МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу

Кафедра інженерії програмного забезпечення

# Пояснювальна записка до дипломної роботи

магістра
(освітній ступінь)

на тему «Експериментальне дослідження технологій будування складних систем електронної комерції»

ХАІ.603.667п1.121.156348.20О

Виконав: студент 6 курсу групи № 667п1
Спеціальність 121 – Інженерія програмного забезпечення
(код та найменування)
Освітня програма Хмарні обчислення та Інтернет речей
(найменування)
Романчук М.В.
(прізвище й ініціали студента)
Керівник Манжос Ю.С.
(прізвище та ініціали)
Рецензент Барковська О.Ю.
(прізвище та ініціали)

Харків – 2020

**Міністерство світи і науки України**
**Національний аерокосмічний університет ім. М. Є. Жуковського**
**«Харківський авіаційний інститут»**

Факультет програмної інженерії та бізнесу
(повне найменування)
Кафедра інженерії програмного забезпечення
(повне найменування)
Рівень вищої освіти другий (магістерський)

Спеціальність 121 – інженерія програмного забезпечення
(код та найменування)
Освітня програма хмарні обчислення та Інтернет речей
(найменування)

**ЗАТВЕРДЖУЮ**
**Завідувач кафедри**
_____ _____І. Б. Туркін_____
(підпис) (ініціали та прізвище)
“_____” _____ 2020 року

# З А В Д А Н Н Я
## НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

*Романчуку Максиму Віталійовичу*
(прізвище, ім’я, по батькові)
1. Тема дипломної роботи *Експериментальне дослідження технологій будування складних систем електронної комерції*
керівник дипломної роботи *Манжос Юрій Семенович, к.т.н., доцент*
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від “__” ___2020 року № __
2. Термін подання студентом роботи _____
3. Вихідні дані до роботи: *система електронної комерції зі застосуванням мікросервісної архітектури та мікросервіс каталогу продуктів на базі Google Cloud Platform.*
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
*провести огляд та аналіз існуючих системи електронної комерції;провести критичний огляд архітектурних стилів та системних інтеграторів, які полегшують управління, розгортання, зберігання даних, моніторинг для систем електронної комерції; розробити математичну модель інтегратора e-commerce додатка для проведення експериментальних досліджень;провести експериментальні дослідження з порівняння архітектур для побудови складних систем електронної комерції;провести аналіз отриманих результатів експериментального дослідження*

5. Перелік графічного матеріалу
*РПЗ – стор. 82, рисунків – 11 шт., таблиць – 5 шт., презентація – 15 слайдів.*

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| 1 | Манжос Ю.С., доц. каф. 603 | | |
| 2 | Манжос Ю.С., доц. каф. 603 | | |
| 3 | Манжос Ю.С., доц. каф. 603 | | |

8. Нормоконтроль _____ __В.А. Постернакова__ «____» _____2020 р.
(підпис) (ініціали та прізвище)

7. Дата видачі завдання _____

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів дипломного проекту | Строк виконання етапів проекту | Примітка |
|---|---|---|---|
| 1 | Отримання і затвердження теми диплому | 03.09.2019 | |
| 2 | Аналіз предметної області | 04.09.2019 | |
| 3 | Постановка задачі | 20.11.2019 | |
| 4 | Проведення теоретичних досліджень | 22.11.2019 | |
| 5 | Розробка прототипу ПЗ | 02.09.2020 | |
| 6 | Підготовка пояснювальної записки | 22.10.2020 | |
| 7 | Оформлення пояснювальної записки до дипломної роботи | 10.11.2020 | |
| 8 | Передзахист дипломної роботи | 27.11.2020 | |
| 9 | Захист дипломної роботи | 16.12.2020 | |

**Студент** _____ ___Романчук М.В._____
( підпис ) (прізвище та ініціали)

**Керівник роботи** _____ ___Манжос Ю.С.___
( підпис ) (прізвище та ініціали)

# РЕФЕРАТ

Пояснювальна записка до дипломної роботи містить 82 стор., 11 рис., 1 додаток, 15 джерел.

*Об'єкт дослідження* – технології будування складних систем електронної комерції.

*Предмет дослідження* – архітектури складних систем електронної комерції.

*Метою роботи* є підвищення ефективності розробки та підтримки складних систем електронної комерції шляхом проектування оптимальної архітектури для побудови складної системи електронної комерції з точки зору швидкості роботи, надійності та безпеки.

*Методи досліджень.* У роботі було використано методи аналізу, синтезу, системного аналізу, порівняння та логічного узагальнення результатів. Методи розробки базуються на технології Java, Spring фреймворк, MySQL база даних, Google Cloud Platform.

*Наукова новизна*. Удосконалено метод побудови складних систем електронної комерції, який на відміну від існуючих оснований на мікросервісній архітектурі, що дасть змогу спростити процес розробки та пришвидшити релізний цикл складних систем електронної комерції.

*Практична значимість отриманих результатів*. Результатом роботи є спроектована система електронної комерції зі застосуванням мікросервісної архітектури та розроблений мікросервіс каталогу продуктів на базі Google Cloud Platform.

МІКРОСЕРВІСНОЇ АРХІТЕКТУРА, СИСТЕМИ ЕЛЕКТРОННОЇ КОМЕРЦІЇ, GOOGLE CLOUD PLATFORM, МІКРОСЕРВІС

# ABSTRACT

Explanatory note to the master's thesis 82 pp., 11 fig., 1 app., 15 sources.

The object of research - technologies for building complex e-commerce systems.

The subject of research - the architecture of complex e-commerce systems.

The aim of the work is to increase the efficiency of development and maintenance of complex e-commerce systems by designing the optimal architecture for building a complex e-commerce system in terms of speed, reliability and security.

Research methods. The methods of analysis, synthesis, system analysis, comparison and logical generalization of results were used in the work. Development methods are based on Java technology, Spring framework, MySQL database, Google Cloud Platform.

Scientific novelty. The method of building complex e-commerce systems has been improved, which, unlike the existing ones, is based on a microservice architecture, which will simplify the development process and speed up the release cycle of complex e-commerce systems.

The practical significance of the obtained results. The result is a designed e-commerce system using microservice architecture and developed a microservice product catalog based on Google Cloud Platform.

MICROSERVICE ARCHITECTURE, E-COMMERCE SYSTEMS, GOOGLE CLOUD PLATFORM, MICROSERVICE

**ЗМІСТ**

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API - Application Programming Interface (прикладний програмний інтерфейс).

AWS - Amazon Web Services GCP - Google Cloud Platform.

HTML - HyperText Markup Language (мова розмітки гіпертекстових документів).

PaaS - Platform as a service (модель надання хмарних обчислень).

REST - Representational State Transfer (підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів).

IT - Information Technologies (інформаційні технології).

ПЗ - програмне забезпечення.

# ВСТУП

Електронна комерція — це діяльність з купівлі або продажу товарів в онлайн-службах або через Інтернет. Електронна комерція спирається на такі технології, як мобільна комерція, електронний переказ коштів, управління ланцюгами постачання, інтернет-маркетинг, обробка онлайн-транзакцій, системи управління складами та автоматизовані системи збору даних.

Підприємства електронної комерції можуть надавати такі послуги:

– інтернет магазин для роздрібних продажів безпосередньо споживачам через веб-сайти та мобільні додатки;

– надання користувачу можливість участі в онлайн-ринках, які потрібні для продажу товарів від третіх сторін до бізнесу або від споживача до споживача;

– купівля та продаж товарів від бізнесу до бізнесу;

– маркетинг для потенційних і зареєстрованих клієнтів електронною поштою;

– онлайн-фінансові біржі для обміну валют або торговельних цілей.

Усі наведені послуги можуть бути повністю реалізовані як веб-додатки та бути розгорнуті у мережі Інтернет. Тому важливим питанням є саме побудова такого веб-додатку для великих підприємств, який би був довговічним та надійним, а його побудова була швидкою та успішною.

У сучасному світі спостерігається тенденція значного збільшення торгівлі онлайн у порівнянні з фізичними магазинами. Багато великих компаній закривають свої фізичні магазини у зв'язку з нерентабельністю та концентруються на розвитку онлайн продажів. Системи електронної комерції стають все більш розумними та багатими функціонально. У них є набагато більше потенційних можливостей спонукати потенціального покупця придбати товар завдяки можливості збору інформації та доступу до різних каналів зв'язку із покупцем, що в результаті значно підвищує прибуток.

Для сучасних комерційних компаній також дуже важливо швидко реагувати на зміни на ринку, скоріше, ніж їхні конкуренти. Тому при проектуванні систем електронної комерції також дуже важливо враховувати можливість максимально швидко та безпечно до системи додавати новий функціонал.

*Об'єктом дослідження* – технології будування складних систем електронної комерції.

*Предмет дослідження* – архітектури складних систем електронної комерції.

*Метою роботи* є підвищення ефективності розробки та підтримки складних систем електронної комерції шляхом проектування оптимальної архітектури для побудови складної системи електронної комерції з точки зору швидкості роботи, надійності та безпеки.

Для досягнення поставленої мети необхідно вирішити ряд завдань:

– провести огляд та аналіз існуючих системи електронної комерції;

– провести критичний огляд архітектурних стилів та системних інтеграторів, які полегшують управління, розгортання, зберігання даних, моніторинг для систем електронної комерції.

– розробити математичну модель інтегратора e-commerce додатка для проведення експериментальних досліджень;

– провести експериментальні дослідження з порівняння архітектур для побудови складних систем електронної комерції;

– провести аналіз отриманих результатів експериментального дослідження.

*Методи досліджень.* У роботі було використано методи аналізу, синтезу, системного аналізу, порівняння та логічного узагальнення результатів. Методи розробки базуються на технології Java, Spring фреймворк, MySQL база даних, Google Cloud Platform.

*Наукова новизна*. Удосконалено метод побудови складних систем електронної комерції, який на відміну від існуючих оснований на мікросервісній архітектурі, що дасть змогу спростити процес розробки та пришвидшити релізний цикл складних систем електронної комерції.

*Практична значимість отриманих результатів*. Результатом роботи є спроектована система електронної комерції зі застосуванням мікросервісної архітектури та розроблений мікросервіс каталогу продуктів на базі Google Cloud Platform.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

## 1.1 Постановка мети й завдань дослідження

Метою роботи є підвищення ефективності розробки та підтримки складних систем електронної комерції шляхом проектування оптимальної архітектури для побудови складної системи електронної комерції з точки зору швидкості роботи, надійності та безопасності.

Для досягнення поставленої мети необхідно вирішити ряд завдань:

– провести огляд та аналіз існуючих системи електронної комерції;

– провести критичний огляд архітектурних стилів та системних інтеграторів, які полегшують управління, розгортання, зберігання даних, моніторинг для систем електронної комерції.

– розробити математичну модель інтегратора e-commerce додатка для проведення експериментальних досліджень;

– провести експериментальні дослідження з порівняння архітектур для побудови складних систем електронної комерції;

– провести аналіз отриманих результатів експериментального дослідження.

## 1.2 Важливість електронної комерції у сучасному світі

Електронна комерція — це сфера цифрової економіки, що включає всі фінансові та торгові транзакції, які проводяться за допомогою комп'ютерних мереж, та бізнес-процеси, пов'язані з проведенням цих транзакцій.

Серед країн, що розвиваються, присутність Китаю в електронній торгівлі продовжує розширюватися з кожним роком. У 2015 році в Китаї було 600 мільйонів користувачів Інтернету (удвічі більше, ніж у США), що зробило його найбільшим онлайн-ринком у світі. Китай також є найбільшим ринком

електронної комерції у світі за величиною продажів, а в 2016 році оцінюється в 899 млрд. Доларів США. Електронна комерція є найшвидше зростаючим ринком роздрібної торгівлі, і, за оцінками, вона сягне понад 4 трлн у 2020 році.

Недавні дослідження чітко вказують на те, що електронна комерція в даний час є найбільш популярним способом, яким люди купують продукцію. Ринок електронної комерції також набув великої популярності серед західних країн, зокрема, Європи та США. Ці країни характеризувалися споживчими товарами.

Індія має групу користувачів Інтернету близько 460 мільйонів станом на грудень 2017 року. Незважаючи на те, що за кількістю користувачів мережі Інтернет Індія посідає третю позицію в світі, порівняно з такими ринками, як США, Великобританія або Франція, але зростає значно швидше, додаючи близько 6 мільйонів нових користувачів щомісяця. Індійський ринок роздрібної торгівлі, як очікується, зросте з 2,5% у 2016 році до 5% у 2020 році.

Електронна комерція стала важливим інструментом для малих і великих підприємств у всьому світі - не лише для продажу товарів клієнтам, але й для їх залучення нових покупців.

Для традиційних підприємств одне дослідження заявило, що інформаційні технології та транскордонна електронна торгівля є гарною можливістю для швидкого розвитку та зростання підприємств. Багато компаній вклали величезні обсяги інвестицій в мобільні додатки. Немає ніяких обмежень у часі та просторі, є більше можливостей для спілкування з клієнтами по всьому світу, а також для скорочення непотрібних проміжних зв'язків, тим самим знижуючи собівартість.

Ринки електронної комерції зростають помітними темпами. Очікується, що онлайн-ринок зросте на 56% у 2015-2020 роках. У 2017 році роздрібні продажі електронної комерції в усьому світі становили 2,3 трлн доларів США. Традиційні ринки лише очікують зростання на 2% за той же час. Багато великих роздрібних підприємств можуть підтримувати як фізичні магазини так і магазини в мережі Інтернет, пов'язуючи фізичні та мережеві пропозиції.

Електронна комерція дозволяє клієнтам долати географічні бар'єри і дозволяє їм купувати продукти в будь-який час і з будь-якого місця. Інтернет і традиційні ринки мають різні стратегії ведення бізнесу. Традиційні роздрібні торговці пропонують менший асортимент продукції через місця на полицях, де інтернет-магазини часто не мають інвентаризації, але відправляють замовлення клієнтів безпосередньо на виробництво. Стратегії ціноутворення також відрізняються для традиційних і інтернет-магазинів. Традиційні роздрібні торговці базують свої ціни на трафіку в магазинах і вартість зберігання запасів. Інтернет-магазини роздрібної торгівлі базують ціни на швидкість доставки.

Для маркетологів існує два способи ведення бізнесу через електронну комерцію: повністю онлайн або онлайн, разом зі складом та фізичним магазином. Інтернет-маркетологи можуть запропонувати більш низькі ціни, більший вибір продукції та високу ефективність. Багато клієнтів віддають перевагу онлайн ринкам, якщо продукти можна швидко доставити за відносно низькою ціною. Тим не менш, інтернет-магазини не можуть запропонувати ефект присутності, який можуть надати традиційні роздрібні магазини. Іншим питанням, що стосується онлайн-ринку, є занепокоєння щодо безпеки онлайн-транзакцій.

Безпека є основною проблемою для електронної торгівлі в розвинених і в країнах, які розвиваються. Безпека електронної комерції захищає веб-сайти та клієнтів від несанкціонованого доступу, використання, зміни чи знищення їх даних. До типів загроз можна віднести: шкідливі коди, небажані програми, фішинг, хакерство та кібер-вандалізм. Веб-сайти електронної комерції використовують різні інструменти для запобігання загроз. Ці інструменти включають брандмауери, програмне забезпечення для шифрування, цифрові сертифікати та паролі.

Інтернет-магазин — це форма електронної комерції, яка дозволяє споживачам безпосередньо купувати товари або послуги від продавця через Інтернет за допомогою веб-браузера. Споживачі знаходять продукт, який їм

цікавий, безпосередньо або відвідуючи веб-сайт роздрібного продавця, або шукаючи серед альтернативних постачальників, використовуючи пошукову систему для покупок, яка відображає доступність та цінову оцінку того ж продукту в різних електронних магазинах. Станом на 2016 рік клієнти можуть робити покупки в Інтернеті, використовуючи різні комп'ютери та пристрої, включаючи настільні комп'ютери, ноутбуки, планшетні комп'ютери та смартфони.

Інтернет-магазини зазвичай дозволяють покупцям використовувати функції "пошуку" для пошуку конкретних моделей, брендів або предметів. Онлайн- клієнти повинні мати доступ до Інтернету та дійсний спосіб оплати для завершення операції, наприклад, кредитної картки або дебетової картки.

Інтернет-магазини зазвичай доступні цілодобово, і багато споживачів у західних країнах мають доступ до Інтернету як на роботі, так і вдома. Інші установи, такі як інтернет-кафе, громадські центри та школи, також надають доступ до Інтернету.

Конфіденційність особистої інформації є важливим питанням для більшості споживачів. Багато споживачів хочуть уникнути спаму та телемаркетингу, які можуть виникнути в результаті постачання контактної інформації в мережі Інтернет. У відповідь на це багато торговців обіцяють не використовувати споживчу інформацію для цих цілей. Багато веб-сайтів відстежують звички споживачів покупок, щоб пропонувати предмети та інші веб-сайти для перегляду. Деякі звертаються за адресою та номером телефону покупця під час процесу покупки, хоча споживачі можуть відмовитися від її надання. Багато великих магазинів використовують адресну інформацію, закодовану на кредитних картах споживачів (часто без їхнього знання), щоб додати їх до списку розсилки каталогу. Тому питання безпеки персональної інформації є також дуже важливим при побудові веб-додатків для електронної комерції.

Побудова систем для електронної комерції є ключовим фактором успішності бізнесу, адже більшість, або навіть усі операції проводяться у мережі Інтернет. Тому розроблені додатки мають бути надійними, швидко працювати, а також мають бути безпечними, як для покупців, так і для продавців.

### 1.3 Порівняння монолітної та мікросервісної архітектур

Монолітна архітектура вважається традиційним способом побудови додатків. Монолітна програма побудована як єдина і неподільна одиниця. Зазвичай, таке рішення містить користувальницький інтерфейс клієнта, серверний додаток і базу даних. Він єдиний і всі функції управляються і обслуговуються в одному місці.

Як правило, монолітні програми мають одну велику кодову базу і відсутність модульності. Якщо розробники хочуть щось оновити або змінити, вони отримують доступ до тієї ж кодової бази. Таким чином, вони вносять зміни відразу до всього стека. Архітектура типового монолітного додатку для електронної комерції наведена на рисунку 1.1.


Рисунок 1.1 - Типова архітектура монолітного додатку для електронної комерції

На відміну від архітектури мікросервісів, монолітні програми набагато легше налагоджувати і тестувати. Оскільки монолітне додаток є єдиною неподільною одиницею, ви можете запускати наскрізне тестування набагато швидше.

