama, ar teisingai sugeneruojami žingsniai su užklausos turiniu.
- *SingleService_Single_Request_WithParameters* – tikrinama, ar teisingai sugeneruojami žingsniai, kurie kreipiasi į testuojamą servisą.
- *SingleService_Single_Request_WithResponseBody* – tikrinama, ar teisingai sugeneruojami žingsniai, tikrinantys serviso atsakymą.
- *AllStatusCodes-AggregatorService* – tikrina, ar sekų diagramose galima naudoti 200-299, 300-399, 400-499, 500-599 statuso kodus.
- *AllHttpMethodTypes-AggregatorService* – tikrina, ar galima naudoti *GET*, *POST*, *PUT*, *PATCH* HTTP metodų tipus.

5. Eksperimentinis testų generavimo iš sekų diagramų įrankio tyrimas

5.1. Eksperimento planas

Eksperimento planą sudarė dvi pagrindinės veiklos:

- Siūlomos metodikos pritaikymas realiems mikroservisų projektams, generuojant testus iš sekų diagramų. Šia dalimi siekta įvertinti, ar iš *UML* sekų diagramų sugeneruoti testai gali būti praktiškai taikomi mikroservisų sistemų elgsenai tikrinti, kai tarp mikroservisų atliekamos *HTTP* užklauskos. Šiai plano daliai numatytos veiklos:
 - atrinkti atviro kodo mikroservisų projektus, tinkamus metodikos taikymui;
 - naudojant dirbtinio intelekto užklausa, kiekvienam pasirinktam galutiniam taškui sugeneruoti *Mermaid* formato sekų diagramas ir perkelti jas į *Magic Systems of Systems Architect* įrankį;
 - naudojant sukurtą testų generatorių, iš sekų diagramų sugeneruoti *Gherkin* formato testų scenarijus;
 - mikroserviso projekte sukonfigūruoti bendrųjų žingsnių biblioteką;
 - vykdyti sugeneruotus testus ir apskaičiuoti kodo padengimą;
 - atlikti mutacinį testavimą naudojant *Stryker.NET* įrankį ir įvertinti aptiktų defektų skaičių [56].
- Apklauskos vykdymas, siekiant nustatyti, ar testų generavimas iš *UML* diagramų gali palengvinti testavimo procesą. Apklauskos metu respondentams buvo pateiktas metodiką demonstruojantis vaizdo įrašas ir surinktas grįžtamasis ryšys apie siūlomos metodikos naudingumą. Šiai plano daliai numatytos veiklos:
 - sudaryti apklausos klausimyną, apimančią respondentų patirties įvertinimą ir metodikos vertinimą;
 - parengti metodiką demonstruojantį vaizdo įrašą;
 - paskelbti apklausą ir surinkti respondentų atsakymus;
 - apibendrinti apklausos rezultatus ir padaryti išvadas apie siūlomos metodikos naudingumą.

5.2. Testų generavimo mikroservisų projektams eksperimentas

Šioje eksperimento dalyje buvo siekta įvertinti siūlomo metodo pritaikomumą realiems mikroservisų projektams. Todėl buvo atrinktos atvirojo kodo saugyklos, kuriose sistemų logika grindžiama kelių mikroservisų tarpusavio sąveika. Sukurtas prototipas generuoja *Gherkin* formato žingsnius, o bendrųjų žingsnių biblioteka yra realizuota naudojant *.NET* karkasą. Dėl šios priežasties eksperimentui buvo atrinkti projektai, naudojantys bent *.NET* 7 karkasą, kad būtų galima tiesiogiai pritaikyti bendrą žingsnių biblioteką. Iš viso eksperimente buvo panaudotos penkios unikalios kodo saugyklos, kiekvienoje iš jų testuojant po vieną pasirinktą mikroservisą. Atrinkti projektai atspindi skirtingus mikroservisų architektūros projektavimo šablonus, todėl leidžia įvertinti metodo taikomumą skirtinguose kontekstuose.

Testų generavimo eksperimento eiga buvo tokia:

1. Atsisiunčiamas pasirinktas mikroserviso projektas.

2. Naudojant *Claude Sonnet 4.6* modelį generuojama *mermaid* formato sekų diagrama (žr. 2 priedą).
3. Pagal sugeneruotą sekų diagramą sukuriama diagrama *Magic Systems of Systems Architect* įrankyje.
4. Mikroserviso projekte sukonfigūruojama bendrųjų žingsnių biblioteka.
5. Naudojant testų generatorių sugeneruojami *Gherkin* scenarijai.
6. Remiantis sugeneruotais testais apskaičiuojami kodo padengimo ir mutacinio testavimo rezultatai.

Toliau pateikiama eksperimentui atrinktų kodo saugyklų analizė, trumpai apibūdinant jų architektūrinės savybes, pasirinktą mikroservisą ir tinkamumą testų generavimui.

Projektas *aws-microservices-dotnet-test-samples* [57] pateikia pavyzdinę *.NET* pagrindu sukurtą mikroservisų sistemą, skirtą įvairioms testavimo strategijoms demonstruoti debesų kompiuterijos kontekste. Sistema orientuota į elektroninės prekybos sritį ir apima tokius procesus kaip prekių valdymas, pirkinių krepšelio operacijos bei užsakymų apdorojimas. Eksperimentui pasirinktas *ShoppingCart* mikroservisas, nes jis sąveikauja su kitais mikroservisais vykdydamas su pirkinių krepšeliu susijusius veiksmus, todėl yra tinkamas mikroservisų sąveikos testavimui.

Projektas *dotnetcore-microservices-poc* [58] pateikia mikroservisų architektūros pavyzdį, modeliuojantį supaprastintą draudimo domeną, kuriame vykdomi procesai nuo pasiūlymų generavimo iki draudimo polisų sukūrimo. Analizei pasirinktas *PolicyService*, kuris apdoroja draudimo pasiūlymus, konvertuoja juos į polisus ir sąveikauja su kitais sistemos komponentais, pavyzdžiui, kainodaros servais. Šis servisas pasirinktas todėl, kad turi sudėtingus domeno kontraktus, todėl leidžia įvertinti, kaip metodika susidoroja su sudėtingesnėmis objektų struktūromis ir jų serializacija.

Projektas *AspnetMicroservices* [59] pateikia mikroservisų architektūros pavyzdį, orientuotą į elektroninės prekybos domeną. Šio projekto išskirtinumas yra agregatoriaus projektavimo šablono taikymas. Agregatoriaus servisas sujungia kelių mikroservisų atsakymus į vieną klientui pateikiamą rezultatą, todėl šis projektas tinkamas vertinti testų generavimą agreguotai duomenų surinkimo logikai.

Projektas *dotnet-microservices* [60] pateikia mikroservisų architektūros pavyzdį finansų srityje, kur skirtingi servais atsakingi už klientų, sąskaitų ir transakcijų valdymą. Sistemoje naudojama su *Ocelot* realizuota *API* vartų (*angl. API Gateway*) architektūra, kuri veikia kaip vienas įėjimo taškas į sistemą ir nukreipia užklausas į atitinkamus mikroservisus. Šis projektas į eksperimentą įtrauktas tam, kad būtų galima įvertinti metodikos veikimą, kai užklausos deleguojamos kitiems mikroservisams.

Projektas *microservice-patterns* [61] pateikia mikroservisų architektūros pavyzdžių rinkinį, kuriame demonstruojami pagrindiniai paskirstytų sistemų projektavimo šablonai, tokie kaip *Saga*, *CQRS* ir *Event Sourcing*. Analizei pasirinkta *Saga* šablono orkestravimo realizacija, kurioje centralizuotas orkestratorius valdo kelių mikroservisų veiksmų seką ir koordinuoja jų vykdymą, įskaitant kompensacinius veiksmus nesėkmės atveju. Toks modelis ypač tinkamas *UML* sekų diagramų analizei ir testų generavimo uždaviniams, nes leidžia tiesiogiai susieti sekų diagramos veiksmus su vykdomų operacijų grandine ir atsakų logika.

5.3. Testų generavimo mikroservisų projektams eksperimento rezultatai

Iš analizuotų projektų atrinkta po 1–3 galutinius taškus (*angl. endpoint*), kuriems taikytas testų generavimas. Kadangi naudotuose projektuose sekų diagramos nebuvo pateiktos, jos buvo sugeneruotos *Mermaid* formatu pasitelkiant *GitHub Copilot* įrankį ir *Claude Sonnet 4.6* modelį. Vėliau šios diagramos buvo perkeltos į *Magic Systems of Systems Architect* įrankį, laikantis siūlomos metodikos reikalavimų (žr. 3 priedą). 5.1 lentelėje pateikiama testuojamų galutinių taškų statistika: nurodomas galutinio taško tipas, užklausa ir atsakymo kūnų buvimas, nuo kiek mikroservisų priklauso galutinio taško logika, logikos išsišakojimų skaičius ir numatyti atsakymo statuso kodai.

5.1 lentelės duomenys rodo, kad analizuoti galutiniai taškai skiriasi ne tik techninėmis savybėmis, bet ir elgsenos sudėtingumu. Pavyzdžiui, E10 galutinis taškas išsiskiria didžiausiu logikos išsišakojimų skaičiumi ir kelių priklausomų mikroservisų orkestravimu, todėl jis yra vienas sudėtingiausių eksperimento atvejų. Tuo tarpu E8 ir E9 galutiniai taškai atlieka užklauskų nukreipimo funkciją ir nekeičia nei užklauskos, nei atsakymo turinio, todėl jų testavimas labiau orientuotas į maršrutizavimo teisingumą. E7 turi tiek pat priklausomų servisų kaip ir E10, tačiau jo logika paprastesnė, nes jis skirtas duomenų agregavimui. Galutiniai taškai E1–E6 orientuoti į esybių kūrimo ir gavimo operacijas, todėl juose papildomai svarbus užklauskų tikrinimas.

5.1 lentelė. Testuojami galutiniai taškai (*angl. endpoint*)

Nr.	Projektas	Pavadinimas	Galutinio taško tipas	Turi užklauskos turinį	Turi atsakymo turinį	Priklausomi servais	Išsišakojimai	Gražinami statuso kodai
E1.	ShoppingCart Service	checkout/{id}	POST	Ne	Ne	2	4	200, 400
E2.	ShoppingCart Service	api/shoppingcart/{id}	GET	Ne	Taip	1	3	200, 404
E3.	ShoppingCart Service	api/shoppingcart	POST	Taip	Taip	1	3	201
E4.	PolicyService	api/policy	POST	Taip	Ne	2	3	200
E5.	PolicyService	api/offer	POST	Taip	Ne	2	3	200
E6.	PolicyService	api/policy/{id}	GET	Ne	Taip	1	3	200
E7.	ShoppingAggregator	api/v1/shopping/{id}	GET	Ne	Taip	3	2	200
E8.	OcelotGateway	accounts/{id}	GET	Ne	Taip	1	1	200, 201
E9.	OcelotGateway	accounts/adding	POST	Taip	Ne	1	1	200, 201

Nr.	Projektas	Pavadinimas	Galutinio taško tipas	Turi užklausos turinį	Turi atsakymo turinį	Priklausomi servais	Išsišaukimai	Gražinami statuso kodai
E10.	TripPlanning Service	api/saga/v1/trips	POST	Taip	Ne	3	5	200

Siekiant įvertinti sekų diagramų transformavimą į testų scenarijus, buvo atlikta eksperimentinė analizė, kurios metu sugeneruoti testai iš dešimties sekų diagramų. Rezultatai apibendrinti 5.2 lentelėje. Joje pateikiamos pagrindinės analizuotų sekų diagramų charakteristikos: išorinių servisų iškvietimų skaičius, gyvavimo linijų skaičius, egzempliorių specifikacijos kiekis, taip pat alternatyvos fragmentų skaičius. Taip pat nurodomas bendras sugeneruotų testų scenarijų skaičius bei panaudoti *Gherkin* žingsniai, kurie apibrėžti 2.1 lentelėje.

Iš 10 sekų diagramų buvo sugeneruoti 23 testų scenarijai. Sekų diagramos, turinčios alternatyvos fragmentus, generuoja daugiau nei vieną scenarijų, nes kiekvienas alternatyvus vykdymo kelias yra transformuojamas į atskirą testavimo scenarijų (pvz., S1, S4, S6, S8, S9, S10). Tuo tarpu diagramos be alternatyvos fragmentų generuoja vieną scenarijų (pvz., S3, S5, S7). Taip pat matoma reikšminga variacija egzempliorių specifikacijų kiekyje. Pavyzdžiui, S7 ir S10 diagramos pasižymi dideliu šių specifikacijų skaičiumi (atitinkamai 11 ir 10), todėl jose dažniau naudojami žingsniai, susiję su atsako ar užklausos duomenų tikrinimu (pvz., SD4, SD7, SD9). Tuo tarpu diagramos su mažesniu egzempliorių kiekiu dažniau apsiriboja užklausos ir atsakymo tikrinimo žingsniais, kurie neturi užklausos ar atsakymo kūno tikrinimo. Analizuojant žingsnių panaudojimą, galima pastebėti, kad dažniausiai naudojami SD5 ir SD6 žingsniai, kurie pasitaiko visuose sugeneruotuose scenarijuose. Tai rodo, kad šie žingsniai yra esminiai kiekvieno testavimo scenarijaus komponentai. Tuo tarpu kiti žingsniai (pvz., SD7, SD8, SD9) naudojami priklausomai nuo konkrečių sekų diagramų struktūrinių ypatybių, tokių kaip duomenų sudėtingumas ar išorinių servisų sąveikų kiekis.

5.2 lentelė. Statistika apie sekų diagramas

Sekų diagramos parametrai					Testų scenarijų generavimo rezultatai	
Nr.	Išorinių servisų iškvietimai	Gyvavimo linijos	Egzempliorių specifikacijos	Alt fragmentai	Iš viso sugeneruota scenarijų	Naudoti žingsniai (2.1 lentelė)
S1.	6	4	4	4	4	SD1(4), SD3(3), SD4(3), SD5(4), SD6(4), SD8(4), SD9(2)
S2.	2	3	1	2	2	SD1(2), SD3(1), SD4(1), SD5(2), SD6(2), SD7(1), SD8(2)
S3.	1	3	2	0	1	SD2(1), SD4(1), SD5(1), SD6(1), SD7(1), SD9(1)

Sekų diagramos parametrai					Testų scenarijų generavimo rezultatai	
Nr.	Išorinių servisų iškvietimai	Gyvavimo linijos	Egzempliorių specifikacijos	Alt fragmentai	Iš viso sugeneruota scenarijų	Naudoti žingsniai (2.1 lentelė)
S4.	3	3	7	3	3	SD2(3), SD3(2), SD4(2), SD5(3), SD6(3), SD8(3), SD9(2)
S5.	2	4	6	0	1	SD2(1), SD3(1), SD4(1), SD5(1), SD6(1), SD9(2)
S6.	1	3	4	3	3	SD1(3), SD3(1), SD4(2), SD5(3), SD6(3), SD7(1), SD8(3)
S7.	4	5	11	0	1	SD1(1), SD4(4), SD5(1), SD6(1), SD7(1), SD8(4)
S8.	1	3	1	3	3	SD1(3), SD3(2), SD4(1), SD5(3), SD6(3), SD7(1), SD8(3)
S9.	1	3	1	3	3	SD2(3), SD3(3), SD5(3), SD6(3), SD9(3)
S10.	5	5	10	4	3	SD2(3), SD4(9), SD5(3), SD6(3), SD9(9)

