Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Удосконалення методів і механізмів керування логуванням у програмних системах
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРІПЗ. 170106.01.05.ПЗ

Виконав студент 2 курсу, група ІПЗм-21-1 _____ С. О. Капітанець
Підпис Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ Г. І. Радельчук
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн. наук, доцент _____ Г. І. Радельчук
Підпис Ініціали, прізвище

До захисту допускаю:
завідувач кафедри інженерії програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

2 грудня 2022 р.

Хмельницький 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
01.09.2022 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Капітанцю Степану Олександровичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Удосконалення методів і механізмів керування логуванням у програмних системах

Керівник проєкту (роботи) Радельчук Галина Іванівна, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.07.2022 р. № 83
2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2022 р.
3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Дослідження предметної області та постановка задачі
2 Концепції, моделі та методи вирішення задачі
3 Технології вирішення задачі
4 Реалізація та тестування програмного засобу
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Гурман І. В., доцент	26.11.22	1.12.22р.
Нормоконтроль	Радельчук Г. І., доцент	26.11.22	1.12.22р.

7. Дата видачі завдання «_01_» __вересня__ 2022 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту	Примітка
1 Вивчення предметної області; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; формування логістичної структури дипломної роботи	01.09-10.09.2021	
2 Робота над розділом 1 дипломної роботи – вивчення літературних та Інтернет-джерел; аналіз відомих моделей, методів та засобів за темою роботи; визначення методологічних підходів до вирішення задачі; висновки до розділу та постановка задач дослідження	11.09-25.09.2021	
3 Робота над розділом 2 дипломної роботи – розробка моделей, методів та алгоритмів вирішення задачі; висновки до розділу	26.09-10.10.2021	
4 Робота над науковими статтями	11.10-30.10.2021	
5 Робота над розділом 3 дипломної роботи – розробка інформаційної технології вирішення задачі (аналіз вимог до програмного засобу та його проектування, аналіз та вибір засобів реалізації програмного засобу тощо); висновки до розділу	11.10-26.10.2021	
6 Робота над розділом 4 дипломної роботи – програмна реалізація спроектованих рішень, результати експериментів та їх аналіз; дослідження ефективності запропонованих рішень; висновки до розділу	27.10-17.11.2021	
7 Попередній захист дипломної роботи	Листопад (згідно графіка)	
8 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків; оформлення пояснювальної записки та графічних матеріалів згідно вимог чинних стандартів	18.11-30.11.2021	
9 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12-04.12.2021	
10 Підготовка до захисту дипломної роботи	з 01.12.2021 р	

Студент _____ С.О. Капітанець
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ _____
Підпис Ініціали, прізвищ

РЕФЕРАТ

Тема дипломної роботи: «Удосконалення методів і механізмів керування логуванням у програмних системах».

Автор роботи: Капітанець Степан Олександрович.

Керівник роботи: Радельчук Галина Іванівна.

Пояснювальна записка: 123 с., 27 рис., 3 табл., 2 дод., 38 джерел.

ЖУРНАЛЮВАННЯ ДАНИХ, ЛОГУВАННЯ, МЕТОДОЛОГІЯ ЖУРНАЛЮВАННЯ, ВИЯВЛЕННЯ АНОМАЛІЙ, ВИЯВЛЕННЯ ПОМИЛОК, АРХІТЕКТУРА СИСТЕМИ ЛОГУВАННЯ ДАНИХ.

Об’єкт дослідження – журналювання даних.

Мета дослідження – розробка та впровадження удосконаленої масштабованої архітектури для проведення аналізу журналів даних у реальному часі у високонавантажених програмних системах.

У процесі дипломного проектування здійснено дослідження існуючих рішень та проблем у сфері логування та аналізу журналів повідомлень. На початковому етапі роботи відбулось ознайомлення із наявними методами та механізмами логування інформації у програмних системах. Були досліджені наукові статті та джерела, існуючі методи та способи організації процесів логування та знаходження аномалій з метою виявлення невирішених проблем. Було проаналізовано передові підходи до аналізу текстових даних що використовуються для виявлення аномалій в логах, визначено їх переваги та недоліки. Нерозв’язані задачі запропоновано вирішити шляхом впровадження нової архітектури, в якій аналіз даних журналу може масштабуватися по горизонталі для швидкої та ефективної роботи з великою кількістю даних журналу.

В ході експерименту було виявлено, що розроблене рішення здатне витримувати високі навантаження, використовуючи при цьому лише обмежені ресурси комп’ютера. Також було доведено, що продуктивність рішення не знижується навіть протягом тривалої обробки великої кількості даних журналу.

Оскільки розроблене рішення не потребує особливих налаштувань, воно

може бути досить легко інтегроване у проект, що дозволить збільшити стабільність його роботи шляхом своєчасного виявлення проблем. Можливість додавання своїх власних правил до запропонованого рішення може зробити даний інструмент набагато більш актуальним як в галузі в цілому, так і для окремо взятих проектів.

Підпис

07.12.22
Дата

ABSTRACT

Master's thesis: «Improvement of logging control methods and mechanisms in software systems».

Author: Kapitanets Stepan.

Head of research: Radelchuk Galina.

Master's thesis consists of: 123 p., 27 pc., 3 tb., 2 add., 38 srs.

DATA LOGGING, LOGGING ANOMALY DETECTION, LOGGING METHODOLOGY, ERROR DETECTION, ARCHITECTURE OF THE DATA LOGGING SYSTEM.

Object of research – data logging.

The purpose of the study is to develop and implement an improved scalable architecture for real-time analysis of data logs in highly loaded software systems.

In the process of diploma design was carried out a study of existing solutions and problems in the field of logging and analysis of message logs. At the initial stage of the work was acquainted with the existing methods and mechanisms of logging information in software systems. Scientific articles and sources, existing methods and ways of organizing logging processes and finding anomalies were studied in order to identify unsolved problems. Advanced approaches to text data analysis used to detect anomalies in logs were analyzed, their advantages and disadvantages were determined. Unsolved problems are proposed to be solved by implementing a new architecture in which the analysis of log data can scale horizontally for fast and efficient work with large amounts of log data.

Understanding the behavior of a framework in its generation environment can be challenging. Commonly, when a framework comes up short, engineers examine the log records to pick up understanding and distinguish the issue. Hence, it is evident that logs play an vital part in investigating and keeping up huge applications, as they capture the occasions giving data approximately the execution of the program.

During the experiment, it was found that the developed solution is able to withstand high loads using only limited computer resources. It was also proved that the performance

of the solution does not decrease even during prolonged processing of a large amount of log data messages.

Since the developed solution does not require special settings, it can be easily integrated into the project, which will increase the stability of its work by timely detection of problems. The ability to add your own rules to the proposed solution can make this tool much more relevant both in the industry as a whole and for individual projects.

Signature

02.12.22

Date

ЗМІСТ

ВСТУП

Моніторинг поведінки програмного забезпечення необхідний при роботі як з малими так і великими програмними системами. Отримана таким чином інформація дозволяє зрозуміти як саме веде себе програма у робочому середовищі. Моніторинг може полягати у відстеженні використання ресурсів системи по типу необхідної кількості оперативної пам’яті, показників завантаженості центрального процесора та пропускної здатності мережі. Моніторинг також може здійснюватися шляхом аналізу ключових показників ефективності програмного забезпечення. Наприклад, такі показники як середня тривалість виконання запитів та їх кількість є типовими для аналізу систем баз даних. Якщо ж таких показників виявляється недостатньо і нам необхідно більше знань про процеси що відбуваються у системі, то чудовим рішенням для цієї проблеми стає використання систем журналювання даних повідомлень.

Зростаючий попит на інструменти, які дають більше представлення про працююче у робочому середовищі програмне забезпечення, підкреслюється ще й тим, що на даний момент на ринку з’являються все нові й нові утиліти, основною метою яких є забезпечення моніторингу поведінки програмних систем за допомогою даних отриманих із логів [1, 2].

Логування – це практика запису подій, яка дозволяє отримати інформацію про процеси, що відбуваються протягом роботи програмної системи [3]. Журнали широко використовуються в якості джерела інформації для аналізу програмного забезпечення, контролю його стабільності та працездатності. У тестовому середовищі розробник має можливість відлагодити програму для того щоби виявити джерело помилки.

Проте в робочому середовищі налагодження неможливе. В таких ситуаціях журнальовані дані можуть сильно допомогти розробникам у виявленні та швидкому виправленні проблем, що виникають із працюючою програмною системою. Таким чином, стає очевидно, що журнали відіграють дуже важливу роль

в процесі налагодження та супроводу великих програмних систем, оскільки вони фіксують події що надають інформацію про процес виконання програми.

По мірі росту та збільшення складності програмного забезпечення збільшується також і об’єм журнальованих даних. Проте, якщо раніше людина могла діагностувати проблеми вручну використовуючи дані журналу (наприклад, фільтруючи їх за допомогою простих ключових слів по типу "error" або "warning"), то для сучасних систем такий підхід перестав бути ефективним. Причиною цього є те, що сучасне програмне забезпечення без проблем може генерувати мільйони логів щоденно. Окрім того, всі ці дані як правило або взагалі не структуровані, або структуровані лише частково. Зрозуміло що аналіз таких великих об’ємів інформації людиною неможливий, а тому автоматизований підхід для вирішення цієї задачі стає необхідністю [4, 5].

Однак навіть перенасиченість логами не гарантує що проблема буде правильно та швидко виявлена. Більшість з наявних механізмів логування даних або взагалі не надають інформації про файл і місце в коді, з якого було створено даний запис, або мають проблеми з продуктивністю за рахунок тих способів, якими дані інструменти отримують цю інформацію. Більше того, серйозні проблеми з продуктивністю можуть виникнути через місце й спосіб яким нові дані вносяться в журнал, та те, в якому вигляді він представлений.

Отже, актуальність теми полягає в тому, що ведення журналів широко застосовується на практиці завдяки своїй простоті та ефективності. Програмні системи ведуть журнали для моніторингу подій та свого поточного стану. Такі журнали в процесі пошуку та усунення несправностей являють собою дуже важливе та цінне джерело інформації для розробників, які можуть вивчати записані логи щоби зрозуміти стан системи, виявляти аномалії та знаходити першопричини їх виникнення.

Проте велика кількість високонавантажених систем в процесі своєї роботи можуть генерувати величезну кількість інформації зі швидкістю в десятки гігабайт на годину. Такий обсяг даних надзвичайно складно обробляти вручну, навіть за допомогою інструментів пошуку та фільтрації, що створює необхідність в розробці

інструменту для автоматичного аналізу даних журналу.

Мета дослідження – створення та впровадження масштабованої архітектури для проведення аналізу журналів даних в реальному часі у високонавантажених програмних системах.

Завдання, які необхідно вирішити для досягнення мети:

– провести аналіз області логування даних;

– дослідити існуючі методи автоматичного аналізу даних журналу;

– удосконалити метод для системи логування, який дозволить обробляти великі обсяги даних одночасно у високонавантажених системах;

– розробити архітектуру на основі удосконаленого методу з можливістю горизонтального масштабування;

– провести тестування та апробацію отриманих результатів.

Об’єкт дослідження – журналювання даних.

Предмет дослідження – методи та механізми збору даних журналів.

Наукова новизна отриманих результатів:

− запропоновано та імплементовано архітектуру для логування даних, яка призначена спеціально для аналізу журналів у високонавантажених програмних системах у режимі реального часу;

− отримала подальший розвиток техніка розподілених обчислень у напрямку її застосування для логування даних, що дозволило отримувати результати у реальному часі або з мінімальними затримками.

Запропонована архітектура може легко масштабуватися у разі необхідності обробки ще більшої кількості даних.

Практична цінність отриманих результатів полягає в успішній реалізації методу, який пропонує масштабовану архітектуру для проведення аналізу журналів даних в реальному часі для високонавантажених систем.

Низькі затримки дозволяють швидко обробляти нові журнальні повідомлення. В свою чергу, це дозволяє своєчасно виявляти проблеми та їх причини, що дає змогу швидко відновити працездатність застосунку в разі виникнення помилок.

В ході тестування було виявлено що розроблене рішення здатне витримувати високі навантаження використовуючи при цьому лише обмежені ресурси комп’ютера. Оскільки розроблене рішення не потребує особливих налаштувань, воно може бути досить легко інтегроване у проект, що дозволить збільшити стабільність його роботи шляхом своєчасного виявлення проблем та аномалій.

За темою дипломної роботи для фахового видання підготовлена стаття «Особливості логування даних та вплив типу додатка на вибір методології логування» (яка прийнята до друку) та доповідь «Керування логуванням у програмних системах: концептуальні засади», яка опублікована у збірнику матеріалів міжнародної наукової конференції.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області, останніх досліджень та джерел

Програмні системи ведуть журнали для моніторингу подій та свого поточного стану. Ведення журналів широко застосовується на практиці завдяки своїй простоті та ефективності. Такі журнали в процесі пошуку та усунення несправностей являють собою дуже важливе та цінне джерело інформації для розробників, які можуть вивчати записані логи щоби зрозуміти стан системи, виявляти аномалії та знаходити першопричини їх виникнення.

Сучасні програмні системи мають тенденцію до масштабування і переходу до розподілених хмарних обчислень. Такі великі системи забезпечують роботу сервісів по типу соціальних мереж, пошукових систем, або електронної комерції. Абсолютна більшість таких систем призначена для роботи в цілодобовому режимі обслуговуючи при цьому мільйони користувачів по всьому світу. Будь-яке відключення (навіть тимчасове) або погіршення якості роботи таких сервісів обходиться дуже дорого, тому можливості своєчасно виявляти будь-які зміни та швидко з'ясовувати першопричину проблеми є надважливими. Проте такі системи в процесі своєї роботи генерують величезну кількість логів зі швидкістю в десятки гігабайт на годину. Такий обсяг даних надзвичайно складно обробити вручну, навіть за допомогою інструментів пошуку та фільтрації. Окрім того, логи зазвичай представляють собою неструктуровані або напівструктуровані стрічки тексту і тому складно піддаються для обробки алгоритмами (якщо не дотримуватись певних правил).

Проте журналювання може використовується не тільки для виявлення аномалій, але і для аудиту та виставлення рахунків також [6, 7]. З іншого боку, всі ці варіанти використання передбачають вільний доступ до всіх відповідних журналів, що може бути проблематично враховуючи розподільний характер сучасних систем.

Логи можуть зберігатися на різних рівнях журналу, щоби можна було відразу визначити важливість повідомлення. Наприклад, логи можуть містити префікси “INFO”, “WARN” чи “ERROR”. Стандарту які саме рівні повинні використовуватись та наскільки логи повинні бути багатослівні немає [8].

Типовий запис в журналі повинен містити мітку часу, рівень повідомлення, та саме повідомлення. Мітка часу як правило представляє собою час у форматі Unix або UTC, що позначає точний час коли даний запис було створено. Це дозволяє відновити правильну послідовність подій.

Неправильний або неточний код журналювання може призвести до плутаниці або й навіть до більш серйозних проблем, таких як відчутне зниження продуктивності чи збій системи [9]. Окрім того, можна також виділити дві основні проблеми, які пов'язані з практикою ведення журналів програмного забезпечення.

1. Відсутність існуючих керівних принципів щодо створення високоякісного коду журналювання. Недавні емпіричні дослідження показують, що існуючих керівництв по веденню журналів для комерційних [10] та систем з відкритим вихідним кодом [11, 12] немає. Розробники пишуть код журналювання виключно на основі своїх власних міркувань. На відміну від решти коду, який може бути перевірений шляхом тестування, перевірити коректність журнальованих даних досить складно. Наприклад, на рисунку 1.1 продемонстрована помилка, знайдена на одному з проектів.

2. Складність підтримки та розвитку коду логування. Оскільки код журналювання сплутується з рештою коду, його дуже складно постійно оновлювати по ходу розвитку програмної системи.

DFSClient.cs

V 4078:
```
LOG.warn("Failed to renew lease for " + clientName + " for "+ (elapsed / 1000)+ "
seconds (>= soft-limit ="+ (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000)+ " seconds.)
"+ "Closing all files being written ...",e)
```

V 5956:
```
LOG.warn("Failed to renew lease for " + clientName + " for "+ (elapsed / 1000)+ "
seconds (>= hard-limit ="+ (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000)+ " seconds.)
"+ "Closing all files being written ...",e)
```

Рисунок 1.1 – Приклад помилки у коді журналювання даних

Найпоширеніші помилки з якими стикаються розробники в процесі логування наступні:

– явне приведення типів або відсутність перевірки на null у динамічних логах, що може призвести до помилки в процесі виконання програми;

– неправильний рівень повідомлення (наприклад, DEBUG замість INFO, або WARN замість ERROR);

– некоректний вивід даних у випадку коли у повідомлення вставляється об’єкт в якого не визначений формат, який зручний для читання людиною (наприклад, якщо у нього не перевизначений метод ToString());

– повтори коду для логування з однаковими повідомленнями;

– занадто довгий код для логування.

Як вже було зазначено раніше, розробники зазвичай покладаються на свої власні відчуття у питаннях організації процесів логування у програмних системах. В загальному, можна виділити три таких питання:

– питання того, де саме потрібно логувати;

– питання того, що саме записувати в журнал;

– питання того, як саме потрібно логувати.

Питання того, де саме потрібно логувати, полягає у визначенні підходящих місць для журналювання інформації. Логування може бути виконано у різноманітних місцях програмного коду. До прикладу, це може бути всередині блоків умов чи всередині блоків try і catch, щоб отримати представлення про поведінку системи під час її роботи. Однак надмірне логування може призвести до додаткових витрат ресурсів [13]. Тому розробникам необхідно бути вибірковими у плані визначення основних точок для журналювання і ставити їх тільки там, де це матиме найбільшу користь.

Основна ідея питання того, що саме записувати в журнал, полягає у наданні достатньої інформації в повідомленні для логу. Статичні тексти повинні надавати короткий опис контексту виконання, а динамічний вміст має вказувати на поточний стан виконання. При написанні статичного повідомлення його текст має бути

чітким і легким для розуміння, а динамічні фрагменти повинні бути послідовними та актуальними [14].

Питання того, як саме потрібно логувати, полягає у необхідності розробки і підтримки високоякісного коду для журналювання. Ведення журналу – це наскрізна проблема, оскільки код для логування розкиданий по всій програмній системі та заплутаний у кодах функцій. Завжди слід мати на увазі і враховувати, що розробляти та підтримувати хороший код для журналювання по мірі росту проекту буде тільки складніше, а тому найкращі практики для цього треба інтегрувати ще на початкових стадіях розробки системи.

Однак, правильно складених повідомлень іноді може бути недостатньо щоб однозначено ідентифікувати точне місце в коді з якого його було створено (як мінімум – щоби зробити це швидко, а як максимум – в коді проекту може бути декілька місць, де викликається функція, яка генерує однакові логи). Наявність даних про трасування стеку значно спрощує налагодження, а тому їх також необхідно включати до логів [15, 16].

Отже, першою ціллю даної роботи буде покращення механізмів що використовуються для створення та збереження журнальних даних з метою пришвидшення швидкодії програмної системи. Другою ціллю буде розробка механізму для автоматичного виявлення аномалій серед великого об’єму моніторингових даних.

Тепер проаналізуємо як та в якій мірі вирішуються ці завдання серед існуючих рішень.

1.2 Порівняльний аналіз переваг та недоліків існуючих рішень

Всі програми для аналізу логів обробляють події, що відповідають записам, які містяться в журналах, а точніше, викликаним операторам у вихідному коді. Однак, простежити у зворотньому порядку, тобто від даних журналу до його оператора у вихідному коді, є нетривіальним завданням. По-перше, створення події

під час виконання впливає на продуктивність системи. Наприклад, такі фреймворки, як Log4j, дозволяють переглядати ім'я класу та номер рядка з якого було створено даний запис. Однак, збір цієї інформації при кожному записі в журнал супроводжується втратою продуктивності. Це відбувається тому що за лаштунками Log4j отримавши рядок оператора журналу, генерує виключення і захоплює згенероване ним трасування стека [17]. Розробники Log4j експериментували з іншими альтернативами, але покищо даний спосіб виявився найбільш ефективним. По-друге, повідомлення що зберігаються у журналі складаються з тексту у довільній формі.

Щодо вирішення проблеми аналізу журналів даних було запропоновано декілька основних стратегій. Найпростіший спосіб полягає у використанні регулярних виразів, запитів та фільтрів. Даний метод не практичний та не масштабується. Він дозволяє виявляти лише одиночні збої, тоді як ланцюжки відмов, які викликані декількома взаємопов’язаними компонентами, залишаться непоміченими. Журнал часто забруднюється не пов’язаними одне з одним трасуваннями стеку, що робить дослідження ще складнішим та заплутанішим. Окрім того, дана стратегія забирає багато часу та не може бути автоматизованою, що робить її непрактичною для реального світу.

Більш інноваційні та масштабовані методи намагаються поєднати аналіз журналів з алгоритмами машинного навчання. Враховуючи високу доступність даних, алгоритм машинного навчання ідеально підходить для обробки і вилучення закономірностей і прогнозів з логів. Виявлення аномалій це область, пов'язана з аналізом журнальованих даних, в якій вже широко використовується машинне навчання. Основна проблема що виникає з подібними методологіями, які застосовуються для аналізу логів, полягає в тому, що задача перетворення вільнотекстових повідомлень у значущі ознаки є нетривіальною [20].

Ще однією життєздатною стратегією є статичний програмний аналіз. Алгоритми статичного аналізу вже добре відпрацьовані та ефективні. Вони здатні сканувати програму, щоб знайти шляхи, які можуть призвести до помилок. Основна проблема полягає в тому, що статичний аналіз обмежений розміром і

складністю цільової системи.

Для вирішення проблеми аналізу журнальних даних також можна використовувати аналітичні моделі. Модель будується та відточується для конкретної системи. Експерти вручну визначають залежності, метрики та взаємозв'язки для розробки стратегії прогнозування. Основним недоліком такого підходу є те, що великомасштабні системи є занадто складними і часто змінюються, що може зробити модель неефективною вже через короткий проміжок часу.

Далі буде представлено кородкий огляд найбільш значущих інструментів для аналізу повідомлень журналів.

Logstash – вільний та відкритий серверний конвеєр для обробки даних. Logstash може поглинати величезні обсяги даних з різних джерел, розбирати їх, а потім відправляти в будь-яке вихідне сховище. Logstash розроблений компанією Elasticsearch B.V. і є частиною стека ELK. Він створений спеціально для збору, аналізу та перетворення логів [18].

Splunk – це платформа для розширеного доступу до даних, потужної аналітики та автоматизації. Інструмент надає безліч корисних функцій, які оптимізують управління та аналіз логів.

LogDNA – це високомасштабоване рішення для управління журналами, яке індексує, агрегує та аналізує дані журналів. LogDNA може поглинати величезну кількість даних з різних джерел. Він також має повнофункціональний веб-інтерфейс, а також різні рішення для пошуку, синтаксичного аналізу, побудови графіків та оповіщення про дані журналів [19].

LogZ використовує стек ELK (Elasticsearch, Logstash, Kibana) всередині для обробки логів. LogZ використовує рішення з відкритим вихідним кодом для виконання аналізу логів, моніторингу інфраструктури та розподіленого трасування.

GrayLog – це комплексне рішення для управління в IT. Воно надає функціонал для управління безпекою та інфраструктурою, об'єднання, збагачення, кореляції, запитів і візуалізації всіх даних журналів в одному місці.

Тепер складемо список з плюсів та мінусів для кожного з перелічених

інструментів. Ця інформація ґрунтується на думках спільноти щодо кожного з інструментів та на порівнянні між кожним з цих рішень.

Порівняльню характеристику існуючих програмних рішень продемонстровано на таблиці 1.1.

Таблиця 1.1 – Порівняння найпопулярніших інструментів для аналізу логів

Інструмент	Плюси	Мінуси
Splunk	- найпопулярніше рішення - багато плагінів - багато інтегрованих рішень для логуваннння - хмарне рішення	- дорогий - із закритим кодом - складна та крута крива навчання - не дуже підходить для аналізу логів
Logstash	- створений спеціально для аналізу логів - гнучкий та легкий в налаштуванні - може бути запущеним як локально так і у хмарі - велика екосистема з плагінів - безкоштовний та з відкритим кодом - може приймати та виводити інформацію з та в різні джерела	- лише для аналізу - немає інформаційної веб панелі - пов’язаний зі стеком ELK - високе споживання ресурсів - на практиці часто може мати недостатню продуктивність - при навантаженні ядра на 100%відбувається збій у відправці даних
LogDNA	- легке налаштування та інтеграція - націлений на інтеграцію з інструментами DevOps - потужний пошук та сповіщення - агрегація логів та керування подіями	- не безкоштовний - користувачі повідомляють про періодичні затримки - не користується популярністю та не розвивається спільнотою
LogZ	- створений спеціально для моніторингу логів - утиліти інтегровані у хмару	- заснований на ELK - не безкоштовний - не користується популярністю
Graylog	- хмарне рішення - хороша документація - легка мова запитів	- не безкоштовний - складно писати комплексні запити

Для того, щоб краще зрозуміти зручність використання та популярність вищезгаданих інструментів було підраховано кількість трафіку в Google, що припадає на них. Результати продемонстровано на рисунку 1.2.

