УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Проф. др Игор Дејановић

Шаблон и упутство за писање завршних радова

Нови Сад, 2025

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА ЗАВРШНИ РАД	Датум:

(Податке уноси предметни наставник - ментор)

Студијски програм:	Софтверско инжењерство и информационе технологије		
Студент:	Стефан Пејиновић	Број индекса:	SV13/2021
Степен и врста студија:	Osnovne академске студије		
Област:	Електротехничко и рачунарско инжењерство		
Ментор:	Милош Симић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ ЗАВРШНОГ РАДА:

Шаблон и упутство за писање завршних радова

ТЕКСТ ЗАДАТКА:

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

Руководилац студијског програма:	Ментор рада:

Примерак за: □ - Студента; □ - Ментора

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски - бечелор рад
Аутор, **АУ**:		Стефан Пејиновић
Ментор, **МН**:		Др Милош Симић, доцент
Наслов рада, **НР**:		Шаблон и упутство за писање завршних радова
Језик публикације, **ЈП**:		српски/ћирилица
Језик извода, **ЈИ**:		српски/енглески
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2025
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад, трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		5/34/23/2/9/0/2
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Примењене рачунарске науке и информатика
Предметна одредница/Кључне речи, **ПО**:		Шаблон, завршни рад, упутство
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Овај документ представља упутство за писање завршних радова на Факултету техничких наука Универзитета у Новом Саду. У исто време је и шаблон за Typst.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		01.12.2025
Чланови комисије, **КО**:	Председник:	Др Горан Сладић, редовни професор
	Члан:	Др Милан Стојков, доцент
	Члан:	
	Члан, ментор:	Др Милош Симић, доцент

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	
Author, **AU**:	Stefan Pejinović
Mentor, **MN**:	Miloš Simić, Phd., asist. professor
Title, **TI**:	Template and tutorial for thesis preparation
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian/English
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2025
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	5/34/23/2/9/0/2
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Applied computer science and informatics
Subject/Key words, **S/KW**:	Template, thesis, tutorial
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad
Note, **N**:	
Abstract, **AB**:	This document provides guidelines for writing final theses at the Faculty of Technical Sciences, University of Novi Sad. At the same time, it serves as a Typst template.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	01.12.2025
Defended Board, **DB**: President:	Goran Sladić, Phd., full professor
Member:	Milan Stojkov, Phd., asist. professor
Member:	
Member, Mentor:	Miloš Simić, Phd., asist. professor

Menthor's sign

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

ИЗЈАВА О НЕПОСТОЈАЊУ СУКОБА ИНТЕРЕСА

Изјављујем да нисам у сукобу интереса у односу ментор – кандидат и да нисам члан породице (супружник или ванбрачни партнер, родитељ или усвојитељ, дете или усвојеник), повезано лице (крвни сродник ментора/кандидата у правој линији, односно у побочној линији закључно са другим степеном сродства, као ни физичко лице које се према другим основама и околностима може оправдано сматрати интересно повезаним са ментором или кандидатом), односно да нисам зависан/на од ментора/кандидата, да не постоје околности које би могле да утичу на моју непристрасност, нити да стичем било какве користи или погодности за себе или друго лице било позитивним или негативним исходом, као и да немам приватни интерес који утиче, може да утиче или изгледа као да утиче на однос ментор-кандидат.

У Новом Саду, дана _____

Ментор

_____

Кандидат

_____

Садржај

Глава 1

Увод

Савремени софтверски системи све више усвајају архитектуре засноване на микросервисима, пре свега због скалабилности, лакшег развоја, независног распоређивања компоненти и могућности бржег одговора на промене у пословној логици [1], [2]. У таквим архитектурама укупна функционалност система је распарчана на велики број малих, независних сервиса који међусобно комуницирају преко мрежних протокола. Иако сваки сервис представља самосталну јединицу, њихове међусобне зависности и комуникациони токови значајно повећавају комплексност система у односу на монолитне архитектуре [3]. Под овим условима и краткотрајни откази, успорени одговори или делимичне деградације појединих компоненти могу имати видљив утицај на функционалност читавог система.

Један од кључних изазова у дистрибуираним архитектурама јесте чињеница да сервис може формално бити жив (енгл. *alive*), док у стварности није у стању да обрађује корисничке захтеве било због засићења ресурса, неконзистентног иницијалног стања или других облика делимичних отказа [4], [5]. Ова појава, позната као делимичан отказ (енгл. *partial failure*), представља један од најчешћих узрока нестабилности у дистрибуираним системима [6]. Управо због тога постоји потреба за механизмима који непрекидно прате здравље сервиса (енгл. *health-check*), детектују неправилности у раду система и омогућавају правовремене реакције система у циљу очувања стабилности.

Иако *Kubernetes* поседује уграђене механизме за *health-check* провере, они су доступни искључиво у *Kubernetes* окружењима [7] и тешко их је прилагодити специфичним захтевима система. Чак и у случајевима када се *Kubernetes* користи, ови механизми имају ограничења у погледу прилагодљивости логике провера, проширивости и интеграције са спољним системима [8]. Због тога постоји потреба за независним, конфигурабилним радним окружењем (енгл. *framework*) који омогућава сопствене типове провера здравља и већу контролу над понашањем система, без обзира на окружење у којем се апликације извршавају.

Потребно је направити *framework* који омогућава паралелно праћење стања више сервиса. Оно треба да обезбеди подршку различитих типова провера као што су покретање (енгл. *startup*), спремност (енгл. *readiness*) и конкурентност (енгл. *liveness*), као и различитих комуникационих протокола, попут *HTTP*, *TCP* и *gRPC* [7]. Осим самог извођења провера, неопходно је обезбедити модел стања који на јасан начин описује тренутну исправност сервиса, као и механизме за аутоматизовано реаговање, попут поновног покретања контејнера, евидентирања догађаја и излагања метрика системима за надзор (енгл. *monitoring*). Централизована конфигурација помоћу *YAML* формата додатно олакшава употребу *framework*-а и његову интеграцију у различита окружења.

Циљ овог рада јесте развој самосталног *health-check framework*-а који омогућава унифицирано праћење исправности сервиса у дистрибуираним системима. Развијено решење подржава *startup*, *readiness* и *liveness* логике провера, ради са *HTTP*, *TCP* и *gRPC* протоколима, извршава провере конкурентно, излаже метрике у *Prometheus* формату и омогућава конфигурацију преко *YAML* датотека. Поред тога, *framework* се интегрише са *Docker runtime*-ом ради аутоматизованих реакција на отказе сервиса.

1.1 Структура рада

Рад је организован у више тематских целина које почињу теоријским основама, а завршавају се конкретном имплементацијом развијеног *framework*-а.

Друго поглавље представља теоријске концепте неопходне за разумевање проблема и контекста у коме *framework* функционише. У њему су обрађени темељи контејнеризације, микросервисних архитектура, дистрибуираних система, механизама провере здравља и основа надгледања (енгл. *observability*), уз осврт на изазове који прате ове технологије.

Треће поглавље дефинише функционалне и нефункционалне захтеве система, формулишући шта *framework* мора да обезбеди, која ограничења треба да испуни и које карактеристике се очекују у погледу перформанси, стабилности и проширивости.

