https://doi.org/10.31891/2219-9365-2024-79-24
УДК 004

КОРНАГА Ярослав
Національний Технічний Університет України “Київський політехнічний інститут імені Ігоря Сікорського”
y.kornaga@kpi.ua
ГУБАРЄВ Олександр
Національний Технічний Університет України “Київський політехнічний інститут імені Ігоря Сікорського”
https://orcid.org/0000-0002-0924-4103
gubarev.alexandr@gmail.com

АЛГОРИТМ ПЕРЕХОДУ ВІД МОНОЛІТНОЇ ДО МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ІЗ ЗАСТОСУВАННЯМ МЕТОДІВ КЛАСТЕРИЗАЦІЇ ГРАФІВ

У даній статті представлений підхід до трансформації монолітної архітектури в мікросервісну з використанням bottom-up стратегії декомпозиції. В данному підході використовується статичний аналіз коду для представлення монолітної системи у вигляді зв’язного графа. На другому етапі, використовуючи алгоритми кластеризаціїі, відбувається кластеризація графа. На третьому — оптимізація отриманих кластерів.

Ключові слова: мікросервіси, декомпозиуія системи, кластеризація графу, комбінаторна оптимізація.

KORNAGA Yaroslav, GUBAREV Oleksandr
National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”

ALGORITHM FOR TRANSITION FROM MONOLITHIC TO MICROSERVICE ARCHITECTURE USING GRAPH CLUSTERING METHODS

The transition from a monolithic architecture to a microservice architecture using graph clustering methods is an effective approach that allows you to divide the system into logically connected components.

Static analysis is the first step in making the transition from a monolithic to a microservice architecture. It allows you to highlight components (nodes) and establish relationships between them, as well as the weight of these relationships, depending on the type of components it connects.

After constructing a connected graph based on static analysis, the next important step is graph clustering. Graph clustering provides an automated approach to partitioning the monolith, which allows for the detection of natural boundaries between components. The use of clustering algorithms, such as Louvain, Spectral Clustering or Girvan-Newman, simplifies and speeds up the process of module identification.

The next important stage in the process of transforming a monolithic architecture into a microservice is the optimization of the resulting clusters. After completing the practical task of clustering a graph with 5 vertices, 2 clusters were obtained, the first included 2 nodes, and the second - 3. But after performing cluster optimization, a different result was obtained - one of the graph vertices was moved from the second to the first cluster. This happened due to the reduction of intercluster dependencies, which ensured the efficient distribution of components and ultimately increased the performance and scalability of the system.

This article presents an approach to the transformation of a monolithic architecture into a microservice one using a bottom-up decomposition strategy. This approach uses static code analysis to represent a monolithic system in the form of a connected graph. At the second stage, the graph is clustered using clustering algorithms. The third is optimization of the obtained clusters.

Key words: microservices, system decomposition, graph clustering, combinatorial optimization.

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ’ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

Перехід від монолітної архітектури до мікросервісної є актуальним напрямком розвитку сучасних програмних систем, що дозволяє підвищити гнучкість, масштабованість і швидкість розробки та впровадження програмного забезпечення. У цьому процесі великі монолітні додатки розбиваються на невеликі, автономні мікросервіси, кожен з яких відповідає за окрему функціональність або бізнес-процес [1].

Основними мотивуючими факторами для переходу до мікросервісної архітектури є:

1. Гнучкість і швидкість вдосконалення: Мікросервіси дозволяють здійснювати окреме вдосконалення і впровадження нових функцій без необхідності розгортання всього додатку.
2. Масштабованість: Кожен мікросервіс може бути масштабований незалежно від інших, що дозволяє оптимізувати використання ресурсів і забезпечити високу доступність системи.
3. Ізоляція помилок: Проблеми в одному мікросервісі зазвичай не впливають на інші, що полегшує відладку і підтримку системи.
4. Технологічна гетерогенність: Різні мікросервіси можуть використовувати різні технології і стеки, що дозволяє використовувати найкращі інструменти для кожного конкретного завдання.

Перехід від монолітної архітектури до мікросервісної, хоча і має багато переваг, зазвичай супроводжується рядом складнощів і викликів, які варто урахувати:

1. Системна складність: Управління багатьма невеликими сервісами потребує ефективного моніторингу, координації і управління версіями.
2. Комунікація між сервісами: Необхідність ефективної організації міжсервісних взаємодій і забезпечення консистентності даних.
3. Тестування і відладка: Складність тестування, особливо інтеграційного, через розподілену природу системи [2].