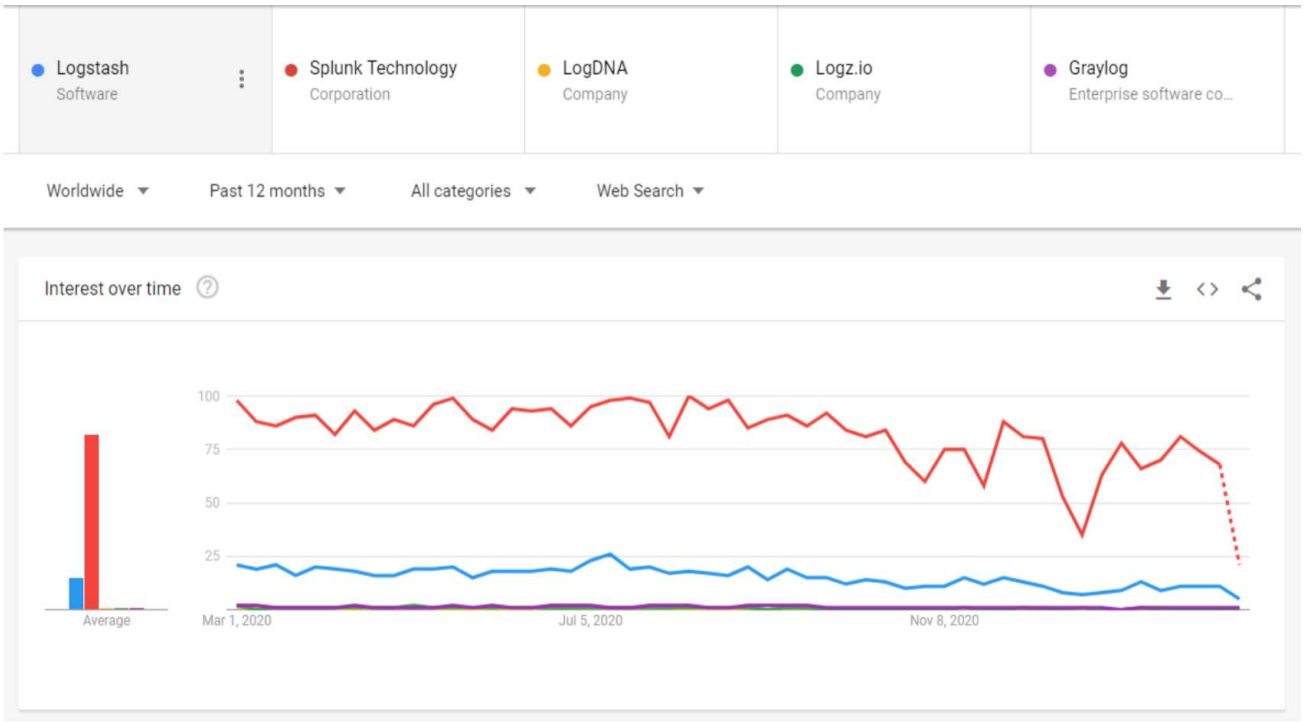


Рисунок 1.2 – Популярність інструментів для аналізу логів

Як видно з рисунка 1.2, у трафіку на Google за останні 12 місяців домінують Splunk та LogStash. Splunk – це корпоративне рішення, в той час як LogStash – це програмне забезпечення з відкритим вихідним кодом. Можна сказати, що популярність LogStash досить стабільна протягом всього часового інтервалу, в той час як у Splunk Technologies спостерігається тенденція до зниження.

1.3 Висновки. Постановка задачі

Логування та аналіз журналів повідомлень були визначені як важливий та потужний інструмент у розробці та підтримці програмного забезпечення. Вони являють собою головне та дуже цінне джерело інформації для розробників після запуску продукту в робоче середовище і дозволяють зрозуміти стан системи,

виявляти аномалії та знаходити першопричини їх виникнення.

Добре організований механізм логування разом з дотриманням основних правил їх оформлення та інструментом для аналізу аномалій, не тільки не нашкодить продуктивності системи, а й надасть додаткові бонуси у вигляді можливості моніторингу системи в реальному часі та стабільнішої роботи всього застосунку за рахунок своєчасного виявлення проблем.

Дослідження, проведені в ході аналізу предметної області, показують що жоден з наявних механізмів логування не здатен надавати інформації про трасування стеку не вдаючись до сумнівних «костилів», які ще й мають чималий вплив на продуктивність всієї системи.

Вивчення інструментів для аналізу логів показало що LogStash є найпопулярнішим (і, до того ж, безкоштовним), не дивлячись на загальні проблеми з продуктивністю та надійністю (можливе здвоєння показників і лавиноподібний ріст навантаження на сховище при спробах повторної відправки даних при високому завантаженні системи). Окрім того, жоден із розглянутих засобів не дозволяв проводити аналіз даних журналу в реальному часі, або дозволяв робити це швидко лише при невисоких навантаженнях.

Таким чином, було встановлено що наявні механізми для логування та аналізу журнальних повідомлень все ще недосконалі і потребують покращень для збільшення швидкодії та інформативності, особливо в контексті роботи із високонавантаженими системами.

2 КОНЦЕПЦІЇ, МОДЕЛІ ТА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ

2.1 Загальний огляд методів для парсингу логів

Типове повідомлення журналу містить мітку часу, рівень докладності (наприклад, WARN) і необроблений вільний текстовий вміст для зручності та гнучкості. Цей необроблений, неструктурований вміст повідомлення дає змогу розробникам пояснити, що сталося на певному шляху виконання, а системним операторам ця інформація може стати в нагоді під час діагностики проблем. Безумовно, вільний текст корисний для людей, але машини не мають ефективного способу обробки природної мови. Відсутність структури є проблемою для автоматизованого аналізу журналу з використанням методів інтелектуального аналізу даних для таких завдань, як виявлення збоїв, виявлення аномалій і прогнозування збоїв. Крім того, вільний текст у повідомленнях журналу передбачає майже нескінченну кількість можливих повідомлень. Це робить пошук закономірностей і виявлення аномалій дуже складним завданням. Синтаксичний аналіз журналу можна використовувати для спрощення кількості шаблонів для розгляду та підвищення ефективності аналізу журналу.

Синтаксичний аналіз журналу – це перетворення необроблених неструктурованих журналів у певні події, які представляють точку створення логу в програмі. Зазвичай необроблений вміст складається з постійної частини та змінної частини. Постійна частина — це фіксований звичайний текст із оператора журналу, який залишається незмінним для кожного повідомлення журналу, що відповідає цьому оператору. Змінна частина містить інформацію про час виконання, таку як значення параметрів. Метою аналізу журналу є розділення постійної та змінної частини повідомлення журналу. Його результат зазвичай називають шаблоном повідомлення або подією журналу, що представляє повідомлення журналу, надруковані тим самим оператором журналу у вихідному коді. На рисунку 2.1 показано повідомлення журналу та відповідний йому шаблон.

```
Authentication failed for user=alice on host 172.26.140.21 - attempt 3
Authentication failed for user=* on host * - attempt *
```

Рисунок 2.1 – Повідомлення журналу та відповідний йому шаблон

Подібно до того, як програми розвиваються з часом, так само розвиваються шаблони журналів у вихідному коді. Шаблони журналів змінюються та додаються нові, що вважається перешкодою для методів аналізу журналів. Повторне застосування аналізу журналу після кожного релізу може бути неможливим і вплинути на аналіз журналу через різні вхідні дані. Щоб вирішити цю проблему, синтаксичний аналізатор журналу повинен мати змогу працювати зі змінами в шаблонах, автоматично виявляючи їх.

Характеризується чотири різні групи методів розбору журналів: евристичні методи, методи кластеризації, методи на основі вихідного коду та інші методи. Ці групи описуються детальніше у наступних підпунктах.

2.1.1 Евристичні методи аналізу журналів даних

Евристика – це техніка, яка використовується для отримання наближеного рішення, якщо точне рішення неможливо знайти за прийнятний час. Приблизне рішення може бути не найкращим рішенням, але може бути цілком адекватним. Для підвищення ефективності евристики можна використовувати в поєднанні з іншими техніками. Часто це правила, засновані на емпіричних даних або теорії. Евристика, що використовується в аналізі журналу, враховує семантику повідомлень журналу. Наприклад, числові значення менш імовірно належать до константної частини повідомлення журналу. Зазвичай вони представляють такі змінні, як ідентифікатори процесу, значення параметрів та IP-адреси.

Окрім числових значень, повідомлення журналу також зазвичай містять спеціальні символи, які використовуються для представлення змінних. Ці спеціальні символи включають двокрапку, дужки та знаки рівності. У повідомленні

журналу «Недійсна сума для валюти USD із сумою=500» знайдені значення змінної будуть «USD» і «500». Застосування більш розширених евристичних методів дозволяє досягти точності 95%. Іншими словами, можна виявити 95% шаблонів повідомлень, використовуючи лише евристику.

Очевидним недоліком використання числової евристики є те, що у логах можуть використовуватись постійні значення, які є числами. Важливі шаблони журналів можуть бути пропущені аналізаторами журналів, які використовують цю евристику. Крім того, семантична евристика змінюється залежно від випадку, тому для налаштування аналізатора журналу потрібна людина зі знанням конкретної програми. Натомість, результати є багатообіцяючими, а підхід є недорогим з точки зору обчислень. Однак, евристика не повинна використовуватися як єдиний механізм аналізу журналів. Її можна використовувати як етап попередньої обробки для більш обчислювально дорогого методу аналізу.

2.1.2 Методи кластеризації

Кластеризація даних – це техніка інтелектуального аналізу даних і машинного навчання, яка групує сутності в кластери. Кожен елемент у групі схожий на інші елементи в цій групі, але відрізняється від елементів в інших групах. Методи кластеризації зазвичай використовуються для класифікації та інтерпретації великих наборів даних. У аналізі журналів кластеризація може бути корисним першим кроком для зменшення кількості елементів, з якими потрібно працювати. Подальший аналіз все ще може бути складним в плані обчислень, але практично можливим через зменшення розміру вхідних даних. Методи кластеризації, як правило, використовують той факт, що слова, які часто зустрічаються, з більшою ймовірністю збігаються. Крім того, ці методи використовують визначене користувачем порогове значення для визначення гранулярності кластерів. Набагато простіший підхід полягає у використанні евристики для визначення змінних. Групування абстрактних повідомлень

відбувається на основі кількості постійних слів і кількості змінних. Наприклад, усі повідомлення, що містять 6 постійних слів і 3 змінні, групуються разом.

2.1.3 Методи засновані на вихідному коді

Замість того, щоб аналізувати повідомлення журналу для отримання набору шаблонів повідомлень, можна проаналізувати вихідний код для досягнення тієї ж мети. При використанні стороннього програмного забезпечення, наприклад, вихідний код може бути недоступним, тому для отримання шаблонів повідомлень потрібно буде використовувати інші методи. Перевага синтаксичного аналізу вихідного коду полягає в тому, що він легкий та швидкий. Крім того, він може забезпечити більш високий рівень точності.

Наприклад, повідомлення, які трапляються рідко, все одно перетворюються на шаблон, тоді як евристика журналу або алгоритми кластеризації можуть їх пропустити. Однак аналіз вихідного коду для аналізу журналу не такий простий, як здається. Для прикладу розглянемо оператор журналу зображений на рисунку 2.2.

```
log.info("Machine " + m + " has state " + STATE);
```

Рисунок 2.2 – Команда для логування на мові Java

При розборі цього твердження виникає декілька труднощів. Синтаксичний аналізатор повинен знати, яка бібліотека журналу використовується та які методи він використовує для журналювання, наприклад, log.info(). У цьому випадку STATE є змінною, однак m може бути екземпляром класу, який перевизначає метод toString() за замовчуванням. Сам цей метод може повертати постійну частину, змішану зі змінними, які важче розібрати. Остання проблема зазвичай притаманна для об’єктно-орієнтованих мов.

Запропоновані парсери нелегко застосувати до широкого діапазону програм, на відміну від різних алгоритмів, заснованих на кластеризації. Це пов’язано з

відмінностями в мовах програмування та бібліотеках журналів. Щоб підготувати синтаксичний аналізатор до різних додатків, необхідно спершу його кастомізувати.

2.1.4 Інші методи парсингу логів

Окрім евристики, кластеризації та методів на основі вихідного коду для розбору журналів, також існують методи, які не належать до цих груп. Вони будуть описані у цьому підпункті.

Метрика відстані редагування, також звана метрикою відстані Левенштейна, є мірою, яка використовується для кількісного визначення подібності між двома рядками. Ця відстань представлена кількістю операцій, необхідних для перетворення одного рядка в інший. Операції вставки, видалення та заміни впливають на відстань редагування.

Під час аналізу журналу ця метрика використовується для визначення того, чи можуть два повідомлення журналу належати одному шаблону повідомлення. Слова в цьому випадку вважаються лексемами. Salfner і Tschirpke використовують метрику відстані редагування після використання евристики для кожної пари повідомлень журналу, щоб підвищити ефективність аналізатора журналу. Їм вдалося зменшити кількість повідомлень на 99% за допомогою евристики та збільшити цей відсоток до 99,92% за допомогою метрики відстані редагування.

Метрика відстані редагування має теоретичну складність часу виконання $O(n^2)$ і тому не підходить для обробки великої кількості журнальних даних. Обмежений набір повідомлень журналу, навчальний набір, можна використовувати для скорочення часу, необхідного для пошуку подій журналу. Однак повідомлення, які з’являються рідше, мають більший шанс бути неправильно згрупованими або навіть не бути виявленими. Техніка може бути придатною після можливого першого етапу кластеризації, який значно зменшує кількість елементів, які слід враховувати.

Існує широкий спектр методів розбору журналів. Від простих евристик до

складних алгоритмів кластеризації. Часто використовується не одна техніка, а поєднання кількох підходів. Те, який саме набір методів підходить, залежить від типу логів і варіанту використання. Найефективніший спосіб розбору журналу базується на аналізі вихідного коду.

2.2 Структури даних

Кластеризація журнальних повідомлень може бути використана для об’єднання корелюючих журнальних повідомлень разом як додатковий рівень абстракції. Повідомлення журналу, які корелюють одне з одним, імовірно, вказують на ту саму подію, що відбувається в коді. Тому, якщо ці повідомлення об’єднати, це може надати нову інформацію під час їхнього спільного аналізу. Застосовуються методи кластеризації, наприклад використання підрахунку частоти слів у даних журналу або перетворення даних журналу в евристичні вектори ознак. Коли справа доходить до пошуку послідовностей, приєднаних до події з часовими ознаками, використовується побудова графа.

Побудова графа для повідомлень журналу виконується для створення зв’язку між кількома повідомленнями журналу. Це надає додатковий вимір для аналізу журналу, з якого можна витягти функції. Маючи побудований граф, можна відкрити широкий спектр алгоритмів, які можна використовувати для аналізу журналів. Наприклад, Sun пропонують техніку, за якої CFG будується після використання підходу аналізу журналу з деревом фіксованої глибини [21, 22]. CFG – це зважений за часом граф недоліків керування, який будується для того, щоб побачити, які шаблони журналу слідують один за одним і в якому інтервалі часу це відбувається. Використовується також інший підхід до побудови графа; за яким повідомлення журналу пов’язані не повністю за тимчасовою ознакою, а за співпадаючими подіями [23]. Спершу необхідно представити події та їхні зв’язки у моделі. Далі потрібен метод для виділення схожих подій у набір шаблонів. Граф у процесі буде побудований таким чином, що події, які відбуваються одна за одною,

представлені з меншою відстанню на графіку. Обидва вище описані рішення зосереджені на пошуку подібності між повідомленнями журналу. Проте вони досить складні в плані обчислень, особливо у випадку якщо їх будуть застосовувати на великих обсягах даних. Тому для обробки великої кількості повідомлень журналу у пропонованому рішенні відповідальність за групування логів за ознаками буде перекладено на програміста, який пише код для логування. До кожного лога буде додано статичний ідентифікатор ознаки та трасування, який дозволить визначати логи із однієї категорії та місце де саме в коді знаходиться дане повідомлення.

2.3 Пасивне навчання

Під пасивним навчанням мається на увазі побудова графових моделей поведінки системи [24]. В нашому випадку це буде зроблено з даних журналу шляхом структурування даних журналу в трасування та моделювання їх у вигляді графу. Отримані графи потім можна використати для подальшого аналізу. Наприклад, для автоматизації виявлення аномалій або вимірювання ефективності. Пасивне навчання відрізняється від активного тим, що активне навчання означає, що системна модель активно запитується, щоб отримати відповіді на різні вхідні дані. Пасивне навчання ґрунтується на отриманих спостереженнях і може порівнюватися з нещодавно отриманими спостереженнями.

Робота Wieman [25] демонструє три різні інструменти, які входять до кола інтересів даного дослідження. По-перше, обговорюється інструмент Synoptic [26], який виводить діаграму стану з файлів журналу для цілей налагодження. Далі в роботі InvariMint [27] обговорюються моделі що журналюють лінії на ребрах, які більше схожі на дерево префіксів. І останній інструмент – DFASAT [28]. Він також використовує префіксну деревоподібну структуру для моделювання ознак журналу. Спільним для всього вищесказаного є те, що моделі виводяться з історичних даних журналу, які можна обробити пізніше та спробувати об’єднати

ознаки на графі, зберігаючи на ньому унікальні шляхи. Ці роботи показують, що структурування даних журналу на графі можна ефективно використовувати для аналізу, а пошук способу моделювання графу в масштабований спосіб може принести для галузі ще більше користі.

2.4 Розподілені системи

Існують уже добре відомі технології, які широко використовуються в галузі логування та аналізу даних журналу. Щоб працювати з великими обсягами даних, можна використовувати кілька видів існуючих технологій. Zhengmin та ін. надають, наприклад, детальний опис масштабованої, високопродуктивної розподіленої системи обробки потоку журналів. Він відразу підсумовує багато найбільш використовуваних технологій які вже були згадані. Спільним знаменником тут є особливо той факт, що для великих обсягів журнальних даних необхідно використовувати горизонтально масштабоване та продуктивне рішення. Іншим загальним фактором є те, що програмне забезпечення від Apache широко використовується в дослідницькому середовищі.

Робота з великими обсягами журнальних даних означає в тому числі й ефективне використання ресурсів, особливо якщо метою є обробка журналів у режимі реального часу, коли комп'ютерна система запущена та працює. У літературі можна знайти чітке розмежування між обробкою в реальному часі та постобробкою. Головним чином через те, що деякі алгоритми важчі для виконання, ніж інші. Як уже згадувалося, алгоритми можуть потребувати ресурсів, і без необхідних запобіжних заходів це може нашкодити до того добре працюючій комп'ютерній системі.

Приклад масштабованої високопродуктивної розподіленої системи обробки потоку журналів уже згадувався, але вона не була здатна виявляти аномалії. У роботі [29] демонструється високоефективна система виявлення аномалій на основі Apache Hadoop. Автор використовує машинне навчання для створення часових

рядів (алгоритм Microsoft Time Series [30]), щоб не лише виявляти аномалії, але й прогнозувати їх. Вміст журналів майже не використовується в цьому методі обробки логувальних даних. Він використовує часові ряди з MapReduce для моніторингу ККД системи, як-от використання диска та ЦП, і створює індекс аномалії на основі максимального та середнього використання ресурсів. Якщо аномалію буде виявлено за допомогою раптової зміни в метриках системи, система зможе відстежити її назад до відповідних повідомлень журналу. Однак, проблемою в системах аналізу журналів у реальному часі є не тільки продуктивність, як уже згадувалося раніше. В роботі автора [29] використовувалась часова характеристика, щоб визначити, чи корелюють записи журналу один з одним. У системі реального часу це означає, що проблему ковзних кордонів стає ще важче вирішити, оскільки не вся інформація відома, і необхідно ухвалити рішення, наскільки великий розмір вікна слід використовувати для обробки журналів.

2.5 Аномалії

Цей підпункт присвячений методам виявлення аномалій і поділений на чотири частини. Це пояснюється тим, що в галузі аналізу журналів література зосереджена в основному на одному або, можливо, двох із цих аспектів одночасно. Різні методи, що використовуються для виявлення аномалій, визначають походження різних видів аномалій. У цьому підпункті буде описано, що це за різні типи аномалій та як вони виявляються. Перший тип описує часове виявлення аномалії, тобто основна частина виявлення аномалії залежить від часу генерації повідомлень журналу. Другий – виявлення аномалії під час вимірювань продуктивності. Третій зосереджений на фактичному вмісті журналів програми. І останній – це візуалізація, яка відрізняється від попередніх трьох способів тим, що він не автоматизований, а потребує ручного аналізу, щоб побачити тенденції до появи аномалій.

Виявлення тимчасової аномалії базується на подіях, що відбуваються в

інтервалах часу. Це можуть бути дві або більше пов’язаних подій, що відбуваються протягом певного інтервалу. Тоді цей інтервал може бути пороговим або автоматично розрахованим на основі попередніх подій. Однак не тільки пов'язані події обмежуються за часом, а й частоти одного типу логарифмічної моделі можуть бути підраховані у певному інтервалі, як це роблять у працях [31, 32]. Вся суть полягає у створеннні трьох типів часових рядів: імпульс (p), ряд імпульсів (SP) і періоду групи імпульсів (PGP). Де (SP) є послідовністю (P) у певному інтервалі і те ж саме для (SP) у (PGP). У поєднанні з кореляційним аналізом аномалію в часових рядах можна простежити до першопричини в журналах додатків, це ще не було повністю досліджено, але попередній аналіз показав, що журнали додатків були синергетичними з часовими рядами. Створення часових рядів є популярною практикою для аналізу журналів. Дана концепція легка для розуміння, але має сенс тільки під час роботи з даними з часовими мітками. Проблема тут також полягає у вже згаданому ковзаючому вікні, оскільки не відомо де обрізати межі під час порівняння поточних журналів із вибраною підмножиною. Важливо, що для цього типу виявлення аномалій можна вибрати будь-який тип методології аналізу журналу. Якщо при аналізі враховується часовий фактор, то це буде дійсним варіантом для аналізу журналу, який намагається виявити аномалії чи збої. Розроблюване рішення також буде значною мірою покладатися на впорядковані журнальні ознаки, а завдяки пасивному навчанню моделі графа часові рамки стають важливим фактором того, наскільки достовірними будуть спостереження під час порівняння даних журналу в реальному часі з побудованою моделлю.

Виявлення аномалій також можна здійснити, перевіривши журнали продуктивності з комп’ютерної системи. Раптові зміни у використанні ресурсів можуть вказувати на те, що щось вийшло з ладу, і ці дані співвідносяться з журналами додатків, як у роботі [29]. Цей спосіб виявлення аномалій можна застосувати майже в будь-якій комп’ютерній системі, а системні журнали такого типу дуже поширені. Хоча коли використання ресурсів зміниться то вже може бути занадто пізно, але цей метод може розширити існуючі методи виявлення аномалій і запобігти майбутнім збоям шляхом аналізу результатів цих методів. Певним

чином розроблюване рішення також виявлятиме проблеми з продуктивністю з точки зору порівняння часу між повідомленнями журналу в трасуванні, що також може вказувати на високе використання ресурсів.

Виявлення аномалій на основі вмісту журналів програми є найбільш ресурсоємним. Моделювання шаблонів журналів може бути досить складним з точки зору того, що структура повідомлень може змінюватися. Використання ковзного вікна враховує менше інформації, але це компроміс, який може бути корисним якщо ретельно вибирати його величину. Також важко передбачити, коли вміст повідомлення є шкідливим. Необхідно мати глибокі знання про контекст параметрів у моделі журналу. Тому запропоноване рішення збереже вибрані частини повідомлень журналу, але виключно для подальшого ручного аналізу.

Візуалізація є неавтоматизованим процесом, але аномалії можна виявити вручну. Візуалізація все ще застосовується на практиці, оскільки можливі аномалії можуть бути представлені візуально. Основне завдання полягає у візуалізації журналів таким чином, щоб виявлення аномалій можна було зробити вручну. Це було зроблено шляхом аналізу журналів за темою, потім використання частоти термінів і, нарешті, створення міжтематичної карти відстаней і гістограми. Це полегшує ручну роботу під час роботи з великими обсягами журнальних даних. У розроблюваному рішенні також буде враховано візуальне представлення графа або частин графа, що допоможе розробнику мати справу з великою структурою, яка була побудована, і все ще мати можливість легко приймати рішення щодо неї.

2.6 Методології логування данних на основі типу розроблюваного додатку

Записи із журналу представляють собою дані, що дозволяють їх читачу витягати інформацію про те, що не так і перетворити її на знання про те, як цьому запобігти та як це виправити. Важливо заздалегідь вирішити, які дані можуть допомогти читачеві журналу, щоби включити їх у запис. Записуючи в журнал все, ми зберігатимемо там дуже велику кількість надлишкової інформації, що призведе

до марного використання пам’яті та, врешті-решт, її переповнення.

Інший аспект полягає в тому, що логування також займає певний час [4] і тому важливо вирішити, коли його краще проводити: чи метод або дію розпочинати до журналювання, паралельно з ним чи після нього. Логування перед завершенням обробки запиту може призвести до затримок, а його виконання перед ним не включатиме результатів проведеної роботи. Існує також інформація, яка ніколи не повинна включатися до лог-файлів (наприклад, паролі або інші конфіденційні дані користувачів).

Проведене дослідження складалося з огляду літератури та досліджень тематичних проектів з розробки програмного забезпечення. У вибірку було включено програмне забезпечення з відкритим кодом розроблене з використанням різних мов програмування. Вибірка складалася з додатків написаних на мовах Java та C#, де проекти програмних додатків на Java являли собою веб-додатки, а проекти програмних систем на C# були десктопними застосунками. Це дуже важливий факт, який пояснює велику кількість відмінностей у результатах проведеного дослідження.

У проектах на Java використовувалися Log for J (Log4J) від Apache [5], та LogBack [6]. Жоден з досліджуваних проектів не використовував нативний вбудований пакет Java.Util.Logging [7]. Розробники програмного забезпечення переходили з Log4J на SLF4J та на фреймворк LogBack через накладні витрати та обмеження Log4J [3]. Усі наймолодші проекти використовували LogBack. Причиною цього слугують менші накладні витрати та більші можливості розробників.

Ведення журналу здійснювалося безпосередньо шляхом виклику відповідного методу на екземплярі логера, а також через аспекти до та після виконання методу. Другий підхід використовувався дуже рідко.

Розробники реєстрували ім'я користувача, який увійшов в систему один раз, в момент запуску програми або після його успішної авторизації в додатку, мітку часу, ім'я java-класу, який виконував операцію та трасування стеку в разі виникнення виключення (в блоках try catch). Повідомлення, які зберігалися в

журналі шляхом ручного виклику логера, містили переважно етапи чистої бізнес-логіки (наприклад, користувач авторизований, права доступні, дію виконано і т.д.).

Журнали містили велику кількість інформації, яка реєструвалася самим застосунком за допомогою фреймворку Spring, а не розробниками. Це призводило до того, що файли ставали надзвичайно великими вже протягом декількох днів. Файли журналів додавалися кожного разу, коли починалася нова транзакція. Максимальний розмір файлу був заданий в налаштуваннях XML, і коли він був досягався, фреймворк логування починав записуватись дані у новий файл без втручання розробника програмного забезпечення. Це налаштування дозволило зменшити кількість лог-файлів.

Однак, кількість зареєстрованих даних все одно була дуже великою і це спричиняло проблеми з доступним обсягом пам'яті. Старі журнали видалялися вручну з періодичністю в декілька місяців або в момент коли служба моніторингу сповіщала про низький об’єм доступного сховища на сервері.

Під час супроводу розроблених додатків, журнали були проаналізовані розробниками програмного забезпечення та адміністраторами середовища для аналізу першопричин виявлених проблем. За результатами аналізу було виявлено що знайти причин небажаної поведінки додатків вдавалося не завжди через запізнілі повідомлення (коли журнали вже були видалені) або через те, що записи в журналах не дозволяли змоделювати етап роботи програми, що призвів до неправильної поведінки програми.

Розробники протоколювали в основному прямі повідомлення або виключення, але вхідні параметри, які могли призвести до таких подій, не були записані в журнал, а записи журналів, включені в ці файли, були лише частково корисними для аналізу першопричини.