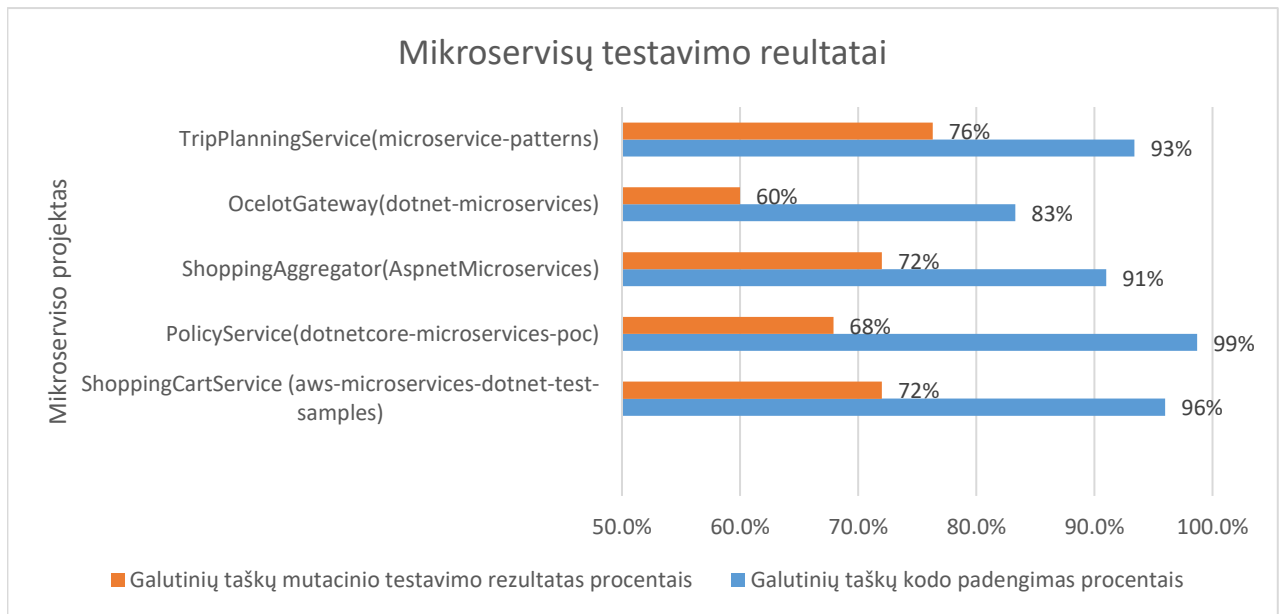
Sugeneruoti testų scenarijai buvo pritaikyti atrinktuose projektuose, kiekvienam jų sukuriant atskirą testų projektą vadovaujantis 4.2.2 skyriuje apibrėžtomis instrukcijomis. Prieš pritaikant metodiką analizuojamuose projektuose, buvo atliktos tam tikros modifikacijos, siekiant užtikrinti jos suderinamumą su skirtingomis mikroservisų sistemų architektūromis. Viena iš pagrindinių modifikacijų buvo galimybė konfigūruoti servisų *URL* adresus, taip sudarant sąlygas testus vykdyti skirtingose aplinkose. Taip pat dalyje projektų tiesioginės duomenų bazės priklausomybės buvo pakeistos mikroservisų sąveika, siekiant geriau atitikti mikroservisų architektūros principus ir užtikrinti testavimo metodo pritaikomumą paskirstytose sistemose. Tokiu būdu buvo siekiama praktiškai įvertinti siūlomo metodo efektyvumą realių mikroservisų sistemų kontekste. 5.3 lentelėje pateikiama galutinių taškų testavimo statistika, gauta naudojant sugeneruotus testus. Lentelėje nurodomas sugeneruotų testų skaičius, padengiamų kodo eilučių kiekis, bendras kodo padengimas procentais, taip pat mutacinio testavimo metu aptiktų ir neaptiktų defektų skaičius. Kodo padengimas buvo matuojamas naudojant integruotus *.NET* įrankius, o mutacinis testavimas atliktas naudojant *Stryker.NET* įrankį. Rezultatai rodo, kad visi analizuoti galutiniai taškai pasiekė aukštą kodo padengimo lygį - daugiau nei 90%, o daugeliu atvejų padengimas siekė 100 % (pvz., E2–E5). Mažiausias padengimas užfiksuotas E7 (91 %) ir E10 (93,4 %) galutiniuose taškuose, tačiau jis vis tiek išlieka aukštas ir leidžia teigti, kad sugeneruoti testai apima didžiąją dalį vykdymo kelių. Kiekvienam sukurtam testų projektui mutacinis testavimas buvo vykdomas atskirai, paleidžiant *Stryker.NET* komandą projekto aplinkoje.

Įrankis kiekvienam mikroservisui sugeneravo mutantus ir automatiškai vykdė susietus testus, o rezultatai buvo kaupiami atskirai kiekvienam galutiniam taškui. Mutacinio testavimo rezultatai rodo, kad sugeneruoti testai geba aptikti dalį dirbtinai įterptų defektų, tačiau ne visi mutantai yra identifikuojami. *OcelotGateway* projekto atveju kodo padengimo ir mutacinio testavimo įvertinti nebuvo galima. Šiame projekte galutiniai taškai nėra realizuoti programiniu kodu, nes užklausų maršrutizavimas aprašomas konfigūracijos failuose. Dėl šios priežasties tokio tipo testavimo metrikos šiam projektui netaikomos.

5.3 lentelė. Sugeneruotų testų statistika

Projektas	Nr.	Pavadinimas	Testų skaičius	Padengiamų eilučių skaičius	Kodo padengimas testais, %	Aptikti defektai	Neaptikti defektai
ShoppingCartService	E1.	checkout/{id}	4	42	94,5	9	3
	E2.	api/shoppingcart/{id}	2	10	100	6	3
	E3.	api/shoppingcart	1	12	100	6	2
PolicyService	E4.	api/policy	3	37	100	7	2
	E5.	api/offer	1	24	100	7	4
	E6.	api/policy/{id}	3	27	96,2	5	3
ShoppingAggregator	E7.	api/v1/shopping/{id}	1	101	91,0	13	5
OcelotGateway	E8.	accounts/{id}	3	-	-	-	-
	E9.	accounts/adding	3	-	-	-	-
TripPlanningService	E10.	api/saga/v1/trips	3	123	93,4	42	15

5.1 pav. pateikiami apibendrinti testavimo rezultatai projektų lygmeniu. Jie leidžia palyginti, kaip metodika veikia skirtinguose architektūriniuose kontekstuose. *OcelotGateway* mikroserviso atveju pateikiama viso projekto, o ne tik galutinių taškų kodo padengimas ir mutacinio testavimo rezultatai.



5.1 pav. Mikroservisų testavimo rezultatai

Eksperimento rezultatai parodė, kad sugeneruoti testai gali aptikti defektus. Jie leidžia nustatyti pašalintus servisų iškvietimus, pakeistus galutinių taškų adresus, modifikuotas sąlygines išraiškas, pakeistas grąžinamų statuso kodų reikšmes ir kitus pakeitimus, darančius įtaką galutiniam atsakymui. Tačiau yra defektų, kurie sugeneruoti testai negali aptikti. Visų pirma, jie nėra tinkami identifikuoti defektų, susijusių su pranešimų registravimo (*angl. logging*) logika, nes ši elgsena nėra reprezentuojama sekų diagramose. Taip pat testai neaptinka defektų, susijusių su vidinėmis validacijomis, kurios nėra aiškiai išreikštos sekų diagramoje. Pavyzdžiui, jei paslauga papildomai tikrina kitų paslaugų grąžinamų atsakymų korektiškumą, tačiau ši logika nėra modeliuota sekų diagramoje, tokie pakeitimai lieka neaptikti. Be to, sugeneruoti testai negali patikrinti įvykių publikavimo (*angl. event publishing*) mechanizmus, kadangi šie aspektai dažnai nėra detalizuojami sekų diagramose. Kadangi sekų diagramose nėra apibrėžiama vykdymo laiko būseną (pvz., dabartinė data ar laikas), sugeneruoti testai negali patikimai aptikti defektų, susijusių su laiko sąlygomis ar terminais. Tai riboja galimybę testuoti ribinius atvejus (*angl. edge case*), tokius kaip galiojimo laikų pasibaigimas ar laikui jautrūs sprendimai.

Pasiektas vidutinis 92,6 % kodo padengimas ir 69,8 % mutacinio testavimo rezultatas rodo, kad siūlomas metodas yra efektyvus aptinkant daugumą su išorine sistemos elgsena susijusių klaidų, tačiau jo taikymas turėtų būti derinamas su kitais testavimo metodais (pvz., vienetinais testais), siekiant užtikrinti visapusišką sistemos kokybės vertinimą.

5.4. Mikroservisų testų generavimo metodikos įvertinimo apklausa

Apklausoje metu siekiama įvertinti, ar siūloma testų generavimo metodika galėtų palengvinti mikroservisų testavimą. Tam buvo sukurta apklausa, sudaryta iš septynių klausimų, kurie pateikiami 5.4 lentelėje. Klausimai K1–K4 skirti įvertinti respondentų programavimo patirtį, darbo su mikroservisų architektūra bei *UML* diagramomis taikymo patirtį. Apklausoje taip pat pateikiamas vaizdo įrašas, kuriame demonstruojama testų generavimo metodika. Jame pateikiamas įvadas į *Gherkin* formato scenarijus, parodoma pavyzdinė mikroservisų sekų

diagrama, pagal kurią sugeneruojami testai, bei demonstruojamas klaidų aptikimo procesas. Po vaizdo įrašo pateikiami papildomi klausimai, skirti įvertinti siūlomą metodiką. Klausimai K5–K7 yra skirti įvertinti respondentų nuomonę apie siūlomą testų generavimo metodiką ir jos praktinę naudą. K5 klausimu siekiama nustatyti, ar metodika palengvina mikroservisų testavimo procesą, vertinant bendrą jos naudingumą. K6 klausimas orientuotas į sugeneruotų testų kokybės ir jų taikymo efektyvumo vertinimą. K7 klausimu siekiama įvertinti metodikos praktinio pritaikymo galimybes realiuose projektuose, kuriuose sistemos elgsena modeliuojama *UML* sekų diagramomis. K8 klausimas yra atviro tipo ir skirtas papildomoms respondentų įžvalgoms bei pastaboms surinkti.

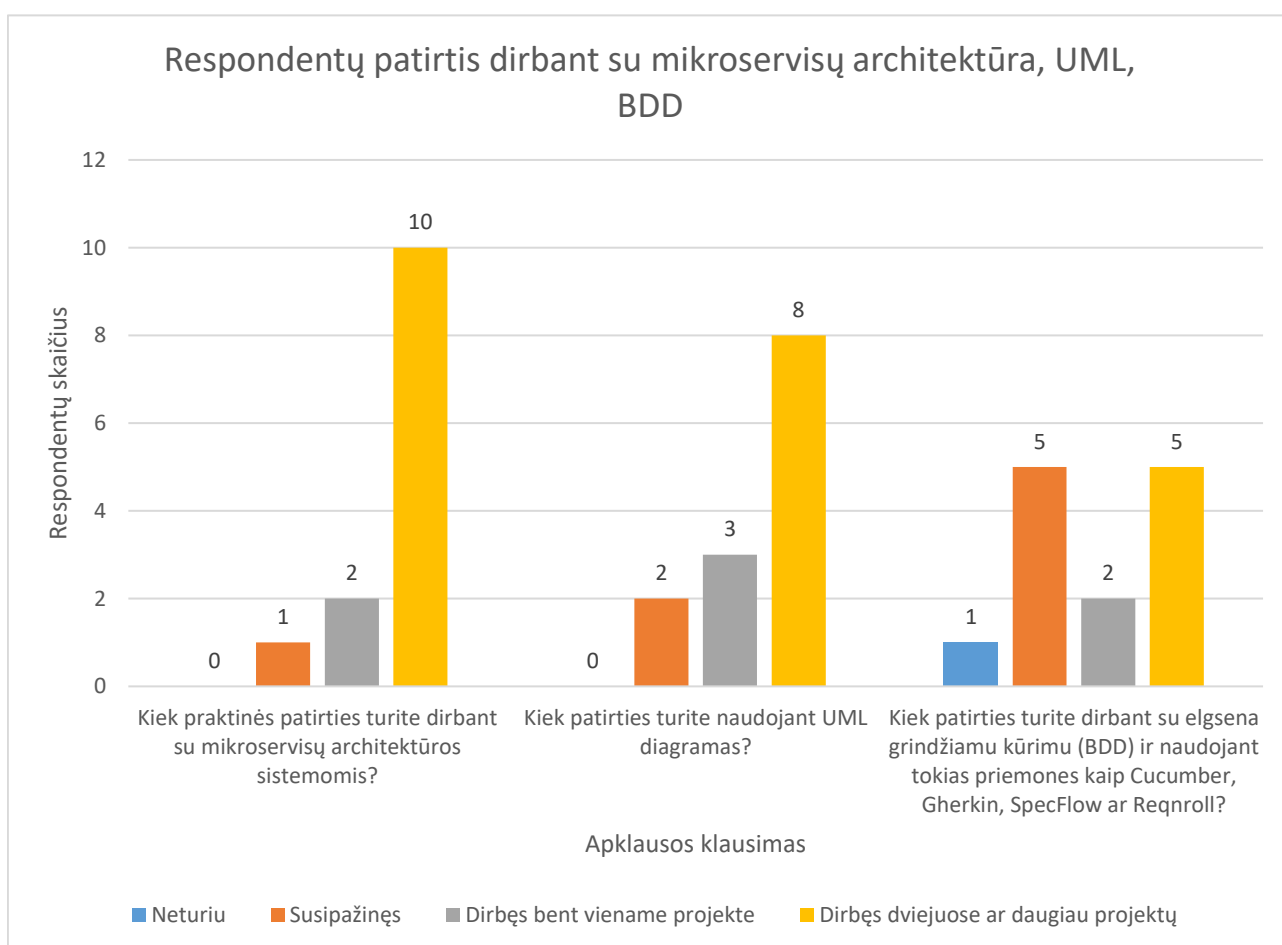
5.4 lentelė. Apklauso klausimai

Nr.	Klausimas	Klausimo tipas	Galimi atsakymo variantai
K1.	Kokia yra Jūsų programinės įrangos kūrimo arba testavimo patirtis metais?	Vieno pasirinkimo	0-2,3-5,6+
K2.	Kiek praktinės patirties turite dirbant su mikroservisų architektūros sistemomis?	Vieno pasirinkimo	Neturiu, esu susipažinęs, esu dalyvavęs kuriant bent vieną sistemą, esu kūręs dvi ar daugiau sistemų
K3.	Kiek patirties turite naudojant <i>UML</i> diagramas?	Vieno pasirinkimo	Neturiu, esu susipažinęs, esu naudojęs kuriant bent vieną sistemą, esu naudojęs kuriant dvi ar daugiau sistemų
K4.	Kiek patirties turite dirbant su elgsena grindžiamu kūrimu (<i>BDD</i>) ir naudojant tokias priemones kaip <i>Cucumber</i> , <i>Gherkin</i> , <i>SpecFlow</i> ar <i>Reqnroll</i> ?	Vieno pasirinkimo	Neturiu, esu susipažinęs, esu taikęs bent vienoje sistemoje, esu taikęs dviejuose ar daugiau sistemų
K5.	Pristatyta testų generavimo iš sekų diagramų metodika palengvintų mikroservisų testavimą.	Likerto skalė	Visiškai nesutinku, nesutinku, Nei sutinku, nei nesutinku, Sutinku, Visiškai sutinku
K6.	Pristatytos metodikos sugeneruoti testai pasirodė naudingi sistemos elgsenai ir mikroservisų tarpusavio sąveikai tikrinti.	Likerto skalė	Visiškai nesutinku, nesutinku, Nei sutinku, nei nesutinku, Sutinku, Visiškai sutinku
K7.	Jei dirbčiau projekte, kuriame sistemos sąveika aprašoma <i>UML</i> sekų diagramomis, taikyčiau šią metodiką testams generuoti.	Likerto skalė	Visiškai nesutinku, nesutinku, Nei sutinku, nei nesutinku, Sutinku, Visiškai sutinku
K8.	Kokias papildomas įžvalgas norėtumėte pateikti apie šią metodiką?	Atviras klausimas	

5.5. Apklauso rezultatai

Apklausa buvo paskelbta balandžio 20 dieną, atsakymai buvo renkami iki balandžio 29 dienos. Kadangi metodika skirta mikroservisų architektūra grindžiamų sistemų testavimui, apklausai buvo tikslingai atrinkti tik šioje srityje dirbantys specialistai – jų nuomonė laikoma aktualiausia vertinant siūlomos metodikos praktinę naudą. Respondentai buvo surinkti iš Lietuvoje veikiančių technologijų įmonių – *Visma Tech*, *PVcase*, *Nord Security* ir *Flo Health*.