Існують різні стратегії рефакторингу монолітної системи в мікросервісну архітектуру. Їх можна класифікувати за трьома категоріями, так званими стратегіями декомпозиції: top-down, bottom-up та hybrid підходи.

Однією з головних проблем, з якою доводиться стикатися при використанні будь-якої стратегії декомпозиції є ідентифікація мікросервісів із високою цілісністю та слабким зв'язком.

ФОРМУЛЮВАННЯ ЦІЛЕЙ СТАТТІ

Метою даної статті є представлення алгоритму декомпозиції моноліта у мікросервіси із високою цілісністю і слабким зв'язком з використанням методів кластеризації графів і методів для комбінаторної оптимізаціїї утворених кластерів.

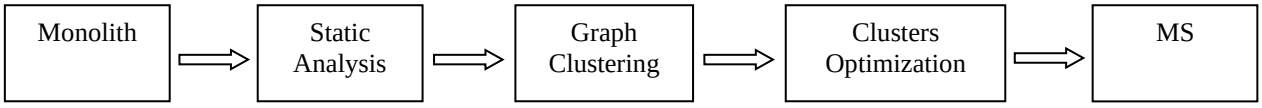

Рис.1.1 Алгоритм переходу від моноліту до мікросервісів

ВИКЛАД ОСНОВНОГО МАТЕРІАЛУ

1. Стратегії рефакторингу монолітної системи

Top-Down підхід (Рис.1.2): орієнтується на початкове визначення високорівневих бізнес-функцій і вимог до системи, які потім розбиваються на менші компоненти - мікросервіси. Цей підхід спрощує процес розробки, покращує масштабованість та забезпечує гнучкість системи. Основними етапами цього підходу є: аналіз бізнес-функцій і вимог (починаючи з високорівневого аналізу, визначаються основні бізнес-функції та вимоги до системи. Цей етап включає розуміння бізнес-процесів, що виконуються системою, та ідентифікацію ключових бізнес-компонентів), розбиття на функціональні області(бізнес-функції і вимоги поділяються на окремі функціональні області, які можуть стати кандидатами для мікросервісної реалізації. Це може включати ідентифікацію окремих процесів або бізнес-правил, які можуть бути реалізовані як окремі сервіси), визначення мікросервісних границь(на основі функціональних областей визначаються мікросервісні границі - інтерфейси та контракти, які визначають, як сервіси взаємодіють між собою та з іншими системами. Цей етап також включає визначення інтерфейсів API, які дозволяють комунікацію між мікросервісами), реалізація мікросервісів(на основі визначених мікросервісних границь розробляються і реалізуються окремі мікросервіси. Кожен мікросервіс відповідає за виконання конкретної функціональної області і має чітко визначені внутрішні правила і логіку), інтеграція і тестування (Після реалізації мікросервісів їх необхідно інтегрувати та виконати комплексне тестування, включаючи інтеграційне тестування для визначення коректності взаємодії між сервісами), моніторинг і підтримка(після впровадження мікросервісів в експлуатацію, необхідно налаштувати моніторинг та забезпечити підтримку, щоб впевнитися в їхній надійності та ефективності).

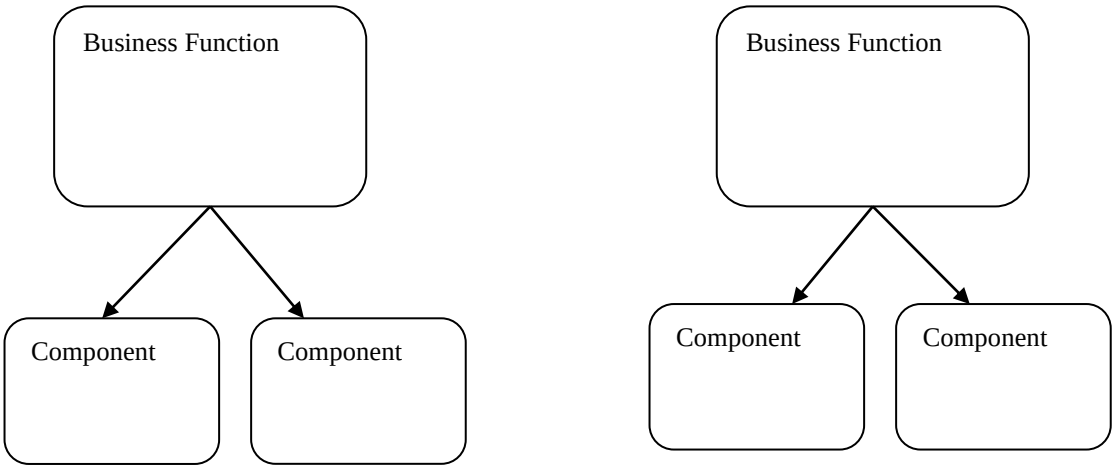

Рис.1.2 Top-Down підхід

До переваг цього підходу можна віднести:

- Простота управління: Цей підхід спрощує процес розробки, оскільки він дозволяє концентруватися на високорівневих бізнес-вимогах і функціях.
- Гнучкість і масштабованість: Завдяки чіткому визначенню мікросервісних границь, система стає більш гнучкою і масштабованою, що дозволяє ефективно відповідати на зміни в бізнес-потребах.

Недоліками є:

- Складність визначення мікросервісних границь: Не завжди просто точно визначити, які функції повинні бути реалізовані як окремі мікросервіси, особливо в складних системах з великою кількістю функціональності.
- Комплексність інтеграції: Інтеграція різних мікросервісів може виявитися складною задачею, особливо при потребі забезпечення консистентності даних і ефективної комунікації.

Top-Down підхід до розробки мікросервісної архітектури є корисним інструментом для організації та створення складних систем, проте його успішність залежить від глибини і точності аналізу бізнес-вимог і функцій.

Bottom-Up підхід (Рис.1.3): розглядається як протилежність Top-Down підходу. Він полягає в початковому розробленні невеликих компонентів або сервісів, які пізніше об'єднуються для створення більш складної системи. Основна ідея полягає у тому, щоб перш ніж визначити високорівневі бізнес-функції і вимоги, розробляти індивідуальні сервіси і поступово їх інтегрувати у єдину систему.

Основними етапами цього підходу є: розробка незалежних мікросервісів: початково розробляються невеликі, самодостатні сервіси, які реалізують обмежену функціональність або певний бізнес-процес. Кожен сервіс має власний інтерфейс, базується на внутрішніх правилах і має власну базу даних (якщо необхідно); інтеграція і комунікація: після створення незалежних сервісів вони інтегруються один з одним із використанням механізмів комунікації, таких як API, шини повідомлень або інші технології. Цей етап включає встановлення зв'язків із сервісами, які можуть бути необхідними для спільного вирішення бізнес-завдань; розширення та масштабування: Після успішної інтеграції і випробувань сервісів можна розглядати їх розширення і масштабування. Цей етап включає оптимізацію та реорганізацію коду, а також впровадження моніторингу та управління версіями; ітерації і підтримка: Bottom-Up підхід сприяє ітеративному покращенню і розвитку системи, оскільки нові функції можуть додаватися шляхом створення нових мікросервісів або розширенням існуючих. Підтримка системи включає надання підтримки, виправлення помилок та впровадження оновлень.

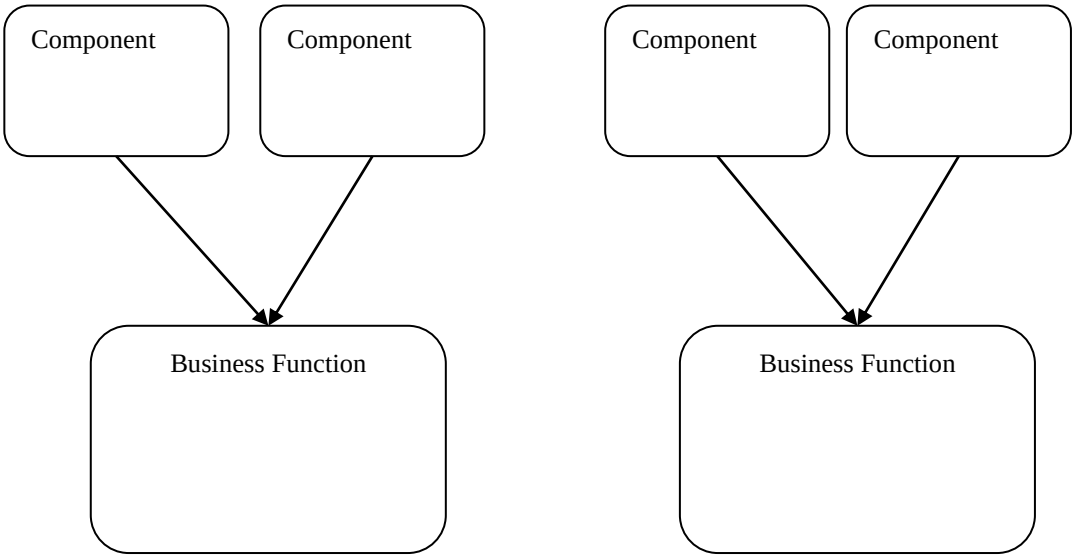

Рис.1.3 Bottom-Up

До переваг цього підходу можна віднести:

- Гнучкість і швидкість розробки: Розробка мікросервісів може відбуватися незалежно один від одного, що сприяє швидкому впровадженню нових функцій та експериментам.
- Масштабованість: Кожен сервіс може масштабуватися незалежно від інших, що дозволяє легше управляти завантаженням і ресурсами.

Недоліками є:

- Складність інтеграції: Інтеграція багатьох незалежних сервісів може бути складною

задачею, особливо при потребі забезпечення консистентності даних та ефективної комунікації.

- Управління залежностями: Важливо правильно керувати залежностями між мікросервісами, щоб уникнути зайвої складності і зберігати стабільність системи.

Bottom-Up підхід є корисним для організацій, які шукають швидкі результати і можливість експериментувати з новими функціями, однак він вимагає уважного управління залежностями і інтеграцією, щоб досягти стабільної і ефективної мікросервісної архітектури.

Hybrid підхід(Рис 1.4): поєднує в собі елементи як Top-Down, так і Bottom-Up підходів. Його основна ідея полягає в тому, щоб використовувати переваги обох підходів для досягнення оптимальних результатів при переході від монолітної до мікросервісної архітектури. Основними етапами цього підходу є: аналіз бізнес-функцій і вимог: Починаючи з високорівневого аналізу бізнес-функцій і вимог до системи, визначаються ключові функціональні області і компоненти, які потрібно реалізувати; розробка початкових мікросервісів: На цьому етапі розробляються початкові мікросервіси, які вже відповідають за окремі функціональні області чи бізнес-процеси. Ці мікросервіси можуть створюватися в рамках Bottom-Up підходу; визначення мікросервісних границь: На основі аналізу бізнес-функцій і початкових мікросервісів визначаються мікросервісні границі і інтерфейси між ними. Цей етап також включає визначення механізмів комунікації між сервісами; ітеративна розробка та інтеграція: Після визначення мікросервісних границь розробка продовжується ітеративно, додаванням нових мікросервісів та їх інтеграцією з існуючими сервісами; тестування і впровадження: Після розробки і інтеграції мікросервісів проводяться комплексне тестування для визначення коректності їх роботи в складі системи. Після успішного тестування мікросервіси впроваджуються в експлуатацію; моніторинг і оптимізація: Після впровадження необхідно налаштувати моніторинг сервісів та проводити їх оптимізацію для забезпечення високої доступності та продуктивності.

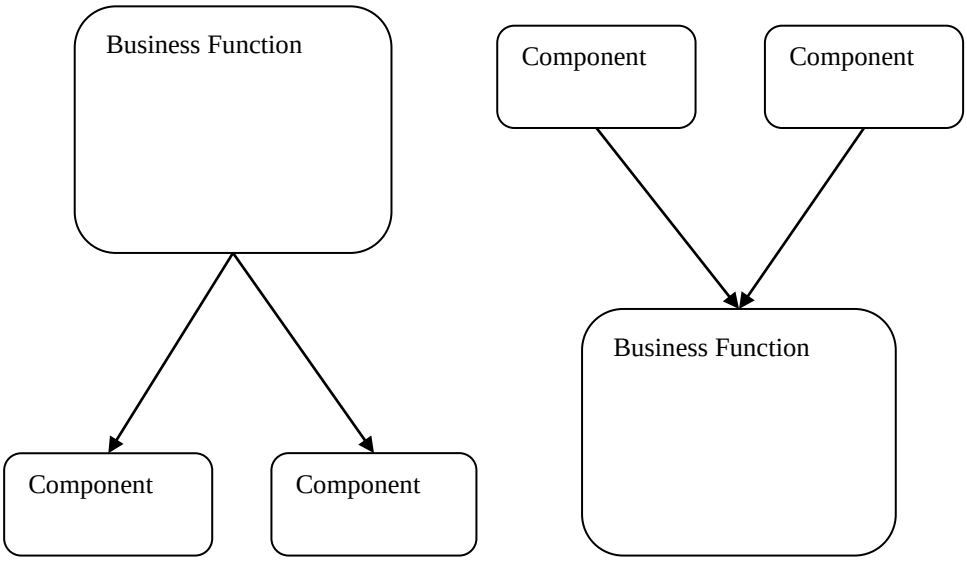

Рис 1.4 Hybrid підхід

До переваг цього підходу можна віднести:

- Гнучкість і адаптивність: Гібридний підхід дозволяє використовувати найкращі аспекти як Top-Down, так і Bottom-Up підходів, залежно від конкретних потреб проекту.
- Ефективність розробки: Розробка починається з швидкої ітерації Bottom-Up, що дозволяє швидко отримувати результати, а потім переходить до більш структурованого визначення мікросервісних границь у вигляді Top-Down підходу.

Недоліками є:

- Складність керування залежностями: Під час інтеграції різних елементів, що були розроблені різними методами, можуть виникати проблеми з керуванням залежностями і сумісністю.
- Потреба в збалансованості: Для успішної реалізації гібридного підходу потрібно правильно збалансувати використання Top-Down і Bottom-Up методів, щоб уникнути перекриття чи недоліків.

Гібридний підхід до розробки мікросервісної архітектури є потужним інструментом для організацій, які шукають компроміс між швидкістю розробки і структурованістю дизайну. Він дозволяє ефективно використовувати переваги обох підходів для досягнення успішного переходу до мікросервісної архітектури.

2. Автоматизований підхід переходу на мікросервісну архітектуру на основі кластеризації графів

Перехід від монолітної до мікросервісної архітектури можна автоматизувати за допомогою підходу, який використовує кластеризацію графів і комбінаторну оптимізацію. Цей підхід зосереджений на аналізі залежностей у монолітній системі та визначенні оптимальних меж для мікросервісів.

Розглянемо покроковий перехід від моноліту до мікросервісів з використанням кластеризації графів.

2.1 Статичний аналіз коду (Static Code Analysis)

Статитчний аналіз- аналіз вихідного коду монолітного додатка для виділення класів, методів і їх взаємозв'язків. Цей аналіз може включати визначення залежностей між класами, методами, а також взаємодії з базою даних. В ході аналізу можуть бути використані анотації із фреймворку, такого як SpringFramework `@Controller`, `@Service`, `@Repository`, `@Entity`. Ці анотації вказують на технічні ролі класів (наприклад, контролери, сервіси, репозиторії, сутності), що є цільовими для виокремлення в окремі мікросервіси.

Далі анотації можуть бути використані для визначення границь між різними мікросервісами. Наприклад, класи з анотацією `@Controller` можуть стати окремими сервісами, які надають API для зовнішніх клієнтів. Класи з анотацією `@Service` можуть представляти бізнес-логіку, яку можна розділити на окремі мікросервіси з різними функціональними обов'язками. Класи, які мають анотації `@Entity` або відповідають доменним об'єктам, таким як моделі бази даних, можуть бути відокремлені в окремі мікросервіси, які управляють певними частинами доменної моделі.

2.2 Побудова графового представлення (GraphRepresentation)

Побудова графового представлення - створення графової структуру, де вузлами будуть представлені класи і методи програми, а ребра відображатимуть взаємозв'язки між ними, такі як виклики методів, використання різних класів тощо. Це дає уявлення про структурні зв'язки в монолітному додатку [3,4].

Вузол графа може бути створений для: кожного класу рівня логіки та репозиторію та для кожного класу сутності. Зв’язки між класами мають бути поміщені на граф у вигляді спрямованих дуг. Також слід визначити типи дуг і їхні ваги. Ваги визначають релевантність зв’язків між вузлами: чим вища вага дуги, тим більша ймовірність, що сутності в її кінцевих точках слід помістити в той самий мікросервіс [5,6]. Після аналізу ваги призначаються відповідно до типу зв’язку.

repository→persistence_entity (w=5),
service → repository (w=4)
service → service (w=3)
service → entity (w=2)
entity → entity (w=1)

Зв’язок repository→persistence_entity має найвищу вагу, оскільки persistence_entity є моделями бази даних, для зв’язків service → repository та service → service встановлена нижчавага і вага зв’язків service → entity і entity → entity нижча, щоб сутності домену було розподілено між різними мікросервісами.

2.3 Кластеризація графа

Метою кластеризації є знаходження груп вузлів (класів, методів), які мають високу внутрішню схожість (висока когезія) і низьку зовнішню залежність (низька зв'язаність). Ця задача математично формулюється через оптимізаційні критерії або евристики, залежно від обраного алгоритму [7].