Файли журналів допомагали лише в обмеженій кількості випадків. В основному в той час коли облікові дані для входу були неправильними, або коли база даних була недоступна.

У середовищі .NET також існує декілька фреймворків логування [33]. До найбільш відомих на сьогоднішній день відносяться Log for .NET (Log4Net) від

Apache [34, 35] та .NET Log (NLog) [36], який набуває все більшої популярності. Ці фреймворки подібні до фреймворків, відомих у мові Java, їх потрібно викликати безпосередньо. Подібно до аспектного програмування в Spring та ведення журналу до та після фактичного виконання методу, що забезпечується анотацією, в C# даний функціонал пропонується бібліотекою PostSharp [37], яка дозволяє протоколювати як вхідні так і вихідні параметри методу також.

Незважаючи на всі ці можливості, розробники програмного забезпечення як правило використовували свої власні імплементації логувальників. У всіх 8 досліджуваних проектах розробники віддавали перевагу своїй власній реалізації. Логувальник як правило моделювався як окремий, синглтонний клас [38], який включається в проект та викликається вручну.

Власноруч реалізований логувальник розпоряджався налаштуваннями, заданими у власному класі і ці налаштування можна було встановити заново у додатку. Таким чином, логувальник працював автоматично після включення у проект додатку. Використання інших фреймворків на кшталт Log4Net або NLog потребувало б наявності XML-файлу налаштувань.

Відповіді на 7 запитань, описаних у попередньому розділі, опираючись на досліджувані проекти показано у таблиці 2.1.

Таблиця 2.1 – Порівняння підходів до логування у Java та C#

Питання для відповіді при логуванні	Підхід, що використовувався у проектах написаних на Java	Підхід, що використовувався у проектах написаних на C#
1	2	3
Що сталося?	Трасування стека виключення в журналі. Прямі повідомлення в лог, що інформують про те, що метод метод був успішно завершений.	Трасування стеку зареєстрованого виключення. Прямі повідомлення журналу, що інформують про те, який метод був виконаний і на якому етапі.
Коли це сталося?	Мітка часу сервера	Мітка часу операційної системи комп'ютера
Де це сталося?	Ім'я Java-класу, метод, що викликається	Ім'я методу виклику, виключення

Кінець таблиці 2.1

1	2	3
Хто спровокував подію?	Ім‘я користувача	Ім‘я користувача
На кого вплинула ця подія?	Ім‘я користувача	Компонент запуску та ім'я користувача
Чому це сталося?	Трасування стека виключення в журналі. Прямі повідомлення в лог, що інформують про те, що метод метод був успішно завершений.	Трасування стеку зареєстрованого виключення. Прямі повідомлення журналу, що інформують про те, який метод був виконаний і на якому етапі.
Що спричинило це?	Трасування стека з винятком	Трасування стеку зареєстрованого виключення

Як бачимо, команди розробників Java та C# обрали абсолютно різні стратегії ведення журналів. Команди розробників C# використовують перевагу знань про кінцевого користувача (оскільки в основному досліджувалися саме .NET проекти орієнтовані на десктопні застосунки) що допомогло їм розробити власний логувальник та власний процес логування. Java-розробники, використовували фреймворки для ведення журналів та можливості, які надає фреймворк Spring (включаючи аспектне логування). Проте, журнали були величезними в порівнянні з журналами C#, записи журналу не завжди були корисними для аналізу першопричини і приводили більше до переповнення пам'яті, ніж до документації помилки. Це говорить про існування великих проблем із високонавантаженими серверними додатками, які існують через неможливість застосувань деяких оптимізацій, які доступні для десктопних застосунків. До цих проблем належить висока складність виявлення аномалій в ручному режимі, через дуже велику кількість даних, яка генерується програмною системою за короткі проміжки часу, та високе навантаження на пам’ять. Це свідчить про необхідність розробки нового методу, який дозволить слідкувати за аномаліями в програмних системах з великим обсягом даних.

2.7 Висновки

Усі дослідження аналізу журналів показують, що існують дійсні методи аналізу даних журналу, які можуть виявити важливі приховані фактори під час розробки чи моніторингу системи. Той факт, що майже будь-яка комп'ютерна система виводить дані журналу, робить це поле досліджень таким, що потенційно може принести велику користь у галузі. Однак, також можна побачити, що немає рішень, які покладаються виключно на дані журналу для аналізу в реальному часі. Наявні засоби здебільшого пропонують лише постобробку. Головним чином це пов'язано з проблемами, які виникають під час аналізу великої кількості отриманих журнальних даних, при тому що неструктуровані дані ще складніші для обробки. Існуючі інструменти не масштабуються або не працюють у режимі реального часу, в той час як створення масштабованого рішення для моніторингу журналів у реальному часі є метою цього дослідження. Таким чином, це дослідження є розширенням галузі аналізу журналів таким чином, що воно може надати доказ концепції того, що аналіз у реальному часі можна проводити та потенційно приносити користь галузі більше, ніж існуючі методи.

3 ТЕХНОЛОГІЇ ВИРІШЕННЯ ЗАДАЧІ

3.1 Аналіз вимог до програмного засобу

Метод, описаний у попередньому розділі полягає в масштабуванні рішення, яке структурує і моделює дані журналів даних так, щоб подальший аналіз був доступний через розподілені екземпляри дочірніх підсистем. Для цього у даному розділі буде розроблено архітектуру для створення середовища в якому аналіз даних журналу може масштабуватися по горизонталі для роботи з великою кількістю операторів журналу.

Дана архітектура повинна включати в себе декілька частин, кожна з яких може горизонтально масштабуватися. Розроблюване рішення повинне працювати через масштабовані блоки, які будують багаточастковий граф і можуть виконувати над ним обчислення, та один основний екземпляр, який періодично дістає часткові графи та контролює повідомлення про аномалії. Дане рішення базується на кількох попередньо встановлених вимогах. Це дає уявлення про те, на що здатне запропоноване рішення, а також дає короткий опис завдань, на яких фокусується дана система:

1. Збір даних журналу
2. Приведення повідомлень журналу до структурованого вигляду
3. Виокремлення ознак
4. Обробка ознак розподіленим способом (для підтримки можливості горизонтального масштабування)
5. Створення структури даних подібної до детермінованого скінченного автомата (ДСА) із ознак, що дозволить проводити їх аналіз
6. Зберігання стану системи у вигляді змодельованого графа
7. Здатність порівнювати поточний стан системи з раніше збереженим з метою виявлення нових шляхів
8. Здатність виявляти проблеми з продуктивністю в реальному часі опираючись на різницю в часі між повідомленнями.

3.2 Загальний огляд компонентів системи

Запропоноване рішення найбільше фокусується саме на розподіленій системі, яка дозволяє обробляти великий обсяг даних журналу. Тому дане рішення залежить від багатьох інших факторів, таких як середовище, в якому воно буде працювати, і яким чином ознаки будуть подаватися в дане рішення. На рисунку 3.1 схематично продемонстровано те, як всі частини системи повинні взаємодіяти між собою.

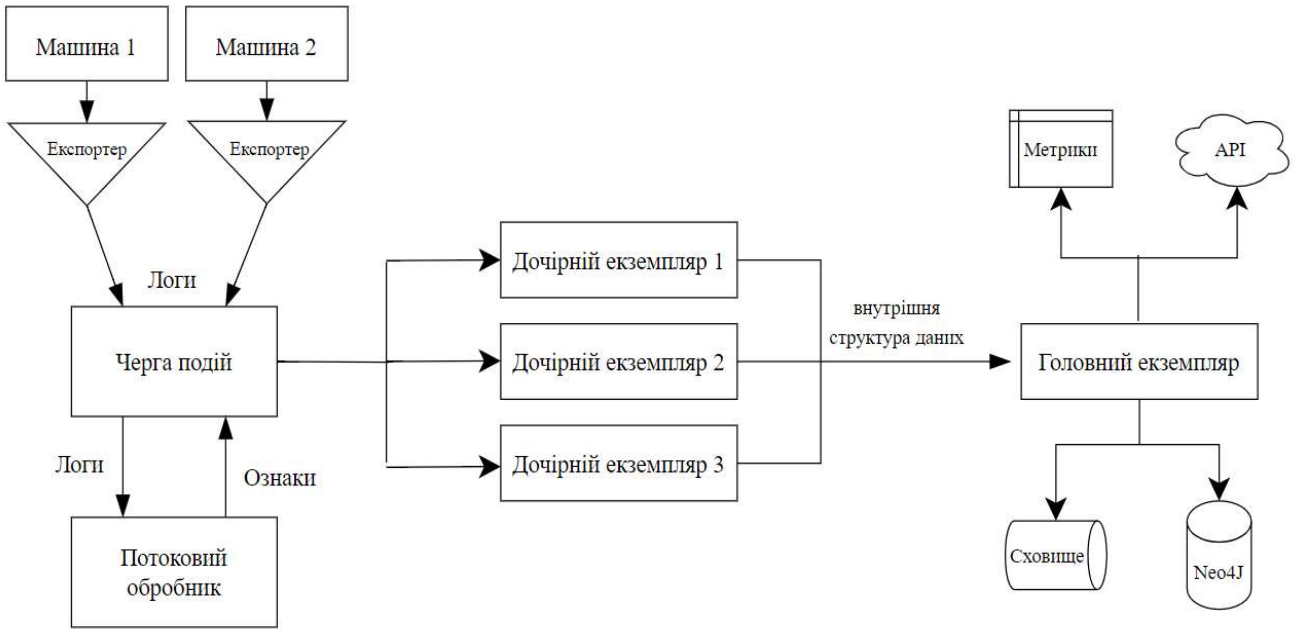


Рисунок 3.1 – Схема взаємодії компонентів системи

3.3 Архітектура системи

На сьогоднішній день існує багато систем, які виробляють велику кількість логів наприклад, системи компаній Google, Facebook, Adyen, Amazon. Структурування ознак та проведення аналізу в реальному часі з використанням цих самих ознак вимагає масштабування ресурсів. Якщо є тільки одна машина, здатна запустити програму для обробки цього потоку ознак, це означає, що єдиним

варіантом є вертикальне масштабування, тобто збільшення потужності цієї однієї машині з точки зору процесора або оперативної пам'яті.

Вертикальне масштабування може означати, що ціни на необхідні ресурси можуть стати дуже високими, і в кінцевому підсумку буде досягнута межа того, що одна машина може витримати з точки зору апаратного забезпечення. Іншим способом масштабування є горизонтальне масштабування, яке вимагає додавання більшої кількості машин, які можуть обробляти ті ж самі вхідні дані, що в багатьох випадках буде дешевше і межа буде залежати від того, скільки машин можна додати. Тому ця архітектура фокусується на тому, щоб кожна частина була горизонтально масштабованою, що полегшить роботу з великими обсягами даних журналів повідомлень.

Існує багато варіацій того, як системи генерують дані журналу. Це може бути одна програма на машині, що виробляє вихідні дані для збору; це може бути декілька програм на одній машині або навіть декілька програм на декількох машинах. Незважаючи на те, що існує багато варіацій, всі дані журналу повинні бути зібрані якимось конкретним загальним способом, після чого вони можуть бути оброблені в подальшому.

У запропонованому рішенні це означає наявність центрального вузла у вигляді системи черги видавця/підписника в середовищі, яке все ще може бути масштабованим горизонтально. Система видавця/підписника здатна отримувати всі рядки журналів і ознаки, після збору яких здатна перерозподіляти їх (використовуючи алгоритм типу round-robin) між абонентами, які можуть продовжити обробку потоку даних. Популярними системами черг подій є Kafka, RabbitMQ, Apache ActiveMQ та NATS.

Існує декілька способів, якими існуюча програма може взаємодіяти з такою системою. Один із способів може бути здійснений без внесення змін до існуючої кодової бази системи шляхом впровадження експортера в необхідне середовище. Експортери, по типу LogStash або Fluentd, збирають дані журналів у системі на основі заданих конфігурацій після чого експортують їх, наприклад, до системи черги подій.

Іншим способом може бути надсилання створених даних журналу безпосередньо до системи керування чергою. На рисунку 3.2 можна побачити діаграму послідовності, яка показує, як дані журналу проходять через систему.

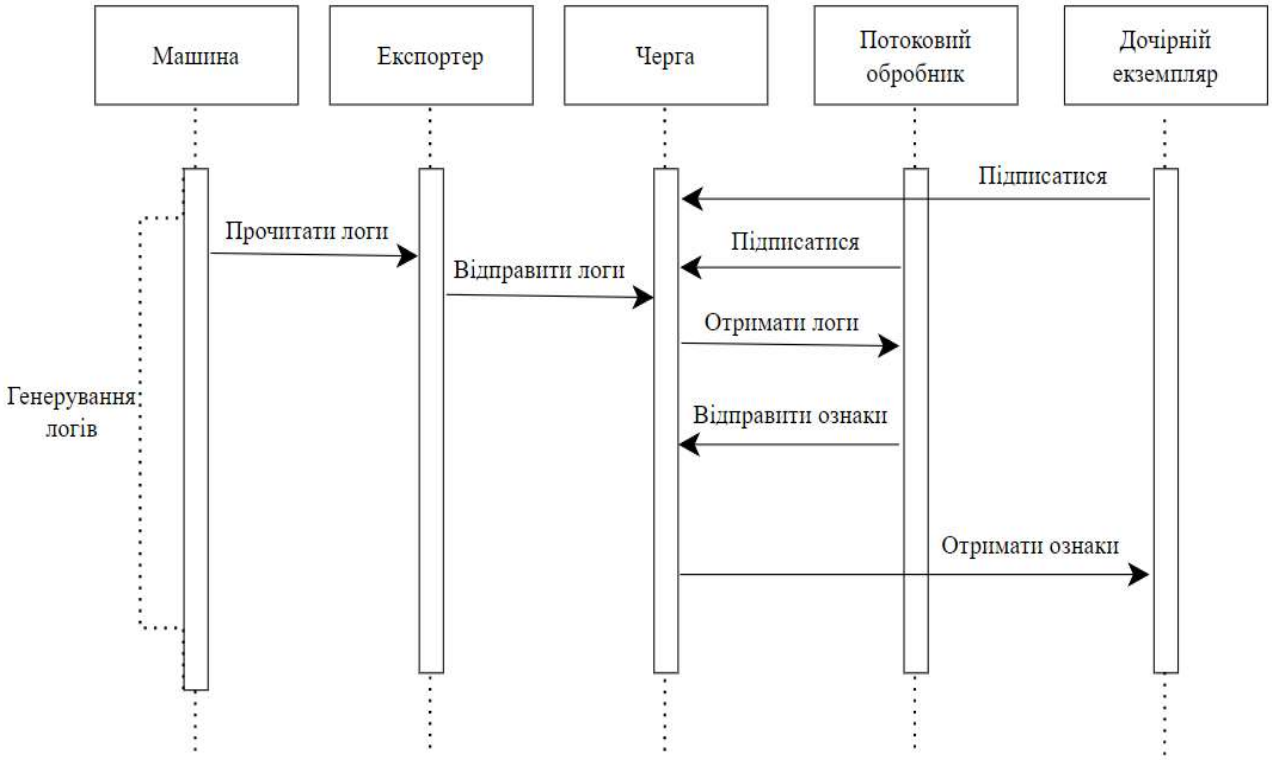


Рисунок 3.2 – Послідовність проходження даних через систему

Процес продовжується після збору логів шляхом пошуку ознак з отриманих повідомлень, тобто перетворенням окремих повідомлень в об'єднані, які представляють собою сліди або ознаки. На рисунку 3.1 це можна побачити в нижньому лівому куті, де рядки журналу надсилаються до потокового процесора, а отримані ознаки надсилаються назад до системи видавця/підписника. Запропоноване рішення вимагає середовища, в якому ці рядки журналу можуть бути переглянуті в часовому вікні за допомогою потокового процесора, такого як Apache Flink або Benthos. Процесор потокового передавання даних повинен працювати за механізмом ковзаючого вікна, щоб мати можливість обробляти вхідні дані, що постійно зростають, і об'єднувати рядки журналу в одну ознаку. За допомогою процесорів потокової обробки даних ми можемо розділити рядки

журналу за часом і, попередньо налаштувавши часовий ліміт між рядками журналу, отримати повне трасування. На прикладі діаграми 3.3 це зрозуміло завдяки обмеженню в двадцять секунд. Вона відображає потік журналу з ідентифікаторами трасування «A» і «B» у першому часовому вікні. За визначенням, ознака «A» вважається завершеною якщо даний ідентифікатор трасування не буде знайдено. Це означає що тепер ця траса може бути зібраною. Врешті-решт це означає, що ознака оброблятиметься після попередньо налаштованого ліміту в секундах і пізніше, ніж останній рядок журналу з його ідентифікатором трасування.

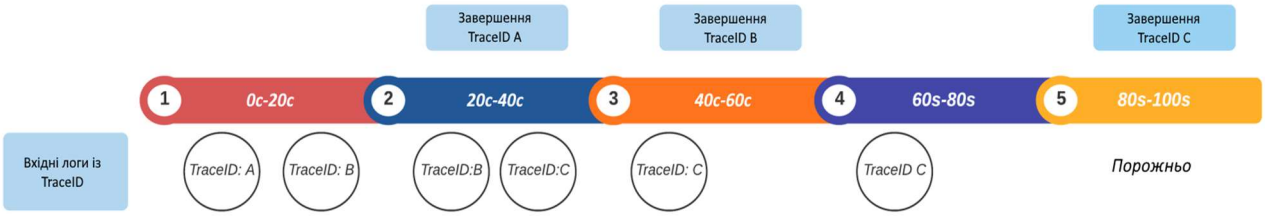


Рисунок 3.3 – Потік журналу

Останньою частиною обробки даних є передача даних журналу відстеження дочірньому екземпляру, так як це показано посередині рисунку 2.1. Цей заключний процес складається з двох частин.

Перша частина, яка представлена головним екземпляром, є загальним менеджером станів, який може зберігати стан, об'єднувати нові стани, експортувати весь стан та обробляти певні запити.

Друга частина, яка представлена дочірніми екземплярами, являє собою масштабовані одиниці, здатні отримувати ознаки, динамічно будувати графік та мати можливість аналізувати їх у реальному часі. Дочірні екземпляри підписуються на чергу і можуть бути розгорнуті окремо, всі дочірні екземпляри витягують ознаки на основі алгоритму циклічного перебору, тому з точки зору обчислень навантаження розподілятиметься. Можна розгорнути будь-яку кількість дочірніх елементів, що робить дане рішення горизонтально масштабованим, і кожен дочірній елемент може виконувати свої власні обчислення на вхідних

ознаках, будувати граф та порівнювати його з попереднім базовим графом. Потім головний екземпляр робить злиття часткових дерев дочірніх екземплярів і пропонує доступ до API, метричну експозицію та можливості для експорту.

Масштабовані дочірні екземпляри здатні виконувати прості обчислення на основі ознак та можуть бути розширені в подальшому. Проте, на даний момент вони мають дещо обмежений функціонал:

- обчислювати різницю в часі між рядками журналу;
- порівнювати середню різницю в часі між лініями журналу із змодельованим базовим графом;
- повідомляти про збільшення різниці в часі між рядками журналу;
- повідомляти про знаходження нових шляхів у порівнянні зі змодельованим базовим графом;
- повідомляти про стрибки кількості обробок певного шляху;
- повідомляти про конкретні ідентифікатори/корисне навантаження в шляхах (наприклад, про ідентифікатор користувача).

Таким чином, головний екземпляр звільняється від вищевказаних обчислювальних обов'язків і відповідає лише за злиття повних структур графів, які він отримує від дочірніх екземплярів. Потім він може зберігати або експортувати свій поточний стан, щоб він міг стати новою основою для дочірніх екземплярів, з якою вони можуть порівнювати свої обчислення. Проте, існує ще кілька незначних обчислень, за які все ще відповідає головний екземпляр. Наприклад, коли дочірні екземпляри знаходять новий шлях, вони звітують про це головному екземпляру, тому він запам’ятовує новий шлях лише один раз, замість того, щоб кожен дочірній екземпляр його запам’ятовував. Це досить легкі операції; однак, вони стосуються лише експорту метрик, які також виконують всі дочірні частини.

Дочірні частини не мають стану, тому вони змушені звертатися до основного екземпляра, щоби отримати від нього всю необхідну інформацію. Після отримання вони запускаються окремо, підписуючись на чергу з ознаками. Це означає, що дочірні екземпляри можуть вимикатися в будь-який момент і їх можна масштабувати незалежно від основного екземпляру.

Головний екземпляр також може перезавантажуватися в будь-який час, він читатиме лише останню або попередньо налаштовану структуру графів, а дочірні частини будуть повторювати спробу надіслати свою інформацію до головного екземпляра, у випадку якщо їм це не вдасться з першого разу.

На рисунку 3.4 показано, як взаємодіють різні частини системи, починаючи із запиту конфігурації, потім підписки на чергу з ознак та обробки кожної з отриманих ознак.

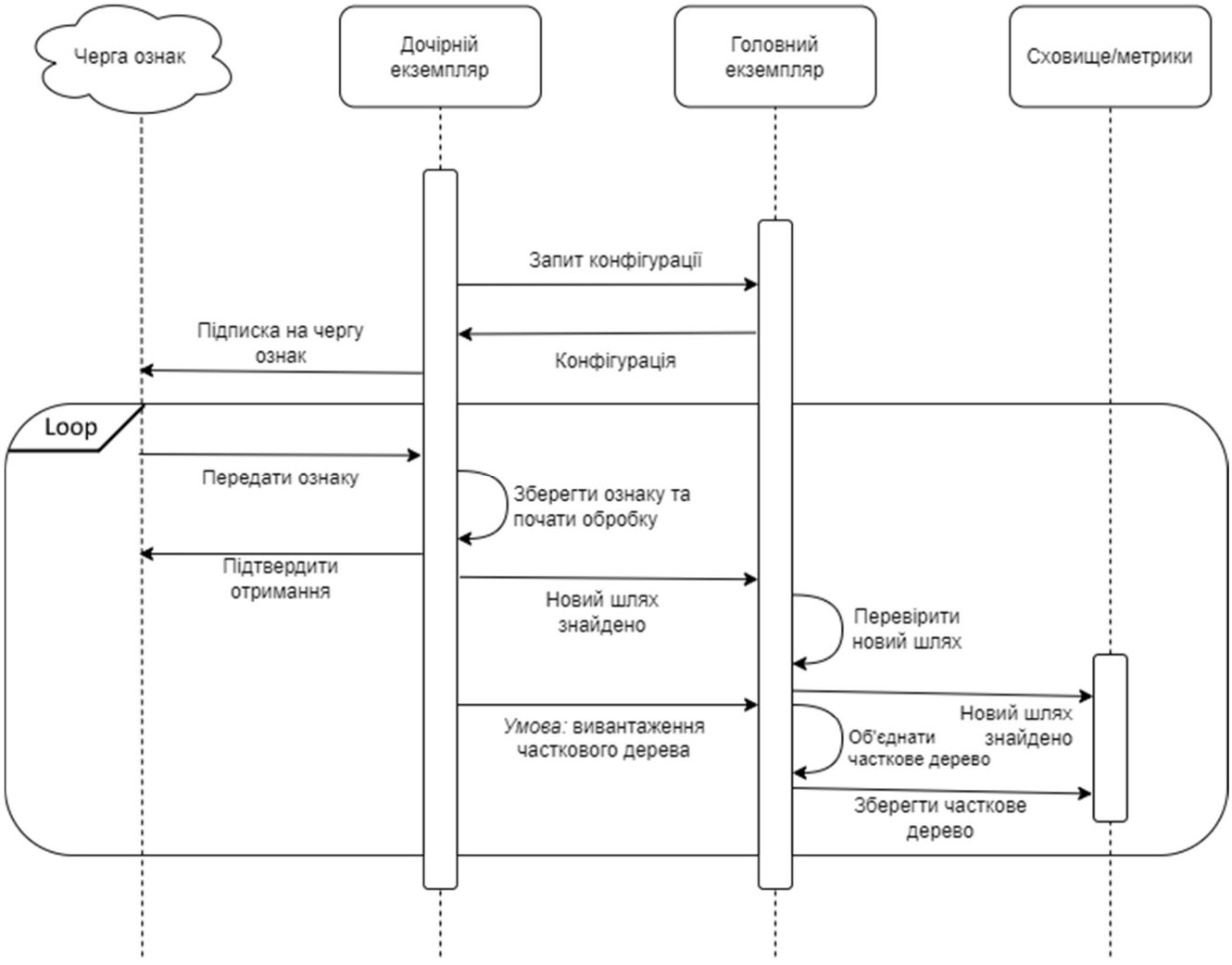


Рисунок 3.4 – Взаємодія компонентів системи

Отже, було описано всі різні частини програмної системи для роботи з великими обсягами журнальних даних. Починаючи з програм або машин, які генерують необроблені дані журналу, які потім збираються експортером, який

надсилає дані журналу до системи масового обслуговування. Там вони обробляються масштабованими потоковими процесорами, які об’єднують окремі рядки журналу в пакети з часовим вікном, з яких можна витягнути ознаки. Потім ці ознаки можуть бути перерозподілені між масштабованими дочірніми екземплярами системою черги видавця/підписників.

Дочірні екземпляри виконуватимуть обчислення та створюватимуть структуру графа, а головний екземпляр отримуватиме часткові дерева, коли буде виконано попередньо зазначену умову, і об’єднуватиме, зберігатиме та потенційно експортуватиме їх.

3.4 Структурування даних журналу та парсинг

За своєю структурою дані журналу можуть відрізнятися в кожній комп'ютерній системі та програмі, але у даних журналу є декілька компонентів, які є стандартними та спільними для багатьох рішень, що вже представлені на ринку. Тому запропоноване рішення вимагає наявності трьох різних частин в журналі для того, щоб мати можливість побудувати з них граф.

Наступні властивості є необхідними для того, щоб запропоноване рішення могло створювати із них графи:

- мітка часу;
- статичний ідентифікатор запису;
- ідентифікатор трасування.

Мітка часу необхідна для того, щоб забезпечити сортування журналів всередині ознаки на основі часу виконання, коли невідомо, чи будуть рядки журналу доставлені в правильній часовій послідовності.

Статичний ідентифікатор запису необхідний для ідентифікації виводу журналу в кодовій базі системи, де повідомлення може відрізнятися кожного разу на основі динамічного виведення значень.

Ідентифікатор трасування необхідний для визначення того, які рядки

журналу були виведені одним і тим же процесом або потоком виконання, які в кінцевому підсумку можуть бути впорядковані за часовою міткою, і результатом яких буде один шлях виконання на графіку.

Всі інші види корисного навантаження, такі як повідомлення або, можливо, поля, визначені у JSON, все ще можуть бути прийняті до уваги, але вони не потрібні для створення графа шляхів виконання.

