УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Језик за опис архитектуре дистрибуираних система базираних на микросервисима

ДОКТОРСКА ДИСЕРТАЦИЈА

Ментор:
проф. др Игор Дејановић

Кандидат:
Ален Суљкановић

Нови Сад, 2023. године

УНИВЕРЗИТЕТ У НОВОМ САДУ **ОБРАЗАЦ – 5**
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА[1]

Врста рада:	Докторска дисертација
Име и презиме аутора:	Ален Суљкановић
Ментор (титула, име, презиме, звање, институција)	Др Игор Дејановић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду
Наслов рада:	Језик за опис архитектуре дистрибуираних система базираних на микросервисима
Језик публикације (писмо):	Српски (ћирилица)
Физички опис рада:	Унети број: Страница _____92__ Поглавља_____6__ Референци ___130__ Табела_____13__ Слика _____12__ Графикона _____0__ Прилога _____0__
Научна област:	Електротехничко и рачунарско инжењерство
Ужа научна област (научна дисциплина):	Примењене рачунарске науке и информатика
Кључне речи / предметна одредница:	моделовање, метамоделовање, софтверски језици, дистрибуирани системи, микросервиси, сервис-оријентисане архитектуре
Резиме на језику рада:	Докторска дисертација се бави проблемом развоја дистрибуираних система базираних на микросервисима. Циљ докторске дисертације је убрзање развоја и побољшање квалитета оваквих система. У оквиру израде дисертације имплементиран је језик *Silvera*, који се од сличних решења издваја по могућности аутоматског генерисања документације, те евалуације архитектуре система помоћу посебно дефинисаних метрика. У циљу евалуације језика, спроведена је анкета заснована на FQAD радном оквиру, где је од учесника тражено да имплементирају једноставан задатак са и без коришћења *Silvera* језика, а потом да попуне упитник. Учесници анкете су задатак имплементирали ~124% брже када су користили Silvera језик. Уз то, коришћење *Silvera* језика је довело и до квалитетнијих решења, са значајно мање грешака. Креирањем *Silvera* језика извршена је и формализација домена моделовања архитектура дистрибуираних система базираних на микросервисима, што представља и оригинални научни допринос дисертације.

[1] Аутор докторске дисертације потписао је и приложио следеће Обрасце:
5б – Изјава о ауторству;
5в – Изјава о истоветности штампане и електронске верзије и о личним подацима;
5г – Изјава о коришћењу.
Ове Изјаве се чувају на факултету у штампаном и електронском облику и не кориче се са тезом.

Датум прихватања теме од стране надлежног већа:	
Датум одбране: (Попуњава одговарајућа служба)	
Чланови комисије: (титула, име, презиме, звање, институција)	Председник: др Бранко Милосављевић, редовни професор, Факултет техничких наука, Нови Сад Члан: др Горан Сладић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду Члан: др Владимир Димитриески, доцент, Факултет техничких наука, Универзитет у Новом Саду Члан: др Владимир Вујовић, ванредни професор., Електротехнички факултет, Универзитет Источно Сарајево Ментор: др Игор Дејановић, редофни професор, Факултет техничких наука, Универзитет у Новом Саду
Напомена:	

UNIVERSITY OF NOVI SAD
FACULTY OR CENTER

KEY WORD DOCUMENTATION[2]

Document type:	Doctoral dissertation
Author:	Alen Suljkanović
Supervisor (title, first name, last name, position, institution)	PhD Igor Dejanović, full professor, Faculty of Technical Sciences, University of Novi Sad
Thesis title:	A language for description of distributed systems
Language of text (script):	Serbian language (cyrillic)
Physical description:	Number of: Pages_____92_____ Chapters _____6_____ References__130_____ Tables_____13_____ Illustrations __12_____ Graphs_____0_____ Appendices___ 0_____
Scientific field:	Electrical and computer engineering
Scientific subfield (scientific discipline):	Applied Computer Science and Informatics
Subject, Key words:	modeling, metamodeling, software languages, distributed systems, microservices, service-oriented arhichitectures
Abstract in English language:	The doctoral dissertation addresses the challenge of developing distributed systems based on microservices. The aim of the doctoral dissertation is to accelerate the development and enhance the quality of such systems. During the dissertation process, the Silvera language was implemented, which stands out from similar solutions due to its capability for automatic documentation generation and system architecture evaluation using specifically defined metrics. To evaluate the language, a survey based on the FQAD framework was conducted, where participants were asked to implement a simple task with and without the use of the Silvera language, followed by filling out a questionnaire. Participants of the survey completed the task ~124% faster when using the Silvera language. Moreover, the utilization of the Silvera language led to higher quality solutions, with significantly fewer errors. The creation of the Silvera language also accomplished the formalization of the domain for modeling architectures of distributed systems based on microservices, which represents an original scientific contribution of the dissertation.

[2] The author of doctoral dissertation has signed the following Statements:
5б – Statement on the authority,
5в – Statement that the printed and e-version of doctoral dissertation are identical and about personal data,
5г – Statement on copyright licenses.
The paper and e-versions of Statements are held at he faculty and are not included into the printed thesis.

Accepted on Scientific Board on:	
Defended: (Filled by the faculty service)	
Thesis Defend Board: (title, first name, last name, position, institution)	President: PhD Branko Milosavljević, full professor, Faculty of Technical Sciences, University of Novi Sad Member: PhD Goran Sladić, full professor, Faculty of Technical Sciences, University of Novi Sad Member: PhD Vladimir Dimitrieski, assistent professor, Faculty of Technical Sciences, University of Novi Sad Member: PhD Vladimir Vujović, associate professor, Faculty of Electrical Engineering, University of East Sarajevo Supervisor: PhD Igor Dejanović, full professor, Faculty of Technical Sciences, University of Novi Sad
Note:	

Желим да изразим своју искрену захвалност ментору, проф. др. Игору Дејановићу, који ми је пружио неизмерну подршку током целог процеса истраживања.

Захваљујем се и колегама и сарадницима који су делили своје знање и искуство са мном, те ми омогућили да проширим своје видике.

Посебну захвалност дугујем својој породици на љубави која ми је помогла да истрајем.

Садржај

Резиме

Развој дистрибуираних система је изазован процес који захтева пажљиво планирање и имплементацију. Од свог настанка, дистрибуирани системи су се развијали са циљем повећања скалабилности, поузданости, перформанси и доступности апликација, те смањења трошкова одржавања. Сваки од ових циљева са собом носи и изазове које је потребно пребродити како би се систем имплементирао на најбољи могући начин. Временом, појављивали су се разни архитектонски стилови који су на различити начин решавали проблеме везане за имплементацију и одржавање дистрибуираних система.

Архитектура базирана на микросервисима представља стил развоја дистрибуираних система као скупа малих, аутономних сервиса. Мали сервиси („микро") су једноставнији за одржавање, проширивање и скалирање. Микросервиси се развијају независно, обично и коришћењем различитих технологија што увећава хетерогеност и комплексност система. Тестирање, одржавања, разумевање и документовање микросервиса је, стога, доста теже и подложније грешкама.

Иако је уложено доста труда у развој алата који би олакшали развој микросервиса, још увек постоји низ проблема у креирању ових система. Језици и алати, данас у употреби, имају бар један од следећих недостатака: а) фокусирање на једну циљну платформу (нпр. подршка само за Java програмски језик), б) немогућност аутоматског генерисања документације, те в) немогућност евалуације моделованог система.

Узимајући у обзир наведене недостатке, циљ ове дисертације је олакшати и убрзати развој дистрибуираних система базираних на микросервисима.

У оквиру израде дисертације имплементиран је језик *Silvera*, базиран на *textX* мета-језику за развој језика специфичних за домен. *Silvera* језик одликују следеће карактеристике: а) једноставна синтакса, б) подршка за хетерогене технологије, в) аутоматско генерисање документације, те г) могућност евалуације архитектуре креираног система помоћу специјалних метрика.

У циљу евалуације језика, спроведена је анкета заснована на FQAD радном оквиру, где је од учесника тражено да имплементирају једноставан задатак

са и без коришћења *Silvera* језика, а потом да попуне упитник. Учесници анкете су задатак имплементирали ∼124% брже када су користили Silvera језик. Уз то, коришћење *Silvera* језика је довело и до квалитетнијих решења, са значајно мање грешака.

Нови Сад, 2023.

Ален Суљкановић

Abstract

The development of distributed systems is a challenging process that requires careful planning and implementation. Since their inception, distributed systems have evolved with the aim of increasing scalability, reliability, performance, and application availability, while reducing maintenance costs. Each of these goals brings challenges that need to be overcome to implement the system in the best possible way. Over time, various architectural styles have emerged, each addressing implementation and maintenance issues in different ways in distributed systems.

The microservices-based architecture represents a development style for distributed systems as a collection of small, autonomous services. Small services ("micro") are easier to maintain, extend, and scale. Microservices are developed independently, often using different technologies, which increases system heterogeneity and complexity. Therefore, testing, maintenance, understanding, and documenting microservices are more difficult and error-prone.

Although considerable effort has been invested in developing tools to facilitate microservice development, there are still a number of issues in creating these systems. The languages and tools currently in use have at least one of the following drawbacks: a) focusing on a single target platform (e.g., support only for the Java programming language), b) inability to automatically generate documentation, and c) inability to evaluate the modeled system.

Considering the mentioned drawbacks, the goal of this dissertation is to faciliate and expedite the development of microservices-based distributed systems.

As part of the dissertation, the *Silvera* language was implemented, based on the textX meta-language for developing DSLs. The Silvera language features the following characteristics: a) simple syntax, b) support for heterogeneous technologies, c) automatic documentation generation, and d) the ability to evaluate the architecture of the created system using special metrics.

To evaluate the language, a survey based on the FQAD framework was conducted, where participants were asked to implement a simple task with and without using the *Silvera* language, and then complete a questionnaire. Participants in the survey completed the task approximately ∼124% faster when using the Silvera language. Furthermore, the use of the *Silvera* language led to

higher quality solutions with significantly fewer errors.

Novi Sad, 2023.

Alen Suljkanović

Поглавље 1

Уводна разматрања

Живот је врло кратак и тескобан за оне који заборave прошлост, занемарују садашњост, и плаше се будућности.

Сенека

Од гломазних рачунара из 1940-тих до преносних рачунара данашњице, развој рачунарства је обликовао начин комуникације између људи, као и начин прикупљања информација. Рачунари су временом постајали не само мањи и моћнији, него и јефтинији. Резултат тога је чињеница да у 2021. години скоро пола домаћинстава у свету има бар један рачунар[1], док скоро 5 милијарди људи у свету поседује паметни телефон[2]. Међутим, нису само рачунари напредовали него и рачунарске мреже. Локалне мреже (енгл. *local-area networks - LAN*) данас могу да опслужују стотине машина унутар зграде тако да информације између машина могу да се пренесу у свега неколико микросекунди. Регионалне мреже (енгл. *wide-area networks - WAN*) омогућавају повезивање милиона машина где се информације преносе брзинама које досежу и неколико гигабита. Захваљујући еволуцији рачунарских мрежа, већина данашњих апликација су дистрибуиране [1].

Развој дистрибуираних система у последњих неколико деценија је знатно убрзан због потребе опслуживања великог броја корисника на мрежи. У свету брзог развоја софтвера, комерцијалне софтверске компаније како би опстале морају брзо одговорити на пословне изазове како би њихови производи

[1] https://www.statista.com/statistics/748551/worldwide-households-with-computer/
[2] https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/

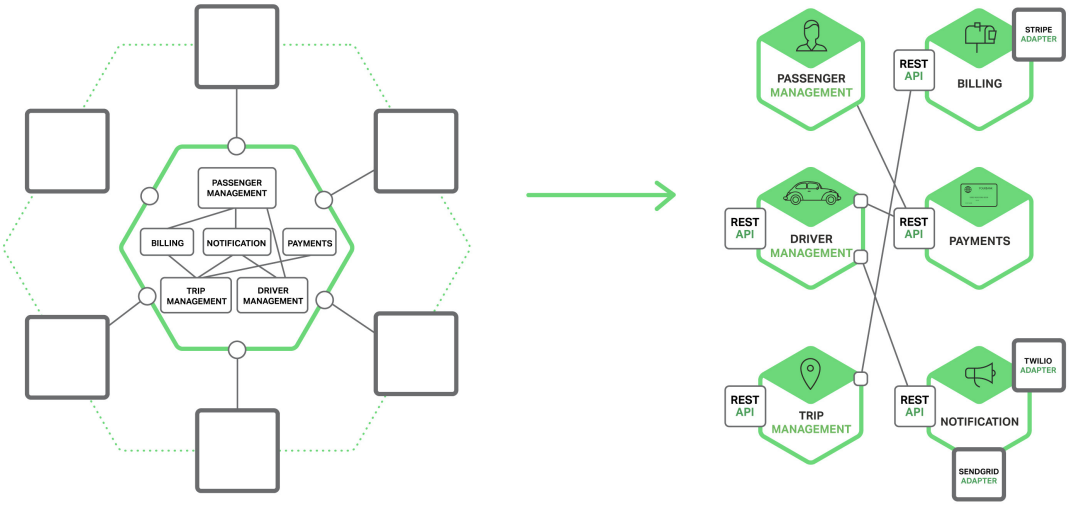
брзо доспели на тржиште [2]. Као резултат жеље за што бржом испоруком софтвера крајњим корисницима, појављује се *DevOps* покрет који за циљ има успостављање културе и окружења у ком се софтвер може развијати и испоручивати на брз и поуздан начин [3]. Кључни концепт унутар ДевОпс-а је континуална испорука (енгл. *continuous delivery - CD*) [4]. CD је приступ развоју софтвера у ком тимови у кратком року имплементирају софтверска решења, при чему та решења могу бити испоручена у сваком тренутку [5]. Међутим, имплементација CD приступа може бити веома изазовна, при чему архитектура софтвера представља кључну баријеру [3].

Архитектура базирана на микросервисима (видети Секцију 1.2.9) представља стил развоја апликације као скупа малих сервиса, при чему се сваки од тих сервиса извршава унутар посебног процеса и комуницира са осталим сервисима путем једноставних механизама (енгл. *lightweight*), као што је на пример *Hypertext Transfer Protocol* (HTTP) [6]. Сваки од ових сервиса је потпуно аутономан што омогућава да се они независно развијају и покрећу. Централизован начин управљања код архитектура базираних на микросервисима је сведен на минимум, јер се чак и управљачка логика развија као независан сервис.

Овакав приступ представља тоталну супротност тзв. монолитној архитектури (видети Секцију 1.2.7), где се апликација развија као једна извршна целина иако она може да се састоји од низа других сервиса, компоненти, итд. У случају једноставнијих апликација, овај приступ је у предности у односу на микросервисе [7]. Међутим, са растом апликације долази и до проблема. Убацивање новог или измена тренутног програмског кода постаје све тежа јер често долази до недоумице у ком делу програмског кода апликације треба извршити измену. Упркос тежњи ка модуларности, границе између модула постају све тање што доводи до тога да се програмски код везан за одређену функционалност распрши по разним модулима апликације [8]. Разлика између ове две врсте архитектура је илустрована на Слици 1.1.

1.1 Предмет истраживања

Истраживање је усмерено ка проблему недостатка софтверских алата који би олакшали развој, увођење, управљање и надгледање микросервиса. На недостатак софтверских алата овог типа указују студије [10] и [11]. Поред тога, развој дистрибуираних апликација базираних на микросервисима окарактерисан је писањем велике количине шаблонског кода како би се успоставила потребна инфраструктура, уместо да се имплементира пословна логика [12].



Слика 1.1: Разлика између монолитне (*лево*) и архитектуре базиране на микросервисима (*десно*) [9]

Истраживање се састоји од следећих целина: (1) проучавање дизајн образаца из области архитектура базираних на микросервисима, (2) дизајнирање и имплементација језика специфичног за домен (ЈСД) за опис архитектура базираних на микросервисима, (3) дизајн и имплементација радног оквира за евалуацију архитектура описаних помоћу поменутог језика, и (4) дизајн и имплементација радног оквира за генерисање костура апликације. Данас постоји неколико алата за моделовање апликација у виду микросервиса, међутим, њихова примена је ограничена на један вид комуникације између микросервиса, као и један начин увођења.

1.2 Појмовни оквир рада

У овом поглављу биће уведени основни појмови битни за разумевање остатка дисертације. Пре свега, ради се о базичним дефиницијама дистрибуираних система, моделовања специфичног за домен, као и појму језика специфичних за домен.

1.2.1 Дистрибуирани системи

Постоји неколико дефиниција дистрибуираних система.

Таненбаум и Ван Стен за дистрибуирани систем кажу да је *колекција независних рачунара који својим корисницима изгледају као један кохерентан систем* [13]. Кулорис дефинише дистрибуирани систем као *систем у коме хардверске или софтверске компоненте смештене на умреженим рачунарима комуницирају и координирају своје акције путем порука* [14].

Бирман дефинише дистрибуирани систем као *скуп рачунарских програма који се извршавају на једном или више рачунара и координишу операције разменом порука* [15].

На основу ових дефиниција можемо закључити да се дистрибуирани систем састоји од аутономних компоненти које сарадњом извршавају одређене операције. Рачунарска мрежа је саставни део већине дистрибуираних система, те служи за пренос порука између компоненти система. Међутим, коришћење рачунарске мреже није предуслов за имплементацију дистрибуираног система. Рачунарски програм где процеси размењују информације преко порука такође је дистрибуирани систем [15].

Основне карактеристике дистрибуираних система су следеће:

- Компоненте унутар система функционишу конкурентно [14].
- Организација компоненти унутар дистрибуираног система, њихове разлике, и начин на који оне комуницирају су већином скривене од крајњег корисника [13].
- Концепт заједничког, глобалног, сата (енгл. *global clock*) не постоји. Унутар дистрибуираног система, свака компонента поседује свој сат. Сатови се усклађују како би били што конзистентнији, међутим није их могуће потпуно ускладити. Као последица овог, код дистрибуираних система понекад није могуће утврдити који од два догађаја се догодио први [16].

Дистрибуирани системи су своју примену нашли скоро у свим сферама наших живота. Користе се успешно за Интернет претраживање (Google, Yahoo, Bing), е-трговину (Amazon, eBay), забаву (YouTube, Steam), образовање (Dualingo), превоз и логистику (GPS, Google Maps, HereWeGo). Данашњи аутомобили су „дистрибуирани системи на точковима”, а још комплекснији пример представљају авиони [17].

Главни разлог употребе дистрибуираних система је да клијентима олакшају *приступ удаљеним ресурсима*, као и контролисан начин дељења тих ресурса [13]. Ресурс може бити било шта. Типични примери дељених ресурса су штампачи, датотеке, и медијуми за складиштење података. Дељење ресурса попут штампача или супер рачунара умањује оперативне трошкове

компаније, али поред тога олакшава сарадњу и размену информација. Лак приступ удаљеним ресурсима довео је до појаве организација у којима групе људи са различитих географских локација сарађују коришћењем софтвера за радне тимове (енгл. *groupware*) [13]. Типичан пример ове врсте софтвера су софтвери за телеконференције.

1.2.1.1 Изазови

Транспарентност. За дистрибуирани систем кажемо да је транспарентан уколико је његово понашање еквивалентно понашању централизованог система [18], односно уколико својим клијентима даје привид да врше интеракцију само са једним рачунарским системом [13]. Концепт транспарентности се може применити на неколико аспеката дистрибуираног система, при чему су најважнији приказани у Табели 1.1.

Табела 1.1: Различити аспекти транспарентности код дистрибуираних система [13].

Аспект транспаретности	**Опис**
Приступ (енгл. *Access*)	Сакрива разлике у репрезентацији података, те начин на ком се приступа дељеним ресурсима
Локација (енгл. *Location*)	Сакрива информацију о локацији дељеног ресурса
Миграција (енгл. *Migration*)	Сакрива чињеницу да ресурс може бити измештен на другу локацију
Пресељење (енгл. *Relocation*)	Сакрива чињеницу да ресурс може бити измештен на другу локацију у току коришћења
Репликација (енгл. *Replication*)	Сакрива чињеницу да је ресурс реплициран
Конкурентност (енгл. *Concurrency*)	Сакрива чињеницу да неколико конкуретних клијената деле ресурс
Отказ (енгл. *Failure*)	Сакрива процес отказивања и опоравка

Транспарентност приступа сакрива разлике у репрезентацији података и начин на који клијенти приступају дељеним ресурсима [13]. Главни циљ код транспарентности приступа је сакрити детаље везане за одређени оперативни систем или машину. Тако клијент може да манипулише са подацима на исти начин, независно од платформе на ком се ти подаци налазе.

Код *транспарентности локације*, клијент не поседује знање о физичкој локацији дељеног ресурса [13]. Код ове врсте транспарентности, именовање дељеног ресурса игра веома важну улогу. Транспарентност локације се врло лако постиже коришћењем логичког именовања дељених ресурса. Пример таквог имена може бити URL `http://ftn.uns.ac.rs/index.html`, који не поседује информацију о физичкој локацији сервера Факултета техничких наука, нити о локацији датотеке *index.html*. Поред тога, поменути URL не одаје информацију о томе да ли је датотека *index.html* увек била на тренутном месту или је ту премештена. За дистрибуирани систем код којих дељени ресурс може да промени своју локацију, без утицаја на начин на којима се приступа том ресурсу, кажемо да обезбеђује *транспарентност миграције* [13]. За дистрибуирани систем код ког дељени ресурс може да промени

своју локацију у време коришћења, без утицаја на рад клијента, кажемо да обезбеђује *транспарентност пресељења* [13].

Транспарентност репликације сакрива информацију о томе да у систему постоји неколико копија одређеног дељеног ресурса [13]. Да би се успешно имплементирала ова врста транспарентности, потребно је да све реплике поседују исти назив. Дистрибуирани систем који подржава транспарентност репликације би требао да у исто време подржава и транспарентност локације [13].

За дистрибуирани систем код ког клијенти могу истовремено користити исти дељени ресурс, без да су свесни да приступају истом ресурсу у исто време, кажемо да обезбеђује *транспарентност* конкурентности [13]. Код ове врсте транспарентности, веома је битно обезбедити да овакав начин приступа дељеном ресурсу увек остави тај ресурс у конзистентном стању. Конзистентност се може постићи помоћу технике закључавања (енгл. *locking*) или коришћењем трансакција.

За дистрибуирани систем код ког клијенти не могу да примете да је систем ушао у стање отказа и да се из тог стања успешно опоравио кажемо да обезбеђује *транспарентност отказа* [13]. Откази су неизбежни код сваког софтверског система, и могу потицати како од хардвера тако и од софтвера. У случају отказа, програми или престају да раде или дају нетачне резултате. Код дистрибуираних система, откази су парцијални, што значи да одређени делови система настављају са радом док су други делови система "под отказом" [14]. Постоји неколико техника за руковање отказима. У неким ситуацијама је могуће детектовати отказ (на пример, коришћењем контролног збира (енгл. *checksum*)), док је у другим ситуацијама то немогуће. Откази се могу и маскирати, што се најчешће ради у ситуацијама када је потребно поново послати захтев ка удаљеном ресурсу без акције клијента [14]. Маскирање отказа представља једну од ствари коју је код дистрибуираних система најтеже имплементирати јер често не можемо направити разлику између ресурса који једноставно споро даје одговор од ресурса који не ради уопште [13]. Откази могу имати директан утицај на доступност система (енгл. *availability.*). Доступност системима се описује процентима (Табела 1.2).

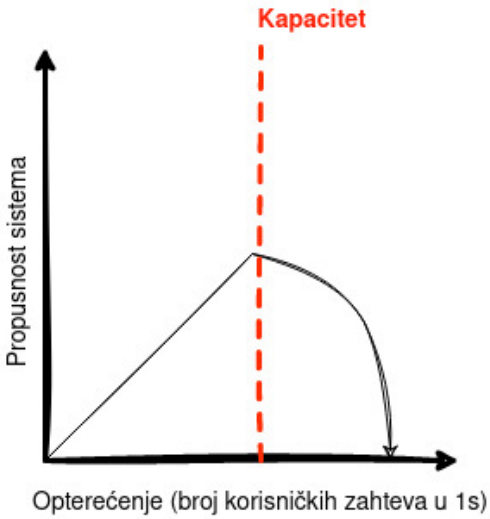
Иако је транспарентност пожељна, у одређеним ситуацијама није практична. Често транспарентност као последицу има пад перформанси. Типичан пример је успостављање конекције са сервером који не ради [13]. У случају транспаретности отказа, клијент би непрестано покушавао успоставити везу са сервером тиме утичући на перформансе читавог система [13]. У том случају, боље решење је одмах пријавити грешку кориснику. Такође, операција ажурирања дељеног ресурса ће видно трајати дуже уколико у систему постоји неколико реплика тог ресурса [13].

Табела 1.2: Однос доступности дистрибуираног система и периода неисправности у току дана. 99.9% се сматра прихватљивом доступношћу. Системи са процентом доступности изнад 99.99% спадају у категорију веома доступних система [19].

Доступност	**Период неисправности у току дана**
90%	2.40 сата
99%	14.40 минута
99.9%	1.44 минута
99.99%	8.64 секунде
99.999%	864 милисекунде

Отвореност (енгл. *openness*). Отвореност дистрибуираног система је одређена степеном доступности његових кључних интерфејса различитим клијентским програмима [13, 14]. Често се интерфејси дистрибуираних система описују посебним језицима за опис интерфејса — *Interface Definition Languages* (IDLs). IDL прецизно дефинише називе доступних фукција, очекиване параметре и њихове типове, тип повратне вредности, те могуће изузетке [13]. Спецификација интерфејса треба да буде потпуна и неутрална. Потпуна спецификација поседује све информације потребне за имплементацију, док неутрална спецификација не прописује како конкретно имплементација треба да изгледа [13]. Потпуност и неутралност су веома битне за интероперабилност (енгл. *interoperability*) и преносивост (енгл. *portability*) [20]. Интероперабилност је способност да две или више компоненти дистрибуираног система, потенцијално развијене од различитих произвођача, имају могућност размене и тумачења захтева те повезаних скупова података [21]. За компоненту кажемо да је преносива уколико може да се из једног система пребаци у други, и да ради без модификација. Стога, отворен систем је такође и лако проширив јер је могуће лако додавати нове компоненте или мењати постојеће.

Скалабилност (енгл. *scalability*). Скалабилност система се може мерити дуж најмање три димензије [22]: нумеричка, географска, и административна. Нумеричку димензију чини број корисника система и број обухваћених објеката и услуга. Географска димензија се састоји од удаљености које систем покрива. Административна димензија се састоји од броја организација које врше контролу над деловима система. Дизајн скалабилног дистрибуираног система са собом носи многе изазове попут [14]: цене проширивања система са новим ресурсима, руковање перформансама система, избегавање уских грла, и сл. Скалабилност може директно да утиче на перформансе система. Перформансе дистрибуираног система су резултат пропустности



Слика 1.2: Приказ односа пропустности и оптерећења система [19]

(енгл. *throughput*) и времена одзива (енгл. *response time*) [19]. Пропустност представља број операција које систем може да процесуира у једној секунди, док је време одзива утрошено време између захтева и одзива [19]. Повећањем оптерећења, систем достиже свој *капацитет* — максимално оптерећење које систем може да издржи. У том случају, могуће је нагло опадање перформанси система (Слика 1.2). Скалабилан дистрибуиран систем може да повећава свој капацитет са повећањем оптерећења. Најједноставнији начин да се то постигне је *вертикално скалирање* (енгл. *vertical scaling или scaling up*), односно, куповина и инсталација јачег хардвера. Међутим, временом систем поново може да досегне врхунац својих перформанси. Алтернатива овој врсти скалирања је *хоризонтално скалирање* (енгл. *horizontal scaling или scaling out*) где се врши расподела оптерећења на више делова система. Постоје три врсте хоризонталног скалирања: функционална декомпозиција (систем се разлаже на више индивидуалних делова), партиционисање (расподела скупа података на неколико независних подсистема), и дупликација (креирање више инстанци система).

