УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Владислав Петковић

Имплементација опсервабилности дистрибуиране микросервисне веб квиз апликације

ЗАВРШНИ РАД
- Основне академске студије -

Нови Сад, 2026

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		
Тип записа, **ТЗ**:		
Врста рада, **ВР**:		
Аутор, **АУ**:		
Ментор, **МН**:		
Наслов рада, **НР**:		
Језик публикације, **ЈП**:		
Језик извода, **ЈИ**:		
Земља публиковања, **ЗП**:		
Уже географско подручје, **УГП**:		
Година, **ГО**:		
Издавач, **ИЗ**:		
Место и адреса, **МА**:		
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		
Научна област, **НО**:		
Научна дисциплина, **НД**:		
Предметна одредница/Кључне речи, **ПО**:		
УДК		
Чува се, **ЧУ**:		
Важна напомена, **ВН**:		
Извод, **ИЗ**:		
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	
	Члан:	Потпис ментора
	Члан, ментор:	

a a., “ number, **ANO**:		
Identification number, **INO**:		
Document type, **DT**:		
Type of record, **TR**:		
Contents code, **CC**:		
Author, **AU**:		
Mentor, **MN**:		
Title, **TI**:		
Language of text, **LT**:		
Language of abstract, **LA**:		
Country of publication, **CP**:		
Locality of publication, **LP**:		
Publication year, **PY**:		
Publisher, **PB**:		
Publication place, **PP**:		
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		
Scientific field, **SF**:		
Scientific discipline, **SD**:		
Subject/Key words, **S**/**KW**:		
UC		
Holding data, **HD**:		
Note, **N**:		
Abstract, **AB**:		
Accepted by the Scientific Board on, **ASB**:		
Defended on, **DE**:		
Defended Board, **DB**:	President:	
	Member:	Menthor's sign
	Member, Mentor:	

	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:	Примењено софтверско инжењерство		
Студент:		Број индекса:	
Степен и врста студија:			
Област:			
Ментор:			

НАСЛОВ ЗАВРШНОГ РАДА:

ТЕКСТ ЗАДАТКА:

Руководилац студијског програма:	Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора

Садржај

Садржај слика **1**

Садржај табела **2**

Списак коришћених скраћеница **3**

1. Увод **5**

1.1. Мотивација и проблем истраживања *5*

1.2. Циљеви рада *6*

1.3. Истраживачка питања *6*

1.4. Методологија истраживања *7*

1.5. Структура рада *7*

2. Теоријска основа **8**

2.1. Микросервисна архитектура *8*

2.1.1. Дефиниција и основни принципи 8

2.1.2. Предности микросервисне архитектуре 8

2.1.3. Изазови микросервисне архитектуре 9

2.2. Посматрачка способност дистрибуираних система *9*

2.2.1. Дефиниција и значај 9

2.2.2. Три стуба посматрачке способности 10

2.2.2.1. Логови 10

2.2.3. Корелација три стуба 12

2.2.4. Service Level Indicators/Objectives/Agreements 12

2.2.5. Изазови посматрачке способности у микросервисима 13

2.3. Kubernetes - оркестрација контејнера *13*

2.3.1. Увод у контернеризацију 13

2.3.2. Kubernetes архитектура 13

2.3.3. Основни *Kubernetes објекти* 14

2.3.4. Механизми за посматрање у Kubernetes-у 15

2.3.5. Хоризонтално аутоматско скалирање 15

2.3.6. Azure Kubernetes Service 16

2.4. ELK Stack – централизовано логовање *16*

2.4.1. *Elasticsearch* 16

2.4.2. Filebeat 17

2.4.3. Kibana 18

2.4.4. Index Lifecycle Management 18

2.5. Prometheus и Grafana - метрике и визуелизација *19*

2.5.1. Prometheus 19

2.5.2. Инструментација .NET апликација 20

2.5.3. Grafana 21

2.6. Дистрибуирано праћење захтева *21*

2.6.1. Jaeger 21

2.6.2. OpenTelemetry 21

2.6.3. Анализа трагова 22

3. Анализа постојећег система **23**

3.1. QuizHub платформа *23*

3.2. Микросервисна архитектура *23*

3.2.1. API *Gateway* (Ocelot) ... 25
3.2.2. User Service ... 26
3.2.3. Quiz Service ... 26
3.2.4. Result Service ... 27
3.2.5. Frontend апликација ... 28

3.3. Анализа недостатака постојећег система ... *29*
3.3.1. Логовање ... 29
3.3.2. Метрике ... 29
3.3.3. Праћење захтева ... 29

3.4. Циљана архитектура посматрачке способности ... *30*

4. Имплементација решења ... 30

4.1. Провизионисање инфраструктуре помоћу Terraform ... *30*
4.1.1. Концепт инфраструктуре као кода ... 30
4.1.2. Структура Terraform конфигурације ... 31
4.1.3. Azure ресурси ... 32
4.1.4. Kubernetes ресурси кроз Terraform ... 32
4.1.5. Параметризација конфигурације ... 33
4.1.6. Параметризација конфигурације ... 33

4.2. Контејнеризација и постављање апликације ... *33*
4.2.1. Контејнеризација сервиса ... 33
4.2.2. База података као StatefulSet ... 35
4.2.3. Конфигурација *Ingress* ресурса ... 35

4.3. Имплементација централизованог логовања (ELK Stack) ... *35*
4.3.1. Постављање Elasticsearch-а ... 36
4.3.2. Конфигурација *Filebeat* агента ... 36
4.3.3. Структурирано логовање у .NET апликацијама ... 37
4.3.4. Kibana за визуелизацију логова ... 37
4.3.5. Управљањем животним циклусом индекса ... 38

4.4. Имплементација прикупљања података – Prometheus ... *38*
4.4.1. Архитектура Prometheus-а у Kubernetes-у ... 38
4.4.2. Инструментација .NET апликација ... 39
4.4.3. Крајње тачке за проверу здравља ... 39

4.5. Имплементација визуелизације – Grafana ... *39*

4.6. Имплементација дистрибуираног праћења - Jaeger ... *41*
4.6.1. Архитектура *Jaeger*-а ... 41
4.6.2. OpenTelemetry интеграција ... 41
4.6.3. Прилагођени спанови и пропагација контекста ... 42

5. Тестирање и резултати ... 43

5.1. Методологија тестирања ... *43*
5.1.1. Тест окружење ... 43
5.1.2. Алати за тестирање ... 43

5.2. Функционално тестирање ... *43*
5.2.1. Валидација централизованог логовања ... 43
5.2.2. Валидација метрика ... 45
5.2.3. Валидација дистрибуираног праћења ... 45

5.3. Тестирање перформанси под оптерећењем ... *46*
5.3.1. Дефинисање сценарија оптерећења ... 46
5.3.2. Резултати тестирања ... 46

5.4. Анализа резултата ... *48*
5.4.1. Перформансе система ... 48

5.4.2. Ефективност посматрачке способности 48
5.5. Поређење са почетним стањем *48*
6. Закључак **49**
6.1. Резиме рада *49*
6.2. Остварени циљеви *50*
6.3. Одговори на истраживачка питања *50*
6.4. Ограничења истраживања *51*
6.5. Правци будућег рада *51*
6.6. Завршна реч *52*
7. Литература **53**
Биографија **55**

Садржај слика

- Слика 2.3.2.1. Kubernetes - архитектура кластера 15
- Слика 2.4.1.1. *Elasticsearch* API - пример индексирања документа 18
- Слика 2.4.1.2. *Elasticsearch* API - пример претраге докумената 18
- Слика 2.4.2.1. *Filebeat* - пример конфигурације 19
- Слика 2.5.1.1. *Prometheus* - архитектура система 20
- Слика 2.5.2.1. *prometheus-net* - Пример конфигурације кориснички дефинисаних метрика 21
- Слика 2.6.1.1. *Jaeger* - архитектура система 22
- Слика 3.2.1. *QuizHub* - архитектура система 25
- Слика 3.2.1.1. *Ocelot* - конфигурација рута унутар *Ocelot.json* датотеке ……..... 26
- Слика 3.4.1. *QuizHub* - Архитектура платформе са опсервабилношћу 31
- Слика 4.2.1.1. Docker - вишефазни *build* процес за .NET сервисе 35
- Слика 4.3.4.1. *Kibana* - интерфејс за претрагу логова 38
- Слика 4.5.2.1. *Grafana* - контролна табла за систем метрике 40
- Слика 5.2.1.1. *Kibana* - филтрирање логова по сервису и нивоу 44
- Слика 5.2.2.1. *Grafana* – Графички приказ метрика система у раду …… 45
- Слика 5.2.3.1. *Jaeger* – комплетан траг захтева кроз све сервисе …….. 46

Садржај табела

- Табела 4.1.2.1. Табела 4.1.2.1. Садржај Terraform директоријума и намена сваке од датотека ……31
- Табела 4.3.5.1. Конфигурација животног циклуса индекса …… 38
- Табела 4.4.2.1 Имплементација кориснички дефинисаних метрика ……..39
- Табела 4.6.1.1 Jaeger – Приступне тачке система ……40
- Табела 5.1.1.1. AKS – Конфигурација тестног окружења ……...42
- Табела 5.2.1.1. Резултати валидације централизованог логовања ……..43
- Табела 5.2.2.1. Резултати валидације метрика ……..44
- Табела 5.3.1.1. Дефиниције сценарија оптерећења ……..44
- Табела 5.3.2.1. Резултати тестирања под нормалним оптерећењем ……...45
- Табела 5.3.2.2. Резултати тестирања под високим оптерећењем ……45
- Табела 5.3.2.3. Резултати стрес теста ……46
- Табела 5.5.1. Поређење аспекта пре и после имплементације решења ……..47

Списак коришћених скраћеница

- **ACL**: *Access Control List*: Листа контроле приступа
- **ACR**: *Azure Container Registry*: Azure сервис за чување Docker слика
- **AKS**: *Azure Kubernetes Service*: Microsoft-ов управљани Kubernetes сервис
- **API**: *Application Programming Interface*: Програмски интерфејс који сервиси излажу
- **APM**: *Application Performance Monitoring*: Категорија алата за праћење перформанси апликација
- **CNI**: *Container Network Interface*: Мрежни додатак који омогућава директно IP адресирање pod-ова у Kubernetes-у
- **CORS**: *Cross-Origin Resource Sharing*: Политика која омогућава комуникацију између различитих домена или портова
- **CPU**: *Central Processing Unit*: Процесорски ресурс
- **DB**: *Database*: База података
- **DSL**: *Domain Specific Language*: Језик специфичан за домен, као што је *Elasticsearch Query DSL* или *PromQL*
- **ELK**: *Elasticsearch, Logstash, Kibana*: Скуп алата за прикупљање, складиштење и претраживање лог записа. У пројекту се користи *Elasticsearch + Filebeat + Kibana*
- **GB**: *Gigabyte*: Гигабајт, јединица меморије (1024 мегабајта)
- **HCL**: *HashiCorp Configuration Language*: Декларативни језик за дефинисање инфраструктурних ресурса у Terraform-у
- **HTTP**: *Hypertext Transfer Protocol*: Протокол за пренос података преко веба
- **IaC**: *Infrastructure as Code*: Инфраструктура као код - пракса управљања инфраструктуром кроз машински читљиве конфигурационе датотеке
- **ILM**: *Index Lifecycle Management*: Управљање животним циклусом индекса у Elasticsearch-у (hot, warm, cold, delete фазе)
- **JSON**: *JavaScript Object Notation*: Формат за размену структурираних података
- **JWT**: *JSON Web Token*: Стандард за представљање и верификацију токена за аутентификацију
- **K8s** / **Kubernetes**: Систем за оркестрацију контејнера који аутоматизује постављање, скалирање и управљање контејнеризованим апликацијама
- **KQL**: *Kibana Query Language*: Синтакса за претраге и филтрирање логова у Kibana интерфејсу
- **LTS**: *Long Term Support*: Верзија са дугорочном подршком и продуженим периодом одржавања
- **MB**: *Megabyte*: Мегабајт, јединица меморије (1024 килобајта)

- **PromQL**: *Prometheus Query Language*: Језик за упите и анализу метрика у Prometheus систему
- **PV**: *PersistentVolume*: Стварни перзистентни диск ресурс у Kubernetes-у који обезбеђује трајно складиште
- **PVC**: *PersistentVolumeClaim*: Захтев у Kubernetes-у за трајни складишни простор који аплицикација користи
- **RAM**: *Random Access Memory*: Меморија са насумичним приступом
- **RBAC**: *Role-Based Access Control*: Контрола приступа заснована на улогама корисника
- **RED**: *Rate, Errors, Duration*: Методологија за дефинисање метрика сервиса фокусирана на брзину захтева, стопу грешака и трајање
- **REST**: *Representational State Transfer*: Архитектурални стил за дизајн веб сервиса заснован на HTTP протоколу
- **RG**: *Resource Group*: Логичка група ресурса у Azure-у која омогућава управљање животним циклусом на једном месту
- **SDK**: *Software Development Kit*: Комплет алата, библиотека и документације за развој софтвера
- **SPA**: *Single Page Application*: Апликација са једном страницом која динамички ажурира садржај без поновног учитавања
- **SQL**: *Structured Query Language*: Језик за управљање и упите релационих база података
- **StatefulSet**: Kubernetes ресурс за управљање stateful апликацијама које захтевају стабилне идентитете и перзистентно складиште
- **TLS**: *Transport Layer Security*: Протокол за шифровану и безбедну комуникацију преко мреже
- **USE**: *Utilization, Saturation, Errors*: Методологија за дефинисање метрика ресурса фокусирана на искоришћеност, засићеност и грешке
- **vCPU**: *Virtual Central Processing Unit*: Виртуелни процесор додељен виртуелној машини или контејнеру
- **VNet**: *Virtual Network* (Azure): Виртуелна мрежа у Azure-у која обезбеђује изолован мрежни простор
- **W3C**: *World Wide Web Consortium*: Међународна заједница која развија отворене стандарде за веб, укључујући *Trace Context* стандард
- **YAML**: *Yet Another Markup Language*: Формат који се често користи за Kubernetes манифест спецификације и конфигурационе датотеке

1. Увод

1.1. Мотивација и проблем истраживања

Савремени софтверски системи све чешће се развијају применом микросервисне архитектуре, која омогућава декомпозицију комплексних апликација на мање, независне сервисе који комуницирају путем мрежних протокола. Овај архитектурални приступ доноси бројне предности као што су независно скалирање појединачних компоненти, флексибилност у избору технологија и олакшано одржавање система [1]. Међутим, дистрибуирана природа микросервиса уводи значајне изазове у домену праћења и дијагностике система.

У монолитним апликацијама, праћење понашања система је релативно једноставно - сви логови се налазе на једном месту, а позиви функција се могу пратити кроз стек трагове унутар једног процеса. Супротно томе, у микросервисном окружењу један кориснички захтев може проћи кроз десетине различитих сервиса, генеришући логове на више локација и укључујући асинхрону комуникацију између компоненти. Ова комплексност отежава идентификацију узрока проблема, анализу перформанси и разумевање целокупног понашања система [2].

Концепт посматрачке способности (енгл. *observability*) представља способност разумевања унутрашњег стања система на основу његових спољашњих излаза [2]. За разлику од традиционалног мониторинга, који се фокусира на праћење унапред дефинисаних метрика и познатих проблема, посматрачка способност омогућава инжењерима да истражују и дијагностикују непознате проблеме постављањем произвољних питања о стању система. Три основна стуба посматрачке способности [2] су:

1. **Логови** (енгл. *logs*) - текстуални записи догађаја са временским печатом који документују шта се десило у систему
2. **Метрике** (енгл. *metrics*) - нумеричке вредности праћене током времена које квантификују понашање система
3. **Трагови** (енгл. *traces*) - записи о путањи захтева кроз дистрибуирани систем

Платформе за оркестрацију контејнера, посебно ***Kubernetes***, постале су де факто стандард за покретање микросервисних апликација у продукционим окружењима [6]. Kubernetes пружа богат екосистем алата за посматрачку способност, укључујући подршку за прикупљање метрика, централизовано логовање и интеграцију са системима за дистрибуирано праћење [36]. Ова интеграција омогућава имплементацију свеобухватне посматрачке способности која може значајно унапредити оперативну ефикасност микросервисних система.