Показані на рисунку 3.5 рядки журналу демонструють приклад необхідних властивостей разом із динамічно створеними під час виконання повідомленнями. Рядки формуються з мітки часу, ідентифікатора журналу, ідентифікатора трасування та повідомлення.

Видно, що в даному прикладі є два різних трасування. Окрім двох різних ідентифікаторів траси, у цьому прикладі є ще п'ять різних ідентифікаторів журналу. Про важливість ідентифікатора журналу свідчать два повідомлення "Отримано 601 користувача" та "Отримано 602 користувача", які відрізняються один від одного за змістом але насправді надходять з одного і того ж місця в кодовій базі. Звідси той самий ідентифікатор журналу, який дозволяє графу формувати статичний шлях від виводу журналу в кодовій базі замість того, щоб засновувати його на повідомленнях, які змінюються динамічно.

```
[2019-10-28 10:05:01:736] {MRAjWwhTHc} {cdddbcdf} Started Request [GET]: /v1/user
[2019-10-28 10:05:02:126] {bZRjxAwnwe} {cdddbcdf} Retrieved 601 users
[2019-10-28 10:05:02:220] {MRAjWwhTHc} {6ce65118} Started Request [GET]: /v1/user
[2019-10-28 10:05:02:223] {TndUJHiQec} {cdddbcdf} Appended states to users succesfully
[2019-10-28 10:05:02:289] {uXvgxEqdoX} {cdddbcdf} Finished Request [GET]: /v1/user
[2019-10-28 10:05:03:120] {bZRjxAwnwe} {6ce65118} Retrieved 602 users
[2019-10-28 10:05:03:120] {tCjySfUrie} {6ce65118} Error appending state to user X
[2019-10-28 10:05:03:160] {uXvgxEqdoX} {6ce65118} Finished Request [GET]: /v1/user
```

Рисунок 3.5 – Приклад рядків журналу

Обидва трасування в прикладі починаються з однакових ідентифікаторів журналу, але йдуть різними шляхами із кодової бази. Нижче на рисунках 3.6 та 3.7 зображено результуючий граф, який складається з цих прикладів рядків журналу як трас та об'єднаний в один графік. Це також показує важливість пошуку

правильної структури даних для зберігання цієї інформації та можливості швидкого зберігання та доступу до цих проаналізованих рядків журналу та отриманих ознак.

Хоча ознаки можуть бути зображені як окремі шляхи в дереві, але в ідеалі, вузли графів, які є однаковими, об'єднуються, коли вони все ще відображають коректні шляхи.

3.5 Побудова префіксних дерев

Для масштабованого структурування ознак необхідна реалізація дерева префіксів, яку можна буде використовувати спільно для всіх розгортаних рішень.

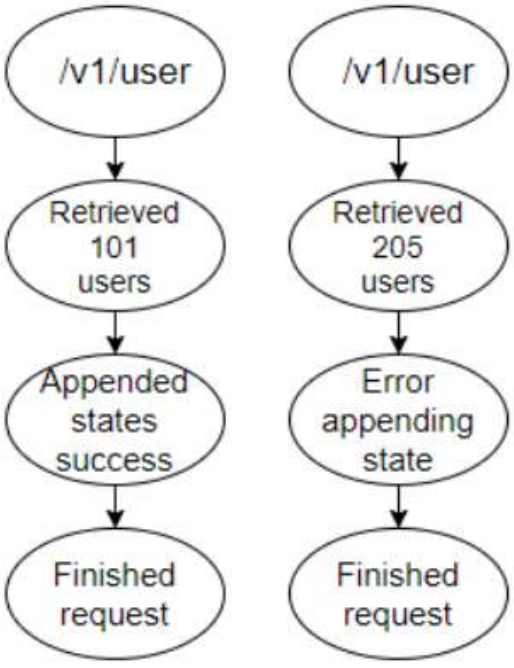


Рисунок 3.6 – Приклад розділених трасувань на графі

Префіксне дерево працює за допомогою створення шляхів у графі на основі інформації, що зберігається в ребрах. Вузли дерева префіксів можна розглядати як заповнювач для даних, але не мають істинного значення з точки зору представлення графа. Це означає, що в якості символів для кожного ребра в дереві

префіксів можна вибрати алфавіт, де для кожного символу в обраному алфавіті створюється ребро з цим символом з попереднього символьного ребра. Тому при побудові дерева префіксів слово в обраному алфавіті буде буде розбито на символи, і для кожного символу, що зчитується з початку слова, буде створено ребро слова, буде створено ребро і наступний символ буде на один рівень глибше.

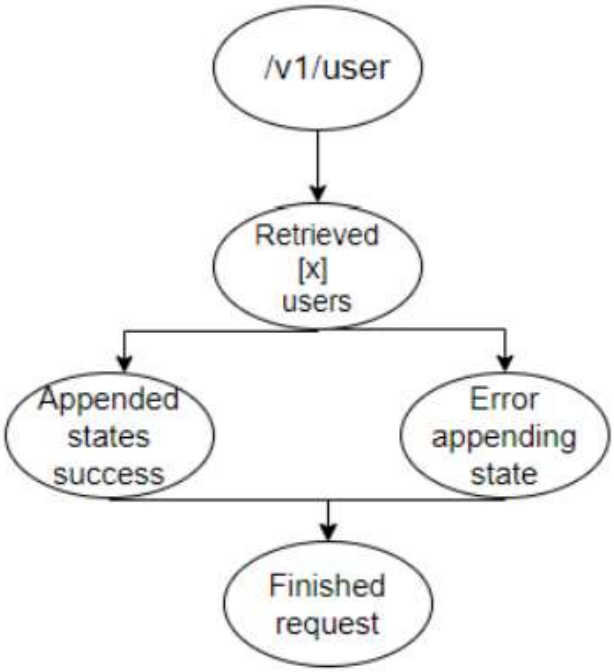


Рисунок 3.7 – Приклад трасувань на графі

Оскільки в даній дипломній роботі мова йде про обробку великих об'ємів повідомлень журналу та їх моделювання у префіксне дерево, то реалізація базується на безпечній для потоків структурі префіксного дерева. Реалізована структура даних префіксного дерева працює з блокуваннями взаємного виключення, які роблять вставку нових шляхів та доступ до існуючих шляхів швидшими за рахунок того, що до них одночасно можуть звертатися декілька потоків. На рисунку 3.8 наведено псевдокод реалізації алгоритму, який включає цю структуру даних та працює з взаємовиключним доступом.

3.6 Комбінування префіксних дерев

Всі дочірні екземпляри будуватимуть власне дерево префіксів на основі отриманих ознак. Побудова дерева префіксів, обчислення і додавання метаданих можуть бути вимогливими до ресурсів в залежності від того, які обчислення виконуються. Однак головний екземпляр, який відповідає за створення остаточного дерева префіксів на основі часткових, які він отримує з часом, не є масштабованим. Тому обчислювальна складність об'єднання префіксних дерев не повинна бути високою, щоб зробити запропоноване рішення підходящим масштабованим способом обробки великих обсягів даних журналів повідомлень.

```
procedure RETRIEVING TRACE
    trace ← queue
    goroutine processTrace.
processTrace:
    Sort trace by timestamp
    for lineIndex in trace do
        trace(lineIndex+1).Performance = delta(trace(lineIndex),trace(lineIndex+1))
        start putLine trace(lineIndex).
    end for
putLine:
    node ← trie
    logIdentifier ← line.static
    for char in logIdentifier do
        node.MutexLock
        child ← node.Children(char)
        if child eq empty then
            child = Node()
        end if
        node.MutexUnLock
        node = child
    end for
    node.MutexLock
    node.Value = NewValue
    node.MutexUnLock
end procedure
```

Рисунок 3.8 – Псевдокод алгоритму обробки ознак

Головний екземпляр буде періодично отримувати часткові дерева префіксів від своїх дочірніх структур. Тому буде побудована черга для злиття часткових дерев по одному з останнім відомим головним деревом. Щоб об'єднати головне дерево з отриманим частковим деревом, створюється нове дерево префіксів. Починаючи з кореня головного дерева, перевірятимуться всі ребра від кореня, якщо вони відомі в отриманому частковому дереві. Якщо ні, весь шлях від кореня можна відразу вставити в нове префіксне дерево. Якщо ребро знайдено в отриманому частковому дереві, то обчислені змінні будуть агреговані, а ребро буде вставлено у нове префіксне дерево. Наступні від взаємних ребра (дочірні), пройдуть через той же процес. Якщо їх не знайдено в отриманому дереві, то негайно додається шлях, що залишився від цього ребра, інакше воно агрегується.

Єдиним кроком, що залишився, є додавання повних шляхів, які були знайдені в отриманому частковому дереві, але не були знайдені в попередньому основному дереві. Додавання цих шляхів може бути виконано шляхом другого обходу отриманого часткового дерева, без додавання взаємних ребер.

Це означає, що у найгіршому випадку для об'єднання двох префіксних дерев потрібно буде обійти усі ребра основного дерева (Em) і всі ребра отриманого часткового дерева (Ep), що зводиться до складності O(Em+Ep).

3.6 Вибір мови програмування для імплементації рішення

Для розробки запропонованого рішення було обрано Golang [8]. Golang — це мова програмування, яка із самого початку розроблялася Google, та має відкритий вихідний код. Однією з причин, чому Golang було обрано як мову для проведення даного дослідження, є те, що дана мова чудово підходить для роботи з кластером і і масштабованими частинами. Golang створено для компіляції в самостійні двійкові файли, які не мають залежностей від бібліотек із коробки, що робить образи докерів для розгортання дуже малими. Docker [4] можна використовувати для створення образів, які називаються контейнерами, які містять усі необхідні програмні

компоненти для запуску програми. По суті, його можна розглядати як інструмент для попереднього налаштування та конфігурації операційної системи, щоб забезпечити всі потреби для певного екземпляру додатка, а потім мати можливість віртуально запускати його на машині. Крім того, що Golang може працювати в невеликих контейнерах, Golang також пропонує дуже ресурсоефективний метод для багатопоточності під назвою goroutines, вбудований у середовище виконання замість того, щоб оброблятися операційною системою. Це дає змогу запускати goroutine для кожної вхідної ознаки, що робить обробку дуже швидкою та не блокує великі обсяги вхідного трафіку.

3.7 Висновки

У даному розділі було проведено аналіз вимог до програмного засобу. Запропоноване рішення було описане в абстрактному та загальному вигляді. Виокремлено та визначено як модулі різні частини запропонованого рішення, які є або обов'язковими, або необов'язковими. Для кожної частини або модуля запропонованого рішення буде надано його опис, а також спосіб його взаємодії з іншими частинами системи. Далі буде детально описана реалізація описаного підходу, зокрема, технології та мови, що використовуються, і чому вони були обрані як прийнятний варіант для проведення дослідження.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

4.1 Імплементація програмного засобу на основі запропонованої архітектури

Дана архітектура була створена з акцентом на відтворюваність. У цьому розділі буде описано, які технології обрано для реалізації рішення, спроектованого у попередньому розділі. Кожне рішення, прийняте в цьому процесі, буде детально описане та пояснено, чому було обрано саме цей метод. Повний огляд архітектури можна побачити на рисунку 4.1, де всі послідовні взаємодії описуються числом, яке слідує за діями, що відбуваються між різними компонентами системи.

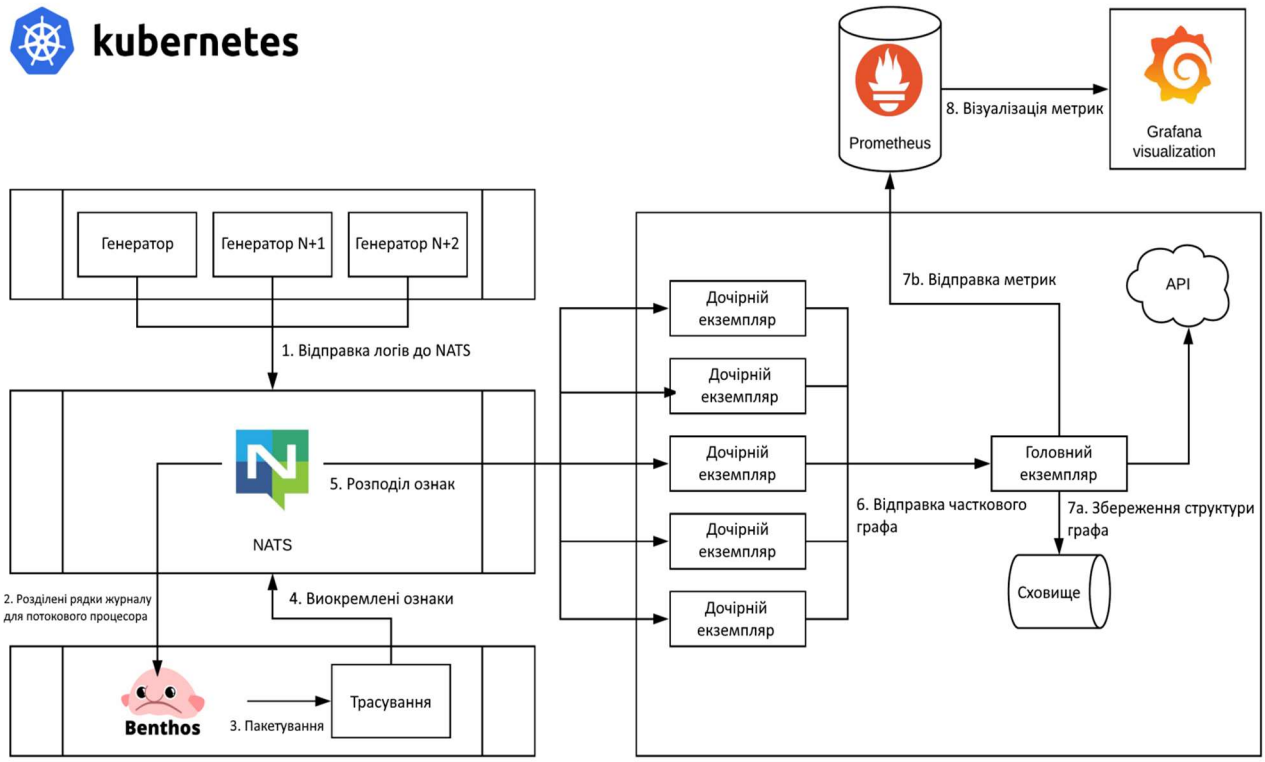


Рисунок 4.1 – Архітектура повного програмного середовища

4.1.1 Kubernetes

Перша частина екосистеми починається з Kubernetes. Вона була розроблена компанією Google, а зараз підтримується Native Computing Foundation. Kubernetes

– це платформа для автоматизованого розгортання, масштабування та управління додатками. Kubernetes працює за допомогою спеціальних конфігураційних ресурсів, які вказують, як має бути розгорнутий додаток, наприклад, які змінні оточення повинен мати додаток і який тип жорсткого диску повинен використовуватись програмою. Це широко використовувана платформа для вирішення саме тих завдань, з якими стикається це дослідження, а саме розгортання великої кількості різних компонентів в одному спільному кластері та можливість їх масштабування за допомогою конфігурацій.

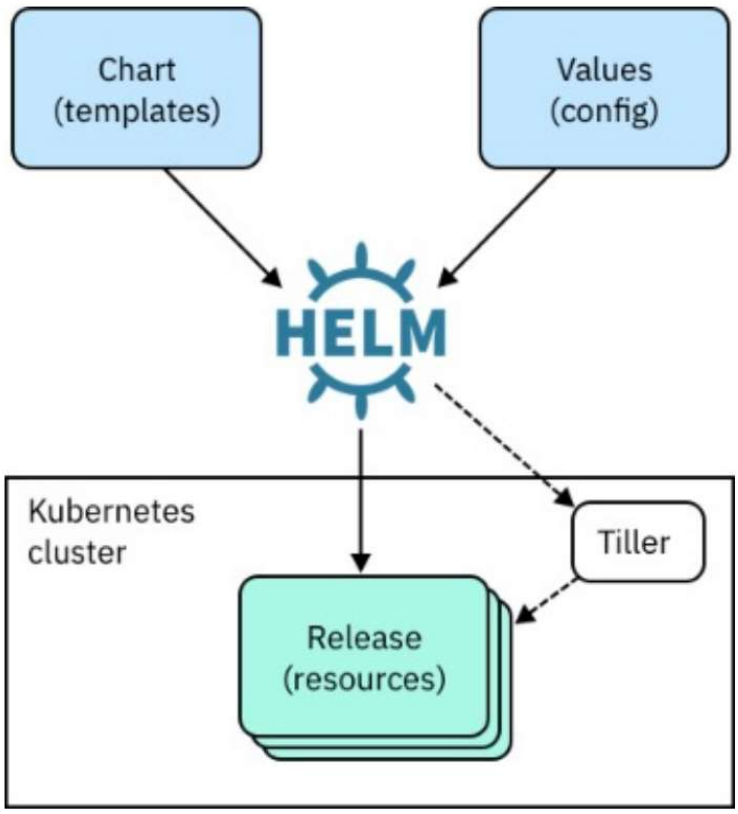


Рисунок 4.2 – Взаємодія діаграми Helm із кластером Kubernetes

Дані конфігурації можуть бути заздалегідь створені за допомогою діаграм Helm (керуючі діаграми), які є своєрідними шаблонами для розгортання, де ви можете досить легко редагувати параметри в шаблоні для легкої зміни конфігурацій. Діаграми Helm також використовуються для легкого відтворення експериментів і для подальшого використання масштабованого аналізу журнальних даних шляхом розгортання запропонованого рішення. На рисунку 4.2

показано схему того, як створюються діаграми Helm і як вони взаємодіють з кластером Kubernetes.

4.1.2 Система черги подій

Як вже було вказано у попередніх розділах, існують різні системи керування чергами подій. Оскільки дана реалізація працює на Kubernetes, вибір був зроблений на користь NATS та NATS Streaming, які досить легко розгортаються на Kubernetes та мають власноруч розроблену та задокументовану бібліотеку для Golang. Окрім того, відмінності між різними системами обслуговування черги подій не є важливими для цього дослідження, оскільки всі вони повинні бути в змозі запропонувати і досягти однакових результатів з незначними відмінностями у продуктивності, в залежності від того, що саме необхідно. Для цього рішення було обрано комбінацію між NATS та NATS Streaming, в той час як NATS використовує принцип At Most Once Delivery, а NATS Streaming – At Least Once Delivery. NATS може бути замінений на NATS Streaming, якщо потрібна стовідсоткова гарантія під час доставки рядків журналу. NATS Streaming – це розширення NATS. Даний засіб пропонує тривалішу підписку та більше можливостей для перерозподілу елементів черги. Це необхідно для циклічного розподілу ознак до кожного дочірнього екземпляра, який підписався на чергу з ознак.

4.1.3 Потоковий обробник

Для потокового процесора було обрано Benthos, оскільки це легкий одноцільовий потоковий процесор, який може бути розширений кодом на мові Golang, що полегшує його інтеграцію з іншими частинами з точки зору програмування. Benthos обробляє прості операції над потоками даних з усіх видів джерел. Прості операції включають пакетну обробку та злиття декількох об'єктів в

один на заздалегідь визначених частинах повідомлення, таких як, наприклад, ідентифікатор ознаки. Це робить Benthos відмінним кандидатом для пакетної обробки та попереднього групування рядків журналу, які належать до однієї ознаки. Окрім того, додатково було написано плагін, який приймає два пакети, що йдуть один за одним, і виводить трасування в одну чергу, а решту зберігає в пам'яті, доки не прийде новий пакет, як описано в попередньому розділі. Його код можна переглянути у Додатку А.

4.1.4 Метрики

Щоб відстежувати всі частини запропонованої екосистеми, система повинна контролюватися таким чином, щоб дати користувачеві загальне уявлення про те, що відбувається. Це може допомогти виявляти вузькі місця та отримувати розуміння того, яка частина використовує які ресурси.

В процесі моніторингу системи навіть кілька основних показників можуть пролити світло на те, наскільки добре система працює. Апаратні метрики є найбільш поширеними для моніторингу, але також може враховуватись пропускна здатність, затримка та інші користувацькі види програмних метрик, які можуть дати ще краще розуміння роботи системи.

Одним з універсальних рішень для відстеження метрик в кластері Kubernetes є перехід до використання Prometheus. Існує широкий спектр експортерів, доступних для Prometheus, які вимірюють всі види програмних та апаратних метрик програмного та апаратного забезпечення і надсилають їх до Prometheus. Prometheus пропонує мову запитів, щоб отримати уявлення про зібрані метрики.

Grafana дозволяє візуалізувати запитувані з Prometheus метрики, а також дозволяє створювати інформаційні панелі з декількома графіками, щоб отримати огляд того, що відбувається в кластері в режимі реального часу.

У цьому рішенні головний екземпляр, а також дочірні екземпляри експортують метрики в Prometheus. До цих метрик, серед інших, належать також

кількість оброблених ознак і нових шляхів, які вдається віднайти. Окрім того, для NATS були встановлені експортери, які дозволяють вимірювати пропускну здатність в режимі реального часу.

До даного рішення включена Grafana, інформаційна панель якої здатна відстежувати наступні показники в режимі реального часу:

- виявлення нових шляхів;
- погіршення продуктивності на окремих шляхах;
- кількість оброблених ознак;
- кількість оброблених рядків журналу;
- лічильник ідентифікаторів;
- NATS RTT;
- пропускна здатність мережі кластера NATS;
- пропускна здатність повідомлень NATS;
- кількість підключень NATS;
- повідомлення про помилки в роботі елементів системи.

Невеликий приклад того, як виглядає ця інформаційна панель, можна побачити на рисунках 4.3 та 4.4.

Рисунок 4.3 – Метрики кількості оброблених логів за хвилину

Рисунок 4.4 – Метрики виявлення нових шляхів

4.2 Результати тестування та їх аналіз

4.2.1 Вплив різних аспектів логування даних на продуктивність моделювання графа

Важливо знати, які саме фактори погіршують продуктивність та визначити вплив відформатованих даних журналу на використання ресурсів комп'ютера. Для цього будуть розглядатися самі дані журналу, алгоритм і структура даних. Параметри, які описують дані журналу та ознаки у вимірюваній формі виглядають наступним чином:

- довжина статичної символьної частини ідентифікатора журналу;
- довжина шляху в графі;
- кількість різних шляхів у графі;
- комбінація кількості шляхів та довжини шляхів;
- кількість оброблених ознак.

Для аналізу впливу різних аспектів логування даних журналу було проведено ряд тестів з різними вхідними даними. Ці тести будуть виконувалися на одному

екземплярі шляхом запуску модульних тестів.

Результати вимірюються в рядках журналу, що обробляються щосекунди, і можуть бути використані при визначенні скільки екземплярів потрібно розгорнути, в той час як результати показують, коли один екземпляр досягає свого максимуму. У таблиці 4.1 наведено вхідні дані для кожного експерименту, що виконується.

Довжина статичної символьної частини ідентифікатора журналу враховується, тоді як дерево префіксів використовує символи статичної частини в даних журналу для побудови шляхів у графі. Очікується, що коли статична частина рядка журналу стає довшою, обчислення для обробки та вставки її в графік також триватимуть довше. Виміряти це можна шляхом генерації рядків журналу зі статичною частиною фіксованої довжини та вимірювання кількості рядків, які будуть оброблені в діапазоні часу. Цей процес буде повторено з різною фіксованою довжиною та порівнянням кількості рядків журналу, оброблених за секунду для кожної з процедур.

Таблиця 4.1 – Вхідні дані для тестування

№ п/п	Кількість шляхів	Глибина шляху	Довжина статичної частини	Діапазон кількості ознак
1	1	10	10 - 1000000	100 - 1000000
2	1	10 - 1000	10	100 - 1000000
3	10 - 1000	10	10	100 - 1000000
4	10 - 1000	10 - 1000	10	100 - 1000000

Довжина шляху на графі визначається кількістю записів журналу для ознаки. Незважаючи на те, що кількість логів в опублікованому додатку може бути визначена, немає стандарту щодо того, скільки рядків журналу міститиме ознака. Побудувавши тест, у якому кількість рядків журналу в кожній ознаці є фіксованою, можна знову виміряти кількість рядків журналу, оброблених за одиницю часу. Зміна кількості рядків журналу на ознаку в кожній повторюваній процедурі призведе до отримання уявлення про те, яка глибина шляху сприяє покращенню

або погіршенню продуктивності обробки даних журналу.

Кількість різних шляхів на графіку визначається кількістю різних послідовностей у всіх оброблених ознаках. Кількість різних шляхів може бути згенерована шляхом створення ознак, які мають фіксовану кількість різних послідовностей в статичних частинах повідомлень, що йдуть один за одним. Повторюючи цей експеримент кілька разів для зростаючої кількості різних послідовностей, результат можна знову виміряти в оброблених рядках каротажу за певний проміжок часу. До тих пір поки відбуватиметься тестування різної кількість шляхів, слід зазначити, що експеримент для 1000 шляхів можна починати лише з 1000 різних ознак.

Кількість оброблених ознак буде вимірюватися для кожного тесту, описаного вище. Це означає, що кожен тест буде виконуватися зі зростаючою кількістю ознак на вході. Для зменшення фактору вхідного лагу всі траси будуть згенеровані до початку експерименту та збережені в оперативній пам'яті комп’ютера, а потім подані на бенчмаркінговий тест. В результаті будуть отримані вимірювання, які покладатимуться виключно на обробку ознак, а не на створення ознак.

Поєднання кількості шляхів і довжини шляхів буде експериментованим, щоб отримати загальне уявлення про продуктивність, коли кілька факторів збігаються разом. Це може означати, що реальна продуктивність буде такою ж.

На рисунку 4.5 чітко видно, що глибина шляхів на графі має найсуттєвіший вплив на продуктивність. При збільшенні ознак глибина графа збільшується і для його побудови виконується більше операцій. Однак, вхідні дані, що обробляються – це в основному ті ж самі шляхи, які замикають структуру даних. Коли вхідні дані розподілений між декількома екземплярами, блокування структури даних стає меншою проблемою, і продуктивність знову зростає.

Окрім розміру ознак, збільшення статичного ідентифікатора призводить до такого ж зниження продуктивності. Ребра побудовані зі статичних частин, тому це зводиться до тієї ж проблеми, що описана вище. У сценаріях реального світу статичний ідентифікатор буде обрано якомога меншим, щоб не захаращувати дані журналу, тому це не буде проблемою для даного підходу.

На рисунку 4.5 також можна побачити, що кількість шляхів не впливають на продуктивність заданого підходу щодо швидкості обробки. Розмір графа збільшується, тому використовується більше оперативної пам’яті. Однак, це ніяк не впливає на швидкість обробки даних журналу.

