РОЗДІЛ ІІ. ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

УДК 004.75
DOI https://doi.org/10.26661/2786-6254-2025-1-09

РОЗРОБКА ПРОТОТИПУ ДОДАТКА ВИЯВЛЕННЯ АНОМАЛІЙ ШЛЯХОМ ВІДСТЕЖЕННЯ ПОДІЙ У РОЗПОДІЛЕНИХ КОМП'ЮТЕРНИХ СИСТЕМАХ НА ПРИКЛАДІ БАНКІВСЬКОЇ СИСТЕМИ СПОВІЩЕНЬ

Вакалюк Т. А.
доктор педагогічних наук, професор,
завідувач кафедри інженерії програмного забезпечення
Державний університет «Житомирська політехніка»
вул. Чуднівська, 103, Житомир, Україна
orcid.org/0000-0001-6825-4697
tetianavakaliuk@gmail.com

Талавер О. В.
асистент кафедри інженерії програмного забезпечення
Державний університет «Житомирська політехніка»
вул. Чуднівська, 103, Житомир, Україна
orcid.org/0000-0002-6752-2175
olegtalaver@gmail.com

Марцева Л. А.
доктор педагогічних наук, доцент,
професор кафедри комп'ютерної інженерії та кібербезпеки
Державний університет «Житомирська політехніка»
вул. Чуднівська, 103, Житомир, Україна
orcid.org/0000-0001-5037-6565
L.a.martseva@gmail.com

Мінтій І. С.
кандидат педагогічних наук, доцент,
доцент кафедри систем автоматизованого проектування
Національний університет «Львівська політехніка»
вул. Степана Бандери, 12, Львів, Україна
orcid.org/0000-0003-3586-4311
mintii@iitlt.gov.ua

Жиляєв Є. В.
здобувач
Державний університет «Житомирська політехніка»
вул. Чуднівська, 103, Житомир, Україна
orcid.org/0009-0000-4396-7826
evgenzilaev8@gmail.com

Ключові слова: *розподілені комп'ютерні системи, виявлення аномалій, аналіз першопричин, система відстеження подій, мікросервісна архітектура, Apache Kafka, Spring Cloud Sleuth, Apache Spark.*

У статті представлено розробку прототипу додатка для виявлення й аналізу аномалій у розподілених комп'ютерних системах шляхом відстеження подій, реалізованого на прикладі банківської системи сповіщень. Розроблений прототип складається з двох основних модулів: модуля виявлення аномалій і модуля аналізу першопричин. Модуль виявлення аномалій здатен аналізувати дані, що генеруються системою під спостереженням через розподілену систему відстеження подій, та ідентифікувати аномальну поведінку, особливо таку, що впливає на досвід користувача. Модуль аналізу першопричин призначений для аналізу виявленої аномальної поведінки в контексті залежностей сервісу та встановлення сервісу-першопричини аномальної поведінки з урахуванням ефекту поширення аномалій. У роботі детально описано архітектуру та принципи функціонування обох модулів, їх взаємодію через систему асинхронної комунікації Apache Kafka, а також особливості реалізації алгоритмів виявлення різних типів аномалій. Особлива увага приділяється процесу виявлення помилок, порушень порогових значень та аналізу часу відгуку сервісів. Представлено інноваційний підхід до реконструкції структури залежностей аномалії на основі часових міток та ідентифікаторів трейсів, що дає змогу точно визначати першопричини проблем навіть за відсутності повної інформації про всі спени. Прототип розроблено з використанням сучасних технологій та фреймворків з відкритим кодом, зокрема Spring Cloud Sleuth для відстеження подій, Apache Kafka для асинхронної комунікації та Apache Spark для обробки даних. Архітектура системи спроєктована з урахуванням можливості масштабування та легкого додавання нових алгоритмів виявлення аномалій. Прототип успішно виявляє різні типи аномалій, включно з помилками, порушенням порогових значень і аномальним часом відгуку сервісів, а також точно визначає сервіси-першопричини проблем, що дає змогу значно прискорити процес діагностики й усунення несправностей у складних розподілених системах.

DEVELOPMENT OF A PROTOTYPE APPLICATION FOR ANOMALY DETECTION BY TRACKING EVENTS IN DISTRIBUTED COMPUTER SYSTEMS ON THE EXAMPLE OF A BANKING NOTIFICATION SYSTEM

Vakaliuk T. A.
Doctor of Pedagogical Sciences, Professor,
Head of the Department of Software Engineering
Zhytomyr Polytechnic State University
Chudnivska str., 103, Zhytomyr, Ukraine
orcid.org/0000-0001-6825-4697
tetianavakaliuk@gmail.com

Talaver O. V.
Assistant at the Department of Software Engineering
Zhytomyr Polytechnic State University
Chudnivska str., 103, Zhytomyr, Ukraine
orcid.org/0000-0002-6752-2175
olegtalaver@gmail.com

Martseva L. A.
Doctor of Pedagogical Sciences, Associate Professor,
Professor at the Department of Computer Engineering and Cybersecurity
Zhytomyr Polytechnic State University
Chudnivska str., 103, Zhytomyr, Ukraine
orcid.org/0000-0001-5037-6565
L.a.martseva@gmail.com

Mintiy I. S.
PhD in Pedagogy, Associate Professor,
Associate Professor at the Department of Computer Aided Design Systems
Lviv Polytechnic National University
Stepan Bandera str., 12, Lviv, Ukraine
orcid.org/0000-0003-3586-4311
mintii@iitlt.gov.ua

Zhylyaev E. V.
Student
Zhytomyr Polytechnic State University
Chudnivska str., 103, Zhytomyr, Ukraine
orcid.org/0009-0000-4396-7826
evgenzilaev8@gmail.com

Key words: *distributed computer systems, anomaly detection, root cause analysis, event tracking system, microservice architecture, Apache Kafka, Spring Cloud Sleuth, Apache Spark.*

The article presents the development of a prototype application for detecting and analysing anomalies in distributed computer systems by tracking events, implemented on the example of a banking notification system. The developed prototype consists of two main modules: an anomaly detection module and a root cause analysis module. The anomaly detection module can analyse data generated by the system under surveillance through the distributed event tracking system and identify anomalous behaviour, especially those that affect the user experience. The root cause analysis module is designed to analyse the detected anomalous behaviour in the context of service dependencies and identify the service root cause of the anomalous behaviour, taking into account the effect of anomaly propagation.