Безбедност. Безбедност информационих ресурса има три компоненте: поверљивост (заштита од откривања неовлашћеним лицима), интегритет (заштита од измене или корупције) и доступност (заштита од ометања средстава за приступ ресурсима) [14]. Приликом решавања проблема безбедности система, потребно је дефинисати безбедносну политику (енгл. *security policy*) као и безбедносне механизме (енгл. *security mechanisms*).

Безбедносна политика прецизно описује које акције су ентитетима дозвољене, а које су забрањене [13]. Ентитети могу бити корисници, услуге, подаци, машине и тако даље [13]. Безбедносни механизми спроводе безбедносне политике. Неки од најзначајних безбедносних механизама су енкрипција, аутентификација, ауторизација, и ревизија. Задатак енкрипције је да трансформише податке у облик неразумљив нападачу, док аутентификација служи за верификацију идентитета клијента [13]. Ауторизација је провера да ли аутентификовани клијент има дозволу да приступи одређеним подацима, док се ревизијом прати који клијенти су приступали којим подацима и на који начин [13].

1.2.2 Архитектуре дистрибуираних система

Дистрибуирани системи су често веома комплексни, те њихови делови могу бити распоређени на више различитих машина. Како бисмо савладали њихову комплексност, овакве системе је потребно организовати на прави начин. Управо то је задатак софтверске архитектуре. Софтверска архитектура је концепт који дефинише начин на који су делови дистрибуираног система повезани, како ти делови размењују поруке и у ком формату, те како су делови конфигурисани тако да чине један јединствен систем [13].

Софтверска архитектура представља опис софтверског решења на високом нивоу. Како би овај опис био што потпунији, потребно је систем сагледати из више различитих перспектива при чему се свакака од њих фокусира на другу врсту детаља у зависности од циљева различитих заинтересованих страна: крајњих корисника, програмера, менаџера, и сл. Модел „4+1” [23] дефинише следећа четири погледа на софтверску архитектуру:

- Логички поглед (енгл. *Logical view*) - фокусира се на функционалне захтеве, тј. описује фукционалности које систем треба да пружи крајњем кориснику. У случају објектно-оријентисаних језика, систем се обично описује помоћу дијаграма класа. Дијаграм класа дефинише класе и везе између њих, као што су: асоцијација, композиција, наслеђивање, итд. Уколико се не користи објектно-оријетисани приступ, систем се може описати помоћу E-R (*Entity-relationship*) дијаграма.

- Поглед из угла процеса (енгл. *Process view*) - узима у обзир и неке нефункционалне захтеве, попут перформанси и доступности. Бави се питањима конкуретности и дистрибуције, интегритета система, отпорности на отказе, те који процес контролише елементе дефинисане унутар логичког погледа. Процес представља групу задатака (енгл. *task*)

који формирају једну извршиву целину. Процеси се могу реплицирати како би се побољшала доступност, или како би се смањило оптерећење система. Задатак представља додатни ниво гранулације, и извржва се унутар једног процеса. Задаци могу бити већи (енгл. *major*) или мањи (енгл. *minor*). Код овог погледа, везе између процеса описују начин комуникације између процеса.

- Физички поглед (енгл. *Physical view*) - описује како се елементи попут мреже, процеса, задатака, и објеката мапирају на различите хардверске компоненте. Постоји могућност креирања различитих конфигурација, при чему се типично креирају посебне конфигурације за развој и за тестирање, па чак и за увођење уколико оно зависи од потреба крајњих корисника. Стога, мапирање софтвера на хардверске компоненте мора бити веома флексибилно те мора имати минималан утицај на сам кôд.

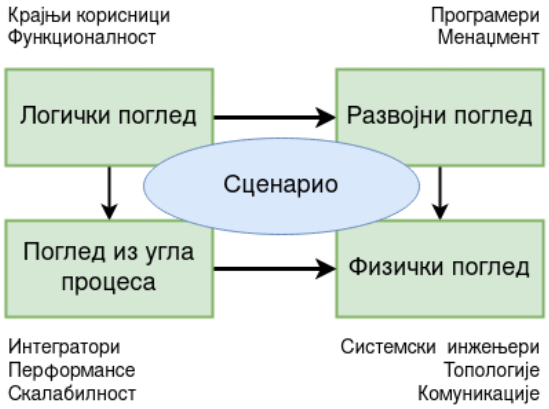
- Развојни поглед (енгл. *Development view*) - фокусира се на декомпозицију система на библиотеке или подсистеме. Сваки подсистем може имати један или више слојева, при чему сваки слој поседује узак и добро дефинисан интерфејс према слоју који се налази хијерархијски изнад њега. Овај поглед се представља путем дијаграма модула или подсистема, те помоћу „увоз" (енгл. *import*) и „извоз" (енгл. *export*) веза.

Везивно ткиво између погледа код „4+1" модела, односно „+1" део, представља сценарио. Сваки сценарио дефинише како различите компоненте унутар одређеног погледа сарађују како би извршиле задатак. На пример, сценарио унутар логичког погледа показује начин сарадње између класа. Модел 4+1 је приказан на Слици 1.3.

Зашто су софтврерске архитектуре важне? Захтеви који се постављају приликом креирања софтверских апликација се могу сврстати у две категорије. Прва категорија су *функционални* захтеви, који дефинишу шта апликација мора да ради. Ови захтеви се обично дефинишу у облику случајева коришћења (енгл. *use case*), те као такви се не описују помоћу софтверске архитектуре. Друга катагорија су захтеви који обезбеђују *квалитет услуге* (енгл. *quality of service*). Ови захтеви могу дефинисати квалитете попут скалабилности, поузданости, одрживости, и сл. Врста софтверске архитектуре има директан утицај на начин на који ће софтвер испунити ове захтеве [24].

Као најчешће коришћене софтверске архитектуре у области дистрибуираних система можемо издвојити следеће [13, 14]:

- Слојевита архитектура (енгл. *layered architecture*) - дистрибуирани систем се састоји од неколико слојева, при чему сваки има тачно



Слика 1.3: Модел „4+1" описује софтверску архитектуру помоћу четири погледа и сценарија који описује како елементи унутар сваког погледа сарађују како би извршили задатак.

одређену одговорност (Секција 1.2.5). Слојеви су организовани хијерархијски, и сваки слој комуницира само са слојем који је директно испод њега у хијерархији. Предност слојевите архитектуре је у томе што је могуће мењати слојеве независно, без утицаја на остатак система. Слојевита архитектура може да се користи у комбинацији са осталим архитектонским стиловима;

- Хексагонална архитектура (енгл. *Hexagonal architecture*) - стил за дизајн софтверских апликација у ком је пословна логика центар апликације, док су све ванске зависности „везане" за центар помоћу адаптера (Секција 1.2.6). Хексагонална архитектура промовише добру организацију кода, јасно дефнисане улазе и излазе, и добро изоловане адаптере.

- Монолитна архитектура (енгл. *monolith architecture*) - традиционалан стил за дизајн софверских апликација. Међутим, због његових недостатака, појавили су се нови стилови развоја дистрибуираних система (Секција 1.2.7);

- Архитектура базирана на објектима (енгл. *object-based architecture*) - сваки део дистрибуираног система је представљен одговарајућим објектом [13]. За разлику од слојевитих архитектура, ток захтева и одговора не прати тачно дефинисану хијерархијску структуру;

- Архитектура усредсређена на податке (енгл. *Data-centered Architecture - DCA*) - делови дистрибуираног система комуницирају преко заједничког

репозиторијума [13]. Репозиторијум може бити активан или пасиван. Понекад, овај заједнички репозиторијум може бити и база података. Овај тип архитектура се користи код система где су потребне високе перформансе и конзистенција података;

- Архитектура базирана на догађајима (енгл. *Event-driven Architecture - EDA*) - делови дистрибуираног система комуницирају пропагацијом догађаја (догађај опционо са собом носи и додатне податке). Уместо да комуницирају директно, делови дистрибуираног система објављују и реагују на догађаје. Овај тип архитектура се користи у системима који захтевају високу скалабилност и мало кашњење;

- Сервис-оријентисана архитектура (СОА, енгл. *Service Oriented Architecture - SOA*) - дистрибуирани систем се састоји од независних сервиса које комуницирају преко мреже (Секција 1.2.8). СОА обезбеђује висок степен флексибилности и скалабилности, али такође повећава и сложеност система.

- Архитектура базирана на микросервисима (МСА, енгл. *Microservice Architecture - MSA*) - дистрибуирани систем се састоји од малих, аутономних сервиса који комуницирају преко мреже коришћењем једноставних механизама (Секција 1.2.9). МСА обезбеђује висок степен одрживости и скалабилности, али такође повећава и сложеност система.

Избор софтверске архитектуре зависи од потреба система који се имплементира. Такође, многе од наведених архитектура могу да се користе заједно. EDA се често користи у комбинацији са МСА јер омогућава асинхрону комуникацију између микросервиса. ДЦА се често комбинује са МСА и СОА како би обезбедила заједнички репозиторијум за размену података између сервиса. Слојевита и хексагонална архитектура могу да се комбинују са свим наведеним архитектурама како би кôд био лакши за одржавање, разумљивији и флексибилнији.

1.2.3 Протоколи за комуникацију

Важно питање код дизајнирања дистрибуираних система је како ће комуницирати делови система који се често налазе на различитим машинама. Најчешће коришћени протоколи за комуникацију су протоколи базирани на порукама (енгл. *messaging*) и позив удаљене процедуре (енгл. *remote procedure call - RPC*) [25].

Протокол базиран на порукама омогућује деловима система да размењују поруке преко две апстрактне операције: *пошаљи* (енгл. *send*), и *прими*

(енгл. *receive*). Приликом имплементације овог начина комуницје потребно је донети следеће одлуке [26]:

- комуникација „један на један" (енгл. *one-to-one*) или „један на више" (енгл. *one-to-many*),
- синхрона или асинхрона комуникација,
- једносмерна (енгл. *one-way*) или двосмерна (енгл. *two-way*) комуникација,
- директна или индиректна комуникација,
- аутоматско или експлицитно бафероване (енгл. *buffering*),
- имплицитан или експлицитан пријем.

Комуникација „један на један" омогућава размену поруке између тачно два дела дистрибуираног система, док комуникација „један на више" омогућава ефикасно емитовање (енгл. *broadcast*) порука које могу имати више пријемника. Код синхроне комуникације, пошиљалац (енгл. *sender*) шаље поруку примаоцу (енгл. *receiver*) и чека на његов одговор. Код асинхроне комуникације, пошиљалац не чека на одговор након слања поруке. Једносмерна комуникација је линеарна и ограничена, јер се одвија само у смеру пошиљалац-прималац. Двосмерна комуникација подразумева и повратну информацију (енгл. *feedback*) од примаоца ка пошиљаоцу и омогућава пошиљаоцу да зна да је порука тачно примљена. Код дирекне комуникације, пошиљалац шаље поруку примаоцу директно, док се та порука у случају индирекне комуникације шаље преко посредника (енгл. *intermediate*). Код експлицитног бафероване, бафер заузима тачно дефинисану количину меморијског простора. Уколико пошиљалац пошаље поруку, а при томе је бафер пун, биће присиљен да чека да се ослободи довољна количина меморије како би се његова порука обрадила. Код аутоматског бафероване ово није случај, јер величина бафера није фиксна, те пошиљалац није приморан на чекање. Експлицитан пријем подразумева пријем поруке од тачно одређеног извора поруке (пошиљаоца), док код имплицитног пријема пошиљалац не мора бити познат унапред.

Семантика код позива удаљене процедуре је иста као и код позива локалне процедуре: *клијент* позива процедуру и чека на њено извршење. Комуникација позивом удаљене процедуре подразумева постојање везивног кôда на клијентској (енгл. *client stub*) и серверској (енгл. *server stub*) страни. Везивни кôд позиваоцу делује као локални, али уместо извршавања

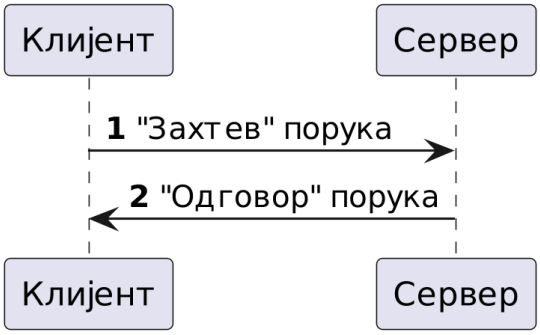
процедуре, овај кôд је задужен за серијализацију и десеријализацију позива процедуре и њено слање преко мреже. Везивни кôд није ништа друго него интерфејс клијентске и серверске апликације. Ови интерфејси су углавном доступни у истом програмском језику у ком су писане клијентске или серверске апликације. Међутим, интерфејс се може дефинисати и помоћу IDL-а. Коришћење IDL-ова драстично поједностављује апликације базиране на позиву удаљене процедуре [13].

Позив удаљене процедуре је по природи синхрон начин комуницире, међутим, постоји и асинхрона варијанта овог начина комуникације.

1.2.4 Комуникациони обрасци

Посебан изазов при имплементацији комуникације представља координација порука на такав начин да се акције везане за поруке извршавају у тачно одређеном низу. Обрасци за размену порука (енгл. *Message exchange patterns - MEPs*) су обрасци у којима је редослед размене порука унапред дефинисан [27].

Најједноставнији, и најучесталији, образац за размену порука је *Захтев-одговор* (енгл. *request-response*) [27](Слика 1.4). Овај образац обезбеђује размену порука на такав начин да се прво шаље порука од захтеваоца до даваоца услуге. Затим, након пријема поруке, даваоц шаље одговор захтеваоцу. При томе, одговор у себи носи информацију о кореспондирајућем захтеву. Образац *Захтев-одговор* се може и за синхрони и за асинхрони стил комуникације.



Слика 1.4: *Захтев-одговор* образац за размену порука. Клијент потражује услугу од сервера тако што пошаље захтев, те чека на одговор.

За имплементацију једносмерног преноса порука служи *Пошаљи-и-заборави* (енгл. *Fire-and-forget*) образац (Слика 1.5). Овај образац се искључиво користи за асинхрони стил комуникације где се не очекује одговор,

и постоје бројне варијанте овог обрасца [27]: а) слање поруке на само једно одредиште (*single destination* образац), б) слање поруке на више одредишта (*multi-cast* и *broadcast* обрасци).

Слика 1.5: *Пошаљи-и-заборави* образац за размену порука.

Ова два обрасца, *Захтев-одговор* и *Пошаљи-и-заборави*, припадају групи „примитивних" образаца, на основу којих се изграђују комплекснији обрасци [27]. Постоји неколико комплексних комуникационих образаца, укључујући:

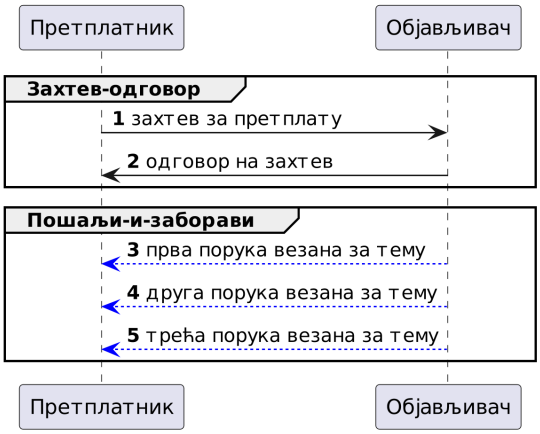
- Образац *објаве и претплате* (енгл. *publish-subscribe*) на поруке. Овај образац уводи нове улоге за сервисе током размене порука: објављивачи (енгл. *publishers*) порука, и примаоци или претплатници (енгл. *subscribers*) [27]. Начин на који овај образац комбинује примитивне обрасце *Захтев-одговор* и *Пошаљи-и-заборави* је следећи: Претплатник шаље *захтев* објављивачу да жели да прима поруке везане за одређену тему, након чега објављивач враћа *одговор* претплатнику везан за његов захтев. Затим, по доступности порука везаних за одабрану тему, објављивач емитује поруке свим претплатницима на ту тему (Слика 1.6). Коришћењем овог обрасца избегавамо чврсту спрегу између пошиљаоца и примаоца.

- *Event-driven* је варијанта обрасца *објаве и претплате* где се, у случају неког догађаја, порука шаље директно заинтересованим примаоцима уместо да се емитује свима.

- *Репродуковање у реалном времену* (енгл. *streaming*) укључује континуално слање података (попут звука или видеа) једном или више прималаца. Примаоци код овог обрасца могу да обрађују податке како пристижу, уместо да чекају да стигну сви подаци. Овај образац често захтева више ресурса него други обрасци комуникације, јер се подаци преносе у реалном времену [14].

- *Дељено стање* (енгл. *shared state*) је комуникациони образац где делови дистрибуираног система комуницирају тако што симултано пишу или

читају са дељене локације за складиштење података [14]. Овај образац се често користи за имплементацију дистрибуираних база података или дистрибуираних система за кеширање.



Слика 1.6: Образац *објаве и претплате* на поруке.

Сваки од ових образаца има своје предности и мане,а избор обрасца који ће се користити зависи од спецификчних захтева апликације.

1.2.5 Слојевита софтверска архитектура

Слојевита софтверске архитектура организује елементе софтвера у посебне слојеве. Сваки слој је задужен за извршавање тачно дефинисаних задатака. Слојеви су организовани хијерархијски, при чему одређени слој зависи од слојева који су нижи у хијерархији. Ова архитектура имплементира логички поглед на архитектуру система.

Најчешће коришћена архитектура из ове категорије јесте трослојна архитектура (енгл. *three-tier architecture*), која слојеве организује у следеће слојеве:

- Презентациони слој (енгл. *Presentation layer*) - садржи кôд који имплементира кориснички интерфејс или екстерни API,
- Слој пословне логике (енгл. *Business logic layer*) - садржи кôд који имплементира пословну логику, и
- Слој за перзистенцију (енгл. *Persistence layer*) - садржи кôд који имплементира интеракцију са базом података.

Највећа мана ових врста архитектура је стварање међузависти између слојева. Приликом компајлирања, виши слојеви зависе од нижих. Тако имамо да презентациони слој зависи од слоја пословне логике, који зависи од слоја за презентацију. То значи да пословна логика, која је обично најважнији део апликације, зависи од детаља имплементације слоја за перзистенцију (и често од постојања базе података) [28]. Тестирање пословне логике код слојевитих архитектура често је тешко и захтева тестну базу података. Слој за перзистенцију се мења често. Историјски, у индустрији се овај слој мења најмање једном на сваке три године [29]. Системи код којих међузависности спречавају ажурирање делова система обично постану застарели, те их је потребно реимплементирати [29].

1.2.6 Хексагонална софтверска архитектура

Хексагонална архитектура [30] је софтверска архитектура где је пословна логика у центру апликације. Ова архитектура имплементира логички поглед на архитектуру система. Хексагонална архитектура је илустрована на Слици 1.7.

Три главне компоненте хексагоналне архитектуре су [24, 30]:

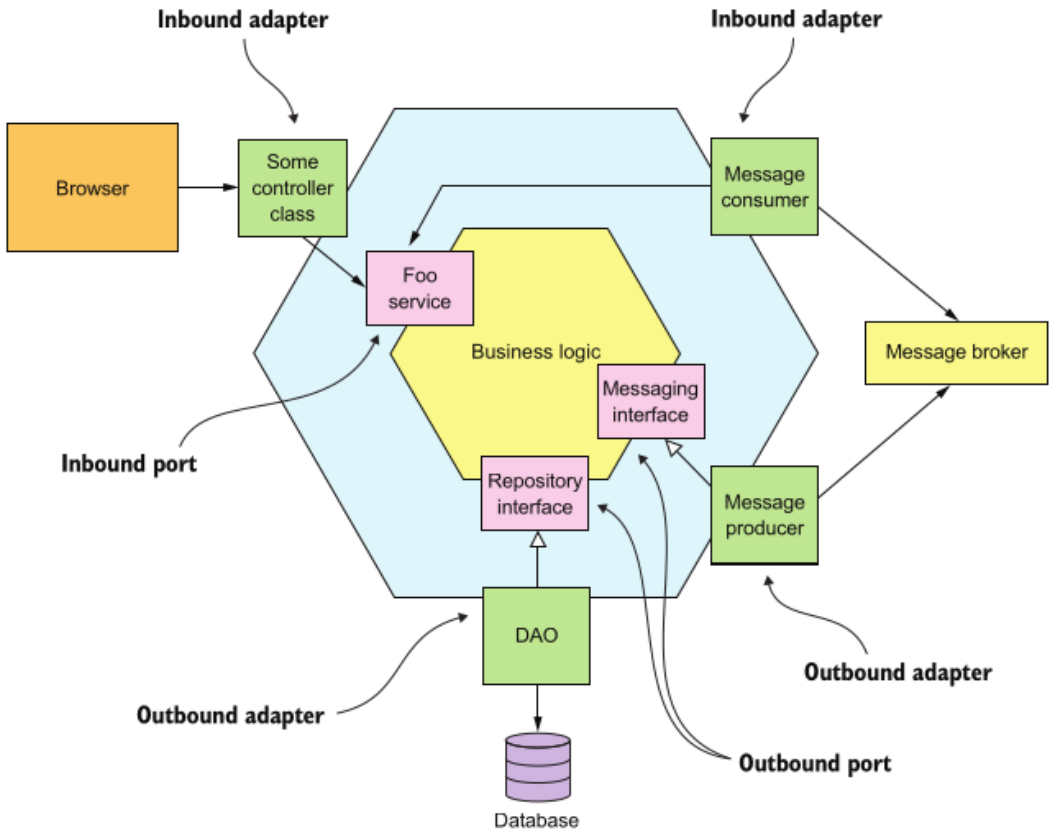
- *Доменски модел* - представља централну пословну логику система и доменске ентитете. Имплементира главну функционалност и правила на основу којих је израђен остатак апликације. Доменски модел мора бити независан од ванских делова, попут базе података, корисничког интерфејса, и сл.

- *Портови* - представљају интерфејсе који дефинишу начин комуникације између пословне логике и ванског света. Постоје две врсте портова: а) примарни или улазни портови (енгл. *primary*, *driving*, или *inbound ports*), који дефинишу операције које пословна логика пружа ванском свету (нпр. кориснички интерфејс), и б) секундарни или излазни портови (енгл. *secondary*, *driven*, или *outbound ports*), који како пословна логика позива методе из ванског света (нпр. интеракција са базом података).

- *Адаптери* - представљају компоненте које имплементирају портове и тиме обезбеђују интеракцију ванског света и пословне логике. Постоје две врсте адаптера: а) примарни или улазни (енгл. *primary*, *driving*, или *inbound adapters*), који обрађују захтеве из ванског света и позивају улазне портове, и б) секундарни или излазни енгл. *secondary*, *driven*, или *outbound adapters*), који имплементирају излазне портове те позивају методе из екстерних апликација.

Највећа предност хексагоналне архитектуре је у томе што раздваја пословну логику од презентационе и логике руковања са подацима, што значи да се пословна логика може тестирати независно од остатка система. Поред тога, ове архитектуре прецизније описују модерне апликације.

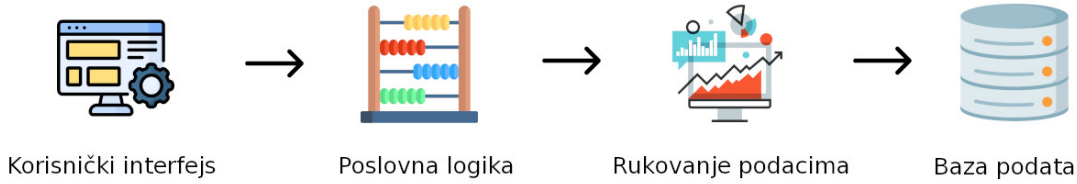
Хексагонална софтверска архитектура је још позната под именима *Ports and Adapters* [30], *Onion* [29], *Clean Architecture* [31].



Слика 1.7: Пример хексагоналне архитектуре, која се састоји од пословне логике и једног или више адаптера који комуницирају са спољашњим системима [24]. Пословна логика има један или више портова. Улазни адаптери, који обрађују захтеве од спољних система, позивају улазне портове. Излазни адаптери имплементирају излазне портове, те позивају спољне системе.

1.2.7 Монолитне софтверске архитектуре

Монолитне софтверске архитектуре (МА) представљају традиционалан модел за дизајн софтверских апликација. Резултат овог приступа развоја софтвера су софтверске апликације које су самосталне (енгл. *self-contained*) и независне од осталих апликација. Монолитна апликација се развија као једна извршна целина, иако она може да се састоји од низа других компоненти. МА се често комбинује са слојевитим (Слика 1.8) или хексагоналним софтверским архитектурама како би се систем логички организовао.



Слика 1.8: Приказ монолитне софтверске апликације по слојевима.

У случају једноставнијих апликација, монолитне софтверске архитектуре представљају добар избор [7]. Програмски кôд код монолитних апликација се обично налази у једном репозиторијуму, што олакшава увођење, управљање конфигурацијама, те праћење перформанси апликације. Међутим, у случају сложенијих апликација долази до неколико проблема:

- велики монолити су тешки за одржавање и надограђивање због своје комплексности [32]. Лоцирање грешака унутар велике кодне базе монолита захтева доста времена [33];
- монолити такође имају проблем са међузависностима (енгл. *dependency hell*) [34], због чега додавање или ажурирање библиотека може резултовати неконзистентним радом система;
- било каква измена у делу монолита захтева поновно покретање читаве апликације. За велике системе, поновно покретање обично изазива значајне застоје у раду (енгл. *downtime*), успорава развој, тестирање, и одржавање система;
- увођење (енгл. *deployment*) монолитних апликација је обично субоптимално због конфликтних захтева над ресурсима које имају њени модули: неки модули могу да захтевају више меморије, док други могу да троше више процесорске снаге, итд. Приликом одабира окружења за увођење, потребно је направити компромис узевши конфигурацију која

задовољава све захтеве, што може бити или скупо или субоптимално имајући у виду захтеве индивидуалних модула [33];

- монолити ограничавају могућност скалирања. Уобичајена стратегија за руковање са повећаним бројем захтева је креирање нових инстанци исте апликације те расподела оптерећења (енгл. *load balancing*) [33]. Међутим, проблем настаје када увећани саобраћај утиче на рад малог подскупа апликације, што изазива доста неподесно алоцирање ресурса за остале делове апликације [32];

- монолити уводе ограничења у виду коришћених технологија, те се због тога у већини случајева користи само један програмски језик и развојна окружења везана за тај језик [33].

Алтернатива монолитним софтверских архитектурама су архитектуре које фаворизују декомпозицију на једноставније компоненте или модуле. Резултат овог начина развоја софвера су апликације које су прилагођене итеративном развоју, лакше су за одржавање, нуде већу хетерогеност у избору технологија, те омогућавају лакше скалирање. Неке од архитектура овог типа су сервис-оријентисане архитектуре (Секција 1.2.8), те архитектуре базиране на микросервисима (Секција 1.2.9).

1.2.8 Сервис-оријентисане архитектуре

Да би јасно дефинисали шта су сервис-оријентисане архитектуре, прво морамо дефинисати појам "сервисне оријентације". Сервисна оријентација, у контексту софтверског инжењерства, је парадигма развоја софтвера у виду мањих, независних делова. То значи да се логика потребна за решавање неког сложеног проблема може боље конструисати, и лакше имплементирати уколико је та логика рашчлањена на мање, повезане делове [27]. При томе, сваки од тих делова се односи на одређени део проблема. Овај начин развоја софтвера је базиран на концепту расподеле надлежности (енгл. *separation of concerns*), и начин на који је расподела надлежности имплементирана је оно што одваја овај приступ од осталих [27].