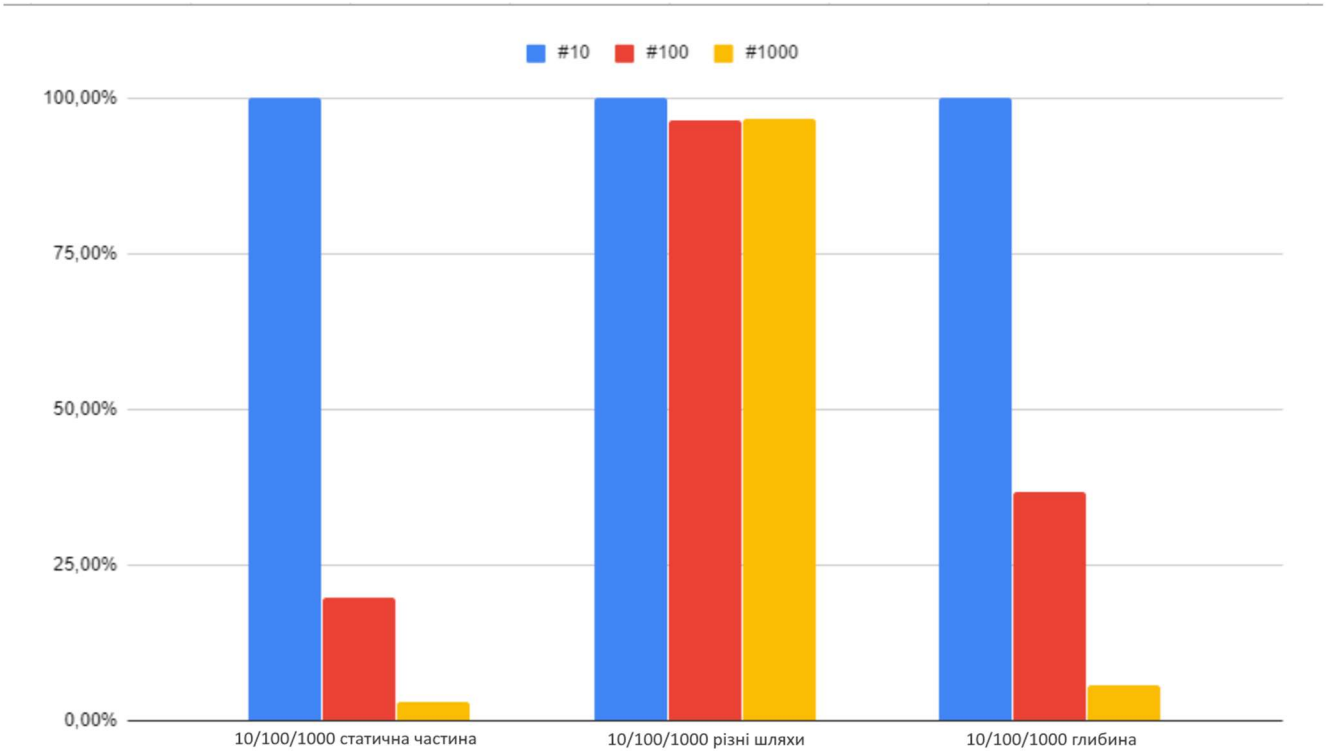


Рисунок 4.5 – Продуктивність при різних вхідних даних для конкретних даних журналу, статичного ідентифікатора, шляхів та глибини

Отже, глибина деревовидної структури на графі має найбільший вплив на продуктивність. Довші ознаки призводять до більшої необхідності в обчислювальних ресурсах. Різні ознаки, що генерують різні шляхи у структурі даних, не призводять до сповільнення виконання.

4.2.2 Вплив кількості даних журналу на роботу системи

Щоб перевірити масштабованість реалізованого рішення, воно буде перевірено на різних обсягах даних журналу. Це буде зроблено в середовищі,

налаштованому з генераторами даних журналу, розробленими для генерації певної кількості повідомлень журналу зі змінними вхідними параметрами на основі описаних вище факторів даних журналу. Подаючи дані журналу в систему у контрольований спосіб результати можна виміряти в кількості рядків журналу, оброблених за хвилину, а функції звітності можна контролювати та вимірювати таким чином, щоб вирахувати затримку впровадженого рішення.

Як видно з результатів тестування, показаних на рисунку 4.6, при збільшення кількості даних журналу не впливає на продуктивність. Це показує, що кількість рядків журналу, оброблених за хвилину, залишається стабільною з часом для будь-яких вхідних даних. Головний екземпляр виконує більше злиття, коли обробляються великі обсяги даних журналу, але злиття досить швидке для того, щоб не бути негативним фактором для масштабуванн. Звіти про шляхи та аномалії майже не показують затримки незалежно від обсягу даних журналу, які обробляються. На рисунку 4.7 показано час проходження в обидві сторони (RTT) для повідомлень, які проходять через NATS і отримують підтвердження.

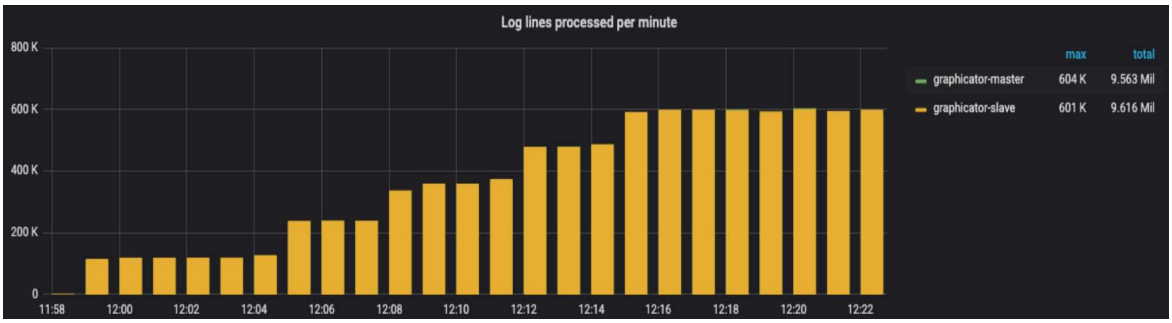


Рисунок 4.6 – Кількість оброблених рядків журналу за хвилину в Grafana з одним дочірнім екземпляром

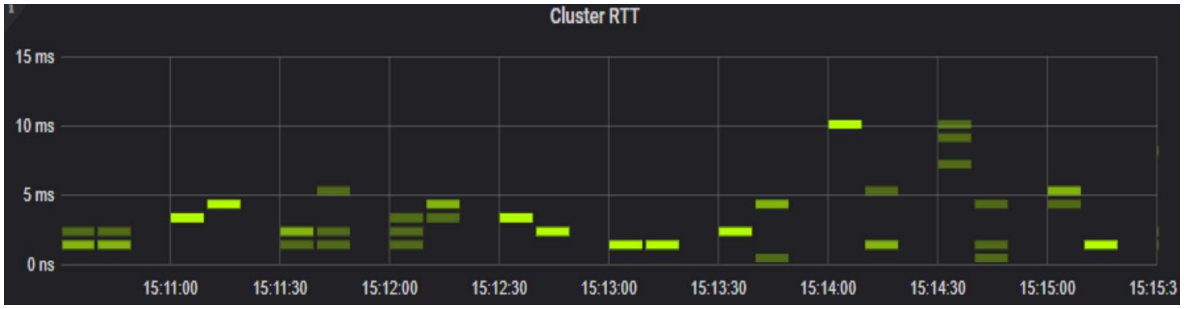


Рисунок 4.7 – Час проходження запитів STAN

Час, необхідний даному програмному засобу для звітування про аномалії, майже дорівнює часовому вікну для пакетування повідомлень і заданому RTT, що означає, що ці дані надходять майже миттєво.

Отже, при обробці більшої кількості даних журналу продуктивність не погіршується. Швидкість обробки даних журналу залишається стабільною при будь-якій кількості даних, що подаються в запропоноване рішення. Звітність про ознаки та аномалії майже не демонструє затримок навіть при зростаючій кількості даних журналу, що подаються в запропоноване рішення.

4.2.3 Оцінка ефективності запропонованого рішення

Щоб отримати уявлення про продуктивність системи, буде налаштовано кластер, описаний у попередньому розділі. Завдяки цьому Google Cloud Monitoring дає уявлення про час використання процесора та його завантаження за хвилину. Крім того, він також надасть використовуваний обсяг ОЗП та диска. Окрім того, додатково налаштовані експортери для Prometheus, надають додаткову інформацію про розгорнутий кластер NATS, щодо кількості отриманих і надісланих повідомлень, встановлених з’єднань, кількості байтів, що проходять через кластер, і часу проходження повідомлень. Крім того, створене рішення також експортуватиме рядки журналу, трасування та нові шляхи до Prometheus. Зрештою, усі ці показники можна переглянути або в інтерфейсі Google Cloud Console, або в Grafana, який візуалізує показники, які були експортовані в Prometheus.

У першу чергу увага при вимірюванні продуктивності буде зосереджена на самостійно створеній реалізації, тоді як інші частини системи є взаємозамінними з іншими рішеннями, доступними на ринку. Однак, важливо пам’ятати, що ці частини є вирішальними для наскрізного процесу роботи з даними журналу, тому для цих частин буде проведено вимірювання та перевірка, щоб вирішити, чи не стануть вони вузьким місцем у всій системі. Зрештою, слід зазначити, що архітектура повного конвеєра налаштована таким чином, що кожен модуль можна

масштабувати, у той час як усі вони вибрані для підтримки розподіленої обробки даних журналу.

Показники, які найбільше впливають на кластер при роботі з великими обсягами даних журналу, це час використання процесора, використання оперативної пам’яті та використання диска для NATS/STAN. Першим екземпляром, який відповідає за створення рядків журналу, є потоковий процесор конвеєра в поєднанні з трасувальником.

Використання ЦП досить низьке, що пояснюється тим, що він виконує дуже мало обчислень, що вцілому відповідає очікуванням. Використання оперативної пам’яті подвоюється з 70 тис. рядків до 140 тис. рядків через те, що потоковий процесор групує кількість вхідних повідомлень, а потім передає їх до трасувальника. Поточний процесор Benthos слід масштабувати раніше, ніж сам трейсер, але даний компонент не є обчислювально важким, якщо ми подивимося на інші компоненти в кластері.

NATS і його компонент STAN потребують набагато більше ресурсів кластера для обробки рядків журналу. NATS, однак, є лише порталом для пропускної здатності повідомлень без будь-якого складного контролю над повідомленнями, що проходять. Значне використання процесора NATS і STAN можна пояснити тим фактом, що було встановлено обмеження на кількість повідомлень, які зберігаються на диску, і постійно перевіряється, чи слід видаляти старіші дані журналу.

Обидва екземпляри використовують набагато більше операцій дискового введення-виведення, що не покращує продуктивність у цьому конкретному кластері, тоді як самі дискові операції досить повільні. Очікуванням на завершення цих операцій, пояснюється більша частина часу використання ЦП.

Фактична реалізація рішення структурування ознак використовує набагато менше ресурсів, ніж NATS. Отримані результати вказують на те, що дочірній екземпляр виконує обчислювальну частину, а головний екземпляр використовує дуже мало ресурсів у кластері.

Тим не менше, дочірній екземпляр використовує набагато більше оперативної пам’яті, ніж його основний аналог, у той час як він містить більше

інформації під час виконання, щоб мати можливість робити обчислення для вимірювання продуктивності та виявлення шляхів.

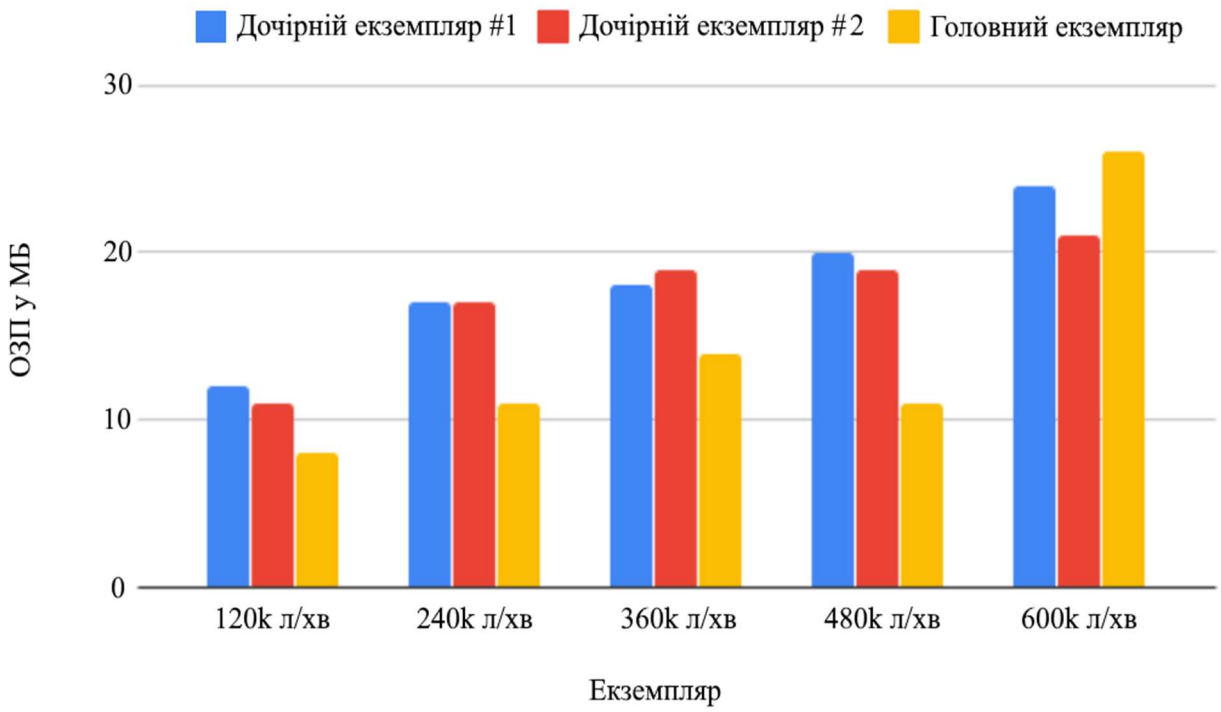


Рисунок 4.8 – Використання ОЗП головним та дочірніми екземплярами

Отже, час використання процесора є одним з показників, який має найбільший вплив на те, чи варто масштабувати ресурси. Рішення виграє за рахунок розподілення робочих навантажень і використовує невелику кількість часу процесора для виконання стандартних обчислень необхідних для побудови структури префіксних дерев. Використання оперативної пам'яті залишається на одному рівні між головним екземпляром і всіма дочірніми екземплярами.

4.2.4 Оцінка можливостей для горизонтального масштабування

Для вимірювання ефективності того, як система оброблятиме кількість даних журналу, які надходять у систему, використовуються показники описані в попередньому підрозділі. Хоча деякі компоненти, як, наприклад, потоковий

процесор і NATS, є готовими рішеннями, і їх можна замінити іншими рішеннями в цій галузі, експерименти з розподіленого обрахунків зосереджуватимуться насамперед на рішенні, реалізованому в рамках даного дослідження.

Щоб зосередитися на рішенні, реалізованому в цьому дослідженні, повні ознаки будуть створені відразу, замість їх представлення як окремих рядків журналу, щоб зменшити кількість і розмір повідомлення, яке проходить через NATS і STAN. Це призведе до того, що реалізоване рішення зможе обробляти більше трасувань і ліній журналу, а отже, стане кращим показником того, як реалізована система реагує на великі обсяги даних журналу.

Щоб дізнатися, як використана розподілена методологія впливає на продуктивність, дочірні частини системи масштабуються окремо. Такі показники, як рядки журналу, що обробляються щосекунди, а також усі вищезазначені показники можна переглядати окремо для кожного розгорнутого екземпляра та порівнювати один з одним. Це дає змогу системі надавати інформацію про те, чи рівномірно розподіляються рядки журналу між окремими екземплярами та як це впливає на використання ресурсів.

Важко визначити точний тригер, коли підхід має розширюватися для підвищення ефективності. Це залежить від наявних ресурсів. Однак найголовніше, на що варто звернути увагу, це те, що використовувана оперативна пам’ять в даному підході (продемонстрованому на рисунку 4.8) не є розподіленою, і кожен розгорнутий екземпляр, дочірній і основний, матиме однакове використання пам’яті. Також можна побачити, що використання оперативної пам’яті є низьким, але теоретично під час виконання складніших обчислень воно може зрости, і це потрібно враховувати в разі інтеграції подібних обчислень в дане рішення.

Використання ЦП майже порівну розподілено між дочірніми екземплярами. Дочірні екземпляри повинні досить легко розширятися з точки зору використання ЦП, коли вони досягають певного попередньо налаштованого порогу. Ще одна причина для розширення дочірніх екземплярів полягає в тому, що коли вхідні дані журналу в основному однакові у великих кількостях, багаторазове розгортання може бути швидшим з точки зору меншого блокування структури даних під час

виконання. Це буде компроміс із додатковою оперативною пам’яттю, яку використовуватимуть нові екземпляри. Головний екземпляр не використовує багато ресурсів для об’єднання часткових графів і майже не є фактором з точки зору масштабованості підходу.

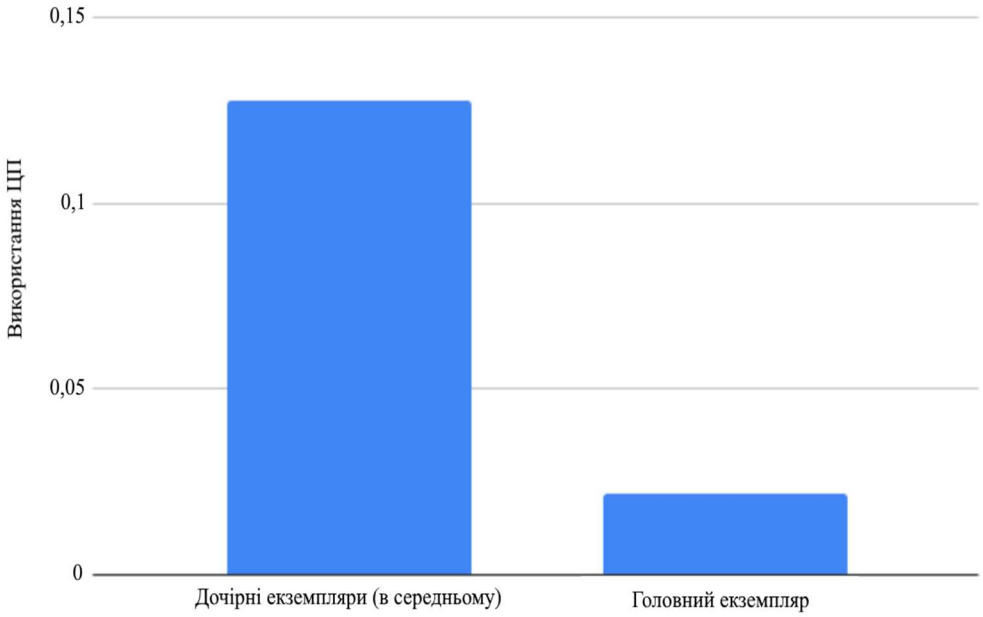


Рисунок 4.9 – Використання ЦП головним та дочірніми екземплярами

Отже, використання ЦП майже рівномірно розподілено між дочірніми екземплярами, в той час як використання оперативної пам’яті не розподілено. Автоматичне масштабування можна встановити, беручи до уваги обмеження ресурсів ЦП машини, на якій запущено дочірній екземпляр. Ще одна причина для збільшення дочірніх екземплярів полягає в тому, що вхідні дані журналу здебільшого однакові у великих обсягах, тому це може значно пришвидшити обробку даних журналу.

4.2.5 Оцінка збереження ефективності роботи підходу з плином часу

В даній частині система буде піддана стрес-тесту в рамках експерименту, щоб побачити, який обсяг даних журналу система може обробити під час роботи

на машинах, налаштованих для цього експерименту. Подаючи системі якомога більше ознак, структура даних буде розширена, а зниження продуктивності можна виміряти за допомогою згаданих раніше показників. Незважаючи на те, що перше тестування вже зможе дати розуміння найважливіших показників ефективності роботи з даними журналу, запуск і вимірювання в повноцінному середовищі зможуть підтвердити або спростувати попередні результати.

У поточній реалізації вбудовано кілька функцій, щоб отримати уявлення про приховані фактори поведінки програмного забезпечення. Дослідження Round-Trip-Time даних журналу, які надсилаються в рішення, доки вони не будуть структуровані, дає змогу досліджувати, наскільки добре рішення може повідомляти про аномалії в режимі реального часу. На даний момент однією з функцій є вимірювання продуктивності в наносекундах від вузла до вузла на кінцевому графіку, де нові оброблені шляхи можуть ініціювати створення звітів про зниження продуктивності. Час проходження в обидві сторони ознаки що деградувала у часі в такому випадку можна визначити дельтою точного моменту подачі деградованої ознаки та точного моменту звітування метрики. На додаток до попередніх умов для цього експерименту слід визначити одну ознаку, яка відрізняється дельтою з попередньо налаштованою межею дельти між вузлами. Коли це налаштовано, експеримент можна проводити неодноразово з різною кількістю рядків журналу, щоб побачити, чи не зменшується час проходження в обидва кінці під час обробки великих обсягів даних журналу.

Додатково буде порівняно базовий граф з вхідними ознаками з точки зору нових шляхів. У цьому випадку також можна виміряти Round-Trip-Time нових ознак, які надходять до системи, доки вона не повідомить про новий шлях. Це ще раз покаже, як система працює з даними журналу в режимі реального часу. Даний експеримент можна провести, використовуючи згадані вище передумови та надсилаючи в систему ознаки з іншим шляхом у порівнянні з базовим графом. Цей показник дозволить рішенню повідомляти про відмінності в поведінці програмного забезпечення таким чином, щоб воно могло знайти зміни в шляхах для різних релізів додатку.

Продуктивність не знижується з часом, поки структура даних передається в головний екземпляр шляхом попереднього налаштування, і він може підтримувати вхідні дані журналу без використання надмірних ресурсів. Під час роботи протягом півгодини на фоні 1,2 мільйона рядків журналу за хвилину, що становить 10 тис. унікальних вузлів і 200 різних шляхів на графіку, необхідні ресурси ЦП не зменшуються з часом.

На рисунках 4.10 та 4.11 показано ресурси, які використовуються одним із чотирьох дочірніх екземплярів протягом певного часу. Можна зробити висновок, що необхідні ресурси ЦП залишаються стабільними з часом, якщо кількість рядків журналу однакова обробляється за хвилину. Використані ресурси оперативної пам'яті залишаються рівномірними з часом, тоді як нові шляхи та вузли не вводяться, а оброблені траси залишаються незмінними з часом. Таким чином, ми можемо зробити висновок, що програмне забезпечення, яке працює у виробництві, зрештою досягне певного використання ресурсів з точки зору обробки журналів, яке може охопити максимуми та мінімуми використання самого програмного забезпечення. Таким чином, можна знайти оптимальне налаштування кластера для роботи з великою кількістю програмного забезпечення та їх журналів у вихідних даних додатків.

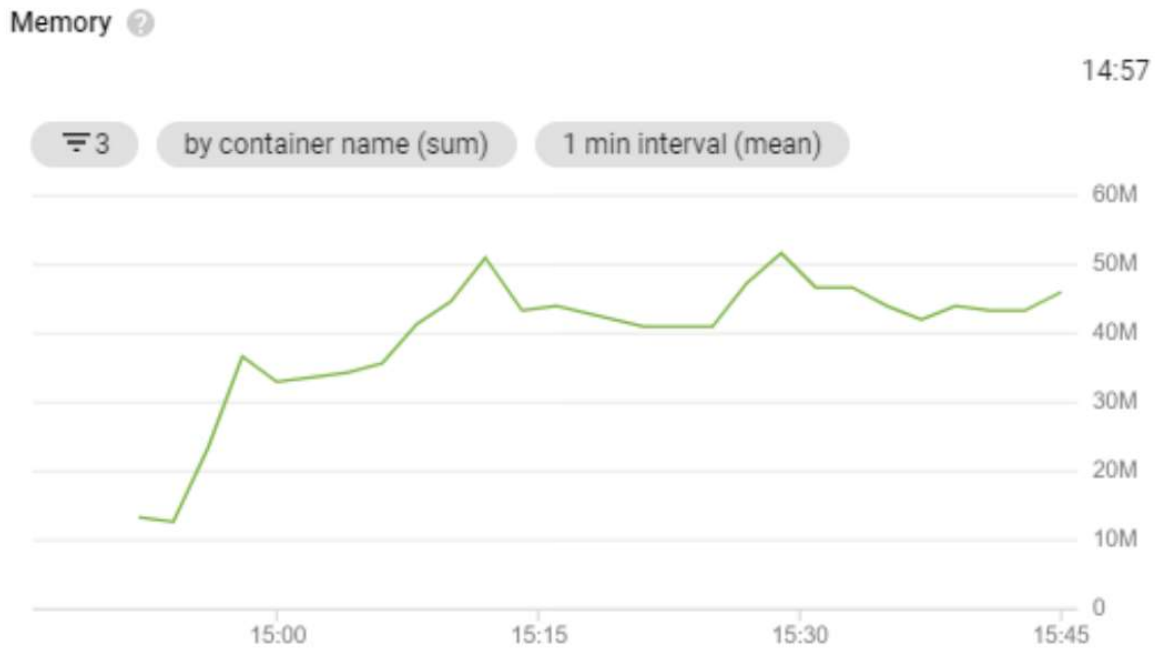


Рисунок 4.10 – Графік використання ОЗП дочірнім екземпляром

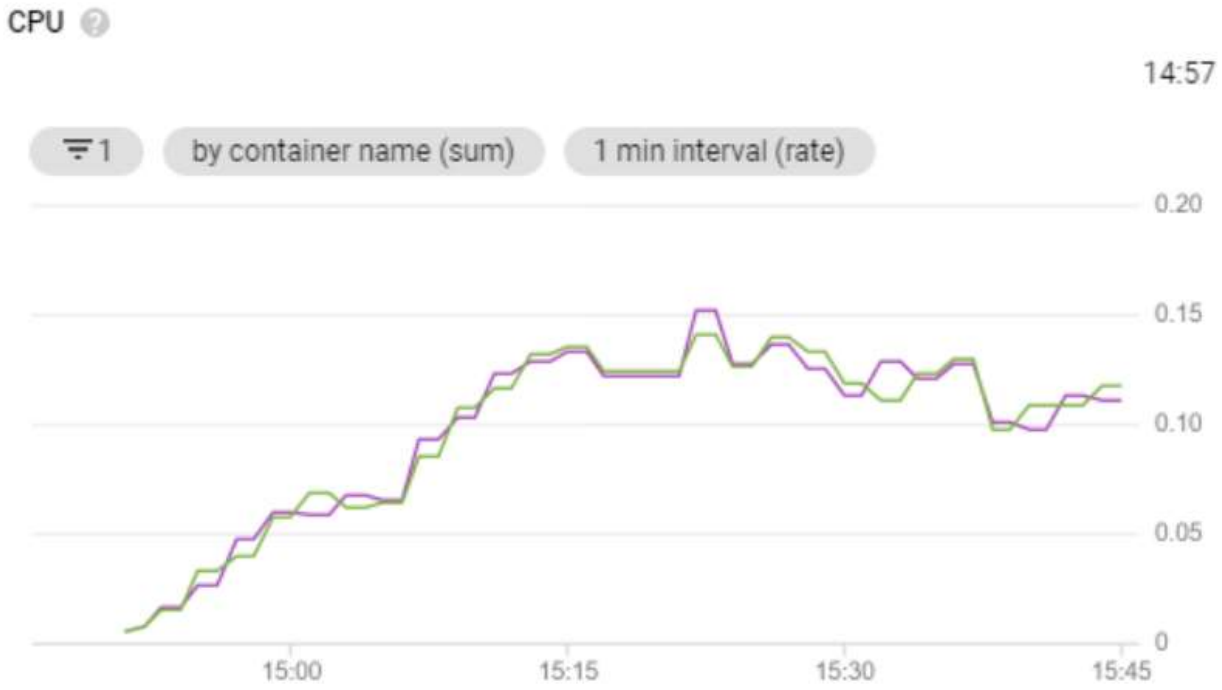


Рисунок 4.11 – Графік використання ЦП дочірнім екземпляром

На рисунках 4.12 та 4.13 показано використання ресурсів для головного екземпляру. Він зберігає кілька часткових графових структур у пам’яті протягом короткого періоду часу під час злиття, і тому йому необхідно більше пам’яті.

