НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО
В.о. завідувача кафедри
_____Олександр КОВАЛЬ
«_____»_____2025р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»
спеціальності 121 «Інженерія програмного забезпечення»
на тему: «Впровадження CI/CD інфраструктури на базі GitLab з автоматизованими механізмами моніторингу, логування та аварійного відновлення»

Виконав:
Студент IV курсу, групи ТВ-12
Крутиголова Володимир Олегович _____
(підпис)

Керівник:
асистент Голець В.О. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент:

_____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
«____»_____2025р.

ЗАВДАННЯ
на дипломну роботу студенту

_____Крутиголові Володимиру Олеговичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи
«Впровадження CI/CD інфраструктури на базі GitLab з автоматизованими механізмами моніторингу, логування та аварійного відновлення»_____
керівник роботи асистент ____Голець Владислав Олександрович_____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "_2_" _червня__ 2025 року №_1875-с_
2. Строк подання студентом роботи _9 червня 2025 р._____
3. Вихідні дані до роботи: мова розмітки YAML, інфраструктурний фреймворк Terraform, система контролю версій GitLab, реверс проксі Apache, система віртуалізації ESXi, система моніторингу Prometheus, платформа візуалізації Grafana, стек логування OpenSearch + FileBeat, середовище розробки VS Code.
4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): Налаштування CI/CD через GitLab, збір та обробка телеметрії, візуальний моніторинг, централізоване логування подій і збоїв, сценарії аварійного відновлення у віртуалізованому середовищі.
5. Перелік ілюстративного матеріалу: приклад користувацького інтерфейсу, приклад директорій, приклад налаштувань_____

6. Дата видачі завдання «28» жовтня___ 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	28.10.2024	виконано
2	Дослідження предметної області	15.03 - 31.03.2025	виконано
3	Постановка вимог до проектування системи	1.04 - 7.04.2025	виконано
4	Дослідження існуючих рішень	7.04 - 17.04.2025	виконано
5	Розробка програмного продукту	14.04 - 16.05.2025	виконано
6	Тестування	12.05 - 16.05.2025	виконано
7	Захист програмного продукту	15.05.2025	виконано
8	Оформлення дипломної роботи	20.05 – 30.05.2025	виконано
9	Передзахист	5.06.2025	виконано
10	Захист	18.06.2025	виконано

Студент _____ Володимир Крутиголова
(підпис) (ім'я, прізвище)

Керівник роботи _____ Владислав Голець
(підпис) (ім'я, прізвище)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 64 сторінок, 31 рисунок, хх додатків та 10 посилань.

Метою роботи було впровадження автоматизованої CI/CD-інфраструктури із вбудованими системами збору метрик, централізованого журналювання подій і механізмами швидкого відновлення працездатності після збоїв.

У процесі виконання проекту були реалізовані наступні кроки:

- здійснено огляд і порівняння існуючих платформ для CI/CD та суміжних сервісів;
- створено інфраструктурну модель з використанням принципів Infrastructure as Code;
- налаштовано конвеєри збірки, тестування та розгортання із застосуванням GitLab Runner;
- інтегровано інструменти моніторингу (Prometheus, Grafana) з повідомленнями про інциденти;
- забезпечено агрегування логів через OpenSearch із можливістю гнучкого аналізу;
- реалізовано періодичне резервування стану системи із подальшим відновленням через vCenter.

У результаті розробки отримаємо ефективну систему, орієнтовану на стабільну та керовану розробку, що дозволяє не лише автоматизувати ключові етапи життєвого циклу ПЗ, а й своєчасно реагувати на збої в інфраструктурі.

Практичне значення одержаних результатів полягає в формуванні гнучкої DevOps-інфраструктури, здатної до масштабування, автоматичного відновлення після критичних подій та підтримки повного циклу розробки програмного забезпечення з мінімальним втручанням адміністратора.

Ключові слова: DevOps, автоматизація, CI/CD, GitLab, Terraform, моніторинг, логування, резервне копіювання, відновлення системи.

