

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Динамічний аналіз поганих практик та аномалій в
розподіленій системі»**

Виконав студент групи ІПЗм-22-1
ТАЛАВЕР Олег Володимирович

Керівник роботи:
МОРОЗОВ Андрій Васильович

Рецензент:
ЧИЖМОТРЯ Олексій Володимирович

Житомир – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«18» вересня 2023 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ТАЛАВЕР Олег Володимирович**

Керівник роботи: **МОРОЗОВ Андрій Васильович**

Тема роботи: **«Динамічний аналіз поганих практик та аномалій в розподіленій системі»,**

затверджена Наказом закладу вищої освіти від **«18» вересня 2023 р. № 479/с**

Вихідні дані для роботи: предметом дослідження є аналіз, обробка та візуалізація зібраної в системі телеметрії задля наглядного відображення з метою знаходження та вчасного реагування на проблеми системи. Об'єктом дослідження є процес аналізу розподіленої інформаційної системи.

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Морозов А.В.	22.09.2023р.	22.09.2023р.
2	Морозов А.В.	22.09.2023р.	22.09.2023р.
3	Морозов А.В.	22.09.2023р.	22.09.2023р.

РЕФЕРАТ

Індивідуальне завдання кваліфікаційної магістерської роботи полягає в науковому дослідженні та обґрунтуванні можливості застосування телеметрії з метою аналізу розподіленої інформаційної системи, виявлення та швидкого реагування на шкідливі архітектурні практики та аномальні події в системі.

Кваліфікаційна магістерська робота на тему «Динамічний аналіз поганих практик та аномалій в розподіленій системі» складається з теоретичного та програмного комплексу системи та пояснювальної записки. Пояснювальна записка до кваліфікаційної магістерської роботи складається зі вступу, чотирьох розділів, списку літератури з 37 найменувань, 42 рисунків, 1 таблиці та 6 додатків. Повний обсяг роботи становить 81 сторінку.

КЛЮЧОВІ СЛОВА: РОЗПОДІЛЕНА СИСТЕМ, ДИНАМІЧНИЙ АНАЛІЗ, ПОГАНІ АРХІТЕКТУРНІ ПРАКТИКИ, ТЕЛЕМЕТРІЯ, АНОМАЛІЇ, OPEN TELEMETRY, ТЕОРІЯ ГРАФІВ, СТАТИСТИЧНИЙ АНАЛІЗ.

ABSTRACT

The individual task of the qualifying master's thesis was a scientific study and justification of the possibility of using telemetry for the analysis of a distributed system, detection and rapid response to architectural bad practices and anomalous events in the system.

The qualifying master's work on the topic "Dynamic analysis of bad practices and anomalies in a distributed system" consists of a theoretical and software complex of the system and an explanatory note. The explanatory note to the qualifying master's thesis consists of an introduction, four chapters, a bibliography of 37 titles, 42 figures, 1 tables and 6 appendices. The full volume of the work is 81 pages.

KEY WORDS: DISTRIBUTED SYSTEMS, DYNAMIC ANALYSIS, BAD ARCHITECTURAL PRACTICES, TELEMETRY, ANOMALIES, OPEN TELEMETRY, GRAPH THEORY, STATISTICAL ANALYSIS.

ЗМІСТ

ЗМІСТ	5
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	7
ВСТУП	8
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ.....	11
1.1. Огляд ПРОБЛЕМАТИКИ МОНИТОРИНГУ РОЗПОДІЛЕНИХ СИСТЕМ	11
1.2. Дослідження ТА РОЗВИТОК СИСТЕМ МОНИТОРИНГУ	12
1.3. Огляд ЗАСТОСУНКІВ МОНИТОРИНГУ ТА АНАЛІЗУ ПРОДУКТИВНОСТІ СИСТЕМИ	14
1.3.1. Jaeger.....	14
1.3.2. Signoz.....	16
1.3.3. Aspecto.....	18
1.3.4. ServiceNow Cloud Observability	19
1.3.5. Honeycomb	21
1.3.6. New Relic.....	23
1.4. АНАЛІЗ МОЖЛИВОГО ФУНКЦІОНАЛУ ДЛЯ ПОДАЛЬШОГО ДОСЛІДЖЕННЯ ...	26
РОЗДІЛ 2. ВИКОРИСТАННЯ ТЕЛЕМЕТРІЇ ДЛЯ ДИНАМІЧНОГО АНАЛІЗУ СИСТЕМИ	30
2.1. Визначення ДАНИХ ТА РОЗРОБКА СХОВИЩА ДАНИХ ДЛЯ АНАЛІЗУ ПОГАНИХ ПРАКТИК	30
2.2. МЕТОДИ І АЛГОРИТМИ ІНТЕЛЕКТУАЛЬНОГО АНАЛІЗУ ДАНИХ ДЛЯ РІШЕННЯ ПРОБЛЕМИ ЗНАХОДЖЕННЯ АНОМАЛІЙ	34
РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ АНАЛІЗУ СИСТЕМИ.....	37
3.1. РОЗРОБКА OPEN TELEMETRY COLLECTOR КОМПОНЕНТА ДЛЯ ТРАНСФОРМАЦІЇ ТА ВИВАНТАЖЕННЯ ДАНИХ.....	37
3.2. Визначення АНОМАЛЬНИХ ПОДІЙ АНАЛІЗУЮЧИ МЕТРИКИ ВИКОРИСТОВУЮЧИ RSA	37

РОЗДІЛ 4. АНАЛІЗ РЕЗУЛЬТАТІВ, ПІДСУМКОВІ ГРАФІКИ ТА ТАБЛИЦІ.	46
4.1. Візуалізація графу сервісів за допомогою засобів Neo4J	46
4.2. Аналіз часового проміжку на присутність аномалій	48
ВИСНОВКИ.....	50
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	53

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

MSA – Microservice architecture

RPC – Remote Procedure Call

DBMS - DataBase Management System

ВСТУП

У сучасному світі розробки програмного забезпечення тема впровадження розподілених рішень стала досить поширеною, через переваги застосування у компаніях з великою кількістю розробників. Традиційний підхід монолітної розробки додатків має очевидні недоліки, коли йдеться про велику кількість розробників, продуктивність, масштабованість, розширюваність. Це призвело до появи нових підходів розробки, коли застосунок розділяють на незалежні функціональні частини, кожна з яких вміщує логіку частини варіантів використання. Дані модулі, що мають назву мікросервіси, формують основу стилю розробки, який швидко набуває поширення. Кожен процес працює відокремлено від інших мережею та спілкується за допомогою передачі повідомлень. Слабозв'язність компонентів дозволяє мікросервісам бути розробленими паралельно та масштабованими за потреби.

Негативна сторона полягає в тому, що при розробці таких систем, особливо коли мова йде про велику кількість розробників, глобальні проблеми проєктування можуть бути неочевидними та призвести до сповільнення процесу розробки або навіть до проблем з локацією помилок чи погіршенням продуктивності системи. Окрім того реагування на деградацію працездатності системи поскладнюється через розподілену природу архітектури, при цьому ручне налаштування правил для повідомлення про проблемні ситуації може займати багато часу і все одно не покривати усі місця, тому автоматичне виявлення можливих аномалій системи задля звернення уваги інженерів (Software Reliability Engineer) має значну цінність. Тому додатки що здатні динамічно аналізувати систему на предмет вищезгаданих проблем має великий потенціал. Наразі тема використання телеметрії для аналізу системи розвивається та отримує поширення, тому подальші дослідження мають цінність.

Метою роботи є теоретичне та практичне обґрунтування можливості застосування телеметрії з метою аналізу розподіленої інформаційної системи,

виявлення та швидкого реагування на шкідливі архітектурні практики та аномальні події.

Встановлена мета обумовлює наступні завдання:

- огляд наявних підходів до аналізу системи;
- огляд наявних популярних систем моніторингу та аналізу продуктивності: Jaeger, Signoz, Aspecto, ServiceNow Cloud Observability, Honeycomb, NewRelic;
- виокремлення сильних та слабких сторін для подальшого дослідження або покращення існуючих варіантів використання;
- моделювання та побудова ETL процесу вивантаження телеметрії;
- аналіз отриманих даних з метою виокремлення шкідливих практик та аномалій.

Об'єктом дослідження є процес аналізу розподіленої інформаційної системи.

Предметом дослідження є аналіз, обробка та візуалізація зібраної в системі телеметрії задля наглядного відображення з метою знаходження та вчасного реагування на проблеми системи.

Методи дослідження: методи аналізу та візуалізації даних (графів та часових рядів), статистичні алгоритми узагальнення даних, результати теорії графів.

Наукова новизна отриманих результатів полягає в наступному:

- використано останній стандарт для роботи з телеметрією (OpenTelemetry) задля визначення поганих архітектурних практик та аномалій;
- розроблено компонент-вивантажувач для OpenTelemetry Collector що перетворює телеметрію в однонаправлений граф структури системи;
- визначення аномальних подій використовуючи статистичний аналіз системи багатьох змінних за допомогою PCA.

Практичне значення полягає в наступному:

- програмний додаток дозволяє проаналізувати роботу розподіленої системи використовуючи дані її роботи шляхом отримання візуалізації агрегованих та оброблених даних для кращого сприйняття;
- розроблений додаток допомагає побачити проблемні місця в архітектурі розподіленої системи;
- розроблений додаток вирішує проблему аналізу зібраних в системі метрик у великих обсягах для подальшого автоматичного визначення можливих аномалій.

Матеріал кваліфікаційної роботи базується на статті опублікованій в журналі «Journal of Edge Computing» 2023 року [1].

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1. Огляд проблематики моніторингу розподілених систем

В останні роки розподілені архітектури, такі як мікросервісна (MSA), отримали велику увагу та популярність завдяки можливостям, які відкриває архітектурний шаблон з точки зору оптимізації, диверсифікації технологічного стеку тощо [2]. Незважаючи на те, що цей підхід до розробки часто представляють такою котра є кращою за монолітний, його не можна розглядати як заміну чи сліпо слідувати йому, оскільки ці архітектурні стилі служать іншим цілям і створюють свої проблеми.

Порівняно з розподіленими системами (DS), моноліти розробляються набагато швидше, оскільки немає потреби думати про те, як взаємодіють різні компоненти, якими даними має володіти конкретна частина та як ділитися даними через мережу [1].

Відсутність фізичних кордонів між модулями в моноліті, хоча й дозволяє швидко впроваджувати нові функції, також призводить до втрати загальної структури, що ускладнює розширення та підтримку системи та призводить до так званого «Big ball of mud» [3].

З іншого боку, DS, якщо вони побудовані правильно, спрощують процес розробки, коли задіяно багато команд, зменшують складність змін або залежність команд одна від одної та прискорюють розробку. Інші переваги, як-от стійкість і легкість розуміння окремих компонентів, є вторинними, оскільки їх переважають додаткові проблеми, які виникають із впровадженням таких систем, не кажучи вже про те, що моноліт все ще можна побудувати таким чином аби він був масштабованим, стійким і структурованим. Додаткові складності, що приносить MSA [4], сповільнюють розробку такої системи, тоді як така загальна перевага, як технологічна неоднорідність, лише збільшує зусилля, необхідні для підтримки кодової бази [5, 6].

Отже, якщо брати до уваги існуючу необхідність певної компанії у використанні MSA, та її розуміння складності такого підходу розробки, виникає потреба у додаткових засобах для аналізу системи та реагування на проблеми. Розробка розподілених інформаційних систем вимагає більше зусиль, особливо коли мова йдеться про моніторинг усієї системи, знаходження проблемних місць [4], адже на відміну від моноліту, така система має безліч компонентів, які до того ж можуть мати структурні помилки закладені на етапі проєктування [3, 7]. Для глобальних проблем зазвичай проводиться аналіз інформаційної системи задля виявлення та швидкого усунення недоліків [8]. Існує безліч підходів, наприклад статичний аналіз кодової бази кожного з компонентів системи або ж аналіз журналу подій системи. Обидва варіанти є досить складними, адже вимагають налаштування під кожну окрему систему, технологію, мову програмування, однак статичний підхід на відміну від динамічного дозволяє аналізувати систему без попереднього запуску, що дозволяє виправити деякі локальні недоліки, але не проблеми системи загалом, по причині низької точності визначення поведінки системи під час її роботи [9]. Тому в даній роботі пропонується оглянути процес динамічного аналізу системи з використанням телеметрії.

1.2. Дослідження та розвиток систем моніторингу

Тема моніторингу та аналізу зібраних показників з системи під наглядом є далеко не новою. Однак як і у випадку вирішення будь-яких інших проблем, все починається з намагань окремих людей вирішити конкретну проблему, в даному випадку аналізу системи, а конкретніше розподіленої системи, аби бачити що відбувається, завчасно реагувати на можливі проблеми. У світі розподілених систем Google вважається піонером дослідження даної проблематики через публікацію 2010-го року під назвою «Dapper - a Large-Scale Distributed Systems Tracing Infrastructure» [10] яка спонукає появу перших систем відстежування та візуалізації запитів Jaeger [11] та Zipkin [12]. Однак ці застосунки вирішували одну проблему різними шляхами та вимагали інтеграцію та прив'язку до

конкретної з них без можливості простої зміни, тому з часом починається розробка стандарту OpenTracing [13], який надає шар для взаємодії між додатком та системами моніторингу задля збору та відстежування запитів. Даний стандарт не вирішував усієї проблеми, тому пізніше був розроблений інший, OpenCensus [14], який концентрувався на зборі метрик системи та журналів, однак також включав в себе альтернативну реалізацію збору телеметрії, що в кінцевому випадку створювало більше проблем, адже тепер розробникам потрібно було обрати між цими двома стандартами. По цій причині в 2019 обидва стандарти були об'єднані в OpenTelemetry [15] аби вирішувати наступні задачі:

- збір інформації про запити, метрики та журнали подій в одному місці;
- знаходження аномалій через графіки;
- знаходження причини та місця аномалій через огляд проблемних запитів.

Також на ринку на той момент вже сформувалося досить багато застосунків, які допомагали в аналізі системи, по цій причині одним із завдань було зберегти сумісність з ними, тому даний стандарт надає специфікацію де описується підходи до збору метрик, конвенції для опису певних процесів таких як взаємодія за допомогою протоколу HTTP, RPC [16]. Як результат, наразі OpenTelemetry є найактивнішим проєктом Cloud Native Computing Foundation [17].

Однією з головних сутностей що вводить даний стандарт є телеметрія - сукупність метрик, журналів подій (logs) та, найголовніше, відбитків запитів (traces) [18], що у даному випадку може бути використана задля побудови моделі системи у вигляді направленого графа [9] та пізніше використовується для аналізу і виявлення поганих практик та проблемних місць системи. Стандарт Open Telemetry відносно новий тому все ще тривають активні дослідження наявних можливостей, але основний напрямок використання - візуалізація запитів (рис.1.1) з можливістю пошуку проблемних місць, наприклад причини поганої продуктивності ресурсу чи першопричини некоректно працюючого бізнес процесу [18].

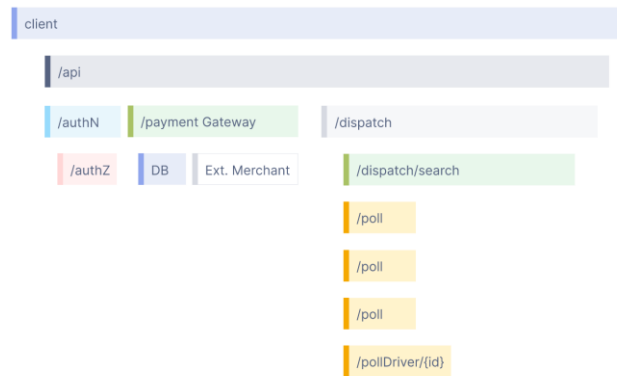


Рис 1.1. Візуалізація відбитків запиту

1.3. Огляд застосунків моніторингу та аналізу продуктивності системи

На ринку представлено безліч продуктів які працюють з телеметрією з метою спрощення аналізу працездатності системи, тому необхідно проаналізувати їх аби знайти місця для подальшого дослідження з можливістю покращення.

1.3.1. Jaeger

Одна з найпростіших систем з відкритим кодом, що в основному дає можливість візуалізувати запити, але нещодавно були додані можливості побудови графіків із зібраних метрик – що є стандартною функціональністю такого роду систем.