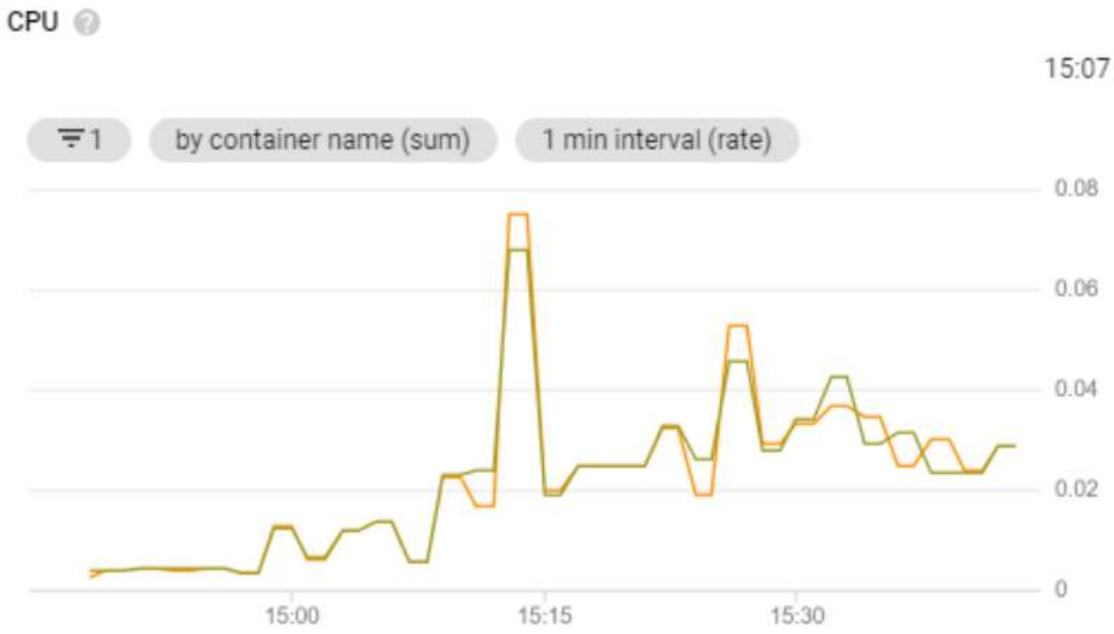


Рисунок 4.12 – Графік використання ЦП головним екземпляром

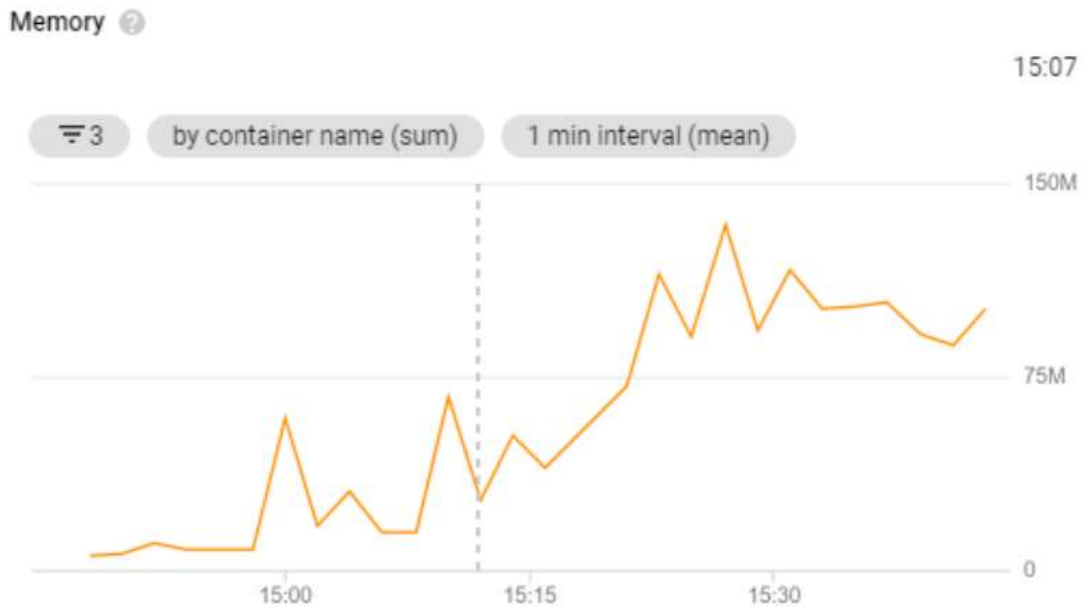


Рисунок 4.13 – Графік використання ОЗП головним екземпляром

У той час як у цьому експерименті використовувався кластер, який запускав NATS із змонтованим звичайним жорстким диском, можна помітити, що пропускна здатність повідомлень більше не може бути оброблена в реальному часі з точки зору дискових операцій.

У Grafana відстежувалися показники рядків журналу, оброблених за хвилину (див. рисунок 4.14), і пропускна здатність мережі (див. рисунок 4.15), і можна побачити, що після 25 хвилин обробки 1,2 мільйонів рядків журналу за хвилину починає тротлити через STAN. Однак STAN можна налаштувати по-іншому, як із твердотілим накопичувачем (SSD) або базою даних, щоб покращити пропускну здатність. Найкраща продуктивність буде досягнута, якщо NATS і STAN працюватимуть на окремо виділених машинах, а не в хмарному кластері, що надається від Google. Після надсилання конкретної ознаки декілька разів, коли вже оброблялося 1,2 мільйона рядків журналу на хвилину, і головний екземпляр повідомив точний момент часу, коли її було отримано, було зроблено висновок, що середній час між надсиланням ознаки та звітом становив 8 мс. На рисунку 4.7 пояснюється, чому існує невелика затримка в надсиланні та отриманні повідомлень, але це вказує на те, що навіть при роботі з великими обсягами журнальних даних виявлення аномалії все одно відбувається майже миттєво.

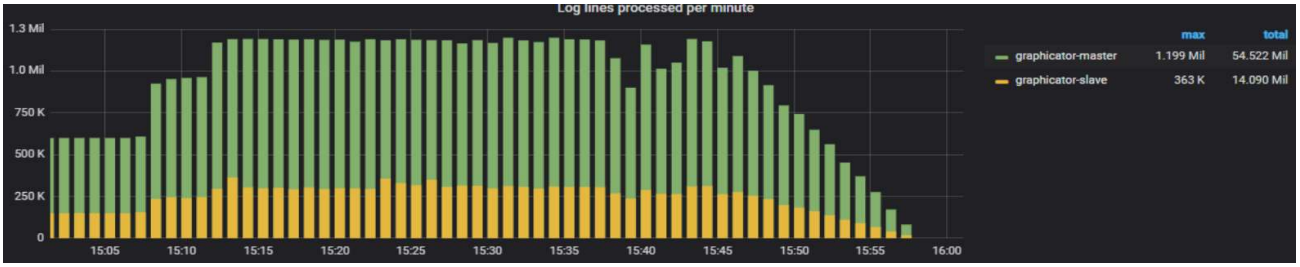


Рисунок 4.14 – Тротлинг STAN

Рисунок 4.15 – Пропускна здатність мережі при 1,2 млн. логів в хвилину

Отже, продуктивність роботи рішення не погіршується з часом. Запуск рішення протягом півгодини із темпом 1,2 мільйона рядків журналу за хвилину зберіг продуктивність на рівні початку роботи.

4.3 Висновки

У даному розділі були представлені компоненти та модулі розробленої архітектури. Дане рішення було розроблено для створення середовища в якому аналіз даних журналу може масштабуватися по горизонталі для роботи з великою кількістю операторів журналу.

Пропонована архітектура включає в себе декілька частин, кожна з яких може горизонтально масштабуватися. Представлене рішення працює через масштабовані блоки, які будують багаточастковий граф і можуть виконувати над ним обчислення, та один основний екземпляр, який періодично дістає часткові графи та контролює повідомлення про аномалії.

Було виявлено, що глибина шляхів на графі, яка дорівнює довжині ознак, має найбільший вплив на продуктивність обробки даних журналу. Довші трасування

призводять до більшої потреби в обчислювальних ресурсах. Різні ознаки, що призводять до різних шляхів у структурі даних, не призводять до сповільнення виконання.

Також було доведено що продуктивність рішення не знижується, при обробці великої кількості даних журналу. Цей процес залишається стабільним незалежно від кількості даних, що подається в запропоноване рішення. Окрім того, затримка для звітування залишається в межах мілісекунд, що є дуже хорошим результатом.

В результаті проведених тестувань було виявилено, що використання ЦП і диска для системи видавця/підписника у запропонованій архітектурі споживала найбільше ресурсів. Дочірні екземпляри реалізованого рішення спільно використовують ЦП при масштабуванні по горизонталі. Під час виконання складніших обчислень у дочірніх екземплярах слід враховувати використання оперативної пам’яті, хоч вона й не є спільною.

Окрім того було досліджено можливості для горизонтального масштабування. Було виявлено що коли зчитується велика кількість однакових інструкцій журналу, раннє масштабування дочірніх екземплярів може зменшити затримку обробки журналу. У більшості випадків найважливішим показником, який слід брати до уваги, є використання ЦП, а дочірні екземпляри можна масштабувати на основі його порогового значення.

Також в результаті проведення експерименту було підтверджено що продуктивність не знижується з часом під час стрес-тестування на відносно невеликому доступному кластері.

ВИСНОВКИ

У першому розділі дипломної роботи було досліджено та проаналізовано галузь логування даних, останні публікації, наукові статті та джерела, існуючі методи та способи організації процесів логування та знаходження аномалій з метою виявлення невирішених проблем. В результаті дослідження було встановлено що наявні механізми для логування та аналізу журнальних повідомлень все ще недосконалі і потребують покращень для збільшення швидкодії та покращення розширюваності, особливо в контексті роботи із високонавантаженими системами.

У другому розділі роботи проаналізовано моделі, концепції та алгоритми вирішення проблем виявлення аномалій. Нерозв’язані задачі запропоновано вирішити шляхом впровадження нової архітектури, в якій аналіз даних журналу може масштабуватися по горизонталі для роботи з великою кількістю операторів журналу. Для аналізу повідомлень журналу були використані техніки розподілених обчислень, що дозволило отримувати результати у реальному часі або з мінімальними затримками. Запропонована архітектура може легко масштабуватися у разі необхідності обробки ще більшої кількості даних.

У третьому розділі було проведено аналіз вимог до програмного засобу. Запропоноване рішення було описане в абстрактному та загальному вигляді. Виокремлено та визначено як модулі різні частини запропонованого рішення, які є або обов'язковими, або необов'язковими. Для кожної частини або модуля запропонованого рішення буде надано його опис, а також спосіб його взаємодії з іншими частинами системи.

У четвертому розділі описано програмну реалізацію та результати навантажувального тестування розробленої системи на основі запропонованої архітектури. Також проведено емпіричне дослідження, спрямоване на доведення працездатності розробленого програмного засобу та його функціональної придатності. Були проведені навантажувальні тестування, які показали що дане рішення здатне обробляти великі обсяги даних в режимі реального часу та що

розроблена система не втрачає продуктивності з часом в ході своєї роботи.

Низькі затримки дозволяють швидко обробляти нові журнальні повідомлення. В свою чергу, це дозволяє своєчасно виявляти проблеми та їх причини, що дає змогу швидко відновити працездатність застосунку в разі виникнення помилок.

Оскільки розроблене рішення не потребує особливих налаштувань, воно може бути досить легко інтегроване у проект, що дозволить збільшити стабільність його роботи шляхом своєчасного виявлення аномалій. Динамічне додавання розрахунків і нових правил до запропонованого рішення може зробити цей інструмент більш актуальним в галузі.

Отже, результати дослідження за темою дипломної роботи мають наукову новизну та практичну цінність.

Копія наукової публікації та довідка про прийняття статті до друку подані у додатку Б.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Sumo Logic cloud based log analysis platform [Online] // Tech Crunch. – Available: https://techcrunch.com/2017/06/27/sumo-logic-lands-75-million-series-f-on-the-road-to-ipo/?guccounter=1.

2. Elastic software monitoring stack [Online] // Investors – Available: https://www.investors.com/news/technology/elastic-ipo-initial-public-offering/.

3. Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. Failure diagnosisusing decision trees. In IEEE International Conference on Autonomic Computing, 2004.

4. Wei Xu, Ling Huang, and and Michael I Jordan. Experience mining google's production console logs, 2010.

5.Kamal Kc and Xiaohui Gu. Elt: Efficient log-based troubleshooting system for cloud computing infrastructures. In IEEE Symposium on Reliable Distributed Systems, 2011.

6. Ding Yuan. Improving software diagnosability via log enhancement. ACM Transactions on Computer Systems (TOCS), 30(1):4, 2012.

7. Qiang Fu et al. Advanced logging. Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014. 2014.

8. Boyuan Chen and Zhen Ming (Jack) Jiang. "Characterizing logging practices in Java-based open source software projects - a replication study in Apache Software Foundation". In: Empirical Software Engineering, 2017.

9. Ding Yuan, Soyeon Park, and Yuanyuan Zhou. "Characterizing logging practices in open-source software". In: 34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland, June 2-9, 2012. 2012.

10. R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A Cost-aware Logging Mechanism for Performance Diagnosis," in Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC), 2015.

11. W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, “Understanding Log Lines Using Development Knowledge,” in Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014.

12. W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, “Detecting large-scale system problems by mining console logs.,” pp. 37–46, 01 2010.

13. A. Chuvakin and G. Peterson, "How to Do Application Logging Right" IEEE Security & Privacy, vol. 8, no. 4, July/Aug. 2010, pp. 82-85.

14. R. Marty, ”Cloud application logging for forensics,” Proc. 2011 ACM Symp. Applied Computing, ACM, March 2011, pp. 178-184

15. OverOps, “The complete guide to Java logging in production” 2017. [Online]. Available: https://land.overops.com/java-logging-in-production-ebook/

16. W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," 2009, pp. 117–132. [Online]. Available: https://doi.org/10.1145/1629575.1629587

17. Apache Log4j 2 [Online]. Available: http://logging.apache.org/log4j/2.x/

18. Simple Logging (SL4J) [Online]. Available: http://www.slf4j.org/

19. Logback Project [Online]. Available: http://logback.qos.ch/

20. Java.Util.Loging Package General Description [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/logging/packagesummary.html

21. Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In 2017 IEEE International Conference on Web Services (ICWS), pages 33–40. IEEE, 2017

22. Kang Sun, Luoming Meng, Shaoyong Guo, Siya Xu, Ying Wang, and Weijian Li. An approach of anomaly diagnosis with logs for distributed services in communication network information system, p. 938–940. IEEE, 2017.

23. LOGSTASH [Online] // Logstash Enchanced – Available https://logz.io/logstash-td/

24. Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. Empirical software engineering, 18(4):791–824,

2013.

25. Rick Wieman. What Does Passive Learning Bring To Adyen? PhD thesis, Master's thesis, Delft University of Technology, the Netherlands, 2017.

26. Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant constrained models. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 267–277, 2011.

27. Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D Ernst, and Arvind Krishnamurthy. Unifying fsm-inference algorithms through declarative specification. In 2013 35th International Conference on Software Engineering (ICSE), pages 252–261. IEEE, 2013.

28. Marijn JH Heule and Sicco Verwer. Exact dfa identification using sat solvers. In International Colloquium on Grammatical Inference, pages 66–79. Springer, 2010.

29. Imam Yagoub, Muhammad Asif Khan, and Li Jiyun. It equipment monitoring and analyzing system for forecasting and detecting anomalies in log files utilizing machine learning techniques. In 2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD), pages 1–6. IEEE, 2018.

30. Microsoft SQL [Online]. Available: https://docs.microsoft.com/en-us/sql/sql-serverlsql-server-technical documentation.

31. Tao Qin, Yuli Gao, Lingyan Wei, Zhaoli Liu, and Chenxu Wang. Potential threats mining methods based on correlation analysis of multi-type logs. IET Networks, 7(5): 299–305, 2017.

32. Perttu Halonen, Markus Miettinen, and Kimmo Hat¨ onen. Computer log anomaly detection using frequent episodes. In IFIP International Conference on Artificial Intelligence Applications and Innovations, pages 417–422. Springer, 2009.

33. Java Logging Tools and Libraries [Online]. Available: http://www.java-logging.com/

34. .NET Logging Tools and Libraries [Online]. Available: http://www.dotnetlogging.com/

35. Apache log4net project, .NET Logging [Online]. Available:

https://logging.apache.org/log4net/

36. NLog, Flexible & free open-source logging for .NET, [Online]. Available: https://logging.apache.org/log4net/

37. PostSharp Diagnostics: Logging and Tracing [Online]. Available: https://www.postsharp.net/diagnostics/net-logging

38. S. Clarke, W. Harrison, H. Ossher and P. Tarr, “Subject-oriented design: towards improved alignment of requirements, design, and code,” 1999, pp. 325-339

39. Капітанець С. О. Керування логуванням у програмних системах: концептуальні засади / С. О. Капітанець, Г. І. Радельчук // Матеріали II Міжнародної наукової конференції «Розвиток наукової думки постіндустріального суспільства: сучасний дискурс» (секція «Комп’ютерна та програмна інженерія»), 18 листопада 2022 р., Львів. – Вінниця: Європейсь-ка наукова платформа, 2022. – С. 223-228. URL: https://archive.mcnd.org.ua/index.php/conference-proceeding/issue/view/18.11.2022

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД

А1. Програмний код файлу task_service.go

```
package services

import (
	"bytes"
	"encoding/gob"
	. "github.com/GoCollaborate/src/artifacts/task"
)

func Encode(maps *map[int]*Task) (payload *TaskPayload, err error) {
	var maps_bytes bytes.Buffer

	enc := gob.NewEncoder(&maps_bytes)

	err = enc.Encode(maps)

	payload = &TaskPayload{
		Payload: maps_bytes.Bytes(),
	}
	return
}
func Decode(payload *TaskPayload) (maps *map[int]*Task, err error) {
	dec := gob.NewDecoder(bytes.NewReader(payload.GetPayload()))
	err = dec.Decode(&maps)
	return
}
```

А2. Програмний код файлу client_stub.go

```
package collaborator

import (
	"github.com/GoCollaborate/src/artifacts/card"
	"github.com/GoCollaborate/src/artifacts/message"
	"github.com/GoCollaborate/src/artifacts/task"
	. "github.com/GoCollaborate/src/collaborator/services"
	"github.com/GoCollaborate/src/constants"
	"github.com/GoCollaborate/src/logger"
	"golang.org/x/net/context"
	"google.golang.org/grpc"
)

type ServiceClientStub struct {
	RPCServiceClient
}

func NewServiceClientStub(endpoint string, port int32, secure ...bool)
(stub *ServiceClientStub, err error) {
	if len(secure) < 1 {
		clientContact := card.Card{endpoint, port, true, "", false}

		conn, err := grpc.Dial(
			clientContact.GetFullIP(),
			grpc.WithInsecure(),
			grpc.WithBlock(),
			grpc.WithTimeout(constants.DEFAULT_RPC_DIAL_TIMEOUT),
		)

		if err != nil {
			logger.LogError("Dialing:", err)
			logger.GetLoggerInstance().LogError("Dialing:", err)
```

```
                return &ServiceClientStub{}, err
            }

            return &ServiceClientStub{NewRPCServiceClient(conn)}, nil
      }
      // todo: change return to TLS client
      return &ServiceClientStub{}, nil
}

func (stub *ServiceClientStub) DistributeAsync(source *map[int]*task.Task)
chan *task.Task {
      ch := make(chan *task.Task)

      go func() {
            defer close(ch)
            enc, _ := Encode(source)
            dec, err :=
stub.RPCServiceClient.Distribute(context.Background(), enc)
            if err != nil {
                  logger.LogError("Connection Error:" + err.Error())
                  return
            }
            result, _ := Decode(dec)
            for _, t := range *result {
                  ch <- t
            }
      }()

      return ch
}

func (stub *ServiceClientStub) Exchange(in *message.CardMessage)
(*message.CardMessage, error) {
      return stub.RPCServiceClient.Exchange(context.Background(), in)
```

```
}
```