2.3.1 Алгоритми кластеризації графів

Спектральна кластеризація використовує спектральні властивості (власні значення і вектори) матриці Лапласіана графа для розбиття графа [10].

Алгоритм:

1. Побудувати матрицю Лапласіана L=D−A, де D — діагональна матриця ступенів, A — матриця суміжності.
2. Обчислити k найменших власних векторів Лапласіана, де k — кількість кластерів.
3. Використовувати k власних векторів для кластеризації вершин за допомогою алгоритму k-середніх (k-means).

Формула:L=D−A

Метод Лувена максимізує модулярність графа, що вимірює густину ребер всередині кластерів порівняно з очікуваною густиною у випадковому графі [8].

Алгоритм:

1. Початково кожен вузол розглядається як окремий кластер.
2. Повторювано переміщуються вузли між кластерами для збільшення модулярності.

3. Злиття вузлів у супер-вузли та повторення процесу на новому рівні абстракції до досягнення оптимальної модулярності.

Формула Модулярності:

$$Q = \frac{1}{2 \cdot m} \sum_{ij} \left[A_{ij} - \frac{k_i \cdot k_j}{2 \cdot m} \right] \cdot \delta(C_i, C_j) \quad (2.1)$$

де:

A_{ij} — елемент матриці суміжності,

k_i, k_j — ступені вузлів і та j,

m — загальна кількість ребер,

$\delta(C_i, C_j)$ — функція Кронекера.

Алгоритм Гірвана-Ньюмана знаходить спільноти в графі шляхом послідовного видалення ребер з високою центральністю між кластерами [9].

Алгоритм:

1. Обчислити Центральність Бетвіна для всіх ребер.
2. Видалити ребро з найбільшою центральністю.
3. Повторювати кроки 1 і 2 до досягнення бажаного числа кластерів.

Формула Центральності Бетвіна:

$$C_b(e) = \sum_{allpairs(s,t)} \frac{\sigma_{st}(e)}{\sigma_{st}}, \quad (2.2)$$

де:

σ_{st}- загальна кількість найкоротших шляхів між s і t,

$\sigma_{st}(e)$ – кількість шляхів, що проходять через ребро e

Метод поширення міток призначає мітки вузлам графа на основі міток їхніх сусідів, припускаючи, що сусідні вузли схильні належати до одного і того ж кластера [11].

Алгоритм:

1. Початково кожному вузлу присвоюється унікальна мітка.
2. Повторювано оновлюються мітки вузлів відповідно до найпоширенішої мітки серед сусідів.
3. Процес повторюється до збіжності (тобто мітки перестають змінюватися).

Кластеризація графа допомагає автоматично розбивати монолітні додатки на логічні групи компонентів, які можуть бути виділені в окремі мікросервіси. Використання алгоритмів, таких як Louvain, дозволяє знайти оптимальні кластери, зменшуючи міжкластерні взаємодії і збільшуючи внутрішньокластерну зв'язаність.

3. Оптимізація кластерів

Мета оптимізації - мінімізувати міжкластерні взаємодії та покращити внутрішньокластерну зв'язаність, що дозволяє досягти більшої автономності та ефективності кожного мікросервісу.

Приклад оптимізації кластеризації

1. Створення початкових кластерів

Розглянемо граф з вузлами та ребрами:

- Вузли: `A`, `B`, `C`, `D`, `E`
- Залежності: `A -> B`, `A -> C`, `B -> D`, `C -> D`, `C -> E`

ISSN 2219-9365

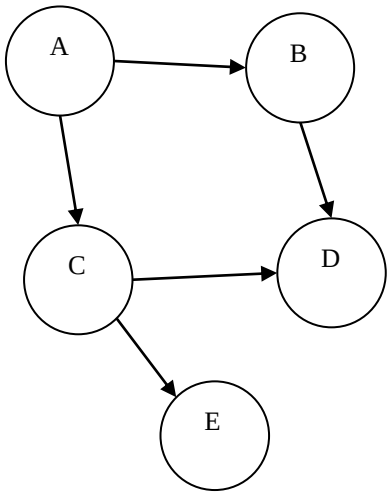

Рис.3.1. Граф з вузлами та ребрами

Для простоти припустимо, що всі залежності мають однакову вагу.

```
import networkx as nx
import community as community_louvain

# Створення графа
G = nx.DiGraph()
edges = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'), ('C', 'E')]
G.add_edges_from(edges)

# Виконання кластеризації з використанням алгоритму Louvain
partition = community_louvain.best_partition(G.to_undirected())

# Виведення початкових кластерів
for node, cluster in partition.items():
 print(f"Node {node} is in cluster {cluster}")
```

Результат роботи програми:

```
Node A is in cluster 0
Node B is in cluster 0
Node C is in cluster 1
Node D is in cluster 1
Node E is in cluster 1
```

3.1. Формулювання оптимізаційної задачі

Мета - мінімізувати кількість міжкластерних залежностей.