ABSTRACT

Structure and scope of the thesis. The work contains 64 pages, 31 figures, xx appendices and 10 references.

The purpose of the work is to implement an automated CI/CD infrastructure with built-in metrics collection systems, centralized event logging and mechanisms for rapid recovery after failures.

During the project, the following steps were implemented:

- a review and comparison of existing platforms for CI/CD and related services was carried out;
- an infrastructure model was created using the Infrastructure as Code principles;
- building, testing and deployment pipelines were configured using GitLab Runner;
- monitoring tools (Prometheus, Grafana) were integrated with incident reports;
- log aggregation via OpenSearch was provided with the possibility of flexible analysis;
- periodic backup of the system state with subsequent recovery via vCenter was implemented.

As a result of the development, we will obtain an effective system focused on stable and managed development, which allows not only to automate key stages of the software life cycle, but also to respond to infrastructure failures in a timely manner.

The practical significance of the obtained results is the formation of a flexible DevOps infrastructure capable of scaling, automatic recovery after critical events, and support for the full software development cycle with minimal administrator intervention.

Keywords: DevOps, automation, CI/CD, GitLab, Terraform, monitoring, logging, backup, system recovery.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

API (англ. Application Programming Interface) — інтерфейс прикладного програмування; набір функцій для взаємодії між компонентами програмного забезпечення.

CD (англ. Continuous Delivery) — безперервне доставлення змін у програмному забезпеченні.

CE (англ. Community Edition) — відкрита версія програмного забезпечення з базовими можливостями.

CI (англ. Continuous Integration) — безперервна інтеграція змін у програмному забезпеченні.

CI/CD — сукупність практик безперервної інтеграції та доставки.

CPU (англ. Central Processing Unit) — центральний процесор.

DNS (англ. Domain Name System) — система доменних імен.

Docker — платформа для контейнеризації застосунків.

ELK (англ. Elasticsearch, Logstash, Kibana) — стек інструментів для логування та аналітики (альтернатива OpenSearch).

ERROR — тип повідомлення в логах, що означає помилку.

ESXi — гіпервізор компанії VMware для віртуалізації серверів.

Filebeat — легкий агент для збору логів, що передає їх у Logstash або OpenSearch.

Git — система керування версіями.

GitLab — система контролю версій з підтримкою CI/CD.

GitLab Runner — агент, який виконує завдання CI/CD пайплайнів GitLab.

Grafana — система візуалізації метрик та створення дашбордів.

ID — ідентифікатор.

JSON (англ. JavaScript Object Notation) — формат обміну даними.

Logstash — компонент збору, обробки та трансформації логів.

OpenSearch — пошукова система і база для зберігання логів та аналітики.

Pipeline — набір автоматизованих етапів виконання у CI/CD.

Prometheus — система збору та зберігання метрик.

RAM *(*англ. Random Access Memory*)* — оперативна пам’ять.

Runner — агент, який виконує jobs у CI/CD.

SSH *(*англ. Secure Shell*)* — протокол безпечного доступу до віддалених машин.

Snapshot — моментальний знімок стану віртуальної машини.

Terraform — інструмент для опису інфраструктури як коду.

VM *(*англ. Virtual Machine*)* — віртуальна машина.

YAML *(*англ. YAML Ain’t Markup Language*)* — мова розмітки для конфігурацій.

gitlab-ci.yml — конфігураційний файл CI/CD для GitLab.

vCenter — центральний інструмент керування VMware-інфраструктурою.

ВСТУП

У сучасному цифровому світі вимоги до швидкості випуску оновлень і одночасного забезпечення високої якості програмного забезпечення постійно зростають. Компанії прагнуть скоротити час від написання коду до попадання його в робоче середовище, водночас мінімізувати ризики та збої в роботі кінцевого продукту. Саме із цим завданням успішно справляється підхід DevOps, який передбачає тісну взаємодію розробників, тестувальників та фахівців з експлуатації інфраструктури. Завдяки єдиному потоку процесів — від планування й написання коду до його доставки і підтримки — досягається синергія між командами, знижується кількість конфліктів і прихованих помилок, а також підвищується прозорість робочих процесів.