Ще однією перевагою, пов'язаною з простотою монолітних додатків, є простіше розгортання. Коли мова йде про монолітні додатки, вам не доведеться обробляти багато розгортань - лише один файл або каталог.

До тих пір, поки монолітний підхід є стандартним способом побудови додатків, будь-яка інженерна команда має відповідні знання та можливості для розробки монолітного застосування.

Але коли монолітне застосування збільшується, це стає занадто складним для розуміння. Важко здійснити зміни в такому великому і складному застосуванні з дуже щільним зв'язком. Будь-яка зміна коду впливає на всю систему, тому вона повинна бути ретельно скоординована. Це робить процес загального розвитку набагато довшим.

Ви не можете масштабувати компоненти самостійно, тільки всю програму в цілому.

Надзвичайно проблематично застосовувати нову технологію в монолітному додатку, тому що тоді вся програма повинна бути переписана.

При розробці програмного додатку системи електронної комерції більшість додатків складаються з декількох рівнів:

- презентаційний рівень - відповідає за обробку HTTP-запитів і відповідей за допомогою HTML або JSON / XML (для API веб-служб);
- рівень бізнес-логіки - бізнес-логіка програми;
- рівень доступ до бази даних - об'єкти доступу до даних, відповідальні за доступ до бази даних;!
- рівень інтеграцій з іншими додатками - інтеграція з іншими службами (наприклад, через повідомлення або REST API).

Незважаючи на логічну модульну архітектуру, програма упаковується і розгортається як монолітний додаток.

Переваги монолітної архітектури:

- проста у розвитку;

- проста для перевірки. Наприклад, ви можете реалізувати комплексне тестування, просто запустивши програму та перевіривши інтерфейс користувача за допомогою Selenium;

- просте розгортання. Вам потрібно просто скопіювати упакований додаток на сервер;

- проста для масштабування по горизонталі шляхом запуску декількох копій за балансування навантаження;

- на ранніх стадіях проекту вона працює добре, і в основному більшість великих і успішних додатків, які існують сьогодні, почалися як моноліт.!

Недоліки монолітної архітектури:

- цей простий підхід має обмеження в розмірах і складності;!

- додаток занадто великий і складний, щоб повністю зрозуміти і внести зміни швидко і правильно;!

- розмір програми може сповільнити час запуску;!

- потрібно розгорнути весь додаток на кожне оновлення;!

- вплив змін, як правило, не дуже добре зрозумілий, що призводить до великого ручного тестування;!

- важко забезпечити постійне розгортання додатку;

- монолітні програми важко масштабувати, коли різні модулі мають конфліктні вимоги до ресурсів;

- надійність. Помилка в будь-якому модулі (наприклад, втрата пам'яті) може потенційно збити весь процес. Більш того, оскільки всі екземпляри програми ідентичні, ця помилка вплине на доступність всієї системи;

- монолітні додатки мають бар'єр для впровадження нових технологій.

Оскільки зміни в фреймворках або мовах впливатимуть на всю програму, це надзвичайно коштовно і в часі, і в витратах.

Якщо монолітна програма є єдиною уніфікованою одиницею, то архітектура мікросервісів розбиває її на набір менших незалежних одиниць. Ці

підрозділи здійснюють кожен процес як окрему послугу. Тому всі послуги мають свою логіку і базу даних, а також виконують певні функції - коротше кажучи, архітектурний стиль мікросервісу - це підхід до розробки єдиного додатка як набору невеликих послуг, кожен з яких працює у власному процесі і спілкується з легкими механізмами, зазвичай API ресурсів HTTP.

В архітектурі мікросервісів вся функціональність розбивається на незалежні модулі, які взаємодіють один з одним за допомогою визначених методів, які називаються API (інтерфейси прикладного програмування). Кожна послуга охоплює свою власну сферу і може бути оновлена, розгорнута і масштабована незалежно. Структурна схема типового мікросервісного додатку показана на рисунку 1.2:

Всі послуги можуть бути розгорнуті та оновлені самостійно, що дає більше гнучкості. Помилка в одному мікросервісі впливає тільки на конкретну послугу і не впливає на всю програму. Крім того, набагато простіше додати нові функції до додатку мікросервісу, ніж монолітні.

Розбивши на більш дрібні та прості компоненти, додаток мікросервісу легше зрозуміти та їм керувати. Ви просто зосереджуєтеся на конкретній послузі, яка пов'язана з вашою бізнес-ціллю.

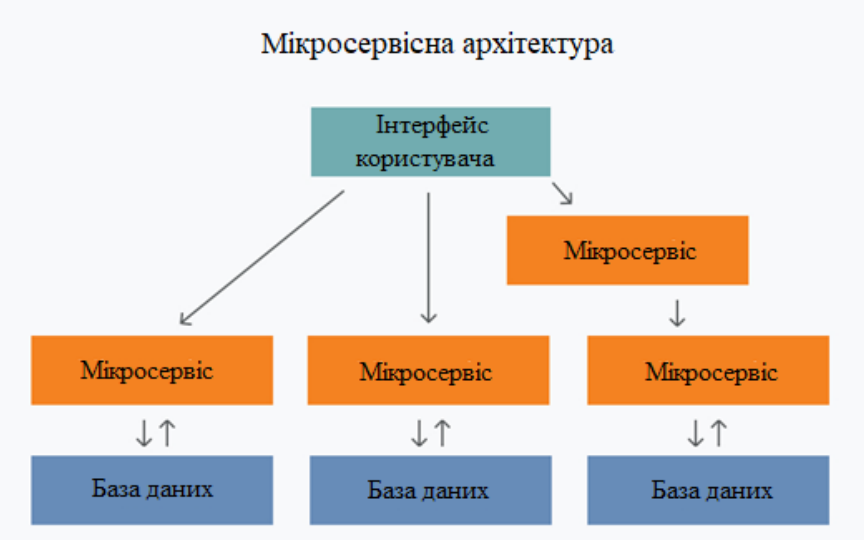


Рисунок 1.2 - Структурна схема типового мікросервісного додатку

Іншою перевагою підходу мікросервісів є те, що кожен елемент можна масштабувати незалежно. Таким чином, весь процес є більш економічним та ефективним у часі, ніж з монолітами, коли всю програму потрібно масштабувати, навіть якщо в ній немає необхідності. Крім того, кожен моноліт має межі масштабування, тому чим більше користувачів ви маєте, тим більше проблем ви маєте з вашим монолітом. Тому багато компаній в кінцевому підсумку перебудовують свої монолітні архітектури.

Інженерні команди не обмежені технологією, обраною з самого початку. Вони можуть вільно застосовувати різні технології та бібліотеки коду для кожного мікросервісу.

Будь-яка помилка в застосуванні мікросервісів впливає тільки на конкретну послугу, а не на все рішення. Тому всі зміни і експерименти реалізуються з меншими ризиками і меншою кількістю помилок.

Оскільки архітектура мікросервісів є розподіленою системою, ви повинні вибрати і встановити з'єднання між усіма модулями і базами даних. Архітектура мікросервісів - це складна система з декількох модулів і баз даних, тому всі з'єднання повинні бути ретельно оброблені.

Створюючи додаток для мікросервісів, вам доведеться мати справу з низкою проблем. Вони включають зовнішню конфігурацію, реєстрацію, показники, перевірки здоров'я та інші. Безліч самостійно розгортаються компонентів робить тестування рішення на основі мікросервісів набагато складнішим.

Ідея полягає в тому, щоб розділити вашу програму на набір менших, взаємопов'язаних служб, замість створення єдиного монолітного додатка. Кожен мікросервіс - це невелика програма, що має свою власну архітектуру, що складається з бізнес-логіки та різних адаптерів. Деякі мікросервіси мають REST, RPC або API на основі повідомлень, а більшість сервісів споживають API, надані іншими службами. Інші мікросервіси можуть реалізовувати веб-інтерфейс.

Схема архітектури мікросервісу значно впливає на взаємозв'язок між програмою та базою даних. Замість спільного використання однієї схеми бази

даних з іншими службами, кожна служба має власну схему бази даних. З одного боку, цей підхід суперечить ідеї моделі даних для всього підприємства. Також це часто призводить до дублювання деяких даних. Проте наявність схеми баз даних для кожного сервісу є важливим, якщо ви хочете скористатися мікросервісами, оскільки вони забезпечують вільний зв'язок. Кожна з служб має власну базу даних. Більш того, служба може використовувати тип бази даних, який найкраще підходить для його потреб, так звана архітектура поліглотів.

Деякі API також можуть адаптуватися для мобільних, настільних та веб-систем. Проте, ці споживачі не мають прямого доступу до внутрішніх сервісів. Натомість, комунікація здійснюється посередником, відомим як шлюз API. Шлюз API відповідає за такі завдання, як балансування навантаження, кешування, керування доступом, вимірювання API і моніторинг.

Переваги архітектури мікросервісів:

— вона вирішує проблему складності, розкладаючи додаток на набір керованих сервісів, які набагато швидше розвиваються, і набагато легше розуміти і підтримувати;

— це дає змогу кожному сервісу самостійно розвиватися командою, яка орієнтована на цей сервіс;

— це зменшує бар'єр прийняття нових технологій, оскільки розробники можуть вільно вибирати, які технології мають сенс для їхнього обслуговування, а не обмежені виборами, зробленими на початку проекту;

— мікросервісна архітектура дозволяє кожному мікросервісу розгортатися незалежно. Як результат, це робить можливим постійне розгортання для складних додатків;

— мікросервісна архітектура дозволяє кожній службі масштабуватися незалежно.

Недоліки архітектури мікросервісів:

— архітектура мікросервісів додає до проекту складність лише тим, що додаток мікросервісів є розподіленою системою. Необхідно вибрати і реалізувати механізм взаємодії між процесами, заснований на обміну

повідомленнями або RPC і писати код для обробки часткової несправності і враховувати інші помилки розподілених обчислень;!

— мікросервіси мають розділену архітектуру бази даних. Бізнес-операції, які оновлюють декілька суб'єктів в додатку, заснованому на мікросервісах, повинні оновлювати кілька баз даних, що належать різним службам. Використання розподілених транзакцій, як правило, не є можливим, і вам доводиться використовувати підхід, що базується на послідовності, що є більш складним для розробників;!

— тестування програми мікросервісів також набагато складніше, ніж у випадку монолітного додатка. Для подібного тесту для сервісу потрібно запустити цей сервіс та усі служби, від яких він залежить (або принаймні налаштувати заглушки для цих служб);

— більш важко реалізувати зміни, які охоплюють кілька служб. У монолітному додатку можна просто змінити відповідні модулі, інтегрувати зміни і розгорнути їх за один раз. У архітектурі мікросервісу необхідно ретельно планувати і координувати зміни в кожній службі;

— розгортання додатків на основі мікросервісів також є більш складним.

Монолітна програма просто розгортається на наборі ідентичних серверів за балансуванням навантаження. На відміну від цього, додаток мікросервісу зазвичай складається з великої кількості сервісів. Кожен сервіс буде мати кілька екземплярів середовища виконання. І кожен екземпляр повинен бути налаштований, розгорнутий, масштабований і постійно бути під контролем. Крім того, також потрібно реалізувати механізм виявлення сервісу. Ручний підхід до операцій не може масштабуватися до такого рівня складності і успішне розгортання мікросервісу вимагає застосування високого рівня автоматизації.

### 1.4 Вимоги до систем електронної комерції

Для того щоб спроектувати систему електронної комерції використовуючи мікросервісний архітектурний стиль, треба виділити основні функціональні

області типового e-commerce додатка. В залежності від складності системи ці функціональні області потрібно бути розбити на менші сервіси та продумати зв'язки між ними.

Типова e-commerce система складається з таких областей: каталог продуктів, управління складськими запасами, обробка замовлення, корзина, персональний аккаунт покупця, обробка оплати замовлення, обробка повернення замовлення, організація доставки, управління кампаніямі (рис 1.3).

Розглянемо деякі з них більш детально:

— каталог продуктів. Він є невід'ємною частиною кожного e-commerce додатка. Чим більший асортимент магазину, тим складнішим може буде імплементація цією функціональної області. У великих магазинів продукт зазвичай розподіляється на декілька типів. Наприклад, у одягу є фіксовані розміри, а у деяких товарів покупець може задавати персональний розмір (тканини, штори у метрах). Деякі товари можна повернути згідно з законодавством, а деякі ні. І всі ці товари можуть потенційно продаватися в одному магазині, що може ускладнити відображення товарів покупцю та обробку операцій з товарами;

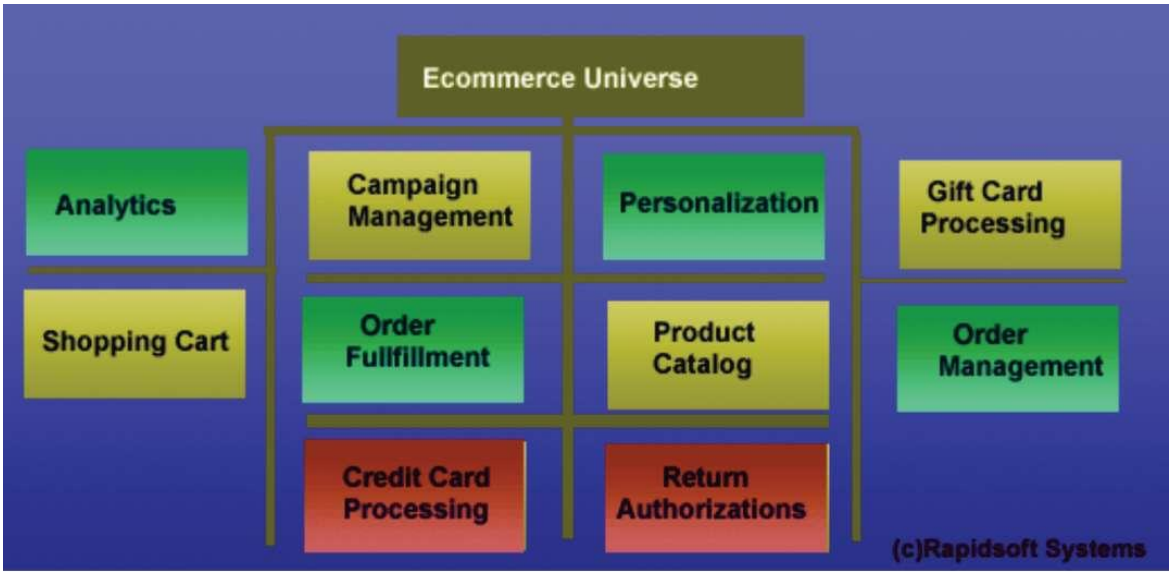


Рисунок 1.3 - Функціональні області у e-commerce системі

— управління складськими запасами. До цієї функціональної області можна віднести управління онлайн запасами, сповіщення покупців про товари, які знову в наявності. У разі якщо магазин перепродає товари інших продавців за управління складськими запасами може відповідати безпосередньо продавець, тому сюди також можна віднести синхронізацію даних про запаси з іншими продавцями;

— обробка замовлення. Є однією з найважливіших частин e-commerce систем та відповідає за резервацію товару на початку обробки вашого замовлення, збір інформації про адресу доставки та підбір наявних методів доставки згідно з нею, збір інформації про метод оплати замовлення та повторна спроба списання коштів у разі невдачі, відправка повідомлення у разі успішного замовлення. Обробка замовлення може бути різною в залежності від типу продукта;

— персональний аккаунт покупця. Включає зберігання даних про адресу покупця, його платіжну інформацію, історію замовлень, переліки побажань та інше;

— обробка оплати замовлення. Може включати в себе обробку оплати різними методами - кредитна картка, PayPal та інші платіжні системи, подарункові карти, кредити аккаунта, відкладена оплата та інші.

Сучасні системи електронної комерції знаходятся в більш жорстким умовах конкуренції, завдяки розповсюдження онлайн торгівлі та можливості замовляти товари з інших стран. Тому дуже важливо, щоб e-commerce система викликала довіру у користувача. Вона потрібна бути надійною та захищеною з точки зору зберігання чутливих даних користувача - це найголовніше. Не менш важливою є прозорість для користувача щодо списання коштів. Інформування користувача про стан його замовлення та інші операції, які він виконував, у різних каналах зв'язку надає йому відчуття спокою та викликає довіру. І звичайно система повинна бути швидкою та комфортною у користуванні.

## 1.5 Висновки по розділу 1

Електронна комерція є тою, що швидко розвивається та актуальною сферою, тому шо приносить великий прибуток. Обсяг даних, що треба обробити стає все більшим. Через це системи стають все більш складнішими та потребують нових архітектурних підходів для того, щоб відповідати вимогам бізнесу та споживачів.

Метою роботи є проектування оптимальної архітектури складної e-commerce системи з точки зору швидкості роботи, надійності, швидкості розробки, швидкості та ціни розгортання, вартості підтримки. Для цього є необхідним порівняти монолітну та мікросервісну архітектури, розглянути наявні на ринку хмарні інтегратори додатків, обрати оптимальний паттерн розгортання сервісу.

Для невеликих систем монолітна архітектура є більш прийнятною завдяки простішому розгортанню, процессу розробки та тестуванню. Але коли система зростає, час між релізами нової функціональності також зростає, що не є прийнятним. Мікросервісна архітектура допомагає вирішити цю проблему завдяки можливості робити окремі релізи для кожного з мікросервісів, тестувати та розгортати кожен із мікросервісів окремо. Також мікросервісна архітектура надає більшу надійність, тому що помилка у одному мікросервісі не може спричинити фатальну помилку усієї системи. Звичайно мікросервіси потребують більше початкових зусиль щодо розгортання та конфігурації, але результат, який ви можете отримати, виправдовує ці зусилля.

# 2 ПЛАНУВАННЯ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ З ПОРІВНЯННЯ АРХІТЕКТУР ПОБУДОВИ СКЛАДНИХ СИСТЕМ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

## 2.1 Аналіз стану розв'язання проблеми

Корпоративні програмні додатки розроблені для полегшення численних бізнес-вимог. Таким чином, дане програмне забезпечення пропонує сотні функцій, і всі такі функції зазвичай складаються в єдиний монолітний додаток. ERP, CRM та інші різноманітні програмні системи - хороші приклади - вони побудовані як моноліти з декількома сотнями функцій. Складність розгортання, усунення несправностей, масштабування та модернізація таких систем зростає з їх складністю.