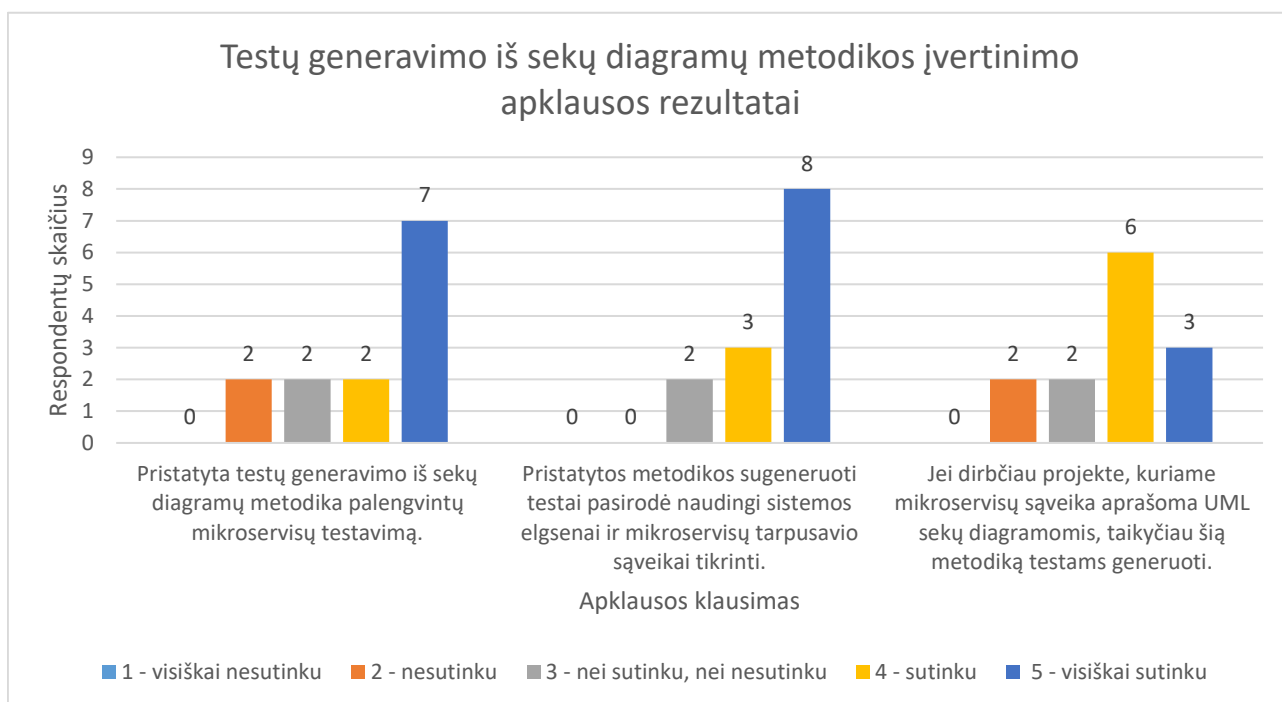
Iš viso buvo surinkta 13 atsakymų (žr. 4 priedą). 5.2 pav. pateiktoje diagramoje pateikiama respondentų patirtis dirbant su mikroservisų architektūra, *UML* diagramomis ir elgsena grindžiamu kūrimu (*angl. behaviour driven development*). Matyti, kad 12 apklaustųjų turi praktinės patirties mikroservisų srityje. 10 respondentų nurodė, kad yra dirbę su daugiau nei viena sistema, o tik nedidelė dalis pažymėjo neturintys patirties. Analizuojant *UML* diagramų naudojimą matoma, kad du respondentai yra tik susipažinę su *UML*, o kiti 11 respondentų turi praktinės taikymo patirties viename ar keliuose projektuose. *BDD* praktikos atveju pastebima, kad daugiau respondentų turi ribotą arba vidutinę patirtį – penki respondentai yra tik susipažinę, o likę septyni yra taikę šį metodą realiuose projektuose. Apibendrintai galima teigti, kad didžioji dalis respondentų turėjo patirties su mikroservisų architektūra, *UML* diagramomis ir elgsena grindžiamu kūrimu, todėl jų nuomonė apie siūlomą metodiką yra ypač reikšminga, nes ji grindžiama praktine patirtimi ir tiesioginiu susidūrimu su šių technologijų taikymu realiuose projektuose.



5.2 pav. Respondentų patirtis dirbant su mikroservisų architektūra, *UML*, *BDD*.

5.3 pav. pateikiami respondentų atsakymai į klausimus, susijusius su testų generavimo iš sekų diagramų metodikos vertinimu (K5–K7). Vertinimas atliktas naudojant Likerto skalę nuo 1 iki 5. Rezultatai rodo, kad dauguma respondentų linkę vertinti metodiką teigiamai. K5 klausimo atveju devyni respondentai sutiko, kad metodika palengvintų testavimą, o tik du respondentai nesutiko su tuo. Panaši tendencija matoma ir K6 klausime, kuriame vertinamas sugeneruotų testų naudingumas sistemos elgsenai ir mikroservisų sąveikai tikrinti. 11 respondentų pasirinko teigiamus įverčius, o du respondentai liko neutralūs. K7

klausimo rezultatai parodo, jog devyni respondentai taikytų siūlomą metodiką testams generuoti.



5.3 pav. Testų generavimo iš sekų diagramų metodikos įvertinimo apklausos rezultatai

Iš 13 respondentų 69,2 % sutiko, kad testų generavimo iš sekų diagramų metodika palengvintų mikroservisų testavimą, 84,6 % nurodė, kad sugeneruoti testai yra naudingi, o 69,2 % nurodė, kad tokią metodiką taikytų praktiškai. Atsižvelgiant į ribotą imties dydį, kurį sudaro 13 respondentų, šie rezultatai turėtų būti interpretuojami kaip orientaciniai. Vis dėlto jie leidžia daryti prielaidą, kad siūloma metodika yra vertinama palankiai ir laikoma naudinga praktikoje.

5.6. Testų generavimo iš sekų diagramų įrankio veikimo ir savybių analizė, kokybės kriterijų įvertinimas

Sukurta metodika geriausiai tinka tada, kai testuojamas mikroservisas sąveikauja su bent vienu kitu mikroservisu *REST* stiliaus sinchroniniais *HTTP* iškvietimais, o sąveika yra pakankamai detalai aprašoma *UML* sekų diagramose. Eksperimento metu testų generavimas pasirinktiems projektams vyko sėkmingai.

Tačiau metodikos taikymas yra apribotas architektūriniais ir modeliavimo aspektais. Jei servisas neturi priklausomų *HTTP* servisų, pavyzdžiui, kreipiasi tik į duomenų bazę ar neturi jokios komunikacijos tarpusavyje, tuomet sekų diagramoje neatsiranda žinučių mainų, kuriuos būtų galima transformuoti į testų scenarijus. Tokiu atveju metodika netaikytina ir servisas turėtų būti dengiamas vienetų arba kitais testais. Servisams, kurių logika realizuota konfigūracijos failuose, o ne programiniame kode, kaip *OcelotGateway* atveju, metodika vis tiek leidžia sugeneruoti ir paleisti scenarijus bei patikrinti, ar maršrutizavimas veikia teisingai.

Metodikos apimtį riboja ir tai, kad ji apsiriboja tik sinchronine *REST* komunikacija. Servisai, kurie sąveikauja per žinučių tarpininkus (pvz., *RabbitMQ*, *Kafka*), naudoja asinchroninius

pranešimus ar įvykiais paremtą (*angl. event-driven*) architektūrą, negali būti pilnai aprėpti šia metodika. Tokiuose kontekstuose sekų diagramomis galima aprašyti tik sinchroninę komunikacijos dalį, o likusi elgsena lieka nepatestuota. Be to, dabartinė versija nepalaiko *HTTP* užklausos antraščių (*angl. headers*) ir *URL* užklausos parametrų (*angl. query parameters*) – jei galutinio taško elgsena priklauso nuo šių elementų, sugeneruoti testai jų netikrina.

Yra ir modeliavimo lygio apribojimų, susijusių su *UML* egzempliorių specifikacijų (*angl. instance specification*) savybėmis. Kadangi egzempliorių specifikacija modeliuoja vieną objekto egzempliorių, *JSON* masyvų gražinimas atsakyme negali būti teisingai perteiktas sekų diagramoje. Norint modeliuoti kolekciją, reikia naudoti apgaubiančiojo objekto šabloną, pavyzdžiui, „{“items”: [{}, {}]}“. Analogiškai, sekų diagramose nėra galimybės aprašyti vykdymo metu generuojamų reikšmių, tokių kaip unikalūs identifikatoriai ar laiko žymės – sugeneruoti testai tokiais atvejais gali tikrinti tik atsakymo statuso kodą arba lauko buvimą, bet ne jo konkretų turinį.

Eksperimentinio tyrimo apribojimai taip pat turi įtakos gautiems rezultatams. Visi penki eksperimente naudoti projektai yra *.NET* pagrindu sukurtos atviro kodo sistemos, todėl negalima teigti, kad rezultatai tiesiogiai perkeltami į kitas technologines platformas (pvz., *Java Spring Boot*, *Node.js*). Sekų diagramos visoms sistemoms buvo kuriamos naudojant *Magic Systems of Systems Architect* įrankį, eksportuojantį į *XMI* formatą. Metodikos suderinamumas su kitų *UML* įrankių eksporto formatais liko nepatikrintas.

Nepaisant minėtų apribojimų, metodiką galima sėkmingai taikyti tada, kai projektas naudoja mikroservisų architektūrą su *REST* stiliaus sinchronine komunikacija ir sistemos sąveika yra aprašyta *UML* sekų diagramose. Tokiomis sąlygomis metodika leidžia automatiškai generuoti elgsenos testus, padengiančius pagrindinius vykdymo kelius. Metodika gali būti naudinga sistemose, kuriose yra keletas mikroservisų su tarpusavio sinchronine sąveika ir kuriose kiekvienas servisas testuojamas izoliuotai – tokiais atvejais sugeneruoti testai gali papildyti esamą testavimo procesą.

5.7. Testų generavimo iš sekų diagramų metodikos taikymo rekomendacijos

Eksperimento metu sekų diagramos buvo kuriamos naudojant *Magic Systems of Systems Architect* įrankį. Kadangi *UML 2.5* standartas nurodo, jog žinutės gali turėti argumentus laisva forma, skirtingi įrankiai šių argumentų formatą gali saugoti skirtingais būdais. Sukurtas prototipas remiasi prielaida, kad užklausų ir atsakymų elementų identifikatoriai argumentuose nurodomi *HTML* formatu, kaip pavaizduota 5.4 pav. *Magic Systems of Systems Architect* įrankio sugeneruotame pavyzdyje.

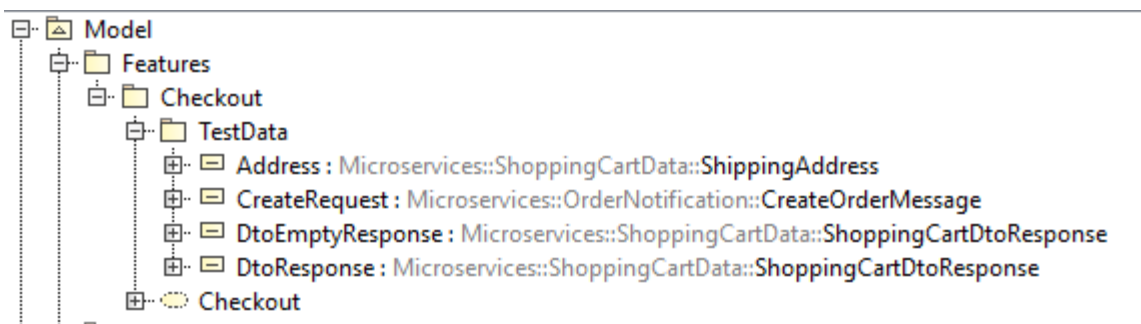
```
<message xmi:type="uml:Message" xmi:id="_2022x_2_5e10205_1764650776215_352034_2221" visibility="public" receiveEvent="_2022x_2_5e10205_1764650776216_708547_2223" sendEvent="_2022x_2_5e10205_1764650776216_469290_2222" signature="_2022x_2_5e10205_1764650267104_567301_2089">
  <argument xmi:type="uml:OpaqueExpression" xmi:id="_2022x_2_5e10205_1764650779412_204696_2230">
    <body>&lt;html light="true"&gt;&lt;body&gt;&lt;a href="mdl://_2022x_2_5e10205_1764651040909_347072_2305" erit:display="NAME" erit:update="AUTOMATIC_UPDATE"
    <language>English</language>
  </argument>
</message>
```

5.4 pav. Užklausos argumento modeliavimas

Modeliuojant alternatyvos ir sąlygos fragmentus būtina užtikrinti teisingą jų struktūrą ir hierarchiją. Nors diagramose gali susidaryti įspūdis, kad šie fragmentai yra įdėti vienas į kitą, modelyje fragmentai gali vis dėl to būtų išdėstyti nuosekliai. Netinkamas fragmentų modeliavimas gali lemti neteisingą sugeneruotų testų scenarijų skaičių arba jų struktūrą.

Jeigu priklausomo serviso galutinio taško atsakymas priklauso nuo užklauskos kūno, nėra galimybės automatiškai generuoti kelių skirtingų užklauskų scenarijų, nes nėra apibrėžta žingsnio, kuris leistų dinamiškai gražinti atsakymą pagal siunčiamos užklauskos turinį. Atsakymo validacija tampa sudėtinga tais atvejais, kai atsakymo duomenys yra generuojami vykdymo metu, pavyzdžiui, kai sukuriamas naujas identifikatorius. Tokiais atvejais metodiką būtų galima patobulinti įtraukiant validacijos pagal pasirinktą formatą ar šabloną mechanizmą.

Metodikoje nėra griežtai apibrėžti mikroservisų klasių pavadinimų standartai. Vykdamas eksperimentą prie mikroserviso pavadinimo buvo pridėdama „Service“, kad būtų aiškiai suprantama, jog vaizduojamas mikroservisas. Kuriant egzempliorių specifikacijas, jos buvo kuriamos šalia sekų diagramos, kad testavimo duomenys būtų šalia elgseną aprašančios sekų diagramos, kaip pateikta 5.5 pav.