Предмет овог завршног рада је имплементација комплетног система посматрачке способности за *QuizHub* платформу - микросервисну апликацију за креирање и решавање квизова. *QuizHub* је првобитно развијен коришћењем ***Docker Compose*** оркестрације са минималном посматрачком способношћу, што је представљало ограничење за ефикасно праћење и дијагностику система у сценаријима са већим оптерећењем.

1.2. Циљеви рада

Примарни циљ овог завршног рада је имплементација свеобухватне посматрачке способности за микросервисну архитектуру коришћењем савремених алата и технологија отвореног кода. Конкретни циљеви укључују:

- **Миграција система на Kubernetes платформу**: Миграција постојеће QuizHub апликације са *Docker-Compose* на ***Azure Kubernetes Service*** **(AKS)** ради искоришћавања напредних могућности Kubernetes-а за посматрачку способност, скалирање и управљање ресурсима.

- **Имплементација централизованог логовања**: Успостављање **ELK (*Elasticsearch*, *Logstash*/*Filebeat*, *Kibana*)** стека за агрегацију логова свих микросервиса на једном месту, са могућношћу претраге, филтрирања и визуелизације логова.

- **Успостављање система за прикупљање метрика**: Конфигурација ***Prometheus*** система за прикупљање метрика о перформансама апликације и инфраструктуре, укључујући HTTP захтеве, време одзива, искоришћеност ресурса и грешке.

- **Имплементација дистрибуираног праћења захтева**: Интеграција ***Jaeger*** система за праћење путање захтева кроз све микросервисе, омогућавајући анализу латенције и идентификацију уских грла у систему.

- **Креирање контролних табли за визуелизацију**: Развој интерактивних ***Grafana*** контролних табли (енгл. *dashboards*) које пружају преглед здравља система, перформанси појединачних сервиса и трендова коришћења.

- **Валидација решења кроз тестирање под оптерећењем**: Спровођење тестова оптерећења ради верификације да имплементирани систем посматрачке способности омогућава ефикасну дијагностику проблема и анализу перформанси.

1.3. Истраживачка питања

Овај рад тежи да одговори на следећа истраживачка питања:

- Како имплементирати свеобухватну посматрачку способност у микросервисној архитектури користећи алате отвореног кода?

- Какве користи пружа Kubernetes оркестрација у контексту посматрања и дијагностике микросервисних система?

- Како ELK стек омогућава ефикасну централизацију, претрагу и анализу логова у дистрибуираном окружењу?

- На који начин комбинација метрика, логова и трагова унапређује способност идентификације и решавања проблема у продукционом окружењу?

- Колики је утицај имплементиране посматрачке способности на перформансе система?

1.4. Методологија истраживања

За потребе овог истраживања примењена је комбинација методолошких приступа:

- **Студија случаја (енгл. *case study*)**: QuizHub платформа коришћена је као студија случаја за имплементацију и евалуацију предложеног решења. Ова методологија омогућава детаљну анализу у реалистичном контексту и практичну верификацију теоријских концепата.

- **Експериментални приступ**: Практична имплементација система посматрачке способности праћена је систематским тестирањем и мерењем перформанси. Експерименти укључују тестове оптерећења, мерење латенције и анализу употребе ресурса.

- **Анализа литературе**: Преглед релевантне академске литературе и индустријских пракси у области посматрачке способности микросервиса, са циљем усвајања најбољих пракси и валидације предложеног приступа.

1.5. Структура рада

Рад је организован у шест поглавља:

- **Поглавље 2: Теоријска основа** - Представља теоријску позадину микросервисне архитектуре, концепта посматрачке способности, Kubernetes платформе и технологија коришћених за имплементацију (ELK, Prometheus, Grafana, Jaeger).

- **Поглавље 3: Анализа постојећег система** - Описује архитектуру QuizHub платформе, анализира тренутно стање посматрачке способности и идентификује недостатке које је потребно адресирати.

- **Поглавље 4: Имплементација решења** - Детаљно документује процес миграције на Kubernetes и имплементацију свих компоненти система посматрачке способности, укључујући конфигурационе датотеке и архитектуралне одлуке.

- **Поглавље 5: Тестирање и евалуација** - Приказује методологију тестирања, резултате тестова оптерећења и анализу ефикасности имплементираног решења.

- **Поглавље 6: Закључак** - Сумира остварене резултате, дискутује доприносе рада и предлаже правце будућег развоја.

2. Теоријска основа

Ово поглавље представља теоријску основу неопходну за разумевање концепата и технологија примењених у имплементацији система посматрачке способности. Започиње се прегледом микросервисне архитектуре, затим се детаљно анализира концепт посматрачке способности, а потом следи опис технологија коришћених у имплементацији.

2.1. Микросервисна архитектура

2.1.1. Дефиниција и основни принципи

Микросервисна архитектура представља архитектурални стил који структурира апликацију као колекцију слабо спрегнутих сервиса, од којих сваки имплементира специфичну пословну функционалност [3]. Кључни принципи микросервисне архитектуре укључују:

- **Декомпозицију по пословним способностима:** Сваки микросервис је организован око специфичне пословне способности (енгл. *business capability*) и садржи све потребне компоненте за испуњење те одговорности - од корисничког интерфејса до базе података. Овај принцип осигурава да промене у једној пословној области захтевају модификације само једног сервиса.

- **Независну могућност постављања:** Микросервиси се могу независно постављати (енгл. *deploy*) без утицаја на остале сервисе у систему. Ова карактеристика омогућава честе испоруке нових верзија и смањује ризик од регресија при ажурирањима.

- **Децентрализовано управљање подацима:** Сваки микросервис управља својом базом података (енгл. *database per service pattern*), што осигурава слабу спрегнутост и омогућава избор оптималне технологије складиштења за сваки сервис. Ова децентрализација захтева пажљиво управљање конзистентношћу података између сервиса.

- **Комуникацију путем API-ја:** Микросервиси комуницирају искључиво путем добро дефинисаних интерфејса, типично REST API-ја или асинхроних порука. Интерна имплементација сервиса је енкапсулирана и непрозирна за друге сервисе.

2.1.2. Предности микросервисне архитектуре

Микросервисна архитектура пружа бројне предности у поређењу са монолитним приступом:

- **Технолошка разноврсност:** Тимови могу бирати оптималне технологије за сваки сервис независно од избора у другим деловима система [4]. На пример, сервис за обраду слика може користити *Python* због богатих библиотека за машинско учење, док сервис за трансакције може користити *Java* због зрелих оквира за управљање трансакцијама.

- **Скалабилност:** Појединачни сервиси се могу скалирати независно према потребама оптерећења [4]. У монолитној апликацији, целокупан систем мора бити скалиран чак и ако само једна компонента захтева додатне ресурсе.

- **Отпорност на грешке:** Изолација сервиса омогућава да квар једног сервиса не узрокује пад целокупног система [4]. Примена образаца као што су *Circuit Breaker* и *Bulkhead* додатно унапређује отпорност система [5].

- **Организациона усклађеност:** Микросервисна архитектура омогућава организовање тимова око појединачних сервиса, што одговара Конвејевом закону о усклађености архитектуре система са комуникационим структурама организације [4].

2.1.3. Изазови микросервисне архитектуре

Упркос бројним предностима, микросервисна архитектура уводи значајне изазове:

- **Комплексност дистрибуираног система** Дистрибуирани системи се суочавају са проблемима мрежне комуникације, парцијалних грешака и евентуалне конзистентности [4]. Инжењери морају управљати латенцијом мреже, прекидима комуникације и синхронизацијом између сервиса.

- **Оперативна комплексност** Управљање десетинама или стотинама сервиса захтева софистицирану инфраструктуру за постављање, мониторинг и дијагностику [4]. Без одговарајућих алата, оперативни трошкови могу надмашити користи од декомпозиције система.

- **Тестирање интеграција** Верификација исправног функционисања целокупног система захтева тестирање интеракција између сервиса, што је значајно комплексније од тестирања монолитне апликације [4].

- **Посматрачка способност** Идентификација узрока проблема у дистрибуираном систему захтева напредне механизме за прикупљање логова, метрика и трагова из свих сервиса [5]. Ово представља централну тему овог завршног рада.

2.2. Посматрачка способност дистрибуираних система

2.2.1. Дефиниција и значај

Термин "посматрачка способност" (енгл. *observability*) потиче из теорије управљања системима, где је дефинисан као мера колико добро се унутрашња стања система могу закључити из познавања спољашњих излаза [4]. У контексту софтверских система, Sridharan дефинише посматрачку способност као "карактеристику система која омогућава разумевање његовог унутрашњег стања испитивањем његових спољашњих излаза" [6].

Кључна разлика између мониторинга и посматрачке способности лежи у приступу дијагностици:

- **Мониторинг** се фокусира на праћење унапред познатих метрика и упозоравање када вредности пређу дефинисане прагове. Ово је **реактиван приступ** који захтева претходно знање о потенцијалним проблемима [6].

- **Посматрачка способност** омогућава истраживачки приступ, где инжењери могу постављати произвољна питања о стању система и добијати одговоре чак и за непредвиђене сценарије Ово је **проактиван приступ** који подржава дијагностику непознатих проблема [6].

2.2.2. Три стуба посматрачке способности

2.2.2.1. Логови

Логови представљају дискретне записе догађаја који се дешавају у систему, обично са временским печатом и структурираним или неструктурираним садржајем. Они пружају детаљан увид у специфичне догађаје и контекстуалне информације неопходне за дијагностику проблема.

Типови логова:

- **Апликативни логови** - записи догађаја унутар пословне логике апликације
- **Системски логови** - записи о функционисању оперативног система и инфраструктуре
- **Логови приступа** - записи о HTTP захтевима и одговорима
- **Логови грешака** - детаљни записи о изузецима и грешкама

Нивои логовања: Индустријски стандард дефинише хијерархију нивоа логовања [7]:

- **TRACE** - најдетаљнији записи, типично за развојно окружење
- **DEBUG** - детаљне информације за дијагностику проблема
- **INFO** - информативни записи о нормалном функционисању
- **WARN** - упозорења о потенцијалним проблемима
- **ERROR** - грешке које захтевају пажњу
- **FATAL** - критичне грешке које узрокују престанак рада

Структурирано логовање: Модерне праксе препоручују структурирано логовање у JSON формату, што олакшава аутоматску обраду и претрагу логова [8]:

```
{
  "timestamp": "2025-11-19T14:32:15.123Z",
  "level": "INFO",
  "service": "user-service",
  "traceId": "abc123def456",
  "message": "User login successful",
  "userId": "user_789",
  "duration_ms": 45
}
```

Метрике представљају нумеричке вредности праћене током времена које квантификују понашање система [31]. За разлику од логова који су дискретни догађаји, метрике су агрегиране вредности које омогућавају **анализу трендова** и **детекцију аномалија.**

Типови метрика:

- ***Counter*** - монотоно растућа вредност која представља кумулативни број догађаја (нпр. укупан број HTTP захтева).
- ***Gauge*** - вредност која може расти и падати, представљајући тренутно стање (нпр. тренутна меморија у употреби).
- ***Histogram*** - дистрибуција вредности у унапред дефинисаним интервалима (нпр. дистрибуција времена одзива).
- ***Summary*** - статистички сажетак вредности укључујући перцентиле (нпр. p50, p95, p99 времена одзива).

Методологије за дефинисање метрика [9]:

1. **RED** метод - фокусиран на **метрике сервиса**:
   - ***Rate*** - брзина обраде захтева (захтева по секунди)
   - ***Errors*** - стопа грешака
   - ***Duration*** - трајање обраде захтева
2. **USE** метод - фокусиран на **метрике ресурса**:
   - ***Utilization*** - проценат искоришћености ресурса
   - ***Saturation*** - степен засићености (ред чекања)
   - ***Errors*** - број грешака ресурса

Дистрибуирано праћење (енгл. *distributed tracing*) омогућава праћење путање једног захтева кроз све сервисе у дистрибуираном систему [10]. Ова техника је есенцијална за микросервисне архитектуре где један кориснички захтев може укључити десетине интерних позива између сервиса.

Кључни концепти:

- ***Trace*** - комплетан запис о путањи захтева кроз систем, укључујући све сервисе који су учествовали у обради.
- ***Span*** - јединица рада у оквиру трага, представљајући операцију у једном сервису. Сваки спан садржи време почетка, трајање, име операције и опционе метаподатке.
- ***Context propagation*** - механизам за пропагацију контекста трага (trace ID, span ID) кроз границе сервиса, типично путем HTTP заглавља.

W3C *Trace Context* стандард: *W3C (*Веб3 Конзорцијум*)* је стандардизовао формат пропагације контекста кроз *traceparent* и *tracestate* HTTP заглавља [11]:

```
traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01
```

Дистрибуирано праћење је революционализовано 2010. године објављивањем Google-овог рада о **Dapper** систему [12], који је послужио као инспирација за бројне системе отвореног кода укључујући **Jaeger** и **Zipkin**.

2.2.3. Корелација три стуба

Стварна вредност посматрачке способности постиже се корелацијом логова, метрика и трагова [6]. На пример:

1. Метрике указују на повећану латенцију сервиса
2. Трагови идентификују који специфичан позив унутар сервиса узрокује успорење
3. Логови пружају детаљан контекст о узроку (нпр. споре SQL упите)

Корелација се постиже коришћењем јединственог идентификатора (*trace* ID или *correlation* ID) који повезује све типове података за исти захтев:

```
Метрика: http_request_duration_seconds{service="user-service"} = 2.5s
↓
Траг:    trace_id="abc123" → UserService (1.8s) → DatabaseCall (1.5s)
↓
Лог:    {
         "traceId":"abc123",
         "message":"Slow query: SELECT * FROM users WHERE…"
        }
```

2.2.4. Service Level Indicators/Objectives/Agreements

Посматрачка способност је фундаментална за дефинисање и праћење нивоа услуге [13]:

- SLI (*Service Level Indicator*) - квантитативна мера аспекта услуге, нпр. проценат захтева обрађених за мање од 200ms.
- SLO (*Service Level Objective*) - циљна вредност за SLI, нпр. 99% захтева треба да буде обрађено за мање од 200ms.
- SLA (*Service Level Agreement*) - формални уговор са дефинисаним последицама ако SLO није испуњен.

Имплементација система посматрачке способности омогућава прецизно мерење SLI-јева и праћење испуњености SLO-ова, што је критично за управљање квалитетом услуге у продукционом окружењу.

2.2.5. Изазови посматрачке способности у микросервисима

Имплементација ефикасне посматрачке способности у микросервисном окружењу суочава се са специфичним изазовима:

- **Обим података:** Велики број сервиса генерише огромне количине логова и метрика које морају бити ефикасно прикупљене, складиштене и претраживе [14].
- **Дистрибуирани контекст:** Праћење захтева кроз границе сервиса захтева механизме за пропагацију контекста и стандардизоване формате [15].
- **Хетерогеност:** Различити сервиси могу користити различите технологије и формате логовања, што отежава централизацију и корелацију [16].
- **Оверхед:** Инструментација за посматрачку способност уноси одређени оверхед који мора бити минимизован да не би утицао на перформансе система [15].

2.3. Kubernetes - оркестрација контејнера

2.3.1. Увод у контернеризацију

Контејнеризација представља технологију виртуализације на нивоу оперативног система која омогућава паковање апликације заједно са свим њеним зависностима у изоловану јединицу - контејнер [17]. **Docker** је 2013. године популаризовао контејнеризацију увођењем једноставног формата за дефинисање контејнера и алата за њихово управљање [18].

Кључне предности контејнеризације укључују:

- **Конзистентност окружења** - идентично понашање на развојној машини и у продукцији
- **Изолација** - независност апликација од хост система и других апликација
- **Ефикасност** - мањи оверхед у поређењу са виртуелним машинама [19]
- **Преносивост** - могућност покретања на различитим платформама [19]

Међутим, управљање великим бројем контејнера у продукционом окружењу захтева систем за оркестрацију који аутоматизује постављање, скалирање и управљање животним циклусом контејнера.