The paper describes in detail the architecture and principles of operation of both modules, their interaction through the Apache Kafka asynchronous communication system, and the specifics of implementing algorithms for detecting various anomalies. Particular attention is paid to detecting errors, violations of thresholds, and analysis of service response times. An innovative approach to reconstructing the structure of anomaly dependencies based on timestamps and trade identifiers is presented, allowing us to accurately determine the root causes of problems even without complete information about all spans. The prototype was developed using modern technologies and open-source frameworks, including Spring Cloud Sleuth for event tracking, Apache Kafka for asynchronous communication, and Apache Spark for data processing. The system's architecture is designed to be scalable and add new anomaly detection algorithms easily. The study's results demonstrate the proposed approach's effectiveness in detecting anomalies and determining their root causes in distributed computer systems. It accurately identifies the root cause services, which can significantly speed up diagnosing and troubleshooting complex distributed systems.

Актуальність. У сучасних умовах стрімкого розвитку розподілених комп'ютерних систем та мікросервісної архітектури особливої актуальності набуває проблема виявлення й аналізу аномалій у їх роботі. Розподілені системи стають все більш складними, що ускладнює процес моніторингу їх стану та виявлення першопричин проблем. Традиційні методи моніторингу часто виявляються неефективними через розподілений характер сучасних систем, де аномалія в роботі одного компонента може викликати каскадний ефект і призвести до збоїв у роботі інших сервісів. При цьому визначення першопричини проблеми стає нетривіальним завданням, оскільки потребує аналізу великої кількості взаємопов'язаних подій і залежностей між сервісами.

Актуальність розробки спеціалізованих інструментів для виявлення аномалій також посилюється зростаючими вимогами до якості обслуговування користувачів та необхідністю швидкого реагування на проблеми в роботі систем. При цьому важливо не лише виявляти аномалії, але й автоматично визначати їх першопричини, що дасть змогу значно скоротити час на діагностику й усунення проблем. Особливої уваги заслуговує можливість виявлення аномалій на основі даних, що надаються розподіленою системою відстеження подій, без необхідності встановлення додаткових датчиків чи модифікації наявної інфраструктури. Це робить рішення більш універсальним і легким у впровадженні для різних типів розподілених систем.

Аналіз останніх досліджень і публікацій. Аналіз наукової літератури свідчить про значну увагу дослідників до проблематики моніторингу та оптимізації роботи розподілених комп'ютерних систем. Robert Heinrich та інші [1] досліджували виклики у сфері оптимізації продуктивності мікросервісів, визначивши ключові напрями досліджень і проблеми, що потребують вирішення. Їхня робота заклала теоретичне підґрунтя для розуміння специфіки роботи з мікросервісною архітектурою.

Важливий внесок у розвиток автоматизованого тестування продуктивності мікросервісів зробили André de Camargo та співавтори [2], запропонувавши архітектуру для автоматизації тестів продуктивності. Їхнє дослідження демонструє практичні підходи до вирішення проблем моніторингу розподілених систем.

Holger Knoche [3] зосередив увагу на підтримці продуктивності під час поступової модернізації монолітного програмного забезпечення до мікросервісної архітектури. Це дослідження особливо актуальне для організацій, що здійснюють перехід до мікросервісної архітектури. Порівняльний аналіз продуктивності сервісів на основі контейнерів та віртуальних машин представлений у роботі T. Salah та інших [4], що дає можливість краще розуміти особливості різних підходів до розгортання мікросервісів. У своїй попередній роботі [5] ці ж автори дослідили еволюцію розподілених систем у напрямі мікросервісної архітектури. T. Ueda, T. Nakaike та M. Ohara [6] провели характеристику робочого навантаження для мікросервісів, що дало змогу краще зрозуміти особливості їх функціонування під навантаженням.

Особливу увагу слід приділити роботам, присвяченим виявленню й аналізу аномалій. T. Pitakrat та співавтори [7] запропонували архітектурно-орієнтований підхід до ієрархічного онлайн-прогнозування збоїв. T. M. Ahmed та інші [8] досліджували ефективність інструментів управління продуктивністю додатків (APM) для виявлення регресій продуктивності. M. Mdini та співавтори [9] зосередились на моніторингу систем моніторингу мережі з використанням розпізнавання патернів для виявлення аномалій. M. Zasadzinski, V. Muntés-Mulero та M. S. Simo [10] запропонували підхід до аналізу першопричин на основі акторів у розподіленому середовищі.

Проведений аналіз літератури свідчить про активний розвиток досліджень у сфері моніторингу й аналізу аномалій у розподілених системах. При цьому особлива увага приділяється автоматизації процесів виявлення проблем та визначення їх першопричин, а також розробці масштабованих рішень для роботи з мікросервісною архітектурою.

Метою статті є розробити прототип додатка виявлення аномалій шляхом відстеження подій у розподілених комп'ютерних системах на прикладі банківської системи сповіщень.

Виклад основного матеріалу. Розробка прототипу додатка виявлення аномалій шляхом відстеження подій у розподілених комп'ютерних системах на прикладі банківської системи сповіщень передбачатиме розробку таких модулів. Модуль виявлення аномалій має бути здатним аналізувати дані, які генерує система під спостереженням, яка, зі свого боку, має розподілену систему відстеження подій, і повідомляти про аномальну поведінку, особливо таку, що має вплив на досвід користувача. Єдиним джерелом інформації при цьому має бути саме розподілена система відстеження подій. Модуль аналізу першопричини має завдання аналізувати прозвітовану аномальну поведінку в контексті залежностей сервісу, що продемонстрував таку аномальну поведінку, та, враховуючи таке явище, як поширення аномальної поведінки, встановлювати сервіс-першопри-

чину аномальної поведінки. Контекст структури залежностей сервісу також має встановлюватися лише з інформації, що надається розподіленою системою відстеження подій.