5.5 pav. Pavyzdinė egzempliorių specifikacijos ir sekų diagramų išdėstymo struktūra

Išvados

1. Mikroservisų architektūros ir testavimo analizė parodė, kad taikant mikroservisų architektūrą reikalinga atlikti kontraktų ir komponentų testavimą, kad būtų ištestuota komunikacija tarp mikroservisų.
2. Išanalizavus *UML* taikymą mikroservisų architektūroje pastebėta, kad iš struktūrinių diagramų naudojamos klasių ir komponentų diagramos, o elgsenai vaizduoti naudojamos sekų diagramos, kurios gali būti naudojamos testų generavimui.
3. Apžvelgus sprendimus, padedančius testuoti sistemas, pastebėta, kad yra sprendimų, kurie generuoja testus, vizualizuoja scenarijus ar testuoja komunikaciją tarp servisų. Tačiau nė vienas iš apžvelgtų sprendimų neatlieka visų 3 funkcijų kartu, todėl yra poreikis kurti įrankį, kuris padėtų generuoti testus mikroservisams iš *UML* sekų diagramų.
4. Kuriant mikroservisų testų generavimo dalykinės srities modelį, buvo pastebėta, kad testams generuoti iš *UML* modelio reikalingas tarpinis mikroservisų komunikacijos modelis, kuris palengvintų testų scenarijų informacijos surinkimą iš sekų diagramų ir testų generavimą.
5. Siekiant realizuoti testų scenarijų generavimą iš *UML* sekų ir klasių diagramose užfiksuotos informacijos, buvo nuspręsta sekų diagramų pranešimus tarp gyvavimo linijų transformuoti į *Gherkin* formato žingsnius pagal apibrėžtus šablonus, iš kurių būtų sudaromi testų scenarijai.
6. Mikroservisų testų generatoriaus prototipas buvo įgyvendintas kaip *MCP* serveris ir integruojamas su *GitHub Copilot* įskiepiu, kas suteikia galimybę *Visual Studio Code* programoje generuoti *Gherkin* formato testų scenarijus naudojant *UML* sekų diagramų modelio failus.
7. Mikroservisų testų generavimo eksperimente sugeneruoti testai pasiekė vidutinį 92,6 % kodo padengimą ir 69,8 % mutacinio testavimo rezultata, kas rodo, kad pagal darbe aprašytą metodiką sugeneruoti testai gali aptikti mikroservisų elgsenos klaidas ir būti praktiškai naudojami.
8. Atliktoje testavimo srities ekspertų apklausoje dalyvavo 13 mikroservisų testavimo specialistų, iš kurių 84,6 % nurodė, kad sugeneruoti testai yra naudingi, o 69,2 % nurodė, kad tokią metodiką taikytų praktiškai. Tai rodo, kad darbe siūloma metodika gali būti taikoma kaip priemonė mikroservisų testavimui palengvinti.

Literatūros sąrašas

1. LEE, C.; KIM, H. F. ir LEE, B. G. A Systematic Literature Review on the Strategic Shift to Cloud ERP: Leveraging Microservice Architecture and MSPs for Resilience and Agility. *Electronics*, t. 13 (2024), nr. 14, p. 2885. Prieiga per internetą: <https://www.mdpi.com/2079-9292/13/14/2885> [žiūrėta 2026 m. gegužės 18 d].
2. G. Blinowski; A. Ojdowska ir A. Przybyłek. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*, t. 10 (2022), p. 20357–20374.
3. F. Carranza-García; C. Rodríguez-Domínguez ir J. L. Garrido. Addressing Expressiveness for a UML Microservices-Based Modeling within the Life Cycle of the Ubiquitous System Development. *Iš: 2021 17th International Conference on Intelligent Environments (IE)*, pp. 1–8.2021.
4. DI FRANCESCO, P.; LAGO, P. ir MALAVOLTA, I. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, t. 150 (2019), p. 77–97. Prieiga per internetą: <https://www.sciencedirect.com/science/article/pii/S0164121219300019>.
5. OZKAYA, M. *Microservices Architecture for Enterprise Large-Scaled Application*. , -04-09, 2023. Prieiga per internetą: <https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a78a> [žiūrėta 2026 m. kovo 29 d].
6. LI, S.; ZHANG, H.; JIA, Z.; ZHONG, C.; ZHANG, C. ir kt. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology*, t. 131 (2021), p. 106449. Prieiga per internetą: <https://www.sciencedirect.com/science/article/pii/S0950584920301993>.
7. RODRÍGUEZ, C.; BAEZ, M.; DANIEL, F.; CASATI, F.; TRABUCCO, J. C. ir kt. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. , pp. 21–39. Cham: Springer International Publishing, 2016. Prieiga per internetą: https://link.springer.com/chapter/10.1007/978-3-319-38791-8_2#citeas [2026 m. kovo 29 d].
8. RICHARDSON, L. ir RUBY, S. *RESTful web services*. " O'Reilly Media, Inc.", 2008.
9. *About gRPC*. Prieiga per internetą: <https://grpc.io/about/> [žiūrėta 2026 m. kovo 29 d].
10. V. Singh; A. Singh; A. Aggarwal ir S. Aggarwal. A Digital Transformation Approach for Event Driven Micro-Services Architecture Residing within Advanced Vcs. *Iš: 2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, pp. 100–105.2021.
11. RICHARDSON, C. *Microservices Patterns: With Examples in Java*. , 2019.
12. OYENIRAN, O. C.; ADEWUSI, A. O.; ADELEKE, A. G.; AKWAWA, L. A. ir AZUBUKO, C. F. Microservices Architecture in Cloud-Native Applications: Design Patterns and Scalability. *International Journal of Advanced Research and Interdisciplinary Scientific Endeavours*, t. 1 (2024), nr. 2, p. 92–106. Prieiga per internetą: <https://ijarise.org/index.php/ijarise/article/view/66> [žiūrėta 2026 m. kovo 30 d].
13. NEELAN, A. The Evolving Role of API Gateways in Scalable Microservices Architecture. *International Journal of Emerging Trends in Computer Science and Information*

- Technology*, t. 6 (2025), nr. 4, p. 4–15. Prieiga per internetą: <https://www.ijetcsit.org/index.php/ijetcsit/article/view/399> [žiūrėta 2026 m. kovo 17 d].
14. C. -Y. Li; S. -P. Ma ir T. -W. Lu. Microservice Migration using Strangler Fig Pattern: A Case Study on the Green Button System. *Iš: 2020 International Computer Symposium (ICS)*, pp. 519–524.2020.
 15. S. Aydin ir C. B. Çebi. Comparison of Choreography Vs Orchestration Based Saga Patterns in Microservices. *Iš: 2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pp. 1–6.2022.
 16. OZKAYA, M. *Saga Pattern for Microservices Distributed Transactions*. , -04-09, 2023. Prieiga per internetą: <https://medium.com/design-microservices-architecture-with-patterns/saga-pattern-for-microservices-distributed-transactions-7e95d0613345> [žiūrėta 2026 m. kovo 29 d].
 17. GHANI, I.; WAN-KADIR, W. M. N.; MUSTAFA, A. ir BABIR, M. I. Microservice Testing Approaches: A Systematic Literature Review. *International Journal of Integrated Engineering*, t. 11 (2019), nr. 8, p. 65–80. Prieiga per internetą: <https://publisher.uthm.edu.my/ojs/index.php/ijie/article/view/3856> [žiūrėta 2026 m. kovo 30 d].
 18. NIDHRA, S. ir DONDETI, J. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, t. 2 (2012), nr. 2, p. 29–50.
 19. ELSNER, D.; BERTAGNOLLI, D.; PRETSCHNER, A. ir KLAUS, R. Challenges in Regression Test Selection for End-to-End Testing of Microservice-Based Software Systems. *Iš: Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pp. 1–5.2022.
 20. KHORIKOV, V. *Unit testing principles, practices, and patterns*. Simon and Schuster, 2020.
 21. VOCKE, H. *The Practical Test Pyramid*. Prieiga per internetą: <https://martinfowler.com/articles/practical-test-pyramid.html> [žiūrėta 2026 m. kovo 17 d].
 22. LEHVÄ, J.; MÄKITALO, N. ir MIKKONEN, T. Consumer-Driven Contract Tests for Microservices: A Case Study. *Iš: International Conference on Product-Focused Software Process Improvement*, pp. 497–512. Springer, 2019.
 23. NIÑO-MARTÍNEZ, V. M.; OCHARÁN-HERNÁNDEZ, J. O.; LIMÓN, X. ir PÉREZ-ARRIAGA, J. C. Microservices Deployment: A Systematic Mapping Study. *Iš: 2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pp. 24–33. IEEE, 2021.
 24. *What is CI/CD? Continuous Integration & Delivery Explained*. Prieiga per internetą: <https://katalon.com/resources-center/blog/ci-cd-introduction> [žiūrėta 2026 m. kovo 29 d].
 25. IRSHAD, M.; BRITTO, R. ir PETERSEN, K. Adapting Behavior Driven Development (BDD) for large-scale software systems. *Journal of Systems and Software*, t. 177 (2021), p. 110944.
 26. HELLESØY, &. Aslak ir WYNNE, M. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. 1st , 2012.
 27. *Installation | Cucumber*. [2026-05-17]. Prieiga per internetą: <https://cucumber.io/docs/installation/> [žiūrėta 2026 m. gegužės 18 d].

28. CALAMO, M.; MONTI, F.; SPAZIANI, F.; LEOTTA, F. ir MECELLA, M. From User Stories to Architectures: Using LLM-powered Agents to Design and Improve Microservice-based Software. *IEEE Software* (2026).
29. SCHÄFER, M.; NADI, S.; EGHBALI, A. ir TIP, F. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, t. 50 (2023), nr. 1, p. 85–105.
30. GU, S.; NASHID, N. ir MESBAH, A. Llm test generation via iterative hybrid program analysis. *arXiv preprint arXiv:2503.13580* (2025).
31. ATIL, B.; AYKENT, S.; CHITTAMS, A.; FU, L.; PASSONNEAU, R. J. ir kt. Non-Determinism of “Deterministic” LLM System Settings in Hosted Environments. *Iš: Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems*, pp. 135–148.2025.
32. HUANG, L.; YU, W.; MA, W.; ZHONG, W.; FENG, Z. ir kt. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, t. 43 (2025), nr. 2, p. 1–55.
33. ZHANG, B.; LIANG, P.; ZHOU, X.; AHMAD, A. ir WASEEM, M. Practices and challenges of using github copilot: An empirical study. *arXiv preprint arXiv:2303.08733* (2023).
34. *What is the Model Context Protocol (MCP)?* . Prieiga per internetą: <https://modelcontextprotocol.io/docs/getting-started/intro> [žiūrėta 2026 m. gegužės 17 d].
35. HASAN, M. M.; LI, H.; FALLAHZADEH, E.; RAJBAHADUR, G. K.; ADAMS, B. ir kt. Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers. *ACM Transactions on Software Engineering and Methodology* (2025).
36. GRANO, G. A New Dimension of Test Quality: Assessing and Generating Higher Quality Unit Test Cases. *Iš: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 419–423.2019.
37. KOCHHAR, P. S.; THUNG, F. ir LO, D. Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems. *Iš: 2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pp. 560–564. IEEE, 2015.
38. ZHANG, L.; MARINOV, D.; ZHANG, L. ir KHURSHID, S. Regression Mutation Testing. *Iš: Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 331–341.2012.
39. About the Unified Modeling Language Specification Version 2.5. Prieiga per internetą: <https://www.omg.org/spec/UML/2.5/> [žiūrėta 2026 m. kovo 18 d].
40. About the XML Metadata Interchange Specification Version 2.4.2. Prieiga per internetą: <https://www.omg.org/spec/XMI/2.4.2/> [žiūrėta 2026 m. gegužės 17 d].
41. PETRASCH, R. Model-Based Engineering for Microservice Architectures using Enterprise Integration Patterns for Inter-Service Communication. *Iš: 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–4. IEEE, 2017.
42. SANTOS, N.; RODRIGUES, H.; FERREIRA, N. ir MACHADO, R. J. Inputs from a Model-Based Approach Towards the Specification of Microservices Logical Architectures: An

- Experience Report. *Iš: International Conference on Product-Focused Software Process Improvement*, pp. 473–488. Springer, 2019.
43. JIN, K. ir LANO, K. Generation of Test Cases from Uml Diagrams-a Systematic Literature Review. *Iš: Proceedings of the 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, pp. 1–10. 2021.
44. CRUZES, D. S.; MOE, N. B. ir DYBÅ, T. Communication between Developers and Testers in Distributed Continuous Agile Testing. *Iš: 2016 IEEE 11th international conference on Global Software Engineering (ICGSE)*, pp. 59–68. IEEE, 2016.
45. FLOREA, R. ir STRAY, V. The skills that employers look for in software testers. *Software Quality Journal*, t. 27 (2019), nr. 4, p. 1449–1479.
46. MA, S.; FAN, C.; CHUANG, Y.; LIU, I. ir LAN, C. Graph-based and scenario-driven microservice analysis, retrieval, and testing. *Future Generation Computer Systems*, t. 100 (2019), p. 724–735.
47. RAHMAN, M. ir GAO, J. A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development. *Iš: 2015 IEEE Symposium on service-oriented system engineering*, pp. 321–325. IEEE, 2015.
48. LI, N.; ESCALONA, A. ir KAMAL, T. Skyfire: Model-Based Testing with Cucumber. *Iš: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 393–400. IEEE, 2016.
49. GIAMATTEI, L.; GUERRIERO, A.; PIETRANTUONO, R. ir RUSSO, S. Assessing Black-Box Test Case Generation Techniques for Microservices. *Iš: International Conference on the Quality of Information and Communications Technology*, pp. 46–60. Springer, 2022.
50. ALI, S.; BRIAND, L. C.; REHMAN, M. J.; ASGHAR, H.; IQBAL, M. Z. Z. ir kt. A state-based approach to integration testing based on UML models. *Information and Software Technology*, t. 49 (2007), nr. 11-12, p. 1087–1106.
51. ROCHA, M.; SIMÃO, A. ir SOUSA, T. Model-based test case generation from UML sequence diagrams using extended finite state machines. *Software Quality Journal*, t. 29 (2021), nr. 3, p. 597–627.
52. ARCURI, A. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software*, t. 38 (2020), nr. 3, p. 72–78.
53. *What is .NET? an Open-Source Developer Platform | .NET*. Prieiga per internetą: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet> [žiūrėta 2026 m. gegužės 18 d].
54. *About Reqnroll*. Prieiga per internetą: <https://reqnroll.net/about/> [žiūrėta 2026 m. gegužės 18 d].
55. *WireMock.NET - API Mocking for .NET*. Prieiga per internetą: <https://wiremock.org/dotnet/> [žiūrėta 2026 m. gegužės 18 d].
56. *Introduction | Stryker Mutator*. Prieiga per internetą: <https://stryker-mutator.io/docs/stryker-net/introduction/> [žiūrėta 2026 m. gegužės 18 d].
57. AWS-SAMPLES. *Aws-Microservices-Dotnet-Test-Samples*. Prieiga per internetą: <https://github.com/aws-samples/aws-microservices-dotnet-test-samples>.
58. ASC-LAB. *dotnetcore-microservices-poc*. Prieiga per internetą: <https://github.com/asc-lab/dotnetcore-microservices-poc>.

59. YUNUSEMREGL. *AspnetMicroservices*. Prieiga per internetą:
<https://github.com/yunusemregl/AspnetMicroservices>.
60. ALPERENKUCUKALI. *Dotnet-Microservices*. Prieiga per internetą:
<https://github.com/alperenkucukali/dotnet-microservices>.
61. DAOHAINAM. *microservice-patterns*. Prieiga per internetą:
<https://github.com/daohainam/microservice-patterns>.

