УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Стефан Пејиновић

Радни оквир за провере активности сервиса у дистрибуираном рачунарству у облаку

ЗАВРШНИ РАД

Osnovne академске студије

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:

Радни оквир за провере активности сервиса у дистрибуираном рачунарству у облаку

ТЕКСТ ЗАДАТКА:

Имплементирати радни оквир за провере активности сервиса у дистрибуираном рачунарству у облаку. Систем треба да омогући надзор микросервисних компоненти путем провера активности заснованих на *HTTP*, *TCP* и *gRPC* протоколима, као и детекцију стања неактивности сервиса. Неопходно је реализовати механизам аутоматског опоравка сервиса у *Docker* окружењу, кроз поновно покретање контејнера након детектованог отказа. Поред тога, потребно је интегрисати излагање метрика у *OpenMetrics* формату и омогућити алармирање путем *Prometheus* и *Alertmanager* система.

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**:		Радни оквир за провере активности сервиса у дистрибуираном рачунарству у облаку
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)		6/40/23/3/17/0/0
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Примењене рачунарске науке и информатика
Предметна одредница/Кључне речи, **ПО**:		провере активности сервиса, микросервиси, дистрибуирани системи, Docker
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Овај рад приказује развој радног оквира за провере активности сервиса у дистрибуираном рачунарству у облаку. Развијено решење омогућава надзор микросервисних компоненти кроз HTTP, TCP и gRPC провере, детекцију неактивности и аутоматски опоравак сервиса у Docker окружењу. Рад обухвата дизајн архитектуре, имплементацију механизама за управљање животним циклусом сервиса, као и излагање метрика у OpenMetrics формату и конфигурацију алармирања путем Prometheus-a и Alertmanager-a.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		04.12.2025
Чланови комисије, **КО**:	Председник:	Др Жељко Вуковић, редовни професор
	Члан:	Др Милан Стојков, доцент
	Члан:	Потпис ментора
	Члан, ментор:	Др Милош Симић, доцент

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**:	Service Health-Check Framework for Distributed Cloud Computing
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)	6/40/23/3/17/0/0
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Applied computer science and informatics
Subject/Key words, **S/KW**:	service health checks, microservices, distributed systems, Docker
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad
Note, **N**:	
Abstract, **AB**:	This thesis presents the development of a service health-check framework for distributed cloud computing. The proposed solution enables monitoring of microservices through HTTP, TCP, and gRPC checks, detection of inactivity, and automated service recovery within a Docker environment. The work includes the system's architectural design, implementation of lifecycle management mechanisms, metric exposition in the OpenMetrics format, and alerting configuration using Prometheus and Alertmanager.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	04.12.2025

Defended Board, **DB**:		
	President:	Željko Vuković, 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 активност сервиса (енгл. *health-check*), детектују неправилности у раду система и омогућавају правовремене реакције система у циљу очувања стабилности.

Иако *Kubernetes* поседује уграђене механизме за провере активности сервиса, 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 оквиром (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 систем 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 активност (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 система и његову интеграцију у различита окружења.

Циљ овог рада јесте развој самосталног радног оквира 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, радни оквир 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 радног оквира.

Друго поглавље представља теоријске концепте неопходне за разумевање проблема и контекста у коме систем 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 систем 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 систем 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 радног оквира у пракси.

Глава 2

Теоријске основе и постојећа решења

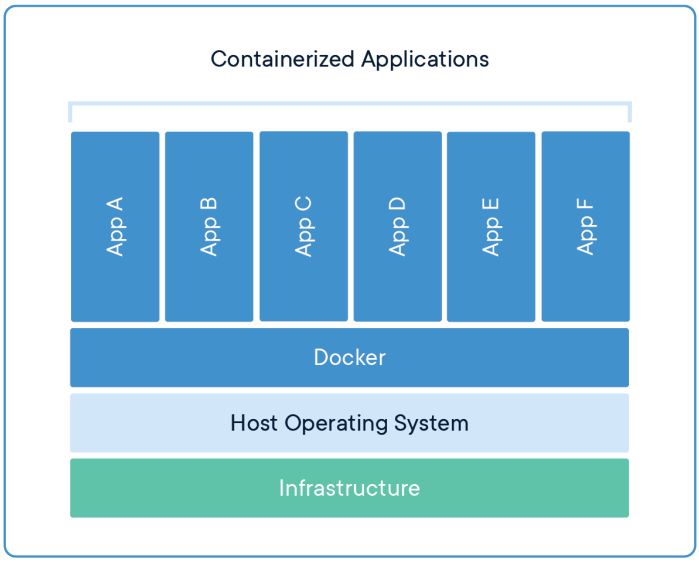
Ово поглавље описује теоријске концепте и технологије које су неопходне за разумевање развијеног решења. На почетку су представљени основни принципи контејнеризације, микросервисних архитектура и дистрибуираних система, као и изазови који се јављају у таквим окружењима. Затим су обрађени механизми провере 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.



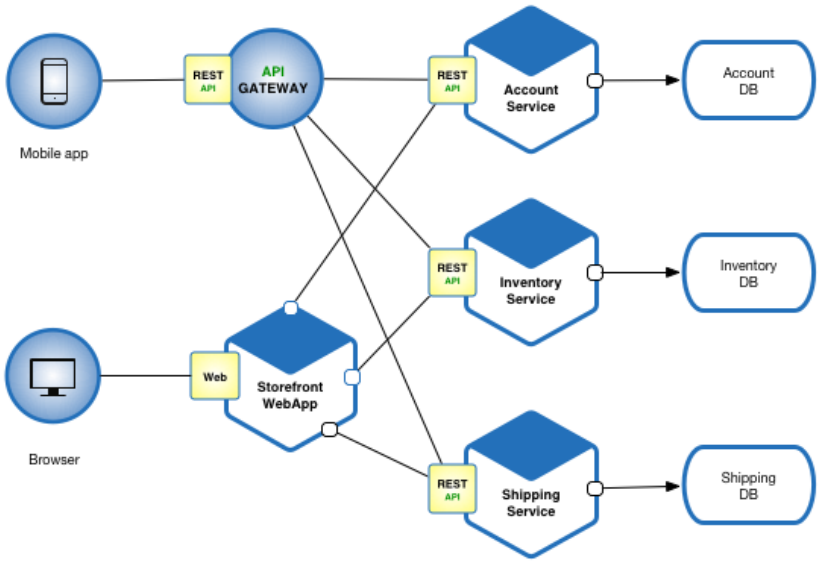
Слика 1: Архитектура контејнеризованог окружења [9]

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.