Сервис-оријентисане архитектуре представљају архитектонски стил у којем је рачунарски програм подељен у мале, независне логичке јединице [27]. Иако су независне, ове јединице морају да поштују принципе које им омогућавају да независно еволуирају, али да при томе задрже довољну количину „заједништва" и стандардизације [27]. Унутар СОА, ове логичке јединице се називају *сервиси*.

Да би задржали независност, сервиси енкапсулирају програмску логику унутар засебног контекста. Овај контекст може бити специфичан за одређени пословни задатак, ентитет, или неки други вид логичког груписања [27]. Задатак који одређени сервис обавља може бити и велики и мали. Стога, величина сервиса и опсег који обухватају може да варира. Такође, композицијом више сервиса може се добити један сложенији сервис.

Како би се успешно обавио задатак, сервиси често морају да комуницирају и размењују податке. Да би та комуникација била успешна, сервиси морају бити свесни једни других. То се постиже помоћу посебних описа сервиса (енгл. *service descriptions*). Ови описи, у најосновнијем облику, садрже назив сервиса, те опис какав тип података сервис очекује, и какав тип података сервис враћа [27]. Због коришћења описа, не постоји чврста спрега између сервиса, па се због тога у СОА користе радни оквири који су углавном базирани на размени порука.

1.2.8.1 Принципи СОА

До сада, могло се уочити да је СОА доста слична претходно постојећим архитектурама дистрибуираних система које су базиране на размени порука и које јасно одвајају свој интерфејс од пословне логике. Међутим, оно што је специфично за СОА је начин на који се имплементирају три главне компоненте овог архитектонског стила, а то су: сервиси, описи сервиса, и поруке [27].

Унутар СОА, сервиси се дизајнирају на основу следећих принципа [27]:

- Сервисни уговори (енгл. *service contracts*) - сервиси се придржавају уговора о комуникацији. Сервисни уговор обезбеђује формалан опис крајњих тачака (енгл. *endpoints*) сервиса, сваке операције сервиса, формат свих улазних и излазних порука, те правила и карактеристике сервиса и његових операција. Сервисни уговор мора пажљиво да се одржава и верзионише јер од њега зависе сервиси који позивају операције сервиса на који се уговор односи.

- Слаба спрега (енгл. *loose coupling*) - сервиси су слабо спрегнути уколико сервис А захтева услуге сервиса Б, а при томе сервис А остаје независан у односу на сервис Б. Слаба спрега између сервиса се постиже управо коришћењем сервисних уговора који сервисима омогућавају комуникацију унутар већ предефинисаних параметара. Ипак, треба напоменути да су сервис и њему одговарајући уговор увек у чврстој спрези.

- Апстракција (енгл. *abstraction*) - сервиси сакривају програмску логику од остака система, коме су доступне само информације присутне унутар сервисног уговора. Не постоји ограничење колику "количину" програмске логике сервис може имплементирати. Сервис може бити дизајниран тако да имплементира неки једноставан задатак, или чак да имплементира читав скуп комплексних задатака.

- Аутономија (енгл. *autonomy*) - сервиси су аутономни контролишу програмску логику коју енкапсулирају. Аутономија не гарантује да сервис има потпуну контролу над програмском логиком. Стога, разликујемо две врсте аутономије: a) *аутономија на нивоу сервиса* — границе сервиса се међусобно разлику, али сервиси могу да деле ресурсе, и b) *чиста аутономија* – где је програмска логика у потпуности под контролом сервиса.

- Поновна искористивост (енгл. *reusability*) - програмска логика је расподељена по сервисима на начин који подстиче поновну искористивост. Сервис може бити поново искоришћен на многе начине. Један од начина је да се сервис имплементира као помоћни (енгл. *utility*) сервис те се као такав позива из остатка система, или да постане део композитног сервиса.

- Могућност композиције (енгл. *composability*) - више сервиса је могуће саставити на начин да креирају један композитни сервис.

- Непостојање стања (*enlg. statelessness*) - сервиси не задржавају информације везане за одређену активност. Сервис задржава ове информације само у току обраде долазне поруке. Уколико би сервис задржавао ове информације, могућност обраде нових порука би могла бити угрожена. Пожељно је придржавати се овог принципа у случају имплементације сервиса који би био поновно искористив.

- Могућност откривања (енгл. *discoverability*) - sервиси се дизајнирају тако да могу бити доступни преко механизама за откривање(попут регистра за сервисе (енгл. *service registry.*).

Ови принципи се не остварују изоловано. Неки од принципа подржавају друге принципе на различите начине. На пример, принципи попут поновне искористивости и могућности композиције бенефицирају применом осталих принципа [27].

1.2.8.2 Једноставан протокол за приступање објектима

Да би се успешно извршио задатак, без обзира на његову природу и комплексност, сервиси често морају да размењују поруке. Радни оквир за размену порука мора бити стандардизован, тако да сви сервиси користе исти формат поруке и протокол за транспорт.

Једноставан протокол за приступање објектима (*Simple Object Access Protocol - SOAP*) је стандардни протокол за размену порука унутар СОА, базиран на *XML (Extensible Markup Language)* формату. Временом, SOAP се ширио те тако постао радни оквир за дефинисање стандарда за формате порука [27].

SOAP порука се састоји из три дела: коверат (енгл. *envelope*), заглавље (енгл. *header*), и тело (енгл. *body* или *payload*) (Слика 1.9). SOAP коверат је контејнер за SOAP поруку. Заглавље поруке садржи метаподатке. Иако је заглавље поруке опционо, оно у већини сервис-оријентисаних решења представља виталан део архитектуре [27]. Тело поруке носи конкретну поруку коју је потребно пренети. Као део тела поруке, може да се налази и секција са логиком за руковање грешкама која се назива *fault*.

Слика 1.9: Структура SOAP поруке.

1.2.8.3 Слојеви СОА

Три кључна слоја апстракције идентификована у СОА су [27]: слој апликативних сервиса (енгл. *application service layer*), слој пословних сервиса (енгл. *business service layer*), и слој оркестрације сервисима (енгл. *orchestration service layer*).

Слој апликативних сервиса. У овом слоју се налазе сервиси који садрже логику која није део пословне логике. Ови сервиси се још називају и помоћни сервиси (енгл. *utility services*), или инфраструктурни сервиси (енгл. *infrastructure services*). Постоје два начина имплементације ових сервиса у зависности од тога да ли постоји слој пословних сервиса или не. Уколико слој пословних сервиса постоји, сервиси из слоја апликативних сервиса се имплементирају као независни, генерички сервиси [27]. На тај начин, сервиси из овог слоја могу да се користе од стране сервиса из пословног слоја за извршавање одређеног задатка. Пак, уколико пословна логика није измештена у посебан слој, сервиси из апликативног слоја могу да имплементирају и део пословне логике (иако се овај приступ не препоручује у СОА) [27]. Сервиси који садрже и апликативну и пословну логику се називају хибридним апликативним сервисима.

Слој пословних сервиса. У овом слоју се налазе сервиси који су уско везани за пословну логику. У већини случајева, овај слој се дели на два додатна слоја: а) сервиси усредсређени на задатак (енгл. *task-centric business service*), б) сервиси усредсређени на ентитет (енгл. *entity-centric business service*) [27]. Сервиси усредсређени на задатак енкапсулирају пословну логику или пословни процес који обухвата два или више ентитета [27]. Њихова поновна искористивост је углавном ограничена. Сервиси усредсређени на ентитет енкапсулирају логику везану за један специфичан ентитет [27]. Ови сервиси имају висок степен поновне искористивости.

Слој сервиса за оркестрацију. У овом слоју се налазе сервиси чији је задатак да координишу другим сервисима како би се извршио пословни задатак [27]. Увођење овог слоја са собом доноси потребу и за посредничким софтвером. Увођење сервера за оркестрацију не представља тривијалан задатак и има значајан утицај како на повећање комплексности, тако и на финансијски аспект софтверског решења [27].

1.2.9 Архитектуре базиране на микросервисима

Архитектура базирана на микросервисима представља стил развоја апликације као скупа малих сервиса, при чему се сваки од тих сервиса извршава унутар посебног процеса и комуницира са осталим сервисима путем

једноставних механизама (енгл. *lightweight*), као што је на пример HTTP (*Hypertext Transfer Protocol*) [6]. Сваки од ових сервиса је потпуно аутономан што омогућава да се они независно развијају и покрећу.

Иако су микросервиси у последњих неколико година доживели велику популарност и своју примену нашли и у гигантским компанијама као што су *Netflix* [35, 36], *Amazon* [37], *LinkedIn* [38], *SoundCloud* [39], и још многим другим, филозофија на којој се они заснивају практично је иста као и филозофија на којој се базира *Unix* оперативни систем: "Ради једну ствар и ради је добро (енгл. *Do one thing and do it well*)". Unix се састоји из великог броја малих програма (команди) чијом се комбинацијом могу креирати сложени изрази (програми), као на пример:

```
ls -al | grep Projects | more
```

1.2.9.1 Принципи МСА

Микросервиси се заснивају на неколико основних принципа [8, 40]:

- *Скривање детаља имплементације и ограничен контекст* - микросервис скрива своју унутрашњу имплементацију иза јавно доступног интерфејса, те не зна ништа о имплементацији осталих микросервиса који га окружују. Ово осигурава да микросервиси имају јасно означене границе [41]. Појам ограниченог контекста се први пут спомиње у [42], где је дата следећа аналогија са ћелијама: "Ћелије постоје због тога што њихове мембране дефинишу шта припада унутрашњости, а шта спољашности ћелије, као и шта може да прође кроз мембрану". Микросервиси би такође требали и сакривати своје базе података. На тај начин се обезбеђује да се избегне један од најчешћих начина креирања зависности међу деловима дистрибуираног система.

- *Моделовање око пословних концепата.* Интерфејси који су моделовани око пословних концепата уместо техничких концепата су се показали стабилнијим. Моделовањем домена у ком систем ради, обезбеђује не само стабилнији интерфејс, него и могућност лакше обраде измена у пословним процесима [8].

- *Култура аутоматизације.* Један од начина да се изборимо са комплексношћу система базираних на микросервисима је аутоматизација. Стога, имплементација скупа аутоматизованих тестова је од кључне важности. Помаже и аутоматизована процедура увођења, посебно у случају униформне команде која увођење обавља на исти или сличан

начин за све микросервисе. Како би се ово постигло, потребно је дефинисати окружење за увођење (енгл. *environment definitions*).

- *„Smart endspoints and dumb pipes”*. Микросервиси избегавају коришћење комплексних *enterprise* решења (попут *enterprise service bus*) због тога што се пословна логика често пребацује у комуникациони слој, уместо да буде део микросервиса. Резултат овог приступа је тај да је сво „знање” део микросервиса, што доприноси већој кохезији унутар апликације.

- *Величина* - фокус на томе да микросервис буде мали („микро”) је кључна карактеристика микросервиса, што доноси бројне предности у смислу одржавања и проширивања: мали сервис се може лако изменити, или чак и написати испочетка чак и са ограниченим ресурсима и временом. Међутим, понекад је тешко одредити колико заправо микросервис треба да буде мали. Многи аутори [8] [43] сматрају да је микросервис довољно мали уколико може да се напише испочетка и уведе у року од две седмице.

- *Независност* - Сваки микросервис је оперативно независан од осталих и комуницира са њима преко јавно доступних интерфејса. Ово је од фундаменталног значаја, јер сваки микросервис може да се мења без угрожавања коректности система све док се поштују интерфејси. Такође, сваки микросервис се уводи независно.

- *Изолација отказа.* МСА је отпорна на грешке уколико се планира унапред да ће до отказа неизбежно доћи. Пропагација грешака се спречава имплементацијом посебног дизајн обрасца (Секција 1.2.9.5).

- *Праћење статуса.* Како бисмо утврдили да систем функционише правилно, није довољно посматрати микросервисе појединачно. Уместо тога, потребан нам је обједињен поглед на целоукупан систем. Стога, у МСА се често користи семантичко праћење статуса (енгл. *semantic monitoring*), тако што се у систем убацује вештачка трансакција (енгл. *synthetic transaction*) која симулира корисничко понашање.

Управо због ових принципа, микросервиси су значајну примену нашли у рачунарству у облаку (енгл. *cloud computing*) [44–47]. Једна од визија рачунарства 21. века јесте да ће корисници приступати услугама на интернету преко лаких, преносних уређаја, чија се снага не може поредити са традиционалним *десктоп* рачунарима. Због тога што неки корисници немају моћне уређаје, поставља се питање ко ће снабдевати кориснике са потребном рачунарском снагом [48]? Одговор на то питање даје управо рачунарство у

облаку. Појам рачунарства у облаку се односи како на апликације које су преко интернета доступне као сервиси, тако и на хардвер и системски софтвер у центрима за податке (енгл. *data centers*) који нуде те сервисе [49].

Поред рачунарства у облаку, микросервиси су своју примену нашли и у модернизацији застарелих (енгл. *legacy*) система [50]. Добар пример модернизације застарелог система је представљен у [51], где је информациони систем Данске банке мигриран са монолитне архитектуре на архитектуру базирану на микросервисима.

1.2.9.2 Поређење MCA и COA

MCA и COA представљају сличне архитектонске стилове. У литератури, у време писања овог рада, није постојао концензус да ли MCA заправо представља нови архитектонски стил [52], или само еволуцију COA. Неки аутори сматрају да је MCA само „COA урађен како треба" [53,54]. Ипак, постоје значајне разлике између ова два архитектонска стила.

MCA и COA деле неколико заједничких карактеристика. Основне компоненте које сачињавају системе базиране на овим архитектурама су сервиси [55]. Такође, применом било које од ових архитектура добијамо модуларне апликације, чији елементи нису чрсто спрегнути [55]. Поред тога, имплементација сервиса је сакривена иза јавно доступног API-а. То значи да било каква измена имплементације сервиса не утиче на остатак система, све док су измене API-а компатибилне са претходном верзијом [55].

Једна од основних разлика између MCA и COA је у карактеристикама њихових сервиса. Ове разлике се огледају у таксономији сервиса, њиховом власништву, те нивоу грануларности. Из Табеле 1.3 можемо видети да MCA има врло ограничену таксономију сервиса. Сервиси у MCA се деле на функционалне и инфраструктурне. Функционални сервиси имплементирају пословну логику, док инфраструктурни сервиси имплементирају нефункционалне задатке попут аутентификације, ауторизације, евидентирања, и надгледања. Инфраструктурни сервиси у MCA су доступни само унутар апликације, те нису видљиви „остатку света" [55]. Са друге стране, код COA, постоји неколико врста сервиса. Ричардс (Табела 1.3) дефинише следеће врсте сервиса у COA: пословни, *enterprise*, апликативни, те инфраструктурни.

Поред таксономије сервиса, разлика између MCA и COA је очигледна и у власништву над сервисима (Табела 1.3). У случају MCA, развојни тимови су власници сервиса и задужени су за развој и одржавање сервиса током његовог животног циклуса (принцип познат као „you build it, you run it" [33]). Такође, сервиси у MCA су мали и концентрисани на уску функционалност, док сервиси у COA могу да варирају у величини, од малих до веома великих.

Табела 1.3: Поређење карактеристика сервиса у MCA и COA [55].

Архитектура	**Таксономија сервиса**	**Власништво сервиса**
MCA	Фунцтионални сервиси	Тимови за развој апликација
	Инфраструктурни сервиси	Тимови за развој апликација
SOA	Пословни сервиси	Пословни корисници.
	Enterprise сервиси	Архитекте или тимови за развој дељених сервиса.
	Апликативни сервиси	Тимови за развој апликација.
	Инфраструктурни сервиси	Тимови за развој апликација или тимови за развој инфраструктуре.

MCA и COA се разликују и по начину дељења података. Филозофија на којој се заснива MCA је да микросервиси деле што мање података међусобно (нпр. дељена база података је нешто што се избегава у MCA), док COA заговора дијаметрално супротну филозофију: сервиси треба да деле што је могуће више података [55]. Због тога, микросервис и његове податке можемо посматрати као једну јединствену целину са минималним зависностима од остатка система, што олакшава одржавање и увођење микросервиса.

Разлике између MCA и COA можемо уочити и по начину на који се имплементира координација сервиса. MCA се фокусира на кореографију, док се COA ослања и на оркестрацију и на кореографију [55]. С обзиром да код кореографије не постоји медијатор, одговорност за интеракцију преузимају сами микросервиси.

Због наведених разлика, системи изграђени на COA имају тенденцију да буду спорији од микросервиса и захтевају више времена за развој, тестирање, увођење и одржавање [55]. Микросервиси обично имају више компоненти са мање функционалности које захтевају мање ресурса, Стога, скалирање микросервиса одузима мање времена и ресурса [56]. Детаљно поређење MCA и COA у погледу перформанси и комплексности је дато у радовима [57] и [58].

1.2.9.3 Поређење са другим начинима декомпозиције

Већина предности архитектура базираних на микросервисима потиче из њихове грануларности [8]. Међутим, поставља се питање да ли други начини декомпозиције сложених система могу постићи исте резултате?

Дељене библиотеке. Један од стандардних начина декомпозиције је коришћење дељених библиотека. Скоро сви програмски језици имају подршку за овај начин декомпозиције. Типичан приступ је да се велика кодна база разбије на неколико библиотека. Дељене библиотеке такође могу бити преузете од трећих страна, или креиране интерно, унутар организације. Дељене библиотеке промовишу поновну искористивост, и лако се користе. Ипак, овај приступ има следеће мане [8]:

- Недостатак хетерогености. Дељене библиотеке унутар једне кодне базе

већином морају бити писане у истом језику.

- Немогућност независног скалирања делова система.
- У случају да се не користе динамички везане библиотеке, није могуће увести нову верзију дељене библиотеке без поновног покретања читаве апликације.
- Недостатак начина да се обезбеди резилијентност система на нивоу архитектуре.

Модули. Неки програмски језици, попут *Erlang*-а, обезбеђују декомпозицију на програмске модуле, при чему сваки модул има свој животни циклус. То значи да се модули могу мењати током изражавања програма. Сличан ефекат се може постићи и у *Erlang* програмском језику уз ослонац на *OSGI*(The Open Source Gateway Initiative). OSGI је радни окрвир за дефинисање екстензија које је могуће инсталирати у Eclipse развојно окружење. Ипак, иако постоји прилика да се користи *Erlang* или *Erlang* уз комбинацију са *OSGI* радним оквиром, декомпозиција на модуле пати од већине проблема који се јављају и код дељених библиотека.

Њуман у [8] представља занимљиву опзервацију: Иако је могуће креирати монолитне апликације које се састоје из добро дефинисаних модула, то се веома ретко дођага. Као разлог, Њуман наводи то да модули веома брзо постану чврсто спрегнути са остатком кода.

1.2.9.4 Изазови у MCA

Микросервиси са собом доносе неке нове изазове, као што су [7, 8, 59, 60]:

- *Декомпозиција*. На први поглед, декомпозиција система на микросервисе делује као једноставан задатак. Међутим, током овог процеса могуће је суочити се са следећим препрекама [24]:
  - Кашњење у мрежи (енгл. *network latency*). Овај проблем може да се јави у случајевима када након декомпозиције имамо велики број захтева између два сервиса. Понекад се кашњење може свести на прихватљив ниво изменом у API-у, тако да се као одговор на захтев врати више објеката одједном. Међутим, у другим ситуацијама, решење је комбинација два сервиса у један.
  - Смањена доступност због синхроног начина комуникације. Иако је овај начин комуникације једноставан, у неким ситуацијама је асинхрони начин комуникације бољи избор.

– Одржавање конзистентности података. Често постоји потреба да одређена операција ажурира податке унутар више микросервиса. Традиционално решење је коришћење двофазног механизма за управљање трансакцијама, базираног на потврди (енгл. *commit*). *De facto* стандард за управљање дистрибуираним трансакцијама је *X/Open XA* модел[3]. *XA* пружа сигурност да ће сви учесници у трансакцији или потврдити трансакцију или вратити на претходно конзистентно стање (енгл. *rollback*). Ипак, постоје два проблема у примени овог модела на МСА: а) Модерне *NoSQL* базе података, (попут *MongoDB* и *Cassandra*) и брокери за поруке (попут *RabbitMQ* и *Apache Kafka*) немају подршку за овај начин управљања, и б) овај начин управљања смањује доступност система, јер се базира синхроном RPC-у. Алтернативу овом приступу представља *Saga*. *Saga* је секвенца локалних трансакција које су координисане разменом порука. Ограничење овог приступа је да обезбеђује евентуалну конзистентност. У случају да ажурирање података мора да буде атомичка операција, ти подаци би морали да се налазе унутар једног микросервиса, што може представљати препреку за декомпозицију.

– Постојање сложених klasa (тзв. *God classes*). Овакве класе типично имплементирају пословну логику која се дотиче разних делова система. Такође, овакве класе се у базама података обично мапирају на табеле са великим бројем колона. С обзиром да овакве класе вежу стање и понашање разних аспеката апликације, њихова декомпозиција може бити велики изазов.

- *Комплексност.* Архитектура базирана на микросервисима заправо представља дистрибуиран систем, те са собом доноси све проблеме који се сусрећу у дистрибуираним системима [8].

- *Тестирање.* Тестирање дистрибуираних система је инхерентно тежак проблем. Откази су неизбежни и непредвидиви, те се не појављују са униформном вероватноћом и учестаности тако да је скоро немогуће креирати тестове који би покривали све могуће сценарије који би могли довести до отказа. Уколико се узме у обзир и чињеница да микросервиси брзо еволуирају, проблем тестирања постаје још већи. методом вештачког убацивања грешке у систем. Најпознатије варијације

[3]Distributed Transaction Processing: The XA Specification - https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf

ове методе су: *GameDays* (Amazon, Google) [61] и *Simian Army* (Netflix) [35, 36].

- *Увођење.* Сваки микросервис представља посебну апликацију и, у великом броју случајева, поседује специфичан процес увођења, као и механизме за скалирање, анализирање и управљање ресурсима [62].

- *Хетерогеност.* Микросервиси се налазе на различитим машинама (понекад и у разним верзијама) и често су имплементирани коришћењем различитих технологија [63], што доводи до велике хетерогености система. Хетерогеност је веома важна карактеристика МСА јер програмерима омогућава да одаберу технологију или алат који највише задовољава њихове потребе. Међутим, увођење превише језика и радних оквира може драстично да смањи разумљивост и могућност одржавања система. Како би се изборили са овим проблемом, неке организације ограничавају избор програмског језика на свега неколико најчешће коришћених [64]. Ипак, могућност коришћења разноврсних технологија за неке организације представља највећи разлог за увођење МСА [65]. Проблем хетерогености се може ублажити дефинисањем API-ја који је независан од језика у ком је имплементиран микросервис. Многи аутори сматрају да би API требало дефинисати чак и пре саме имплементације микросервиса (енгл. *API-first approach*) [66, 67].

- *Недостатак квалификованих програмера.* Иако имамо пораст броја дистрибуираних система развијених у облику МСА, студије представљене у [65, 68, 69] јасно показују да је проналазак квалификованих програмера, упознатих са МСА, и даље велики проблем. МСА захтева од програмера да се у кратком временском року упозна са широком лепезом технологија и алата.

- *Документација.* Документовати МСА систем такође може бити изазов. Честе измене у архитектури често доводе до неажурних модела [70]. Додатни проблем може бити и чињеница да ови модели често не садрже информације о томе како микросервиси комуницирају [70]. С обзиром да је развој микросервиса високо децентрализован, одржавање целоукупног модела на једном месту такође представља велики проблем [71, 72].

- *Комуникација.* Протокол који се користи за комуникацију микросервиса најчешће је HTTP. HTTP се у дистрибуираним системима најчешће примењује због једноставног захтев-одговор обрасца комуникације. Ипак, због непостојања концепта сесије и чињенице да велики број

микросервиса захтева постојање исте, HTTP поставља тешко бреме на плећа програмера који ручно мора да имплементира механизам који би надокнадио овај недостатак [73].

- *Валидација.* Велики део кода било које врсте дистрибуираног система је писан само због валидације улазних порука [74]. Микросервиси нису изузетак.

- *Руковање грешкама.* Механизам за руковање грешкама се имплементира ручно. Према [75], недостатак или неправилна имплементација механизама за руковање грешкама је најчешћи узрок престанка рада микросервиса.

- *Компатибилност.* Механизам за обезбеђивање компатибилности са старијим верзијама (енгл. *backward compatibility*) се имплементира ручно.

1.2.9.5 Дизајн обрасци у МСА

Дизајн образац је поновно искористиво решење за проблем који се јавља у одређеном контексту. Ричардсон у [24] даје предлог таксономије и описује дизајн обрасце који се користе у МСА. У наставку дисертације, биће разматрани дизајн обрасци из следећих категорија: обрасци увођења (енгл. *deployment patterns*), обрасци комуникације (енгл. *communication patterns*), обрасци за екстерни API (енгл. *external API patterns*), обрасци поузданости (енгл. *reliability patterns*), и обрасци за откривање сервиса (енгл. *service discovery patterns.*). Свака од наведених група садржи неколико различитих дизајн образаца, као што је приказано на Табели 1.4.

Микросервиси често морају да сарађују како би се обавио одређени задатак. Такође, микросервиси понекад морају дати одговор и на захтеве од екстерних клијената. Задатак комуникационих дизајн образаца је да понуде решење за комуникационе проблеме између различитих микросервиса. Најчешће коришћени обрасци за комуникацију код система базираних на микросервисима су *Позив удаљене процедуре* и *Размена порука.* Код позива удаљене процедуре, микросервиси могу да комуницирају на синхрони или асинхрони начин. Комуникација између микросервиса која је базирана на размени порука је увек асинхрона. Према Аксакалију и сарадницима, није могуће утврдити који је најпопуларнији образац за комуникацију јер се одабир обрасца често мења како систем еволвира [76].

Дизајн обрасци за откривање сервиса описују начин на који микросервиси успостављају комуникацију. Образац *Регистар сервиса* обебеђује постојање

регистра за сервисе — базе података која садржи податке о микросервисима који су доступни [77]. Регистар сервиса је критичан део система, те увек мора бити доступан. Одговорност за регистрацију и одјављивање микросервиса из регистра може бити на самом микросервису (енгл. *self-registration*) или на трећој страни (енгл. *third-party registration*). Слично, одговорност за откривање микросервиса може бити на клијентској (образац *Откривање на клијентској страни*) или на серверској страни (образац *Откривање на серверској страни*).

Дизајн обрасци из категорије *Екстерни API* описују начин на који се обавља комуникације између екстерних клијената и микросервиса. Образац *API пролаз* обезбеђује јединствену улазну тачку у систем за све екстерне клијенте. Како би се избегао *single-entry failure* проблем, обично постоји више инстанци API пролаза. Додатно, често се обезбеђује и посебан API пролаз за сваки тип клијента (*web* апликација, мобилна апликација, и сл.). Ова варијација обрасца *API пролаз* се назива *Back-end за front-end*.

Микросервиси се имплементирају са очекивањем да ће отказати (принцип *built-to-fail* [8]). Да пад одређеног микросервиса не би ланчано пропагирао на остале микросервисе, користи се образац *Осигурач*, који припада категорији образаца за поузданост.

1.2.10 Језици специфични за домен

Језик специфичан за домен (ЈСД, енгл. *Domain Specific Language - DSL*) је језик који кроз одговорајуће нотације и апстракције нуди повећање експресивности фокусирану и обично ограничену на одређени домен проблема [78]. Задатак ЈСД-а није да реши све врсте рачунарских проблема, чак нити велике класе таквих проблема [79], што омогућава да језик буде веома експресиван за проблеме који потпадају у домен који ЈСД покрива. ЈСД подстиче изградњу заједница стручњака из домена који „говоре истим језиком” [80].