Сервісно-орієнтована архітектура (SOA) була розроблена для подолання проблем, що виникають внаслідок монолітних додатків, шляхом запровадження концепції «сервіса». Таким чином, за допомогою SOA програмне забезпечення розробляється як комбінація сервісів. Концепція SOA не обмежує реалізацію сервісу як моноліт, але найпопулярніші веб-сервіси впровадження рекламують програмне забезпечення, яке буде реалізовуватися як моноліт, що включає грубі веб-сервіси, які працюють у одному і тому самому середовищу виконання. Подібно до монолітних програмних додатків, ці служби мають звичку зростати з часом, накопичуючи різні функції. Таке зростання незабаром перетворює ці додатки в декілька великих монолітів, які не відрізняються від звичайних монолітних застосувань.

Є кілька інших проблем з цим підходом. Моноліт повинен масштабуватися як єдиний додаток і його важко масштабувати за допомогою конфліктуючих вимог до ресурсів (наприклад, одна послуга, що вимагає більше ЦП, а інша вимагає більшої кількості пам'яті). Одна нестабільна послуга може призвести до зниження всього додатку, і в цілому важко впроваджувати нові технології та фреймворки.

Такі характеристики призвели до того, що з'явилася мікросервісна архітектура. Давайте розглянемо проблеми монолітної архітектури, які вирішують мікросервіси — відсутність міцних кордонів модуля, неможливість незалежного розгортання, відсутність технологічного різноманіття.

Першою великою перевагою мікросервісів є сильні межі модуля. Це є важливою перевагою хоча і дивний, тому що немає жодної причини, в теорії, чому мікросервіси повинні мати сильніші кордони модулів, ніж моноліт.

Отже, що мається на увазі під сильним кордоном модуля? Більшість людей погодиться, що це добре розділити програмне забезпечення на модулі: частини програмного забезпечення, які відокремлені один від одного. Ви хочете, щоб ваші модулі працювали так, що якщо мені потрібно змінити частину системи, то більшу частину часу потрібно витратити лише на те, щоб зрозуміти невелику частину цієї системи, щоб внести зміни, і цю невелику частину знайти досить легко. Хороша модульна структура корисна в будь-якій програмі, але стає експоненціально більш важливою, оскільки програмне забезпечення зростає в розмірах. Можливо, що важливіше, вона стає все більш важливою, оскільки команда, що розвивається, зростає.

Прихильники мікросервісів швидко впроваджують закон Конвея, уявлення про те, що структура програмної системи відображає комунікаційну структуру організації, яка її побудувала. З великими командами, особливо якщо ці команди розташовані в різних місцях, важливо структурувати програмне забезпечення, щоб визнати, що комунікації між командами будуть менш частими і більш формальними, ніж у команді. Мікросервіси дозволяють кожній команді доглядати за відносно незалежними одиницями з такою схемою спілкування.

Немає жодної причини, чому монолітна система не повинна мати хорошу модульну структуру. Але багато людей помітили, що це здається рідкісним, отже, "Великий клубок бруду" є найпоширенішою архітектурною схемою. Дійсно, це розчарування спільною долею монолітів — це те, що змусило декілька команд перейти до мікросервісів. Розділення на модулі працює, оскільки межі модуля є бар'єром для посилання між модулями. Біда в тому, що,

з монолітною системою, звичайно досить легко порушити бар'єр. Це може бути корисним тактикою для швидкого створення функцій, але це глобально підриває модульну структуру та знищує продуктивність команди. Виділення модулів в окремі сервіси робить межі жорсткішими, що ускладнює пошук цих злоякісних шляхів.

Важливим аспектом цього зв'язку є дані, що постійно зберігаються. Однією з ключових характеристик мікросервісів є Decentralized Data Management (Децентралізований менеджмент даних), який говорить, що кожен сервіс керує своєю власною базою даних, і будь-який інший сервіс повинна пройти через API сервісу, щоб потрапити до них. Це виключає такий анті-патерн як "Інтеграційні бази даних", які є основним джерелом неприємних зв'язків у великих системах.

Важливо підкреслити, що цілком можливо мати тверді модульні кордони з монолітом, але це вимагає дисципліни. Аналогічним чином ви можете отримати "Великий клубок мікросервісного бруду", але він вимагає більше зусиль, щоб зробити неправильну річ. Використання мікросервісів збільшує ймовірність того, що ви отримаєте кращу модульність. Якщо ви впевнені в дисципліні вашої команди, то це, ймовірно, виключає цю перевагу, але, як команда зростає, її стає все важче утримувати дисциплінованою, так само, як стає важливішим підтримувати межі модуля.

Ця перевага стає перешкодою, якщо ви не розумієте свої межі. Це одна з двох основних причин стратегії "Monolith First", і чому навіть ті, хто більш схильний працювати з мікросервісами з самого початку підкреслюють, що ви можете зробити це тільки з добре зрозумілим доменом.

Але можна тільки сказати, наскільки добре система зберегла модульність після того, як час пройшов. Таким чином, ми тільки можемо дійсно оцінити, чи призводять мікросервіси до кращої модульності, як тільки ми побачимо систему мікросервісів, яка існує щонайменше кілька років. Все, що можна надати на даний момент - це ранні докази від людей, хто використовував цей стиль. Їхнє судження полягає в тому, що значно простіше підтримувати свої модулі.

Ключовим принципом мікросервісів є те, що послуги є складовими і, таким чином, незалежно розгортаються. Так що тепер, коли ви робите зміни, ви повинні тільки перевірити і розгорнути невеликий сервіс. Якщо цей сервіс не буде працювати коректно, це не виведе з ладу всю систему. Зрештою, внаслідок необхідності обробки невдачі, навіть повна невдача вашого компонента не повинна заважати роботі інших частин системи, хоча і з деякою формою поступової деградації.

Маючи багато мікросервісів, які потребують частого розгортання, важливо, щоб ваше розгортання діяло разом. Саме тому швидке розгортання додатків та швидке надання інфраструктури є передумовами мікросервісів. Для всього, що виходить базового, вам потрібно робити безперервну доставку.

Великою перевагою безперервної доставки є скорочення часу циклу між ідеєю та запущеним програмним забезпеченням. Організації, які роблять це, можуть швидко реагувати на ринкові зміни та впроваджувати нові функції швидше, ніж їх конкуренція. Цей момент є дуже важливим для e-commerce додатків, тому що у зв'язку з їх специфікою реагувати на ринкові зміни для них потрібно швидче ніж їхні конкуренти для отримання достатнього прибутку.

Можливість самостійного розгортання послуг є частиною визначення мікросервісів. Отже, доцільно сказати, що набір послуг, які мають координовані розгортання, не є архітектурою мікросервісу. Доцільно також сказати, що багато команд, які намагаються створити архітектуру мікросервісу, потрапляють у неприємності, оскільки вони змушені координувати розгортання сервісів.

Оскільки кожна мікросервіс є самостійно розгортаючимся, у вас є значна свобода у виборі технології. Мікросервіси можуть бути написані на різних мовах, використовувати різні бібліотеки і використовувати різні сховища даних. Це дозволяє командам обирати відповідний інструмент для роботи, деякі мови та бібліотеки краще підходять для певних проблем.

Обговорення технічної різноманітності часто базується на найкращому інструменті для роботи, але найчастіше найбільшою перевагою мікросервісів є більш прозаїчна проблема версій. У моноліті можна використовувати лише одну

версію бібліотеки - це ситуація, яка часто призводить до проблемних оновлень. Одна частина системи може вимагати оновлення для використання нових функцій, але не може бути оновлена, тому що оновлення виводить з ладу іншу частину системи. Робота з проблемами бібліотечних версій є однією з тих проблем, які стають експоненціально складнішими, оскільки кодова база стає більшою.

Не варто недооцінювати цінність можливості експериментувати. З монолітною системою ранні рішення щодо мов і рамок важко змінити у подальшому. Через десятиліття такі рішення можуть заблокувати команди в незручних технологіях. Мікросервіси дозволяють командам експериментувати з новими інструментами, а також поступово переносити системи по одному сервісу за раз, якщо передові технології стають актуальними.

Основою архітектури мікросервісу (MSA) є розробка єдиної програми як набору малих і незалежних сервісів, які працюють у власному процесі, розробляються і розгортаються незалежно.

Більшість визначень MSA пояснюють це як архітектурну концепцію, зосереджену на розділенні сервісів, які є в моноліті, на набір незалежних послуг. Проте, мікросервісна архітектура полягають не тільки в тому, щоб розділити послуги, що надаються в моноліті, на незалежні сервіси.

Врахуємо це, розглянувши функції, запропоновані монолітом, визначивши бізнес-можливості, необхідні для програми, тобто те, що програма повинна робити, щоб бути корисною. Потім ці бізнес-можливості можуть бути реалізовані як повністю незалежні, дрібнозернисті та самодостатні (мікро) сервіси. Вони можуть бути реалізовані на базі різних технологічних стеків, але, незважаючи на це, кожен сервіс буде відповідати за дуже специфічний і обмежений обсяг діяльності.

Як ви можете бачити, на основі бізнес-вимог можна виділити додатковий мікросервіс, створений з оригінального набору послуг, які були там в моноліті. Отже, очевидно, це не просто просто розщеплення служб і має під собою більш складні підстави.

## 2.2 Порівняльна характеристика e-commerce платформ

Зазвичай великі e-commerce системи будуються на базі e-commerce платформ. Є декілька найбільш популярних, але вони схожі тим, що надають базові для e-commerce дотатків функції - сутність "продукт" та можливі з нею операції; управління кількістю товарів у наявності; аккаунт користувача; корзина; базові сутності "замовлення", "доставка" та "оплата"; процес купівлі товару; власна ORM (Об'єктно-реляційна проекція); додаток для адміністраторів системи і т.д. Зазвичай платформи надають багато можливостей та через це поріг входження для розробників є досить високим. Тому треба закладати час на проходження розробниками тренінгів, бо знайти спеціаліста під конкретну платформу буває складно. Нижче будуть розглянуті найбільш популярні плафторми для екосистеми Java.

Платформа Oracle Commerce — це технологія Omnichannel компанії Oracle для створення бездоганного досвіду клієнтів і зшивання даних з різних каналів клієнтів. Хоча це традиційно роль CRM-додатків або веб-додатків в Інтернеті, сьогодні постачальники поєднують продажі, маркетинг і сервіс у тих, що називаються комерційними платформами.

Подібно до своїх конкурентів, Oracle запропонувала платформу Commerce для того щоб інтегрувати дані про клієнтів, що генеруються продажами, маркетингом, сервісом, мобільними, соціальними та іншими інформаційними елементами, та об'єднати ці дані в єдиний, 360-градусний огляд клієнта. Постачальники все частіше доповнюють дані про клієнтів, що знаходяться в системах CRM компаній або базах даних облікових записів клієнтів, з даними інших відділів, таких як маркетинг і сервіс, а також дані третіх сторін, які надають більш розумну інформацію про поведінку клієнтів, переваги та готовність до покупки. Об'єднуючи маркетингові, сервісні та цифрові поведінкові дані в мережі, компанії можуть розробити більш вичерпні погляди на клієнтів і перспективи.

Необхідність зв'язування та кращого розуміння цих даних стала першорядною, оскільки подорожі клієнтів через різні канали стають менш лінійними та більш складними. В той час, як раніше, споживачі могли просто подорожувати до магазину або проводити мінімальні дослідження онлайн продуктів, сьогодні вони вільно подорожують між цифровими та фізичними каналами, а їхні шляхи можуть вказувати на їхні уподобання або наступні покупки. Ідея полягає в тому, щоб використовувати технологію для створення бездоганного досвіду, який виходить за межі каналів, або для створення рідкого, omnichannel досвіду.

Інші виробники розробляють подібні платформи для створення бездоганного досвіду клієнтів та інтеграції інформації про клієнтів. У Salesforce є свій Commerce Cloud, а SAP має платформу Beyond CRM/Hybris.

Oracle ATG (платформа ATG) є основою що налаштовується, конфігурується для створення та підтримки веб-сайтів, зокрема сайтів, що використовуються для електронної комерції. Платформа ATG включає кілька шарів.

Базовий платформний шар — платформа "Dynamo Application Framework" (DAF). Вона забезпечує середовище розробки компонентів, що складається з JavaBeans і JavaServer Pages (JSP). Розробники збирають додатки з компонентних компонентів (на основі стандартних класів ATG або користувальницьких класів Java), поєднуючи їх за допомогою конфігураційних файлів у Nucleus, відкритому об'єктному середовищі ATG.

Модуль персоналізації "Personalization module" (DPS) надає код для підтримки вмісту веб-сайту, який динамічно змінюється для кожного користувача. За допомогою цього модуля ви можете створити та підтримуєте профілі користувачів та бізнес-правила, які визначають, який вміст показувати кому. Цей шар також підтримує цільові повідомлення електронної пошти.

Модуль сценаріїв "Scenarios module" (DSS) розширює можливості таргетованого контенту модуля персоналізації. Сценарії - це чутливі до часу

кампанії, орієнтовані на події, призначені для управління взаємодією між відвідувачами сайту та вмістом протягом тривалого періоду.

Платформа включає модулі B2B і B2C для кращої підтримки різних моделей бізнесу та продажу. Платформа ATG працює на одному з трьох серверів додатків: Oracle WebLogic, JBoss або IBM WebSphere.

Oracle Commerce є платформою з закритим початковим кодом, яка поширюється на комерційній основі. Вона складається з декількох модулів, які відповідають за різні глобальні функції — процесс покупки, управління профайлом користувача, управління товарами і т.д. Через те що платформа була представлена досить давно, вона мимоволі штовхає вас на створення монолітного e-commerce додатку. Тому чим складніше стає додаток, тим складніше проводити часті релізи, а це є дуже пощиренною та одною із головних вимог бізнес замовників. Хоча платформа досить підтримується, через її складність підтримка нових версій Java надається через деякий час, тому з часом команди починають працювати з неактуальною версією. Використання нових технологій стає дуже складним, тому шо платформа рідко оновлюється. Також її використання є досить дорогим.

SAP Hybris - одне з провідних рішень в світі для створення масштабних інтернет-магазинів, eCommerce-проектів. Має високу стійкість до відмов і надає багаті функціональні можливості, які задовільнять різні вимоги бізнесу до майбутньої майданчику для онлайн-продажів. SAP hybris використовують понад 400 найбільших компаній, лідерів ринку електронної комерції (eCommerce) по всьому світу. Це багатопрофільні інтернет-магазини, B2B-системи торгових дистриб'юторів, інтернет-каталоги з продажу аудіо- та відео-контенту.

Переваги SAP Hybris:

— одна з провідних платформ для електронної комерції в сегменті великого бізнесу;

— готова інтеграція з усіма рішеннями SAP;

— багата функціональність і надійність;

— додаткові інструменти для аналізу та управління онлайн-продажами, електронною комерцією.

SAP Hybris створює нові можливості для розвитку електронної комерції (онлайн-продажів). Дозволяє реалізувати стратегії омніканального збуту (omni-channel), ефективне управління замовленнями та підтримку продажів через колл- центр з можливістю редагування замовлення в інтернет-магазині. Також платформа спрощує створення мобільного додатку, надає готову інтеграцію інтернет-магазину з популярними платіжними системами, можливість створення каталогів товарів зі спеціальними цінами, умовами оплати та доставки для кожного регіону, країни.

Платформа надає готове рішення для обробки замовлень, часткового відвантаження, автоматичний розрахунок доставки та інтеграцію з зовнішніми сервісами доставки.

Також корисним стане SEO - модуль. Просування інтернет-магазину в пошукових системах буде ще ефективніше.

Hybris використовує Spring Framework, JSP, IBM у якості сервера, може інтегруватися з популярними базами даних. Поріг входження може бути дещо легшим завдяки тому що у цій платформі використовується Spring, але все одно навчання потребуватиме багато часу. Використання цієї платформи також є досить дорогим, і взагалі недоліки використання є схожими на попередній аналог.

Платформи e-commerce допомагають зекономити час, який можна використати на те, щоб додати нову функціональність для проекту замість того щоб будувати базис з нуля. Вони прекрасно підходять для не надто великих систем, які тільки починають отримувати достатній дохід. Але для величезних систем вони у якійсь момент починають накладати обмеження та ускладнюють розвиток. Кастомізація рішення все одно буде необхідно для будь якого проекту і у деяких випадках додавання нової функціональності може бути значно легшим без обмежень, які накладає платформа. Для полегшення ситуації великі проекти можуть почати міграцію з e-commerce платформ до мікросервісної

архітектури, знаючи об'єм робіт та маючи можливість розбити моноліт на мікросервіси більш вдало для отримання усіх переваг цього підходу.

### 2.3 Шаблони розгортання мікросервісних додатків

Розгортання монолітного додатка означає виконання декількох ідентичних копій одного, зазвичай великого додатка. Зазвичай ви надаєте N серверів (фізичних або віртуальних) і запускаєте M екземплярів програми на кожному. Розгортання монолітного додатка не завжди є цілком простим, але набагато простіше, ніж розгортання мікросервісного додатка.

Додаток мікросервісів складається з десятків або навіть сотень сервісів. Послуги написані різними мовами та за допомогою різних фреймворків. Кожен з них являє собою міні-додаток із власними специфічними вимогами до розгортання, ресурсів, масштабування та моніторингу. Наприклад, потрібно запустити певну кількість екземплярів кожної служби на основі потреби в цій службі. Також кожен екземпляр служби повинен бути забезпечений відповідними ресурсами CPU, пам'яті та I/O. Ще більш складним є те, що, незважаючи на цю складність, розгортання послуг має бути швидким, надійним і економічно ефективним. Існує кілька різних схем розгортання мікросервісу:

— Multiple Service Instances per Host Pattern

Один із способів розгортання ваших мікросервісів — шаблон "декілька екземплярів сервіса на хості". Використовуючи цей шаблон, ви надаєте один або більше фізичних або віртуальних хостів і запускаєте декілька екземплярів служби на кожному. Багато в чому це традиційний підхід до розгортання додатків. Кожен екземпляр служби виконується у відомому порту на одному або декількох вузлах.

Шаблон декількох екземплярів сервіса на хості має як переваги, так і недоліки. Однією з головних переваг є відносно ефективне використання ресурсів. Кілька екземплярів служб спільно використовують сервер і його операційну систему. Це ще більш ефективно, якщо процес або группа процесів

запускає кілька екземплярів служб, наприклад, кілька веб-додатків, що використовують один і той же сервер Apache Tomcat і JVM.