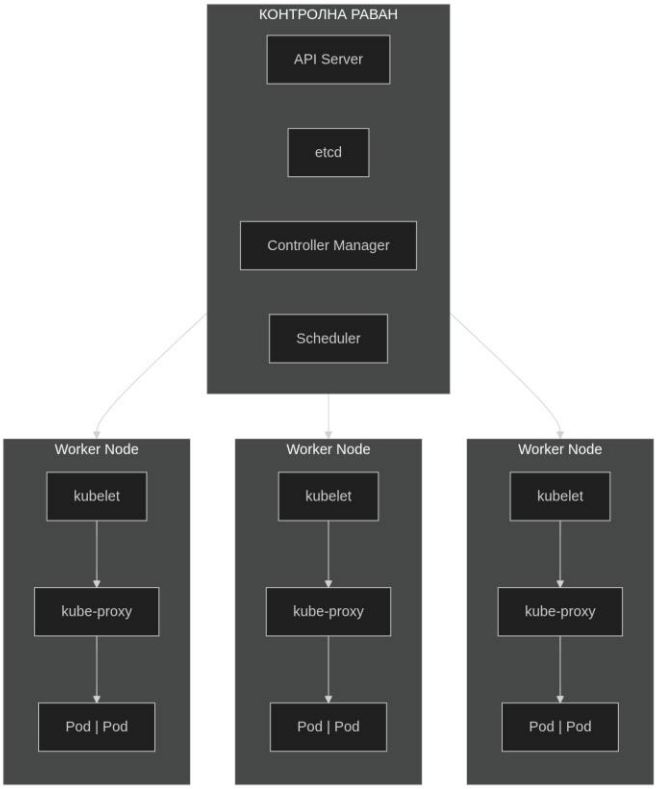
2.3.2. Kubernetes архитектура

Kubernetes (K8s) је систем отвореног кода за аутоматизацију постављања, скалирања и управљања контејнеризованим апликацијама [20]. Развијен је у Google-у на основу искуства са *Borg* системом који је управљао милионима контејнера [21].

Архитектура кластера:

Kubernetes кластер се састоји од **контролне равни** (енгл. *control plane*) и **радничких чворова** (енгл. *worker nodes*) [22]:

1. **Контролна раван** садржи компоненте за управљање кластером:
   - ***kube-apiserver*** - REST API за све операције над кластером
   - ***etcd*** - дистрибуирано складиште за стање кластера
   - ***kube-scheduler*** - распоређује *Pod*-ове на чворове
   - ***kube-controller-manager*** - управља контролерима (*ReplicaSet*, *Deployment*, итд.)
2. **Радни чворови** извршавају контејнеризоване апликације:
   - **kubelet** - агент који управља *Pod*-овима на чвору
   - **kube-proxy** - одржава мрежна правила за комуникацију
   - **извршно окружење контејнера** (енгл. *container runtime*) - извршава контејнере (*containerd*, *CRI-O*)



Слика 2.3.2.1. Kubernetes – архитектура кластера

2.3.3. Основни *Kubernetes објекти*

***Pod* - Капсула** је најмања јединица за извршавање у *Kubernetes*-у и може садржати један или више контејнера који деле мрежни простор и складиште [23]. Контејнери унутар *pod*-а комуницирају путем *localhost* адресе - интерном комуникацијом.

Deployment - *Deployment* је контролер који управља животним циклусом *pod*-ова, обезбеђујући декларативно ажурирање и одржавање жељеног броја реплика [23].

Service - *Service* пружа стабилну мрежну апстракцију за приступ групи p*od*-ова, омогућавајући сервисно откривање (енгл. *service discovery*) путем DNS-а [23].

ConfigMap и ***Secret*** - *ConfigMap* складишти неосетљиве конфигурационе податке, док *Secret* чува осетљиве информације као што су лозинке и API кључеви [23].

Namespace - *Namespace* омогућава логичку изолацију ресурса унутар кластера, подржавајући сценарије вишеструких корисника (енгл. *multi-tenant*) и организацију по окружењима [23].

2.3.4. Механизми за посматрање у Kubernetes-у

Kubernetes пружа уграђене механизме који подржавају посматрачку способност:

Здравље Pod-ова (*Health Probes*) [24]:

- ***Liveness Probe*** - проверава да ли је контејнер жив; ако не, *Kubernetes* рестартује *Pod*
- ***Readiness Probe*** - проверава да ли је контејнер спреман да прима саобраћај
- ***Startup Probe*** - намењен апликацијама са дугим временом покретања

```
livenessProbe:
  httpGet:
    path: /health
    port: 80
  initialDelaySeconds: 30
  periodSeconds: 10

readinessProbe:
  httpGet:
    path: /ready
    port: 80
  initialDelaySeconds: 10
  periodSeconds: 5
```

Metrics Server - *Metrics Server* прикупља метрике о искоришћености CPU-а и меморије од свих Pod-ова, омогућавајући хоризонтално аутоматско скалирање [25].

Kubernetes Events - Систем догађаја бележи важне промене стања објеката, пружајући основну посматрачку способност као уграђену функционалност [26].

Labels и ***Annotations*** - *Labels* омогућавају организацију и селекцију ресурса, док *annotations* пружају мета податке које алати за посматрање користе за обогаћивање података [26].

2.3.5. Хоризонтално аутоматско скалирање

Horizontal Pod Autoscaler (*HPA*) аутоматски скалира број *pod*-ова на основу метрика о искоришћености ресурса или прилагођених метрика [27]:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
```

```
metadata:
  name: user-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: user-service
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

2.3.6. Azure Kubernetes Service

AKS (*Azure Kubernetes Service*) је управљана Kubernetes услуга у ***Microsoft Azure*** облаку која апстрахује комплексност управљања контролном равни [28].

AKS пружа:

- Аутоматско ажурирање *Kubernetes* верзија,
- *Azure Monitor* за метрике и логове,
- ***Azure Container Registry*** за складиштење слика контејнера,
- Виртуелне мреже и мрежне политике [29]

2.4. ELK *Stack* – централизовано логовање

ELK *Stack* представља комбинацију три пројекта отвореног кода: ***Elasticsearch***, ***Logstash*** и ***Kibana***, коју је развила компанија ***Elastic*** [30]. Ова комбинација се широко користи за централизовано прикупљање, складиштење, претрагу и визуелизацију логова.

2.4.1. *Elasticsearch*

Elasticsearch је дистрибуирана претраживачка и аналитичка машина базирана на *Apache Lucene* библиотеци [31]. Његове кључне карактеристике укључују:

Архитектура:

- **Кластер** - скуп чворова који заједно складиште податке
- **Индекс** - колекција докумената сличне структуре
- **Документ** - основна јединица информације у JSON формату
- ***Shard*** - подела индекса за хоризонтално скалирање
- ***Replica*** - копија *shard*-а за високу доступност

***RESTful API**:* Elasticsearch чини доступним све операције кроз REST-ful API:

```
PUT /logs-quizhub-2025.11.19/_doc/1

{
  "timestamp": "2025-11-19T14:00:00Z",
  "service": "user-service",
  "level": "INFO",
  "message": "User login successful"
}
```

Слика 2.4.1.1. *Elasticsearch API* – пример индексирања документа

```
GET /logs-quizhub-*/_search

{
  "query": {
    "bool": {
      "must": [
        { "match": { "service": "user-service" } },
        { "match": { "level": "ERROR" } }
      ]
    }
  }
}
```

Слика 2.4.1.2. *Elasticsearch API* – пример претраге докумената

Query DSL: *Elasticsearch Query DSL* је језик који омогућава комплексне упите укључујући пуно-текстуалну претрагу, филтрирање, агрегације и географске упите [32].

2.4.2. Filebeat

Filebeat је лагани агент за прослеђивање логова који је оптимизован за минималну потрошњу ресурса [33]. У *Kubernetes* окружењу, *Filebeat* се типично поставља као *DaemonSet* тако да сваки чвор има један примерак који прикупља логове контејнера.

Кључне функционалности:

- ***Autodiscover*** - аутоматско откривање контејнера и прилагођавање конфигурације
- ***Kubernetes metadata enrichment*** - додавање мета података о *pod*-у, *namespace*-у, *labels*
- ***Multiline processing*** - спајање вишелинијских логова (нпр. *stack trace*)
- ***JSON* парсирање**- парсирање структурираних логова

```
 filebeat.inputs:
- type: container
  paths:
    - /var/log/containers/*.log
  processors:
    - add_kubernetes_metadata:
```

```
        host: ${NODE_NAME}
    - decode_json_fields:
        fields: ["message"]
        target: ""

output.elasticsearch:
  hosts: ['elasticsearch:9200']
  index: "logs-quizhub-%{[kubernetes.labels.app]}-%{+yyyy.MM.dd}"
```

Слика 2.4.2.1. *Filebeat* – пример конфигурације

2.4.3. Kibana

Kibana је платформа за визуелизацију која омогућава истраживање и анализу података складиштених у *Elasticsearch*-у [34].

Главне функционалности:

- ***Discover*** - интерактивна претрага и филтрирање логова у реалном времену
- ***Visualizations*** - креирање графикона, табела, метрика и других визуелизација
- ***Dashboards*** - композитни прикази који комбинују више визуелизација
- ***Alerting*** - дефинисање правила за аутоматске нотификације
- ***Canvas*** - напредне инфографике и презентације

2.4.4. Index Lifecycle Management

Index Lifecycle Management (ILM) у оквиру *Elasticsearch* омогућава аутоматизовано управљање животним циклусом индекса дефинисањем полиса које контролишу како се индекси понашају током времена, кроз унапред дефинисане фазе: *hot*, *warm*, *cold* и *delete* [35]. Овај механизам је кључан за оптимизацију трошкова складиштења**,** перформансе претраге **и** дефинисање политика ретенције лог података**.**

ILM полиса представља декларативну конфигурацију која дефинише:

- фазе кроз које индекс пролази,
- услове за прелазак између фаза (нпр. старост индекса, величина, број докумената),
- акције које се извршавају у свакој фази (нпр. *rollover*, *shrink*, *force merge*, *delete*).

Полисе се типично примењују на ***time-series*** **индексе** (временски континуално сортирани индекси нпр. логови), што омогућава аутоматско управљање великим количинама података без ручне интервенције.

У ***hot*** **фази**, индекс је активно доступан за упис и претрагу и смештен је на чворовима са високим перформансама. У овој фази је могуће дефинисати ***rollover*** **механизам**, којим се аутоматски креира нови индекс када се достигну задати услови као што су величина, број докумената или старост индекса, чиме се одржава стабилност перформанси.

***Warm* фаза** је намењена индексима који се више не ажурирају активно, али се и даље повремено претражују. Индекси се премештају на мање ресурсно интензивне чворове и могу се додатно оптимизовати ради смањења потрошње системских ресурса.

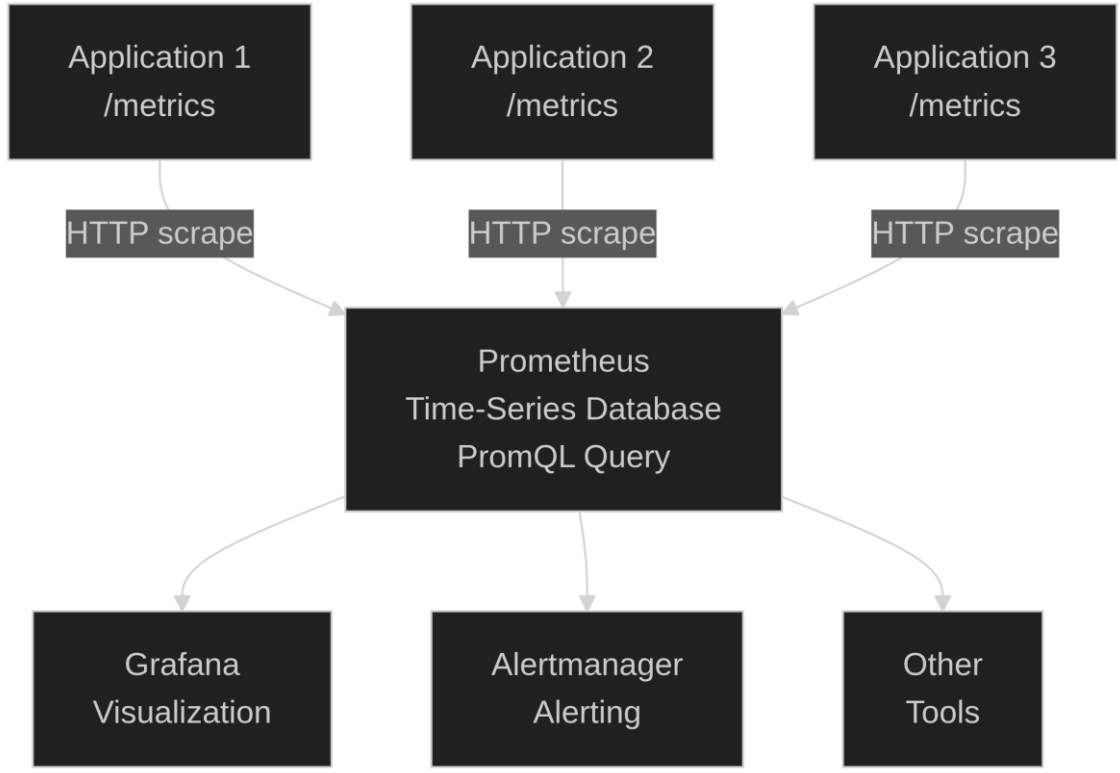
У ***cold* фази**, индекси садрже архивске податке који се ретко користе и складиште се на инфраструктури са минималним ресурсима. Претраге су спорије, али је обезбеђено дугорочно задржавање података уз минималне трошкове, након чега се у ***delete* фази** индекси трајно уклањају у складу са дефинисаном политиком ретенције.

2.5. Prometheus и Grafana - метрике и визуелизација

2.5.1. Prometheus

Prometheus је систем за мониторинг и упозоравање отвореног кода развијен у *SoundCloud*-у, сада под окриљем *Cloud Native Computing Foundation [36]*.

Архитектура*:*



Слика 2.5.1.1. *Prometheus* – архитектура система

***Pull* модел**: *Prometheus* активно "скрејпује" метрике са циљева у конфигурисаним интервалима, за разлику од *push* модела где апликације шаљу метрике [37]. Овај приступ омогућава:

- Централизовану конфигурацију
- Лакше детектовање недоступних сервиса
- Једноставнију имплементацију на страни апликације

***Service Discovery*:** У Kubernetes окружењу, *Prometheus* користи Kubernetes API за аутоматско откривање циљева [38]:

```
scrape_configs:
  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
        regex: true
```

PromQL (*Prometheus Query Language*): омогућава моћне упите над временским серијама [39]:

- Стопа HTTP захтева по сервису у последње 5 минута:

```
rate(http_requests_total{job="user-service"}[5m])
```

- Претрага 95-тог перцентила латенције:

```
histogram_quantile(0.95,
   sum(rate(http_request_duration_seconds_bucket[5m])) by (le, service)
)
```

- Стопа грешака:

```
sum(rate(http_requests_total{status=~"5.."}[5m]))
   / sum(rate(http_requests_total[5m])) * 100
```

2.5.2. Инструментација .NET апликација

За прикупљање метрика из .NET апликација користи се ***prometheus-net*** библиотека [40]:

```
// Program.cs
app.UseMetricServer();  // Излаже /metrics интерфејс

// Дефинисање метрика
private static readonly Counter RequestCounter = Metrics
    .CreateCounter("http_requests_total", "Total HTTP requests",
        new CounterConfiguration {
            LabelNames = new[] { "method", "endpoint", "status" }
        });

private static readonly Histogram RequestDuration = Metrics
    .CreateHistogram("http_request_duration_seconds", "Request duration",
        new HistogramConfiguration {
            LabelNames = new[] { "method", "endpoint" },
            Buckets = new[] { 0.01, 0.05, 0.1, 0.5, 1, 5 }
        });
```

Слика 2.5.2.1. *prometheus-net*– Пример конфигурације кориснички дефинисаних метрика

2.5.3. Grafana

Grafana је платформа отвореног кода за аналитику и интерактивну визуелизацију [41]. Подржава бројне изворе података укључујући *Prometheus*, *Elasticsearch*, *InfluxDB* и друге [42].