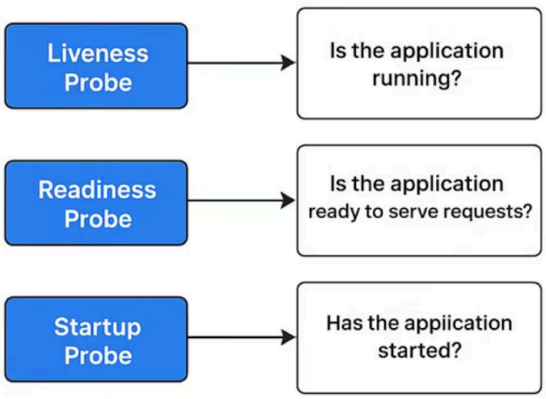
Надгледање дистрибуираних система је знатно сложеније од надгледања монолитних апликација [8]. Информације о стању система су разложене на десетине независних сервиса, од којих сваки генерише сопствене логове и метрике. Због тога су механизми за праћење здравља система критични за очување стабилности,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

Радни оквир 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 провере активности сервиса, омогућавајући радном оквиру 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 радним оквиром за проверу активности 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 систем 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 радног оквира 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 активности сервиса,
- 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 активности сервиса

Систем мора да периодично извршава провере покретања, спремности и активности 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 активност сервиса.

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.

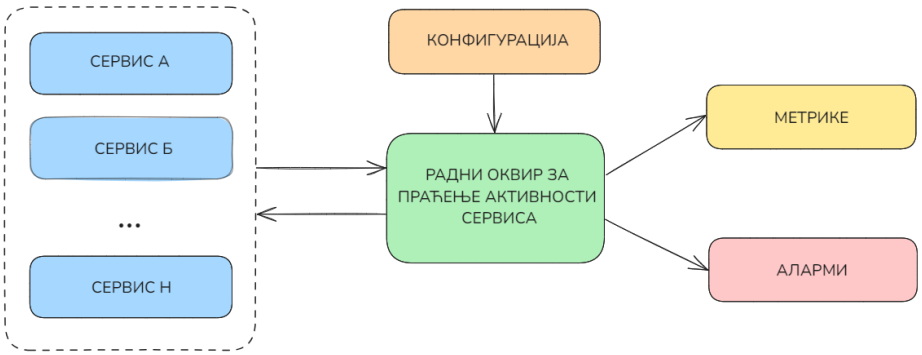
Глава 4

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 система, 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 радног оквира 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 радном оквиру, чиме се постиже флексибилност и лако одржавање.

4.1.2 Радни оквир за праћење активности сервиса

Радни оквир за праћење активности сервиса 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. Радни оквир функционише као централни чвор који координише интеракцију између свих осталих компоненти система.

4.1.3 Надгледани сервиси

Надгледани сервиси представљају апликације, микросервисе или друге функционалне јединице чија је активност предмет праћења. Радни оквир се периодично повезује са сваким сервисом како би утврдио да ли он функционише у очекиваном режиму. На основу добијене повратне информације, радни оквир 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, радни оквир 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 радни оквир реализује. Оне представљају основне токове извршавања у оквиру система и описују на који начин радни оквир 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 радног оквира за проверу активности сервиса.

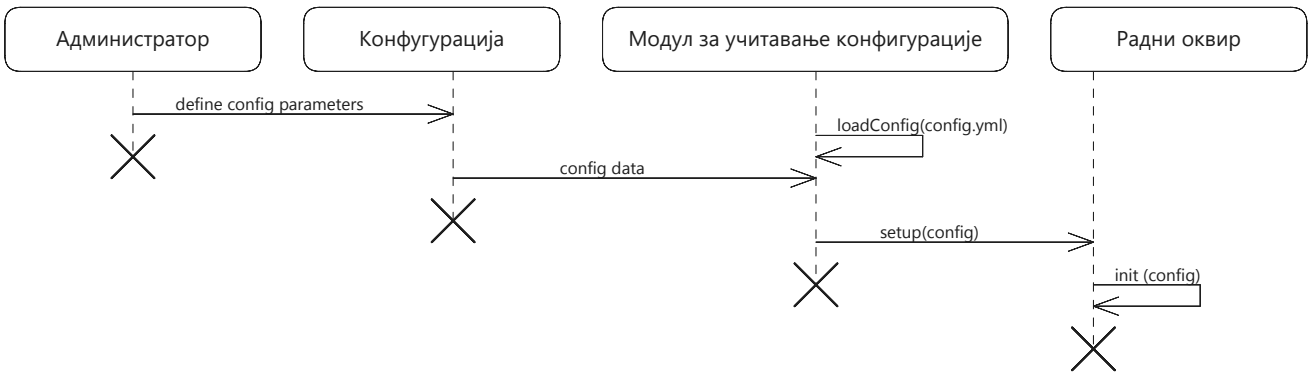
Функционалност	Опис
Иницијализација система	Радни оквир чита *YAML* фајл, поставља почетна стања и припрема компоненте за рад.
Извршавање циклуса провера активности сервиса	Радни оквир периодично упућује захтеве дефинисаним сервисима како би проверио њихову доступност.
Обрада резултата провере	Одговори добијени од сервиса анализирају се како би се утврдила успешност провера.
Управљање животним циклусом сервиса	На основу резултата систем ажурира тренутно стање посматраног сервиса.
Руковање неуспесима и покретање корективних акција	Када се детектује више узастопних неуспеха, систем 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 систем генерише аларме.

4.2.1 Иницијализација система

Радни оквир 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, радни оквир 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. Радни оквир за проверу активности сервиса затим обрађује преузете податке, иницијализује своје унутрашње компоненте, поставља почетна стања сервиса и припрема систем за извршавање периодичних циклуса провере активности. 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 систем чека пре извршавања прве провере.
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.