Ще однією перевагою цієї моделі є те, що розгортання екземпляра служби є відносно швидким. Ви просто копіюєте сервіс на хост і запускаєте її. Якщо служба написана на Java, скопіюйте файл JAR або WAR. У будь-якому випадку, кількість байтів, скопійованих по мережі, є відносно невеликим. Крім того, через відсутність накладних витрат, запуск сервісу зазвичай дуже швидкий.

Незважаючи на свою привабливість, шаблон "декілька екземплярів сервіса на хості" має деякі суттєві недоліки. Одним з основних недоліків є те, що ізоляція службових екземплярів практично відсутня, якщо кожен екземпляр служби не є окремим процесом. Незважаючи на те, що ви можете точно контролювати використання ресурсів кожного екземпляра служби, ви не можете обмежити ресурси, які використовує кожен екземпляр. Можливо, що неправильно функціонуючий екземпляр служби споживає всю пам'ять або інші ресурси хоста.

Ізоляції взагалі немає, якщо декілька екземплярів служб виконуються в одному процесі. Усі екземпляри можуть, наприклад, спільно використовувати одну і ту ж JVM. Неповноцінний екземпляр служби може легко зламати інші служби, що працюють у цьому ж процесі. Крім того, ви не маєте можливості контролювати ресурси, що використовуються кожним екземпляром служби.

Іншою істотною проблемою цього підходу є те, що devops група, яка розгортає службу, повинна знати конкретні деталі того, як це зробити. Послуги можуть бути написані різними мовами та рамками, тому є багато деталей, якими команда розробників має поділитися з devops інженерами. Ця складність збільшує ризик помилок під час розгортання.

Іншим способом розгортання ваших мікросервісів є шаблон "Екземпляр сервісу на хост". Коли ви використовуєте цей шаблон, ви запускаєте кожен екземпляр служби окремо на своєму хості. Існує дві різні спеціалізації цієї схеми: Служба екземпляра на віртуальну машину та екземпляр служби на контейнер.

— Service Instance per Virtual Machine Pattern

Якщо ви використовуєте екземпляр сервісу на віртуальну машину, ви упаковуєте кожну службу як зображення віртуальної машини (VM), наприклад, AMI AM2 EC Amazon. Кожен екземпляр сервісу — це віртуальна машина (наприклад, екземпляр EC2), яка запускається з використанням зображення VM.

Існують різноманітні інструменти, які можна використовувати для створення власних віртуальних машин. Ви можете налаштувати сервер безперервної інтеграції (CI) (наприклад, Jenkins), щоб викликати Aminator, щоб упакувати ваші послуги як EC2 AMI. Packer.io — це ще один варіант для автоматичного створення зображень VM. На відміну від Aminator, він підтримує різні технології віртуалізації, включаючи EC2, DigitalOcean, VirtualBox і VMware.

Шаблон "екземпляр сервісу на віртуальну машину" має ряд переваг. Основною перевагою віртуальних машин є те, що кожен екземпляр служби працює в повній ізоляції. Він має фіксований обсяг процесора і пам'яті і не може вкрасти ресурси з інших служб.

Ще однією перевагою розгортання ваших мікросервісів як віртуальних машин є те, що ви можете використовувати зрілі хмарні інфраструктури. Такі хмари, як AWS та Google Cloud, надають корисні функції, такі як балансування навантаження та автоматичне масштабування.

Інша велика перевага розгортання служби як віртуальної машини полягає в тому, що вона інкапсулює технологію реалізації вашої служби. Після того, як послуга була упакована як VM, вона стає чорним ящиком. API керування VM стає API для розгортання служби. Розгортання стає набагато простішим і надійнішим.

Шаблон "екземпляр сервісу на віртуальну машину" має певні недоліки. Одним з недоліків є менш ефективне використання ресурсів. Кожен екземпляр служби має накладні витрати на всю VM, включаючи операційну систему. Більш того, у типовому суспільному IaaS віртуальні машини мають фіксований розмір і можливо, що віртуальна машина буде недостатньо використана.

Більш того, загальнодоступний IaaS зазвичай стягує плату за віртуальні машини, незалежно від того, чи вони зайняті або бездіяльні. IaaS, такий як AWS, забезпечує автоматичне масштабування, але важко швидко реагувати на зміни попиту. Отже, вам часто доводиться надавати VM, що збільшує вартість розгортання.

Іншим недоліком цього підходу є те, що розгортання нової версії служби зазвичай відбувається повільно. Зображення віртуальної машини, як правило, повільні для побудови через їх розмір. Крім того, віртуальні машини, як правило, повільні для інстанції, знову через їх розмір. Крім того, для запуску операційної системи зазвичай потрібен деякий час. Зауважте, однак, що це не є універсальним, оскільки існують легкі віртуальні машини.

— Service Instance per Container Pattern

При використанні шаблону "екземпляр сервісу на контейнер" кожен екземпляр служби запускається у власному контейнері. Контейнери - це механізм віртуалізації на рівні операційної системи. Контейнер складається з одного або більше процесів, що виконуються в пісочниці. З точки зору процесів, вони мають свій власний простір імен портів і кореневу файлову систему. Ви можете обмежити пам'ять контейнера та ресурси ЦП. Деякі реалізації контейнерів також мають обмеження швидкості введення/виводу. Приклади контейнерних технологій включають зони Docker і Solaris.

Щоб скористатися цим шаблоном, пакуйте службу як зображення контейнера. Зображення контейнера — це зображення файлової системи, що складається з програм і бібліотек, необхідних для запуску служби. Деякі зображення контейнерів складаються з повної кореневої файлової системи Linux. Інші більш легкі. Для розгортання сервісу Java, наприклад, ви створюєте зображення контейнера, що містить середовище виконання Java, можливо сервер Apache Tomcat, а також скомпільований Java-додаток.

Після того, як ви упакуєте свою службу як зображення контейнера, ви запускаєте один або більше контейнерів. Зазвичай ви запускаєте декілька контейнерів на кожному фізичному або віртуальному хості. Ви можете

використовувати менеджер кластера, наприклад Kubernetes або Marathon, для керування вашими контейнерами. Менеджер кластерів розглядає хости як пул ресурсів. Він вирішує, де розміщувати кожен контейнер на основі ресурсів, необхідних для контейнера, і ресурсів, наявних на кожному хості.

Модель "екземпляр сервісу на контейнер" має як переваги, так і недоліки. Переваги контейнерів подібні до переваг віртуальних машин. Вони ізолюють екземпляри служб один від одного. Можна легко контролювати ресурси, що споживаються кожним контейнером. Також, як і віртуальні машини, контейнери інкапсулюють технологію, яка використовується для реалізації ваших послуг. API керування контейнерами також служить API для керування вашими сервісами.

Однак, на відміну від віртуальних машин, контейнери — це легка технологія. Зображення контейнерів зазвичай дуже швидко збираються. Контейнери також стартують дуже швидко, оскільки немає довгого механізму завантаження ОС. Коли стартує контейнер, сервіс починає виконуватися.

Є деякі недоліки використання контейнерів. Хоча контейнерна інфраструктура швидко розвивається, вона не настільки зріла, як інфраструктура для віртуальних машин. Крім того, контейнери не настільки безпечні, як віртуальні машини, оскільки контейнери розділяють ядро хост-ОС один з одним.

Крім того, контейнери часто розгортаються в інфраструктурі з ціноутворенням за віртуальною машиною. Отже, як було описано раніше, ви, ймовірно, понесете додаткові витрати на переоформлення віртуальних машин для обробки збоїв навантаження.

### 2.4 Огляд системних інтеграторів

Розгортання мікросервісних додатків є більш складним, ніж розгортання монолітних додатків, хоча завдяки цьому є більш гнучким. Для того щоб полегшити цей процес можно використати існуючі на ринку IaaS, PaaS рішення.

Окрім полегшення розгортання вони також надають такі необхідні речі моніторінг, зберігання журналів, сповіщення про помилки у різних каналах зв'язку та інші необхідні застосування для зручної підтримки мікросервісних додатків. Далі будуть розглянуті найбільші з таких інтеграторів - AWS та Google Cloud Platform.

### 2.4.1 Google Cloud Platform (GCP)

GCP складається з набору фізичних активів, таких як комп'ютери та жорсткі диски, і віртуальні ресурси, такі як віртуальні машини (VM), які містяться в центрах даних Google по всьому світу. Кожне розташування центру обробки даних знаходиться в глобальному регіоні. Регіони включають США, Західну Європу та Східну Азію. Кожен регіон являє собою набір зон, які є ізольованими один від одного в межах регіону. Кожна зона ідентифікується назвою, що поєднує в собі ідентифікатор букви з назвою регіону. Такий розподіл ресурсів надає декілька переваг, включаючи зменшення затримки за допомогою розміщення ресурсів ближче до клієнтів. Цей розподіл також вводить деякі правила про те, як можна використовувати ресурси разом.

У хмарних обчисленнях те, що зазвичай називається програмне та апаратне забезпечення, стає сервісами. Ці сервіси надають доступ до базових ресурсів. Список доступних послуг GCP досить великий, і він постійно зростає. Коли ви розробляєте свій веб-сайт або додаток на GCP, ви поєднуєте ці сервіси у комбінаціях, які надають потрібну вам інфраструктуру, а потім додаєте код, щоб увімкнути сценарії, які потрібно створити.

GCP надає вам такі варіанти для обчислень і хостингу: працювати у безсерверному середовищі; використовувати платформу керованих програм; використовувати контейнерні технології, щоб отримати велику гнучкість; створити власну хмарну інфраструктуру, щоб мати найбільший контроль і гнучкість. Ви можете уявити собі спектр, де, з одного боку, ви маєте більшу частину відповідальності за управління ресурсами, а з іншого боку, Google має більшу частину цих обов'язків (рис 2.1).

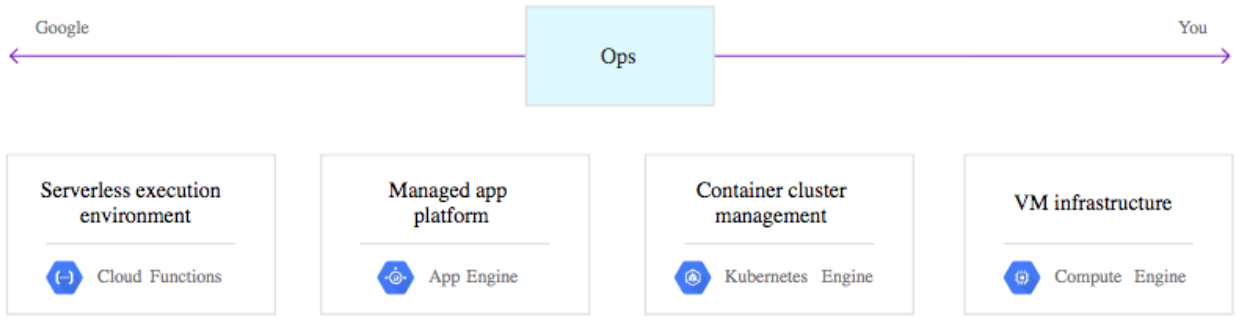


Рисунок 2.1 - Google Cloud Platform engines

Google Cloud Functions, функції GCP як сервіс (FaaS) надають середовище виконання сервера без створення серверів і підключення хмарних сервісів. За допомогою Cloud Functions ви можете писати прості, одноцільові функції, які приєднуються до подій, що піднімаються за допомогою хмарної інфраструктури та служб. Cloud Function запускається, коли піднята спостережувана подія. Ваш код виконується в повністю керованому середовищі; Вам не потрібно надавати будь-яку інфраструктуру або турбуватися про керування будь-якими серверами.

Ви пишете Cloud Function, використовуючи JavaScript, і виконуйте їх у середовищі Node.js v6.11.5 на GCP. Ви можете запустити Cloud Function у будь-якому стандартному середовищі виконання Node.js, що полегшує портативність і локальне тестування. Хмарні функції є гарним вибором для випадків використання, які включають:

— обробка даних та операції ETL для сценаріїв, таких як транскодування відео та потокові дані IoT;

— Webhooks реагують на тригери HTTP;

— легкі API, які містять вільно пов'язану логіку в додатках;

— мобільні функції бекенда.

Google App Engine - це платформа GCP як сервіс (PaaS). За допомогою App Engine Google виконує більшу частину керування ресурсами. Наприклад, якщо ваша програма потребує більшої кількості обчислювальних ресурсів, оскільки трафік на ваш сайт збільшується, Google автоматично масштабує систему, щоб

забезпечити ці ресурси. Якщо системному програмному забезпеченню потрібне оновлення для системи безпеки, яке також використовується для вас. Коли ви створюєте свою програму на App Engine, ви можете:

— створити свою програму на базі стандартного середовища запуску App Engine на мовах, які підтримує стандартне середовище, включаючи: Python 2.7, Java 8, Java 7, PHP 5.5 і Go 1.8, 1.6;

— створити свою програму на базі гнучких середовищ App Engine на мовах, які підтримує гнучка програма App Engine, зокрема: Python 2.7 / 3.6, Java 8, Go 1.8, Node.js, PHP 5.6, 7, .NET і Ruby. Або використати спеціальний режими виконання, щоб використовувати альтернативну реалізацію підтримуваної мови або будь-якої іншої мови;

— дозволити Google керувати хостингом програм, масштабуванням, моніторингом та інфраструктурою;

— використати SDK App Engine для розробки та тестування на локальному комп'ютері в середовищі, що імітує App Engine на GCP;

— легко використовувати технології зберігання, призначені для підтримки в стандартних і гнучких середовищах. Google Cloud SQL — це база даних SQL, яка підтримує MySQL або PostgreSQL. Google Cloud Datastore є сховищем даних NoSQL. Google Cloud Storage надає простір для великих файлів. У стандартному середовищі можна також вибрати з безлічі баз даних сторонніх виробників для використання з такими програмами, як Redis, MongoDB, Cassandra і Hadoop. У гнучкому середовищі можна легко використовувати будь-яку базу даних сторонніх виробників, яку підтримує ваша мова, якщо ця база даних доступна з екземпляра Google App Engine. У будь-якому середовищі ці бази даних третіх сторін можуть розміщуватися на комп'ютері Compute Engine, розміщеному на іншому постачальнику послуг у хмарі, розміщеному на місці або керованим постачальником сторонніх розробників;

— використати Cloud Endpoints у стандартному середовищі для створення API та бібліотек клієнтів, які можна використовувати для спрощення доступу до

даних з інших програм. Кінцеві точки спрощують створення веб-інтерфейсу для веб-клієнтів і мобільних клієнтів, таких як Android або iOS;

— використати вбудовані керовані служби для таких дій, як керування електронною поштою та користувачам;

— використати Cloud Security Scanner для виявлення вразливостей безпеки як доповнення до існуючих безпечних процесів проектування та розробки.

Програму може бути розгорнута за допомогою програми GUI для запуску App Engine на MacOS або Microsoft Windows або за допомогою командного рядка.

За допомогою обчислень на базі контейнерів ви можете зосередитися на своєму коді програми, а не на розгортанні та інтеграції в середовище хостингу. Google Kubernetes Engine, контейнери GCP як сервіс (CaaS), побудований на відкритій системі Kubernetes, що дає вам гнучкість локальних або гібридних хмар, на додаток до інфраструктури публічної хмари GCP. Коли ви будуєте з двигуном Kubernetes, ви можете:

— створювати та керувати групами екземплярів Compute Engine, що виконують Kubernetes, звані кластерами. Двигун Kubernetes використовує екземпляри Compute Engine як вузли кластера. Кожен вузол запускає середовище виконання Docker, агент вузла Kubernetes, який контролює стан вузла, і простий мережний проксі;

— визначити вимоги до ваших контейнерів Docker, створивши простий конфігураційний файл JSON;

— використати Google Container Registry для безпечного приватного зберігання зображень Docker. Ви можете покласти зображення до реєстру, а потім витягувати зображення до будь-якого екземпляра Compute Engine або власного обладнання за допомогою кінцевої точки HTTP;

— створити одно- та багатоконтейнерні поди (pod — це найменший, найосновніший об'єкти у Kubernetes. Pod представляє один екземпляр запущеного процесу у вашому кластері.). Кожна пачка являє собою логічний

хост, який може містити один або більше контейнерів. Контейнери в блоці працюють разом, обмінюючись ресурсами, такими як мережеві ресурси. У сукупності набір подів може містити цілий додаток, мікросервіс або один шар у багаторівневій програмі;

— створити та керувати "replication controllers", які керують створенням і видаленням реплікантів на основі шаблону. Контролери реплікації допомагають забезпечити, щоб у вашій програмі були ресурси, необхідні для надійного та відповідного масштабування;

— створити та керувати сервисами. Сервіси створюють шар абстракції, який відокремлює клієнтів інтерфейсу від подів, які надають функції бекенда. Таким чином, клієнти можуть працювати без занепокоєння щодо того, які поди створюються та видаляються в будь-який момент;

— створити балансування зовнішньої мережі.

Некерованим обчислювальним сервісом GCP є Google Compute Engine. Ви можете вважати, що Compute Engine надає інфраструктуру як послугу (IaaS), оскільки система забезпечує надійну обчислювальну інфраструктуру, але ви повинні вибрати і налаштувати компоненти платформи, які ви хочете використовувати. Завдяки Compute Engine ваша відповідальність - налаштувати, адмініструвати та контролювати системи. Google забезпечить наявність, надійність і готовність ресурсів для використання, але ви повинні приготувати їх і керувати ними. Перевага тут полягає в тому, що ви маєте повний контроль над системами і необмежену гнучкість. Коли ви будуєте на Compute Engine, ви можете:

— використовувати віртуальні машини (VMs), які називаються екземплярами, щоб побудувати свою програму, подібно до того, як це було б, якщо б ви мали власну апаратну інфраструктуру. Ви можете вибрати з різних типів екземплярів, щоб налаштувати конфігурацію відповідно до ваших потреб і бюджету;

— обрати, які глобальні регіони та зони розгортатимуть свої ресурси, надаючи вам контроль над тим, де зберігаються й використовуються ваші дані;

— обрати, які операційні системи, стеки розробки, мови, фреймворки, служби та інші технології програмного забезпечення ви віддаєте перевагу;

— створити екземпляри із відкритих або приватних зображень;

— використати технології зберігання даних GCP або будь-які інші технології, які ви віддаєте перевагу;

— використати Google Cloud Launcher для швидкого розгортання попередньо настроєних програмних пакетів;

— створити групи екземплярів, щоб легше керувати кількома екземплярами разом;

— використати автоматичне масштабування з групою екземплярів для автоматичного додавання та видалення можливостей;

— при необхідності приєднати та від'єднати диски;

— використати SSH для підключення безпосередньо до своїх екземплярів.