А3. Програмний код файлу collaborator.go

```
package collaborator

import (
	"fmt"
	"github.com/GoCollaborate/src/artifacts/card"
	"github.com/GoCollaborate/src/artifacts/iexecutor"
	"github.com/GoCollaborate/src/artifacts/iworkable"
	"github.com/GoCollaborate/src/artifacts/message"
	"github.com/GoCollaborate/src/artifacts/stats"
	"github.com/GoCollaborate/src/artifacts/task"
	"github.com/GoCollaborate/src/cmd"
	"github.com/GoCollaborate/src/constants"
	"github.com/GoCollaborate/src/helpers/cardHelper"
	"github.com/GoCollaborate/src/helpers/messageHelper"
	"github.com/GoCollaborate/src/logger"
	"github.com/GoCollaborate/src/store"
	"github.com/GoCollaborate/src/utils"
	"github.com/GoCollaborate/src/web"
	"github.com/gorilla/mux"
	"net/http"
	"strconv"
	"time"
)

const (
	MAX_IDLE_CONNECTIONS int           = 20
	REQUEST_TIMEOUT      time.Duration = 5
	UPDATE_INTERVAL      time.Duration = 1
```

```
)

// Collaborator is a helper struct to operate on any inner trxs.
type Collaborator struct {
	CardCase Case
	Workable iworkable.Workable
}

// Return an instance of Collaborator
func NewCollaborator() *Collaborator {
	return &Collaborator{*newCase(), iworkable.Dummy()}
}

func newCase() *Case {
	return &Case{cmd.Vars().CaseID, &Exposed{make(map[string]*card.Card),
time.Now().Unix()}, &Reserved{*card.Default(), card.Card{}}}
}

// Join a master to the collaborator network.
func (clbt *Collaborator) Join(wk iworkable.Workable) {
	clbt.Workable = wk

	err := clbt.CardCase.readStream()
	if err != nil {
		panic(err)
	}

	// update if the read in timestamp is later than now
	if now := time.Now().Unix(); clbt.CardCase.GetDigest().GetTimeStamp()
> now {
		clbt.CardCase.TimeStamp = now
		logger.LogWarning("The read in timestamp conflicts with local
clock")
```

```
            logger.GetLoggerInstance().LogNormal("The read in timestamp
conflicts with local clock")
      }

      logger.LogNormal("Case Identifier:")
      logger.GetLoggerInstance().LogNormal("Case Identifier:")
      logger.LogListPoint(clbt.CardCase.CaseID)
      logger.GetLoggerInstance().LogListPoint(clbt.CardCase.CaseID)

      ip := utils.GetLocalIP()

      cards := clbt.CardCase.GetDigest().GetCards()

      logger.LogNormal("Cards:")
      logger.GetLoggerInstance().LogNormal("Cards:")
      cardHelper.RangePrint(cards)

      local := clbt.CardCase.Local

      // update if self does not exist
      if card, ok := clbt.CardCase.Cards[local.GetFullIP()]; !ok ||
!card.IsEqualTo(&local) {
            clbt.CardCase.Cards[local.GetFullIP()] = &local
            clbt.CardCase.Stamp()
      }

      local.IP = ip

      logger.LogNormal("Local:")
      logger.GetLoggerInstance().LogNormal("Local:")
      cardHelper.RangePrint(map[string]*card.Card{local.GetFullIP():
&local})

      // start active message handler
```

```
	clbt.CardCase.Action()


	// launch RPC server
	LaunchServer(":"+strconv.Itoa(int(clbt.CardCase.Local.GetPort())),
clbt.Workable)


	go func() {
		for {
			clbt.Catchup()
			<-time.After(constants.DEFAULT_SYNC_INTERVAL)
			// clean collaborators that are no longer alive
			clbt.Clean()
			// dump data to local store
			clbt.CardCase.writeStream()
		}
	}()

}

// Catchup with peer servers.
func (clbt *Collaborator) Catchup() {
	var (
		c = &clbt.CardCase

		dgst  = c.GetDigest()
		cards = dgst.GetCards()

		max  = cmd.Vars().GossipNum
		done = 0
		seed = 0
	)

	for key, e := range cards {
```

```
            // stop gossipping if already walk through max nodes
            if done > max && seed > 0 {
                  break
            }

            if e.Alive && (!e.IsEqualTo(&c.Local)) {
                  client, err := NewServiceClientStub(e.IP, e.Port)

                  if err != nil {
                        logger.LogWarning("Connection failed while bridging")
                        logger.GetLoggerInstance().LogWarning("Connection
failed while bridging")
                        c.Cards[key].SetAlive(false)
                        c.writeStream()
                        continue
                  }

                  from := c.Local.GetFullExposureCard()
                  to := e
                  var in *message.CardMessage =
message.NewCardMessageWithOptions(
                        c.GetCluster(),
                        &from,
                        to,
                        dgst.GetCards(),
                        dgst.GetTimeStamp(),
                        message.CardMessage_SYNC,
                  )
                  var out *message.CardMessage = message.NewCardMessage()
                  var out2 *message.CardMessage = message.NewCardMessage()
                  // first exchange
                  out, err = client.Exchange(in)
                  // local update exchange
```

```
                out2, err = messageHelper.Exchange(out)
                // second exchange
                _, err = client.Exchange(out2)
                if err != nil {
                      logger.LogWarning("Calling method failed while
bridging")
                      logger.GetLoggerInstance().LogWarning("Calling method
failed while bridging")
                      continue
                }
           }

           // if the collaborator is not a seed, server should attempt to
contact a seed
           if e.IsSeed() {
                seed++
           }
           done++
     }
}

// Clean up the case, release terminated servers.
func (clbt *Collaborator) Clean() {

     cards := clbt.CardCase.GetDigest().GetCards()

     for k, c := range cards {
          if !c.Alive {
                clbt.CardCase.Terminate(k)
          }
     }

     cards = clbt.CardCase.GetDigest().GetCards()
```

```
	cardHelper.RangePrint(cards)
}

// Start handling server routes.
func (clbt *Collaborator) Handle(router *mux.Router) *mux.Router {

	// register dashboard
	router.HandleFunc("/", web.Index).Methods("GET").Name("Index")
	router.PathPrefix("/static/").Handler(http.StripPrefix("/static/",
http.FileServer(http.Dir(constants.LIB_UNIX_DIR+"static/")))).Name("Static"
)
	router.HandleFunc("/dashboard/profile",
web.Profile).Methods("GET").Name("Profile")
	router.HandleFunc("/dashboard/routes",
web.Routes).Methods("GET").Name("Routes")
	router.HandleFunc("/dashboard/logs",
web.Logs).Methods("GET").Name("Logs")
	router.HandleFunc("/dashboard/stats",
web.Stats).Methods("GET").Name("Stats")

	// register tasks existing in publisher
	// reflect api endpoint based on exposed task (function) name
	// once api get called, use distribute
	fs := store.GetInstance()
	for _, jobFunc := range fs.SharedJobs {
		logger.LogNormal("Job Linked:")
		logger.LogListPoint(jobFunc.Signature)
		logger.GetLoggerInstance().LogNormal("Job Linked:")
		logger.GetLoggerInstance().LogListPoint(jobFunc.Signature)

		// shared registry
		clbt.HandleShared(router, jobFunc)
	}

	for _, jobFunc := range fs.LocalJobs {
```

```
            logger.LogNormal("Job Linked:")
            logger.LogListPoint(jobFunc.Signature)
            logger.GetLoggerInstance().LogNormal("Job Linked:")
            logger.GetLoggerInstance().LogListPoint(jobFunc.Signature)
            // local registry
            clbt.HandleLocal(router, jobFunc)
      }
      return router
}

// Distribute tasks to peer servers, the tasks will be sequentially sent.
func (clbt *Collaborator) DistributeSeq(sources map[int]*task.Task)
(map[int]*task.Task, error) {

      var (
            cc                          = clbt.CardCase
            dgst                        = cc.GetDigest()
            l1                          = len(sources)
            l2                          = len(dgst.GetCards())
            result  map[int]*task.Task = make(map[int]*task.Task)
            counter int                 = 0
      )

      // task channel
      chs := make(map[int]chan *task.Task)

      if l2 < 1 {
            return result, constants.ERR_NO_PEERS
      }
      // waiting for rpc responses
      printProgress(counter, l1)

      for k, v := range sources {
```

```
            var l = k % l2
            e := cc.ReturnByPos(l)

            // publish to local if local Card/Card is down
            if e.IsEqualTo(&cc.Local) || !e.Alive {
                  // copy a pointer for concurrent map access
                  p := *v
                  chs[k] = clbt.DelayExecute(&p)
                  continue
            }

            // publish to remote
            service, err := NewServiceClientStub(e.IP, e.Port)
            if err != nil {
                  // re-publish to local if failed
                  logger.LogWarning("Connection failed while connecting")
                  logger.LogWarning("Republish task to local")
                  logger.GetLoggerInstance().LogWarning("Connection failed
while connecting")
                  logger.GetLoggerInstance().LogWarning("Republish task to
local")
                  // copy a pointer for concurrent map access
                  p := *v
                  chs[k] = clbt.DelayExecute(&p)
                  continue
            }

            // copy a pointer for concurrent map access
            s := make(map[int]*task.Task)
            s[k] = v
            chs[k] = service.DistributeAsync(&s)
      }
```

```
	for {
		for i, ch := range chs {
			select {
			case t := <-ch:
				counter++
				result[i] = t
				printProgress(counter, l1)
			}
		}
		if counter >= l1 {
			break
		}
	}

	logger.LogProgress("All task responses collected")
	logger.GetLoggerInstance().LogProgress("All task responses collected")
	return result, nil
}

// Handle local Job routes.
func (clbt *Collaborator) HandleLocal(router *mux.Router, jobFunc *store.JobFunc) {

	var (
		f = func(w http.ResponseWriter, r *http.Request) {
			var (
				bg      = task.NewBackground()
				counter = 0
			)

			jobFunc.F(w, r, bg)
```

```
                go func() {
                        job := bg.Done()
                        defer bg.Close()
                        for s := job.Front(); s != nil; s = s.Next() {
                                exes, err := job.Exes(counter)
                                if err != nil {
                                        logger.GetLoggerInstance().LogError(err)
                                        break
                                }
                                err = clbt.LocalDistribute(&s.TaskSet, exes)
                                if err != nil {
                                        logger.GetLoggerInstance().LogError(err)
                                        break
                                }
                                counter++
                        }
                }()
        }
        fs = store.GetInstance()
    )

    lim, err := fs.GetLimiter(jobFunc.Signature)
    if err == nil {
        f = utils.AdaptLimiter(lim, f)
    }

    f = utils.AdaptStatsHits(f)

    f = utils.AdaptStatsRouteHits(jobFunc.Signature, f)

    router.HandleFunc(jobFunc.Signature,
f).Methods(jobFunc.Methods...).Name("Local Tasks")
}
```

```
// Handle shared Job routes.
func (clbt *Collaborator) HandleShared(router *mux.Router, jobFunc
*store.JobFunc) {

	var (
		f = func(w http.ResponseWriter, r *http.Request) {
			var (
				bg      = task.NewBackground()
				counter = 0
			)

			jobFunc.F(w, r, bg)

			go func() {
				job := bg.Done()
				defer bg.Close()
				for s := job.Front(); s != nil; s = s.Next() {
					exes, err := job.Exes(counter)
					if err != nil {
						logger.GetLoggerInstance().LogError(err)
						break
					}

					err = clbt.SharedDistribute(&s.TaskSet, exes)
					if err != nil {
						logger.GetLoggerInstance().LogError(err)
						break
					}
					counter++
				}
			}()
		}
```

```
		fs = store.GetInstance()
	)

	lim, err := fs.GetLimiter(jobFunc.Signature)
	if err == nil {
		f = utils.AdaptLimiter(lim, f)
	}

	f = utils.AdaptStatsHits(f)

	f = utils.AdaptStatsRouteHits(jobFunc.Signature, f)

	router.HandleFunc(jobFunc.Signature,
f).Methods(jobFunc.Methods...).Name("Shared Tasks")
}

// The function will process the tasks locally.
func (clbt *Collaborator) LocalDistribute(pmaps *map[int]*task.Task, stacks
[]string) error {

	// stats
	sm := stats.GetStatsInstance()
	sm.Record("tasks", len(*pmaps))

	var (
		err  error
		fs   = store.GetInstance()
		maps = *pmaps
	)

	for _, stack := range stacks {
		var (
			exe iexecutor.IExecutor
```

```
            )
            exe, err = fs.GetExecutor(stack)

            if err != nil {
                  return err
            }

            switch exe.Type() {
            // mapper
            case constants.EXECUTOR_TYPE_MAPPER:
                  maps, err = exe.Execute(maps)
                  if err != nil {
                        return err
                  }
            // reducer
            case constants.EXECUTOR_TYPE_REDUCER:
                  maps, err = exe.Execute(maps)
                  if err != nil {
                        return err
                  }
            default:
                  maps, err = exe.Execute(maps)
                  if err != nil {
                        return err
                  }
            }
      }
      *pmaps = maps
      clbt.Workable.Enqueue(*pmaps)

      return nil
}
```

```
// The function will process the tasks globally within the cluster network.
func (clbt *Collaborator) SharedDistribute(pmaps *map[int]*task.Task,
stacks []string) error {

	// stats
	sm := stats.GetStatsInstance()
	sm.Record("tasks", len(*pmaps))

	var (
		err  error
		fs   = store.GetInstance()
		maps = *pmaps
	)
	for _, stack := range stacks {
		var (
			exe iexecutor.IExecutor
		)
		exe, err = fs.GetExecutor(stack)

		if err != nil {
			return err
		}

		switch exe.Type() {
		// mapper
		case constants.EXECUTOR_TYPE_MAPPER:
			maps, err = exe.Execute(maps)
			if err != nil {
				return err
			}
			maps, err = clbt.DistributeSeq(maps)
			// reducer
		case constants.EXECUTOR_TYPE_REDUCER:
```

```
                maps, err = exe.Execute(maps)
                if err != nil {
                        return err
                }
        default:
                maps, err = exe.Execute(maps)
                if err != nil {
                        return err
                }
        }
    }
    *pmaps = maps

    return nil
}

// Execute the tasks after satisfying some certain conditions.
func (clbt *Collaborator) DelayExecute(t *task.Task) chan *task.Task {
    ch := make(chan *task.Task)

    go func() {
        defer close(ch)
        err := clbt.Workable.Done(t)
        if err != nil {
            logger.LogError("Execution Error:" + err.Error())
            ch <- &task.Task{}
        }
        ch <- t
    }()
    return ch
}

func Delay(sec time.Duration) {
```

```
	tm := time.NewTimer(sec * time.Second)
	<-tm.C
}

func printProgress(num interface{}, tol interface{}) {
	logger.LogProgress("Waiting for task responses[" + fmt.Sprintf("%v",
num) + "/" + fmt.Sprintf("%v", tol) + "]")
	logger.GetLoggerInstance().LogProgress("Waiting for task responses["
+ fmt.Sprintf("%v", num) + "/" + fmt.Sprintf("%v", tol) + "]")
}
```

ДОДАТОК Б

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

МАТЕРІАЛИ
ІІ МІЖНАРОДНОЇ
НАУКОВОЇ КОНФЕРЕНЦІЇ

Міжнародний Центр Наукових Досліджень

РОЗВИТОК НАУКОВОЇ ДУМКИ ПОСТІНДУСТРІАЛЬНОГО СУСПІЛЬСТВА: СУЧАСНИЙ ДИСКУРС

18 ЛИСТОПАДА 2022 РІК
м. Львів, Україна

Вінниця, Україна
«Європейська наукова платформа»
2022

УДК 001 (08)
Р 64

https://doi.org/10.36074/mcnd-18.11.2022

Організація, від імені якої випущено видання:
ГО «Міжнародний центр наукових досліджень»

Голова оргкомітету: Рабей Н.Р.

Верстка: Білоус Т.В.
Дизайн: Бондаренко І.В.

Р 64 **Розвиток наукової думки постіндустріального суспільства: сучасний дискурс**: матеріали II Міжнародної наукової конференції, м. Львів, 18 листопада, 2022 р. / Міжнародний центр наукових досліджень. — Вінниця: Європейська наукова платформа, 2022. — 356 с.

ISBN 978-617-8037-97-0
DOI 10.36074/mcnd-18.11.2022

Викладено матеріали учасників II Міжнародної спеціалізованої наукової конференції «Розвиток наукової думки постіндустріального суспільства: сучасний дискурс», яка відбулася 18 листопада 2022 року у місті Львів.

УДК 001 (08)

ISBN 978-617-8037-97-0

СЕКЦІЯ XIV.
КОМП'ЮТЕРНА ТА ПРОГРАМНА ІНЖЕНЕРІЯ

КЕРУВАННЯ ЛОГУВАННЯМ У ПРОГРАМНИХ СИСТЕМАХ: КОНЦЕПТУАЛЬНІ ЗАСАДИ

Капітанець Степан Олександрович
здобувач вищої освіти факультету інформаційних технологій
Хмельницький національний університет, Україна

Радельчук Галина Іванівна
ORCID ID: 0000-0002-9728-4390
канд. техн. наук, доцент, доцент кафедри інженерії програмного забезпечення
Хмельницький національний університет, Україна

Програмні системи (ПС) ведуть журнали для моніторингу подій та свого поточного стану. Ведення журналів широко застосовується на практиці завдяки своїй простоті та ефективності. Такі журнали у процесі пошуку та усунення несправностей є дуже важливим та цінним джерелом інформації для розробників ПС, які можуть вивчати записані логи з метою розуміння стану системи, виявлення аномалії та знаходження першопричин їх виникнення.

Сучасні ПС мають тенденцію до масштабування і переходу до розподілених хмарних обчислень. Такі великі системи забезпечують роботу сервісів подібно до соціальних мереж, пошукових систем чи систем електронної комерції. Абсолютна більшість таких систем призначена для роботи у цілодобовому режимі, обслуговуючи при цьому мільйони користувачів по всьому світу. Будь-яке відключення (навіть тимчасове) або погіршення якості роботи таких сервісів обходиться дуже дорого, тому можливості своєчасно виявляти будь-які зміни та швидко з'ясовувати першопричину проблеми є надважливими. Проте такі системи у процесі своєї роботи генерують величезну кількість логів зі швидкістю в десятки гігабайт на годину. Такий обсяг даних надзвичайно складно обробити вручну, навіть за допомогою інструментів пошуку та фільтрації. Окрім того, логи зазвичай є неструктурованими або напівструктурованими рядками тексту і тому складно піддаються обробці алгоритмами (якщо не дотримуватись певних правил).

Проте журналювання може використовуватись не лише для виявлення аномалій, але також і для аудиту та виставлення рахунків. З іншого боку, всі ці варіанти використання передбачають вільний доступ до всіх відповідних журналів, що може бути проблематичним, враховуючи розподільний характер сучасних систем.

Логи можуть зберігатися на різних рівнях журналу, щоб можна було відразу визначити важливість повідомлення. Наприклад, логи можуть містити префікси «INFO», «WARN» чи «ERROR». Стандарту, які саме рівні повинні використовуватись та наскільки логи мають бути багатослівними, немає [1].

Типовий запис у журналі має містити мітку часу, рівень повідомлення та саме повідомлення. Мітка часу, як правило, представляє собою час у форматі Unix або UTC (Temps Universel Coordonne), що позначає точний час, коли запис було створено. Це дозволяє відновити правильну послідовність подій.

Неправильний або неточний код журналювання може призвести до плутанини або й навіть до більш серйозних проблем (таких, наприклад, як відчутне зниження продуктивності чи збій системи).

Окрім того, можна також виділити дві основні проблеми, які пов'язані з практикою ведення журналів програмного забезпечення (ПЗ):

– відсутність існуючих керівних принципів щодо створення високоякісного коду журналювання;

– складність підтримки та розвитку коду логування.

Недавні емпіричні дослідження показують, що існуючих керівництв по веденню журналів для комерційних [2] та систем з відкритим вихідним кодом [3] немає. Розробники пишуть код журналювання виключно на основі своїх власних міркувань. На відміну від решти коду, який може бути перевірений шляхом тестування, перевірити коректність журнальованих даних досить складно. Наприклад, на рис. 1 продемонстрована помилка, знайдена на одному із проаналізованих програмних проектів.

DFSClient.cs

V 4078:
```
LOG.warn("Failed to renew lease for " + clientName + " for "+ (elapsed / 1000)+ "
seconds (>= soft-limit ="+ (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000)+ " seconds.)
"+ "Closing all files being written ...",e)
```

V 5956:
```
LOG.warn("Failed to renew lease for " + clientName + " for "+ (elapsed / 1000)+ "
seconds (>= hard-limit ="+ (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000)+ " seconds.)
"+ "Closing all files being written ...",e)
```

Рис. 1. **Приклад помилки у коді журналювання даних**

Оскільки код журналювання сплутується з рештою коду, його дуже складно постійно оновлювати по ходу розвитку ПС.

Найпоширеніші помилки, з якими стикаються розробники у процесі логування є наступними:

– явне приведення типів або відсутність перевірки на NULL у динамічних логах, що може призвести до помилки у процесі виконання програми;

– неправильний рівень повідомлення (наприклад, DEBUG замість INFO, або WARN замість ERROR);

– некоректне виведення даних у випадку, коли у повідомлення вставляється об'єкт, у якого не визначений формат, зручний для читання людиною (наприклад, якщо у ньому не перевизначений метод ToString());

– повтори коду для логування з однаковими повідомленнями;

– занадто довгий код для логування.

Як вже було зазначено, розробники зазвичай покладаються на свої власні відчуття у питаннях організації процесів логування у ПС. У загальному можна виділити три таких питання:

– питання того, де саме потрібно логувати;

– питання того, що саме записувати у журнал;

– питання того, як саме потрібно логувати.

Питання того, де саме потрібно логувати, полягає у визначенні підходящих місць для журналювання інформації. Логування може бути виконано у різноманітних місцях програмного коду. До прикладу, це може бути всередині блоків умов чи всередині блоків try і catch, щоб отримати представлення про поведінку системи під час її роботи. Однак надмірне логування може призвести до додаткових витрат

ресурсів [4]. Тому розробникам необхідно бути вибірковими у плані визначення точок для журналювання і ставити їх там, де це матиме найбільшу користь.

Основна ідея питання того, що саме записувати в журнал, полягає у наданні достатньої інформації у повідомленні для логу. Статичні тексти мають надавати короткий опис контексту виконання, а динамічний вміст має вказувати на поточний стан виконання. При написанні статичного повідомлення його текст має бути чітким і легким для розуміння, а динамічні фрагменти повинні бути послідовними та актуальними [5].

Питання того, як саме потрібно логувати, полягає у необхідності розробки та підтримки високоякісного коду для журналювання. Ведення журналу – це наскрізна проблема, оскільки код для логування розкиданий по всій програмній системі та заплутаний у кодах функцій. Завжди слід мати на увазі і враховувати, що розробляти та підтримувати хороший код для журналювання по мірі росту проекту буде все складнішим, а тому найкращі практики для цього треба інтегрувати ще на початкових стадіях розробки ПЗ.

Однак, правильно складених повідомлень інколи може бути недостатньо для того, щоб однозначено ідентифікувати точне місце в коді, з якого його було створено (як мінімум – щоб зробити це швидко, а як максимум – у коді проекту може бути декілька місць, де викликається функція, яка генерує однакові логи). Наявність даних про трасування стеку значно спрощує налагодження, а тому їх також необхідно включати до логів.

Отже, першим напрямком є удосконалення методів, які використовуються для створення та збереження журнальних даних з метою покращення швидкодії ПС. Другим напрямком є розробка механізму для автоматичного виявлення аномалій серед великого обсягу моніторингових даних. Проаналізуємо, як та в якій мірі вирішуються ці завдання серед існуючих рішень.

Всі програми для аналізу логів обробляють події, що відповідають записам, які містяться в журналах (а точніше, викликаним оператором у вихідному коді). Однак, простежити у зворотньому порядку (тобто від даних журналу до його оператора у вихідному коді) є нетривіальним завданням. По-перше, створення події під час виконання впливає на продуктивність системи. Наприклад, такі фреймворки, як Log4j, дозволяють переглядати ім'я класу та номер рядка, з якого було створено даний запис. Однак, збір цієї інформації при кожному записі в журнал супроводжується втратою продуктивності. Це відбувається тому, що за лаштунками Log4j, отримавши рядок оператора журналу, генерує виключення і захоплює згенероване ним трасування стека. Розробники Log4j експериментували з іншими альтернативами, але поки що саме цей спосіб виявився найефективнішим. По-друге, повідомлення, що зберігаються у журналі, складаються з тексту у довільній формі.

Щодо вирішення проблеми аналізу журналів даних можна запропонувати декілька основних стратегій. Найпростіший спосіб полягає у використанні регулярних виразів, запитів та фільтрів. Однак цей метод не масштабується і є непрактичним. Він дозволяє виявляти лише одиничні збої ПС, тоді як ланцюжки відмов, які викликані декількома взаємопов'язаними компонентами, залишаться непоміченими. Журнал часто «забруднюється» не пов'язаними один з одним трасуваннями стеку, що робить дослідження ще складнішим та заплутанішим. Окрім того, ця стратегія забирає багато часу та не може бути автоматизованою, що робить її непрактичною для реального світу.

Більш інноваційні та масштабовані методи намагаються поєднати аналіз журналів з алгоритмами машинного навчання. Враховуючи високу доступність даних, алгоритм машинного навчання ідеально підходить для обробки і вилучення закономірностей та прогнозів з логів.

Виявлення аномалій – це область, пов'язана з аналізом журнальованих даних, в якій вже широко використовується машинне навчання. Основна проблема, що виникає з подібними методологіями, які застосовуються для аналізу логів, полягає в тому, що задача перетворення вільнотекстових повідомлень у значущі ознаки є нетривіальною [6].

Ще однією життєздатною стратегією є статичний програмний аналіз. Алгоритми статичного аналізу вже добре відпрацьовані та є ефективними. Вони здатні сканувати програму, щоб знайти шляхи, які можуть призвести до помилок. Основна проблема полягає в тому, що статичний аналіз обмежений розміром і складністю цільової системи.

Для вирішення проблеми аналізу журнальних даних також можна використовувати аналітичні моделі. Модель будується та відточується для конкретної системи. Експерти вручну визначають залежності, метрики та взаємозв'язки для розробки стратегії прогнозування. Основним недоліком такого підходу є те, що великомасштабні системи є занадто складними і часто змінюються, що може зробити модель неефективною вже через короткий проміжок часу.

Далі представлено короткий огляд найбільш значущих інструментів для аналізу повідомлень журналів.

Logstash – вільний та відкритий серверний конвейєр для обробки даних. Logstash може поглинати величезні обсяги даних з різних джерел, розбирати їх, а потім відправляти в будь-яке вихідне сховище. Logstash розроблений компанією Elasticsearch B.V. і є частиною стека ELK (Elasticsearch, Logstash, Kibana). Він створений спеціально для збору, аналізу та перетворення логів.

Splunk – платформа для розширеного доступу до даних, потужної аналітики та автоматизації. Інструмент надає безліч корисних функцій, які оптимізують управління та аналіз логів.

LogDNA – високомасштабоване рішення для управління журналами, яке індексує, агрегує та аналізує дані журналів. LogDNA може поглинати величезну кількість даних з різних джерел. Він також має повнофункціональний веб-інтерфейс, а також різні рішення для пошуку, синтаксичного аналізу, побудови графіків та оповіщення про дані журналів.

LogZ використовує стек ELK всередині для обробки логів. LogZ використовує рішення з відкритим вихідним кодом для виконання аналізу логів, моніторингу інфраструктури та розподіленого трасування.

GrayLog – комплексне рішення для управління в IT. Воно надає функціонал для управління безпекою та інфраструктурою, об'єднання, збагачення, кореляції, запитів і візуалізації всіх даних журналів в одному місці.

Порівняльна характеристика існуючих програмних рішень подана у табл. 1.

Таблиця 1

Порівняння популярних інструментів для аналізу логів

Інструмент	Достоїнства	Недоліки
Splunk	– найпопулярніше рішення; – багато плагінів; – багато інтегрованих рішень для логування; – хмарне рішення;	– дорогий; – із закритим кодом; – складна та крута крива навчання; – не дуже підходить для аналізу логів;

Продовження табл. 1

Інструмент	Достоїнства	Недоліки
Logstash	– створений спеціально для аналізу логів; – гнучкий та легкий в налаштуванні; – може бути запущеним як локально, так і у хмарі; – велика екосистема з плагінів; – безкоштовний та з відкритим кодом; – може приймати та виводити інформацію з/та в різні джерела;	– лише для аналізу; – немає інформаційної веб-панелі; – пов'язаний зі стеком ELK; – значне споживання ресурсів; – на практиці часто може мати недостатню продуктивність; – при навантаженні ядра на 100% відбувається збій у відправці даних;
LogDNA	– легке налаштування та інтеграція; – націлений на інтеграцію з інструментами DevOps; – потужний пошук та сповіщення; – агрегація логів та керування подіями;	– не безкоштовний; – користувачі повідомляють про періодичні затримки; – не користується популярністю та не розвивається спільнотою;
LogZ	– створений спеціально для моніторингу логів; – утиліти інтегровані у хмару;	– заснований на ELK; – не безкоштовний; – не користується популярністю;
Graylog	– хмарне рішення; – хороша документація; – легка мова запитів.	– не безкоштовний; – складно писати комплексні запити.

[авторська розробка]

Для того, щоб краще зрозуміти зручність використання та популярність вищезгаданих інструментів, було підраховано кількість трафіку в Google, що припадає на них. Результати продемонстровано на рис.2.

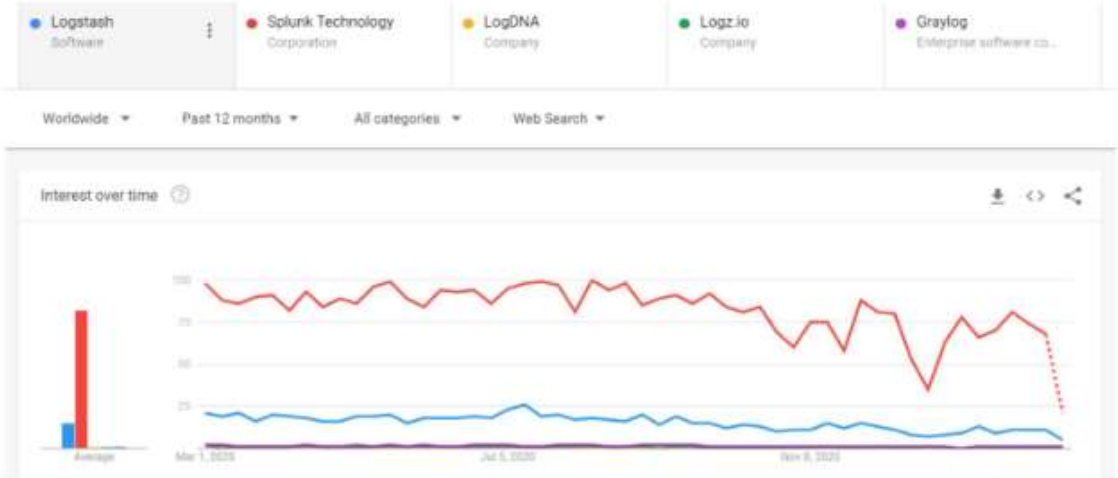


Рис. 2. **Популярність інструментів для аналізу логів**

Як видно з рис. 2, у трафіку на Google за останні 12 місяців домінують Splunk та LogStash. Splunk – це корпоративне рішення, в той час як LogStash – це ПЗ з відкритим вихідним кодом. Можна сказати, що популярність LogStash є досить стабільною протягом усього часового інтервалу, у той час як у Splunk Technologies спостерігається тенденція до зниження.