ЈСД-ови чији су домен, апстракције и нотације блиско усклађени са начином изражавања доменских експерата (који су често не-програмери), омогућавају експертима да лако читају програмски кôд писан на ЈСД-у јер није претрпан ирелевантним детаљима имплементације [81]. Стручњаци из домена често и сами пишу програмски кôд [81]. Iако ниво укључености доменских експерата може варирати, врло је важно да ЈСД обезбеди синтаксу коју могу да прочитају и разумеју јер такав ЈСД може послужити како алат за дизајн и имплементацију решења, тако и као медиј за опис корисничких захтева [82].

Многе емпиријске студије потврђују да употреба ЈСД-ова повећава

флексибилност, продуктивност, поузданост, и употребљивост [83,84]. Студија коју су спровели Јохансон и Хаселбринг [85] показује да доменски експерти постижу знатно већу прецизност и троше мање времена на решавање задатака када користе ЈСД уместо решења заснованог на језицима опште намене (ЈОН, енгл. *General Purpose Language - GPL*). Студија коју су спровели Косар и сарадници [86] показује да су учесници ефикаснији када користе ЈСД уместо ЈОН, посебно у доменима у којима су имали мање искуства. Студија, такође, сугерише да ће програмери боље обављати посао коришћењем ЈСД-а, у случају да је он доступан.

Најлакши начин да се дизајнира ЈСД јесте да се базира на постојећем језику. ЈСД-ови изграђени на овај начин се називају *Интерни ЈСД-ови*. Предност овог приступа је у томе што не мора да се имплементира додатна инфраструкура, међутим, мана је нефлексибилност која се огледа у синтакси која је ограничена могућностима базног језика [87]. Други приступ је дизајн тзв. *Екстерних ЈСД-ова*. Екстерни ЈСД-ови се граде „од нуле", те им је потребна сопствена инфраструктура у виду парсера, преводиоца, или интерпретера [87].

1.2.11 Моделовање специфично за домен

Програмери углавном праве разлику између моделовања и кодирања. Модели се користе како би се систем дизајнирао или боље разумео. Модел је спецификација свих потребних функционалности система, и уједно служи и као његова документација. Са друге стране, кôд представља конкретну имплементацију модела.

Код традиционалног развоја софтвера, не постоји веза између модела и кôда. Програмери анализирају моделе, а затим кодирају. Током овог процеса, често долази до ситуације да се модели не ажурирају или се у потпуности одбаце након што је кодирање завршено. Разлог томе је тај што је цена одржавања истих информација на два места, кôду и моделима, доста висока јер је то ручни процес, склон грешкама [88].

Инжењерство управљано моделима (енгл. *Model-Driven Engineering - MDE*) [89,90] је методологија развоја софтвера у којој се модел употребљава током читавог циклуса развоја софтвера. Код MDE, за разлику од традиционалног развоја софтвера, модели представљају формалну спецификацију решења која се аутоматизованим трансформацијама преводи у извршива софверска решења.

Моделовање специфично за домен (енгл. *Domain-Specific Modeling - DSM*) је варијација инжењерства управљаног моделима које се фокусира на креирање језика специфичних за домен и алата за моделовање и

имплементацију софтверских решења у одређеном домену. Циљ моделовања специфичног за домен је да постигне две ствари [88]:

- Подизање нивоа апстракције креирањем спецификације решења на језику који директно користи концепте и правила из домена, и
- Генерисање извршивих решења на одабраном програмском језику на основу апстракне спецификације решења.

Табела 1.4: Обрасци за дизајн микросервиса. Опис свих дизајн образаца је преузет из [24].

Намена	**Назив**	**Опис**
Увођење	Једна инстанца по хост рачунару	Сваки хост рачунар има само једну инстанцу сервиса
	Више инстанци по хост рачунару	Један хост рачунар може имати више инстанци сервиса
	Једна инстанца по контејнеру	Свака инстанца сервиса се уводи коришћењем контејнера
	Бесерверско (енгл. *serverless*) увођење	Инстанце сервиса се уводе путем инфраструктуре која „сакрива" постојање сервера.
Комуникација	Позив удаљене процедуре	Сервиси за комуникацију користе протокол базиран на позиву удаљене процедуре.
	Размена порука	Сервиси комуницирају асинхроно, путем размене порука.
Екстерни API	API пролаз (енгл. *API gateway)*	Обезбеђује једну улазну тачку у систем за све екстерне клијенте.
	Back-end за front-end	Обезбеђује посебну улазну тачку у систем за сваки од екстерних клијената.
Поузданост	Осигурач (енгл. *Circuit breaker*)	Спречава да се пад мреже или сервиса пропагира ка осталим сервисима.
Откривање сервиса	Откривање на клијентској страни (енгл. *Client-side discovery*)	Клијентска страна открива локацију серверског сервиса
	Откривање на клијентској страни (енгл. *Server-side discovery*)	Клијентски сервиси захтеве ка серверу шаљу преко мрежног усмеривача (*енгл. router*).
	Регистар сервиса	Садржи информације о локацији сваке инстанце сервиса.

Поглавље 2

Преглед постојећих истраживања у оквиру теме

Кључно је бити у друштву људи који вас уздижу, чија присутност захтева ваше најбоље.

Епиктет

У овом поглављу биће дат приказ постојећих решења за опис архитектура базираних на микросервисима. С обзиром да постоји већи број решења за опис МСА, у овој секцији биће детаљније приказана само решења чији је кôд јавно доступан.

Преглед решења пре свега ће бити усредсређен на начин на који се описује микросервисна архитектура, те на скуп дизајн образаца који су подржани.

2.1 Језици за опис архитектуре

MAGMA (*Maven Archetype for Generating Microservice Architectures*) је алат за развој базиран на Мавен систему за управљање изградњом програма. Његов циљ је да убрза развој архитектура базираних на микросервисима генерисањем инфраструктурног кода који је: (а) специјално конфигурисан за циљни домен апликације; (б) директно извршив; (в) проширив са шаблонским кодом који је дефинисан од стране корисника [91]. Корисници могу једноставно да креирају своје системе кроз једноставни дијалог за унос података. Сервиси као што су сервис за откривање, сервис за сигурност, сервис за управљање корисницима и сервис за ресурсе су унапред

имплементирани у MAGMA алату и доступни су у дијалогу за унос података. Дијалог за унос података садржи мали прегледни панел, који приказује архитектуру система који се генерише. Иако је једноставан за коришћење, подесити MAGMA алат може бити проблематично и временски захтевно, посебно за програмере који нису навикли на Мавен. Такође, пошто MAGMA генерише само школьке пројеката за сервисе који су дефинисани кроз шаблоне корисника, потребно је ручно имплементирати доменски модел, заједно са API-јем и пословном логиком. Поред тога, MAGMA нема подршку за итеративни развој.

AjiL је алат за креирање и описивање MCA базиран на Eclipse радном оквиру за моделовање (*Eclipse Modeling Framework - EMF*). AjiL се састоји из два дела: графичког едитора и генератора кода базираног на шаблонима. Генератор кôда преводи AjiL дијаграме у извршиве апликације базиране на Javi и Spring радном оквиру [12]. Графички едитор AjiL-а је лепо дизајниран и једноставан за коришћење. AjiL, као и MAGMA, долази са скупом унапред имплементираних сервиса, што корисницима омогућује да брзо креирају моделе својих апликације. Међутим, генерисање кôда из модела није једноставно. Графички едитор и кôд генератор код овог алата су два засебна пројекта између којих није дефинисана никаква веза. Стога, да би се генерисао кôд из модела, потребно је копирати модел у пројекат генератора. Поред тога, AjiL тренутно не пружа механизам за интеграцију ручно писаног кода са генерисаним кôдом.

JHipster је алат за генерисање, развој и увођење веб апликација и апликација базираних на микросервисима. Архитектуре базиране на микросервисима се у JHipster алату моделују помоћу посебног језика који се зове JHipster Domain Language (JDL). Једноставна синтакса JDL језика омогућује корисницима да брзо креирају моделе апликација. Модели се, потом, аутоматски преводе у извршиве Java апликације помоћу JHipster генератора кода.

LEMMA (*Language Ecosystem for Modeling Microservice Architecture*) је скуп језика за моделовање базираних на Eclipse радном оквиру који служе за развој архитектура базираних на микросервисима [92]. Ови језици пружају различите погледе на модел за сваку од улога у тиму за развој микросервиса [93]. Помоћу погледа, LEMMA разлаже систем на мање, специјализованије делове. Због тога, свака улога располаже само са информацијама релевантним за ту улогу [93]. Баш као и AjiL, LEMMA је такође заснован на Eclipse екосистему.

Jolie је програмски језик за имплементацију сервис-оријентисаних архитектура [94]. Сервис у Jolie се састоји од два дела: део који описује понашање, и део који описује увођење. Део за опис понашања дефинише

имплементацију функционалности сервиса, док део за опис увођења дефинише неопходне информације за успостављање комуникационих веза између сервиса [94]. Jolie интерпретер је имплементиран у Java програмском језику. За интеракцију са интепретером могуће је користити Java API.

2.2 Језици за комуникацију

Постоји неколико решења која у многоме олакшавају имплементацију комуникације између микросервиса. Нека од тих решења су: *Apache Trift*, *Apache Avro*, *Protocol Buffers*, *Babel*, *RAML*, *Swagger*, итд.

Apache Thrift [95] је радни оквир који омогућује да две апликације писане у различитим програмским језицима могу да комуницирају коришћењем позива удаљене процедуре. Thrift обезбеђује језик за опис интерфејса помоћу ког се дефинише и сам API апликације. Thrift IDL синтаксом подсећа на програмски језик C, а на основу њега се аутоматски генеришу комуникациони слојеви на клијентској и серверској страни. Приликом RPC позива, параметри методе се могу серијализовати у: JSON формат, бинарни формат, или компактни бинарни формат. Одабир формата за серијализацију знатно може утицати на перформансе. Тако имамо да је серијализација и десеријализација значајно бржа код бинарних формата него код JSON формата. При томе, компактни бинарни формат је и меморијски ефикаснији. Предност JSON формата је у томе што је читљив и подржан од стране web претраживача.

Apache Avro [96] је радни оквир за серијализацију података развијен као део Hadoop пројекта. Avro обезбеђује разне формате за перзистенцију и размену података између чворова у Hadoop мрежи, као и механизам за позив удаљене процедуре. API сервиса, као и начин на који ће се подаци серијализовати и десеријализовати, описује се помоћу посебне шеме која се задаје о облику JSON документа. Avro se лако интегрише са динамичким језицима попут Python-a ili Javascript-a јер не захтева генерисање посебних комуникационих слојева. Приликом позива удаљене процедуре, чворови прво врше размену шема, па тек потом размену података.

Protocol Buffers [97] је механизам за серијализацију структуираних података развијен од стране компаније Google. Формат података се описује у виду *шеме* поруке која се шаље од једног сервиса ка другом. Компајлирањем шеме поруке помоћу Protocol Buffers програмског преводиоца, генерише се код за серијализацију и десеријализацију порука. Свака порука садржи јединствено нумерисане атрибуте. Сваки атрибут поседује назив и тип, при чему тип може бити неки од уграђених типова (*integer*, *floating-point*, *boolean*,

string или *enumeration*) или кориснички дефинисан тип. Свака порука се приликом слања серијализује у ефикасан бинарни формат који примаоц поруке може брзо и ефикасно да десеријализује. За разлику од радних оквира Thrift и Avro, Protocol Buffers не обезбеђује инфракструктуру за коришћење позива удаљене процедуре. У ову сврху се користи **gRPC** [98], који користи Protocol Buffers као механизам за серијализацију података, али и као IDL у ком се дефинише сервис (gRPC проширује формат за спецификацију *шеме* поруке са додатним елементима који омогућују дефинисање сервиса у виду скупа метода са дефинисаним параметрима и повратним вредностима).

Babel [99] је алат који обезбеђује интероперабилност између апликација писаних у различитим програмским језицима. Дизајниран је са сврхом да реши посебне функционалне проблеме и проблеме са перформансама у развоју великих симулација у чији развој је интегрисано више математичких модела, библиотека, те апликација имплементираних у различитим програмским језицима. За опис интерфејса, Бабел користи посебан језик - SIDL (*Scientific Interface Definition Language*). SIDL поред основних типова података има подршку и за комплексне, мултидимензионалне типове. SIDL поред тога нуди и потпуну подршку за полиморфизам и преко граница програмских језика. На пример, код писан у Python-u може да изврши редефинисање методе писане у Fortran модулу на начин да она баца изузетак који је имплементиран у програмском језику C++.

RAML(*RESTful API Modeling Language*) [100] је језик за дефиницију API-а који су базирани на принципима REST протокола. RAML обезбеђује механизам за дефинисање RESTful API-ја, креирање изворног кода за клијент/сервер архитектуру, и свеобухватно документовање API-ја за крајње кориснике. API се специфицира у облику YAML документа. RAML језик поред текстуалне нотације нуди и графичку. API се може дефинисати графички помоћу следећих алата:

- API Workbench - интегрисано развојно окружење (IDE) за дизајнирање, тестирање, документовање и размену RESTful API-а,
- API Designer - алат за дизајн API-а, базиран на web технологијама.

Swagger [101] представља скуп алата за текстуално или графичко креирање API-а. API се може дефинисати у виду JSON или YAML документа, на основу којих је могуће аутоматски генерисати комуникациони слој за сервер и/или клијент. Swagger подржава генерисање кода за око четрдесетак програмских језика, укључујући и најпопуларније: Python, Java, C++, C#, Ruby, Haskell, Skala, итд.

Поглавље 3

Silvera језик

Не будите мудри у речима -
будите мудри у делима.

Марко Аурелије

Silvera је декларативни језик који је развијен за домен архитектура базираних на микросервисима. Овај тип ЈСД-а називамо „техничким ЈСД-ом" или „хоризонталним ЈСД-ом". Језик је дизајниран на начин да директно имплементира дизајн обрасце везане за домен МСА.

Постоји неколико приступа имплементацији језика специфичних за домен. Класификацију ових приступа, у виду скупа дизајн образаца, први је одрадио Спинелис [102]. Рад Спинелиса су, потом, проширили Мерник и сарадници [103]. Обрасци су везани за фазе развоја језика специфичних за домен, те се могу поделити у следеће категорије:

- обрасци одлучивања (енгл. *decision patterns*) - обрасци који описују скуп познатих ситуација где су се језици специфични за домен показали као успешни,
- обрасци за анализу (енгл. *analysis patterns*) - обрасци који дају смернице како препознати проблем унутар домена, и како прикупити релевантно доменско знање,
- обрасци за дизајн (енгл. *design patterns*) - обрасци који дају смернице како дизајнирати језик специфичан за домен, те
- обрасци за имплементацију (енгл. *implementation patterns*) - обрасци који дају смернице како одабрати најпогоднији приступ за имплементацију.

Развој Silvera језика се одвијао кроз четири сукцесивне фазе: *фаза анализе*, *фаза одлучивања*, *фаза дизајна*, те *фаза имплементације*.

Током фазе анализе, већина доменског знања је прикупљена анализирањем доступне литературе, кôда и документације доступних система. Домен је анализиран на неформалан начин, и већина литературе је прикупљена коришћењем *snowballing* приступа. Почетни скуп литературе је проширен тако што је анализирана литература која је цитирана у радовима из почетног скупа (*backward snowballing* приступ). Током ове фазе прикупљена је терминологија коришћена унутар домена и семантика појединачних термина.

Одлука о креирању новог језика специфичног за домен донесена је на основу примећених недостатака постојећих решења, пре свега на могућност а) аутоматског генерисања инфраструктуре апликације, и б) доменске анализе и евалуације система базираног на микросервисима.

Током фазе дизајна, потребно је одлучити у којој мери се језик специфичан за домен ослања на постојеће језике, те да ли ће језик бити описан формално или неформално [103]. Постоје два начина креирања језика специфичних за домен: базирање на постојећем језику, или креирање потпуно независног језика. Оба приступа имају своје предности и мане (видети Секцију 1.2.10). Silvera спада у групу екстерних ЈСД-ova, те нема директну везу ни са једним од постојећих програмских језика.

Мерник и сарадници [103] уводе разлику између формалног и неформалног дизајна. Код неформалног дизајна, спецификација језика је обично у неком облику природног језика. У случају формалног дизајна, синтакса језика се специфицира кроз регуларне изразе и граматику, док се семантика специфицира кроз формализме попут машине стања [103]. Формални дизајн има неколико предности, попут: а) отрива проблеме пре него што се ЈСД zaista implementira, и б) може се аутоматски имплементирати преко алата за развој језика, што увелико скраћује време развоја новог ЈСД-а. Синтакса Silvera језика је специфицирана у облику PEG (*Parsing Expression Grammar*) граматике, док је семантика дефинисана од стране генератора кода.

Постоји неколико образаца који се могу користити током фазе имплементације [103]. Током ове фазе, коришћен је образац који подразумева имплементацију посебног програмског преводиоца (компајлера). Иако је овај приступ „скупљи” од осталих, ако се узме у обзир време потребно за имплементацију, ту „цену” оправдава флексибилност коју нуди. Коришћењем овог обрасца, могуће је: креирати синтаксу која је приближна нотацијама које користе доменски експерти, обезбедити добар систем за пријаву грешака [103], те умањити време потребно да корисници напишу коректан програм [104].

3.1 Апстрактна синтакса Silvera језика

У овој секцији ће бити представљена апстрактна синтакса Silvera језика. Апстрактна синтакса Silvera језика је специфицирана у виду метамодела. Поједностављена верзија метамодела је приказана на Слици 3.1.

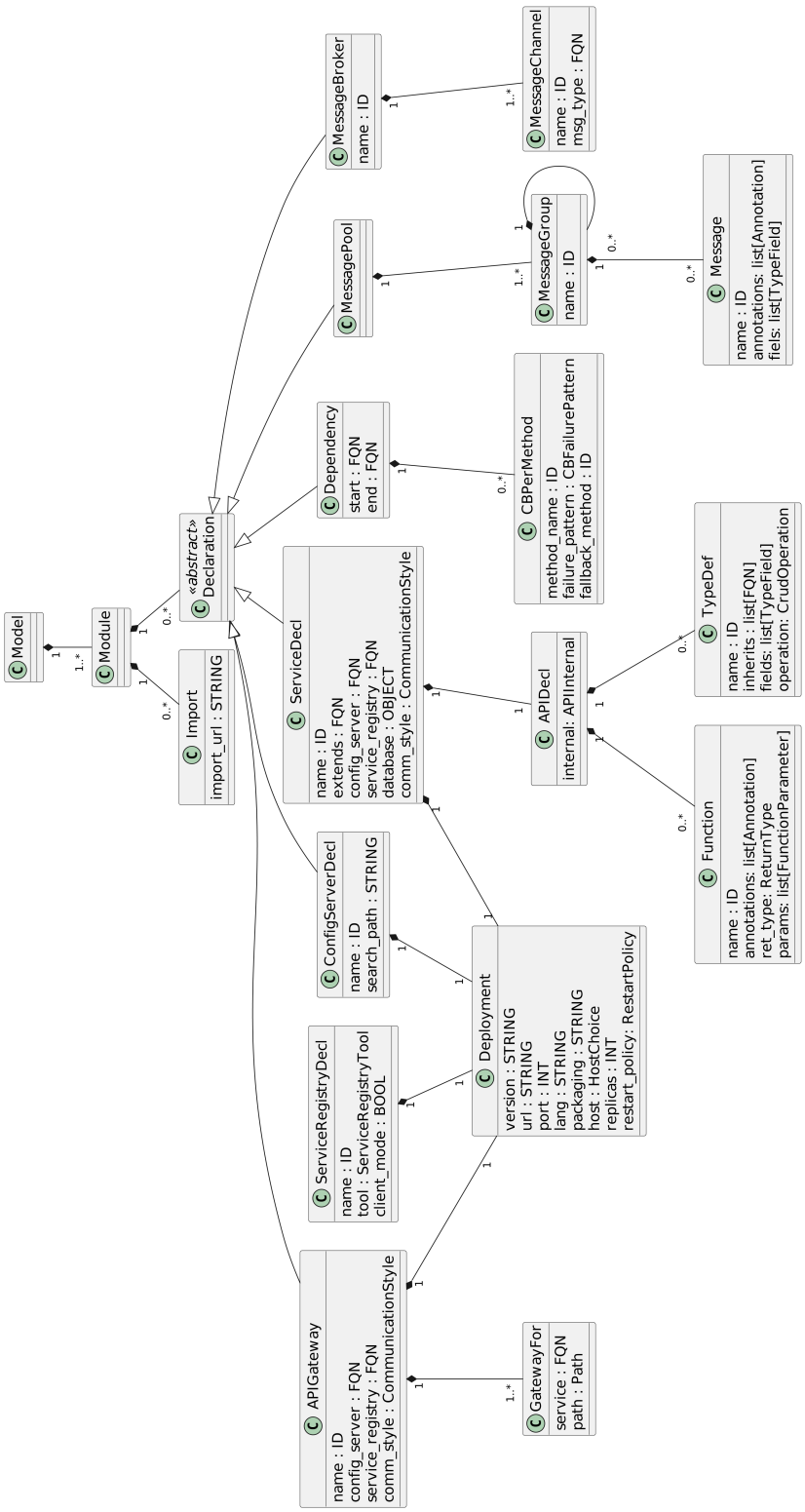
Главни концепт Silvera метамодела је `Model`, који се састоји од једног или више модула (`Module`). Модул омогућава корисницима да логички организују свој Silvera код. Сваки модул може садржати декларације микросервиса (`ServiceDecl`), API пролаза (`APIGateway`), регистра сервиса (`ServiceRegistryDecl`), конфигурационих сервера (`ConfigServerDecl`), зависности (`Dependency`), скупа порука (`MessagePool`) или посредника за поруке (`MessageBroker`). Свака декларација се може идентификовати по свом јединственом имену на нивоу модула. Јединствени идентификатор за декларације унутар модела је њихово пуно име (*Fully Qualified Name -FQN*), које се добија по следећој формули:

<putanja_do_modula>.<naziv_modula>.<naziv_deklaracije>

Модул може да референцира декларације из другог модула путем импорта. За сваки микросервис је могуће дефинисати његово име, API, стратегију распоређивања, стил комуникације, да ли микросервис треба да се региструје унутар регистра сервиса, и да ли треба да преузима податке о конфигурацији са конфигурационог сервера. Декларација API-ја (`APIDecl`) se састоји од дефиниција функција и дефиниција објеката специфичних за сервис који се користе за моделовање пословних ентитета микросервиса. Свака дефиниција функције се састоји од имена функције, параметара функције, типа повратне вредности и анотације (опционо), док се дефиниција објекта специфичног за сервис (`TypeDef`) састоји од имена и једног или више поља (`TypeField`). Свако поље има своје име и тип података, али може имати и посебан атрибут ID, који се користи за идентификацију поља током серијализације и десеријализације порука у бинарном формату, и за осигуравање компатибилности са предходним верзијама. Због тога, вредност додељена атрибуту ID не би требала да се мења.

У Silvera језику, сваки микросервис има посебан протокол који се користи за слање и примање порука. Тренутно је могуће изабрати између *messaging* и *RPC* (видети Секцију 1.2.3) протокола. Формат порука које се могу послати и примити од микросервиса је дефинисан његовим API-јем. Микросервиси који користе RPC да позивају методе из других микросервиса морају да их дефинишу као зависности. Silvera подржава и синхрони и асинхрони начин комуникације приликом коришћења RPC протокола.

Будући да се отказ код микросервиса може десити у било ком тренутку, MCA морају бити дизајниране тако да могу да успешно рукују са грешкама



Слика 3.1: Поједностављена верзија Silvera метамодела.

[6]. Отказ једног микросервиса не би требао да доведе до пада читавог система. Један од дизајн образаца који помаже у ублажавању овог проблема је *Осигурач*. Овај образац је директно подржан у Silveri. Стратегија за опоравак од грешке мора бити дефинисана за сваку API функцију коју почетни микросервис (`start`) позива из крајњег микросервиса (`end`). Табела 3.1 приказује стратегије за опоравак од грешке које подржава Silvera.

Табела 3.1: Стратегије за опоравак од грешке које подржава Silvera.

Назив стратегије	**Опис**
fail_fast	Подиже изузетак на клијентској страни ако API позив не успе (подразумевано понашање)
fail_silent	Враћа празан одговор
fallback_static	Враћа одговор са подразумеваним вредностима
fallback_stubbed	Одговор је сложени објекат где свако поље има или подразумевану вредност или вредност прорачунату на основу захтева.
fallback_cache	Враћа кеширану верзију одговора, уколико она постоји. У супротном, враћа празан одговор као *fail_silent*
fallback_method	Дефинише методу која ће бити позвана у случају да позив оригиналне методе изазове изузетак.

Приликом коришћења протокола базираног на порукама, микросервиси комуницирају асинхроно. Сваки тип поруке (`Message`) који се користи унутар система, мора бити дефинисан унутар скупа порука (`MessagePool`). Овај скуп порука је глобално доступан и може се референцирати из било ког модула. Поруке се достављају на дестинацију помоћу посредника или брокера (`MessageBroker`). Сваки брокер поседује један или више канала за комуникацију (`MessageChannel`). Брокер креира канале за поруке, и сваки канал може имати више произвођача (енгл. *producer*) и/или потрошача (енгл. *consumer*). Потрошач је сваки микросервис који чита поруке са канала. Аналогно, произвођач је микросервис који уписује поруке у канал. Канали за поруке у Silveri су типизирани, што значи да је сваки канал посвећен одређеном типу поруке. Поруке се могу конзумирати или производити само путем API метода. Да би микросервис послао поруку, његова API метода мора бити регистрована као произвођач на каналу чији је специфични циљ комуницирање тог типа поруке. Слично, да би микросервис примио поруку одређеног типа, његова API метода мора бити регистрована као потрошач канала који садржи дати тип поруке. Метода се може регистровати као потрошач путем `@consumer` анотације, и као произвођач путем `@producer` анотације. Метода, такође, може у исто време бити и потрошач и произвођач порука. Канали за поруке су логичке адресе у систему за слање порука. Начин

на који се заправо имплементирају канали зависи од имплементације система за слање порука [105].

У МСА, клијентске апликације обично морају да прикупе податке из више од једног микросервиса. Ако је комуникација директна, клијент мора да комуницира са више микросервиса да би прикупио податке. Таква комуникација је неефикасна и повећава зависност између клијента и микросервиса [24]. Алтернатива је имплементација API пролаза. API пролаз представља једну улазну тачку у систем за све клијенте и може да обрађује захтеве на један од два начина: а) захтеви се прослеђују ка одговарајућем сервису, и б) захтеви се деле на више микросервиса. У Silveri, API пролаз је посебан сервис имплементиран у облику објекта (`APIGateway`). Његов атрибут `gateway_for` одређује који ће микросервиси бити стављени иза API пролаза. У тренутној имплементацији Силвере, API пролаз служи само као рутер захтева. У будућности планирано је проширење ове имплементације. Посебан акценат ће бити на имплементацији обрасца за API композицију (*API Composition*) [24], и функционалности везаних за сигурност. Образац за API композицију се ослања на композитора како би имплементирао комплексне упите. API композитор позива појединачне микросервисе који поседују податке, а затим спаја резултате и шаље одговор.

У Silveri, сваки микросервис може имати специфичне захтеве за увођење. Начин на који ће се спровести увођење микросервиса је описано `Deployment` објектом, са следећим атрибутима: *version*, *url*, *port*, *lang*, *packaging*, *host*, *replicas*, и *restart_policy*. Атрибут *version* дефинише верзију микросервиса. Атрибути *url* и *port* дефинишу локацију микросервиса на мрежи. Атрибути *lang* и *packaging* дефинишу програмски језик у ком ће микросервис бити имплементиран и у ком ће облику бити коришћен (изворни код или бинарни). Атрибут *host* дефинише да ли ће микросервис радити на физичкој машини, виртуелној машини или унутар контејнера, док атрибут *replicas* дефинише број инстанци микросервиса. На крају, атрибут *restart_policy* дефинише када микросервис треба да се поново покрене (након отказа, увек, итд.). Овај атрибут тренутно се може користити само ако је хост контејнер.