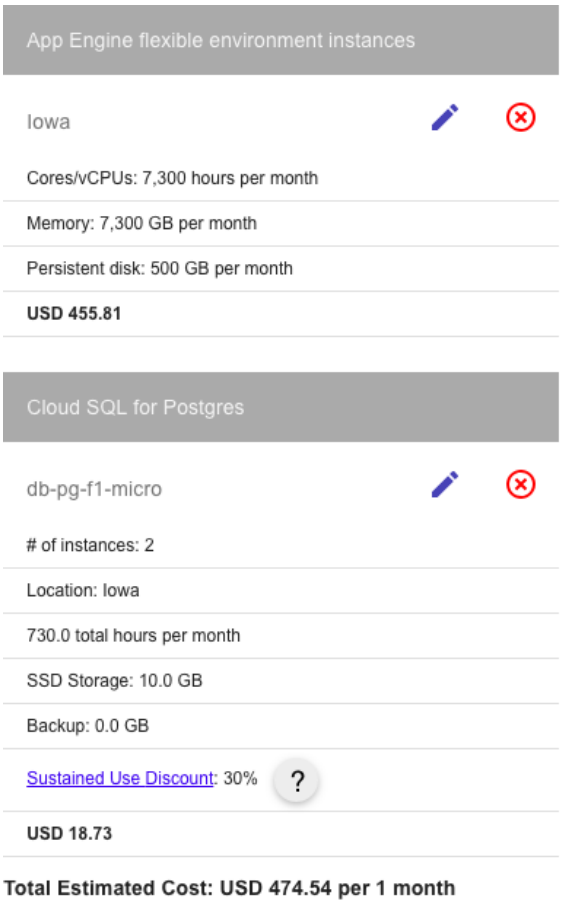


Рисунок 2.2 - Вартість послуг GCP на місяць

Також GCP дозволяє використовувати різні сервіси, які можуть бути корисними для вашого додатку — Google Maps API, Big Query для зберігання бізнес-аналітики, що є дуже необхідним для e-commerce додатків, Google Pub/Sub для асинхронного спілкування між вашими мікросервісами та багато інших.

Для розрахунку потенціальних витрат у GCP є зручний ціновий калькулятор. Вартість залежить від того, якими сервісами ви користуєтесь, скільки у вас екземплярів бази даних, який обсяг даних ви зберігаєте, скільки у вас екземплярів ваших додатків та за допомогою якого з engine перерахованих вище ви будуєте свій сервіс.

### 2.4.2 Amazon Web Services (AWS)

Amazon Web Services (AWS) є одним із лідерів на ринку IaaS (інфраструктура як сервіс) і PaaS (платформа як сервіс) для хмарних екосистем. пов'язані з наданням інфраструктури (обчислення, зберігання та мережа) та управління.

За допомогою AWS ви можете вибрати конкретні рішення, які вам потрібні, і платити тільки за те, що ви використовуєте, що призводить до зниження капітальних витрат і швидшого часу для оцінки, не жертвуючи при цьому продуктивністю програми або користувачем.

Amazon пропонує цілий універсал продуктів і послуг для побудови або розширення вашого середовища в хмарі. А саме створення віртуальних машин (екземпляри EC2), зберігання даних (S3), машинне навчання, розробка ігор і багато іншого (більше 8000 сервісів). І на основі подібності між цими послугами, вони класифікуються в різні розділи консолі AWS. Опишемо Compute сервіси:

— EC2 (Elastic Compute Cloud): їх можна розглядати як віртуальні машини, які можна побудувати всередині облачної платформи AWS. А AWS не обмежує вас віртуальними машин, ви також можете створювати фізичні виділені машини за допомогою сервісу EC2;

— EC2 Container Service (ECS): Ця послуга дозволяє легко запускати, масштабувати та захищати програми Docker на платформі AWS;

— Elastic Beanstalk: З сервісом Elastic Beanstalk, розробники, які не розуміють AWS концепції можуть завантажувати свої веб-додатків, а потім Elastic Beanstalk сервіс автоматично обробить розгортання, балансування навантаження, буде автоматично масштабуватися і моніторити здоров'я програми;

— Lambda: За допомогою сервісу AWS Lambda ви можете завантажити свій код і запустити його на платформі AWS. При цьому вам не потрібно керувати серверами, віртуальними машинами або будь-якими базовими речами. Таким чином, це легка і дуже популярна послуга серед громади. В даний час ця послуга дозволяє коди, написані на мовах Node.js, Java, C #, Go і Python;

— Lightsail: Lightsail — віртуальна приватна послуга Amazon (служба VPS). Це означає, що він призначений для звичайних користувачів, які не розуміють нічого про концепції AWS. Таким чином, у пакеті пропонуються віртуальна машина, пам'ять, управління DNS і статичний IP. Але в інших службах, таких як EC2, буде надаємо екземпляр EC2 і решта функцій буде встановлена і налаштована;

— Batch: за допомогою Batch сервісу програміст може виконувати сотні або тисячі пакетних обчислень у хмарі AWS.

Розглянемо наявні сервіси бази даних:

— RDS (служба реляційних баз даних): за допомогою цієї служби ви можете отримати реляційні бази даних, такі як MySQL, MSSQL, PostgreSQL, Oracle, Arora (версія MySQL від Amazon) і т.д;

— DynamoDB: це нереляційна служба баз даних (NoSQL). Це повністю керована хмарна база даних, яка підтримує моделі зберігання документів і ключів;

— Elasticache: Це служба кешування доступна для баз даних. Це допомагає покращити продуктивність веб-додатків, отримуючи інформацію з кешу, замість того, щоб повністю покладатися на більш повільні бази даних на дисках. Таким

чином, ви можете кешувати важливі дані в кеш і використовувати цю послугу для підвищення продуктивності.

— Red Shift: За допомогою служби Red Shift ви можете зберігати ваші дані та використовувати інструменти бізнес-аналітики для ефективного аналізу вартості даних. Таким чином, першим кроком до створення сховища даних є запуск набору вузлів, які називаються кластером Amazon Redshift.

Таким чином, як AWS так і GCP інтегратори надають багато різноманітних сервісів, які задовільняють вимогам більшості додатків, у тому числі і e- commerce. У кожного з них є зручна веб-консоль завдяки якій ви можете моніторити свої додатки, підключати нові сервіси, слідкувати за витраченими коштами та інше. Це є дуже зручним для мікросервісних додатків, тому що всі їх дані та конфігурація будут зручно розташовані в одному місці.

## 2.5 Математична модель інтегратора e-commerce додатка

Субмодель контейнера (Container sub-model або CSM) є 3-мірним безперервним ланцюгом Маркова (Continuous Time Markov Chain CTMC) зі станами, що позначені як (i, j, k), де і позначає кількість запитів в глобальній черзі мікрообслуговування, j позначає число працюючих контейнерів на платформі, k показує кількість активних віртуальних машин у групі хоста користувача. Іншими словами, ми припускаємо, що всі періоди між подіями розподілені експоненціально. Кожна віртуальна машина (ВМ) може розмістити до M контейнерів, які встановлюються користувачем. Так як число потенційних користувачів є високим, і один користувач зазвичай виконує запити в один і той самий час з низькою ймовірністю, прибуття запитів можна адекватно змоделювати як процес Пуассона зі швидкістю $\lambda$. Нехай $\varphi$ — швидкість, з якою контейнер може бути розгорнутий на ВМ і $1/\mu$ — середня тривалість життя контейнерів (тобто обидва експоненціально розподілені). Отже, загальна Швидкість обслуговування для кожної ВМ — це добуток числа запуску

контейнерів $\mu$. Припустимо, $\beta+$ і $\beta-$ є швидкість, за який мікросервіса платформа (MSP) може захопити і звільнити VM відповідно. CSM запитує нову віртуальну машину від хмари бекенда, коли явно упорядковано користувачем мікросервісної платформи або коли використання хост групи є рівним або більшим ніж задане значення. Для стану *(i, j, k),* використання (тобто u) визначається наступним чином,

$$u = \frac{i + j}{k * M}, \tag{2.1}$$

де M — максимальна кількість контейнерів, які можуть виконуватися на одній віртуальній машині. Значення u вказує співвідношення активних контейнерів до всіх доступних контейнерів у кожному стані. З іншого боку, якщо використання стає меншим ніж попередньо визначене значення, CSM звільнить одну BM для оптимізвції витрат. Віртуальну машину можна звільнити, якщо на ній немає запущених контейнерів, так що віртуальна машина повинна бути повністю виведена з експлуатації заздалегідь. Також CSM захоплює мінімальну кількість віртуальних машин у групі хоста незалежно від використання, щоб зберегти доступність сервіса (s). Користувач може також встановити інше значення для свого додатка (S) вказуючи, що мікросервіса платформа не може запитувати більше ніж S віртуальних машин від інфраструктури макросервісу від імені користувача. Таким чином, програма розширюється не більше ніж до S віртуальних машину у разі високого трафіку та звужується до s віртуальних машин під час низького трафіку. Ми встановлюємо розмір глобальної черги (Lq) до загальної кількості контейнерів, які він може розміщуюти на повну потужність (тобто Lq = S x M). Зверніть увагу на те, що запит буде заблоковано, якщо користувач досяг своєї ємності, незалежно від стану глобальної черги.

Стан (0, 0, s) вказує, що у черзі відсутні запити, запущені контейнери відсутни і хост група складається мінімальної кількісті віртуальних машин, що підтримуються користувачем хост групи.

Розглянемо довільне стан, такий як (i, j, k), в яких може відбутися п'ять переходів.

Перехід 1. Після прибуття нового запиту система з швидкістю X рухається до стану (i + 1, j, k), якщо користувач все ще має ємність (тобто i + j <S $^{x}$ M), інакше запит буде заблоковано, і система залишиться в поточному стані;

Перехід 2. CSM створює контейнер зі швидкістю ф для запиту у голові глобальної черги і переходить до (i - 1, j + 1, k);

Перехід 3. Час життя контейнерів експоненціально розподілено і закінчується зі швидкістю j $\mu$ і система переходить до (i, j - 1, k);

Перехід 4. Якщо коефіцієнт використання вищий ніж поріг, мікросервісна платформа запитує нову віртуальну машину, і система переходить у стан (i, j, k + 1) зі швидкістю $\beta +$.

Перехід 5. Або, використання падає нижче певного значення і мікросервісна платформа виводить з експлуатації віртуальну машину, і система звільняється від віртуальної машини, що не використовується, так, що переходить до (i, j, k - 1) зі швидкістю $\beta -$.

Зауважимо, що CSM розрахована лише для одного користувача (або для одного додатку в іншому сенсі); в цій роботі ми припустимо, що користувачі однорідні, так що нам потрібно тільки вирішити одну CSM незалежно від кількості користувачів мікросервісної платформи. Припустимо, що $\pi_{(i,j,k)}$ стаціонарна ймовірність для CSM бути в стан (i, j, k). Таким чином, ймовірність блокування в CSM може бути розрахована наступним чином:

$$bp_{(i,j,k)} = \sum \pi_{(i,j,k)},\ i + j = L_q. \tag{2.2}$$

Нас також цікавлять дві ймовірності, за якими мікросервісна платформа запитує ($P_{req}$) або звільняє ($P_{rel}$) віртуальну машину.

$$\begin{aligned} p_{req} &= \sum \pi_{(i,j,k)}, u \geq high - util \ \& \ k < S, \\ p_{rel} &= \sum \pi_{(i,j,k)}, u \geq loq - util \ \& \ k < S. \end{aligned} \tag{2.3}$$

Використовуючи ці ймовірності, швидкість, за якою мікросервісна платформа запитує ($\lambda_c$) або звільнює ($\eta_c$) віртуальну машину може бути розрахована таким чином:

$$\begin{aligned} \lambda_c &= \lambda \times p_{req}, \\ \eta_c &= p_{rel} \times \mu. \end{aligned} \tag{2.4}$$

Щоб розрахувати середній час очікування в черзі, ми спочатку обчислимо кількість запитів у черзі:

$$\bar{q} = \sum_{i=0}^{L_q} i \cdot \pi_{(i,j,k)}. \tag{2.5}$$

Застосувавши закон Літтла, розрахуємо середній час очікування в глобальній черзі:

$$\overline{wt_q} = \frac{\bar{q}}{\lambda(1 - bp_q)}. \tag{2.6}$$

Час вводу до експлуатації для контейнерів ( $1/\varphi$ ) буде додано до $wt_q$ для отримання загальної затримки в мікросервісній платформі. CSM взаємодіє з фізичною підмоделью забезпечення машин (PMSM) і підмоделью забезпечення віртуальних машин (VMSM). $\lambda_c$ і $\eta_c$ використовуються PMSM і VMSM відповідно.

### 2.6 План експериментальних досліджень

На підставі розглянутих переваг мікросервісної архітектури перед монолітною для подальшого аналізу та створення прототипу був обраний

мікросервісний архітектурний стиль. У якості системного інтегратора був обраний GCP через наявність усіх необхідних для роботи інструментів, пробного безкоштовного періоду користування та можливості використання паттерну розгортання Service Instance per Container Pattern, який відповідає технічним вимогам, що були висунуті до e-commerce систем. Беручі усі ці фактори до уваги, перейдемо до планування експериментальних досліджень:

— розбити типову систему електронної комерції монолітної архітектури на окремі компоненти, використовуючи мікросервісний архітектурний стиль;

— обрати оптимальне сховище даних згідно з функціональністю мікросервіса;

— детально дослідити розробку прототипа одного з мікросервісів, використовуючи мову програмування Java та Google APIs у разі потреби;

— створити дескриптор розгортання для одного з мікросервісів використовуючи Google Cloud Platform та Service Instance per Container Pattern паттерн розгортання;

— створити порівняльну характеристику традиційної архітектури (монолітної) та мікросервісного підходів побудови великих систем для електронної комерції, викорустовуючи метод лінійної аддитивної згортки з ваговими коефіцієнтами;

— дослідити стабільність роботи мікросервіса, час відповіді на запити, вартість підтримки, враховуючи вартість використання необхідних Google сервісів (Cloud SQL, Datastore, Cloud Pub/Sub і т.д.).

## 2.7 Висновки по розділу 2

Були розглянуті існуючі рішення для побудування систем електронної комерції, а саме e-commerce платформи Oracle Commerce та SAP Hybris, які є гарним варіантом для e-commerce систем середньої складності на початку роботи інтернет-магазину, але накладають великі обмеження на розвиток системи.

Таким чином, раніше чи пізніше системам електронної комерції доводиться відмовлятися від використання таких платформ.

У якості альтернативи був розглянутий підхід побудування систем електронної комерції з використанням мікросервісного архітектурного стилю та таких інтеграторів як Google Cloud Platform або AWS, які можуть значно спростити не тільки розгортання додатків, але і побудування інфраструктури системи у цілому. Також вони надають різні корисні сервіси, такі як SQL або NoSQL сховище, сервіс асинхронного обміну повідомленнями Publisher/Subscriber, обмеження доступу до ваших кінцевих точок API за допомогою ключів. У залежності від того який тип машини ви обрали, ви можете взагалі не думати про масштабування та інфраструктуру вашого додатка та довіритися інтегратору, який може автоматично масштабувати ваш додаток, або отримати доступ до всіх конфігурацій вашого додатка та мати усе під контролем.

На підставі розглянутих переваг мікросервісної архітектури перед монолітною для подальшого аналізу та створення прототипу був обраний мікросервісний архітектурний стиль. У якості системного інтегратора був обраний GCP через наявність усіх необхідних для роботи інструментів, пробного безкоштовного періоду користування та можливості використання паттерну розгортання Service Instance per Container Pattern, який відповідає технічним вимогам, що були висунуті до e-commerce систем.

# 3 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ З ПОРІВНЯННЯ АРХІТЕКТУР ПОБУДОВИ СКЛАДНИХ СИСТЕМ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

## 3.1 Опис архітектури системи

У попередніх розділах були розглянуті основні функції, вимоги до систем електронної комерції та рекомендації щодо їх проектування. Розглянемо систему, спроектовану згідно цих рекомендацій (рисунок 3.1).

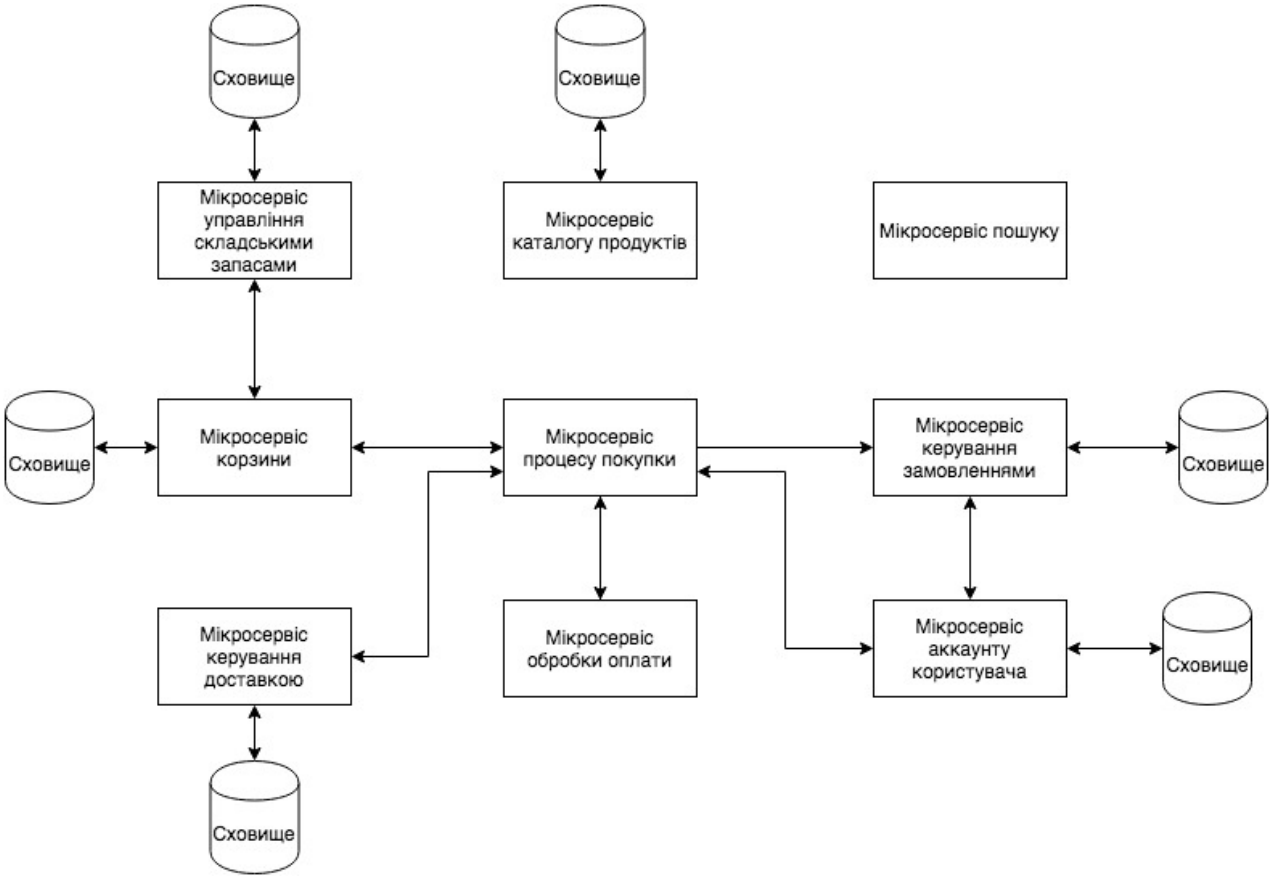


Рисунок 3.1 - Мікросервісна архітектура e-commerce системи

Спроектована система виконує основні функції типової e-commerce системи. Для уявлення про розмір кожного з мікросервісів слід сказати, що

зазвичай над розробкою одного мікросервіса працюють 1-2 команди (у команді приблизно 4-8 інженерів). Слід звернути увагу на те, що у кожного мікросервіса у разі потреби є своє власне сховище. Це робить їх цілком незалежними один від одного, якщо казати про розгортання. Двосторонніми стрілками зображені синхронні зв'язки між мікросервісами, а саме зв'язок за допомогою REST API та запитів у форматі JSON. Односторонніми стрілками зображені асинхронні зв'язки між мікросервісами (Publish/Subscribe Messaging).