У четвртом поглављу дата је архитектура предложеног решења. Приказана је организациона структура компоненти, модели стања, механизми конкурентног извршавања провера и начин на који *framework* обрађује и реагује на различита стања сервиса.

Пето поглавље описује имплементацију система. Представљена је реализација појединачних врста провера (енгл. *probes*), подршка за *HTTP*, *TCP* и *gRPC* протоколе, начин излагања метрика и интеграција са *Docker* окружењем у циљу аутоматске реакције на отказе.

Рад се завршава у шестом поглављу, које садржи закључак и предлог могућих праваца даљег развоја *framework*-а у пракси.

Глава 2

Теоријске основе и постојећа решења

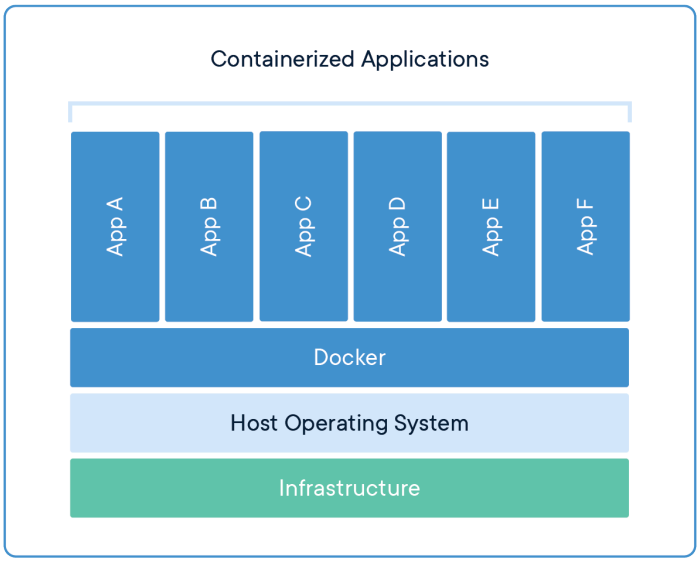
Ово поглавље описује теоријске концепте и технологије које су неопходне за разумевање развијеног решења. На почетку су представљени основни принципи контејнеризације, микросервисних архитектура и дистрибуираних система, као и изазови који се јављају у таквим окружењима. Затим су обрађени механизми провере здравља сервиса и основе надгледања система путем метрика. У посебној целини описане су и технологије коришћене у имплементацији решења, као што су *Docker*, *Docker Compose*, *Prometheus* и *Alertmanager*. Поглавље се завршава прегледом постојећих решења, у ком је приказан *Kubernetes health-check* механизам који је послужио као референтни модел при дизајну система.

2.1 Теоријски концепти релевантни за разумевање решења

2.1.1 Контејнеризација

Контејнеризација представља приступ извршавању апликација у изолованим и преносивим окружењима која се називају контејнери. Они обједињују апликацију и све њене зависности и библиотеке у једну стандардизовану јединицу софтвера, обезбеђујући да се апликација понаша предвидљиво независно од окружења у ком се извршава [9]. Овај приступ решава проблем различитог понашања софтвера на развојним, тест и продукционим окружењима, чинећи контејнере кључном компонентом савремених дистрибуираних и архитектура у облаку (енгл. *cloud*) [10].

За разлику од виртуелних машина, контејнери деле језгро оперативног система домаћина, што резултује минималним оптерећењем ресурса и изузетно брзим покретањем [11]. Ова ефикасност омогућава истовремено извршавање већег броја контејнера на једној машини без значајног утицаја на перформансе. Уз то, контејнери обезбеђују снажну изолацију процеса, мрежних ресурса и система датотека [9], чинећи их погодном основом за системе који захтевају стабилно и скалабилно извршавање већег броја независних микросервиса. На слици 1 приказана је поједностављена архитектура контејнеризованог окружења.



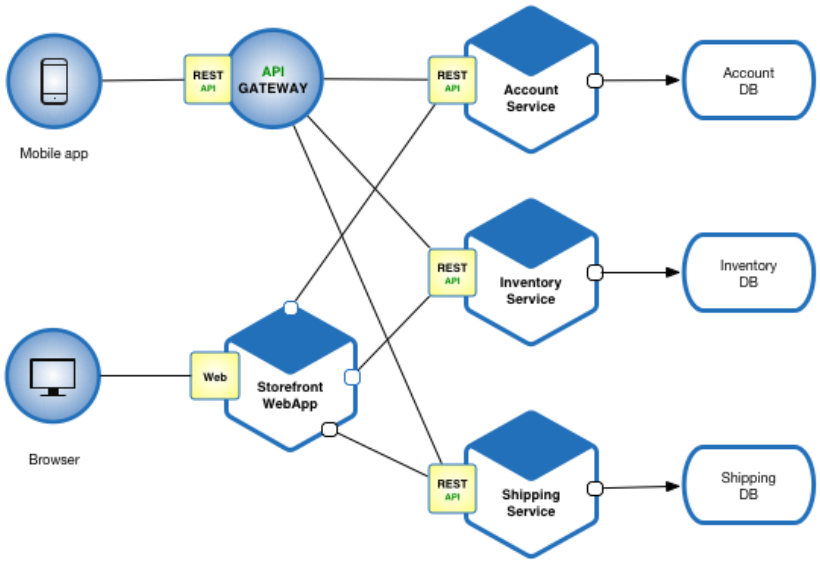
Слика 1: Архитектура контејнеризованог окружења [9]

У микросервисним архитектурама, контејнери омогућавају да сваки сервис буде развијан, тестиран и распоређен независно од осталих делова система [1]. Ова изолација олакшава управљање животним циклусом апликација и омогућава тимовима да користе различите технологије и верзије библиотека без

конфликата. Међутим, динамична природа контејнеризованих окружења уводи нове изазове у надгледању здравља система, што захтева механизме који могу континуирано да прате доступност и исправност сервиса у реалном времену [8]. *Health-check* системи постају кључни за детекцију проблема попут исцрпљених ресурса, мрежних прекида или неодговарајућих сервиса, обезбеђујући аутоматско опорављање и одржавање високе доступности апликација [7].

2.1.2 Микросервисна архитектура

Микросервисна архитектура представља приступ развоју софтвера у коме је сложена апликација подељена на више малих, независних сервиса, од којих је сваки задужен за јасно дефинисану пословну функционалност [1]. Сваки сервис се развија, тестира и распоређује самостално, најчешће користи сопствену базу података и може бити имплементиран у различитом програмском језику [2]. На слици 2 дат је поједностављен приказ микросервисне архитектуре. Иако ова декомпозиција омогућава већу флексибилност и одрживост, она истовремено уводи комплексне ланце позива: један захтев може пролазити кроз више сервиса, а здравље сваке компоненте директно утиче на доступност и перформансе целокупног система [6].



Слика 2: Основни концепт микросервисне архитектуре [12]

Микросервисне платформе функционишу у динамичним окружењима у којима се инстанце сервиса континуирано покрећу, заустављају, скалирају и премештају између чворова кластера [3]. Ова динамика, у комбинацији са међусобном завишношћу сервиса, повећава ризик од каскадних отказа, где проблем у једном сервису може довести до деградације рада читавог система [6].

2.1.3 Дистрибуирани системи

Дистрибуирани систем представља скуп независних компоненти које комуницирају преко мреже како би заједно извршавале одређене задатке [4]. Микросервисна архитектура природно гради дистрибуирани систем, јер сваки сервис функционише као засебна компонента која сарађује са другима путем мрежних протокола. За разлику од монолитних апликација где компоненте комуницирају директним позивима функција у оквиру истог процеса, дистрибуирани системи се суочавају са додатним изазовима који проистилазе из мрежне комуникације [5].

Мрежна комуникација није поуздана, што доводи до непредвидивог понашања. Посебно су проблематични парцијални откази, где један део система нормално функционише док је други недоступан или ради са лошим перформансама [6]. Овакви откази су теже уочљиви од потпуних, јер није увек јасно да ли сервис не одговара због сопственог проблема, спорог извршавања или застоја на мрежи.

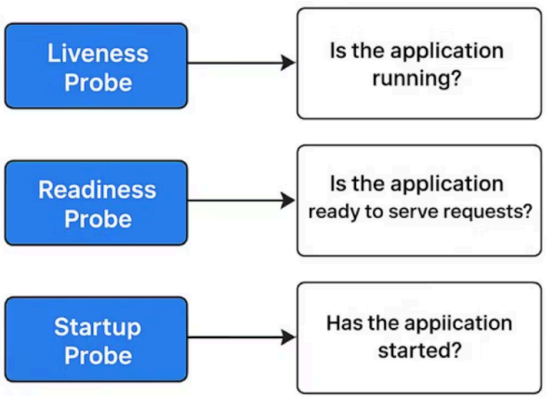
Надгледање дистрибуираних система је знатно сложеније од надгледања монолитних апликација [8]. Информације о стању система су разложене на десетине независних сервиса, од којих сваки генерише сопствене логове и метрике. Због тога су механизми за праћење здравља система критични за очување стабилности, доступности и благовремено реаговање на отказе сервиса [7].

2.1.4 Системи за проверу здравља

Health-check представља механизам за аутоматско праћење исправности сервиса у дистрибуираним системима. Његова основна сврха је да утврди да ли је сервис у стању да обрађује захтеве и да омогући реаговање у случају неправилног рада [7]. У микросервисним архитектурама, где сваки сервис функционише као независна компонента, континуирана провера здравља је неопходна за очување доступности и спречавање ширења отказа кроз остатак система [13].

Разлози за увођење *health-check* механизама проистичу из природе дистрибуираних система. Сервиси могу постати недоступни услед грешака у извршавању, узајамне блокаде (енгл. *deadlock*), преоптерећења или проблема на мрежи. У таквим ситуацијама процес апликације може и даље постојати, али без могућности да правилно обрађује захтеве. Без аутоматизованих провера, други делови система наставили би да му шаљу саобраћај, што може довести до значајних кашњења или каскадних отказа [13].

Да би се ови проблеми открили на време, уводе се различити типови провера здравља који покривају различите фазе животног циклуса сервиса. *Kubernetes* дефинише три основне категорије провера: *liveness*, *readiness* и *startup*, од којих свака решава специфичан аспект понашања апликације [7]. На слици 3 дат је поједностављен приказ основних типова *health-check* провера.



Слика 3: Основни типови *health-check* провера [14]

Liveness probe служи за утврђивање да ли је апликација у исправном стању. Она детектује ситуације у којима процес постоји, али више није функционалан, као што су *deadlock* или бесконачне петље. Када ова провера више пута узастопно не успе, систем може аутоматски поново покренути контејнер, што побољшава доступност и омогућава брз опоравак од отказа [13].

Readiness probe одређује да ли је сервис спреман да прима саобраћај. Она не служи за детекцију кварова, већ за регулисање оптерећења у динамичним условима. Ако провера не успе, сервис се привремено искључује из скупа доступних инстанци, али се сам контејнер не покреће поново [7]. Ово је важно током иницијализације или у условима привременог оптерећења.

Startup probe намењена је апликацијама које имају дужи процес покретања. Она осигурава да се *liveness* и *readiness* провере не активирају прерано и не изазову поновно покретање пре него што се апликација у потпуности иницијализује. Ово је посебно значајно за *legacy* системе или апликације са сложеном фазом иницијализације [7].

Провере здравља изводе се на различите начине, у зависности од природе апликације и доступних интерфејса [13]:

- *HTTP* провера: систем шаље *HTTP GET* захтев на дефинисану путању (нпр. /*health*), при чему се статуси 200–399 сматрају успешним.
- *TCP* провера: покушава се успостављање *TCP* конекције на одређени порт сервиса.
- *gRPC* провера: користи се *gRPC health protocol* за верификацију исправности сервиса.

Ове технике омогућавају униформан и поуздан начин праћења здравља различитих сервиса у дистрибуираним системима.

2.1.5 Метрике и основе надгледања

Observability представља способност система да пружи јасан увид у своје унутрашње стање на основу података које сам генерише током рада. Метрике су квантитативни показатељи који се прикупљају периодично или након одређених догађаја и описују понашање система кроз време. Примери могу бити број успешно обрађених захтева, латенција одговора или број неуспелих провера здравља.

Метрике играју кључну улогу како за администраторе, тако и за аутоматизоване механизме попут распоређивача (енгл. *scheduler*), компоненти за аутоматско скалирање (енгл. *autoscaler*) и самообављајућих (енгл. *self-healing*) компоненти, које се ослањају на метрике при доношењу одлука у реалном времену [15]. Системи за надгледање користе метрике да би детектовали неправилности попут наглог пораста грешака, пада доступности или неуобичајених образаца коришћења ресурса [16]. Алармирање засновано на метрикама омогућава благовремено обавештавање администратора или активирање аутоматских механизама за опоравак, чиме се значајно скраћује време реаговања на проблеме.

Health-check системи који излажу метрике пружају додатну вредност тиме што омогућавају прецизније праћење стања сервиса током времена. Метрике као што су укупан број провера здравља, број неуспелих провера и трајање појединачних провера омогућавају идентификацију трендова и потенцијалних аномалија. Агрегација ових података на нивоу чворова, сервиса или читавих кластера пружа свеобухватан увид у здравље система [15]. Излагање метрика у стандардизованим форматима олакшава интеграцију са постојећим алатима за надгледање.

2.2 Технологије коришћене у решењу

Framework је развијен у програмском језику *Go* [17], који пружа подршку за конкурентно извршавање (енгл. *goroutines*) и богату стандардну библиотеку за *HTTP*, *TCP* и *gRPC* комуникацију. Конфигурација система дефинисана је у *YAML* формату, чиме смо омогућили једноставан опис сервиса и свих параметара неопходних за провере и откривање проблема.

За интеграцију са *Docker* окружењем коришћена је *Docker Engine API* [18], која омогућава управљање животним циклусом контејнеру укључујући поновно покретање неиспавних инстанци. Метрике о стању сервиса излажу се у *OpenMetrics* формату [19] компатибилном са *Prometheus* екосистемом, док *Alertmanager* [20] прихвата те метрике и шаље обавештења на основу дефинисаних правила. *Docker Compose* [21] је коришћен за управљањем свим компонентама током локалног развоја и тестирања.

2.2.1 *Docker и Docker Compose*

Docker је платформа за контејнеризацију која омогућава паковање апликација заједно са свим њиховим зависностима у изоловане контејнере [9]. *Docker Engine API* [18] обезбеђује програмски приступ за управљање животним циклусом контејнера, укључујући креирање, покретање, заустављање и брисање. У контексту овог рада, *Docker API* се користи за аутоматизовано реаговање на неправилности откривене током *health-check* провера, омогућавајући *framework*-у да поново покрене неисправне контејнере без мануелне интервенције.

Docker Compose је алат за дефинисање и покретање вишеконтејнерских апликација [21]. Помоћу *YAML* конфигурационе датотеке (*docker-compose.yml*) могуће је описати све сервисе, мреже и дељене директоријуме (енгл. *volumes*) који чине апликацију, а затим једном командом (*docker-compose up*) покренути целокупно окружење. Ово поједностављује управљање локалним развојним окружењем и тестирање система који се састоји од више компоненти. У овом раду *Docker Compose* се користи за покретање и управљање *health-check framework*-ом заједно са пратећим сервисима (*Prometheus*, *Alertmanager*) и тестним апликацијама, омогућавајући брзо подизање и рушење комплетног тестног окружења.

2.2.2 *Prometheus*

Prometheus је систем за прикупљање и складиштење метрика, дизајниран специјално за праћење дистрибуираних система и *cloud-native* апликација [22]. Заснива се на *pull* моделу где *Prometheus* сервер периодично узима (енгл. *scrape*) метрике са дефинисаних endpoint-а који излажу податке у *OpenMetrics* формату [19]. Метрике се чувају као временске серије и могу се испитивати помоћу *PromQL* упитног језика, што омогућава флексибилну анализу и визуелизацију стања система.

У оквиру овог рада, *Prometheus* прикупља метрике које *health-check framework* излаже попут број извршених провера, број неуспешних провера и број поновних покретања контејнера, пружајући увид у здравље праћених сервиса током времена.

2.2.3 *Alertmanager*

Alertmanager је компонента *Prometheus* екосистема задужена за обраду и управљање упозорењима (енгл. *alerts*) [20]. Прима аларме које генерише *Prometheus* на основу дефинисаних правила (енгл. *alerting rules*) и врши њихово груписање, филтрирање и прослеђивање одговарајућим каналима обавештавања. Подржани су бројни начини нотификација, укључујући *email*, *Slack* и још многе друге интеграције.

У контексту овог рада, *Alertmanager* прима аларме засноване на метрикама о поновним покретањима контејнера и аутоматски шаље *email* обавештења администраторима, што омогућава брзу реакцију на уочене проблеме.

2.3 Постојећа решења

Kubernetes је најкоришћенија платформа за управљање контејнеризованим апликацијама и садржи уграђен систем за аутоматско праћење здравља сервиса [7]. Провере здравља извршава *kubelet*, агент који ради на сваком чвору кластера и брине о томе да *pod*-ови (основна јединица извршавања у *Kubernetes*-у) раде у исправном стању.

Kubernetes подржава више типова провера здравља, а то су *liveness*, *readiness* и *startup probe* које се дефинишу у конфигурацији *pod*-а и аутоматски извршавају током рада апликације. Сваки *probe* може користити различите механизме провере, укључујући *HTTP* захтеве, *TCP* конекцију, као и *gRPC health protocol*, што омогућава проверу различитих типова сервиса.

Систем на основу резултата ових провера одлучује када треба уклонити pod из саобраћаја, када да га поново укључи или када је потребно извршити поновно покретање контејнера. На овај начин *Kubernetes* обезбеђује стабилан рад и високу доступност апликација без мануелне интервенције.

Због јасно дефинисане логике и широке примене у индустрији, *Kubernetes health-check* механизам је у овом раду коришћен као референтни модел приликом дизајна независног *framework*-а за проверу здравља сервиса.

Глава 3

Функционални и нефункционални захтеви

Основна сврха система представља надзор и праћење дефинисаних сервиса, као и адекватно реаговање на њихове отказе. Сервиси, као и параметри неопходни за њихов правилан надзор дефинишу се у конфигурационим фајловима. Решење мора бити лако прошириво, поуздано и инфраструктурно независно.

3.1 Функционални захтеви

У наставку су дефинисани функционални захтеви које систем мора да обезбеди у погледу понашања, начина рада и интеракције са надгледаним сервисима. Главни захтеви обухватају:

- дефинисање и учитавање конфигурације,
- извршавање провера здравља система,
- управљање животним циклусом контејнера,
- евидентирање резултата провера,
- интеграција са екстерним системима.

3.1.1 Дефинисање и учитавање конфигурације

Систем мора да омогући учитавање конфигурационог фајла у *YAML* формату. Конфигурација треба да садржи листу контејнера, типове провера и параметре неопходне за њихово извршавање (интервали, временска ограничења, број покушаја и сл.). Потребно је омогућити проверу исправности конфигурације, при чему неисправан или непотпун унос мора резултирати обуставом рада система.

3.1.2 Извршавање провера здравља

Систем мора да периодично извршава провере *liveness*, *readiness* и *startup* над дефинисаним сервисима. Свака провера мора да примени параметре дефинисане у конфигурацији, као што су интервал, број поновљених покушаја, временско ограничење и тип провере. Резултати провера морају бити евидентирани и доступни осталим компонентама система.

3.1.3 Управљање животним циклусом контејнера

Систем мора да буде у стању да, услед прекорачења дефинисаног прага неуспеха, иницира поновно покретање контејнера. Након поновног покретања систем мора започети нови циклус провера, укључујући startup провере пре него што контејнер буде сматран функционалним.

3.1.4 Евендирање резултата провере

Систем мора да омогући вођење локалне евиденције резултата провера у формату погодном за накнадну анализу. Поред локалне евиденције, систем мора да излаже метрике у *OpenMetrics* формату, укључујући број успешних и неуспешних провера, број поновних покретања контејнера, као и друга релевантна стања која описују здравље система.

3.1.5 Интеграција са екстерним системима

Систем мора да омогући интеграцију са алатима који подржавају *OpenMetrics* формат, као што су *Prometheus* и *Alertmanager*. Правилним излагањем метрика у стандарду погодном за анализу, екстерни алати могу да извршавају аутоматско алармирање и обавештавање крајњих корисника. На систему је да обезбеди прецизне и стабилне метрике на основу којих се могу дефинисати правила за даљу обраду.

3.2 Нефункционални захтеви

У наставку су дефинисани нефункционални захтеви које систем треба да обезбеди.

3.2.1 Перформансе и ефикасност

Систем мора да извршава провере у предвиђеним интервалима без значајног кашњења. Употреба меморије и процесора мора бити минимална и не сме утицати на перформансе надгледаних сервиса. Излагање метрика мора бити брзо и ефикасно.

3.2.2 Скалабилност

Систем мора подржати произвољан број контејнера без значајног пада перформанси. Додавање нових сервиса мора бити могуће искључиво изменом конфигурације, без измене кода. Систем мора лако да подржи нове типове провера.

3.2.3 Одрживост и проширивост

Систем мора бити модуларно организован тако да различите компоненте (конфигурација, провере, метрике, управљање контејнерима) буду лако одрживе. Додавање нових формата метрика или различитих механизама реаговања мора бити могуће уз минималне измене у постојећем коду. Структура кода мора омогућити једноставану интеграцију са постојећим алатима за надгледање.