Змінні: x_{ij} – двійкова змінна, яка приймає значення 1, якщо вузол і призначено кластеру j, і 0 в іншому випадку.

Цільова функція:

$$Minimize \sum_{(i,k)\in E} \sum_{j\neq l} x_{ij} \cdot x_{kl}, \tag{3.1}$$

Обмеження:

Кожен вузол призначається тільки одному кластеру:

$$\sum_j x_{ij} = 1 \forall i, \tag{3.2}$$

Кількість кластерів обмежена:

$$\sum_i x_{ij} \leq max \quad \forall j, \tag{3.3}$$

3.2. Розв'язання задачі з використанням ILP

Використовуємо бібліотеку PuLP для розв'язання ILP:

```
!pip install pulp
from pulp import LpProblem, LpMinimize, LpVariable, lpSum, LpBinary, value

# Вузли і ребра
nodes = ['A', 'B', 'C', 'D', 'E']
edges = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'), ('C', 'E')]

# Кластери
clusters = [0, 1]  # Припустимо, що ми хочемо 2 кластери

# Створення ILP задачі
problem = LpProblem("ClusterOptimization", LpMinimize)

# Змінні
x = LpVariable.dicts("assignment", [(i, j) for i in nodes for j in clusters], cat=LpBinary)

# Допоміжні змінні для мінімізації міжкластерних залежностей
y = LpVariable.dicts("inter_cluster", [(i, k) for i, k in edges], cat=LpBinary)

# Обмеження: кожен вузол в одному кластері
for i in nodes:
    problem += lpSum(x[(i, j)] for j in clusters) == 1

# Обмеження: небільше max_nodes_per_cluster вузлівукластері (опційно)
max_nodes_per_cluster = 3
for j in clusters:
    problem += lpSum(x[(i, j)] for i in nodes) <= max_nodes_per_cluster

# Обмеження для допоміжних змінних
for (i, k) in edges:
    for c1 in clusters:
        for c2 in clusters:
            if c1 != c2:
                problem += y[(i, k)] >= x[(i, c1)] + x[(k, c2)] - 1

# Цільова функція
problem += lpSum(y[(i, k)] for (i, k) in edges)

# Розв'язання задачі
problem.solve()

# Виведення результатів
cluster_assignment = {i: j for i in nodes for j in clusters if x[(i, j)].varValue == 1}
print("Cluster assignment:")
for node, cluster in cluster_assignment.items():
    print(f"Node {node} is in cluster {cluster}")

print("Objective value:", value(problem.objective))
```

Результат роботи програми:

```
Cluster assignment:
Node A is in cluster 0
Node B is in cluster 1
```

Node C is in cluster 0
Node D is in cluster 1
Node E is in cluster 0
Objective value: 2.0

ВИСНОВКИ З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

Перехід від монолітної архітектури до мікросервісної з використанням методів кластеризації графів є ефективним підходом, який дозволяє розділити систему на логічно зв'язані компоненти.

Статичний аналіз є першим етапом здійснення переходу від монолітної до мікросервісної архітектури. Він дозволяє виділити компоненти (вузли) та встановити зв’язки між ними, а також вагу цих зв’язків, в залежності від типу компонентів, які він з’єднує.

Після побудови зв’язного графа на основі статичного аналізу, наступним важливим кроком є кластеризація графів. Кластеризація графів забезпечує автоматизований підхід до розбиття моноліту, що дозволяє виявити природні межі між компонентами. Використання алгоритмів кластеризації, таких як Louvain, Spectral Clustering або Girvan-Newman, спрощує і прискорює процес ідентифікації модулів.