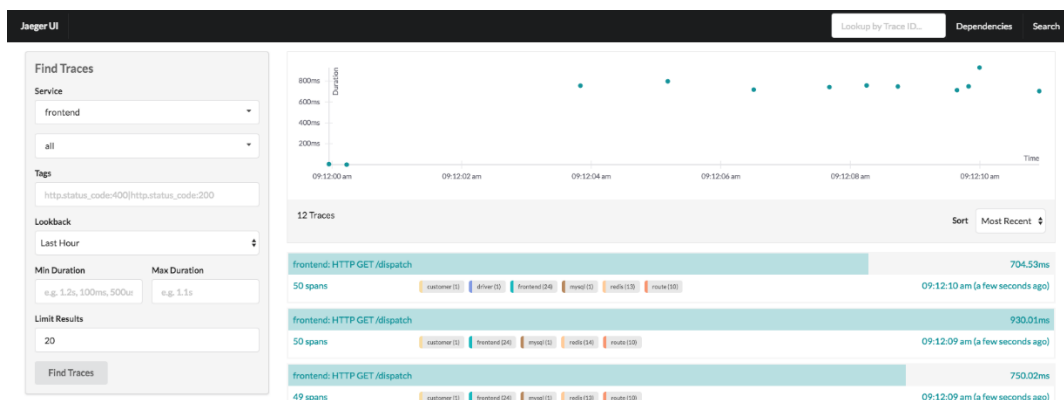


Рис 1.2. Перегляд запитів в системі за фільтром в Jaeger

Аби переглянути запити, потрібно спочатку налаштувати фільтр (рис.1.2), який вимагає вказання певного сервісу для перегляду його запитів. Тобто акцент

робиться на тому що ми вже знаємо який запити з якого сервісу нас цікавлять і це є проблемою, адже система може складатися з безлічі компонентів і ми не зможемо знаходити проблеми просто переглядаючи запити до кожного з сервісів. Фільтр має можливість фільтрації певних тегів, однак потрібно точно знати їх назву, адже при вводі підказки відсутні. Після застосування, нам відображається список запитів та графік. Графік відображає лише один набір показників: співвідношення кількості компонентів запитів а також час їх виконання. Список запитів відображає присутність помилки в запиті, компоненти, час з можливістю сортування.

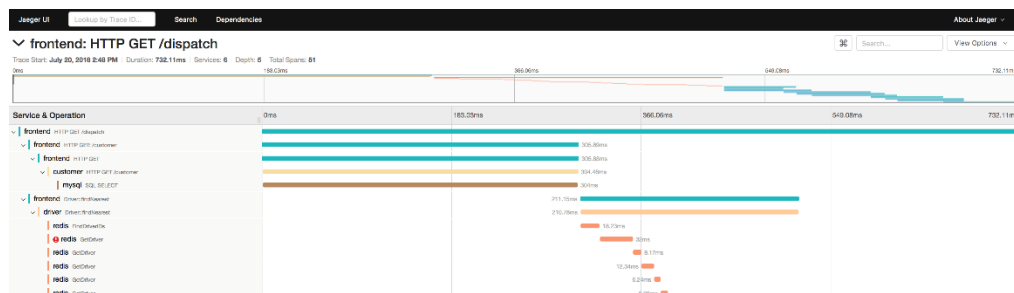


Рис 1.3. Перегляд запиту в Jaeger

При відкритті певного запиту (рис.1.3), відображається сторінка з часовим графіком, а також діаграма – водопад, яка наглядно показує послідовність та залежність під-запитів. Кожна ланка відображає назву компонента, назву інтервалу (операції), присутність помилки а також інших подій (насічки на інтервалах).

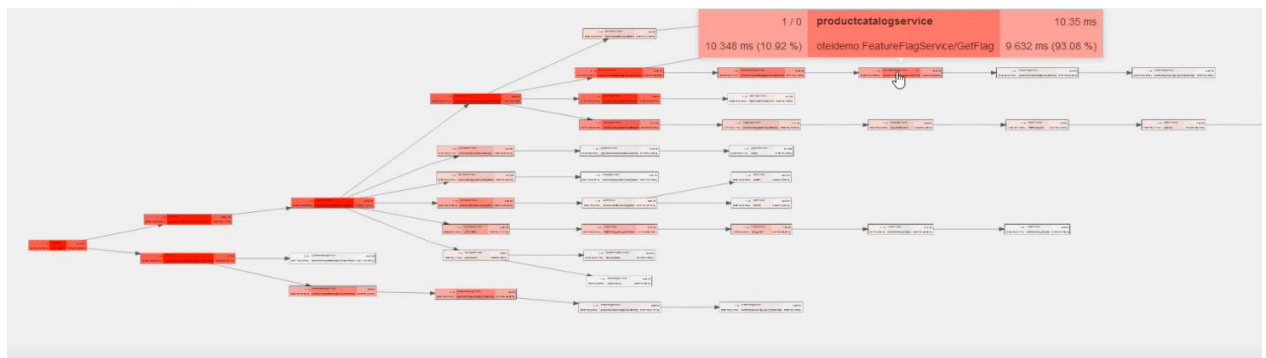


Рис 1.4. Перегляд графу викликів з візуалізацією відносного затраченого часу в Jaeger

Такий формат перегляду є стандартним, але також можливий перегляд в інших, наприклад ациклічний граф (рис.1.4) залежностей з візуалізацією часу затраченого на кожен виклик.

Висновок:

- досить проста система, з відкритим кодом що надає стандартні можливості;
- відсутність зручностей та підказок можливих значень при фільтрації запитів;
- можливості візуалізації запиту як графа, з можливістю перегляду затраченого часу.

1.3.2. Signoz

Дана система також має відкриту кодову базу, однак є відносно новою та має набагато більше можливостей.



Рис 1.5. Перегляд графіків, операцій сервісу та статистики для кожної в Signoz

Використання метрик та відбитків запитів поєднується аби відображатися графіки для певних сервісів а також статистику для певної групи запитів, які є частиною однієї операції (рис.1.5).

Сторінка перегляду запитів (рис.1.6) має можливість перегляду запитів з усієї системи, а також використання розвинутого фільтру, який представляє

фільтрацію по популярним загальноприйнятим полям, таким як наприклад «http.request.method», «http.response.status_code». Також на сторінці представлений налаштовуваний графік з можливістю групування.

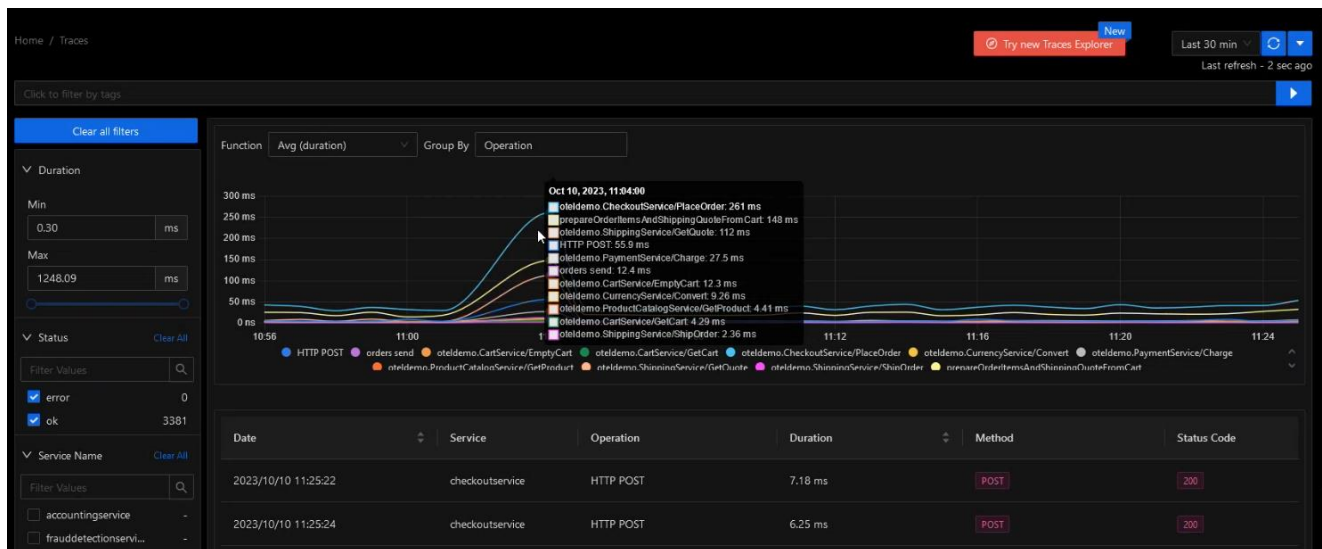


Рис 1.6. Перегляд запитів системи в Signoz

Сторінка перегляду запиту (рис.1.7) має лише один варіант перегляду графіку, інтервали мають схоже відображення, однак події не відображаються на прямих, що робить аналіз складнішим.

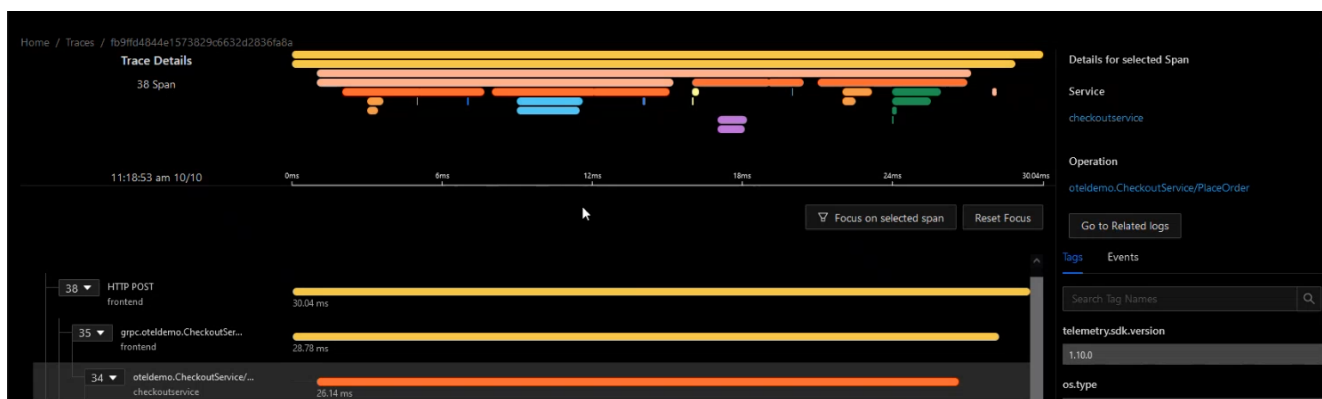


Рис 1.7. Перегляд запиту в Signoz

Також, як і інші інструменти моніторингу продуктивності, налічує функціонал для створення та перегляду сповіщень, виняткових ситуацій в системі (з групуванням однакових).

Наявна можливість перегляду графу компонентів системи залежностями між ними (рис.1.8). На графі також можна побачити кількість запитів на секунду, присутність певного відсотку помилок, затримку запитів.

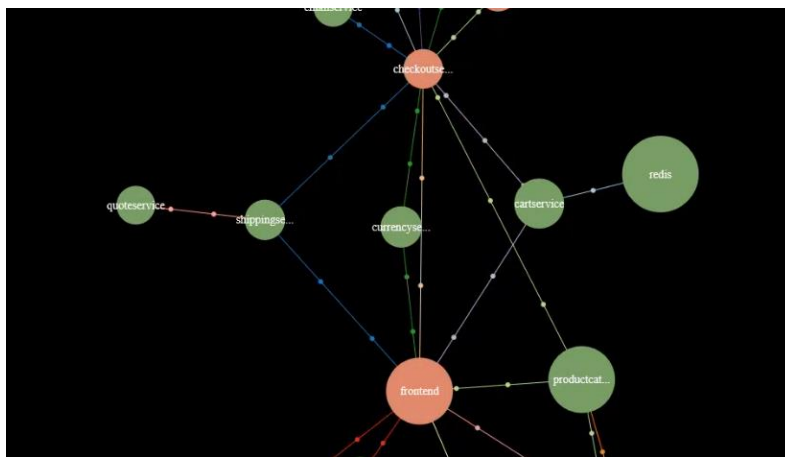


Рис 1.8. Візуалізація системи як графа Signoz

Висновок:

- гнучке налаштування графіків: для певних сервісів, для усього додатку, конкретних вхідних точок;
- сторінка перегляду запитів має продвинутий фільтр та налаштовуваний графік;
- посередній функціонал перегляду запитів без можливості змінити візуалізації;
- можливість візуалізації системи як графа з додатковими метриками – деякі з відображених компонентів не налаштовані на відправлення телеметрії і визначені за рахунок компонентів які викликають їх, наприклад сховища даних, які рідко інструментуються.

1.3.3. Aspecto

Достатньо проста платформа для візуалізації запитів та перегляду деяких метрик без можливості гнучкої конфігурації. Даний продукт, на відміну від попередніх, має закритий код та прогресивні плани в залежності від кількості даних що проходить через платформу.

Сторінка перегляду запитів (рис.1.9) має схожий до Signoz функціонал, однак дана платформа також додає візуалізацію запиту у вигляді графа, що значно спрощує перегляд та аналіз.

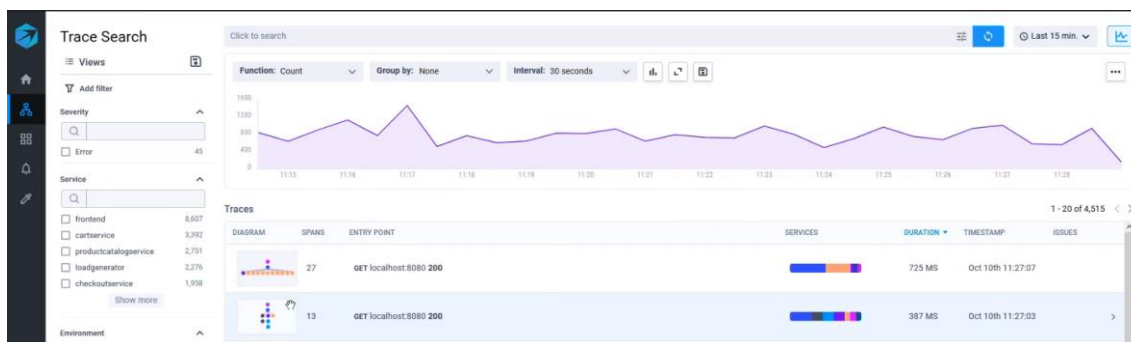


Рис 1.9. Сторінка перегляду запитів в Aspero

Графік запиту відображає (рис.1.10) тип компонента: сервіс, сховище даних, систему обміну повідомленнями.

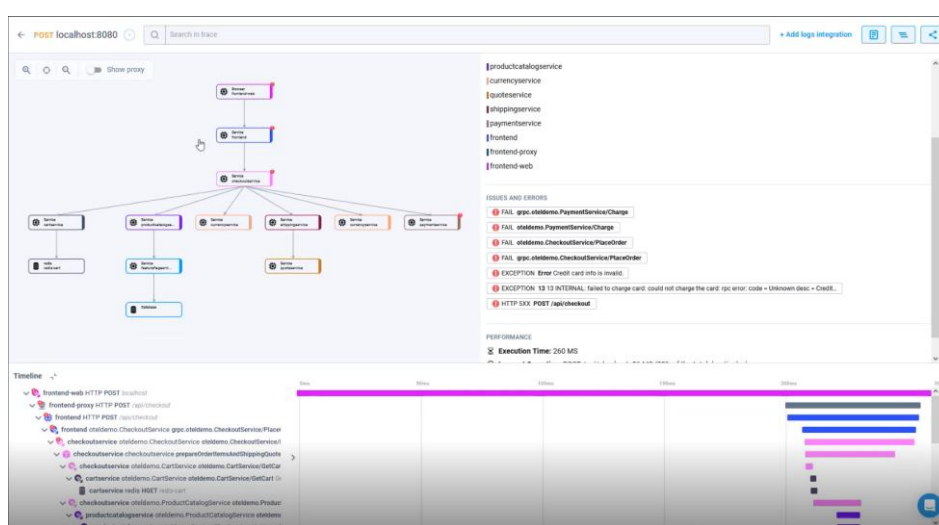


Рис 1.10. Сторінка перегляду запиту в Aspero

Висновки: проста платформа, яка має продвинуті можливості візуалізації запитів.

1.3.4. ServiceNow Cloud Observability

Наступна закрита платформа налічує багаточисельні можливості аналізу метрик та побудови графіків що мають різне представлення (рис.1.11). На графіках можна відображати одразу декілька агрегацій, метрик, переглядати запити що потрапляють в діапазон аналізу, чітко бачити запити з помилками.

Окрім того є можливість візуалізації взаємодії між сервісами, що підпадають під запит графіка (рис.1.12). З діаграми можна побачити яку частку затримки усього запиту вносить окремий сервіс, сервіси з помилками.

Рис 1.11. Побудова графіків в ServiceNow

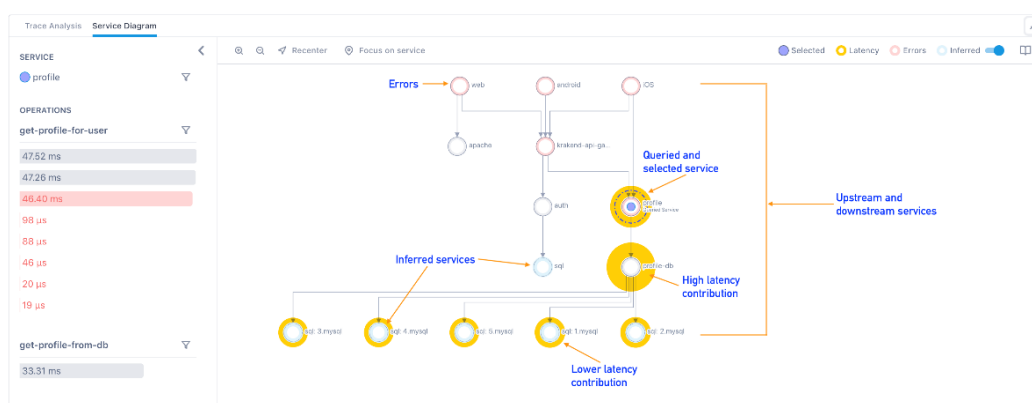


Рис 1.12. Діаграма взаємодії сервісів в ServiceNow

Також в додатку присутній механізм аналізу аномалій (рис.1.13) шляхом порівняння змін в атрибутів запитів двох проміжків, один з яких нормальний – базовий, а інший проблемний – з відхиленнями. Як результат виводиться список атрибутів відсортованих по коефіцієнту кореляції. При цьому для кожного з атрибутів можна одразу ж переглянути графік аби побачити чи дійсно запити з цим тегом сприяли відхиленню.