Однією з ключових практик DevOps є безперервна інтеграція (CI) та безперервна доставка (CD). При їх впровадженні кожен внесений розробником коміт одразу відправляється у централізовану систему контролю версій, де активується автоматизований конвеєр. Цей конвеєр виконує низку послідовних кроків: збірку програмних артефактів, запуск юніт- та інтеграційних тестів, статичний аналіз коду, а також перевірку на відповідність внутрішнім стандартам безпеки та якості. Лише після успішного проходження всіх перевірок артефакти стають доступними для подальшої доставки в підготовлене середовище тестування або безпосередньо в продакшен, якщо цього вимагає політика випуску. Такий підхід дозволяє значно скоротити час на виявлення та усунення дефектів, адже проблеми локалізуються одразу, не накопичуючись у кілька релізів.

Проте автоматизовані збірки та деплой — це лише основа стабільності. Для підтримки роботи системи на високому рівні необхідно впроваджувати рішення для моніторингу показників працездатності та централізованого логування. Збір часових метрик — навантаження на сервіси, використання ресурсів, час відгуку — дозволяє формувати точні запити на виявлення аномалій і запускати оповіщення при перевищенні встановлених порогів. Водночас система агрегації логів збирає події з різних компонентів, класифікує їх за рівнем критичності та надає інтерфейс

для швидкого пошуку й аналізу причин збоїв. Інтерактивні панелі з графіками та гістограмами допомагають інженерам реагувати на інциденти в режимі реального часу та оптимізувати продуктивність додатків.

Не менш важливим етапом є забезпечення безперервності бізнес-процесів у випадку критичних збоїв: впровадження стратегій резервного копіювання й аварійного відновлення. Регулярне створення знімків віртуальних середовищ або бекапів ключових сервісів дозволяє швидко повернути систему до працездатного стану після апаратних чи програмних відмов. Детально описані процедури відновлення та періодичне тестування «відкатів» гарантують, що навіть у найскладніших сценаріях простої будуть мінімальними, а втрата даних — непомітною для кінцевого користувача.

Поєднуючи безперервну інтеграцію та доставку, цілодобовий моніторинг і централізоване логування з продуманими механізмами відновлення, організації отримують змогу не лише швидко випускати нові релізи, а й підтримувати високу якість і доступність сервісів. Такий комплексний підхід лежить в основі сучасних DevOps-культур і стає запорукою успіху будь-якого програмного проекту.

1 МЕТА І ЗАВДАННЯ РОБОТИ

1.1 Формулювання загальної мети роботи

У сучасних умовах цифрової трансформації все частіше виникає необхідність у системах, здатних працювати самостійно — стабільно, передбачувано, без втручання оператора. Якщо раніше акцент робився на функціональність програмного продукту, то сьогодні набагато важливішим стає його надійний супровід. І тут ідеться не лише про технічну підтримку, а про повноцінну інфраструктуру, яка може не просто існувати поряд із кодом, а бути його частиною.

Саме на побудову такої інфраструктури і спрямовано цю роботу. В її основі лежить ідея створити систему, яка могла б забезпечити безперервний супровід програмного забезпечення після етапу розробки. Мова йде не про один компонент чи окремий інструмент, а про комплексне рішення, яке охоплює одразу кілька ключових напрямів: спостереження за станом системи, централізований збір та обробку подій, а також забезпечення стабільності шляхом періодичного резервування і відновлення.

Платформа, яку планується реалізувати, повинна не просто фіксувати технічні збої, а й давати можливість їх розпізнавати завчасно. Це означає наявність механізмів для збору телеметрії — цифрових відбитків стану системи, які можуть слугувати основою для прийняття рішень або запуску реакцій. Ідеально, коли така система сама знає, коли щось пішло не так. І знає, як це виправити.

Пріоритетом також є зменшення залежності від людського фактору. Підхід, що закладається в основу, передбачає мінімізацію ручної роботи на кожному етапі — від налаштування до повсякденної експлуатації. В ідеалі користувач не повинен навіть знати, що система виконує якісь дії: усе має працювати прозоро, у фоні, без постійної взаємодії.

