Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет ________________Комп’ютерних наук________________________
(повна назва)

Кафедра_________________Програмної інженерії_________________________
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти - другий (магістерський)

Дослідження методів та оптимізація процесу розробки додатків за допомогою мікросервісної архітектури
(тема)

Виконав: студент 6 курсу, групи ПЗСм-18-1__
___________Радченко О. А.______________
(прізвище, ініціали)

спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-професійної програми________
(тип програми)

___Програмне забезпечення систем___
(повна назва освітньої програми)

Керівник __доцент каф. програмної інженерії Чуприна А. С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____________________ З.В.Дудар

2019 р.

Харківський національний університет радіоелектроніки

Факультет_Комп’ютерних наук____

Кафедра_Програмної інженерії______________________________________

Рівень вищої освіти - другий (магістерський)

Спеціальність _121-Інженерія програмного забезпечення_________________
(код і повна назва)

Тип програми освітньо-професійна програма

Освітня програма_ Програмне забезпечення систем__________

ЗАТВЕРДЖУЮ:

Зав. кафедри ______________
(підпис)
«____»_____________ 20 ___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові _Радченку Олександру Андрійовичу____________________________
(прізвище, ім’я, по батькові)

1. Тема роботи _ Дослідження методів та оптимізація процесу розробки додатків за допомогою мікросервісної архітектури _______________________________

затверджена наказом університету від “___” _______20 ___ р № _____

заповнюється вручну після отримання наказу

2. Термін подання студентом роботи до екзаменаційної комісії
05 грудня 2019 р.

3. Вихідні дані до роботи *методи розробки мікросервісної архітектури, алгоритм перевтілювання монолітної архітектури на мікросервісну*______

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз проблемної галузі і постановка задачі, огляд методів створення додатку з сервісоорієнтованою архітектурою. Огляд методів перетворення додатку з монолітною архітектурою на додаток з мікросервісною архітектурою. Методи оптимізації та масштабування мікросервісної архітектури, безпека мікросервісної архітектури.*

5 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	доц. каф. ПІ Чуприна А.С.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка*
1.	Аналіз предметної галузі	19 вересня 2019р.	
2.	Огляд існуючих методів	27 вересня 2019р.	
3.	Методи швидкого детектування відрізків ліній		
4.	Підготовка пояснювальної записки	25 жовтня 2019р.	
5.	Спецчастина	26 жовтня 2019р.	
6.	Підготовка презентації та доповіді	28 жовтня 2019р.	
7.	Попередній захист	30 жовтня 2019р.	
8.	Нормоконтроль, рецензування	02 жовтня 2019р.	
9.	Занесення диплома в електронний архів	03 листопада 2019р.	
10.	Допуск до захисту у зав. кафедри	04 листопада 2019р.	
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання _____ _____________2019 р.

Студент _____________________________________
(підпис)

Керівник роботи ______________________ __доц. каф. ПІ Чуприна А.С._
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Атестаційна робота магістра містить: 69 с., 13 рис.,3 табл., 33 джер.

JAVA, REST, МІКРОСЕРВІСНА АРХІТЕКТУРА, MSA, МІКРОСЕРВІС, МОНОЛІТ SPRING, SOA, SOAP, MVC, APACHE TOMCAT.

Об'єктом дослідження являється міросервісна архітектура, її переваги та недоліки, обмеження її використання, загальна практика переходу з моноліту на мікросервісну архітектуру.

Метою роботи є розробка програмного забезпеченняз використання мікросервісної архітектури, виділення обмежень мікросервісної архітектури, розробка моделей оцінювання.

В ході виконання дипломної роботи була розроблена архітектура додатку з використанням мікросервісної архітектури, виявлені обмеження використання мікросервісної архітектури, розроблені моделі оцінювання. На базі архітектурирозроблено програмний додаток для демонстрування роботи мікросервісної архітектури.

JAVA, REST, MICROSERVICE ARCHITECTURE, MSA, MICROSERVICE, MONOLITH, SPRING, SOA, SOAP, MVC, APACHE TOMCAT.

The object of research is a microservice architecture and itsusage limitations.

The aim is to develop software which based on microservice architecture, highlight usage limitations and develop evaluation models.

In the course of implementation of diploma work was developed an app architecture based on microservices approach, highlighted microservice architecture usage limitation, developed evaluation models. Based on the obtained architecturethere was developed an application for displaying microservices architecture functionality.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 6
ВСТУП 7
1 ВИКОРИСТАННЯ МІКРОСЕРВІСІВ У ВЕБ-РОЗРОБЦІ 8
1.1 Загальні поняття та проблеми веб-розробки 8
1.2 Переваги використання мікросервісної архітектури 9
1.3 Обмеження використання мікросервісної архітектури 10
1.4 Існуючі аналоги, їх переваги та недоліки 13
1.5 Мікросервісна архітектура та типи комунікації мікросервісів 17
1.6 Якісна постановка задачі 21
2 ТЕОРЕТИЧНІ ОСНОВИ ОЦІНЮВАННЯ 23
2.1 Загальна постановка задачі 23
2.2 Методи оцінювання багатокритеріальних об’єктів 24
2.3 Опис методів, що використовуються 26
2.3.1 Метод ПАКС 26
2.3.2 Агрегування якісних ознак у просторі зниженої розмірності 31
2.3.3 Приклад агрегування критеріїв 34
3 ЕТАПИ РОЗРОБКИ ДОДАТКУ З ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ 36
3.1 Розбиття моноліту на частини 36
3.2 Розгортання мікросервісів та їх масштабування 37
3.2.1 Способи розгортання мікросервісної архітектури 38
3.2.2. Масштабування мікросервісів 41
3.3 Нефункціональні вимоги 44
3.4 Інструменти для побудови MSA 44
3.4.1 Єдина точка входу (API Gateway) 45
3.4.2 Виявлення сервісів (Service Discovery) 49
3.4.3 Автоматичний вимикач (Circuit Breaker) 49
3.4.4 Docker 50
3.5 Засоби розробки 52
3.5.1 Java 52
3.5.2 Spring framework 53
СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ 57
ДОДАТОК А 60
ДОДАТОК Б 67

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

MSA – MicroService Architecture (мікросервісна архітектура)

SOA – Service-Oriented Architecture (сервіс-орієнтована архітектура)

SOAP – Simple Object Access Protocol

API – Application Programming Interface, набір інтерфейсів для взаємодії різнотипного програмного забезпечення

JSON – JavaScript Object Notation, текстовий формат обміну даними

CI – Continuous Integration

CD – Continuous Delivery

REST – Representational State Transfer, архітектурний стиль

ООП – Об'єктно-орієнтоване програмування

AWS – Amazon Web Services, інфраструктура хмарних платформ

J2EE – Java Enterprise Edition;

OSGi – Open Service Gateway Initiative;

HTTPS – Hyper Text Transfer Protocol Secure, захищений HTTP

GWT – Google Web Toolkit, Java-фреймворк;

XML – розширювана мова розмітки.

ВСТУП

У сучасному інформаційному світі проблема проектування та створення якісного програмного забезпечення є надзвичайно важливою. Під час розвитку IT-індустрії було знайдено багато різних методів та способів будування складних програмних систем. Одним з показників гарно побудованої програми є її архітектура, якщо вона вірно описує предметну область та є формальною моделлю системи. Можна вважати архітектурою програмної системи набір певних структурних компонентів зв'язаних між собою, що задають поведінку усієї системи. Управління складністю та доцільне відображення предметної галузі - основна мета архітектури. Основною задачею архітектури є управління складністю, елегантне та доцільне відображення предметної області. Монолітна архітектура означає: що вся система фізично розташовується на єдиній машині, запускається в одному процесі та виконує всі бізнес-операції.

Монолітна система масштабується виключно горизонтально, шляхом запуску декількох окремих серверів із кожним окремим монолітом. З часом знаходилися інші ідеї та підходи, саме таким стала сервісоорієнтована архітектура (SOA), на відміну від монолітної системи, при SOA вся програма являє собою розподілену систему, яка обмінюється повідомленнями за певним протоколом. Вся система складається з набору незалежних сервісів, які фокусуються на власній задачі.

В цілому, мікросервісна архітектура не є єдиним вирішенням усіх проблем, а вирішує лише певний набір задач і залежить від різних обставин. Тому в даній роботі буде розглянуто, які обмеження висуваються до MSA та доцільність використання цієї архітектури.

1 ВИКОРИСТАННЯ МІКРОСЕРВІСІВ У ВЕБ-РОЗРОБЦІ

1.1 Загальні поняття та проблеми веб-розробки

Термін "мікро веб-сервіси" був спочатку використаний доктором Пітером Роджерсом під час конференції із хмарних обчислень в 2005. Самі мікросервіси були показані вперше на заході для архітекторів програмного забезпечення в 2011, де термін був використаний, щоб описати стиль архітектури, з якою експериментували багато відвідувачів в той час. Netflix і Amazon були серед ранніх піонерів мікросервісів.

Популярність мікросервісів нещодавно підвищувалася, тому що вони можуть вирішити багато поточних проблем IT, таких як збільшуюча швидкість, масштабованість програмних систем і швидких випробувальних процесів.

Мікросервісна архітектура, або просто мікросервіси, є окремим методом розробки систем програмного забезпечення, який зосереджується на будівництві модулів з єдиною функцією та з чітко визначеними інтерфейсами та операціями. Тенденція стала популярною в останні роки, оскільки підприємства сподіваються ставати більш гнучкими, розвивати DevOps та безперервне тестування.

Мікросервісний стиль архітектури розробляє складне прикладне програмне забезпечення з маленьких, окремих програм, які спілкуються один з одним використовуючи незалежні від мови програмування інтерфейси (API). Компанії стикаються з проблемою, у випадку коли вони не здатні виміряти монолітну архітектуру, яка розвивалася з часом, коли їхню архітектуру важко модернізувати, або обслуговування стає занадто складним. Мікросервіси можуть бути відповіддю на цю проблему, оскільки вони поділяють складні завдання на менші процеси, які працюють один незалежно від одного.

1.2 Переваги використання мікросервісної архітектури

Мікросервісна архітектура має деякі переваги порівнюючи з монолітною. Головною метою даного стилю, як і будь-якої архітектури, це управління складністю. Ось основні переваги мікросервісної архітектури приведені нижче.

Перший це простота. Програми стає легше побудувати і підтримати, коли вони розділені на ряд менших, складових фрагментів. Управління кодом також стає легшим, бо кожний мікросервіс, насправді, окрема частина кодексу. Сервіси можуть бути здійснені, використовуючи різні мови програмування, бази даних і навколишнє середовище програмного забезпечення. Це дозволяє кожному сервісу бути розгорнутим, відновленим, повторно розгорнутим і організованим незалежно. Наприклад, якщо мікросервіс потребує занадто багато пам 'яті або завантажує процесор, це торкнеться тільки цей сервіс. Взагалі кажучи, будь-яка проблема з мікросервісами не буде впливати на всю систему, і відмова окремих мікросервісів може бути компенсована відносно швидко. Також, це дозволяє вводити кожен мікросервіс в експлуатацію легко один за іншим.

Зосередження на бізнес-функціоналі. Мартін Фаулер підкреслює, що мікросервіси дозволяють будувати продукт замість проектів. Дійсно, мікросервісна архітектура рекомендує командам зосереджуватися на створенні бізнес-функціональності замість того, щоб писати проміжний код. Іншими словами, групи розробників зосереджені на функціоналі, а не технологіях. Це означає, що сервіси пристосовані до використання в декількох контекстах. Ті ж самі сервіси можуть бути знову використані більше ніж в одному бізнес-процесі залежно від потреби. Кожен член команди відповідальний за конкретний сервіс, що призводить до будування розумної, багатопрофільної команди.

Покращена продуктивність і швидкість. Мікросервісна архітектура вирішує проблему продуктивності і швидкості, розкладаючи програми на керовані сервіси, які швидко розвиваються. Різні команди можуть працювати над різними компонентами одночасно, не маючи необхідність чекати одна одну. Окремі

мікросервіси легше визначити змінювати та лагодити. Цей тип архітектури дуже зручний для прискорення гарантії якості, оскільки кожний сервіс можна перевірити індивідуально, і можна протестувати компоненти, які були вже розроблені, в той час як програмісти працюють над іншими.

Масштабованість і гнучкість. Кожний мікросервіс може бути написан, використовуючи різну технологію. Це спрощує вибір відповідної технології для певних потреб вашого сервісу. Мікросервісна архітектура дозволяє розчепленим сервісам, написаним на різних мовах програмування, мирно співіснувати з іншими фрагментами. Це також гарна новина, якщо ви сподіваєтесь масштабувати своє рішення в майбутньому. З мікросервісами ви можете додати нові компоненти до системи безболісно або масштабувати сервіси окремо один від одного.
Автономні, багатопрофільні команди. Мікросервіси дуже зручні для розподілених команд. Розвиток великої системи моноліту може бути складним, якщо ви працюєте з підрозділами у всьому світі. Мікросервіси надають розробникам більше незалежності, щоб працювати автономно і приймати технічні рішення швидко в менших групах. Так, якщо ви очікуєте, що ваше програмне рішення буде громіздким, слід розглянути мікросервісну архітектуру.

1.3 Обмеження використання мікросервісної архітектури

Одні команди розробників знаходять в мікросервісной архітектурі переваги перед монолітною. Інші ж вважають мікросервіси зайвим навантаженням, знижує продуктивність. Як і кожен архітектурний стиль, мікросервіси несуть в собі і переваги, і недоліки. Розглянемо які можуть зустрітися обмеження використання мікросервісної архітектури.

Найбільший недолік мікросервісної архітектури - збільшена складність. Складність заснованого на мікросервісаї застосування безпосередньо корелює з кількістю залучених сервісів. У цього типу архітектури є набагато більше рухомих

частин, ніж у традиційних програмних систем, що вимає додаткового зусилля, планування, і автоматизації, щоб контролювати міжсервісне спілкування, моніторинг, тестуванням та розгортання.

Ініціатива будування програмної системи мікросервісного типу вимагатиме структурної зміни в команді. Вона вимагає зрілої та гнучкої культури DevOps. Із заснованою на мікросервісах програмою командам потрібно вміти управляти всім життєвим циклом усіх сервісів. Це часто вимагає мігруючих компетенцій та прийняття різних рішень від менеджерів і архітекторів окремих команд. Ця зміна в ієрархії може бути важкою для компанії. Тому, гарантія втягнення досвідчених учасників і верхній частини управління персоналом в ініціативу це важливий перший крок. Крім того, зв 'язок між людьми і командами стає набагато більш складним, оскільки у команд може не бути видимості усієї картини і того, як окремі послуги повинні працювати один з одним, щоб створити повноцінне програмне забезпечення. Організація повинна буде також визначити, чи володіють їх люди навичками і мають необхідний досвід, щоб розробляти засновану на мікросервісах програмну систему. Оскільки команда може бути відповідальною за один сервіс, розробники повинні бути добре обізнані про розвиток, розгортання, тестування та контроль застосування. Останньою вимогою буде мати хоча б одного DevOps розробника з відповідними навичками на команду.

Серед інших недоліків мікросервісів - витрати. Сервіси повинні будуть спілкуватися один з одним, наводячи до великої кількості запросів. Ці віддалені запроси призводять до більш високих цін, пов 'язаних з мережевим часом очікування і обробки, порівняно з традиційною архітектурою. Розробники захочуть прикласти максимальних зусиль, щоб скоротити кількість запросів. Інший фактор збільшеної вартості - більш високий попит ресурсу, оскільки кожний сервіс вимагатиме свого власного середовища і центрального процесора. Це - вимога, щоб зберегти кожен сервіс ізольованим. Крім того, через кожний сервіс, що використовує його власну мову і технологіЇ, розробка додатку та неоднорідність архітектури можуть збільшити ресурси, які організація витрачає на управління і обслуговування.

Мікросервіси ставлять проблеми безпеки. Мікросервіси можуть поставити величезні проблеми безпеки через збільшень комунікації міжсервісне спілкування по мережі. Всі ці взаємодії створюють можливість зовнішнім елементам отримати доступ до системи.

Мікросервіси можуть стати причиною низької продуктивності. Якщо сервіс викликає п'ять віддалених сервісів, кожен з яких викликає ще п'ять віддалених сервісів, то загальний час затримки може стати надмірно великим. Для вирішення цієї проблеми було придумано декілька способів. Можна робити менше віддалених викликів, збільшивши їх деталізацію, але це ускладнює модель програмування, так як тепер потрібно думати про те, як об'єднувати взаємодії між сервісами. У цього підходу теж є ліміт - кожен сервіс все одно потрібно буде викликати хоча б один раз. Ще одним варіантом вирішення буде асинхронність. Якщо зробити декілька асинхронних викликів одночасно, то загальний час очікування буде визначатися самим повільним викликом, а не сумою всіх. Це може дати великий виграш в продуктивності, але тягне за собою іншу проблему - асинхронність складно програмувати і ще важче налагоджувати.

Ще одна проблема при будування програмної системи мікросервісного типу - надійність. Очікується, що внутрішні виклики завжди будуть працювати, але віддалений виклик може відмовити в будь-який момент. З ростом кількості мікросервісів потенційних точок відмови стає більше. Розробники намагаються проектувати систему з урахуванням відмов. Існують техніки, які потрібні для реалізації асинхронного взаємодії, вони добре підходять для роботи з відмовами, підвищуючи відмовостійкість. Однак, це не повна компенсація, тому що як і раніше залишаються додаткові складності з виявлення наслідків відмови для кожного зовнішнього виклику. І це тільки дві основні проблеми розподілених обчислень. По-перше, озвучені проблеми зростають в монолітних системах по мірі їх зростання. Мало монолітних систем дійсно самодостатніх. Як правило, існують інші системи, з якими доводиться взаємодіяти. Взаємодія з ними відбувається по мережі і ми стикаємося з тими ж проблемами. Саме тому команди, які працюють з віддаленими системами, схильні швидше переходити на мікросервіси. У цьому

питанні сильно допомагає досвід - більш вмілі команди краще справляються з проблемами розподіленності.

Мікросервіси схильні до проблеми неузгодженості. У монолітної системі можуть бути оновленими безліч об'єктів в одній транзакції. З мікросервісами буде потрібно кілька ресурсів для оновлення та розподілені транзакції не схвалюються (з вагомих причин). Як результат, розробники повинні пам'ятати про можливість неузгодженості і думати про те, як визначати моменти рассинхронізації, перш ніж програмувати щось, що потрібно буде покращувати з часом.

Один з недоліків - складний онбордінг. Існує точка зору, що оскільки кожен сервіс невеликий, його простіше зрозуміти. Але небезпека полягає в тому, що складність не зникає, вона просто зміщується на взаємозв'язку між сервісами. Це може маскувати збільшення складності експлуатації, наприклад, труднощі при налагодженні функцій, які включають кілька сервісів. Якісний вибір кордонів сервісів зменшує цю проблему, інший - робить її набагато складніше.

1.4 Існуючі аналоги, їх переваги та недоліки

Існує декілька підходів до створення програмних систем, але усі можна поділити на два типа архітектури:

- багатошарова монолітна архітектура
- розподілена сервіс-орієнтована архітектура

Кожен з типів має свої недоліки та переваги. Багатошаровість - давня ідея побудови програмної системи, що зараз є найпопулярнішою. Багатошаровість означає, що система умовно поділяється на окремі частини (зазвичай на три), кожна з яких виконує свою функцію. На рисунку 1.1 зображена тришарова архітектура.

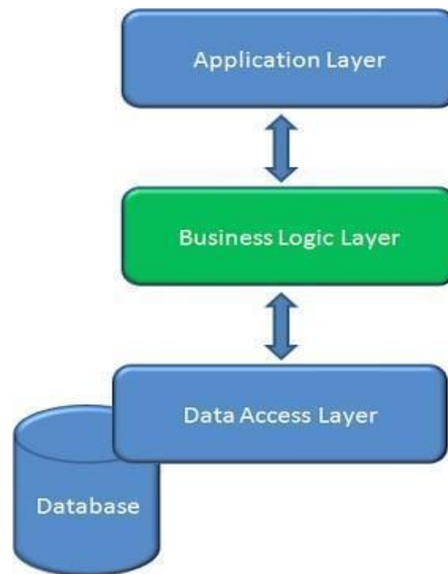

Рисунок 1.1 – Приклад класичної тришарової архітектури

Нижче приведені основні функції кожного з модулів.

В основі системи лежить перший шар, відповідний за доступ до даних.

Він надає інтерфейс для доступу до джерел даних для наступного шару нашої програмної системи.

Наступний шар відповідає за бізнес-логіку. Він містить алгоритми, які задають предметну область. Дана частина системи є найбільш гнучкою, тому що саме вона буде змінюватися в майбутньому більш за все. Використовує API доступу до даних, щоб реалізувати свої прикладні методи.

Останній рівень - рівень представлення. Цей рівень відповідає за взаємодію з користувачем та є точкою входу даних у програму.

Іншим способом для створення високонавантажених систем вважають сервісорієнтовану архітектуру (SOA), яка базується на ідеї розділення однієї програми, що виконується на одному фізичному комп’ютері, на окремі сервіси, що взаємодіючи один з одним через стандартизовані протоколи та інтерфейси обмінюються даними.

Оскільки кожен сервіс незалежний від інших, він може виконуватися взагалі на іншій машині, а взаємодія з іншими мікросервісами здійснюється по мережі. В найпростішому випадку існує три основних елементи: постачальник сервісу,

споживач сервісу та реєстр сервісів. Схема взаємодії наведена на рисунку 1.2. Взаємодія між окремими модулями буде мати такий вид:

Сервіси спілкуються за допомогою стандартизованого протоколу SOAP, постачальник сервісу зберігає сервіси до реєстру, споживач сервісу звертається до реєстру з запитом. Взаємодія між даними елементами виглядає наступним чином: постачальник сервісу реєструє свої сервіси, а споживач звертається до реєстру із запитом. Спілкування відбувається за певним уніфікованим протоколом передачі даних(SOAP, XML).

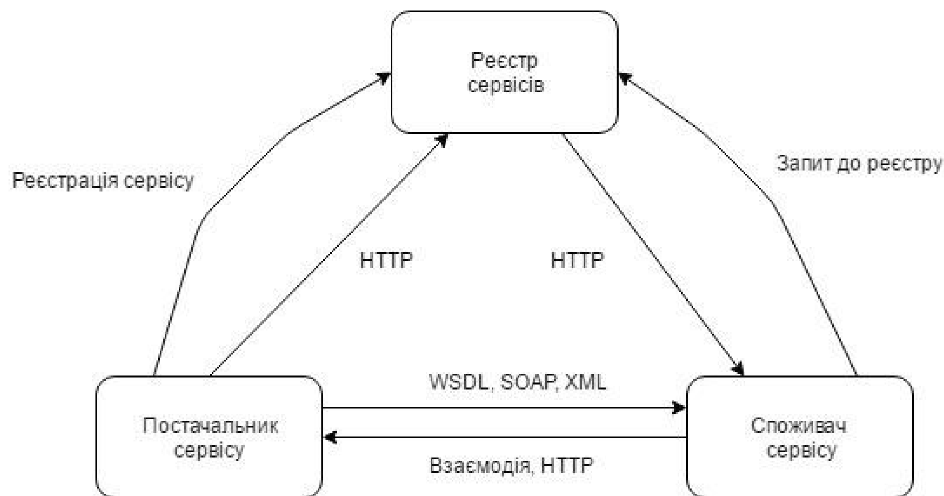

Рисунок 1.2 – Базова схема SOA

SOA базується на інтерфейсах. Вони ніколи не мають залежати від реалізації, які можуть визначатися різного роду фактором, такими як мова програмування чи операційна система [4].

Таблиця 1.1 – Основні переваги трирівневої архітектури та SOA

Трирівнева архітектура	SOA
Простота	Часткове розгортання
Узгодженість	Відмовостійкість
Міжмодульний рефакторинг	Відсутність стану
	Гетерогенність

Розглянемо переваги кожної з вище описаних архітектурних стилів, які представлені в таблиці 1.1

Для монолітної трирівневої архітектури:

- простота. Даний архітектурний підхід є простішим в реалізації, управлінні та розгортанні.
- узгодженість. Для монолітного додатку простіше слідкувати за узгодженістю коду, опрацьовувати програмні помилки.
- міжмодульний рефакторинг. Єдиний кодовий репозиторій та цілісність структури полегшує роботу в ситуаціях, коли декілька модулів повинні взаємодіяти між собою або у випадку, коли необхідно перемістити деяку програмну логіку з одного модуля в інший. У сервісоорієнтованому випадку, ми чітко обмежені границями модулів.

Таким чином, сервіс-орієнтована архітектура володіє необхідною гнучкістю, яку вимагають бізнес-процеси.

Перейдемо до основних переваг SOA:

- відсутність стану. SOA гарантує відсутність спільних станів між модулями.
- відмовостійкість. Коли один з сервісив дає відмову, уся система продовжує роботу.
- часткове розгортання. Розгортання всієї системи складається з розгортання окремих сервісів-модулів, які є незалежними. У випадку монолітного додатку, при будь-якій зміні проекту – нам необхідно перезбирати та знову розгортати всю систему.
- гетерогенність. Незалежність та стандартизованість інтерфейсів сервісів та протоколів передачі даних дозволяють створювати сервіси використовуючи різні мови програмування та різні фреймворки.

Ідеї SOA розвинув, так званий, мікросервісний підхід. В цілому, SOA представляє гарні рішення, але не існує чітких правил та настанов як саме правильно досягти успіху в сервіс-орієнтованій архітектурі. Найбільші недоліки

SOA – це проблема визначення місць розділу системи на окремі частини, проблема продуктивної взаємодії окремих незалежних частин використовуючи один протокол (SOAP).

1.5 Мікросервісна архітектура та типи комунікації мікросервісів

Архітектурний стиль мікросервісів (MicroServices Architecture, MSA) – це підхід, при якому вся система являє собою набір невеликих сервісів, кожен з яких розгортається та працює в окремому процесі та взаємодіє з іншими використовуючи легкі механізми, як правило HTTP. Сервіси побудовані так щоб задовольняти потреби бізнесу, використовуючи при цьому різні технології та мови програмування, фреймворки, бази даних тощо.

Мікросервісна архітектура взагалі є підмножиною типу SOA, що показано на рисунку 1.3. Мікросервіси розділені та відносно незалежні один від одного, так само як і SOA, використовує протоколи обміну даними. Мікросервісну архітектуру слід вважати підходом до реалізації сервісо-орієнотованої архітектури.

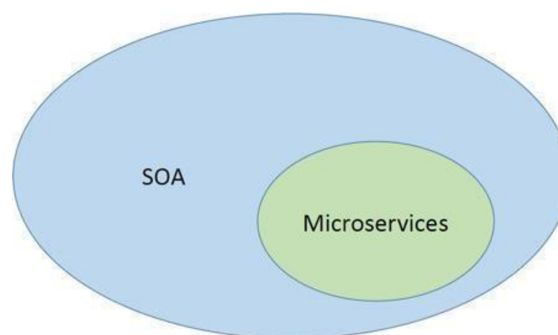

Рисунок 1.3 – Взаємовідношення MSA та SOA

Зазвичай один мікросервіс відповідає за одну проблему яку він вирішує. Це правило гарно корелює з принципом єдиної відповідальності, дотримуватися до якого є гарною практикою при розробці будь-якої програмної системи [5].

Кожний сервіс має використовувати та надавати інтерфейси і також використовувати впровадження залежностей (Dependency Injection), що дозволяє змінювати одну частину незалежно від іншого. Сервіс також повинен мати усі інструменти (методи) для вирішення своєї проблеми.

Найважливіша частина програмної системи - організація даних. У мікросервісній архітектурі зазвичай використовується децентралізоване управління даними, як це зображено на рисунку 1.4. Монолітна система користується однією базою даних, що обслуговує усі компоненти програми.

В MSA кожний сервіс може мати свою базу даних, до якої інші сервіси не мають доступу. Будь-які дані елемента доступні іншим тільки через прикладний інтерфейс. Завдяки цьому можна використовувати у системі різні технології та фреймворки для роботи з даними, наприклад нереляційну бази поруч із реляційною. Даний метод управління даними називають Polyglot Persistence.

Підхід децентралізації даних серед різних мікросервісів змінює саму структуру даних, та їх оновлення. Зазвичай для того, щоб забезпечити консистентність даних, використовуються транзакції при кожній зміні. Цей спосіб набув поширення серед монолітних програмних систем, тому що він гарантує виконання правил ACID (Atomicity, Consistency, Isolation, Durability. Атомарність, ізольованість, тривалість та узгодженість).

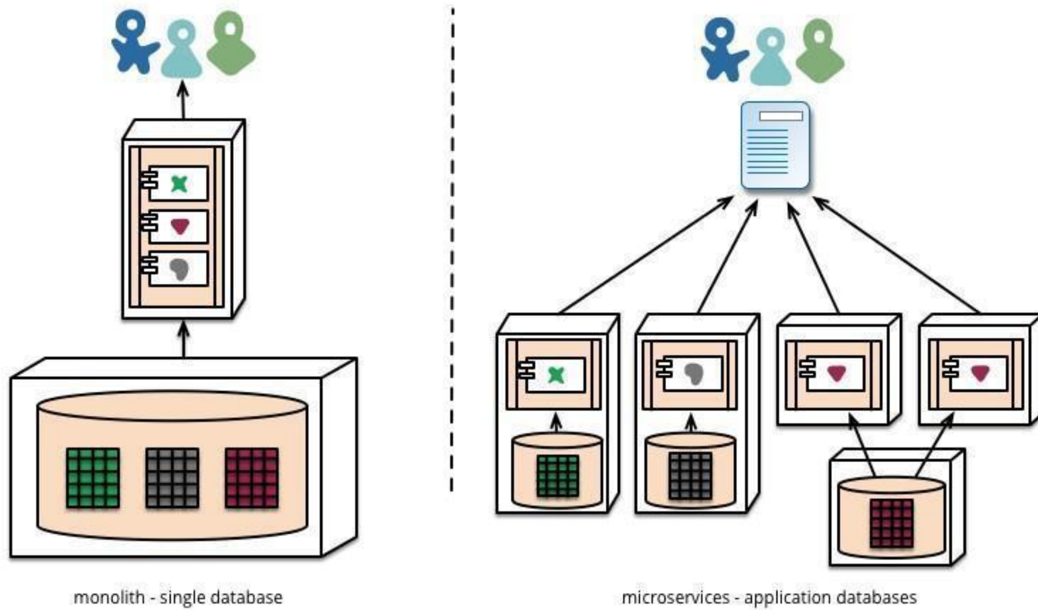

Рисунок 1.4 – Порівняння підходів до управління даними [6]

На відміну від моноліта, втілення транзакцій у проект з сервісоорієнтованою архітектурою є не таким тривіальним. Щоб реалізувати транзакційний механізм використовують принцип події. Тоді мікросервіс чекає на змін стану та генерує подію, на яку інші мікросервіси, в свою чергу, можуть бути підписані. Для них ця подія буде приводом для створення нової події. І так далі по ланцюжку, доки весь необхідний функціонал не буде виконано. Архітектура заснована на подіях також має бути узгодженою (eventual consistency) в кінці "ланцюжку подій" усіх сервісів. Система може бути в неузгодженому стані деякий час, поки всі події не будуть опрацьовані. Дизайн заснований на подіях має свої недоліки, оскільки, незважаючи на те, що дані в результаті узгоджені, система повинна мати справу з суперечливими даними. Крім цього, можлива ситуація повторення однієї й теж самої події, яка має бути знайдена та, найчастіше, ігнорована.

Ще одна проблема організації мікросервісної архітектури - запит до декількох модулів "одночасно". На даний момент існує два типа комунікації сервісів між собою: синхронний (REST, RPC), асинхронний.

Нижче розглянуто синхронний тип обміну повідомленнями на прикладі REST.

REST (Representational State Transfer, передача репрезентативного стану) - клієнт-серверний архітектурний стиль, що базується на HTTP протоколі, який він використовує як платформу для CRUD операцій між клієнтом і сервером.

Дуже важливим у стилі REST є поняття ресурсів. Для сервісу ресурс - сутність, до якої він має доступ, наприклад сутність “User”, що відповідає за користувача. Клієнт намагається отримати JSON користувача, щоб мати представлення цього об’єкта. Після того як клієнт отримав представлення, він має можливість запитувати його зміну або інколи навіть видалення. Отримавши запит на зміну, сервер має вибір опрацьовувати його чи ні.

HTTP має декілька властивостей, які дуже допомагають програмістам при створенні RESTful програмної системи. HTTP має свої власні методи (GET, PUT, POST, DELETE, UPDATE тощо), які REST використовує для реалізації механізму читання, зміни та видалення даних CRUD (Create, Read, Update, Delete). GET читає ресурс, POST зазвичай створює новий, UPDATE і DELETE змінюють та видаляють ресурс відповідно.

Щодо асинхронної комунікації між сервісами на основі механізму подій, то тут можна виділити дві задачі. Перша - як саме надати сервісам інформацію про подію і як саме слухачу цієї події визначити час її настання. Цими проблемами зазвичай займаються брокери повідомлень, наприклад RabbitMQ.

Використовуючи API постачальники генерують події для брокера, який, в свою чергу, опрацьовує підписки та доставляє інформацію до споживача. Інколи брокери повідомлень можуть опрацьовувати стан споживача.

Декілька постачальників повідомлень можуть відправляти їх в одну й ту саму чергу. Декілька споживачів можуть отримувати повідомлення з однієї черги. Таких споживачів повідомлень ще називають підписник. Підписник очікує на повідомлення.

Такі програмні системи розробляються гнучкими, з великими можливостями їх змінити масштабувати. Хоча розгортання системи може бути дещо складніше, асинхронний спосіб обміну повідомлень між сервісами часто є ефективним.

1.6 Якісна постановка задачі

При використанні сервісоорієнтованої архітектури (SOA), на відміну від монолітної системи, вся програма розділена на сервіси, що комунікують один з одним за протоколом, та які відповідальні за власне коло задач. Використання SOA може надати певні переваги, але питання правильної та якісної організації сервіс-орієнтованої архітектури досить складне.

Об’єктивно немає обмежень на кількість мікросервісів у системі. Кожен сервіс має бути відповідальним за одну бізнес - задачу. Для комунікування сервіси використовують уніфіковані протоколи передачі даних. Зазвичай кожний мікросервіс в архітектурі має своє API, за допомогою якого інші мікросервіси можуть з ним спілкуватися. Всі модулі в програмній системі можуть бути розроблені за допомогою різних технологій, мов програмування та фреймворків. Дані у системі децентралізовані, тобто кожний модуль має свою базу даних.

Переваги MSA такі:

- нескладний процес розгортання, бо кожний компонент розгортається незалежно
- надійність. Якщо один з компонентів не працює, інші можуть працювати.
- висока зчепленість коду в окремих сервісах.
- низька зв’язність між окремими сервісами.
- можливість використання різних технологій на різних модулях
- невеликі групи програмістів, що відповідають за власний мікросервіс.
- легке горизонтальне масштабування окремих сервісів і вертикальне масштабування системи в цілому.

Недоліки MSA такі:

- додатковий шар брокера повідомлень збільшує часові витрати як при розробці, так і при очікуванні відповіді від серверу.
- задача версіонування стає складнішою.

— задача інтеграційного тестування стає складнішою.
— складність розробки напряму корелює з кількістю технологій, що використовуються при розробці кожного з сервісів.

MSA стає дійсно зручною, якщо система буде розгортатися в хмарі. Моноліт майже не масштабується горизонтально.

Метою даної роботи є дослідження особливостей побудови додатків на базі мікросервісів, огляд існуючих платформ для створення мікросервісів, їх порівняння, видідення обмежень та розробка моделей їх оцінювання.

З урахуванням актуальності теми і загального стану проблеми необхідно вирішити наступні задачі:

— оглянути літературні джерела щодо використовування мікросервісної архітектури, розглянути деякі існуючі програмні рішення, виділити їх особливості, переваги та недоліки;
— сформулювати постановку задачі дослідження;
— розробити бізнес-процеси та бізнес-правила;
— сформувати функціональні та нефункціональні вимоги для програмного рішення, що розробляється;
— виділити обмеження використання мікросервісної архітектури та розробити до них відповідні моделі оцінювання;
— розробити моделі оцінювання обмежень використання мікросервісної архітектури.

2 ТЕОРЕТИЧНІ ОСНОВИ ОЦІНЮВАННЯ

2.1 Загальна постановка задачі

Задача багатокритеріального вибору формується наступним чином. Дається множина варіантів $A_1, \ldots, A_p A_1, \ldots, A_p$, кожен з яких характеризується певними критеріями $K_1, \ldots, K_m K_1, \ldots, K_m$. Кожен критерій $K_i K_i$ має шкалу $X_i = \{x_i^1, \ldots, x_i^{g_i}\}, i = 1,..m X_i = \{x_i^1, \ldots, x_i^{g_i}\}, i = 1,..m$, дискретні числові або вербальні градації в якій у більшості випадків є впорядкованими. Необхідно, на основі вподобань особи, яка приймає рішення, обрати один або декілька найкращих варіантів із множини представлених.

Основною складністю є те, що необхідно одночасно аналізувати як кількісні, так і якісні показники різного ступеня важливості, велика кількість яких сильно ускладнює порівняння та впорядкування веб-сайтів. Додатковою обставиною є те, що для оцінки якості веб-сайту не існує єдиної моделі оцінки якості.

У тому разі, коли проводиться оцінювання багатьох об’єктів з багатьма властивостями, виникає проблема, яка полягає у тому, що порівняння лише за одними значеннями атрибутів стає неможливим, а спроби зменшити кількість критеріїв оцінювання призводить до зниження якості остаточного результату через його віддалення від дійсності.

Ця проблема вимагає знайти спосіб, який забезпечив би вирішення задачі багатокритеріального вибору у просторі великої розмірності шляхом скорочення кількості вимірів, спираючись на правила предметної області та специфіку порівнюваних об’єктів. Скорочення кількості вимірів буде використано для агрегування декількох критеріїв до одного з отриманням шкали вимірювання с градацією, яка залежить від уподобань особи, що приймає рішення.

2.2 Методи оцінювання багатокритеріальних об'єктів

Вирішення подібної задачі можна побудувати на основі задачі пошуку екстремуму однієї або багатьох функції корисності(цінності) [11]. Для виконання задачі необхідно вивести узагальнений критерій із числа багатьох числових критеріїв шляхом їх згортання та знаходження зваженої суми. При великій кількості критеріїв такий метод є занадто трудомістким, тому що потребує від аналітика предметної області великих витрат часу на виведення приблизної функції корисності, а також коефіцієнтів важливості(ваги), які необхідно призначити кожній взятій до уваги властивості, що само по собі є задачею з неоднозначним рішенням. Також недоліком подібного методу є те, що використання агрегованих показників не дозволяє відтворити вхідні дані, з чого витікає неможливість легкого пояснення результатів порівняння [12].

Використання огрублених множин у класифікації багатокритеріальних об'єктів полягає у використанні наборів правил, що визначаються ОПР для віднесення альтернатив до певного класу з різним ступенем точності. Метод є достатньо складним, тому що велике число правил класифікації ускладнює їх аналіз. Крім того, метод потребує попереднє налагодження на підготовлених масивах даних [13].

Для впорядкування об'єктів в цілому або за багатьма критеріями часто використовують методи, що засновані на парному порівнянні об'єктів.

Повне впорядкування об'єктів відбувається, якщо можливо порівняти усі пари варіантів та вподобання ОПР є транзитивними. Якщо деякі із пари неможливо порівняти, то буде отримано часткове упорядкування. В методах аналітичної ієрархії [13] варіанти впорядковуються за показником їх пріоритетності, який послідовно обчислюється шляхом попарного порівняння варіантів , критеріїв їх оцінки та учасників по відношенню до глобальної цілі задачі, що вирішується. Недоліком є чутливість до контексту вибору, що призводить до кардинальної зміни впорядкування після додавання/виключення певного варіанту.

Існують наступні методи порівняння:

- пряме сортування об'єктів за заданими класами є найбільш популярними методом класифікації унаслідок простоти використання. Кожному об'єктові присвоюється певний клас в залежності від його оцінки за єдиним числовим критерієм;
- інтерактивна процедура класифікації передбачає опис вподобань ОПР через функцію корисності, що є зваженою сумою багатьох скалярних критеріїв [14].

Незалежно від способу отримання інформації від ОПР відомі методи прийняття рішень незадовільно працюють у великому просторі властивостей. Із цим пов'язані наступні проблеми:

- велика трудомісткість роботи ОПР з побудови функції корисності, знаходження ваг критеріїв, попарному порівнянні варіантів;
- складність інтерпретації результатів через неможливість відновлення вихідних даних на основі агрегованих критеріїв;
- необґрунтованість процедур переводу вербальних вимірів у числові оцінки.

Для подолання цих недоліків та спрощення процедури порівняння багатокритеріальних об'єктів за їх властивостями, необхідно обрати відповідний метод агрегації великого числа критеріїв у невелике число нових критеріїв із власними шкалами градації оцінок, які відображають вподобання ОПР.

Враховуючи слабку структурованість задачі, можна звернутися до методології вербального аналізу рішень. Згідно цієї методології, властивості варіантів описуються за допомогою якісних критеріїв, які мають словесні формулювання градацій на шкалах оцінок [15]. В такому випадку числові коефіцієнти важливості критеріїв та цінності варіантів не розраховуються та не використовуються, а вербальні оцінки ознак не перетворюються у числові показники. Таким чином, використовуючи тільки якісні виміри, на множині кортежів багатокритеріальних оцінок задаються відношення верховенства та еквівалентності варіантів рішення, за допомогою яких можна здійснити вибір кращого варіанту.

2.3 Опис методів, що використовуються

2.3.1 Метод ПАКС

Для рішення задачі багатокритеріального вибору був обраний метод ПАКС. Даний метод характеризується використанням методів вербального аналізу для зменшення розмірності простору критеріїв. Метод був обраний завдяки тому, що оцінювання за ієрархічною системою складених якісних критеріїв дозволить отримати змістовну та зрозумілу оцінку з найменшими витратами часу на побудову системи оцінювання для ОПР.

Процедура рішення методом ПАКС має у складі три етапи. Першим етапом є побудова ієрархічної системи агрегованих критеріїв методом ИСКРА із врахуванням переконань ОПР. Процес побудови полягає у створенні інтегральних показників, які характеризують обрані на основі понять предметної області властивості варіантів, які агрегують початкові характеристики. Процедура агрегування показників носить послідовний характер, тобто отримані групи критеріїв об'єднуються послідовно в нові групи наступного рівня ієрархії і так далі аж до єдиного інтегрального критерію найвищого рівня, якщо це необхідно.

На рисунку 2.1 зображена діаграма діяльності рішення задачі багатокритеріального вибору з послідовним зниженням розмірності простору ознак.

Кроки методу:

- вибрати тип завдання T. Можливі наступні завдання: T1 - вибрати кращий варіант; T2 - упорядкувати варіанти; T3 – розділити варіанти на впорядковані групи;
- сформувати безліч $A_1, \dots, A_p$ $A_1, \dots, A_p$ варіантів, p≥2 рішення задачі T;

— сформувати безліч вихідних показників $K_1,\ldots,K_m$ $K_1,\ldots,K_m$ (ознак) , m≥2;

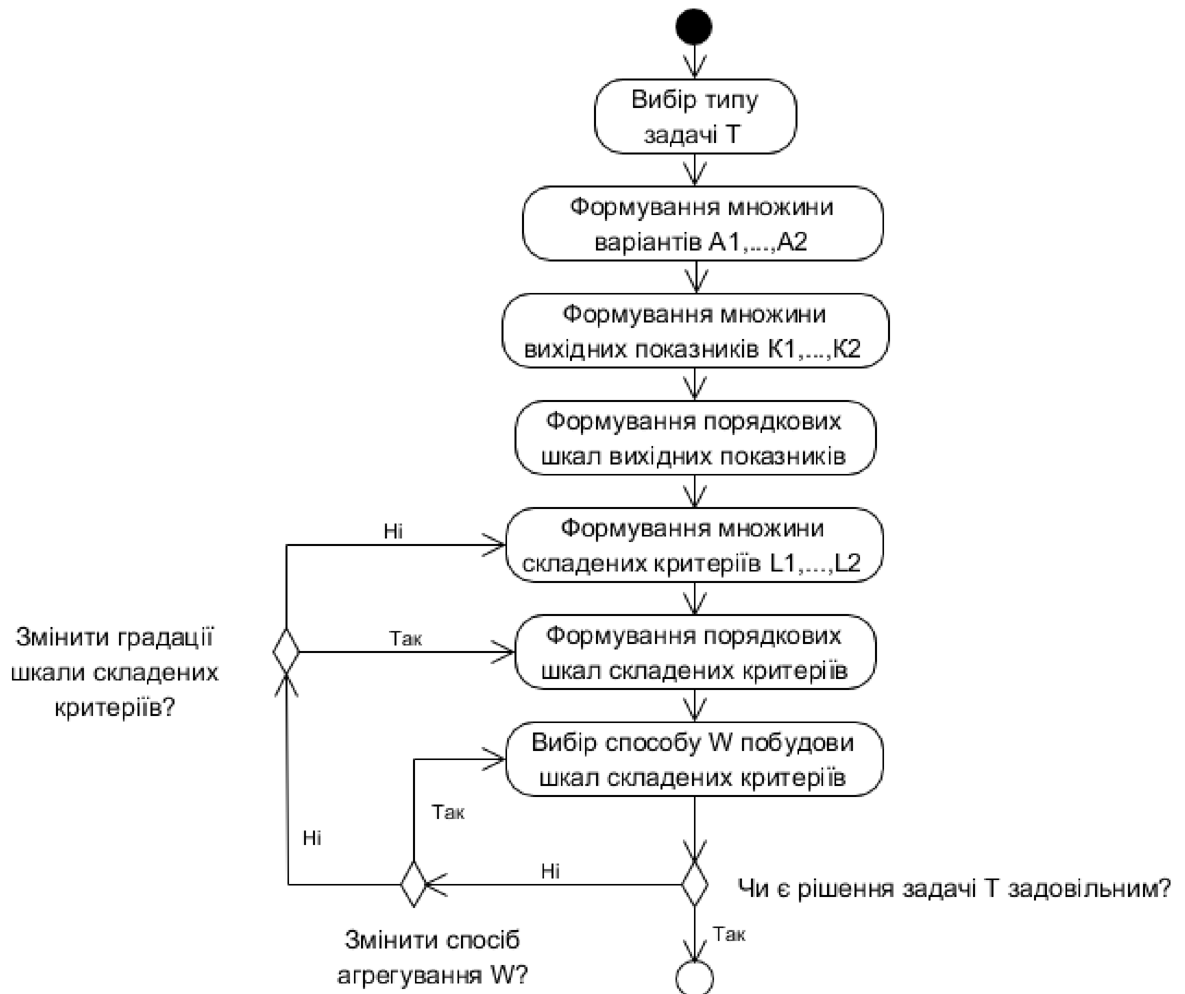

Рисунок 2.1 - Діаграма діяльності алгоритму побудови ієрархічної системи критеріїв

— сформувати порядкові шкали $X_i=\{x_i^1,\ldots,x_i^{g_i}\}, i=1,..m$ $X_i=\{x_i^1,\ldots,x_i^{g_i}\}, i=1,..m$ вихідних показників в залежності від типу завдання T;

— сформувати безліч складових критеріїв $L_1 L_1$, ..., $L_n L_n$, n <m, тобто інтегральних показників, що визначають вбрання ЛПР властивість варіантів, які агрегує вихідні характеристики $K_1 K_1$, ..., $K_m K_m$;

— сформувати порядкові шкали $Y_i = \{y_j^1, \dots, y_j^{h_i}\}, j = 1,..n$ $Y_i = \{y_j^1, \dots, y_j^{h_i}\}, j = 1,..n$ складових критеріїв. Кожна градація шкали складеного критерію є комбінацією градацій оцінок вихідних показників;

— вибрати спосіб W побудови шкал складових критеріїв (агрегування показників). Можливі такі способи: W1 - стратифікація кортежів; W2 - багатокритеріальна порядкова класифікація кортежів; W3 - ранжування кортежів;

— побудувати шкали складових критеріїв всіх ієрархічних рівнів, включаючи верхній рівень, використовуючи кілька різних методів агрегування показників і/або комбінації методів;

— вирішити задачу T (за багатьма критеріями вибрати кращий варіант, упорядкувати варіанти, побудувати порядкову класифікацію варіантів). Якщо отриманий задовільний результат рішення, то алгоритм завершує роботу, інакше перехід до кроку 10;

— якщо результат, отриманий на кроці 9, не задовольняє ОПР, то пропонується або змінити спосіб побудови шкали складеного критерію W (перехід до кроку 7), або змінити градації шкали складеного критерію (перехід до кроку 6), або сформувати новий безліч складових критеріїв Y (перехід до кроку 5) [16].

У рамках задачі порядкової класифікації виконується послідовна побудова шкали кожного складеного критерію, що полягає у використанні комбінації градацій оцінок вихідних показників у якості об’єктів що класифікуються. Класами виступають градації оцінок складеного критерію, унаслідок чого кожна комбінація градацій вихідних оцінок буде відповідати деякій градації оцінок на шкалі

складеного критерію. У загальному випадку для побудови шкал складових критеріїв можна використовувати практично будь-який метод ранжирування або класифікації багатокритеріальних альтернатив, що дозволяє представити кожну градацію шкали складеного критерію у вигляді комбінації градацій оцінок базових показників.

Такий підхід дозволяє при вирішенні конкретної практичної задачі вибрати як найкращий набір складових критеріїв, так і метод або сукупність методів їх побудови.

На третьому етапі виконується кінцеве рішення задачі вибору в отриманому просторі комплексних критеріїв меншої розмірності за допомогою методу АРАМИС (Агрегирование и Ранжирование Альтернатив около Многопризнаковых Идеальных Ситуаций) [17], що дозволяє ранжувати об'єкти, описані багатьма періодичними кількісними або якісними атрибутами $K_1, \dots, K_m$ $K_1, \dots, K_m$, без побудови індивідуальних рангувань об'єктів. Багатокритеріальні об'єкти $A_1, \dots, A_p A_1, \dots, A_p$ розглядаються як точки метричного простору мультимножин з деякою метрикою, які порівнюються і упорядковуються за показником відносної близькості до найкращого (ідеального) об'єкту $A_+ A_+$ або найгіршого (антиідеального) $A_- A_-$ в цьому просторі. Найкращий і найгірший об'єкти мають відповідно найвищі і найнижчі оцінки за всіма критеріями. Всі об'єкти упорядковуються по близькості до найкращого об'єкту $A_+ A_+$, по віддаленості від найгіршого об'єкта А- або за значенням показника відносної близькості до найкращого об'єкту

$$l(A_q) = d(A_+,A_q) / [d(A_+,A_q) + d(A_-,A_q)] l(A_q) = d(A_+,A_q) / [d(A_+,A_q) + d(A_-,A_q)],$$

де d($A_+ A_+$, $A_q A_q$) - відстань до найкращого об'єкта $A_+ A_+$;

d ($A_- A_-$, $A_q A_q$) - відстань до найгіршого об'єкта $A_- A_-$.

Розглянемо особливості застосування технології ПАКС. Агрегування ознак базується на вподобаннях ОПР. Спочатку за участю ОПР формується набір вихідних характеристик розглянутих варіантів. В залежності від специфіки завдання ці характеристики можуть бути або задані заздалегідь, або сформовані в

процесі аналізу проблеми з допомогою аналітика-консультанта або експерта. Для кожного вихідного показника будується шкала, яка може мати числові (крапкові, інтервальні) або вербальні градації оцінок. Шкали вихідних показників можуть збігатися з зазвичай використовуваними на практиці, або конструювати спеціально. Для задач вибору найкращого варіанту і ранжирування варіантів доцільно розглядати тільки ті градації оцінок результативних показників, які зустрічаються в описі наявних варіантів $A_1, \dots, A_p A_1, \dots, A_p$. Таким способом можна попередньо скоротити розмірність вихідного простору ознак. Якщо спочатку не заданий набір реальних альтернатив, необхідно розглядати множину всіх можливих кортежів оцінок в просторі ознак, утворену декартовим добутком градацій оцінок на шкалах критеріїв. Далі, ґрунтуючись на досвіді і інтуїції ОПР, будується ієрархічна система критеріїв. ОПР на свій розсуд визначає число, склад і зміст критеріїв кожного рівня ієрархії. В якості критерію можна вибрати один з вихідних показників або декілька вихідних характеристик, об'єднаних в складений критерій. ОПР встановлює, які вихідні показники вважатимуться самостійними критеріями, а які будуть віднесені до складеного критерію. Процедура агрегування показників носить послідовний характер, тобто отримані групи критеріїв об'єднуються послідовно в нові групи наступного рівня ієрархії і так далі аж до єдиного інтегрального критерію найвищого рівня, якщо це необхідно.

ОПР визначає також смисловий зміст критеріїв та градацій шкал оцінок. Критерії повинні мати такі шкали оцінок, які, з одного боку, будуть відображати агреговані властивості об'єктів, а з іншого боку, будуть зрозумілі ОПР при остаточному впорядкування та класифікації об'єктів. Рекомендується будувати шкали критеріїв з невеликим (3-5) числом вербальних градацій.

Щоб зменшити вплив особливостей різних методів, за допомогою яких конструюються шкали складових критеріїв, пропонується на різних етапах технології ПАКС застосовувати кілька різних методів і / або їх поєднання. Використання багатьох різних способів побудови шкал складових критеріїв і інтегрального показника перетворює вихідну задачу вибору в задачу колективного вибору, в якій варіанти представляються декількома об'єктами, заданими багатьма

нечисловими ознаками. Тому слід її вирішувати за допомогою методів групового вербального аналізу рішень [18].

2.3.2 Агрегування якісних ознак у просторі зниженої розмірності

Формально задача зниження розмірності простору ознак має наступний вигляд:

$$X_1X_1 \times \ldots \times X_mX_m \to Y_1Y_1 \times \ldots \times Y_nY_n, \; n < m,$$

де X1, ..., Xm - вихідний набір ознак;

Y1, ..., Yn - новий набір ознак;

M - розмірність вихідного простору ознак;

n - розмірність нового простору ознак.

Кожен з ознак має свою власну шкалу $X_i = \{x_i^1, \ldots, x_i^{g_i}\}, i = 1,..m$ $X_i = \{x_i^1, \ldots, x_i^{g_i}\}, i = 1,..m$, $Y_i = \{y_j^1, \ldots, y_j^{h_i}\}, j = 1,..nY_i = \{y_j^1, \ldots, y_j^{h_i}\}, j = 1,..n$ з упорядкованою градацією якісних (символьних, вироблених) оцінок.

Будемо розглядати задачу зниження розмірності простору ознак як задачу багатокритеріальної класифікації, в якій різні комбінації вихідних ознак (кортежі оцінок) послідовно агрегуються в менші набори нових ознак, що мають для ОПР цілком певний сенс. Підсумком є ієрархічна система критеріїв, верхній рівень якої визначається змістом практичної проблеми. Процедура агрегування показників має багаторівневої ієрархічної структурою зі «слабкими» зв'язками, в якій елемент нижчого рівня (оцінки по вихідних показників) підпорядкований двом і більше вершин вищого рівня (оцінками по складовим критеріям). Переходячи крок за кроком на більш високий рівень ієрархії, ОПР може сконструювати прийнятні для нього складові критерії аж до одного єдиного. Уявімо процес побудови шкал

складових критеріїв у вигляді однотипних блоків. Блоки змістовно виділяються ОПР в залежності від специфіки розв'язуваної задачі. Кожен блок класифікації ii-го рівня ієрархії складається з деякого набору ознак і одного складеного критерію. Як об'єкти класифікації виступають всі градації оцінок на шкалах ознак. Класами рішень ii-го рівня служать градації оцінок на шкалі складеного критерію. У блоці класифікації $(i+1)(i+1)$-го рівня ієрархії складові критерії ii-го рівня вважаються ознаками, безліч градацій оцінок яких є нові об'єкти класифікації в скороченому просторі ознак, а класами рішень будуть тепер градації оцінок на шкалі складеного критерію $(i+1)(i+1)$-го рівня. Процедура повторюється до тих пір, поки не залишиться єдиний складовою критерій верхнього рівня, який є шуканим інтегральним показником, шкала оцінок якого утворює впорядковані класи рішень $D_1, \dots, D_q D_1, \dots, D_q$. Тим самим встановлюється відповідність між класами рішень $D_1, \dots, D_q D_1, \dots, D_q$ і сукупністю вихідних показників - безліччю X_1 X_1 × ... × $X_m X_m$ всіх можливих комбінацій градацій оцінок на шкалах критеріїв $X_i = \{x_i^1, \dots, x_i^{g_i}\}, i = 1,..m X_i = \{x_i^1, \dots, x_i^{g_i}\}, i = 1,..m$ і знаходяться межі класів, що дозволяє легко побудувати класифікацію реальних варіантів (альтернатив) $A_1, \dots, A_p A_1, \dots, A_p$, оцінених за багатьма критеріями $K_1, \dots, K_m K_1, \dots, K_m$.

Кожен блок ii-го рівня ієрархії являє собою зв'язний двочастковий граф $G_i G_i$ $= < U, E >= < U, E >$, де U - вершини графа, E – дуги. Схематичне зображення елементу ієрархії зображено на рисунку 2.2.

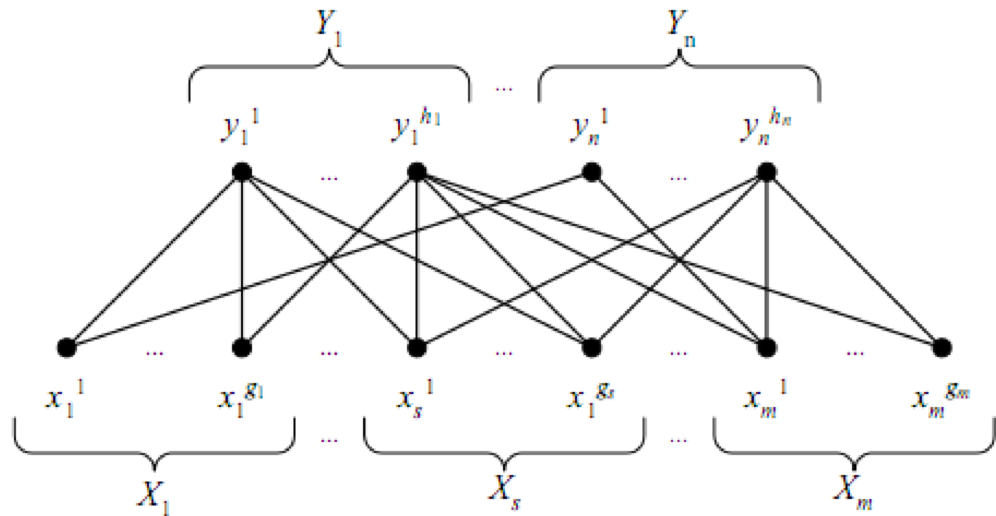

Рисунок 2.2 - Схематичне зображення елементу ієрархії

Множиною вершин U = X ∪ Y є безлічі значень вихідних ознак $XX = X_1X_1 \cup \ldots \cup X_mX_m$ і градацій шкал складових критеріїв $Y\ Y = Y_1Y_1 \cup \ldots \cup Y_nY_n$. Дуги E графічно висловлюють набори вирішальних правил, на підставі яких шикуються кортежі оцінок, які формують градації шкал складових критеріїв (фактично - це одна з форм смислової інтерпретації переваг ОПР).

Між вершинами, що належать до різних множин, є кратні дуги, тобто граф G_i G_i є мультиграфом. Кратність дуг графа Gi дає можливість уявити градацію $y_j^{f_j}y_j^{f_j}$ ($f_jf_j = 1,\ldots,h_j = 1,\ldots,h_j$, $j = 1,\ldots,nj = 1,\ldots,n$) шкали складеного критерію Y_jY_j як множину з повторюваними елементами або мультимножину [19]:

$$A_{fj} = \{k_{A_{fj}}(x_1^1)\text{o}x_1^1, \ldots, k_{A_{fj}}(x_1^{g_1})\text{o}x_1^{g_1}, \ldots, k_{A_{fj}}(x_m^1)\text{o}x_m^1, \ldots, k_{A_{fj}}(x_m^{g_m})\text{o}x_m^{g_m}\}$$

$$A_{fj} = \{k_{A_{fj}}(x_1^1)\text{o}x_1^1, \ldots, k_{A_{fj}}(x_1^{g_1})\text{o}x_1^{g_1}, \ldots, k_{A_{fj}}(x_m^1)\text{o}x_m^1, \ldots, k_{A_{fj}}(x_m^{g_m})\text{o}x_m^{g_m}\}$$

,

де $k_{A_{fj}}(x_i^{e_i})k_{A_{fj}}(x_i^{e_i})$ – кількість разів входження значення вихідної ознаки $x_i^{e_i}x_i^{e_i}$ ($e_ie_i = 1,\ldots,g_i, i = 1,\ldots,m)1,\ldots,g_i, i = 1,\ldots,m)$ в кортежі, які формують градацію шкали складеного критерію $y_j^{f_j}y_j^{f_j}$;

oo – кратність оцінки $x_i^{e_i}x_i^{e_i}$.

Використання понять мультиграфу і мультимножини дозволяє побудувати єдину схему формалізації поняття складеного критерію і по-новому вирішувати як відомі завдання, в яких є певні складності, наприклад, завдання розпізнавання ієрархічних структур, так і нові види завдань.

Для формування шкал оцінок по складовим критеріям ОПР може скористатися кількома способами з арсеналу засобів вербального аналізу рішень.

Найбільш простим і легко сприймаються як ОПР способом конструювання порядкової шкали складеного критерію є метод стратифікації кортежів, в якому використовуються однотипні (наприклад, з однаковим числом градацій) набори порядкових вербальних шкал вихідних показників. Аналогічно методам векторної стратифікації [20] метод стратифікації кортежів заснований на перетині багатовимірного дискретного простору ознак паралельними гіперплощинами. Кожен шар (страта) складається з однотипних комбінацій градацій оцінок на шкалах критеріїв $X_i X_i$, а число таких перетинів визначається ОПР із змістовних міркувань. Максимально можливе число шарів можна розрахувати за формулою $s = 1 - m + \sum_{i=1}^{m} g_i s = 1 - m + \sum_{i=1}^{m} g_i$. Кожен шар утворюється як комбінація кортежів градацій оцінок, сума номерів яких фіксована. Число класів r≤s. У більш складних процедурах побудови шкал критеріїв використовуються методи вербального аналізу рішень ЗАПРОС або ОРКЛАСС [21], які оперують на безлічі всіх можливих кортежів оцінок в просторі ознак, утвореному декартовим добутком градацій оцінок на шкалах критеріїв $X_1 X_1 \times \ldots \times X_m X_m$. У цих випадках число можливих комбінацій оцінок (альтернатив) одно t = П = miig1.

2.3.3 Приклад агрегування критеріїв

Розглянемо невеликий ілюстративний приклад. Нехай ОПР необхідно побудувати шкалу складеного критерію D з градацій оцінок базових показників A,

В і С. Припустимо, що всі критерії A, B, C і D мають шкали з трьома вербальними градаціями порядкових оцінок

A = {a^0a^0, a^1a^1, a^2a^2}, B {b^0b^0, b^1b^1, b^2b^2}, C = {c^0c^0, c^1c^1, c^2c^2} і D = {d^0d^0, d^1d^1, d^2d^2},

де e^0e^0 - відмінно, e^1e^1 - середньо, e^2e^2 - погано.

Застосовуючи для побудови порядкової шкали складеного критерію метод стратифікації кортежів, ОПР може, наприклад, об'єднати градації оцінок вихідних критеріїв в узагальнені градації складеного критерію за таким принципом: все кращі оцінки за базовими показниками утворюють одну кращу оцінку по складеному критерію, все середні оцінки - одну середню оцінку, все гірші оцінки - одну гіршу оцінку. Приклад побудови шкали методом стратифікації кортежів зображений на рисунку 2.3.

	$a^0b^0c^1$	$a^0b^0c^2$	$a^0b^2c^0$	$a^2b^0c^0$	$a^1b^2c^2$	
$a^0b^0c^0$	$a^0b^1c^0$	…	$a^1b^1c^1$	…	$a^2b^1c^2$	$a^2b^2c^2$
	$a^1b^0c^0$	$a^1b^1c^2$	$a^1b^2c^1$	$a^2b^1c^1$	$a^2b^2c^1$	
$\mathbf{d^0}$			$\mathbf{d^1}$		$\mathbf{d^2}$	

Рисунок 2.3 - Приклад побудови шкали методом стратифікації кортежів

3 ЕТАПИ РОЗРОБКИ ДОДАТКУ З ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1 Розбиття моноліту на частини

Найчастіше причиною з’явлення мікросервісної архітектура є надзвичайно громіздкий моноліт. Відомі випадки і коли новий проект був закритий через надмірно складку мікросервісну архітектуру. На початку свого шляху проект краще пристосовується до постійно змінюючихся вимог, якщо це монолітний проект. Працює одне з основних правил розробки програмного забезпечення YAGNI (You Ain`t Gonna Need It), що дослівно перекладається як “Вам це не знадобиться”. Це правило означає, що не слід розробляти того, що з великою долею вірогідності може не знадобитися. Тому не треба ускладнювати архітектуру системи, якщо вона тільки бере початок та має невелику кількість функціоналу. До того ж, невеликий монолітний набагато швидше та дешевше розробляти.

Ось чому багато компаній обирають для початку нового монолітну архітектуру. Ніхто й ніколи не знає до чого призведе розвиток проекту та чи буде він потребувати складнішу систему. Саме такий архітектурний вибір пояснює, чому процес розробки мікросервісної архітектури це майже завжди процес перетворення монолітного застосування на мікросервісне, та є різновидом модернізації системи. Це щось, що розробники робили протягом багатьох десятиліть. В результаті їх роботи є деякі ідеї, які ми можемо використовувати коли займаємося рефакторингом нашої програмної системи.

Однією стратегією, що не рекомендують використовувати є "Big Bang", переписування. Це - коли ви зосереджує всі свої зусилля з розвитку старого на розробку нового мікросервісного додатку з нуля. Хоча це здається привабливим, це надзвичайно небезпечно і ймовірно закінчиться невдачею. Мартін Фаулер казав про цей метод так: "Єдиною річчю, що гарантує Big Bang, є великий вибух!"

Замість повного переписування, ви повинні поступово змінювати свою монолітну систему. Ви поступово створюєте нову програму, що складається з мікросервісів, таким чином щоб вони працювали паралельно з монолітом. Коли новий сервіс готов, його тестують та замінюють їм відповідний функціонал в моноліті. З часом функціонал моноліту звужується, або зникає повністю, або моноліт просто стає іншим мікросервісом. Ми створимо нову програмну систему, що складається з мікросервісів, навколо застарілого застосування, яке в кінцевому рахунку помре. Ще один спосіб - це проектування моноліт з самого початку припускаючи, що він може в майбутньому перетворитися на мікросервісну архітектуру. Тоді моноліт логічно буде поділено на різний за логікою функціонал, не має бути такого поняття, як "спагетті код", коли один функціонал в декількох різних місцях залежить від декількох інших місць іншого за логікою функціоналу, особливо якщо це не інтерфейси, а їх реалізації. Грамотно побудований моноліт матиме декілька незалежних модулів з передумовами для власної бази даних та власного API [7].

Але не всі компанії використовують такий підхід до створення програмної системи. Деякі все ж таки наполягають на тому, щоб відразу створювати мікросервісний додаток. Основний аргумент за такий підхід це простота у розподіленні людських ресурсів. Компанії не треба змінювати багато процесів та на ходу займатися пошуком спеціалістів в нових областях. Програмістам не треба переходити з однієї архітектури на іншу, оскільку вони відразу звикли до сервісів. Також не треба вирішувати проблему розділення моноліту, що не є легким і швидким процесом, навіть якщо він грамотно розроблен.

3.2 Розгортання мікросервісів та їх масштабування

Розгортання монолітних програмних систем передбачає, що ви запускаєте декілька ідентичних копій однієї, зазвичай великечної програми. Це головним

чином зроблено за допомогою N серверів, фізичних чи віртуальних, і виконання G копій програми на кожному. Незважаючи на те, що це виглядає досить легко, як правило, це не так. Однак це набагато легше, ніж розгортати мікросервіси.

3.2.1 Способи розгортання мікросервісної архітектури

Якщо ви плануєте розгорнути мікросервісну програму, то ви маєте бути знайомі з безліччю фреймворків і мов програмування, які використані в цих сервісах. Це одне з найскладніших завдань, так як кожен з цих сервісів має свої певні вимоги до розгортання, потреби в ресурсах, вимоги до масштабування. Крім розгортання послуг має бути швидким, надійним, і ефективним.

Можливо, самий традиційний підхід до розгортання системи є підхід, коли розробники програмного забезпечення "піднімають" один чи декілька фізичних чи віртуальних серверів (хости) та запускають декілька сервісів на кожному з хостів. У цього методу є декілька варіацій, включаючи варіант, коли кожний сервіс це окремий процес, або коли декілька сервісів це один процес.

Серед переваг такого підходу - легке розгортання. Серед недоліків - недостатньо контролю над сервісом, недостатній рівень ізоляції.

Інший спосіб розгорнути мікросервіси - один сервісний екземпляр (інстанс) на один хост. Це дозволяє Вам запускати кожний інстанс окремо на своєму сервері. У цього способа є два різновиди: сервісний інстанс на одну віртуальну машину та інстанс на один контейнер.

Сервісний інстанс на одну віртуальну машину надає Вам можливість "пакувати" кожний сервіс як зображення віртуальної машини як Amazon EC2 AMI. Кожен інстанс це віртуальна машина, яка запущена використовуючи це зображення. Одним з популярних додатків, використовуючих цей спосіб є Netflix для свого сервісу потокової передачі відео. Щоб побудувати вашу власну

віртуальну машину, ви можете налаштувати безперервний сервер інтеграції як Дженкінс.

Серед переваг такого підходу - контроль над ресурсами. Серед недоліків - віртуальна машина достатньо громіздка, тому її досить довго розгортати.

В наступному підході кожен сервісний інстанс працює в його відповідному контейнері, який є механізмом віртуалізації на рівні операційної системи. Деякі популярні контейнерні технології - Solaris Zones і Docker.

Для використання цього способу ви маєте запакувати свій сервіс як образ файлової системи, включаючи додатки і бібліотеки, що потрібні для виконання сервіса, зазвичай відоме як контейнерний образ. Як тільки обслуговування упаковано як контейнерний образ, тоді ви повинні запустити один або декілька контейнерів і можете запускати кілька контейнерів на фізичному чи віртуальному хості.

Щоб запускати декілька контейнерів та зручно керувати їми, багатьом розробникам подобається використовувати спеціальне програмне забезпечення, таке як Kubernetes або Marathon.

Серед переваг такого підходу - контроль над ресурсами, швидко розгортати. Серед недоліків - не так безпечно, як віртуальна машина, оскільки розділяє ядро операційної системи хоста.

Перша задача це CI/CD (Continuous integration/continuous delivery, безперервна інтеграція і безперервна доставки).

Основна мета CI - синхронізація шляхом перевірки свіжого коду на можливість інтегрування в існуючий. Щоб досягти цієї мети CI-сервер перевіряє код на здатність бути скомпільованим та перевіряє чи не “впали” тести (показали помилку в коді). Часто емулюється новий сервер, на якому працюють мікросервіси вже зі зміненим східним кодом. На цьому сервері проходить подальше тестування, якщо воно потрібно. Ці віртуальні тимчасові сервери створюються та використовуються зазвичай один раз на кожну зміну коду.

Якщо потрібно повторне використання цього серверу, можна розмістити його в спеціальне місце, що називають CI-інструментарієм. В найпростішому випадку весь код зберігається в одному репозиторії. Будь-яка зміна коду запускає збірку, далі верифікація усіх мікросервісів.

Рисунок 3.1 – Збирання за спільним сховищем коду

Цей шаблон не рекомендується використовувати, але на стадії розвитку проекту його використання припустиме.

Дана модель має свої недоліки. Кожна зміна в будь-який сервіс змушує збирати те перевіряти усю систему цілком, що є на самим оптимальним рішенням.

Ще один варіант цього методу це деревоподібна структура для всього коду і декілька CI-збірок. Нижче наведено приклад такого збирання (рис 3.2). Цей підхід вирішує декілька проблем, однак він усе одно передбачає повну перевірку систему, незважаючи на те, що збірка буде здійснена для кожного з мікросервісів.

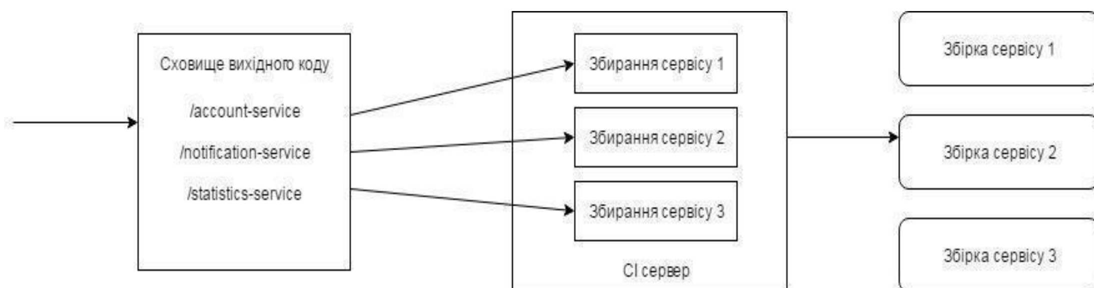

Рисунок 3.2 – Збирання на базі деревоподібної структури

Найбільш оптимальним є спосіб CI, який зображений на рис 2.9. Кожен сервіс має свій репозиторій для коду та власну CI-збірку. При набутті системою

змін, перевіряється одна збірка. Однак звищився рівень складності при зміні коду, бо задіяно багато репозиторіїв.

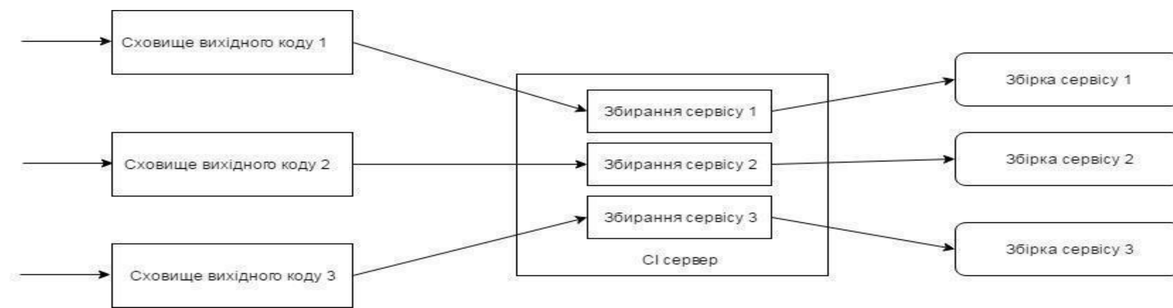

Рисунок 3.3 – Збирання при наявності репозиторію для кожного сервісу

Ідея конвеєрного збирання дає можливість моніторингу нашої програми та результати її перевірки на будь-якій стадії.

3.2.2. Масштабування мікросервісів

Поняття "мікросервіс" та архітектура породжують групу проблем коли справа доходить до масштабованості. Але це зовсім не означає, що мікросервіси не масштабуються. Їх вважають більш масштабованими, ніж їхні попередні монолітні додатки, але вони дійсно вимагають зовсім іншого підходу до масштабування.

Команда розробників завжди прагне до системи, що автоматично масштабується залежно від рівню навантаження. Але такі рішення не завжди оптимальні з точки зору затрат ресурсів.

Існують наступні вимоги до масштабування системи:

- доступність
- час затримки
- збереження даних

Існує велика різниця між масштабуванням моноліту там масштабуванням мікросервісів. Щоб масштабувати моноліт не потрібно замислюватися відразу про

багато складових, бо працездатність програми залежить тільки від часу, за який виконується байткод. Щодо мікросервісної архітектури - тут усе значно складніше. Крім виконання свого коду, що може бути написан на будь-якій мові програмування, сервіси ще мають спілкуватися один з одним. Потрібно враховувати поведінку системи в різних станах з навантаженням на один чи декілька сервісів.

Існує таке поняття як ідемпотентна операція. Її особливість у тому, що результат змінюється тільки якщо вона виконується вперше. Усі подальші виконання ідемпотентної операції не призводять до змін у стану системи. Такі операції є інколи дуже зручним способом повторного відновлення повідомлення у випадку коли ми не знаємо чи була вже виконана операція.

Є дві основні причини для масштабування. Перша - потрібен механізм проти збоїв. У випадку помилки сервісу буде корисним мати ще один такий сервіс, який відразу візьме новий запит в обробку.

Ще одна причина - продуктивність. Потрібен механізм, що дозволяє впоратися з мінливим рівнем навантаження.

Є декілька різних способів масштабування. Нижче наведено їх застосування разом з MSA.

Нарощування потужностей. Виконуючи більшу кількість операцій за одиницю часу ми таким чином зменшуємо час затримки. Такий різновид масштабування називається вертикальним. Він не завжди є оптимальним, оскільки часто дешевше мати декілька менших серверів ніж один великий.

Розділення навантаження. Цей спосіб означає, що кожний компонент повинен мати власний хост.

Балансування навантаження. Ідея полягає у тому, щоб для навантаженого сервісу мати кілька хостів з інстансами цього сервісу. Але коли є запит на цей сервіс, він спочатку потрапляє на балансувальник навантаження, який за допомогою алгоритму та даних навантаження на всі хости перенаправляє запит на пріоритетний хост.

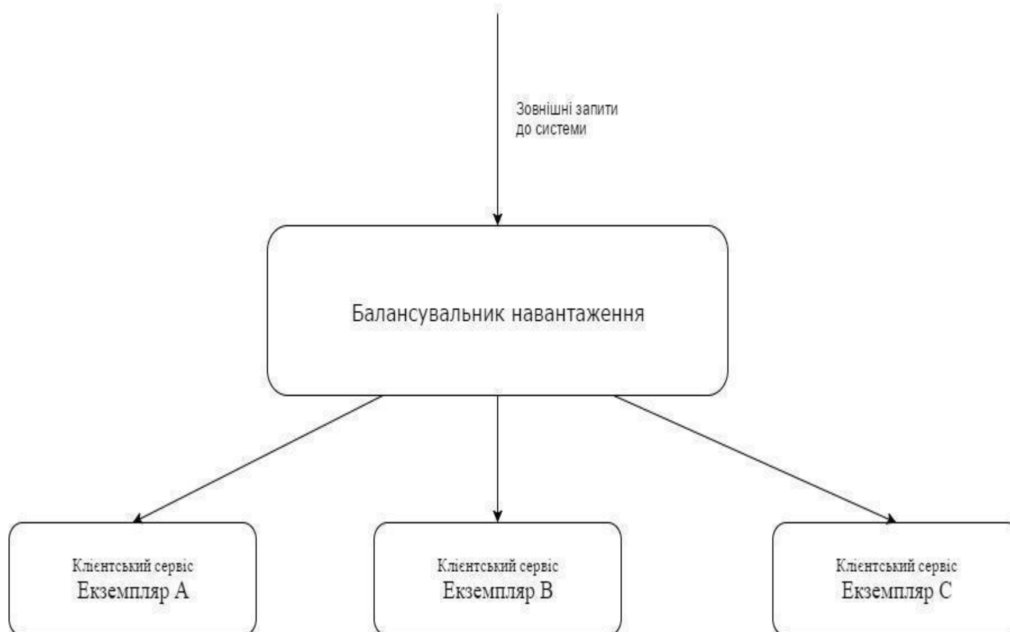

Рисунок 3.4 – Схема розподілу навантаження

Система за принципом виконавець. Даний спосіб добре показує себе коли є піки навантаження. Нові інстанси підіймаються в момент великого навантаження.

Однією з дуже важливих проблем є проблема масштабування баз даних. Сервіси масштабуються просто, бо вони не мають стану. Масштабування різних баз даних має бути проходити по-разному. Зазвичай більшість сервісів лише зчитують дані з бази, а не пишуть нові дані. Для таких сервісів може бути корисним кешування. В різних СУБД зазвичай реалізован механізм створення реплік. Сервіс може направляти запити на запис на оригінальний вузол, а всі запити на зчитування розподіляти між репліками (рис. 3.5).

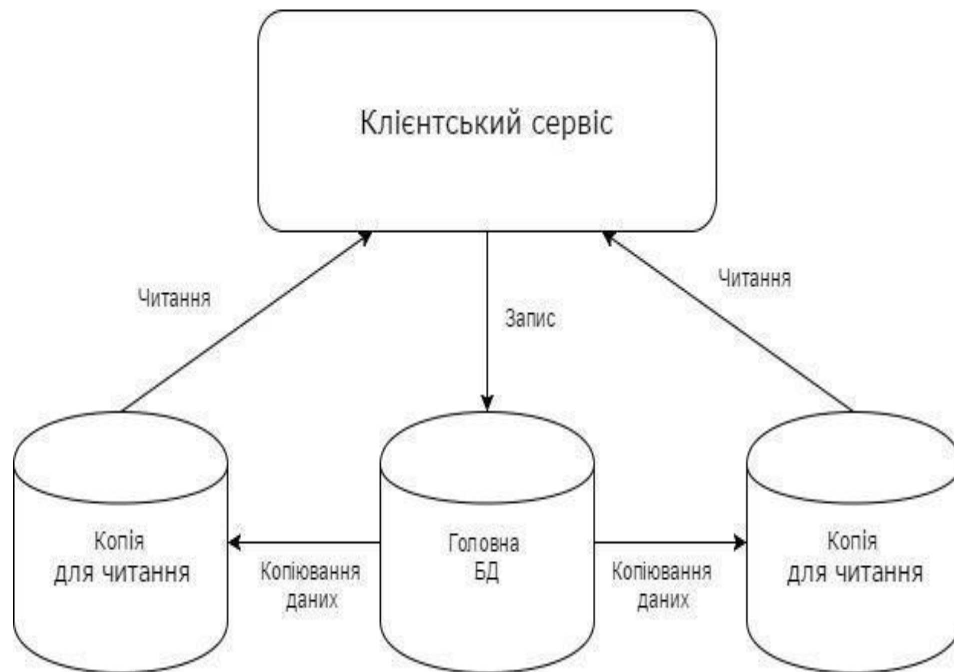

Рисунок 3.5 – Приклад організації взаємодії з БД

Резервні копії бази створюються через деякий час після запиту на запис. Якщо зчитати дані до реплікації - вони будуть застарілі. Актуальні дані стануть доступні лише через деякий час.

3.3 Нефункціональні вимоги

Нефункціональні вимоги - вимоги до програми, умови запуску програмної системи. Основні атрибути якості MSA можна побачити нижче (рис. 3.6).

3.4 Інструменти для побудови MSA

Щоб реалізувати взаємодію сервісів з остальною системою можна скористатися спеціальними інструментами.

3.4.1 Єдина точка входу (API Gateway)

Коли архітектура зростає настільки, що кількість сервісів обчислюється десятками, з'являється проблема як клієнту зручно отримати відповідь на свій запит. Як йому комунікувати з усіма компонентами? Було придумано шаблон API Gateway. Цей шаблон є точкою входу для декількох сервісів або системи в цілому.

Приклад єдиного шлюзу для входу наведено на рисунку 3.7. Кожен запит клієнта перенаправляється на відповідний компонент. Як єдина точка входу до системи, API Gateway також відповідає за підтримку безпеки, перевірки OAuth-токенів та перевірки прав доступу клієнтів до конкретних сервісів.

Важливо підкреслити, що на рисунках 3.7 та 3.9, використовується єдиний сервіс API Gateway, який відповідає на запити з різних клієнтів. Це може бути серьозним ризиком, тому що сервіс API Gateway буде рости і розвиватися на основі багатьох різних вимог із клієнтів. В кінцевому рахунку він буде роздут через ті різні вимоги і за ефективністю буде подібен до монолітної системи або до монолітного сервісу. Ось чому наполегливо рекомендується розділити API Gateway на декілька менших сервісів, наприклад один API Gateway сервіс на клієнт. Зазвичай це не оптимальне рішення мати єдиний API Gateway, що з 'єднує всі внутрішні мікросервіси вашої програмної системи. Якщо це так, це буде працювати як монолітний накопичувач і порушувати мікросервісну автономію.

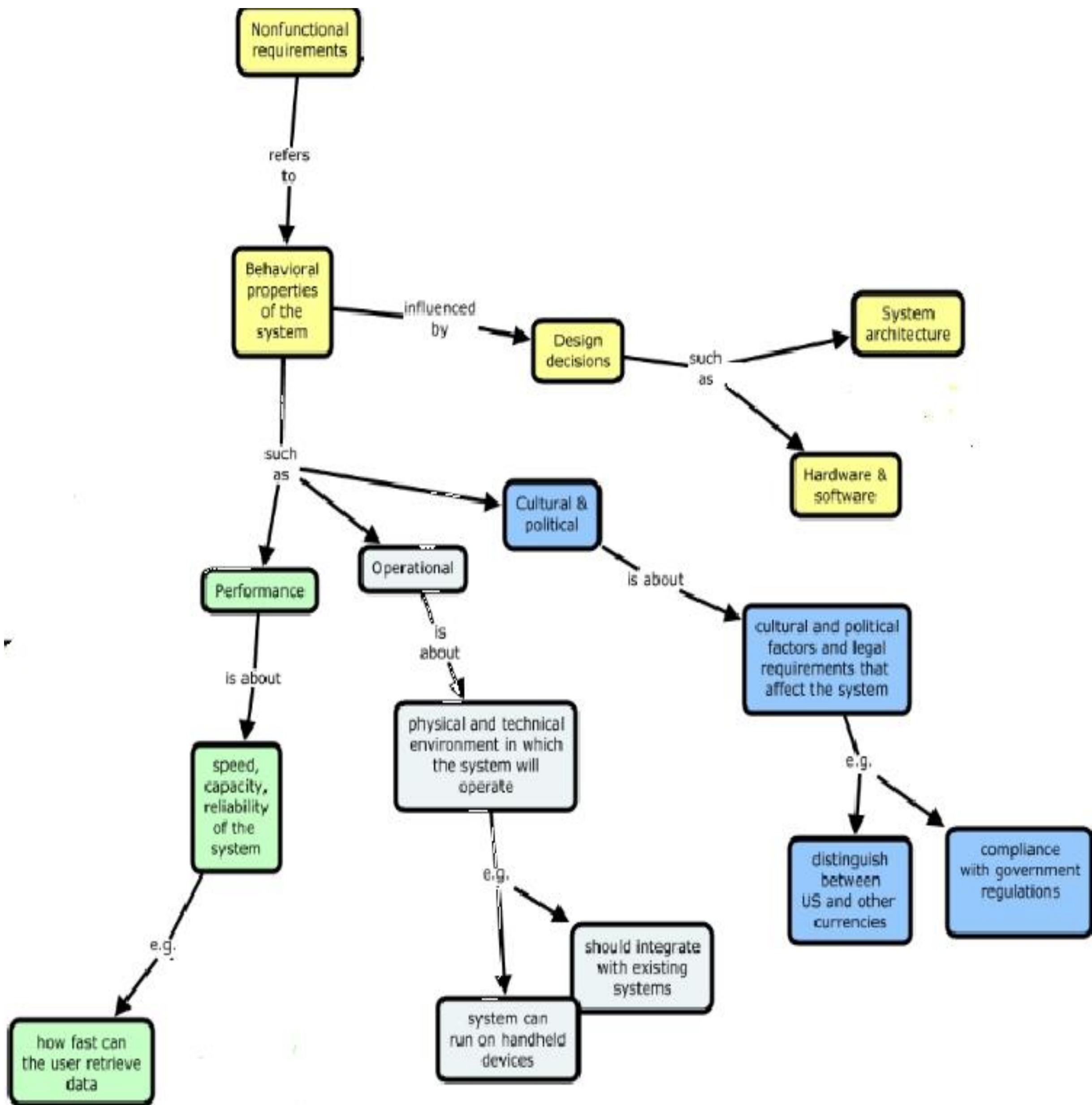

Рисунок 3.6 – Нефункціональні вимоги

Розглянемо дві схеми для більшого розуміння необхідності API Gateway у розробці мікросервісної архітектури. На рисунку 3.7 зображена архітектура додатку без використання API Gateway, а на рисунку 3.8 – з використанням API Gateway.

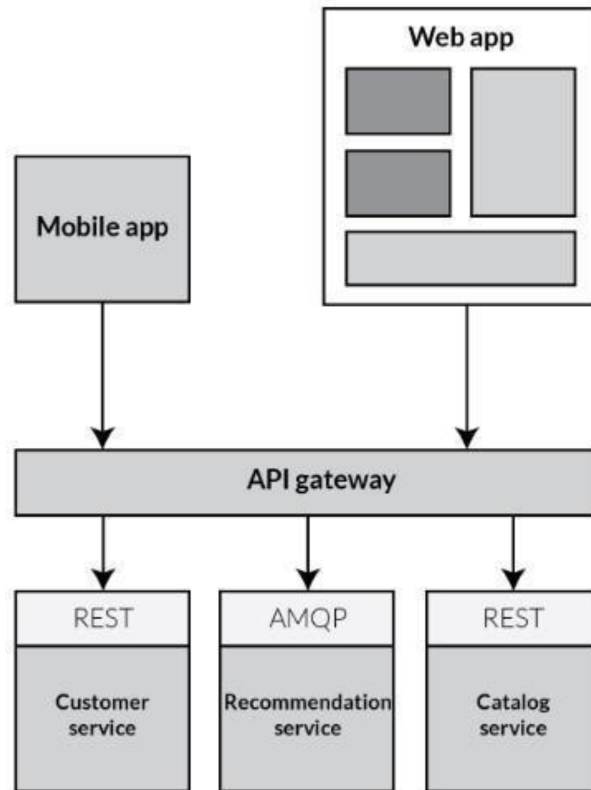

Рисунок 3.7 – Взаємодія зовнішніх клієнтів з мікросервісами[8]

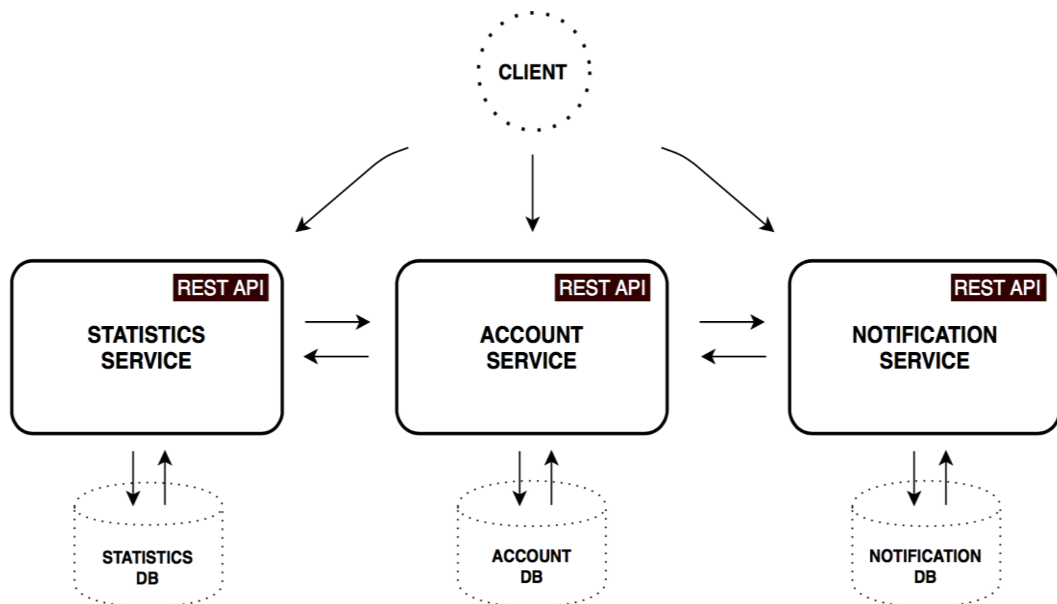

Рисунок 3.8 – Архітектура без API Gateway

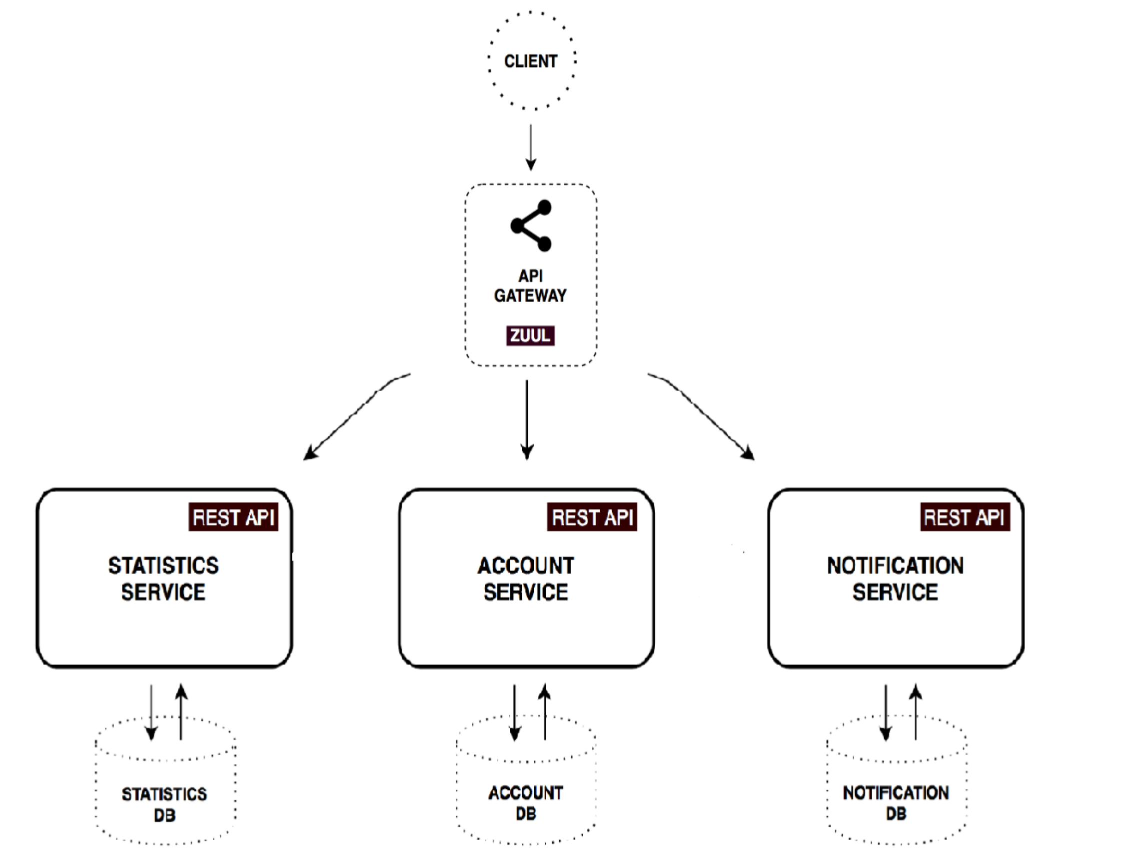

Рисунок 3.9 – Архітектура з API Gateway

Переваги використання спільної точки входу [9]:

— зручність використання

— гнучкісь системи

— можливість кешування

— можливість композитних відповідей з декількох сервісів

API Gateway є аналогом ООП-шаблону фасад, який надає єдиний зручний інтерфейс для роботи зі складною системою.

Недоліки шаблону такі:

— ускладнення системи шляхом додавання нового сервісу

— постійне оновлення API для шлюза.

Незважаючи на ці недоліки, зазвичай використання цього паттерну має сенс для більшості програмних систем.

3.4.2 Виявлення сервісів (Service Discovery)

В умовах масштабування програмна система стає дуже динамічною, вона змінюється, постійно оновлює свої компоненти. Компоненти можуть додаватися та видалятися в режимі реального часу. Зазвичай порті і сервери призначаються автоматично. Щоб новий сервіс можна була знайти клієнту або іншому сервісу, була придумана концепція виявлення сервісів. Так, згідно з цією концепцією треба створити новий компонент - реєстр. Він повинен займатися моніторингом системи та відстежувати всі мікросервіси. Кожен раз коли сервіс запускається, він повинен проходити реєстр, користуючись при цьому клієнтом самого сервіса, або ж через сторонній сервіс, що контролює інстанси мікросервісів і зберігає статус в реєстрі.

Реєстр сервісів це основний компонент для виявлення сервісу. Він зберігає мережеві розташування екземплярів сервісу. РС завжди повинен бути актуальним. Інколи клієнту можуть використовувати механізм кешування задля отримання необхідних сервісів.

3.4.3 Автоматичний вимикач (Circuit Breaker)

Зазвичай мікросервіси можуть обробляти запит разом. Але коли один сервіс викликає інший, є вірогідність, що мікросервіс матиме великий час відгуку. Таким чином викликаючий компонент на має можливості обробляти інші запити. Відмова одного сервісу ланцюжком ламає роботу системи.

Система використовує паттерн автовимикач щоб забезпечити надійність (рис 3.10). Він огортає виклик собою та моніторить виключну ситуацію. Коли кількість викликів несправних сервісів сягає порогового значення, вимикач повертає усі наступні виклики з помилкою, а перший виклик ігнорується.

Вимикачі зберігають ресурси системи, зменшують навантаження на сервер, уникають ситуацію очікування таймаутів. Цей патерн використовується в різного роду програмних системах, щоб надати надійності від помилок в інших частинах. Я автовимикач реалізован вірно, ланцюжкові помилки стають неможливі.

Реалізаціями даного патерну для мікросервісів є:

— Netflix Hystrix
— Nginx Circuit Breaker

3.4.4 Docker

Кількість сервісів у програмній системі може бути великим, та кожен сервіс може мати відмінну технологію від інших. З таким різноманіттям технологій існує проблема управління середовищами. Щоб спростити процес створення різних середовищ використовують технологію контейнерів. Одним з найвідоміших інструментів з підтримкою контейнеризації є Docker.

Докер - популярний загальнодоступний проект на основі контейнерів Linux. Докер написаний на мові GO і розвинений з Dotcloud (компанія PaaS). Докер - контейнерний двигун, який використовує функції ядра Linux, такі як простір імен та контрольні групи, щоб створити контейнери зверх операційної системи і автоматизувати розгортання на контейнері.

Найкраща функція Докера - можливість співпраці. Образи докера можуть бути вигружені в сховище і можуть бути загружені на будь-який інший хост, щоб управляти контейнерами образу. Крім того, сховище Докера має тисячі образів створених іншими користувачами, і ви можете загрузити ті образи на свій хост, залежно від вимог до вашої програмної системи.

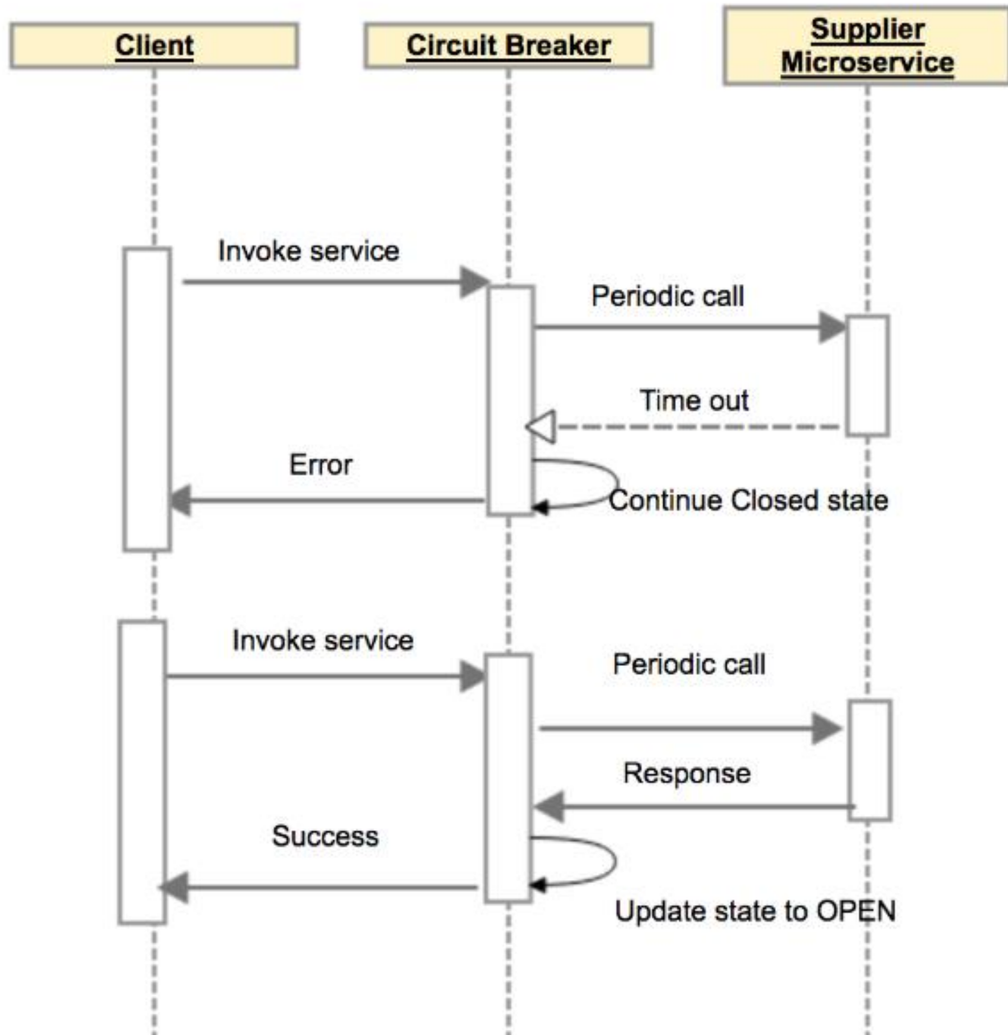

Рисунок 3.10 – Приклад роботи автоматичного вимикача

Контейнеризацію завжди порівнювали як підхід до розгортання за віртуалізацією. Віртуальні машині використовувались як спосіб оптимізації, бо на одному компьютері можливо запустити багато віртуальних машин. Віртуальні машини створюють середовище для кожної програми. Але віртуальні машини складно масштабувати, оскільки віртуалізація потребує багато процесорного часу та пам’яті.

Мікросервіси не є великими та громіздкими, зазвичай це компактні програмні рішення, яке потребує своє окреме середовище. Тому створення віртуальної машини для такої задачі не є оптимальним рішенням. Docker вимагає набагато менше системних ресурсів. За допомогою Docker можна розгорнути багато сервісів на одному сервері.

Основні переваги використання Docker:

- час запуску
- час розгортання
- легке масштабування
- оптимізація
- підтримка великої кількості технологій [11].

3.5 Засоби розробки

3.5.1 Java

Java – на даний момент одна з найвідоміших мов програмування. Програми на Java транслюються в байт-код, який виконується віртуальною машиною Java(JVM) – програмою, яка опрацьовує байтовий код та передає інструкції обладнанню як інтерпретатор. Перевага подібного способу виконання програм є повна незалежність байт-коду від операційної системи та апаратури, що дозволяє виконувати Java-додатки на будь-якому пристрої, для якого існує відповідна віртуальна машина.

Java є найпопулярнішою мовою для побудови мікросервісів. Багато відомих компаній створюють свої системи саме на Java прикладом є Netflix, Amazon.

3.5.2 Spring framework

Spring framework – найпотужніша Java-бібліотека, яка дозволяє створювати програмні системи будь-якої складності. Виник як альтернатива J2EE платформи. Для побудови мікросервісів найважливіші модулі Spring це Spring Boot та Spring Cloud.

Spring Boot – складова екосистеми бібліотеки Spring. Spring Boot дозволяє легко створювати повноцінні, ефективні Spring-додатки. Для налаштування програмної системи необхідно відносно мало конфігурацій, в порівнянні з Spring MVC.

Основні особливості Spring Boot:

- створення повноцінних Spring додатків;
- вбудований сервер Tomcat або Jetty;
- автоматична конфігурація Spring framework;
- забезпечує можливостями моніторингу стану системи;
- не вимагає написання конфігураційних файлів на XML [14].

Spring Boot є основою для більш складного фреймворку Spring Cloud, який призначений для створення розподілених систем та має широкий інструментарій для розробки.

З основних переваг використання Spring Cloud виділимо наступні:

- наявність реєстру сервісів та системи виявлення сервісів, маршрутизації, міжсервісних викликів;
- балансування навантаження;
- наявність автоматичного вимикача;
- розподілений обмін повідомленнями.

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Рисунок 3.13 – Приклад створення програми за допомогою

Spring Cloud надає декларативний підхід до створення програмного забезпечення (рис 3.13). В даному прикладі використовується лише єдина анотація *@EnableDiscoveryClient* для введення в систему механізму виявлення сервісів. Також Spring Cloud має чудову інтеграцію з Netflix OSS, та надає дані інструменти «з коробки»:

```
@SpringBootApplication
@EnableEurikaServer
public class ServiceRegistrationServer {
    public static void main(String[] args) {
        System.setProperty("spring-config.yml",
                "registration-server");
        SpringApplication.run(ServiceRegistrationServer.class, args);
    }
}
```

Рисунок 3.14 – Інтеграція з Netflix OSS

Вище наведено приклад використання компоненту Netflix OSS через Spring Cloud. Eurika – це сервер, який виконує функцію реєстрації сервісів.

Для включення балансувальника навантаження також необхідно докласти мінімально зусиль, а саме включити анотацію *@LoadBalanced* над відповідним Spring-компонентом.

```
@Autowired
@LoadBalanced
protected RestTemplate restTemplate;
```

Рисунок 3.15 – Включення балансувальника навантаження

Також Spring Cloud надає горизонтально масштабоване сховище конфігурацій для розподілених систем. Як джерело даних на даний момент

підтримується Git, Subversion та прості файли, що зберігаються локально. За замовчуванням Spring Cloud Config віддає файли, які відповідають імені викликаючого Spring додатку.

Spring Cloud надає зручні анотації та автоконфігурації для забезпечення аутентифікації, створення токенів OAuth2 для доступу до ресурсів бекенду. Spring Cloud спрощує підключення до сервісів та отримання можливостей середовища в хмарних платформах, таких як Cloud Foundry і Heroku. Особлива підтримка Spring-додатків через Java і XML-конфігурації робить підключення до хмарних сервісів тривіальним завданням. Ви можете використовувати існуючі хмарні коннектори або написати власний для вашої хмарної платформи. "З коробки" підтримуються найбільш популярні сервіси (реляційні СУБД, MongoDB, Redis, Rabbit), але також можливе розширення для ваших сервісів. Жоден з сервісів не вимагає зміни самого Spring Cloud, досить просто додати необхідну вам jar бібліотеку в область видимості classpath [14].

ВИСНОВКИ

Під час написання дипломної роботи було досліджено основні ідеї побудови програмних систем на базі мікросервісної архітектури. Були розглянуті основні переваги та недоліки даного типу архітектури а також деякі особливості побудови мікросервісної програмної системи. Була проаналізована мікросервісна архітектура у порівнянні з найпопулярнішим типом архітектури - моноліт. Було зроблено висновок, що рекомендовано застосовувати моноліт, якщо це невелика програма, стартап або некомерційний проект. Було розглянуто як з ростом складності програми та її функціоналу моноліт стає все менш ефективним. Оптимальним рішенням у такому випадку буде сервіс-орієнтована архітектура, прикладом якої є мікросервіси.

В даній роботу було розглянуто основні принципи проектування мікросервісних програмних систем, деякі ключові моменти та основні паттерни для успішного розгортання MSA. Було детально розглянуто деякі компоненти мікросервісної архітектура, до яких відносяться API Gateway, балансувальник навантаження, автовимикач. Значну роль відіграє у розробці програмної системи необхідне середовище, яке забезпечується системами віртуалізації та постійної інтеграції. Було розглянуто систему автоматичного контейнерного розгортання Docker. Були продемонстровані ключові моменти безпеки мікросервіснох архітектури. Було розглянуто масштабування мікросервісів, мета, проблеми, паттерни їх вирішення. У роботі було розглянуто як саме створювати мікросервісну архітектуру, деякі підходи для переходу з моноліту на мікросервіси.

Як підсумок можна визначити наступне: мікросервісна архітектура не є панацея, але вирішує широке коло проблем, які не міг вирішити моноліт. Мікросервіси є більш гнучкий спосіб представлення програмного забезпечення, але мікросервіси не слід використовувати без аналізу предметної області. Мікросервісна архітектура стає дедалі все популярніше, багато провідних компаній використовують мікросервіси для розробки своїх продуктів.

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. М. Фаулер. Архитектура корпоративных программных приложений / М. Фаулер. – Издательский дом Вильямс, 2006 – 544 с.
2. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.: Питер, 2016 – 304 с.
3. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.
4. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74 p.
5. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans – Addison-Wesley, 2003 – 560 p.
6. Martin Fowler – Microservices – Режим доступу: http://martinfowler.com/articles/microservices.html – Дата доступу: 28.05.2019
7. Nadareishvili. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O'Reilly Media, 2016 – 146 p.
8. Introduction to microservices. – Режим доступу: https://nginx.com/blog/introduction-to-microservices/ – Дата доступу: 29.05.2019
9. Using an API Gateway. – Режим доступу: https://nginx.com/blog/buildingmicroservices-using-an-api-gateway/ – Дата доступу: 27.05.2019
10. Service Discovery. – Режим доступу: https://nginx.com/blog/service-discovery-ina-microservices-architecture/ – Дата доступу: 27.05.2019
11. Офіційний сайт Docker. – Режим доступу: https://docker.com/ – Дата доступу: 27.05.2019
12. Офіційний сайт C++ Micro Services. – Режим доступу: http://cppmicroservices.org/ – Дата доступу: 29.05.2017

13. Офіційний сайт Pistache framework. – Режим доступу: http://pistache.io/ – Дата доступу: 29.05.2019

14. Офіційний сайт Spring framework. – Режим доступу: https://spring.io – Дата доступу: 29.05.2019

15. Офіційний сайт Spark framework. – Режим доступу: https://sparkjava.com/ – Дата доступу: 29.05.2019

16. Офіційний сайт Restlet. – Режим доступу: https://restlet.com/ – Дата доступу: 29.05.2017

17. Офіційний сайт Flask. – Режим доступу: http://flask.pocoo.org/ – Дата доступу: 29.05.2017 18. Офіційний сайт Tornado. – Режим доступу: http://tornadoweb.org/ – Дата доступу: 29.05.2019

18. Офіційний сайт Nameko. – Режим доступу: https://nameko.readthedocs.io – Дата доступу: 29.05.2019

19. Закон України «Про охорону праці» від 14.10.1992 № 2694 – XII

20. В.І. Голінько Основи охорони праці: підручник / В.І. Голінько – Д.: НГУ, 2014. – 271 с

21. ДСаНПіН «Гігієнічна класифікація праці за показниками шкідливості та небезпечності факторів виробничого середовища, важкості ті напруженості трудового процесу», МОЗУ Наказ від 08.04.2014 №248.

22. ДСН 3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень» від 01.12.1999.

23. ДБН В.2.5-28-2006. Державні будівельні норми. Природне і штучне освітлення. К.: Мінбуд України,-Київ: 2006.

24. В.В. Березуцький Основи професійної безпеки та здоров’я людини: підручник / В.В. Березуцький – Харків: НТУ «ХПІ», 2018. – 553 с.

25. СанПіН 2.2.2/2.4.1340-03 «Гігієнічні вимоги до персональних електронно-обчислювальних машин та організації роботи» зі змінами від 21.07.2016.

26. ДСТУ Б В.1.1-36:2016 «Визначення категорій приміщень, будинків та зовнішніх установок за вибухопожежною та пожежною небезпекою» від 01.01.2017.

27. В.Ц. Жидецький Основи охорони праці / В.Ц. Жидецький – Львів: Афіша, 2004. – 250 с.

28. Закон України «Про охорону навколишнього природного середовища» № 1264-XII від 25 червня 1991 року.

29. Отходы ПК и компьютерной техники: как правильно утилизировать [Електронний ресурс]. – Режим доступу: https://ecologia.life/othody/tehnika/kompyuternaya-tehnika.html (дата звернення 04.05.2019). – Назва з екрану

30. ДСаНПіН 3.3.6-069-2002 Державні санітарні норми і правила при роботі з джерелами електромагнітних полів // Затвердженно наказом МОЗУ від 18.12.2002 №476.

31. ДСТУ ISO 9241-5;2004 Ергономічні вимоги до роботи з відеотерміналами в офісі. Частина 5. Вимоги до компонування робочого місця та до робочої пози. – Чинний від 01.01.2006.

32. ДСТУ Б. В.1.1-36:2016 Національний стандарт України. Норми визначення категорій приміщень, будинків та зовнішніх установок за вибухопожежною та пожежною небезпекою. – К.: Мінрегіонбуд України, 2016.

33. Правила улаштування електроустановок. – Чинний з 20.11.2014. Затв. Наказом Міністра енергетики та вугільної промисловості України від 20 червня 2014 р. №469.