Magistro baigiamojo projekto rezultatų publikavimas (IVUS 2026 konferencijoje)

<div style="background-color: #c00000; color: white; border-radius: 15px; padding: 5px; display: inline-block;"> IVUS 2026 <small>MAY 12</small> </div>	
INFORMATION SOCIETY AND UNIVERSITY STUDIES	
08:30 – 09:00	REGISTRATION EDU_Lab (Studentų st. 50)
09:00 – 09:15	OPENING CEREMONY Assoc. Prof. Audrius Lopata, Faculty of Informatics, Kaunas University of Technology
<div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="background-color: #c00000; color: white; border-radius: 15px; padding: 5px; display: inline-block;"> In-person </div> <div style="background-color: #c00000; color: white; border-radius: 15px; padding: 5px; display: inline-block;"> Online </div> </div>	
09:15 – 11:00 Session 1, chaired by Assoc. Prof. Audrius Lopata EDU_Lab, Room 108	
09:15 – 09:30	Cvidas Ambrozaitis and Mantas Jurgelaitis Rethinking Model Differencing for SysML v2: Challenges and Future Directions
09:30 – 09:45	Ieva Valatkevičiūtė, Lina Čeponienė and Tomas Valatkevičius The Use of UML Use Case Diagrams and Their Specifications for Test Case Generation
09:45 – 10:00	Arnas Bosas and Eglė Butkevičiūtė AI-Assisted Clinical Decision Support for Urology Oncology: Baseline Machine Learning Experiments on Noisy EHR Data
10:00 – 10:15	Deividas Urbonas and Ramūnas Kubilūnas Updating The Professional Knowledge of Electronics Engineers in the Context of Rapidly Changing Technologies
10:15 – 10:30	Tomas Kanapickas, Ugnius Lukošius and Andrius Kriščiūnas Geo-Localization in OpenStreetMap-Derived Semantic Map Space Using Structural Embeddings
10:30 – 10:45	Tomas Valnoris and Lina Čeponienė Automated Test Generation for Microservices from UML Sequence Diagrams
10:45 – 11:00	Kyrylo Kolesnichenko and Martas Ambraziunas Can Structured Prompting Overcome Knowledge
09:15 – 11:00 Session 1, chaired by Prof. Daina Gudoniene EDU_Lab, Room 106	
09:15 – 09:30	Jakub Dras and Piotr Pilszczek Hawk-Dove Dynamics in Continuous Spaces: Escaping Local Optima via Game-Theoretic Swarm Regulation
09:30 – 09:45	Maksim Iavich, Vladimir Svanadze and Ketevani Grdzeldze Development of an Information Security Governance Framework in Fragmented Circumstances
09:45 – 10:00	Malika Baizakova and Kadyralieva Nurayim A Federated Graph Neural Network for Integrating Clinical, Imaging, and Genetic Data Across Hospital Systems
10:00 – 10:15	Piotr Pilszczek The Effect of Providing LSTM-Based Neural Networks Used for Time-Series Predictions with Relational Data
10:15 – 10:30	Zhibek Primova and Gulzat Muktaranova Explainable AI for Fair and Transparent Resource Allocation in Smart Cities
10:30 – 10:45	Bohdan Kostyk An Algorithm Inspired by the Ant Colony System for Solving NxN Sliding Puzzles
10:45 – 11:00	Mukhriddin Arabboev, Begmatov Shohruh, Sukhrob Bobojanov, Nor Erne Nazira Bazin and Akhram Nishanov

Užklausa naudota generuoti sekų diagramas iš mikroservisų programinio kodo

```
You are a senior .NET code analyst.
Your task is to analyze a given .NET endpoint and map all possible execution
flows starting from the entry point.
---
## 1. Start from the Endpoint
* Identify the controller and specific endpoint method (e.g., HTTP GET/POST
action).
* Note input parameters, attributes, and routing details.
---
## 2. Trace the Call Chain
* Follow all method calls initiated by the endpoint.
* Traverse through:
  * Controllers → Services → Clients → Repositories → External integrations
* Continue recursively until you reach terminal operations (e.g., database
calls, external APIs, or return statements).
---
## 3. Identify Branching Logic
* Detect all conditional paths:
  * if/else
  * switch statements
  * polymorphism / interface implementations
  * exceptions and error handling
* Treat each unique branch as a separate flow.
---
## 4. Track Dependencies
* Identify injected dependencies (via constructor or DI container).
* Resolve which concrete implementations are used where possible.
---
## 5. Define Unique Flows
* A “flow” is a distinct execution path from the endpoint to a terminal outcome.
* Count and list all unique flows.
### HARD RULES:
* ONLY include flows explicitly defined in code.
* DO NOT include implicit or unhandled failures.
  * Example: if HttpClient could return 500 but no handling exists → IGNORE it.
* Flows must differ by:
  * Branching decisions
  * Different downstream calls
  * Different explicitly defined outcomes
---
## 6. Response Mapping Rules (STRICT – ZERO INFERENCE)
Every flow MUST end with an HTTP response explicitly present in the code.
### Format:
* `statusCode(responseModel)` → when a response body/model exists
* `statusCode` → when no response body is returned
```

```

### HARD CONSTRAINTS:
* Status codes MUST be integers (e.g., 200, 201, 400, 404).
* ONLY include status codes that are:
  * Returned directly in code (`return Ok(...)`, `return BadRequest(...)`,
  `StatusCode(...)`, etc.)
  * OR explicitly mapped in visible code (e.g., custom exception handling shown)
### STRICTLY FORBIDDEN:
* DO NOT infer or assume any responses
* DO NOT include:
  * 500 Internal Server Error (unless explicitly returned in code)
  * 502 / 503 / downstream failures
  * Framework/default ASP.NET behavior
  * HttpClient implicit failures
### VALIDATION (MANDATORY):
For EACH status code:
* Quote or reference the exact code location that produces it
* If you cannot find it → REMOVE it
---
## 7. Generate Mermaid Sequence Diagram (REQUIRED)
Create a Mermaid `sequenceDiagram` representing ONLY high-level external
interactions.
### Participants:
* External client (MUST be the first caller)
* This service (as a SINGLE black-box participant)
* Other microservices / external systems
### STRICT VISIBILITY RULES:
DO NOT show:
* Controllers, services, repositories
* Internal method calls
* Internal branching logic
### WHAT TO SHOW:
* Calls from client → this service
* Calls from this service → external systems
* Responses from external systems
* Final response back to client
### BRANCHING:
* Use `alt` / `else` ONLY for flows explicitly defined in code
### RESPONSE FORMAT (MANDATORY):
* Use:
  * `statusCode(responseModel)`
  * or `statusCode`
* Apply SAME strict response rules (no inference)
### FORBIDDEN IN DIAGRAM:
* Any unhandled or implicit failures
* Any status codes not proven from code
### VALIDATION:
* If code defines only success → diagram MUST show only success
* It is VALID to have zero non-2xx responses
### GOAL:

```

Represent the service as a **black box**, showing ONLY inter-service communication.