Концепти контролних табли:

- ***Panel*** - појединачна визуелизација (график, табела, метар)
- ***Row*** - хоризонтална група панела
- ***Dashboard*** - колекција панела и редова
- ***Variable*** - динамичка вредност за филтрирање (нпр. избор сервиса)

2.6. Дистрибуирано праћење захтева

2.6.1. Jaeger

Jaeger је систем за дистрибуирано праћење отвореног кода развијен у *Uber*-у, инспирисан *Google*-овим *Dapper*-ом и *OpenZipkin*-ом [43]. Jaeger је дизајниран за праћење захтева у микросервисним архитектурама, са циљем откривања латенција, уских грла и грешака у дистрибуираним системима.

Архитектура Jaeger-а је модуларна и састоји се од клијентских библиотека које генеришу спанове, агента који их прима и прослеђује, колектора који врши валидацију и обраду трагова, као и позадинског система за складиштење као што су *Elasticsearch* или *Cassandra*:

Слика 2.6.1.1. *Jaeger* – архитектура система

- ***Jaeger Client*** - библиотека интегрисана у апликацију
- ***Jaeger Agent*** – позадински процес (*daemon*) који прима спанове од клијената
- ***Jaeger Collector*** - валидира, обрађује и складишти трагове
- Позадински систем за складиштење (енгл. ***Storage Backend***) - *Elasticsearch*, *Cassandra* или *in-memory*
- ***Jaeger Query*** - кориснички интерфејс за претрагу и визуелизацију трагова, што омогућава ефикасну анализу понашања система у реалном времену.

2.6.2. OpenTelemetry

OpenTelemetry је неутралан стандард независан од произвођача (енгл. ***vendor-neutral***) који обједињује функционалности пројеката *OpenTracing* и *OpenCensus* [44]. Циљ *OpenTelemetry*-ја је да обезбеди јединствен и конзистентан начин прикупљања трагова, метрика и логова, независно од конкретног *backend* система.

OpenTelemetry дефинише заједничке API-је и SDK-ове за велики број програмских језика и подржава како ручну, тако и аутоматску инструментацију популарних библиотека и радних оквира. Прикупљени подаци се затим могу извозити ка различитим системима за анализу и визуелизацију, укључујући *Jaeger* и *Prometheus*.

2.6.3. Анализа трагова

Jaeger UI омогућава детаљну анализу дистрибуираних трагова кроз више комплементарних приказа [45].

Хронолошки приказ (*timeline view*) омогућава увид у редослед и трајање спанова, док **граф зависности сервиса** (*service dependency graph*) визуализује комуникацију између микросервиса. Поред тога, Jaeger подржава поређење трагова и приказ агрегираних статистика, што је посебно корисно за идентификацију перформансних деградација и регресија током времена.

Ови механизми чине дистрибуирано праћење кључном компонентом modernog приступа опсервабилности у *cloud-native* системима.

3. Анализа постојећег система

3.1. QuizHub платформа

QuizHub представља веб апликацију намењену креирању и решавању квизова, развијену као практична студија случаја за истраживање имплементације посматрачке способности у микросервисним архитектурама. Платформа је конципирана као образовни алат који омогућава корисницима да тестирају своје знање кроз интерактивне квизове из различитих области.

Систем разликује три типа корисника према њиховим улогама. Обични корисници могу да се региструју, пријаве на платформу, прегледају доступне квизове, решавају их и прате своју историју резултата. Наставници (енгл. *Teacher*) имају проширена овлашћења која им омогућавају креирање нових квизова са питањима и понуђеним одговорима. Администратори поседују потпуна овлашћења над системом, укључујући могућност креирања, ажурирања и брисања квизова, као и управљања корисничким налозима и промоције корисника у више улоге.

Аутентификација се реализује путем JWT (JSON *Web Token*) стандарда, при чему сваки токен садржи информације о кориснику укључујући његову улогу у систему. Овај приступ омогућава безстатусну (енгл. stateless) верификацију идентитета на нивоу сваког микросервиса, без потребе за централизованим сервисом сесија.

Кориснички профили обухватају основне податке као што су корисничко име, електронска адреса, име и презиме, док је лозинка сачувана у облику криптографског хеша са додатном соли (енгл. *salt*) ради повећане безбедности. Систем такође подржава чување профилних слика за персонализацију корисничких профила.

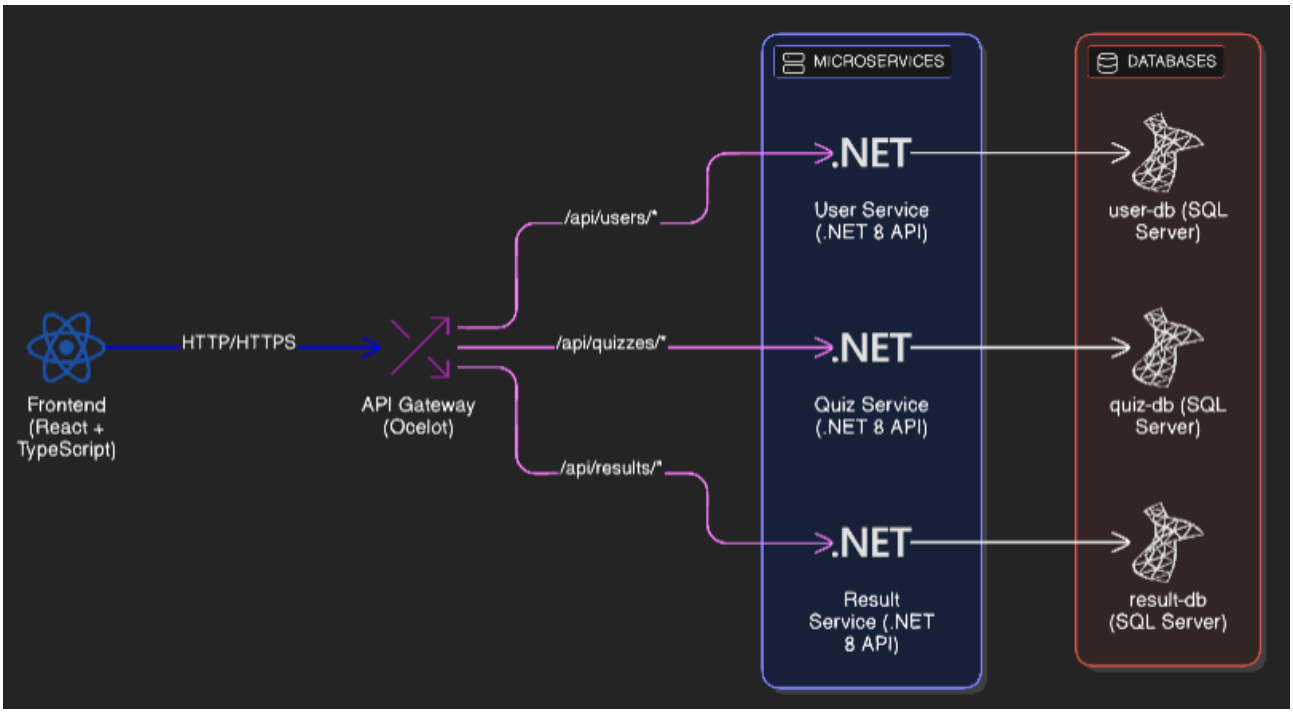
Квизови се састоје од наслова, описа, категорије и скупа питања. Свако питање може имати произвољан број понуђених одговора, при чему је тачно један означен као исправан. Након решавања квиза, систем аутоматски рачуна резултат на основу броја тачних одговора и бележи га заједно са временом завршетка и трајањем решавања.

Аналитичке функционалности омогућавају корисницима увид у сопствену историју решавања квизова, укључујући постигнуте резултате и статистику по категоријама. Глобална ранг листа (енгл. *leaderboard*) приказује најуспешније кориснике платформе, пружајући елемент такмичења и мотивације.

3.2. Микросервисна архитектура

QuizHub систем је имплементиран као микросервисна архитектура заснована на REST API топологији. Ова архитектура користи API капију (енгл. API Gateway) као јединствену тачку приступа систему, омогућавајући комуникацију клијената са позадинским сервисима искључиво путем REST API позива. Таква топологија је погодна за веб апликације умерене сложености јер обезбеђује централизовано рутирање, аутентификацију и праћење захтева на једном месту.

Архитектура се састоји од четири главна сервиса: API *Gateway*, *User Service*, *Quiz Service* и *Result Se*rvice, уз *React* апликацију која служи као кориснички интерфејс. Сваки сервис је одговоран за специфичну доменску област и управља сопственом базом података, што омогућава независан развој, тестирање и постављање појединачних компоненти.



Слика 3.2.1. *QuizHub* – архитектура система

У овој архитектури, јасна подела одговорности између сервиса представља кључни принцип. *User Service* се фокусира искључиво на управљање корисничким налозима, аутентификацију и ауторизацију. *Quiz Service* обрађује све операције везане за квизове, укључујући креирање, ажурирање и преузимање квизова са питањима. *Result Service* је одговоран за складиштење и анализу резултата решавања квизова, укључујући израчунавање статистика и вођење ранг листе.

Принцип изоловане базе података по сервису (енгл. *database-per-service*) примењен је доследно кроз целу архитектуру. Сваки микросервис има сопствену SQL Server инстанцу, што елиминише директне зависности на нивоу података између сервиса. Када је једном сервису потребан податак из домена другог сервиса, комуникација се остварује искључиво путем HTTP позива ка REST API-ју одговарајућег сервиса. На пример, *Result Service* при приказивању ранг листе позива *User Service* ради преузимања имена корисника, уместо директног приступа бази корисника.

Овакав приступ доноси неколико предности битних за клауд окружење. Сервиси се могу независно скалирати према оптерећењу — ако *Quiz Service* захтева више ресурса током периода интензивног решавања квизова, само он се хоризонтално проширује без утицаја на остале компоненте. Такође, сервиси се могу ажурирати и постављати независно, што скраћује циклус испоруке нових функционалности и смањује ризик од неуспешних постављања.

3.2.1. API *Gateway* (Ocelot)

API *Gateway* представља јединствену тачку уласка у систем и централно место за примену политика приступа. Имплементиран је коришћењем ***Ocelot* библиотеке** за .NET платформу, која пружа лаку и флексибилну имплементацију API капије погодну за микросервисне архитектуре.

Примарна функција *Gateway*-а је рутирање долазних захтева ка одговарајућим позадинским сервисима. На основу путање захтева, *Gateway* одређује који сервис треба да обради захтев и прослеђује га на одговарајућу адресу унутар *Kubernetes* кластера. Захтеви ка */api/users/** се рутирају ка *User Service*-у, захтеви ка */api/quizzes/** ка *Quiz Service*-у, а захтеви ка */api/results/** ка *Result Service*-у.

Поред рутирања, *Gateway* обавља проверу JWT токена за заштићене путање. Сви захтеви осим регистрације и пријаве морају садржати валидан JWT токен у заглављу захтева. *Gateway* верификује потпис токена и проверава његову валидност пре прослеђивања захтева ка циљном сервису, чиме се централизује логика аутентификације.

CORS (енгл. ***Cross-Origin Resource Sharing***) политика је такође конфигурисана на нивоу Gateway-а, омогућавајући *Frontend* апликацији да комуницира са API-јем са различитог домена или порта. Ово је неопходно за стандардни развојни ток где се *React* апликација сервира са једног порта, а API са другог.

Конфигурација рута се дефинише декларативно у JSON формату, што олакшава одржавање и проширивање система:

```
"Routes": [
  {
    "DownstreamPathTemplate": "/api/users/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
      { "Host": "user-service", "Port": 80 }
    ],
    "UpstreamPathTemplate": "/api/users/{everything}",
    "UpstreamHttpMethod": ["GET", "POST", "PUT", "DELETE", "OPTIONS"],
    "AuthenticationOptions": {
      "AuthenticationProviderKey": "JwtBearer",
    }
  },
  {
    "DownstreamPathTemplate": "/api/quizzes/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
      { "Host": "quiz-service", "Port": 80 }
    ],
    "UpstreamPathTemplate": "/api/quizzes/{everything}",
    "UpstreamHttpMethod": ["GET", "POST", "PUT", "DELETE", "OPTIONS"],
    "AuthenticationOptions": {
      "AuthenticationProviderKey": "JwtBearer",
    }
  }
]
```

Слика 3.2.1.1. *Ocelot* – конфигурација рута унутар *Ocelot.json* датотеке

3.2.2. User Service

User Service представља централну компоненту система за управљање корисничким идентитетима и контролу приступа. Сервис је развијен коришћењем .NET 8 *Web* API радног оквира са *Entity Framework Core* за приступ SQL *Server* бази података.

Сервис обезбеђује функционалности регистрације нових корисника, при чему се лозинка никада не чува у изворном облику већ се примењује криптографски хеш алгоритам са насумичном соли. Приликом пријаве, сервис верификује унете акредитиве и генерише JWT токен који садржи идентификатор корисника, корисничко име и улогу. Токен је потписан тајним кључем и има ограничено време важења.

Систем улога омогућава врло грануларну контролу приступа. Обични корисници имају улогу User и могу само читати и решавати квизове. Корисници са улогом *Teacher* могу додатно креирати нове квизове. Администратори са улогом *Admin* имају потпуна овлашћења укључујући брисање и ажурирање квизова других корисника, као и промоцију корисника у више улоге.

Доменски модел корисника обухвата следеће атрибуте:

```
public class User
{
    public Guid Id { get; set; }
    public required string Username { get; set; }
    public required string Email { get; set; }
    public required byte[] PasswordHash { get; set; }
    public required byte[] PasswordSalt { get; set; }
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
    public required string Role { get; set; }
    public byte[] AvatarImage { get; set; }
}
```

3.2.3. Quiz Service

` *Quiz Service* представља централну компоненту за управљање квизовима и питањима. Овај сервис омогућава креирање богатих квизова са различитим типовима питања, категоризацијом и подешавањем нивоа тежине.

Квизови се организују по категоријама (нпр. историја, наука, спорт) и нивоима тежине (лако, средње, тешко), што олакшава корисницима проналажење садржаја који одговара њиховим интересовањима и способностима. Сваки квиз може имати временско ограничење изражено у секундама, након чијег истека се решавање аутоматски прекида.

Систем подржава четири типа питања: питања са једним тачним одговором (енгл. *single choice*), питања са више тачних одговора (енгл. *multiple choice*), тачно/нетачно питања и питања са уносом текста. Свако питање може носити различит број поена и опционо може садржати објашњење које се приказује након одговарања.

Контрола приступа за операције над квизовима реализована је кроз ASP.NET Core атрибуте ауторизације. Креирање квизова је дозвољено корисницима са улогом *Admin* или *Teacher*, док су ажурирање и брисање резервисани искључиво за администраторе:

```
[HttpPost]
[Authorize(Roles = "Admin,Teacher")]
public async Task<IActionResult> CreateQuiz([FromBody] CreateQuizRequestDto
createQuizDto)
{
    // Креирање квиза...
}

[HttpDelete("{id}")]
[Authorize(Roles = "Admin")]
public async Task<IActionResult> DeleteQuiz(Guid id)
{
    // Брисање квиза...
}
```

Доменски модели *Quiz Service*-а обухватају ентитете *Quiz*, *Question* и *Answer* са следећом структуром:

```
public class Quiz
{
    public Guid Id { get; set; }
    public required string Title { get; set; }
    public string? Description { get; set; }
    public required string Category { get; set; }
    public Difficulty od { get; set; }  // Easy, Medium, Hard
    public int TimeLimitSeconds { get; set; }
    public Guid CreatedByUserId { get; set; }
    public bool IsDeleted { get; set; }
    public virtual ICollection<Question> Questions { get; set; }
}

public class Question
{
    public Guid Id { get; set; }
    public Guid QuizId { get; set; }
    public required string Text { get; set; }
    public QuestionType NT { get; set; }  // Single, Multiple, TrueFalse, FillIn
    public int Points { get; set; }
    public string? Explanation { get; set; }
    public int Order { get; set; }
    public required ICollection<Answer> Answers { get; set; }
}
```

3.2.4. Result Service

Result Service је задужен за трајно складиштење и анализу резултата решавања квизова. Сваки пут када корисник заврши квиз, *Result Service* прима информације о одговорима и рачуна постигнути резултат на основу поена дефинисаних за свако питање.