Реалізація модулю виявлення аномалій. Розглянемо насамперед *виклики, з якими можемо стикнутися* під час реалізації прототипу додатка виявлення аномалій шляхом відстеження подій у розподілених комп'ютерних системах на прикладі банківської системи сповіщень. Першим викликом, який потрібно вирішити, є той факт, що немає ніякої попередньої інформації про систему під спостереженням. Оскільки виявлення аномалій має працювати з будь-якою програмою, який має розподілену систему відстеження подій, єдиним попередньо доступним знанням є поведінка Spring Cloud Sleuth в контексті конкретних подій.

Наступною вимогою до системи є виявлення аномалій у режимі онлайн. Це означає, що дані мають бути класифіковані як нормальні або аномальні, як тільки вони стають доступними. Однак обчислення порогових значень прийнято виконувати перед початком виявлення аномалії.

Крім того, цей прототип обмежено використанням лише даних, що доступні з розподіленої системи відстеження подій. Досить часто використовуються апаратні датчики, наприклад, для збору інформації щодо завантаження ресурсів ЦПУ. Розроблений прототип обмежений в об'ємі доступної інформації до лише тієї, що надається Spring Cloud Sleuth. Без додаткових налаштувань це буде лише інформація про тривалість спену й інформація для виявлення відповідної служби та методу, де згадується той самий спен. Однак із використанням додаткових налаштувань для збору інформації додатково отримується інформація щодо завантаження ресурсів пам'яті JVM та середнього використання ресурсів ЦПУ в останню хвилину.

Нарешті, модуль для виявлення аномалій повинен підтримувати модульний підхід, який було встановлено під час проєктування системи. Це потребує асинхронної комунікації з розподіленої системи відстеження подій до та із системою аналізу першопричини після виявлення аномалій з використанням теми системи Kafka. Крім того, алгоритми, які використовуються для виявлення аномалій, повинні бути реалізовані таким чином, щоб декілька алгоритмів могли працювати паралельно, а додавання нового алгоритму могло бути легко здійснено.

Проведемо огляд робочого процесу виявлення аномалій. Базову концепцію реалізації системи виявлення трьох типів аномалій, описаних вище, показано на рисунку 1. Для вилучення інформації з Kafka використовується загальновживаний модуль, адже цей процес є однаковим для всіх трьох алгоритмів. Звітування про аномалію також схоже для всіх алгоритмів, у кінці інформація про всі аномалії має бути передана в той самий топік Kafka в попередньо визначеному форматі (JSON).

Рис. 1. Концепція робочого процесу виявлення аномалій

Програму має бути розроблено таким чином, щоб довільна кількість алгоритмів для виявлення аномалій могла бути встановлена та запущена паралельно. Вони можуть виявляти як ті самі, так і різні аномалії – за умови, що вони роблять це на основі даних, що стосуються спенів. Інтерпретація цих даних про аномальні спени є завданням модуля аналізу першопричини.

Перед тим, як будь-який з алгоритмів виявлення аномалій може почати індивідуальну обробку даних, програма Spark потребує доступ до даних, які були прозвітовані розподіленою системою відслідковування подій і збережені в Kafka за допомогою інструментарію Spring Cloud Sleuth. Топік містить об'єкти спенів у форматі JSON. Це є контейнером для інформації про певну кількість спенів, а також сервіс, якого вони стосуються. JSON об'єкт також поєднується з додатковою інформацією у заголовку перед ним.

Найбільш цікава інформація міститься в об'єктах спенів. Особливо тривалість та теги, що являють собою атрибути спену та використовуються алгоритмами виявлення аномалій. Ідентифікатори, ім'я та інформація про хост слугуватимуть для ідентифікації та групування запитів, які представляються відповідними спенами.

Для створення початкового потоку з вхідними даними з топіку Kafka використовується спеціальний коннектор з Apache Spark Streaming, який, зі свого боку, використовується для реалізації виявлення аномалій. Після того як потік даних відкрито, він може бути змінений за допомогою операцій фільтрації та меппінгу. На першому етапі видаляється інформація заголовка, тому залишається лише JSON, який, зі свого боку, перетворюється в об'єкт спену для Spring Cloud Sleuth для полегшеної обробки.

Оскільки аномалії виявлятимуться на рівні спенів, вони мають бути отримані з листа все-

редині об'єкта спену. Після вилучення окремих спен- об'єктів вони поєднуються з відповідним об'єктом хоста як tuple (структура даних). Це робиться для збереження інформації про хост під час обробки даних алгоритмами для виявлення аномалій. Отримана структура даних використовується в потоці даних, який і обробляється алгоритмами для обчислення та маніпуляції.

Виявлення помилок виконується, як показано на рис. 2. Це процедура відсіву на вхідному потоці даних спенів, який надається алгоритму. Відсів здійснюється з використанням http-кодів і тегів помилок, що надаються Spring Cloud Sleuth. Ті спени, що залишаються в кінці, вважаються аномаліями.

На першому етапі відсіюються всі спени, які мають код стану, визначний як нормальний, або ж ті, що мають порожній тег помилки. У прототипі нормальним кодом стану вважається 200 (ок) та 201 (створено). Однак у разі потреби цей список можна розширити.

Рис. 2. Процес виявлення помилок

На наступному етапі відсіюються всі спени з кодом стану 404 (не знайдено). У системі цей код статусу надавався, лише якщо користувач запитував адресу, якої в системі не було. Це б призвело до збільшення кількості звітованих аномалій, якби велика кількість користувачів неправильно вводили URL-адреси. Коли ж сервіс внутрішньо надсилає запит до іншого сервісу, який не є активним, тоді код статусу буде 500 (внутрішня помилка сервера). Це робиться для розділення цих двох різних випадків. Однак цей фільтр можна видалити в разі потреби.

Наступний етап потрібен для протидії деяким наслідкам обробки винятків фреймворком. Замість того щоб зупинити процес відслідковування події на спені, на якому було повідомлено про виключення, класи для обробки винятків викликаються дещо пізніше. Це призводить до того, що інформація щодо події не обмежується конкретним спеном, але також міститиме виклики класів для обробки виключень. Щоб запобігти цим додатковим спенам, можна використати налаштування Spring Cloud Sleuth. Може статися, що виняток в одному сервісі спричинить виняток в іншому сервісі – наприклад, коли очікуваний код статусу є 200 (ок), але отриманий код є 500 (внутрішня помилка сервера). Такий виняток не буде відфільтрований, адже в нього буде окрема інформація щодо помилки. Встановлення того, який із винятків є першопричиною, є завданням модуля аналізу першопричини.