Таким чином, логування та аналіз журналів повідомлень визначаються як важливий та потужний інструмент у розробці та підтримці ПЗ. Вони є головним та дуже цінним джерелом інформації для розробників після запуску продукту у робоче середовище і дозволяють зрозуміти стан ПС, виявляти аномалії та знаходити першопричини їх виникнення.

Добре організований механізм логування разом з дотриманням основних правил їх оформлення та інструментом для аналізу аномалій не лише не нашкодить продуктивності системи, а й надасть додаткові бонуси у вигляді можливості моніторингу системи у реальному часі та стабільнішої роботи застосунку за рахунок своєчасного виявлення проблем.

Вивчення інструментів для аналізу логів показало, що LogStash є найпопулярнішим (і, до того ж, безкоштовним) інструментом, не дивлячись на загальні проблеми з продуктивністю та надійністю (можливе здвоєння показників та лавиноподібне зростання навантаження на сховище при спробах повторної відправки даних при високому завантаженні системи).

У ході аналізу предметної області встановлено, що наявні механізми для логування та аналізу журнальних повідомлень все ще є недосконалими і потребують покращень для збільшення швидкодії та інформативності. Для цього планується вирішити наступні завдання:

– збільшити фокус на дослідження методів статичного аналізу тексту для суттєвої оптимізації процесу трасування стеку, що використовується для виявлення точного місця у програмному коді, з якого було створено новий запис у журналі;

– визначити основні переваги методів аналізу логів з метою знаходження точок їх сходження, щоб вивести комплексний метод та покращити відмовостійкість системи для аналізу, моніторингу та виявлення аномалій у ПС;

– виконати програмну реалізацію бібліотеки для логування на основі удосконаленого методу.

Наступним кроком у вдосконаленні методів та процесів логування інформації у ПС стане виконання подальших досліджень з пошуком варіантів вирішення поставлених завдань.

Список використаних джерел:

1. Yuan, D., Zheng, J., Park, S., Zhou Y. & Savage, S. Improving Software Diagnosability via Log Enhancement. *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*. March 5-11, 2011, Newport Beach, CA, USA.
2. Qiang Fu, Jieming Zhu, Wenlu Hu & Jian-Guang Lou. Where Do Developers Log? An Empirical Study on Logging Practices In Industry. *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. May 31 – June 07, 2014, Hyderabad, India.
3. Yuan, D., Park, S. & Zhou, Y. Characterizing Logging Practices in Open-Source Software. *Proceedings of the 34th International Conference on Software Engineering* (pp. 102-112). June 2-9, 2012, Zurich, Switzerland.
4. Ding, R., Zhou, H., Lou, J.-G., Zhang, H., Liu, Q., Fu, Q., Zhang, D. & Xie, T. Log2: A Cost-aware Logging Mechanism for Performance Diagnosis. *Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC'15)* (pp. 139-150), July 8–10, 2015, Santa Clara, CA, USA.
5. Shang, W., Nagappan, M., Hassan, A. E. & Jiang, Z. M. Understanding Log Lines Using Development Knowledge. *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'*2014) (pp. 21-30). 29 September – 03 October, 2014, Victoria, BC, Canada.
6. Xu, W., Huang, L., Fox, A., Patterson, D. & Jordan, M. Detecting Large-Scale System Problems by Mining Console Logs. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (pp. 37-46), June 21-24, 2010, Haifa, Israel.

ДОВІДКА № 6/22-14

Затверджує те, що стаття «ОСОБЛИВОСТІ ЛОГУВАННЯ ДАНИХ ТА ВПЛИВ ТИПУ ДОДАТКА НА ВИБІР МЕТОДОЛОГІЇ ЛОГУВАННЯ» авторів КАПІТАНЕЦЬ С. О., РАДЕЛЬЧУК Г. І. (Хмельницький національний університет) – прийнята до опублікування у науковому журналі "Вісник Хмельницького національного університету" № 6 за 2022 р., серія "Технічні науки", який включено до наукометричних баз (Index Copernicus, Google Scholar) та затверджений як фахове видання Постановою президії ВАК України від 28.125.2019 № 1643

Начальник відділу
інтелектуальної власності та трансферу технологій,
відповідальний секретар Вісника ХНУ Ю. В. Кравчик

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Кафедра інженерії програмного забезпечення

Удосконалення методів і механізмів керування логуванням у програмних системах

Виконав:
Студент II курсу, група ІПЗм-21-1,
Капітанець Степан Олександрович

Керівник:
доцент, кандидат технічних наук,
Радельчук Галина Іванівна

Об’єкт, предмет і мета дослідження

- **Об’єкт дослідження.** Журналювання даних.
- **Предмет дослідження.** Методи та механізми збору даних журналів.
- **Мета дослідження.** Метою цього дослідження є створення та впровадження масштабованої архітектури для проведення аналізу журналів даних в реальному часі у високонавантажених програмних системах.

Актуальність теми

Актуальність теми полягає в тому, що ведення журналів широко застосовується на практиці завдяки своїй простоті та ефективності. Програмні системи ведуть журнали для моніторингу подій та свого поточного стану. Такі журнали в процесі пошуку та усунення несправностей являють собою дуже важливе та цінне джерело інформації для розробників, які можуть вивчати записані логи щоби зрозуміти стан системи, виявляти аномалії та знаходити першопричини їх виникнення.

Проте велика кількість високонавантажених систем в процесі своєї роботи можуть генерувати величезну кількість логів зі швидкістю в десятки гігабайт на годину. Такий обсяг даних надзвичайно складно обробляти вручну, навіть за допомогою інструментів пошуку та фільтрації, що створює необхідність в розробці інструменту для автоматичного аналізу журнальованих даних.

Завдання дослідження

Відповідно до мети необхідно вирішити наступні завдання дослідження:

- провести теоретичний аналіз сфери журналювання даних;
- виконати аналіз та порівняння методів що застосовуються для логування;
- дослідити методи що застосовуються для виявлення аномалій;
- виявити недоліки існуючих рішень;
- удосконалити метод для системи логування, який дозволить обробляти великі обсяги даних одночасно у високонавантажених системах;
- розробити архітектуру на основі удосконаленого методу з можливістю горизонтального масштабування;
- дослідити ефективність запропонованого рішення;
- проаналізувати отримані результати та сформувати рекомендації щодо доцільності впровадження запропонованої архітектури на реальні проекти.

Аналіз галузі та виділення проблем

Основні проблеми притаманні системам для журналювання даних:

- високі затримки в обробці та збереженні повідомлень;
- високе навантаження на систему в процесі роботи;
- складність підключення та налаштування;
- непристосованість до масштабування та роботи з розподіленими системами;
- неможливість отримання даних від кількох джерел одночасно.

Архітектура для системи логування даних

Метод, представлений в цій роботі, полягає в масштабуванні рішення, яке структурує і моделює дані журналів даних так, щоб подальший аналіз був доступний через розподілене розгортання.

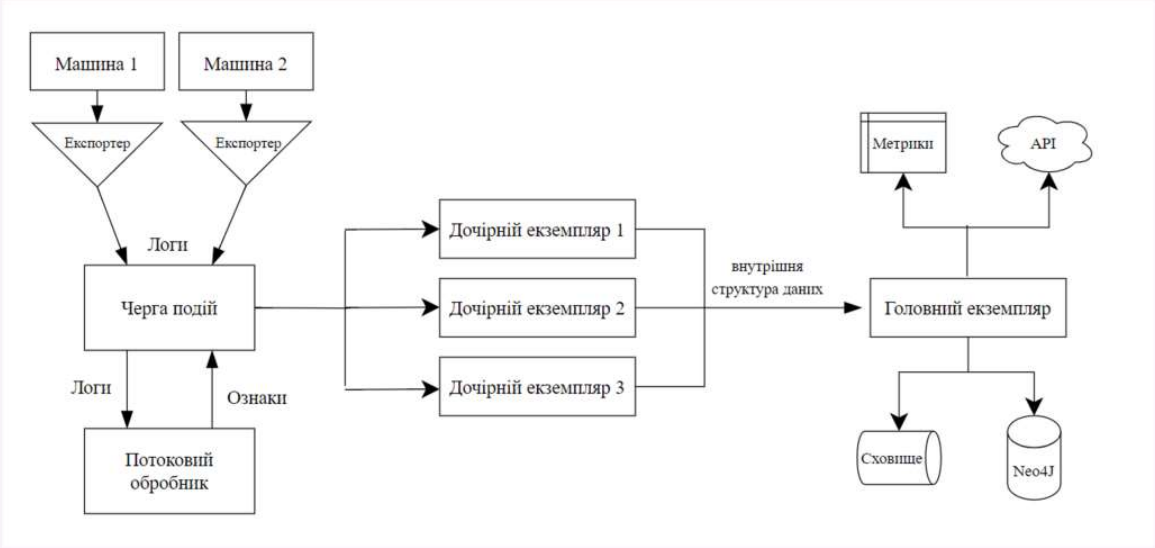
Архітектура розроблена для створення середовища в якому аналіз даних журналу може масштабуватися по горизонталі для роботи з великою кількістю операторів журналу.

Пропонована архітектура включає в себе декілька частин, кожна з яких може горизонтально масштабуватися.

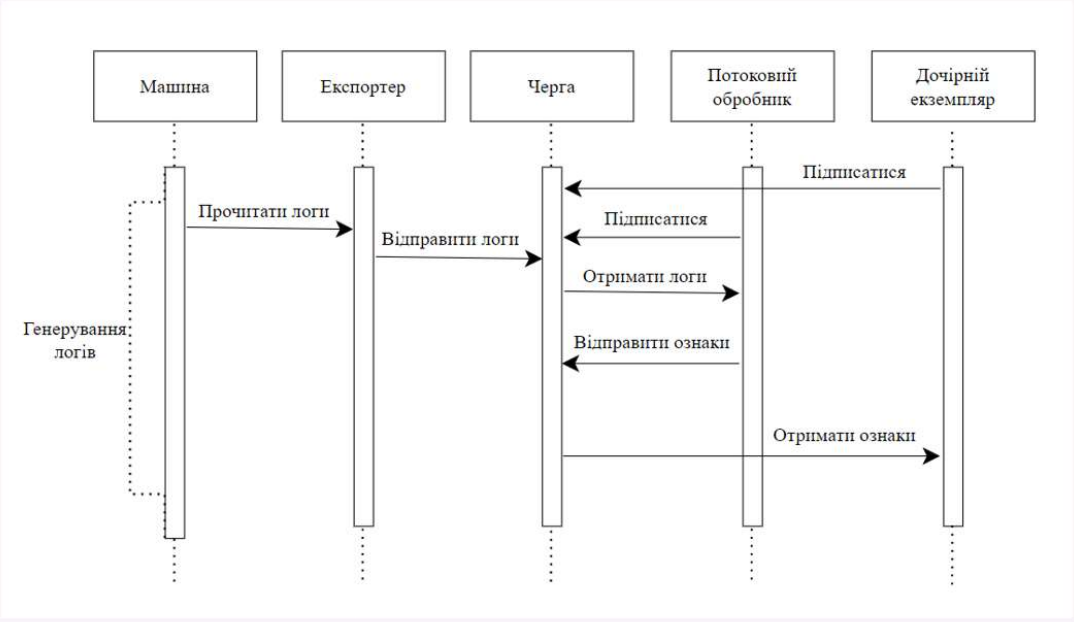
Представлене рішення працює через масштабовані блоки, які будують багаточастковий граф і можуть виконувати над ним обчислення, та один основний екземпляр, який періодично дістає часткові графи та контролює повідомлення про аномалії.

Наявні в галузі інструменти з підтримкою масштабування, такі як Prometheus, використовуються для аналізу статистики.

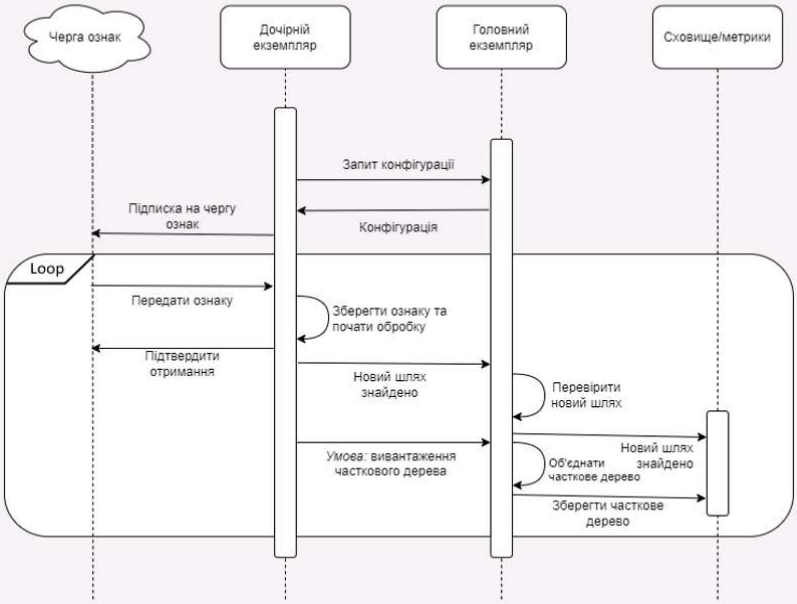
Архітектура для системи логування даних



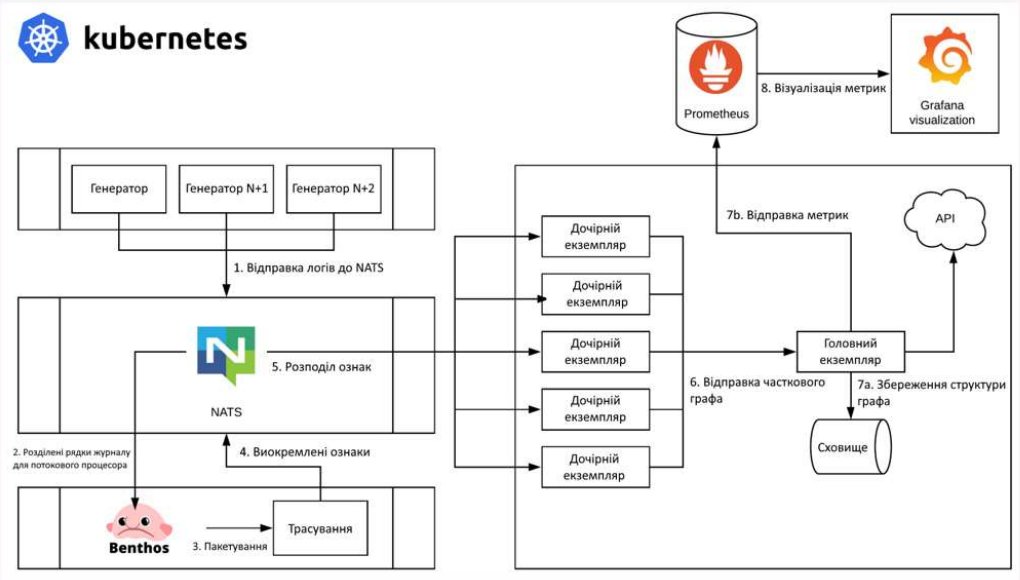
Послідовність проходження даних через систему



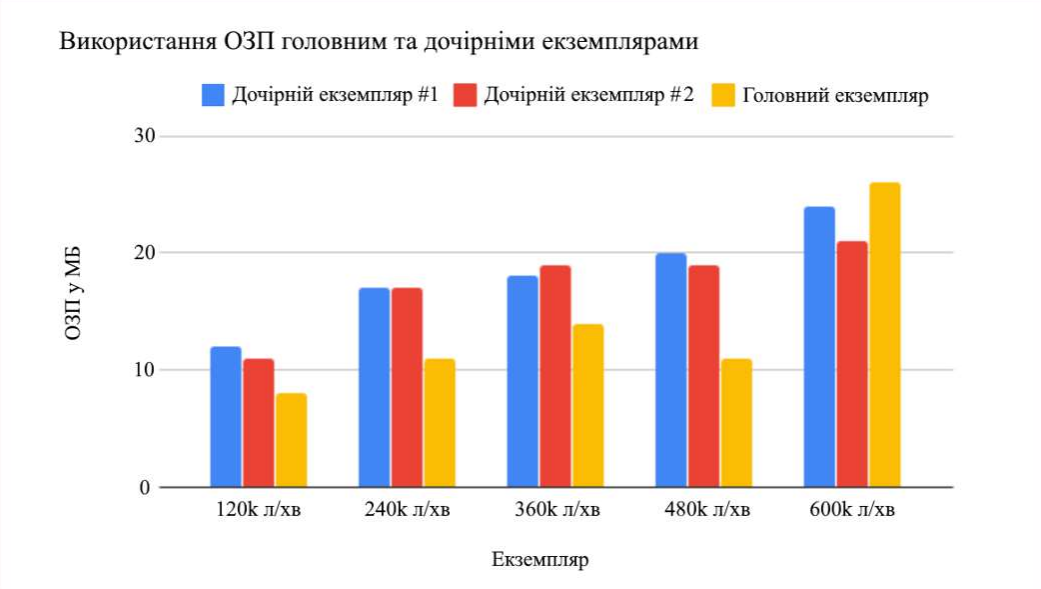
Послідовність проходження даних через систему



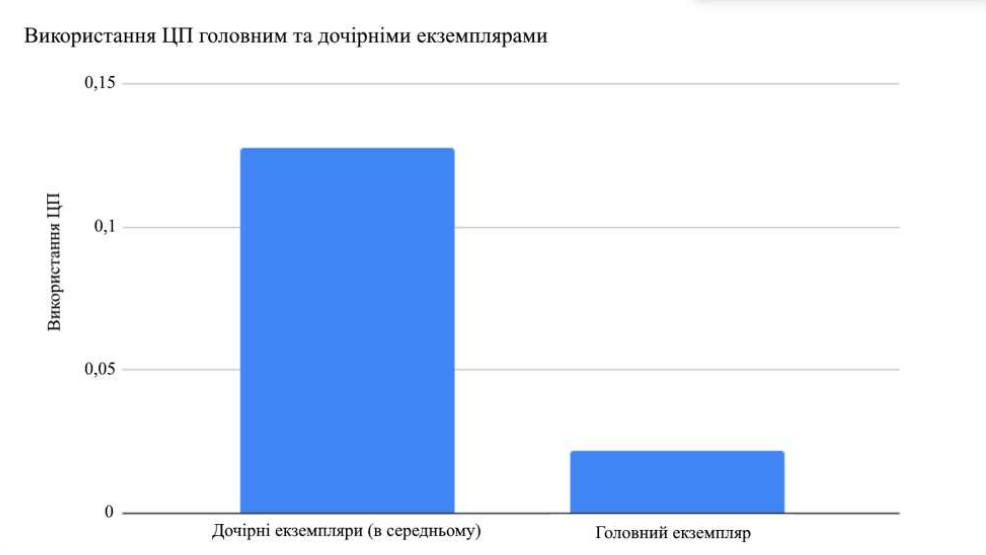
Архітектура повного програмного середовища



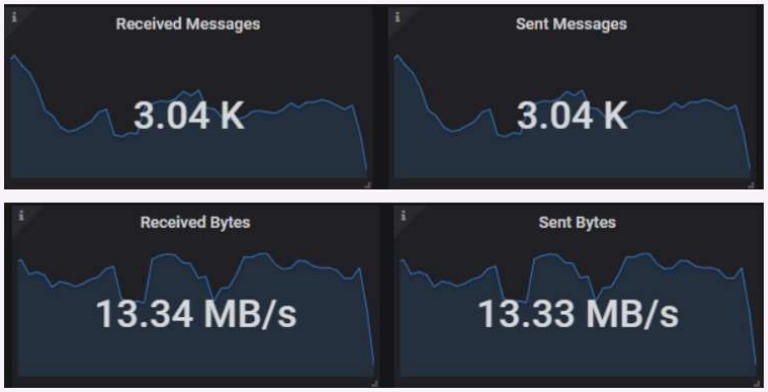
Апробація результатів



Апробація результатів



Апробація результатів



Пропускна здатність мережі при 1,2 млн логів в хвилину

Наукова новизна

Наукова новизна отриманих результатів:

- запропоновано та імплементовано архітектуру для логування даних, яка призначена спеціально для аналізу журналів у високонавантажених програмних системах у режимі реального часу;
- отримала подальший розвиток техніка розподілених обчислень у напрямку її застосування для логування даних, що дозволило отримувати результати у реальному часі або з мінімальними затримками.

Практична цінність

Практична цінність отриманих результатів полягає в успішній реалізації методу який пропонує масштабовану архітектуру для проведення аналізу журналів даних в реальному часі для високонавантажених систем.

Низькі затримки дозволяють швидко обробляти нові журнальні повідомлення. В свою чергу, це дозволяє своєчасно виявляти проблеми та їх причини, що дає змогу швидко відновити працездатність застосунку в разі виникнення помилок.

Висновки та рекомендації

В результаті виконання дипломної роботи було здійснено дослідження існуючих рішень та проблем в сфері логування та аналізу журналів повідомлень.

Запропоновано метод та рішення, яке дозволяє впроваджувати масштабовану архітектуру для проведення аналізу журналів даних в реальному часі з використанням удосконаленого методу виявлення аномалій у високонавантажених програмних системах.

В ході експерименту було виявлено що розроблене рішення здатне витримувати високі навантаження використовуючи при цьому лише обмежені ресурси комп’ютера.

Оскільки розроблене рішення не потребує особливих налаштувань, воно може бути досить легко інтегроване у проект, що дозволить збільшити стабільність його роботи шляхом своєчасного виявлення аномалій.

Динамічне додавання розрахунків і нових правил до запропонованого рішення може зробити цей інструмент більш актуальним в галузі.

Дякую за увагу!

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Капітанця С. О.
факультет ІТ, 2 курс, група ІПЗм-21-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

02.12.22
дата

підпис

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1013184908

Дата перевірки:
05.12.2022 11:47:21 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
05.12.2022 11:49:44 EET

ID користувача:
100005589

Назва документа: **Магістерська_Капітанця_антиплагіат**

Кількість сторінок: **78** Кількість слів: **15293** Кількість символів: **118468** Розмір файлу: **2.24 MB** ID файлу: **1012948700**

1.02% Схожість

Найбільша схожість: **0.57%** з джерелом з Бібліотеки (ID файлу: **1012913624**)

0.37% Джерела з Інтернету	15	Сторінка 80
0.77% Джерела з Бібліотеки	53	Сторінка 80

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0% Вилучень

Немає вилучених джерел

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 12.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 8%**

ID: 108938 Назва: КРМ на тему: «Удосконалення методів і механізмів керування логуванням у програмних системах» Додано в БД: 2022-12-05 Автора: Капітанець С.О. Керівники: Радельчук Г.І. Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	105809	432	13474 (13%)	96 (22%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
107867	Назва: Звіт з переддипломної практики Додано в БД: 2022-10-08 Автора: С. О. Капітанець Керівники: Радельчук Г.І. Консультанти: Опоненти:	13041 (12.0%)	109 (25.0%)

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

освітнього ступеня «магістр»

Магістр Капітанець Степан Олександрович

Тема Удосконалення методів і механізмів керування логуванням у програмних системах

Спеціальність 121 « Інженерія програмного забезпечення»

Обсяг дипломної роботи:

Кількість сторінок дипломної роботи 123.

1. Короткий зміст роботи та прийнятих рішень У процесі дипломного проектування досліджено галузь логування даних та сучасні методи і способи виявлення аномалій у програмних системах, визначено невирішені проблеми у галузі, на основі чого запропоновану масштабовану архітектуру для аналізу даних журналу в реальному часі, виконано її програмну реалізацію у вигляді бібліотеки, навантажувальне тестування реалізованого засобу.

2. Висновок про відповідність роботи дипломному завданню Дипломна робота освітнього ступеня «магістр» відповідає поставленому завданню як у теоретичній, так і в практичній її частині.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступі обґрунтовується актуальність теми роботи, формулюються цілі і завдання дослідження, описується наукова новизна та практична значимість отриманих результатів. У першому розділі дипломної роботи досліджено та проаналізовано галузь логування даних, останні публікації, наукові статті та джерела, існуючі методи та способи організації процесів логування та виявлення аномалій з метою виявлення невирішених проблем. У другому розділі роботи проаналізовано моделі, концепції та алгоритми вирішення проблем виявлення аномалій. Нерозв'язані задачі запропоновано вирішити шляхом впровадження нової архітектури, в якій аналіз даних журналу може масштабуватися по горизонталі для роботи з великою кількістю операторів журналу. У третьому розділі описана технологія реалізації запропонованої архітектури. У четвертому розділі описано програмну реалізацію та результати навантажувального тестування розробленої системи на основі запропонованої архітектури. Також проведено емпіричне дослідження, спрямоване на доведення працездатності розробленого програмного засобу та його функціональної придатності. Обґрунтована ефективність створених методів та засобів та розроблено рекомендації з їх застосування.

4. Позитивні сторони роботи У дипломній роботі запропоновано метод та рішення, яке дозволяє впроваджувати масштабовану архітектуру для проведення аналізу журналів даних в реальному часі з використанням удосконаленого методу виявлення аномалій у високонавантажених програмних системах. Низькі затримки дозволяють швидко обробляти нові журнальні повідомлення. В свою чергу, це дозволяє своєчасно виявляти проблеми та їх причини, що дає змогу швидко відновити працездатність застосунку в разі виникнення помилок. В ході експерименту було виявлено що розроблене рішення здатне витримувати високі навантаження використовуючи при цьому лише обмежені ресурси комп'ютера.

5. Негативні сторони роботи Запропоноване рішення забезпечує масштабований спосіб аналізу журналу в реальному часі на графі. Однак, на даний момент, виконуються лише прості обчислення, і вихідні дані все ще можуть бути важко зрозумілими самі по собі. Рішення включає в себе функцію звітування, яка допомагає виявляти аномалії в роботі програмних систем, але тільки в загальному вигляді. Можливість динамічного додавання розрахунків і нових правил до запропонованого рішення може зробити цей інструмент набагато більш актуальним в галузі.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми дипломної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про роботу в цілому В цілому дипломна робота заслуговує позитивної оцінки. Матеріал дипломної роботи структурований та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для вирішення поставленої задачі.

8.Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінки «задовільно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Мартинюк Валерій Володимирович,
завідувач кафедри «Автоматизації та комп'ютерно-інтегрованих технологій»

« 2 » грудня 2022 р.

(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Удосконалення методів і механізмів керування логуванням у програмних системах»
Автор: Капітанець Степан Олександрович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Радельчук Галина Іванівна, кандидат технічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посилання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає **1.02%** і адресується до 15 джерел з Інтернету та 53 джерел з бібліотеки, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь дипломної роботи.

Керівник _____ Г. І. Радельчук

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк