Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет «Комп'ютерні технології і системи»

Кафедра «Комп'ютерні інформаційні технології»

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему: «Проектування та розгортання маштабованоі системи для збору й аналізу телеметрії в Kubernetes за допомогою Elasticsearch і Grafana»

за освітньою програмою: «12 Інженерія програмного забезпечення»

зі спеціальності: «121 Інженерія програмного забезпечення»

Виконав: студент групи «ПЗ2016»

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

(підпис студента) (Ім’я ПРІЗВИЩЕ)

|  |  |  |  |
| --- | --- | --- | --- |
| Керівник: |  |  |  |

(підпис) (посада, Ім’я ПРІЗВИЩЕ)

|  |  |
| --- | --- |
| Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань. | |
| Студент |  |

(підпис)

Дніпро – 2025 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty «Computer technologies and systems»

Department «Computer information technology»

Explanatory Note

to Bachelor’s Thesis

on the topic: «Design and deployment of a scalable system for collecting and analyzing telemetry in Kubernetes using Elasticsearch and Grafana»

according to educational curriculum «12 Software engineering»

in the Speciality: «121 Software engineering»

|  |  |  |
| --- | --- | --- |
| Done by the student of the group PZ2016: |  |  |
| Scientific Supervisor: |  |  |
| Normative controller: |  |  |

Dnipro – 2025

Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет: «Комп'ютерні технології і системи»

Кафедра: «Комп'ютерні інформаційні технології»

Рівень вищої освіти: бакалавр

Освітня програма: «12 Інженерія програмного забезпечення»

Спеціальність: «121 Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри КІТ

/Вадим ГОРЯЧКІН/

(підпис)

Дата

**ЗАВДАННЯ**

На кваліфікаційну роботу

бакалавра студенту

1. Тема роботи: «Проектування та розгортання маштабованоі системи для збору й аналізу телеметрії в Kubernetes за допомогою Elasticsearch і Grafana»

Керівник роботи: Шаповал Ірина Вікторівна, старший викладач

затверджені наказом № 328ст від 03.03.2025.

|  |  |  |
| --- | --- | --- |
| 2. Строк подання студентом роботи: | | 19.06.2025 |
| 3. Вихідні дані до роботи: |  |  |
| 4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати): | | |
| вступ, збір вимог до моніторингу, зовнішнє і внутрішнє | | |
| проектування, тестування та налагодження, висновки, література. | | |
| 5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень): презентація, відео-демонстрація роботи програми. | | |

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
| 1 | Постановка задачі | 03.04.2025 – 05.04.2025 |  |
| 2 | Огляд літератури та аналіз аналогів | 06.04.2025 – 09.04.2025 |  |
| 3 | Розробка структур вхідних і вихідних даних | 10.04.2025 – 24.04.2025 |  |
| 4 | Визначення вимог до програми. Вибір та обґрунтування мови програмування | 25.04.2025 – 02.05.2025 |  |
| 5 | Узгодження та затвердження ТЗ | 03.05.2025 – 10.05.2025 | 30% |
| 6 | Розробка та програмування логіки програми | 11.05.2023 – 18.05.2023 |  |
| 7 | Розробка і реалізація графічного відоюраження метрик (Kibana, Grafana) | 19.05.2025 – 26.05.2025 | 60% |
| 8 | Відлагодження програми | 27.05.2025 – 02.06.2025 |  |
| 9 | Розробка, узгодження та затвердження програмної документації | 03.06.2025 – 18.06.2025 | 100% |
| 10 | Подання кваліфікаційної роботи до кафедри | 19.06.2025 – 26.06.2025 |  |
| 11 | Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії | 27.06.2025 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент |  |  |  |

(підпис (Ім’я ПРІЗВИЩЕ)

|  |  |  |  |
| --- | --- | --- | --- |
| Керівник роботи |  |  |  |

**(підпис (Ім’я ПРІЗВИЩЕ)**

## РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра 82 с., 10 рис., 9 табл., 12 джерел.

**Метою роботи** було проектування та розгортання масштабованої системи для збору й аналізу телеметрії в Kubernetes з використанням Elasticsearch і Grafana. Основними завданнями було створення надійної та гнучкої архітектури, що дозволяє ефективно збирати, зберігати та візуалізувати дані про роботу контейнеризованих додатків та інфраструктури, а також налаштування алертингу для проактивного виявлення проблем. За результатами роботи було створено інтегроване рішення, що відповідає всім вимогам сучасних підходів до моніторингу розподілених систем.

Робота складається з 7 розділів:

* вступ — у цьому розділі описується актуальність проблеми моніторингу контейнеризованих додатків, мета та завдання дослідження. Складається з 3 сторінок;
* збір вимог до програмного забезпечення — у цьому розділі аналізуються існуючі системи моніторингу, розглядаються теоретичні основи контейнеризації та оркестрації, принципи роботи Kubernetes, системи збору та аналізу логів і метрик. Складається з 20 сторінок;
* зовнішнє і внутрішнє проектування — у цьому розділі описується функціональне призначення системи, визначаються вхідні і вихідні дані, проводиться проектування архітектури системи моніторингу, тестового додатку, системи алертів та дашбордів для візуалізації даних. Складається з 23 сторінок;
* реалізація та тестування — цей розділ містить інформацію про розгортання Kubernetes кластера, налаштування компонентів системи моніторингу (ELK Stack, Prometheus, Grafana), розгортання тестового додатку та результати тестування системи. Складається з 18 сторінок;
* висновки — підсумовуються результати роботи та визначаються перспективи подальших досліджень. Складається з 2 сторінок;
* список літератури — включає бібліографічний список використаної літератури. Складається з 2 сторінок;
* додатки — містять конфігураційні файли та код, що використовувався для розгортання та налаштування системи, а також приклади дашбордів. Складається з 14 сторінок.

Кількість таблиць: 9 штук. Кількість рисунків: 10 штук.

Ключові слова: KUBERNETES, МОНІТОРИНГ, ELASTICSEARCH, GRAFANA, PROMETHEUS, КОНТЕЙНЕРИЗАЦІЯ, ОРКЕСТРАЦІЯ, ЛОГИ, МЕТРИКИ, ТЕЛЕМЕТРІЯ, МАСШТАБОВАНІСТЬ, АНАЛІЗ ДАНИХ, ВІЗУАЛІЗАЦІЯ, АЛЕРТИНГ.

2 ЗОВНІШНЄ І ВНУТРІШНЄ ПРОЕКТУВАННЯ

ЗМІСТ

[ВСТУП 11](#_Toc138003767)

[1 ЗБІР ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 13](#_Toc138003768)

[1.1 13](#_Toc138003769)

[1.1.1 Огляд платформи «DataDog»     14](#_Toc138003770)

[1.1.2 Огляд платформи «Dynatrace»     15](#_Toc138003771)

[1.1.3 Огляд стеку ELK](#_Toc138003772).

[1.1.4 Огляд Prometheus Stack     18](#_Toc138003773)

[1.2.1 Принципи роботи Kubernetes. 19](#_Toc138003774)

[Висновки до пункту 1 Error: Reference source not found](#_Toc138003775)

2 ЗОВНІШНЄ І ВНУТРІШНЄ ПРОЕКТУВАННЯ

2.1 Зовнішнє проектування

2.1.1 Функціональне призначення

2.1.2 Експлуатаційне призначення

2.1.3 Функціональні вимоги

2.1.4 Вхідні та вихідні дані

2.1.4.1 Вхідні дані

2.1.4.2 Вихідні дані

2.1.5 Опис зовнішнього інформаційного середовища

2.2 Внутрішнє проектування

2.2.1 Аналіз зовнішніх специфікацій систем

2.2.1.1 Моделювання словника системи

2.2.2 Проектування архітектури системи

2.2.2.1 Загальний огляд архітектури

2.2.2.2 Компоненти системи

2.2.2.3 Топологія та взаємодія компонентів

2.2.2.4 Масштабованість та відмовостійкість

2.2.3 Проектування підсистеми збору метрик

2.2.4 Проектування підсистеми збору та аналізу логів

2.2.4.1 Архітектура підсистеми логування

2.2.4.2 Конфігурація збирача логів Fluent Bit

2.2.4.3 Налаштування Elasticsearch для зберігання логів

2.2.4.4 Структура індексів та шаблони індексів Elasticsearch

2.2.4.5 Управління життєвим циклом даних у Elasticsearch

2.2.4.6 Конфігурація Kibana для візуалізації та аналізу логів

2.2.4.7 Парсери для структурованих логів

2.2.4.8 Механізми безпеки та контролю доступу

2.2.4.9 Інтеграція з Grafana для уніфікованого доступу

2.2.5 Проектування підсистеми візуалізації та дашбордів

2.2.5.1 Архітектура підсистеми візуалізації

2.2.5.2 Типи дашбордів та їх призначення

2.2.5.3 Методи візуалізації даних

2.2.5.4 Шаблонізація та змінні в дашбордах

2.2.5.5 Автоматизація створення дашбордів

2.2.6 Проектування підсистеми оповіщень

2.2.6.1 Архітектура підсистеми оповіщень

2.2.6.2 Категорії оповіщень

2.2.6.3 Правила оповіщень

2.2.6.4 Маршрутизація та ескалація оповіщень

2.2.6.5 Інтеграція з системами управління інцидентами

2.2.7 Проектування механізмів масштабування та високої доступності

2.2.7.1 Стратегії масштабування компонентів

2.2.7.2 Забезпечення високої доступності

Висновки до пункту 2

[3 ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ 56](#_Toc138003788)

[Висновки до пункту 3 Error: Reference source not found](#_Toc138003789)

[ВИСНОВОК Error: Reference source not found](#_Toc138003790)

[ЛІТЕРАТУРА 66](#_Toc138003791)

[ДОДАТКИ Error: Reference source not found](#_Toc138003792)

[ДОДАТОК А – ТЕКСТ ПРОГРАМИ 69](#_Toc138003793)

ПЕРЕЛІК УМОВНИХ ПОЗНАК, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ (програмне забезпечення) – програмний продукт, який застосовується на комп’ютерних системах таких як персональні комп’ютери, ноутбуки, мобільні пристрої, планшети, телефони а також деколи і на різній побутовій техніці на кшталт пральних машин, мікрохвильовок, холодильників та інше.

Кастомізація – підлаштовування, налаштування чогось під себе. Те ж саме що й персоналізація.

Нота – ступінь або звук, що має характеристики: октава, знак альтерації, протяжність та інші.

Тональність – це висотне розташування ладу. Усього існує 30 тональностей.

Лад – це сукупність звуків, які на основі спорідненості між ними об’єднані в систему, що має тоніку.

Тоніка – базова нота тональності або ладу, тобто та нота, від якої йде побудова тональності чи ладу.

Півтон – одиниця виміру музичних інтервалів, є найменшою відстанню між звуками за висотою. Зазвичай, візуально на фортепіано, у ролі півтону виступає одна клавіша.

Тон – одиниця виміру музичних інтервалів, містить в собі два півтони. Зазвичай, візуально на фортепіано, у ролі тону виступає дві клавіші підряд.

Ступінь – це звук музичної системи (звукоряду). Усього основних ступенів сім – «до», «ре», «мі», «фа», «соль», «ля», «сі».

Знак альтерації – знак, що відображає підвищення або зниження ступеню на півтона або тон. Таких знаків декілька: дієз (♯), дубль-дієз, бемоль (♭), дубль-бемоль, а також бекар (♮) – знак відміни дій інших знаків.

Нотний стан – це система з п’яти паралельних горизонтальних ліній, на якій розміщують ноти.

Октава – інтервал від «до» до «сі» включаючи обидві ноти, відображає звукоряд певної висоти. Таких октав може бути на різних інструментах різну кількість, і усі октави мають власні назви та власну висоту звучання.

Мажор – музичний лад, аккорди якого будуються на великій терції. Зазвичай цей лад має позитивне, радісне звучання.

Мінор – музичний лад, аккорди якого будуються на малій терції. Зазвичай цей лад має похмуре, сумне звучання.

# ВСТУП

Музика, як і у минулому, сьогодні, так і надалі залишатиметься невід’ємною частиною нашого життя. Технології просунулася вперед і тим самим змінилася сама музика – з’явились безліч нових інструментів та різноманітних жанрів, покращилась якість звучання, спростився процес створення нових композицій, пісень, аранжувань. Але будь-який інструмент у руках людини без гарної практики та базових знань ­– просто не розкритий інструмент, на якому неможливо відтворити щось дійсно приголомшливе. У всіх сферах є умовна "база знань", де потрібно досягти максимального рівня навичок, щоб отримати фундамент, завдяки якому людина стане майстром справи. У музиці важлива кожна нота, зіграна на інструменті, адже кожна нота задає тон та настрій композиції. Але все ж таки, чому музика настільки важлива для нашого світу? А важлива вона тим, що завдяки їй часто люди піднімають собі настрій, задають атмосферу у своєму повсякденному житті, мають різні спогади, пов’язані з окремими піснями. В тяжкі для людини часи, музика надає натхнення рухатись вперед та направляє на різноманітні вчинки, надає мотивацію, підтримує дух людини аби не дати їй впасти духом та поринути у депресію. Словом, «Музика дозволяє рухатись вперед і бути закоханим у неї». А ще є такі люди, як меломани, що просто не уявляють своє життя без музики, адже меломани обожнюють абсолютно різні жанри.

Завдяки технологіям нашого часу можна знайти різноманітні програми, що допомагають музикантам у вивченні та виявленні особистих проблем у музиці. Такі програми допомагають музикантові у його становленні, як висококваліфікованого фахівця. Тому, аби допомогти усім музикантам, а саме: як початківцям, так і більш досвідченим, – було вирішено створити ПЗ «Elegia», яка сприятиме у відточенні музичних навичок та знань музичної теорії, методом використовування запропонованих інструментів ПЗ для досягнення почуття, що передає та сама фраза: «Музика допомагає рухатися вперед…».

Завданням цієї дипломної роботи є вдосконалення навичок знань музиканта. Робота передбачає за собою написану музичну програму, що має тренажерні тести та інструментарій для музикантів із простим інтерфейсом для легкого використання. Для користувача ПЗ, здебільшого, є два режими у кожному розділі – режим тренування та режим інструменту, завдяки яким він зможе відточити свої знання в музичній теорії на практиці. Показувати результат тестування користувачеві – важлива складова навчання. Окрім зміцнення знань, «Elegia» матиме ряд інструментів із роботою для піаніно. Користувачем може бути будь-яка зацікавлена у музиці ​​людина – як музикант-початківець, так і професійний музикант зі стажем.

# 1 ЗБІР ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 1.1 Перегляд аналогових конкурентів

Серед сучасних рішень для моніторингу контейнеризованих середовищ та інфраструктури, окрім Prometheus, варто виділити дві потужні комерційні платформи — DataDog та Dynatrace. DataDog представляє собою комплексну SaaS-платформу для моніторингу інфраструктури та додатків, що відрізняється швидким розгортанням завдяки готовим інтеграціям з понад 450 технологіями. Платформа використовує агентну модель збору даних, де легкий агент встановлюється на кожний сервер або контейнер і збирає метрики, логи та трейси, забезпечуючи візуальну кореляцію між ними. Особливу увагу варто звернути на розширені можливості DataDog для моніторингу Kubernetes, включаючи автоматичне виявлення подів, сервісів, відстеження життєвого циклу контейнерів та візуалізацію взаємозалежностей між компонентами системи. В свою чергу, Dynatrace пропонує унікальний підхід "один агент для всього" через технологію OneAgent, яка автоматично виявляє всі компоненти середовища та збирає необхідну телеметрію без ручного налаштування. Ключовою перевагою Dynatrace є запатентована система штучного інтелекту Davis, що аналізує мільярди залежностей для виявлення причинно-наслідкових зв'язків між подіями та автоматичного визначення кореневих причин проблем. Обидві платформи пропонують першокласну інтеграцію з Kubernetes, включаючи спеціалізовані оператори для розгортання агентів, глибокий моніторинг кластера та компонентів, а також потужні аналітичні інструменти. Проте, на відміну від відкритого Prometheus, ці рішення мають значні недоліки — високу вартість при масштабуванні, зберігання даних у хмарній інфраструктурі стороннього провайдера та закриту екосистему з обмеженими можливостями кастомізації.