Регистар сервиса је посебан сервис који садржи информације о броју инстанци и локацијама сваког микросервиса у систему. У Silveri, регистар сервиса је имплементиран у облику објекта `ServiceRegistryDecl`. Помоћу атрибута `client_mode` регистар сервиса може да се региструје унутар другог регистра сервиса. Будући да је то посебан тип микросервиса, регистар сервиса такође може да специфицира процедуру увођења користећи објекат `Deployment`. Микросервис се региструје унутар регистра сервиса наводећи референцу на објекат `ServiceRegistryDecl` у његовом атрибуту `service_registry`.

У Silvera језику, микросервиси могу да учитавају конфигурационе фајлове са спољног сервера за конфигурацију. Сервер за конфигурацију је имплементиран у облику објекта `ConfigServerDecl`.

3.2 Конкретна синтакса Silvera језика

У овој секцији биће представљена конкретна синтакса језика Silvera. Конкретна синтакса дефинише како се концепти апстрактне синтаксе приказују кориснику [81]. За једну апстрактну синтаксу, могуће је креирати више конкретних синтакси (текстуалне, графичке, итд.). Конкретна синтакса Silvera је доступна у облику текстуалне нотације. У тексту који следи биће такође дат приказ граматике за најважније концепте језика. Због једноставности, неки делови граматике нису приказани. Пуна верзија граматике је доступна на GitHub-у[1].

3.2.1 Декларација микросервиса

На Листингу 3.1 је дат поједностављен приказ правила за декларацију микросервиса, из граматике Silvera језика.

Листинг 3.1: Део правила граматике за декларацију микросервиса

```
ServiceDecl:
    'service' name=ID '{'
        ('config_server' '=' config_server=FQN)?
        ('service_registry' '=' service_registry=FQN)?
        (deployment=Deployment)?
        (api=APIDecl)?
    '}'
;
```

Правило `ServiceDecl` почиње са кључном речју `service`, након које следи атрибут `name` који одговара правилу `ID`, уграђеном у *textX* [106]. *textX* је слободан алат отвореног кода који је у потпуности развијен у Python програмском језику. Предност овог алата је у томе што скраћује време од тренутка измене граматике језике до његовог коришћења [106]. Ова карактеристика је веома битна за развој језика специфичних за домен, јер су такви језици склони честим изменама [103]. Након тога следи литерал „{” (линија 2). На почетку тела декларације микросервиса се могу наћи

[1]Граматика Silvera језика – https://github.com/alensuljkanovic/silvera/blob/master/silvera/lang/silvera.tx

две опционе промењиве. Прва промењива чува референцу на одговарајући сервер са екстерном конфигурацијом. Дефиниција ове промењиве почиње са кључном речју `config_server`, након које следи литерал „=”, а затим и атрибут `config_server` који одговара правилу `FQN` (линија 3). Следећа промењива је `service_registry` и дефинише се на сличан начин (линија 4). Ова промењива чува референцу на регистар сервиса. Затим, следи опциони атрибут `deployment` који је дефинисан правилом `Deployment` (линија 5), те атрибут `api` дефинисан правилом `APIDecl` (linija 6). Затварајућа заграда завршава декларацију микросервиса (линија 7).

На листингу 3.2 дат је пример дефиниције `User` микросервиса. `User` микросервис је регистрован у `ServiceRegistry` регистру за сервисе (линија 3). Приликом увођења овог микросервиса биће коришћена контејнерска технологија, а захтеве ће слушати на порту 8080. С обзиром да атрибут `host` није дефинисан, биће коришћена подразумевана вредност — `http://localhost`. API овог микросервиса се састоји од `User` доменског објекта, те неколико јавно доступних метода. CRUD методе за `User` доменски објекат ће бити генерисане аутоматски због `@crud` анотације. Ова анотација представља скраћени начин дефинисања CRUD metoda. Исти ефекат би се могао постићи коришћењем следећих анотација: `@create`, `@read`, `@update`, і `@delete`. Поред CRUD метода, `User` микросервис поседује још три додатне методе: `listUsers`, `userExists`, те `userEmail`. Све методе овог микросервиса комуницирају путем REST протокола. URL мапирање за сваку од метода ће се генерисати аутоматски, на основу URL-а микросервиса, назива методе, те одабраног HTTP метода дефинисаног унутар `@rest` анотације. На пример, методи `listUsers` се може приступити преко следећег URL-а: `http://localhost:8080/user/listusers`. URL мапирање је могуће мењати преко `mapping` атрибута `@rest` анотације: `@rest(method=GET,mapping=<user_defined_mapping>)`.

```
1  service User {
2
3      service_registry=ServiceRegistry
4
5      deployment {
6          version="0.1"
7          port=8080
8          host=container
9      }
10
11     api{
12         @crud
13         typedef User {
14             @id str username
```

```
15              @required str password
16              @required @unique str email
17
18              int age  //optional
19          }
20
21          @rest(method=GET)
22          list<User> listUsers()
23
24          @rest(method=GET)
25          bool userExists(str username)
26
27          @rest(method=GET)
28          str userEmail(str username)
29      }
30  }
```

Листинг 3.2: Пример дефиниције микросервиса помоћу Silvera језика

3.2.2 Декларација регистра за сервисе

На листингу 3.2 дат је пример дефиниције регистра за сервисе помоћу Silvera језика. Назив регистра за сервисе је `ServiceRegistry`. Овај регистар се неће регистровати као клијент унутар неког другог регистра за сервисе (линија 2). `ServiceRegistry` ће бити доступан на адреси `http://registry.example.com` (линија 6), и захтеве ће очекивати на 9091 HTTP порту (линија 5). Тренутна верзија регистра је `0.0.1` (линија 4). Приликом увођења регистра биће коришћена контејнерска технологија (линија 7).

```
1  service-registry ServiceRegistry {
2      client_mode=False
3      deployment {
4          version="0.0.1"
5          port=9091
6          url="http://registry.example.com"
7          host=container
8      }
9  }
```

Листинг 3.3: Пример дефиниције регистра за сервисе помоћу Silvera језика

3.2.3 Декларација API пролаза

На листингу 3.2 дат је пример дефиниције API пролаза (`EntryGateway`) помоћу Silvera језика. `EntryGateway` обезбеђује једну улазну тачку у систем. Ово корисницима олакшава интеракцију са системом јер је довољно запамтити само један URL. API пролаз прослеђује захтеве до одговарајућих микросервиса, при чему је потребно дефинисати URL мапирања за сваки од микросервиса (линија 12). На пример, да би се позвала метода `listUsers` из `User` микросервиса, потребно је користити следећи URL: `http://entry.example.com:9095/api/u/listusers`. `EntryGateway` је, такође, регистрован унутар `ServiceRegistry` регистра за сервисе.

```
1  api-gateway EntryGateway {
2
3      service_registry=ServiceRegistry
4
5      deployment {
6          version="0.0.1"
7          port=9095
8          url="http://entry.example.com"
9      }
10
11     gateway-for{
12         User as /api/u
13     }
14 }
```

Листинг 3.4: Пример дефиниције API пролаза помоћу Silvera језика

3.2.4 Декларација зависности између микросервиса

На листингу 3.5 дат је пример дефинисања зависности између `Order` и `User` микросервиса. У овом примеру, микросервис `Order` зависи од микросервиса `User`. Конкретно, микросервис `Order` позива методе `userExists` и `userEmail` које припадају микросервису `User`. За сваку од метода, потребно је дефинисати и стратегију опоравка од отказа. Тако имамо да се при позиву методе `userExists` користи стратегија `fallback_static`, док се стратегија `fail_silent` користи приликом позива методе `userEmail`.

```
1 dependency Order -> User {
2     userExists[fallback_static]
3     userEmail[fail_silent]
4 }
```

Листинг 3.5: Дефинисање зависности између *Order* и *User* микросервиса

3.2.5 Комуникација базирана на порукама

Како би се омогућила комуникација базирана на порукама, потребно је дефинисати скуп порука (`MessagePool`) и бар једног посредника (`MessageBroker`).

Листинг 3.6 приказује дефиницију скупа порука. Поруке унутар овог скупа су груписане унутар `UserMsgGroup` групе, коју чине три поруке: `UserAdded`, `UserUpdated`, `UserDeleted`. Све три поруке садрже следећа поља `userId` и `userEmail`.

Листинг 3.7 приказује дефиницију посредника под именом `Broker`. Овај посредник поседује три канала за поруке: `EV_USER_ADDED_CHANNEL` канал за `UserAdded` поруку, `EV_USER_UPDATED_CHANNEL` канал за `UserUpdated` поруку, те `EV_USER_DELETED_CHANNEL` канал за `UserDeleted` поруку. Приликом инстанцирања канала, потребно је користити његов пуни назив (FQN).

Листинг 3.8 приказује како ажурирати `User` микросервис тако да са остатком система комуницира разменом порука. Као што се може видети, свака од CRUD анотација може да дефинише која порука ће бити послана, којем посреднику, те на који канал. На пример, приликом креирања корисника, порука `UserAdded` из групе `UserMsgGroup` ће бити послана на канал `EV_USER_ADDED_CHANNEL`, који припада `Broker` посреднику (линија 4).

```
1  msg-pool {
2      group UserMsgGroup {
3          msg UserAdded {
4              str userId
5              str userEmail
6          }
7
8          msg UserUpdated {
9              str userId
10             str userEmail
11         }
12
13         msg UseDeleted {
14             str userId
15             str userEmail
16         }
17     }
18 }
```

Листинг 3.6: Пример дефиниције скупа порука у Silvera језику

```
1  msg-broker Broker {
2
3      channel EV_USER_ADDED_CHANNEL(UserMsgGroup.UserAdded)
```

```
4       channel EV_USER_UPDATED_CHANNEL(UserMsgGroup.UserUpdated)
5       channel EV_USER_DELETED_CHANNEL(UserMsgGroup.UserDeleted)
6   }
```

Листинг 3.7: Пример дефинисања скупа порука

```
1   service User {
2
3       api{
4           @create(UserMsgGroup.UserAdded ->
                Broker.EV_USER_ADDED_CHANNEL)
5           @read
6           @update(UserMsgGroup.UserUpdated ->
7                   Broker.EV_USER_UPDATED_CHANNEL)
8           @delete(UserMsgGroup.UserDeleted ->
9                   Broker.EV_DELETED_DELETED_CHANNEL)
10          typedef User {
11              ...
12          }
13          ...
14      }
15  }
```

Листинг 3.8: Микросервис User који комуникацију врши разменом порука

Размену порука је могуће дефинисати и помоћу методама са `@producer` и `@consumer` анотацијама. Листинг 3.9 показује на који начин се користе наведене анотације за дефисање размене порука.

```
1   service User {
2
3       api{
4           typedef User {
5               ...
6           }
7           ...
8           @publish(UserMsgGroup.UserAdded ->
                Broker.EV_USER_ADDED_CHANNEL)
9           void createUser(str username, str password, str email)
10
11          @publish(UserMsgGroup.UserUpdated ->
                Broker.EV_USER_UPDATED_CHANNEL)
12          void updateUser(str username, str password, str email)
13
14          @publish(UserMsgGroup.UserDeleted ->
                roker.EV_DELETED_DELETED_CHANNEL)
15          void deleteUser(str username)
16      }
17  }
```

```
18
19 service EventLogger {
20
21     api {
22         typedef Log {
23             ...
24         }
25
26         internal {
27             ...
28             @consumer(UserMsgGroup.AssignTask <-
                   Broker.EV_USER_ADDED_CHANNEL)
29             void assignTask()
30             ...
31         }
32     }
33 }
```

Листинг 3.9: Микросервиси User и EventLogger комуникацију разменом порука, при чему је начин размене порука дефинисан анотацијама @producer и @consumer

3.3 Silvera преводилац

Silvera преводилац се састоји од два логички одвојена дела, а то су: предњи део (енгл. *front-end*) и задњи део (енгл. *back-end*). *Front-end* се даље дели на модуле за анализу и евалуацију, док *back-end* чини скуп генератора кода везаних за одређени програмски језик.

3.3.1 Front-end

Front-end Silvera преводиоца обавља лексичку анализу, парсирање, те семантичку анализу. Парсер за овај део преводица се динамички креира на основу *textX* алата и граматике Silvera језика.

Парсер парсира Silvera програме и креира граф Python објеката (модел), где је сваки објекат инстанца одговарајуће класе из Silvera метамодела. На овај начин, уместо генерисања апстрактног синтаксног стабла (енгл. *Abstract Syntax Tree - AST*), *textX* креира готов `Model` објекат (Секција 3.1).

Front-end Silvera преводиоца детектује како синтаксне, тако и семантичке грешке. Синтаксне грешке детектује *textX*, током парсирања Silvera програма. Семантичке грешке се детектују током анализе `Model` објекта. С обзиром да

је Silvera језик независан од развојног окружења, грешке се пријављују тек током процеса превођења.

Пре него што се проследи *back-end* делу Silvera преводиоца, над *Model* објектом се ради додатно процесирање. Прво, модел пролази кроз процесор за разрешавање информација везаних за комуникацију (*Communication Resolving Processor - CRP*), а затим кроз процесор за евалуацију архитектуре решења (*Architecture Evaluation Processor - AEP*). С обзиром да микросервиси могу користити различите начине комуникације, циљ CRP-а је да изврши валидацију модела у складу са правилима коришћених комуникационих протола, те да обезбеди додатне информације које су потребне *back-end* делу преводица. Сваки комуникациони стил поседује одговарајући CRP.

Са друге стране, циљ AEP-а је да евалуира модел помоћу унапред дефинисаних метрика. Метрике за евалуацију архитектура базираних на микросервисима дефисане су од стране Богнера и сарадника [107]. Резултати евалуације модела могу се користити у back-end-u за генерисање оптимизованих апликација. Приликом евалуације користе се следеће метрике: *Weighted Service Interface Count (WSIC)*, *Number of Versions per Service (NVS)*, *Services Interdependence in the System (SIY)*, *Absolute Importance of the Service (AIS)*, *Absolute Dependence of the Service (ADS)*, и *Absolute Criticality of the Service (ACS)*.

$WSIC(S)$ [108] је метрика која показује број јавних API функција микросервиса. Ниже вредности $WSIC(S)$ указују на то да ће одржавање микросервиса потенцијално бити лакше. Пошто апсолутне вредности за ову метрику саме по себи нису значајне, потребно је израчунати просечну вредност за целоукупан систем — $WSIC_{AVG}$. Поређењем појединачних вредности са просечном, могуће је идентификовати највећи микросервис у систему. Сервис идентификован као највећи је пожељно разложити на мање микросервисе.

$NVS(S)$ [108] је метрика која показује колико различитих верзија истог микросервиса S је коришћено унутар система. Висока просечна вредност ове метрике, NVS_{AVG}, указује на високу комплексност система. Висока комплексност има значајан утицај на одржавање система.

SIY [109] је метрика која показује број парова микросервиса који су бидирекционо међузависни. Руд и сарадници [109] тврде да постојање парова бидирекционо зависних микросервиса указује на лош дизајн система. Уколико такви парови постоје, једно од могућих решења је спојити их у један микросервис [107].

$AIS(S)$ [109] је метрика која показује број микросервиса који зависе од микросервиса S. Поређењем појединачних AIS вредности са просечном вредношћу система AIS_{AVG} могуће је идентификовати најважније микросервисе

у систему.

$ADS(S)$ [109] је метрика која показује број микросервиса од којих микросервис S зависи. И за ову метрику, такође, је потребно израчунати просечну вредност за целоукупан систем, или ADS_{AVG}.

Метрика $ACS(S)$ комбинује $AIS(S)$ и $ADS(S)$ метрике како би открила потенцијално најпроблематичније делове система. Руд и сарадници [109] тврде да су накритичнији микросервиси са највише клијената, али и који највише зависе од других микросервиса.

Silvera модел се евалуира сваки пут приликом превођења. Пред тога, модел се може евалуирати и путем команде (`silvera evaluate <directory_path>`). На крају процеса евалуације, АЕР креира детаљан извештај са резултатима за сваки микросервис.

Зашто је евалуација модела важна? АЕП можемо упоредити са програмима за анализу кода (енгл. *linters*), као што је *Pylint*[2] за анализу *Python* програмског кода. Неке од предности коришћења програма за анализу кода су следеће [110, 111]: а) стилски доследан кôд, б) рано откривање грешака, в) унапређење перформанси, итд. Слично као и програми за анализу кода, АЕП може идентификовати грешке или делове система који су непотребно сложени. Откривање грешака пре него што се оне појаве у продукцији је значајан бенефит. Поред тога, студија представљена у [112] показује да постојање стандардног скупа правила током развоја софтвера побољшава комуникацију, смањује број грешака, те побољшава квалитет кода.

3.3.2 Back-end

Након што *front-end* заврши са обрадом `Model` објекта, објекат се прослеђује *back-end* делу као улаз.

Back-end Silvera програмског преводиоца итерира кроз сваки модул, те прослеђује декларације (објекти класе `Decl`) генераторима кода. Број генератора кода није ограничен јер Silvera преводилац нуди могућност регистровања екстерних преводилаца (Секција 3.3.3). Ипак, у актуелној верзији Silvera језика (верзија 0.3.1), постоји само један генератор кода уграђен у преводилац. Овај генератор кода трансформише моделе у извршиве Java апликације, базиране на Spring радном оквиру, коришћењем посебних *шаблона* (енгл. *template*). Шаблон представља апстрактну репрезентацију циљног текста [113]. Сваки шаблон се састоји из статичког дела (текст који се у резултујућем тексту појављује у оригиналној форми), те динамичког

[2]PyLint - https://pylint.readthedocs.io/en/stable/

дела (текст који садржи делове метакôда који описују логику генерисања кода) [113]. Пример шаблона је дат на листингу 3.10.

```
1  def my_function(arg1, arg2):
2      {% if operation == "+" %}
3      return arg1 + arg2
4      {% else %}
5      return arg1 - arg2
6      {% endif %}
7
8  # ukoliko je uslov 'operation == "+"' zadovoljen, rezultat je:
9  def my_function(arg1, arg2):
10     return arg1 + arg2
11 # ... u suprotnom, rezultat je:
12 def my_function(arg1, arg2):
13     return arg1 - arg2
```

Листинг 3.10: Пример шаблона. Изрази на линијама 2, 4 и 6 представљају динамички део шаблона и неће се видети у резултујућем тексту (линије 9-13).

Шаблони су веома погодни за итеративни развој софтвера, јер се врло лако могу креирати на основу постојећег кôда. Свака декларација унутар Silvera модела има одговарајући сет шаблона. Који сет шаблона ће бити коришћен зависи првенствено од типа декларације и одабраног циљног програмског језика. У случају уграђеног генератора кода, у тренутној верзији Silvera језика, циљни програмски језик је Java 17. Када је одабран одговарајући сет шаблона, генератор кода анализира декларацију те из ње извлачи податке релевантне за генерисање кода. Ови подаци се потом користе унутар динамичких делова шаблона.

Уграђени генератор кôда генерише SpringBoot апликацију за сваки микросервис из модела. Већина кôда унутар ових апликација се генерише аутоматски, међутим, у неким случајевима је потребна ручна интервенција програмера како би се имплементирала пословна логика. Како би се ручно унесени кôд сачувао између сукцесивних позива генератора кода, уграђени генератор имплементира *generation gap* образац [114]. Употреба овог обрасца омогућује програмерима да на неинвазиван начин унесу ручно писани кôд у апликацију путем интерфејса, при чему класе из ручно писаног кôда имплементирају инферфејсе из генерисаног кôда.

Уграђени генератор кода је имплементиран тако да поштује најбоље праксе дефинисане од стране Хофмана и сарадника [115], тако да свака генерисана SpringBoot апликација има следеће модуле: *domain*, *controller*, *repository*, и *service*. Микросервиси који комуницирају преко порука поседују још два додатна модула: *config* и *messages*.

Модул *domain* поседује класе које специфицирају доменски модел (пословне ентитете) апликације. Ове класе су изведене на основу дефиниција типова које се налазе у API-ју микросервиса. Модул *service* поседује класе које имплементирају пословну логику апликације. Овај модул садржи два подмодула: *base* и *impl*. Модул *base* садржи дефиницију Јава интерфејса са методама наведеним у API-у микросервиса, док модул *impl* садржи класе које имплементирају дате инферфејсе. Модул *impl* се разликује од осталих генерисаних модула у томе што се ручне измене кôда у датотекама унутар овог модула чувају током сукцесивних позива генератора кода. Насупрот томе, остале генерисане датотеке се увек преписују.

Модул *repository* садржи имплементацију *repozitorijum* обрасца (енгл. *repository*), у облику `MongoRepository` објекта који пружа *MongoDB*. Тренутно, уграђени генератор кода има подршку само за MongoDB базу података. Међутим, подршка за произвољну базу података се може додати или проширењем постојећег генератора кода, или регистрацијом новог генератора кода.

Све поруке дефинисане унутар скупа порука се генеришу унутар *messages* модула. У тренутној верзији Silvera језика, поруке се преко мреже шаљу у виду JSON објеката.

Модул *config* садржи класе које дефинишу како ће микросервис комуницирати са брокером. Ове класе су: а) `KafkaConfig` - класа која дефинише на који начин ће се микросервис регистровати у *Kafka* кластер, и б) `MessageDeserializer` - класа која дефинише како ће се JSON порука мапирати на одговарајућу поруку из *messages* модула. Класа `MessageDeserializer` се генерише само у случају да микросервис има методе које очекују поруке од стране брокера.

Модул *controller* садржи класе које имплементирају REST API генерисаног микросервиса. Класе унутар овог модула садрже све методе дефинисане унутар API-а микросервиса, с тим да клијенти не могу приступити методама које су дефинисане као интерне.

За сваки микросервис базиран на REST протоколу, *back-end* генерише OpenAPI документ под називом *openapi.json*. OpenAPI датотеке пружају информације о томе где се API налази, које операције нуди, те шта су очекивани улази и излази, и сл. Поред тога, генеришу се и додатне две датотеке: *pom.xm* - који се користи за управљање зависностима унутар Maven алата, те *bootstrap.properties* - који се користи за подешавање апликација које користе Spring Boot радни оквир.

API пролази и регистри за сервисе се такође генеришу као посебне апликације базиране на Spring Boot радном оквиру. API пролаз се генерише

као *Zuul Proxy*[3] сервер. *Zuul* је API пролаз развијен од стране Netflix тима и пружа услуге динамичког рутирања и надгледања, као и могућности динамичког реаговања на отказе, те дефинисања сигурносних правила. Генерисани код за API пролаз је прост и садржи само једну класу са `main` функцијом, те *application.properties* датотеку која дефинише на који начин ће API пролаз рутирати захтеве, те начин на који ће користити регистар сервиса за проналажење одговарајућег микросервиса. Регистар сервиса је, такође, једноставна апликација са једном класом и *bootstrap.properties* датотеком. Ова датотека специфицира назив регистра за сервисе, HTTP порт на ком ће регистар очекивати захтеве, те да ли ће овај регистар бити регистрован унутар неког другог регистра сервиса.

За сваку генерисану апликацију, уграђени генератор кода генерише специјалну скрипту за покретање и *Docker* датотеку уколико ће се апликација извршавати унутар контејнера. Скрипта за покретање користи *Maven* како би се генерисала и покренула `jar` датотека.

3.3.3 Проширивост Silvera преводиоца

Проширивост Silvera преводиоца се заснива на концептима екстензионе тачке (енгл. *extension point*) и екстензије (енгл. *extension*). Ектензиона тачна дефинише начин на који се ектерни програми могу „уградити" у ту тачку.

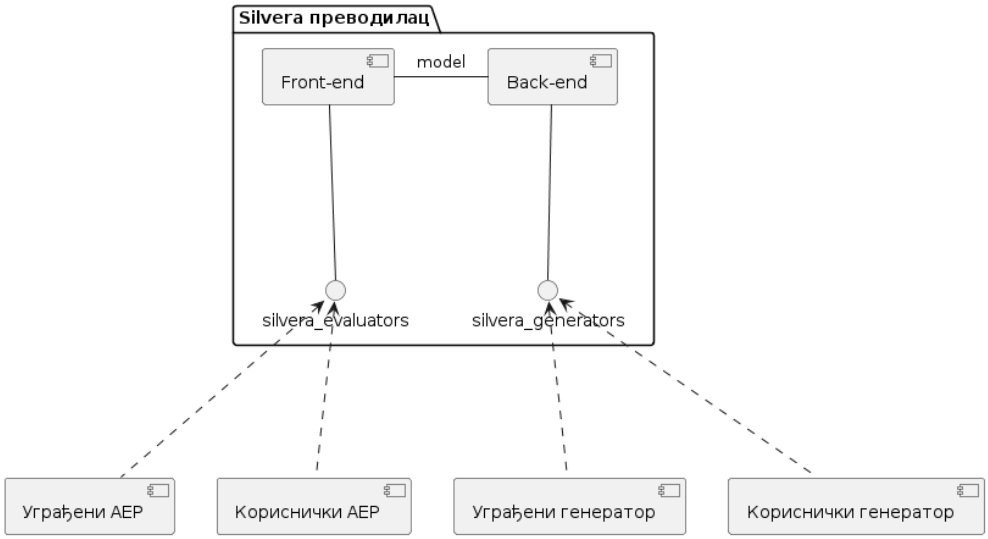
За имплементацију овог механизма коришћен је `pkg_resources` модул који је део модула `setuptools`, доступног унутар програмског језика Python. Silvera поседује екстензионе тачке за регистровање нових генератора кода, и процесора за евалуацију архитектуре (Слика 3.2): Обе тачке су дефинисане унутар `setup.py` датотеке која се налази у главном директоријуму пројекта.

Регистрација екстензија на обе тачке се врши на сличан начин. Стога, у тексту који следи биће описан само процес регистровања новог генератора кода.

Регистрација новог генератора кода се врши у два корака. Први корак је креирање инстанце `GeneratorDesc` класе. Инстанца ове класе поседује информације о циљном језику генератора, опис генератора, те референцу на функцију која обавља генерисање кода. Ова функција (listing 3.11) има три параметра. Први параметар је објекат `Decl` класе. Други параметар представља путању до директоријума где ће кôд бити генерисан, док трећи параметар показује да ли генератор ради у режиму за исправљање грешака.

Циљ другог корака је омогућити Silvera преводицу да детектује нови генератор кода. За то је потребно регистровати објекат класе `GeneratorDesc`

[3]Netlix Zuul – https://github.com/Netflix/zuul



Слика 3.2: Приказ повезивања екстензионих тачака и екстензија.

```
from silvera.generator.gen_reg import GeneratorDesc

def generate(decl, output_dir, debug):
    """Entry point function for code generator.

    Args:
        decl(Decl): can be declaration of service registry or
            config
                    server.
        output_dir(str): output directory.
        debug(bool): True if debug mode activated. False
            otherwise.
    """
    ...

python = GeneratorDesc(
    language_name="python",
    language_ver="3.7.4",
    description="Python 3.7.4 code generator",
    gen_func=generate
)
```

Листинг 3.11: Имплементација *GeneratorDesc* објекта и прототип *generate* функције

на екстензиону тачку `silvera_generators` унутар `setup.py` датотеке (Листинг 3.12).

```
from setuptools import setup

setup(
    ...
    entry_points={
        'silvera_generators': [
            'python=pygen.generator:python'
        ]
    }
)
```

Листинг 3.12: Регистровање новог генератора кода на екстензиону тачку *silvera_generators*

Испис свих доступних генератора кода у текућем развојном окружењу се може добити коришћењем команде (`silvera list-generators`).

Поглавље 4

Студија случаја

Инжењери воле да решавају проблеме. Ако немају проблеме при руци, створиће своје сопствене.