Щоб позбавитися спенів, які не містять корисної інформації, останнім кроком фільтра для виявлення помилок є відсів усіх спенів із кодом стану 400 (неправильний запит) і 500 (внутрішня помилка сервера), що не мають тегу помилки. Відфільтрування таких спенів без тегу помилки не призводить до втрати інформації про жодний із винятків, але покращує звітування про аномалії, що є актуальними.

Спени, які залишаються після всіх етапів відсіву, будуть прозвітовані як аномальні і потім оброблені за допомогою модуля аналізу першопричини. Це було свідомим рішенням аналізувати спени не лише з кодом статусу 400 і 500, оскільки це дає можливість виявляти випадки, які ще не відомі, але можуть опинитися і в іншій системі. Якщо ж зосередитися лише на відомих помилках, такі нові випадки будуть просто проігноровані.

Цей модуль виявлення аномалій є досить простим. Його можна запускати в декілька копій працюючих паралельно з різними налаштуваннями. Можна також вказати, який з тегів потрібно очікувати, тобто яку саме метрику з тих, що збираються розподіленою системою відслідковування подій, необхідно порівнювати з визначеними пороговими значеннями (рис. 3).

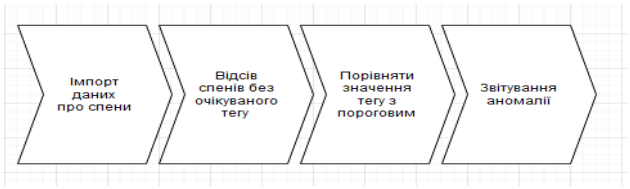

Рис. 3. Процес виявлення порушень порогових значень

Крім того, можна також визначити, чи такий тег очікується присутнім у кожному спені, чи лише в деяких. Якщо тег позначено як такий, що очікується лише на деяких спенах, першим кроком є видалення всіх спенів, на яких цього тегу немає. У випадку, якщо тег очікується на кожному спені, діапазони без тегу звітуватимуться як аномальні.

Після цього кроку відсіву значення тегу буде вилучено для порівняння його з пороговим значенням, що встановлюється попередньо. Якщо значення перевищує порогове, спен буде звітований як аномальний і спрямований для подальшої обробки за допомогою модуля аналізу першопричини.

Найскладнішим алгоритмом виявлення аномалій у цьому прототипі є алгоритм виявлення занадто високого часу відгуку. Цей алгоритм має фазу тренування для розрахування порогових і базових / очікуваних значень. Результат цієї фази

потім використовується під час фази виявлення аномалій.

Тренувальна фаза. Ця фаза потребує набору історичних даних, для чого є дуже корисним функціонал сховища Kafka. Kafka зберігає усі дані з топіку на основі визначеної політики. Коли новий клієнт починає зчитувати дані з топіку, він може вибрати отримати всі дані з топіку після моменту під'єднання клієнта. Іншим варіантом, який і використовується для тренування, є отримання всіх даних топіку.

Під час імпорту тренувальних даних програма збирає такі ідентифікатори: назва сервісу; IP-адреса та порт хоста; URL-адреса сервісу; http-метод або ж назва класу відповідного контролера. Уся ця інформація доступна за усталеним налаштуванням за допомогою Spring Cloud Sleuth. З її використанням створюється набір усіх сервісів, що повідомили про той самий спен під час тренування. Сервіси, які ніколи раніше не повідомляли про діапазон, не розпізнаються алгоритмом і не можуть спостерігатися під час наступної фази.

Наступним кроком в обробці даних є розділення його на декілька менших наборів, по одному на кожний окремий сервіс. Оскільки можна припустити, що кожен сервіс матиме різний характер поведінки, цей крок має сенс – він забезпечує основу для визначення очікуваних значень, не покладаючись на алгоритми машинного навчання для правильного групування. Це дає змогу зменшити кількість функцій на наступному етапі навчання для кожної з моделей. Для кожного з піднаборів буде розраховано окремі базові та порогові значення. Для цього прототипу лише тривалість спену включена в тренування, а включення інших показників не покращило виявлення аномалій. Для врахування факту, що набір даних для тренування сам по собі можете містити аутлаєри, було виключено 0,1 % даних, які мали найбільші значення тривалості. Завдяки цьому кінцевий результат для визначення базових значень буде менш зміщений у бік екстремально великих значень. Чим менші екстремальні найбільші значення, тим менший вплив матиме цей крок на кінцеві результати. Якщо ж видалити більший відсоток даних, може бути видалено занадто велику кількість екстремальних значень, що призведе до збільшення кількості хибнопозитивних спрацювань на етапі визначення аномалій.

Після цього цей набір даних обробляється для навчання моделлю KMeans з k = 1 для кожного кластерного центру. Це робиться за допомогою імплементації цієї моделі від Apache Spark Mlib. Визначається єдиний кластерний центр, оскільки кожен сервіс матиме окрему модель. Кластерний центр цієї моделі є точкою відліку для визначення аномалій. Для прототипу кластер тренуватиметься, використовуючи тривалість спенів.

Додатково до точки відліку потрібне також порогове значення. Воно являє собою дистанцію від точки відліку, що є прийнятною. Тільки якщо відстань буде вища за порогове значення, точка даних буде визначена аномальною. У кінці тренування моделі дистанція кожної з точок даних у тренувальному наборі від точки відліку буде розрахована й відповідно до цієї відстані буде надано п'ять значень: максимальна відстань; медіанна відстань; середня відстань; 95-й процентиль відстаней; 99-й процентиль відстаней.

Для прототипу використовується 99-й процентиль відстаней, оскільки він показав найбільш збалансовані результати щодо виявлення аномалій і за кількістю помилкових спрацювань. Залежно від розподілу значень у тренувальному наборі даних може знадобитися інший набір критеріїв для визначення порогового значення.