## Огляд платформи DataDog

DataDog — це комплексна SaaS-платформа для моніторингу інфраструктури, додатків та цифрових користувацьких взаємодій. Заснована у 2010 році, сьогодні вона є одним із провідних рішень для спостереження (observability) та моніторингу в корпоративному сегменті.

### Архітектура та принципи роботи

DataDog використовує агентну модель збору даних. Легкий агент встановлюється на кожний сервер, контейнер або віртуальну машину. Він збирає метрики, логи та трейси, відправляючи їх у централізоване сховище DataDog для подальшої обробки та аналізу.

Основні компоненти архітектури DataDog:

1. **DataDog Agent** — легкий, відкритий програмний агент, написаний на Go, який збирає метрики, логи та трейси з хостів і сервісів.
2. **Інтеграції** — готові модулі для збору даних від популярних технологій та сервісів.
3. **Central DataDog Platform** — хмарна платформа для зберігання, обробки та візуалізації даних.
4. **API і SDK** — інструменти для інтеграції та кастомізації.



Рисунок 1.1 – Приклад дашборда DataDog

Переваги:

* **Швидкість впровадження**: готові інтеграції з понад 450 технологіями дозволяють швидко почати збір даних;
* **Уніфікований моніторинг**: єдина платформа для метрик, логів та трейсів;
* **Масштабованість**: здатність обробляти дані від десятків тисяч серверів;

Недоліки:

* **Висока вартість при масштабуванні**: ціна залежить від кількості хостів та функцій, що використовуються.
* **Зберігання даних у хмарі**: чутливі дані зберігаються на серверах DataDog, що може не відповідати вимогам безпеки деяких організацій.
* **Складність налаштування просунутих функцій**: деякі можливості вимагають глибокого розуміння платформи.

## 1.1.2 Огляд платформи Dynatrace

Dynatrace — це інтелектуальна платформа для моніторингу та обсерваційності, що використовує штучний інтелект для автоматичного виявлення та діагностики проблем у складних ІТ-середовищах. Заснована у 2005 році в Австрії, компанія Dynatrace зробила значний перехід від традиційного APM (Application Performance Management) до повноцінної платформи обсерваційності з власною AI-системою Davis. (сам приклад дашборду наведено на рис. 1.2).

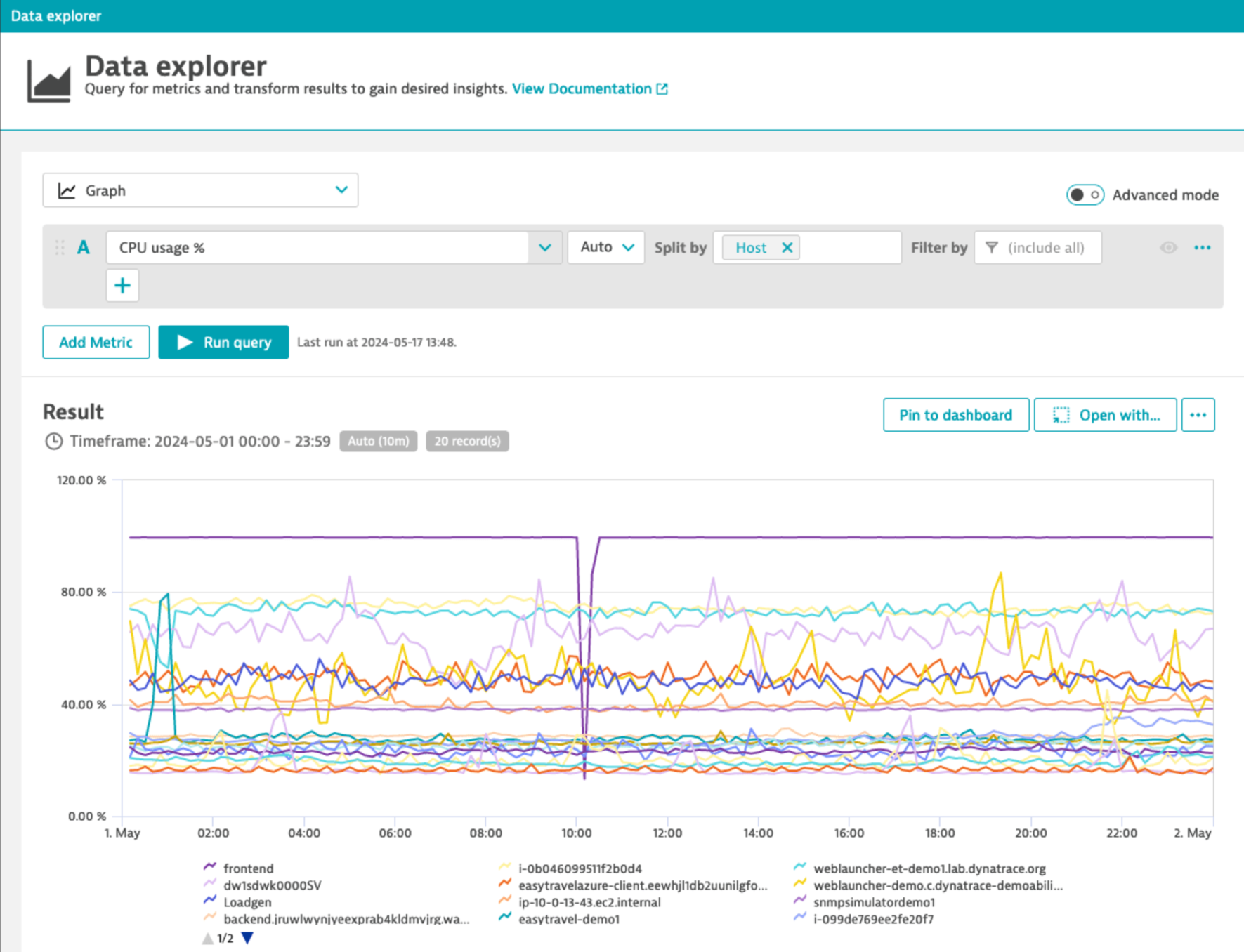


Рисунок 1.2 – Приклад дашборду «Dynatrace»

Переваги:

* **Автоматизація та AI** — повністю автоматичне виявлення та діагностика проблем без необхідності ручного налаштування;
* **Повна видимість** — єдина платформа для моніторингу від інфраструктури до користувацького досвіду;
* **Causation-Based AI** — визначення кореневих причин проблем, а не просто кореляцій між подіями;
* **Самоадаптація** — автоматична адаптація до змін в середовищі.

Недоліки:

* **Висока вартість ліцензії** — одне з найдорожчих рішень для моніторингу на ринку;
* **Складність впровадження** — потребує значного часу для повноцінного впровадження та налаштування;
* **Вимоги до ресурсів** — може потребувати значних ресурсів для роботи в великих середовищах;
* **Закрита екосистема** — менша гнучкість порівняно з відкритими рішеннями;
* **Складна крива навчання** — потребує часу для повного освоєння всіх можливостей;

## 1.1.3 Огляд стеку ELK.

ELK Stack представляє собою інтегроване рішення з відкритим вихідним кодом, спрямоване на комплексну роботу з логами: від їх збору та обробки до зберігання й візуалізації. Назва "ELK" походить від перших літер ключових компонентів: Elasticsearch, Logstash та Kibana. З розвиткомтехнології до стеку додалися легкі агенти Beats, що розширили його функціональність.

Рисунок 1.3 ілюструє архітектуру ELK Stack (Elasticsearch, Logstash, Kibana), доповнену компонентами Beats. Діаграма відображає повний потік даних у системі логування та моніторингу.

Зліва зображено сімейство Beats - легкі агенти для збору різних типів даних:

* Filebeat - для збору логів з файлів
* Winlogbeat - для збору логів подій Windows
* Metricbeat - для збору системних та прикладних метрик
* Packetbeat - для аналізу мережевого трафіку

Ці компоненти виконують функцію крайового збору даних (Edge Data Collection).

У центрі розташовано Logstash - систему обробки та трансформації даних, що представлена кількома робочими вузлами (Nodes) з увімкненими персистентними чергами (Persistent Queues Enabled). Logstash отримує дані від агентів Beats, обробляє їх та передає до Elasticsearch.

Справа розміщено два компоненти:

1. Elasticsearch - розподілене сховище для зберігання і пошуку даних, що складається з:
   * Master Nodes (3) - вузли управління кластером
   * Data Nodes - Hot (X) - вузли для зберігання активних даних
   * Data Nodes - Warm (X) - вузли для зберігання менш активних даних
2. Kibana - платформа візуалізації з кількома екземплярами (Instances (X))

Стрілки показують напрямок руху даних: від Beats до Logstash, і далі від Logstash до Elasticsearch, а Kibana підключається до Elasticsearch для відображення та аналізу зібраних даних.

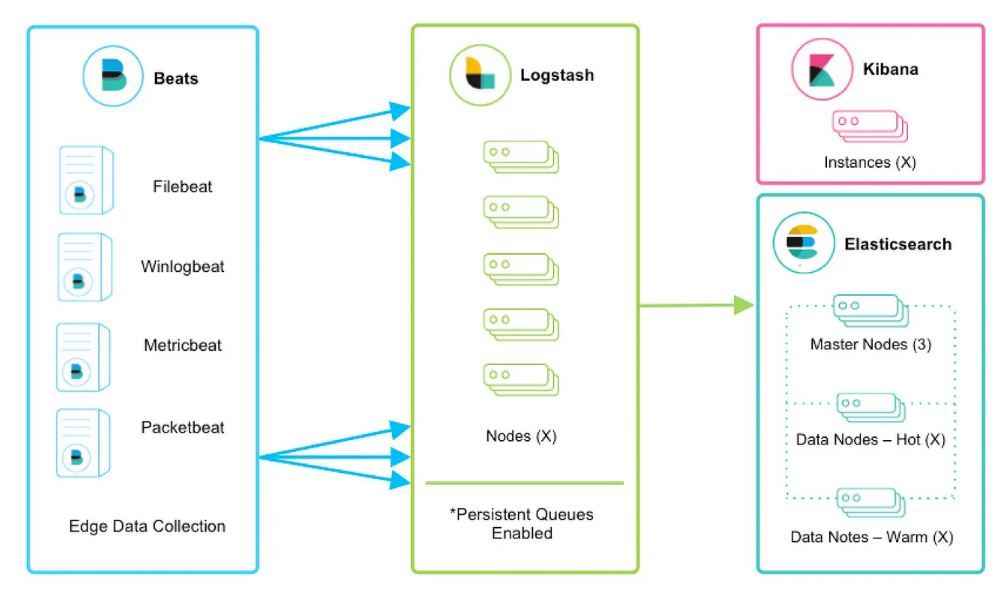


Рисунок 1.3 – ELK Stack

Переваги:

* **Прозорість реалізації**: можливість аудиту та розуміння внутрішньої роботи;
* **Спільнота розробників**: активна підтримка та розвиток компонентів;
* **Безкоштовне базове використання**: зниження витрат на впровадження;
* **Відсутність прив'язки до вендора**: незалежність від політики окремих компаній;
* **Можливість модифікації**: адаптація під специфічні потреби організації;

Недоліки:

* **Вимоги до ресурсів:** Одним з найсуттєвіших недоліків ELK Stack є високі вимоги до обчислювальних ресурсів;

### **Складність налаштування та підтримки:** ELK Stack вимагає значних зусиль для налаштування та підтримки;

### **Обмеження в роботі з метриками:** Незважаючи на можливість збору метрик через Metricbeat, ELK Stack має певні обмеження;

## 1.1.4 Огляд Prometheus Stack

Prometheus з Grafana формують потужний стек для моніторингу інфраструктури та додатків, який став стандартом де-факто для середовищ на базі Kubernetes. Цей стек пропонує спеціалізоване рішення для збору, зберігання, аналізу та візуалізації метрик.

## Архітектура та принципи роботи

Prometheus Stack побудований на кількох ключових компонентах:

* **Prometheus Server**: центральний компонент, що збирає та зберігає метрики як часові ряди
* **Exporters**: спеціалізовані агенти для надання метрик з різних систем
* **Alertmanager**: компонент для управління оповіщеннями
* **Pushgateway**: проміжний сервіс для короткоживучих завдань
* **Grafana**: платформа для візуалізації метрик у вигляді дашбордів

Архітектура стеку базується на моделі "pull", де Prometheus активно опитує (scrapes) цільові системи, а не чекає відправки даних, що забезпечує більшу надійність та контроль.

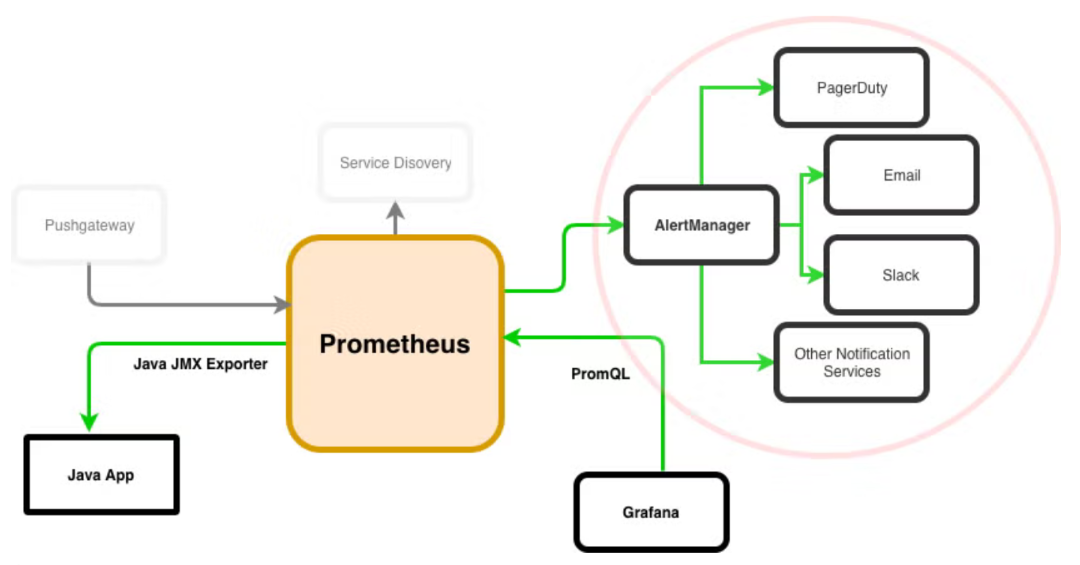


Рисунок 1.4 – Архітектура Prometheus

## 1.2.1 Принципи роботи Kubernetes.

Kubernetes (K8s) — це відкрита платформа для автоматизації розгортання, масштабування та управління контейнеризованими застосунками. Розроблена початково Google на основі внутрішньої системи Borg, Kubernetes стала стандартом де-факто для оркестрації контейнерів у промисловому масштабі.

#### Архітектура Kubernetes

Kubernetes має розподілену архітектуру типу "клієнт-сервер", що складається з наступних основних компонентів:

**Control Plane (Площина управління)**

Площина управління відповідає за глобальні рішення щодо кластера (наприклад, планування) та виявлення і реагування на події кластера. Компоненти площини управління включають:

1. **API Server (kube-apiserver)**: центральний компонент, що експонує API Kubernetes, обробляє запити та оновлює відповідні об'єкти в etcd. Всі взаємодії з кластером проходять через API Server.
2. **etcd**: розподілене сховище ключ-значення, яке зберігає всі дані кластера. Забезпечує надійне збереження конфігурації кластера та його стану.
3. **Scheduler (kube-scheduler)**: відповідає за відстеження новостворених подів, які не призначені до жодної ноди, та вибір ноди для їх запуску. Фактори для планування включають:
   * Вимоги до ресурсів
   * Обмеження апаратного/програмного забезпечення
   * Афініті та анти-афініті
   * Баланс навантаження між нодами
4. **Controller Manager (kube-controller-manager)**: запускає контролери, що обробляють рутинні завдання в кластері. Контролери включають:
   * Node Controller: моніторинг стану нод і реагування на їх відмови
   * Replication Controller: забезпечення правильної кількості подів
   * Endpoints Controller: зв'язування сервісів з подами
   * Service Account & Token Controllers: створення облікових записів і API токенів
5. **Cloud Controller Manager (cloud-controller-manager)**: інтегрує Kubernetes з API провайдерів хмарних послуг. Дозволяє основній кодовій базі Kubernetes залишатися незалежною від специфічних хмарних провайдерів.