З поміж розглянутих можливостей, даний застосунок дає схожий до Signoz функціонал перегляду конкретного запиту, а також можливість налаштування сповіщень відповідно до побудованих графіків.

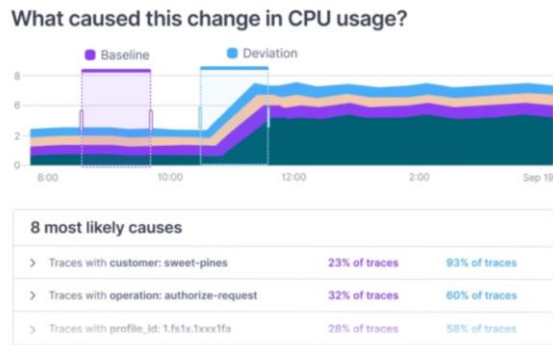


Рис 1.13. Аналіз кореляцій між зміною метрик в ServiceNow

Висновки:

- продвинутый функціонал побудови графіків;
- можливість побудови графів взаємодії (карти) сервісів;
- аналіз аномалій в системі, однак відсутній функціонал по знаходженню аномалій, адже сповіщення покривають не всі можливі проблеми, а постійно дивитися на графіки - недоцільно;
- дозволяє додавати в систему маркери для відображення події (наприклад вивантаження певної версії) на усіх графіках.

1.3.5. Honeycomb

Даний продукт концентрує свою увагу на знаходженні причин виникнення аномальних ситуацій, таких як наприклад певна кількість довгих запитів. Для цього на сторінці списку запитів є функціонал візуалізації запитів.

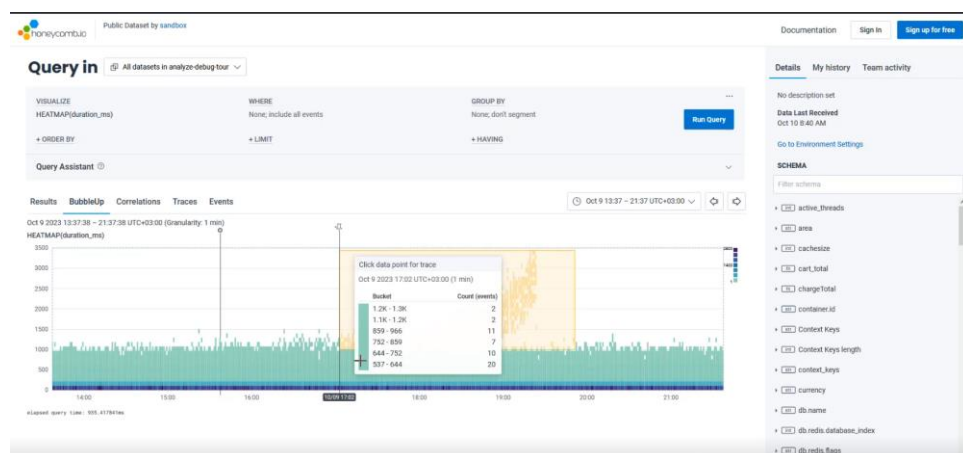


Рис 1.14. Виділення проблемних запитів для порівняння з іншими в Honeycomb

Щоб порівняти запити, потрібно обрати достатню кількість, після цього буде відображені атрибути та статистика відхилень для кожного (рис.1.15). Це дозволяє нам змінювати запит, виключаючи певні атрибути в запит та звужувати коло причин певної проблеми. Агрегації такого плану стають можливими завдяки використанню колонкових баз даних [19], які краще підходять для аналітичних завдань.

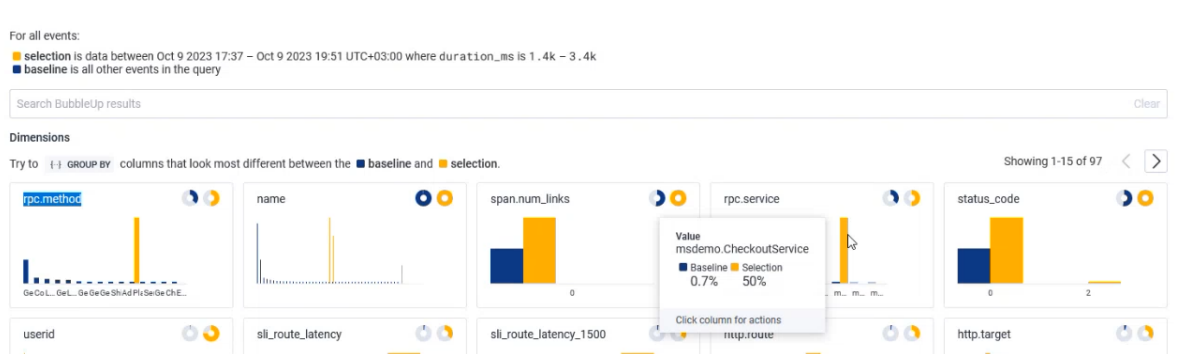


Рис 1.15. Графіки порівняння значень атрибутів між виділеними запитами та іншими в Honeycomb

Сторінка перегляду певного запиту виглядає звичайною. Також застосунок має модуль, який дозволяє переглядати карту системи (рис.1.16), на якій добре видно сервіси через які проходить більша кількість запитів, які сервіси залежать від багатьох інших.

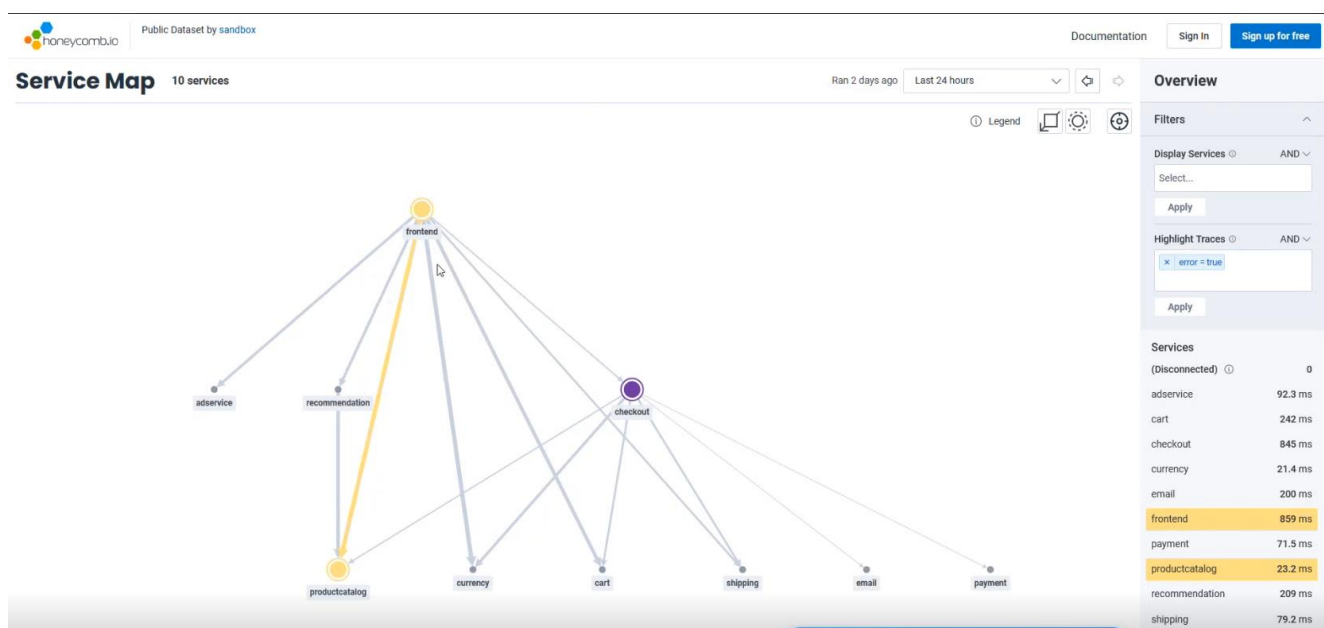


Рис 1.16. Перегляд карти системи в Honeycomb

Окрім вищезазначеного, присутній звичний функціонал налаштування сповіщень.

Висновки:

- продвинута карта сервісів системи;
- аналіз аномалій в системі, однак відсутній функціонал по знаходженню аномалій, адже сповіщення покривають не всі можливі проблеми, а постійно дивитися на графіки - недоцільно;
- дозволяє додавати в систему маркери для відображення події (наприклад вивантаження певної версії) на усіх графіках.

1.3.6. New Relic

Наразі найбільш розвинута та дорога система з тих що оглядалися, в якій окрім плати за використані ресурси, як в попередніх платформах, також потрібно платити за користувачів, відповідно до кількості доступних їм можливостей системи.

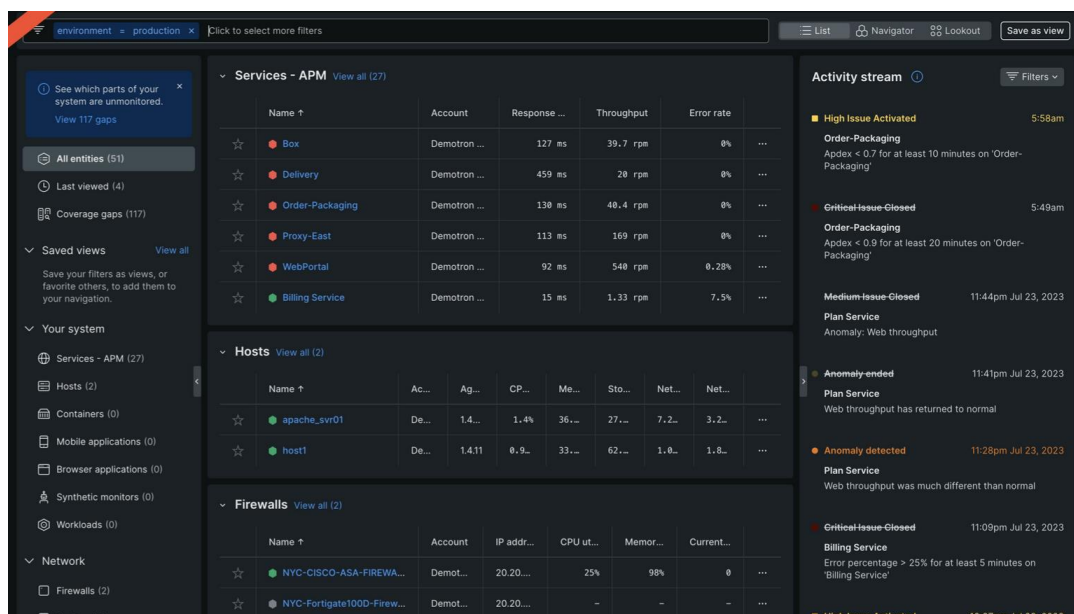


Рис 1.17. Сторінка сутностей в New Relic

Окрім цього потрібно відзначити, що присутній функціонал, як наприклад знаходження аномалій, не повністю підтримує стандарт OpenTelemetry, тому для цих можливостей потрібно використовувати надані компоненти від цієї

платформи аби збирати необхідні дані, що нівелює гнучкість та прив'язує до цієї платформи.

Сторінка усіх сутностей (рис.1.17) є ключовою та надає доступ до усіх ключових точок продукту. На цій сторінці можна одразу побачити список усіх сервісів, груп сервісів (workloads), сповіщення, а також статус та метрики усіх цих сутностей. При чому статус сервісу визначається як комбінація статусів усіх сповіщень для цього сервісу, а статус групи відповідно комбінації сервісів групи.

Наявний режим перегляду сервісів за якого вони відображаються як кола розміром відповідно до степені відхилення від нормального стану для певної ключової метрики (рис.1.18).

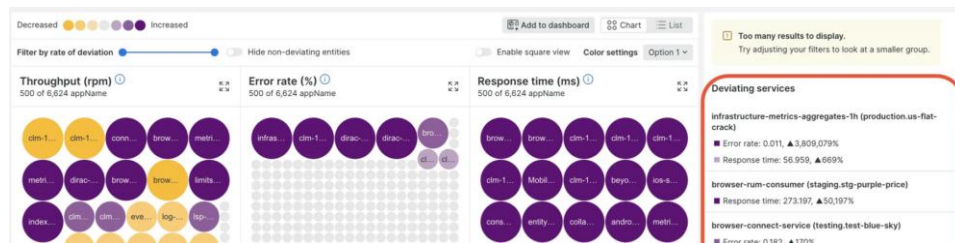


Рис 1.18. Режим перегляду відхилення сутностей від нормальних показників в New Relic

При натисканні на певний сервіс є можливість перегляду відхилень за певними атрибутами на певному проміжку часу (рис.1.19), схожим чином до вже розглянутого ServiceNow. Також є можливість перегляду схожих змін в ключових метриках сервісів які взаємодіють з даним, а ще відображення певних патернів поведінки системи в цей час.

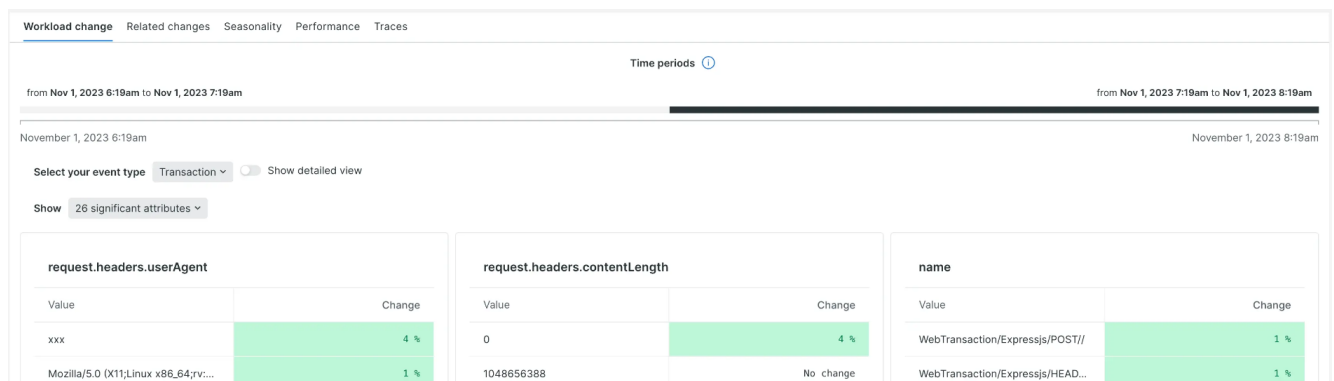


Рис 1.19. Сторінка перегляду відхилень атрибутів для певного сервісу в New Relic

На сторінці сервісу, можна переглянути підготовані графіки усіх ключових метрик, таких як відсоткова частка помилок, пропускна здатність (рис.1.20). Також відображається список найповільніших операцій в системі, а також статистику по усіх серверах де наразі розгорнутий даний сервіс.

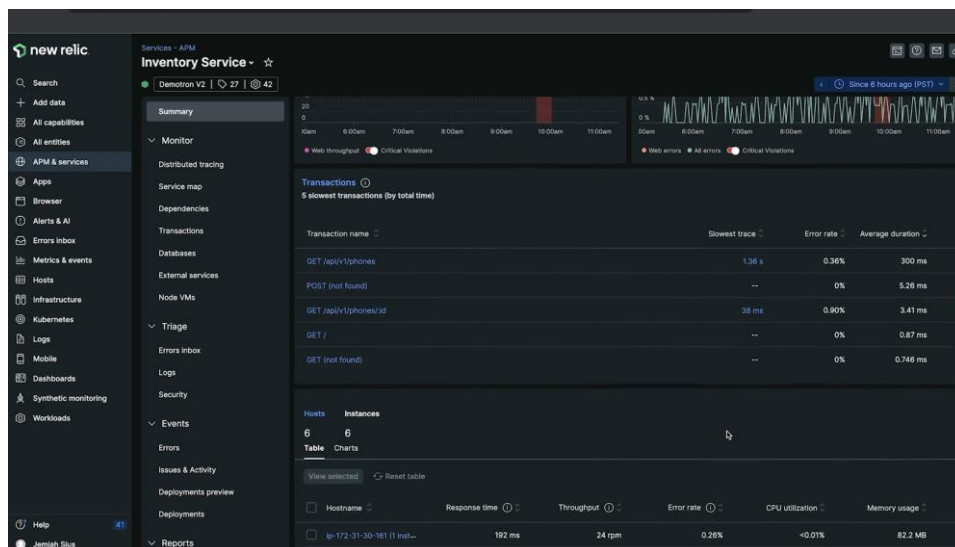


Рис 1.20. Сторінка перегляду сервісу в New Relic

Сторінка перегляду запиту відображає карту запиту через усі сервіси, а також помилки, аномалії, і корисну статистику відносно обраного інтервалу (рис.1.21).