8. Output Mermaid in a Separate File

* Create a new file named: `endpoint_flows.md`

* The file must contain ONLY the Mermaid diagram inside:

```mermaid

sequenceDiagram

...

\* No explanations in this file.

---

## ## 9. Output Format (Main Response)

Provide:

### ### A. Endpoint Summary

\* What the endpoint does

### ### B. Dependency Chain (text)

Example:

Controller → Service → Client → External API

### ### C. Unique Flows (numbered)

For EACH flow:

\* Step-by-step execution path (INCLUDING internal components)

\* Key decisions/branches

\* Final outcome using required format:

\* `statusCode(responseModel)` or `statusCode`

\* Reference to exact code that produces the response

### ### D. Total Flow Count

---

## ## 10. Determinism Requirements

\* Do NOT guess

\* Do NOT assume framework behavior

\* Do NOT “fill in missing cases”

\* If something is unclear → explicitly state uncertainty

\* Prefer UNDER-reporting over incorrect reporting

---

## ## FINAL GOAL

Produce:

1. A precise, code-grounded breakdown of all execution flows

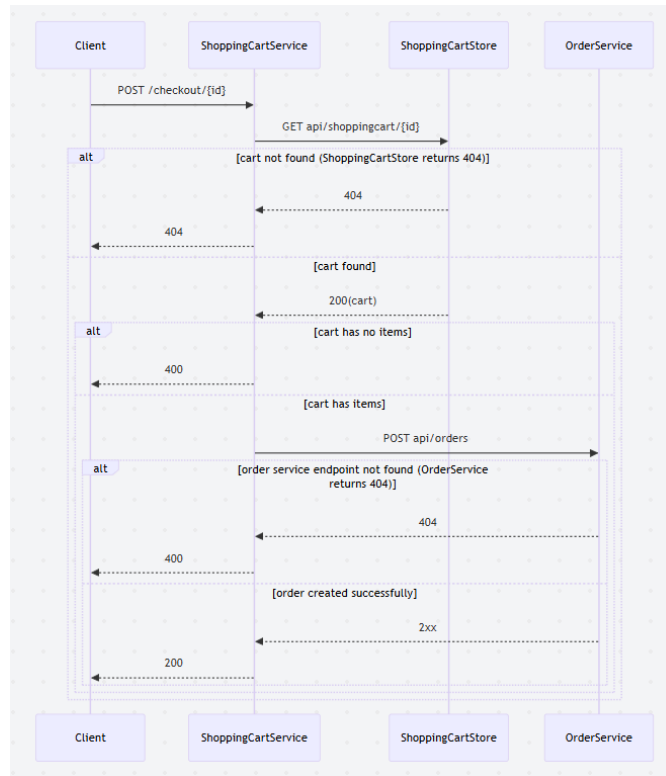
2. ZERO hallucinated responses or flows

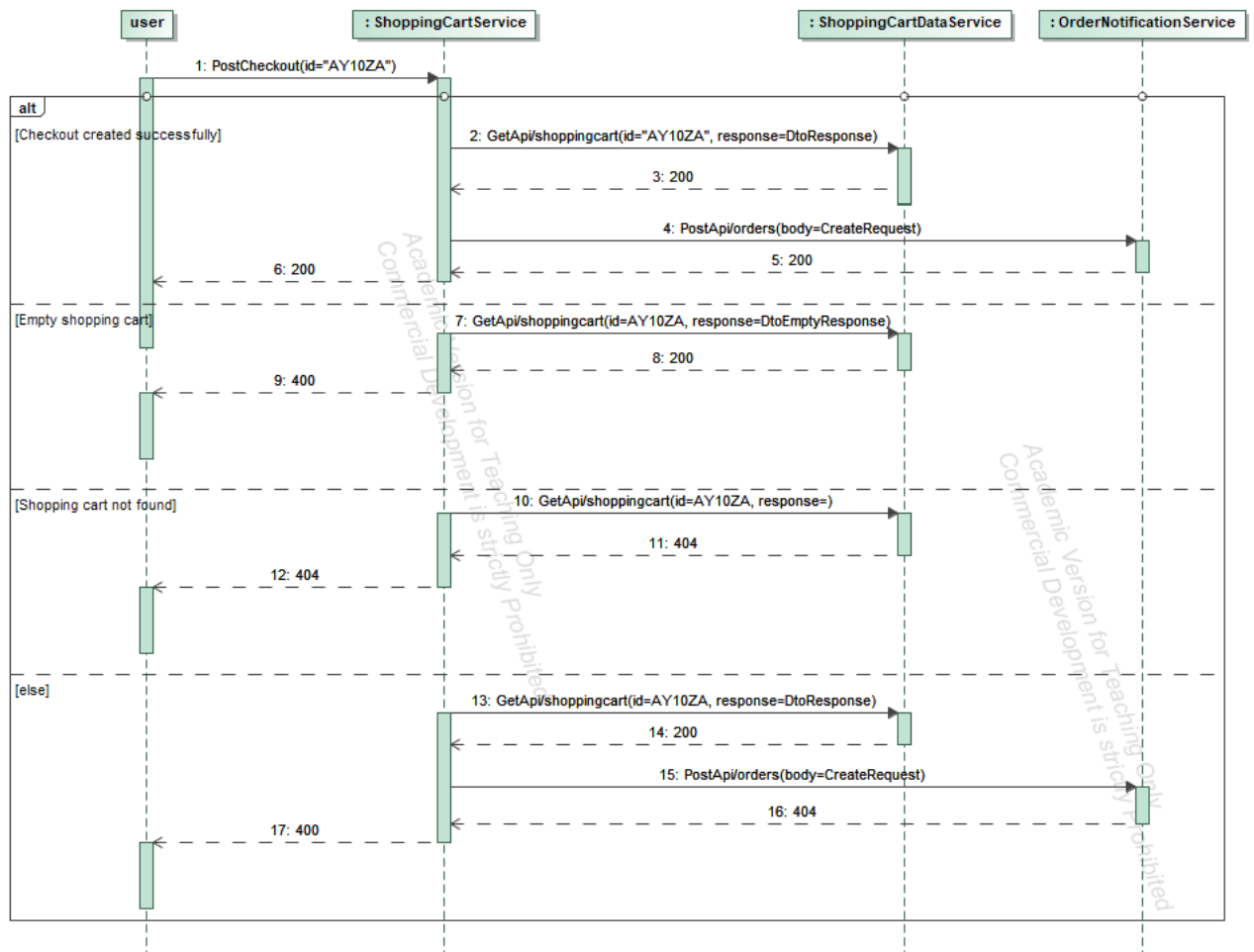
3. A clean sequence diagram showing ONLY real, external interactions

### Eksperimente naudotos sekų diagramos

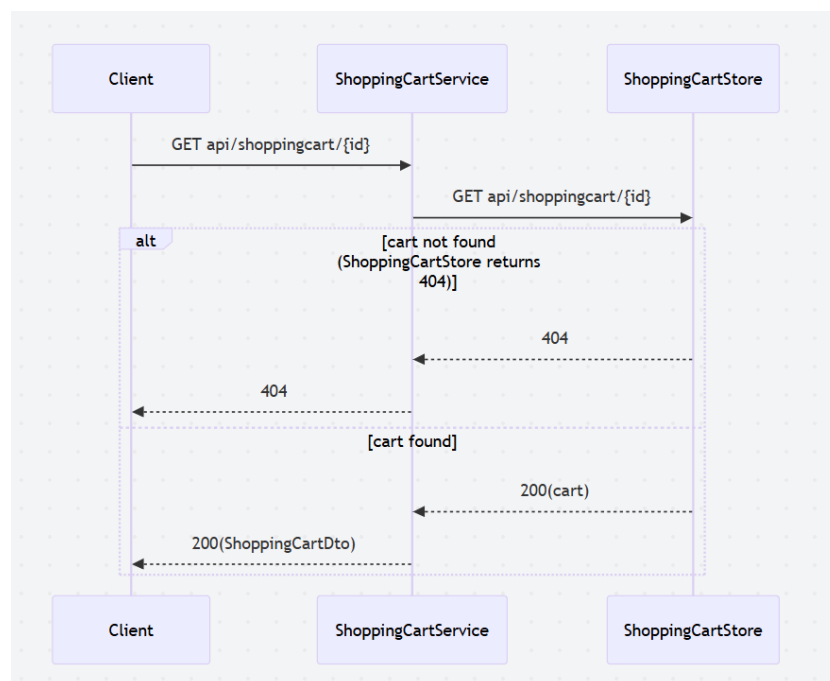
Šiame priede pateiktos sekų diagramos *Mermaid* formatu ir analogiškos sekų diagramos nubraižytos *Magic Systems of Systems Architect* įrankyje.

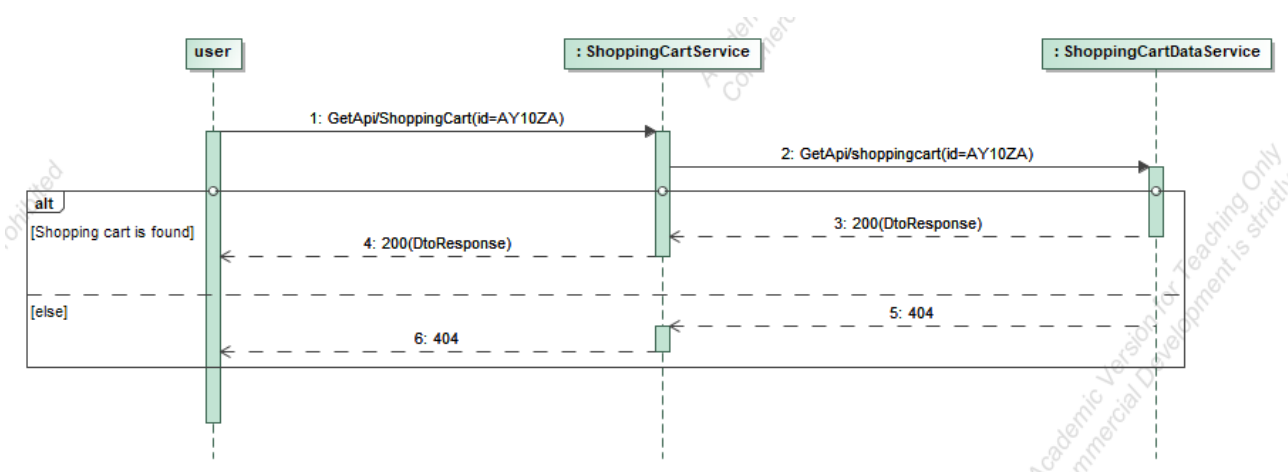
Žemiau pateikiamos ShoppingCartService mikroserviso *API* galutinio taško *checkout* sekų diagramos.