Наприкінці фази тренування ми отримуємо точку відліку та порогове значення для кожного сервісу. Оскільки це вся інформація, що є необхідною для етапу визначення аномалій, не має значення, скільки даних потребується для отримання цих результатів. Як тільки дані наявні, можна розпочинати наступний етап.

Фаза виявлення. Після завершення фази тренування можна розпочинати виявлення аномалій у режимі онлайн. Головним завданням цієї фази є оцінка вхідних даних і визначення, чи відхиляються вони від очікуваної норми, чи ні.

Такий розрахунок є досить простим. Для кожного вхідного спену розраховується ідентифікатор сервісу. Потім цей ідентифікатор використовується для визначення правильної точки відліку та порогового значення. Якщо відстань між вхідним спеном і точкою відліку перевищує порогове значення, то спен вважатиметься аномальним.

Якщо вхідний спен матиме ідентифікатор сервісу, який ще не зустрічався, він буде проігнорований. Причиною цього є те, що за такого підходу формування моделі немає гарантії, що модель охоплюватиме всі сервіси. Щоб запобігти великої кількості помилкових повідомлень про аномальність даних, такі сервіси ігноруватимуться до того моменту, як для них буде створено відповідну модель. Під час наступного перерахунку моделей до них буде додано інформацію і про новий сервіс, таким чином модель може адаптуватися до мінливого середовища, навіть якщо попередні дані про систему відсутні.

У поточній реалізації прототипу потрібно перезапустити всю систему, щоб оновити базові моделі

алгоритму. Однак, оскільки єдиною інформацією, яка необхідна для виявлення аномалій, є карта даних, що містить точку відліку та порогове значення для кожного сервісу, така карта даних може бути розрахована окремо і потім використана в уже запущеному алгоритмі під час його роботи.

Як тільки спен буде ідентифіковано як аномальний, про це потрібно відзвітувати і топік Kafka. Саме звідти модуль аналізу першопричини отримає дані про аномальні спени. Тому всі детектори аномалій мають використовувати спільний формат JSON і записувати дані про виявлені аномалії в той самий топік. Вибраний формат зображено на рис. 4.

```
{
    "endpointIdentifier": "service@host:port/uri:::methodName",
    "spanId": "1234567890",
    "traceId": "1234567890",
    "anomalyDescriptor": "splittedKMeans",
    "begin": "100000000",
    "end": "999999999",
    "parentId": "1234567890"
}
```

Рис. 4. JSON-формат звітованих аномалій

Прозвітований JSON є спрощеним представленням спену. Він містить лише ту інформацію, що є необхідною для аналізу першопричини. Це означає, що все теги та інформація про тривалість, які були важливі саме для виявлення аномалії, тепер видалені, оскільки аналіз першопричини знає, що всі вхідні спени є аномальними. Ідентифікатори ж зберігаються. Також додається окремий дескриптор алгоритму, оскільки та сама аномалія може бути прозвітована різними алгоритмами.

Операція зі створення екземпляра producer для Kafka є дуже дорогою, але він є необхідним для публікації даних до топіку з програми. Через це потрібно повторно використовувати producer якомога довше. У разі використання нового producer для кожного окремого аномального спену, швидкодія програми значно погіршується.

Якщо нові алгоритми, додані до системи, також зберігають вихідний формат JSON, як продемонстровано у фрагменті коду 02, їх можна легко інтегрувати в той самий робочий процес паралельно з існуючими алгоритмами, а їх результати також будуть включені в аналіз першопричини аномалії.

Реалізація модуля аналізу першопричини. Ті виклики, що були згадані в контексті модуля виявлення аномалій, також стосуються і модуля аналізу першопричини. Для цього прототипу все ще немає інформації про структуру сервісу. Аналіз першопричини також має виконуватися в режимі онлайн і базуватися лише на інформації, отриманій від розподіленої системи відслідковування подій. Крім того, модульний підхід має продовжуватися і тут.

Особливо останній пункт є важливим для аналізу першопричини. Вхідними даними для цього етапу є звітовані аномальні спени, отримані від різних алгоритмів. Оскільки таки алгоритми можуть виконуватися паралельно один до одного, є можливість, що деякі спени були прозвітовані декілька разів різними алгоритмами, що відрізняється від підходу в модулі виявлення аномалій, де кожен спен спостерігався як унікальний.

І, нарешті, найбільшим викликом на цьому етапі є зменшення обсягу інформації, через що адміністратору потрібно буде продивлятися для розуміння першопричини аномалії. Це означає, що вихідна інформація, яка повертається на цьому етапі, має бути настільки мінімальною, наскільки це можливо, при цьому без втрати необхідної інформації, такої, що стосується самої аномалії, та ідентифікаторів сервісу, що її спричинив.

Підготовка даних. Основною метою цього кроку є зчитування даних з Kafka, вирішення проблеми дублюючих звітів на ті самі спени та надання решти даних у певному форматі для подальшої обробки.

Інформація на цьому етапі визначається форматом вихідних даних із модуля виявлення аномалій. У цьому випадку це Span ID та Trace ID; ідентифікатор сервісу, що створив спен; часова позначка початку і кінця спену; дескриптор аномалії (який налаштовується окремо); ідентифікатор батьківського елемента, якщо такий є.

Щоб об'єднати продубльовані звіти про ті самі спени, є підхід Apache Spark Streaming до обробки даних у мініпакетах даних. Адже з припущенням, що всі звіти, що стосуються того самого спену, надійдуть до системи приблизно в той самий час, вони всі будуть у тому самому мініпакеті даних. Для кожного процесу обробки всі спени групуються за своїми ідентифікаторами, після чого вони можуть бути об'єднані у єдиний звіт, об'єднавши їх дескриптори аномалії та залишаючи всю іншу інформацію, яка повторюється. У кінці цього етапу потік аномалій матиме лише один запис у топіку на кожен унікальний спен, без дублікатів. Наступний крок аналізу першопричини може припускати, що кожен аномальний спен є унікальним.