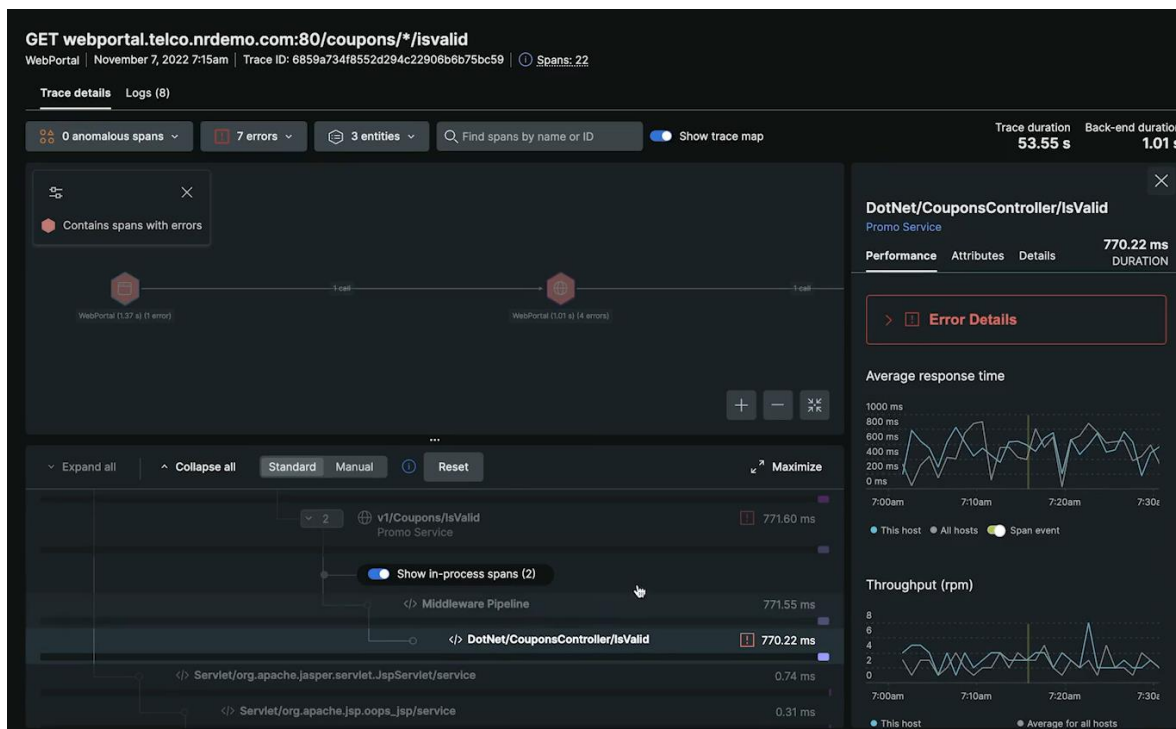


Рис 1.21. Сторінка перегляду запиту в New Relic

Присутній функціонал автоматичного виявлення аномалій для кожної з ключових метрик. Також є можливість налаштування виявлення аномалій вручну для власних метрик.

Як і у випадку з попередніми платформами, наявні можливості перегляду проблем системи, однак в даній до цього ще додаються події спрацювання сповіщень, аномалій а також можливість реагування на новостворені інциденти з можливістю створення помилок системи з різною пріоритетністю, контролювати виконання та додавання коментарів.

Висновки:

- інтегрованість інфраструктури та всіляких автоматичних перевірок, можливість перегляду метрик для хостів, сервісів що розгорнуті на них, зазначення статусів готовності сервісу, групи сервісів, відповідність Service Level Objectives (SLO);
- відображення сесій користувачів – впродовж користування додатком, користувач робить багато кліків, переходів між сторінками, провокує запити;
- можливість перегляду змін ключових метрик сервісу;
- автоматичне виявлення аномальних ситуацій ключових метрик сервісу;
- інтегрований функціонал по взаємодії з інцидентами: групування, коментування, визначення відповідальних за вирішення.

1.4. Аналіз можливого функціоналу для подальшого дослідження

Виходячи з розглянутих системи, ми отримали інформацію про більшість варіантів використання що є стандартними та ключовими – інноваційними для застосунків моніторингу продуктивності систем (табл.1.1).

Обов'язкові можливості:

- перегляд сервісів системи з ключовими метриками (відсоток помилок за останній час, проценти швидкості виконання запитів сервісу);

- можливість побудови сторінок з графіками системи (визначених запитами);
- перегляд проблем та виключень які зустрічаються в запитах системи;
- фільтрація та перегляд відбитків запитів з візуалізацією;
- створення сповіщень для реагування на якісь події визначені запитами;
- побудова графа компонентів - карти системи з визначенням залежних але не інструментованих (налаштованих для відправлення телеметрії) компонентів, таких як бази даних.

Таблиця 1.1

Порівняння можливостей застосунків для моніторингу

Застосунок	Обов'язкові можливості					Ключові можливості		
	Перегляд сервісів та метрик	Перегляд виключень в системі	Пошук та візуалізація відбитків запитів	Побудова графа системи	Створення сповіщень	Автоматичне визначення аномалій	Інтегрованість із зовнішніми системами	Визначення причин аномалій
Jaeger	-	-	+	-	-	-	-	-
Signoz	+	+	+	+	+	-	-	-
Aspecto	-	-	++	-	-	-	-	-
ServiceNow	++	-	++	+	+	-	++	+
Honeycomb	+	-	++	++	++	-	+	+
New Relic	++	++	++	++	++	++	++	++

Ключові та інноваційні можливості:

- порівняння груп відбитків запитів задля знаходження факторів що сприяють зміні працездатності;
- автоматичне визначення присутності неполадок в системі (anomalies detection);

- автоматичне визначення причин виникнення неполадок в системі (root cause analysis);
- інтегрованість із інфраструктурою (сервера, Kubernetes...) – відображення розгорнутих сервісів на певних серверах, відображення метрик;
- інтегрованість із зовнішніми системами по типу github, платформами Continuous Integration (CI) Continuous Delivery (CD) задля швидкого переходу до проблемних місць, збору інформації про події вивантаження нових версій сервісів, збору додаткових метаданих.

Після аналізу вищезазначених функціональних можливостей було визначено два можливих нових варіанти використання:

- визначення поганих архітектурних практик;
- визначення аномалій в системі.

Розглянемо варіанти використання розроблюваної системи:

- визначення шкідливих практик в системі [8, 1]:
 - вузьке місце - компонент від якого залежить багато інших компонентів з використанням синхронних запитів. Може призвести до крихкості системи коли даний компонент буде недоступним;
 - вузол – кластер з компонентів, що сильно залежать один від одного (high coupling). Дана практика вказує на невірно розділені обов'язки компонентів;
 - нано-сервіс що залежить від багатьох інших. Зачасту це означає, що занадто малий та додає більше проблем в систему з його підтримкою, додає накладних розходів під час запитів через меншу швидкість мережі порівняно з запитами в межах одного процесу;
- визначення аномалій в системі – аналіз ключових метрик компонентів системи з метою знаходження проблемних місць.

Висновок до розділу 1

В даному розділі було розглянуто основну проблематику моніторингу розподілених систем, а саме складність аналізу купи запитів що проходять через багаточисельні компоненти системи. Було оглянуто коротку історію дослідження та розвитку стандартів збору метрик з подальшим описом OpenTelemetry стандарту. Опісля було оглянуто застосунки для моніторингу та аналізу що використовують даний стандарт, в порівнянні брали участь популярні системи з відкритим та закритим кодом, серед яких NewRelic є представником застосунків з продвинутим набором функціоналу для аналізу запитів та груп запитів для знаходження аномалій та їх причин. Однак, при цьому автоматичний аналіз проходить по кожній з ключових метрик окремо, замість того аби аналізувати систему разом, що є одним із завдань дослідженням у цій роботі. Інше завдання – використання телеметрії для аналізу архітектури системи на присутність структурних недоліків, адже телеметрія дозволяє покроково відслідковувати запити, в тому числі асинхронні. Виходячи з цього було поставлено завдання на розробку.

РОЗДІЛ 2. ВИКОРИСТАННЯ ТЕЛЕМЕТРІЇ ДЛЯ ДИНАМІЧНОГО АНАЛІЗУ СИСТЕМИ

2.1. Визначення даних та розробка сховища даних для аналізу поганих практик

Ідея використання телеметрії для покращення структури системи просліджується в декількох дослідницьких роботах, випущених останніми роками [20, 21, 22], та налічує досить малий список проблем, які можуть бути виявлені, що відкриває можливості для подальшого вивчення цієї теми [8].

Запропонована аналітична система отримує постійний потік даних телеметрії (traces, metrics, logs), агрегує їх оновлюючи модель системи, та паралельно зберігаючи дані в аналітичну систему. Після цього, окремий сервіс робить запити до сховищ та відображає дані користувачу.

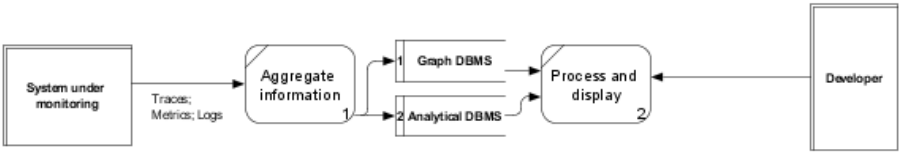


Рис 2.1 Загальна структура процесу отримання та обробки телеметрії

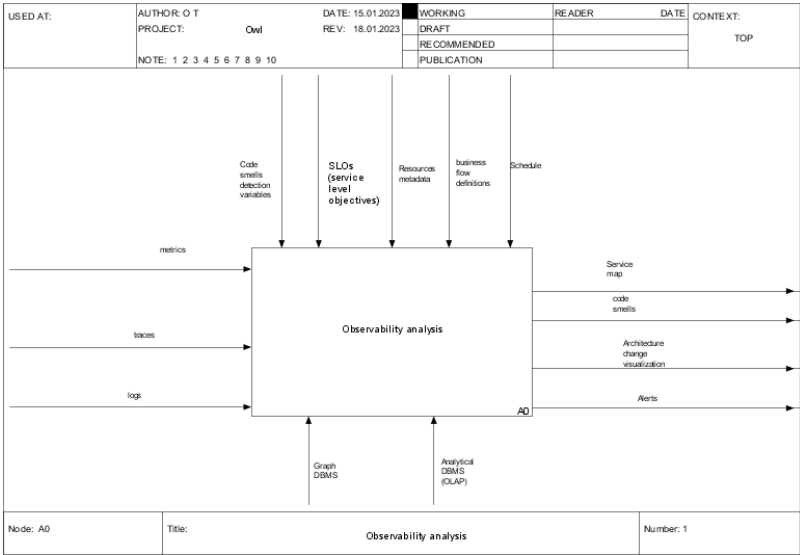


Рис 2.2. Загальна модель процесу аналізу телеметрії

Процес аналізу телеметрії (Рис 2.3) складається з етапів побудови моделі системи у вигляді направленого графа, пошук структурних недоліків, аналіз

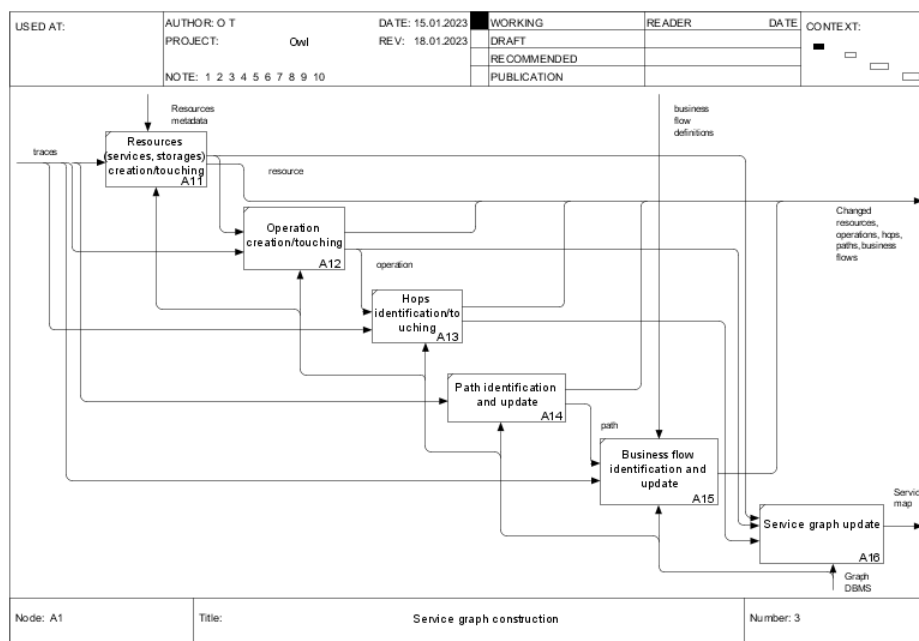


Рис 2.5. Процес побудови графової моделі системи

Для побудови карти сервісів з подальшим аналізом, сховище даних повинно мати таку інформацію:

- ресурси – компоненти системи що взаємодіють. Ресурс повинен мати назву, тип (сервіс, сховище, браузер), має дату створення та останнього використання;
- операції – визиваються іншими ресурсами; мають статистику по кількості викликів, помилок, останньої дати створення, використання;
- виклики – зв'язок між компонентами в межах одного запиту. Має дату створення, останнього використання, тип (синхронний, асинхронний), відсоткову кількість помилок.

Для визначення поганих практик, потрібно мати інформацію про кількість викликів з сервісу та до нього.

Як сховище структури системи було обрано Neo4j зважаючи на те що дане рішення фізично зберігає дані як граф, що дає можливість використовувати алгоритми обходу графа задля визначення поганих практик в системі, а саме:

- коефіцієнт кластеризації - вимірює ступінь зв'язаності вершин; допоможе визначити вузли в системі [23];

- ступінь центральності (degree centrality) - виміряє кількість з'єднань між вершинами; дає можливість підраховувати метрики спорідненості компонентів в системі [24];
- сильно зв'язані компоненти (strongly connected components) – знаходить групи, де кожна вершина доступна з будь-якої іншої; допомагає виявити циклічні залежності в системі [25].

Виходячи з попередніх пунктів, графова СУБД повинна мати наступну структуру.



Рис 2.6. Спрощена діаграма структури графової СУБД

Сховище даних має вершини двох типів: ресурси та операції. Ресурси що пов'язані із операціями відношенням «Provides», надають їй. Щоб показати виклики, використовується відношення «Calls», яке агрегує статистику для всіх ідентичних викликів від одного ресурсу до операції іншого ресурсу.

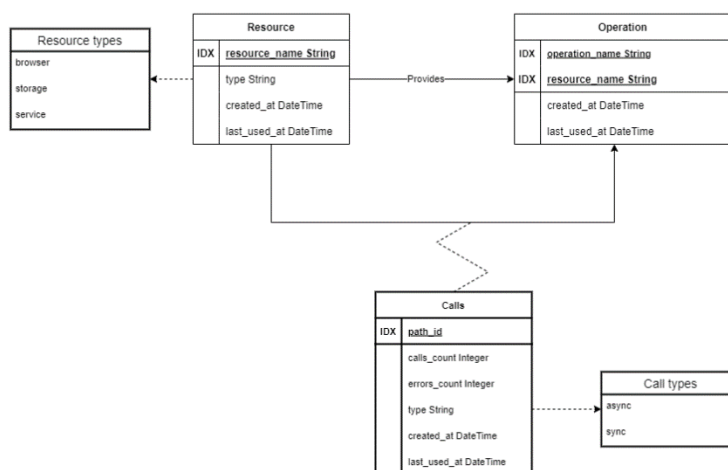


Рис 2.7. Повна діаграма структури графової СУБД

Процес Extract, Transform, Load (ETL) складається з трьох етапів: витягування даних, їх перетворення та вивантаження в аналітичну систему. Витягування телеметрії з системи починається з її інструментації – встановлення

плагінів, компонентів та необов'язкового дописування внутрішнього коду задля більш деталізованого огляду певного процесу в системі. Хоч ці компоненти платформи-залежні, через зрілість стандарту OpenTelemetry налічується багато готових до використання, офіційно підтримуваних додатків, особливо якщо говорити про популярні бібліотеки [26]. Загалом, ці компоненти займаються збиранням простих даних про запити, створення інтервалів, вимірювання метрик, які після цього відправляються в компонент-збирач (рис.2.8) – окремий модульний додаток, розроблений авторами стандарту [27], що дозволяє уніфікувати процес збору, обробки та вивантаження обробленої телеметрії та налічує багато готових компонентів.

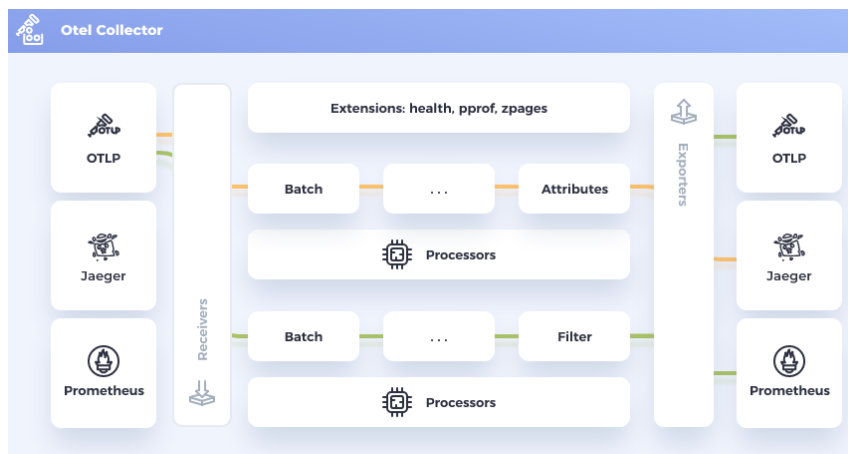


Рис 2.8. Загальна схема компонентів ETL процесу обробки телеметрії

Однак усі ці компоненти покривають широкий спектр варіантів використання, в нашому випадку потрібно перетворювати дані у незвичній манері та вивантажувати це все в незвичну СУБД, тому для цього доцільно створити компонент-вивантажувач власноруч.