Наступним важливим етапом у процесі перетворення монолітної архітектури в мікросервісну є оптимізація отриманих кластерів. Виконавши практичну задачу кластеризаціїї графа з 5 вершинами, було отримано 2 кластери, перший включав 2 вузли, а другий – 3. Але після виконання кластерної оптимізаціїї було отримано інший результат – одна з вершин графу була переміщена з другого до першого кластеру. Відбулося це завдяки зменшенню міжкластерних залежностей, що забезпечило ефективний розподіл компонентів і в кінцевому результаті підвищило продуктивність і масштабованість системи.

До переваг підходу можна віднести:

1. Аналіз графу залежностей дозволяє виявити приховані зв'язки між компонентами, які можуть бути неочевидними в традиційному аналізі. Це допомагає більш точно визначити межі мікросервісів та забезпечити їх ефективну взаємодію.
2. Кластеризація графів допомагає розбити систему на менші, незалежні модулі з високою внутрішньою зв'язаністю і мінімальними зовнішніми залежностями.
3. Автоматизоване визначення кластерів зменшує складність ручного аналізу і розбиття моноліту, забезпечуючи більш об'єктивний і точний розподіл.

Виклики та обмеження:

1. Якість даних:
- Успішність кластеризації графів залежить від якості даних про залежності між компонентами. Неповні або неточні дані можуть призвести до некоректного розбиття.
- Необхідно забезпечити точний збір і аналіз метрик залежностей.
2. Складність налаштування:
- Вибір відповідного алгоритму кластеризації і його параметрів може бути складним завданням, яке вимагає спеціальних знань і досвіду.
- Потрібно експериментувати з різними підходами і параметрами, щоб досягти оптимального результату.
3. Інтеграція з існуючою системою:
- Процес переходу від моноліту до мікросервісів може вимагати значних змін в існуючій інфраструктурі і організаційних процесах.
- Важливо забезпечити безперервність роботи системи під час переходу та мінімізувати вплив на користувачів.

Розбиття монолітної архітектури на мікросервіси за допомогою кластеризації графів є потужним і ефективним підходом, який дозволяє автоматизувати процес ідентифікації модулів і зменшити складність ручного аналізу. Це допомагає підвищити модульність системи, знизити ризик помилок і забезпечити більш ефективний перехід до мікросервісної архітектури.

Незважаючи на виклики, пов'язані зі складністю аналізу та розподілу, правильна стратегія та інструменти дозволяють досягти значних переваг і успішно впровадити мікросервісну архітектуру.

Література

1. Newman S. Building Microservices: Designing Fine-Grained Systems. O’Reilly Media, Inc.; 2015.
2. S. Hassan, R. Bahsoon, and R. Kazman, Microservice transition and its granularity problem: A systematic mapping study, Software: Practice and Experience, vol. 50, no. 9, pp. 1651–1681, 2020.
3. Trabelsi, M. Abdellatif, A. Abubaker, N. Moha, S. Mosser, S. Ebrahimi-Kahou, and Y.-G. Gue´he´neuc, From legacy to microser- vices: A type-based approach for microservices identification using machine learning and semantic analysis, Journal of Software: Evolution and Process, 2022.
4. M. Abdellatif, R. Tighilt, N. Moha, H. Mili, G. El Boussaidi, J. Privat, Y.-G. Gue´he´neuc, A type-sensitive service identification approach for legacy-to-soa migration, International Conference on Service- Oriented Computing, pp. 476–491, Springer, 2020.

5. Selmadji, A.-D. Seriai, H. L. Bouziane, R. O. Mahamane, P. Zaragoza, andC. Dony, From monolithic architecture style to microservice one based on a semi-automatic approach, ICSA. IEEE, 2020, pp. 157–168
6. Levcovitz A, Terra R, Valente MT. Towards a technique for extracting microservices from monolithic enterprise systems, 2016.
7. M. E. Newman, Modularity and community structure in networks, Proceedings of the national academy of sciences, vol. 103, no. 23, pp. 8577–8582, 2006.
8. Scott Emmons ,Stephen Kobourov,Mike Gallant,Katy Börner Analysis of Network Clustering Algorithms and Cluster Quality Metrics at Scale URL:https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0159161
9. Lj. Despalatović, T. Vojković, D.Vukičević Community structure in networks: improving the Girvan-Newman algorithm
10. Scott White, Padhraic Smyth A Spectral Clustering Approach To Finding Communities in Graphs
11. Kamal Berahmanda, Sogol Haghanib, Mehrdad Rostamic, Yuefeng Lia A new attributed graph clustering by using label propagation in complex networks, Journal of King Saud University- Computer and Information Sciences, vol 34, May 2022, Pages 1869-1883