Огляд робочого процесу визначення першопричини. Початкова ситуація, ще до того, як аналіз першопричини розпочинає свою роботу, показана на рис. 6. Аномальна поведінка сервісу E спричиняє аномальні результати в сервісах A та B через поширення помилок, а тому спени із сервісів A та B також будуть ідентифіковані як аномальні, хоча поведінка самих сервісів була нормальною. Отже,

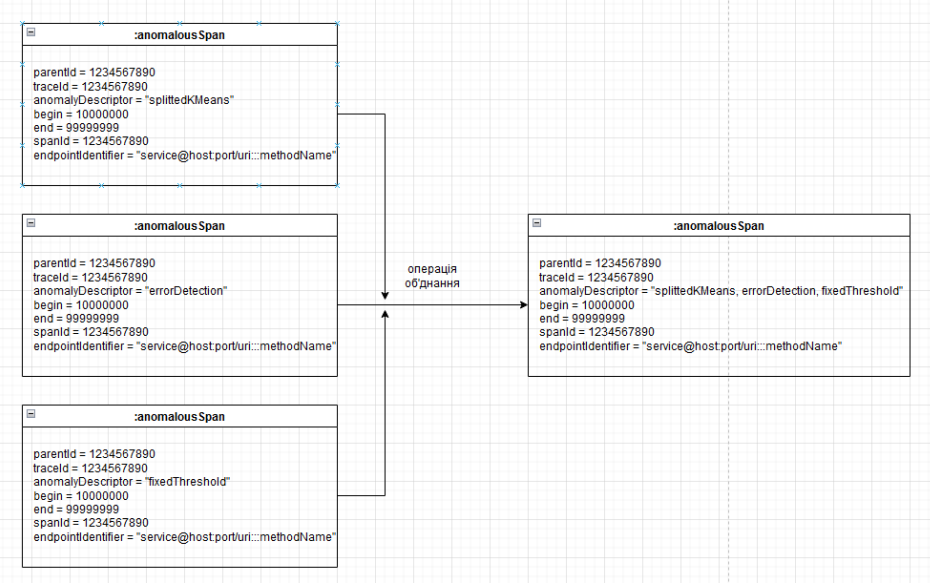

Рис. 5. Об'єднання дублікатів звітів на той самий спен

з погляду звітування про аномальну поведінку це є помилковим спрацюванням системи.

Щоб зменшити кількість таких випадків, потрібно проаналізувати контекст виникнення аномальних спенів і виділити сервіс, який є першопричиною аномальної поведінки.

Не вся доступна інформація може бути корисною для такої цілі. Наприклад, ідентифікатор спену не несе в собі ніякої корисної інформації. Будучи лише випадково згенерованим числом, він не надає жодної інформації про порядок генерації цих спенів.

Ідентифікатор сервісу та дескриптор аномалії можуть містити корисну інформацію, як тільки першопричина ідентифікована, оскільки вони надають інформацію про те, де саме в сервісі виникла аномалія. Однак до того, як сервіс-першопричину буде встановлено, користі з цієї інформації також немає.

Проте ідентифікатор трейсу містить досить корисну інформацію. Трейс є колекцією спенів, які разом становлять історію єдиного запиту. Кожен спен у межах того самого трейсу стосується того самого запиту. Це означає, що якщо виявлена аномалія має ефект поширення, це відобразиться на інших спенах у межах того самого трейсу. Тому ідентифікатор трейсу може бути використаний для групування спенів разом.

Ідентифікатор батьківського процесу може мати дуже обмежене використання в цьому випадку. У теорії ідентифікатор батьківського процесу мав би надати інформацію для виявлення відношень спенів у межах трейсу між собою. Однак проблема в тому, що це може спрацювати, лише якщо ми маємо інформацію про весь трейс, що в цій ситуації не є гарантовано. Аналіз першопричини може лише працювати з тими спенами трейсу, які були прозвітовані на етапі виявлення аномалій. Це може бути і весь трейс, але може статися так, що деякі спени не були ідентифіковані як аномальні. При цьому ідентифікатор батьківського процесу може бути використаний, але лише як запасний варіант.

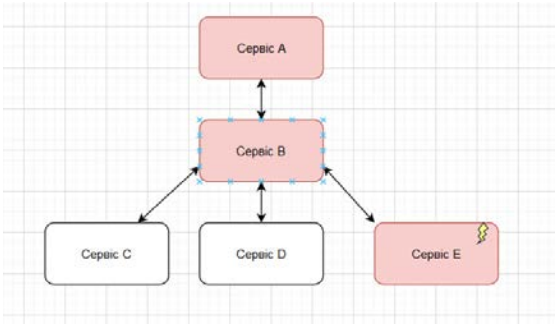

Рис. 6. Поширення аномалії – поширення аномалії з сервісу E вплинуло на поведінку сервісів A та B (заповнені червоним)

Останніми значеннями, що містяться у звіті про аномальні спени, є часові мітки, якими позначені початок і кінець спену. Вони і допоможуть нам віднайти структуру залежностей. Це можна зробити завдяки тому, як працює механізм створення спенів. Кожен із них розпочинає свою дію, лише коли запит доходить до відповідного сервісу, і завершає свою роботу, коли сервіс закінчує опрацювання запиту.

У мікросервісній архітектурі з використанням http-запитів для комунікації ці запити обро-

блюються відповідним класом-контролером (controller). Цей контролер може як опрацювати запит власноруч, так і викликати інші сервіси за допомогою інших http-запитів або ж опрацювати запит в іншому класі того самого сервісу. Незалежно від того, як саме проходить опрацювання запиту, спен уже створений, як тільки запит надійшов до сервісу, і закінчиться, як тільки запит буде оброблено. Завдяки цьому на основі інформації про часові мітки можна робити припущення щодо того, який зі спенів був першим.

Спен створюється, коли запит досягає відповідного сервісу. Ба більше, батьківський процес викликає дочірній процес. А тому і створення дочірнього процесу має бути лише після створення батьківського. Протилежне ствердження є правильним стосовно завершення роботи спену.

Однак, крім цього очікуваного порядку спенів, також було помічено два інші під час спостереження за роботою прототипу. Другий випадок, найімовірніше, є результатом роботи всіх сервісів на тій самій віртуальній машині. Через відсутність затримок через мережу можна було спостерігати спени, у яких і батьківський, і дочірній процеси мали ідентичну часову мітку початку роботи. У реальній розподіленій системі, де наявні затримки в мережі, такі випадки не повинні виникати.