Скот Адамс

У овој секцији биће дат приказ имплементације једноставне апликације под називом е-Ресторан. Апликација е-Ресторан је дистрибуирана апликација која корисницима пружа могућност поруџбине хране путем Интернета. Архитектура ове апликације је у потпуности моделована помоћу Silvera језика. На основу модела је генерисан Java кôд.

4.1 Архитектура апликације е-Ресторан

Апликација е-Ресторан се састоји од пет микросервиса и имплементирана је у сврху приказивања могућности Silvera језика. Микросервиси који сачињавају ову апликацију су приказани у Табели 4.1. API сваког појединачног микросервиса је базиран на REST протоколу. Архитектура ове апликације је илустрована на Слици 4.1.

Како би се ванъским корисницима обезбедила само једна улазна тачка у систем, дефинисан је API пролаз са називом *PristupnaTacka* (Листинг 4.1). Сви микросервиси е-Ресторан апликације су регистровани унутар регистра за сервисе, назван *Registar* (Листинг 4.2). У тексту који следи, биће описан доменски модел и API сваког микросервиса понаособ.

Микросервис *Korisnik* поседује само један ентитет — *Korisnik* који поседује следеће атрибуте: *korisnickoIme*, *lozinka*, те *ePosta*. API овог

```
// datoteka: pritupnatacka.si

import "korisnik.si"
import "jelo.si"
import "porudzbina.si"
import "magacin.si"
import "obavestenja.si"
import "podesavanja.si"


api-gateway PristupnaTacka {
    service_registry = Registar

    deployment {
        version="0.0.1"
        port=9095
        url="http://localhost"
        host=container
    }

    gateway-for {
        Korisnik as /api/kor
        Jelo as /api/jel
        Porudzbina as /api/por
        Magacin as /api/mag
        Obavestenja as /api/oba
    }
}
```

Листинг 4.1: Дефиниција API пролаза под називом *PristupnaTacka*.

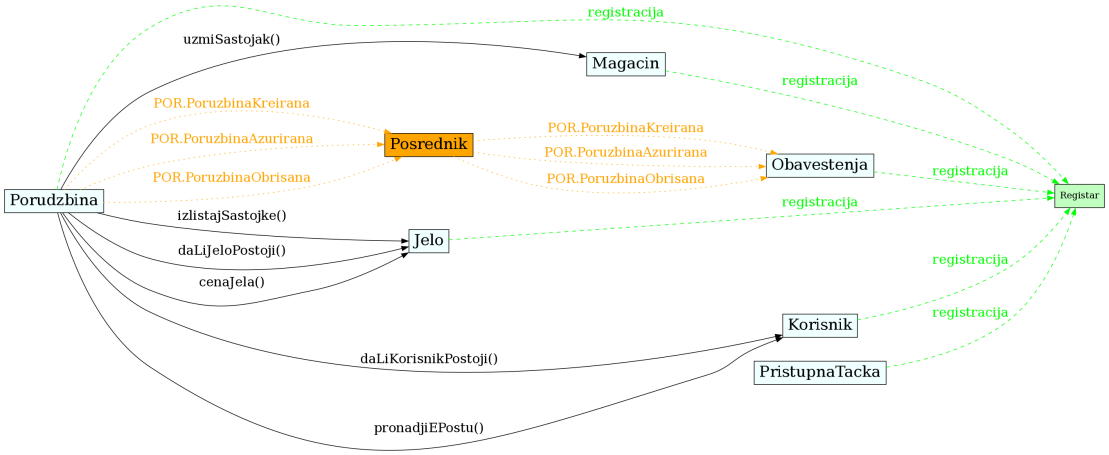
```
// datoteka: podesavanja.si

service-registry Registar {
    client_mode=False
    deployment {
        version="0.0.1"
        port=9091
        url="http://localhost"
        host=container
    }
}
```

Листинг 4.2: Дефиниција регистра за сервисе под називом *Регистар*.

Табела 4.1: Микросервиси од којих се састоји апликација е-Ресторан.

Микросервис	**Опис**
Korisnik	Обезбеђује операције везане за додавање, измену и брисање корисника.
Jelo	Обезбеђује операције за додавање, измену и брисање јела.
Porudzbina	Обезбеђује операције креирање поруџбине.
Magacin	Садржи информације о доступности састојака потребних за спремање јела.
Obavestenja	Обавештава корисника о статусу поруџбине путем е-поште.



Слика 4.1: Архитектура е-Рачунар апликације. Црне, неиспрекидане везе представљају међузависности. Лабеле на овим везама приказују назив методе над којом се дефинише међузависност. Зелене, испрекидане линије представљају регистрацију микросервиса у регистар. Наранџасте, испрекидане везе представљају ток размене порука. Лабеле на овим везама дефинишу назив поруке.

микросервиса се састоји од CRUD метода, као и три додатне методе: а) *izlistajKorisnike* - која враћа податке о свим корисницима унутар система, б) *daLiKorisnikPostoji* - која проверава да ли корисник са датим корисничким именом постоји у систему, те в) *pronadjiEPostu* - која на основу корисничког имена враћа кориснкову е-пошту.

Микросервис *Jela* (Листинг 4.3) поседује два ентитета: *Jelo* и *Sastojak*. Ентитет *Jelo* има атрибуте *naziv*, *cena*, и *sastojci*, док ентитет *Sastojak* поседује атрибуте *naziv* и *kolicina*. API овог микросервиса се састоји од CRUD метода дефинисаних за ентитет *Jelo*, као и три додатне методе: а) *izlistajJela* - која враћа листу свих јела унутар базе података, б) *daLiJeloPostoji* - која проверава да аи јело са датим именом постоји у бази података, те б) *cenaJela* - која враћа цену за јело са датим именом, г) *izlistajSastojke* - која за дато јело враћа listu потребних састојака.

```
// datoteka: jela.si

import "podesavanja.si"

service Jela {

    service_registry=Registar

    api{
        @crud
        typedef Jelo {
            @id str naziv
            @required double cena
            @required list<Sastojak> sastojci
        }

        typedef Sastojak {
            str naziv
            double kolicina
        }

        @rest(method=GET)
        list<Jelo> izlistajJela()

        @rest(method=GET)
        bool daLiJeloPostoji(str naziv)

        @rest(method=GET)
        double cenaJela(str naziv)

        @rest(method=GET)
        list<Sastojak> izlistajSastojke(str name)

    }
}
```

Листинг 4.3: Дефиниција микросервиса *Jela*.

Микросервис *Magacin* поседује само један ентитет — *Sastojak*, који поседује атрибуте *naziv* и *kolicina*. Од метода, API овог микросервиса поседује CRUD методе за ентитет *Sastojak*, као и две додатне методе: а) *izlistajSastojke* - која враћа листу свих тренутно доспутних састојака, и б) *uzmiSastojak* - која узима састојак из магацина, уколико је то могуће.

Микросервис *Porudzbina* (Листинг 4.4) поседује два ентитета: *Porudzbina*, и *StavkaPorudzbine*. Свака поруџбина поседује информације о томе који корисник је креирао поруџбину, затим, листу ставки, те израчунату цену поруџбине. Ентитет *StavkaPorudzbine* поседује информације о порученом јелу, као што су назив јела и количина. API овог микросервиса се састоји од CRUD метода дефинисаних за ентитет *Porudzbina*, те још и методе *izlistajPorudzbine* која враћа листу свих поруџбина из базе података. Методе за креирање, ажурирање, и брисање поруџбине, након успешног извршетка, објављују одговарајуће поруке посреднику за поруке. Ове поруке користи микросервис *Obavestenja* који корисника обевештава о статусу поруџбине.

Микросервис *Obavestenja* (Листинг 4.5) има један ентитет *Obavestenje*, који садржи ID наруџбине и е-пошту корисника као атрибуте. API овог микросервиса има методе *izlistajObavestenja* и *izlistajObavestenjaZaKorisnika*. Прва метода преузима сва обавештења из базе података, док друга метода преузима обавештења само за изабраног корисника. Осим јавно доступних метода, овај микросервис такође садржи три интерне API методе: а) *porudzbinaKreirana*, б) *porudzbinaIzmenjena*, и в) *porudzbinaObrisana*. Ове методе се покрећу након што је наруџбина у микросервису *Porudzbina* креирана, ажурирана или обрисана.

4.2 Евалуација архитектуре апликације

Евалуација архитектуре апликације је извршена током превођења Silvera модела у Јава кôд. Као што приказује Табела 4.2, постоји неколико изузетака за метрике *WSIC*, *AIS*, и *ADS*.

Вредности за метрику *WSIC* код микросервиса *Korisnik* и *Jela* су веће од системског просека. Ова метрика, зарад лакшег одржавања микросервиса, фаворизује вредности које су ниже од системског просека. Ипак, у случају апликације е-Ресторан, овакве вредности за метрику *WSIC* не представљају проблем и очекивано је да имају више јавно доступних API функција од осталих микросервиса у апликацији.

У случају метрике *ADS*, може се уочити да се микросервис *Porudzbina* посебно истиче. На основу вредности ове метрике, може се увидети да је у тренутној имплементацији е-Ресторан апликације, микросервис *Porudzbina*

```
// datoteka: porudzbina.si

// importi...

service Porudzbina {

    service_registry=Registar

    api{

        @create(PorudzbinaPoruke.PorudzbinaKreirana ->
            Broker.KANAL_ZA_KREIRANE_PORUDZBINE)
        @read
        @update(PorudzbinaPoruke.PorudzbinaIzmenjena ->
            Broker.KANAL_ZA_IZMENJENE_PORUDZBINE)
        @delete(PorudzbinaPoruke.PorudzbinaObrisana ->
            Broker.KANAL_ZA_OBRISANE_PORUDZBINE)
        typedef Porudzbina {
            @required str korisnik
            @required list<StavkaPorudzbine> stavke
            double cena
        }


        typedef StavkaPorudzbine {
            @id str nazivJela
            @required i32 kolicina
        }

        @rest(method=GET)
        list<Porudzbina> izlistajPorudzbine()
    }
}

dependency Porudzbina -> Korisnik {
    daLiKorisnikPostoji[fail_fast]
    pronadjiEPostu[fail_fast]
}

dependency Porudzbina -> Jelo {
    izlistajSastojke[fail_fast]
    daLiJeloPostoji[fail_fast]
    cenaJela[fail_fast]
}

dependency Porudzbina -> Magacin {
    uzmiSastojak[fail_fast]
}
```

Листинг 4.4: Дефиниција микросервиса *Porudzbina*.

```
// datoteka: obavestenja.si

import "podesavanja.si"
import "poruke.si"

service Obavestenja {

    service_registry=Registar

    api{

        typedef Obavestenje {
            @required str idPorudbine
            @required str korisnickaEPosta
        }

        @rest(method=GET)
        list<Obavestenje> izlistajObavestenja()

        @rest(method=GET, mapping="/{korisnickaEPosta}")
        list<Obavestenje> izlistajObavestenjaZaKorisnika(str
            korisnickaEPosta)

        internal {
            @consumer(PorudzbinaPoruke.PorudzbinaKreirana <-
                Broker.KANAL_ZA_KREIRANE_PORUDZBINE)
            void porudzbinaKreirana()

            @consumer(PorudzbinaPoruke.PorudzbinaIzmenjena <-
                Broker.KANAL_ZA_IZMENJENE_PORUDZBINE)
            void porudzbinaIzmenjena()

            @consumer(PorudzbinaPoruke.PorudzbinaObrisana <-
                Broker.KANAL_ZA_OBRISANE_PORUDZBINE)
            void porudzbinaObrisana()
        }
    }
}
```

Листинг 4.5: Дефиниција микросервиса *Obavestenja*.

потенцијално најпроблематичнији део система. Заиста, овај микросервис има зависности према три микросервиса из система, а то су: *Korisnik*, *Jela*, и *Magacin*. Ипак, овај микросервис представља централни део апликације и имплементира главни део пословне логике, те су ове вредности очекиване. Како би се повећала поузданост апликације, могуће је увести више инстанци овог микросервиса, или ажурирати апликацију на такав начин да се комуникација између значајних делова извршава разменом порука. С обзиром да не постоји микросервис који директно зависи од микросервиса *Porudzbina*, вредност *AIS* и *ACS* метрике је нула.

Микросервиси *Korisnik*, *Jela*, *Magacin* и *Obavestenja* немају дефинисане зависности ка другим микросервисима. Стога, вредност *AIS* и *ACS* метрике за ове микросервисе је нула.

Табела 4.2: Резултати евалуације архитектуре е-Ресторан.

Метрике	**Системски просек**	**Korisnik**	**Jela**	**Magacin**	**Porudzbina**	**Obavestenja**
WSIC	3.2	**4**	**5.5**	3	1	2.5
NVS	1	1	1	1	1	1
AIS	0.6	**1**	**1**	**1**	0	0
ADS	0.6	0	0	0	**3**	0
ACS	0	0	0	0	0	0

4.3 Генерисани кôд

Као што је наведено у Секцији 3.3, Silvera програмски преводилац може имати произвољан број генератора кода. У случају е-Ресторан апликације, коришћен је уграђени генератор кода.

За сваки од микросервиса је извршено поређење броја линија (енгл. *Lines of code - LOC*) ручно писаног и генерисаног кода, како би се утврдио њихов однос. Прво је израчунат број линија аутоматски генерисаног кода. У ову сврху, коришћен је *cloc*[1] алат отвореног кода. Потом, додат је ручно писани кôд у сваки од микросервиса, након чега је поново покренут *cloc*, те су израчунате разлике у броју линија кода. Као што показује Табела 4.3, велика већина кода је генерисана аутоматски, док је мањи део кода додат ручно.

Кели и Толванен [88] су показали да коришћење ЈСД-ова резултује апликацијама бољег квалитета из два разлога. Први разлог је тај да ЈСД-ови у себи садрже валидације које спречавају корисника да специфицира апликацију на некоректан начин. Тиме се значајно смањује број грешака

[1] cloc – https://github.com/AlDanial/cloc

Табела 4.3: Поређење количине ручно писаног и аутоматски генерисаног кода у апликацији е-Ресторан.

Микросервис	**Silvera**	**Java**		**Maven**		**Укупно**			**% ген. кода**
		Генерисано (LOC)	*Ручно писано (LOC)*	*Генерисано (LOC)*	*Ручно писано (LOC)*	*Генерисано (LOC)*	*Ручно писано (LOC)*	*Укупно (LOC)*	
Korisnik	19	159	8	77	0	236	8	244	96.72
Jela	26	204	10	77	0	154	10	164	93.9
Magacin	16	142	9	77	0	154	9	163	94.48
Porudzbina	37	523	27	81	0	162	27	189	85.71
Obavestenja	24	379	59	81	4	162	63	225	72
Registar	10	11	0	62	0	124	0	124	100
PristupnaTacka	23	14	0	60	0	120	0	120	100

у коду. Свакако, елиминација грешака у почетку је доста „јефтинија" него проналазак и исправка тих истих грешака касније [88]. Други разлог је у томе што генератори кода аутоматски преводе ЈСД спецификације на језике ниже апстракције, те је број потребних ручних интервенција сведен на минимум. Кибурц и сарадници [116] су поређењем апликација писаних ручно и путем ЈСД-ова у контексту тестова који падају, установили да приступ базиран на ЈСД-овима даје коректније резултате.

На основу ове две студије, као и резултата из Табеле 4.3, могуће је закључити да Silvera језик поред убрзања развоја дистрибуираних система базираних на микросервисима, такође, унапређује и њихов квалитет. Унапређење квалитета се базира на следеће две чињенице: а) Silvera поседује валидације које спечавају корисника да специфицира апликацију на некоректан начин, те б) већина кода апликације је аутоматски генерисана и нема потребе да се мења ручно.

S циљем тестирања перформанси Silvera језика, превођење апликације е-Ресторан је извршено десет пута узастопно, уз коришћење уграђене Linux shell команде *time*. Резултати су показали да је за превођење ове апликације у просеку утрошено 0.5 секунди[2].

[2]Тестирање је извршено на лаптоп рачунару са Intel Core™ i7-6700HQ процесором и 8Gb радне меморије.

Поглавље 5

Евалуација Silvera језика

Учење је као веслање узводно,
чим се престане, одмах се креће
назад.

Лао Це

Како би се тестирао Silvera језик, развијено је неколико апликација. Међутим, за објективну процену језика, потребне су повратне информације од корисника који нису учествовали у развоју језика. Стога, спроведена је анкета заснована на FQAD (*A Framework for Qualitative Assessment of DSLs*) радном оквиру [117] како би се утврдило да ли Silvera испуњава зацртане циљеве. FQAD се темељи на ISO/IEC 25010:2011 стандарду и дефинише скуп карактеристика квалитета које треба узети у обзир при дизајнирању ЈСД-а. При процени ЈСД-а може учествовати више заинтересованих страна, при чему свака страна може одабрати неке од следећих карактеристика: функционална погодност, употребљивост, поузданост, одрживост, продуктивност, проширивост, компатибилност, изражајност, поновна искористивост, и интеграбилност.

Евалуација Silvera језика базирана је на методологији приказаној од стране Волина и сарадника [118], а начин извештавања на раду приказаном од стране Једличке и сарадника [119].

5.1 Опсег

Дефинисањем циљева истраживања одређујемо и опсег самог истраживања [118]. За дефинисање циљева истраживања коришћен је GQM шаблон, дефинисан од стране Басилија и Ромбаха [120].

Циљ истраживања је био да се анализира Silvera језик како би се утврдило да ли језик задовољава следеће карактеристике квалитета ЈСД-а: функционална погодност, употребљивост, поузданост, продуктивност, и изражајност.

Истраживање је спроведено уз помоћ студената и програмера са искуством у развоју микросервисних архитектура, или софтверских архитектура уопштено.

5.2 Хипотезе

Пре самог почетка истраживања, дефинисане су следеће хипотезе.

Хипотеза $H0_1$: Silvera није прикладна за спецификацију MCA. Алтернативна хипотеза $H1_1$: Silvera јесте прикладна за спецификацију MCA.

Хипотеза $H0_2$: Елементи Silvera језика нису разумљиви. Алтернативна хипотеза $H1_2$: Елементи Silvera језика јесу разумљиви.

Хипотеза $H0_3$: Концепти и нотација Silvera језика се не могу научити и памтити. Алтернативна хипотеза $H1_3$: Концепти и нотација Silvera језика се могу научити и памтити.

Хипотеза $H0_4$: Silvera није прикладна корисничким потребама за развој MCA. Алтернативна хипотеза $H1_4$: Silvera јесте прикладна корисничким потребама за развој MCA.

Хипотеза $H0_5$: Silvera не штити кориснике од прављења грешака. Алтернативна хипотеза $H1_5$: Silvera штити кориснике од прављења грешака.

Хипотеза $H0_6$: Silvera не скраћује време потребно за развој MCA. Алтернативна хипотеза $H1_6$: Silvera скраћује време потребно за развој MCA.

Хипотеза $H0_7$: Silvera не смањује број људских ресурса потребних за развој MCA. Алтернативна хипотеза $H1_7$: Silvera смањује број људских ресурса потребних за развој MCA.

Хипотеза $H0_8$: Silvera не пружа један и само један добар начин за изражавање сваког концепта од интереса. Алтернативна хипотеза $H1_8$: Silvera пружа један и само један добар начин за изражавање сваког концепта од интереса.

Хипотеза $H0_9$: Неки конструкти Silvera језика не представља тачно један концепта из домена. Алтернативна хипотеза $H1_9$: Сваки конструкт Silvera језика представља тачно један концепта из домена.

Хипотеза $H0_{10}$: Silvera садржи конфликтне елементе. Алтернативна хипотеза $H1_{10}$: Silvera не садржи конфликтне елементе.

Хипотеза $H0_1$ тестира прикладност Silvera језика. Прикладност је подкарактеристика функционалне погодности. Функционална погодност се

односи на степен до ког је ЈСД у потпуности развијен [117]. ЈСД треба да садржи све потребне функционалности из домена, али не треба да садржи функционалности које не припадају домену.

Хипотезе $H0_2$, $H0_3$, и $H0_4$ тестирају разумљивост, лакоћу учења, те корисничку перцепцију језика, респективно. Унутар FQAD оквира, ове карактеристе су подкарактеристике употребљивости. Употребљивост је дефинисана као „мера у којој ЈСД може бити коришћен за постизање одређених циљева".

Хипотеза $H0_5$ тестира поузданост Silvera језика. Кахраман и Билген [117], дефинишу поузданост као „својство језика које помаже при креирању поузданих програма (могућност провере модела, спречавање неочекиваних веза између елемената, и сл.)".

Хипотезе $H0_6$ и $H0_7$ тестирају продуктивност. Продуктивност је карактеристика која се односи на количину ресурса које корисник утроши како би постигао одређене циљеве [117].

Хипотезе $H0_8$, $H0_9$, и $H0_{10}$ тестирају јединственост, ортогоналност, и постојање конфликтних елемената. У FQAD оквиру, ове карактеристике су подкарактеристике експресивности, која се дефинише као „степен до дог се стратегија решавања проблема може природно мапирати у програм".

5.3 Одабир учесника

Циљна популација истраживања били су програмери са искуством у дизајнирању архитектура базираних на микросервисима. Такође су укључени програмери који имају значајно искуство у дизајнирању осталих врста софтверских архитектура, али са мање искуства у области МСА.

Учесници су изабрани на основу њихове доступности и спремности да учествују у истраживању. За учешће у истраживању нису понуђене никакве награде. Сви учесници су идентификовани путем професионалних мрежа аутора. Учесницима је дато тридесет дана да заврше истраживање. За учешће у истраживању није коришћен никакав посебан простор, те су учесници могли да раде од куће, у свом темпу. Учесницима су на располагању били видео туторијали, документација са примерима, и контакт е-пошта аутора у случају питања. Након што су проучили дате материјале, од учесника је тражено да имплементирају једноставну апликацију са и без употребе Silvera језика (Секција 5.4).

Истраживање није било анонимно из два разлога. Прво, било је потребно обезбедити да сваки учесник у истраживању може да учествује само једном. Друго, учесници су могли бити контактирани у случају да је било потребно

затражити додатна објашњења. Сви учесници су били информисани о условима учествовања у анкети, и добили су гаранцију да њихови подаци неће бити подељени са трећим странама.

5.4 Задатак

Задатак који су учесници морали имплементирати је следећи.

> Имплементирати дистрибуирани систем базиран на микросервисима за објављивање научних радова. Систем садржи следеће микросервисе: `User`, `SciPaper` и `Library`. Микросервис `User` омогућава корисницима да се региструју и пријаве. Приликом регистрације, корисник уноси корисничко име (ID), лозинку (обавезно), име (обавезно), презиме (обавезно) и е-пошту (опционо). За пријаву, корисник мора да унесе корисничко име и лозинку. Микросервис `SciPaper` омогућава пријављеним корисницима да пишу нове научне радове. Сваки рад има аутора, наслов и произвољан број секција. Свака секција има свој назив и садржај. Овај микросервис има посебну методу, `publish`, која за дати ID рада објављује `PUBLISH_PAPER` поруку на брокер за поруке. Порука садржи ID рада, наслов рада и аутора. Микросервис `Library` има методу која слуша `PUBLISH_PAPER` поруку и чува податке из поруке у бази података. Овај микросервис такође има јавну методу која излистава све податке из базе података. Корисник није свестан архитектуре апликације и користи `APIgateway` као улазну тачку. Сви сервиси су регистровани у регистру сервиса.

Иако је задатак једноставан, конципиран је тако да обухвата све основне концепте Silvera језика. Да би успешно решили задатак, учесници су морали да науче како да дефинишу микросервис и његов API, API пролаз, те регистар сервиса. Затим, како да региструју микросервис у регистар сервиса, и како да омогуће приступ микросервису кроз API prolaz. Да би успешно имплементирали пословну логику, учесници су морали да користе оба стила комуникације: RPC і размену порука. За комуникацију путем порука, морали су да дефинишу све поруке које се размењују унутар система. Да би имали функционалну RPC комуникацију, корисници су морали да науче како да дефинишу зависности између микросервиса.

Задатак је морао бити имплементиран и у Silvera језику, и ручно. У случају ручне имплементације, учесници су имали пуну слободу да изаберу програмски језик, радне оквире и алате у којима ће имплементирати задату

апликацију. Учесници су бележили време потребно за имплементацију у оба случаја.

5.5 Упитник

Након решавања задатка, учесници су попуњавали online анкету која се састојала од 13 питања, са одговорима представљеним у виду петостепене Ликертове скале. Учесници су могли да одаберу само један одговор, а питања су била подељења у следеће групе: *Искуство*, *Функционална погодност*, *Употребљивост*, *Поузданост*, *Продуктивност*, и *Изражајност*. Питања су представљена у Табели 5.1.

Табела 5.1: Питања из анкете. Сва питања су била затвореног типа и могао се дати само један одговор.

Бр.	**Група питања**	**Питање**
Q1	Искуство	Како бисте описали своје претходно искуство у дизајнирању софтверских архитектура?
Q2	Искуство	Како бисте описали своје претходно искуство у дизајнирању MCA?
Q3	Искуство	Како бисте описали своје претходно искуство са CASE алатима?
Q4	Функционална погодност	Silvera је прикладна за спецификацију MCA.
Q5	Употребљивост	Елементи Silvera језика су разумљиви.
Q6	Употребљивост	Концепти и нотација Silvera језика се могу научити и памтити.
Q7	Употребљивост	Silvera је прикладна корисничким потребама за развој MCA.
Q8	Поузданост	Silvera штити кориснике од прављења грешака помоћу уграђених валидација.
Q9	Продуктивност	Silvera скраћује време потребно за развој MCA.
Q10	Продуктивност	Silvera смањује број људских ресурса потребних за развој MCA.
Q11	Изражајност	Silvera pruža jedan i samo jedan dobar način za izražavanje svakog koncepta od interesa.
Q12	Изражајност	Сваки конструкт Silvera језика представља тачно један концепта из домена.
Q13	Изражајност	Silvera не садржи конфликтне елементе.

За питања из групе *Искуство*, учесници су могли да одаберу следеће одговоре:

1. *Релативно неискусан*,

2. *Средње искусан*,

3. *Релативно искусан*, и

4. *Искусан*.

За питања из осталих група, корисници су могли одабрати неки од следећих одговора:

1. *Уопште се не слажем*,

2. *Не слажем се*,

3. *Не могу да проценим*,

4. *Слажем се*, и

5. *У потпуности се слажем*.

Корисници су, такође, имали могућност да оставе своје коментаре на крају анкете. Подаци прикупљени путем Ликертове скале у истраживању су квантитативни.

5.6 Редослед и правила истраживања

Пре почетка истраживања, дефинисана су следећа правила.

Редослед имплементације задатка. Сваки учесник је задатак (5.4) имплементирао два пута. Једна група корисника је била упућена да задатак имплементира прво коришћењем Silvera језика, док је друга група била упућена да прво имплеменира задатак ручно, коришћењем неког од одабраних програмских језика. Учесници су насумично додељивани у једну од те две групе. На тај начин је избегнут ефекат да учесници имају боље резултате приликом друге имплементације, због познавања проблема који се решава. Учесници су предавали своје имплементације након сваког круга.

Комплетност имплементације. Како би резултати истраживања били валидни, свака имплементација се проверавала путем скупа аутоматизованих тестова. Ови тестови су такође коришћени за рачунање процента у ком је задатак испуњен. Учесници нису имали приступ овим тестовима.

5.7 Анализа и резултати истраживања

Укупно, педесет особа је позвано да учествују у истраживању, од чега су четрдесет и четири прихватиле позив. Међутим, само осамнаест испитаника је заправо учествовало у истраживању. Од тога, четири учесника су програмери са три или више година радног искуства, а четрнаест учесника су студенти завршне године *bachelor* студија. Као што показује Табела 5.2, искуство учесника је варирало, са око 33% испитаника који су се сматрали релативно искусним или искусним у дизајнирању софтверских архитектура, око 17% који су се сматрали релативно искусним или искусним у дизајнирању MCA, те око 5,5% испитаника који су се сматрали релативно искусним или искусним са CASE alatima. Будући да је један од циљева истраживања био испитати да ли програмери који немају искуства са MCA могу лако користити Silvera језик, закључено је да испитаници задовољавају критеријуме истраживања.