3.2.4 Портабилност

Систем мора бити извршив у сваком *Docker*-компатибилном окружењу, независно од оперативног система и платформе. Конфигурација система мора бити лако прилагодљива различитим окружењима, без додатних измена у коду. Систем мора бити развијен тако да минималне промене у окружењу не доведу до промене у начину извршавања.

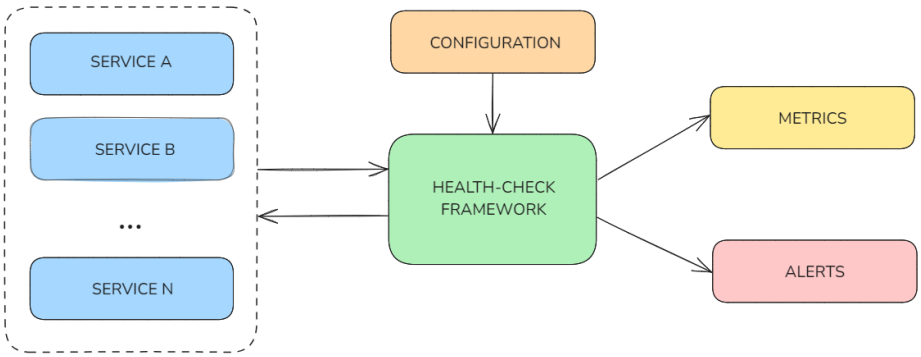
Глава 4

Архитектура система

У овом поглављу описана је архитектура развијеног система кроз две целине. Најпре је представљен концептуални модел решења, који приказује логичку организацију система, његове главне компоненте и односе између њих. Након тога дат је преглед кључних функционалности *framework*-а, које описују токове надзора сервиса, ажурирање њиховог стања, руковање неуспесима, евидентирање метрика и генерисање алармних сигнала.

4.1 Концептуални модел система

Архитектура система организована је тако да јасно раздваја кључне функционалности и омогућава једноставно проширивање и одржавање решења. У основи, систем се састоји од конфигурационог слоја, централног *framework*-а за надзор, сервиса који се проверавају и компоненти за метрике и аларме. Ове целине међусобно сарађују кроз једноставне и добро дефинисане токове података. На слици 4 приказан је концептуални модел система.



Слика 4: Концептуални модел система.

4.1.1 Конфигурациони слој

Конфигурациони слој представља улазну тачку система у којој се дефинишу сви релевантни параметри надзора. У њему се наводе сервиси који се прате, типови провера које треба извршавати, интервали и прагови на основу којих се доносе одлуке о исправности. Овај слој омогућава да се понашање система прилагоди различитим окружењима без потребе за изменама у самом *framework*-у, чиме се постиже флексибилност и лако одржавање.

4.1.2 Health-check framework

Health-check framework представља језгро система и задужен је за извршавање свих операција надзора. На основу учитане конфигурације, овај модул иницира периодичне провере, обрађује резултате добијене од сервиса и одређује у ком се стању рада они налазе. У оквиру овог модула доносе се и одлуке о покретању корективних акција, попут поновног покретања сервиса. *Framework* функционише као централни чвор који координише интеракцију између свих осталих компоненти система.

4.1.3 Сервиси који се проверавају

Сервиси који се проверавају представљају апликације, микросервисе или друге функционалне јединице чије је здравље предмет праћења. *Framework* се периодично повезује са сваким сервисом како би утврдио да ли он функционише у очекиваном режиму. На основу добијене повратне информације, *framework* процењује доступност, исправност и стање сваке јединице. У концептуалном моделу сервиси су логички груписани у оквиру извршног окружења, али остају независни од конкретне технологије или платформе на којој се извршавају.

4.1.4 Компоненте за метрике и аларме

Компонента за метрике служи за бележење информација о успешно или неуспешно извршеним проверама, као и о променама у стању сервиса током времена. Ове информације представљају основу за каснију анализу понашања система и могу се користити за визуелизацију или праћење историје стабилности. Поред тога, *framework* генерише и алармне сигнале који указују на критичне проблеме, као што су већи број узастопних неуспеха или потпуна недоступност сервиса. Метрике и аларми представљају излазни слој система и не зависе од конкретног механизма за њихову даљу обраду.

4.2 Функционалности система

У наставку су издвојене кључне функционалности које *framework* реализује. Оне представљају основне токове извршавања у оквиру система и описују на који начин *framework* учитава конфигурацију, покреће периодичне провере, обрађује резултате, доноси одлуке о стању сервиса и иницира корективне мере када је то неопходно. У табели 1 дат је преглед свих функционалности на високом нивоу и њихови описи.

Табела 1: Преглед функционалности *Health-check framework*-а

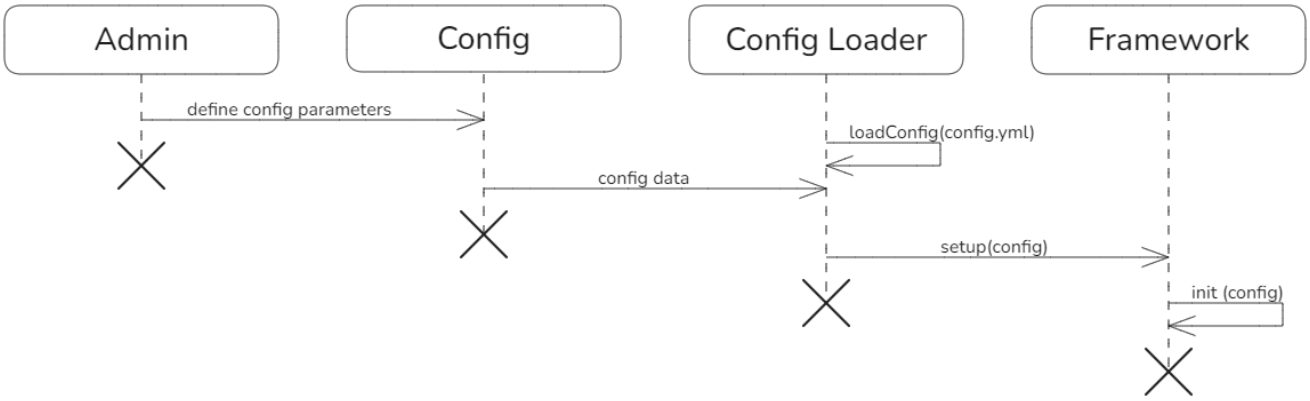
Функционалност	Опис
Иницијализација система	*Framework* чита *YAML* фајл, поставља почетна стања и припрема компоненте за рад.
Извршавање *health-check* циклуса	*Framework* периодично упућује захтеве дефинисаним сервисима како би проверио њихову доступност.
Обрада резултата провере	Одговори добијени од сервиса анализирају се како би се утврдила успешност провера.
Управљање животним циклусом сервиса	На основу резултата *framework* ажурира тренутно стање посматраног сервиса.
Руковање неуспесима и покретање корективних акција	Када се детектује више узастопних неуспеха, *framework* активира корективне мере.
Генерисање и бележење метрика	Током рада система бележе се метрике о успесима, неуспесима и променама стања сервиса, које се користе за анализу система.
Генерисање аларма	При критичним стањима или продуженој недоступности сервиса *framework* генерише аларме.