Третій випадок є дуже контрінтуїтивним, якщо виходити з припущень першого випадку. Адже в процесі першого випадку є неможливим, щоб батьківський процес завершувався до того, як завершується дочірній. Однак такі випадки мали місце.

З усіма цими знаннями про структуру відносин процесів / спенів тепер можна розробити алгоритм, здатний реконструювати структуру трейсу, навіть не маючи інформації про всі спени в ньому. Для цього потрібно побудувати додаткову структуру даних, яку можна назвати просто Anomaly і яка показана на рис. 7. Вона слугує сховищем для всієї інформації щодо аномального спену та надає можливість реконструювати структуру трейсу.

Оскільки ця структура даних призначена для реструктуризації структури залежностей трейсу, підготовлені аномальні спени мають бути згруповані за їх ідентифікатором трейсу. Після чого вони мають буди впорядковані за часом початку їх роботи, що може бути зроблено завдяки сортуванню за часовою міткою початку, а потім за часовою міткою завершення роботи. Однак, якщо згадати випадок, коли вони можуть бути ідентичні на деяких спенах, це може призвести до неправильного порядку. Проте у випадку http-запитів ідентифікатор спену контролера закінчується префіксом mvc, завдяки чому їх можна відсортувати, навіть коли часові мітки є ідентичними. Але це є

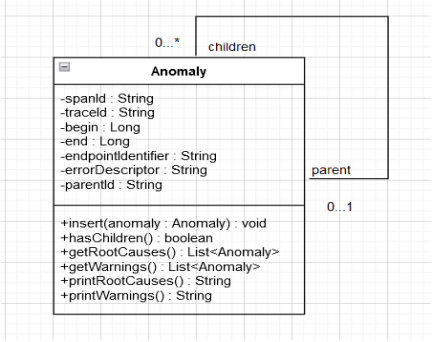

Рис. 7. UML схема класу Anomaly

слабким місцем цього алгоритму, тож покращення методу ідентифікації порядку спенів має бути пріоритетом у майбутньому.

Після такого сортування аномальні спени конвертуються в об'єкти типу нової структури даних Anomaly. Такий об'єкт, який розпочав роботу перед іншими, є головним, і всі інші об'єкти стають лише його частиною за допомогою методу insert, імплементацію якого можна побачити нижче на рисунку 8. Ця функція є рекурсивною, завдяки чому об'єкти поміщаються один в одного, згідно з їх старшинством: від найстаршого до наймолодшого.

Після того як таке «дерево» аномалій створено, метод getRootCauses() поверне список аномальних об'єктів, що є вихідними вузлами «дерева».

```
def insert(anomaly: Anomaly): Unit = {
    var inserted = false
    for (child <- children) {
        if (anomaly.parentId==this.spanId
        ||(anomaly.begin > child.begin && anomaly.end <= child.end)
        ||anomaly.begin >= child.begin && anomaly.end < child.end) {
            if (!inserted) {
            child.insert(anomaly)
            inserted = true}
        }
    }
    if (!inserted) {
        children.append(anomaly)
        anomaly.parent = this
    }
}
```

Рис. 8. Insert метод для об'єкта типу Anomaly

Це є аномалії, чиї відповідні сервіси, найімовірніше, і є першопричиною аномалії. Інший же метод, getWarnings(), повертає список сервісів, які є на шляху до вузла-першопричини, які також були прозвітовані як аномальні і, вірогідно, їх аномальність була викликана лише поширенням аномалії-першопричини. Ідентифікатор сервісів-першопричин та відповідні дескриптори зберігаються для подальшого кроку – візуалізації.

Відсіювання сервісів, що не є істиною першопричиною. Під час візуалізації виконується ще один крок, який покращує надані результати аналізу першопричини аномалії. Оскільки весь процес базується на обробці мініпакетів даних, що містять усі спени одного трейсу, може бути, що інформація про

деякі з спенів відсутня, тому що вони були оброблені разом з іншим мініпакетом даних. Якщо же ці, відсутні, спени містять інформацію про першопричини, інший (неправильний) ідентифікатор сервісу може бути визначений як першопричина, що в іншому випадку був би лише результатом поширення аномалії-першопричини. А тому всі такі аномалії, про які інформація міститься в мініпакеті з уже визначеною першопричиною, мають бути деприорітизовані. Що означає, що вони мають займати нижчу позицію у звіті про першопричину аномалії.

Як можна побачити за допомогою простої візуалізації на рисунку 9, прототип здатен звітувати про виникнення помилок або ж виключень у роботі сервісу. Якщо звітується виключення, то надається назва виключення, у випадку ж помилки надається код помилки. Також в інформацію звіту додається назва сервісу, де ця помилка виникла.

На рисунку 10 показано приклад того, як прототип здатен виявляти, коли сервіс потребує занадто багато часу для обробки запиту. У прикладі візуалізації звіт сортується від найдовшого часу відгуку до найменшого. Також згадується назва сервісу та URI-адреса запиту.

Дискусія. Розроблений прототип демонструє подібність із підходом, запропонованим T. Pitakrat та співавторами [7], однак цей прототип відрізняється тим, що фокусується не лише на прогнозуванні збоїв, але й на виявленні різних типів аномалій та визначенні їх першопричин. Порівняно з позицією T. M. Ahmed та інших [8] авторський прототип має перевагу в тому, що він не потребує встановлення додаткових датчиків чи модифікації наявної інфраструктури.

M. Mdini та співавтори [9] зосередилися на моніторингу систем моніторингу мережі з використанням розпізнавання патернів для виявлення аномалій. Авторський прототип доповнює цей підхід, пропонуючи не тільки виявлення аномалій, але й автоматичне визначення їх першопричин, що значно скорочує час на діагностику й усунення проблем у складних розподілених системах.

Особливо цікавим є порівняння авторського підходу з роботою M. Zasadzinski, V. Muntés-Mulero та M. S. Simo [10], де запропоновано підхід до аналізу першопричин на основі акторів у розподіленому середовищі. Авторський прототип використовує інноваційний підхід до реконструкції структури залежностей аномалії на основі часових міток та ідентифікаторів трейсів, що дає змогу точно визначати першопричини проблем навіть за відсутності повної інформації про всі спени.