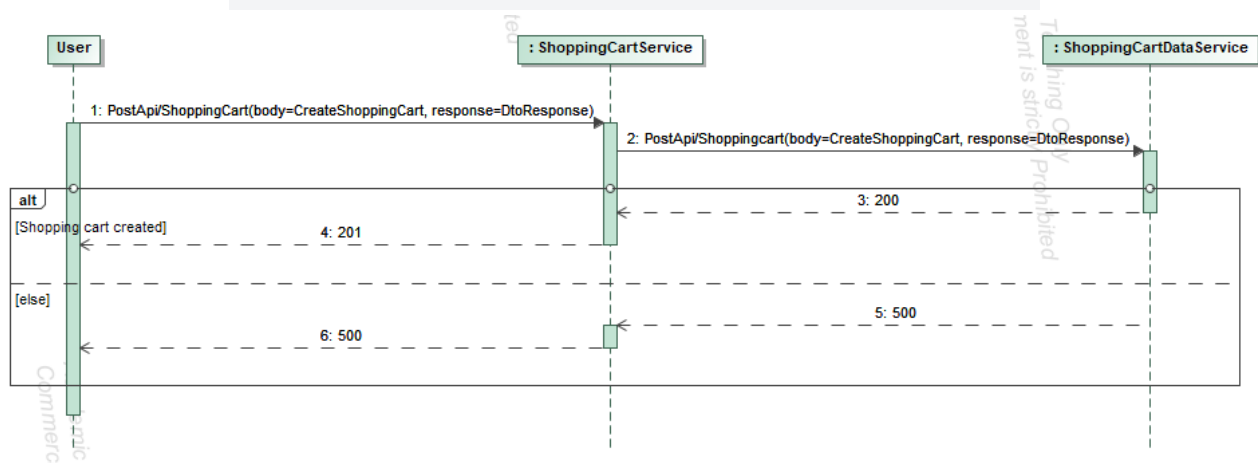
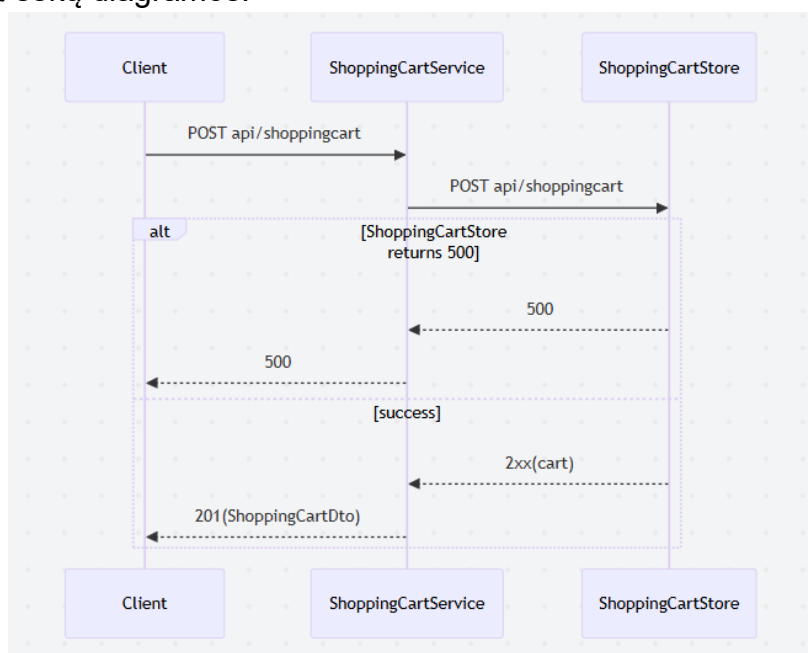


Žemiau pateikiamos ShoppingCartService mikroserviso API galutinio taško GET api/shoppingcart sekų diagramos.

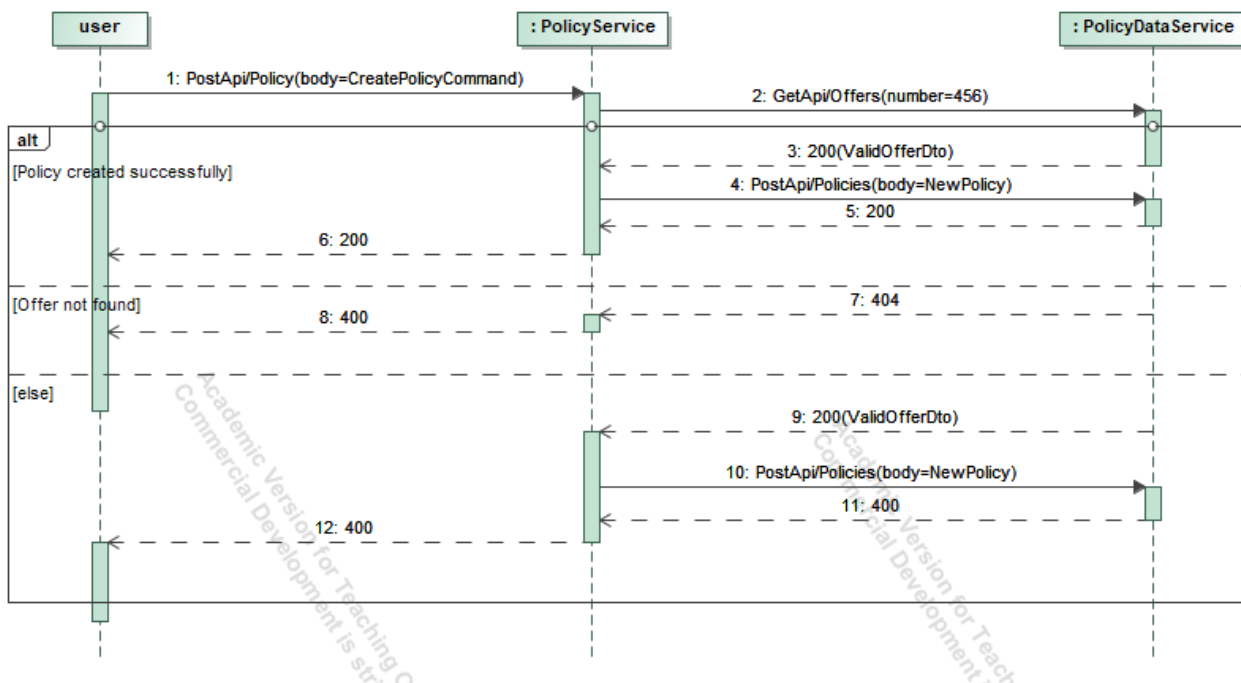
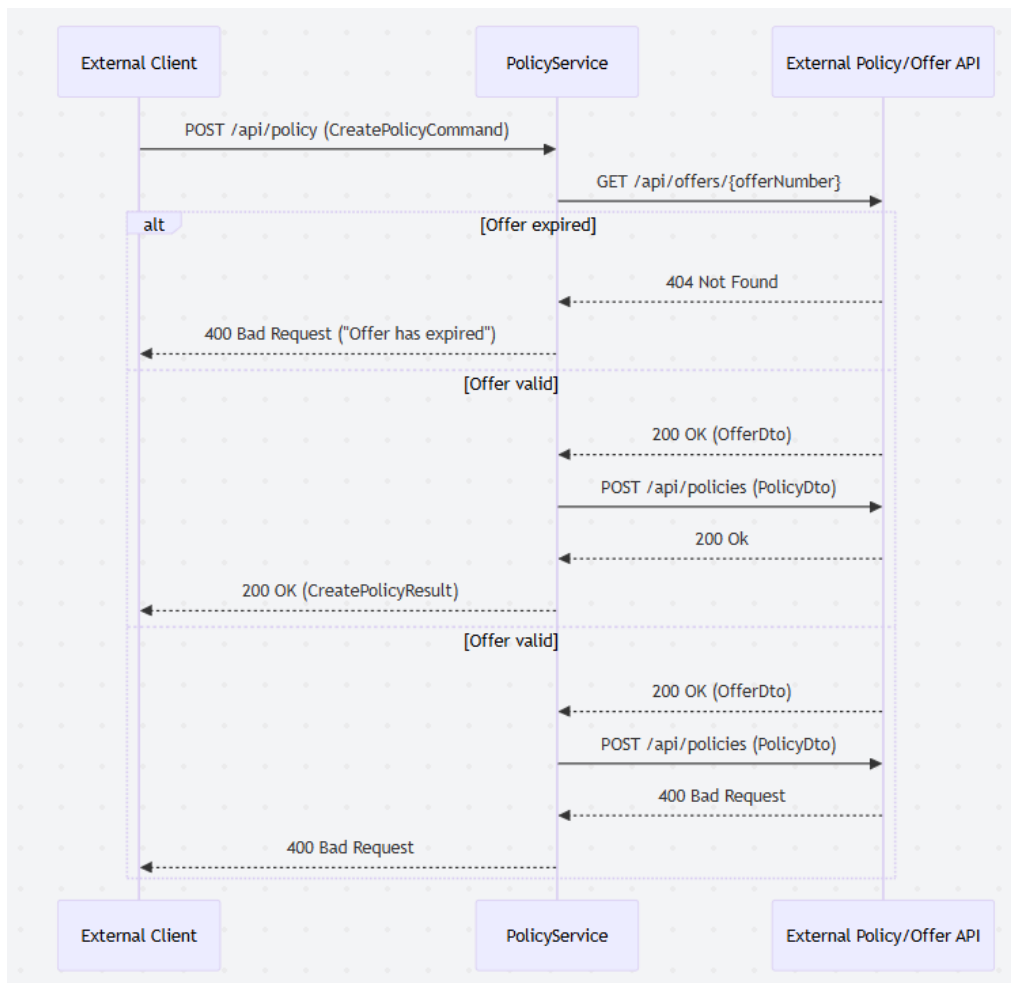




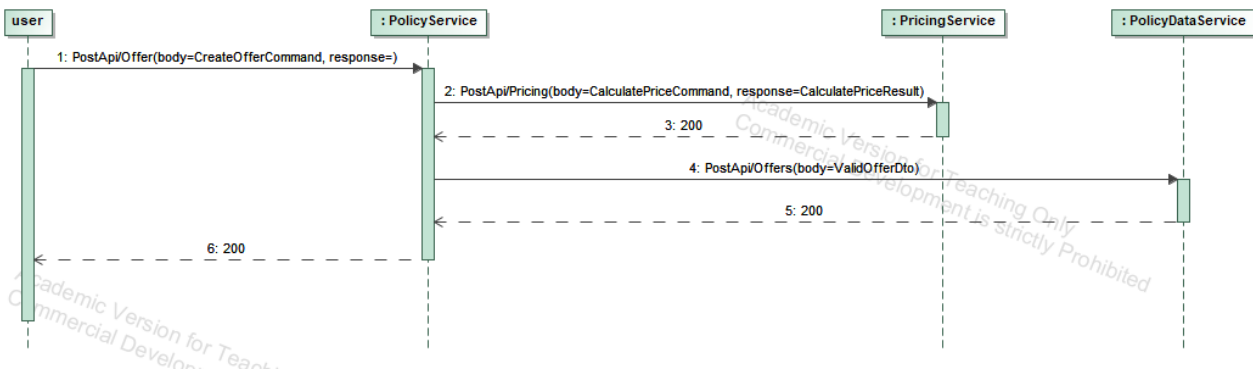
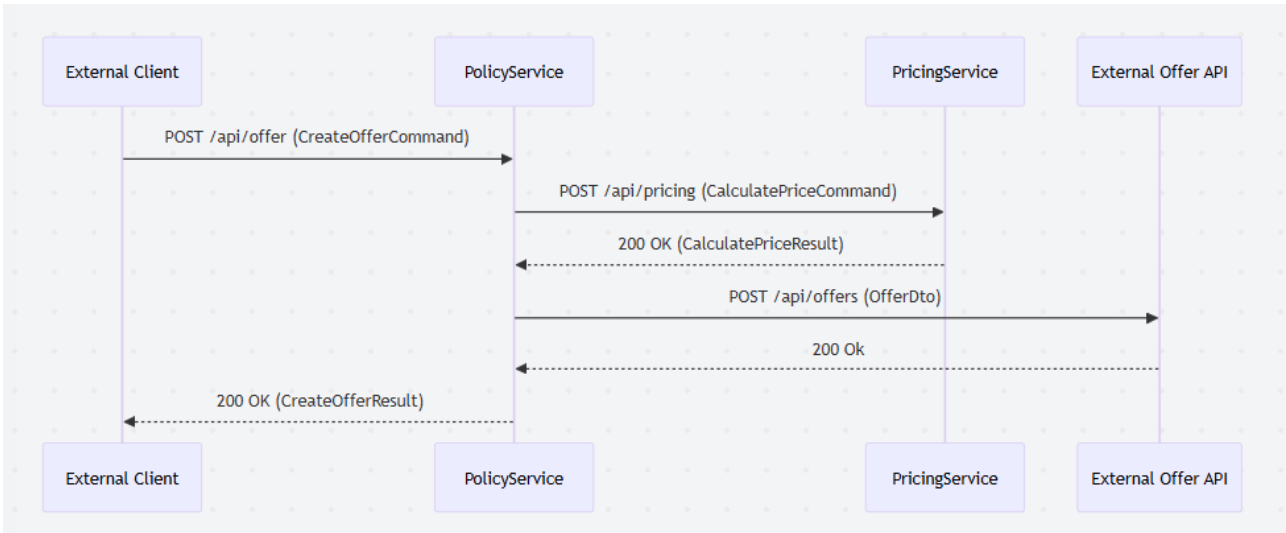
Žemiau pateikiamos ShoppingCartService mikroserviso API galutinio taško *POST api/shoppingcart* sekų diagramos.



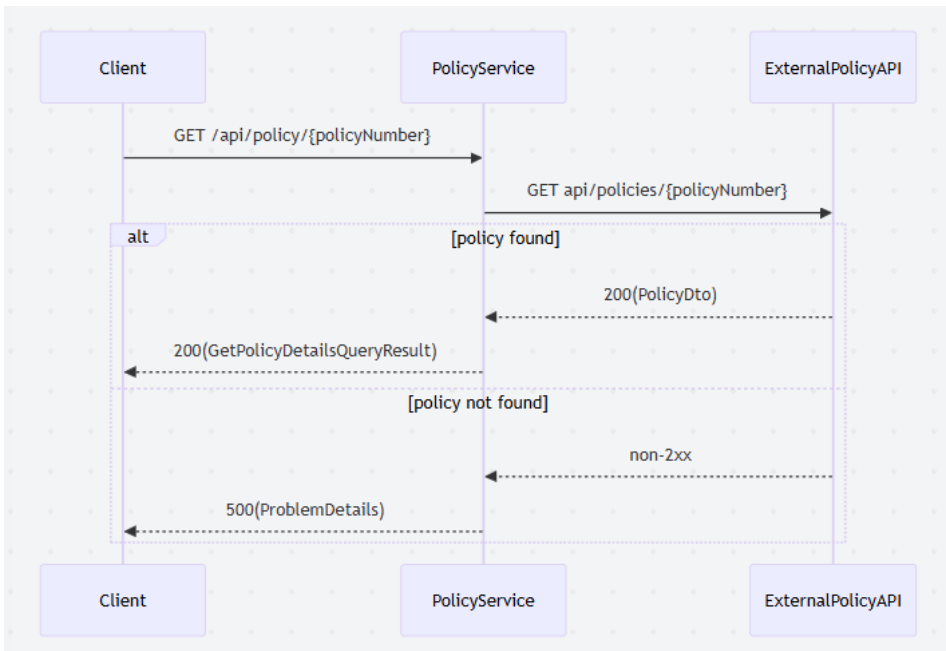
Žemiau pateikiamos PolicyService mikroserviso API galutinio taško *POST api/policy* sekų diagramos.

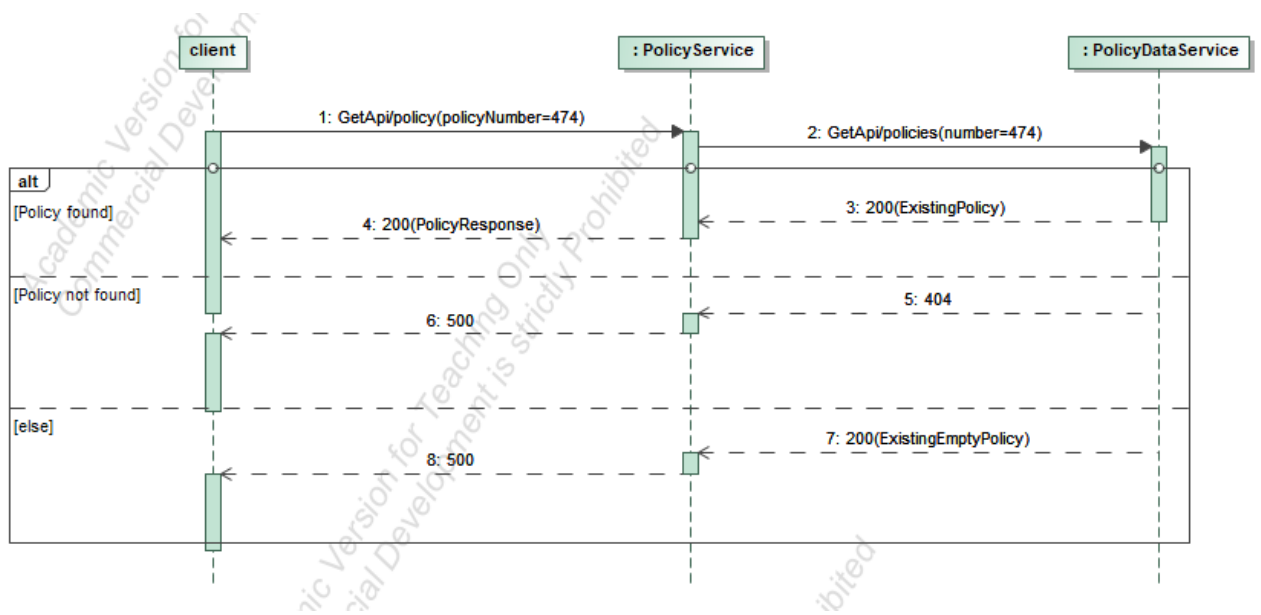


Žemiau pateikiamos *PolicyService* mikroserviso API galutinio taško *POST api/offer* sekų diagramos.

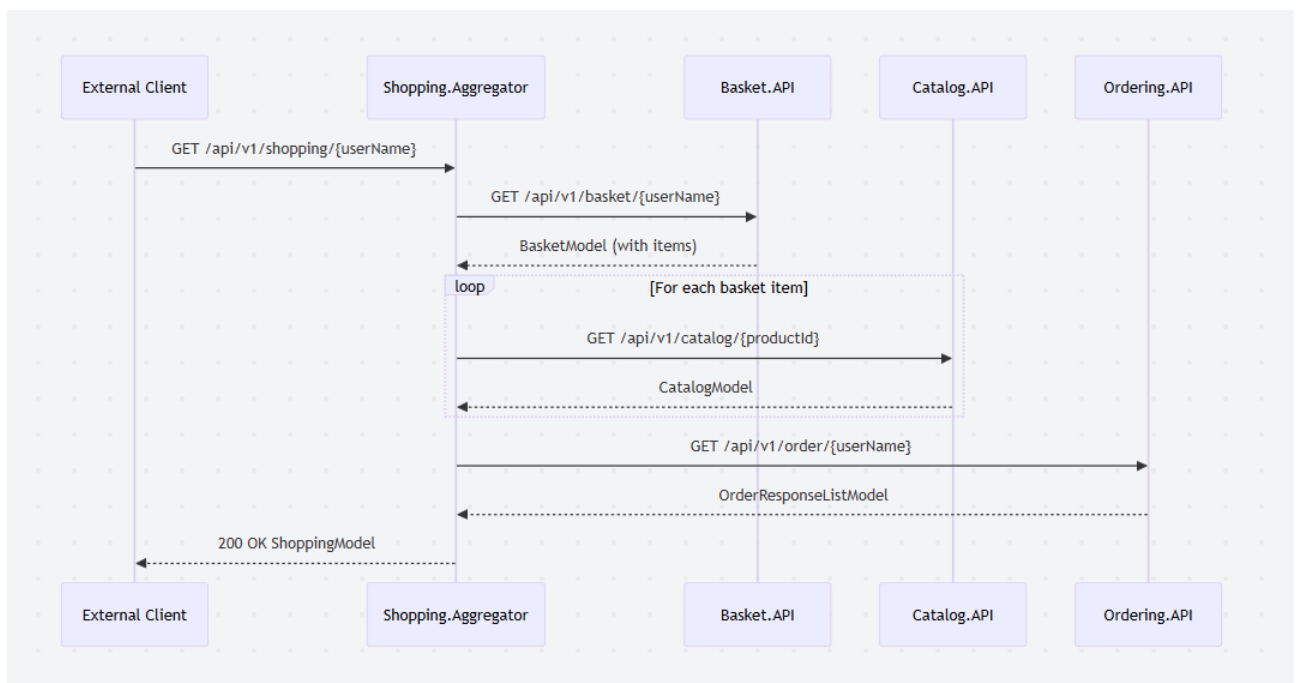


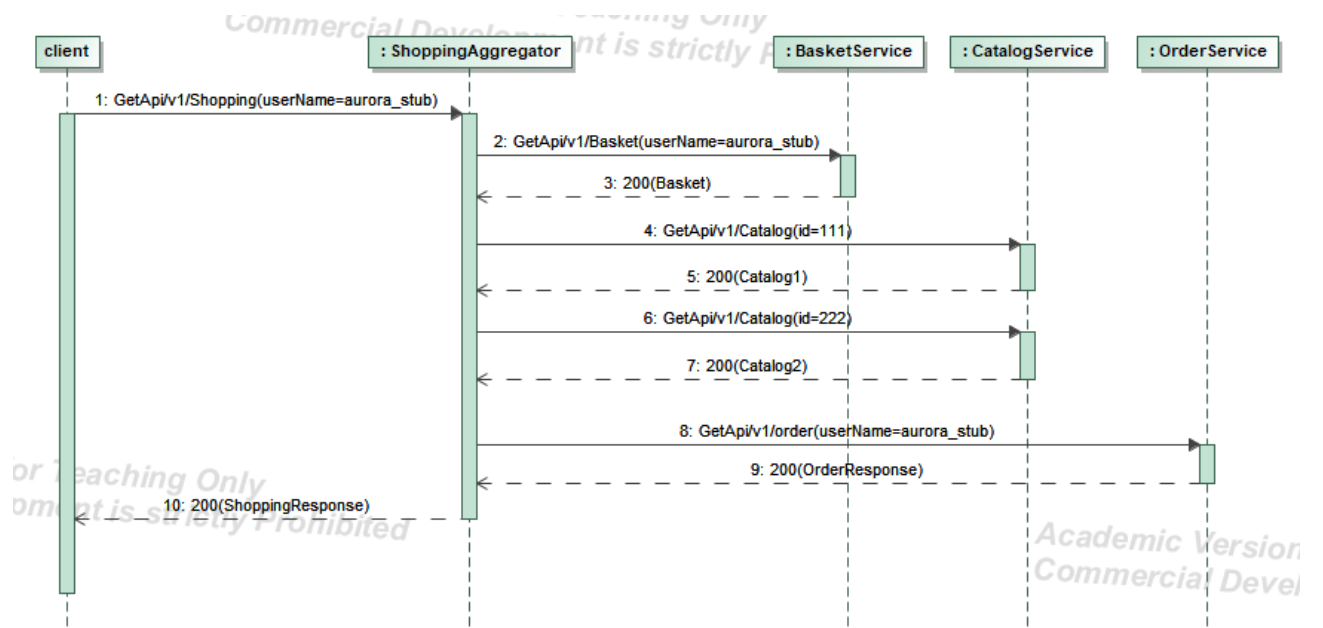
Žemiau pateikiamos *PolicyService* mikroserviso *API* galutinio taško *GET api/policy* sekų diagramos.



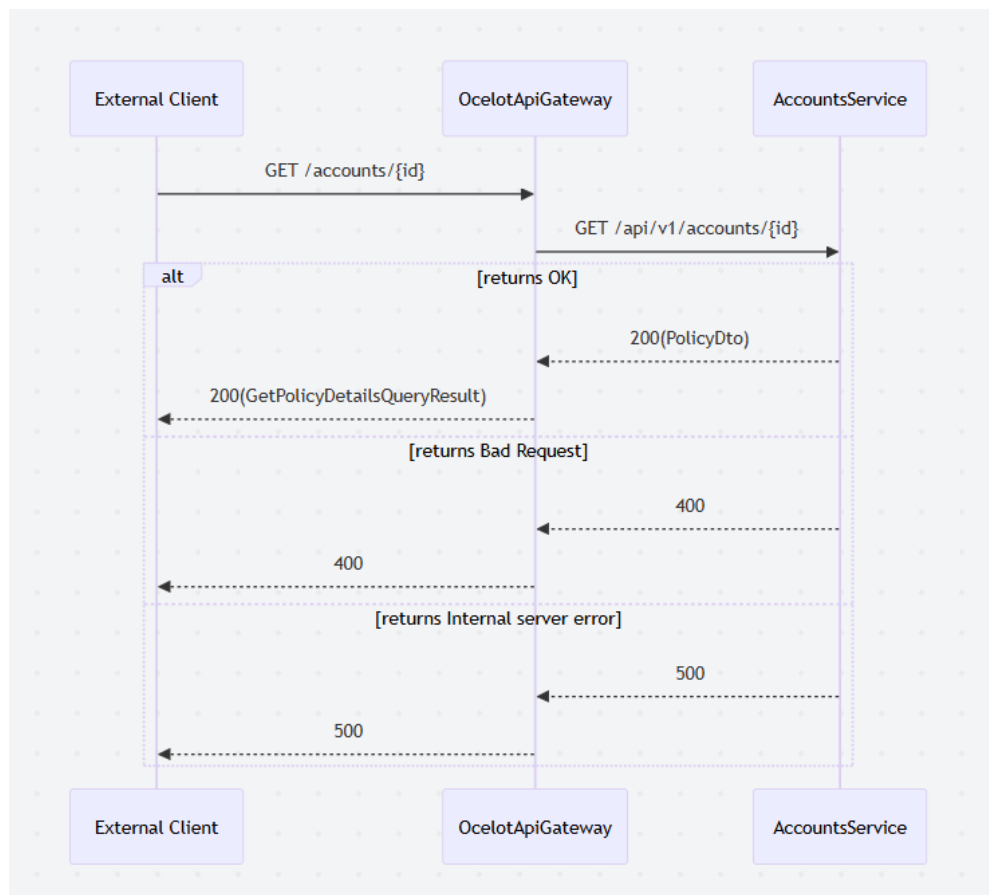


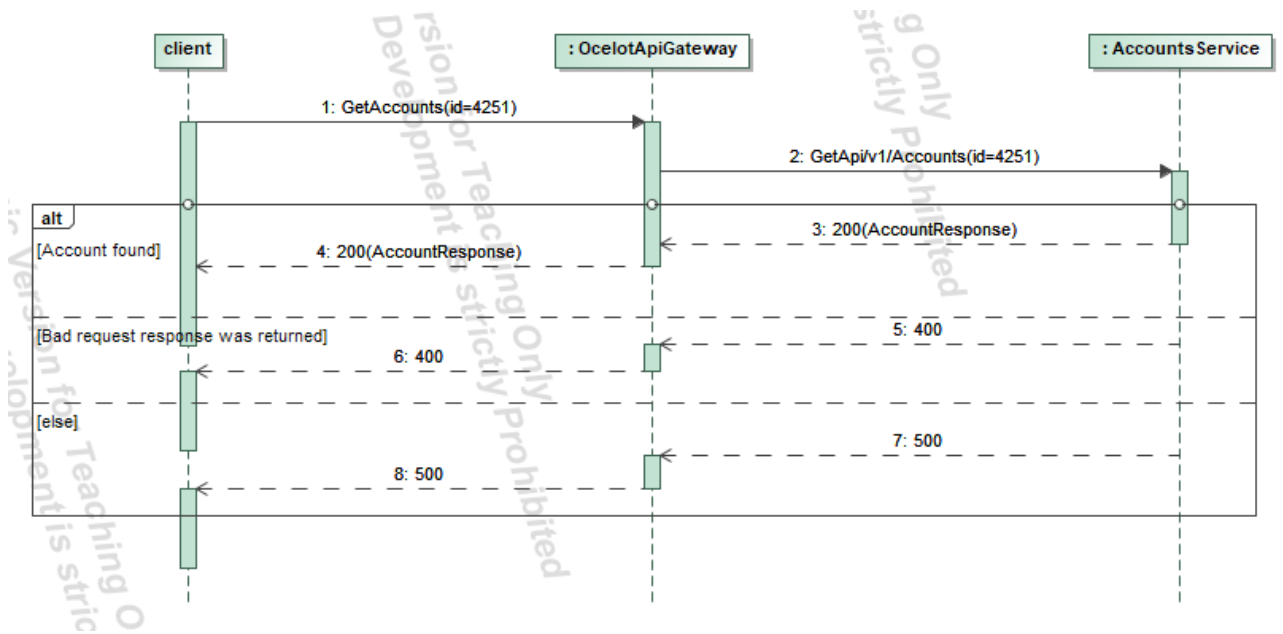
Žemiau pateikiamos *ShoppingAggregator* mikroserviso API galutinio taško *GET api/v1/shopping* sekų diagramos.



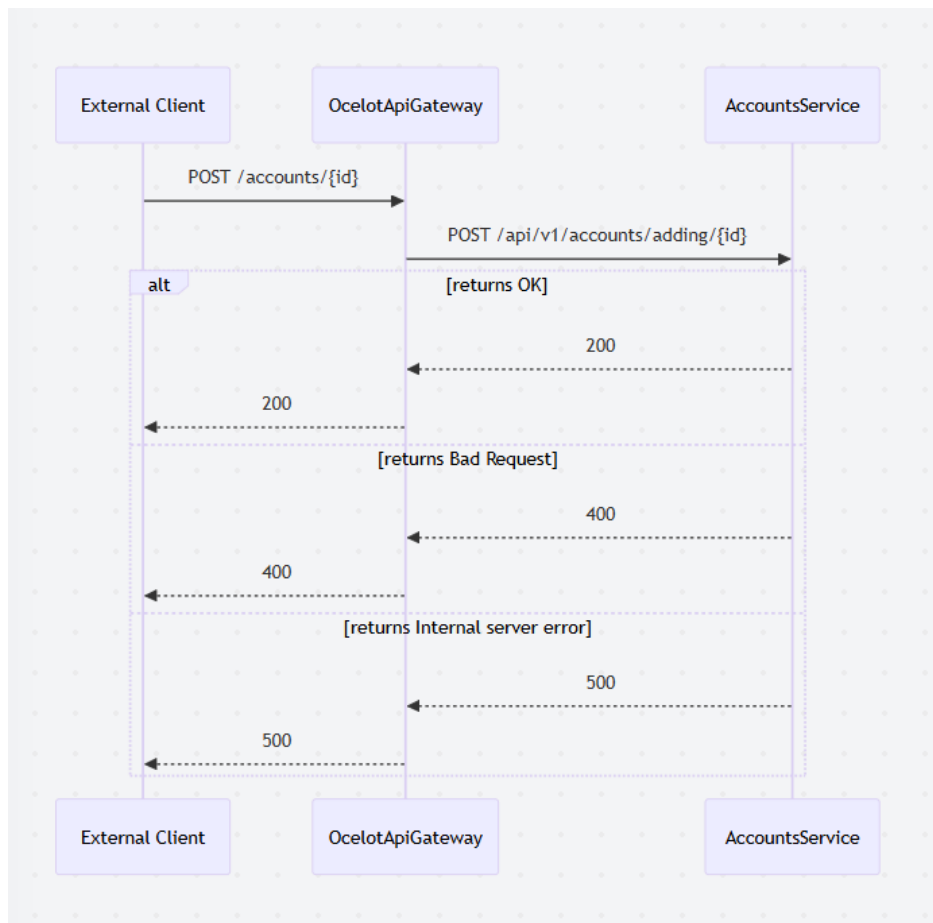


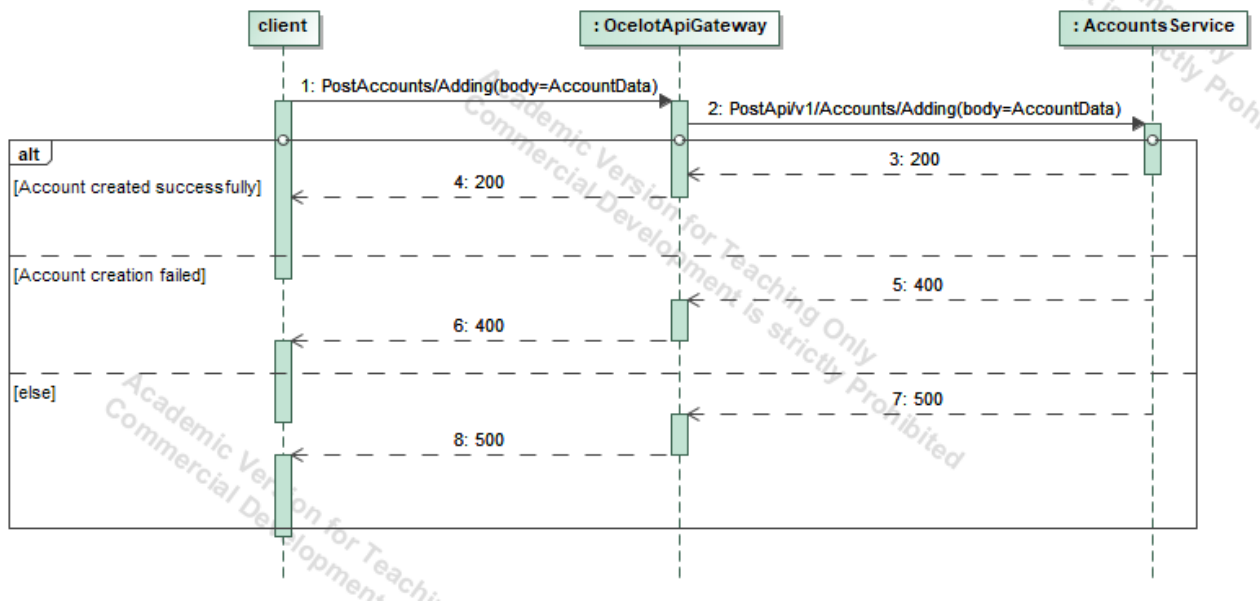
Žemiau pateikiamos *OcelotApiGateway* mikroserviso API galutinio taško *GET ap/accounts* sekų diagramos.





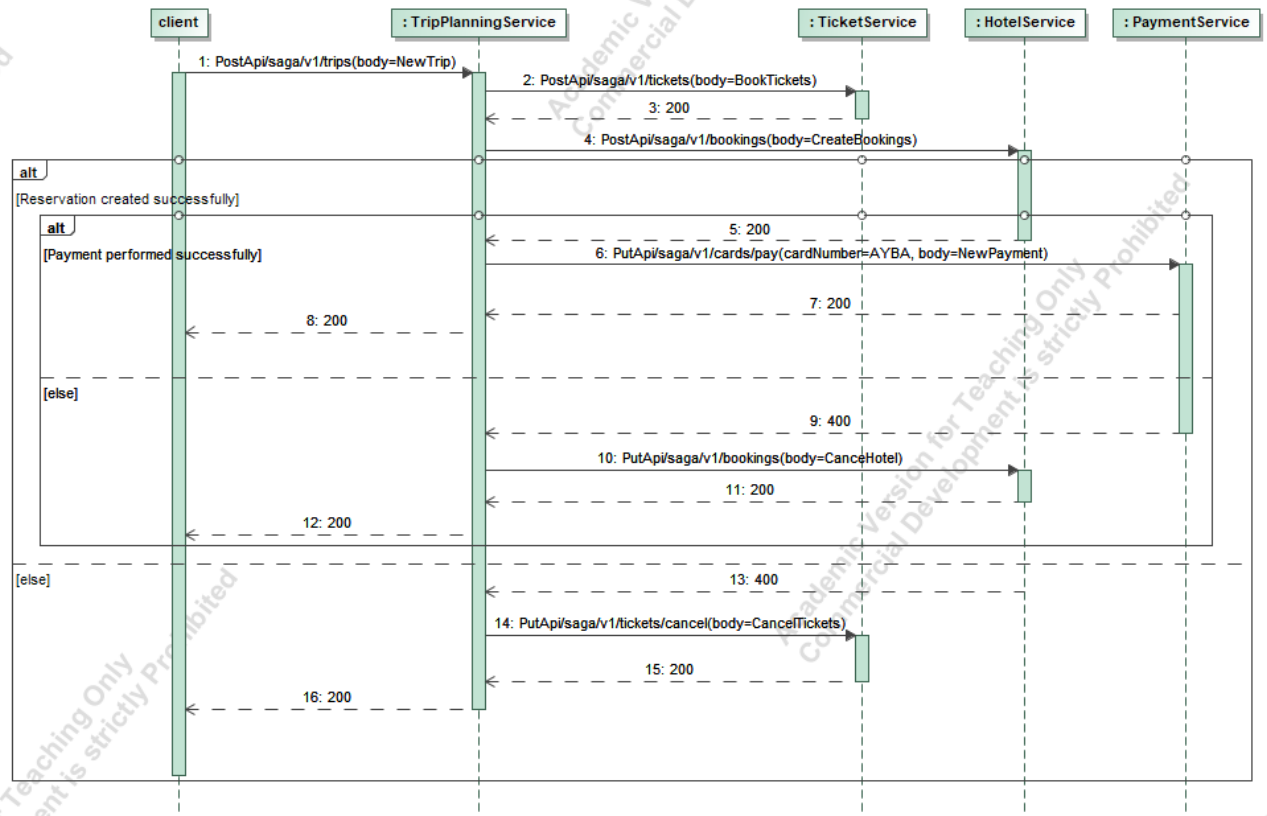
Žemiau pateikiamos *OcelotApiGateway* mikroserviso API galutinio taško *Post ap/accounts* sekų diagramos.





Žemiau pateikiamos *TripPlanningService* mikroserviso API galutinio taško `POST api/saga/v1/trips` sekų diagramos.