Табела 5.2: Преглед одговора (n=18) на питања из групе *Искуство*.

Питање	**Неискусан** (%)	**Релативно неискусан** (%)	**Средње искусан** (%)	**Релативно искусан** (%)	**Искусан** (%)
Q1: Како бисте описали своје претходно искуство у дизајнирању софтверских архитектура?	5.55	5.55	55.55	27.77	5.55
Q2: Како бисте описали своје претходно искуство у дизајнирању MCA?	33.33	33.33	16.67	11.11	5.55
Q3: Како бисте описали своје претходно искуство са CASE алатима?	50.00	38.89	5.55	5.55	0

5.7.1 Анализа перформанси учесника

У Табели 5.3, дат је приказ анализе перформанси учесника са фокусом на ефикасност и ефективност. Сви учесници истраживања су имплементирали задатак (Секција 5.4) и ручно и помоћу Silvera језика (Табела 5.3). Учесницима је помоћ била потребна укупно десет пута, што је резултирало средњом вредношћу од *0.98*, као што је приказано у Табели 5.3). Помоћ учесницима је пружана вербално, преко Интернета, при чему су учесницима

дате инструкције да паузирају сат за мерење времена имплементације у периоду док су чекали на одговор.

Просечно време завршетка имплементације задатка када се користи Silvera језик било је нешто испод 3.5 сата. У просеку, учесници су имплементирали задатак ∼124% брже када су користили Silvera језик него када су исти задатак имплементирали ручно. Дужине имплементације су упоређене и статистички. Помоћу *R* језика, урађен је *t-test* са два узорка како би се упоредиле средње вредности времена потрошених на имплементацију. На овај начин је тестирано да ли Silvera језик заиста убрзава развој МСА, при чему је добијена $p\text{-}vrednost(p - vrednost = 5.539e - 05)$ која је нижа од нивоа значајности ($\alpha = 0.01$). Статистика теста и величина ефекта износе *V=-5.185483* и *0.91*, респективно. С обзиром да ниво стручности учесника у МСА варира у већој мери, високе вредности стандардне девијације за времена имплементације задатка су очекиване.

Комплетност имплементација је мерена извршењем скупа аутоматизованих тестова. Као што приказује Табела 5.3, оба приступа дају сличне резултате, при чему Silvera језик у просеку има нешто већу меру извршења задатка од ∼95% у односу на ∼91% код ручног приступа. Док је максимална вредност за меру извршења задатка идентична за оба приступа, минимална вредност за меру извршења задатка је нешто већа код ручног приступа. Минимална вредност мере извршења задатка је нижа од очекиване јер неки од учесника нису именовали своје функције онако како је наведено у задатку, што је довело до пада одређених тестова. Медијана за оба приступа је идентична. Међутим, *модус* (вредност која се најчешће појављује) показује да је већина учесника имала меру извршења задатка од 100% при коришћењу Silvera језика. Заиста, 50% учесника је у потпуности тачно имплементирало задатак при коришћењу Silvera језика, насупрот 33% код ручног приступа. Модус за колоне везане за време извршавања није израчунат, јер су сви корисници пријавили јединствена времена извршавања.

Табела 5.3: Анализе перформанси учесника са фокусом на ефикасност и ефективност.

	Средња вредност	**Медијана**	**Модус**	**Стд. Дев.**	**Мин**	**Макс**
Мера извршења задатка - Silvera	94.84%	89.26%	100%	8.06%	71.42%	100%
Мера извршења задатка - ручно	91.26%	89.26%	85.71%	7.96%	78.57%	100%
Број асистенција	0.55	2	0	0.98	0	3
Време утрошено на имплементацију задатка - Silvera (сат:мин:сек)	03:21:56	03:08:12	-	01:16:07	01:13:24	06:45:23
Време утрошено на имплементацију задатка - ручно (сат:мин:сек)	07:30:31	06:00:05	-	03:48:31	02:55:25	17:23:20

Као што је приказано Табели 5.4, већина учесника сматра да је Silvera језик функционано погодан за спецификацију дистрибуираних система

Табела 5.4: Процентуални приказ одговора (н=18) за следеће групе питања: *Функционална погодност*, *Употребљивост*, *Поузданост*, *Продуктивност*, і *Изражајност*.

Питање	**Уопште се не слажем (%)**	**Не слажем се (%)**	**Не могу да проценим (%)**	**Слажем се (%)**	**У потпуности се слажем (%)**
Q4: Silvera је прикладна за спецификацију МСА.	0	0	5.56	72.22	22.22
Q5: Елементи Silvera језика су разумљиви.	0	0	0	27.78	72.22
Q6: Концепти и нотација Silvera језика се могу научити и памтити.	0	0	0	33.33	66.67
Q7: Silvera је прикладна корисничким потребама за развој МСА.	0	0	11.11	55.56	33.33
Q8: Silvera штити кориснике од прављења грешака помоћу уграђених валидација.	0	0	11.11	61.11	27.78
Q9: Silvera скраћује време потребно за развој МСА.	0	0	5.56	33.33	61.11
Q10: Silvera смањује број људских ресурса потребних за развој МСА.	0	5.56	11.11	66.67	16.67
Q11: Silvera пружа један и само један добар начин за изражавање сваког концепта од интереса.	0	11.11	22.22	55.56	11.11
Q12: Сваки конструкт Silvera језика представља тачно један концепта из домена.	0	0	11.11	77.78	11.11
Q13:Silvera не садржи конфликтне елементе.	0	0	5.56	72.22	22.22

базираних на микросервисима. Међутим, учесници су уочили и следеће проблеме:

- *Грешка у синтакси.* Један од учесника је уочио да генерисани кôд у његовом случају није могао да се изврши, због синтаксне грешке. Грешку је узроковала неисправна имплементација унутар шаблона, те је ова грешка одмах уклоњена.

- *Форматирање генерисаног кôда.* Неколико учесника се жалило на форматирање генерисаног кôда. У појединим случајевима, размак између линија кôда је био превелики. Овај проблем је решен убрзо након пријављивања. Свакако, у будућим верзијама Silvera језика посебна пажња ће бити обраћена на форматирање генерисаног кôда.

- *Недовољно искоришћење могућности Spring Boot радног оквира.* Неки

учесници су приметили да Spring Boot радни оквир поседује уграђене механизме који би могли поједноставити неке делове генерисаног кода. На пример, за имплементацију комуникације између микросервиса, могуће је искористити механизам *Feign client*.

- *Документација.* Један учесник је прокоментарисао да видео туторијали и документација нису ажурни. Видео туторијали су снимљени за претходну верзију Silvera језика, али су, међутим, све битне разлике назначене и у видео материјалима и у документацији. Ипак, на ажурност документације и видео материјала биће обраћена посебна пажња у идућим верзијама језика.

5.7.2 Дескриптивна статистика

Помоћу R [121] језика, израчунате су мере централне тенденције и мере дисперзије одговора. Медијана (50%-перцентил) за свако питање има вредности 4 или 5, као што је приказано у Табели 5.5. У зависности од питања, вредности модуса су такође 4 или 5. Ове вредности медијане и модуса указују на то да су подаци симетрично распоређени [118]. Ниске вредности интерквантилних разлика указују на то да постоји концензус међу учесницима да Silvera језик задовољава следеће карактеристе квалитета: функционална погодност, употребљивост, поузданост, продуктивност, и изражајност.

5.7.3 Тестирање хипотеза

Тестирање хипотеза је извршено помоћу Вилкоксоновог теста означених рангова са једним узорком [122]. Овај тест утврђује постоји ли статистички значајна разлика између посматраног и подразумеваног узорка, и изабран је због тога што су подаци из Ликертове скале редни, и не могу се нормално дистрибуирати. За свако питање из анкете, проверено је да ли је вредност теста већа од подразумеване вредности. За подразумевану вредност је одабран број 3. Да би се контролисала стопа грешке у целом истраживању, коришћена је Бонферонијева корекција, тако што је свака израчуната *p-вредност* помножена са бројем хипотеза ($n_{hip} = 13$).

Подаци из Табеле 5.6 показују да је p-вредност сваког питања нижа од нивоа значајности ($\alpha = 0.05$). На основу ових резултата, нулте хипотезе ($H0_1$ - $H0_{10}$) су одбачене и прихваћене алтернативне хипотезе ($H1_0$ - $H1_{10}$). Стога, утврђено је да Silvera језик задовољава следеће карактеристике квалитета: функционална погодност, употребљивост, поузданост, продуктивност, и изражајност.

Табела 5.5: Мере централне тенденције и мере дисперзије одговора (н=18). Вредности *медијана* и *модуса* показују мере централне тенденције, док *интерквантилне разлике* показују дисперзију одговора.

Питање	**Kvantili**					**Модус**	**Интерквантилне разлике**
	0(%)	25(%)	50(%)	75(%)	100(%)		
Q4: Silvera је прикладна за спецификацију МСА.	3	4	4	4	5	4	0
Q5: Елементи Silvera језика су разумљиви.	4	4.25	5	5	5	5	0.75
Q6: Концепти и нотација Silvera језика се могу научити и памтити.	4	4	5	5	5	5	1
Q7: Silvera је прикладна корисничким потребама за развој МСА.	3	4	4	5	5	4	1
Q8: Silvera штити кориснике од прављења грешака помоћу уграђених валидација.	3	4	4	4.75	5	4	0.75
Q9: Silvera скраћује време потребно за развој МСА.	3	4	5	5	5	5	1
Q10: Silvera смањује број људских ресурса потребних за развој МСА.	2	4	4	4	5	4	0
Q11: Silvera пружа један и само један добар начин за изражавање сваког концепта од интереса.	2	3	4	5	5	4	1
Q12: Сваки конструкт Silvera језика представља тачно један концепта из домена.	3	4	4	4	5	4	0
Q13: Silvera не садржи конфликтне елементе.	3	3	4	4	5	4	0

5.8 Претње по валидност истраживања

Циљ истраживања је био развити језик који ће убрзати развој дистрибуираних система базираних на микросервисима. Иако је истраживање спроведено темељно, постоје одређене претње по валидност истраживања које ће бити дискутоване у овој секцији. У тексту који следи, биће представљене претње по конструктивну, интерну, и екстерну валидност.

5.8.1 Конструктивна валидност

Конструктивна валидност се односи на генерализацију резултата истраживања на концепт или теорију која стоји иза истраживања [118]. Неке претње се односе на дизајн истраживања, док се друге односе на социјалне факторе.

Претње по валидност, социјалне природе, тичу се проблема везаних за

Табела 5.6: Вредности статистике теста, p-вредност и величина ефекта за свако питање из група *Функционална погодност*, *Употребљивост*, *Поузданост*, *Продуктивност*, і *Изражајност*.

Питање	**Статистика теста**	**p-вредност**	**Величина ефекта**
Q4: Silvera је прикладна за спецификацију MCA.	V = 153	0.0008	0.98
Q5: Елементи Silvera језика су разумљиви.	V = 171	0.0006	1.00
Q6: Концепти и нотација Silvera језика се могу научити и памтити.	V = 171	0.0006	0.99
Q7: Silvera је прикладна корисничким потребама за развој MCA.	V = 136	0.002	0.93
Q8: Silvera штити кориснике од прављења грешака помоћу уграђених валидација.	V = 153	0.002	0.94
Q9: Silvera скраћује време потребно за развој MCA.	V = 129	0.001	0.96
Q10: Silvera смањује број људских ресурса потребних за развој MCA.	V = 92	0.004	0.86
Q11: Silvera пружа један и само један добар начин за изражавање сваког концепта од интереса.	V = 136	0.045	0.68
Q12: Сваки конструкт Silvera језика представља тачно један концепта из домена.	V = 153	0.0008	0.97
Q13: Silvera не садржи конфликтне елементе.	V = 120	0.0007	0.98

понашање учесника у истраживању. Учесници који учествују у истраживању могу да се понашају другачије током истраживања, него што би то радили иначе. Претње овог типа које су могле да угрозе резултате истраживања су: *нагађање хипотезе*, те *страх од евалуације*.

Нагађање хипотезе. Ова претња по валидност огледа се у томе да учесници у истраживању могу покушати да одгонетну шта су циљ и очекивани резултати истраживања, те да дају своје одговоре у складу са тим [118]. Иако учесници у истраживању нису били свесни постављених хипотеза, постојала је могућност да дају одговоре који би фаворизовали Silvera језик. Ова претња по валидност је избегнута (или умањена), тако што су учесници охрабривани да дају искрене одговоре. Учесници су били свесни да ће једино на тај начин Silvera језик бити прецизно евалуиран, што даје прецизније смернице за будући развој језика.

Страх од евалуације. Људи имају тенденцију да покушају да дају боље резултате током евалуације, што може да утиче на резултате истраживања. Чињеница да истраживање није било анонимно могло је да изазове страх од евалуације. Како би се избегао овај ефекат, учесници су обавештени да нису они ти који се евалуирају, него Silvera језик. Такође, учесници нису били свесни других учесника, нити њихових резултата (сваки учесник је радио од куће, својим темпом), како би се избегла било каква форма такмичења између

њих.

5.8.2 Интерна валидност

Интерна валидност представља сигурност да су закључци изведени током истраживања примењиви на случајеве који се истражују. Претње по интерну валидност, у погледу узрочности, могу утицати на закључак истраживања без знања истраживача. Претње по интерну валидност, релевантне за истраживање, биће описане у даљем тексту.

Сазревање. Дужина истраживања може на различите начине да утиче на учеснике у истраживању. Како би успешно одговорили на питања из упитника, учесници су прво морали да се упознају са Silvera језиком преко видео туторијала, документације, и примера. Након тога, од учесника се захтевало да имплементирају једноставну апликацију са и без употребе Silvera језика (Секција 5.4). Сви ови кораци су временски захтевни, те су учесницима узимали доста времена што је могло довести до незаинтересованости и досаде. Како би се ублажила ова претња по валидност, задатак је осмишљен тако да буде што једноставнији. Такође, документација је структуирана на начин да учесници могу брзо да подесе окружење за рад.

Инструментација. У Секцији 4.3 показано је да је већина кода, у случају приказане апликације, генерисана аутоматски те да нема потребе да се мења. Ипак, однос између количине аутоматски генерисаног и ручно писаног кода зависи од стила програмирања и искуства програмера. Такође, резултати би могли бити другачији и у случају да је коришћен други алат за рачунање броја линија кода. Даље, иако презентована апликација покрива све концепте Silvera језика, количина ручно писаног кода би могла бити другачија у за неку другу апликацију.

5.8.3 Екстерна валидност

Претње по екстерну валидност директно утичу на могућност генерализације резултата истраживања на ширу популацију. Претње по екстерну валидност, релевантну за истраживање, биће описане у даљем тексту.

Селекција учесника. Само осамнаест учесника је учествовало у истраживању. Разлог мало броја учесника може бити у томе што је потребно доста времена како би се учесници упознали са Silvera језиком и успешно урадили задатак и испунили упитник. Такође, циљ је био прикупити учеснике који имају доста искуства у развоју апликација базираних на микросервисима, међутим ниво искуства учесника у истраживању је различит. Стога, може

се закључити да би резултати истраживања могли бити другачији у случају већег броја учесника, или у случају већег броја искуснијих учесника.

Околности. Утицај на резултате истраживања је такође могла имати и комплексност задатка који је било потребно имплементирати. Задатак је осмишљен тако да буде што једноставнији, али да покрива све основне концепте Silvera језика (Секција 5.4). На тај начин, учесници су могли даураде задатак у разумном временском року. Међутим, једноставност задатка је на резултате истраживања могла утицати на разне начине. На пример, тестирање Silvera језика на комплекснијим задацима би реалније показало подобност за домен ($H0_1$) и употребљивост ($H0_2 - H0_4$) Silvera језика. Додатно, очекивано је да је број грешака код једноставнијих апликација мањи него код комплекснијих апликација. Стога, комплекснији задатак би дао бољи увид у то да ли Silvera штити кориснике од прављења грешака ($H0_5$), какав је утицај на продуктивност ($H0_6$) и алокацију ресурса ($H0_7$). Недостатак интегрисаног развојног окружења је, такође, могао имати утицај на резултате истраживања. Учесници су могли користити произвољан текстуални едитор за имплементирање задатка, међутим, због тога што Silvera није интегрисана ни са једним од постојећих едитора, грешке су се могле уочити тек током процеса компајлирања. Како би се ублажио овај проблем, посебна пажња је посвећена детаљном пријављивају грешака, као и перформансама Silvera преводиоца. Ово је омогућило учесницима да брзо открију и поправе грешке. Ипак, како би се унапредило корисничко искуство, планирана је подршка за интеграцију са популарнијим тескстуалним едиторима.

Због присутних претњи по екстерну валидност, не можемо генерализовати резултате истраживања на друге групе испитаника, нити другачије околности спровођења истраживања. Стога, планирана су детаљнија истраживања у будућности.

Поглавље 6

Ограничења Silvera језика

Стално побољшање је боље од одгођеног савршенства.

Марк Твен

Тренутна имплементација Silvera језика има неколико ограничења. Иако за већину примећених ограничења постоје начини да се она заобиђу, њиховим систематским решавањем би се значајно унапредило корисничко искуство као и квалитет генерисаних апликација. У тексту који следи биће описана сва примећена ограничења, као и могући начини унапређења Silvera језика како би се ова ограничења одстранила.

6.1 Децентрализован начин развоја модела

Silvera даје најбоље резултате када је читав систем описан у моделу. Међутим, то не значи нужно да је Silvera централизовано решење. Током развоја дистрибуираних система базираних на микросервисима, неки тимови се могу одлучити да користе Silvera jezik, док други тимови могу да одаберу неки други језик. Овај приступ има неколико ограничења, јер су информације о систему распршене на неколико Silvera модела, или других начина спецификације MCA. Ова ограничења ће бити описана у тексту који следи.

Недостатак информација о међузависностима. Микросервиси дефинисани унутар Silvera модела не могу дефисати зависности ка микросервисима дефинисаним изван модела. Као резултат тога, процесор за евалуацију архитектуре (Секција 3.3.1), који зависи о информацијама о међузависностима, може да да некомплетне или нетачне резултате евалуације. Такође, Silvera

преводилац на основу веза о међузависностима генерише кôд за комуникацију између микросервиса. У овом случају, тај кôд би се морао додати ручно.

Регистар за сервисе није део Silvera модела. У овом случају, регистар за сервисе се налази унутар другог модела, или је имплементиран ручно. Микросервиси који су дефинисани унутар модела где не постоји дефиниција регистра за сервисе не могу се аутоматски регистровати унутар регистра. Због тога, кôд за регистрацију би се морао додати ручно.

API улаз није део модела. У овом случају, конфигурација API улаза се мора ажурирати ручно у случају било каквих измена на API-у микросервиса који су део Silvera модела.

Некомплетна дефиниција API улаза. У овом случају, API улаз и микросервиси су или унутар различитих модела или су неки од њих имплементирани ручно. Микросервиси који нису део истог модела као API улаз, нису „видљиви" API улазу. Стога, конфигурација API улаза се мора ажурирати ручно, као и у предходном случају.

Неке од порука нису дефинисане. Ако постоји више Silvera модела, или су неки од микросервиса имплементирани ручно, сваки од делова система поседује само мали део укупног скупа порука који се користе унутар система. Због уграђене валидације унутар Silvera језика, немогуће је дефинисати произвођача или потрошача за одређену поруку уколико та порука није дефинисана унутар модела. Због тога, сваки скуп порука мора бити ажуриран са недостајућим порукама. С обзиром на то да сваки модел има свој скуп порука, имамо да је један развојни тим задужен за један модел/скуп порука. Тимови могу кроз документацију комуницирати које поруке се користе. Иако овај приступ решева проблем порука које нису дефинисане, може створити други проблем у ком иста порука, због редудантности, треба да се ажурира у више скупова порука.

Иако постоје начини за превазилажење горе поменутих проблема, планирано је темељно истраживање како би се ови проблеми решили систематски. На пример, механизам за увоз декларација (микросервиса, регистара услуга, и сл.) из других модела би могао омогућити да се апликација опише путем више Silvera модела. Овај механизам би морао да ради и у случајевима када су модели на удаљеним машинама, и не сме да нарушава перформансе.

6.2 Недостатак интегрисаног развојног окружења

ЈСД-ови су најефикаснији када имају подршку у виду специјализованих алата, укључујући, али не и ограничавајући се на, интегрисано развојно окружење (IDE) са едиторима прилагођеним језику.

Тренутно, Silvera нема подршку за IDE, што може значајно утицати на корисничко искуство у случају сложенијих модела. Додавање функционалности попут аутоматског довршавања (енгл. *auto-complete*), скока на дефиницију (енгл. *go to definition*), или приказа документације приликом преласка курсором (енгл. *documentation on hover*) за програмски језик захтева значајан труд. Традиционално, овај посао се понављао за сваки развојни алат, јер сваки алат пружа различит API за имплементацију исте функционалности. Како би се Silvera интегрисала у више развојних окружења, планирана је имплементација подршке за протокол језичког сервера (енгл. *Language Server Protocol - LSP*). LSP стандардизује начин на који језички сервер, коју пружа „језичку интелигенцију", комуницира са развојним окружењем. За све ЈСД-ове засноване на *textX* алату, подршка за имплементацију језичког сервера се пружа путем *textX-LS* пројекта са отвореним кôдом.

6.3 Пословна логика није део модела

Због проширивог програмског преводиоца, Silvera може подржати произвољан број генератора кода. Сваки генератор може генерисати кôд у специфичном програмском језику. Ова карактеристика Silvere, пружа могућност корисницима да брзо мигрирају имплементацију микросервиса са једног на други програмски језик (на пример, са Java програмског језика на Python). Међутим, пословна логика имплементирана у појединим API функцијама ипак мора да се мигрира ручно. Ове измене нису видљиве на нивоу модела, јер се обављају на нивоу кода.

Постоје два начина која омогућавају дефинисање пословне логике на нивоу модела. Први начин је проширење Silvera језика са императивним језиком за дефинисање акције (енгл. *action language*). Овај језик би требао да има подршку за стандардне језичке конструкције као што су декларације, инструкције, и изрази. Уобичајена критика ЈСД-ова је да захтевају редизајн таквих ствари попут инструкција и израза, што је тешко направити исправно и потпуно [79]. Друга могућност би била да се понуди опција уградње (енгл. *embedding*) кода циљног језика у модел. Предност овог приступа је у томе што би корисници имали доступну пуну експресивност циљног језика, али је

мана у томе што ограничава преносивост на друге програмске језике.

6.4 Додатни дизајн обрасци за МСА

У Секцији 1.2.9.5 су описани тренутно имплементирани обрасци за дизајн МСА. Међутим, број имплементираних образаца је могуће и повећати. У овој секцији, биће размотрени дизајн обрасци који недостају.

Безбедност. Iако развој дистрибуираних система у облику МСА има доста предности, безбедност и даље представља један од највећих изазова [123]. Iпак, због филозофије приступа, микросервиси поседују неколико карактеристика које су важне за безбедност: а) микросервиси обично имају малу кодну базу, што доводи и до мање „површине" за напад, б) мало је вероватно да ће се малициозне измене убачене од стране нападача у одређену инстанцу микросервиса задржати након поновног покретања те инстанце, уколико је микросервис непромењив (енгл. *immutable*), в) микросервиси приступају само оним подацима и сервисима који су им потребни, што умањује штету уколико би одређена инстанца била компромитована, и г) хетерогеност технологија штити микросервисе од експлоатација на ниском нивоу. Ипак, у индустрији су имплементиране следеће безбедносне праксе [124]: а) Mutual Transport Layer Security (MTLS) са сопственом инфраструктуром јавних кључева (енгл. *Public Key Infrastructure - PKI*) као метод за заштиту интерне комуникације међу микросервисима, и б) Безбедносни токени (енгл. *Security tokens*) за локалну аутентификацију. Тренутно, Silvera генерише такав кôд где је ове праксе неопходно имплементирати ручно. План је проширити Silvera језик тако да генерише кôд са свим најбољим безбедносним праксама.

Управљање подацима. Трансакције које обухватају више микросервиса морају се имплементирати ручно. То је директна последица немогућности изражавања пословне логике у Silvera моделима (Секција 6.3). Када пословна логика постане део модела, биће омогућена имплементација дизајн образаца као што су: Saga [8], Command Query Responsibility Segregation (CQRS) [24], и API composer [24]. Осим тога, микросервиси се дизајнирају тако да немају стање (енгл. *stateless*). Овакви микросервиси су аутономни и изоловани. Међутим, дистрибуиране апликације често морају одржавати глобално стање (као што је показано у [125]). Начин на који ЈСД може обезбедити очување глобалног стања и постизање концензуса у МСА, показано је у [126].

6.5 Проширивање Silvera језика

Циљ ЈСД-а није да реше све врсте рачунарских проблема, нити чак велике класе проблема из одређеног домена [79]. ЈСД је изражајан за проблеме који спадају у дати домен и не може се користити за решавање других проблема. ЈСД чак може бити неадекватан и за проблеме који се налазе на самом рубу домена. Због оваквих проблема, обично постоји притисак да се ЈСД прошири и прерасте свој оригинални домен [79].

Тешко је предвидети скуп концепата који ће бити потребни током животног века програмског језика. Са изменама у домену за који је дефинисан, долази и до промене ЈСД-а. Silvera језик је дизајниран на начин који лако омогућава да се временом прилагођава потребама МСА. Нови концепти се могу додати проширивањем граматике и Silvera преводиоца. Међутим, ово такође може бити и проблем. Увођењем великог броја нових концепата, језик може постати претрпан (енгл. *bloated*), што може значајно утицати на одржавање језика, као и на могућност корисника да брзо овладају концептима језика. Ипак, имплементација подршке за одређену технологију или радни оквир на нивоу преводица је олакшан, јер се генератори кода могу регистровати унутар Silvera преводица (Секција 3.3.3).

6.6 Уграђени генератор кода

Током евалуације Silvera језика, уочена су два недостатка уграђеног генератора кода.

Трансформације модела у текст (енгл. *model-to-text transformations).* Уграђени генератор кода (Секција 3.3.2) користи трансформације модела у текст базиране на шаблонима за генерисање Java апликација базираних на Spring Boot радном оквиру. Значајан недостатак генератора кода базираних на шаблонима је у томе што такав генератор нема увид у структуру генерисаног кода [81]. Самим тим, није могуће извршити никакве валидације или трансформације над резултујућим кодом. Предност овог приступа је у томе што се генератор кода може брзо имплементирати, зашто је овај приступ имплементације и одабран.

Недостајуће функционалности. Увек постоји могућност да генерисани кôд не покрива све функционалности доступне у Spring Boot радном оквиру.

Ипак, корисници могу да превазиђу наведене проблеме изменом постојећег генератора кода, или регистрацијом нових (Секција 3.3.3).

6.7 Метрике за евалуацију архитектуре

Приликом евалуације моделоване архитектуре, Silvera језик користи метрике које су намењене за евалуацију сервис-оријентисаних система [107]. Ове метрике, такође, су примењиве на архитектуре базиране на микросервисима [107].

У време писања овог текста, коришћене метрике нису емпиријски евалуиране на индустријском нивоу како би се одредили да ли су комплетне или погодне за МСА. Метрике $WSIC(S)$ и $NVS(S)$ су евалуиране у само једној студији, али у контексту СОА [108]. Метрике $AIS(S)$, $ADS(S)$, и $ACS(S)$ грубо одговарају метрикама за мерење спреге између класа (енгл. *class coupling*) и објеката (енгл. *object coupling*) које се користе у објектно-оријентисаном програмирању. Постоји неколико емпиријских студија где су евалуиране метрике спреге класа и објеката. Ове студије су представљене у [127].