Рис 2.9. Поверхнева модель збирання, обробки та вивантаження телеметрії

2.2. Методи і алгоритми інтелектуального аналізу даних для рішення проблеми знаходження аномалій

Проблема знаходження та аналізу аномалій є досить поширеною в computer science та часто варіюється в залежності від домену в якому відбувається аналіз. Так наприклад при зчитуванні даних з датчиків для подальшого аналізу, важливо знаходити та виправляти викиди. Під час аналізу бізнес процесу буває необхідно знаходити незвичайні події щоб аналізувати що до них призвело. В домені розробки та підтримки програмного забезпечення (Software reliability engineering), тема швидкості реагування на проблеми (mean time to detect) – один з найважливіших показників, адже якщо проблема буде знайдена раніше - вона буде вирішена вирішена.

Аналіз аномальних змін вже присутній як мінімум в New Relic, однак він присутній на рівні окремих сервісів, а не усієї системи. Хоч і немає достатніх даних аби стверджувати, дана платформа аналізує метрики, в тому числі ключові, використовуючи підхід Exponential Smoothing [28], що є методом прогнозування однієї змінної та в залежності від варіації може брати до уваги сезонність [29]. Однак, визначати аномалії можна й оберненим способом – від більшого, використовуючи алгоритми що беруть до уваги одразу декілька змінних, що й буде використано в цій роботі.

Аномалія – це не нормальна ситуація, що по суті визначається як сильна різниця між очікуваними та реальними вимірами. Виходячи з цього, можна зрозуміти, що процес визначення аномалії включає процес передбачення значення певної величини базуючись на досі отриманих даних [30].

Проблема знаходження аномалій з великою кількістю вхідних параметрів (multivariate) доволі популярна та критична, адже великий обсяг даних вимагає відповідних алгоритмів, одновимірні набори даних зустрічаються рідко [31].

За типом навчальних алгоритми поділяються на:

- Без учителя – усі дані що застосовуються для навчання моделі не мають позначення аномальності окремих спостережень.
- Напіврозмічені – деяка частина даних позначена, зазвичай тільки нормальні дані.

- Розмічені – всі дані розмічені, найрідше використаний тип алгоритмів так як вимагає повністю позначені дані.

По причині складності отримання розмічених даних, алгоритми без учителя – найбільш популярні, при цьому можна додати можливість зворотної реакції, корекції при використанні алгоритмів для напіврозмічених даних. В даній роботі буде використовуватися саме алгоритм без учителя, при цьому «Жоден із неконтрольованих методів статистично не кращий за інші» [30], що пов'язано із складністю тренування на нерозмічених даних, при якому лишні параметри тільки заважають, тому було вирішено обрати Principal Component Analysis (PCA) - статистичний метод аналізу даних, який використовується для виявлення основних структурних компонентів в наборі даних. Головна мета PCA - зменшити розмірність даних, зберігаючи при цьому якнайбільше інформації, що зважаючи на простоту підходу добре підходить для багато розмірних даних та робить цей алгоритм найбільш поширеним.

Основна ідея PCA полягає у перетворенні початкових змінних (кореляційно пов'язаних) на нові лінійні комбінації, які називаються головними компонентами. Перша головна компонента визначається таким чином, щоб вона пояснювала найбільшу частину дисперсії даних. Кожна наступна головна компонента обирається таким чином, щоб вона була ортогональною до попередніх і пояснювала якомога більше залишкової дисперсії [32].

Висновок до розділу 2

В цьому розділі було проаналізовано процес збору телеметрії, описано структуру відбитків запитів, обрано та визначено структуру сховище даних Neo4j, що з урахуванням графової структури бази дозволить використовувати наявні алгоритми теорії графів для обрахування певних показників для оцінки наявності поганих практик. В другій частині оглянуто підходи аналізу аномалій в системі, обрано PCA алгоритм як той що буде використаний для аналізу метрик в системі.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ АНАЛІЗУ СИСТЕМИ

3.1. Розробка Open Telemetry Collector компонента для трансформації та вивантаження даних

Компонент збірника телеметрії що займається вивантаженням телеметрії в інші системи отримує на вхід групи відбитків запитів, після чого даний компонент робить прохід по кожному інтервалу в запиті (структуру відбитку з коментарями див. додаток А). Один інтервал визначає якусь операцію в системі (див. рис.2.40), це може бути початок обробки отриманого сервісом запиту, якоюсь внутрішньої операції або запитом до іншого сервісу, залежно від чого встановлюється тип інтервалу: «Internal», «Server», «Client», «Producer», «Consumer». Так як запит – є ланцюгом послідовних операцій, усі окрім першого, кореневого, інтервалу мають батьківський. Слідуючи по ланцюгу, ми можемо відрізняти окремі операції, ресурси та виклики (додаток В). Всі ці дані вставляються в базу одним запитом (додаток Б), фрагмент якого можна побачити нижче:

Лістинг фрагменту запиту Cypher по створенню чи оновленню операції в системі:

```
MERGE (operation:Operation {name: $toOperationName, resourceName:
$toResourceName})
ON CREATE
SET
operation.createdAt = timestamp(),
operation.lastUsedAt = timestamp()
ON MATCH
SET operation.lastUsedAt = timestamp()
...
```

3.2. Визначення аномальних подій аналізуючи метрики використовуючи PCA

У випадку визначення аномалій в системі нас цікавить наступна інформація про компоненти системи:

- Виклики: к-ть вхідних вихідних та внутрішніх викликів (синхронних та асинхронних у випадку використання черги або інших брокерів повідомлень) з та без помилок.
- Тривалість: час витрачений на обробку запитів.

Для отримання необхідних даних у вигляді метрик, а також групування та пакування потрібно використати спеціальний компонент-конектор, що трансформує інформацію відбитків запитів у метрики викликів та тривалості. Таким чином збирач отримує інформацію про відбитки запитів по протоколу Open Telemetry Protocol (OTLP), після чого групує, витягає необхідні метрики та вивантажує в файл (повна конфігурацію див. додаток Д).

Лістинг конфігурації процесу агрегації та експортування метрик:

```
service:
  pipelines:
    metrics:
      receivers: [spanmetrics]
      processors: [batch]
      exporters: [file]
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [spanmetrics]
```

В результаті в файл експортується метрики, фрагменти яких можна побачити нижче (повний приклад див. додаток Е):

Лістинг фрагменту метрики виклику:

```
{
  "name": "calls",
  "sum": {
    "dataPoints": [
      {
        "attributes": [
          {
            "key": "service.name",
            "value": {
              "stringValue": "quoteservice"
            }
          }
        ]
      }
    ]
  }
}
```

```

    },
    ...
  ],
  "startTimeUnixNano": "1685509043760171242",
  "timeUnixNano": "1685509058790156649",
  "asInt": "1"
},
],
"aggregationTemporality": 1,
"isMonotonic": true
}
}

```

Метрики за певний період збираються для кожного з компонентів (ресурсів) системи. Метрики мають різні типи значень, так наприклад даний тип – сума, що є лічильником певних подій за період і в даному випадку є монотонною послідовністю, адже кількість викликів ніколи не зменшується (рис.3.1).

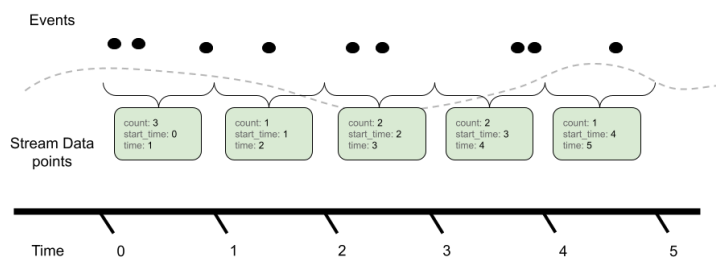


Рис 3.1. Приклад метрики-лічильника

Також важливо зазначити що метрики повертаються як дельта (значення `aggregationTemporality` дорівнює 1), а не кумулятивне значення, адже нас цікавить кількість викликів за певний період, а не абсолютна величина. Кожна метрика може мати декілька точок, які уособлюють різні виміри, що визначаються атрибутами (виміри налаштовуються), таким чином було налаштовано окремі лічильники для різних типів запиту (`span.kind`) та результатів (`status.code`).

Лістинг фрагменту метрики тривалості виклику:

```

{
  "name": "duration",
  "unit": "ms",

```

```

"histogram": {
  "dataPoints": [
    {
      "attributes": [
        {
          "key": "service.name",
          "value": {
            "stringValue": "quoteservice"
          }
        },
        ...
      ],
      "startTimeUnixNano": "1685509043760171242",
      "timeUnixNano": "1685509058790174364",
      "count": "1",
      "sum": 0.006665,
      "bucketCounts": [
        "1",
        "0",
        ...
      ],
      "explicitBounds": [
        0.1,
        1,
        ...
      ],
      "exemplars": [
        {
          "timeUnixNano": "1685509058790174364",
          "asDouble": 0.006665,
          "spanId": "ade03fcb73f18048",
          "traceId": "7f6cf387237813d1f3891b5f21b09be2"
        }
      ]
    },
    ...
  ],
  "aggregationTemporality": 1
}

```

Якщо брати метрику тривалості виклику, то у цьому випадку ми маємо гістограму, що є певною агрегацією значень та розподілення їх по певним проміжкам задля візуалізації.

Але в такому вигляді ми не зможемо використати ці дані, попередньо їх треба обробити та привести до прийнятного вигляду. Першим чином потрібно відокремити усі метрики для кожного сервісу

Лістинг програмного коду розділення метрик:

```
import json
from functools import reduce
from preprocessor import extract_attributes, CallsMetric, DurationMetric

metrics = open('../data/otel/metrics.json', 'r')
Lines = metrics.readlines()
for line in Lines:
    parsed = json.loads(line)
    for resourceMetrics in parsed['resourceMetrics']:
        serviceName = resourceMetrics
        metadata = extract_attributes(resourceMetrics['resource']['attributes'],
{'service.name': 'serviceName'})
        allMetrics = reduce(lambda a, b: a + b, map(lambda o: o.get('metrics',
[]), resourceMetrics['scopeMetrics']))

        for metric in allMetrics:
            name = metric['name']
            if CallsMetric.match(name):
                metric = CallsMetric(metric)
                metric.write(metadata)
            if DurationMetric.match(name):
                metric = DurationMetric(metric)
                metric.write(metadata)
metrics.close()
```

Кожен рядок в файлі метрик – набір метрик який обробляється окремо. За кожний тип метрик відповідає окремий клас.

Лістинг обробника метрик виклику:

```
class CallsMetric:
    @staticmethod
    def match(name):
        return name == 'calls'

    def __init__(self, metric):
        self.metrics = extract_sum_metric(metric,
self.additional_info_extractor)
```

```

def additional_info_extractor(self, data_point, result):
    metadata = extract_attributes(data_point['attributes'], {'span.kind':
'requestType', 'status.code': 'status'})
    result['type'] = span_kind_to_type(metadata['requestType'])
    if metadata['requestType'] == 'SPAN_KIND_INTERNAL':
        return result

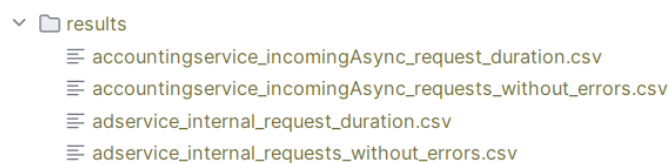
    result['isError'] = metadata['status'].find("ERR") != -1
    return result

def write(self, metadata):
    files_content = dict()
    for metric in self.metrics:
        error_part = 'without_errors'
        if metric.get('isError', False):
            error_part = 'with_errors'
        file_name = '{}_{}_requests_{}.csv'.format(metadata['serviceName'],
metric['type'], error_part)
        if file_name not in files_content:
            files_content[file_name] = []
        files_content[file_name].append(metric)

    for file_name, metrics in files_content.items():
        f = get_metric_file("results/"+file_name)
        for metric in metrics:
            f.write('{}{}\n'.format(metric['time'], metric['value']))
        f.close()

```

При створенні, відбувається витягання інформації про метрики з додатковими даними (чи виклики з помилками, тип). Після цього метрики вивантажуються в окремі файли формату csv з часом та значенням змінної. Як результат створюється 54 файли.



```

v results
├── accountingservice_incomingAsync_request_duration.csv
├── accountingservice_incomingAsync_requests_without_errors.csv
├── adservice_internal_request_duration.csv
└── adservice_internal_requests_without_errors.csv

```

Рис 3.2. Файли з метриками кожного сервісу

```
time,value
1685509058790506925,3.201241
1685509073760723132,86.654854
1685509088760464981,4.221619
1685509103760203727,4.63424825
1685509118760201716,4.838916
```

Рис 3.3. Приклад вмісту файлу метрик

Після цього йде етап об'єднання усіх метрик в один файл. Для цього кожен файл зчитується в об'єкт `DataFrame` бібліотеки `pandas`, після чого йде трансформація формату часу з наносекунд в дату та процес агрегації за якої визначаються середні значення кожної метрики з вікном у хвилину. Вкінці всі файли об'єднуються та видаляються рядки з пустими значеннями.

time	frauddetectionservice_incomingAsync_requests	currencyservice_outgoing_requests_without_	adserv
2023-05-31 04:57:00	0.141367	4.500000	
2023-05-31 04:58:00	0.227574	6.500000	
2023-05-31 04:59:00	0.191880	6.333333	
2023-05-31 05:00:00	0.287567	5.000000	
2023-05-31 05:01:00	0.135161	9.500000	
...	...	...	
2023-05-31 07:33:00	0.094923	9.000000	
2023-05-31 07:34:00	0.076495	7.333333	

Рис 3.4. Результуючий файл з метриками

Лістинг фрагменту об'єднання метрик:

```
import os
import pandas as pd

resampled = []
for file_name in os.listdir("results"):
    df = pd.read_csv("results/" + file_name)
    df['time'] = pd.to_datetime(df['time'])
    df = df.resample('T', on='time').value.mean()
    df.rename(file_name.split('.')[0], inplace=True)
    resampled.append(df)

df = pd.concat(resampled, axis=1)
df.dropna(inplace=True)
```

З проміжних результатів можна чітко побачити кореляцію між різними метриками компонентів системи (рис.3.5), що підтверджується картою кореляцій (рис.3.6).

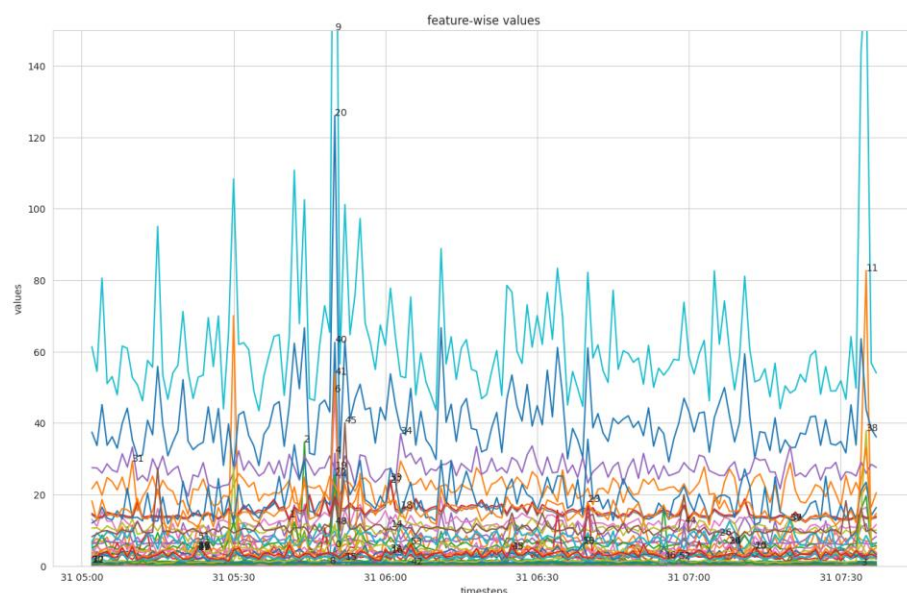


Рис 3.5. Графік значень метрик від часу

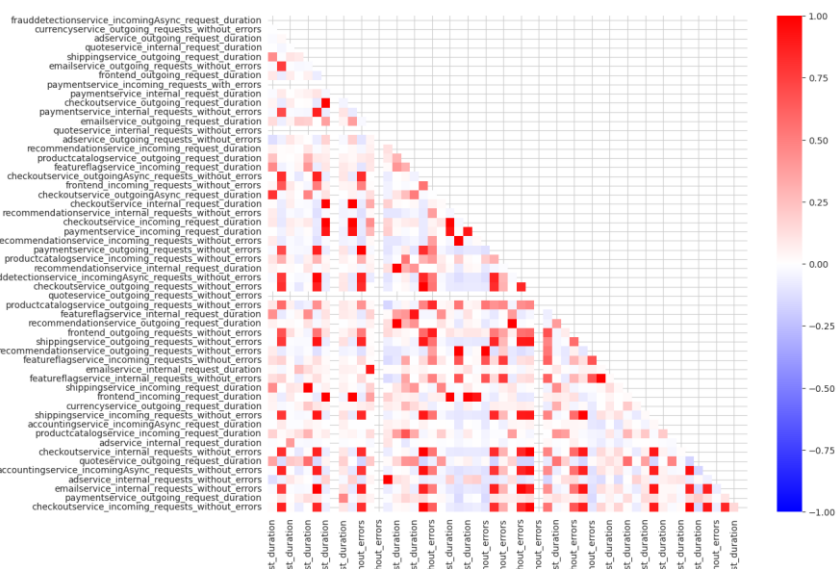


Рис 3.6. Карта кореляції метрик

Процес визначення аномалій відбувається шляхом розбиття вибірки даних на два періоди, перший використовується для тренування статистичної моделі РСА, другий для порівняння з результатами отриманими від моделі та відповідно оцінки помилки для метрик кожної хвилини та навпаки за весь час кожної окремої метрики. Вибірка розбивається 10 разів, при цьому кожного разу точка розділення рухається вперед з певним кроком (TimeSeriesSplit відповідає за розділення впорядкованих даних, на відміну від інших підходів які можуть робити це випадковим чином).