**Worker Nodes (Робочі вузли)**

Worker ноди — це машини, які запускають контейнеризовані застосунки. Кожна нода містить:

1. **Kubelet**: агент, що запускається на кожній ноді. Забезпечує, що контейнери запускаються в поді. Kubelet:
   * Отримує специфікації подів від API сервера
   * Гарантує, що вказані контейнери працюють справно
   * Звітує про стан подів і ноди до API сервера
   * Запускає проби готовності та життєздатності
2. **Container Runtime**: програмне забезпечення, відповідальне за запуск контейнерів (containerd, CRI-O, Docker Engine та ін.)
3. **Kube-proxy**: мережевий проксі, що працює на кожній ноді. Підтримує мережеві правила та забезпечує зв'язок із подами. Реалізує частину концепції Kubernetes Service, керуючи перенаправленням з'єднань до відповідних подів.

#### Основні абстракції Kubernetes

Kubernetes оперує рядом абстракцій, що представляють стан системи:

**1. Pod (Под)**

Под — це найменша і найпростіша одиниця в Kubernetes, що представляє одиночний екземпляр застосунку. Под інкапсулює:

* Один або кілька контейнерів
* Спільний простір зберігання (томи)
* Специфічні налаштування мережі (IP-адреса, яку поділяють всі контейнери)
* Опції для запуску контейнерів

Поди, за своєю природою, є ефемерними (тимчасовими) і можуть бути замінені іншим подом при необхідності масштабування, оновлення або в разі збоїв.

**2. ReplicaSet**

ReplicaSet гарантує, що вказана кількість реплік подів запущена в будь-який момент часу. Забезпечує:

* Масштабованість застосунків
* Відмовостійкість через автоматичне відновлення подів
* Декларативне визначення бажаного стану

**3. Deployment**

Deployment — це вищий рівень абстракції над ReplicaSet, що надає декларативні оновлення для подів та ReplicaSets. Забезпечує:

* Стратегії оновлення (Rolling Update, Recreate)
* Відстеження історії розгортань
* Можливість відкату до попередніх версій
* Паузу та продовження розгортання

**4. Service**

Service — це абстракція, що визначає логічний набір подів і політику доступу до них. Сервіси дозволяють слабке зв'язування між залежними подами. Типи сервісів:

* ClusterIP: внутрішня IP-адреса, доступна лише всередині кластера
* NodePort: експонує сервіс на статичному порту кожної ноди
* LoadBalancer: використовує зовнішній балансувальник навантаження провайдера хмари
* ExternalName: повертає DNS CNAME запису

**5. Volume**

Volume вирішує дві проблеми: втрату даних, коли контейнер перезапускається, і спільне використання даних між контейнерами в поді. Типи томів:

* emptyDir: тимчасове порожнє сховище
* hostPath: монтування файлу або директорії з файлової системи ноди
* persistentVolume: постійне сховище, незалежне від життєвого циклу подів
* configMap/secret: монтування конфігурацій та секретів як файлів

**6. Namespace**

Namespace забезпечує віртуальне розділення кластера Kubernetes:

* Ізоляція ресурсів для різних команд, проектів або застосунків
* Керування доступом на рівні namespace
* Призначення квот ресурсів для окремих проектів

**7. StatefulSet**

StatefulSet керує розгортанням і масштабуванням набору подів з гарантіями щодо порядку розгортання та унікальності:

* Стабільні, унікальні імена мережі
* Стабільне, постійне сховище
* Упорядкований, плавний розгортання і масштабування
* Упорядковане оновлення та видалення

**8. DaemonSet**

DaemonSet забезпечує запуск копії поду на всіх (або деяких) нодах кластера:

* Ідеально для агентів моніторингу
* Збір логів на кожній ноді
* Агенти зберігання даних

**9. Job і CronJob**

Job створює один або кілька подів і гарантує, що вказана кількість успішно завершиться. CronJob створює Jobs за розкладом, подібно до crontab в Unix системах.

**10. ConfigMap і Secret**

ConfigMap зберігає конфігураційні дані у форматі ключ-значення для використання подами. Secret подібний до ConfigMap, але призначений для чутливих даних (паролі, токени, ключі).

# Висновки до пункту 1

У першому розділі було проведено ґрунтовний аналіз існуючих рішень для моніторингу контейнеризованих середовищ та детально розглянуто принципи роботи Kubernetes як основної платформи для оркестрації контейнерів.

Аналіз комерційних платформ моніторингу, таких як DataDog та Dynatrace, показав, що вони мають потужний функціонал для роботи з Kubernetes, включаючи автоматичне виявлення компонентів, збір метрик, логів та трейсів, а також аналітичні можливості. Однак цим рішенням притаманні суттєві недоліки: висока вартість ліцензій, особливо при масштабуванні; зберігання даних у хмарній інфраструктурі стороннього провайдера, що може не відповідати політикам безпеки деяких організацій; закрита екосистема з обмеженими можливостями кастомізації.

Детальний огляд відкритих стеків моніторингу ELK (Elasticsearch, Logstash, Kibana) та Prometheus з Grafana дозволив виявити їх переваги та обмеження. ELK Stack забезпечує потужні можливості для збору та аналізу логів, але має високі вимоги до ресурсів та обмеження в роботі з метриками. Prometheus Stack є оптимальним для збору та аналізу метрик, використовує модель активного опитування (pull model) і добре інтегрується з Kubernetes, проте має обмежені можливості для роботи з логами.

Дослідження принципів роботи Kubernetes показало, що ця система має розподілену архітектуру типу "клієнт-сервер" з поділом на площину управління (control plane) та робочі вузли (worker nodes). Розуміння основних абстракцій Kubernetes, таких як Pod, ReplicaSet, Deployment, Service, Volume, Namespace, є критично важливим для проектування ефективної системи моніторингу.

Проведений аналіз дозволяє зробити висновок, що для створення масштабованої та гнучкої системи моніторингу в Kubernetes доцільно використовувати комбінацію відкритих рішень: Prometheus для збору та аналізу метрик, Elasticsearch для зберігання та індексації логів, Fluent Bit для збору логів та Grafana для візуалізації даних. Такий підхід дозволить забезпечити повне покриття моніторингом всіх аспектів роботи кластера та розгорнутих додатків, уникаючи обмежень комерційних рішень.

Результати даного аналізу є основою для подальшого проектування та розгортання системи моніторингу, що буде розглянуто в наступних розділах.

# 2 ЗОВНІШНЄ І ВНУТРІШНЄ ПРОЕКТУВАННЯ

## 2.1 Зовнішнє проектування

### 2.1.1 Функціональне призначення

Функціональним призначенням розроблюваної системи є забезпечення повного циклу збору, зберігання, аналізу та візуалізації телеметричних даних у середовищі Kubernetes. Система призначена для:

* Збору метрик стану та продуктивності компонентів Kubernetes-кластера (вузлів, подів, сервісів)
* Збору логів контейнеризованих додатків з усіх вузлів кластера
* Централізованого зберігання та індексації телеметричних даних
* Аналізу зібраних даних з можливістю виявлення аномалій та тенденцій
* Візуалізації телеметрії через інтерактивні інформаційні панелі
* Налаштування системи оповіщень для проактивного виявлення проблем
* Масштабування компонентів системи відповідно до розміру кластера та обсягу даних

Система інтегрується з Kubernetes API для динамічного виявлення нових компонентів інфраструктури та застосовує принципи "zero-configuration" моніторингу, що дозволяє автоматично виявляти та починати збір даних з нових компонентів без ручного втручання.

## 2.1.2 Експлуатаційне призначення

Експлуатаційне призначення розроблюваної системи включає:

* Моніторинг стану та продуктивності Kubernetes-кластерів у режимі реального часу
* Діагностику проблем продуктивності та надійності розподілених додатків
* Аналіз трендів використання ресурсів для планування потужностей інфраструктури
* Ретроспективний аналіз інцидентів для визначення їх першопричин
* Забезпечення дотримання угод про рівень обслуговування (SLA) через моніторинг ключових показників
* Підтримку DevOps-практик через інтеграцію з CI/CD-системами
* Забезпечення аудиту безпеки через централізований збір та аналіз логів
* Підтримку мультитенантності для ізоляції даних у спільній інфраструктурі моніторингу

Система розрахована на використання різними категоріями фахівців:

* Адміністраторами Kubernetes для моніторингу здоров'я кластера
* Розробниками для діагностики та оптимізації додатків
* DevOps-інженерами для автоматизації операційних процесів
* Фахівцями з інформаційної безпеки для виявлення потенційних загроз

Очікується, що система функціонуватиме в режимі 24/7, забезпечуючи безперервний збір телеметрії та доступ до історичних даних.

## 2.1.3 Функціональні вимоги

Розроблювана система має відповідати наступним функціональним вимогам:

### Збір метрик:

* Автоматичне виявлення та моніторинг вузлів, подів, контейнерів, сервісів та інших об'єктів Kubernetes
* Збір системних метрик (CPU, пам'ять, мережа, диск) з усіх компонентів кластера
* Підтримка кастомних метрик додатків через Prometheus API

### Збір логів:

* Збір логів з усіх контейнерів у кластері незалежно від їх розташування
* Підтримка різних форматів логів (JSON, plain text, структурований)
* Корреляція логів з метриками за часовими мітками і ідентифікаторами

### Зберігання даних:

* Масштабоване зберігання метрик і логів в Elasticsearch
* Конфігурована політика ротації та видалення даних

### Візуалізація:

* Набір попередньо налаштованих дашбордів Grafana для моніторингу Kubernetes
* Інтерактивні графіки з можливістю налаштування користувачем

### Аналіз та алертинг:

* Налаштування порогових значень для ключових метрик
* Попередньо сконфігуровані правила оповіщень для типових проблем Kubernetes

### Масштабування та надійність:

* Горизонтальне масштабування всіх компонентів системи
* Відмовостійкість і автоматичне відновлення при збоях
* Буферизація даних при недоступності компонентів зберігання
* Механізми обмеження навантаження та захисту від перевантаження

## 2.1.4 Вхідні та вихідні дані

### 2.1.4.1 Вхідні дані

Вхідними даними для системи моніторингу та аналізу телеметрії в Kubernetes є:

1. **Метрики рівня інфраструктури:**
   * Утилізація CPU (відсоток використання, навантаження)
   * Використання оперативної пам'яті (загальний обсяг, використано, доступно, кеш)
   * Використання дискового простору (загальний обсяг, використано, доступно, I/O операції)
2. **Метрики Kubernetes:**
   * Стан вузлів (готовність, навантаження, кількість подів)
   * Стан подів (фаза життєвого циклу, рестарти, готовність)
   * Використання ресурсів на рівні подів та контейнерів
3. **Логи:**
   * Системні логи контейнерів (stdout, stderr)
   * Логи додатків з файлів всередині контейнерів
   * Логи компонентів Kubernetes (api-server, scheduler, controllers)
4. **Конфігураційні дані:**
   * Kubernetes API об'єкти (pods, deployments, services)
   * Конфігурація мережі та безпеки
   * Анотації та мітки ресурсів Kubernetes
   * Змінні середовища контейнерів
   * Секрети та конфігмапи (у безпечному форматі)
5. **Дані про події:**
   * Kubernetes events (створення, оновлення, видалення ресурсів)
   * Повідомлення про збої та відновлення

Формати вхідних даних включають:

* Prometheus метрики у форматі text/plain або OpenMetrics
* Логи у форматах plain text, JSON, structured logging
* API-дані у форматі JSON
* Конфігураційні файли у форматах YAML, JSON, INI

### 2.1.4.2 Вихідні дані

Вихідними даними системи моніторингу та аналізу телеметрії в Kubernetes є:

1. **Візуалізовані дані у вигляді дашбордів Grafana:**
   * Оперативні дашборди стану кластера з зведеною інформацією
   * Деталізовані графіки використання ресурсів за часовими рядами
   * Панелі перегляду та аналізу логів
2. **Аналітичні звіти:**
   * Звіти про використання ресурсів з прогнозом трендів
   * Аналіз аномалій та викидів у метриках
   * Аналіз помилок та їх кореляцій
3. **Оповіщення (алерти):**
   * Сповіщення про перевищення порогових значень метрик
   * Повідомлення про аномалії в роботі системи
4. **Дані для API та інтеграцій:**
   * JSON-відповіді на запити до API Grafana та Elasticsearch
   * Експортовані дані для CI/CD систем та автоматизації
5. **Експортовані файли:**
   * Вивантажені дані у форматах CSV, JSON, Excel
   * Архівовані логи та метрики для довгострокового зберігання
   * Резервні копії конфігурацій та правил оповіщень
6. **Дані аудиту та безпеки:**
   * Журнали доступу до системи моніторингу
   * Статистика доступу до даних за користувачами та ролями

Вихідні дані надаються через наступні інтерфейси:

* Веб-інтерфейс Grafana для інтерактивної взаємодії
* REST API Elasticsearch для прямого доступу до даних
* API Grafana для програмного доступу до дашбордів

## 2.1.5 Опис зовнішнього інформаційного середовища

Система збору й аналізу телеметрії в Kubernetes функціонує в складному інформаційному середовищі, взаємодіючи з різними компонентами інфраструктури та зовнішніми системами:

### Взаємодія з Kubernetes-кластером:

* **Kubernetes API Server** - основне джерело інформації про стан кластера та його ресурси. Система використовує API для отримання даних про поди, вузли, сервіси та інші об'єкти Kubernetes, а також для відстеження змін у конфігурації.
* **Kubelet** - агент на кожному вузлі кластера, який надає дані про стан вузла та контейнерів, що на ньому виконуються. Взаємодія відбувається через API або напряму через зчитування метрик з cAdvisor.
* **контейнери/поди** - основні одиниці розгортання додатків, які виробляють логи та метрики. Система збирає stdout/stderr потоки, а також експоновані метрики через HTTP endpoints.

### Середовище зберігання та обчислення:

* **Постійне сховище** (PersistentVolumes в Kubernetes) - використовується для зберігання даних Elasticsearch та конфігурацій системи.
* **Мережеве середовище** - включає внутрішню мережу Kubernetes та зовнішні мережеві компоненти (LoadBalancers, Ingress).
* **Обчислювальні ресурси** (CPU, пам'ять) - розподіляються між компонентами системи моніторингу з урахуванням пріоритетів та QoS.

### Інтерфейси взаємодії з користувачами:

* **Веб-інтерфейси** (Grafana, Kibana) - забезпечують візуальний доступ до даних моніторингу через HTTP/HTTPS.
* **API інтерфейси** - надають програмний доступ до даних та функцій системи моніторингу.
* **CLI інструменти** - забезпечують командний інтерфейс для взаємодії з системою моніторингу.

### Обмеження та вимоги середовища:

* **Політики безпеки** - визначають обмеження на збір та зберігання чутливих даних.
* **Мережеві політики** - регулюють доступ між компонентами системи та зовнішнім світом.
* **Квоти ресурсів** - обмежують використання обчислювальних ресурсів компонентами моніторингу.

Система спроектована з урахуванням усіх цих аспектів зовнішнього інформаційного середовища для забезпечення безперебійної роботи та інтеграції з існуючими IT-процесами організації.

## 2.2 Внутрішнє проектування

### 2.2.1 Аналіз зовнішніх специфікацій систем

При проектуванні системи збору й аналізу телеметрії в Kubernetes необхідно враховувати специфікації та API зовнішніх компонентів, з якими відбуватиметься інтеграція. Такий аналіз дозволяє забезпечити сумісність та ефективну взаємодію між різними частинами архітектури.

#### Kubernetes API

Kubernetes API є основним інтерфейсом для взаємодії з кластером і надає доступ до ресурсів через RESTful HTTP API:

* **Формат даних**: JSON
* **Методи аутентифікації**: сертифікати x509, Bearer tokens, OIDC
* **Версіонування API**: стабільні (наприклад, v1), бета (v1beta1) та альфа (v1alpha1) версії
* **Ресурси, що використовуються**:
  + /api/v1/nodes - інформація про вузли кластера
  + /api/v1/pods - інформація про поди
  + /api/v1/namespaces - доступ до просторів імен
  + /apis/apps/v1/deployments - дані про розгортання
  + /apis/metrics.k8s.io/v1beta1 - метрики ресурсів (через Metrics Server)

Для роботи з Kubernetes API система використовуватиме офіційні клієнтські бібліотеки та SDK.

#### Prometheus API

Prometheus надає API для збору та запиту метрик:

* **Формат даних**: специфічний текстовий формат Prometheus, JSON для HTTP API
* **Endpoint для scraping**: зазвичай /metrics на цільових сервісах
* **Query API**:
  + /api/v1/query - для виконання миттєвих запитів
  + /api/v1/query\_range - для запитів за часовим діапазоном
  + /api/v1/series - для пошуку серій часових рядів
* **Формат запитів**: PromQL (Prometheus Query Language)

#### Elasticsearch API

Elasticsearch забезпечує потужний RESTful API для зберігання, пошуку та аналізу даних:

* **Формат даних**: JSON
* **Основні endpoint'и**:
  + /{index}/\_doc - індексація документів
  + /{index}/\_search - пошук документів
  + /{index}/\_update - оновлення документів
  + /\_cluster/health - перевірка стану кластера
* **Мова запитів**: Elasticsearch Query DSL
* **API версіонування**: семантичне версіонування (8.x, 9.x)

#### Grafana API

Grafana надає HTTP API для взаємодії з дашбордами та іншими ресурсами:

* **Формат даних**: JSON
* **Основні endpoint'и**:
  + /api/dashboards - управління дашбордами
  + /api/datasources - управління джерелами даних
  + /api/alerts - управління оповіщеннями
  + /api/user - управління користувачами
* **Особливості**: підтримка організацій (multi-tenancy)

#### Протоколи взаємодії

Для забезпечення взаємодії між компонентами системи використовуються такі протоколи:

* **HTTP/HTTPS**: основний протокол для взаємодії з API
* **TCP/UDP**: для передачі логів (Syslog, Fluentd forward protocol)

Аналіз цих специфікацій дозволяє правильно спроектувати точки інтеграції, формати даних та механізми взаємодії між компонентами розроблюваної системи.

### 2.2.1.1 Моделювання словника системи

Для забезпечення єдиного розуміння і використання термінів у розроблюваній системі збору та аналізу телеметрії в Kubernetes, необхідно визначити основні сутності, що становлять словник системи. Кожна сутність описується з точки зору її атрибутів, взаємозв'язків та операцій.

#### Основні сутності системи:

**1. Метрика (Metric)**

Представляє вимірювану величину, що характеризує стан або продуктивність компонента системи.

Атрибути:

* name - унікальний ідентифікатор метрики
* value - числове значення метрики
* timestamp - час вимірювання
* labels - набір ключ-значення для категоризації та фільтрації
* type - тип метрики (counter, gauge, histogram, summary)
* unit - одиниця вимірювання (bytes, seconds, percent)
* source - джерело метрики (компонент, що її виробляє)

Операції:

* Збір (collect)
* Агрегація (aggregate)
* Фільтрація (filter)
* Візуалізація (visualize)

**2. Лог (Log)**

Запис про подію, що сталася в системі, містить текстове повідомлення та метадані.

Атрибути:

* message - текстове повідомлення
* timestamp - час створення запису
* level - рівень важливості (info, warning, error, debug)
* source - джерело (под, контейнер, компонент)
* namespace - простір імен Kubernetes
* pod\_name - ім'я поду
* container\_name - ім'я контейнера
* labels - додаткові мітки для категоризації
* trace\_id - ідентифікатор трасування (для розподілених викликів)

Операції:

* Збір (collect)
* Парсинг (parse)
* Фільтрація (filter)
* Пошук (search)
* Аналіз (analyze)

**3. Оповіщення (Alert)**

Повідомлення про перевищення порогового значення або виявлення аномалії.

Атрибути:

* name - назва оповіщення
* status - статус (firing, resolved, pending)
* severity - серйозність (critical, warning, info)
* timestamp - час виникнення
* value - значення, що викликало оповіщення
* description - опис проблеми
* labels - мітки для категоризації
* annotations - додаткові відомості (посилання, інструкції)
* silenced - чи заглушено оповіщення
* acknowledged - чи підтверджено отримання

Операції:

* Створення (create)
* Відправка (send)
* Заглушення (silence)
* Підтвердження (acknowledge)
* Розв'язання (resolve)

**4. Дашборд (Dashboard)**

Візуальне представлення набору метрик та інших даних моніторингу.

Атрибути:

* title - назва дашборду
* uid - унікальний ідентифікатор
* version - версія
* panels - набір панелей з візуалізаціями
* variables - змінні для фільтрації/налаштування
* timeRange - діапазон часу за замовчуванням
* owner - власник дашборду
* tags - теги для категоризації
* permissions - права доступу

Операції:

* Створення (create)
* Оновлення (update)
* Видалення (delete)
* Експорт (export)
* Імпорт (import)
* Перегляд (view)

**5. Ресурс Kubernetes (K8sResource)**

Об'єкт Kubernetes, що є джерелом метрик та логів.

Атрибути:

* kind - тип ресурсу (Pod, Deployment, Service)
* name - ім'я ресурсу
* namespace - простір імен
* uid - унікальний ідентифікатор
* labels - мітки, присвоєні ресурсу
* annotations - анотації з додатковою інформацією
* creationTimestamp - час створення
* status - поточний статус
* spec - специфікація конфігурації

Операції:

* Виявлення (discover)
* Моніторинг (monitor)
* Відстеження життєвого циклу (track lifecycle)
* Збір метрик (collect metrics)
* Збір логів (collect logs)

**6. Агент збору даних (CollectionAgent)**

Компонент, відповідальний за збір телеметрії з джерел.

Атрибути:

* type - тип агента (Prometheus exporter, Fluent Bit, etc)
* version - версія агента
* status - статус (running, failed, etc)
* configuration - параметри конфігурації
* resources - виділені ресурси (CPU, memory)
* targetSelectors - селектори цілей для збору даних
* scrapeInterval - інтервал збору даних
* bufferSize - розмір буфера для тимчасового зберігання

Операції:

* Розгортання (deploy)
* Конфігурування (configure)
* Збір даних (collect)
* Фільтрація (filter)
* Відправка даних (send)
* Моніторинг стану (health check)

**7. Сховище даних (DataStore)**

Компонент для зберігання та індексації зібраної телеметрії.

Атрибути:

* type - тип сховища (Elasticsearch, Prometheus, etc)
* version - версія
* status - статус (healthy, degraded)
* capacity - ємність (в ГБ або кількості документів)
* retention - політика зберігання даних
* indices - індекси або бази даних
* replicationFactor - фактор реплікації
* nodes - кількість вузлів сховища
* performance - метрики продуктивності

Операції:

* Індексація (index)
* Запит (query)
* Масштабування (scale)
* Резервне копіювання (backup)
* Відновлення (restore)
* Ротація даних (rotate)

**8. Правило оповіщення (AlertRule)**

Визначає умови, при яких генерується оповіщення.

Атрибути:

* name - ім'я правила
* expression - вираз або умова для спрацювання
* for - тривалість, протягом якої умова має виконуватись
* severity - серйозність оповіщення
* labels - мітки, що будуть присвоєні оповіщенню
* annotations - додаткові відомості
* enabled - чи активне правило
* notificationChannels - канали для відправки оповіщень

Операції:

* Створення (create)
* Оновлення (update)
* Видалення (delete)
* Активація (enable)
* Деактивація (disable)
* Тестування (test)

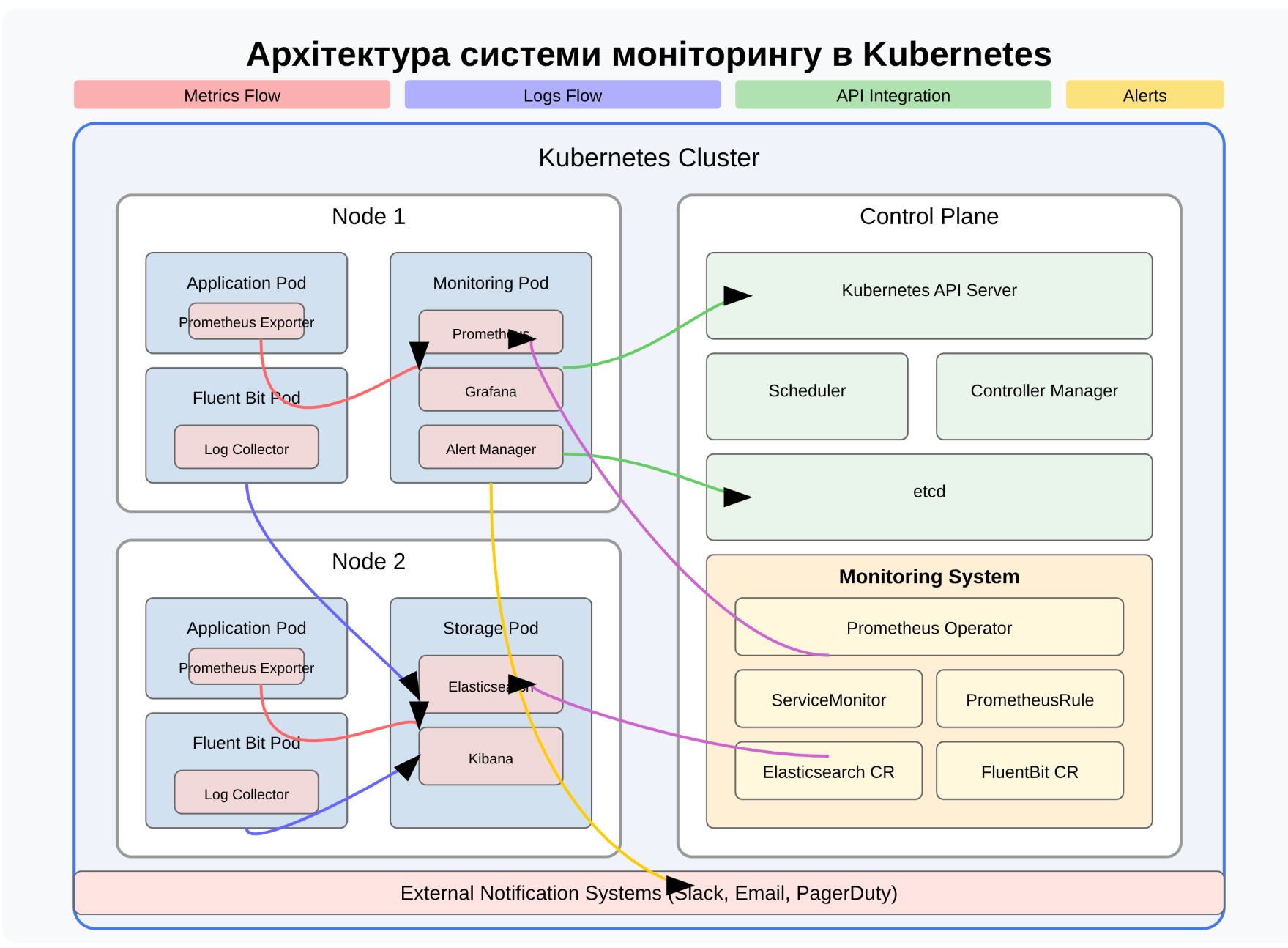
Моделювання словника системи допомагає забезпечити однозначне розуміння компонентів і процесів усіма учасниками проекту та є основою для подальшого детального проектування.

2.2.2 Проектування архітектури системи

Архітектура системи для збору й аналізу телеметрії в Kubernetes повинна бути розроблена з урахуванням вимог до масштабованості, надійності та ефективності. Система складається з декількох взаємопов'язаних компонентів, кожен з яких відповідає за окремий аспект моніторингу.

### 2.2.2.1 Загальний огляд архітектури

Архітектура системи моніторингу розроблена за мікросервісним принципом, де кожен компонент виконує чітко визначену функцію та може масштабуватися незалежно від інших, наведена на рис 1.5. Система розгортається повністю в середовищі Kubernetes, що забезпечує її портативність та спрощує управління життєвим циклом.

(Рис. 1.5 Архітектура системи моніторингу)

Основні компоненти системи:

**Колектори метрик** - відповідають за збір метричних даних з різних джерел

1. **Колектори логів** - забезпечують збір, парсинг та фільтрацію логів
2. **Сховища даних** - зберігають зібрані метрики та логи
3. **Підсистема візуалізації** - надає інтерфейс для перегляду та аналізу даних
4. **Підсистема оповіщень** - відстежує аномалії та відправляє сповіщення
5. **Оператори Kubernetes** - керують розгортанням та конфігурацією компонентів

### 2.2.2.2 Компоненти системи

#### Компоненти збору метрик:

* **Prometheus** - основний компонент для збору та зберігання метрик. Розгортається як StatefulSet для забезпечення постійного зберігання даних.
* **Prometheus Exporters** - спеціалізовані агенти, що збирають метрики з різних джерел та експонують їх у форматі Prometheus. Включають:
  + Node Exporter - збирає системні метрики з вузлів Kubernetes
  + kube-state-metrics - надає метрики про стан об'єктів Kubernetes
  + Custom Exporters - специфічні експортери для додатків
* **Prometheus Operator** - Kubernetes оператор, що спрощує розгортання та конфігурацію компонентів Prometheus.
* **ServiceMonitor** та **PodMonitor** - Custom Resource Definitions (CRDs), що визначають, які сервіси та поди моніторити.

#### Компоненти збору логів:

* **Fluent Bit** - легкий агент для збору логів, що розгортається на кожному вузлі кластера як DaemonSet.
* **Elasticsearch** - розподілене сховище для індексації та пошуку логів.
* **Elasticsearch Operator** - Kubernetes оператор для управління розгортанням Elasticsearch.
* **Kibana** - веб-інтерфейс для пошуку та аналізу логів в Elasticsearch.

#### Компоненти візуалізації:

* **Grafana** - платформа для створення дашбордів та візуалізації метрик.
* **Grafana Operator** - Kubernetes оператор для управління розгортанням Grafana.
* **Dashboards as Code** - дашборди, визначені у форматі JSON та розгорнуті через ConfigMaps.

#### Компоненти оповіщень:

* **Alertmanager** - компонент Prometheus, що відповідає за обробку оповіщень.
* **PrometheusRule** - CRD для визначення правил оповіщень.
* **Notification Channels** - шлюзи для відправки оповіщень через різні канали (Slack, Email, PagerDuty).

#### Інфраструктурні компоненти:

* **Persistent Volumes** - для зберігання даних Prometheus, Elasticsearch та Grafana.
* **Service Accounts** та **RBAC** - для контролю доступу до ресурсів Kubernetes.
* **NetworkPolicies** - для обмеження мережевого трафіку між компонентами.
* **Resource Quotas** - для обмеження використання ресурсів.

### 2.2.2.3 Топологія та взаємодія компонентів

Взаємодія компонентів системи відбувається за наступними принципами:

1. **Збір метрик**:
   * Prometheus періодично опитує (scrapes) експортери за визначеними endpoints.
   * ServiceMonitor визначає, які сервіси та з якою частотою опитувати.
   * Зібрані метрики зберігаються локально в Prometheus та можуть бути федеровані у центральний Prometheus.
2. **Збір логів**:
   * Fluent Bit збирає логи з файлів логів контейнерів та журналів системи.
   * Зібрані логи парсяться, фільтруються та збагачуються метаданими Kubernetes.
   * Оброблені логи передаються в Elasticsearch для індексації та зберігання.
3. **Візуалізація та аналіз**:
   * Grafana підключається до Prometheus як джерела даних для візуалізації метрик.
   * Kibana підключається до Elasticsearch для пошуку та аналізу логів.
   * Користувачі взаємодіють з системою через веб-інтерфейси Grafana та Kibana.
4. **Оповіщення**:
   * Prometheus оцінює правила оповіщень на основі зібраних метрик.
   * При спрацюванні правила, оповіщення передається в Alertmanager.
   * Alertmanager групує, дедуплікує та маршрутизує оповіщення до відповідних каналів сповіщення.

### 2.2.2.4 Масштабованість та відмовостійкість

Архітектура системи забезпечує масштабованість та відмовостійкість через:

1. **Горизонтальне масштабування**:
   * Elasticsearch може масштабуватися горизонтально через додавання нових вузлів.
   * Prometheus може федерувати дані між екземплярами для масштабування.
2. **Шардинг та реплікація**:
   * Elasticsearch використовує шардинг для розподілу навантаження та реплікацію для відмовостійкості.
   * Prometheus підтримує функціонал high availability через розгортання декількох реплік.
3. **Автоматичне відновлення**:
   * Kubernetes автоматично перезапускає контейнери, що вийшли з ладу.
   * StatefulSets забезпечують стабільні ідентифікатори для екземплярів Prometheus та Elasticsearch.

Архітектура системи представлена на рис. 1.5, що ілюструє компоненти та їх взаємодію в середовищі Kubernetes.

## 2.2.4 Проектування підсистеми збору та аналізу логів

Підсистема збору та аналізу логів відповідає за ефективний збір, обробку, зберігання та аналіз логів з усіх компонентів Kubernetes-кластера та розгорнутих додатків. Ця підсистема є критичною для діагностики проблем, розслідування інцидентів та аудиту дій.

### 2.2.4.1 Архітектура підсистеми логування

Підсистема базується на поєднанні компонентів ELK Stack (Elasticsearch, Logstash, Kibana) з легким агентом Fluent Bit для ефективного збору логів. Загальна архітектура підсистеми включає:

1. **Збір логів** - Fluent Bit агенти на кожному вузлі кластера
2. **Обробка та трансформація** - фільтрація, парсинг та збагачення логів
3. **Зберігання та індексація** - кластер Elasticsearch
4. **Пошук та візуалізація** - Kibana для інтерактивного дослідження логів

### 2.2.4.2 Конфігурація збирача логів Fluent Bit

Fluent Bit розгортається як DaemonSet на кожному вузлі кластера для забезпечення повного покриття збору логів. У конфігурації Fluent Bit передбачено:

* Збір логів з файлової системи (/var/log/containers/\*.log)
* Збагачення логів метаданими Kubernetes (простір імен, назва поду, контейнера)
* Фільтрація системних логів для зменшення обсягу даних
* Безпечну передачу логів до Elasticsearch з підтримкою TLS
* Обробку помилок та спроби повторної відправки у разі недоступності сховища

### 2.2.4.3 Налаштування Elasticsearch для зберігання логів

Elasticsearch розгортається як StatefulSet з використанням ECK (Elastic Cloud on Kubernetes) оператора для автоматизації управління життєвим циклом. Архітектура Elasticsearch включає:

* Розділення ролей вузлів на master (для координації кластера) та data (для зберігання даних)
* Виділені ресурси залежно від ролі вузла та очікуваного навантаження
* Постійне зберігання даних з використанням Persistent Volumes
* Оптимізацію налаштувань JVM та кеш-пам'яті залежно від доступних ресурсів

### 2.2.4.4 Структура індексів та шаблони індексів Elasticsearch

Для оптимізації зберігання та пошуку логів розроблено шаблон індексу з оптимальним налаштуванням відображення полів:

* Структура індексів у форматі "kubernetes-YYYY.MM.DD" для логічного розділення даних за часом
* Оптимізація типів полів для ефективного пошуку (keyword для точного пошуку, text для повнотекстового)
* Налаштування кількості шардів та реплік залежно від обсягу даних та вимог до доступності
* Спеціальні відображення для полів Kubernetes для покращення фільтрації та агрегації

### 2.2.4.5 Управління життєвим циклом даних у Elasticsearch

Для ефективного управління даними логів налаштовано політику життєвого циклу індексів (ILM):

* Фаза "hot" - активне індексування і часті запити, зберігання на швидких дисках
* Фаза "warm" - менш активні дані, оптимізація (об'єднання сегментів, зменшення кількості шардів)
* Фаза "cold" - рідко використовувані дані, перенесення на більш повільні, але дешевші носії
* Фаза "delete" - автоматичне видалення даних після визначеного періоду (90 днів)

### 2.2.4.6 Конфігурація Kibana для візуалізації та аналізу логів

Kibana розгортається за допомогою ECK оператора для забезпечення інтерфейсу пошуку та аналізу логів. Конфігурація включає:

* Масштабування до декількох реплік для забезпечення високої доступності
* Інтеграція з Elasticsearch для аутентифікації та авторизації
* Попередньо налаштовані дашборди для поширених сценаріїв аналізу
* Налаштування візуалізацій для типових патернів запитів

### 2.2.4.7 Парсери для структурованих логів

Для ефективного парсингу та структурування логів різних форматів налаштовано парсери Fluent Bit:

* Парсер для стандартного формату контейнерних логів Docker
* Парсер для системних логів у форматі syslog-rfc5424
* Спеціалізовані парсери для Java Exception стек-трейсів
* Гнучка система парсингу на основі регулярних виразів для специфічних форматів логів додатків

### 2.2.4.8 Механізми безпеки та контролю доступу

Для забезпечення безпеки підсистеми логування реалізовано наступні механізми:

1. **Шифрування даних**:
   * TLS для передачі даних між компонентами
   * Шифрування даних у стані спокою для Elasticsearch
2. **Автентифікація та авторизація**:
   * Інтеграція з RBAC Kubernetes
   * Налаштування ролей Elasticsearch для обмеження доступу до даних
3. **Маскування чутливих даних**:
   * Фільтрація чутливої інформації (паролі, токени) перед зберіганням

### 2.2.4.9 Інтеграція з Grafana для уніфікованого доступу

Для забезпечення централізованого доступу до всіх видів телеметрії налаштовано інтеграцію логів Elasticsearch з Grafana:

* Налаштування Elasticsearch як джерела даних у Grafana
* Створення комбінованих дашбордів з метриками та логами
* Налаштування кореляції між логами та метриками за часовими мітками
* Використання Loki як альтернативного джерела логів для спрощеної інтеграції

Підсистема збору та аналізу логів спроектована для ефективної обробки великих обсягів даних з високою доступністю та надійністю. Використання легкого агента Fluent Bit для збору та потужного Elasticsearch для зберігання забезпечує оптимальний баланс між ефективністю використання ресурсів та функціональністю системи.

Є ще декілька важливих підрозділів, які варто включити до другого розділу для повноти висвітлення проектування системи моніторингу. Давайте продовжимо з наступними частинами:

## 2.2.5 Проектування підсистеми візуалізації та дашбордів

Підсистема візуалізації відіграє критичну роль у представленні зібраних телеметричних даних у зрозумілому та інформативному вигляді. Графічне представлення метрик та логів забезпечує швидке розуміння стану системи та спрощує виявлення аномалій.

### 2.2.5.1 Архітектура підсистеми візуалізації

Підсистема візуалізації базується на Grafana як основному інструменті створення дашбордів і візуалізацій. Архітектура включає:

* Розгортання Grafana як StatefulSet для збереження конфігурацій та дашбордів
* Інтеграцію з різними джерелами даних (Prometheus, Elasticsearch)
* Механізми автоматичного створення та оновлення дашбордів
* Підтримку користувацьких ролей та дозволів

### 2.2.5.2 Типи дашбордів та їх призначення

Для забезпечення повного покриття моніторингом різних аспектів системи розроблено наступні типи дашбордів:

1. **Оглядові дашборди**:
   * Загальний огляд стану кластера
   * Основні показники ресурсів (CPU, пам'ять, диск, мережа)
   * Агреговані метрики доступності та продуктивності
2. **Інфраструктурні дашборди**:
   * Детальний моніторинг вузлів Kubernetes
   * Моніторинг компонентів control plane
   * Моніторинг мережевих компонентів
3. **Дашборди додатків**:
   * Моніторинг специфічних для додатків метрик
   * Візуалізація бізнес-метрик
   * JVM-метрики для Java-додатків
4. **Дашборди для логів**:
   * Візуалізація та аналіз потоків логів
   * Кореляція логів з метриками
   * Аналіз помилок та винятків
5. **Дашборди аналізу продуктивності**:
   * Історичні тренди використання ресурсів
   * Порівняння метрик до та після змін
   * Аналіз вузьких місць системи

### 2.2.5.3 Методи візуалізації даних

Для ефективного представлення різних типів даних використовуються різноманітні методи візуалізації:

* **Графіки часових рядів** для відображення змін метрик у часі
* **Теплові карти** для візуалізації розподілу значень
* **Гістограми** для аналізу розподілу затримок та інших показників
* **Таблиці** для детального перегляду числових даних
* **Геокарти** для візуалізації географічного розподілу запитів
* **Діаграми Sankey** для візуалізації потоків трафіку між сервісами
* **Діаграми стану** для моніторингу здоров'я компонентів

### 2.2.5.4 Шаблонізація та змінні в дашбордах

Для забезпечення гнучкості та повторного використання дашбордів реалізовано систему шаблонізації:

* **Змінні дашбордів** для динамічної фільтрації даних (простори імен, поди, сервіси)
* **Змінні запитів** для динамічного формування списків значень на основі даних
* **Інтервальні змінні** для зміни часового діапазону аналізу
* **Шаблонні запити** для повторного використання запитів у різних панелях

### 2.2.5.5 Автоматизація створення дашбордів

Для спрощення управління дашбордами та забезпечення їх відтворюваності реалізовано систему автоматизації:

* **Дашборди як код** - зберігання конфігурацій дашбордів у репозиторії
* **Граф. Operator** для автоматичного створення та оновлення дашбордів
* **CI/CD інтеграція** для автоматичного розгортання змін
* **Бібліотека панелей** для повторного використання стандартних елементів

## 2.2.6 Проектування підсистеми оповіщень

Підсистема оповіщень забезпечує своєчасне виявлення та сповіщення про проблеми в інфраструктурі та додатках, дозволяючи швидко реагувати на інциденти та мінімізувати їх вплив.

### 2.2.6.1 Архітектура підсистеми оповіщень

Підсистема оповіщень базується на Prometheus AlertManager та включає:

* Правила оповіщень, визначені як PrometheusRules
* AlertManager для обробки, групування та маршрутизації оповіщень
* Інтеграцію з різними каналами сповіщень (Slack, Email, PagerDuty)
* Механізми дедуплікації та групування оповіщень

### 2.2.6.2 Категорії оповіщень

Оповіщення розділено на категорії залежно від їх критичності та джерела:

1. **Критичні оповіщення**:
   * Недоступність ключових компонентів
   * Висока утилізація ресурсів (>90%)
   * Проблеми з доступністю сервісів
2. **Попереджувальні оповіщення**:
   * Підвищення утилізації ресурсів (>75%)
   * Збільшення затримки відповіді
   * Аномалії в метриках
3. **Інформаційні оповіщення**:
   * Планове масштабування додатків
   * Перевищення порогів, що не впливають на роботу
   * Тренди використання ресурсів

### 2.2.6.3 Правила оповіщень

Розроблено набір правил оповіщень для різних компонентів системи:

1. **Оповіщення для інфраструктури**:
   * Високе навантаження на CPU вузлів
   * Низький об'єм вільної пам'яті
   * Проблеми з дисковим простором
   * Аномалії в мережевому трафіку
2. **Оповіщення для Kubernetes**:
   * Недоступність компонентів control plane
   * Проблеми з планувальником (scheduler)
   * Поди у стані CrashLoopBackOff
   * Проблеми з PersistentVolumes
3. **Оповіщення для додатків**:
   * Високий рівень помилок у відповідях
   * Підвищена затримка відповіді
   * Аномалії в бізнес-метриках
   * Проблеми з підключенням до зовнішніх сервісів

### 2.2.6.4 Маршрутизація та ескалація оповіщень

Для ефективної обробки оповіщень реалізовано стратегію маршрутизації та ескалації:

* **Маршрутизація за критичністю** - різні канали для різних рівнів критичності
* **Маршрутизація за командами** - направлення оповіщень відповідальним командам
* **Часова маршрутизація** - різні отримувачі залежно від часу доби та дня тижня
* **Ескалація** - автоматична ескалація невирішених інцидентів до більш високого рівня

### 2.2.6.5 Інтеграція з системами управління інцидентами

Для ефективного управління інцидентами реалізовано інтеграцію з системами управління інцидентами:

* **PagerDuty** для ескалації та управління чергуваннями
* **Jira** для створення та відстеження тікетів
* **Slack** для командної комунікації та швидкого реагування
* **OpsGenie** для управління чергуваннями та ескалаціями

## 2.2.7 Проектування механізмів масштабування та високої доступності

Для забезпечення надійності та продуктивності системи моніторингу при зростанні навантаження розроблено механізми масштабування та забезпечення високої доступності.

### 2.2.7.1 Стратегії масштабування компонентів

Для кожного компонента системи розроблено стратегію масштабування:

1. **Масштабування сховищ даних**:
   * Горизонтальне масштабування Elasticsearch через додавання вузлів
   * Шардування індексів Elasticsearch для паралельної обробки
   * Архівація старих даних у холодне сховище
2. **Масштабування візуалізації**:
   * Балансування навантаження між екземплярами Grafana
   * Кешування запитів для зменшення навантаження на сховища даних
   * Оптимізація запитів для покращення продуктивності

### 2.2.7.2 Забезпечення високої доступності

Для забезпечення безперервної роботи системи моніторингу реалізовано наступні механізми:

1. **Автоматичне відновлення**:
   * Kubernetes Liveness та Readiness проби для виявлення проблем
   * Автоматичне перезапускання контейнерів при відмові
   * Відновлення стану з персистентних томів

# Висновки до пункту 2

У другому розділі було проведено комплексне проектування масштабованої системи для збору й аналізу телеметрії в середовищі Kubernetes з використанням Elasticsearch і Grafana. Розроблена архітектура системи відповідає сучасним підходам до моніторингу розподілених систем та забезпечує всі необхідні функціональні вимоги.

На етапі зовнішнього проектування було визначено функціональне та експлуатаційне призначення системи, сформульовано детальні функціональні вимоги та проаналізовано вхідні й вихідні дані. Описано зовнішнє інформаційне середовище, в якому функціонуватиме система, включаючи взаємодію з кластером Kubernetes та зовнішніми системами.

Внутрішнє проектування включало аналіз зовнішніх специфікацій систем та моделювання словника системи, що забезпечило створення спільної термінології та розуміння між усіма учасниками проекту. Розроблено детальну архітектуру системи моніторингу, яка складається з таких ключових компонентів:

1. Підсистема збору метрик на базі Prometheus, що забезпечує ефективний збір та зберігання метрик продуктивності та стану інфраструктури й додатків.
2. Підсистема збору та аналізу логів, яка використовує Fluent Bit для збору логів і Elasticsearch для їх зберігання та індексації, що дозволяє ефективно обробляти та аналізувати великі обсяги логів.
3. Підсистема візуалізації на базі Grafana, яка надає гнучкі можливості для створення інформативних дашбордів та візуалізацій метрик і логів.
4. Підсистема оповіщень на базі AlertManager, що забезпечує своєчасне виявлення та повідомлення про проблеми в інфраструктурі та додатках.

Особлива увага була приділена проектуванню механізмів масштабування та забезпечення високої доступності системи. Розроблені стратегії масштабування компонентів та механізми автоматичного відновлення дозволяють системі ефективно функціонувати в умовах зростаючого навантаження та адаптуватися до змін у моніторинговому середовищі.

Кожна підсистема спроектована з урахуванням необхідності інтеграції з іншими компонентами, що забезпечує цілісний підхід до збору, зберігання, аналізу та візуалізації телеметричних даних.

Розроблена архітектура системи моніторингу:

* Використовує кращі практики та патерни для розгортання в Kubernetes
* Забезпечує ефективне використання ресурсів кластера
* Підтримує горизонтальне масштабування та відмовостійкість
* Має гнучку систему управління життєвим циклом даних
* Надає різноманітні можливості для візуалізації та аналізу телеметрії

Запропоноване рішення є всеосяжним та може бути адаптоване для моніторингу різних типів Kubernetes-кластерів та розгорнутих додатків, що робить його універсальним інструментом для DevOps-команд та адміністраторів системи.