Поред коришћеног скупа метрика, постоји још неколико различитих скупова метрика који су или дизајнирани специфично за МСА, или су примењиве на тај стил [128–130] те могу бити коришћене током евалуације. Додатно, искусни програмери могу намерно занемарити резултате евалуације. На пример, микросервис може зависити од неколико других микросервиса јер садржи имплементацију главне пословне логике. Програмери могу чак измислити свој сопствени скуп метрика које примењују у својим пројектима. Како би се омогућила флексибилност при избору метрика, Silvera омогућава корисницима да региструју своје функције за евалуацију (Секција 3.3.3).

Закључак и правци даљег развоја

Предмет истраживања дисертације припада области дистрибуираних система, као и области инжењерства софтверских језика.

Дистрибуирани системи су све присутнији у савременом рачунарству јер крајњим корисницима доносе многе бенефите као што су побољшана доступност и приступачност. Међутим, дизајнирање савремених дистрибуираних система може бити веома изазовно због њихове инхерентне сложености и потребе за координацијом између више делова система.

У области дистрибуираних система посебну пажњу у послењих неколико година привлаче архитектуре базиране на микросервисима јер су системи базирани на овим архитектурама флексибилнији, лакше скалирају, и једноставнији су за одржавање. Ипак, имплементација система базираних на микросервисима са собом доноси и нове изазове. Овакви сустеми су комплекснији, тежи за тестирање и документовање, те од програмера изискију познавање широког дијапазона технологија (Секција 1.2.9.4).

Циљ ове дисертације био је креирати језик који би убрзао развој дистрибуираних система базираних на микросервисима. Креирани језик, Silvera, се одликује једноставном синтаксом која је лака за учење и разумевање. Једноставна, али семантички богата нотација обезбеђује бржи начин развоја микросервиса. Поред тога, Silvera језик је дизајниран на начин да подржава коришћење великог броја различитих технологија, што програмерима даје могућност одабира технологије која је најприкладнија проблему који решавају. Додатно, аутоматским генерисањем документације, језик решава проблем застареле документације која је обично последица честих измена у системима базираних на микросервима. Такође, језик нуди и могућност евалуације архитектуре креираног система помоћу метрика специјално намењених домену микросервисних архитектура.

Како би се објективно проценио Silvera језик, спроведена је анкета заснована на FQAD радном оквиру. Од учесника је тражено да имплементирају једноставан задатак са и без коришћења Silvera језика, а потом да попуне упитник. Укупно, осамнаест испитаника је учествовало у анкети. Резултати анкете јасно показују да Silvera језик драстично убрзава

развој дистрибуираних система базираних на микросервисима. У просеку, учесници су имплементирали задатак ∼124% брже када су користили Silvera језик (Секција 5.7.1). Додатно, учесници су коришћењем Silvera језика креирали и квалитетније програме. Чак 50% учесника је у потпуности тачно имплементирало задатак при коришћењу Silvera језика, насупрот 33% у случају када нису користили Silvera језик.

Правци даљег истраживања и развоја могу бити следећи:

- Детаљније истраживање које би евалуирало Silvera језик на примерима из индустрије. Од посебног интереса би било сазнати у којој мери Silvera језик утиче на продуктивност и квалитет микросервисних апликација у случајевима када сваки тим имплементира по један микросервис. Такође, било би значајно сазнати и у којој мери евалуација креираног система, базирана на метрикама, доприноси квалитету имплементираног система;

- Имплементација механизма за увоз декларација из других модела. Имплементација овог механизма обезбедила би бољу модуларност Silvera модела, као и потенцијално поновну искористивост одређених делова модела;

- Интеграција Silvera језика са неким од графичких едитора;

- Имплементација додатних дизајн образаца из области MCA. Од посебног интереса су обрасци везани за безбедност и управљање подацима.

- Проширење Silvera језика са концептима који би омогућили опис пословне логике унутар Silvera модела. Могућност описа пословне логике унутар Silvera модела би била од великог значаја јер би са собом донела следеће бенефите: а) број ручно писаних линија кода би био сведен на минимум, б) могућност имплементације додатних дизајн образаца којима би се описале трансакције које обухватају више микросервиса.

- Имплементација генератора кода за различите програмске језике и радне оквире.

Библиографија

[1] W. Lamersdorf, "Paradigms of distributed software systems: Services, processes and self-organization," in *International Conference on E-Business and Telecommunications*, pp. 33–40, Springer, 2011.

[2] M. Pikkarainen, J. Haikara, O. Salo, P. Abrahamsson, and J. Still, "The impact of agile practices on communication in software development," *Empirical Software Engineering*, vol. 13, no. 3, pp. 303–337, 2008.

[3] L. Chen, "Microservices: Architecting for continuous delivery and devops," 03 2018.

[4] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[5] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[6] M. Fowler and J. Lewis, "Microservices." https://www.martinfowler.com/articles/microservices.html. Accessed: 2022-01-10.

[7] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014.

[8] S. Newman, *Building microservices*. "O'Reilly Media, Inc.", 2015.

[9] C. Richardson, "Building microservices: Inter-process communication in a microservices architecture," 2015. Accessed: 2022-01-15.

[10] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *CLOSER (1)*, pp. 137–146, 2016.

[11] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in

Software Architecture (ICSA), 2017 IEEE International Conference on, pp. 21–30, IEEE, 2017.

[12] J. Sorgalla, "Ajil: A graphical modeling language for the development of microservice architectures," in *Extended Abstracts of the Microservices 2017 Conference*, 2017.

[13] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[14] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 5th ed., 2011.

[15] K. P. Birman, *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Berlin, Heidelberg: Springer-Verlag, 2005.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] D. Beuche, A. A. Fröhlich, R. Meyer, H. Papajewski, F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk, "On architecture transparency in operating systems," in *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating system*, pp. 147–152, 2000.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[19] R. Vitillo, *Understanding Distributed Systems: What every developer should know about large distributed applications*. Roberto Vitillo, 2022.

[20] G. S. Blair and J.-B. Stefani, *Open distributed processing and multimedia*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[21] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci, "The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 410–430, Springer, 2011.

[22] B. C. Neuman, "Scale in distributed systems," in *Readings in Distributed Computing Systems*, pp. 463–489, IEEE Computer Society Press, 1994.

[23] P. B. Kruchten, "The 4+ 1 view model of architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995.

[24] C. Richardson, *Microservice patterns*. Manning Publications, 2017.

[25] J. Silcock and A. Gościński, *Message passing, remote procedure calls and distributed shared memory as communication paradigms for distributed systems*. Deakin University, School of Computing and Mathematics, 1995.

[26] J. Wu, *Distributed system design*. CRC press, 2017.

[27] T. Erl, "Service-oriented architecture: Concepts, technology, and design," 2005.

[28] S. Smith, "Architect modern web applications with asp .net core and azure," *Redmond, Washington 98052-6399: Microsoft Corporation, 2019.–113 p*, 2022.

[29] J. Palermo, "The onion architecture," 2009. Accessed: 2023-05-12.

[30] A. Cockburn, "Hexagonal architecture," 2005. Accessed: 2022-10-15.

[31] R. C. Martin, "Clean architecture," 2017.

[32] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pp. 128–141, Springer, 2019.

[33] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *arXiv preprint arXiv:1606.04036*, 2016.

[34] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.

[35] Y. Izrailevsky and A. Tseitlin, "The netflix simian army," 2011. Accessed: 2023-05-09.

[36] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[37] S. Kramer, "The biggest thing amazon got right: The platform," 2016. Accessed: 2023-05-10.

[38] S. Ihde, "From a monolith to microservices + rest: the evolution of linkedin's service architecture," Mar. 2015. Accessed: 2023-05-10.

[39] P. Calçado, "Building products at soundcloud - part i: Dealing with the monolith," June 2014. Accessed: 2023-05-10.

[40] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How to make your application scale," *arXiv preprint arXiv:1702.07149*, 2017.

[41] V. Gucer, S. Narain, *et al.*, *Creating Applications in Bluemix Using the Microservices Approach*. IBM Redbooks, 2015.

[42] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[43] H. Zeiner, M. Goller, V. J. E. Jiménez, F. Salmhofer, and W. Haas, "Secos: Web of things platform based on a microservices architecture and support of time-awareness," *e & i Elektrotechnik und Informationstechnik*, vol. 133, no. 3, pp. 158–162, 2016.

[44] C. Esposito, A. Castiglione, and K.-K. R. Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, 2016.

[45] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pp. 179–182, IEEE, 2016.

[46] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-service for microservices-based cloud applications," in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pp. 50–57, IEEE, 2015.

[47] J. Lin, L. C. Lin, and S. Huang, "Migrating web applications to clouds with microservice architectures," in *Applied System Innovation (ICASI), 2016 International Conference on*, pp. 1–4, IEEE, 2016.

[48] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, "Cloud computing: Distributed internet computing for it and scientific research," *IEEE Internet computing*, vol. 13, no. 5, pp. 10–13, 2009.

[49] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[50] D. Linthicum, "From containers to microservices: Modernizing legacy applications." https://techbeacon.com/containers-microservices-how-modernize-legacy-applications. Accessed: 2023-05-10.

[51] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "Microservices: Migration of a mission critical system," *arXiv preprint arXiv:1704.04173*, 2017.

[52] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *Ieee Software*, vol. 35, no. 3, pp. 50–55, 2018.

[53] O. Zimmermann, "Do microservices pass the same old architecture test? or: Soa is not dead-long live (micro-) services," in *Microservices Workshop at SATURN conference, SEI*, 2015.

[54] K. Indrasiri, "Microservices in pratice-key architectural concepts of an msa," *Wso2 White Paper*, vol. 150, 2016.

[55] M. Richards, "Microservices vs. service-oriented architecture," 2015.

[56] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pp. 318–325, IEEE, 2016.

[57] V. Raj and R. Sadam, "Performance and complexity comparison of service oriented architecture and microservices architecture," *International Journal of Communication Networks and Distributed Systems*, vol. 27, no. 1, pp. 100–117, 2021.

[58] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, 2018.

[59] L. Krause, "Microservices: Patterns and applications," *Lucas Krause*, vol. 1, 2015.

[60] C. Richardson, "Introduction to microservices," 2015. Accessed: 2023-05-10.

[61] T. Limoncelli, J. Robbins, K. Krishnan, and J. Allspaw, "Resilience engineering: learning to embrace failure," *Communications of the ACM*, vol. 55, no. 11, pp. 40–47, 2012.

[62] C. Richardson, "Choosing a microservices deployment strategy," 2016. Accessed: 2023-05-10.

[63] H. Knoche and W. Hasselbring, "Drivers and barriers for microservice adoption-a survey among professionals in germany," *Enterprise Modelling and Information Systems Architectures (EMISAJ)-International Journal of Conceptual Modeling*, vol. 14, no. 1, pp. 1–35, 2019.

[64] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: an exploratory study," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–44, 2021.

[65] S. Baškarada, V. Nguyen, and A. Koronios, "Architecting microservices: practical opportunities and challenges," *Journal of Computer Information Systems*, 2018.

[66] P. Robles, "The emergence of api-first development," Jan. 2014. Accessed: 2023-05-10.

[67] K. Hoffman, "An api-first approach for cloud-native app development," July 2016. Accessed: 2023-05-10.

[68] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: intentions, strategies, and challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 481–490, IEEE, 2019.

[69] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Microservices in industry: insights into technologies, characteristics, and software quality," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 187–195, IEEE, 2019.

[70] M. Kleehaus and F. Matthes, "Challenges in documenting microservice-based it landscape: A survey from an enterprise architecture management perspective," in *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 11–20, IEEE, 2019.

[71] M. Waseem, P. Liang, and M. Shahin, "A systematic mapping study on microservices architecture in devops," *Journal of Systems and Software*, vol. 170, p. 110798, 2020.

[72] V. Bushong, A. S. Abdelfattah, A. A. Maruf, D. Das, A. Lehman, E. Jaroszewski, M. Coffey, T. Cerny, K. Frajtak, P. Tisnovsky, and M. Bures, "On microservice analysis and architecture evolution: A systematic mapping study," *Applied Sciences*, vol. 11, no. 17, 2021.

[73] P. Thiemann, "An embedded domain-specific language for type-safe server-side web scripting," *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 1, pp. 1–46, 2005.

[74] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proceedings of the 11th international conference on World Wide Web*, pp. 396–407, ACM, 2002.

[75] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: systematic resilience testing of microservices," in *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pp. 57–66, IEEE, 2016.

[76] I. Karabey Aksakalli, T. Çelik, A. Can, and B. Tekinerdogan, "Deployment and communication patterns in microservice architectures: A systematic literature review," *Journal of Systems and Software*, vol. 180, p. 111014, 06 2021.

[77] Z. Houmani, D. Balouek-Thomert, E. Caron, and M. Parashar, "Enhancing microservices architectures using data-driven service discovery and qos guarantees," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 290–299, IEEE, 2020.

[78] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[79] E. Visser, "Webdsl: A case study in domain-specific language engineering," in *International summer school on generative and transformational techniques in software engineering*, pp. 291–373, Springer, 2007.

[80] T. Kosar, Z. Lu, M. Mernik, M. Horvat, and M. Črepinšek, "A case study on the design and implementation of a platform for hand rehabilitation," *Applied Sciences*, vol. 11, no. 1, 2021.

[81] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[82] I. Dejanović, M. Dejanović, J. Vidaković, and S. Nikolić, "Pyflies: A domain-specific language for designing experiments in psychology," *Applied Sciences*, vol. 11, no. 17, 2021.

[83] D. Wile, "Lessons learned from real dsl experiments," in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pp. 10–pp, IEEE, 2003.

[84] J. Gray and G. Karsai, "An examination of dsls for concisely representing model traversals and transformations," in *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pp. 10–pp, IEEE, 2003.

[85] A. N. Johanson and W. Hasselbring, "Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment," *Empirical Software Engineering*, vol. 22, no. 4, pp. 2206–2236, 2017.

[86] T. Kosar, S. Gaberc, J. C. Carver, and M. Mernik, "Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2734–2763, 2018.

[87] M. Fowler, *Domain-specific languages*. Addison-Wesley Professional, 2010.

[88] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, 2008.

[89] D. C. Schmidt *et al.*, "Model-driven engineering," *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

[90] S. Kent, "Model driven engineering," in *International conference on integrated formal methods*, pp. 286–298, Springer, 2002.

[91] P. Wizenty, J. Sorgalla, F. Rademacher, and S. Sachweh, "Magma: build management-based generation of microservice infrastructures," in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pp. 61–65, ACM, 2017.

[92] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, and A. Zündorf, "Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations," *SN Computer Science*, vol. 2, no. 6, pp. 1–25, 2021.

[93] F. Rademacher, S. Sachweh, and A. Zündorf, "Aspect-oriented modeling of technology heterogeneity in microservice architecture," in *2019 IEEE International Conference on Software Architecture (ICSA)*, pp. 21–30, IEEE, 2019.

[94] F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with jolie," in *Web Services Foundations*, pp. 81–107, Springer, 2014.

[95] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[96] "Welcome to apache avro!." https://avro.apache.org/. Accessed: 2023-05-10.

[97] "Protocol buffers." https://developers.google.com/protocol-buffers/. Accessed: 2023-05-10.

[98] "grpc." https://grpc.io/. Accessed: 2023-05-10.

[99] T. G. Epperly, G. Kumfert, T. Dahlgren, D. Ebner, J. Leek, A. Prantl, and S. Kohn, "High-performance language interoperability for scientific computing through babel," *The International Journal of High Performance Computing Applications*, vol. 26, no. 3, pp. 260–274, 2012.

[100] "Raml." https://raml.org/. Accessed: 2023-05-10.

[101] "Swagger." https://swagger.io/. Accessed: 2023-05-10.

[102] D. Spinellis, "Notable design patterns for domain-specific languages," *Journal of systems and software*, vol. 56, no. 1, pp. 91–99, 2001.

[103] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[104] T. Kosar, P. E. Martı, P. A. Barrientos, M. Mernik, *et al.*, "A preliminary study on various implementation approaches of domain-specific language," *Information and software technology*, vol. 50, no. 5, pp. 390–405, 2008.

[105] G. Hohpe and B. Woolf, *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[106] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković, "Textx: A python tool for domain-specific languages implementation," *Knowledge-Based Systems*, vol. 115, p. 1–4, 2017.

[107] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service-and microservice-based systems: a literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, pp. 107–115, 2017.

[108] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani, "A metrics suite for evaluating flexibility and complexity in service oriented architectures," in *International Conference on Service-Oriented Computing*, pp. 41–52, Springer, 2008.

[109] D. Rud, A. Schmietendorf, and R. Dumke, "Product metrics for service-oriented infrastructures," *IWSM/MetriKon*, pp. 161–174, 2006.

[110] J. Berg and E. Gralén, "The effects of continuous code inspection on code quality," *Lund University, EDAN80-Coaching of programming teams*, p. 8, 2019.

[111] S. Habchi, X. Blanc, and R. Rouvoy, "On adopting linters to deal with performance concerns in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 6–16, 2018.

[112] X. Li and C. Prasad, "Effectively teaching coding standards in programming," in *Proceedings of the 6th conference on Information technology education*, pp. 239–244, 2005.

[113] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Computer Languages, Systems & Structures*, vol. 52, pp. 43–62, 2018.

[114] J. Vlissides, *Pattern hatching: Design patterns applied*. Addison-Wesley Longman Ltd., 1998.

[115] M. Hofmann, E. Schnabel, K. Stanley, *et al.*, *Microservices Best Practices for Java*. IBM Redbooks, 2017.

[116] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, "A software engineering

experiment in software component generation," in *Proceedings of IEEE 18th International Conference on Software Engineering*, pp. 542–552, IEEE, 1996.

[117] G. Kahraman and S. Bilgen, "A framework for qualitative assessment of domain-specific languages," *Software & Systems Modeling*, vol. 14, no. 4, pp. 1505–1526, 2015.

[118] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, "Experimentation in software engineering: an introduction," 2000.

[119] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to advanced empirical software engineering*, pp. 201–228, Springer, 2008.

[120] V. R. Basili and H. D. Rombach, "The tame project: Towards improvement-oriented software environments," *IEEE Transactions on software engineering*, vol. 14, no. 6, pp. 758–773, 1988.

[121] R. C. Team *et al.*, "R: A language and environment for statistical computing," 2013.

[122] R. Woolson, "Wilcoxon signed-rank test," *Wiley encyclopedia of clinical trials*, pp. 1–3, 2007.

[123] A. Ghosh, A. Mukherjee, and S. Misra, "Sega: Secured edge gateway microservices architecture for iiot-based machine monitoring," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 3, pp. 1949–1956, 2021.

[124] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 11–20, IEEE, 2018.

[125] R. Belafia, P. Jeanjean, O. Barais, G. Le Guernic, and B. Combemale, "From monolithic to microservice architecture: The case of extensible and domain-specific ides," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 454–463, IEEE, 2021.

[126] H. A. El-Ghareeb, "Neutrosophic-based domain-specific languages and rules engine to ensure data sovereignty and consensus achievement in microservices architecture," in *Optimization Theory Based on Neutrosophic and Plithogenic Sets*, pp. 21–43, Elsevier, 2020.

[127] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical study of object-oriented metrics.," *J. Object Technol.*, vol. 5, no. 8, pp. 149–173, 2006.

[128] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 550–562, 2014.

[129] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, "Evaluation of microservice architectures: A metric and tool-based approach," in *International Conference on Advanced Information Systems Engineering*, pp. 74–89, Springer, 2018.

[130] F. Haupt, F. Leymann, A. Scherer, and K. Vukojevic-Haupt, "A framework for the structural analysis of rest apis," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 55–58, IEEE, 2017.

Индекс слика

Индекс листинга

Индекс табела

Биографија

Ален Сулькановић је рођен у 1.4.1991. године у Градачцу, БиХ. Дипломирао је на Факултету техничких наука у Новом Саду, на одсеку Електротехника и Рачунарство, смер Примењене рачунарске науке и информатика са просечном оценом 8,03. Дипломски рад из предмета „Софтверски обрасци и компоненте" на тему „Проширриви алат за визуализацију структуре програмског кода базиран на софтверским компонентама" одбранио је 13.9.2013. године са оценом 10.

На мастер студије Факултета техничких наука, смер Рачунарство и аутоматика, усмерење Рачунарске науке, уписао се 2013/14 године и положио све испите предвиђене планом и програмом са просечном оценом 9.71. Мастер рад на тему „Имплементација *CodeBlock* компоненте коришћењем *Arpeggio* парсера" одбранио је 27.9.2014. године.

Од 2014. године ради као софтверски инжењер у компанији Typhoon HIL, Novi Sad. Области интересовања су му језици специфични за домен, развој софтвера управљан моделима, и дистрибуирани системи. Активан је члан Python заједнице у Србији.

Ожењен је, и живи у Новом Саду. Од страних језика говори енглески језик.

Овај Образац чини саставни део докторске дисертације, односно докторског уметничког пројекта који се брани на Универзитету у Новом Саду. Попуњен Образац укоричити иза текста докторске дисертације, односно докторског уметничког пројекта.

План третмана података

Назив пројекта/истраживања
Језик за опис архитектуре дистрибуираних система базираних на микросервисима
Назив институције/институција у оквиру којих се спроводи истраживање
а) Факултет техничких наука, Универзитет Нови Сад
Назив програма у оквиру ког се реализује истраживање
Рачунарство и аутоматика – докторска дисертација
1. Опис података
1.1 Врста студије *Укратко описати тип студије у оквиру које се подаци прикупљају* **Докторска дисертација.** 1.2 Врсте података **а) квантитативни** **б) квалитативни** 1.3. Начин прикупљања података **а) анкете, упитници, тестови** б) клиничке процене, медицински записи, електронски здравствени записи в) генотипови: навести врсту _____

г) административни подаци: навести врсту _____

д) узорци ткива: навести врсту_____

ђ) снимци, фотографије: навести врсту_____

е) текст, навести врсту _**Актуелна литература у области истраживања**___

ж) мапа, навести врсту _____

з) остало: описати _____

1.3 Формат података, употребљене скале, количина података

1.3.1 Употребљени софтвер и формат датотеке:

а) Excel фајл, датотека _.csv_____

b) SPSS фајл, датотека _____

c) PDF фајл, датотека _____

d) Текст фајл, датотека _____

e) JPG фајл, датотека _____

f) Остало, датотека _.r_____

1.3.2. Број записа (код квантитативних података)

а) број варијабли _13_____

б) број мерења (испитаника, процена, снимака и сл.) ___18_____

1.3.3. Поновљена мерења

а) да

б) не

Уколико је одговор да, одговорити на следећа питања:

а) временски размак између поновљених мера је _____

б) варијабле које се више пута мере односе се на _____

в) нове верзије фајлова који садрже поновљена мерења су именоване као _____

Напомене: _____

Да ли формати и софтвер омогућавају дељење и дугорочну валидност података?

а) Да

б) Не

Ако је одговор не, образложити _____

_____

2. Прикупљање података

2.1 Методологија за прикупљање/генерисање података

2.1.1. У оквиру ког истраживачког нацрта су подаци прикупљени?

а) експеримент, навести тип _**поређење перформанси програмера у случају када се користи језик специфичан за домен и неки од одабраних језика опште намене, као и утврђивање да ли језик специфичан за домен задовољава квалитативне карактеристике дефинисане FQAD радним оквиром**____

б) корелационо истраживање, навести тип _____

ц) анализа текста, навести тип _**Анализа доступне литературе**_____

д) остало, навести шта _____

2.1.2 Навести врсте мерних инструмената или стандарде података специфичних за одређену научну дисциплину (ако постоје).

_____

_____

2.2 Квалитет података и стандарди

2.2.1. Третман недостајућих података

а) Да ли матрица садржи недостајуће податке? Да **Не**

Ако је одговор да, одговорити на следећа питања:

а) Колики је број недостајућих података? _____

б) Да ли се кориснику матрице препоручује замена недостајућих података? Да Не

в) Ако је одговор да, навести сугестије за третман замене недостајућих података

2.2.2. На који начин је контролисан квалитет података? Описати

Квалитет података контролисан је помоћу валидације уграђене у форму за унос података, те скупом аутоматизованих тест скрипти.

2.2.3. На који начин је извршена контрола уноса података у матрицу?

Контрола уноса података у матрицу изведена је на бази експертског знања.

3. Третман података и пратећа документација

3.1. Третман и чување података

*3.1.1. Подаци ће бити депоновани у **Репозиторијуму докторских дисертација на Универзитету у Новом Саду.***

3.1.2. URL адреса _____

3.1.3. DOI _____

3.1.4. Да ли ће подаци бити у отвореном приступу?

а) Да

б) Да, али после ембарга који ће трајати до _____

в) Не

Ако је одговор не, навести разлог _____

3.1.5. Подаци неће бити депоновани у репозиторијум, али ће бити чувани.

Образложење

3.2 Метаподаци и документација података

3.2.1. Који стандард за метаподатке ће бити примењен? **Стандард који примењује Репозиторијум докторских дисертација Универзитета у Новом Саду**

3.2.1. Навести метаподатке на основу којих су подаци депоновани у репозиторијум.

_____

_____

Ако је потребно, навести методе које се користе за преузимање података, аналитичке и процедуралне информације, њихово кодирање, детаљне описе варијабли, записа итд.

_____

_____

_____

_____

3.3 Стратегија и стандарди за чување података

3.3.1. До ког периода ће подаци бити чувани у репозиторијуму? _____

3.3.2. Да ли ће подаци бити депоновани под шифром? **Да** Не

3.3.3. Да ли ће шифра бити доступна одређеном кругу истраживача? **Да** Не

3.3.4. Да ли се подаци морају уклонити из отвореног приступа после извесног времена?

Да **Не**

Образложити

_____

_____

4. Безбедност података и заштита поверљивих информација

Овај одељак МОРА бити попуњен ако ваши подаци укључују личне податке који се односе на учеснике у истраживању. За друга истраживања треба такође размотрити заштиту и сигурност података.

4.1 Формални стандарди за сигурност информација/података

Истраживачи који спроводе испитивања с људима морају да се придржавају Закона о заштити података о личности *(https://www.paragraf.rs/propisi/zakon_o_zastiti_podataka_o_licnosti.html)* и одговарајућег институционалног кодекса о академском интегритету.

4.1.2. Да ли је истраживање одобрено од стране етичке комисије? Да **Не**

Ако је одговор Да, навести датум и назив етичке комисије која је одобрила истраживање

_____

4.1.2. Да ли подаци укључују личне податке учесника у истраживању? Да **Не**

Ако је одговор да, наведите на који начин сте осигурали поверљивост и сигурност информација везаних за испитанике:

а) Подаци нису у отвореном приступу

б) Подаци су анонимизирани

ц) Остало, навести шта

_____

_____

5. Доступност података

5.1. Подаци ће бити

а) јавно доступни

б) доступни само уском кругу истраживача у одређеној научној области

ц) затворени

Ако су подаци доступни само уском кругу истраживача, навести под којим условима могу да их користе:

_____

_____

Ако су подаци доступни само уском кругу истраживача, навести на који начин могу приступити подацима:

5.4. Навести лиценцу под којом ће прикупљени подаци бити архивирани.

Ауторство - некомерцијално

6. Улоге и одговорност

6.1. Навести име и презиме и мејл адресу власника (аутора) података

Ален Суљкановић, biohazard1491@gmail.com

6.2. Навести име и презиме и мејл адресу особе која одржава матрицу с подацима

Ален Суљкановић, biohazard1491@gmail.com

6.3. Навести име и презиме и мејл адресу особе која омогућује приступ подацима другим истраживачима

Ален Суљкановић, biohazard1491@gmail.com