Мікросервіс управління складськими запасами відповідає за зберігання інформації про наявність тих чи інших продуктів. Саме тому йому потрібно сховище (наприклад SQL база даних). Він зв'язаний з мікросевісом корзини, який робить запит на резервування на деякий час певної кількості товару при додаванні товару у корзину.

Мікросервіс корзини відповідає за попереднє формування замовлення користувача. Має власне сховище для зберігання стану попереднього замовлення на деякий час поки користувач не почне процес покупки замовлення. Має зв'язок з мікросервісом процесу покупки, якому він передає зібрані дані про замовлення.

Мікросервіс процесу покупки відповідає за оформлення замовлення користувача. Він зв'язаний з декількома іншими мікросервісами - мікросервісом корзини, мікросервісом аккаунту користувача, мікросервісом керування доставкою, мікросервісом здійснення оплати та мікросервісом керування замовленням. У процесі покупки він викликає мікросервіс користувача для того, щоб дізнатися чи є в користувача збережені адреси доставки, спеціальні бонуси або знижки та збережена кредитна карта для того, щоб максимально спростити процес покупки користувачу. Після отримання адреси доставки (збереженої або наданої у процесі) викликається мікросервіс керування доставкою для того, щоб дізнатися можливі опції доставки та їх вартість. Далі після того як визначена остаточна сума замовлення викликається сервіс обробки оплати. Для цього знову таки може використовуватися збережена кредитна картка, або надана у процесі покупки. Також можуть використовуватись інші методи оплати такі як PayPal, подарункова карта та інші. У разі успішною обробки оплати мікросервіс процесу

покупки публікує повідомлення для мікросервісу керування замовленнями для його подальшого опрацювання.

Мікросервіс керування доставкою відповідає за зберігання наявних методів доставки замовлення та їх вартості та визначення підходящого методу доставки згідно з продуктом та адресою доставки. Має власне сховище та пов'язаний з мікросервісом процесу покупки.

Мікросервіс обробки оплати відповідає за опрацювання оплати замовлення різними методами, інтегруючись для цього з певними провайдерами оплати. Також опрацьовує повернення коштів у разі повернення замовлення. Пов'язаний з мікросервісом процесу покупки.

Мікросервіс керування замовленням відповідає за опрацювання замовлення, відправку email підтвердження користувачу, зміну або повернення замовлення. Має власне сховище для зберігання замовлень. Також надає API для отримання усіх замовлень користувача, для того, щоб показати їх у аккаунті користувача. Підписаний на повідомлення мікросервісу процесу покупки.

Мікросервіс аккаунту користувача відповідає за зберігання та керування даними користувача (адреси, кредитні картки, ім'я, телефон, адреса електронної пошти та інше) та процес авторизації у системі. Має власне сховище для зберігання даних користувача. Надає збережену інформацію, необхідну для оформлення замовлення, мікросервісу процесу покупки.

Мікросервіс пошуку відповідає за пошук по продуктам у системі та пропозиції пошуку.

Мікросервіс каталогу продуктів відповідає за зберігання каталогу продуктів та презентацію їх користувачу. Має власне сховище для зберігання продуктів.

Для докладного розгляду побудови мікросервісу та аналізу переваг та недоліків цього архітектурного стилю був обраний мікросервіс каталогу продуктів. У ході роботи був пройдений шлях від проектування мікросервіса до його розгортання з використанням платформи Google Cloud Platform. Його архітектура буде розглянута у наступному підрозділі більш детально.

### 3.2 Опис архітектури мікросервісу каталогу продуктів

Мікросервіс каталогу продуктів складається з різних модулів, кожен з яких виконує одну задачу. Структурна схема мікросервісу каталогу продуктів наведена на рисунку:

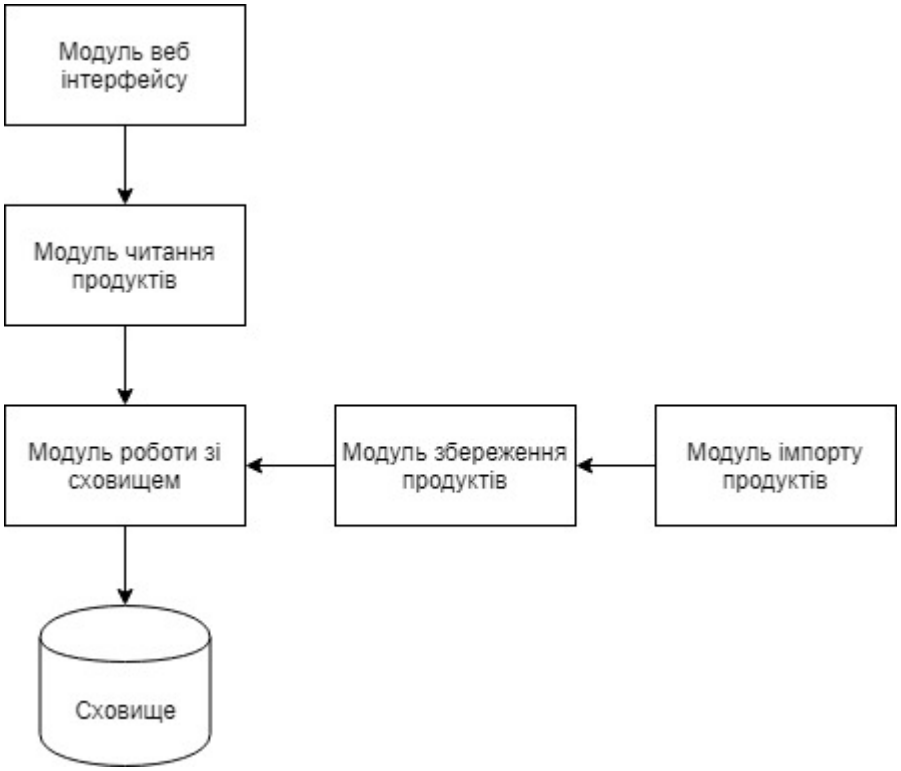


Рисунок 3.2 - Структурна схема мікросервісу каталогу продуктів

Модуль веб-інтерфейсу працює безпосередньо з клієнтом - отримує запити та відправляє відповіді на них.

Модуль читання продуктів використовує модуль роботи зі сховищем для отримання записів продуктів. В свою чергу модуль роботи зі сховищем має в собі усю необхідну інформацію про сховище де зберігається уся інформація про продукти. Таким чином, модуль читання продуктів не залежить від сховища і сховище можна в будь-який час змінити на новий тип і уся система буде у працездатному стані.

Для заповнення сховища продуктами створений модуль імпорту продуктів. Цей модуль використовує дані із різних джерел та використовує модуль збереження продуктів, який в свою чергу використовує вже описаний модуль роботи зі сховищем, для запису інформації про продукти до сховища.

Діаграма взаємодії побудованого мікросервісу показана на рисунку 3.3.

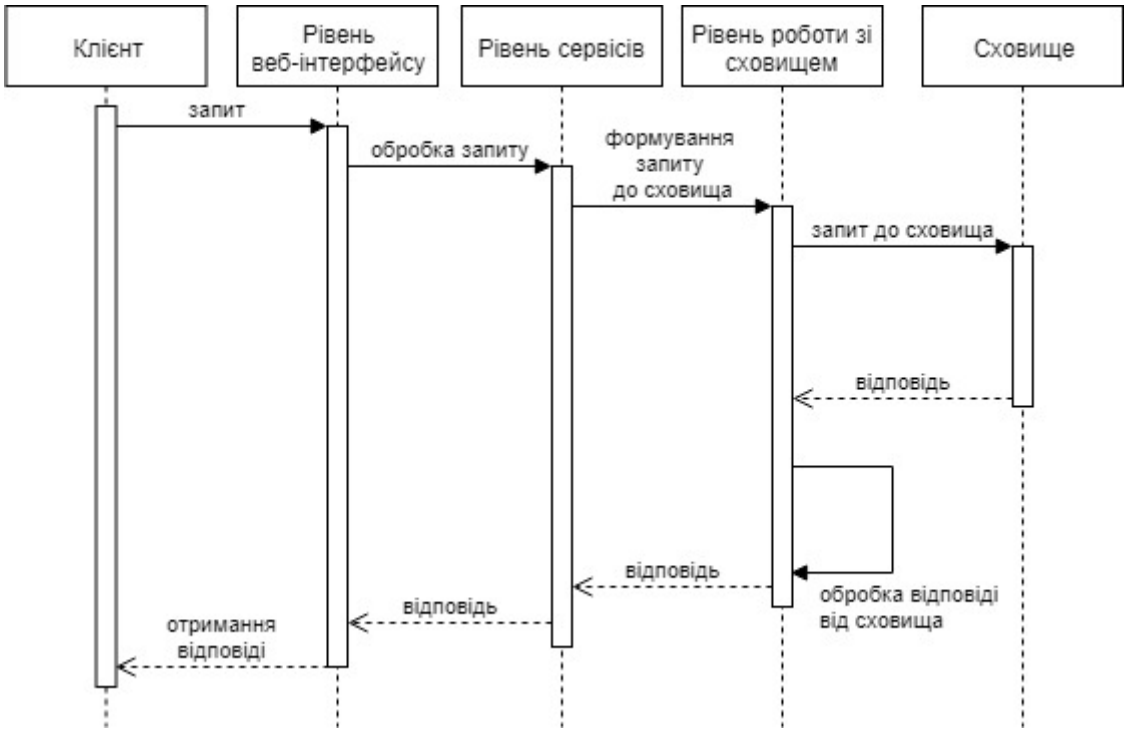


Рисунок 3.3 - Діаграма послідовностей мікросервісу каталогу продуктів

На цій діаграмі показано як працює мікросервіс починаючи від запита від клієнта. Мікросервіс каталогу продуктів розбитий на різні рівні - рівень веб-інтерфейсу, рівень сервісів та рівень роботи зі сховищем. Кожен рівень виконує свою окрему функцію. За рахунок цього забезпечується чітка та проста архітектура побудованого додатку. Рівень веб-інтерфейсу працює тільки х мережею, рівень сервісів виконує усю бізнес логіку, а рівень роботи зі сховищем має усю необхідну інформацію про сховище та працює тільки з ним.

### 3.3 Розробка прототипу мікросервісу каталогу продуктів

Для реалізації даного завдання була обрана мова програмування Java та Spring Framework. Він був обраних через те, що він надає абстракції, які можуть бути використані для роботи з Google Pub/Sub сервісом для реализації асинхронних зв'язків та Feign для реалізаціх синхронних зв'язків між мікросервісами. Для цього до проекту необхідно підключити модуль Spring Cloud.

Spring Cloud надає розробникам інструменти для швидкої побудови деяких загальних шаблонів у розподілених системах (наприклад, керування конфігураціями, виявлення сервісу, вимикачі, інтелектуальна маршрутизація, мікропроксі, шина керування, одноразові маркери, глобальні замки, вибори керівництва, розподілені ліки сесій, стан кластера). Його особливості:

– зосереджується на забезпеченні гарного досвіду роботи для типових випадків використання і механізму розширюваності для охоплення інших систем;

– розподілена / версійна конфігурація;

– реєстрація та виявлення послуг;

– маршрутизація;

– балансування навантаження;

– розподілені повідомлення.

Spring Cloud має декларативний підхід і часто ви отримуєте багато можливостей лише конфігурацію або анотацію.

У якості інтегратора хмарних додатків був обраний GCP, а у якості сховища — база даних MySQL, звернення до якої організовані через GCP сервіс Cloud SQL. Для розгортання додатку у кластері GCP були використані Docker та Kubernetes, які будуть розглянуті більш докладно нижче.

### 3.3.1 Docker

Docker (https://www.docker.com) — це програма, яка виконує віртуалізацію на рівні операційної системи. Docker використовується для запуску пакетів програм, які називаються контейнерами. Контейнери відокремлені один від одного і з'єднують власні програми, інструменти, бібліотеки та конфігураційні файли; вони можуть спілкуватися один з одним через чітко визначені канали. Всі контейнери виконуються одним ядром операційної системи і, таким чином, є більш легкими, ніж віртуальні машини. Контейнери створюються з зображень, які вказують їх точний вміст. Зображення часто створюються шляхом комбінування та модифікації стандартних зображень, завантажених з відкритих сховищ.

Docker розробляється в першу чергу для Linux, де він використовує можливості ізоляції ресурсів ядра Linux, такі як cgroups і простори імен ядра, а також файлову систему з можливістю об'єднання, наприклад OverlayFS та інші, щоб дозволити незалежним контейнерам працювати в одному екземплярі Linux, уникаючи накладні витрати на запуск і підтримку віртуальних машин. Підтримка просторів імен у ядрі Linux здебільшого ізолює вигляд програми від операційного середовища, включаючи дерева процесів, мережу, ідентифікатори користувачів і змонтовані файлові системи, тоді як cgroups ядра забезпечують обмеження ресурсів для пам'яті і процесора. На додаток до можливостей, які надаються ядром Linux (насамперед, cgroups і іменами), контейнер Docker, на відміну від віртуальної машини, не вимагає і не включає окремої операційної системи. Натомість, він спирається на функціональність ядра і використовує ізоляцію ресурсів для процесора і пам'яті, а також окремі простори імен, щоб ізолювати вигляд програми від операційної системи. Докер отримує доступ до функцій віртуалізації ядра Linux або безпосередньо за допомогою бібліотеки libcontainer.

Програмне забезпечення Docker — це служба, що складається з трьох компонентів:

– програмне забезпечення. Docker Daemon, який називається dockerd, є постійним процесом, який управляє контейнерами Docker і обробляє об'єкти контейнера. Демон слухає запити, надіслані за допомогою API Docker Engine. Клієнтська програма Docker, звана докером, надає інтерфейс командного рядка, який дозволяє користувачам взаємодіяти з демонами Docker;

– об'єкти. Об'єкти Docker - це різні об'єкти, які використовуються для збирання програми в Docker. Основними класами об'єктів Docker є зображення, контейнери та служби.

– реєстри. Реєстр Docker є сховищем зображень Docker. Клієнти Docker підключаються до реєстрів для завантаження зображень для використання або завантаження створених ними зображень. Реєстри можуть бути відкритими або приватними. Двома основними відкритими регістрами є Docker Hub і Docker Cloud.

Docker Compose — це інструмент для визначення та запуску багатокамерних докерівських додатків. Він використовує файли YAML для налаштування служб програми і виконує процес створення і запуску всіх контейнерів за допомогою однієї команди. Утиліта CLI-докера дозволяє створювати команди на декількох контейнерах одночасно, наприклад, створювати зображення, масштабувати контейнери, запускати контейнери, які були зупинені, і багато іншого. Команди, пов'язані з маніпуляцією зображеннями або інтерактивними параметрами користувача, не мають відношення до Docker Compose, оскільки вони адресують один контейнер. Файл docker-compose.yml використовується для визначення служб програми та включає різні параметри конфігурації. Наприклад, параметр build визначає параметри конфігурації, такі як шлях Dockerfile, параметр команди дозволяє перевизначити команди Docker за замовчуванням тощо. Docker Swarm забезпечує функціональність кластеризації для контейнерів Docker, що перетворює групу двигунів Docker в єдиний віртуальний движок Docker. Утиліта CLI дозволяє користувачам запускати контейнери Swarm, створювати маркери виявлення, перелічувати

вузли в кластері тощо. Утиліта CLI докерівського вузла дозволяє користувачам запускати різні команди для керування вузлами.

### 3.3.2 Kubernetes

Kubernetes (https://kubernetes.io) — це система оркестрування контейнерів з відкритим кодом для автоматизації розгортання, масштабування та управління додатками. Спочатку вона була розроблена компанією Google і тепер підтримується Cloud Computing Foundation. Вона має на меті забезпечити платформу для автоматизації розгортання, масштабування та роботи контейнерів додатків по кластерах хостів. Він працює з низкою контейнерних інструментів, включаючи Docker. Багато хмарних послуг пропонують платформу або інфраструктуру на основі Kubernetes як послугу, на якій Kubernetes можуть бути розгорнуті як сервіс, що надає платформу. Багато виробників також надають свої власні дистрибутиви Kubernetes.

Kubernetes визначає набір будівельних блоків, які спільно забезпечують механізми розгортання, підтримки та масштабування додатків на основі процесора, пам'яті або користувацьких метрик. Kubernetes вільно пов'язаний і розширюється для задоволення різних робочих навантажень. Ця розширюваність в значній мірі забезпечується API Kubernetes, який використовується внутрішніми компонентами, а також розширеннями та контейнерами, що працюють на Kubernetes. Платформа здійснює свій контроль над обчислювальними ресурсами та ресурсами зберігання, визначаючи ресурси як об'єкти, які потім можна керувати як такі. Основним блоком планування в Kubernetes є pod. Це додає більш високий рівень абстракції, групуючи контейнерні компоненти.