Водночас структура рішення має бути гнучкою. Змінність вимог, оновлення окремих компонентів або зміна логіки роботи не повинні призводити до потреби

переробляти всю систему. Це вимагає чіткого поділу обов’язків між складовими, модульності та логічної ізоляції функцій.

Мета, яка ставиться у межах цієї роботи, охоплює не лише створення окремих інструментів — важливо зібрати їх в одне ціле. Так, щоб кожна частина знала свою роль, могла взаємодіяти з іншими, але залишалася автономною настільки, наскільки це потрібно для стійкої роботи в умовах збою.

1.2 Система моніторингу стану інфраструктури

Жодна система не може вважатися повноцінною, якщо вона не знає, що з нею відбувається. Моніторинг — це не просто діагностика, це здатність бачити себе зсередини в режимі реального часу. Саме тому одним із базових напрямів розробки стало впровадження такої підсистеми, яка могла б забезпечити збір, фіксацію та подання технічного стану інфраструктури у зручному для інтерпретації вигляді.

Моніторинг дозволяє виявляти проблеми ще до того, як вони стають критичними. Це — превентивний інструмент. В ідеалі система повинна помітити підвищене навантаження, перебої у роботі окремого компонента чи затримки у відповідях ще до того, як ці збої почнуть впливати на користувачів або розробників.

У контексті даної роботи моніторинг охоплює як загальні метрики продуктивності (навантаження, споживання ресурсів, час відповіді), так і специфічні показники, що пов’язані з внутрішніми процесами: відслідковування стабільності виконання певних операцій, час доступності окремих вузлів, а також реакцію на зовнішні запити. Це багаторівневий підхід, який дозволяє одночасно бачити і загальну картину, і деталі — коли це потрібно.

Особливу увагу приділено способу представлення інформації. Цифри самі по собі мало що дають, поки вони не організовані. Тому важливо не лише зібрати метрики, а й надати до них доступ через зручний інтерфейс: у вигляді графіків, таблиць, індикаторів або попереджень. Така візуалізація дозволяє відстежувати

зміни у динаміці, бачити тренди та реагувати на відхилення не постфактум, а тоді, коли ще можна щось змінити.

І ще один важливий аспект — система моніторингу має бути стійкою. Тобто здатною працювати навіть тоді, коли інші частини системи вже дають збої. Вона повинна бути першим, що помічає проблему, і останнім, що відключається. У деяких випадках саме моніторинг може стати єдиним джерелом інформації про те, що саме сталося.

Таким чином, упровадження моніторингової частини — це не допоміжна функція, а структурно важлива складова. Вона не просто додає зручності — вона є гарантом того, що система не залишиться "сліпою" у критичний момент.

1.3 Централізоване логування подій та збоїв

Якщо моніторинг показує, що щось відбувається — логування пояснює, що саме. Це два різних інструменти, які не можна змішувати, але й не можна розривати між собою. Логи — це хроніка подій. Вони не лише відображають помилки, а й фіксують дії: хто, коли, що запустив, які ресурси використовувались, як довго тривала операція, і чим вона завершилась.

Одна з головних проблем розподілених систем полягає в тому, що інформація про події розпорошена. Кожен компонент веде свій власний журнал, часто в різних форматах, з різною деталізацією, і зберігає його локально. У разі збою це стає головним викликом — знайти, де саме щось пішло не так, у морі розрізнених повідомлень.

Тому в рамках цієї роботи розглядалася задача не просто логування, а саме централізованого логування. Ідея полягає в тому, щоб усі повідомлення — незалежно від їх джерела — потрапляли до єдиного сховища. Це не лише спрощує пошук і аналіз, але й дозволяє автоматизувати обробку: визначати шаблони, фільтрувати за важливістю, формувати звіти або запускати дії у відповідь на певні події.

Важливо, щоб така система логування була масштабованою. Інакше з часом вона сама перетвориться на проблему. Тому враховується структура даних, політика зберігання, можливість архівації старих записів, а також сценарії швидкого доступу до найсвіжішої інформації.