Резултат се чува заједно са мета подацима који укључују идентификатор корисника, идентификатор квиза, постигнути и максимални могући број поена, време завршетка и

трајање решавања у секундама. Додатно, чувају се појединачни одговори корисника ради накнадне анализе и приказа грешака.

Сервис излаже API за преузимање историје резултата корисника, преглед свих резултата за одређени квиз и генерисање глобалне ранг листе. Контрола приступа обезбеђује да обични корисници могу видети само сопствене резултате, док администратори имају увид у све податке.

При комуникацији са другим сервисима, Result Service примењује принцип међусервисних HTTP позива. На пример, за приказ детаља на ранг листи, сервис позива *User Service* ради преузимања корисничких имена уместо директног приступа бази корисника. Овај приступ одржава изолованост сервиса и поштује границе домена.

Доменски модел резултата:

```
public class Result
{
    public Guid Id { get; set; }
    public Guid UserId { get; set; }
    public Guid QuizId { get; set; }
    public float Score { get; set; }
    public float MaxPossibleScore { get; set; }
    public int TimeTakenSeconds { get; set; }
    public DateTime CompletedAt { get; set; }
    public required ICollection<ResultAnswer> ResultAnswers { get; set; }
}
```

3.2.5. Frontend апликација

Кориснички интерфејс је имплементиран као апликација са једном страницом (енгл. ***Single Page Application*** - SPA) коришћењем *React* библиотеке верзије 18 са ***TypeScript*** програмским језиком за типску безбедност. Апликација комуницира са позадинским системом искључиво путем REST API позива ка *Gateway* сервису.

React Router библиотека омогућава рутирање на клијентској страни, при чему се различите странице (пријава, листа квизова, решавање квиза, резултати) приказују без поновног учитавања целе странице. ***Axios*** библиотека се користи као HTTP клијент за комуникацију са API-јем, пружајући једноставан интерфејс за слање захтева и обраду одговора.

Стање аутентификације се управља на нивоу апликације коришћењем ***React Context*** API-ја. Након успешне пријаве, JWT токен се чува у локалном складишту претраживача (енгл. *localStorage*) и аутоматски се додаје у заглавље сваког одлазног захтева. Кориснички интерфејс динамички прилагођава доступне опције на основу улоге пријављеног корисника - на пример, дугме за креирање квиза је видљиво само наставницима и администраторима.

За визуелни изглед користе се SCSS стилови који омогућавају угнеждавање правила и коришћење варијабли, што олакшава одржавање конзистентног дизајна кроз целу апликацију. Апликација је оптимизована за приказ на различитим величинама екрана укључујући мобилне уређаје.

3.3. Анализа недостатака постојећег система

Пре имплементације посматрачке способности, QuizHub систем је имао значајне недостатке у погледу видљивости унутрашњег стања и могућности дијагностике проблема. Ови недостаци су типични за микросервисне архитектуре које нису опремљене одговарајућим алатима за посматрање.

3.3.1. Логовање

У почетној имплементацији, логовање је било имплементирано само на нивоу API *Gateway* сервиса коришћењем ***Serilog*** библиотеке. *Gateway* је бележио све долазне захтеве и одговоре у локални фајл (*logs/gateway-.log*) са дневном ротацијом, што је пружало основни увид у саобраћај који пролази кроз систем.

Међутим, остали микросервиси (*User Service, Quiz Service, Result Service*) нису имали конфигурисано структурирано логовање. Користили су подразумеване .NET логове који су били доступни само кроз стандардни излаз контејнера. Приступ овим логовима захтевао је ручно повезивање на сваки контејнер појединачно коришћењем команде ***docker logs*** или ***kubectl logs*****,** што је било непрактично приликом дијагностике проблема.

Чак и логови са **Gateway**-а, иако структурирани, били су ограничени на локални фајл систем контејнера. При рестартовању или поновном постављању *Gateway* pod-а, историјски логови би били изгубљени јер волумен за логове није био перзистентан. Ово је значајно отежавало *post-mortem* анализу инцидената.

Додатно, није постојао механизам за корелацију логова између различитих сервиса. Иако је *Gateway* бележио захтеве, није било могуће пратити шта се дешавало са тим захтевима унутар позадинских сервиса. Формат логова такође није био конзистентан - *Gateway* је користио структурирани JSON формат преко *Serilog*-а, док су остали сервиси имали неформатиране текстуалне логове.

3.3.2. Метрике

Систем није располагао метрикама на нивоу апликације. Једине доступне метрике биле су оне које пружа *Kubernetes* инфраструктура - потрошња CPU-а и меморије по контејнеру. Иако корисне за основно праћење ресурса, ове метрике не пружају увид у понашање апликације.

Без апликативних метрика, није било могуће мерити кључне индикаторе перформанси као што су број захтева у секунди, дистрибуција латенције (P50, P95, P99), стопа грешака по endpoint-у или време одговора базе података. Последично, деградација перформанси могла је проћи незапажено све док корисници не би почели да пријављују проблеме.

Није постојао систем за аутоматско упозорење (енгл. *alerting*). Тим за операције морао је ручно проверавати стање система, што је представљало реактиван приступ уместо проактивног. Капацитетно планирање такође није било засновано на подацима, већ на претпоставкама и повременим ручним мерењима.

3.3.3. Праћење захтева

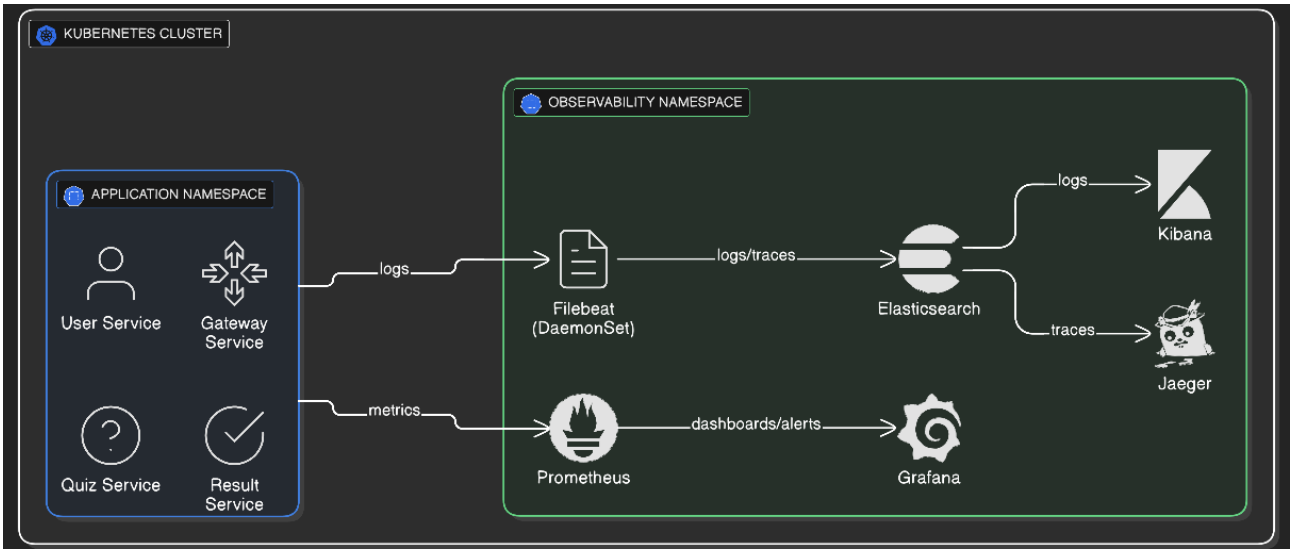
Немогућност праћења појединачног захтева кроз сервисе представљала је један од најзначајнијих недостатака. Када би корисник пријавио да је одређена операција спора, није

постојао начин да се утврди који сервис или која операција унутар сервиса узрокује кашњење.

Захтеви нису имали јединствени идентификатор (енгл. *trace* ID) који би их пратио кроз целу обраду. Ово је онемогућавало утврђивање зависности између сервиса и препознавање уских грла у обради. Дијагностика латенције захтевала је ручну анализу временских ознака из различитих логова, што је било непрактично и подложно грешкама.

3.4. Циљана архитектура посматрачке способности

На основу анализе недостатака, дефинисана је циљана архитектура која имплементира сва три стуба посматрачке способности:



Слика 3.4.1.*QuizHub* – Архитектура платформе са опсервабилношћу

4. Имплементација решења

4.1. Провизионисање инфраструктуре помоћу *Terraform*

Пре постављања апликације на Kubernetes кластер, неопходно је припремити *cloud* инфраструктуру која ће угостити целокупни систем. За аутоматизовано и поновљиво креирање Azure ресурса примењен је приступ инфраструктуре као кода (енгл. *Infrastructure as Code* — IaC) коришћењем *Terraform* алата.

4.1.1. Концепт инфраструктуре као кода

Инфраструктура као код представља праксу управљања и провизионисања рачунарске инфраструктуре кроз машински читљиве конфигурационе датотеке, уместо ручне конфигурације преко графичког интерфејса или интерактивних команди [46]. Овај приступ доноси неколико кључних предности у контексту DevOps методологије.

Декларативна природа IaC омогућава дефинисање жељеног стања инфраструктуре, при чему алат аутоматски одређује кораке потребне за достизање тог стања. Конфигурационе датотеке се чувају у систему за контролу верзија (*Git*), што обезбеђује потпуну историју промена, могућност враћања на претходно стање и колаборацију између чланова тима. Исте

конфигурације се могу применити за креирање идентичних окружења (развојно, тестно, продукционо), елиминишући проблем разлика између окружења.

Terraform је алат отвореног кода компаније *HashiCorp* [47] који подржава провизионисање инфраструктуре на више од 3000 *cloud* провајдера, укључујући *Azure*, *AWS* и *Google Cloud Platform*. *Terraform* користи сопствени декларативни језик HCL (енгл. ***HashiCorp Configuration Language***) за дефинисање ресурса.

Животни циклус *Terraform* конфигурације се састоји од четири основне фазе. У фази иницијализације (terraform init) преузимају се потребни провајдери — у случају *QuizHub* пројекта то су *azurerm* за *Azure* ресурсе, *kubernetes* за *Kubernetes* објекте и *helm* за *Helm* чартове. **Фаза планирања** (*terraform plan*) анализира тренутно стање инфраструктуре и генерише план промена без стварног извршавања. **Фаза примене** (*terraform apply*) извршава планиране промене и креира или модификује ресурсе. Коначно, **фаза уништавања** *(terraform destroy*) уклања све ресурсе дефинисане у конфигурацији [47].

4.1.2. Структура Terraform конфигурације

Terraform конфигурација за QuizHub пројекат организована је у директоријуму terraform/azure/ са следећом структуром датотека:

Датотека	Намена
main.tf	Главна дефиниција свих Azure и Kubernetes ресурса
variables.tf	Декларација улазних варијабли са подразумеваним вредностима
outputs.tf	Дефиниција излазних вредности након провизионисања
terraform.tfvars	Конкретне вредности варијабли за продукционо окружењ
deploy.sh	Bash скрипта за аутоматизовано постављање
destroy.sh	Скрипта за потпуно уклањање инфраструктуре

Табела 4.1.2.1. Садржај Terraform директоријума и намена сваке од датотека

Конфигурација захтева **три *Terraform* провајдера** са специфичним верзијама ради обезбеђивања компатибилности [48]:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.23"
```

```
    }
    helm = {
      source  = "hashicorp/helm"
      version = "~> 2.11"
    }
  }
}
```

4.1.3. Azure ресурси

Terraform конфигурација креира следеће *Azure* ресурсе неопходне за хостовање *QuizHub* апликације:

- ***Resource Group*** представља логички контејнер који групише све Azure ресурсе пројекта. Омогућава управљање животним циклусом свих ресурса на једном месту, брисањем групе аутоматски се бришу сви ресурси унутар ње [49].
- ***Virtual Network*** обезбеђује изолован мрежни простор за AKS кластер. Конфигурисан је са адресним опсегом 10.0.0.0/8 који пружа преко 16 милиона IP адреса, што је више него довољно за скалирање кластера [49].
- ***Azure Kubernetes Service (AKS)*** кластер представља срце инфраструктуре [50]. Конфигурисан је са следећим карактеристикама:
  - *Kubernetes* верзија 1.28.3 (**LTS** верзија са **дугорочном подршком**)
  - ***Azure CNI* мрежни додатак** који омогућава директно IP адресирање pod-ова
  - **Аутоматско скалирање чворова** од 2 до 4 инстанце типа *Standard_B2s* (2 vCPU, 4GB RAM)
  - Систем управљаних идентитета за безбедан приступ другим *Azure* сервисима
  - Интеграција са *Log Analytics* за централизовано праћење логова
- ***Azure Container Registry (ACR)*** служи као приватни регистар *Docker* слика. Сви микросервиси се прво изграђују локално, затим означавају *ACR* путањом и објављују у регистар [51]. *AKS* кластер има аутоматски додељену *AcrPull* улогу која му омогућава преузимање слика без експлицитне аутентификације.
- ***Log Analytics Workspace*** прикупља логове и метрике са *AKS* кластера кроз *Azure Container Insights* функционалност [52]. Ретенција је подешена на 30 дана, што представља баланс између потреба за дијагностиком и трошкова складиштења
- ***Public IP*** обезбеђује статичку јавну IP адресу која се додељује *NGINX Ingress* контролеру. Статичка алокација гарантује да IP адреса остаје непромењена током рестартовања или поновног постављања *Ingress* контролера

4.1.4. Kubernetes ресурси кроз Terraform

Након креирања *AKS* кластера, *Terraform* **аутоматски** конфигурише ***Kubernetes* и *Helm* провајдере** добијене од кластера. Ово омогућава креирање *Kubernetes* ресурса у истом *Terraform* пролазу.

***Namespace*-ови (окружења)** раздвајају апликативне ресурсе од инфраструктуре за посматрање [23]. Окружење quizhub садржи све микросервисе и базе података, док *observability* садржи *ELK stack*, *Prometheus*, *Grafana* и *Jaeger.*

NGINX Ingress Controller се инсталира као *Helm* чарт из званичног *Kubernetes* репозиторијума [53][54]. Конфигурисан је да користи претходно креирану статичку IP адресу и да излаже *Prometheus* метрике.

4.1.5. Параметризација конфигурације

Terraform варијабле омогућавају прилагођавање инфраструктуре без измене главне конфигурације. Све варијабле имају подразумеване вредности оптимизоване за студентска Azure окружења са ограниченим буџетом.

4.1.6. Параметризација конфигурације

Након успешног провизионисања, Terraform излаже кључне информације неопходне за даљу конфигурацију [55]:

```
output "configure_kubectl" {
  value = "az aks get-credentials --resource-group
${azurerm_resource_group.quizhub.name} --name
${azurerm_kubernetes_cluster.quizhub.name}"
}
output "acr_login_server" {
  value = azurerm_container_registry.quizhub.login_server
}
output "ingress_ip" {
  value = azurerm_public_ip.ingress.ip_address
}
```

Аутоматизована скрипта орхестрира целокупан процес провизионисања у седам корака: провера предуслова (*Terraform*, *Azure* CLI, *kubectl*), аутентификација на *Azure*, иницијализација *Terraform* провајдера, валидација конфигурације, генерисање плана извршења, примена промене на *Azure*, и конфигурација *kubectl* за приступ кластеру. Цео процес траје приближно 10-15 минута, од чега највећи део одлази на креирање AKS кластера [55].

4.2. Контејнеризација и постављање апликације

Након успешног провизионисања *Azure* инфраструктуре, следећи корак представља припрему микросервиса за извршавање у Kubernetes окружењу. Овај процес обухвата контејнеризацију апликација, дефинисање Kubernetes манифеста и конфигурисање мрежног приступа.

4.2.1. Контејнеризација сервиса

Сваки микросервис је контејнеризован применом вишефазног *Docker build* процеса (енгл. *multi-stage build*), који раздваја фазу компилације од финалне продукционе слике. Овај

приступ значајно смањује величину коначне слике јер се алати и међупродукти компилације не укључују у продукциону слику.

Прва фаза користи *mcr.microsoft.com/dotnet/sdk:8.0* базну слику која садржи комплетан .NET SDK неопходан за компилацију. У овој фази се прво копирају само .csproj датотеке пројеката и извршава команда *dotnet restore* за преузимање *NuGet* зависности. Овакав редослед искоришћава *Docker* кеширање слојева — све док се зависности не промене, овај слој се не мора поново извршавати.

Друга фаза користи минималну *mcr.microsoft.com/dotnet/aspnet:8.0* базну слику која садржи само *ASP.NET Core runtime* окружење. Величина ове слике је приближно 220MB у поређењу са 900MB за SDK слику. Компилирана апликација се копира из прве фазе, конфигурише се порт 80 за HTTP саобраћај и дефинише улазна тачка за покретање сервиса.