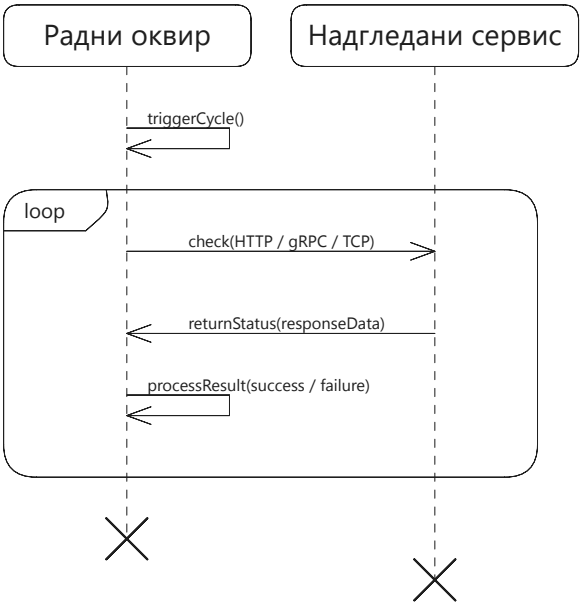
4.2.2 Извршавање циклуса провере активности сервиса

Радни оквир периодично покреће циклус провере активности сервиса у складу са интервалима дефинисаним у конфигурационој датотеци. Током сваког циклуса радни оквир приступа сервисима који су наведени за надзор и одређује врсту провере (покретање, спремност, активност) као и метод комуникације који је предвиђен за тај сервис (*HTTP*, *TCP* или *gRPC*). Након припреме параметара провере радни оквир иницира комуникацију упућивањем захтева за проверу активности 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, систем 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 система 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 циклуса провере активности сервиса 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 циклуса провере активности сервиса, комуникацију са сервисом, као и обраду и бележење резултата провере.



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

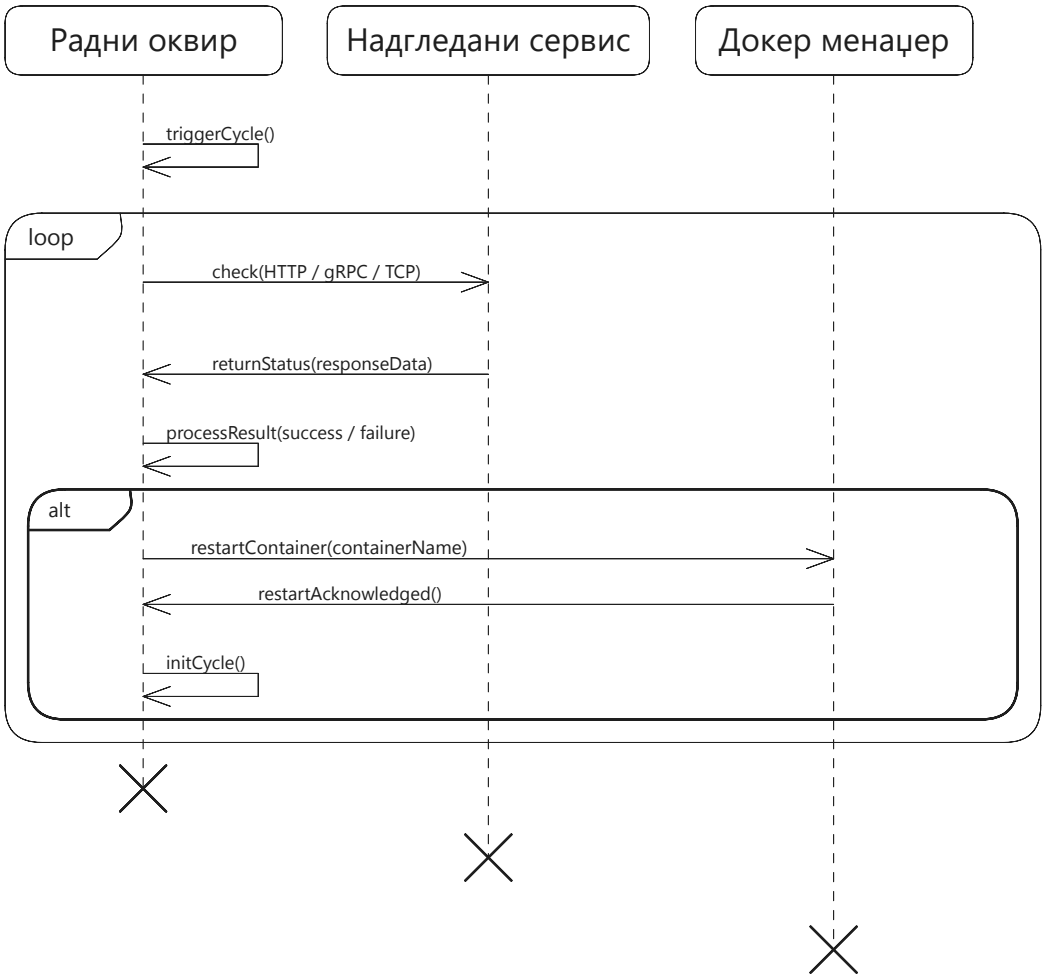
4.2.4 Управљање животним циклусом сервиса

Након обраде резултата појединачних провера, систем ажурира стање сервиса на основу више узастопних успеха или неуспеха. Оцену не одређује једна провера, већ се посматра понашање сервиса током више узастопних циклуса провере активности, 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 провера активности, односно као спреман (енгл. *Ready*) или неспреман (енгл. *NotReady*) у случају провера спремности. 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 систем 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 систем активира корективну меру са циљем опоравка сервиса. У овом систему корективна мера подразумева поновно покретања сервиса у оквиру *Docker* окружења, чиме се омогућава да систем аутоматски реагује на детектовани проблем.