З іншого боку, потрібно пам’ятати, що логи — це джерело чутливої інформації. Вони можуть містити конфігурації, облікові дані, шляхи до системних ресурсів. Тому логування має бути не тільки інформативним, але й безпечним. Захист каналів передачі, контроль доступу до перегляду, обмеження на редагування чи видалення — це все частини однієї картини.

Централізація логів не зменшує обсяг роботи, яку вони виконують, але значно підвищує її цінність. Коли все зібрано в одному місці, ти отримуєш не просто набір повідомлень, а реальну історію того, як жила система — з усіма її помилками, повторюваними проблемами, випадковими збоями. А це вже — основа для вдосконалення.

1.4 Резервне копіювання та аварійне відновлення

Кожна система рано чи пізно стикається з моментом, коли щось йде не за планом. Це не питання "чи станеться", це лише "коли". І саме в цей момент стає зрозуміло, наскільки вона була готова до несподіваного.

Резервне копіювання — це базовий, але критично важливий елемент будь-якої стійкої архітектури. Неважливо, наскільки надійно працює інфраструктура зазвичай — одного збою може бути достатньо, щоб зупинити все. Особливо, якщо втрачені дані або конфігурації неможливо відновити звичайними засобами.

У рамках цієї роботи передбачено впровадження механізму, який дозволяє створювати копії поточного стану системи — регулярно, автоматично, з фіксацією часу та змісту. Мова не тільки про файли чи бази даних. Йдеться про повноцінні знімки середовища, яке можна повернути до робочого стану буквально за кілька хвилин.

Це особливо важливо в умовах, коли система розподілена, а зміни відбуваються часто. Тестування нових функцій, оновлення конфігурацій, зміни у поведінці компонентів — усе це створює ризик непередбачуваних наслідків. Резервна копія в таких випадках — це не підстраховка, а єдина точка повернення.

Сам механізм резервування повинен бути максимально незалежним від решти інфраструктури. Якщо щось ламається — саме він має залишатися працездатним. Тому важливо, щоб процес створення копій не залежав від стану інших сервісів, а самі копії зберігалися в безпечному та ізольованому середовищі.

Але копія — це лише половина справи. Друга — це відновлення. Воно має бути швидким, простим і контрольованим. Не повинно виникати ситуації, коли для повернення до стабільного стану потрібно вручну відновлювати все по частинах або шукати потрібну версію десь у старих архівах. Ідеальний варіант — кілька кліків і все знову працює, як раніше.

Така стратегія дозволяє не боятися помилок. Замість уникати змін, система стає готовою до них. Це не лише про стабільність — це про впевненість у тому, що будь-який експеримент, оновлення чи збій не знищать все безповоротно.

1.5 Інтеграція компонентів в єдину інфраструктуру

Центральним елементом розробленої інфраструктури виступає вузол, через який здійснюється повне керування всіма допоміжними компонентами. Йдеться про систему, що не лише виконує власні задачі, а й ініціює розгортання інших частин інфраструктури, координує їхню появу, конфігурацію, запуск і взаємодію між собою.

На практиці це виглядає так: на одному з віртуальних серверів, розгорнутих у керованому середовищі, встановлюється головний контролер. У ньому зосереджено логіку автоматичного створення інших машин — тих, що відповідають за моніторинг, логування, обробку подій, збирання технічних даних. Весь цей процес ініціюється сценаріями, що описані у вигляді декларативних

конфігурацій. Від користувача не вимагається ручного втручання — запуск одного пайплайну ініціює повний цикл побудови нових частин інфраструктури.

Таким чином, інтеграція досягається не через ручне з’єднання компонентів, а через єдину точку керування, яка точно знає, у якій послідовності потрібно створювати модулі, які параметри передати на вхід, і коли вважати систему повністю готовою до роботи. Така централізація знижує ймовірність несумісностей, помилок у конфігураціях, дублювання налаштувань.