# 3 ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ

## 3.1 Розгортання тестового середовища

Для тестування розробленої системи моніторингу було створено експериментальне середовище на базі Minikube — локальної версії Kubernetes, що дозволяє запустити кластер на одній машині. Minikube підходить для початкового тестування, оскільки забезпечує всі основні функції повноцінного Kubernetes кластера.

Параметри тестового середовища:

* Kubernetes версії v1.32.0
* Контейнеризація на базі Docker v0.40.8
* Розгортання компонентів за допомогою Helm v3
* Операційна система: Fedora Linux 42 (Workstation Edition)

Для розгортання системи моніторингу було використано Helm chart Prometheus, що включає всі необхідні компоненти для побудови базової системи моніторингу:

* Prometheus сервер
* Alertmanager
* kube-state-metrics
* node-exporter
* Grafana

### 3.1.1 Підготовка інфраструктури

Розгортання тестового кластера Minikube виконувалось з наступними параметрами:

**minikube start --cpus=4 --memory=8g --driver=docker**

Після успішного запуску Minikube було перевірено стан кластера:

kubectl cluster-info

*Kubernetes control plane is running at https://localhost:42203*

*CoreDNS is running at https://localhost:42203/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy*

*To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.*

## 3.2 Розгортання компонентів системи моніторингу

### 3.2.1 Встановлення Prometheus Stack

Для розгортання стеку Prometheus було використано офіційний Helm chart. Спочатку додано відповідний репозиторій та оновлено список доступних пакетів:

**helm repo add prometheus-community https://prometheus-community.github.io/helm-charts**

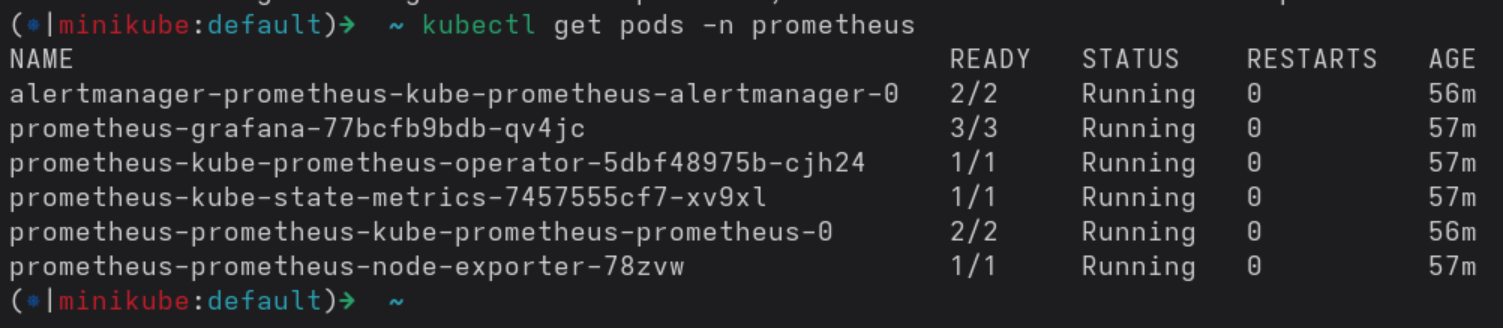
**helm repo update**

**helm install prometheus prometheus-community/kube-prometheus-stack --namespace prometheus —create-namespace**

Після завершення розгортання було перевірено статус розгорнутих подів:

kubectl get pods -n prometheus

Як видно з наведеного знімку екрана (рисунок 3.1), всі компоненти системи моніторингу успішно запущені та функціонують:

Рисунок 3.1 — Компоненти системи моніторингу в Kubernetes

Серед розгорнутих компонентів:

* alertmanager-prometheus-kube-prometheus-alertmanager-0
* prometheus-grafana-5693f9bdb-qv4jc
* prometheus-kube-prometheus-operator-5dbf48975b-cjh24
* prometheus-kube-state-metrics-7457555cf7-xv9xl
* prometheus-prometheus-kube-prometheus-prometheus-0
* prometheus-prometheus-node-exporter-78zvw

Всі компоненти мають статус "Running", що свідчить про успішне розгортання.

### 3.2.2 Налаштування моніторингу компонентів

Для забезпечення надійної роботи всіх компонентів системи моніторингу було реалізовано механізми перевірки стану та автоматичного відновлення за допомогою вбудованих у Kubernetes probe-механізмів:

1. **Readiness Probe** — перевіряє готовність сервісу обробляти запити
2. **Liveness Probe** — перевіряє працездатність сервісу

Приклад конфігурації probe для Prometheus сервера:

livenessProbe:

httpGet:

path: /-/healthy

port: 9090

initialDelaySeconds: 30

timeoutSeconds: 30

readinessProbe:

httpGet:

path: /-/ready

port: 9090

initialDelaySeconds: 30

timeoutSeconds: 30

Для Elasticsearch було налаштовано подібні перевірки з урахуванням особливостей роботи цього компонента:

livenessProbe:

tcpSocket:

port: 9200

initialDelaySeconds: 60

periodSeconds: 10

failureThreshold: 5

readinessProbe:

httpGet:

path: /\_cluster/health

port: 9200

initialDelaySeconds: 60

periodSeconds: 10

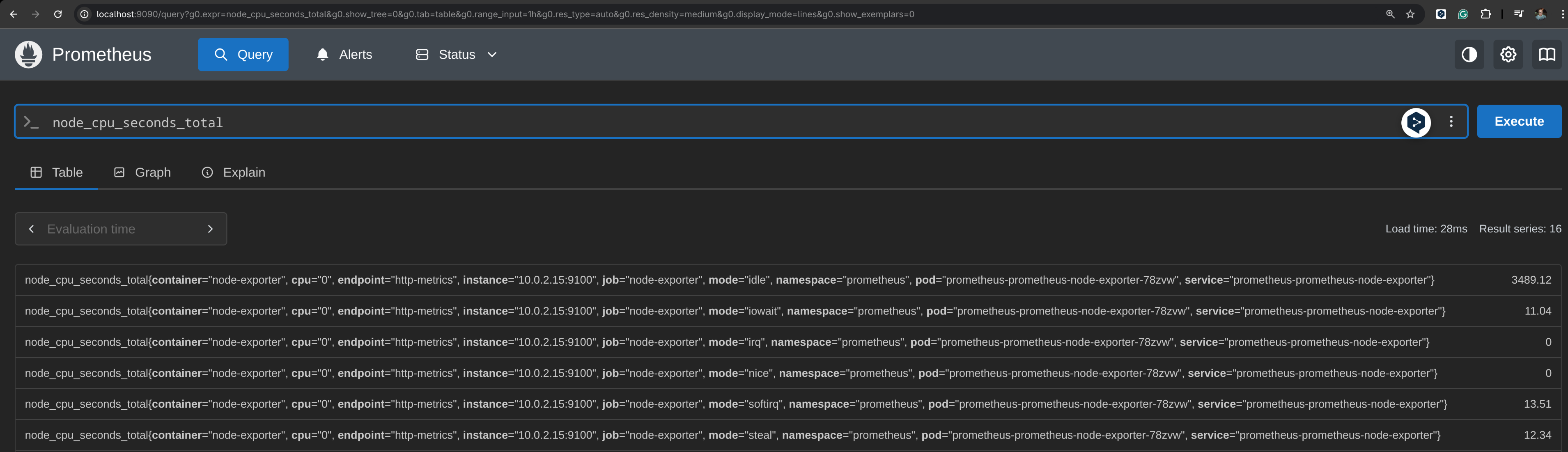
## 3.3 Тестування функціональності системи

### 3.3.1 Тестування збору метрик

Для перевірки коректності збору метрик було проведено серію тестів. Спочатку було перевірено доступність Prometheus через порт-форвардинг:

kubectl port-forward -n prometheus svc/prometheus-kube-prometheus-prometheus 9090:9090

Після встановлення порт-форвардингу було відкрито веб-інтерфейс Prometheus ([http://localhost:9090](http://localhost:9090/)) та перевірено наявність базових метрик.

Рис. 2 Домашня сторінка Prometheus

Для тестування повноти збору даних виконано запит до API Prometheus:

curl -s "http://localhost:9090/api/v1/query?query=up" | jq

{

"status": "success",

"data": {

"resultType": "vector",

"result": [

{

"metric": {

"\_\_name\_\_": "up",

"endpoint": "https-metrics",

"instance": "10.0.2.15:10250",

"job": "kubelet",

"metrics\_path": "/metrics/cadvisor",

..................

Результати запиту підтвердили, що всі таргети успішно скрейпяться (метрика "up" має значення 1 для всіх компонентів).

Для перевірки працездатності node-exporter було виконано запит на отримання метрик навантаження системи:

curl -s "http://localhost:9090/api/v1/query?query=node\_load1" | jq

{

"status": "success",

"data": {

"resultType": "vector",

"result": [

{

"metric": {

"\_\_name\_\_": "node\_load1",

"container": "node-exporter",

"endpoint": "http-metrics",

"instance": "10.0.2.15:9100",

"job": "node-exporter",

"namespace": "prometheus",

"pod": "prometheus-prometheus-node-exporter-78zvw",

"service": "prometheus-prometheus-node-exporter"

},

"value": [

1747773720.966,

"0.48"

]

}

]

}

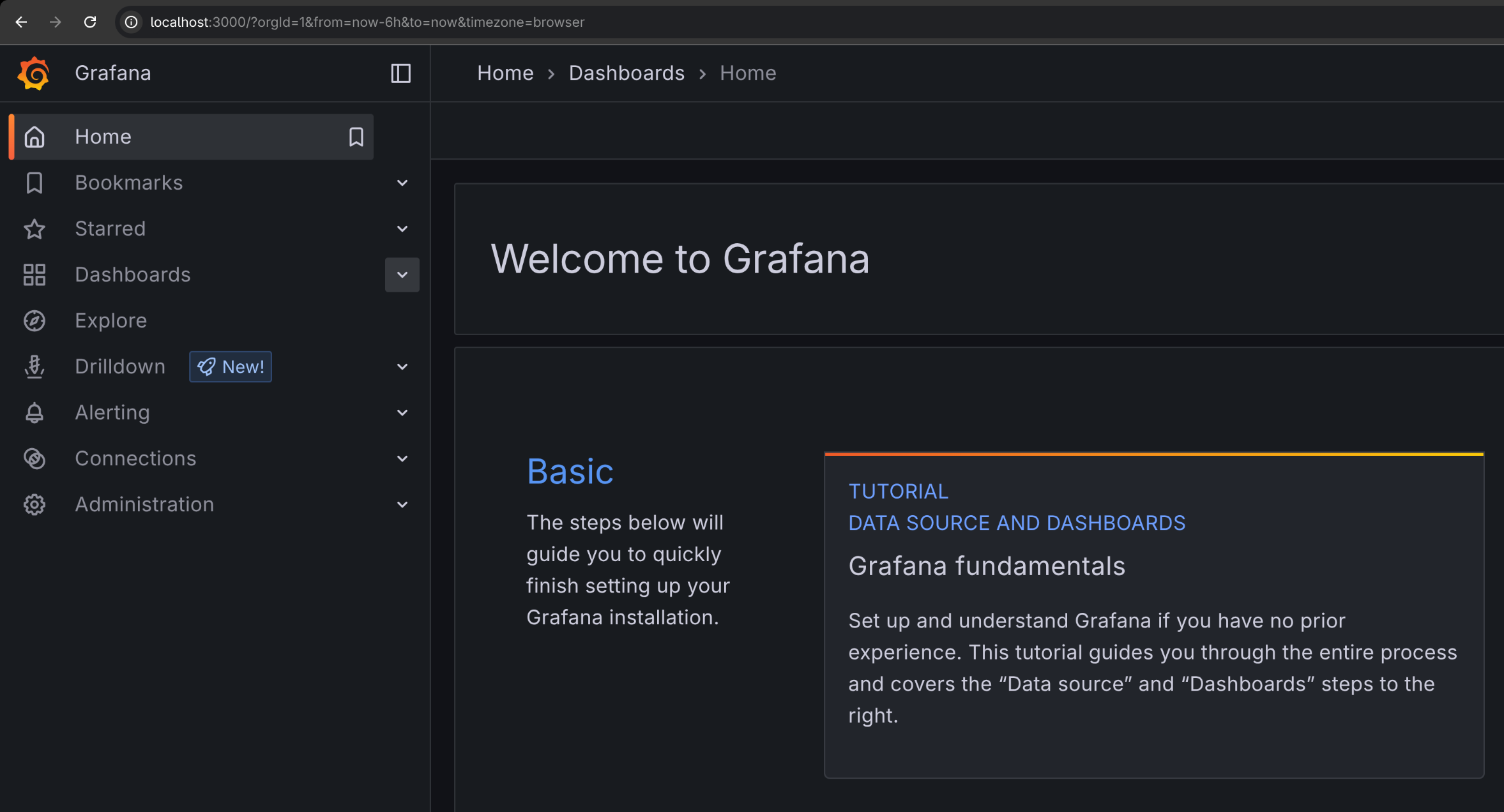
Отримано коректні значення, що підтверджує правильне функціонування збору метрик хоста.

3.3.2 Тестування візуалізації даних

Для перевірки коректності роботи Grafana було налаштовано порт-форвардинг:

kubectl port-forward -n prometheus svc/prometheus-grafana 3000:80

Після входу в веб-інтерфейс Grafana (http://localhost:3000) з використанням стандартних облікових даних було перевірено наявність і функціональність дашбордів.

Рис. 3 Домашня сторінка Grafana

В системі успішно створено та протестовано дашборди:

"Kubernetes Cluster Overview" — для моніторингу загального стану кластера

"Node Exporter" — для детального моніторингу стану вузлів

"Kubernetes Pods" — для моніторингу стану подів

Для тестування алертів було тимчасово зупинено один з компонентів системи:

kubectl scale deployment prometheus-kube-state-metrics -n prometheus --replicas=0

Система моніторингу успішно виявила проблему, і в інтерфейсі Alertmanager з'явилось відповідне повідомлення.

### 3.3.3 Тестування збору та аналізу логів

Для тестування підсистеми збору та аналізу логів було розгорнуто тестовий додаток, що генерує логи різних рівнів:

kubectl apply -f test-app.yaml

Після перевірки доступності Kibana через порт-форвардинг:

kubectl port-forward -n logging svc/kibana 5601:5601

Було виконано пошук логів тестового додатку в Kibana. Результати пошуку підтвердили, що логи успішно збираються, індексуються та доступні для пошуку та аналізу.

### 3.3.4 Тестування масштабованості

Для перевірки здатності системи масштабуватися було проведено тест зі збільшенням кількості реплік одного з компонентів:

kubectl scale deployment prometheus-kube-state-metrics -n prometheus --replicas=3

Система успішно створила додаткові репліки, а Prometheus автоматично виявив нові екземпляри та почав збирати з них метрики.

Також було протестовано горизонтальне масштабування Elasticsearch шляхом збільшення кількості вузлів даних:

kubectl apply -f elasticsearch-scale.yaml

Elasticsearch успішно здійснив шардинг і ребаланс даних між вузлами, що підтвердило коректність роботи механізмів масштабування.

## 3.4 Налагодження виявлених проблем

У процесі тестування було виявлено та вирішено кілька проблем:

1. **Проблема з виділенням ресурсів**: На початковому етапі тестування Elasticsearch-контейнери часто завершувались з помилкою OOMKilled через недостатню кількість виділеної пам'яті. Проблему вирішено шляхом коригування запитів на ресурси:

resources:

requests:

memory: "1Gi"

cpu: "0.5"

limits:

memory: "2Gi"

cpu: "1"

**Проблема з тривалим запуском Elasticsearch**: На етапі ініціалізації кластера Elasticsearch спостерігалась затримка в переході сервісу у стан готовності. Проблему вирішено налаштуванням параметрів readinessProbe:

readinessProbe:

httpGet:

path: /\_cluster/health

port: 9200

initialDelaySeconds: 120 # збільшено з 60

periodSeconds: 15

failureThreshold: 8

## Висновки до пункту 3

У третьому розділі було проведено комплексне тестування та налагодження розробленої системи моніторингу в Kubernetes. Для цього було створено тестове середовище на базі Minikube, що дозволило перевірити функціональність всіх компонентів системи в умовах, близьких до реальних.

Тестування підтвердило, що система відповідає поставленим вимогам: успішно збирає метрики з компонентів Kubernetes-кластера, забезпечує центральне зберігання та індексацію логів, надає інструменти для візуалізації та аналізу даних та реагує на виникнення проблем через систему оповіщень.

У процесі тестування було виявлено та вирішено низку проблем, пов'язаних з виділенням ресурсів, налаштуванням компонентів та механізмами автоматичного відновлення після збоїв. Внесені корективи підвищили надійність та стабільність роботи системи.

Тести на масштабованість показали, що система здатна ефективно адаптуватися до зростання навантаження через горизонтальне масштабування компонентів. При цьому процеси виявлення нових екземплярів, шардингу та балансування даних відбуваються автоматично, без переривання роботи системи.

# ЛІТЕРАТУРА

1. **Burns, B.** Kubernetes: Up and Running: Dive into the Future of Infrastructure / B. Burns, J. Beda, K. Hightower. – 2nd ed. – O'Reilly Media, 2019. – 318 p.
2. **Lukša, M.** Kubernetes in Action / M. Lukša. – Manning Publications, 2018. – 624 p.
3. **Hausenblas, M.** Programming Kubernetes: Developing Cloud-Native Applications / M. Hausenblas, S. Schimanski. – O'Reilly Media, 2019. – 356 p.
4. **Gormley, C.** Elasticsearch: The Definitive Guide / C. Gormley, Z. Tong. – O'Reilly Media, 2015. – 724 p.
5. **Kamps, J.** Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management / J. Kamps, R. Dolin. – Syngress, 2013. – 512 p.
6. **Godard, B.** Prometheus: Up & Running: Infrastructure and Application Performance Monitoring / B. Godard, J. Goodman. – O'Reilly Media, 2018. – 396 p.
7. **Campbell, L.** Monitoring with Prometheus / L. Campbell, B. Brazil. – O'Reilly Media, 2018. – 234 p.

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | ЗАТВЕРДЖУЮ Проректор Українського державного університету науки і технологій \_\_\_\_\_ Анатолій РАДКЕВИЧ 07.05.2025 |  |  |
|  | |  |  |  |
| Розробка застосунку « Проектування та розгортання маштабованоі системи для збору й аналізу телеметрії в Kubernetes за допомогою Elasticsearch і Grafana» | | |  |  |
| Текст програми ЛИСТ ЗАТВЕРДЖЕННЯ  1116130.01305-01 12 01-ЛЗ | | |  |  |
|  | |  |  |  |
|  | | Представники підприємства-розробника Завідувач кафедри КІТ 07.05.25 |  | |
|  | Керівник розробки 07.05.25 | | |  |
|  | Виконавець  07.05.25 | | |  |
|  | Нормконтролер \_\_\_\_\_ Світлана ВОЛКОВА 07.05.25 | | |  |

2025

# ДОДАТОК А – ТЕКСТ ПРОГРАМИ

ЗАТВЕРДЖЕНО

1116130.01305-01 12 01-ЛЗ

Розробка застосунку « Проектування та розгортання маштабованоі системи для збору й аналізу телеметрії в Kubernetes за допомогою Elasticsearch і Grafana»

Текст програми

1116130.01305-01 12 01

Листів 23

2025

# ТЕКСТ ПРОГРАМИ

Backend

*import* os

*import* json

*import* random

*import* time

*import* logging

*from* datetime *import* datetime

*from* flask *import* Flask, jsonify, request

*from* elasticsearch *import* Elasticsearch

*from* threading *import* Thread

*import* uuid

app *=* Flask(\_\_name\_\_)

*# Configure logging*

logging.basicConfig(level*=*logging.INFO)

logger *=* logging.getLogger(\_\_name\_\_)

*# Elasticsearch configuration*

ES\_HOST *=* os.environ.get('ELASTICSEARCH\_HOST', 'elasticsearch')

ES\_PORT *=* os.environ.get('ELASTICSEARCH\_PORT', '9200')

ES\_INDEX *=* os.environ.get('ELASTICSEARCH\_INDEX', 'app-logs')

*# Initialize Elasticsearch client*

es *=* None

*def* *init\_elasticsearch*():

*global* es

*try*:

es *=* Elasticsearch([*f*'http://{ES\_HOST}:{ES\_PORT}'])

*# Wait for Elasticsearch to be ready*

*for* \_ *in* range(30):

*if* es.ping():

logger.info("Connected to Elasticsearch")

*# Create index if it doesn't exist*

*if* *not* es.indices.exists(index*=*ES\_INDEX):

es.indices.create(index*=*ES\_INDEX, body*=*{

"mappings": {

"properties": {

"timestamp": {"type": "date"},

"level": {"type": "keyword"},

"service": {"type": "keyword"},

"message": {"type": "text"},

"user\_id": {"type": "keyword"},

"request\_id": {"type": "keyword"},

"response\_time": {"type": "float"},

"status\_code": {"type": "integer"},

"endpoint": {"type": "keyword"},

"method": {"type": "keyword"},

"ip\_address": {"type": "ip"},

"user\_agent": {"type": "text"},

"error\_type": {"type": "keyword"}

}

}

})

*return* True

time.sleep(2)

logger.error("Failed to connect to Elasticsearch")

*return* False

*except* Exception *as* e:

logger.error(*f*"Error initializing Elasticsearch: {e}")

*return* False

*# Sample data for random log generation*

LOG\_LEVELS *=* ['DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL']

ENDPOINTS *=* ['/api/users', '/api/products', '/api/orders', '/api/auth/login', '/api/auth/logout',

'/api/search', '/api/checkout', '/api/payment', '/api/inventory', '/api/reports']

METHODS *=* ['GET', 'POST', 'PUT', 'DELETE', 'PATCH']

USER\_AGENTS *=* [

'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',

'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36',

'Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36',

'Mozilla/5.0 (iPhone; CPU iPhone OS 14\_6 like Mac OS X) AppleWebKit/605.1.15',

'Mozilla/5.0 (Android 11; Mobile; rv:89.0) Gecko/89.0 Firefox/89.0'

]

ERROR\_MESSAGES *=* [

'Database connection timeout',

'Invalid authentication token',

'Resource not found',

'Permission denied',

'Internal server error',

'Bad request format',

'Service temporarily unavailable',

'Rate limit exceeded',

'Invalid input parameters',

'Payment processing failed'

]

ERROR\_TYPES *=* ['DatabaseError', 'AuthenticationError', 'NotFoundError', 'PermissionError',

'ServerError', 'ValidationError', 'ServiceError', 'RateLimitError']

*# Background log generator*

*def* *generate\_random\_log*():

"""Generate a random log entry"""

level *=* random.choice(LOG\_LEVELS)

*# Higher chance for INFO logs*

*if* random.random() *<* 0.7:

level *=* 'INFO'

log\_entry *=* {

'timestamp': datetime.utcnow().isoformat(),

'level': level,

'service': 'flask-app',

'request\_id': str(uuid.uuid4()),

'user\_id': *f*'user\_{random.randint(1000, 9999)}',

'endpoint': random.choice(ENDPOINTS),

'method': random.choice(METHODS),

'ip\_address': *f*'{random.randint(1, 255)}.{random.randint(1, 255)}.{random.randint(1, 255)}.{random.randint(1, 255)}',

'user\_agent': random.choice(USER\_AGENTS),

'response\_time': round(random.uniform(0.01, 2.0), 3)

}

*# Add status code based on level*

*if* level *in* ['ERROR', 'CRITICAL']:

log\_entry['status\_code'] *=* random.choice([400, 401, 403, 404, 500, 502, 503])

log\_entry['message'] *=* random.choice(ERROR\_MESSAGES)

log\_entry['error\_type'] *=* random.choice(ERROR\_TYPES)

*else*:

log\_entry['status\_code'] *=* random.choice([200, 201, 204, 301, 302])

log\_entry['message'] *=* *f*"Successfully processed {log\_entry['method']} request to {log\_entry['endpoint']}"

*return* log\_entry

*def* *log\_generator\_worker*():

"""Background worker that generates logs continuously"""

*while* True:

*try*:

*if* es *and* es.ping():

*# Generate 1-5 logs every iteration*

*for* \_ *in* range(random.randint(1, 5)):

log\_entry *=* generate\_random\_log()

*# Send to Elasticsearch*

es.index(index*=*ES\_INDEX, body*=*log\_entry)

*# Also log to console*

logger.info(*f*"Generated log: {json.dumps(log\_entry)}")

*# Wait 1-5 seconds before next batch*

time.sleep(random.uniform(1, 5))

*else*:

logger.warning("Elasticsearch not available, retrying...")

time.sleep(5)

init\_elasticsearch()

*except* Exception *as* e:

logger.error(*f*"Error in log generator: {e}")

time.sleep(5)

*# Start background log generator*

log\_thread *=* Thread(target*=*log\_generator\_worker, daemon*=*True)

log\_thread.start()

*# Flask routes*

*@app.route*('/')

*def* *home*():

*return* jsonify({

'service': 'Flask Log Generator',

'status': 'running',

'elasticsearch': es.ping() *if* es *else* False,

'index': ES\_INDEX

})

*@app.route*('/health')

*def* *health*():

es\_status *=* es.ping() *if* es *else* False

*return* jsonify({

'status': 'healthy' *if* es\_status *else* 'unhealthy',

'elasticsearch': es\_status

}), 200 *if* es\_status *else* 503

*@app.route*('/generate', methods*=*['POST'])

*def* *generate\_logs*():

"""Manually trigger log generation"""

*try*:

count *=* request.json.get('count', 10) *if* request.json *else* 10

logs\_generated *=* []

*for* \_ *in* range(min(count, 100)): *# Limit to 100 logs per request*

log\_entry *=* generate\_random\_log()

*if* es *and* es.ping():

es.index(index*=*ES\_INDEX, body*=*log\_entry)

logs\_generated.append(log\_entry)

*return* jsonify({

'status': 'success',

'logs\_generated': len(logs\_generated),

'sample': logs\_generated[:5] *# Return first 5 as sample*

})

*except* Exception *as* e:

*return* jsonify({'status': 'error', 'message': str(e)}), 500

*@app.route*('/stats')

*def* *stats*():

"""Get statistics about generated logs"""

*try*:

*if* es *and* es.ping():

*# Get total count*

count *=* es.count(index*=*ES\_INDEX)['count']

*# Get log level distribution*

aggs *=* es.search(index*=*ES\_INDEX, body*=*{

"size": 0,

"aggs": {

"levels": {

"terms": {"field": "level"}

},

"status\_codes": {

"terms": {"field": "status\_code"}

}

}

})

*return* jsonify({

'total\_logs': count,

'log\_levels': aggs['aggregations']['levels']['buckets'],

'status\_codes': aggs['aggregations']['status\_codes']['buckets']

})

*else*:

*return* jsonify({'error': 'Elasticsearch not available'}), 503

*except* Exception *as* e:

*return* jsonify({'error': str(e)}), 500

*if* \_\_name\_\_ *==* '\_\_main\_\_':

*# Initialize Elasticsearch connection*

*if* init\_elasticsearch():

logger.info("Starting Flask application...")

app.run(host*=*'0.0.0.0', port*=*5000, debug*=*False)

*else*:

logger.error("Failed to initialize Elasticsearch connection")

exit(1)

K8s manifests

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: flask-log-generator

namespace: default

spec:

replicas: 1

selector:

matchLabels:

app: flask-log-generator

template:

metadata:

labels:

app: flask-log-generator

spec:

containers:

- name: flask-app

image: flask-log-generator:latest

imagePullPolicy: Never *# For local Minikube images*

env:

- name: ELASTICSEARCH\_HOST

value: "elasticsearch"

- name: ELASTICSEARCH\_PORT

value: "9200"

- name: ELASTICSEARCH\_INDEX

value: "app-logs"

ports:

- containerPort: 5000

name: http

resources:

limits:

memory: 256Mi

cpu: 200m

requests:

memory: 128Mi

cpu: 100m

readinessProbe:

httpGet:

path: /health

port: 5000

initialDelaySeconds: 10

periodSeconds: 5

livenessProbe:

httpGet:

path: /health

port: 5000

initialDelaySeconds: 30

periodSeconds: 10

---

apiVersion: v1

kind: Service

metadata:

name: flask-log-generator

namespace: default

spec:

selector:

app: flask-log-generator

ports:

- name: http

port: 5000

targetPort: 5000

type: NodePort

requirements.txt  
Flask*==*3.0.0

elasticsearch*==*7.17.9

requests*==*2.31.0

python-json-logger*==*2.0.7

Dockerfile  
*FROM* python:3.11-slim

*WORKDIR* /app

*# Install dependencies*

*COPY* requirements.txt .

*RUN* pip install --no-cache-dir -r requirements.txt

*# Copy application code*

*COPY* app.py .

*# Expose port*

*EXPOSE* 5000

*# Run the application*

*CMD* ["python", "app.py"]

K8s manifests for Prometheus

apiVersion: v1

items:

- apiVersion: v1

kind: Pod

metadata:

annotations:

kubectl.kubernetes.io/default-container: alertmanager

creationTimestamp: "2025-05-28T20:10:07Z"

generateName: alertmanager-prometheus-kube-prometheus-alertmanager-

labels:

alertmanager: prometheus-kube-prometheus-alertmanager

app.kubernetes.io/instance: prometheus-kube-prometheus-alertmanager

app.kubernetes.io/managed-by: prometheus-operator

app.kubernetes.io/name: alertmanager

app.kubernetes.io/version: 0.28.1

apps.kubernetes.io/pod-index: "0"

controller-revision-hash: alertmanager-prometheus-kube-prometheus-alertmanager-6dbc8b78c4

statefulset.kubernetes.io/pod-name: alertmanager-prometheus-kube-prometheus-alertmanager-0

name: alertmanager-prometheus-kube-prometheus-alertmanager-0

namespace: prometheus

ownerReferences:

- apiVersion: apps/v1

blockOwnerDeletion: true

controller: true

kind: StatefulSet

name: alertmanager-prometheus-kube-prometheus-alertmanager

uid: a8eb1353-7261-4082-8262-852d368dd9b5

resourceVersion: "17171"

uid: 8ecd71c2-3ccf-4b53-b144-cd7a773f9105

spec:

affinity:

podAntiAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- podAffinityTerm:

labelSelector:

matchExpressions:

- key: app.kubernetes.io/name

operator: In

values:

- alertmanager

- key: alertmanager

operator: In

values:

- prometheus-kube-prometheus-alertmanager

topologyKey: kubernetes.io/hostname

weight: 100

automountServiceAccountToken: true

containers:

- args:

- --config.file=/etc/alertmanager/config\_out/alertmanager.env.yaml

- --storage.path=/alertmanager

- --data.retention=120h

- --cluster.listen-address=

- --web.listen-address=:9093

- --web.external-url=http://prometheus-kube-prometheus-alertmanager.prometheus:9093

- --web.route-prefix=/

- --cluster.label=prometheus/prometheus-kube-prometheus-alertmanager

- --cluster.peer=alertmanager-prometheus-kube-prometheus-alertmanager-0.alertmanager-operated:9094

- --cluster.reconnect-timeout=5m

- --web.config.file=/etc/alertmanager/web\_config/web-config.yaml

env:

- name: POD\_IP

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: status.podIP

image: quay.io/prometheus/alertmanager:v0.28.1

imagePullPolicy: IfNotPresent

livenessProbe:

failureThreshold: 10

httpGet:

path: /-/healthy

port: http-web

scheme: HTTP

periodSeconds: 10

successThreshold: 1

timeoutSeconds: 3

name: alertmanager

ports:

- containerPort: 9093

name: http-web

protocol: TCP

- containerPort: 9094

name: mesh-tcp

protocol: TCP

- containerPort: 9094

name: mesh-udp

protocol: UDP

readinessProbe:

failureThreshold: 10

httpGet:

path: /-/ready

port: http-web

scheme: HTTP

initialDelaySeconds: 3

periodSeconds: 5

successThreshold: 1

timeoutSeconds: 3

resources:

requests:

memory: 200Mi

securityContext:

allowPrivilegeEscalation: false

capabilities:

drop:

- ALL

readOnlyRootFilesystem: true

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: FallbackToLogsOnError

volumeMounts:

- mountPath: /etc/alertmanager/config

name: config-volume

- mountPath: /etc/alertmanager/config\_out

name: config-out

readOnly: true

- mountPath: /etc/alertmanager/certs

name: tls-assets

readOnly: true

- mountPath: /alertmanager

name: alertmanager-prometheus-kube-prometheus-alertmanager-db

- mountPath: /etc/alertmanager/web\_config/web-config.yaml

name: web-config

readOnly: true

subPath: web-config.yaml

- mountPath: /etc/alertmanager/cluster\_tls\_config/cluster-tls-config.yaml

name: cluster-tls-config

readOnly: true

subPath: cluster-tls-config.yaml

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-wrmrd

readOnly: true

- args:

- --listen-address=:8080

- --web-config-file=/etc/alertmanager/web\_config/web-config.yaml

- --reload-url=http://127.0.0.1:9093/-/reload

- --config-file=/etc/alertmanager/config/alertmanager.yaml.gz

- --config-envsubst-file=/etc/alertmanager/config\_out/alertmanager.env.yaml

- --watched-dir=/etc/alertmanager/config

command:

- /bin/prometheus-config-reloader

env:

- name: POD\_NAME

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: metadata.name

- name: SHARD

value: "-1"

image: quay.io/prometheus-operator/prometheus-config-reloader:v0.82.2

imagePullPolicy: IfNotPresent

name: config-reloader

ports:

- containerPort: 8080

name: reloader-web

protocol: TCP

resources: {}

securityContext:

allowPrivilegeEscalation: false

capabilities:

drop:

- ALL

readOnlyRootFilesystem: true

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: FallbackToLogsOnError

volumeMounts:

- mountPath: /etc/alertmanager/config

name: config-volume

readOnly: true

- mountPath: /etc/alertmanager/config\_out

name: config-out

- mountPath: /etc/alertmanager/web\_config/web-config.yaml

name: web-config

readOnly: true

subPath: web-config.yaml

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-wrmrd

readOnly: true

dnsPolicy: ClusterFirst

enableServiceLinks: true

hostname: alertmanager-prometheus-kube-prometheus-alertmanager-0

initContainers:

- args:

- --watch-interval=0

- --listen-address=:8081

- --config-file=/etc/alertmanager/config/alertmanager.yaml.gz

- --config-envsubst-file=/etc/alertmanager/config\_out/alertmanager.env.yaml

- --watched-dir=/etc/alertmanager/config

command:

- /bin/prometheus-config-reloader

env:

- name: POD\_NAME

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: metadata.name

- name: SHARD

value: "-1"

image: quay.io/prometheus-operator/prometheus-config-reloader:v0.82.2

imagePullPolicy: IfNotPresent

name: init-config-reloader

ports:

- containerPort: 8081

name: reloader-web

protocol: TCP

resources: {}

securityContext:

allowPrivilegeEscalation: false

capabilities:

drop:

- ALL

readOnlyRootFilesystem: true

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: FallbackToLogsOnError

volumeMounts:

- mountPath: /etc/alertmanager/config

name: config-volume

readOnly: true

- mountPath: /etc/alertmanager/config\_out

name: config-out

- mountPath: /etc/alertmanager/web\_config/web-config.yaml

name: web-config

readOnly: true

subPath: web-config.yaml

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-wrmrd

readOnly: true

nodeName: minikube

preemptionPolicy: PreemptLowerPriority

priority: 0

restartPolicy: Always

schedulerName: default-scheduler

securityContext:

fsGroup: 2000

runAsGroup: 2000

runAsNonRoot: true

runAsUser: 1000

seccompProfile:

type: RuntimeDefault

serviceAccount: prometheus-kube-prometheus-alertmanager

serviceAccountName: prometheus-kube-prometheus-alertmanager

subdomain: alertmanager-operated

terminationGracePeriodSeconds: 120

tolerations:

- effect: NoExecute

key: node.kubernetes.io/not-ready

operator: Exists

tolerationSeconds: 300

- effect: NoExecute

key: node.kubernetes.io/unreachable

operator: Exists

tolerationSeconds: 300

volumes:

- name: config-volume

secret:

defaultMode: 420

secretName: alertmanager-prometheus-kube-prometheus-alertmanager-generated

- name: tls-assets

projected:

defaultMode: 420

sources:

- secret:

name: alertmanager-prometheus-kube-prometheus-alertmanager-tls-assets-0

- emptyDir:

medium: Memory

name: config-out

- name: web-config

secret:

defaultMode: 420

secretName: alertmanager-prometheus-kube-prometheus-alertmanager-web-config

- name: cluster-tls-config

secret:

defaultMode: 420

secretName: alertmanager-prometheus-kube-prometheus-alertmanager-cluster-tls-config

- emptyDir: {}

name: alertmanager-prometheus-kube-prometheus-alertmanager-db

- name: kube-api-access-wrmrd

projected:

defaultMode: 420

sources:

- serviceAccountToken:

expirationSeconds: 3607

path: token

- configMap:

items:

- key: ca.crt

path: ca.crt

name: kube-root-ca.crt

- downwardAPI:

items:

- fieldRef:

apiVersion: v1

fieldPath: metadata.namespace

path: namespace

status:

conditions:

- lastProbeTime: null

lastTransitionTime: "2025-06-03T20:01:11Z"

status: "True"

type: PodReadyToStartContainers

- lastProbeTime: null

lastTransitionTime: "2025-05-28T20:10:20Z"

status: "True"

type: Initialized

- lastProbeTime: null

lastTransitionTime: "2025-06-03T20:01:18Z"

status: "True"

type: Ready

- lastProbeTime: null

lastTransitionTime: "2025-06-03T20:01:18Z"

status: "True"

type: ContainersReady

- lastProbeTime: null

lastTransitionTime: "2025-05-28T20:10:07Z"

status: "True"

type: PodScheduled

containerStatuses:

- containerID: docker://54af44c7a0d8c52d364b28d6d95ecff2d492e4a5230d4d7116c4ddf0385ae36e

image: quay.io/prometheus/alertmanager:v0.28.1

imageID: docker-pullable://quay.io/prometheus/alertmanager@sha256:27c475db5fb156cab31d5c18a4251ac7ed567746a2483ff264516437a39b15ba

lastState:

terminated:

containerID: docker://6606d045895af8c770559a5c79ef54a6d6410d1ae500d9f494c16684fbac4f5e

exitCode: 0

finishedAt: "2025-06-01T19:31:24Z"

reason: Completed

startedAt: "2025-06-01T15:11:02Z"

name: alertmanager

ready: true

restartCount: 2

started: true

state:

running:

startedAt: "2025-06-03T20:01:11Z"

volumeMounts:

- mountPath: /etc/alertmanager/config

name: config-volume

- mountPath: /etc/alertmanager/config\_out

name: config-out

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /etc/alertmanager/certs

name: tls-assets

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /alertmanager

name: alertmanager-prometheus-kube-prometheus-alertmanager-db

- mountPath: /etc/alertmanager/web\_config/web-config.yaml

name: web-config

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /etc/alertmanager/cluster\_tls\_config/cluster-tls-config.yaml

name: cluster-tls-config

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-wrmrd

readOnly: true

recursiveReadOnly: Disabled

- containerID: docker://dcb346af3243c27ed0de5b37ae74fb58b6c0e0f3eff783105dd88c66fe4164a7

image: quay.io/prometheus-operator/prometheus-config-reloader:v0.82.2

imageID: docker-pullable://quay.io/prometheus-operator/prometheus-config-reloader@sha256:710458fdccd42d56bb5b20b548fe9a2f832c12aa26256e80cda48a9edb0454ff

lastState:

terminated:

containerID: docker://c1a4a25359ae75b90473d54fc1b06bfcd0dd73997c91059ec94ad573786aa41f

exitCode: 0

finishedAt: "2025-06-01T19:31:24Z"

reason: Completed

startedAt: "2025-06-01T15:11:02Z"

name: config-reloader

ready: true

restartCount: 2

started: true

state:

running:

startedAt: "2025-06-03T20:01:11Z"

volumeMounts:

- mountPath: /etc/alertmanager/config

name: config-volume

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /etc/alertmanager/config\_out

name: config-out

- mountPath: /etc/alertmanager/web\_config/web-config.yaml

name: web-config

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-wrmrd

readOnly: true

recursiveReadOnly: Disabled

hostIP: 192.168.49.2

hostIPs:

- ip: 192.168.49.2

initContainerStatuses:

- containerID: docker://a962ad4cba0cdce1878d2cb00bea6e79e8989091b1899822d96e44242312d5ee

image: quay.io/prometheus-operator/prometheus-config-reloader:v0.82.2

imageID: docker-pullable://quay.io/prometheus-operator/prometheus-config-reloader@sha256:710458fdccd42d56bb5b20b548fe9a2f832c12aa26256e80cda48a9edb0454ff

lastState: {}

name: init-config-reloader

ready: true

restartCount: 0

started: false

state:

terminated:

containerID: docker://a962ad4cba0cdce1878d2cb00bea6e79e8989091b1899822d96e44242312d5ee

exitCode: 0

finishedAt: "2025-06-03T20:01:11Z"

reason: Completed

startedAt: "2025-06-03T20:01:11Z"

volumeMounts:

- mountPath: /etc/alertmanager/config

name: config-volume

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /etc/alertmanager/config\_out

name: config-out

- mountPath: /etc/alertmanager/web\_config/web-config.yaml

name: web-config

readOnly: true

recursiveReadOnly: Disabled

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-wrmrd

readOnly: true

recursiveReadOnly: Disabled

phase: Running

podIP: 10.244.0.47

podIPs:

- ip: 10.244.0.47

qosClass: Burstable

startTime: "2025-05-28T20:10:07Z"

- apiVersion: v1

kind: Pod

metadata:

annotations:

checksum/config: 0e9cbd0ea8e24e32f7dfca5bab17a2ba05652642f0a09a4882833ae88e4cc4a3

checksum/sc-dashboard-provider-config: e70bf6a851099d385178a76de9757bb0bef8299da6d8443602590e44f05fdf24

checksum/secret: 032056e9c62bbe9d1daa41ee49cd3d9524c076f51ca4c65adadf4ef08ef28712

kubectl.kubernetes.io/default-container: grafana

creationTimestamp: "2025-05-28T20:09:53Z"

generateName: prometheus-grafana-77bcfb9bdb-

labels:

app.kubernetes.io/instance: prometheus

app.kubernetes.io/name: grafana

app.kubernetes.io/version: 12.0.0

helm.sh/chart: grafana-9.0.0

pod-template-hash: 77bcfb9bdb

name: prometheus-grafana-77bcfb9bdb-w7gnp

namespace: prometheus

ownerReferences:

- apiVersion: apps/v1

blockOwnerDeletion: true

controller: true

kind: ReplicaSet

name: prometheus-grafana-77bcfb9bdb

uid: 4faa5a7e-907b-4e8e-aa1d-d3fe347da039

resourceVersion: "17001"

uid: 5884f0b5-a10e-452c-8bb9-ab82eeacff3b

spec:

automountServiceAccountToken: true

containers:

- env:

- name: METHOD

value: WATCH

- name: LABEL

value: grafana\_dashboard

- name: LABEL\_VALUE

value: "1"

- name: FOLDER

value: /tmp/dashboards

- name: RESOURCE

value: both

- name: NAMESPACE

value: ALL

- name: REQ\_USERNAME

valueFrom:

secretKeyRef:

key: admin-user

name: prometheus-grafana

- name: REQ\_PASSWORD

valueFrom:

secretKeyRef:

key: admin-password

name: prometheus-grafana

- name: REQ\_URL

value: http://localhost:3000/api/admin/provisioning/dashboards/reload

- name: REQ\_METHOD

value: POST

image: quay.io/kiwigrid/k8s-sidecar:1.30.0

imagePullPolicy: IfNotPresent

name: grafana-sc-dashboard

resources: {}

securityContext:

allowPrivilegeEscalation: false

capabilities:

drop:

- ALL

seccompProfile:

type: RuntimeDefault

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

volumeMounts:

- mountPath: /tmp/dashboards

name: sc-dashboard-volume

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-9hkjk

readOnly: true

- env:

- name: METHOD

value: WATCH

- name: LABEL

value: grafana\_datasource

- name: LABEL\_VALUE

value: "1"

- name: FOLDER

value: /etc/grafana/provisioning/datasources

- name: RESOURCE

value: both

- name: REQ\_USERNAME

valueFrom:

secretKeyRef:

key: admin-user

name: prometheus-grafana

- name: REQ\_PASSWORD

valueFrom:

secretKeyRef:

key: admin-password

name: prometheus-grafana

- name: REQ\_URL

value: http://localhost:3000/api/admin/provisioning/datasources/reload

- name: REQ\_METHOD

value: POST

image: quay.io/kiwigrid/k8s-sidecar:1.30.0

imagePullPolicy: IfNotPresent

name: grafana-sc-datasources

resources: {}

securityContext:

allowPrivilegeEscalation: false

capabilities:

drop:

- ALL

seccompProfile:

type: RuntimeDefault

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

volumeMounts:

- mountPath: /etc/grafana/provisioning/datasources

name: sc-datasources-volume

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-9hkjk

readOnly: true

- env:

- name: POD\_IP

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: status.podIP

- name: GF\_SECURITY\_ADMIN\_USER

valueFrom:

secretKeyRef:

key: admin-user

name: prometheus-grafana

- name: GF\_SECURITY\_ADMIN\_PASSWORD

valueFrom:

secretKeyRef:

key: admin-password

name: prometheus-grafana

- name: GF\_PATHS\_DATA

value: /var/lib/grafana/

- name: GF\_PATHS\_LOGS

value: /var/log/grafana

- name: GF\_PATHS\_PLUGINS

value: /var/lib/grafana/plugins

- name: GF\_PATHS\_PROVISIONING

value: /etc/grafana/provisioning

image: docker.io/grafana/grafana:12.0.0

imagePullPolicy: IfNotPresent

livenessProbe:

failureThreshold: 10

httpGet:

path: /api/health

port: 3000

scheme: HTTP

initialDelaySeconds: 60

periodSeconds: 10

successThreshold: 1

timeoutSeconds: 30

name: grafana

ports:

- containerPort: 3000

name: grafana

protocol: TCP

- containerPort: 9094

name: gossip-tcp

protocol: TCP

- containerPort: 9094

name: gossip-udp

protocol: UDP

- containerPort: 6060

name: profiling

protocol: TCP

readinessProbe:

failureThreshold: 3

httpGet:

path: /api/health

port: 3000

scheme: HTTP

periodSeconds: 10

successThreshold: 1

timeoutSeconds: 1

resources: {}

securityContext:

allowPrivilegeEscalation: false

capabilities:

drop:

- ALL

seccompProfile:

type: RuntimeDefault

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

volumeMounts:

- mountPath: /etc/grafana/grafana.ini

name: config

subPath: grafana.ini

- mountPath: /var/lib/grafana

name: storage

- mountPath: /tmp/dashboards

name: sc-dashboard-volume

- mountPath: /etc/grafana/provisioning/dashboards/sc-dashboardproviders.yaml

name: sc-dashboard-provider

subPath: provider.yaml

- mountPath: /etc/grafana/provisioning/datasources

name: sc-datasources-volume

- mountPath: /var/run/secrets/kubernetes.io/serviceaccount

name: kube-api-access-9hkjk

readOnly: true

dnsPolicy: ClusterFirst

enableServiceLinks: true

nodeName: minikube

preemptionPolicy: PreemptLowerPriority

priority: 0

restartPolicy: Always

schedulerName: default-scheduler

securityContext:

fsGroup: 472

runAsGroup: 472

runAsNonRoot: true

runAsUser: 472

serviceAccount: prometheus-grafana

serviceAccountName: prometheus-grafana

shareProcessNamespace: false

terminationGracePeriodSeconds: 30

tolerations:

- effect: NoExecute

key: node.kubernetes.io/not-ready

operator: Exists

tolerationSeconds: 300

- effect: NoExecute

key: node.kubernetes.io/unreachable

operator: Exists

tolerationSeconds: 300