Лістинг фрагменту тренування:

```

from sklearn.model_selection import TimeSeriesSplit
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np

rec_errors_samples = {}
rec_errors_features = {}

for i, (past_id, future_id) in enumerate(
    TimeSeriesSplit(10).split(df)
):
    scaler = StandardScaler()
    pca = PCA(0.7, random_state=33)
    pca.fit(scaler.fit_transform(df.iloc[past_id]))

    df_inverse = pca.inverse_transform(
        pca.transform(
            scaler.transform(df.iloc[future_id])
        )
    )
    time = df.iloc[past_id[-1]].name
    diff = scaler.transform(df.iloc[future_id]) - df_inverse
    rec_errors_samples[time] = np.linalg.norm(diff, axis=1)
    rec_errors_features[time] = np.linalg.norm(diff, axis=0)

```

Висновок до розділу 3

Даний розділ містить налаштування системи Open Telemetry Collector, опис розробленого компонента, що вивантажує інформацію про запити в сховище neo4j. Друга частина містить опис алгоритму структуризації зібраних метрик, їх розділення, групування, та подальший аналіз за допомогою PCA.

РОЗДІЛ 4. АНАЛІЗ РЕЗУЛЬТАТІВ, ПІДСУМКОВІ ГРАФІКИ ТА ТАБЛИЦІ

Як тестову систему було використано проєкт OpenTelemetry Demo [33], що спеціально розроблена для тестування додатків що працюють з телеметрією. Дана система складається з багатьох компонентів що використовують різні технології розробки. Його разом з базою Neo4j було розгорнуто з використанням docker-compose (додаток Г). Конфігурацію компонентів додатку-збирача можна знайти в додатку Д. Сам тестовий проєкт – розподілена система, написана з використанням мікросервісного підходу, що автоматично навантажується з використанням відповідних сервісів.

4.1. Візуалізація графу сервісів за допомогою засобів Neo4j

Після запуску усієї системи з разом з обробником телеметрії, через деякий час графова база має наступні дані (рис.4.1), з рисунку нижче видно, що граф має багато вершин з типом операції а також дещо менше сервісів, між ними видно зв'язки власності та залежності від певної операції.

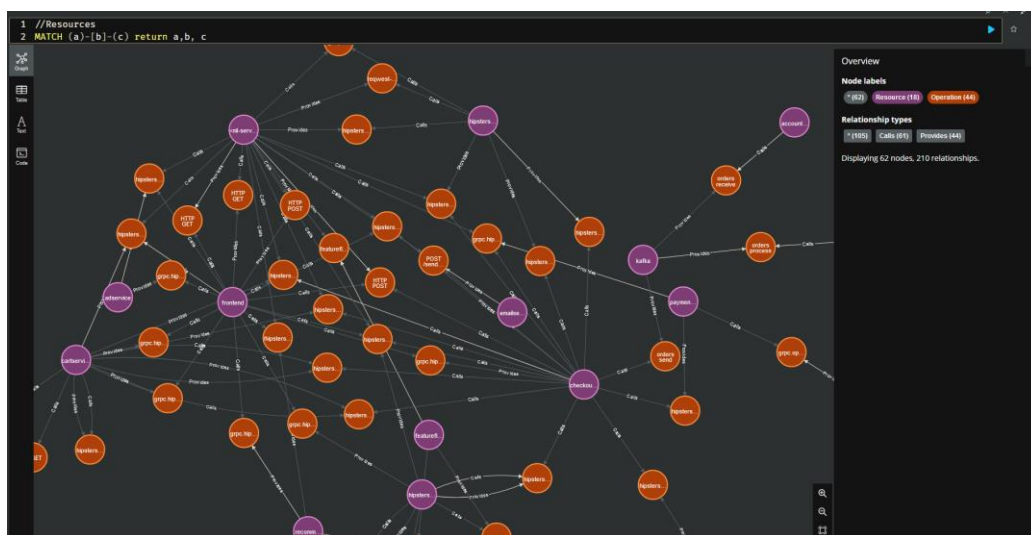


Рис 4.1. Візуалізація повного графу сервісів, операцій та зв'язків між ними за допомогою Neo4j Browser

Для того щоб спростити граф, можна використати функцію з бібліотеки АРОС [34] для Neo4j з метою візуалізації проєкції графа, щоб залежність сервісів була краще видна (рис.4.2).

Лістинг запиту візуалізації тимчасових ребр:

```
MATCH (r1:Resource)-[:Calls]->(:Operation)<-[:Provides]-(r2:Resource)
RETURN r1, r2, apoc.create.vRelationship(r1,'DependsOn',{}, r2) as rel
```

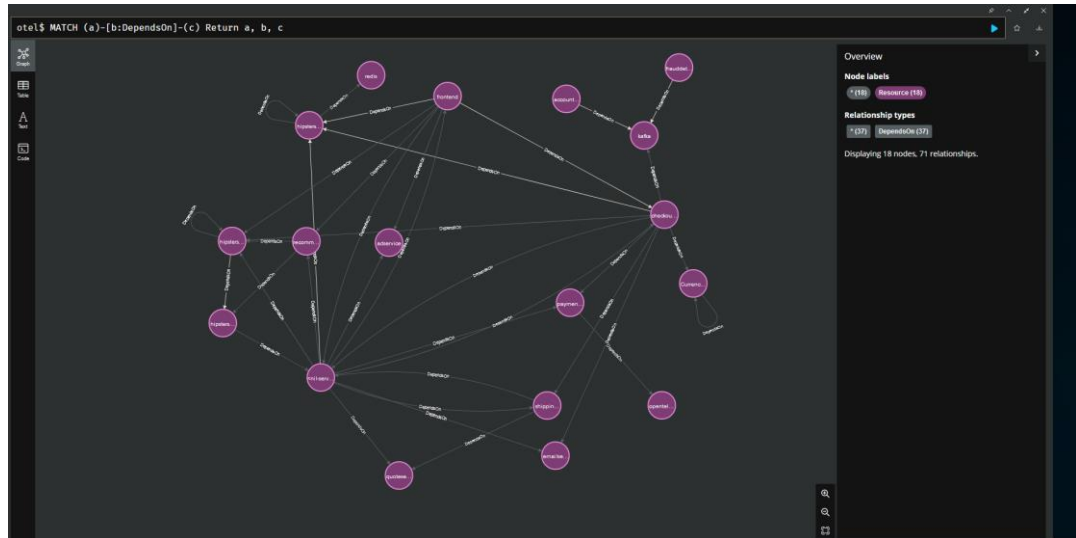


Рис 4.2. Візуалізація графу залежностей в Neo4j Browser

З діаграми вище можна добре побачити залежність сервісу checkout від багатьох інших. Щоб це підтвердити використаємо додаток під назвою Neo4j Bloom для візуалізації алгоритмів кластеризації (Local Clustering Coefficient [35]) та центральності (Degree Centrality [36]).

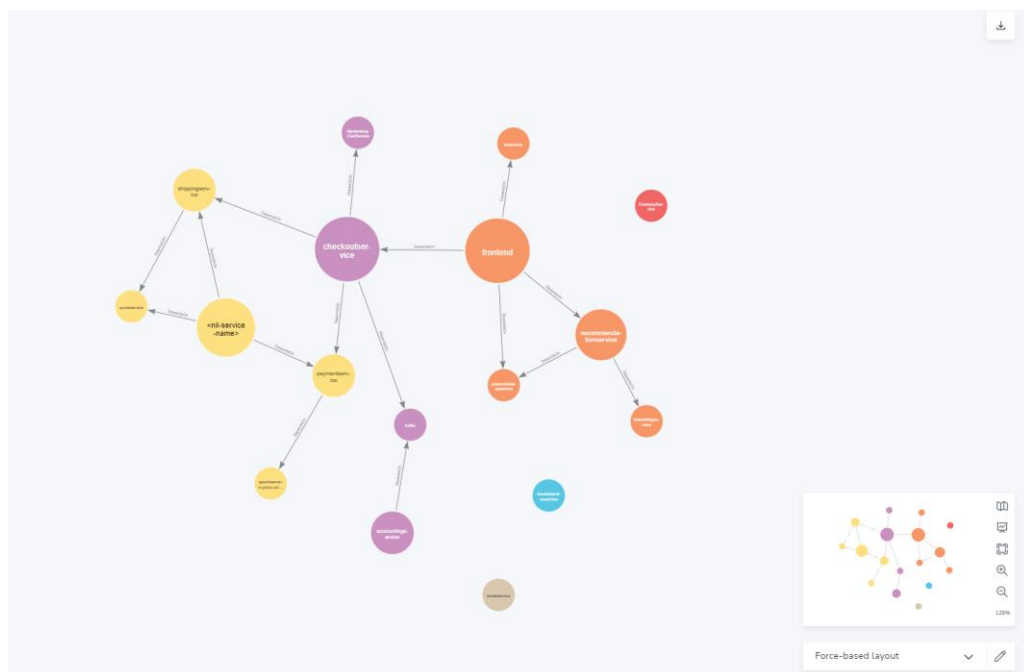


Рис 4.3. Візуалізація графу залежностей сервісів з урахуванням алгоритмів кластеризації та центральності в Neo4j Bloom

На отриманій діаграмі (рис.4.3) кластери позначені різними кольорами, впливовість сервісів – їх розміром, при цьому тут також видно що checkout сервіс має велику впливовість. Таким чином можна швидко проаналізувати архітектуру додатку, частини які можуть бути ключовими в системі та повинні належним чином контролюватися аби не призвести до ситуації, коли при падінні даного компоненту, весь додаток перестав працювати.

4.2. Аналіз часового проміжку на присутність аномалій

Для аналізу на присутність аномалій за час роботи системи було обрано часовий проміжок в декілька годин. Його було оброблено за допомогою відповідного алгоритму та після отримання помилок для кожної з часових точок, можна провести візуальний аналіз на присутність різких збільшень помилки (рис.4.4).

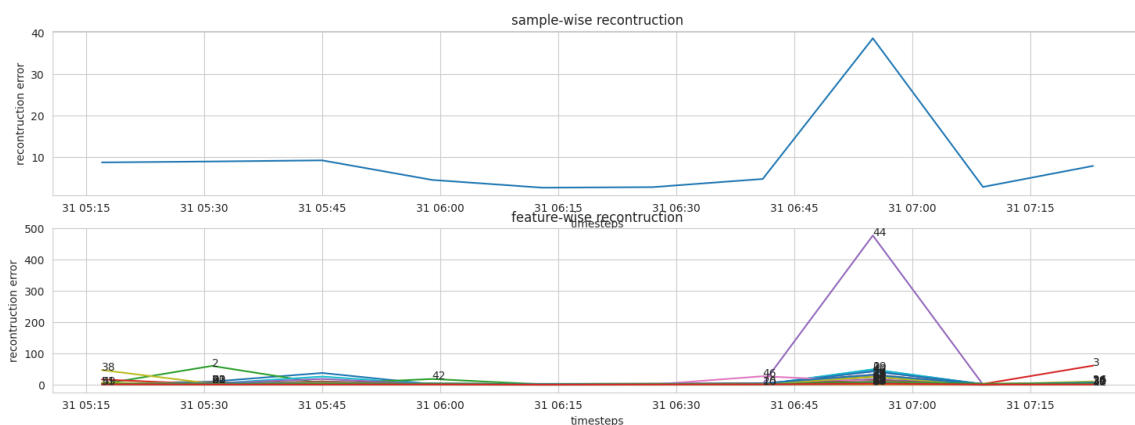


Рис 4.4. Результат виведення помилки опису даних для всіх та окремих метрик

Як видно графіка, в період з 6:50 по 7му годину відбулися якісь зміни, що призвели до великої помилки. З графіку помилок для кожної з ознак, видно що ознака 44 (accountingservice_incomingAsync_request_duration) причетна до цієї помилки, тому провівши більш детальний аналіз значень даної метрики, ми можемо побачити що всі значення тримаються близько 0, в той час коли присутній викид зі значенням близько 12 (рис.4.5).

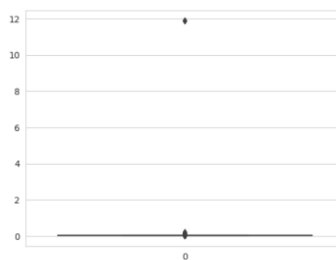


Рис 4.5. Коробковий графік ознаки
accountingservice_incomingAsync_request_duration

Це також підтверджує вибірка значень за відповідний час аномалії (рис.4.6).

time	
2023-05-31 06:51:00	0.020574
2023-05-31 06:52:00	0.036617
2023-05-31 06:53:00	0.021172
2023-05-31 06:54:00	0.028862
2023-05-31 06:55:00	0.028755
2023-05-31 06:56:00	0.018919
2023-05-31 06:57:00	0.018767
2023-05-31 06:58:00	0.023667
2023-05-31 06:59:00	11.874191
2023-05-31 07:00:00	0.044219

Рис 4.6. Значення ознаки accountingservice_incomingAsync_request_duration в період аномалії

Висновок до розділу 4

Аналіз результатів використання програмного модулю щодо знаходження поганих практик з використанням графу системи та алгоритмів теорії графів продемонстрував про можливість та доцільність використання такого підходу для динамічного аналізу системи, адже результуюча діаграма дає побачити групи сервісів, ті що надто зв'язані, сильно залежать один від одного.

Статистична модель РСА продемонструвала достатню продуктивність задля того щоб бути використаною в системах достатньо великого розміру. Даний підхід аналізу дозволив не лише визначити час аномалії, але й відповідну метрику, зміна якої є аномальною, що дає можливість подальшого аналізу причини. Звісно помилка – значення відносне, тому й визначення граничного значення, подолання якого буде вважатися аномалією повинно бути відносним та виставлятися за потреби.

ВИСНОВКИ

У кваліфікаційній роботі магістра було розглянуто тему використання телеметрії для динамічного аналізу системи на предмет аномальних подій та поганих архітектурних практик.

У першому розділі було проведено аналіз проблематики розробки розподілених систем, що дозволило визначити необхідність в застосунках для спостереження та швидкого реагування на проблеми у великій системі. Дослідження використання телеметрії задля динамічного аналізу системи, що були опубліковані останніми роками, показали потенціал даного підходу, а також наявні шляхи розвитку. Було розглянуто історію розвитку теми моніторингу системи та основні аспекти наразі останнього стандарту OpenTelemetry. Далі було проаналізовано одні з найпопулярніших застосунків для моніторингу та аналізу працездатності системи, виокремлено ключові можливості, які пізніше були поділені на ті що зустрічаються у більшості системи та продвинуті, які достатньо сильно спрощують процес вирішення проблем, але досить рідко зустрічаються в більшості систем. За результатами аналізу було виокремлено декілька варіантів використання для подальшого дослідження: визначення поганих архітектурних практик та аномалій в системі.

В другому розділі були визначені основні потоки даних, які потрібні для покриття варіантів використання визначених у попередньому розділі, а також побудована модель графової СУБД, яка налічує такі сутності: операції, ресурси та зв'язки, які визначають напрямок залежність ресурсів та належність операцій. Опісля було розглянуто структуру завантаження, обробки та вивантаження телеметрії за допомогою додатка OpenTelemetry Collector. Було розглянуто основні види алгоритмів виявлення аномалій та обрано статистичний метод PCA для аналізу нерозмічених даних телеметрії.

У третьому розділі розроблено власний компонент додатка-збирача для побудови та вивантаження інформації в базу. Визначено необхідні ознаки що будуть використовуватися, а також процес збору відповідних метрик кількості викликів та тривалості цих викликів всередині системи для знаходження

аномалій. Було описано алгоритм збору метрик. Наступна частина описала підхід побудови статистичної моделі PCA та визначення аномалій.

У четвертому розділі було використано агреговані дані графової СУБД для аналізу поганих практик. Було розглянуто декілька можливих візуалізацій графу залежностей за допомогою Neo4j, Neo4j Bloom, застосовано алгоритми кластеризації, центральності для візуального виявлення проблемних місць в архітектурі системи. Також було проаналізовано результати роботи статистичної моделі на основі алгоритму Principal Component Analysis. Точності даної моделі достатньо для визначення аномальних подій, сутності проблеми та подальшого аналізу для системи великого розміру.

Використання телеметрії для знаходження поганих архітектурних практик має багато шансів на розвиток [21], адже зібраних даних достатньо для визначення більш складних патернів: надто довгі синхронні запити, довгі ланцюги синхронних запитів, а якщо до цього додати інформацію про контексти та групи сервісів [1] також проблеми пов'язані з залежностями різних частин додатку, використання однієї бази в різних частинах додатку. Якщо порівнювати з підходами статичного аналізу, динамічний дозволяє побачити реальну картину роботи усієї системи, усі можливі шляхи запитів які використовуються, точно вказати на компоненти які викликають проблему в продуктивності системи в даний момент, на відміну від статичного аналізу окремих модулів. Також в перспективі ці дані повинні відображатися на головній карті компонентів системи, що наразі не було знайдено в жодному з існуючих, в тому числі комерційних, додатків.

Практичне використання простого статистичного підходу без-учителя PCA продемонструвало можливість використання такої моделі задля визначення аномалій, що дозволяє значно спростити роботу інженерів, адже замість того аби слідкувати за десятками графіків та сервісів та реагувати на повідомлення користувачів в підтримці, даний статистичний аналіз завчасно говорить про виникнення проблем в системі. Якщо порівнювати з підходами аналізу кожної метрики системи окремо (за допомогою відповідних статистичних методів,

наприклад що використовуються в NewRelic [28]), даний підхід дає системну картинку, дозволяючи зрозуміти ситуації в усій системі цілком, але також надає причину виникнення проблеми. В порівнянні з іншими моделями, особливо нейронними мережами, в даному варіанті реалізації не потрібно перетреновувати модель задля адаптації до нормальних змін (наприклад природного збільшення кількості користувачів системи), адже аналіз відбувається в певному вікні, хоча безсумнівно це вікно повинно бути достатнім аби охопити достатню кількість даних для тренування та аналізу і при цьому не бути надто чутливим до «сезонних» змін (наприклад активності вдень проти активності вночі), що потрібно тестувати та визначати на реальній системі. Подальший розвиток аналізу аномалій включає в собі інтеграцію з додатком моніторингу продуктивності системи, можливість конфігурації граничних значень для помилки, додання власноруч створених потоків метрик для аналізу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Talaver, O.V. and Vakaliuk, T.A., 2023. Reliable distributed systems: review of modern approaches. Journal of Edge Computing [Online], 2(1), pp.84–101. Available from: <https://doi.org/10.55056/jec.586>
2. P. D. Francesco, I. Malavolta and P. Lago. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. // International Conference on Software Architecture (ICSA). – 2017. – С. 21–30. DOI: 10.1109/ICSA.2017.24
3. Soldani, Jacopo; Tamburri, Damian Andrew; Van Den Heuvel, Willem-Jan. The pains and gains of microservices: A Systematic grey literature review. // Journal of Systems and Software. – 2018. – №146. – С. 215–232.
4. Söylemez, Mehmet; Tekinerdogan, Bedir; Tarhan, Ayça Kolukısa. Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. // Applied Sciences. – 2022. – №12.
5. A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen and M. Mazzara. From Monolithic to Microservices: An Experience Report from the Banking Domain. // IEEE Software. – 2018. – №35. – С. 50–55.
6. M. Villamizar, al. el. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. // 10th Computing Colombian Conference (10CCC). – 2015. – С. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476
7. Niedermaier, Sina; Koetter, Falko; Freymann, Andreas; Wagner, Stefan. On Observability and Monitoring of Distributed Systems – An Industry Interview Study. // International Conference on Service-Oriented Computing. – 2019.
8. Pigazzini, Ilaria; Fontana, Francesca; Lenarduzzi, Valentina; Taibi, Davide. Towards Microservice Smells Detection. // TechDebt '20: International Conference on Technical Debt. – 2020. DOI: 10.1145/3387906.3388625. Available from: <https://www.researchgate.net/publication/346678327>

9. Cerny, Tomas; Abdelfattah, Amr S. ; Bushong, Vincent; Maruf, Abdullah Al; Taibi, Davide. Microservice Architecture Reconstruction and Visualization Techniques: A Review. // IEEE International Conference on Service-Oriented System Engineering (SOSE). – 2022.
10. Sigelman, Benjamin H., et. el. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure [Электронный ресурс] // Google. – 2010. – Режим доступа до ресурсу: <https://static.googleusercontent.com/media/research.google.com/zh-CN//archive/papers/dapper-2010-1.pdf>
11. Jaeger [Электронный ресурс] – Режим доступа до ресурсу: <https://www.jaegertracing.io/>.
12. Zipkin [Электронный ресурс] – Режим доступа до ресурсу: <https://zipkin.io/>.
13. The OpenTracing Semantic Specification [Электронный ресурс] – Режим доступа до ресурсу: <https://opentracing.io/specification/>.
14. OpenCensus Standard [Электронный ресурс] – Режим доступа до ресурсу: <https://opencensus.io/introduction/>.
15. OpenTelemetry [Электронный ресурс] – Режим доступа до ресурсу: <https://opentelemetry.io/docs/what-is-opentelemetry/>.
16. OpenTelemetry Semantic Conventions [Электронный ресурс] – Режим доступа до ресурсу: <https://opentelemetry.io/docs/specs/semconv/>.
17. OpenTelemetry Project Journey Report [Электронный ресурс]. – 2023. – Режим доступа до ресурсу: <https://www.cncf.io/reports/opentelemetry-project-journey-report/>.
18. Observability Primer [Электронный ресурс] – Режим доступа до ресурсу: <https://opentelemetry.io/docs/concepts/observability-primer/>.
19. Vondrak, A. Why Observability Requires a Distributed Column Store [Электронный ресурс]. – 2022. – Режим доступа до ресурсу: <https://www.honeycomb.io/blog/why-observability-requires-distributed-column-store>.

20. Garrett, Parker; Kim, Samuel; Černý, Tom; Maruf, Abdullah Al. Visualizing Anti-Patterns in Microservices at Runtime: A Systematic Mapping Study. // IEEE Access. – 2023. – С. 4434–4442. DOI: 0.1109/ACCESS.2023.3236165
21. Udara, Isuru; Perera, Indika Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach. // MERCon. – 2021.
22. Guo, Xiaofeng; Peng, Xin; et al. Graph-Based Trace Analysis for Microservice Architecture. // 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. – 2020.
23. Neo4j Local Clustering Coefficient [Электронный ресурс] – Режим доступа до ресурсу: <https://neo4j.com/docs/graph-data-science/current/algorithms/local-clustering-coefficient/>.
24. Neo4j Strongly Connected Components [Электронный ресурс] – Режим доступа до ресурсу: <https://neo4j.com/docs/graph-data-science/current/algorithms/strongly-connected-components/>.
25. Instrumentation [Электронный ресурс] – Режим доступа до ресурсу: <https://opentelemetry.io/docs/instrumentation/>.
26. Instrumentation [Электронный ресурс] – Режим доступа до ресурсу: <https://opentelemetry.io/docs/instrumentation/>.
27. Data Collection [Электронный ресурс] – Режим доступа до ресурсу: <https://opentelemetry.io/docs/concepts/data-collection/>.
28. Boone, Nadya Duke. Dynamic Baseline Alerts Now Automatically Find the Best Algorithm for You [Электронный ресурс]. – 2017. – Режим доступа до ресурсу: <https://newrelic.com/blog/how-to-relic/baseline-alerts-algorithm>.
29. J. Brownlee A Gentle Introduction to Exponential Smoothing for Time Series Forecasting in Python [Электронный ресурс]. – 2020. – Режим доступа до ресурсу: <https://machinelearningmastery.com/exponential-smoothing-for-time-series-forecasting-in-python/>.

30. Songqiao Han, Xiyang Hu, Hailiang Huang, Minqi Jiang, Yue Zhao ADBench: Anomaly Detection Benchmark [Электронный ресурс]. – 2022. – Режим доступа до ресурсу: <https://arxiv.org/abs/2206.09426>.
31. A Systematic Review of Anomaly Detection within High Dimensional. // INTERNATIONAL JOURNAL ON INFORMATICS VISUALIZATION. – 2023. – С. 122–130.
32. Principal component analysis [Электронный ресурс] – Режим доступа до ресурсу: https://en.wikipedia.org/wiki/Principal_component_analysis.
33. OpenTelemetry Demo [Электронный ресурс] – Режим доступа до ресурсу: <https://github.com/open-telemetry/opentelemetry-demo>.
34. Neo4j APOC Library [Электронный ресурс] – Режим доступа до ресурсу: <https://neo4j.com/developer/neo4j-apoc/>.
35. Local Clustering Coefficient [Электронный ресурс] – Режим доступа до ресурсу: <https://neo4j.com/docs/graph-data-science/current/algorithms/local-clustering-coefficient/>.
36. Degree Centrality [Электронный ресурс] – Режим доступа до ресурсу: <https://neo4j.com/docs/graph-data-science/current/algorithms/degree-centrality/>.
37. Xin, Ruyue; Chen, Peng; Zhao, Zhiming CausalRCA: Causal Inference based Precise. // International Conference on Service-Oriented Computing. – 2018.

ДОДАТКИ

Додаток А.

Proto схема структури відбитку запиту

```

// TracesData represents the traces data that can be stored in a persistent
storage,
// OR can be embedded by other protocols that transfer OTLP traces data but do
// not implement the OTLP protocol.
//
// The main difference between this message and collector protocol is that
// in this message there will not be any "control" or "metadata" specific to
// OTLP protocol.
//
// When new fields are added into this message, the OTLP request MUST be updated
// as well.
message TracesData {
    // An array of ResourceSpans.
    // For data coming from a single resource this array will typically contain
    // one element. Intermediary nodes that receive data from multiple origins
    // typically batch the data before forwarding further and in that case this
    // array will contain multiple elements.
    repeated ResourceSpans resource_spans = 1;
}

// A collection of ScopeSpans from a Resource.
message ResourceSpans {
    reserved 1000;

    // The resource for the spans in this message.
    // If this field is not set then no resource info is known.
    opentelemetry.proto.resource.v1.Resource resource = 1;

    // A list of ScopeSpans that originate from a resource.
    repeated ScopeSpans scope_spans = 2;

    // This schema_url applies to the data in the "resource" field. It does not
    apply
    // to the data in the "scope_spans" field which have their own schema_url
    field.
    string schema_url = 3;
}

// A collection of Spans produced by an InstrumentationScope.

```

```

message ScopeSpans {
    // The instrumentation scope information for the spans in this message.
    // Semantically when InstrumentationScope isn't set, it is equivalent with
    // an empty instrumentation scope name (unknown).
    opentelemetry.proto.common.v1.InstrumentationScope scope = 1;

    // A list of Spans that originate from an instrumentation scope.
    repeated Span spans = 2;

    // This schema_url applies to all spans and span events in the "spans" field.
    string schema_url = 3;
}

// A Span represents a single operation performed by a single component of the
// system.
//
// The next available field id is 17.
message Span {
    // A unique identifier for a trace. All spans from the same trace share
    // the same `trace_id`. The ID is a 16-byte array. An ID with all zeroes OR
    // of length other than 16 bytes is considered invalid (empty string in
    // OTLP/JSON
    // is zero-length and thus is also invalid).
    //
    // This field is required.
    bytes trace_id = 1;

    // A unique identifier for a span within a trace, assigned when the span
    // is created. The ID is an 8-byte array. An ID with all zeroes OR of length
    // other than 8 bytes is considered invalid (empty string in OTLP/JSON
    // is zero-length and thus is also invalid).
    //
    // This field is required.
    bytes span_id = 2;

    // trace_state conveys information about request position in multiple
    // distributed tracing graphs.
    // It is a trace_state in w3c-trace-context format:
    // https://www.w3.org/TR/trace-context/#tracestate-header
    // See also https://github.com/w3c/distributed-tracing for more details about
    // this field.
    string trace_state = 3;

```

```

// The `span_id` of this span's parent span. If this is a root span, then this
// field must be empty. The ID is an 8-byte array.
bytes parent_span_id = 4;

// A description of the span's operation.
//
// For example, the name can be a qualified method name or a file name
// and a line number where the operation is called. A best practice is to use
// the same display name at the same call point in an application.
// This makes it easier to correlate spans in different traces.
//
// This field is semantically required to be set to non-empty string.
// Empty value is equivalent to an unknown span name.
//
// This field is required.
string name = 5;

// SpanKind is the type of span. Can be used to specify additional
relationships between spans
// in addition to a parent/child relationship.
enum SpanKind {
    // Unspecified. Do NOT use as default.
    // Implementations MAY assume SpanKind to be INTERNAL when receiving
UNSPECIFIED.
    SPAN_KIND_UNSPECIFIED = 0;

    // Indicates that the span represents an internal operation within an
application,
    // as opposed to an operation happening at the boundaries. Default value.
    SPAN_KIND_INTERNAL = 1;

    // Indicates that the span covers server-side handling of an RPC or other
    // remote network request.
    SPAN_KIND_SERVER = 2;

    // Indicates that the span describes a request to some remote service.
    SPAN_KIND_CLIENT = 3;

    // Indicates that the span describes a producer sending a message to a
broker.
    // Unlike CLIENT and SERVER, there is often no direct critical path latency
relationship
    // between producer and consumer spans. A PRODUCER span ends when the

```

```

message was accepted
    // by the broker while the logical processing of the message might span a
much longer time.
    SPAN_KIND_PRODUCER = 4;

    // Indicates that the span describes consumer receiving a message from a
broker.
    // Like the PRODUCER kind, there is often no direct critical path latency
relationship
    // between producer and consumer spans.
    SPAN_KIND_CONSUMER = 5;
}

// Distinguishes between spans generated in a particular context. For example,
// two spans with the same name may be distinguished using `CLIENT` (caller)
// and `SERVER` (callee) to identify queueing latency associated with the
span.
SpanKind kind = 6;

// start_time_unix_nano is the start time of the span. On the client side,
this is the time
// kept by the local machine where the span execution starts. On the server
side, this
// is the time when the server's application handler starts running.
// Value is UNIX Epoch time in nanoseconds since 00:00:00 UTC on 1 January
1970.
//
// This field is semantically required and it is expected that end_time >=
start_time.
fixed64 start_time_unix_nano = 7;

// end_time_unix_nano is the end time of the span. On the client side, this is
the time
// kept by the local machine where the span execution ends. On the server
side, this
// is the time when the server application handler stops running.
// Value is UNIX Epoch time in nanoseconds since 00:00:00 UTC on 1 January
1970.
//
// This field is semantically required and it is expected that end_time >=
start_time.
fixed64 end_time_unix_nano = 8;

```

```

// attributes is a collection of key/value pairs. Note, global attributes
// like server name can be set using the resource API. Examples of attributes:
//
//      "/http/user_agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36"
//      "/http/server_latency": 300
//      "abc.com/myattribute": true
//      "abc.com/score": 10.239
//
// The OpenTelemetry API specification further restricts the allowed value
types:
// https://github.com/open-telemetry/opentelemetry-
specification/blob/main/specification/common/README.md#attribute
// Attribute keys MUST be unique (it is not allowed to have more than one
// attribute with the same key).
repeated opentelemetry.proto.common.v1.KeyValue attributes = 9;

// dropped_attributes_count is the number of attributes that were discarded.
Attributes
// can be discarded because their keys are too long or because there are too
many
// attributes. If this value is 0, then no attributes were dropped.
uint32 dropped_attributes_count = 10;

// Event is a time-stamped annotation of the span, consisting of user-supplied
// text description and key-value pairs.
message Event {
    // time_unix_nano is the time the event occurred.
    fixed64 time_unix_nano = 1;

    // name of the event.
    // This field is semantically required to be set to non-empty string.
    string name = 2;

    // attributes is a collection of attribute key/value pairs on the event.
    // Attribute keys MUST be unique (it is not allowed to have more than one
    // attribute with the same key).
    repeated opentelemetry.proto.common.v1.KeyValue attributes = 3;

    // dropped_attributes_count is the number of dropped attributes. If the
value is 0,
    // then no attributes were dropped.
    uint32 dropped_attributes_count = 4;

```

```

}

// events is a collection of Event items.
repeated Event events = 11;

// dropped_events_count is the number of dropped events. If the value is 0,
then no
// events were dropped.
uint32 dropped_events_count = 12;

// A pointer from the current span to another span in the same trace or in a
// different trace. For example, this can be used in batching operations,
// where a single batch handler processes multiple requests from different
// traces or when the handler receives a request from a different project.
message Link {
    // A unique identifier of a trace that this linked span is part of. The ID
    is a
    // 16-byte array.
    bytes trace_id = 1;

    // A unique identifier for the linked span. The ID is an 8-byte array.
    bytes span_id = 2;

    // The trace_state associated with the link.
    string trace_state = 3;

    // attributes is a collection of attribute key/value pairs on the link.
    // Attribute keys MUST be unique (it is not allowed to have more than one
    // attribute with the same key).
    repeated opentelemetry.proto.common.v1.KeyValue attributes = 4;

    // dropped_attributes_count is the number of dropped attributes. If the
    value is 0,
    // then no attributes were dropped.
    uint32 dropped_attributes_count = 5;
}

// links is a collection of Links, which are references from this span to a
span
// in the same or different trace.
repeated Link links = 13;

// dropped_links_count is the number of dropped links after the maximum size

```

```

was
    // enforced. If this value is 0, then no links were dropped.
    uint32 dropped_links_count = 14;

    // An optional final status for this span. Semantically when Status isn't set,
    it means
    // span's status code is unset, i.e. assume STATUS_CODE_UNSET (code = 0).
    Status status = 15;
}

// The Status type defines a logical error model that is suitable for different
// programming environments, including REST APIs and RPC APIs.
message Status {
    reserved 1;

    // A developer-facing human readable error message.
    string message = 2;

    // For the semantics of status codes see
    // https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/api.md#set-status
    enum StatusCode {
        // The default status.
        STATUS_CODE_UNSET = 0;
        // The Span has been validated by an Application developer or Operator to
        // have completed successfully.
        STATUS_CODE_OK = 1;
        // The Span contains an error.
        STATUS_CODE_ERROR = 2;
    };

    // The status code.
    StatusCode code = 3;
}

```

Запит на мові Cypher для створення ресурсів та зв'язків між ними

```
MERGE (operation:Operation {name: $toOperationName, resourceName:
$toResourceName})
```

```
  ON CREATE
```

```
  SET
```

```
    operation.createdAt = timestamp(),
```

```
    operation.lastUsedAt = timestamp()
```

```
  ON MATCH
```

```
  SET operation.lastUsedAt = timestamp()
```

```
WITH operation
```

```
OPTIONAL MATCH (r:Resource)
```

```
  WHERE $toResourceName =~ "(?i).*" + r.name + ".*"
```

```
  OR r.name =~ "(?i).*" + $toResourceName + ".*"
```

```
WITH CASE r IS NULL
```

```
  WHEN true THEN $toResourceName
```

```
  ELSE r.name
```

```
  END as resourceName, operation
```

```
MERGE (toResource:Resource {name: resourceName})
```

```
  ON CREATE
```

```
  SET
```

```
    toResource.createdAt = timestamp(),
```

```
    toResource.lastUsedAt = timestamp(),
```

```
    toResource.type = $toResourceType
```

```
  ON MATCH
```

```
  SET toResource.lastUsedAt = timestamp()
```

```
WITH operation, toResource
```

```
MERGE
```

```
  (toResource)-[provides:Provides]->(operation)
```

```
  ON CREATE
```

```
  SET
```

```
    provides.createdAt = timestamp(),
```

```
    provides.lastUsedAt = timestamp()
```

```
  ON MATCH
```

```
  SET provides.lastUsedAt = timestamp()
```

```
WITH operation, toResource
```

```

OPTIONAL MATCH (r:Resource)
  WHERE $fromResourceName =~ "(?i).*" + r.name + ".*"
  OR r.name =~ "(?i).*" + $fromResourceName + ".*"
WITH CASE r IS NULL
  WHEN true THEN $fromResourceName
  ELSE r.name
  END as resourceName, operation, toResource
MERGE (fromResource:Resource {name: resourceName})
ON CREATE
SET
  fromResource.createdAt = timestamp(),
  fromResource.lastUsedAt = timestamp(),
  fromResource.type = $fromResourceType
ON MATCH
SET fromResource.lastUsedAt = timestamp()

WITH operation, toResource, fromResource

MERGE
  (fromResource)-[c:Calls]->(operation)
ON CREATE
SET
  c.createdAt = timestamp(),
  c.lastUsedAt = timestamp(),
  c.callsCount = 1,
  c.errorsCount = CASE $isErrorHop
    WHEN true THEN 1
    ELSE 0
  END,
  c.type = CASE $isAsyncHop
    WHEN true THEN "async"
    ELSE "sync"
  END
ON MATCH
SET c.lastUsedAt = timestamp(),
  c.callsCount = c.callsCount + 1,
  c.errorsCount = c.errorsCount + CASE $isErrorHop
    WHEN true THEN 1
    ELSE 0
  END

```

Впроваджений алгоритм ідентифікації ресурсів та зв'язків

```

func (s *storage) pushTraceData(ctx context.Context, td ptrace.Traces) error
{
    structuredSpans := make(map[pcommon.SpanID]SpanAndResource,
td.ResourceSpans().Len())
    childrenSpans := make(map[pcommon.SpanID][]SpanAndResource,
td.ResourceSpans().Len())

    rss := td.ResourceSpans()
    for i := 0; i < rss.Len(); i++ {
        rs := rss.At(i)
        resource := rs.Resource()

        ilss := rs.ScopeSpans()
        for j := 0; j < ilss.Len(); j++ {
            ils := ilss.At(j)
            spans := ils.Spans()

            for k := 0; k < spans.Len(); k++ {
                span := spans.At(k)

                spanAndResource := SpanAndResource{
                    Span:      &span,
                    Resource: &resource,
                }
                structuredSpans[span.SpanID()] = spanAndResource
                if !span.ParentSpanID().IsEmpty() {
                    childrenSlice, childSliceExists :=
childrenSpans[span.ParentSpanID()]
                    if !childSliceExists {
                        childrenSlice = make([]SpanAndResource, 2)
                        childrenSpans[span.ParentSpanID()] = childrenSlice
                        childrenSlice = append(childrenSlice, spanAndResource)
                    }
                }
            }
        }
    }

    for _, spanAndResource := range structuredSpans {
        span := spanAndResource.Span

```

```

    if span.Kind() != ptrace.SpanKindClient && span.Kind() !=
ptrace.SpanKindServer && span.Kind() != ptrace.SpanKindProducer && span.Kind()
!= ptrace.SpanKindConsumer {
        s.Logger.Info(fmt.Sprintf("skip spanKind %v", span.Kind()))
        continue
    }

    parentAndResource, hasParent := structuredSpans[span.ParentSpanID()]
    _, hasChildrenSpans := childrenSpans[span.SpanID()]
    hasNoCorrespondingServerSpan := !hasChildrenSpans

    // a call produces two spans, one of the client side, one on the server
side, the root span is usually a server call
    // while the leaf can be a client (e.g. when db is called by the
terminating service in a path) - same for producer/consumers

    // if a client span with a corresponding server span in children, then
continue, we will process the parent
    if (span.Kind() == ptrace.SpanKindClient || span.Kind() ==
ptrace.SpanKindProducer) && !hasNoCorrespondingServerSpan {
        continue
    }

    isAsyncRequest := span.Kind() == ptrace.SpanKindProducer || span.Kind() ==
ptrace.SpanKindConsumer
    isHasError := extractIsError(*span)

    hop := Hop{}
    if hasNoCorrespondingServerSpan {
        s.Logger.Info(fmt.Sprintf("No children (server) span, current: %v",
span.SpanID()))
        // a client/producer span may not have corresponding server/consumer
handler if the part is not
        // instrumented (usually databases are not) or is outside of this
system's scope, for situations like that
        hop = Hop{
            IsAsynchronous: isAsyncRequest,
            IsWithError:      isHasError,
            From:             s.newResource(*spanAndResource.Resource),
            To:
s.inferToOperationFromClientSpan(*spanAndResource.Span),
        }
    } else if !hasParent {

```

```

        s.Logger.Info(fmt.Sprintf("No parent (client) span, current: %v,
parent: %v", span.SpanID(), span.ParentSpanID()))
        // usually indicates the root span which can be service or browser
        hop = Hop{
            IsAsynchronous: isAsyncRequest,
            IsWithError:      isHasError,
            From:
s.inferFromResourceFromServerSpan(*spanAndResource.Span),
            To:              s.newOperationFromSpanAndResource(spanAndResource),
        }
    } else {
        hop = Hop{
            IsAsynchronous: isAsyncRequest,
            IsWithError:      isHasError,
            From:            s.newResource(*parentAndResource.Resource),
            To:              s.newOperationFromSpanAndResource(spanAndResource),
        }
    }

    err := s.Writer.WriteHop(&hop)
    if err != nil {
        s.Logger.Error(fmt.Sprintf("Error in writing hops to neo4j: %v", err))
    }
}

return nil
}

```

Частина docker compose впровадженого алгоритму ідентифікації ресурсів та зв'язків

```
otelcol:
  image: signoz/signoz-otel-collector:latest
  container_name: otel-col-custom
  deploy:
    resources:
      limits:
        memory: 100M
  restart: unless-stopped
  command: [ "--config=/etc/otelcol-config.yml" ]
  volumes:
    - ./src/otelcollector/otelcol-custom-config.yml:/etc/otelcol-config.yml
  ports:
    - "4317"          # OTLP over gRPC receiver
    - "4318"          # OTLP over HTTP receiver
    - "9464"          # Prometheus exporter
    - "8888"          # metrics endpoint
  depends_on:
    jaeger:
      condition: service_started
    neo4j:
      condition: service_healthy
  logging: *logging

neo4j:
  image: neo4j:5.4-enterprise
  container_name: signoz-neo4j
  ports:
    - "7474:7474"
    - "7687:7687"
  restart: unless-stopped
  volumes:
    - ./data/neo4j/conf:/conf
    - ./data/neo4j/data:/data
    - ./data/neo4j/import:/import
    - ./data/neo4j/logs:/logs
    - ./data/neo4j/plugins:/plugins
  environment:
    - NEO4J_PLUGINS=["graph-data-science", "bloom", "apoc"]
```

```
-  
NEO4J_dbms_security_procedures_unrestricted=jwt.security.*,bloom.*,gds.*,apoc.*  
-  
NEO4J_dbms_security_procedures_allowlist=apoc.coll.*,apoc.load.*,gds.*,bloom.*,a  
poc.*  
- NEO4J_ACCEPT_LICENSE_AGREEMENT=yes  
- NEO4J_AUTH=neo4j/password  
- NEO4J_apoc_export_file_enabled=true  
healthcheck:  
  test: wget http://localhost:7474 || exit 1  
  interval: 1s  
  timeout: 10s  
  retries: 20  
  start_period: 3s
```

Конфігурація додатка-збирача телеметрії

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318
        cors:
          allowed_origins:
            - "http://*"
            - "https://*"
  jaeger:
    protocols:
      grpc:
        endpoint: 0.0.0.0:14250
      thrift_http:
        endpoint: 0.0.0.0:14268
      # thrift_compact:
      #   endpoint: 0.0.0.0:6831
      # thrift_binary:
      #   endpoint: 0.0.0.0:6832

exporters:
  neo4jtraces:
    datasource: bolt://neo4j:7687/?username=neo4j&password=password&db=otel
  logging:
    #   verbosity: detailed
    sampling_initial: 1
  spanmetrics:
    histogram:
      explicit:
        buckets: [ 100us, 1ms, 2ms, 6ms, 10ms, 100ms, 250ms ]
    dimensions:
      - name: http.status_code
    dimensions_cache_size: 1000
    aggregation_temporality: "AGGREGATION_TEMPORALITY_DELTA"
    metrics_flush_interval: 15s

processors:
  batch:
```

```
    send_batch_size: 10000
    send_batch_max_size: 11000
    timeout: 10s

service:
  pipelines:
    traces:
      receivers: [jaeger, otlp]
      processors: [batch]
      exporters: [neo4jtraces, logging, spanmetrics]
    metrics:
      receivers: [spanmetrics]
      processors: [batch]
      exporters: [file]
```

Приклад вивантажених метрик одного ресурсу

```
{
  "resourceMetrics": [
    {
      "resource": {
        "attributes": [
          {
            "key": "service.namespace",
            "value": {
              "stringValue": "opentelemetry-demo"
            }
          },
          {
            "key": "service.name",
            "value": {
              "stringValue": "quoteservice"
            }
          },
          {
            "key": "host.name",
            "value": {
              "stringValue": "ae6709c99026"
            }
          },
          {
            "key": "host.arch",
            "value": {
              "stringValue": "x86_64"
            }
          },
          {
            "key": "os.type",
            "value": {
              "stringValue": "linux"
            }
          },
          {
            "key": "os.description",
            "value": {
              "stringValue": "5.15.90.1-microsoft-standard-WSL2"
            }
          }
        ]
      }
    }
  ]
}
```

```

    },
    {
      "key": "os.name",
      "value": {
        "stringValue": "Linux"
      }
    },
    {
      "key": "os.version",
      "value": {
        "stringValue": "#1 SMP Fri Jan 27 02:56:13 UTC 2023"
      }
    },
    {
      "key": "process.pid",
      "value": {
        "intValue": "18"
      }
    },
    {
      "key": "process.executable.path",
      "value": {
        "stringValue": ""
      }
    },
    {
      "key": "process.owner",
      "value": {
        "stringValue": "www-data"
      }
    },
    {
      "key": "process.runtime.name",
      "value": {
        "stringValue": "apache2handler"
      }
    },
    {
      "key": "process.runtime.version",
      "value": {
        "stringValue": "8.1.13"
      }
    },
  ],

```

```

{
  "key": "telemetry.sdk.name",
  "value": {
    "stringValue": "opentelemetry"
  }
},
{
  "key": "telemetry.sdk.language",
  "value": {
    "stringValue": "php"
  }
},
{
  "key": "telemetry.sdk.version",
  "value": {
    "stringValue": "0.0.15"
  }
}
]
},
"scopeMetrics": [
  {
    "scope": {
      "name": "spanmetricsconnector"
    },
    "metrics": [
      {
        "name": "calls",
        "sum": {
          "dataPoints": [
            {
              "attributes": [
                {
                  "key": "service.name",
                  "value": {
                    "stringValue": "quoteservice"
                  }
                }
              ],
              "key": "span.name",
              "value": {
                "stringValue": "calculate-quote"
              }
            }
          ]
        }
      }
    ]
  }
]
}

```

```

    },
    {
      "key": "span.kind",
      "value": {
        "stringValue": "SPAN_KIND_INTERNAL"
      }
    },
    {
      "key": "status.code",
      "value": {
        "stringValue": "STATUS_CODE_UNSET"
      }
    }
  ],
  "startTimeUnixNano": "1685509043760171242",
  "timeUnixNano": "1685509058790156649",
  "asInt": "1"
},
{
  "attributes": [
    {
      "key": "service.name",
      "value": {
        "stringValue": "quoteservice"
      }
    },
    {
      "key": "span.name",
      "value": {
        "stringValue": "/getquote"
      }
    },
    {
      "key": "span.kind",
      "value": {
        "stringValue": "SPAN_KIND_SERVER"
      }
    },
    {
      "key": "status.code",
      "value": {
        "stringValue": "STATUS_CODE_OK"
      }
    }
  ]
}

```

```

    }
  ],
  "startTimeUnixNano": "1685509043760171242",
  "timeUnixNano": "1685509058790156649",
  "asInt": "1"
}
],
"aggregationTemporality": 1,
"isMonotonic": true
}
},
{
  "name": "duration",
  "unit": "ms",
  "histogram": {
    "dataPoints": [
      {
        "attributes": [
          {
            "key": "service.name",
            "value": {
              "stringValue": "quoteservice"
            }
          },
          {
            "key": "span.name",
            "value": {
              "stringValue": "calculate-quote"
            }
          },
          {
            "key": "span.kind",
            "value": {
              "stringValue": "SPAN_KIND_INTERNAL"
            }
          },
          {
            "key": "status.code",
            "value": {
              "stringValue": "STATUS_CODE_UNSET"
            }
          }
        ]
      }
    ]
  }
}
],

```

```

    "startTimeUnixNano": "1685509043760171242",
    "timeUnixNano": "1685509058790174364",
    "count": "1",
    "sum": 0.006665,
    "bucketCounts": [
        "1",
        "0",
        "0",
        "0",
        "0",
        "0",
        "0",
        "0",
        "0"
    ],
    "explicitBounds": [
        0.1,
        1,
        2,
        6,
        10,
        100,
        250
    ],
    "exemplars": [
        {
            "timeUnixNano": "1685509058790174364",
            "asDouble": 0.006665,
            "spanId": "ade03fcb73f18048",
            "traceId": "7f6cf387237813d1f3891b5f21b09be2"
        }
    ]
},
{
    "attributes": [
        {
            "key": "service.name",
            "value": {
                "stringValue": "quoteservice"
            }
        },
        {
            "key": "span.name",
            "value": {

```

```

        "stringValue": "/getquote"
    }
},
{
    "key": "span.kind",
    "value": {
        "stringValue": "SPAN_KIND_SERVER"
    }
},
{
    "key": "status.code",
    "value": {
        "stringValue": "STATUS_CODE_OK"
    }
}
],
"startTimeUnixNano": "1685509043760171242",
"timeUnixNano": "1685509058790174364",
"count": "1",
"sum": 2.562971,
"bucketCounts": [
    "0",
    "0",
    "0",
    "1",
    "0",
    "0",
    "0",
    "0"
],
"explicitBounds": [
    0.1,
    1,
    2,
    6,
    10,
    100,
    250
],
"exemplars": [
    {
        "timeUnixNano": "1685509058790174364",
        "asDouble": 2.562971,

```

```
        "spanId": "261f75fbfb9f369e",
        "traceId": "7f6cf387237813d1f3891b5f21b09be2"
      }
    ]
  },
  ],
  "aggregationTemporality": 1
}
}
```