```
# Фаза компилације
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["QuizService.Api/QuizService.Api.csproj", "QuizService.Api/"]
RUN dotnet restore "QuizService.Api/QuizService.Api.csproj"
COPY . .
RUN dotnet publish "QuizService.Api/QuizService.Api.csproj" -c Release -o
/app/publish

# Продукциона фаза
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY --from=build /app/publish .
ENV ASPNETCORE_URLS=http://+:80
EXPOSE 80
ENTRYPOINT ["dotnet", "QuizService.Api.dll"]
```

Слика 4.2.1.1. Docker - вишефазни build процес за .NET сервисе

Постављање микросервиса у Kubernetes захтева дефинисање два основна ресурса за сваки сервис: ***Deployment*** који управља животним циклусом pod-ова и ***Service*** који обезбеђује стабилну мрежну адресу.

Deployment ресурс дефинише жељено стање апликације укључујући број реплика, слику контејнера, ресурсне захтеве и провере здравља. За Quiz Service конфигурисане су две реплике ради високе доступности. Анотације prometheus.io/* омогућавају Prometheus-у аутоматско откривање и прикупљање метрика са крајње тачке /metrics.

Контејнер преузима слику из *Azure Container Registry*-ја (*quizhubacr.azurecr.io/quiz-service:latest*). Конекциони стринг за базу података инјектује се из Kubernetes ***Secret*-а**, чиме се избегава чување осетљивих података у манифестима. Адреса *Jaeger* агента прослеђује се као променљива окружења за слање трагова.

Ресурсни захтеви и лимити осигуравају предвидљиво понашање под оптерећењем. Захтеви (енгл. *requests*) од 256Mi меморије и 200m CPU-а гарантују минималне ресурсе за сваку реплику. Лимити од 512Mi и 500m спречавају да појединачни pod преузме превише ресурса од других радних оптерећења.

Провере здравља имају кључну улогу у одржавању доступности сервиса. Провера активности (енгл. *liveness probe*) периодично проверава крајњу тачку /*health* и рестартује контејнер ако сервис не одговара. Провера спремности (енгл. *readiness probe*) проверава крајњу тачку */health/ready* и уклања pod из балансирања оптерећења ако није спреман да прихвати саобраћај, на пример током иницијализације или повезивања са базом података.

Service ресурс типа ***ClusterIP*** креира стабилну интерну DNS адресу (*quiz-service.quizhub.svc.cluster.local)* којом други сервиси могу приступити *Quiz Service*-у. Kubernetes аутоматски балансира саобраћај између свих здравих реплика.

4.2.2. База података као StatefulSet

SQL Server базе података захтевају другачији приступ од микросервиса без стања. *Deployment* ресурс није погодан јер не гарантује стабилност идентитета *pod*-а и не подржава перзистентно складиште на одговарајући начин. Уместо тога, примењен је ***StatefulSet*** ресурс који пружа:

- Стабилне мрежне идентитете: Сваки pod добија предвидљиво DNS име (нпр. *quiz-db-0.quiz-db.databases.svc.cluster.local*) које остаје непромењено током рестартовања.
- Уређено скалирање: Pod-ови се креирају и бришу секвенцијално, омогућавајући контролисану иницијализацију.
- Перзистентно складиште: Шаблон за захтев волумена (енгл. *VolumeClaimTemplate*) аутоматски креира *PersistentVolumeClaim* за сваку реплику, обезбеђујући да подаци преживе рестартовање.

SQL Server 2022 Express Edition је конфигурисан са минималним ресурсима погодним за развојно и тест окружење (1GB RAM захтев, 2GB лимит). Лозинка системског администратора се преузима из Kubernetes ***Secret***-а ради безбедности. Волумен */var/opt/mssql* садржи све податке базе и перзистира на **Azure *Managed Disk***-у капацитета 5GB.

4.2.3. Конфигурација *Ingress* ресурса

NGINX Ingress Controller, инсталиран *кроз Terraform Helm release*, рутира екстерни HTTP/HTTPS саобраћај ка одговарајућим сервисима унутар кластера. ***Ingress*** ресурс дефинише правила рутирања базирана на путањи захтева.

Захтеви ка путањама које почињу са */api/* прослеђују се *Gateway* сервису који даље рутира ка одговарајућим микросервисима. Сви остали захтеви се усмеравају ка *Frontend* сервису који сервира *React* апликацију. Анотација *nginx.ingress.kubernetes.io/rewrite-target* омогућава уклањање префикса путање пре прослеђивања захтева позадинском сервису.

CORS конфигурација на нивоу *Ingress*-а дозвољава захтеве са било ког порекла (*), што је неопходно за развојно окружење где се *Frontend* може сервирати са различите адресе. У продукционом окружењу ова вредност би требало да буде ограничена на конкретан домен апликације.

4.3. Имплементација централизованог логовања (ELK *Stack*)

Централизовано логовање представља фундаменталну компоненту посматрачке способности која омогућава агрегацију, претрагу и анализу логова из свих микросервиса на

једном месту. За имплементацију је изабран ELK Stack (Elasticsearch, Filebeat, Kibana) који представља де факто стандард у индустрији за управљање логовима [48].

4.3.1. Постављање Elasticsearch-а

Elasticsearch служи као централно складиште за све логове система. Постављен је као ***StatefulSet*** ресурс у Kubernetes-у ради обезбеђивања перзистентног складишта података. У развојном окружењу користи се једна инстанца у режиму појединачног чвора (енгл. ***single-node***), док би продукционо окружење захтевало кластер са најмање три чвора ради високе доступности.

Конфигурација *Elasticsearch*-а подразумева коришћење званичне *Docker* слике верзије 8.11.0 са следећим кључним подешавањима:

- ***discovery.type=single-node***: Омогућава рад без потребе за откривањем других чворова у кластеру
- ***xpack.security.enabled=false***: Искључује безбедносне функције ради поједностављења развојног окружења
- ***ES_JAVA_OPTS=-Xms512m -Xmx512m***: Ограничава JVM *heap* меморију на 512 MB ради економичног коришћења ресурса

Ресурсни захтеви су конфигурисани са минимумом од 1 GB RAM-а и 500m CPU-а, уз лимите од 2 GB и 1000m. Подаци се перзистирају на *Azure Managed Disk*-у капацитета 10 GB монтираном на путању */usr/share/elasticsearch/data.*

4.3.2. Конфигурација *Filebeat* агента

Filebeat представља лаки агент за прослеђивање логова који се покреће на сваком чвору кластера путем ***DaemonSet*** ресурса [51]. Овакав модел постављања гарантује да се логови прикупљају са свих контејнера без обзира на ком чвору се извршавају.

Кључна карактеристика *Filebeat* конфигурације је механизам аутоматског откривања (енгл. ***autodiscover***) који динамички препознаје нове контејнере и аутоматски започиње прикупљање њихових логова. Конфигурација користи Kubernetes провајдер који се ослања на Kubernetes API за откривање подова:

```
filebeat.autodiscover:
  providers:
   - type: kubernetes
     node: ${NODE_NAME}
     hints.default_config:
       type: container
       paths:
         - /var/log/containers/*${data.kubernetes.container.id}.log
```

Процесори обогаћују сваку лог поруку Kubernetes метаподацима укључујући име пода, именски простор, ознаке и име контејнера. Додатно, JSON декодер покушава да парсира структуриране логове ради омогућавања претраге по појединачним пољима. Филтер искључује логове из системских именских простора (*kube-system*) и самог Filebeat-а ради смањења шума.

Излазна конфигурација усмерава логове ка *Elasticsearch*-у са динамичким именовањем индекса по шаблону *logs-quizhub-{app}-{date}*, што омогућава ефикасну ротацију и филтрирање по сервису.

4.3.3. Структурирано логовање у .NET апликацијама

Да би централизовано логовање било ефективно, неопходно је да све апликације генеришу логове у конзистентном, структурираном формату. За .NET сервисе имплементирано је структурирано логовање коришћењем ***Serilog*** библиотеке са JSON форматером.

Конфигурација *Serilog*-а обухвата неколико кључних елемената. Минимални ниво логовања подешен је на *Information*, уз потискивање прекомерних *Microsoft* логова. Сваки лог запис се обогаћује контекстом који укључује име сервиса и окружење. JSON форматер осигурава да све лог поруке буду машински читљиве и лако парсибилне:

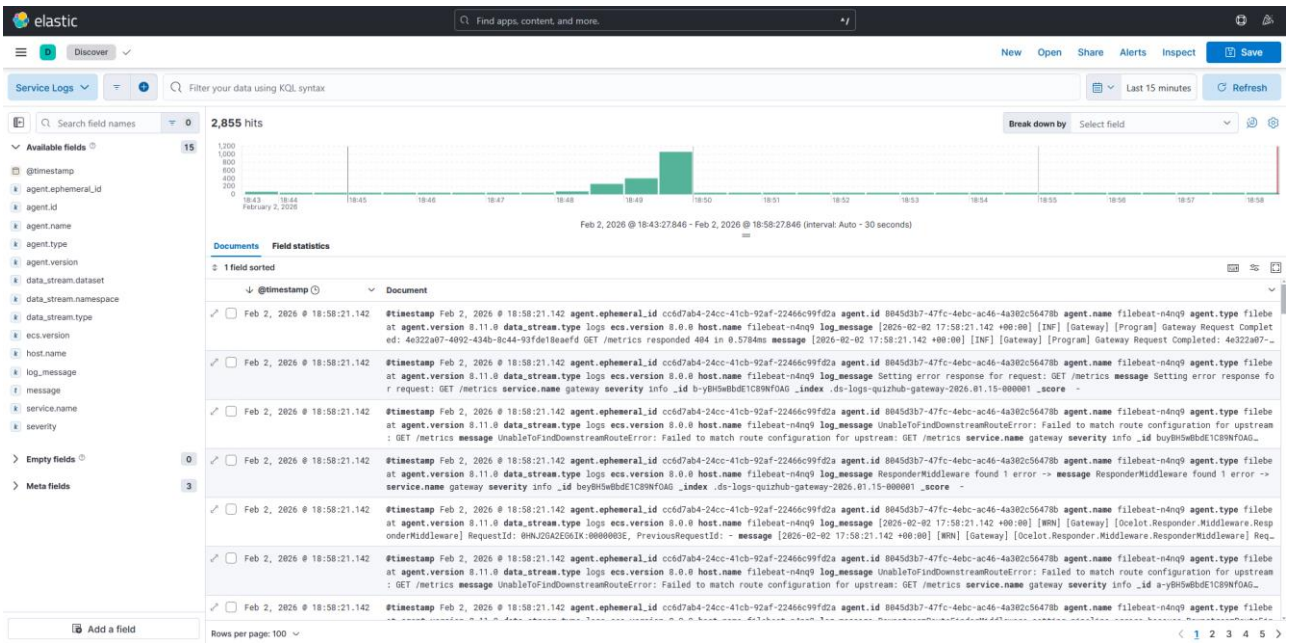
```
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Information()
    .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
    .Enrich.FromLogContext()
    .Enrich.WithProperty("Service", "QuizService")
    .WriteTo.Console(new JsonFormatter())
    .CreateLogger();
```

Посебан *middleware* за корелацију захтева додаје јединствени идентификатор (енгл. *correlation ID*) сваком HTTP захтеву. Ако долазни захтев већ садржи овај идентификатор у заглављу *X-Correlation-ID*, он се преузима и прослеђује даље; у супротном се генерише нови. Овај идентификатор се додаје у контекст логовања и укључује у одговор, омогућавајући праћење целокупног тока захтева кроз све сервисе.

4.3.4. Kibana за визуелизацију логова

Kibana пружа веб интерфејс за претрагу, филтрирање и визуелизацију логова складиштених у *Elasticsearch*-у [52]. Постављена је као ***Deployment*** ресурс са једном репликом и повезана је са *Elasticsearch*-ом путем интерне Kubernetes *Service* адресе.

Интерфејс омогућава креирање прилагођених претрага коришћењем ***Kibana Query Language*** (KQL) синтаксе. Типичне претраге укључују филтрирање по сервису (*kubernetes.labels.app: "quiz-service"),* нивоу логовања (*level*: "*Error*") или корелационом идентификатору за праћење конкретног захтева кроз систем.



Слика 4.3.4.1. Kibana - интерфејс за претрагу логова

4.3.5. Управљањем животним циклусом индекса

Како би се контролисали трошкови складиштења и одржале перформансе Elasticsearch-а, имплементирана је политика управљања животним циклусом индекса (енгл. ***Index Lifecycle Management*** - ILM) [53]. Политика дефинише три фазе:

Фаза	Старост	Акције
Hot	0-7 дана	Примарно складиште, ротација при 10 GB или 1 дан
Warm	7-30 дана	Смањење броја шардова, спајање сегмената
Delete	>30 дана	Аутоматско брисање старих логова

Табела 4.3.5.1. Конфигурација животног циклуса индекса

Овакав приступ балансира потребу за историјским подацима и оперативне трошкове, при чему се најновији логови држе на брзом складишту, а старији се компресују или бришу.

4.4. Имплементација прикупљања података – *Prometheus*

Prometheus представља систем за прикупљање и складиштење метрика временских серија. Његов *pull* модел прикупљања, где *Prometheus* активно преузима метрике са циљних крајњих тачака, посебно је погодан за динамично Kubernetes окружење.

4.4.1. Архитектура Prometheus-а у Kubernetes-у

Prometheus је постављен као Deployment ресурс са перзистентним складиштем за временске серије. Кључне конфигурационе опције укључују период задржавања података од 15 дана (*storage.tsdb.retention.time=15d*) и омогућен *lifecycle* API за динамичко учитавање конфигурације без рестарта.

Централни део конфигурације представља механизам аутоматског откривања сервиса (енгл. ***service discovery***) који користи Kubernetes API за проналажење циљева за прикупљање метрика [56]. Конфигурисана су четири посла прикупљања:

1. ***kubernetes-apiservers***: Прикупља метрике са Kubernetes API сервера;
2. ***kubernetes-nodes***: Прикупља метрике са свих чворова кластера;
3. ***kubernetes-pods***: Прикупља метрике са подова означених анотацијом ***prometheus.io /scrape: "true"***;
4. ***cadvisor***: Прикупља метрике о ресурсима контејнера (CPU, меморија, мрежа).

Посебно је значајна конфигурација за подове која омогућава декларативно означавање сервиса који излажу метрике. Путем анотација, сваки сервис може да специфира путању (*prometheus.io/path*) и порт (*prometheus.io/port*) на којима излаже метрике.

4.4.2. Инструментација .NET апликација

За излагање метрика из .NET апликација користи се ***prometheus-net*** библиотека која имплементира *Prometheus* клијентски протокол [58]. Библиотека аутоматски излаже стандардне HTTP метрике и омогућава дефинисање прилагођених метрика.

Имплементиране су следеће прилагођене метрике специфичне за QuizHub домен:

Метрика	Тип	Опис
quizhub_quizzes_created_total	Бројач	Укупан број креираних квизова
quizhub_quizzes_solved_total	Бројач	Укупан број покушаја решавања
quizhub_quiz_solve_duration_seconds	Хистограм	Дистрибуција трајања решавања
quizhub_active_users	Гејџ	Тренутни број активних корисника

Табела 4.4.2.1. Имплементација кориснички дефинисаних метрика

Бројач метрике бележе монотоно растуће вредности попут броја догађаја. Хистограм метрике омогућавају израчунавање перцентила латенције груписањем вредности у унапред дефинисане кошеве (30s, 60s, 120s, 300s, 600s, 1200s за трајање решавања квиза). Гејџ метрике представљају тренутне вредности које могу расти и опадати.

4.4.3. Крајње тачке за проверу здравља

Поред метрика, имплементиране су стандардизоване крајње тачке за проверу здравља сервиса коришћењем *ASP.NET Core Health Checks* библиотеке. Крајња тачка */health* враћа агрегирано стање сервиса, док */health/ready* проверава спремност за примање саобраћаја укључујући повезаност са базом података.

4.5. Имплементација визуелизације – *Grafana*

Grafana је конфигурисана са три извора података који покривају све аспекте посматрачке способности:

- ***Prometheus***: Примарни извор за метрике, постављен као подразумевани
- ***Elasticsearch***: Извор за претрагу и анализу логова са индексним шаблоном *logs-quizhub-**
- ***Jaeger***: Извор за визуелизацију дистрибуираних трагова

Конфигурација извора података врши се декларативно путем ***ConfigMap*** ресурса који се монтира у *Grafana* контејнер. Ово омогућава верзионисање конфигурације и аутоматско провизионисање при постављању.

- **4.5.2. Контролне табле за преглед система**

Креиране су специјализоване контролне табле прилагођене потребама QuizHub система. Главна контролна табла **„*QuizHub System Overview*"** приказује кључне индикаторе здравља система на једном месту:

- **Статус подова**: Број активних реплика сваког сервиса
- **Стопа захтева**: Број HTTP захтева у секунди по сервису, израчунат *PromQL* упитом *sum(rate(http_requests_received_total[5m])) by (service)*
- **Стопа грешака**: Проценат 4xx и 5xx одговора са праговима упозорења (жуто >1%, црвено >5%)
- **P95 латенција**: 95. перцентил времена одговора израчунат функцијом *histogram_quantile*(0.95, ...)
- **Ресурси**: Графикони искоришћености CPU-а и меморије по сервису

Контролне табле подржавају интерактивно филтрирање по временском опсегу и именском простору, омогућавајући брзу дијагностику проблема.

Табела 4.4.2.1. Grafana – контролна табла за систем метрике

4.6. Имплементација дистрибуираног праћења - *Jaeger*

Дистрибуирано праћење омогућава визуелизацију целокупног пута захтева кроз микросервисну архитектуру, идентификујући уска грла и зависности између сервиса [32]. ***Jaeger*** је изабран као решење отвореног кода компатибилно са ***OpenTelemetry*** **стандардом**.

4.6.1. Архитектура *Jaeger*-а

Jaeger је постављен у *„all-in-one*" конфигурацији која обједињује агент, колектор и *query* сервис у једном контејнеру. За перзистентно складиштење трагова користи се *Elasticsearch* као позадински систем уместо подразумеваног *in-memory* складишта, чиме се омогућава анализа историјских трагова.

Jaeger излаже неколико портова за различите протоколе:

Порт	Протокол	Намена
6831/UDP	*Thrift compact*	Примање трагова од агената
16686/TCP	HTTP	Веб интерфејс за претрагу
14268/TCP	HTTP	Директан пријем од клијената
14250/TCP	gRPC	Пријем од *OpenTelemetry* колектора

Табела 4.6.1.1. Jaeger – Приступне тачке система

4.6.2. OpenTelemetry интеграција

За инструментацију .NET апликација користи се ***OpenTelemetry SDK*** који представља произвођачки неутрални стандард за телеметрију [33]. Интеграција обухвата аутоматску инструментацију за ASP.NET *Core*, *HttpClient* и *Entity Framework Core*, као и могућност креирања прилагођених спанова.

Конфигурација *OpenTelemetry*-ја у *Program.cs* укључује:

```
builder.Services.AddOpenTelemetry()
    .ConfigureResource(resource => resource
        .AddService(serviceName: "quiz-service", serviceVersion:
"1.0.0"))
    .WithTracing(tracing => tracing
        .AddAspNetCoreInstrumentation()
        .AddHttpClientInstrumentation()
        .AddEntityFrameworkCoreInstrumentation()
        .AddJaegerExporter());
```

Аутоматска инструментација бележи све долазне HTTP захтеве, одлазне HTTP позиве ка другим сервисима и упите ка бази података. Филтер искључује провере здравља и метрике ради смањења количине телеметријских података.

4.6.3. Прилагођени спанови и пропагација контекста

Поред аутоматске инструментације, имплементирани су прилагођени спанови за праћење доменских операција. Класа *ActivitySource* користи се за креирање спанова који обухватају пословну логику:

```
using var activity = ActivitySource.StartActivity("CreateQuiz");
activity?.SetTag("quiz.title", dto.Title);
activity?.SetTag("quiz.questions_count", dto.Questions.Count);
```

Контекст трага аутоматски се пропагира кроз HTTP позиве захваљујући *AddHttpClientInstrumentation()*. Када *Quiz Service* позива *User Service* за валидацију токена, идентификатор трага се преноси у HTTP заглављима, омогућавајући повезивање свих операција у јединствен траг.

5. Тестирање и резултати

Евалуација имплементираног решења спроведена је кроз систематско тестирање које обухвата функционално тестирање компоненти посматрачке способности, тестирање перформанси под различитим нивоима оптерећења и валидацију ефективности целокупног решења.

5.1. Методологија тестирања

5.1.1. Тест окружење

Тестирање је извршено на *Azure Kubernetes Service* кластеру провизионисаном путем Terraform конфигурације описане у поглављу 4.1. Карактеристике тест окружења су:

Компонента	**Спецификација**
Kubernetes верзија	1.28.3
Број чворова	2
Тип виртуелних машина	*Standard_B2s* (2 vCPU, 4GB RAM)
Регион	*West Europe* (Амстердам)
Укупни рачунарски ресурси	4 vCPU, 8GB RAM

Табела 5.1.1.1. AKS – Конфигурација тестног окружења

Сваки микросервис је конфигурисан са две реплике ради симулације продукционог окружења са високом доступношћу. Ресурсни захтеви по реплици износе 200m CPU и 256Mi меморије за позадинске сервисе, односно 100m CPU и 128Mi за *Frontend*.

5.1.2. Алати за тестирање

За спровођење тестова коришћена је комбинација алата:

- ***Python*** скрипте: Прилагођени асинхроне скрипте базирани на *aiohttp* библиотеци за генерисање конкурентног оптерећења
- ***Apache Bench*** (*ab*): Стандардни алат за тестирање HTTP перформанси за брзе ад-хок тестове
- ***kubectl***: Команда за праћење стања подова, ресурса и догађаја у кластеру током тестирања

5.2. Функционално тестирање

5.2.1. Валидација централизованог логовања

Тестирана је исправност прикупљања, индексирања и претраге логова кроз ELK *Stack*. Тест је обухватио генерисање корисничких акција (регистрација, пријава, преузимање квизова) и верификацију да се одговарајући логови појављују у *Kibana* интерфејсу.

Типичан лог запис структуриран је према следећем формату, обогаћен Kubernetes метаподацима од стране *Filebeat* агента:

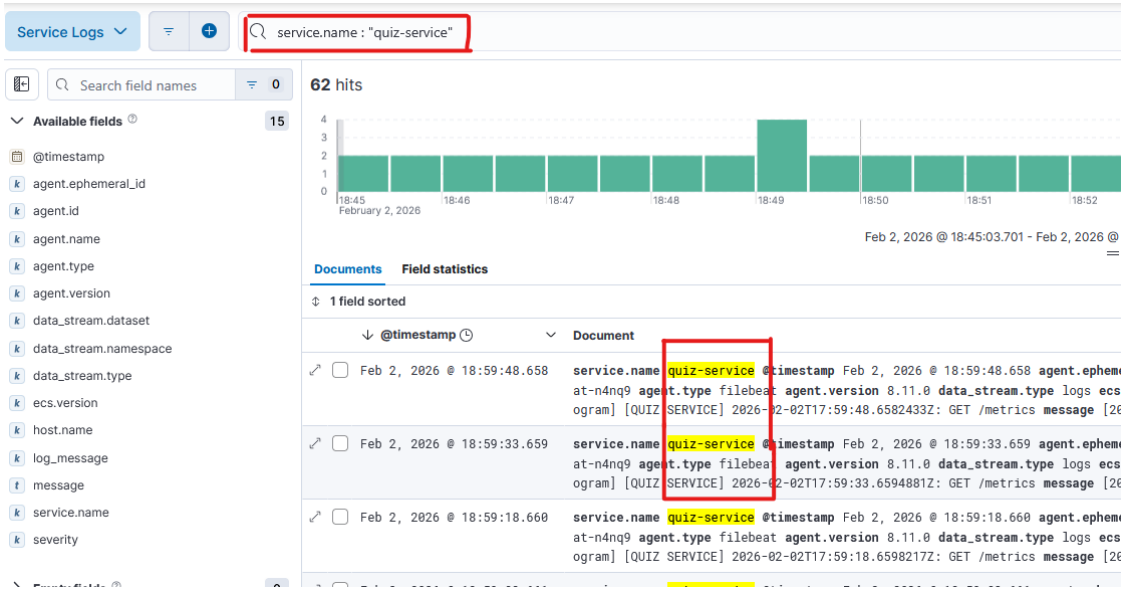
```
{
  "@timestamp": "2024-11-19T14:30:22.451Z",
  "level": "Information",
  "message": "User registered successfully",
  "service": "user-service",
  "CorrelationId": "3f4d2a1c-8b7e-4c9d-a1b2-3c4d5e6f7a8b",
  "kubernetes.namespace": "quizhub",
  "kubernetes.pod.name": "user-service-7d8f9c6b5-x2k4j",
  "kubernetes.labels.app": "user-service"
}
```

Резултати валидације су:

Критеријум	Статус	Напомена
Логови свих сервиса присутни	Успех	*Gateway, User, Quiz, Result* сервиси
Структурирани JSON формат	Успех	Омогућава претрагу по пољима
Kubernetes метаподаци	Успех	Под, именски простор, ознаке
Корелациони идентификатор	Успех	Повезивање логова истог захтева
Индексирање у реалном времену	Успех	Кашњење мање од 5 секунди

Табела 5.2.1.1. Резултати валидације централизованог логовања

Претрага логова тестирана је коришћењем KQL упита у *Kibana* интерфејсу. Филтрирање грешака по сервису (*kubernetes.labels.app: "quiz-service" AND level: "Error"*) и претрага по корелационом идентификатору успешно су враћали очекиване резултате.



Слика 5.2.1.1. Kibana - филтрирање логова по сервису и нивоу

5.2.2. Валидација метрика

Тестирана је доступност и тачност метрика излажених од стране микросервиса и прикупљених од стране *Prometheus*-а. Верификовано је да крајња тачка */metrics* враћа метрике у *Prometheus exposition* формату.

Тестирани *PromQL* упити и њихови резултати:

Упит	Опис	Резултат
sum(rate(http_requests_received_total[5m])) by (service)	Стопа захтева по сервису	*user: 12,3/s, quiz: 8,7/s*
histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (le))	P95 латенција	78 ms
sum(rate(http_requests_received_total{code=~"5.."}[5m]))	Стопа серверских грешака	0,02/s

Табела 5.2.2.1. Резултати валидације метрика

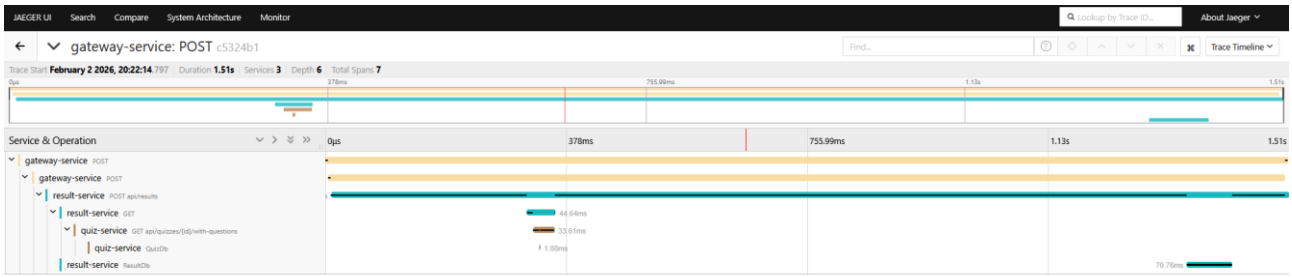
Слика 5.2.2.1. Grafana – Графички приказ метрика система у раду

5.2.3. Валидација дистрибуираног праћења

Тестирано је праћење захтева који пролазе кроз све сервисе у ланцу. Типичан сценарио укључује:

1. *Frontend* шаље захтев за решавање квиза
2. *Gateway* рутира захтев и проверава JWT токен
3. *Quiz Service* преузима питања из базе
4. *Result Service* чува резултат решавања

Jaeger интерфејс успешно приказује целокупан траг са свим спасовима и њиховим трајањима.



Слика 5.2.3.1. Jaeger – комплетан траг захтева кроз све сервисе

5.3. Тестирање перформанси под оптерећењем

5.3.1. Дефинисање сценарија оптерећења

Дефинисана су три сценарија за симулацију различитих нивоа оптерећења према очекиваним обрасцима коришћења:

Сценарио	Конкурентних корисника	Трајање	Опис
Нормално	50	5 мин	Типично дневно оптерећење
Високо	200	5 мин	Вршно оптерећење (испити)
Стрес	500	10 мин	Преоптерећење система

Табела 5.3.1.1. Дефиниције сценарија оптерећења

Скрипте за генерисање оптерећења имплементиране су у *Python*-у коришћењем асинхроног *aiohttp* клијента. Сваки виртуелни корисник извршава типичне операције: пријава, преузимање листе квизова, решавање квиза и чување резултата.

5.3.2. Резултати тестирања

Сценарио 1: Нормално оптерећење (50 конкурентних корисника):

Метрика	Вредност
Укупно захтева	15.000
Успешност	99.9%
Просечна латенција	45 ms
P95 латенција	89 ms

P99 латенција	156 ms
Пропусност	50 захтева/с

Табела 5.3.2.1. Резултати тестирања под нормалним оптерећењем

Grafana метрике током теста показују стабилну искоришћеност ресурса: CPU око 35%, меморија око 55%. Стопа грешака је минимална (0,1%), углавном услед спорадичних мрежних прекида.

Сценарио 2: Високо оптерећење (200 конкурентних корисника):

Метрика	Вредност
Укупно захтева	60.000
Успешност	99,2%
Просечна латенција	128 ms
P95 латенција	342 ms
P99 латенција	523 ms
Пропусност	198 захтева/с

Табела 5.3.2.2. Резултати тестирања под високим оптерећењем

При овом оптерећењу, искоришћеност CPU-а достиже 78%, а меморије 72%. Систем остаје стабилан, али се примећује благи пораст латенције. Стопа грешака од 0,8% и даље је у прихватљивим границама.

Сценарио 3: Стрес тест (500 конкурентних корисника):

Метрика	Вредност
Укупно захтева	150.000
Успешност	94%
Просечна латенција	456 ms
P95 латенција	1.234 ms

P99 латенција	2.567 ms
Пропусност	235 захтева/с

Табела 5.3.2.3. Резултати стрес теста

Стрес тест открива границе тренутне конфигурације. Искоришћеност CPU-а достиже 95%, активира се ***Horizontal Pod Autoscaler*** који покреће додатне реплике *Quiz Service*-а. Стопа грешака од 6% указује на потребу за додатним ресурсима при овом нивоу оптерећења.

5.4. Анализа резултата

5.4.1. Перформансе система

Резултати показују да систем успешно обрађује захтеве у нормалним условима са ниском латенцијом (P95 < 100 ms) и минималном стопом грешака (< 0,5%). При високом оптерећењу систем одржава прихватљиве перформансе уз благи пораст латенције.

Идентификована су следећа уска грла:

1. ***Quiz Service***: Показује највећу латенцију због сложених SQL упита са *JOIN* операцијама за преузимање квизова са питањима и одговорима
2. **Ресурси чворова**: При преко 200 конкурентних корисника, ограничени ресурси *Standard_B2s* инстанци постају фактор
3. **База података**: При интензивним *спасовима* операцијама (чување резултата) уочава се повећана латенција због коришћења мање спасовима базе података због уштеде ресурса на *Azure* платформи.

5.4.2. Ефективност посматрачке способности

Током тестирања, имплементирана посматрачка способност омогућила је:

- Детекцију проблема у реалном времену: *Grafana* контролне табле моментално су приказале пораст латенције и стопе грешака током стрес теста. Конфигурисани прагови упозорења би активирали аларме при прекорачењу дефинисаних вредности.
- Идентификацију узрока: *Jaeger* трагови јасно су показали да *Quiz Service* узрокује највеће кашњење у ланцу захтева. Детаљна анализа спанова открила је да SQL упит за преузимање квиза са питањима троши 60% укупног времена обраде.
- Корелацију између компоненти: Корелациони идентификатор омогућио је праћење проблематичних захтева кроз логове свих сервиса. Граф зависности сервиса у *Jaeger* интерфејсу визуализовао је утицај *Quiz Service*-а на остале компоненте.

5.5. Поређење са почетним стањем

Сума побољшања у опсервабилности система након имплементације посматрачке способности:

Аспект	Пре имплементације	После имплементације
Време детекције проблема	15–30 мин (ручно)	< 1 мин (аутоматски)
Дијагностика узрока	1–2 сата	5–15 минута
Корелација захтева	Немогућа	Потпуна (*correlation* ID)
Историја метрика	Не постоји	15 дана (*Prometheus*)
Историја логова	Губи се при рестарту	30 дана (*Elasticsearch*)

Табела 5.5.1. Поређење аспекта пре и после имплементације решења

6. Закључак

6.1. Резиме рада

Овај рад представио је свеобухватан приступ имплементацији посматрачке способности микросервисне архитектуре коришћењем *Kubernetes* платформе и савремених алата отвореног кода. Кроз практичну имплементацију на *QuizHub* платформи, демонстрирани су кључни концепти и технике за постизање видљивости у дистрибуираним системима.

Главни доприноси рада укључују:

1. **Систематски приступ посматрачкој способности** - Имплементација сва три стуба посматрачке способности (логови, метрике, трагови) у интегрисано решење које омогућава корелацију података из различитих извора
2. **Практична примена у Kubernetes окружењу** - Демонстрација интеграције алата за посматрање са Kubernetes екосистемом искоришћењем механизама као што су *DaemonSet* за агенте, аутоматско откривање сервиса и анотације
3. **Централизовано управљање логовима** - Имплементација ELK *Stack*-а за прикупљање, складиштење и анализу логова са структурираним форматом и корелацијом између сервиса
4. **Метрике апликације и инфраструктуре** - Интеграција *Prometheus*-а и *Grafana* за прикупљање и визуелизацију метрика са прилагођеним контролним таблама
5. **Дистрибуирано праћење** - Имплементација *Jaeger*-а са *OpenTelemetry* инструментацијом за праћење захтева кроз целокупан систем
6. **Инфраструктура као код** - Примена *Terraform*-а за декларативно провизионисање *Azure* инфраструктуре, обезбеђујући поновљивост и верзионисање.

6.2. Остварени циљеви

Сви постављени циљеви истраживања успешно су остварени:

Ц1. **Миграција на Kubernetes** - Систем је успешно мигриран на *Azure Kubernetes Service* са комплетном *Terraform* конфигурацијом за провизионисање инфраструктуре и *Kubernetes* манифестима за све компоненте.

Ц2. **Централизовано логовање** - ELK *Stack* је имплементиран и интегрисан са свим сервисима. *Filebeat* агенти прикупљају логове са свих чворова, *Elasticsearch* их индексира, а *Kibana* омогућава претрагу и визуелизацију.

Ц3. **Прикупљање метрика** - *Prometheus* успешно прикупља метрике са свих сервиса путем механизма аутоматског откривања. *Grafana* пружа визуелизацију кроз прилагођене контролне табле са кључним индикаторима здравља система.

Ц4. **Дистрибуирано праћење** — *Jaeger* са *OpenTelemetry* инструментацијом омогућава потпуно праћење захтева кроз све сервисе, укључујући визуелизацију зависности и идентификацију уских грла.

Ц5. **Евалуација решења** — Спроведено је систематско тестирање под различитим нивоима оптерећења које је валидира функционалност и демонстрирало вредност имплементираног решења.

6.3. Одговори на истраживачка питања

П1: Како ефикасно организовати посматрачку способност у Kubernetes окружењу?

Ефикасна организација захтева примену Kubernetes примитиван прилагођених специфичним потребама сваке компоненте. *DaemonSet* је оптималан избор за агенте попут *Filebeat*-а који морају да раде на сваком чвору. Анотације на подовима омогућавају декларативно означавање сервиса за прикупљање метрика без централне конфигурације. Централизовани namespace за алате за посматрање олакшава управљање и изолује инфраструктуру од апликација. Кључно је искоришћење Kubernetes мета података (ознаке, имена подова, namespace) за обогаћивање телеметријских података.

П2: Које су предности и изазови ELK *Stack*-а за микросервисне логове?

Предности: Моћна претрага комплетног текста са *Lucene* синтаксом, флексибилно индексирање са подршком за структуриране и не структуриране податке, богата визуелизација у *Kibana* интерфејсу, скалабилност хоризонталним проширењем *Elasticsearch* кластера [49], [50].

Изазови: Значајна потрошња ресурса — *Elasticsearch* захтева минимум 1 GB RAM-а чак и за минималне инсталације. Комплексност конфигурације за продукционо окружење укључује подешавање шар динга, репликације и ILM политика. Управљање животним циклусом индекса захтева пажљиво планирање ради балансирања трошкова складиштења и потреба за историјским подацима.

П3: Како интегрисати три стуба посматрачке способности?

Интеграција се постиже кроз неколико механизама. Корелациони идентификатори (*trace* ID, *span* ID) повезују логове са траговима — исти идентификатор се бележи у логовима и укључује у *Jaeger* спанове. *Grafana* служи као јединствена платформа која

агрегираних податке из *Prometheus*-а (метрике), *Elasticsearch*-а (логови) и *Jaeger*-а (трагови). Конзистентно означавање по сервису и окружењу омогућава филтрирање и корелацију у свим алатима.

П4: Који су мерљиви бенефити имплементираног решења?

Квантитативни бенефити укључују: смањење времена детекције проблема са 15–30 минута на мање од једног минута, смањење времена дијагностике са 1–2 сата на 5–15 минута, и омогућавање потпуне корелације захтева која претходно није била могућа. Квалитативни бенефити укључују проактивни приступ одржавању система уместо реактивног, боље разумевање понашања система и могућност капацитетом планирања заснованог на подацима.

6.4. Ограничења истраживања

Рад има следећа ограничења која треба узети у обзир при интерпретацији резултата:

1. Скала тестирања - Тестирање је извршено на *капацитетом* кластеру са ограниченим ресурсима (2 чвора, 8GB RAM). Продукциони системи са већим обимом података и саобраћаја захтевају додатно скалирање и оптимизацију, посебно *Elasticsearch* кластера.
2. Технолошки стек - Резултати су специфични за .NET технолошки стек и конкретну архитектуру *QuizHub*-а. Друге технологије (*Java, Od, Node.js*) захтевају одговарајуће библиотеке за инструментацију.
3. Финансијска анализа - Није детаљно анализирано финансијско оптерећење у продукционом окружењу. Трошкови *Azure* ресурса, посебно за *Elasticsearch* складиштење, могу бити значајни при већем обиму података.
4. Безбедност - Безбедносни аспекти (TLS енкрипција, RBAC, аутентификација за *Kibana* и *Grafana*) нису у фокусу овог рада. Продукциона имплементација захтева додатне безбедносне мере.

6.5. Правци будућег рада

На основу искустава из овог истраживања, идентификовано је неколико праваца за будући рад:

1. Аутоматско скалирање базирано на метрикама - Имплементација Kubernetes *Horizontal Pod Autoscaler*-а који користи прилагођене метрике из *Prometheus*-а (нпр. број захтева у реду) уместо само CPU/меморије.

2. Напредно упозоравање — Конфигурација *Od*-а за интелигентно груписање и рутирање аларма, интеграција са каналима обавештавања (*Od*, email, *Texams*).

3. Service exams интеграција — Истраживање интеграције са *Istio* или *Linkerd* решењима сервисних мрежа која пружају додатну телеметрију без модификације кода апликација.

4. Машинско учење за детекцију аномалија - Примена ML алгоритама на прикупљене метрике за аутоматску детекцију аномалија и предикцију проблема.

5. GitOps приступ - Имплементација *ArgoCD* или *Flux* алата за декларативно управљање Kubernetes конфигурацијом из *Git* репозиторијума.

6.6. Завршна реч

Посматрачка способност представља критичну компоненту модерних микросервисних архитектура. Како системи постају све сложенији и више дистрибуирани, традиционални приступи мониторингу базирани на провери доступности постају недовољни. Потребан је холистички приступ који комбинује логове, метрике и трагове у јединствену слику понашања система.

Овај рад демонстрирао је да комбинација *Kubernetes*-а, ELK *Stack*-а, *Prometheus*-а, *Grafana* и *Jaeger*-а пружа робустно и флексибилно решење за постизање потпуне видљивости у дистрибуираним системима. Сви коришћени алати су отвореног кода и имају активне заједнице, што обезбеђује дугорочну одрживост решења.

Иако имплементација захтева значајан почетни напор у погледу учења, конфигурације и интеграције, дугорочни бенефити у погледу смањења времена детекције и решавања проблема, као и бољег разумевања понашања система, оправдавају улагање. У ери када се од софтверских система очекује висока доступност и перформансе, посматрачка способност није луксуз већ неопходност.

Како микросервисне архитектуре настављају да еволуирају, тако ће се развијати и алати и праксе за њихово посматрање. *OpenTelemetry* стандард представља значајан корак ка унификацији телеметрије, а интеграција са екосистемом заснованим на облаку наставиће да се продубљује. Овај рад пружа солидну основу за разумевање и примену ових концепата у пракси.

7. Литература

[1] G. Blinowski, A. Ojdowska, and A. Przybyłek, “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,” *IEEE Access*, vol. 10, no. 2169–3536, 2022.

[2] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, “A Survey on Observability of Distributed Edge & Container-based Microservices,” *IEEE Access*, pp. 1–1, 2022.

[3] N. Dragoni *et al.*, “Microservices: Yesterday, Today, and Tomorrow,” *Present and Ulterior Software Engineering*, pp. 195–216, 2017.

[4] S. Baškarada, V. Nguyen, and A. Koronios, “Architecting Microservices: Practical Opportunities and Challenges,” *Journal of Computer Information Systems*, vol. 60, no. 5, pp. 1–9, Sep. 2018.

[5] M. Mohammad, “Resilient Microservices: A Systematic Review of Recovery Patterns, Strategies, and Evaluation Frameworks,” arXiv, Dec. 2025

[6] C. Sridharan, Distributed Systems Observability: A Guide to Building Robust Systems. Sebastopol, CA, USA: O’Reilly Media, 2018.

[7] Red Hat, “Logging Attributes — Supported Log Levels,” Red Hat Documentation, 2025.

[8] V. Legeza, A. Golubtsov, and B. Beyer, “Structured Logging: Crafting Useful Message Content”, vol. 44, no. 2, Summer 2019.

[9] “RED and USE Metrics for Monitoring and Observability,” BetterStack Community Guides, 2025.

[10] “Distributed Tracing in Microservices,” GeeksforGeeks.org, 2026.

[11] “What is W3C Trace Context?” Dynatrace Engineering Blog, 2025.

[12] B. H. Sigelman et al., “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure”, Google Technical Report, 2010.

[13] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Site Reliability Engineering: How Google Runs Production Systems, Sebastopol, CA, USA: O’Reilly Media, 2016.

[14] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, “Cloud monitoring: A survey,” Computer Networks, vol. 57, no. 9, pp. 2093–2115, 2013.

[15] W3C, “Trace Context Level 1,” World Wide Web Consortium, Recommendation, 2020.

[16] D. Taibi and V. Lenarduzzi, “On the definition of microservice bad smells,” IEEE Software, vol. 35, no. 3, pp. 56–62, 2018.

[17] P. Pahl, “Containerization and the PaaS Cloud,” IEEE Cloud Computing, vol. 2, no. 3, pp. 24–31, May–Jun. 2015.

[18] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” Linux Journal, no. 239, Mar. 2014.

[19] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An Updated Performance Comparison of Virtual Machines and Linux Containers,” IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015.

[20] Kubernetes Authors, “Kubernetes: Production-Grade Container Orchestration,” Cloud Native Computing Foundation, 2023.

[21] A. Verma et al., “Large-Scale Cluster Management at Google with Borg,” Proceedings of the European Conference on Computer Systems (EuroSys), 2015.

[22] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” Communications of the ACM, vol. 59, no. 5, pp. 50–57, 2016.

[23] Kubernetes Documentation, Kubernetes.io, 2023.

[24] Kubernetes Authors, “Configure Liveness, Readiness and Startup Probes,” Kubernetes.io, 2023.

[25] Kubernetes SIG Autoscaling, “Metrics Server,” Kubernetes.io, 2023.

[26] Kubernetes Authors, “Kubernetes Events, Labels, Annotations,” Kubernetes.io, 2023.

[27] Kubernetes Authors, “Horizontal Pod Autoscaling,” Kubernetes.io, 2023.

[28] Microsoft, “Azure Kubernetes Service (AKS) Documentation,” Microsoft Learn, 2023.

[29] Microsoft, “AKS Architecture and Best Practices,” Microsoft Azure Architecture Center, 2023.

[30] Elastic N.V., “The Elastic Stack: Elasticsearch, Kibana, Beats, and Logstash,” Elastic Documentation, 2023.

[31] Elastic N.V., “Elasticsearch: The Definitive Guide,” O’Reilly Media, 2015.

[32] Elastic N.V., “Elasticsearch Query DSL,” Elastic Documentation, 2023.

[33] Elastic N.V., “Filebeat Overview,” Elastic Documentation, 2023.

[34] Elastic N.V., “Kibana User Guide,” Elastic Documentation, 2023.

[35] Elastic N.V., “Index Lifecycle Management,” Elastic Documentation, 2023.

[36] Prometheus Authors, “Prometheus: Monitoring System & Time Series Database,” Prometheus.io, 2023.

[37] J. Turnbull, The Prometheus Monitoring System, O’Reilly Media, 2018.

[38] Prometheus Authors, “Service Discovery,” Prometheus Documentation, 2023.

[39] Prometheus Authors, “Querying Prometheus,” Prometheus Documentation, 2023.

[40] D. Kolpakov et al., “prometheus-net: .NET Client Library for Prometheus,” GitHub Repository Documentation, 2023.

[41] Grafana Labs, “Grafana Documentation,” Grafana Labs, 2023.

[42] A. Wiggins et al., “Monitoring Modern Distributed Systems,” IEEE Cloud Computing, vol. 7, no. 2, pp. 78–87, 2020.

[43] Y. Chen et al., “Jaeger: End-to-End Distributed Tracing,” Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, vol. 1, no. 4, 2017.

[44] OpenTelemetry Authors, “OpenTelemetry Documentation,” Cloud Native Computing Foundation, 2023.

[45] B. Sigelman et al., “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,” Communications of the ACM, vol. 54, no. 4, pp. 51–58, 2010.

[46] HashiCorp, Infrastructure as Code, HashiCorp Documentation.

[47] Terraform, Terraform Core Concepts and Workflow, HashiCorp.

[48] HashiCorp, Terraform Provider Registry Documentation.

[49] Microsoft, Azure Resource Manager and Virtual Network Documentation.

[50] Azure Kubernetes Service, AKS Architecture and Networking, Microsoft Learn.

[51] Azure Container Registry, Azure Container Registry Documentation, Microsoft.

[52] Azure Log Analytics, Azure Monitor and Container Insights, Microsoft.

[53] Helm, Helm Documentation;

[54] NGINX Ingress Controller, Ingress-NGINX Documentation.

[55] HashiCorp, Terraform Outputs and Automation Best Practices.

Биографија

Владислав Петковић је рођен 06.09.2002. год. у Новом Саду у Србији. Средњу електротехничку школу у Новом Саду завршио је маја 2021. године. Исте године уписао се на основе академске студије примењеног софтверског инжењерства на Факултету техничких наука. Положио је све испите предвиђене планом и програмом.