Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

**МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА**

на тему:

# СИСТЕМА АВТОМАТИЗАЦІЇ МОНІТОРИНГУ НА ПЛАТФОРМІ KUBERNETES

Виконав студент 2 курсу, групи 2КІ-22м
спеціальності 123 — Комп'ютерна інженерія
Кулібабчук І. П.
Керівник к.т.н., проф. каф. ОТ
Захарченко С. М.
" 7 " грудня 2023р.
Опонент к.т.н., доц. каф. ПЗ
Ткаченко О. М.
" 5 " грудня 2023р.

**Допущено до захисту**
д.т.н., проф. Азаров О.Д.
"08" 11 2023 р.

ВНТУ 2023

**ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ**

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Галузь знань — Інформаційні технології

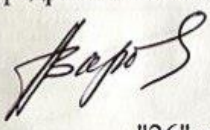
Освітній рівень — магістр

Спеціальність — 123 Комп'ютерна інженерія

Освітньо-професійна програма — Комп'ютерна інженерія

**ЗАТВЕРДЖУЮ**

Завідувач кафедри обчислювальної техніки



О.Д. Азаров

"26" вересня 2023 р.

**З А В Д А Н Н Я**

**НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ**

студенту **Кулібабчуку Івану Павловичу**

1 Тема роботи «Система автоматизації моніторингу на платформі Kubernetes» керівник роботи Захарченко Сергій Михайлович професор, затверджено наказом вищого навчального закладу від 18.09.2023 року № 247

2 Строк подання студентом роботи 09.12.2023 р.

3 Вихідні дані до роботи: 2 віртуальних машини, Microk8s на одній, Jenkins та Ansible на іншій, моніторинг платформи Kubernetes.

4 Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): вступ, аналіз методів проектування моніторингу, дослідження методів автоматизації, розробка системи автоматизації моніторингу на платформі Kubernetes, економічна частина.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): графік масштабування ресурсів за розподілом Гаусса, графік масштабуваня ресурсів за сумуванням рядів формули Рімана, схема архітектури розгортання системи, графік ефективності Kubernetes кластеру, графік подій у Kubernetes кластері.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1— Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| 1-3 | Захарченко Сергій Михайлович проф. | 1.09.2023 | 1.09.2023 |
| 4 | Небава Микола Іванович проф., к.е.н | 1.11.2023 | 1.11.2023 |

7 Дата видачі завдання 19.09.2023р.

8 Календарний план виконання МКР приведений в таблиці 2.

Таблиця 2 — Календарний план

| № з/п | Назва етапів МКР | Строк виконання | Підпис |
|---|---|---|---|
| 1 | Постановка задачі | 10.09.2023 | |
| 2 | Огляд існуючих рішень | 25.09.2023 | |
| 3 | Дослідженя архітектури Kubernetes та моніторингу її ключових компонентів | 5.10.2023 | |
| 4 | Розробка автоматизації доставки та розгортання моніторингу в Kubernetes | 12.10.2023 | |
| 5 | Розрахунок економічної частини | 19.10.2023 | |

Продовження таблиці 2

| | | | |
|---|---|---|---|
| 6 | Оформлення пояснювальної записки | 26.10.2023 | |
| 7 | Виконання магістерської кваліфікаційної роботи | 2.11.2023 | |
| 8 | Перевірка якості виконання магістерської кваліфікаційної роботи та усунення недоліків | 9.11.2023 | |
| 9 | Підписи супроводжувальних документів у керівника, опонента, нормоконтролера | 16.11.2023 | |
| 10 | Перевірка «Антиплагіат» | 23.11.2023 | |
| 11 | Попередній захист | 30.11.2023 | |

Студент _____ Кулібабчук Іван Павлович

Керівник _____ проф. Захарченко Сергій Михайлович

# АНОТАЦІЯ

УДК 004.75

Кулібабчук І.П. Система автоматизації моніторингу на платформі Kubernetes. Магістерська кваліфікаційна робота зі спеціальності 123 — Комп'ютерна Інженерія, Вінниця: ВНТУ, 2023 — 116 с. На укр. мові. Бібліогр.: 30 назв; рис.: 31; табл. 5.

Розглянуто Kubernetes, як програмне забезпечення для оркестрування контейнеризованих додатків, його компоненти та практики і методи його моніторингу. Створено архітектуру автоматизованого налаштування моніторингу . Розроблено автоматизацію для встановлення моніторингового стеку компонентів.

Ключові слова: Kubernetes, інфраструктура як код, система управління конфігураціями, моніторинговий стек.

# ANNOTATION

УДК 004.75

Kulibabchuk I.P. Monitoring automation system on the Kubernetes platform. Master's thesis on specialty 123 — Computer Engineering, Vinnytsia: VNTU, 2023 — 116 p. In Ukrainian speech Bibliography: 30 titles; Fig.: 31; table 5.

In the general part of the work, Kubernetes - software for orchestrating containerized applications, its components and practices and monitoring methods are considered. In the calculation and design part, the architecture of the automated monitoring setup was created. In the technological part, automation will be developed and a technology monitoring stack will be installed.

Keywords: Kubernetes, infrastructure as code, configuration management system, monitoring stack.

## ЗМІСТ

## ВСТУП

Моніторинг поточного стану програми є одним із найефективніших способів передбачити проблеми, що впливають на клієнтів, і виявити вузькі місця у робочому середовищі. Проте це також одна з найбільших проблем, з якою стикаються майже всі організації, що займаються розробкою програмного забезпечення.

Мікросервіси за своєю суттю взаємозалежні, незначна зміна у виробництві може зламати всю систему. Технологією, яка координує всі ці мікросервіси, є, звичайно, Kubernetes. Незважаючи на те, що Kubernetes значно спрощує розгортання мікросервісу та керування ним, його швидке та постійне прийняття рішень також ускладнює визначення того, що насправді відбувається у виробництві.

У світі моніторингу Kubernetes існує так багато змінних, які потрібно відстежувати, що потрібні нові інструменти, нові методи для ефективного збору важливих даних. Отже, моніторинг є важливим процесом, який потребує автоматизованого встановлення та налаштування.

**Метою** даної роботи є прискорення та автоматичне покриття моніторингом створеної цифрової інфрастуктури на базі Kubernetes.

Для досягнення мети необхідно розв’язати такі задачі:

— проаналізувати архітектуру Kubernetes та її критично важливі компоненти;

— проаналізувати існуючі моніторингові стеки;

— визначити ключові метрики та системні події, які слугують індикаторами проблем;

— запропонувати спосіб автоматизації підняття моніторинг компонентів на Kubernetes кластері;

— запропонувати розв’язок задачі з аналізу агрегованих даних, для очищення шуму в даних та отримання точніших результатів про стан роботи платформи Kubernetes.

**Об'єкт дослідження** — процес автоматизації розгортання кластерної системи моніторингу на платформі Kubernetes.

Предмет дослідження — методи проектування моніторингової системи.

**Новизна роботи** запропоновано метод автоматизації розгортання моніторингу за рахунок використання інструментів оркестрації та збору релізу, що дозволило уникнути фактору людської помилки та пришвидшення налаштування контролю за Kubernetes кластером.

**Практична цінність** розроблено систему автоматизації встановлення моніторинг компонентів на платформу Kubernetes. Розроблено програмне забезпечення для додаткового аналізу агрегованих даних у моніторинговому компоненті.

**Апробацію** результатів наукової роботи було проведено на двох наукових конференціях : «Молодь в науці: дослідження, проблеми, перспективи (МН — 2024)», доповідь на тему “Особливості розгортання інструментів моніторингу на основі кластерних систем”.

# 1 ОГЛЯД АРХІТЕКТУРИ KUBERNETES ТА МОНІТОРИНГУ ЇЇ КЛЮЧОВИХ КОМПОНЕНТІВ

Kubernetes — це платформа з відкритим вихідним кодом для управління контейнеризованими робочими навантаженнями та супутніми службами. Її основні характеристики — кросплатформенність, розширюваність, успішне використання декларативної конфігурації та автоматизації. Вона має гігантську, швидкопрогресуючу екосистему.

Назва Kubernetes походить з грецької та означає керманич або пілот. Google відкрив доступ до вихідного коду проекту Kubernetes у 2014 році. Kubernetes побудовано на базі п'ятнадцятирічного досвіду, що Google отримав, оперуючи масштабними робочими навантаженнями у купі з найкращими у своєму класі ідеями та практиками, які може запропонувати спільнота [1].

У даному розділі буде розглянуто архітектуру та важливі складові Kubernetes кластеру, показники яких вказують на проблеми з продуктивністю середовища, порівняння різних моніторингових стеків.

## 1.1 Аналіз кластерної архітектури

Kubernetes запускає програми, розміщуючи контейнери в поди для запуску на вузлах (Nodes). Залежно від кластера, вузол може бути віртуальною чи фізичною машиною. Кожен вузол містить послуги, необхідні запуску подів, керованих площиною управління. Зазвичай у вас є кілька вузлів у кластері; однак у середовищі навчання або середовищі з обмеженими ресурсами у вас може бути лише один. Компоненти на вузлі включають kubelet, середовище виконання контейнера та kube-proxy [2].

Назва ідентифікує вузол. Два вузли не можуть мати однакові назви одночасно. Kubernetes також припускає, що ресурс із такою ж назвою є тим самим об’єктом. У випадку вузла неявно передбачається, що екземпляр, який використовує те саме ім’я, матиме той самий стан (наприклад, параметри мережі, вміст кореневого диска) та атрибути, такі як мітки вузла. Це може призвести до

неузгодженості, якщо екземпляр було змінено без зміни назви. Якщо Node потрібно замінити або суттєво оновити, існуючий об’єкт Node потрібно спочатку видалити з сервера API та повторно додати після оновлення.

Kubernetes має шаблон API «хаб-і-спиця». Усе використання API з вузлів (або модулів, які вони запускають) завершується на сервері API. Жоден з інших компонентів рівня керування не призначений для надання віддалених служб. Сервер API налаштований на прослуховування віддалених з’єднань на безпечному порту HTTPS (зазвичай 443) з увімкненою однією або кількома формами автентифікації клієнта. Слід увімкнути одну або кілька форм авторизації, особливо якщо дозволені анонімні запити або маркери облікових записів служби [3].

Вузли повинні мати загальнодоступний кореневий сертифікат для кластера, щоб вони могли безпечно підключатися до сервера API разом із дійсними обліковими даними клієнта. Хорошим підходом є те, що облікові дані клієнта, надані kubelet, мають форму сертифіката клієнта. Перегляньте завантаження kubelet TLS для автоматичного надання клієнтських сертифікатів kubelet.

Модулі, які бажають підключитися до сервера API, можуть зробити це безпечно, скориставшись обліковим записом служби, щоб Kubernetes автоматично вставляв загальнодоступний кореневий сертифікат і дійсний маркер носія в пакет під час його створення. Служба kubernetes (у просторі імен за умовчанням) налаштована з віртуальною IP-адресою, яка переспрямовується (через kube-proxy) на кінцеву точку HTTPS на сервері API.

Компоненти площини керування також спілкуються з сервером API через безпечний порт.

Як наслідок, робочий режим за замовчуванням для підключень від вузлів і модулів, що працюють на вузлах, до рівня керування захищений за замовчуванням і може працювати через ненадійні та/або загальнодоступні мережі.

Існує два основних шляхи зв’язку від рівня керування (сервер API) до вузлів. Перший — від сервера API до процесу kubelet, який виконується на кожному вузлі в кластері. Другий — від сервера API до будь-якого вузла, модуля чи служби через проксі-сервер API.

Ці з’єднання завершуються на кінцевій точці HTTPS kubelet. За замовчуванням сервер API не перевіряє сертифікат обслуговування kubelet, через що з’єднання стає предметом атак типу «людина посередині» та стає небезпечним для роботи в ненадійних та/або загальнодоступних мережах.

З’єднання від сервера API до вузла, модуля або служби за замовчуванням використовують звичайні HTTP-з’єднання, тому вони не автентифіковані та не зашифровані. Їх можна запускати через безпечне з’єднання HTTPS, додавши префікс https: до імені вузла, модуля або служби в URL-адресі API, але вони не перевірятимуть сертифікат, наданий кінцевою точкою HTTPS, і не нададуть облікові дані клієнта. Таким чином, хоча з’єднання буде зашифровано, воно не забезпечить жодних гарантій цілісності. Ці підключення наразі небезпечні для використання через ненадійні або загальнодоступні мережі.

Kubernetes підтримує SSH-тунелі для захисту шляхів зв’язку між рівнем керування та вузлами. У цій конфігурації сервер API ініціює SSH-тунель до кожного вузла в кластері (підключається до SSH-сервера, який прослуховує порт 22) і пропускає весь трафік, призначений для kubelet, вузла, модуля або служби, через тунель. Цей тунель гарантує, що трафік не виходить за межі мережі, в якій працюють вузли.

## 1.2 Опис робочого навантаження у Kubernetes

Робоче навантаження — це програма, що працює на Kubernetes. Незалежно від того, чи є ваше робоче навантаження одним компонентом чи кількома, які працюють разом, у Kubernetes запускаєте його в наборі модулів. У Kubernetes Pod представляє набір запущених контейнерів у вашому кластері.

Модулі Kubernetes мають певний життєвий цикл. Наприклад, коли модуль працює у вашому кластері, тоді критична помилка на вузлі, де працює цей пакет, означає, що всі модулі на цьому вузлі виходять з ладу. Kubernetes розглядає цей рівень відмови як остаточний: потрібно буде створити новий Pod для відновлення, навіть якщо вузол пізніше стане справним [4].

Однак, щоб значно полегшити життя, не потрібно безпосередньо керувати кожним модулем. Натомість можете використовувати ресурси робочого навантаження, які керують набором модулів від вашого імені. Ці ресурси налаштовують контролери, які гарантують, що запущено потрібну кількість модулів потрібного типу відповідно до вказаного стану.

Kubernetes надає кілька вбудованих ресурсів робочого навантаження:

Deployment та ReplicaSet (заміна застарілого ресурсу ReplicationController). Deployment добре підходить для керування робочим навантаженням додатків без стану на вашому кластері, де будь-які модулі в розгортанні є взаємозамінними та можуть бути замінені за потреби.

StatefulSet дозволяє запускати один або кілька пов’язаних модулів, які якимось чином відстежують стан. Наприклад, якщо ваше робоче навантаження постійно записує дані, можете запустити StatefulSet, який зіставляє кожен Pod з PersistentVolume. Код, що працює в контейнерах для цього StatefulSet, може копіювати дані на інші контейнери в тому самому наборі StatefulSet, щоб покращити загальну стійкість [5].

DaemonSet визначає модулі, які надають засоби, локальні для вузлів. Кожного разу, коли додаєте вузол до свого кластера, який відповідає специфікації в DaemonSet, рівень керування планує Pod для цього DaemonSet на новому вузлі. Кожен модуль у DaemonSet виконує роботу, подібну до системного демона на класичному сервері Unix / POSIX. DaemonSet може бути основоположним для роботи вашого кластера, як-от плагін для запуску мережі кластера, він може допомогти керувати вузлом або може забезпечувати необов’язкову поведінку, яка покращує платформу контейнера, яку використовуєте.

Job і CronJob пропонують різні способи визначення завдань, які виконуються до завершення, а потім зупиняються. CronJob можна використовувати для запуску одного і того ж завдання кілька разів відповідно до розкладу.

У ширшій екосистемі Kubernetes можна знайти сторонні ресурси робочого навантаження, які забезпечують додаткову поведінку. Використовуючи спеціальне визначення ресурсу, можете додати сторонній ресурс робочого навантаження,

якщо потрібна певна поведінка, яка не є частиною ядра Kubernetes. Наприклад, якщо хочете запустити групу Pods для своєї програми, але припинити роботу, доки всі Pods не будуть доступні (можливо, для якогось високопродуктивного розподіленого завдання), можете запровадити або встановити розширення, яке надає цю функцію.

Поди — це найменші розгорнуті обчислювальні одиниці, які можете створювати та керувати ними в Kubernetes. Pod — це група з одного або кількох контейнерів зі спільним сховищем і мережевими ресурсами, а також специфікацією того, як запускати контейнери. Вміст модуля завжди розміщується разом і за розкладом, а також працює в спільному контексті. Pod моделює специфічний для програми «логічний хост»: він містить один або більше контейнерів програми, які відносно тісно пов’язані. У нехмарних контекстах програми, які виконуються на одній фізичній або віртуальній машині, аналогічні хмарним програмам, які виконуються на тому самому логічному хості. Окрім контейнерів додатків, Pod може містити контейнери ініціалізації, які запускаються під час запуску Pod.

### 1.3 Огляд сервісів, балансування навантаження та мережі

Кожен Pod у кластері отримує власну унікальну IP-адресу для всього кластера. Це означає, що не потрібно явно створювати зв’язки між модулями, і майже ніколи не потрібно мати справу з відображенням портів контейнера на порти хосту.

Це створює чисту, зворотно сумісну модель, де Pods можна розглядати так само, як віртуальні машини або фізичні хости з точки зору виділення портів, іменування, виявлення служб, балансування навантаження, конфігурації програми та міграції [6].

Kubernetes висуває наступні основні вимоги до будь-якої мережевої реалізації (за винятком будь-якої навмисної політики сегментації мережі):

Ця модель не тільки менш складна загалом, але й принципово сумісна з бажанням Kubernetes забезпечити легке перенесення додатків із віртуальних машин у контейнери. Якщо ваше завдання раніше виконувалося у віртуальній

машині, ваша віртуальна машина мала IP і могла спілкуватися з іншими віртуальними машинами у вашому проекті. Це та сама базова модель.

IP-адреси Kubernetes існують в області Pod — контейнери в Pod спільними є простори імен мережі — включно з IP-адресою та MAC-адресою. Це означає, що всі контейнери в Pod можуть досягати портів один одного на локальному хості. Це також означає, що контейнери всередині Pod повинні координувати використання портів, але це не відрізняється від процесів у віртуальній машині. Це називається моделлю «IP-per-pod».

Те, як це реалізовано, є деталями конкретного середовища виконання контейнера, яке використовується [7].

Можна запитувати порти на самому вузлі, які пересилають на ваш Pod (так звані порти хосту), але це дуже спеціальна операція. Те, як це пересилання реалізовано, також є деталлю середовища виконання контейнера. Сам Pod не бачить існування чи неіснування хост-портів.

У Kubernetes служба — це метод для надання доступу до мережевої програми, яка працює як один або кілька модулів у вашому кластері.

Основна мета служб у Kubernetes полягає в тому, що не потрібно змінювати наявну програму для використання незнайомого механізму виявлення служб. Ви можете запускати код у Pods, незалежно від того, чи це код, створений для хмарного світу, чи старіша програма, яку ви контейнеризували. Ви використовуєте службу, щоб зробити цей набір модулів доступним у мережі, щоб клієнти могли з ним взаємодіяти.

Якщо ви використовуєте розгортання для запуску програми, це розгортання може динамічно створювати та знищувати модулі. З одного моменту до наступного ви не знаєте, скільки з цих Pods працюють і здорові; Ви можете навіть не знати, як називаються ці здорові стручки. Kubernetes Pods створюються та знищуються відповідно до бажаного стану вашого кластера. Поди є ефемерними ресурсами (не варто очікувати, що окремий Pod надійний і довговічний).

Кожен Pod отримує власну IP-адресу (Kubernetes очікує, що мережеві плагіни гарантуватимуть це). Для певного розгортання у вашому кластері набір модулів,

запущених в один момент часу, може відрізнятися від набору модулів, що запускають цю програму через мить.

Ingress надає маршрути HTTP та HTTPS поза кластером службам у кластері. Маршрутизація трафіку контролюється правилами, визначеними на ресурсі Ingress. Ось простий приклад, коли Ingress надсилає весь свій трафік до однієї служби (рисунок 1.1).

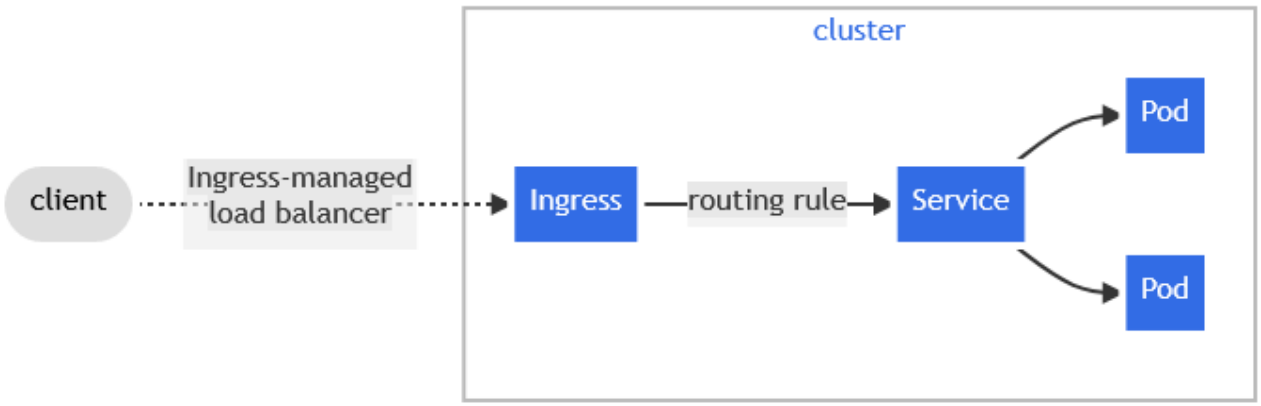



Рисунок 1.1 — Ingress

Ingress може бути налаштований для надання сервісам зовнішніх URL-адрес, балансування трафіку, припинення SSL/TLS і пропонування віртуального хостингу на основі імен. Контролер Ingress відповідає за виконання Ingress, як правило, з балансувальником навантаження, хоча він також може налаштувати ваш межовий маршрутизатор або додаткові інтерфейси для обробки трафіку.

Ingress не відкриває довільні порти чи протоколи. Надання служб, відмінних від HTTP і HTTPS, в Інтернеті зазвичай використовує службу типу Service.Type=NodePort або Service.Type=LoadBalancer.

### 1.4 Аналіз архітектури журналювання та метрик

Журнали програми можуть допомогти вам зрозуміти, що відбувається у вашій програмі. Журнали особливо корисні для налагодження проблем і моніторингу активності кластера. Більшість сучасних програм мають певний

механізм журналювання. Подібним чином механізми контейнерів розроблені для підтримки журналювання. Найпростішим і найпоширенішим методом журналювання для контейнерних програм є запис у стандартний вихід і стандартні потоки помилок.

Однак нативної функціональності, наданої механізмом контейнерів або середовищем виконання, зазвичай недостатньо для повного рішення журналювання [8].

Наприклад, ви можете отримати доступ до журналів програми, якщо контейнер виходить з ладу, модуль вилучається або вузол вмирає.

У кластері журнали повинні мати окреме сховище та життєвий цикл незалежно від вузлів, модулів або контейнерів. Ця концепція називається журналюванням на рівні кластера. Архітектури журналювання на рівні кластера вимагають окремого сервера для зберігання, аналізу та запитів журналів. Kubernetes не надає власного рішення для зберігання даних журналу. Натомість існує багато рішень для журналювання, які інтегруються з Kubernetes. У наступних розділах описано, як обробляти та зберігати журнали на вузлах. Середовище виконання контейнера обробляє та перенаправляє будь-який вихід, згенерований до потоків stdout і stderr контейнерної програми. Різні середовища виконання контейнерів реалізують це по-різному; однак інтеграція з kubelet стандартизована як формат журналювання CRI. За замовчуванням, якщо контейнер перезапускається, kubelet зберігає один завершений контейнер із його журналами. Якщо контейнер вилучається з вузла, усі відповідні контейнери також вилучаються разом із їхніми журналами.

Kubelet робить журнали доступними для клієнтів за допомогою спеціальної функції Kubernetes API. Звичайним способом доступу до цього є запуск журналів kubectl logs.

Хоча Kubernetes не надає рідного рішення для журналювання на рівні кластера, є кілька поширених підходів, які ви можете розглянути.

Реалізувати журналювання на рівні кластера, можна включивши агент журналювання на рівні вузла на кожному вузлі. Агент журналювання — це

спеціальний інструмент, який відкриває журнали або надсилає журнали до серверної частини (рисунок 1.2).

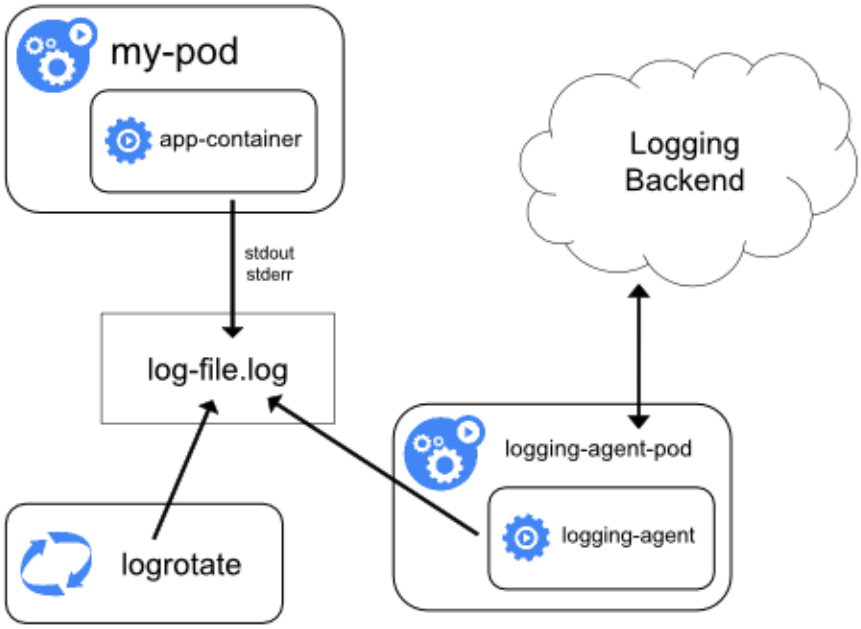



Рисунок 1.2 — Агент журналювання вузла

Зазвичай агент журналювання — це контейнер, який має доступ до каталогу з файлами журналу з усіх контейнерів програми на цьому вузлі. Оскільки агент журналювання має працювати на кожному вузлі, рекомендується запускати агент як DaemonSet. Журналування на рівні вузла створює лише одного агента на вузол і не потребує жодних змін у програмах, запущених на вузлі. Контейнери записують у stdout і stderr, але без погодженого формату. Агент рівня вузла збирає ці журнали та пересилає їх для агрегації.

Якщо контейнери sidecar записують у власні потоки stdout і stderr, можна скористатися перевагами kubelet і агента журналювання, які вже працюють на кожному вузлі. Контейнери коляски читають журнали з файлу, сокета або журналу. Кожен контейнер із колясками друкує журнал у власний потік stdout або stderr (рисунок 1.3).

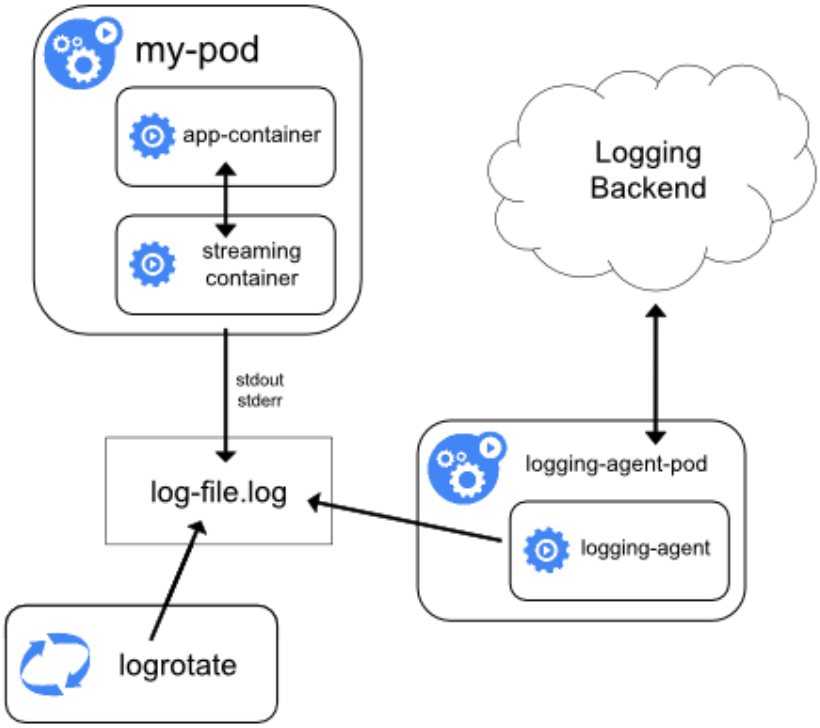



Рисунок 1.3 — Потоковий контейнер з sidecar

Цей підхід дозволяє відокремити кілька потоків журналу з різних частин програми, деякі з яких можуть не підтримувати запис у stdout або stderr. Логіка переспрямування журналів мінімальна, тому це не є значним накладним. Крім того, оскільки stdout і stderr обробляються kubelet, ви можете використовувати такі вбудовані інструменти, як журнали kubectl logs. Під час експлуатації кластера Kubernetes компоненти системи моніторингу мають вирішальне значення для забезпечення працездатності, продуктивності та надійності кластера та його робочих навантажень [9]. Для хоста, який працює майстром, важливо враховувати використання центрального процесора та оперативної пам'яті, кількість подій для виявлення проблем та стан дискового простору.

Для вузлів важливими показниками є використання ресурсів, кількість та стан запущених контейнерів, використання мережевих ресурсів та кількість невдало запущених контейнерів.

При оцінці робочих навантажень слід звертати увагу на використання CPU та пам'яті для конкретного робочого навантаження.

Отже, у наступному підрозділі будуть розглянуті системи, які здатні стежити за даними метриками і надсилати попередження у разі проблем з ними.

## 1.5 Моніторинг стеки та порівняльна характеристика

Сучасні проекти програмного забезпечення базуються на складних технологічних стеках, що складаються з численних компонентів. Правильний вибір та ефективний моніторинг цих компонентів є вирішальним для успішності проекту. У цьому розділі ми розглядаємо методи моніторингу та проводимо порівняльний аналіз популярних технологій [10].

Стеки моніторингу відповідають за аналіз кожного аспекту роботи комп'ютера в певний момент часу — чи це відсоток ресурсів, який використовується конкретними процесами, чи певний процес аварійно завершує роботу та часто перезапускається (що може свідчити про помилку чи помилку), і навіть статистичні дані, наприклад, хто або який процес запустив або завершив роботу протягом певного періоду часу — вони постійно спостерігають за системами, щоб переконатися, що все працює гладко та ефективно [11].

Prometheus- Grafana stack — стек Grafana, Prometheus і Alertmanager є стандартом де-факто для створення сигналів про інфраструктуру.

Prometheus — це інструмент моніторингу з відкритим кодом, який зберігає дані у форматі часових рядів. Він отримує показники від різних експортерів і зберігає їх як часові ряди. Кожен часовий ряд можна унікально ідентифікувати за допомогою комбінації назв показників і міток, які знаходяться в парах ключ-значення.

Мітки допомагають розрізняти часові ряди з однаковою назвою метрики[12].

Наприклад, якщо налаштували Prometheus на збирання метрик експортера вузлів, що працює на двох різних вузлах. Імена метрик, надіслані експортером вузла, будуть однаковими, але відповідні мітки відрізнятимуться, оскільки він працює на різних вузлах (рисунок 1.4).


Рисунок 1.4 — Схема поєднання додатків та Prometheus

Цілі повинні бути визначені у файлі конфігурації Prometheus для збору показників. Кожна ціль відповідає експортерам, які надають показники. На ринку є багато експортерів, які дозволяють краще контролювати послуги та інфраструктуру.

Prometheus також надає мову запитів під назвою PromQL, яка дозволяє надсилати запити щодо отриманих показників. Це дозволяє агрегувати дані часових рядів за допомогою попередньо створених функцій.

Grafana — це інструмент візуалізації даних, який допомагає контролювати стан інфраструктури та програм. Він може запитувати дані, створювати інформаційні панелі та надсилати сповіщення на різні канали. Інформаційні панелі пропонують кращу видимість даних і допомагають уникнути проблем із виробництвом, постійно контролюючи систему.

Спеціальні плагіни можна використовувати для інтеграції Grafana з іншими джерелами даних і візуалізаціями даних. Він підключається до таких баз даних, як Prometheus, Elasticsearch, InfluxDB, MySQL, PostgreSQL, Graphite тощо.

Після підключення до цих джерел даних можна створювати інформаційні панелі. Ці інформаційні панелі можуть бути обмежені користувачами з певними правами через керування доступом користувачів. Окрім рішення з відкритим вихідним кодом, Grafana пропонує хмарні та корпоративні рішення.

Alertmanager — це інструмент попередження, який спілкується з Prometheus і надсилає сповіщення на різні канали. Канали можуть бути в PagerDuty, Email, Telegram, Slack тощо. Ми також можемо надсилати ці сповіщення на інші платформи, як-от Teams, за допомогою веб-хуків. Ці веб-хуки мають бути створені на каналі, а URL-адресу потрібно вказати в конфігурації Alertmanager.

Правила, які потрібно контролювати, наведені в конфігурації Prometheus. Коли спрацьовує одне з цих правил, Prometheus передає тригер Alertmanager. Потім Alertmanager надішле сповіщення на канал, щоб сповістити користувачів про активацію правила. Alertmanager дозволяє вимкнути сповіщення, щоб запобігти повторному сповіщенню про тригери тих самих правил.

Prometheus збирає показники з різних кінцевих точок програми та зберігає їх у своїй внутрішній базі даних. Потім Grafana зчитує ці показники з Prometheus і відображає візуалізації та інформаційні панелі на основі цих показників у своєму інтерфейсі користувача. Сповіщення Prometheus також можна налаштувати для надсилання сповіщень зовнішнім системам, таким чином, коли відбувається подія, Prometheus надсилає дані сповіщень до Alertmanager, який, у свою чергу, динамічно направляє сповіщення до різних отримувачів, таких як електронна пошта, Slack або PagerDuty [13].

Деякі програми не повідомляють вам свої показники через спеціальні кінцеві точки, а натомість покладаються на додатковий компонент під назвою «експортер», щоб виконати свою роботу. Експортер читає внутрішні дані певного процесу, а потім відкриває їх через дану кінцеву точку. Вони широко використовуються багатьма популярними базами даних, такими як Postgres, Redis тощо.

Для показників, які не працюють із моделлю Push Prometheus up, використовується Push-шлюз (рисунок 1.5).


Рисунок 1.5 — Архітектура Prometheus-Grafana

ELK стек — це група продуктів з відкритим вихідним кодом від Elastic, розроблених, щоб допомогти користувачам отримувати дані з будь-якого типу джерела та в будь-якому форматі, а також шукати, аналізувати та візуалізувати ці дані в реальному часі. Група продуктів раніше була відома як ELK Stack для основних продуктів у групі — Elasticsearch, Logstash і Kibana — але була перейменована в Elastic Stack. Згодом до стека було додано четвертий продукт, Beats. Elastic Stack можна розгорнути на місці або надати як програмне забезпечення як послуга (SaaS). Elasticsearch підтримує Amazon Web Services (AWS), Google Cloud Platform і Microsoft Azure.

Elasticsearch — це система розподіленого пошуку та аналітики, створена на основі Apache Lucene. Підтримка різних мов, висока продуктивність і документи JSON без схем роблять Elasticsearch ідеальним вибором для різних аналізів журналів і випадків використання пошуку.

Logstash — це інструмент із відкритим вихідним кодом для прийому даних, який дозволяє збирати дані з різних джерел, перетворювати їх і надсилати в бажане місце призначення. Завдяки вбудованим фільтрам і підтримці понад 200 плагінів Logstash дозволяє користувачам легко завантажувати дані незалежно від джерела чи типу даних[14].

Logstash — це легкий конвеєр обробки даних із відкритим вихідним кодом на стороні сервера, який дозволяє збирати дані з різних джерел, перетворювати їх на льоту та надсилати в бажане місце призначення. Він найчастіше використовується як конвеєр даних для Elasticsearch, аналітичної та пошукової системи з відкритим кодом. Завдяки тісній інтеграції з Elasticsearch, потужним можливостям обробки журналів і понад 200 готових плагінів з відкритим кодом, які можуть допомогти вам легко індексувати ваші дані, Logstash є популярним вибором для завантаження даних в Elasticsearch.

Kibana — це інструмент візуалізації та дослідження даних, який використовується для аналітики журналів і часових рядів, моніторингу додатків і випадків використання оперативної аналітики. Він пропонує потужні та прості у використанні функції, такі як гістограми, лінійні графіки, секторні діаграми, теплові карти та вбудовану геопросторову підтримку. Крім того, він забезпечує тісну інтеграцію з Elasticsearch, популярною аналітичною та пошуковою системою, що робить Kibana вибором за умовчанням для візуалізації даних, що зберігаються в Elasticsearch (рисунок 1.6).

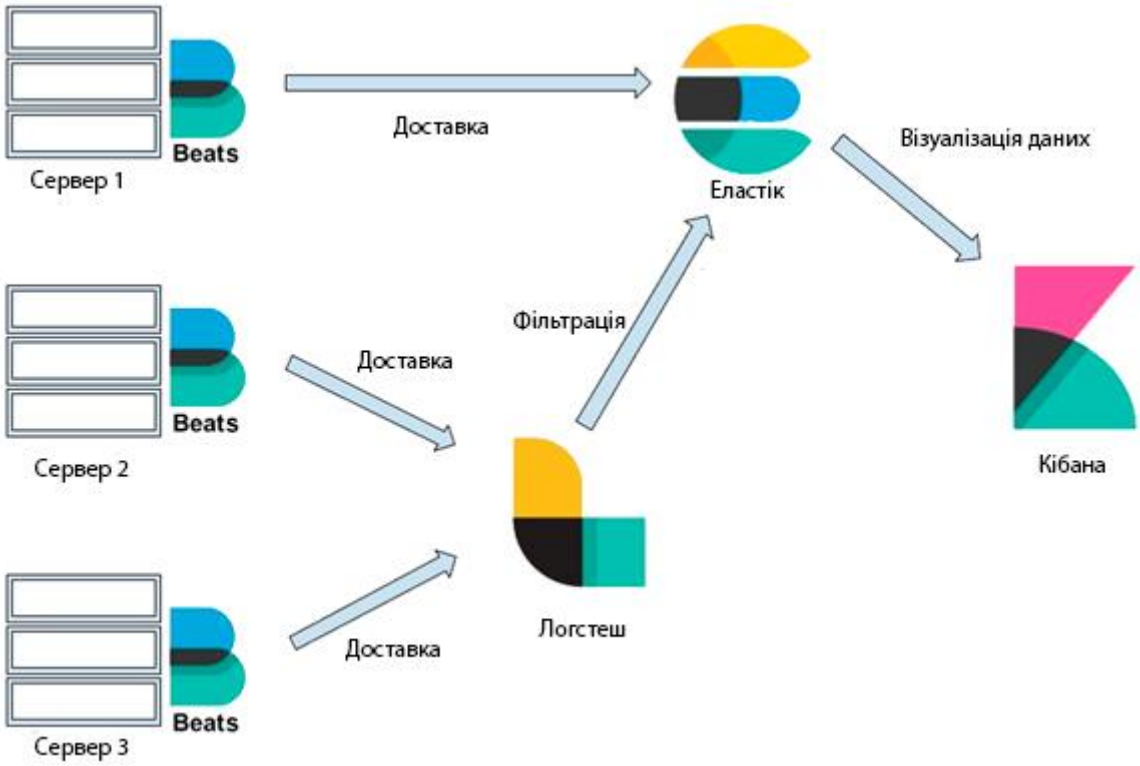



Рисунок 1.6 — Архітектура ELK

Loki stack — це багатокористувацька система агрегації журналів із відкритим кодом. Його можна використовувати з Grafana та Promtrail для збору та доступу до журналів, подібно до стеку ELK/EFK [15]. У той час як Kibana і Elasticsearch можна використовувати для розширеного аналізу даних і візуалізації, стек журналювання на основі Loki зосереджується на тому, що він легкий і простий в експлуатації. Loki надає мову запитів під назвою LogQL, яка дозволяє користувачам запитувати журнали. Його натхненно Prometheus PromQL, і його можна вважати розподіленим «grep», який агрегує джерела журналів. Однією з основних відмінностей від звичайних систем журналювання є те, що Loki індексує лише метадані, а не весь вміст журналів. Таким чином, індекс стає меншим, що зменшує споживання пам’яті та, зрештою, знижує витрати. Одним із недоліків цього дизайну є те, що запити можуть бути менш продуктивними, ніж якщо все проіндексовано та завантажено в пам’ять. Журнали зберігаються безпосередньо в хмарному сховищі, такому як Amazon S3 або GCS, без необхідності зберігати файли на диску. Це

спрощує операції та дозволяє уникнути таких проблем, як нестача місця на диску (рисунок 1.7).

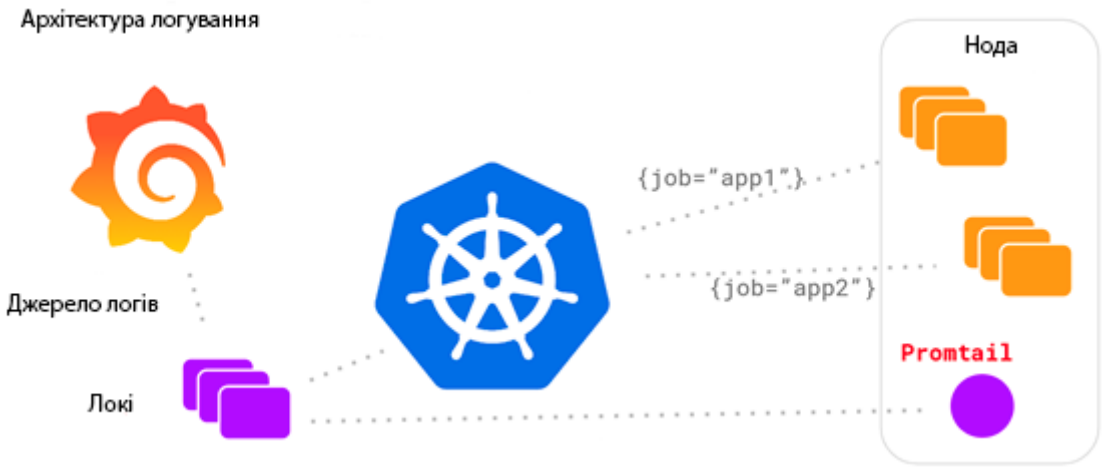



Рисунок 1.7 — Архітектура Loki

Jaeger, натхненний Dapper і OpenZipkin, є розподіленою системою трасування, випущеною Uber Technologies як відкритий код.

OpenTracing складається зі специфікації API, фреймворків і бібліотек, які реалізували специфікацію, і документації для проекту. OpenTracing дозволяє розробникам додавати інструменти до коду своєї програми за допомогою API, які не прив’язують їх до якогось конкретного продукту чи постачальника.

OpenTracing визначає специфікацію, яку реалізують різні проекти з відкритим кодом або комерційні проекти. Найпопулярнішим з них є Jaeger. Це система відстеження з відкритим кодом, створена Uber. У термінології OpenTracing вони називаються трасувальниками. Jaeger дозволяє розробникам візуалізувати дані OpenTracing.

Розподілені платформи спостереження за трасуванням, такі як Jaeger, необхідні для сучасних програмних додатків, які створені як мікросервіси. Jaeger відображає потік запитів і даних, коли вони проходять через розподілену систему. Ці запити можуть здійснювати дзвінки до кількох служб, що може призвести до власних затримок або помилок. Jaeger поєднує ці різнорідні компоненти, допомагаючи виявити вузькі місця продуктивності, усунути помилки та підвищити

загальну надійність додатків. Jaeger є на 100% відкритим вихідним кодом, вбудованим у хмару та нескінченно масштабованим (рисунок 1.8).

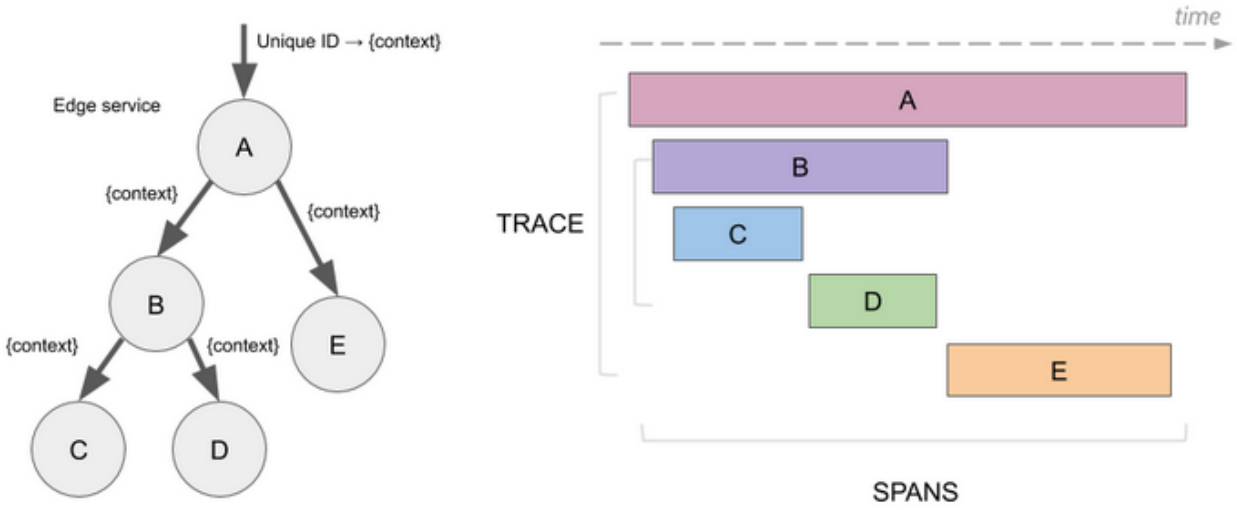



Рисунок 1.8 — Орієнтований ациклічний граф (DAG) проміжків

Отже, якщо порівнювати дане ПЗ, то можна зробити такі висновки.

ELK, відомий також як Elastic Stack, - це пошукова система, спрямована на аналітику та візуалізацію. Його основна задача полягає у зборі, зберіганні та аналізі логів та інших типів даних. Основний компонент для зберігання даних в ELK - це Elasticsearch, який надає можливості для швидкого пошуку і аналізу великих обсягів даних.

Promtail є агентом для збору та надсилання логів до Loki, системи зберігання логів від Grafana Labs. Promtail розроблений спеціально для інтеграції з Loki та використовує її формат для ефективного зберігання. Він може відстежувати зміни у лог-файлах, завдяки чому його часто використовують для збору логів з таких систем, як Kubernetes [16].

Fluent Bit, з іншого боку, є високопродуктивним і легковаговим обробником даних та пересиланням логів. Він є частиною екосистеми Fluentd і розроблений для того, щоб бути невеликим та ефективним рішенням, особливо в обчислювальних умовах з обмеженими ресурсами, таких як IoT або Kubernetes. Fluent Bit підтримує

численні вихідні плагіни, що дозволяє відправляти дані до різних місць зберігання, включаючи Loki, Elasticsearch, Kafka та багато інших.

В загальному, як Promtail, так і Fluent Bit служать для збору та пересилання логів, але вони мають різні особливості та оптимізовані для різних сценаріїв використання. Promtail оптимізований специфічно для роботи з Loki, тоді як Fluent Bit пропонує більше універсальності та гнучкості завдяки своєму розширюваному підходу.

Loki, з іншого боку, спеціалізується на зборі та зберіганні логів. Він часто порівнюється з ELK у контексті обробки логів. Loki створений для інтеграції з Grafana, що дозволяє візуалізувати логи в популярному інструменті візуалізації.

Grafana — це інструмент для візуалізації даних, який може інтегруватися з різноманітними джерелами даних, включаючи Elasticsearch і Prometheus. Його основна задача - це візуалізація метрик, логів та трейсів, що робить його універсальним рішенням для моніторингу (рисунок 1.9).

| Критерій | ELK | Loki | Grafana | Prometheus |
|---|---|---|---|---|
| **Тип** | Пошукова система, аналітика, візуалізація | Логування | Візуалізація | Моніторинг та алертинг |
| **Основна задача** | Збір, зберігання та аналіз логів | Збір і зберігання логів | Візуалізація даних | Збір метрик і алертинг |
| **Зберігання даних** | Elasticsearch | Власна база даних (підтримує різні backends) | Немає (використовує джерела даних) | Time-series database |
| **Масштабованість** | Горизонтальна | Горизонтальна | Залежить від джерела даних | Горизонтальна |
| **Інтеграція** | Інтегрується з багатьма джерелами логів | Пряма інтеграція з Grafana | Підтримує багато джерел даних | Інтегрується з багатьма системами |
| **Ліцензія** | Open Source (з комерційними опціями) | Open Source | Open Source | Open Source |

Рисунок 1.9 — Порівняльна характеристика моніторинг компонентів

Prometheus, на відміну від інших — це система моніторингу та алертингу, яка спеціалізується на зборі метрик у реальному часі. Він зберігає дані у власній time-series базі даних і активно інтегрується з Grafana для візуалізації своїх метрик. У підсумку, ELK і Loki зосереджені на обробці логів, але мають різні підходи і особливості; Grafana - це інструмент візуалізації для різних типів даних; Prometheus зосереджений на моніторингу систем і застосунків через збір метрик [17].

Таким чином, усі засоби моніторингу наведені у цьому підрозділі, важливі. Є певні подібності у функціоналі, проте дані програмні продукти є унікальними і кожен з них знаходить своє місце у високонавантажених системах.

# 2 ТЕХНОЛОГІЯ БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ ТА ДОСТАВКИ

CI/CD — це метод частої доставки додатків клієнтам шляхом впровадження автоматизації на етапах розробки додатків. Основні концепції CI/CD — безперервна інтеграція, безперервна доставка та безперервне розгортання. CI/CD — це рішення проблем, які інтеграція нового коду може спричинити для команд розробки та операцій (відомих також як «інтеграційне пекло»).

Зокрема, CI/CD запроваджує постійну автоматизацію та постійний моніторинг протягом життєвого циклу додатків, від етапів інтеграції та тестування до доставки та розгортання. У сукупності ці підключені практики часто називають «конвеєром CI/CD» і підтримуються командами розробки та операцій, які працюють разом у гнучкий спосіб за допомогою підходу DevOps або розробки надійності сайту (SRE).

Розробка надійності сайту (SRE) — це практика використання програмних засобів для автоматизації завдань IT-інфраструктури, таких як керування системою та моніторинг додатків. Організації використовують SRE, щоб забезпечити надійність своїх програмних програм серед частих оновлень від команд розробників. SRE особливо підвищує надійність масштабованих програмних систем, оскільки керування великою системою за допомогою програмного забезпечення є більш стійким, ніж керування сотнями машин вручну [18].

У даному розділі будуть описані можливість застосування математичних формул, моделей для покращення взаємодії додатків всередині Kubernetes кластера, буде розглянуто GitOps підхід та його моделі доставки ПЗ та ключові метрики, які використовуються для моніторингу стану сервісів та базових систем.

## 2.1 Математичне моделювання Kubernetes кластеру

Kubernetes є відомою платформою для оркестрації контейнерів, і важливо правильно моделювати та оптимізувати ресурси для досягнення максимальної

продуктивності та ефективності кластера. На додачу, правильне управління ресурсами може допомогти уникнути витоків ресурсів, простою та інших проблем.

Теорія масового обслуговування — це розділ математики, який вивчає, як утворюються лінії, як вони функціонують і чому вони не працюють. Теорія масового обслуговування розглядає кожен компонент очікування в черзі, включаючи процес прибуття, процес обслуговування, кількість серверів, кількість місць у системі та кількість клієнтів, якими можуть бути люди, пакети даних, машини чи будь-що інше [19].

Застосування теорії масового обслуговування в реальному житті охоплює широкий спектр підприємств. Його результати можуть бути використані для забезпечення швидшого обслуговування клієнтів, збільшення потоку трафіку, покращення доставки замовлень зі складу або проектування мереж передачі даних і кол-центрів.

Черги можуть виникати щоразу, коли ресурси обмежені. Деякі черги допустимі в будь-якому бізнесі, оскільки повна відсутність черги означатиме дорогий надлишок потужностей. Теорія масового обслуговування спрямована на розробку збалансованих систем, які обслуговують клієнтів швидко та ефективно, але не коштують занадто багато, щоб бути стійкими. На самому базовому рівні теорія масового обслуговування передбачає аналіз прибуття в установу, наприклад банк або ресторан швидкого харчування, а також аналіз процесів, які зараз існують для їх обслуговування. Кінцевим результатом є набір висновків, спрямованих на виявлення будь-яких недоліків у системі та пропонування способів їх усунення.

Дослідження з використанням теорії масового обслуговування розбиває її на шість елементів: процес прибуття, процес обслуговування та відправлення, кількість доступних серверів, дисципліна черги (наприклад, перший прийшов, перший вийшов), місткість черги , і номери, що обслуговуються. Створення моделі всього процесу від початку до кінця дозволяє визначити причину або причини перевантаження та усунути їх [20].

Для розрахунку ресурсів в Kubernetes за допомогою теорії черг, давайте розглянемо найпростіший випадок: систему з однією чергою та одним сервером, де:

Середнє завантаження системи (робоча інтенсивність):

$$\rho = \frac{\lambda}{\mu}$$

де ρ — середнім завантаження системи;

λ — інтенсивність надходження заявок (запитів на додаток, що розгортається в Kubernetes);

μ — інтенсивність обслуговування (здатність нашого серверу обслуговувати заявки).

Середній час відгуку:

$$W = \frac{1}{\mu - \lambda}$$

де W — середнім часом відгуку.

Середня кількість заявок у системі:

$$L = \lambda \times W$$

де L — середня кількість заявок у системі.

Підставляючи ці значення у наші формули:

$$\rho = 100/125 = 0.8,$$
$$W = 1/(125 - 100) = 0.04,$$
$$L = 100 \times 0{,}04 = 4$$

Отже, знаючи середній час відгуку та інші параметри, можна робити висновки щодо оптимальності розміщення ресурсів та потреби в масштабуванні вашого застосунку в Kubernetes.

Моделювання ресурсів у Kubernetes допомагає визначити, скільки ресурсів (CPU, пам'ять, мережа тощо) потрібно для вашої системи. Для цього часто використовуються історичні дані, моніторинг та профілювання. Точне "математичне моделювання" може варіюватися в залежності від конкретного застосунку та його потреб, але ось базова ідея [21].

Середнє використання ресурсів обчислюється як відношення суми використаних ресурсів протягом певного часу до кількості вимірювань.

Пікове використання визначається як найвище значення використання ресурсів протягом визначеного періоду.

Буфер обчислюється як пікове використання, помножене на відсоток буфера, який служить додатковим резервом для врахування можливих піків використання.

Загальні потреби в ресурсах розраховуються як сума середнього використання та буфера, вказуючи необхідну кількість ресурсів для ефективної роботи системи.

Ці формули можуть слугувати основою для простого моделювання ресурсів. Проте важливо зазначити, що реальне середовище Kubernetes може мати численні особливості та залежності, які слід враховувати під час моделювання.

Створимо практичну математичну модель розпаралелювання в Kubernetes. Для цього ми будемо використовувати базові принципи розпаралелювання та особливості Kubernetes.

Модель:

$$T = S \times (P/N + (1 - P) + V)$$

де T — час виконання в паралель;

S — час виконання в одиночному режимі;

P — коефіцієнт паралелізму;

N — кількість ядер (вузлів);

V — накладні витрати часу.

$$Q = S/T$$

де Q — прискорення виконання завдання.

Приклад:

Нехай у нас є Kubernetes кластер з 4 ядрами (вузлами):

Порахуємо час:

$$T = 100 \times (0{,}94 + 0{,}1 + 2) = 32{,}5$$

Підставимо ці дані:

$$E = 100/32{,}5 = 3.08$$

де E — ефективність паралелізму.

Отже, з допомогою розпаралелювання в Kubernetes, наша задача виконується приблизно в 3 рази швидше, ніж без розпаралелювання.

Звісно, в реальних умовах можуть бути додаткові фактори та обмеження, які впливають на розпаралелювання, але дана модель допоможе отримати загальне уявлення про вигоди розпаралелювання в Kubernetes.

Для високої точності рекомендується використовувати моніторингові системи, такі як Prometheus, щоб зібрати реальні дані про використання ресурсів, а також провести навантажувальне тестування для визначення пікового використання та інших ключових метрик [22].

Також, можна звернути увагу на дослідження щодо скейлінгу в Kubernetes. Наприклад використання розподілу Гаусса. Розподіл Гаусса, також відомий як нормальний розподіл, є безперервним розподілом ймовірностей, симетричним відносно свого середнього (середнього) значення. Він часто використовується в статистиці для моделювання розподілу випадкових величин, які відбуваються

природним чином у світі, наприклад розподілу зросту або ваги людей, похибок вимірювання та багатьох інших явищ (рис 2.1).

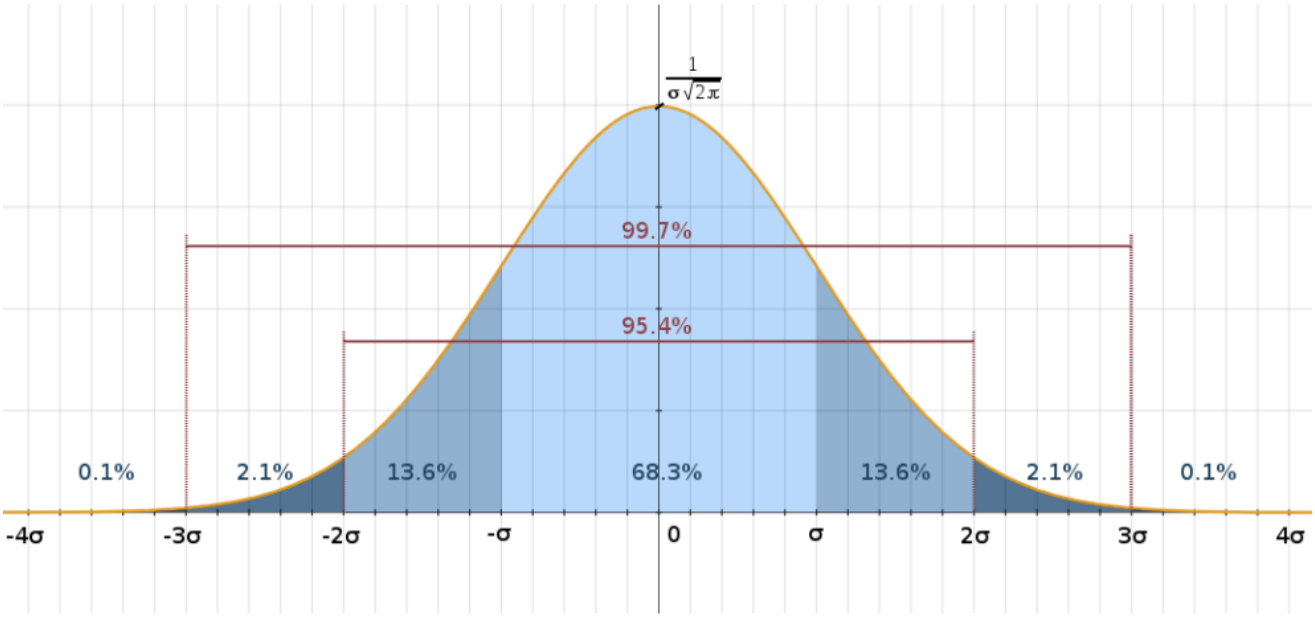



Рисунок 2.1 — Розподіл Гаусса у Kubernetes

Формула:

$$p(\mathrm{x} \mid \mu, \sigma^2) = \mathrm{N}(\mathrm{x};\ \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma}}\ exp\ (-\frac{(x-\ \mu)^2}{2\sigma^2})$$

де μ — очікуване значення;

σ — стандартним відхиленням.

Застосування формул Рімана, дозволяють зробити дещо інші обчислення. В контексті сумування рядів формули Рімана використовуються для опису умов, при яких ряд є збіжним. Одна з таких формул, яку можна використовувати для абсолютно збіжних рядів, називається "Критерієм Рімана" (рисунок 2.5).

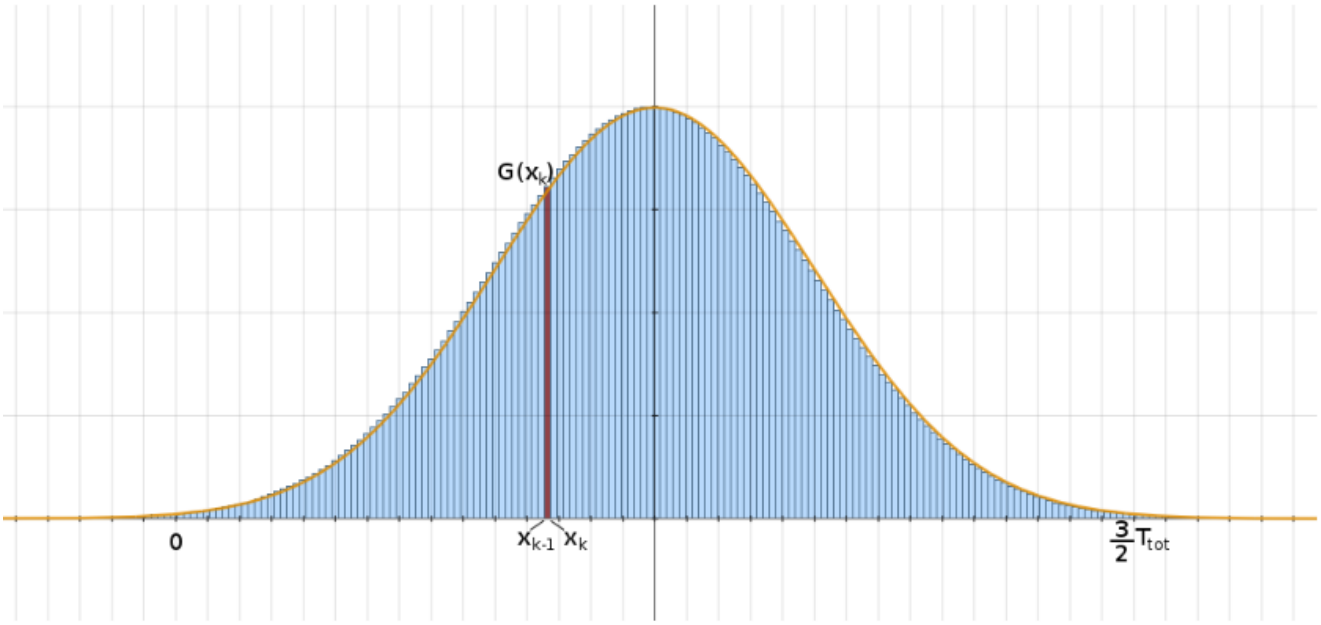



Рисунок 2.2 — Сумування рядів формули Рімана

Формула:

$$\sum_{i=1}^{n} \mathrm{f}(x_i)\Delta \mathrm{x}$$

де f є функцією для наближення (гауссівський метод у нашому випадку).

Додатково, можна спробувати привести декілька спрощених формул оптимізації, які можуть бути загальними стратегіями оптимізації Kubernetes.

Оптимізація використання ресурсів:

$$\text{Потр в рес} = \text{Серед викор} + (\text{Буфер безпеки} \times \text{Пікове використання})$$

Тут ми враховуємо і середнє використання ресурсів, і пікове, додаючи до нього додатковий буфер безпеки.

Оптимізація часу відгуку:

$$\text{Час відгуку} = \text{Час обробки} + \text{Час очікування}$$

Тут ми прагнемо зменшити як час обробки, так і час очікування завдань у черзі.

Оптимізація затрат:

$$\text{Заг витр} = (\text{Вартість ресурсу} \times \text{Час використання}) + \text{Додаткові витрати}$$

Тут мета полягає в мінімізації вартості використаних ресурсів, а також врахуванні додаткових витрат, таких як ліцензії, підтримка тощо.

Реальні сценарії оптимізації можуть бути значно складнішими. Важливо проводити ретельний моніторинг та аналіз розгортання Kubernetes, використовувати відповідні інструменти та регулярно перевіряти систему на предмет можливих поліпшень.

## 2.2 GitOps та порівняння push та pull моделей

GitOps — це операційна структура, заснована на практиках DevOps, таких як безперервна інтеграція/безперервна доставка (CI/CD) і контроль версій, що автоматизує інфраструктуру та керує розгортанням програмного забезпечення. Це дозволяє розробникам зберігати бажаний стан своєї інфраструктури та використовувати його для автоматизації операційних процесів. GitOps — це набір передових практик, які застосовуються від початку робочого процесу розробки до розгортання [23].

Це підхід, орієнтований на розробника, який покладається на інструменти, з якими розробники вже знайомі. Розробники вже використовують Git для вихідного коду програми — GitOps поширює цю практику на конфігурацію програми, інфраструктуру та робочі процедури. GitOps зберігає всі аспекти інфраструктури проекту, включаючи інфраструктуру у вигляді файлів коду, файлів конфігурації та файлів коду програми, у сховищах Git. Усі зміни додатків та інфраструктури автоматично синхронізуються з живим середовищем.

GitOps можна використовувати для керування розгортанням будь-якої інфраструктури. Це особливо корисно для розробників програмного забезпечення та інженерів платформ, які працюють з Kubernetes і хочуть рухатися до моделей безперервної роботи. GitOps спрощує впровадження безперервного розгортання

для хмарних додатків. Це робиться шляхом гарантування негайного відтворення хмарної інфраструктури на основі стану сховища Git.

GitOps —це набір методів розгортання, тоді як DevOps — це парадигма або ще краще — спосіб мислення. Їхні спільні принципи полегшують командам адаптацію робочого процесу GitOps для існуючих методів DevOps.

GitOps розширює практику використання Git для вихідного коду програми на конфігурацію, інфраструктуру та робочі процедури програми. Він зберігає всі аспекти інфраструктури проекту, включаючи інфраструктуру у вигляді файлів коду, файлів конфігурації та файлів коду програми, у сховищах Git, що полегшує керування розгортанням програмного забезпечення та наданням інфраструктури.

Використовуючи декларативний формат, ви можете описати, як працює ваша інфраструктура, а також програми, які працюють на ній. Це дає змогу відстежувати зміни, внесені до будь-якого середовища, і дає змогу відкочувати, відновлювати та атрибути самовідновлення за допомогою будь-якої системи керування джерелами. Відмовтеся від імперативних adhoc сценаріїв до декларативної конфігурації на всіх рівнях (програма та інфра).

Декларативні описи, що зберігаються в репозиторії, підтримують незмінність, керування версіями та історію керування версіями. Наприклад, використання Git для декларацій, згаданих вище, дає вам єдине місце, з якого все для вашої програми буде отримано та керовано. Це дозволяє вам у будь-який час легко визначити будь-які внесені зміни. Не намагайтеся вручну зрозуміти, чим відрізняються два середовища. Просто подивіться на всі зміни, знайдені в історії контролю версій, маючи гарантію, що платформа завжди відповідає тому, що там описано [24].

Використання GitOps означає, що ви використовуєте програмні агенти, які завжди працюють у кластері, автоматично витягуючи стан Git через регулярні проміжки часу та перевіряючи його на поточний стан кластера. Таким чином ви завжди знаєте, чи версія в Git збігається з поточним станом чи ні.

Робота в замкнутому циклі гарантує, що бажаний стан системи відповідає заявленому. Це одна з найважливіших функцій, оскільки вона забезпечує зворотній

зв’язок, що дозволяє вам і вашій команді краще контролювати свої операції та робочий процес (рисунок 2.3).

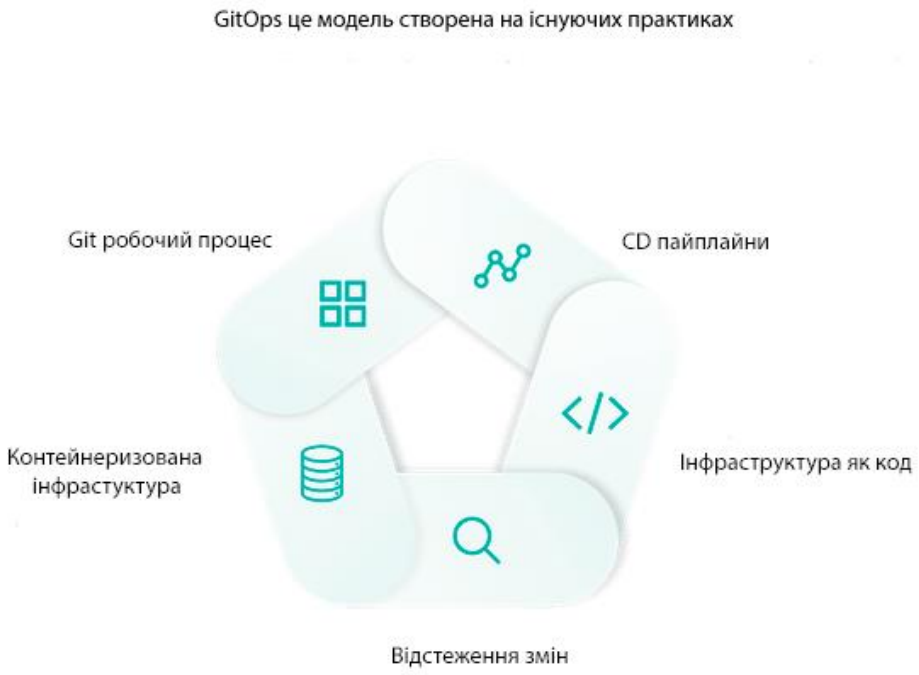



Рисунок 2.3 — Модель GitOps

При використанні з вашим репозиторієм програмні агенти GitOps можуть виконувати різні функції, забезпечуючи самовідновлення. Агенти автоматизують виправлення у разі збою, виконують процеси контролю якості для вашого робочого процесу та захищають від помилок людини чи ручного втручання.

Розробники можуть використовувати GitOps, щоб оптимізувати свій робочий процес, оскільки він використовує можливості контролю версій Git для керування змінами інфраструктури. Коли розробник вносить зміни до репозиторію Git, процес GitOps автоматично ініціює конвеєр розгортання. Це позбавляє розробників від необхідності вручну контролювати кожне розгортання, що робить його швидшим і зменшує ризик помилок. Крім того, використання запитів на внесення змін уможливлює процес рецензування, що покращує контроль якості [25].

Інженери платформ можуть скористатися перевагами GitOps для покращення управління інфраструктурою. Зберігаючи конфігурації інфраструктури в сховищі Git, вони можуть легко відстежувати зміни, відкочувати їх у разі потреби та

досягати узгодженості в багатьох середовищах. GitOps також надає систему для автоматизації розгортань і оновлень, заощаджуючи час і зменшуючи ручні зусилля (рисунок 2.4).

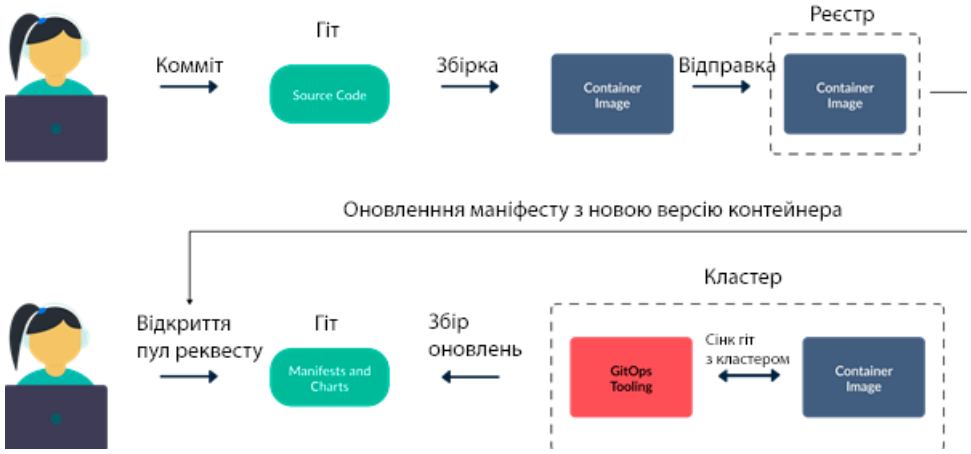



Рисунок 2.4 — Базовий робочий процес GitOps для Kubernetes

Для команд, які розгортають контейнерні програми в Kubernetes або інших хмарних середовищах, GitOps надає ефективний спосіб керувати розгортаннями. Декларативний характер GitOps добре узгоджується з архітектурою Kubernetes, дозволяючи йому автоматично узгоджувати будь-які відмінності між бажаним станом, описаним у Git, і фактичним станом кластера. GitOps також допомагає керувати складними розгортаннями Kubernetes у кількох кластерах і середовищах.

Розглянемо push та pull моделі. У push моделі використовується конвеєр CI/CD (безперервна інтеграція/безперервна доставка), щоб внести зміни у своє середовище. Конвеєр запускається фіксацією або злиттям коду.

Оскільки push використовує добре зрозумілі інструменти CI/CD, він буде найдоступнішим для найширшого кола інженерів у вашій організації. Часто це означає, що ви можете почати швидше та мати ширший пул співпраці.

Сьогодні більшість агентів GitOps у стилі витягування працюють лише в Kubernetes. Якщо ви хочете розгорнути будь-що інше, крім Kubernetes, вам потрібно буде надіслати конфігурацію в це середовище. Існує можливість використання агента Pull GitOps, який працює в кластері Kubernetes, для

оркестрування розгортання на зовнішніх цілях, таких як віртуальні машини та «голі метали», але в кінцевому підсумку ви знову повертаєтеся до виконання push. Агент просувається до цих середовищ. Зрештою вам доведеться керувати багатьма проблемами безпеки, пов’язаними зі збереженням облікових даних поза межами середовища та відкриттям портів мережевого брандмауера, щоб дозволити вхідні з’єднання [26].

Стандартизація однакової методології розгортання для хмарних і традиційних робочих навантажень може підвищити ефективність. Зменшуючи когнітивні витрати, ви спрощуєте адаптацію та навчання нових членів команди, водночас гарантуючи, що існуючі члени команди зосереджуються на вирішенні проблем, а не на розумінні багатьох технологій.

Наявність агента Kubernetes, який постійно опитує ваше сховище Git щодо змін, може не бути проблемою для кількох кластерів, але в масштабі ви можете відчути значне споживання мережевих ресурсів. Крім того, ваш Git і інструменти реєстру контейнерів можуть зазнати значного навантаження, якщо їм потрібно обробляти запити від сотень кластерів, які постійно опитують стан. Ви можете зменшити інтервал опитування, але тоді ви також збільшите затримку між фіксацією коду та розгортанням. Завдяки GitOps у стилі push ви отримуєте найкраще з обох світів, де у вас є швидке розгортання з низькою затримкою, мінімально обкладаючи мережу та інструменти.

За допомогою pull GitOps агент, що працює у вашому середовищі, постійно опитує ваше сховище Git та/або реєстр контейнерів на наявність змін. Коли він виявляє невідповідність між визначеним станом і запущеним станом, агент витягує визначену конфігурацію в середовище.

Якщо всі або більшість ваших робочих навантажень виконуються в Kubernetes, простим і ефективним вибором може бути використання агента для залучення розгортань. Використання агента CD і керування ним відбувається подібно до всіх інших операторів Kubernetes.

Можете задовольнитись зниженою безпекою та ризиком відповідності за допомогою pull GitOps. Оскільки агент CD працює всередині кластера, немає

необхідності зберігати облікові дані у зовнішньому КІ. Так само ви можете зменшити або усунути діри у вашому брандмауері, які дозволяють вхідні з'єднання.

Push GitOps зазвичай працює лише в одному напрямку, від сховища Git до середовища. Pull GitOps працює і в протилежному напрямку. Агент може не тільки опитувати ваше сховище Git і реєстр контейнерів на наявність змін, він також може порівнювати стан кластера з визначеним станом у Git. Це може виявити та виправити дрейф конфігурації у випадку, якщо зміни внесено до кластера вручну або з інших джерел. CI/CD у стилі Push можна налаштувати на регулярне опитування вашого середовища, порівняння робочого стану з визначеним станом і повернення до визначеного стану в разі відхилення конфігурації, але це щось додаткове, що вам потрібно налаштувати. Більшість агентів Pull GitOps оснащено цим двостороннім порівнянням як функцією, якою ви можете скористатися, просто використовуючи агент без додаткового налаштування.

Обидві моделі мають свої переваги та недоліки. Вибір залежить від конкретних потреб, інфраструктури та сценаріїв застосування. Зазвичай pull-модель вважається більш придатною для GitOps завдяки її декларативній природі та здатності автоматично відновлювати стан.

## 2.3 Золоті сигнали моніторингу

З розвитком архітектурних стратегій і технологічним прогресом служби додатків стали більш стійкими. Вони дуже масштабовані, швидше випускаються та легше тестуються.

Потрібна стратегія постійного моніторингу, щоб визначити доступність базової інфраструктури та продуктивність, необхідну для надання високоякісних послуг клієнтам. Тому щоразу, коли служба дає збій, операційній групі потрібно швидко визначити точну проблему, щоб запобігти погіршенню якості обслуговування та негативному впливу на бізнес [27].

Сучасні розподілені системи генерують сотні показників, включаючи показники бази даних, показники хостів, показники додатків, показники

інфраструктури тощо. Постійне відстеження всіх цих показників є недоцільним, але вибір показників для моніторингу є критично важливим для досягнення високодоступного та надійного сервісу (рисунок 2.6).

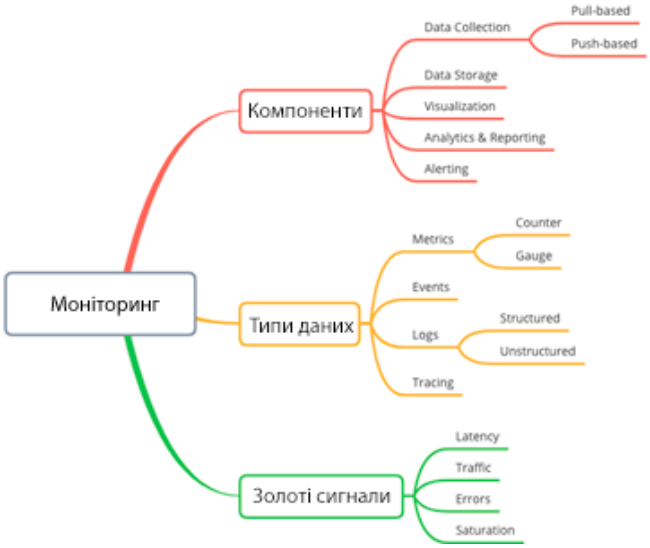



Рисунок 2.6 — Загальне представлення моніторингу

Золоті сигнали — це сигнали, які допомагають узгоджено відстежувати працездатність служби в усіх програмах та інфраструктурі. Є чотири золоті сигнали: затримка, трафік, помилки та насиченість, які стали основними складовими для ефективного моніторингу. Крім того, вони скорочують кількість необхідних показників до чотирьох, забезпечуючи комплексне уявлення про якість обслуговування з точки зору клієнта.

Затримка — це загальний час, потрібний користувачеві, щоб надіслати запит і отримати відповідь. Наприклад, якщо веб-служба зв’язується зі службою бази даних на сервері для перевірки користувача, час, необхідний для виконання запиту до бази даних, вимірюється як частина розрахунку затримки.

Зазвичай затримка вимірюється на стороні сервера; однак ми також можемо виміряти його на стороні клієнта. Обидва мають бути кількісно визначені, але останній важливіший для взаємодії з користувачем. Крім того, ви можете використовувати 95-й процентиль зібраних точок даних про затримку, щоб

оцінити, наскільки добре працює ваша програма. Збільшена затримка є основною ознакою зниження продуктивності. Отже, чим менша затримка, тим кращий сервіс.

Частота помилок визначає кількість невдалих запитів. Операційна команда повинна відстежувати як загальну частоту помилок системи, так і частоту помилок, що виникають у конкретних кінцевих точках служби. Ці помилки вказують на неправильну конфігурацію інфраструктури, збої в роботі, недоліки в коді нашої програми або пошкоджені залежності. Наприклад, раптовий сплеск частоти помилок може означати збій служби, збій бази даних або збій мережі.

Щоб зрозуміти працездатність служби, потрібно розуміти помилки та класифікувати їх на критичні та некритичні сегменти. Це також дає вам змогу швидко діяти та вживати коригувальні заходи.

Обсяг запитів і відповідей, що переміщуються через мережу, вимірюється як трафік. Це може бути інший тип трафіку, наприклад HTTP, FTP тощо, який потрапляє на наш веб-сервер або кінцеві точки API. Крім того, це дозволяє нашим інженерам відрізнити проблеми з пропускною здатністю та неприємні помилки навіть за мінімального трафіку.

Ви можете відстежувати тенденції трафіку, щоб виявити проблеми з пропускною здатністю, неправильні конфігурації, прогнози тощо. Моніторинг трафіку у вашій програмі також допомагає підготуватися до майбутнього попиту.

Компоненти системи, такі як апаратні диски, пам’ять і мережі, досягають точки насичення, коли попит перевищує потужність послуги. У двох словах, насиченість допомагає зрозуміти загальну потужність служби через використання ресурсів. Щоразу, коли система наближається до повного використання своїх ресурсів, це може призвести до зниження потужності. Затримка хвоста може бути ненавмисним наслідком обмеження ресурсів на рівні системи або програми [28].

Імовірно, насичення виникає для будь-яких ресурсів, необхідних програмі, таких як пам’ять, IOPS, ЦП або запити DBS. Що стосується затримки, це може бути 99-й процентиль затримки запиту на обслуговування, який може діяти як індикатор попередження. У багаторівневій системі насиченість може мати каскадний ефект, коли ваш вихідний поток буде нескінченно довго чекати відповіді нижчого сервісу

або, нарешті, тайм-аут, але це також призведе до появи нових запитів у черзі, що призведе до браку ресурсів.

На жаль, коли є балансувальники навантаження та інші автоматизовані методи масштабування, насиченість просто не помітити. Однак балансувальники навантаження іноді можуть не виконувати свої функції через неправильно налаштовані системи, нерегулярне масштабування та інші проблеми. Відстежуючи насиченість, команди можуть виявити проблеми, змінити свій підхід до їх вирішення та уникнути повторення проблем.

Крім того, заслуговують на увагу два додаткові методи: RED (швидкість, помилка, тривалість) і USE (використання, насиченість, помилка). RED зосереджується виключно на моніторингу ваших послуг з точки зору клієнта, незалежно від його інфраструктури. У той же час метод USE зосереджується на використанні ресурсів для швидкого виявлення типових вузьких місць; цей підхід використовує помилки запитів лише як зовнішню ознаку несправності та не може виявити проблеми, пов’язані з затримкою, які також можуть завдати шкоди вашим системам.

Ці сигнали називаються «золотими», оскільки вони спрямовані на оцінку змінних, які мають безпосередній вплив на кінцевих користувачів і компоненти, які виробляють вихід; іншими словами, вони забезпечують точні вимірювання важливих змінних.

Golden Signals намагаються поєднати найкращі аспекти обох цих підходів. Зрештою, усі вони спрямовані на оптимізацію вашої складної розподіленої системи для покращення реагування на кризові ситуації, що робить їх більш цінними, ніж менш точні вимірювання, такі як ЦП, оперативна пам’ять, використання інтерфейсу, затримка та безліч інших показників. Крім того, ви можете використовувати ці золоті сигнали кількома способами, наприклад, для попередження, виявлення аномалій, точного налаштування та планування потужності.

Золоті сигнали є багатим і захоплюючим предметом дослідження, оскільки багато служб досі не відстежують або не надають належних показників. Під час

визначення цілей щодо рівня обслуговування та вибору плану моніторингу для забезпечення надійності вашої програми ці чотири сигнали є чудовою відправною точкою.

Ці ключові показники дозволяють забезпечити стабільну роботу системи, вчасно реагувати на виклики. Важливість ефективного моніторингу полягає в здатності оперативно реагувати на зміни та негайно вирішувати проблеми, що виникають, для забезпечення стабільної та надійної роботи інфраструктури.

# 3 РОЗРОБКА АВТОМАТИЗАЦІЇ ДОСТАВКИ ТА РОЗГОРТАННЯ МОНІТОРИНГУ В KUBERNETES

Автоматизація доставки та розгортання моніторингу є процесом, який включає в себе використання інструментів, практик та методологій для спрощення, стандартизації та прискорення розгортання систем моніторингу в обчислювальному середовищі. У контексті Kubernetes це може означати автоматизацію розгортання та налаштування таких інструментів моніторингу, як Prometheus, Grafana або інших, з метою забезпечення стабільності, продуктивності та безпеки кластера. Це допомагає командам оперативно реагувати на проблеми, оптимізувати ресурси та забезпечувати високу якість сервісу [29].

Автоматизація доставки та розгортання моніторингу стає ключовим компонентом сучасних IT-інфраструктур, особливо у середовищах, що масштабуються і вимагають високої доступності, таких як Kubernetes.

Суть автоматизації полягає в зменшенні ручної роботи та можливих помилок при встановленні, налаштуванні та оновленні систем моніторингу. Це досягається завдяки використанню скриптів, шаблонів, конфігураційних файлів та інших інструментів, які дозволяють автоматично виконувати більшість операцій.

Ціна людської помилки без автоматизації розгортання моніторингу може виявитися досить високою для організації. Коли моніторинг не встановлюється автоматично, існує більший ризик неправильного конфігурування або пропусків у налаштуванні, що може призвести до відсутності важливих сповіщень про проблеми в системі. Неправильно налаштовані або пропущені параметри моніторингу можуть ускладнити виявлення і реагування на критичні проблеми, що, в свою чергу, може призвести до тривалого простою системи або втрати даних. Додатково, коли проблеми з системою не виявляються вчасно через прогалини в моніторингу, організація може зіткнутися з репутаційними збитками, втратою довіри клієнтів та збільшенням загальних витрат на роботу з відновленням системи і вирішенням проблем.

У даному розділі створимо Jenkins пайплайн, опишемо Ansible сценарії та розробимо програму, яка зробить агрегацію даних з моніторинг компоненту та виводитиме звіт про стан помилок у Kubernetes кластері.

## 3.1 Створення Jenkins пайплайну

Jenkins — це інструмент DevOps для автоматизації безперервної інтеграції/безперервної доставки та розгортання (CI/CD) з відкритим кодом, написаний мовою програмування Java. Він використовується для реалізації робочих процесів CI/CD, які називаються конвеєрами [30].

Конвеєри автоматизують тестування та звітування про окремі зміни у більшій кодовій базі в режимі реального часу та полегшують інтеграцію розрізнених гілок коду в основну гілку. Вони також швидко виявляють дефекти в кодовій базі, створюють програмне забезпечення, автоматизують тестування своїх збірок, готують кодову базу для розгортання (доставки) і, зрештою, розгортають код у контейнерах і віртуальних машинах, а також на чистих і хмарних серверах. Є кілька комерційних версій Jenkins. Це визначення описує лише вихідний проект із відкритим кодом.

Jenkins може бути встановлений декількома варіантами: через пакетний менеджер Linux дистрибутива, докер, хельм чарт Kubernetes. Для реалізації підходять перший та другий варіант. Проте якщо нема гіт репозиторію, то залишається перший.

Якщо виконувати встановлення через докер, то краще зібрати власний докер імедж з встановленим ansible заздалегідь. Це робиться за допомогою створення Dockerfile:

```
FROM jenkins/jenkins:lts
USER root
# Install prerequisites
RUN apt-get update
# Install Ansible
RUN apt-get install -y ansible
# Return to the jenkins user
```

USER jenkins

Та збирається за допомогою такої команди: docker build -t jenkins-ansible:latest.

Третій варіант не зовсім підходить, адже його краще застосовувати у межах Kubernetes кластеру, для розгортання додатків чи якихось інших операцій, що краще розмежовувати відповідно до середовищ, що живуть на цих кластерах, не варто прив'язувати менеджмент кластерами з якогось одного кластеру. Одним з прикладів може бути бажання видалити кластер, в наслідок чого прийдеться переносити дженкінс конфігурації, крім того прийдеться робити вікно у робочому графіку для технічного обслуговування і повідомити заздалегідь про втручання.

Після інсталяції та авторизації, відкриється вікно з налаштуваннями (рисунок 3.1).

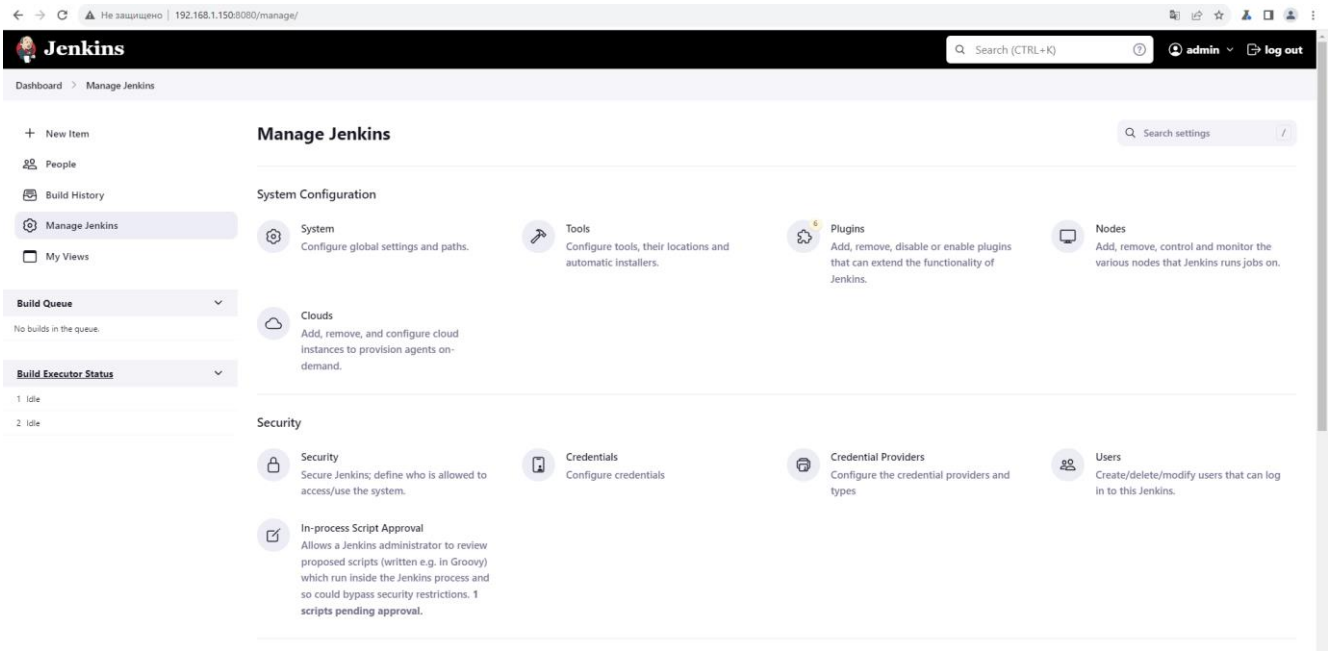



Рисунок 3.1 — Меню налаштування Jenkins

Jenkins пропонує вибір між рекомендованими плагінами та власним вибором. Рекомендовані плагіни зазвичай включають основні інструменти для неперервної інтеграції та розгортання.

Далі, необхідно створити адміністративний обліковий запис. Цей обліковий запис слід захистити надійним паролем, оскільки він має повний доступ до системи Jenkins.

Також важливо налаштувати конфігурацію з'єднання: URL сервера, проксі (якщо вони використовуються), а також налаштування електронної пошти для сповіщень.

Додаткові налаштування безпеки включають настройку автентифікації та авторизації. Jenkins підтримує різні методи автентифікації, включаючи власну базу даних, LDAP або інтеграцію з системами сторонніх виробників [31].

Щодо агентів збірки: якщо планується використовувати додаткові машини для збірки, необхідно налаштувати агентів збірки, вказавши методи з'єднання, такі як SSH або JNLP, та надати їм необхідні права.

На кінець, користувач може налаштувати робочі процеси, додавати додаткові плагіни, налаштувати зовнішні інструменти збірки та задачі CI/CD згідно зі своїми потребами.

Створимо звичайний пайплайн, що буде запускати локальну інсталяцію ansible, який в свою чергу запускатиме сценарій для інсталяції моніторингу на Kubernetes кластері.

Код пайплайну:

```
pipeline {
    agent any
    stages {
        stage('Deploy Kubernetes monitoring using Ansible') {
            steps {
                script {
                    ansiblePlaybook(
                        playbook: 'path-to-playbook.yml',
                        inventory: 'path-to-inventory.ini',
                        credentialsId: 'ansible-ssh-credentials'
                    )
                }
```

```
        }
      }
   }
}
```

На даному етапі це простий пайплайн, його можна покращити додавши можливість застосування cloud git репозиторію, який можна буде викачувати при кожному запуску процесу збірки, для того щоб не зберігати це локально.

Приклад коду:

```
  stages {
     stage('Checkout') {
        steps {
           // Get the code from a version control system.
           checkout scm
        }
     }
```

Створимо простий ansible сценарій, який допоможе перевірити як відпрацює Jenkins job та запустить playbook з перевіркою на ping, яка покаже що є доступ до Kubernetes кластеру.

Ansible playbook:

```
---
- name: Ping Hosts
  hosts: all
  tasks:
    - name: Check if host is reachable
      ping:
```

У даному випадку вказано брати усі хости з інвенторі файлу

Inventory файл:

```
[kubernetes]
kube-master ansible_host=10.0.0.100
```

Якщо запускати плейбук напряму, з консолі то команда буде виглядати так: ansible-playbook -i hosts.ini ping-hosts.yml.

Запустивши у дженкінсі джобу з плейбуком, вивід буде наступним:

```
Started by user Homelab
[Pipeline] Start of Pipeline
[Pipeline] node
```

```
Running on Jenkins in /var/jenkins_home/workspace/Setup Kubernetes Monitoring/Check Kubernetes Master Connecton
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Run Ansible Playbook)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ ansible -i /etc/ansible/hosts -m win_ping win --limit titan
titan | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Отже доступ є і можна переходити до створення ansible сценарію, який буде потрібен для автоматизації встановлення моніторингу на Kubernetes кластері.

### 3.2 Опис ansible сценарію

Ansible — це програмне забезпечення для IT-автоматизації з відкритим вихідним кодом, написане на мові Python. Він може налаштовувати системи, розгортати програмне забезпечення та оркеструвати розширені робочі процеси для підтримки розгортання програм, оновлень системи тощо [32].

Основними перевагами Ansible є простота та легкість використання. Він також має значну увагу на безпеці та надійності, має мінімальну кількість рухомих частин. Він використовує OpenSSH для транспортування (з іншими транспортними засобами та режимами витягування як альтернативою).

Ansible — це радикально простий механізм автоматизації IT, який автоматизує хмарне забезпечення, керування конфігурацією, розгортання додатків, оркестрування всередині служби та багато інших потреб IT.

Будучи розробленим для багаторівневого розгортання з першого дня, Ansible моделює вашу IT-інфраструктуру, описуючи взаємозв’язок усіх ваших систем, а не просто керуючи однією системою за раз.

Ansible працює, підключаючись до ваших вузлів і надсилаючи їм скрипти під назвою «модулі Ansible». Більшість модулів приймають параметри, які описують бажаний стан системи. Потім Ansible виконує ці модулі (через SSH за замовчуванням) і видаляє їх після завершення. Ваша бібліотека модулів може розташовуватися на будь-якій машині, і не потрібні сервери, демони чи бази даних.

Ви можете писати свої власні модулі, хоча спершу слід подумати, чи варто це робити. Як правило, ви працюєте зі своєю улюбленою термінальною програмою, текстовим редактором і, можливо, системою контролю версій, щоб відстежувати зміни у вашому вмісті. Ви можете писати спеціалізовані модулі будь-якою мовою, яка може повертати JSON (Ruby, Python, bash тощо).

Плагіни розширюють основні функції Ansible. У той час як модулі виконуються на цільовій системі в окремих процесах (зазвичай це означає на віддаленій системі), плагіни виконуються на вузлі керування в процесі /usr/bin/ansible. Плагіни пропонують опції та розширення для основних функцій Ansible - перетворення даних, журналювання виводу, підключення до інвентарю тощо. Ansible поставляється з низкою зручних плагінів, і ви можете легко написати свій власний. Наприклад, ви можете написати плагін інвентаризації для підключення до будь-якого джерела даних, яке повертає JSON. Плагіни мають бути написані мовою Python [33].

За замовчуванням Ansible представляє машини, якими він керує, у файлі (INI, YAML тощо), який поміщає всі ваші керовані машини в групи за вашим власним вибором.

Для додавання нових машин не потрібно залучати додатковий сервер підпису SSL, тому ніколи не буде проблем із вирішенням, чому певну машину не було підключено через незрозумілі проблеми з NTP або DNS.

Якщо у вашій інфраструктурі є інше джерело правди, Ansible також може підключитися до нього. Ansible може отримувати інвентаризацію, групову та змінну інформацію з таких джерел, як EC2, Rackspace, OpenStack тощо.

Playbooks може точно оркеструвати кілька фрагментів вашої топології інфраструктури з дуже детальним контролем над тим, скільки машин потрібно опрацьовувати одночасно. Ось де Ansible починає ставати найцікавішим.

Підхід Ansible до оркестровки полягає в тонко налаштованій простоті, оскільки ми вважаємо, що ваш код автоматизації повинен мати ідеальний сенс для вас через роки, і потрібно дуже мало пам’ятати про спеціальний синтаксис або функції.

Модулі, утиліти модулів, плагіни та ролі можуть знаходитися в різних місцях. Якщо ви напишете власний код для розширення основних функцій Ansible, у вас може бути кілька файлів із схожими або однаковими іменами в різних місцях на вашому вузлі керування Ansible. Шлях пошуку визначає, які з цих файлів Ansible виявить і використає під час будь-якого запуску п’єси.

Шлях пошуку Ansible поступово збільшується протягом циклу. Оскільки Ansible знаходить кожну п’єсу та роль, включену в даний запуск, він додає будь-які каталоги, пов’язані з цією ігрою чи роллю, до шляху пошуку. Ці каталоги залишаються в області дії протягом виконання, навіть після завершення виконання ролі.

Опишемо сценарій для розгортання моніторингу в Kubernetes. У даному випадку буде інтеграція нашого кластеру з Grafana Cloud.

Код сценарію:

```
---
- name: Deploy Grafana Cloud Manifest on Kubernetes
  hosts: kubernetes_master
  become: yes
  tasks:
    - name: Check if kubectl is installed
      command: kubectl version --client
      register: kubectl_result
```

```
    changed_when: false
    failed_when: "'Client Version' not in kubectl_result.stdout"

  - name: Set Kubernetes context (if necessary)
    # Ensure you have the right context or credentials here
    command: kubectl config use-context your-context-name
    changed_when: false

  - name: Check if manifest directory exists
    stat:
      path: "/path/to/jenkins/cloned/repo"
    register: repo_stat
    failed_when: not repo_stat.stat.exists

  - name: Apply Grafana Cloud manifest
    command: kubectl apply -f https://path-to-your-grafana-cloud-manifest.yaml
    register: apply_result
    changed_when: "'created' in apply_result.stdout or 'configured' in apply_result.stdout"
```

У даному сценарії йде підключення на Kubernetes мастер, перевіряється чи встановлена утиліта kubectl. Далі перевіряємо чи є папка з клонованим власним репозиторієм, де є маніфест. Після цього застосовується маніфест і можна перевіряти результат.

Перевірка:

```
athena@owl:~$ microk8s kubectl -n monitoring get po
NAME                                      READY   STATUS    RESTARTS      AGE
grafana-agent-integrations-ds-rf2pd       2/2     Running   0             10m
grafana-agent-logs-54dhp                  2/2     Running   0             10m
grafana-agent-operator-6564b87c8f-q7vw5   1/1     Running   0             10m
grafana-agent-0                           2/2     Running   0             10m
grafana-agent-integrations-deploy-6d96dbd478-ppzbv   2/2   Running   0   10m
kube-state-metrics-5595845f7c-8msdt       1/1     Running   0             10m
```

Усі поди піднялися, це означає що моніторинг успішно встановився. Графіки та логи будуть розглянуті у четвертому розділі дипломної роботи.

Окрім цього встановлювати моніторинг компоненти, можна не тільки через маніфести, але й за допомогою хельм чартів.

Helm chart — це пакет, який містить усі необхідні ресурси для розгортання програми в кластері Kubernetes. Це включає файли конфігурації YAML для розгортань, служб, секретів і конфігураційних карт, які визначають бажаний стан вашої програми.

Helm об’єднує файли YAML і шаблони, які можна використовувати для створення додаткових файлів конфігурації на основі параметризованих значень. Це дозволяє налаштовувати файли конфігурації відповідно до різних середовищ і створювати багаторазові конфігурації для використання в кількох розгортаннях. Крім того, кожну діаграму Helm можна контролювати й керувати незалежно, що полегшує підтримку кількох версій програми з різними конфігураціями.

Архітектура Helm складається з двох основних компонентів: клієнта та бібліотеки [34].

Клієнт Helm — це утиліта командного рядка для кінцевих користувачів, яка дозволяє контролювати розробку локальної діаграми та керувати репозиторіями та випусками. Подібно до використання клієнта бази даних MySQL для виконання команд MySQL, ви використовуєте клієнт Helm для запуска команд Helm.

Бібліотека Helm виконує всю важку роботу. Він містить фактичний код для виконання операцій, зазначених у команді Helm. Поєднання файлів конфігурації та діаграм для створення будь-якого випуску обробляється бібліотекою Helm.

Архітектура Helm значно покращилася між версіями 2 і 3. Версія 2 використовувала сервер Tiller для посередництва між клієнтом Helm і сервером Kubernetes API. Він відстежував усі ресурси, створені за допомогою Helm. На користь клієнтської архітектури версія 3 видалила сервер Tiller, натомість використовуючи пряме з’єднання API для взаємодії з сервером Kubernetes API.

Бібліотека програм Helm використовує діаграми для визначення, створення, встановлення та оновлення програм Kubernetes. Діаграми Helm дозволяють керувати маніфестами Kubernetes без використання інтерфейсу командного рядка

(CLI) Kubernetes або запам’ятовування складних команд Kubernetes для керування кластером.

Розглянемо практичний сценарій, коли Helm корисний. Припустімо, ви хочете розгорнути свою програму у робочому середовищі з десятьма репліками. Ви вказуєте це у файлі YAML розгортання для програми та запускаєте розгортання за допомогою команди kubectl.

Тепер запустіть ту саму програму в проміжному середовищі. Припустімо, що вам потрібні три репліки в проміжному середовищі, і ви запустите внутрішню збірку програми в проміжному середовищі. Для цього оновіть кількість реплік і тег зображення Docker у файлі YAML розгортання, а потім використовуйте його в проміжному кластері Kubernetes.

Оскільки ваша програма стає складнішою, кількість файлів YAML збільшується. Згодом кількість настроюваних полів у файлі YAML також збільшується. Незабаром оновлення багатьох файлів YAML для розгортання однієї програми в різних середовищах стає складним для керування.

Використовуючи Helm, ви можете параметризувати поля залежно від середовища. У попередньому прикладі замість використання статичного значення для реплік і зображень Docker ви можете взяти значення для цих полів з іншого файлу. Цей файл називається values.yaml.

Тепер можете підтримувати файл значень для кожного середовища з відповідними значеннями для кожного. Helm допомагає відокремити настроювані значення полів від фактичної конфігурації YAML.

Є кілька ознак того, що проект може отримати користь від використання Helm. Наприклад, припустімо, що ваш проект включає кілька програм Kubernetes, якими потрібно керувати та розгортати разом. У цьому випадку Helm може допомогти, дозволяючи вам упакувати ці програми в єдину діаграму, що спростить керування ними та розгортання їх разом.

Якщо ваш проект передбачає часте оновлення та розгортання програм Kubernetes, Helm може допомогти, надавши інструменти для керування життєвим

циклом ваших програм. Це включає їх розгортання, оновлення та відкат. Як наслідок, оновленнями додатків можна легше керувати та розгортати їх.

Нарешті, якщо ваш проект включає кілька команд або учасників, яким потрібно співпрацювати над розробкою та розгортанням програм Kubernetes, Helm може допомогти, надавши можливості керування версіями та спільного використання для ваших діаграм. Це полегшує роботу команд і забезпечує узгодженість між розгортаннями.

Опишемо сценарій Ansible з застосуванням хельм чарту. За допомогою нього встановимо Opensearch.

Код сценарію:

```
---
- name: Helm Chart Deployment
  hosts: kubernetes_master
  become: yes
  vars:
    helm_repo_name: "{{ helm_repo_name | default('my_default_repo_name') }}"
    helm_repo_url: "{{ helm_repo_url }}"
    helm_chart_name: "{{ helm_chart_name }}"
    helm_release_name: "{{ helm_release_name }}"
    helm_namespace: "{{ helm_namespace }}"
    helm_values_file: "{{ helm_values_file | default('') }}"

  tasks:
    - name: Check if Helm is installed
      command: helm version --client
      register: helm_check
      changed_when: false
      failed_when: "'Version:' not in helm_check.stdout"

    - name: Add Helm repository
      command: "helm repo add {{ helm_repo_name }} {{ helm_repo_url }}"
      when: helm_repo_url is defined
      changed_when: true
```

```
  - name: Update Helm repositories
    command: helm repo update
    changed_when: false

  - name: Install or upgrade Helm chart
    command: >
      helm upgrade --install {{ helm_release_name }} {{ helm_repo_name }}/{{ helm_chart_name }}
      --namespace {{ helm_namespace }}
      --create-namespace
      {{ '-f ' + helm_values_file if helm_values_file else '' }}
    register: helm_install
    changed_when: "'Release \"' + helm_release_name + '\" has been upgraded' in helm_install.stdout or 'Release \"' + helm_release_name + '\" does not exist. Installing it now.' in helm_install.stdout"
```

У даному сценарії перевіряємо чи helm утиліта встановлена, далі додаємо бажаний хельм репозиторій. Оновлюємо інформацію про його релізи і встановлюємо необхідний компонент.

Створимо додатково дженкінс пайплайн для запуску плейбуку.

Код пайлайну:

```
pipeline {
    agent any

    parameters {
        string(name: 'helm_repo', defaultValue: 'my_repo', description: 'Helm Repository Name')
        string(name: 'repo_url', defaultValue: 'https://example.com/charts', description: 'Helm Repository URL')
        string(name: 'chart_name', defaultValue: 'my_chart', description: 'Name of the Helm Chart')
        string(name: 'release_name', defaultValue: 'my_release', description: 'Helm Release Name')
        string(name: 'namespace', defaultValue: 'default', description: 'Kubernetes Namespace')
    }
    stages {
        stage('Checkout') {
            steps {
```

```
                checkout scm
            }
        }
        stage('Deploy Helm Chart') {
            steps {
                sh """
                ansible-playbook helm_deploy.yml -e helm_repo=${params.helm_repo} -e
repo_url=${params.repo_url} -e chart_name=${params.chart_name} -e
release_name=${params.release_name} -e namespace=${params.namespace}
                """
            }
        }
    }
}
```

Як альтернатива, хельм чарти можна автоматично встановлювати по пулл моделі GitOps.

Одним з варіантів може бути Fleet. Fleet — це механізм керування контейнерами та розгортання, розроблений, щоб запропонувати користувачам більше контролю над локальним кластером і постійний моніторинг через GitOps. Fleet зосереджується не лише на можливості масштабування, але також надає користувачам високий ступінь контролю та видимості для моніторингу того, що саме встановлено в кластері.

Fleet може керувати розгортаннями з git необробленого Kubernetes YAML, Helm charts, Kustomize або будь-якої комбінації з трьох. Незалежно від джерела, усі ресурси динамічно перетворюються на діаграми Helm, а Helm використовується як двигун для розгортання всіх ресурсів у кластері. У результаті користувачі можуть насолоджуватися високим ступенем контролю, узгодженості та можливості аудиту своїх кластерів.

Fleet — це, по суті, набір користувацьких визначень ресурсів Kubernetes (CRD) і контролерів, які керують GitOps для одного кластера Kubernetes або великомасштабного розгортання кластерів Kubernetes. Це розподілена система

ініціалізації, яка дозволяє легко налаштовувати програми та керувати кластерами високої доступності з єдиної точки (рисунок 3.4).

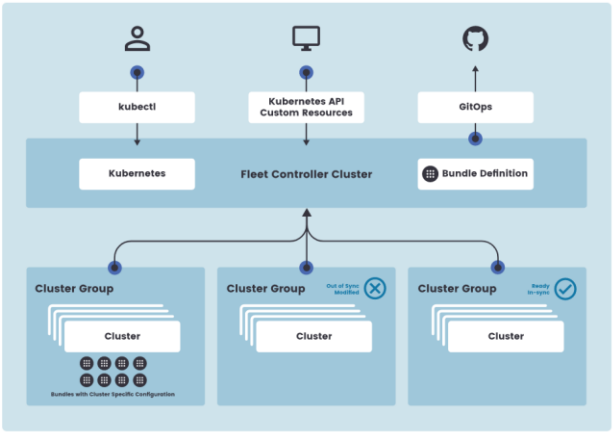



Рисунок 3.4 — Архітектура Fleet

До речі, встановити Fleet можна і через вже готовий дженкінс пайплайн. Спочатку встановити хельм чарт, а далі передати йому конфігураційний файл у вигляді маніфесту, який теж можна застосувати через власну автоматизацію.

Приклад маніфесту:

```
kind: GitRepo
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: helm
spec:
  repo: https://github.com/rancher/fleet-examples
  paths:
  - single-cluster/manifests
```

Структуру гітхаб репозиторію можна подивитися у прикладах надані розробником. Після застосування цих змін fleet буде дивитися за вказаним репозиторієм та перевстановлювати сервіси, якщо знайдуться нові зміни.

Отже, було створено два ansible сценарії для встановлення моніторинг компонентів. Один взаємодіє через маніфести з Kubernetes, інший через хельм. У наступному підрозділі буде створення програми, мета якої дати більш агреговану інформацію щодо подій на кластері, взаємодіючи з API Opensearch. Також

продемонстровано, що можна застосовувати GitOps підхід для автоматичного встановлення компонентів або сервісів.

## 3.3 Автоматизація генерування статистики подій Kubernetes

Події Kubernetes є багатим джерелом інформації. Ці об’єкти можна використовувати для моніторингу програми та стану кластера, реагування на збої та виконання діагностики. Події генеруються, коли ресурси кластера, такі як модулі, розгортання або вузли, змінюють стан.

Події Kubernetes автоматично генеруються, коли над об’єктами в кластері виконуються певні дії, наприклад, коли створюється пакет, створюється відповідна подія. Іншими прикладами є зміни статусу пакета на очікування, успішне або невдале. Це включає такі причини, як виселення контейнера або збій кластера.

Події також генеруються, коли відбувається зміна конфігурації. Зміни конфігурації для вузлів можуть включати горизонтальне масштабування шляхом додавання реплік або вертикальне масштабування шляхом оновлення пам’яті, ємності вводу/виводу диска або ядер вашого процесора.

Сценарії планування або невдалого планування також генерують події. Збої можуть виникати через недійсний доступ до сховища образів контейнера, недостатню кількість ресурсів або якщо контейнер не пройшов тест живучості чи готовності. Це не вичерпний список, є ще багато причин для цього.

Збирати Kubernetes події та зберігати можна як в Loki та через Grafana відображати у realtime режимі, результати якого будуть зображені у четвертому розділі, так і в Elasticsearch/Opensearch для подальшого створення репорту про проблеми на кластері. Використовувати будемо для цього мову програмування Python.

Для того щоб підключитись до Opensearch треба встановити бібліотеку pip install opensearch-py.

Описуємо підключення і назви подій, які цікавлять.

Код програми:

```
import argparse
```

```
import yaml
from opensearchpy import OpenSearch

client = OpenSearch("https://user:pass@host:9200", use_ssl=True, verify_certs=False,
                    ssl_show_warn=False)

general_dict = {}

core_component_reasons = ['Evicted', 'Failed', 'FailedScheduling', 'ImageGCFailed', 'InternalError',
'NodeNotReady', 'NodeStartFailure', 'PolicyViolation', 'ProvisioningFailed', 'Rebooted', 'SystemOOM',
'Unexpected', 'UpdateFailed']
env_reasons = ['BackOff', 'BackoffLimitExceeded', 'Unhealthy']
```

Списками зазначені дві категорії подій, одна стосується середовищ/namespace, що живуть на кластері, інші стосуються самого кластеру, а саме його компонентів.

Код програми:

```
def create_report(cluster):
    with open(r"+cluster+'-alerts-report.yaml', 'w') as file:
        yaml.safe_dump(general_dict, file)
        print(cluster + "-alerts-report.yaml created")
def show_clusters(time_range):
    query_body = {
        "query": {"bool": {"filter": [{"range": {"@timestamp": {"gte": "now-"+time_range, "lte":
"now"}}},
                                      {"query_string": {"query": "(type.keyword=\"Warning\")"}}]}},
        "size": 0,
        "aggs": {
            "cluster": {
                "terms": {
                    "field": "cluster.keyword",
                    "size": 1000,
                    "order": {"_count": "desc"}
                }
            }
        },
```

```
    }
    resp = client.search(index="homelab-*", body=query_body)

    clusters = []
    print("Top clusters by warnings:")
    for key in resp['aggregations']['cluster']['buckets']:
        print(key['key'], key['doc_count'])
        clusters.append(key['key'])

    return clusters
```

Створено дві функції, перша потрібна для того щоб генерувати репорт у yaml форматі, інша зробить пошук о заданому індексу і виведе список кластерів підключених до Opensearch.

Інші функції будуть створені для того щоб шукати причини проблем, namespace, імена сервісів та додавати кількість разів, коли проблема дала про себе знати. Повний код програми буде в додатках.

Проте, хочу звернути увагу на головну функцію, яка дозволить працювати додатку як консольна утиліта через яку можна передавати параметри.

```
def main(cluster, namespace_query_size, reasons, name_query_size, time_range):
    if type(reasons) == list:
        reasons.sort()
    if cluster == "all":
        cluster_list = top_clusters_by_warnings(time_range)
    else:
        cluster_list = cluster
    for cluster_name in cluster_list:
        if reasons == "all":
            top_warning_reasons_by_cluster(cluster_name, time_range, name_query_size, name_query_size)
        else:
            top_namespaces_by_reason(cluster_name, reasons, namespace_query_size, time_range, name_query_size)
```

```
if name == '__main__':
    parser = argparse.ArgumentParser()

    parser.add_argument('-c', '--cluster', nargs='*', type=str, help='Set flag list of clusters. Ex: psf-preprod bf-preprod', default="all")
    parser.add_argument('-ns', '--namespace_query_size', type=int, help='Set flag top number of namespaces. Ex: 5, 10', default=5)
    parser.add_argument('-r', '--reason', nargs='*', type=str, help='Set flag core, env or list of custom reasons. Ex: BackOff', default="all")
    parser.add_argument('-nsq', '--name_query_size', type=int, help='Set flag top number of names. Ex: 5, 10', default=5)
    parser.add_argument('-t', '--time_range', type=str, help='Set flag in min, hours, days. Ex: 15m, 3h, 7d', default="30d")
    args = parser.parse_args()
    cluster, namespace_query_size, reason, name_query_size, time_range = args.cluster, args.namespace_query_size, args.reason, args.name_query_size, args.time_range
    main(cluster, namespace_query_size, reason, name_query_size, time_range)
```

Приймаємо параметри назв кластерів, розміру неймспейсів та імен сервісів у них. Також маємо опцію вказати за який період часу цікавить статистика.

У головній функції викликаємо пошук кластерів та проблем у них. Після відпрацювання програми отримаємо репорт про наявні проблеми.

Результат перевірки кластеру:

```
homelab:
  BackOff:
  - monitoring:
      grafana-7d6d878cc7-pjhkv: 2
```

У процесі розробки автоматизації доставки та розгортання моніторингу в Kubernetes було визначено ключові потреби користувачів та основні виклики, які виникають при масштабуванні та управлінні додатками. Автоматизоване рішення дозволяє значно спростити процес встановлення та конфігурації моніторингових інструментів, гарантуючи при цьому високу надійність та продуктивність системи.

Було досліджено і враховано ряд ключових аспектів. Перш за все, актуальність такого рішення обумовлена постійно зростаючою складністю архітектур та потребою в надійних моніторингових системах для Kubernetes.

Виявлено, що традиційні методи моніторингу можуть бути неефективними в контексті мікросервісних архітектур та контейнерізації. Тому було вирішено розробити автоматизоване рішення, яке б не лише враховувало особливості Kubernetes, але й надавало гнучкість у виборі та конфігурації моніторингових інструментів.

Додатково, з урахуванням зростаючої потреби в швидкому виявленні проблем та мінімізації часу на їх усунення, автоматизація процесів стає критично важливою. Це дозволяє IT-командам відслідковувати стан ресурсів, взаємодії сервісів та забезпечувати високу доступність сервісів.

Однією з ключових особливостей, які були додано до системи моніторингу, є інтеграція з Kubernetes events. Це дозволяє командам оперативно відслідковувати важливі події в життєвому циклі ресурсів, таких як створення, оновлення чи видалення контейнерів і подів. Така інтеграція поліпшує можливість виявлення та реагування на різні інциденти, що можуть виникнути в Kubernetes кластері.

# 4 ТЕСТУВАННЯ CI/CD ПАЙПЛАЙНУ ТА МОНІТОРИНГОВОЇ СИСТЕМИ

Розділ має на меті дослідити та оцінити ефективність та надійність Continuous Integration/Continuous Deployment (CI/CD) пайплайну та моніторингової системи в контексті розробки програмного забезпечення. У цьому розділі буде проведено ретельний аналіз процесів автоматизованої поставки (CI/CD) та системи моніторингу, спрямований на виявлення можливих проблем, покращень та оптимізацій.

## 4.1 Верифікація встановлення моніторингу

Запустимо створену автоматизацію в Jenkins для розгортання моніторинг компонентів.

Вивід консолі:

```
Started by user Homelab
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/Setup Kubernetes Monitoring/GrafanaCloud
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Run Ansible Playbook)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ ansible-playbook -i /etc/ansible/hosts /etc/ansible/playbook/setup_mon.yaml

PLAY [Apply Grafana Cloud Values Manifest] *************************************

TASK [Gathering Facts] *********************************************************
ok: [athena]

TASK [Apply Grafana Cloud Values Manifest] *************************************
changed: [athena]

PLAY RECAP *********************************************************************
athena                     : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
```

[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

Переглянемо зміни що відбулися на кластері, а саме стан подів (рис 4.2).

```
athena@owl:~$ microk8s kubectl -n monitoring get po
NAME                                                READY   STATUS
grafana-agent-operator-6564b87c8f-q7vw5             1/1     Running
grafana-agent-integrations-ds-rf2pd                 2/2     Running
kubernetes-event-exporter-6bc76f867d-j8j69          1/1     Running
grafana-agent-integrations-deploy-6d96dbd478-ppzbv  2/2     Running
grafana-agent-0                                     2/2     Running
grafana-agent-logs-54dhp                            2/2     Running
kube-state-metrics-5595845f7c-8msdt                 1/1     Running
athena@owl:~$
```

Рисунок 4.2 — Стан подів

У неймспейсі monitoring успішно піднялися поди GrafanaCloud компоненту. Не завадить перевірити чи створився service, deployment та statefullset (рис 4.3).

```
athena@owl:~$ microk8s kubectl -n monitoring get po
NAME                                                READY   STATUS    RESTARTS        AGE
grafana-agent-operator-6564b87c8f-q7vw5             1/1     Running   143 (31h ago)   140d
grafana-agent-integrations-ds-rf2pd                 2/2     Running   283 (31h ago)   140d
kubernetes-event-exporter-6bc76f867d-j8j69          1/1     Running   147 (31h ago)   153d
grafana-agent-integrations-deploy-6d96dbd478-ppzbv  2/2     Running   282 (31h ago)   140d
grafana-agent-0                                     2/2     Running   293 (31h ago)   140d
grafana-agent-logs-54dhp                            2/2     Running   285 (31h ago)   140d
kube-state-metrics-5595845f7c-8msdt                 1/1     Running   145 (31h ago)   140d
athena@owl:~$ microk8s kubectl -n monitoring get svc
NAME                     TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)             AGE
kube-state-metrics       ClusterIP   None         <none>        8080/TCP,8081/TCP   140d
grafana-agent-operated   ClusterIP   None         <none>        8080/TCP            140d
athena@owl:~$ microk8s kubectl -n monitoring get deploy
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
grafana-agent-operator              1/1     1            1           140d
grafana-agent-integrations-deploy   1/1     1            1           140d
kubernetes-event-exporter           1/1     1            1           153d
kube-state-metrics                  1/1     1            1           140d
athena@owl:~$ microk8s kubectl -n monitoring get sts
NAME            READY   AGE
grafana-agent   1/1     140d
```

Рисунок 4.3 — Типи встановлених ресурсів у Kubernetes

На кластері використовується StorageClass, який створює локальне сховище на воркері (рис 4.4).

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"storage.k8s.io/
      v1","kind":"StorageClass","metadata":{"annotations":{"storageclass.kubernetes.io/
      is-default-class":"true"},"name":"microk8s-hostpath"},"provisioner":"microk8s.io/
      hostpath","reclaimPolicy":"Delete","volumeBindingMode":"WaitForFirstConsumer"}
    storageclass.kubernetes.io/is-default-class: "true"
  creationTimestamp: "2023-06-18T22:21:05Z"
  name: microk8s-hostpath
  resourceVersion: "1137982"
  uid: 7a427d8d-e422-4d39-9fc4-5f5cd64dae3e
provisioner: microk8s.io/hostpath
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

Рисунок 4.4 — Опис microk8s-hostpath StorageClass

Окремо, треба зазначити що на кластері попередньо має бути StorageClass, який дозволить динамічне створення місця для зберігання даних будь-яких сервісів.

Отже, моніторинг успішно встановлений, у наступному підрозділі відбувається перевірка отримання даних з Kubernetes кластеру.

## 4.2 Перевірка збору метрик та логів

Переходимо у Grafana та оглядаємо список панелей, які доступні (рис 4.5).


Рисунок 4.5 — Панелі у Grafana

Панелей дуже багато та вони розділені по секціям, перша секція пов’язана з Kubernetes інтеграцією, інша ж з станом самого воркеру та його операційною системою.

Переглянемо першу секцію, у ній цікавить в першу чергу як використовуються ресурси кластеру (рис 4.6).


Рисунок 4.6 — Використання CPU ресурсів сервісами

На вище зазначеному рисунку показано загальне поточне використання ресурсів Kubernetes кластеру, а також скільки ж сервіси дійсно потребують для нормальної роботи. Виходячи з результатів, кластер доволі гарно оптимізований. Не зайвим буде перевірити використання RAM, адже якщо існує bottleneck, то це дуже критичний випадок у роботі сервісів (рис 4.7).

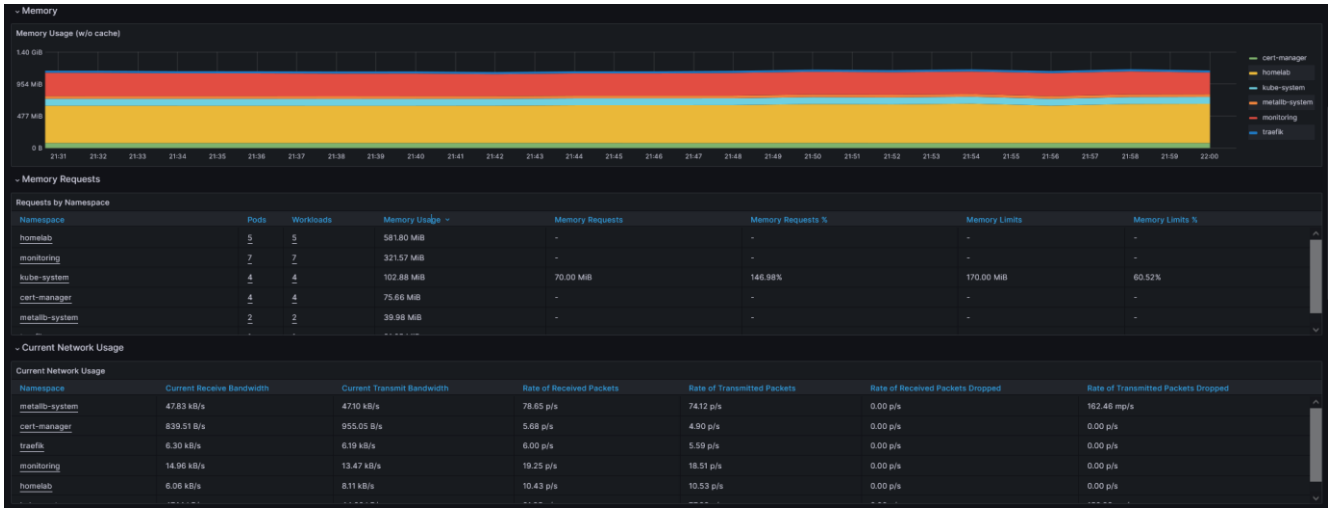



Рисунок 4.7 — Використання RAM ресурсів сервісами

Перейдемо до панелі безпосередньо з розрахунками ефективності кластеру, там отримаємо візуалізацію використання ресурсів процесора, оперативної пам’яті та локального сховища (4.8).

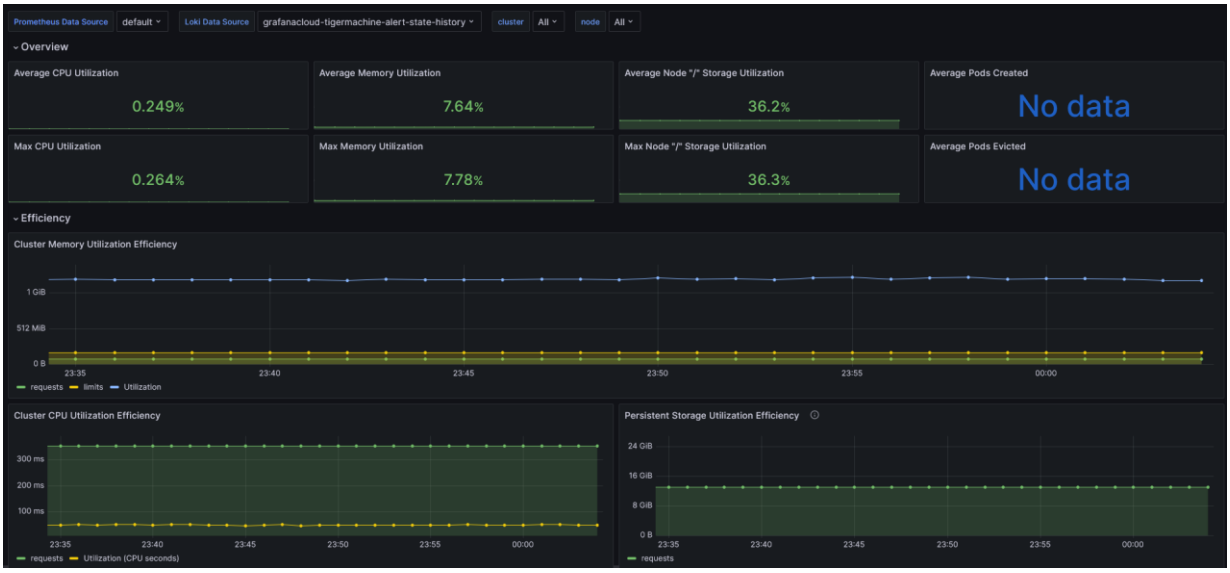



Рисунок 4.8 — Загальна ефективність використання ресурсів кластеру

На малюнку гарно видно, що кластер має необхідну кількість ресурсів для забезпечення стабільної роботи робочого навантаження.

Моніторинг логів сервісів та важливих компонентів кубернетіс кластеру є не менш пріоритетною задачею.

На рисунку 4.9 зображено логи одного з неймспейсів та сервісів, що працюють у ньому.

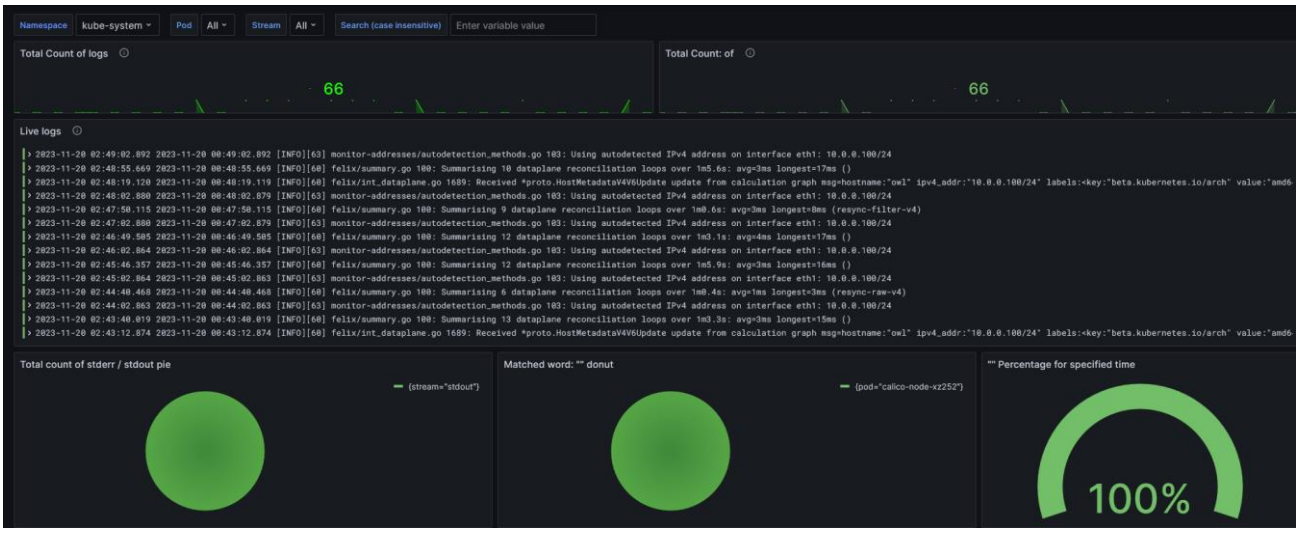



Рисунок 4.9 — Кластерні логи

Окрім того є можливість гнучкої фільтрації та відображення кількості згенерованих логів подами.

## 4.3 Верифікація роботи воркера

У даному підрозділі розглядаємо воркер, як віртуальну машину, на якій працює платформа Kubernetes.

Тобто цікавити буде стан машини в цілому з усіма працюючими процесами. Це потрібно для того, щоб перевірити чи дійсно ніщо не заважає стабільній роботі Kubernetes на виділених потужностях.

На рисунку 4.10 зображено основні загальні відомості про стан роботи віртуальної машини.

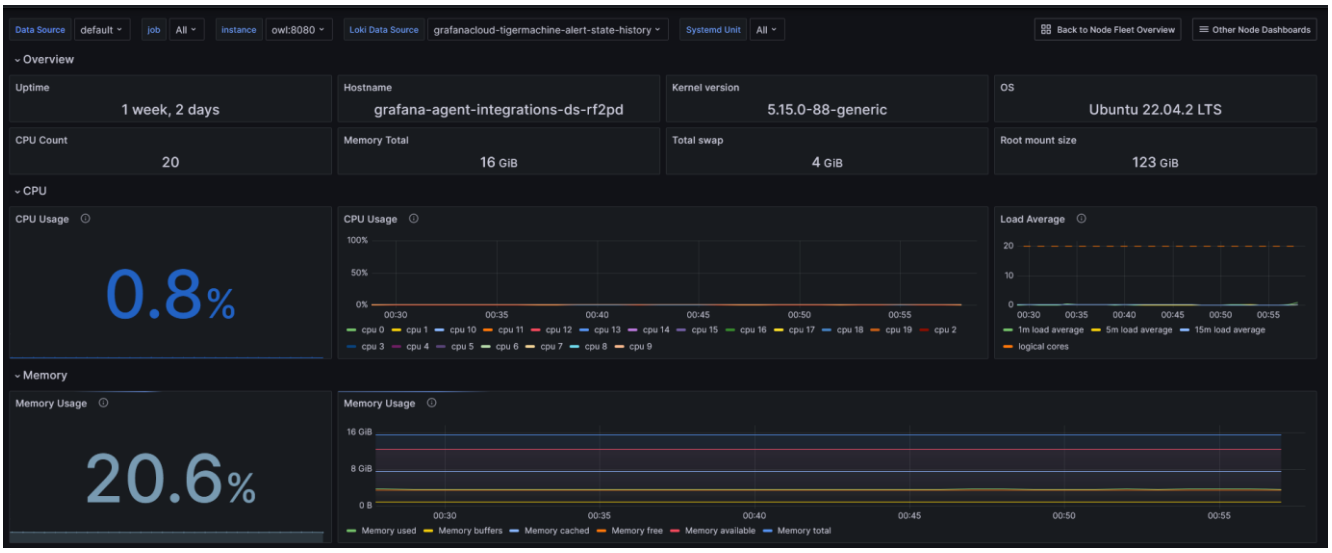



Рисунок 4.10 — Продуктивність машини

Вище зазначені головні метрики, такі як CPU, RAM, DISK I/O, OS. Актуальність операційної системи має велике значення для можливості оновлення версії кубернетіс кластеру.

Додатково, важливо перевірити стан мережі, адже якщо кластер складається з декількох нод і більше, то погана мережа може погіршити стан роботи кластеру (рис 4.11).


Рисунок 4.11 — Показники мережі

На рисунку можна побачити що існує невеликий error rate у роботі TCP протоколу, це не критично але потребує додаткової перевірки природи таких подій.

## 4.4 Верифікація роботи Kubernetes кластеру

У попередніх підрозділах була проаналізовано роботу сервісів, що працюють у кластері та стан віртуальної машини де кубернетіс розгорнутий.

Буде продемонстровано аналіз головних ознак, що власне сам Kubernetes, як платформа оркестрації контейнерів працює стабільно.

Для цього потрібно перевірити відомості про стан kubelet (рис 4.12).

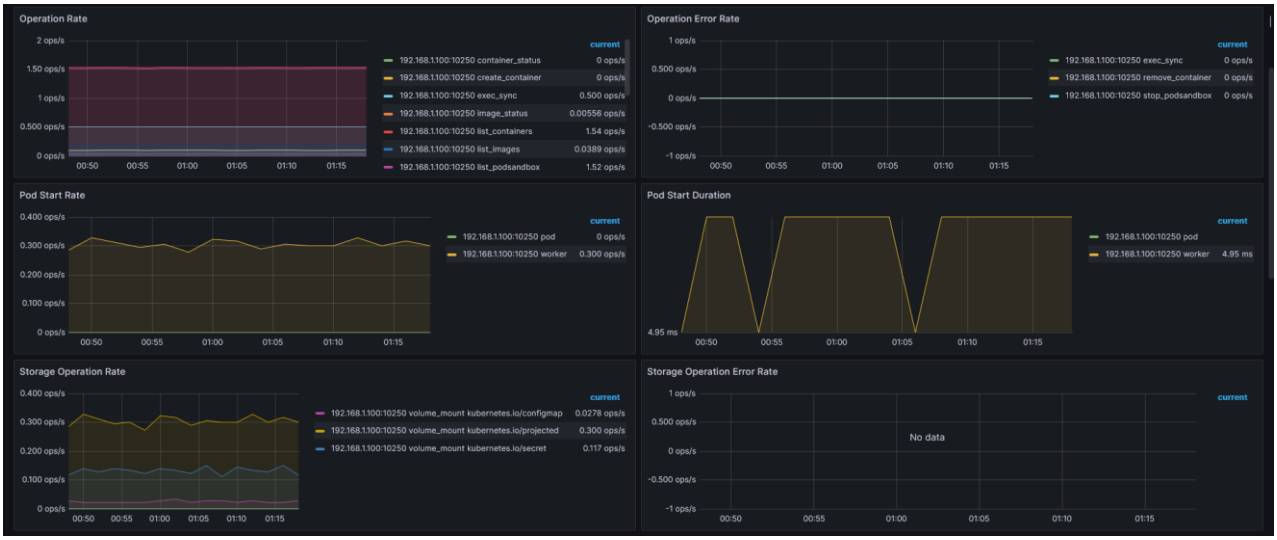



Рисунок 4.12 — Показники kubelet

З даного рисунку можна побачити, що контейнери на воркері успішно стартують, нема error rate. Поди на кластері стартують доволі швидко. Також є відомості про операції пов’язані зі сховищем але кількість операцій в секунду доволі низка, що означає що сервіси не часто звертаються до нього.

Отже проблеми з роботою кублета відсутні, тепер можна звернути увагу на події у кубернетіс кластері. Вони дозволяють побачити непрацюючі сервіси та компоненти.

Для цього відкриємо панель з Kubernetes Events (рис 4.13).


Рисунок 4.13 — Статистика проблемних подій

На цьому рисунку є детальна інформація про кількість проблем, що відбулася за певний проміжок часу. Виділені основні причини та враховано чи було збільшення їх кількості. Також є фільтрація по простору імен компоненту, типу та причинам  які звітують про наявність проблем.

У ходу аналізу був невеликий сплеск Warning подій, проте це було тимчасове явище і воно не відноситься до критичних проблем.

Окрім цього варто зазначити, що це візуалізація показників подій у кластері. Написане програмне забезпечення у третьому розділі роботи дозволяє згенерувати зручний репорт в yaml форматі (рис 4.14).

```yaml
homelab:
  BackOff:
    test-env:
      summary: 12
      seq-1: 1
      vault-2: 11
    summary: 12
  BackoffLimitExceeded:
    test-vault:
      summary: 1
      vault-etcd-bkp-1: 1
    summary: 1
```

Рисунок 4.14 — Згенерована статистика

Отже, стан роботи кластеру та сервісів що працюють на ньому, можна вважати задовільним. Моніторинг успішно встановлений та покриває з різних сторін оцінку продуктивності Kubernetes платформи.

# 5 ЕКОНОМІЧНА ЧАСТИНА

## 5.1 Оцінювання комерційного потенціалу розробки

Метою проведення технологічного аудиту є оцінювання комерційного потенціалу розробки, створеної в результаті науково-технічної діяльності.

Магістерська кваліфікаційна робота за темою “Система автоматизації моніторингу на платформі Kubernetes” передбачає розробку системи автомитизації, що встановлює моніторинг на Kubernetes платформу.

Ціна аналогу становить 92160 грн/місяць.

Проведемо оцінювання комерційного потенціалу даної розробки. Для проведення технологічного аудиту було залучено 3-х незалежних експертів: Захарченко Сергій Михайлович — керівник магістерської кваліфікаційної роботи, професор, викладач кафедри комп’ютерних наук; Кадук Олександр Володимирович, Крупельницький Леонід Віталієвич. В таблиці 5.1 наведено критерії оцінювання комерційного потенціалу розробки та їх оцінки в балах

Таблиця 5.1 — Критерії оцінювання комерційного потенціалу розробки

| Критерії оцінювання та бали (за 5-ти бальною шкалою) | | | | | |
|---|---|---|---|---|---|
| Кри-терій | 0 | 1 | 2 | 3 | 4 |
| Технічна здійсненність концепції: | | | | | |
| 1 | Достовірність концепції не підтверджена | Концепція підтверджена експертними висновками | Концепція підтверджена розрахунками | Концепція перевірена на практиці | Перевірено роботоздатність продукту в реальних умовах |

Продовження таблиці 5.1

| | | | | | |
|---|---|---|---|---|---|
| Ринкові переваги (недоліки): | | | | | |
| 2 | Багато аналогів на малому ринку | Мало аналогів на малому ринку | Кілька аналогів на великому ринку | Один аналог на великому ринку | Продукт не має аналогів на великому ринку |
| 3 | Ціна продукту значно вища за ціни аналогів | Ціна продукту дещо вища за ціни аналогів | Ціна продукту приблизно дорівнює цінам аналогів | Ціна продукту дещо нижче за ціни аналогів | Ціна продукту значно нижче за ціни аналогів |
| 4 | Технічні та споживчі властивості продукту значно гірші, ніж в аналогів | Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів | Технічні та споживчі властивості продукту на рівні аналогів | Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів | Технічні та споживчі властивості продукту значно кращі, ніж в аналогів |
| 5 | Експлуатаційні витрати значно вищі, ніж в аналогів | Експлуатаційні витрати дещо вищі, ніж в аналогів | Експлуатаційні витрати на рівні експлуатаційних витрат аналогів | Експлуатаційні витрати трохи нижчі, ніж в аналогів | Експлуатаційні витрати значно нижчі, ніж в аналогів |
| Ринкові перспективи | | | | | |
| 6 | Ринок малий і не має позитивної динаміки | Ринок малий, але має позитивну динаміку | Середній ринок з позитивною динамікою | Великий стабільний ринок | Великий ринок з позитивною динамікою |
| 7 | Активна конкуренція великих компаній на ринку | Активна конкуренція | Помірна конкуренція | Незначна конкуренція | Конкурентів немає |

Продовження таблиці 5.1

| Практична здійсненність | | | | | |
|---|---|---|---|---|---|
| 8 | Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї | Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців | Необхідне незначне навчання фахівців та збільшення їх штату | Необхідне незначне навчання фахівців | Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї |
| 9 | Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні | Потрібні незначні фінансові ресурси. Джерела фінансування відсутні | Потрібні значні фінансові ресурси. Джерела фінансування є | Потрібні незначні фінансові ресурси. Джерела фінансування є | Не потребує додаткового фінансування |
| 10 | Необхідна розробка нових матеріалів | Потрібні матеріали, що використовуються у військово-промисловому комплексі | Потрібні дорогі матеріали | Потрібні досяжні та дешеві матеріали | Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві |
| 11 | Термін реалізації ідеї більший за 10 років | Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років | Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років | Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років | Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років |

Продовження таблиці 5.1

| | | | | | |
|---|---|---|---|---|---|
| 12 | Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту | Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу | Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу | Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту | Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту |

Результати оцінювання комерційного потенціалу розробки потрібно звести в таблицю за зразком таблиці 5.2.

Таблиця 5.2 — Результати оцінювання комерційного потенціалу розробки

| Критерії | Прізвище, ініціали, посада експерта | | |
|---|---|---|---|
| | 1-Захарченко С.М. | 2-Кадук О.В. | 3-Крупельницький Л.В. |
| | Бали, виставлені експертами: | | |
| 1 | 4 | 4 | 3 |
| 2 | 4 | 3 | 4 |
| 3 | 4 | 3 | 4 |
| 4 | 4 | 3 | 3 |
| 5 | 3 | 3 | 3 |
| 6 | 4 | 4 | 4 |

Продовження таблиці 5.2

| | | | |
|---|---|---|---|
| 7 | 4 | 3 | 3 |
| 8 | 4 | 3 | 3 |
| 9 | 3 | 4 | 4 |
| 10 | 4 | 3 | 4 |
| 11 | 3 | 3 | 3 |
| 12 | 4 | 4 | 2 |
| Сума балів | $СБ_1$=45 | $СБ_2$=40 | $СБ_1$=40 |
| Середньоарифметична сума балів СБ | $\overline{СБ}\frac{\sum_1^3 СБ_i}{3} = \frac{125}{3} = 41.67$ | | |

За даними таблиці 5.2 можна зробити висновок, щодо рівня комерційного потенціалу розробки. Зважимо на результат й порівняємо його з рівнями комерційного потенціалу розробки, що представлено в таблиці 4.3.

Таблиця 5.3 — Рівні комерційного потенціалу розробки.

| Середньоарифметична сума балів СБ, розрахована на основі висновків експертів | Рівень комерційного потенціалу розробки |
|---|---|
| 0 –10 | Низький |

Продовження таблиці 5.3

| | |
|---|---|
| 11–20 | Нижче середнього |
| 21–30 | Середній |
| 31–40 | Вище середнього |
| 41–48 | Високий |

Рівень комерційного потенціалу розробки, становить 41,67 балів, що відповідає рівню «високий».

Сфера розробки моніторинг компонентів є досить розвинутою, оскільки в сучасному світі зростає затребуваність швидкого реагування на зміни в роботі інфраструктури. Таким чином автоматизація встановлення моніторингу на платформу Kubernetes є актуальна.

## 5.2 Прогнозування витрат на виконання наукової роботи та впровадження результатів

Проведемо прогнозування витрат на виконання науково-дослідної, дослідно-конструкторської та конструкторсько-технологічної роботи для розробки програмного забезпечення.

Виконаємо розрахунок витрат  приймаючи до уваги те, що для розробки інформаційної технології було залучено одного розробника програмного забезпечення. Основна заробітна плата кожного із розробників (дослідників) $З_о$, якщо вони працюють в наукових установах бюджетної сфери:

$$З_о = \frac{М}{T_p} \cdot t \text{ [грн]}, \quad (5.1)$$

де М — місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн;

$Т_р$ — число робочих днів в місяці; приблизно $Т_р$ = (21…23) дні;

t — число робочих днів роботи розробника (дослідника), розробка програмного забезпечення триває 80 днів.

Зроблені розрахунки внесені до таблиці 5.4:

Таблиця 5.4 — Основна заробітна плата розробників.

| Найменування посади виконавця | Місячний посадовий оклад, грн. | Оплата за робочий день, грн. | Число днів роботи | Витрати на оплату праці, грн. | Примітка |
|---|---|---|---|---|---|
| Програміст | 32000 | 1455.04 | 80 | 116403,2 | |
| Науковець | 16000 | 727,27 | 80 | 58181,76 | |
| Всього | | | | $\sum З_о$ | 174584,96 |

Додаткова заробітна плата $З_Д$ всіх розробників та робітників, які брали участь у виконанні даного етапу роботи, розраховується як (10…12%) від суми основної заробітної плати розробників та робітників розраховується за формулою:

$$З_Д = 0.10 \cdot 174584,96 = 17458,49(\text{грн}).$$

Нарахування на заробітну плату $Н_{ЗП}$ розробників та робітників, які брали участь у виконанні даного етапу роботи, розраховується за формулою:

$$Н_{зп} = (З_0 + З_Д) \cdot \frac{\beta}{100} \text{ [грн]}, \quad (5.2)$$

де $З_0$ — основна заробітна плата розробника, грн.;

$З_Д$ — додаткова заробітна плата розробника, грн.;

β — ставка єдиного внеску на загальнообов’язкове державне соціальне страхування — 22%.

$$Н_{ЗП} = (174584,96 + 17458,49) \cdot 0,22 = 42249,56(грн)$$

Амортизація обладнання, комп'ютерів та приміщень А, які використовувались під час (чи для) виконання даного етапу роботи.

Дані відрахування розраховують по кожному виду обладнання, приміщенням тощо.

У спрощеному вигляді амортизаційні відрахування А в цілому бути розраховані за формулою:

$$А = \frac{Ц \cdot Т}{12 \cdot Т_В}[грн], \quad (5.3)$$

де Ц — загальна балансова вартість всього обладнання, комп'ютерів, приміщень тощо, що використовувались для виконання даного етапу роботи, грн;

Т — фактична тривалість використання, міс;

$Т_В$— термін, використання обладнання, приміщень тощо, місяці, роки.

Зроблені розрахунки наведено в таблиці 5.4

Таблиця 5.5 — Амортизаційні відрахування

| Найменування | Балансова вартість, грн | Термін використання, роки | Фактична тривалість використання, міс. | Величина амортизаційних відрахувань, грн |
|---|---|---|---|---|
| Офісне приміщення | 500000 | 20 | 4 | 8333.33 |
| Ноутбук | 35000 | 2 | 4 | 5833.33 |
| Всього | | | | 14166.66 |

Під час розробки програмного продукту використовувались лише безкоштовні програмні засоби.

Витрати на силову електроенергію $В_е$ розраховуються за формулою:

$$В_е = В \cdot П \cdot Ф \cdot К_п \ [грн], \quad (5.4)$$

де В — вартість 1 кВт-год. електроенергії, 6.4 грн/кВт;

П — установлена потужність обладнання, кВт;

Ф — фактична кількість годин роботи обладнання, годин;

$К_п$ — коефіцієнт використання потужності.

Потужність використовуваного комп'ютера становить П=0.6 кВт.

Фактична кількість годин роботи обладнання — 640 год (80 робочих днів по 8 годин на день).

$$В_е = 6.4 \cdot 0{,}6 \cdot 640 \cdot 0{,}6 = 1428.48 \text{ (грн)}.$$

Сума всіх попередніх статей витрат дає витрати на виконання даної частини розділу роботи В.

$$В=174584{,}96 + 17458{,}49 + 42249{,}56+14166{,}66+1428{,}48 =249888{,}15$$

Розрахунок загальних витрат на виконання даної роботи.

$$В_{заг} = \frac{В}{\alpha} [грн], \quad (5.5)$$

$$В_{заг} = \frac{249888{,}15}{2} = 124944{,}07[грн],$$

Прогнозування загальних витрат на виконання та впровадження результатів виконаної роботи. Прогнозування витрат ЗВ на виконання та впровадження виконаної роботи здійснюється за формулою:

$$ЗВ = \frac{В_{заг}}{\beta} \text{ [грн]}, \quad (5.6)$$

де $\beta$ — коефіцієнт, який характеризує етап (стадію) виконання даної роботи.

Так, як розробка знаходиться:

— на стадії розробки дослідного зразка, то $\beta \approx 0{,}5$;

— на стадії технічного проектування , то $\beta \approx 0{,}2$;

— на стадії розробки конструкторської документації то $\beta \approx 0{,}3$;

— на стадії розробки технології, то $\beta \approx 0{,}4$;

— на стадії науково-дослідних робіт, то $\beta \approx 0{,}1$;

— на стадії промислового зразка, $\beta \approx 0{,}7$;

— на стадії впровадження, то $\beta \approx 0{,}9$.

$$ЗВ = \frac{124944{,}07}{0{,}7} = 178491{,}53 \text{ (грн)}.$$

Отже, прогноз загальних витрат на виконання та впровадження результатів становить 178491,53 грн.

## 5.3 Прогнозування комерційних ефектів від реалізації результатів розробки

У даному підрозділі проведемо кількісне прогнозування, яку вигоду, зиск можна отримати у майбутньому від впровадження результатів виконаної наукової роботи. В умовах ринку узагальнюючим позитивним результатом, що його отримує підприємство від впровадження результатів тієї чи іншої розробки, є збільшення чистого прибутку підприємства. Зростання чистого прибутку можна оцінити у теперішній вартості грошей.

Зростання чистого прибутку забезпечить підприємству надходження додаткових коштів, які дозволять покращити фінансові результати діяльності.

Виконання даної наукової роботи та впровадження її результатів складає приблизно 1 рік.

Позитивні результати від впровадження розробки очікуються на другий рік впровадження.

Проведемо детальніше прогнозування позитивних результатів та кількісне їх оцінювання по роках.

Обчислимо збільшення чистого прибутку підприємства $\Delta П_i$ для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки, розраховується за формулою:

$$\Delta П_я = \sum_1^n (\Delta П_я \cdot N + П_я \cdot \Delta N)_n \text{ [грн]}, \quad (5.7)$$

де $\Delta П_я$ — покращення основного якісного показника від впровадження результатів розробки у даному році;

N — основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN — покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки;

$П_я$ — основний якісний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n — кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки.

Припустимо, що внаслідок впровадження результатів наукової розробки покращується якість, що дозволяє підвищити ціну його реалізації на 200 грн, а кількість одиниць реалізованої послуги збільшиться: протягом першого року — на 0 од., протягом другого року — на 6000 од., протягом третього року — ще на 3500 од.

Орієнтовно: реалізація послуг до впровадження результатів наукової розробки складала 15 шт., а її ціна — 150рн.

Спрогнозуємо збільшення чистого прибутку підприємства від впровадження результатів наукової розробки у кожному році відносно базового.

Збільшення чистого прибутку підприємства $\Delta П_1$ протягом першого року складе:

$$\Delta П_1 = 0 \text{ (грн)}.$$

Обчислимо збільшення чистого прибутку підприємства $\Delta П_2$ протягом другого року:

$$\Delta П_2 = (15+150)\ (6000) = 990000 \text{ (грн)}.$$

Збільшення чистого прибутку підприємства $\Delta П_3$ протягом третього року становитиме:

$$\Delta П_3 = (15+150)\ \cdot (6000+3500) = 1567500 \text{ (грн)}.$$

Отже, розрахунки показують, що відповідно прогнозуванню комерційний ефект від впровадження розробки виражається у значному збільшенні чистого прибутку підприємства.

## 5.4 Визначення економічної доцільності фінансування наукової розробки

Основними показниками, які визначають доцільність фінансування наукової розробки певним інвестором, є абсолютна і відносна ефективність вкладених інвестицій та термін їх окупності.

Розрахунок ефективності вкладених інвестицій передбачає:

1-й крок — розрахунок теперішньої вартості інвестицій PV, що вкладаються в наукову розробку. Такою вартістю ми можемо вважати прогнозовану величину загальних витрат ЗВ на виконання та впровадження результатів НДДКР, тобто ЗВ = PV = 124944,07(грн).

2-й крок — розрахунок очікуваного збільшення прибутку $\Delta П_i$, що його отримає підприємство (організація) від впровадження результатів наукової

розробки, для кожного із років, починаючи з першого року впровадження проведено вище.

3-й крок — приведена вартість всіх чистих прибутків ПП розраховується за формулою:

$$ПП = \sum_{1}^{т} \frac{\Delta П_i}{(1+\tau)^t}, \quad (5.8)$$

де $\Delta П_i$ — збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т — період часу, протягом якого виявляються результати впровадженої НДДКР, роки;

$\tau$ — ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні - 0,1;

t — період часу (в роках) від моменту отримання чистого прибутку до точки „0”.

$$ПП = \frac{178491,53}{(1+0,1)^1} + \frac{0}{(1+0,1)^2} + \frac{990000}{(1+0,1)^3} + \frac{1567500}{(1+0,1)^4} = 197690,27(грн).$$

$$Е_{абс} = 197690,27 — 178491,53 = 1798198,74(грн).$$

Оскільки $Е_{абс} > 0$, результат від проведення наукових досліджень щодо розробки програмного продукту та їх впровадження принесе прибуток, тобто є доцільним, але це ще не свідчить про те, що інвестор буде зацікавлений у фінансуванні даної програми.

4-й крок — розраховують відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $Е_в$ за формулою:

$$E_в = \sqrt[T_ж]{1+\frac{E_{абс}}{PV}} - 1, \quad (5.9)$$

де $E_{абс}$ — абсолютна ефективність вкладених інвестицій, грн;

PV — теперішня вартість інвестицій PV = ЗВ, грн;

$T_ж$ — життєвий цикл наукової розробки, роки.

$$E_в = \sqrt[4]{1 + \frac{1798198,74}{178491,53}} - 1 = 0,972 \text{ або } 97,2\%$$

Порівняємо $E_в$ з мінімальною (бар'єрною) ставкою дисконтування $\tau_{мін}$, яка визначає ту мінімальну дохідність, нижче за яку інвестиції вкладатися не будуть.

Спрогнозуємо величину $\tau_{мін.}$ У загальному вигляді мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau = d + f, \quad (5.10)$$

де d — середньозважена ставка за депозитними операціями в комерційних банках; d = 0,14;

f — показник, що характеризує ризикованість вкладень; величина f = 0,3.

$$\tau = 0,14 + 0,3 = 0,44$$

Припустимо, що за даних умов прибуток буде збільшуватись, то у інвестора є потенційна зацікавленість у фінансуванні даної наукової розробки.

5-й крок — розраховують термін окупності вкладених у реалізацію наукового проекту інвестицій $T_{ок}$ за формулою:

$$T_{ок} = \frac{1}{E_в} \text{ [грн]}. \quad (5.11)$$

$$T_{ok} = \frac{1}{0,972} = 1.028 \quad \text{(роки)}.$$

Оскільки термін окупності вкладених у реалізацію наукового проекту інвестицій менше трьох років ( $T_{ок} < 3$ років), то фінансування нової розробки є доцільним.

## 5.5 Результати економічного аналізу

В даному розділі було здійснено оцінювання комерційного потенціалу розробки системи автоматизації моніторингу на платформі Kubernete

Проведено технологічний аудит з залученням трьох експертів. Аналіз експертних даних показав, що рівень комерційного потенціалу розробки вище середнього. Дослідження комерційного потенціалу розробки показав, що програмний продукт за своїми характеристиками випереджає аналогічні програмні продукти і є перспективною розробкою. Він має кращі функціональні показники, а тому є конкурентоспроможним товаром на ринку.

Згідно із розрахунками всіх статей витрат на виконання науково-дослідної, дослідно-конструкторської та конструкторсько-технологічної роботи загальна вартість витрат на розробку і впровадження складає 178491,53 грн.

Розрахована абсолютна ефективність вкладених інвестицій в сумі 1798198,74 грн свідчить про отримання прибутку інвестором від впровадження програмного продукту у діяльність підприємства.

Щорічна ефективність вкладених в наукову розробку інвестицій складає 97,2%, що вище за мінімальну бар'єрну ставку дисконтування, яка складає 44%. Це означає потенційну зацікавленість інвесторів у фінансуванні розробки.

Термін окупності складає 1,028 років, що також свідчить про доцільність фінансування.

Усе це, узяте разом, забезпечує прийняття рішення про доцільність виготовлення нового продукту.

# ВИСНОВКИ

У результаті аналізу архітектури платформи Kubernetes та моніторингових стеків виявлено важливі аспекти їхньої взаємодії. Проведена порівняльна характеристика моніторингових компонентів дозволила визначити їхні переваги та недоліки.

Запропонований метод автоматизації розгортання моніторингу, використовуючи інструменти оркестрації та збору релізу, виявився ефективним для уникнення людських помилок та прискорення налаштувань контролю за Kubernetes кластером.

Аналіз золотих сигналів моніторингу та використання GitOps підходу під час інсталяції компонентів підкреслює важливість структурованого та автоматизованого підходу до моніторингу.

Розроблена математична модель, заснована на теорії черг, визначає завантаженість Kubernetes кластеру, що важливо для ефективного масштабування та розподілу ресурсів.

Програмне забезпечення для аналізу подій Kubernetes кластеру та відокремлення шуму від цінної інформації покращує якість виведених даних про проблеми та сприяє швидшому реагуванню на них.

Розроблені системи автоматизації розгортання моніторингу та аналізу даних відображають сучасний підхід до управління Kubernetes кластерами, забезпечуючи ефективний та надійний контроль за їхнім станом.

**ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ**

1. Kubernetes Documentation | Kubernetes https://kubernetes.io/docs/

2. Kubernetes Concepts https://kubernetes.io/docs/concepts/

3. Getting Started with Kubernetes https://kubernetes.io/docs/setup/

4. Kubernetes API Reference https://kubernetes.io/docs/reference/

5. Kubernetes Glossary https://kubernetes.io/docs/reference/glossary/

6. Kubernetes Interactive Tutorials https://kubernetes.io/docs/tutorials/

7. Kubernetes Best Practices https://kubernetes.io/docs/setup/best-practices/

8. Kubernetes Security Best Practices https://kubernetes.io/docs/concepts/security/

9. Kubernetes Networking Concepts https://kubernetes.io/docs/concepts/cluster-administration/networking/

10. Kubernetes Networking Concepts https://kubernetes.io/docs/concepts/cluster-administration/networking/

11. Kubernetes API Overview https://kubernetes.io/docs/reference/kubernetes-api/

12. Kubernetes RBAC Role-Based Access Control https://kubernetes.io/docs/reference/access-authn-authz/rbac/

13. Kubernetes Ingress Controllers https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

14. Kubernetes Monitoring and Debugging https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application/

15. "6 Open Source Kubernetes Monitoring Tools You Should Know." https://lumigo.io/kubernetes-monitoring/6-open-source-kubernetes-monitoring-tools-you-should-know/

16. "8 Kubernetes Monitoring Tools: Head to Head."https://komodor.com/learn/8-kubernetes-monitoring-tools-head-to-head/

17. "How to Expose Your Services with Kubernetes Ingress | by Randal Kamradt Sr | BetterProgramming." https://betterprogramming.pub/how-to-expose-your-services-with-kubernetes-ingress-7f34eb6c9b5a

18. Monitoring Kubernetes Cluster with Prometheus and Grafana https://dev.to/vinayakmehta_/monitoring-kubernetes-cluster-with-prometheus-and-grafana-3p09

19. Kubernetes Monitoring Guide https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application/

20. Container log monitoring on Microk8s with Loki, Grafana and Promtail | Nucu Labs https://nuculabs.dev/2022/02/17/container-log-monitoring-on-microk8s-with-loki-grafana-and-promtail/#jp-carousel-2094

21. Smart Monitoring - Ai Outcome https://aioutcome.ai/smart-monitoring/

22. Monitoring Kubernetes Performance Metrics | Datadog https://www.datadoghq.com/blog/monitoring-kubernetes-performance-metrics/

23. VictoriaMetrics · The High Performance Open Source Time Series Database & Monitoring Solution https://docs.victoriametrics.com/

24. ContainIQ | Kubernetes Monitoring Instantly https://www.containiq.com/

25. Pull or Push: How to Select Monitoring Systems? - Alibaba Cloud Community https://www.alibabacloud.com/blog/pull-or-push-how-to-select-monitoring-systems_599007

26. Monitoring Kubernetes Architecture | InfluxData https://www.influxdata.com/blog/monitoring-kubernetes-architecture/

27. The Rise of Full-Stack Monitoring | by Lauren Detweiler | OpsMatters | Medium https://medium.com/opsmatters/the-rise-of-full-stack-monitoring-4397cb5140e5

28. Monitoring stack setup Part 1: Prometheus & Grafana | by Shishir Khandelwal | Medium https://shishirkh.medium.com/monitoring-stack-setup-part-1-prometheus-grafana-372e5ae25402

29. Ansible https://docs.ansible.com/ansible/latest/index.html

30. The 4 Golden Signals of Monitoring - Site24x7 https://www.site24x7.com/learn/4-golden-signals.html

# ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

проф., д.т.н.. Азаров О.Д..

“29” вересня 2023 р.

## ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи

“Система автоматизації моніторингу на платформі Kubernetes”
08-54.МКР.032.00.000.ТЗ

Науковий керівник: проф. каф.ОТ

_____ Захарченко С.М.

Студент групи 2КІ-22м

_____ Кулібабчук І.П.

ВНТУ 2023

1 Підстава для виконання магістерської кваліфікаційної роботи (МКР)

1.1 Важливим є актуальність дослідження у напрямку магістерської роботи, яка обумовлена тим, що Kubernetes став стандартом для розгортання, оркестрації та управління контейнеризованими додатками. Зростання популярності цієї платформи призводить до великого попиту на рішення, які полегшують моніторинг та управління цими середовищами. Дослідження в області систем автоматизації моніторингу на платформі Kubernetes визначається потребою розробки ефективних інструментів для підтримки сучасних технологічних інфраструктур та забезпечення надійності та продуктивності додатків.

1.2 Наказ про затвердження теми МКР.

2 Мета МКР і призначення розробки

2.1 Мета роботи — є прискорення та автоматичне покриття моніторингом створеної цифрової інфрастуктури на базі Kubernetes.

2.2 Призначення розробки — визначається необхідністю створення ефективного та автоматизованого інструменту для моніторингу і управління ресурсами в середовищі Kubernetes.

3 Вихідні дані для виконання МКР

3.1 Проведення аналізу існуючих методів та принципів;

3.2 Розробка програмного рішення системи автоматизації моніторингу на платформі Kubernetes;

3.4 Проведення верифікації та аналізу отриманих результатів

3.5 Виконання розрахунків для доведення доцільності нової розробки з економічної точки зору;

4 Вимоги до виконання МКР

Головна вимога — що система моніторингу повинна надавати повну інформацію про стан всіх компонентів, які працюють на платформі Kubernetes.

5 Етапи МКР та очікувані результати

Етапи роботи та очікувані результати приведено в Таблиці А.1.

Таблиця А.1 — Етапи МКР

| № етапу | Назва етапу | Термін виконання | | Очікувані результати |
|---|---|---|---|---|
| | | початок | кінець | |
| 1 | Огляд архітектури kubernetes та моніторингу її ключових компонентів | 18.09.2023 | 25.09.2023 | Аналітичний огляд джерел, задачі досліджень, розділ 1 |
| 2 | Технологія безперервної інтеграції та доставки | 28.09.2023 | 5.10.2023 | Розділ 2 |
| 3 | Розробка автоматизації доставки та розгортання моніторингу в Kubernetes | 5.10.2023 | 12.10.2023 | Розділ 3 |
| 4 | Підготовка економічної частини | 12.10.2023 | 19.10.2023 | Розділ 4 |
| 5 | Апробація та впровадження результатів дослідження | 18.10.2023 | 26.10.2023 | Тези доповідей |
| 6 | Оформлення пояснювальної записки, графічного матеріалу і презентації | 27.10.2023 | 2.11.2023 | ПЗ, графічний матеріал і презентація |

Продовження таблиці А.1

| | | | | |
|---|---|---|---|---|
| 7 | Підготовка і підпис супроводжуючих документів, нормоконтроль та тест на плагіат | 2.11.2023 | 10.11.2023 | Оформленні документи |

6 Матеріали, що подаються до захисту МКР

До захисту подаються: пояснювальна записка МКР, графічні і ілюстративні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до МКР українською та іноземною мовами.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання МКР

8.1 При оформлюванні МКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— ГОСТ 2.104–2006 «Єдина система конструкторської документації. Основні написи»;

— методичні вказівки до виконання магістерських кваліфікаційних робіт зі спеціальності 123 — «Комп’ютерна інженерія»;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання МКР викладено в «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ–03.02.02 П.001.01:21

## ДОДАТОК Б

Лістинг Ansible сценарію

```
---
- name: Deploy Grafana Cloud Manifest on Kubernetes
  hosts: kubernetes_master
  become: yes
  tasks:

    - name: Check if kubectl is installed
      command: kubectl version --client
      register: kubectl_result
      changed_when: false
      failed_when: "'Client Version' not in kubectl_result.stdout"

    - name: Set Kubernetes context (if necessary)
      # Ensure you have the right context or credentials here
      command: kubectl config use-context your-context-name
      changed_when: false

    - name: Check if manifest directory exists
      stat:
        path: "/path/to/jenkins/cloned/repo"
      register: repo_stat
      failed_when: not repo_stat.stat.exists

    - name: Apply Grafana Cloud manifest
      command: kubectl apply -f https://path-to-your-grafana-cloud-manifest.yaml
      register: apply_result
      changed_when: "'created' in apply_result.stdout or 'configured' in apply_result.stdout"
```

```
---
- name: Helm Chart Deployment
  hosts: kubernetes_master
  become: yes
  vars:
    helm_repo_name: "{{ helm_repo_name | default('my_default_repo_name') }}"
    helm_repo_url: "{{ helm_repo_url }}"
    helm_chart_name: "{{ helm_chart_name }}"
    helm_release_name: "{{ helm_release_name }}"
    helm_namespace: "{{ helm_namespace }}"
    helm_values_file: "{{ helm_values_file | default('') }}"

  tasks:
    - name: Check if Helm is installed
      command: helm version --client
      register: helm_check
      changed_when: false
      failed_when: "'Version:' not in helm_check.stdout"

    - name: Add Helm repository
      command: "helm repo add {{ helm_repo_name }} {{ helm_repo_url }}"
      when: helm_repo_url is defined
      changed_when: true

    - name: Update Helm repositories
      command: helm repo update
      changed_when: false

    - name: Install or upgrade Helm chart
      command: >
```

```
    helm upgrade --install {{ helm_release_name }} {{ helm_repo_name }}/{{ helm_chart_name }}
    --namespace {{ helm_namespace }}
    --create-namespace
    {{ '-f ' + helm_values_file if helm_values_file else '' }}
  register: helm_install
  changed_when: "'Release \"' + helm_release_name + '\" has been upgraded' in helm_install.stdout or 'Release \"' + helm_release_name + '\" does not exist. Installing it now.' in helm_install.stdout"
```

## ДОДАТОК В

Лістинг Jenkins пайплайну

```
pipeline {
    agent any

    parameters {
        string(name: 'helm_repo', defaultValue: 'my_repo', description: 'Helm Repository Name')
        string(name: 'repo_url', defaultValue: 'https://example.com/charts', description: 'Helm Repository URL')
        string(name: 'chart_name', defaultValue: 'my_chart', description: 'Name of the Helm Chart')
        string(name: 'release_name', defaultValue: 'my_release', description: 'Helm Release Name')
        string(name: 'namespace', defaultValue: 'default', description: 'Kubernetes Namespace')
    }
    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }
        stage('Deploy Helm Chart') {
            steps {
                sh """
                ansible-playbook helm_deploy.yml -e helm_repo=${params.helm_repo} -e repo_url=${params.repo_url} -e chart_name=${params.chart_name} -e release_name=${params.release_name} -e namespace=${params.namespace}
                """
```

```
            }
        }
    }
}

pipeline {
    agent any

    stages {
        stage('Deploy Kubernetes monitoring using Ansible') {
            steps {
                script {
                    ansiblePlaybook(
                        playbook: 'path-to-playbook.yml',
                        inventory: 'path-to-inventory.ini',
                        credentialsId: 'ansible-ssh-credentials'
                    )
                }
            }
        }
    }
```

## ДОДАТОК Г

Лістинг програми

```
import argparse

import yaml

from opensearchpy import OpenSearch


client = OpenSearch("https://user:pass@host:9200", use_ssl=True, verify_certs=False,

                    ssl_show_warn=False)


general_dict = {}


core_component_reasons = ['Evicted', 'Failed', 'FailedScheduling', 'ImageGCFailed',
'InternalError', 'NodeNotReady', 'NodeStartFailure', 'PolicyViolation',
'ProvisioningFailed', 'Rebooted', 'SystemOOM', 'Unexpected', 'UpdateFailed']

env_reasons = ['BackOff', 'BackoffLimitExceeded', 'Unhealthy']



def yaml_out(cluster):

    with open(r"+cluster+'-alerts.yaml', 'w') as file:

        yaml.safe_dump(general_dict, file)

        print(cluster + "-alerts.yaml created")



def clusters(time_range):

    query_body = {

        "query": {"bool": {"filter": [{"range": {"@timestamp": {"gte": "now-
"+time_range, "lte": "now"}}},
```

```
                        {"query_string": {"query": "(type.keyword=\"Warning\")"}}]}},
    "size": 0,
    "aggs": {
      "cluster": {
        "terms": {
          "field": "cluster.keyword",
          "size": 1000,
          "order": {"_count": "desc"}
        }
      }
    },
  }
  resp = client.search(index="homelab-*", body=query_body)

  clusters = []
  print("Top clusters by warnings:")
  for key in resp['aggregations']['cluster']['buckets']:
    print(key['key'], key['doc_count'])
    clusters.append(key['key'])

  return clusters


def reasons(cluster, time_range, namespace_query_size, name_query_size):
  query_body = {
```

```
        "query": {"bool": {"filter": [{"range": {"@timestamp": {"gte": "now-
"+time_range, "lte": "now"}}}, {
            "query_string": {"query": "cluster.keyword=(\"" + cluster + "\") AND
(type.keyword=\"Warning\")"}}]}},
        "size": 0,
        "aggs": {
            "reason": {
                "terms": {
                    "field": "reason.keyword",
                    "size": 1000,
                    "order": {"_count": "desc"}
                }
            }
        },
    }
    resp = client.search(index="homelab-*", body=query_body)
    reason_list = []
    for key in resp['aggregations']['reason']['buckets']:
        reason_list.append(key['key'])
    top_namespaces_by_reason(cluster, reason_list, namespace_query_size, time_range,
name_query_size)



def namespaces(cluster, reasons, namespace_query_size, time_range,
name_query_size):
    document = {cluster: {}}
    global general_dict
```

```
    if reasons == ["core"]:
        reason_list = core_component_reasons
    elif reasons == ["env"]:
        reason_list = env_reasons
    else:
        reason_list = reasons
    for reason in reason_list:
        query_body = {
            "query": {
                "bool": {"filter": [{"range": {"@timestamp": {"gte": "now-"+time_range,
"lte": "now"}}}, {"query_string": {
                    "query": "cluster.keyword=(\"" + cluster + "\") AND
(type.keyword=\"Warning\") AND ("
                                                    "reason.keyword=\"" + reason + "\")"}}]}},
            "size": 0,
            "aggs": {
                "namespace": {
                    "terms": {
                        "field": "namespace.keyword",
                        "size": namespace_query_size,
                        "order": {"_count": "desc"}
                    }
                }
            },
        }
        resp = client.search(index="homelab-*", body=query_body)
```

```
        if resp['hits']['total']['value']!=0:
            general_dict = {cluster: {reason: {}}}
            document[cluster][reason] = {}
        for key in resp['aggregations']['namespace']['buckets']:
            if not key['key']:
                document [cluster][reason]['empty'] = {}
            else:
                document [cluster][reason][key['key']] = {}
            top_names_by_namespace(cluster, key['key'], reason, name_query_size,
document)
    yaml_output(cluster)



def names(cluster, namespace, reason, name_query_size, document):
    global general_dict
    check_namespace = 0
    if not namespace:
        check_namespace = 1
    query_body = {
        "query": {"bool": {"filter": [{"range": {"@timestamp": {"gte": "now-"+
time_range, "lte": "now"}}}, {"query_string": {
            "query": "cluster.keyword=(\"" + cluster + "\") AND
(type.keyword=\"Warning\") AND (reason.keyword=\"" + reason + "\") AND
(namespace.keyword=\"" + namespace + "\")"}}]}},
        "size": 0,
        "aggs": {
            "name": {
```

```
            "terms": {
                "field": "name.keyword",
                "size": name_query_size,
                "order": {"_count": "desc"}
            }
        }
    },
}
    resp = client.search(index="homelab-*", body=query_body)
    for key in resp['aggregations']['name']['buckets']:
        name = key['key']
        warns = key['doc_count']
        if not name:
            name = 'empty'
        if check_namespace == 1:
            document[cluster][reason]['empty'][name] = warns
        elif check_namespace == 0:
            document[cluster][reason][namespace][name] = warns
    general_dict = {**general_dict, **document}


def main(cluster, namespace_query_size, reasons, name_query_size, time_range):
    if type(reasons) == list:
        reasons.sort()
    if cluster == "all":
```

```
        cluster_list = top_clusters_by_warnings(time_range)
    else:
        cluster_list = cluster
    for cluster_name in cluster_list:
        if reasons == "all":
            top_warning_reasons_by_cluster(cluster_name, time_range, name_query_size, name_query_size)
        else:
            top_namespaces_by_reason(cluster_name, reasons, namespace_query_size, time_range, name_query_size)



if name == '__main__':
    parser = argparse.ArgumentParser()

    parser.add_argument('-c', '--cluster', nargs='*', type=str, default="all")
    parser.add_argument('-ns', '--namespace_query_size', type=int, default=5)
    parser.add_argument('-r', '--reason', nargs='*', type=str, default="all")
    parser.add_argument('-nsq', '--name_query_size', type=int, default=5)
    parser.add_argument('-t', '--time_range', type=str, default="30d")

    args = parser.parse_args()
    cluster, namespace_query_size, reason, name_query_size, time_range = args.cluster, args.namespace_query_size, args.reason, args.name_query_size, args.time_range
    main(cluster, namespace_query_size, reason, name_query_size, time_range)
```

# ДОДАТОК Д

Графік масштабування ресурсів за розподілом Гаусса

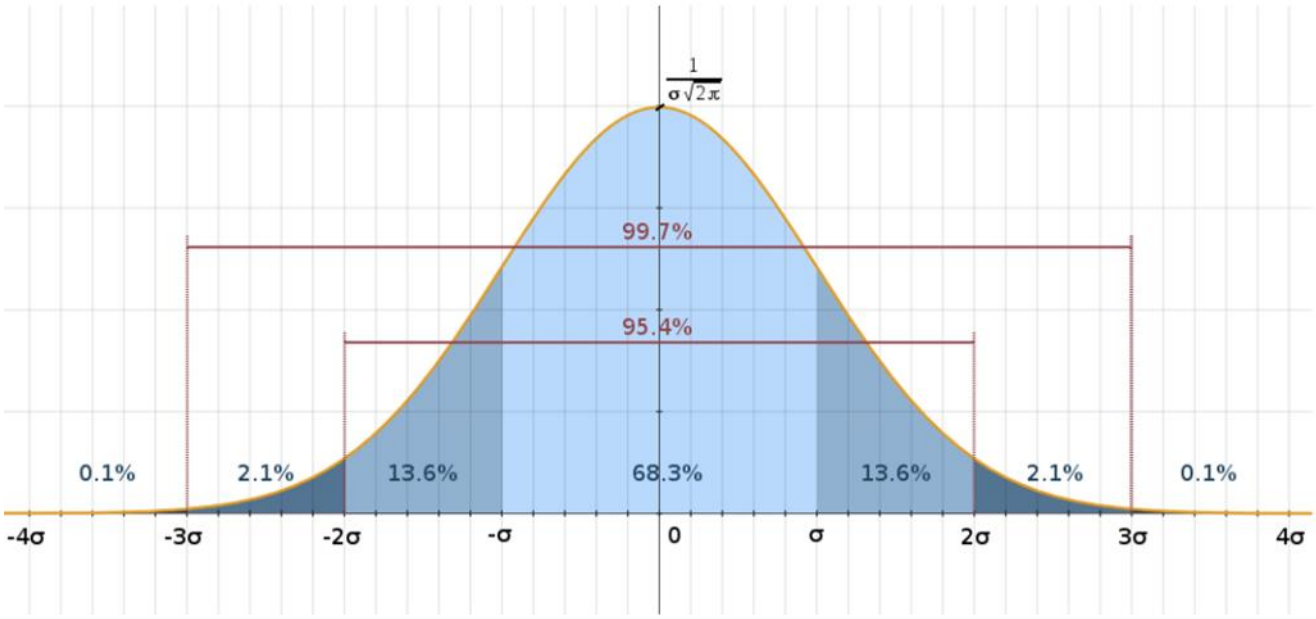



Рисунок Д.1 — Графік масштабування ресурсів за розподілом Гаусса

# ДОДАТОК Е

Графік масштабування ресурсів за сумуванням рядів формули Рімана

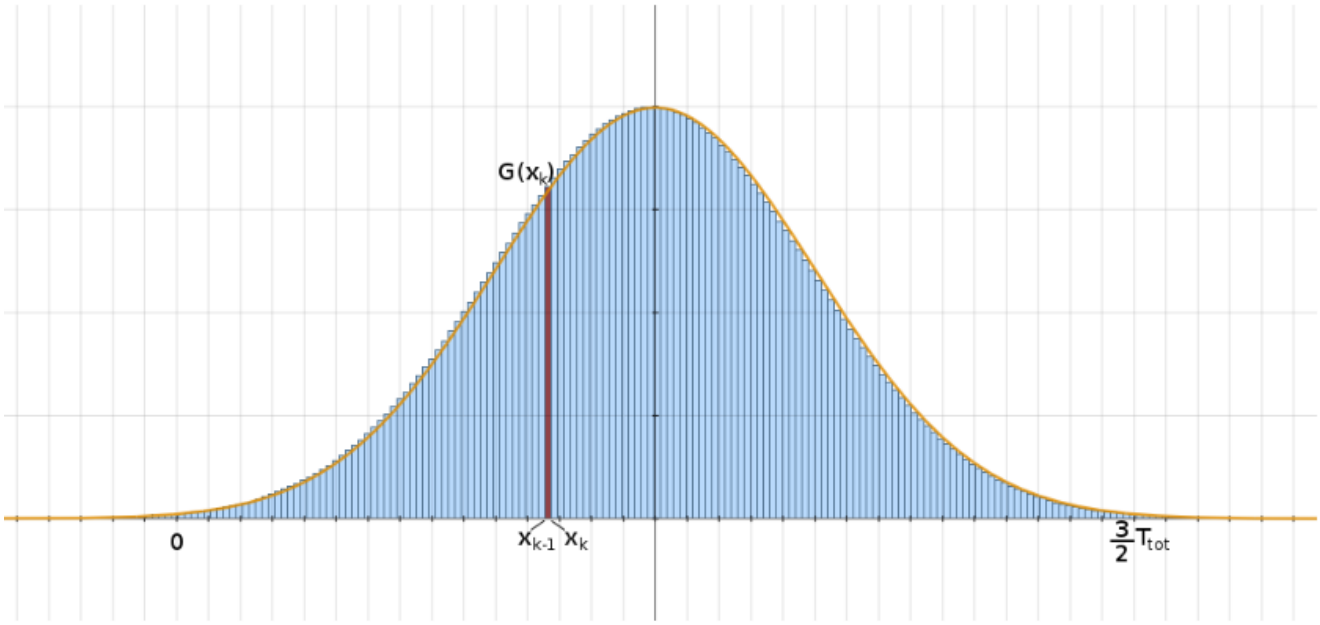



Рисунок Е.1 — Графік масштабування ресурсів за сумуванням рядів формули Рімана

# ДОДАТОК Ж

Схема архітектури розгортання системи

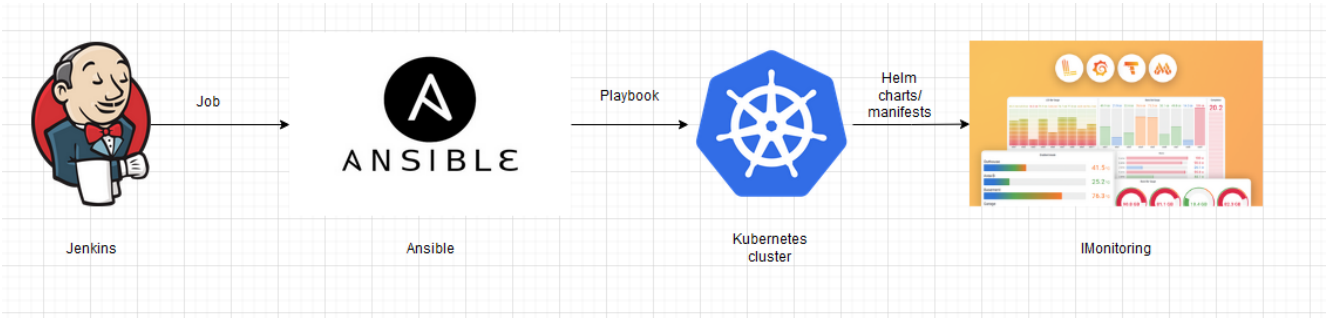



Рисунок Ж.1 — Схема архітектури розгортання системи

# ДОДАТОК И

Графік ефективності Kubernetes кластеру


Рисунок И.1 — Графік ефективності Kubernetes кластеру

# ДОДАТОК К

Графік подій у Kubernetes кластері

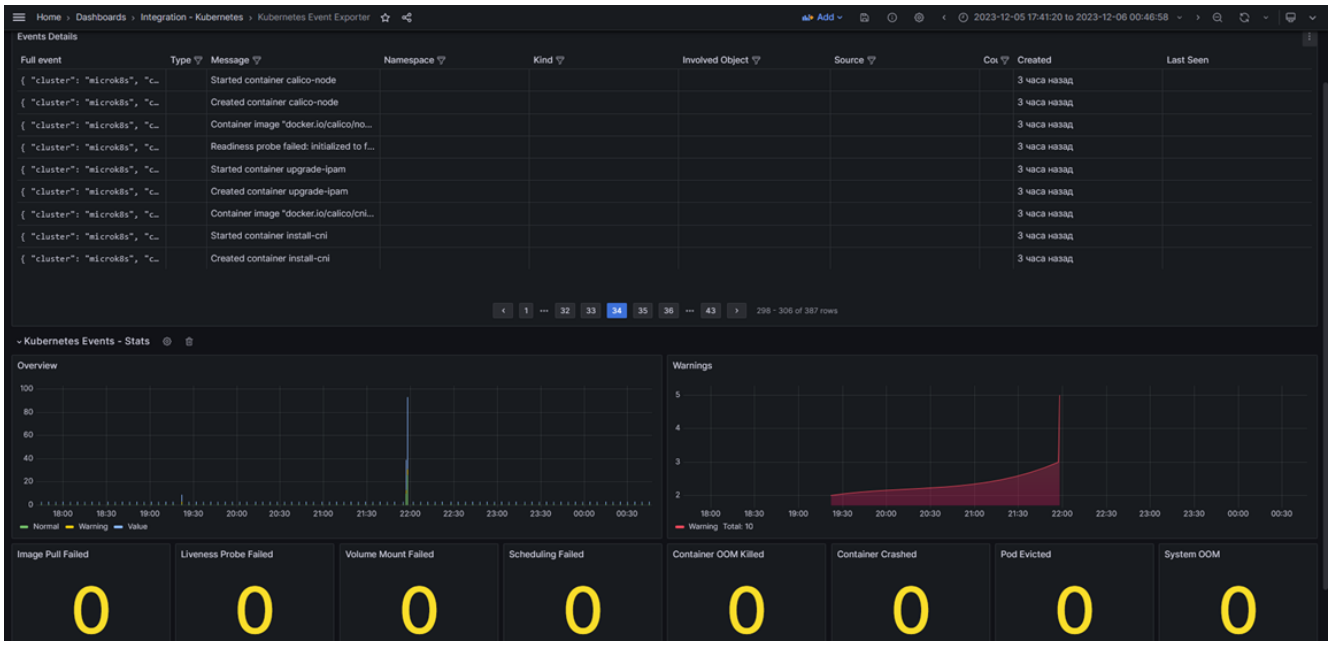



Рисунок К.1 — Графік подій у Kubernetes кластері

# ДОДАТОК Л

Протокол перевірки кваліфікаційної роботи

## ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи: **Система автоматизації моніторингу на платформі Kubernetes.**

Тип роботи: кваліфікаційна робота

Підрозділ: кафедра обчислювальної техніки, ФІТКІ, 2–КІ–22м

Науковий керівник: проф. Захарченко С.М.

| **Unicheck** | |
|---|---|
| **Оригінальність** | **96.8%** |
| Схожість | 3.2% |

**Аналіз звіту подібності**

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

-

Заявляю, що ознайомлений з повним звітом подібності, який був згенерований Системою щодо роботи «Система автоматизації моніторингу на платформі Kubernetes».

Автор _____ Кулібабчук І.П.

(підпис) (прізвище, ініціали)

Опис прийнятого рішення: **допустити до захисту**

Особа, відповідальна за перевірку _____ Захарченко С.М.

(підпис) (прізвище, ініціали)

Експерт _____ _____

(за потреби) (підпис) (прізвище, ініціали, посада)