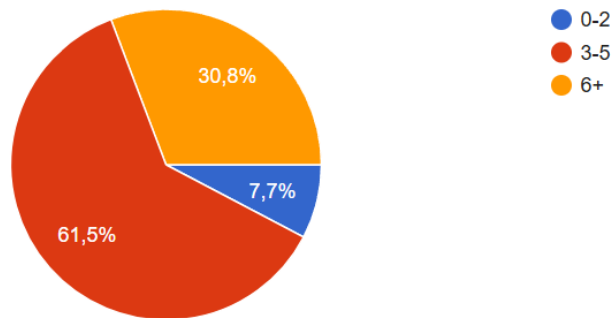


## Apklauso rezultatai

Kokia yra Jūsų programinės įrangos kūrimo arba testavimo patirtis metais?

[Kopijuoti diagramą](#)

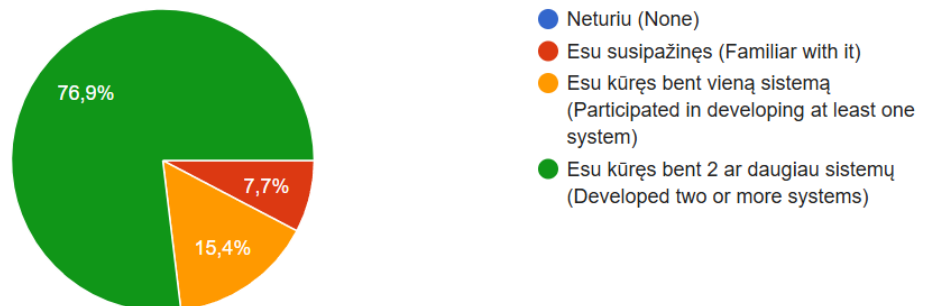
13 atsakymų



Kiek praktinės patirties turite dirbant su mikroservisų architektūros sistemomis?

[Kopijuoti diagramą](#)

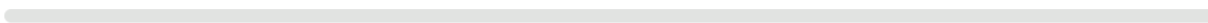
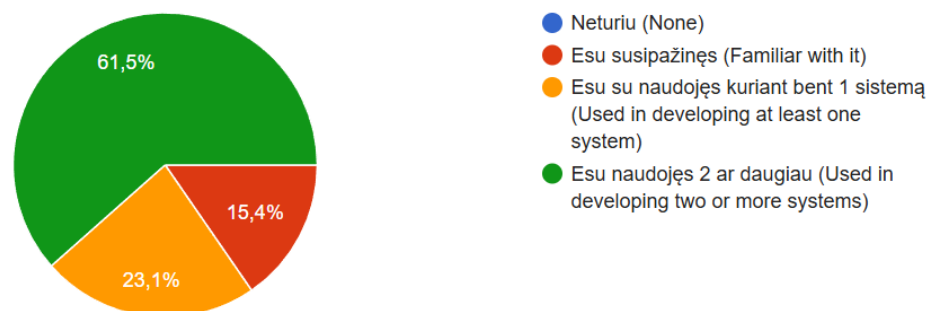
13 atsakymų



Kiek patirties turite naudojant UML diagramas?

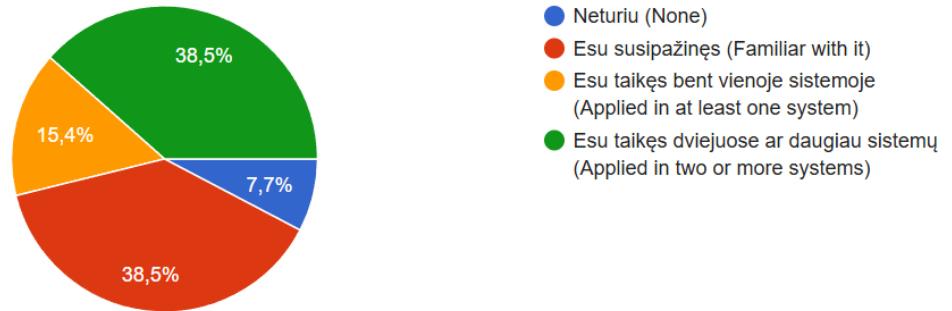
[Kopijuoti diagramą](#)

13 atsakymų



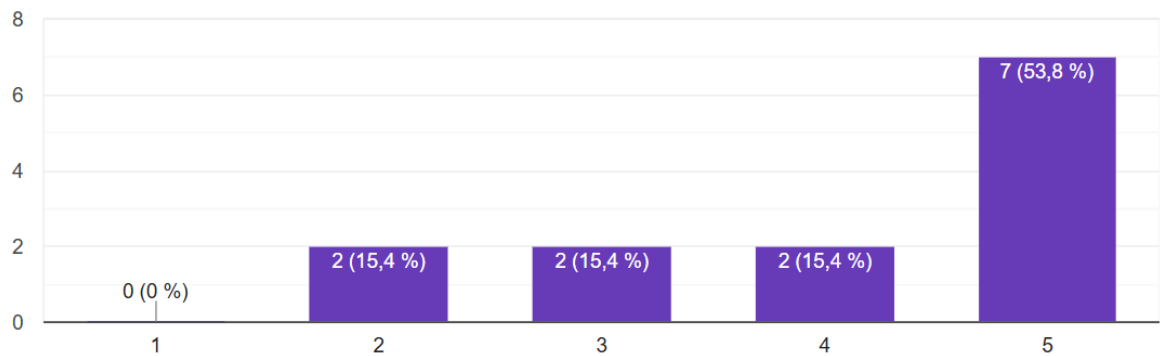
Kiek patirties turite dirbant su elgsena grindžiamu kūrimu (BDD) ir naudojant tokias priemones kaip Cucumber, Gherkin, SpecFlow ar Reqnroll?

13 atsakymų



Pristatyta testų generavimo iš sekų diagramų metodika palengvintų mikroservisų testavimą.

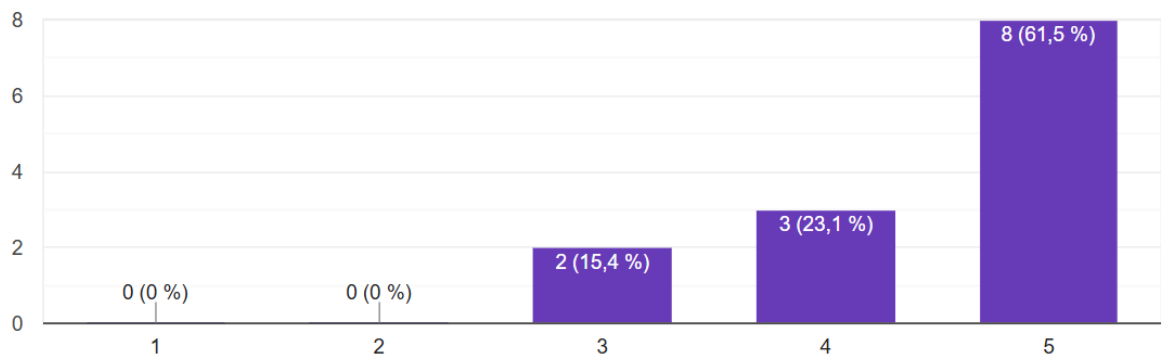
13 atsakymų



Pristatytos metodikos sugeneruoti testai pasirodė naudingi sistemos elgsenai ir mikroservisų tarpusavio sąveikai tikrinti.

[Kopijuoti diagramą](#)

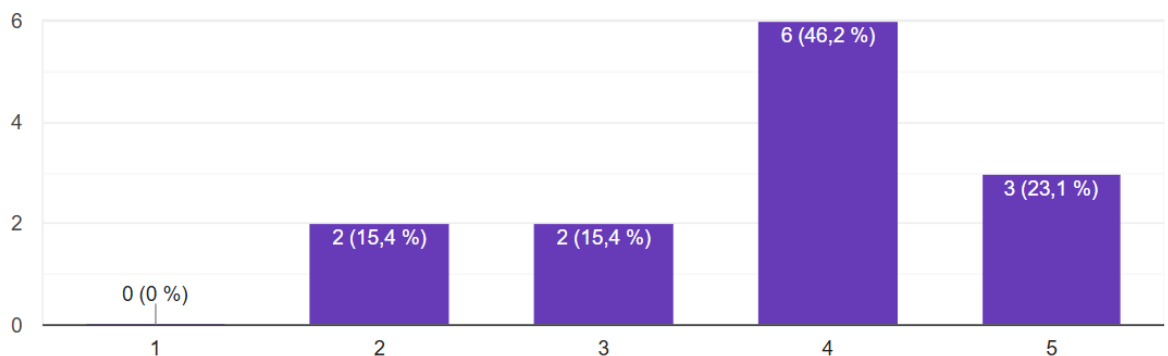
13 atsakymų



Jei dirbčiau projekte, kuriame mikroservisų sąveika aprašoma UML sekų diagramomis, taikyčiau šią metodiką testams generuoti.

[Kopijuoti diagramą](#)

13 atsakymų



Kokias papildomas įžvalgas norėtumėte pateikti apie šią metodiką?

2 atsakymai

BDD is aimed to be for non-technical people. Although in practice usually a Dev team works with it. So although by the book not correct, it solves practical problems.

Manau puikus įrankis, jei projektuose jau naudojamos sekų diagramos.