4.2.1 Иницијализација система

Framework се покреће на основу конфигурационе датотеке у *YAML* формату коју је дефинисао корисник или администратор, а чија је логичка структура приказана у табели 2. Након покретања, *framework* приступа конфигурацији и преузима податке о сервисима који се надзиру, врстама провера које треба да се извршавају и параметрима који одређују понашање система. *Config Loader* учитава садржај конфигурације, валидира структуру и припрема све неопходне информације за даљу употребу. *Health-check framework* затим обрађује преузете податке, иницијализује своје унутрашње компоненте, поставља почетна стања сервиса и припрема систем за извршавање периодичних *health-check* циклуса. На слици 5 приказан је дијаграм секвенце који илуструје описани ток учитавања конфигурације и иницијализације система.

Табела 2: Структура конфигурационе *YAML* датотеке

Параметар	Опис
containers	Листа сервиса који се надзиру. Сваки елемент садржи назив сервиса и параметре провера.
name	Јединствени идентификатор сервиса у оквиру система надзора.
startupProbe / *livenessProbe* / *readinessProbe*	Дефиниције различитих типова провера које се извршавају над сервисом.
httpGet / *tcpSocket* / *grpcCheck*	Параметри специфични за изабрани метод провере: путања и порт за *HTTP*, порт за *TCP*, назив *gRPC* сервиса за *gRPC*.
path	Рута контролера или endpoint-а на коју се упућује *HTTP* захтев у циљу провере стања сервиса.
port	Порт на којем сервис очекује проверу (важи за све методе).
host	Име сервиса или контејнера у оквиру извршног окружења преко кога се успоставља комуникација.
initialDelaySeconds	Време које *framework* чека пре извршавања прве провере одређеног probe-а.
periodSeconds	Интервал између узастопних провера током рада система.
failureThreshold	Број узастопних неуспеха након којих се сервис сматра неисправним.



Слика 5: Дијаграм секвенце који описује ток иницијализације система.

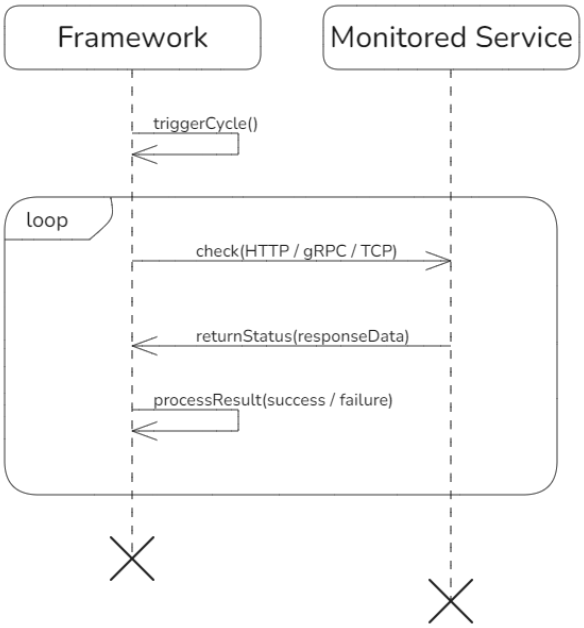
4.2.2 Извршавање *health-check* циклуса

Framework периодично покреће *health-check* циклус у складу са интервалима дефинисаним у конфигурационој датотеци. Током сваког циклуса *framework* приступа сервисима који су наведени за надзор и одређује врсту провере (*startup*, *liveness* или *readiness*) као и метод комуникације који је предвиђен за тај сервис (*HTTP*, *TCP* или *gRPC*). Након припреме параметара провере *framework* иницира комуникацију упућивањем *health-check* захтева ка сервису. Сервис затим враћа информацију о свом тренутном стању, као што су доступност или грешка у раду.

4.2.3 Обрада резултата провере

По пријему одговора сервиса, *framework* приступа интерпретацији добијених података како би одредио исход текуће провере. У овој фази изводи се анализа која класификује резултат као успех или неуспех, без обзира на то који је метод комуникације коришћен. Обрађени резултат се затим бележи у унутрашње структуре *framework*-а и служи као улаз за наредне кораке као што су ажурирање животног циклуса сервиса, генерисање метрика и по потреби, активирање механизама за корективне акције или алармирање.

Извршавање *health-check* циклуса и обрада резултата представљају две уско повезане фазе које чине једну логичку целину надзора сервиса. Прво се иницира и извршава провера стања сервиса, а затим се добијени резултат тумачи и евидентира као основ за даље одлуке у систему. На слици 6 приказан је ток који обухвата покретање *health-check* циклуса, комуникацију са сервисом, као и обраду и бележење резултата провере.



Слика 6: Дијаграм секвенце који описује ток комуникације са сервисом који се надзире.

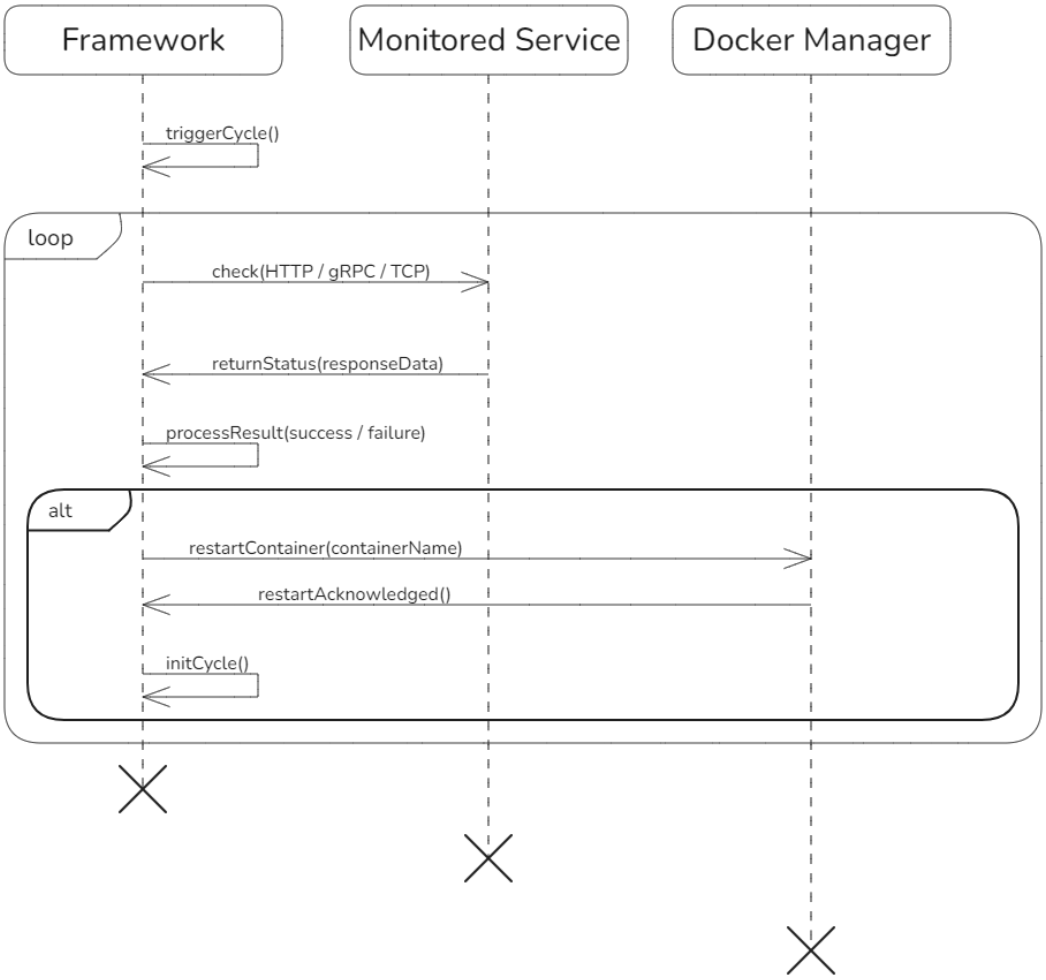
4.2.4 Управљање животним циклусом сервиса

Након обраде резултата појединачних провера, *framework* ажурира стање сервиса на основу више узастопних успеха или неуспеха. Оцену не одређује једна провера, већ се посматра понашање сервиса током више узастопних *health-check* циклуса, како би се разликовале краткотрајне сметње од стварних проблема у раду. На овај начин сервис може бити означен као здрав (енгл. *Healthy*) или нездрав (енгл. *Unhealthy*) у контексту *liveness* провера, односно као спреман (енгл. *Ready*) или неспреман (енгл. *NotReady*) у случају *readiness* провера. Прелазак из једног у друго стање заснива се на праговима дефинисаним у конфигурационој датотеци (броју узастопних неуспеха након којих се сервис сматра неисправним).

4.2.5 Руковање неуспесима и покретање корективних акција

Након што *framework* ажурира стање сервиса на основу резултата провера, он паралелно прати и број узастопних неуспеха. Уколико тај број пређе праг дефинисан у конфигурационој датотеци, сервис се означава као *Unhealthy*, а *framework* активира коративну меру са циљем опоравка сервиса. У овом систему коративна мера подразумева поновно покретања сервиса у оквиру *Docker* окружења, чиме се омогућава да систем аутоматски реагује на детектовани проблем.

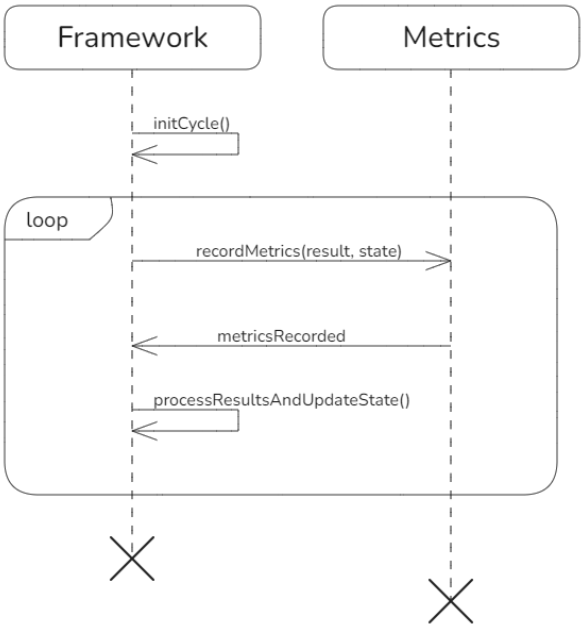
Поступак поновног покретања се спроводи тако што *framework* упућује команду *Docker*-у да поново покрене одговарајући контејнер. Након рестарта сервис улази у нови *health-check* ток, где се поново процењује његово стање на основу резултата наредних провера. На слици 7 приказан је дијаграм секвенце који илуструје ток доношења одлука о поновном покретању и наставак health-check циклуса након коративне акције.



Слика 7: Дијаграм секвенце који описује *health-check* циклус са коративним акцијама.

4.2.6 Генерисање и бележење метрика

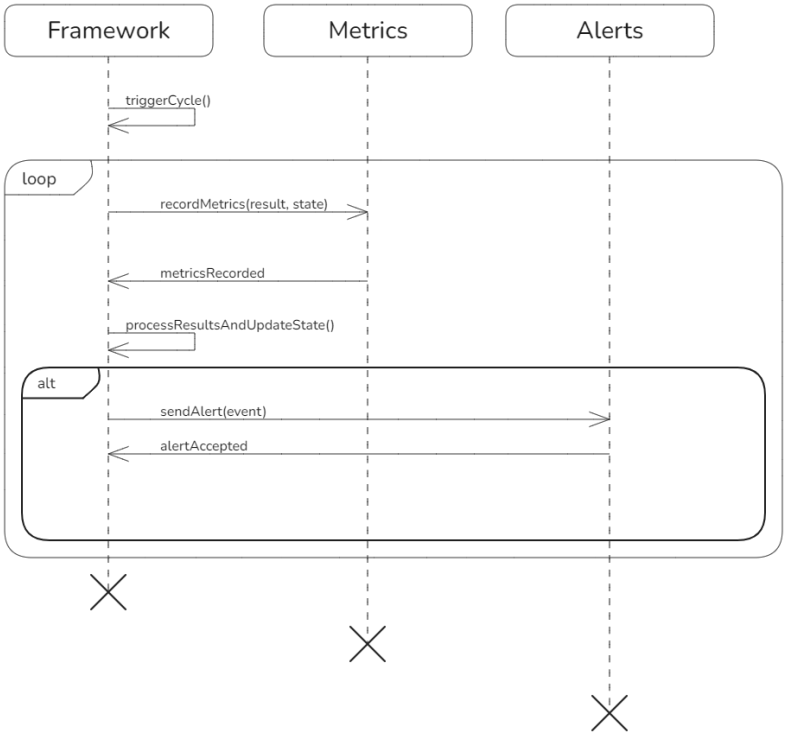
Током рада система *framework* континуирано генерише метрике које описују понашање надзираних сервиса и самог механизма надзора. На основу резултата појединачних провера и ажурираних стања сервиса бележе се, између осталог, информације о броју извршених провера, броју успешних и неуспешних провера по сервису, као и о променама стања (нпр. прелазак из *Healthy* у *Unhealthy* или обрнуто). На овај начин се гради историја која омогућава да се уоче обрасци понашања и дугорочни трендови у стабилности система. Овако прикупљене метрике излажу се кроз јединствен интерфејс који може да користи спољашњи систем за надгледање. На слици 8 приказан је дијаграм секвенце који описује циклус генерисања и бележења метрика.



Слика 8: Дијаграм секвенце који описује циклус генерисања и бележења метрика.

4.2.7 Генерисање аларма

Поред бележења метрика, *framework* је задужен и за препознавање критичних стања сервиса и генерисање алармних сигнала када је то неопходно. На основу интерног стања сервиса, броја узастопних неуспеха, покушаја поновног покретања или дужине периода недоступности, *framework* процењује да ли је потребно да се о конкретном проблему обавесте корисници или спољашњи систем за алармирање. Када су услови за аларм испуњени, формира се опис догађаја који садржи релевантне информације (нпр. који сервис је погођен, који је тип проблема и када је настао) и тај опис се прослеђује компоненти задуженој за даље дистрибуирање аларма. На слици 9 приказан је дијаграм секвенце који описује циклус генерисања аларма.



Слика 9: Дијаграм секвенце који описује циклус генерисања аларма.

Глава 5

Закључак

У закључку дајте кратак преглед онога шта урађено, са освртом на проблеме који су решени, предности и мане решења и правце даљег развоја.

Списак слика

Списак листинга

Списак табела

Списак коришћених скраћеница

Скраћеница	Опис
API	Application Programming Interface (апликациони програмски интерфејс)
AWS	Amazon Web Services (Амазон веб сервиси)
CI/CD	Continuous Integration / Continuous Delivery (континуирана интеграција / континуирана испорука)
CORS	Cross-Origin Resource Sharing (размена ресурса између извора и дестинације различитог порекла)
CSS	Cascading Style Sheets (језик за описивање стилова)
DOM	Document Object Model (објектни модел документа)
DTO	Data Transfer Object (објекат за пренос података)
HTTP	HyperText Transfer Protocol (протокол за пренос хипертекста)
JSON	JavaScript Object Notation (формат за размену података)
JWT	JSON Web Token (сигурносни токен заснован на JSON формату)
RLS	Row-Level Security (сигурност на нивоу реда)
REST	Representational State Transfer (скуп правила за комуникацију између клијента и сервера)
RPC	Remote Procedure Call (позив удаљене процедуре)
SQL	Structured Query Language (структурирани упитни језик)
TLS	Transport Layer Security (безбедност транспортног слоја)
UML	Unified Modeling Language (језик за моделовање дијаграма)
URL	Uniform Resource Locator (јединствени идентификатор и локатор ресурса)
UI	User Interface (кориснички интерфејс)
UUID	Universally Unique Identifier (универзално јединствени идентификатор)
WAL	Write-Ahead Logging (записивање операција унапред)

Списак коришћених појмова

Појам	Објашњење
Асихнрони рад	Рад који се изводи независно од главног тока извршавања, омогућавајући наставак других операција без чекања на његов завршетак.
Bucket (S3)	Логичка јединица за складиштење у AWS S3 сервису, која организује фајлове у облаку.
Read-Only	Режим рада у коме су подаци само за читање, без могућности измене.
Cross-platform	Способност софтвера да се извршава на више различитих оперативних система из истог кода.
Connection pool	Механизам за управљање и поновну употребу веза са базом података како би се побољшале перформансе апликације.

Биографија

Стефан Пејиновић рођен је 12. јула 2002. године у Сомбору. Основно образовање стекао је у Основној школи „Иво Лола Рибар“ у Сомбору. Средњу школу завршио је у Средњој техничкој школи у Сомбору, где је за изузетан успех добио Вукову диплому и награду за ђака генерације.

Факултет техничких наука у Новом Саду, смер Софтверско инжењерство и информационе технологије, уписао је 2021. године. Током студија успешно је положио све испите предвиђене планом и програмом.

Литература

[1] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, 2018. [На Интернету]. Доступно на https://www.manning.com/books/microservices-patterns

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015. [На Интернету]. Доступно на https://www.oreilly.com/library/view/building-microservices/9781491950340/

[3] J. Lewis и M. Fowler, „Microservices: a definition of this new architectural term“. Приступљено: 03. Децембар 2025. [На Интернету]. Доступно на https://martinfowler.com/articles/microservices.html

[4] A. S. Tanenbaum и M. V. Steen, *Distributed Systems: Principles and Paradigms*, 3rd изд. CreateSpace Independent Publishing Platform, 2017. [На Интернету]. Доступно на https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28G%C3%B6schka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf

[5] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, 2017. [На Интернету]. Доступно на https://dokumen.pub/designing-data-intensive-applications-the-big-ideas-behind-reliable-scalable-and-maintainable-systems-9781491903100-9781449373320-1491903104.html/

[6] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd изд. Pragmatic Bookshelf, 2018. [На Интернету]. Доступно на https://pragprog.com/titles/mnee2/release-it-second-edition/

[7] Kubernetes, „Configure Liveness, Readiness and Startup Probes“. Приступљено: 03. Децембар 2025. [На Интернету]. Доступно на https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

[8] B. Burns, J. Beda, и K. Hightower, *Kubernetes: Up and Running*, 2nd изд. O'Reilly Media, 2018. [На Интернету]. Доступно на https://www.oreilly.com/library/view/kubernetes-up-and/9781492046523/

[9] I. Docker, „What is a Container?“. Приступљено: 03. Децембар 2025. [На Интернету]. Доступно на https://www.docker.com/resources/what-container/

[10] Cloud Native Computing Foundation, „Who We Are“. Приступљено: 03. Децембар 2024. [На Интернету]. Доступно на https://www.cncf.io/about/who-we-are/

[11] D. Merkel, „Docker: Lightweight Linux Containers for Consistent Development and Deployment“, *Linux Journal*, том 2014, изд. 239, 2014, [На Интернету]. Доступно на https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

[12] C. Richardson, „Microservices Architecture“. Приступљено: 03. Децембар 2025. [На Интернету]. Доступно на https://microservices.io/patterns/microservices.html

[13] N. Relic, „Kubernetes Fundamentals: How to Use Kubernetes Health Checks“. Приступљено: 03. Децембар 2025. [На Интернету]. Доступно на https://newrelic.com/blog/how-to-relic/kubernetes-health-checks

[14] K. A. Luniya, „Kubernetes Liveness, Readiness and Startup Probes: Keys to Container Health and Resilience“. Приступљено: 03. Децембар 2025. [На Интернету]. Доступно на https://hackernoon.com/kubernetes-liveness-readiness-and-startup-probes-keys-to-container-health-and-resilience

[15] T. Ranković, N. Pokornić, A. Pavlović, и M. Simić, „Monitoring the distributed cloud: Metric collection and aggregation“, у *2024 IEEE 22nd International Symposium on Intelligent Systems and Informatics (SISY)*, IEEE, 2024. [На Интернету]. Доступно на https://www.eventiotic.com/eventiotic/files/Papers/URL/4c596074-0ea5-4d06-979d-5128f2110041.pdf

[16] B. Beyer, C. Jones, J. Petoff, и N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. [На Интернету]. Доступно на https://sre.google/books/

[17] The Go Authors, „The Go Programming Language“. Приступљено: 03. Децембар 2024. [На Интернету]. Доступно на https://go.dev/

[18] Docker Inc., „Docker Engine API“. Приступљено: 03. Децембар 2024. [На Интернету]. Доступно на https://docs.docker.com/engine/api/

[19] OpenMetrics Working Group, „OpenMetrics Specification“. Приступљено: 03. Децембар 2024. [На Интернету]. Доступно на https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md

[20] Prometheus Authors, „Alertmanager“. Приступљено: 03. Децембар 2024. [На Интернету]. Доступно на https://prometheus.io/docs/alerting/latest/alertmanager/

[21] Docker Inc., „Docker Compose“. Приступљено: 03. Децембар 2024. [На Интернету]. Доступно на https://docs.docker.com/compose/

[22] Prometheus Authors, „Prometheus - Monitoring system & time series database“. Приступљено: 03. Децембар 2024. [На Интернету]. Доступно на https://prometheus.io/docs/introduction/overview/