Важливо й те, що кожен компонент, який створюється, одразу підключається до загального механізму спостереження та реагування. Це означає, що щойно з’являється новий логічний вузол, він уже має попередньо прописані сценарії взаємодії — зібрані метрики, шаблони повідомлень, правила зберігання логів тощо. Все це дозволяє зосередити контроль у одному місці й уніфікувати процеси, які в інших випадках довелося б налаштовувати окремо.

Завдяки такій архітектурі кожен компонент не є "сам по собі" — він є результатом узгодженого, послідовного процесу, керованого головним середовищем. І саме ця послідовність — основа надійності всієї системи загалом.

Висновки до першого розділу

Перший розділ роботи дозволив визначити ключову мету проекту та сформувати комплексне уявлення про задачі, які стоять перед розробником інфраструктурного рішення. Система, що розробляється, не є ізольованим сервісом або черговим модулем — це багаторівнева структура, яка об'єднує в собі кілька критично важливих напрямів: спостереження, контроль, обробку подій і відновлення.

У процесі формулювання цілей було зроблено акцент на автоматизації та незалежності. Основна ідея — створити рішення, здатне працювати стабільно та передбачувано без постійної участі людини. Це стосується і способу керування

компонентами, і логіки взаємодії між ними, і принципів реагування на позаштатні ситуації.

Кожен функціональний блок має свою роль, але їх цінність розкривається лише тоді, коли вони діють разом. Саме тому стільки уваги приділено питанням узгодженості: від способу запуску окремих сервісів — до інтеграції їх у єдину керовану структуру.

Окремо слід відзначити важливість підходу до резервного копіювання. Замість класичної «страховки на випадок проблем» у роботі реалізовано цілеспрямовану модель, яка дозволяє впевнено вносити зміни, знаючи, що є чіткий і контрольований шлях до стабільного стану системи.

Загалом, у цьому розділі сформовано концептуальний каркас усього проекту. Він задає не лише напрям реалізації, а й відповідає на питання: що саме має вміти система, як вона має поводитись, і чому саме такий підхід є доцільним у заданому контексті.

2 АНАЛІЗ ПОДІБНИХ ІСНУЮЧИХ СИСТЕМ

2.1 Альтернативи систем керування версіями і репозиторіями

У сучасній розробці систему контролю версій обирають не тільки як засіб зберігання коду. Вона стала центральним вузлом усієї взаємодії команди: від рев'ю, до автоматизації, безпеки, планування й релізів. І саме тому вибір платформи — це вже не про git як такий, а про інструмент, на який лягає вся логіка життєвого циклу проекту.

2.1.1 GitHub

GitHub — це не просто сервіс для репозиторіїв. Це екосистема. Вона включає трекер задач, систему пул-реквевстів, CI/CD (GitHub Actions), менеджер пакетів, секретів, та навіть хостинг сторінок. Але найбільше — це де-факто стандарт для open-source. Якщо твій проект там, тебе знайдуть. Якщо ні — вже треба пояснювати чому.

Його інтерфейс зручний, хоч і місцями перевантажений. Навіть без реєстрації користувач може побачити код, історію змін, коментарі до пул-реквестів — усе це прозоро і знайомо мільйонам.

Панель репозиторію зі списком файлів, кнопкою "Fork", табами "Code", "Issues", "Pull Requests" — класика, яку знають навіть ті, хто ніколи не кодив (рис. 2.1).