Структура складається з одного або декількох контейнерів, які гарантовано розташовуються на головній машині і можуть спільно використовувати ресурси.

Кожному пакунку Kubernetes присвоюється унікальний IP-адрес в межах кластера, який дозволяє програмам використовувати порти без ризику

конфлікту. Всередині контейнера всі контейнери можуть посилатися один на одного на локальному хості, але контейнер в межах одного блоку не має можливості безпосередньо звернутися до іншого контейнера в інший блок; для цього потрібно використовувати IP-адресу Pod. Підсилювач може визначити об'єм, наприклад, каталог локальних дисків або мережевий диск, і виставити його на контейнери в модулі. Керуючими панелями можна керувати вручну за допомогою API Kubernetes, або їх управління може бути делеговано контролеру. Такі томи також є основою для функцій ConfigMaps Kubernetes (для забезпечення доступу до конфігурації через файлову систему, видиму для контейнера) і Секретів (для забезпечення доступу до облікових даних, необхідних для безпечного доступу до віддалених ресурсів, надаючи лише ті облікові дані на файловій системі до уповноважених контейнерів).

Служба Kubernetes — це набір pod, які працюють разом, наприклад, один рівень багаторівневої програми. Набір pod, що складають послугу, визначається селектором міток. Kubernetes надає два режими виявлення послуг, використовуючи змінні середовища або використовуючи Kubernetes DNS. Відкриття служби призначає службі стабільну IP-адресу та ім'я DNS, а баланс навантаження здійснюється круговим способом до мережевих з'єднань цієї IP-адреси серед pod, що відповідають селектору (навіть якщо збій викликає переміщення пакунків від машини до машини). За замовчуванням служба виставляється в кластері (наприклад, зворотники заднього кінця можуть бути згруповані в службу, а запити з фронтальних блоків балансуються між ними), але сервіс також може бути виставлений за межі кластера (наприклад, для клієнтів для досягнення фронтових стручків).

Файлові системи в контейнері Kubernetes за промовчанням забезпечують ефемерне зберігання. Це означає, що перезавантаження контейнера знищить будь- які дані на таких контейнерах, і, отже, ця форма зберігання досить обмежена в будь-яких, крім тривіальних, програмах. Обсяг Kubernetes надає постійне сховище, яке існує протягом всього життя самої панелі. Таке сховище також може використовуватися як спільний дисковий простір для контейнерів у

модулі. Обсяги монтуються на певних точках монтування в контейнері, які визначаються конфігурацією пакуна, і не можуть встановлюватися на інші томи або посилання на інші томи. Той же обсяг може бути встановлений в різних точках дерева файлової системи різними контейнерами.

Kubernetes надає розбиття ресурсів, якими він керує, на неперекриваються набори, що називаються просторами імен. Вони призначені для використання в середовищах з багатьма користувачами, які поширюються на декілька команд або проектів, або навіть розділяють середовища, такі як розробка, тестування та виробництво.

Надамо опис ключових елементів Kubernetes YAML файлу, які стануть у нагоді для конфігурації екземпляру мікросервіса.

Розгортання Kubernetes Yaml містить такі основні характеристики: apiVersion, Kind, metadata, spec. ApiVersion вказує версію API об'єкта розгортання Kubernetes. Це змінюється між версіями Kubernetes. Kind описує тип об'єкта/ресурсу, який буде створено. У нашому випадку це об'єкт розгортання. Наведемо основний список об'єктів / ресурсів, що підтримуються Kubernetes: configmaps, deployments, endpoints, ingress, jobs, namespaces, nodes, pods, services. Metadata — це набір даних для однозначної ідентифікації об'єкта Kubernetes. Наведемо ключові метадані, які можна додати до об'єкта: labels, name, namespace, annotations. У розділі spec,ми оголошуємо бажаний стан і характеристики об'єкта, який ми хочемо мати. Наприклад, в специфікації розгортання ми б вказали кількість реплік, ім'я зображення тощо. Spec має три важливі підполя:

– Replicas — це забезпечить кількість екземплярів що працює весь час після розгортання;

– Selector — визначає мітки, які відповідають підрозділам для керування розгортаннями;

– Template — має власні метадані та специфікації. Spec буде мати всю інформацію про контейнер, яку повинен мати екземпляр. Інформація про

зображення контейнера, інформація про порт, змінні середовища, аргументи команд тощо.

Файл деплоймента мікросервіса продуктів містить 2 контейнери — один для самого додатку каталогу продуктів, а інший — cloudsql-proxy контейнер для спілкування з Cloud SQL сервісом. Усі змінні середовища містяться у окремому файлі configmap. Це наприклад пароль до БД, який зберігається у спеціальній сутності k8s secret, тому що це чуттеві дані.

Описаний вище підхід до розгортання додатків у кластері GCP дозволяє конфігорувати ващ мікросервіс на низькому рівні, що надає йому гнучкості та дозволяє контролювати обсяг ресурсів, що використовуються, та насолоджуватися перевагами мікросервісів у полній мірі.

Таким чином, розгортання розгортання мікросервісу у кластері GCP будо проведено на базі Docker та Kubernetes. Для цього треба створити Dockerfile, який містить у собі інструкції з побудування зображення та файли дескриптори розгортання Kubernetes, написаний мовою YAML. Kubernetes відображає застосування як Pods, які є одиницями, що представляють собою контейнер (або групу щільно з'єднаних контейнерів). Pod - це найменша розгортаюча одиниця на Kubernetes. За допомогою команди kubectl deploy *ваш дескриптор розгортання* ви змушуєте Kubernetes створити Deployment, який керує кількома копіями вашої програми, що називаються репліками, і планує їх виконання на окремих вузлах вашого кластера. У дескрипторі розгортання вашого додатку ви можете конфігурати багато різних параметрів, а саме: порт, на якому буде виконуватись ваш додаток; кількість екземплярів (pods) вашого додатку; кількісті ресурсів, виділених для єкземпляру додатка; змінні середовища, необхідні для запуску екземпляра та багато інших.

### 3.4 Порівняння мікросервісної та монолітної архітектур

Порівняння мікросервісного та монолітного архітектурних стилей з точки зору різних важливих аспектів проведемо на приклади інтернет-магазинів, які

надають один і тий самий функціонал, але реалізовані за допомогою різних стилей.

Мікросервісна архітектура вимагає Continuous Integration (CI) і Continuous Delivery (CD), щоб забезпечити часті розгортання різних сервісів.

Continuous Integration — є ключовим компонентом практики Agile Development. Основа даної практики полягає у постійному попаданні коду до центрального репозиторію після успішного запуску тестів. Основні цілі Continuous Integration - пошук і усунення потенційних проблем якомога швидше, поліпшення якості ПЗ і скорочення часу для випуску оновлень (рис. 3.4).

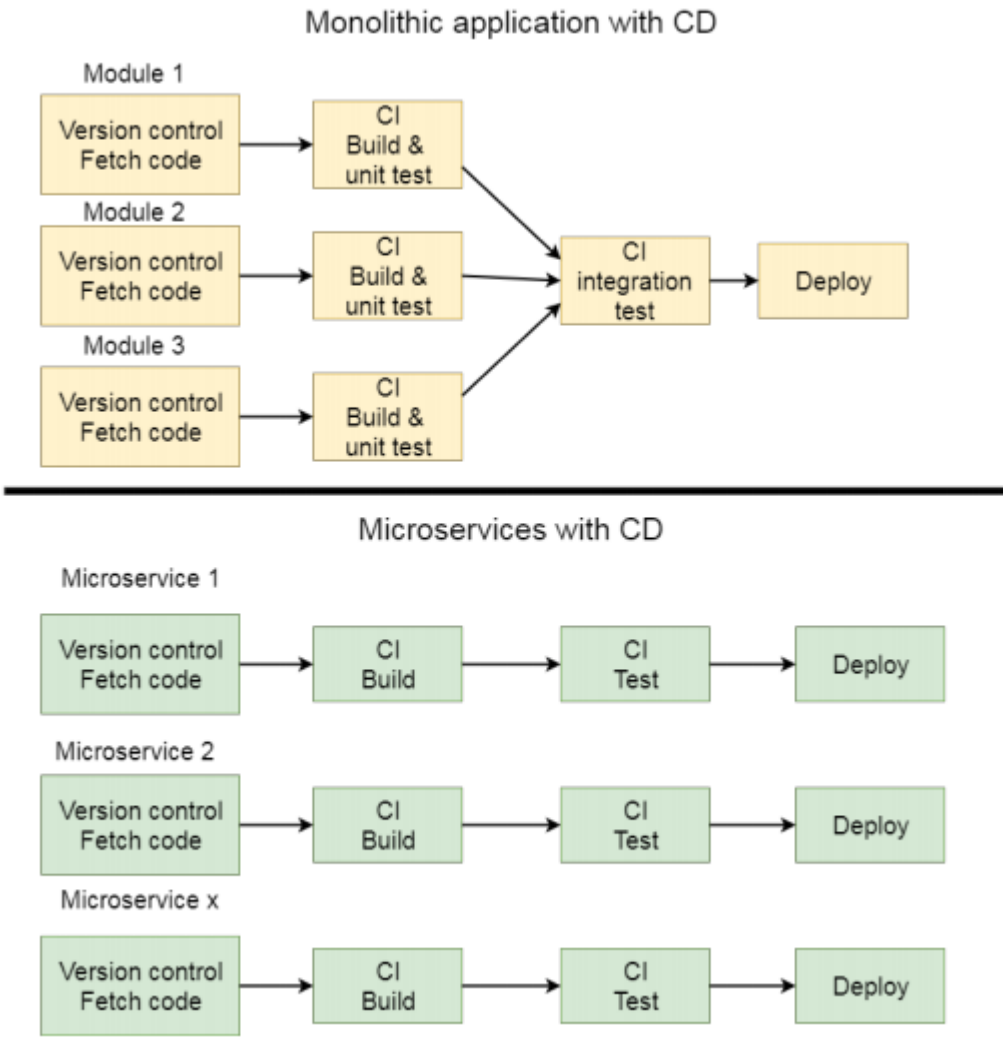


Рисунок 3.4 - CI та CD у мікросервісах та моноліті

Continuous Delivery — це серія практик, спрямованих на те, щоб оновлення програмного забезпечення відбувалися практично постійно. Дані методи гарантують швидке розгортання не змінюючи існуючий функціонал. Continuous Delivery є можливою завдяки різним оптимізаціям на ранніх етапах процесу розробки.

І мікросервісна архітектура, і монолітна архітектура можуть використовувати CI. Використання CD з мікросервісами набагато простіше, ніж з монолітом. Незважаючи на те, що CD можна використовувати з монолітом, це вимагає багато роботи, і тому CD рідко використовується. За допомогою CI і CD пайплайнів (коммунікаційних ліній) архітектура мікросервісу забезпечує більш швидкі цикди релізів, ніж монолітна архітектура. Це означає, що організації, що використовують мікросервіси разом з CD, можуть швидше реагувати на потреби клієнтів, ніж організації з монолітним додатком, що не використовує CD.

Таблиця 3.1 ілюструє основні відмінності при використанні Continious Delivery (CD) з монолітним додатком у порівнянні з використанням CD з мікросервісами.

Таблиця 3.1 Відмінность монолітного та мікросервісного додатків

| Категорія | Моноліт | Мікросервіс |
|---|---|---|
| Швидкість релізного циклу | Швидкий на початку, стає довшим зі зростанням бази коду | Повільний напочатку через технічні складноші, які мають мікросервіси. Далі швидше |
| Рефакторінг | Важко виконати, тому що зміни можуть впливати на кілька місць. | Легше і безпечніше, тому що зміни містяться всередині мікросервісу. |
| Розгортання | Весь моноліт повинен бути завжди розгорнутий. | Можуть бути розгорнуті частково, тільки один сервіс одночасно. |

*Продовження таблиці 3.1*

| Категорія | Моноліт | Мікросервіс |
|---|---|---|
| Мова програмування | Важко змінити, тому що база коду велика | Мову та інструменти можна вибрати для кожного сервісу. |
| Масштабування | Масштабування означає розгортання всього моноліту. | Масштабування може виконуватися для кожного сервісу. |
| DevOps навички | Не потрібно багато, оскільки кількість технологій обмежена | Кілька різних технологій, вимагається багато навичок DevOps |
| Зрозумілість | Важко зрозуміти, через високу складність. | Легко зрозуміти, тому що база коду суворо модульна, а сервіси використовують SRP. |
| Транзакції | Легко використовувати ACID транзакції, що постачаються СУБД | Важко реалізувати. Остаточна послідовність має бути узгоджена для деяких випадків. |
| CI, CD | CI можливий і повинен використовуватися. CD важко досягти | CI є необхідним, CD. має бути використаний CD забезпечує більш швидкий цикл релізу. |

Як ми можемо бачити CI фаза тестування може стати вузьким місцем при використанні монолітної архітектури та CI. Крім того, фаза розгортання набагато більше з монолітною архітектурою, тому що ця фаза відповідає за розгортання цілого додатку, порівняно з фазою розгортання мікросервісів, яка відповідає за розгортання тільки одного сервісу.

Для розробітників мікросервіси можуть бути привабливими завдяки тому, що команда може обрати для мікросервіса будь-яку мову програмування та будь-які фреймворки; час збірки проекту та виконання тестів є значно меншим для окремого мікросервіса, що проскорює процес розробки; код мікросервісу зазвичай є значно чистішим, завздякі тому, що база коду порівнянно невелика і якість коду легко контролювати та проводити рефакторінг за необхідністю.

З точку зору бізнесу мікросервіси є привабливими завдяки можливості реалізавути нові функції додатка незалежно одна від одної; короткому релізному циклу; зменшенню ризикованності релізу - несправність у одному з мікросервісів не призведе до несправності усього додатка; можливості масштабувати тільки окремий сервіс, якому це дійсно потрібно.

Далі буде складена порівняльна характеристика часу розгортання мікросервісного додатку з монолітним додатком, а також порівняльна характеристка витрат на інфраструктуру.

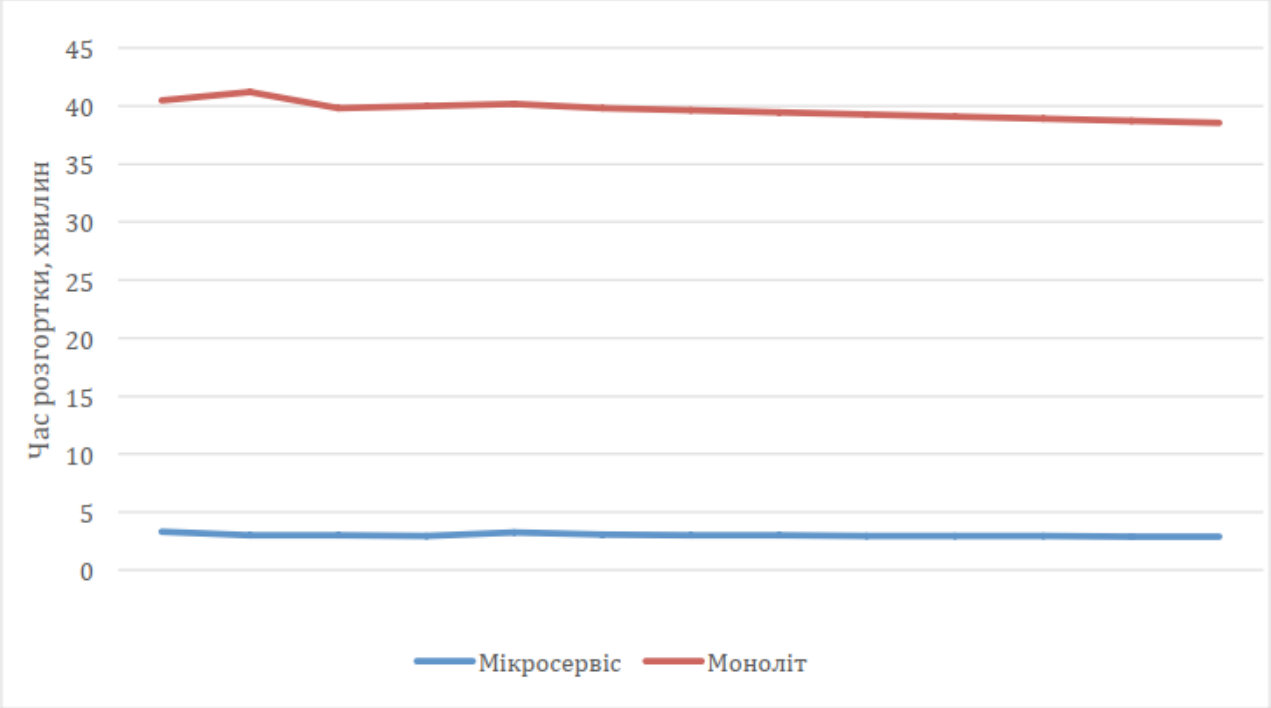


Рисунок 3.5 - Графік порівняльної характеристики часу розгортання мікросервісного та монолітного додатку

На рисунку можна бачити, що приблизний час розгортай мікросервісного додатку становить 4 хвилини. Цей відносно малий час забезпечується малими розмірами додатку. В той самий час, розгортка монолітного додатку займає приблизно 40 хвилин. Це стається через те, що монолітний додаток більш важкий та включає в себе багато модулів та функцій, в той час як мікросервіс виконує лише одну функцію.

Усі результати досліджень наведені у таблиці 3.2.

Як бачимо, з точки зору часу розгортання, мікросервісний додаток значно випереджає аналогічний монолітний додаток.

Таблиця 3.2 - Час розгортання мікросервіса та моноліта

| Тривалість розгортання мікросервісу, хв. | Тривалість розгортання моноліту, хв. |
|---|---|
| 3,33 | 40,5 |
| 3 | 41,2 |
| 3,02 | 39,8 |
| 2,96 | 39,99 |
| 3,23 | 40,2 |
| 3,036 | 39,795 |
| 3,012 | 39,614 |
| 2,988 | 39,433 |
| 2,964 | 39,252 |
| 2,94 | 39,071 |
| 2,916 | 38,89 |
| 2,892 | 38,709 |
| 2,868 | 38,528 |

На рисунку можна бачити, що плата за розгортання мікросервісного додатку значно нижче, ніж плата за розгортання монолітного додатку. В першку чергу, така значна різниця виникає за рахунок розміру додатку. Великий перепад місячної плати за розгортання монолітного додатку виникає через те, що мікросервіси можно незалежно масштабувати при збільшенні навантаженя, у той час як у випадку монолітних додатків треба бути розгорнути додаткові екземпляри усього додатку. Витрати на інфоаструктуру були разоаховані за допомогою GCP Pricing Calculator, який дозволяє обрати Google Cloud Services, які буде використовувати ваш додаток, кількість екземплярів, та інші параметри.