Загалом розроблений прототип додатка виявлення аномалій шляхом відстеження подій у розподілених комп'ютерних системах являє собою інтеграцію різних підходів, описаних у проаналізованій літературі, з важливим рядом удосконалень.

Висновки. Дійсно, аномалії можливо виявляти на основі лише інформації, що надається розподіленою системою відслідковування подій. Реалізовані алгоритми мали змогу виявляти всі цільові аномалії, а саме помилки, порушення попередньо визначених порогових значень і занадто довгий час відгуку сервісу порівняно з очікуваним.

Logback Error Messages	Logback Error Messages	0	2	configurer
SLF4J Error Messages	SLF4J Error Messages	0	8	sms-consumer
BadRequest	org.springframework.web.client.HttpClientErrorException$BadRequest	0	3	sms-consumer
Logback Error Messages	Logback Error Messages	0	3	sms-consumer
HTTP Error Code : 400	HTTP error code : 400	0	18	producer
HTTP Error Code : 400	HTTP error code : 400	0	21	api-gateway
HTTP Error Code : 400	HTTP error code : 400	0	15	in-app-consumer

Рис. 9. Виявлення помилок і виключень за допомогою прототипу

12/19/22 2:13:52 PM	18,281	/status-provider/api/v2/notifications/items/...	/status-provider/api/v2/notification...	api-gateway	api-gateway-84-6qbw2
12/20/22 5:36:18 PM	15,786	/status-provider/api/v2/notifications/items/...	/status-provider/api/v2/notification...	api-gateway	api-gateway-84-qslbb
12/19/22 4:32:50 PM	14,351	/status-provider/api/v2/notifications/items/...	/status-provider/api/v2/notification...	api-gateway	api-gateway-84-qslbb
12/20/22 11:03:36 PM	6,463	-	/producer/v4/notifications/sms	matcher	matcher-29-bxpvr
12/19/22 11:32:00 AM	5,308	/status-provider/api/v2/notifications/items/...	/status-provider/api/v2/notification...	api-gateway	api-gateway-84-qslbb
12/19/22 11:44:43 AM	5,247	-	/producer/v4/notifications	matcher	matcher-29-bxpvr

Рис. 10. Виявлення занадто повільного часу відгуку сервісу за допомогою прототипу

Ба більше, прототип здатен реконструювати структуру залежностей аномалії, виходячи лише з інформації, яка знову ж таки, надається розподіленою системою відслідковування подій. Це дає можливість уникати помилкових звітів про аномальну поведінку та виявляти сервіс-першопричину аномальної поведінки.

Сама архітектура прототипу розроблена таким чином, щоб його можна було легко змінювати в майбутньому й адаптувати до нових викликів. Розподілена система відслідковування подій, модуль виявлення аномалій і модуль аналізу першопричину спілкуються між собою за допомогою системи асинхронної коммунікації Apache Kafka. Це означає, що доки формат повідомлень між сервісами не змінюється, кожен із модулів можна замінити. Крім того, структура також дає змогу запускати декілька алгоритмів паралельно один до одного. Отже, у випадку виявлення нового типу аномалій можна написати новий тип алгоритму, а потім підключити його до решти вже існуючих. Або ж, якщо існуючий алгоритм не працює так, як потрібно, його можна видалити не впливаючи на робочі процеси інших алгоритмів.

І нарешті, весь прототип та інструментарій побудовані на фреймворках з відкритим кодом і з врахуванням можливого збільшення масштабу системи. Особливо Apache Spark та Apache Kafka за своїм дизайном ідеально підходять для використання в контексті розподілених комп'ютерних систем, що дає змогу в разі потреби легко збільшувати обчислювальну потужність системи.

ЛІТЕРАТУРА

1. Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. Performance Engineering for Microservices: Research Challenges and Directions. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion). Association for Computing Machinery, New York, NY, USA, 2017. P. 223–226. https://doi.org/10.1145/3053600.3053653.
2. André de Camargo, Ivan Salvadori, Ronaldo dos Santos Mello, and Frank Siqueira. An architecture to automate performance tests on microservices. In Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services (iiWAS '16). Association for Computing Machinery, New York, NY, USA, 2016. P. 422–429. https://doi.org/10.1145/3011141.3011179.
3. Holger Knoche. Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16). Association for Computing Machinery, New York, NY, USA, 2016. P. 121–124. https://doi.org/10.1145/2851553.2892039.
4. T. Salah, M.J. Zemerly, C.Y. Yeun, M. Al-Qutayri and Y. Al-Hammadi. Performance comparison between container-based and VM-based services. 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Paris, France, 2017. P. 185–190. DOI: 10.1109/ICIN.2017.7899408.
5. T. Salah, M. Jamal Zemerly, Chan Yeob Yeun, M. Al-Qutayri and Y. Al-Hammadi. The evolution of distributed systems towards microservices architecture. 11th International Conference for Internet Technology and Secured Transactions (ICITST), Barcelona, Spain, 2016. P. 318–325. DOI: 10.1109/ICITST.2016.7856721.
6. T. Ueda, T. Nakaike and M. Ohara. Workload characterization for microservices. IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, USA, 2016. P. 1–10. DOI: 10.1109/IISWC.2016.7581269.
7. T. Pitakrat, D. Okanovic, A. Van Hoorn and L. Grunske. An Architecture-Aware Approach to Hierarchical Online Failure Prediction. 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), Venice, Italy, 2016. P. 60–69. DOI: 10.1109/QoSA.2016.16.
8. T.M. Ahmed, C.-P. Bezemer, T.-H. Chen, A.E. Hassan and W. Shang. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 2016. P. 1–12.
9. M. Mdini, A. Blanc, G. Simon, J. Barotin and J. Lecoeuvre. Monitoring the network monitoring system: Anomaly Detection using pattern recognition. IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Lisbon, Portugal, 2017. P. 983–986. DOI: 10.23919/INM.2017.7987418.
10. M. Zasadzinski, V. Muntés-Mulero and M.S. Simo. Actor Based Root Cause Analysis in a Distributed Environment. IEEE/ACM 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), Buenos Aires, Argentina, 2017. P. 14–17. DOI: 10.1109/SEsCPS.2017.3.