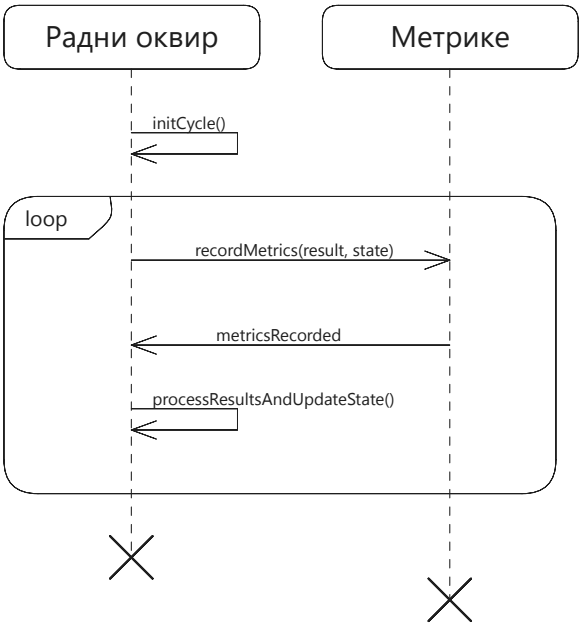
Поступак поновног покретања се спроводи тако што систем упућује команду *Docker*-у да поново покрене одговарајући контејнер. Након рестарта сервис улази у нови циклус провере активности сервиса, где се поново процењује његово стање на основу резултата наредних провера. На слици 7 приказан је дијаграм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



Слика 7: Дијаграм секвенце који описује циклус провере активности сервиса са корективним акцијама.

4.2.6 Генерисање и бележење метрика

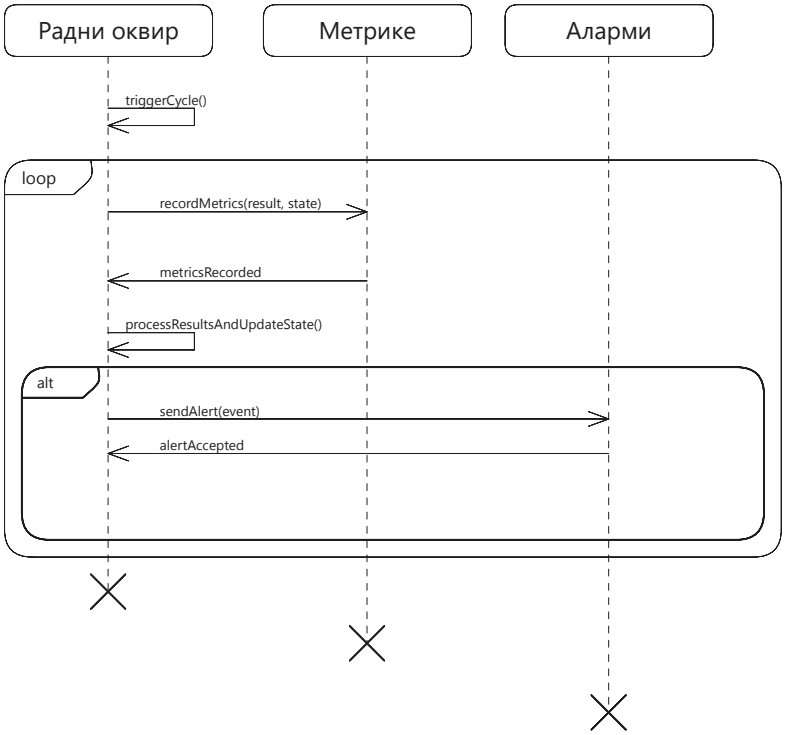
Током рада система систем континуирано генерише метрике које описују понашање надзираних сервиса и самог механизма надзора. На основу резултата појединачних провера и ажурираних стања сервиса бележе се, између осталог, информације о броју извршених провера, броју успешних и неуспешних провера по сервису, 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, систем 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, систем 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

Имплементација система

Имплементациони део рада разматра конкретну реализацију развијеног радног оквира и описује како су концепти представљени у архитектури претворени у функционалну целину. У овом поглављу приказана су кључна техничка решења која омогућавају учитавање конфигурације, извршавање провера активности сервиса, управљање стањем сервиса, спровођење корективних мера, као и генерисање метрика и алармних сигнала. Радни оквир је имплементиран у програмском језику Go, чији механизми конкуренције засновани на горутинима (енгл. *goroutines*) и модуларна структура пакета омогућавају ефикасно паралелно извршавање провера, јасну организацију кода и једноставну интеграцију са екстерним компонентама.

У наставку се разматра конкретна програмска реализација главних функција система. Приказана је структура пројекта, намена основних пакета, обрада конфигурационе датотеке и кључни делови програмске логике који управљају проверама и стањем сервиса. Додатно су описани механизми конкурентног извршавања, комуникација са *Docker* окружењем и интеграција са системима за метрике и алармирање. На овај начин поглавље пружа увид у техничке аспекте радног оквира и објашњава како су идеје из претходног поглавља реализоване у пракси.

5.1 Организација пројекта

Пројекат је организован у складу са *Go* конвенцијама, где је код подељен на пакете према функционалној одговорности. У кореном директоријуму налазе се улазна тачка апликације и конфигурациона датотека, док је имплементација радног оквира смештена у директоријум *internal*, који садржи доменску логику надзора, интеграцију са *Docker* окружењем, систем метрика и пратеће помоћне модуле. Оваква организација обезбеђује јасно раздвајање одговорности и спречава употребу интерних компоненти од стране спољашњих пакета. У табели 3 приказани су основни пакети и њихова улога у систему.

Табела 3: Организација пројекта

Пакет / датотека	Улога у систему
config/	Садржи *config.yml* и модул *config.go* задужен за учитавање, парсирање и валидирање конфигурације.
internal/communication	Комуникација са сервисима који се прате (*HTTP*, *TCP*, *gRPC*). Јединствен интерфејс који логика за проверу активности користи без обзира на протокол.
internal/helpers	Пакет са помоћним функцијама које се користе у више модула система.
internal/metrics	Имплементација метрика (нпр. бројачи успешних, неуспешних провера…).
internal/monitoring	Садржи конфигурационе датотеке за *Prometheus* и *Alertmanager*.
internal/operations	Имплементација операција над *Docker* окружењем (нпр. поновно покретање сервиса).
internal/probes	Логика за rad са свим типовима проверама (покретање, спремност, активност). Повезује конфигурацију са конкретним типом провере и начином извршавања.
internal/runner	Покретање горутина за периодичне провере и координација свих компоненти надзора.
main.go	Покреће главни циклус радног оквира. Учитава конфигурацију и иницијализује компоненте.

5.2 Конфигурација система

Конфигурација представља један од кључних елемената развијеног система, јер одређује које се сервиси посматрају, које се провере над њима извршавају и под којим условима се доносе одлуке о исправности. Уместо да се ови параметри уграђују у изворни код, они су издвојени у посебну конфигурациону датотеку у *YAML* формату, што омогућава да се понашање система прилагоди различитим окружењима без потребе за изменама у имплементацији. Конфигурацију обично припрема особа која познаје карактеристике сервиса који се надзиру, јер управо од правилно постављених интервала, прагова и типова провера зависи стабилност и поузданост надзорног система.

5.2.1 Структура *YAML* конфигурације

Конфигурациона датотека организована је као листа контејнера (енгл. *containers*), где за сваки контејнер постоји назив сервиса и дефиниције провера покретања, спремности и активности. Свака провера садржи опис метода комуникације (*HTTP*, *TCP* или *gRPC*), адресу сервиса (путања, порт, *host*), и временске параметре као што су почетно кашњење (енгл. *initialDelaySeconds*), интервал између провера (енгл. *periodSeconds*), праг за број узастопних неуспеха (енгл. *failureThreshold*) и опциони timeout (енгл. *timeoutSeconds*). На листингу 1 приказан је пример исправне *YAML* конфигурације за један сервис.

```yaml
containers:
  - name: service-a
    startupProbe:
      httpGet:
        path: /startup
        port: 8080
        host: service-a
      initialDelaySeconds: 0
      periodSeconds: 2
      failureThreshold: 30

    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        host: service-a
      initialDelaySeconds: 5
      periodSeconds: 10
      failureThreshold: 3

    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
        host: service-a
      initialDelaySeconds: 2
      periodSeconds: 5
      failureThreshold: 3
```

Листинг 1: Пример исправне *YAML* конфигурације.

Овај пример илуструје случај HTTP провера за сервис који подржава три различита типа провера. Иста структура примењује се и за сервисе који користе *gRPC* или *TCP* провере, уз одговарајућу измену секција *grpc*: или *tcpSocket*:.

5.2.2 Мапирање конфигурације на *Go* структуре

Структура *YAML* датотеке директно је пресликана на модел података у програмском језику *Go*. Основни тип *Config* садржи листу контејнера, док је сваки контејнер представљен структуром *Container*. Параметри провера описани су структуром *Probe*, која подржава три различита модела комуникације путем опционих поља. На листингу 2 приказане су *Go* структуре које описују конфигурацију.

```
type Config struct {
  Containers []Container `yaml:"containers"`
}

type Container struct {
  Name           string `yaml:"name"`
  StartupProbe   *Probe `yaml:"startupProbe,omitempty"`
  LivenessProbe  *Probe `yaml:"livenessProbe,omitempty"`
  ReadinessProbe *Probe `yaml:"readinessProbe,omitempty"`
}

type Probe struct {
  HTTPGet             *HTTPGet   `yaml:"httpGet,omitempty"`
  TCPSocket           *TCPSocket `yaml:"tcpSocket,omitempty"`
  GRPC                *GRPC      `yaml:"grpc,omitempty"`
  InitialDelaySeconds int        `yaml:"initialDelaySeconds,omitempty"`
  PeriodSeconds       int        `yaml:"periodSeconds"`
  FailureThreshold    int        `yaml:"failureThreshold"`
  TimeoutSeconds      int        `yaml:"timeoutSeconds,omitempty"`
}

type HTTPGet struct {
  Path string `yaml:"path"`
  Port int    `yaml:"port"`
  Host string `yaml:"host"`
}

type TCPSocket struct {
  Port int    `yaml:"port"`
  Host string `yaml:"host"`
}

type GRPC struct {
  Port    int    `yaml:"port"`
  Host    string `yaml:"host,omitempty"`
  Service string `yaml:"service,omitempty"`
}
```

Листинг 2: Пример исправне YAML конфигурације.

5.2.3 Учитавање и валидирање конфигурације

При покретању система конфигурациона датотека се учитава коришћењем *YAML unmarshal* механизма, који садржај датотеке преводи у одговарајуће *Go* структуре. Након парсирања врши се додатна валидација, чији је циљ да се открију логичке грешке у конфигурацији пре него што надзор почне са радом.

Примери валидаторских правила укључују:

- листа *containers* не сме бити празна
- сваки контејнер мора имати бар један дефинисан тип провере
- вредности *periodSeconds* и *failureThreshold* морају бити веће од нуле
- комбинација параметара мора бити логички исправна (нпр. *TCP* провера не сме садржати *path*)

У случају погрешне или непотпуне конфигурације систем одмах враћа грешку, спречавајући покретање система провере активности сервиса са неисправним параметрима.

5.3 Имплементација механизма провере активности сервиса

Механизам активности сервиса представља језгро радног оквира. Он повезује конфигурацију дефинисану у YAML датотеци са конкретним *HTTP*/*TCP*/*gRPC* проверама које се периодично извршавају над циљаним сервисима. Основна идеја је да се свака провера (покретање, спремност, активност) опише на декларативан начин, а да радни оквир преузме бригу о њеном извршавању.

5.3.1 Интерни модел провера

Након учитавања конфигурације, свака дефинисана провера се пресликава у интерни модел који садржи:

- тип провере (*ProbeType*: *liveness*, *readiness*, *startup*),
- протокол (*HTTP*, *TCP* или *gRPC*),
- циљ (*target*: *hostname*, порт и, по потреби, путања),
- параметре извршавања:
  - период понављања провере (*periodSeconds*),
  - максимално трајање једног покушаја (*timeoutSeconds*),
  - број узастопних успеха и неуспеха након којих се мења стање (*successThreshold*, *failureThreshold*),
  - почетно кашњење пре прве провере (*initialDelaySeconds*).

На тај начин систем не ради директно над *YAML* структуром, већ над јасно дефинисаним објектима који описују шта тачно треба проверавати за сваки сервис и сваки тип провере.

5.3.2 Покретање провера

За сваки описани тип провере систем покреће посебну позадинску горутину. Свака таква горутина представља једну петљу провере која:

- по потреби сачека иницијално кашњење,
- у регуларним интервалима (према *periodSeconds*) иницира појединачну проверу,
- прикупља исход провере (успех или неуспех, трајање извршавања, опис грешке),
- резултат шаље на заједнички канал који користе компоненте за управљање животним циклусом и за метрике.

Оваква организација омогућава да се сваки тип провере извршава независно од других, без блокирања главне нити програма. Систем користи контексте (енгл. *context*) како би могао да заустави све активне провере приликом гашења или поновног покретања.

5.3.3 *HTTP* провере

За HTTP провере конструише се *URL* на основу *hostname*-а, порта и путање задате у конфигурацији. Провера се извршава као *HTTP GET* захтев, уз ограничење максималног трајања (енгл. *timeout*).

Провера се сматра успешном ако је:

- успостављена мрежна конекција,
- стигао одговор у оквиру задатог временског ограничења,
- статусни код у опсегу 2xx или 3xx.

У случају да дође до прекорачења ограниченог максималног трајања или статусног кода у формату (4xx, 5xx), провера се бележи као неуспех. Поред самог исхода, мери се и трајање захтева, што се накнадно користи за метрике. На листингу 3 приказана је функција *HTTP* провере.

```
func HTTPCheck(ctx context.Context, t Target) (bool, error) {
  timeout := helpers.GetTimeout(t.Timeout)
  client := &http.Client{Timeout: timeout}

  url := helpers.BuildHTTPURL(t.Host, t.Port, t.Path)
  req, _ := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)

  resp, err := client.Do(req)
  if err != nil {
    return false, err
  }
  defer resp.Body.Close()

  return helpers.IsSuccessStatus(resp.StatusCode), nil
}
```

Листинг 3: Функција *HTTP* провере.

5.3.4 *TCP* провере

TCP провере су намењене сервисима који немају *HTTP* интерфејс, али је битно да је порт доступан. У том случају, провера се своди на покушај успостављања *TCP* конекције ка задатом *host*-у и порту, у оквиру дефинисаног максималног временског трајања.

- Ако је конекција успешно успостављена и одмах затворена, провера се сматра успешном.
- Ако успостављање конекције истекне или јави грешку, провера се сматра неуспешном.

Оваква провера је лагана и не захтева разумевање протокола изнад *TCP*-а.

5.3.5 *gRPC* провере

За *gRPC* провере систем користи стандардизовани *gRPC health checking* протокол (сервис *grpc.health.v1.Health*). Да би провера функционисала, циљни *gRPC* сервис треба да имплементира овај сервис.

Приликом овог типа провере радни оквир ради следеће:

- успоставља *gRPC* конекцију ка задатом *host*-у и порту, са задатим временским ограничењем,
- позива метод *Check*, уз назив *gRPC* сервиса који се проверава,
- тумачи добијени статус:
  - ако је статус *SERVING* онда провера је успешна,
  - сваки други статус или грешка приликом позива говори да се провера сматра неуспешном.

И у овом случају се бележе и исход и трајање, како би се накнадно ажурирале метрике.

5.3.6 Механизам за обраду резултата

Иако *HTTP*, *TCP* и *gRPC* провере користе различите протоколе, систем их на излазном нивоу третира на исти начин. Сваки појединачни покушај провере производи структуру резултата која садржи:

- идентификацију провере (тип, име, контејнер, протокол),
- логички исход (успех/неуспех),
- евентуалну грешку (нпр. *status code* 500),
- трајање провере,
- временски печат.

Ови резултати се шаљу на заједнички канал који користе:

- компонента за управљање животним циклусом сервиса, која на основу резултата провере конекције и дозвољеног броја грешки мења стање сервиса и евентуално иницира поновно покретање сервиса;
- компонента за метрике, која резултате преводи у *Prometheus counter* метрике.

5.4 Имплементација управљања животним циклусом сервиса

Након што сисетм прикупи резултате свих провера (*HTTP*/*TCP*/*gRPC*), потребно је донети одлуку о томе у ком се стању циљни сервис тренутно налази, као и како наш радни оквир да реагује. Управљање животним циклусом представља механизам који на основу бројача неуспешних провера, броја дозвољених неуспешних провера и типова провера одређује:

- да ли је сервис активан или није (*Healthy* или *Unhealthy*),
- да ли је сервис спреман или није (*Ready* или *NotReady*),
- да ли је фаза покретања успешно завршена,
- да ли је потребно иницирати поновно покретање сервиса,
- када се мења унутрашње стање система.

Цео механизам може се посматрати као континуирана обрада резултата које шаљу позадинске горутине задужене за провере активности сервиса.

5.4.1 Модел стања сервиса

Сваки сервис који систем надгледа има три независне компоненте стања:

1. Стање активности (*liveness*) Одређује да ли је сервис активан, односно да ли је у функционалном стању да настави са радом. Ако ова провера узастопно не успева, сервис се сматра *Unhealthy* и систем може извршити поновно покретање сервиса.
2. Стање спремности (*readiness*) Одређује да ли је сервис спреман да прихвата саобраћај. Ова провера не подразумева да сервис није активан, већ само да можда тренутно није у стању да успешно обрађује захтеве.
3. Стање покретања (*startup*) Представља посебну фазу рада сервиса, у којој се очекује да сервис стабилизује своје окружење пре него што се укључи логика за проверу активности сервиса. Уколико постоје, прво се изврше ове провере па тек након успешноси крећу провере активности и спремности сервиса.

5.4.2 Управљање стањем активности

Стање активности представља основну способност сервиса да функционише без проблема. Ово стање се утврђује на основу резултата провера активности које се извршавају у задатим интервалима. Сваки успешан исход провере указује на то да је сервис активан, док сваки неуспех повећава бројач неуспешних провера. Када број узастопних неуспеха достигне дефинисан праг толеранције, сервис се сматра неактивним, односно нефункционалним. На листингу 4 приказан је део кода који реализује ову логику.

```
ok, _ := service.Check(ctx, s.Target)
if ok {
    // логика када систем задржава активно стање
    return
}

*fail++
if !*bad && *fail >= s.FailureThreshold {
    // логика када систем треба да пређе у неактивно стање
    }
}
```

Листинг 4: Обрада исхода провере стања активности.

Уколико сервис пређе у стање неактивности, систем иницира поновно покретање одговарајућег Docker контејнера. Након успешног поновног покретања, сви бројачи се враћају на почетне вредности, а цео животни циклус надзора почиње поново. На овај начин систем не само да детектује квар, већ и спроводи механизам аутоматског опоравка, обезбеђујући континуитет рада система.

5.4.3 Управљање стањем спремности

Стање спремности одражава да ли је сервис у датом тренутку способан да прихвата и обрађује захтеве. За разлику од стања активности, прелазак у стање неспремности не доводи до поновног покретања контејнера, већ служи искључиво за контролу усмеравања саобраћаја у оквиру система.

Сваки неуспех провере спремности повећава бројач узастопних неуспеха, а када он достигне праг отказа, сервис се означава као неспреман. Уколико се провере накнадно стабилизују, сервис се враћа у стање спремности. Ова логика омогућава да сервис остане видљив у систему као функционалан, али привремено недоступан за обраду захтева.

5.4.4 Фаза покретања и прелазак у стабилно стање

Стање покретања представља уводну фазу рада сервиса, током које се очекује да сервис изврши све иницијалне радње пре него што буде третиран као у потпуности стабилан. Током ове фазе систем не реагује на неуспехе провера активности и спремности, јер се одређени степен нестабилности сматра уобичајеним приликом подизања сервиса.

Када провере покретања покажу да је сервис стабилан, он прелази у редовно оперативно стање у којем почињу да важе механизми за надзор активности и спремности. Уколико сервис дуже време не успе да превазиђе фазу покретања, односно ако број узастопних неуспеха достигне праг отказа, сервис се означава као нефункционалан без уласка у редовни животни циклус. На овај начин спречава се да сервис остане у стању трајне иницијалне нестабилности.

5.5 Интеракција са *Docker* окружењем и механизам поновног покретања

Поновно покретање контејнера представља механизам опоравка система у случају када сервис пређе у стање неактивности. Ова функционалност се ослања на резултате провера стања активности и омогућава да систем аутоматски реагује на поремећаје у раду сервиса. Пошто се надгледани сервиси извршавају у *Docker* контејнерима, неопходна је директна интеграција са *Docker* окружењем ради контролисаног управљања животним циклусом контејнера.

5.5.1 Повезивање са Docker окружењем

Интеграција са *Docker*-ом реализована је у посебном слоју, кроз структуру *Manager* која садржи *Docker API* клијент. При иницијализацији система креира се један *Docker* клијент који користи подешавања окружења (променљиве окружења и верзију *API*-ја), након чега се преко њега приступа *Docker Engine*-у.

На тај начин систем може да пронађе контејнер на основу његовог логичког имена, да приступи основним информацијама о његовом стању и, што је најважније, да спроведе операцију поновног покретања. Сама апликација која ради у контејнеру не мора да буде свесна постојања нашег радног оквира, јер се све управљачке операције извршавају на нивоу *Docker* окружења.

5.5.2 Детекција потребе за поновним покретањем

Потреба за поновним покретањем сервиса настаје онда када провере стања активности дуже време не успевају. Систем за сваки сервис води бројач узастопних неуспеха и, када овај бројач достигне дефинисани праг отказа, сервис се означава као неактиван.

У том тренутку механизам управљања животним циклусом шаље сигнал у унутрашњи канал за пријаву проблема, којим се обавештава главна контролна петља да је потребно спровести поновно покретање контејнера. На тај начин резултати провера стања активности имају директан оперативни ефекат.

5.5.3 Поновно покретање контејнера и понашање система након тога

Операција поновног покретања контејнера реализована је у оквиру *Manager* структуре, која преко *Docker API*-ја позива одговарајућу функцију за поновно покретање. На листингу 5 приказана је функција која реализује ову логику.

```
func (m *Manager) RestartContainer(ctx context.Context, name string) error {
    // позив Docker API-ја за поновно покретање контејнера
    return m.cli.ContainerRestart(ctx, name, opts)
}
```

Листинг 5: Поновно покретање *Docker* контејнера преко *Docker API*-ја.

Функција *RestartContainer* прима контекст извршавања и име контејнера, припрема опције за заустављање са задатим временским ограничењем и затим позива *Docker API*. На овај начин логика поновног покретања издвојена је у посебан слој, што поједностављује њену употребу у главној петљи и омогућава да се промена начина комуникације са *Docker* окружењем обави на једном месту.

Након успешног поновног покретања контејнера, систем враћа бројаче неуспеха на почетне вредности и сервис улази у фазу покретања. У тој фази се најпре примењују провере стања покретања, а затим, по стабилизацији сервиса, и провере стања активности и спремности.

5.6 Механизам метрика и алармирања

Поред провера активности и механизма поновног покретања, радни оквир обезбеђује и систем метрика који омогућава праћење понашања сервиса током времена, као и алармирање у случају поремећаја. Метрике пружају увид у динамику рада система и представљају основу за благовремено уочавање неправилности. Алармирање користи управо те податке како би обавестило администраторе када сервис дуже време није у исправном стању.

5.6.1 Механизам метрика

Бележење метрика реализује се у складу са *OpenMetrics* форматом, који је у потпуности усаглашен са *Prometheus*-ом. Систем бележи резултате свих провера активности сервиса и излаже их на путањи */metrics* у задатом формату. За сваку проверу записују се:

- исход (успех или неуспех),
- трајање провере,
- број узастопних неуспеха у тренутку мерења,
- назив сервиса и тип провере.

Метрике су конфигурисане као бројачи (*counter*), што је омогућило једноставно праћење трендова током времена. На листингу 6 приказано је део имплементације који дефинише метрике.

```
// Дефиниција метрике за успешне провере
ProbeSuccess = prometheus.NewCounterVec(
    prometheus.CounterOpts{
        Name: "wh_probe_success_total",
        Help: "Number of successful probe checks.",
    },
    []string{"probe", "target", "container", "probe_type"},
)
```

Листинг 6: Дифинисање метрика за успешне провере.

5.6.2 Механизам алармирања

Алармирање у систему реализовано коришћењем стандардног *Prometheus* екосистема. Систем једино излаже метрике у *OpenMetrics* формату, док *Prometheus* преузима те податке и на основу њих покреће правила за аларме дефинисана у *Alertmanager*-у. На тај начин систем остаје фокусирани механизам за надзор и опоравак сервиса, док се логика алармирања препушта специјализованом алату који је за то предвиђен.

Најважнија метрика за алармирање је *wh_container_restarts_total*, која бележи колико пута је сервис поново покренут услед неактивности. На основу ње дефинисан је аларм који се активира ако број поновних покретања прешао задати праг у одређеном временском периоду. На листингу 7 приказано је дефинисање правила које детектује учестала поновна покретања контејнера на основу метрике *wh_container_restarts_total*.

```
groups:
  - name: whirlpool_launcher_restart_alerts
    interval: 30s
    rules:
      - alert: ContainerRestartBurst
        expr: increase(wh_container_restarts_total[${RESTART_WINDOW_MIN}m]) >=
${RESTART_THRESHOLD}
        labels:
          severity: critical
        annotations:
          summary: "Container {{ $labels.container }} restarted too many times!"
          description: |
            Container: {{ $labels.container }}
            Target endpoint: {{ $labels.target }}
            Probe: {{ $labels.probe }}
            Probe type: {{ $labels.probe_type }}
            Number of restarts: {{ $value }}
```

Листинг 7: Дифинисање правила за аларм.

Alertmanager је конфигурисан да у случају активирања овог правила пошаље *e-mail* поруку администратору. У глобалној секцији подешени су *SMTP* параметри, а унутар пријемника *email-alerts* дефинисан је *HTML* шаблон за поруку. На листингу 8 приказана је конфигурација пријемника који шаље *e-mail* обавештење администратору.

```
receivers:
  - name: 'email-alerts'
    email_configs:
      - to: '${ALERT_EMAIL_TO}'
        headers:
          Subject: 'WHIRLPOOL_LAUNCHER ALERT: {{ .GroupLabels.container }}'
        html: |
          <h2>Container restarted too many times!</h2>
          {{ range .Alerts }}
          <h3>{{ .Annotations.summary }}</h3>
          <ul>
            <li><strong>Container:</strong> {{ .Labels.container }}</li>
            <li><strong>Target endpoint:</strong> {{ .Labels.target }}</li>
            <li><strong>Probe:</strong> {{ .Labels.probe }}</li>
            <li><strong>Probe type:</strong> {{ .Labels.probe_type }}</li>
            <li><strong>Restarts:</strong> {{ .Annotations.description }}</li>
          </ul>
          <hr>
          {{ end }}
```

Листинг 8: Конфигурација *Alertmanager* пријемника за *e-mail* обавештења.

На овај начин је реализована потпуна аутоматизација обавештавања где систем излаже метрике,*Prometheus* их прикупља, а *Alertmanager* детектује учестала поновна покретања и шаље упозорење администратору путем *e-mail*-а.

5.7 Тестирање система

Тестирање система изведено је коришћењем посебних тест сервиса који симулирају различита понашања надгледаних сервиса. За сваки подржани протокол *HTTP*, *TCP* и *gRPC* припремљене су варијанте сервиса који могу да враћају позитиван одзив или контролисано прелазе у негативно стање након дефинисаног временског периода. На овај начин омогућено је тестирање исправности свих механизама система, као што су провера стања, реакције на деградацију и поновно покретање контејнера у случају учесталих неуспеха.

Систем је тестирањем изложен сценаријима стабилног рада, постепеног погоршавања одзива и потпуне недоступности сервиса. Негативни сценарији омогућили су проверу да ли систем правилно детектује стање неактивности и иницира поновно покретање контејнера након достизања прага за отказ. Тиме је потврђено да систем исправно покрива очекиване случајеве у реалним условима рада.

Глава 6

Закључак

Развијено решење представља самосталан радни оквир за надзор и опоравак микросервисних компоненти, заснован на периодичним проверама активности и аутоматском управљању животним циклусом сервиса. Уведена логика за детекцију неактивности, праћење стања компоненти и иницирање поновног покретања омогућила је стабилнији и предвидљивији рад система у динамичким условима. Кључни механизми провера покретања, спремности и активности примењени су у стандардизованом облику, што омогућава флексибилну имплементацију у различитим архитектурама. Интеграција са *Docker* окружењем и *Prometheus* екосистемом остварена је без измена у самом сервису, што решење чини лако примењивим у постојећим инфраструктурним системима.

Радни оквир обезбеђује јасно раздвојене компоненте за конфигурацију, надзор, управљање и извештавање, што доприноси његовој проширивости и читљивости кода. Систем је проверен у реалистичним условима симулирања различитих типова отказа, чиме је потврђена исправност кључних механизама. На основу резултата тестирања може се закључити да предложени приступ успешно решава постављене циљеве и представља поуздану основу за даљи развој.

6.1 Предности развијеног решења

Главна предност решења је једноставност имплементације и коришћења. Систем не захтева измене у апликационом коду надгледаних сервиса и може се применити у различитим окружењима без зависности од специфичне платформе. Модуларна архитектура омогућава лако проширивање новим типовима провера, метрика или политика опоравка. Интеграција са *Docker*-ом и *Prometheus*-ом реализована је на стандардан начин, што олакшава укључивање у постојеће токове рада. Флексибилна конфигурација дозвољава брзо прилагођавање различитим захтевима без потребе за већим интервенцијама у систему.

6.2 Мане и ограничења

Тренутна имплементација не излаже све метрике које би биле корисне за детаљнији увид у понашање система. Недостаје комплетан сет аларма за све релевантне случајеве деградације или отказа сервиса. Логика опоравка је ограничена на поновно покретање контејнера без напреднијих стратегија. Систем такође не врши детаљну анализу узрока проблема нити пружа дубљу дијагностику стања пре иницирања опоравка

6.3 Правци даљег развоја

Наредна фаза развоја обухватиће увођење нових провера, проширење скупа метрика и имплементацију комплетнијег система алармирања. Природа продукционог окружења у коме ће систем бити коришћен захтева да ове компоненте буду имплементиране као одвојени модули који ће радити на различитим локацијама у инфраструктури. Овакав распоређени приступ омогућиће боље покривање потреба система и детаљнији увид у понашање сервиса, уз очување једноставности и модуларности основног решења.

Списак слика

Списак листинга

Списак табела

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/