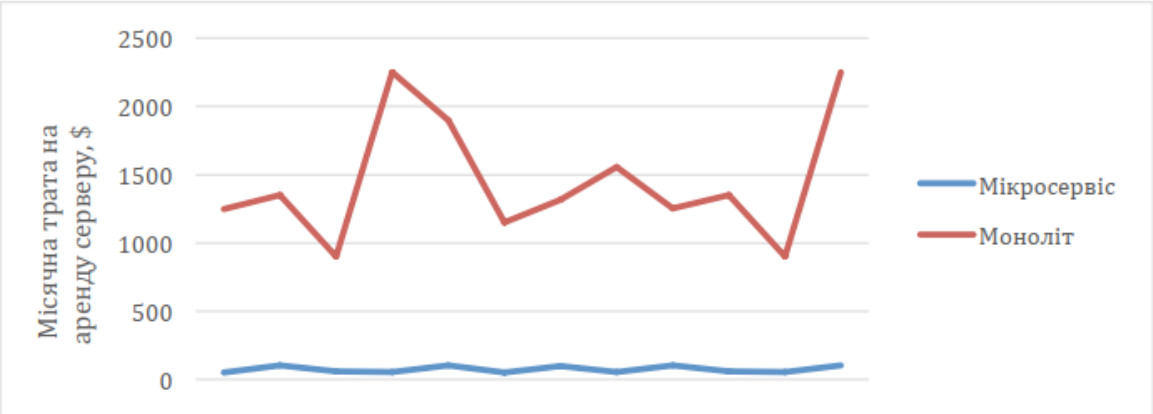


Рисунок 3.6 - Графік порівняльної характеристики місячних витрат на інфраструктуру для розгортання мікросервісного та монолітного додатку

У таблиці 3.3 наведені дані щодо вартості у залежності від можливих конфігурацій додатка - вартості в залежності кількості екземплярів та в залежності від сервісів, шо використовуються.

Для того, щоб розрахувати приблизну вартість підтримки інтернет-магазину на базі мікросервісної архітектури, схема якого була розглянута у розділі 3, треба помножити середню вартість підтримки одного мікросервіса на кількість мікросервісів у додатку, що в нашому випадку складає 9. Таким чином

підтримка додатку на базі мікросервісів буде коштувати 677,99 дол. США, а підтримка аналогічного монолітного додатку буде коштувати 1152,23 дол. США.

Таблиця 3.3 - Місячні витрати на інфраструктуру для мікросервісного та монолітного додатків

| Аренда сервера для мікросервісного додатку, дол. США | Аренда сервера для монолітнгого додатку, дол. США |
|---|---|
| 52,61 | 1250,93 |
| 105,17 | 1349,64 |
| 59,18 | 899,76 |
| 53,89 | 2249,41 |
| 105,22 | 1897,74 |
| 52,66 | 1150,91 |
| 98,6 | 1319,56 |
| 53,61 | 1557,14 |
| 105,18 | 1251,93 |
| 59,19 | 1349,65 |
| 53,9 | 899,77 |
| 105,23 | 2249,42 |

Як бачимо, з точки зору місячних витрат на інфраструктуру додатка, мікросервісний додаток значно випереджає аналогічний монолітний додаток.

Для порівняння мікросервісної та монолітної архітектури в якрості рішення для e-commerce додатків використаємо такий метод приняття рішень як лінійна аддитивная згортка з вісовими коеффіцієнтами. Базуючись на особливостях e- commerce додатків та на загальних вимогах до веб-додатків оберемо критерії, за якими ми будемо порівнювати альтернативи:

— тривалість розгортання;

— вартість підтримки;
— масштабуємість;
— складність впровадження на початковому етапі;
— тривалість релізного циклу;
— продуктивність;
— простота тестування;
— стабільність системи.

Далі визначимо вагові коефіцієнти для кожного з критеріїв методом ранжування. Найбільш важливий критерій отримає 8 балів, кому що кількість критеріїв 8, менш важливий отримає 7 балів і так далі. Потім кількість балів для кожного з критеріїв поділимо на (1+2+..8) і отримаємо вагу критерію.

Після цього кожна альтернатива отримає оцінки по кожному критерію. Оцінка по критерію буде помножена на ваговий коефіцієнт відповідного критерія, а сума таких оцінок для альтернативи визначить її підсумкову оцінку. Цей процес може бути описаний такою формулою:

$$Z^* = \max \sum_{j=1}^{n} \alpha_j \beta_j a_{ij}, \tag{3.1}$$

де $\alpha_j$ - нормуючий множник,

$\beta_j$ - ваговий коефіцієнт, відображаючий відносний кожного критерія у загальний критерій.

Наведемо отримані вагові коефіцієнти у таблиці 3.4. Далі надаємо оцінку кожній альтернативі за критерієм та нормалізуємо її. Кінцевий результат можна побачити у таблиці 3.5.

Можна зробити висновок, що мікросервісна архітектура більше відповідає вимогам великої e-commerce системи за заявленими крітеріями, але відрив між ними не є великим. Обрані крітерії є загальними для систем і для більш точного порівняння альтернатив треба враховувати особливості конкретного проекту. Слід сказати, що кожна з архітектур займає свою нішу та може показати себе з

гарної сторони на різних проектах. Також для успіху кожного окремого проекту має значення не тільки архітектура, що використовується, а і виконання базових принципів побудування ПО, наприклад такі як SOLID.

Таблиця 3.4 - Вагові коефіцієнти критеріїв

| Крітерій | Ваговий коефіцієнт |
|---|---|
| Тривалість релізного циклу | 0,22 |
| Продуктивність | 0,19 |
| Стабільність системи | 0,16 |
| Вартість підтримки | 0,16 |
| Масштабуємість | 0,1 |
| Простота тестування | 0,08 |
| Тривалість розгортання | 0,06 |
| Складність впровадження на початковому етапі | 0,03 |

Таблиця 3.5 - Результати порівняння мікросервісної та монолітної архітектур для використання у e-commerce додатках

| Критерій/альтернатива | Моноліт | Мікросервіс |
|---|---|---|
| Тривалість релізного циклу | 0,36 * 0,22 = 0,079 | 0,64 * 0,22 = 0,14 |
| Продуктивність | 0,57 * 0,19 = 0,108 | 0,43 * 0,19 = 0,082 |
| Стабільність системи | 0,33 * 0,16 = 0,053 | 0,67 * 0,16 = 0,107 |
| Вартість підтримки | 0,38 * 0,16 = 0,061 | 0,62 * 0,16 = 0,099 |
| Масштабуємість | 0,31 * 0,1 = 0,031 | 0,69 * 0,1 = 0,069 |
| Простота тестування | 0,64 * 0,08 = 0,0512 | 0,36 * 0,08 = 0,029 |
| Тривалість розгортання | 0,31 * 0,06 = 0,019 | 0,69 * 0,06 = 0,041 |

*Закінчення таблиці 3.5*

| Критерій/альтернатива | Моноліт | Мікросервіс |
|---|---|---|
| Складність впровадження на початковому етапі | 0,75 * 0,03 = 0,023 | 0,25 * 0,03 = 0,008 |
| Підсумок | 0,425 | 0,575 |

Зважаючи на отримані дані можна зробити висновок, що на початковому етапі розробки треба проектувати систему як моноліт і впроваджувати мікросервісну архітектуру у разі зіткнення з її обмеженнями, які виникають зі зростанням коду. Маючи робочий додаток, який надає бажану функціональність стає значно легше виділити функціональні області, які можуть стати окремими мікросервісами так, щоб вони відповідали принципу єдиного обов'язку.

### 3.5 Висновки по розділу 3

Для того щоб підтвердити перевагу мікросервісної архітектури для систем електронної комерції був обраний один із мікросервісів типової e-commerce системи, а саме каталог продуктів. Він побудований на базі Spring Framework з використанням таких мов програмування з екосистеми Java, як Java, Kotlin та Groovy. Мікросервіс розгортається на базі Google Cloud Platform за допомогою Kubernetes Engine та відповідного дескріптору розгортання, тобто з використанням патерну розгортання "Service Instance per Container". У якості сховища використовується база даних MySQL, яка також знаходиться у хмарному середовищі та доступ до якої надається за допомогою використання Cloud SQL сервісу.

У ході роботи проаналізовані такі параметри як час розгортання, витрати на підтримку додатку виходячи з використаних Google Cloud Platform сервісів та

їх характеристик, таких як пам'ять та CPU, доступність додатку, час відклику, складність розробки та розгортання. Ці дані є практично корисними для складних e-commerce систем, які навштохнулся на обмеження традиційної монолітної архітектури та які незадоволені швидкістю релізів нової функціональності.

# ВИСНОВКИ

У ході роботи були проаналізовані традиційний підхід до побудування складних систем електронної комерції та мікросервісний. Були виявлені переваги та недоліки обох підходів та умови, в яких доцільно використовувати тий чи інший. Також були розглянуті існуючі рішення для побудування систем електронної комерції, а саме e-commerce платформи Oracle Commerce та SAP Hybris, які є гарним варіантом для e-commerce систем середньої складності на початку роботи інтернет-магазину, але накладають великі обмеження на розвиток системи. Таким чином, раніше чи пізніше системам електронної комерції доводиться відмовлятися від використання таких платформ.

У якості альтернативи був розглянутий підхід побудування систем електронної комерції з використанням мікросервісного архітектурного стилю та таких інтеграторів як Google Cloud Platform або AWS, які можуть значно спростити не тільки розгортання додатків, але і побудування інфраструктури системи у цілому. Також вони надають різні корисні сервіси, такі як SQL або NoSQL сховище, сервіс асинхронного обміну повідомленнями Publisher/Subscriber, обмеження доступу до ваших кінцевих точок API за допомогою ключів. У залежності від того який тип машини ви обрали, ви можете взагалі не думати про масштабування та інфраструктуру вашого додатка та довіритися інтегратору, який може автоматично масштабувати ваш додаток, або отримати доступ до всіх конфігурацій вашого додатка та мати усе під контролем.

Таким чином, для того щоб підтвердити перевагу мікросервісної архітектури для систем електронної комерції був обраний один із мікросервісів типової e-commerce системи, а саме каталог продуктів. Він побудований на базі Spring Framework з використанням таких мов програмування з екосистеми Java, як Java, Kotlin та Groovy. Мікросервіс розгортається на базі Google Cloud Platform за допомогою Kubernetes Engine та відповідного дескріптору розгортання, тобто з використанням патерну розгортання "Service Instance per Container". У якості

сховища використовується база даних MySQL, яка також знаходиться у хмарному середовищі та доступ до якої надається за допомогою використання Cloud SQL сервісу.

У ході роботи проаналізовані такі параметри як час розгортання, витрати на підтримку додатку виходячи з використаних Google Cloud Platform сервісів та їх характеристик, таких як пам'ять та CPU, доступність додатку, час відклику, складність розробки та розгортання. Ці дані є практично корисними для складних e-commerce систем, які навштохнулся на обмеження традиційної монолітної архітектури та які незадоволені швидкістю релізів нової функціональності.

Можна зробити висновок, що монолітний додаток може задовільнити усі потреби як бізнесу, так і інженерів проекту на початковій стадії його розвитку. В той час як мікросервісна архітектура більш складна у впроваджені, хоча і має багато переваг та вирішує деякі проблеми монолітної архітектури. Тому можна порекомендувати мікросервісну архітектуру великим та зрілим e-commerce рішенням, для яких управління розробкою та релізним циклом монолітного додатка стає дедалі складнішим. Для невеликих та "молодих" рішень зусилля витрачені на розробку та підтримку мікросервісних додатків можуть бути більшими ніж отриманий результат.

# ПЕРЕЛІК ПОСИЛАНЬ

1. Martin Fowler Microservice Trade-Offs [Електронний ресурс] // Сайт Мартіна Фаулера URL: https://www.martinfowler.com/articles/microservice-trade-offs.html (дата звернення: 17.04.2020)

2. Kasun Indrasiri Microservices in Practice - Key architectural concepts of an MSA [Електронний ресурс] // WSO2 API Management Platform Library URL: https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of- an-msa/ (дата звернення: 17.04.2020)

3. Chris Richardson Choosing a Microservices Deployment Strategy [Електронний ресурс] // Технічний блог nginx 10.02.2016 URL: https://www.nginx.com/blog/deploying-microservices/ (дата звернення: 17.04.2020)

4. AWS documentation [Електронний ресурс] // Офіційний сайт Amazon Web Services URL: https://docs.aws.amazon.com/index.html?nc2=h_ql_doc (дата звернення: 17.04.2020)

5. Google Cloud Platform documentation documentation [Електронний ресурс] // Офіційний сайт Google Cloud Platform URL: https://cloud.google.com/docs/ (дата звернення: 17.04.2020)

6. Michael Hoffman, Erin Schnabel, Katherine Stanley Microservices Best Practices for Java [Електронний ресурс] // IBM Redbooks First Edition (December 2016) URL: https://www.redbooks.ibm.com/redbooks/pdfs/sg248357.pdf (дата звернення: 17.04.2020)

7. Holger Knoche, Wilhelm Hasselbring Using Microservices for Legacy Software Modernization [Електронний ресурс] // IEEE Software 35 2018 URL:https://www.researchgate.net/publication/324961770_Using_Microservices_for_L egacy_Software_Modernization (дата звернення: 17.04.2020)

8. Miika Kalske Transforming monolithic architecture towards microservice architecture [Електронний ресурс] // University of Helsinki, Department of Computer

Science 19.11.2017 URL: https://core.ac.uk/download/pdf/157587910.pdf (дата звернення 4.06.2020)

9. Hamzeh Khazaei, Cornel Barna, Marin Litoiu Performance Modeling of Microservice Platforms Considering the Dynamics of the underlying Cloud Infrastructure [Електронний ресурс] // Cornell University 9.02.2019 URL: https://arxiv.org/pdf/1902.03387.pdf (дата звернення 5.06.2020)

10. Константин Горский Структурирование множества альтернатив с использованием критериев [Електронний ресурс] URL: http://www.gorskiy.ru/Articles/Dmss/part06.html (дата звернення 5.06.2019)

11. Sam Newman Building Microservices: Designing Fine-Grained Systems :навч. посіб. - O'Reilly Media, 2015. - 280 с.

12. Chris Richardson Microservices Patterns: навч. посіб. - Manning, 2016. - 452 с.

13. Kubernetes documentation [Електронний ресурс] // Офіційний сайт Kubernetes URL: https://kubernetes.io/docs/home/ (дата звернення 5.06.2020)

14. Spring Cloud documentation documentation [Електронний ресурс] // Офіційний сайт Spring URL: https://spring.io/projects/spring-cloud (дата звернення 5.06.2020)

15. Rauf Aliev Oracle Commerce vs SAP Hybris Commerce, both On-prem and Cloud [Електронний ресурс] // Hybris competency center EPAM Systems URL: https://hybrismart.com/2018/05/20/a-detailed-comparison-oracle-commerce-cloud-and- on-premise-aka-atg-vs-sap-hybris-commerce-cloud/ (дата звернення 5.06.2020)

# ДОДАТОК А

## *Файли розгортання мікросервісу за допомогою Kubernestes*

```
kind: Service apiVersion: v1 metadata:
    name: product-catalog namespace: diploma labels:
        app: product-catalog environment: stubs serviceExposition: internal spec:
    type: ClusterIP selector:
        app: product-catalog environment: stubs ports:
        - name: http protocol: TCP port: 80
            targetPort: http-proxy

kind: Deployment apiVersion: apps/v1beta1 metadata:
    name: product-catalog namespace: diploma spec:
    replicas: 2 minReadySeconds: 10 selector:
        matchLabels:
            app: product-catalog environment: stubs strategy:
        type: RollingUpdate rollingUpdate: maxSurge: 25% maxUnavailable: 0
template: metadata: labels:
                app: product-catalog
      environment: stubs spec:
  restartPolicy: Always terminationGracePeriodSeconds: 30
  automountServiceAccountToken: false securityContext: fsGroup: 101
  runAsNonRoot: true containers:
  - name: product-catalog
      image: ${GOOGLE_IMAGE_TAG}:${PRODUCT_API_VERSION}
      imagePullPolicy: Always
      ports:
      - containerPort: 8080
          name: http resources: requests:
            memory: "500Mi" cpu: "1000m" limits:
            memory: "1000Mi" cpu: "1000m" securityContext:
          readOnlyRootFilesystem: true runAsUser: 1001 env:
          - name: DB_USER valueFrom:
              secretKeyRef:
                name: product-catalog-mysql-inst-1 key: user
          - name: DB_PASSWORD valueFrom:
              secretKeyRef:
                name: product-catalog-mysql-inst-1 key: password
      envFrom:
          - configMapRef:
```

```
name: product-catalog-env-config readinessProbe: httpGet:
path: /api/v1/admin/health port: 8080 initialDelaySeconds: 60
periodSeconds: 10
failureThreshold: 10 livenessProbe: httpGet:
path: /api/v1/admin/health port: 8080 initialDelaySeconds: 60
periodSeconds: 10 failureThreshold: 10 - name: cloudsql-proxy
image: gcr.io/cloudsql-docker/gce-proxy:1.11 command:
["/cloud_sql_proxy",
"-instances=product-flex:europe-west1:
product-catalog-mysql-inst-1=tcp:3306",
" -
credential_file=/secrets/cloudsql/credentials"] securityContext:
runAsUser: 2 # non-root user allowPrivilegeEscalation: false
volumeMounts:
- name: product-catalog-mysql-inst-1-credentials mountPath:
/secrets/cloudsql readOnly: true volumes:
name: product-catalog-mysql-inst-1-credentials secret:
secretName: product-catalog-mysql-inst-1-
credentials
name: product-catalog-mysql-inst-1-credentials secret:
secretName: product-catalog-mysql-inst-1-
credentials

apiVersion: v1 kind: ConfigMap metadata:
name: product-catalog-env-config namespace: diploma data:
OTHER SERVICE URL: other-service:80
```