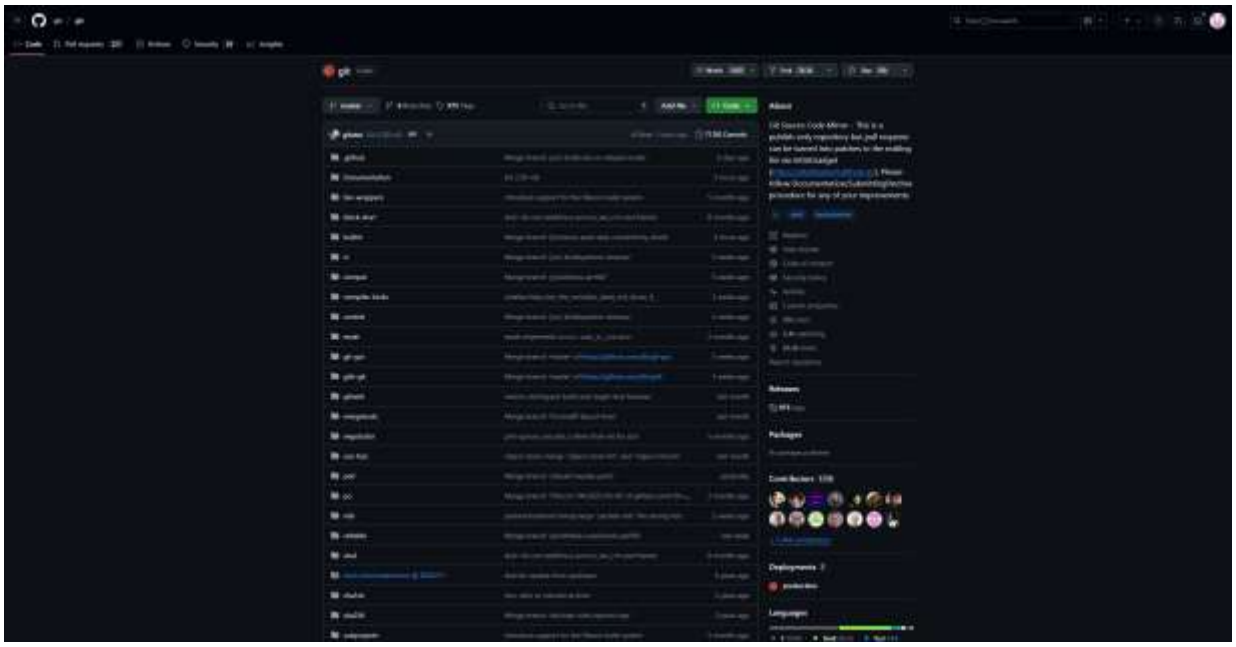


Рисунок 2.1 — головна сторінка репозиторію на GitHub

Та є й зворотна сторона. GitHub — хмарне рішення. У локальному середовищі його не розгорнеш. Можна, звісно, використовувати GitHub Enterprise Server, але це окрема історія: ресурси, ліцензії, супровід. Крім того, для приватних проектів із жорсткими вимогами до доступу GitHub може виявитися надто публічним, навіть якщо він формально «приватний».

2.1.2 Bitbucket

Bitbucket виглядає стримано. Але коли він інтегрований з Jira — перетворюється на серйозну машину для командної роботи. Це одна з його головних фішок: зв’язок із задачами, гілками, комітами — все в єдиній ланці, без потреби додаткових тулів. В Atlassian цьому приділили багато уваги, і результат відчутний у проектах, де водночас йде розробка, тестування, менеджмент.

Інтерфейс у Bitbucket простіший, ніж у GitHub, але менш інтуїтивний для новачків. Навігація між репозиторіями, гілками, пул-реквестами потребує звички. Проте перегляд змін, рев’ю, система захисту гілок — усе працює стабільно.

Основний недолік — менша гнучкість в автоматизації. Bitbucket Pipelines — це окрема штука, зі своїм YAML-форматом. Він непоганий, але не такий універсальний, як GitHub Actions чи Jenkins. Крім того, розгортання self-hosted Bitbucket вимагає вже ліцензії, і його складно інтегрувати без Atlassian-оточення.

2.1.3 Gitea

Gitea — це контраст. Легкий, самодостатній, open-source. Можна поставити за 15 хвилин. І для локальної роботи, тестування, або внутрішніх проектів — ідеально. Особливо якщо не потрібні великі інтеграції, а просто зручний git-сервер із веб-інтерфейсом.

Там усе на місці: репозиторії, гілки, pull request'и, юзери, доступи, навіть CI через сторонні інтеграції. Інтерфейс — схожий на GitHub, але спрощений (рис. 2.2). Іноді — це плюс.

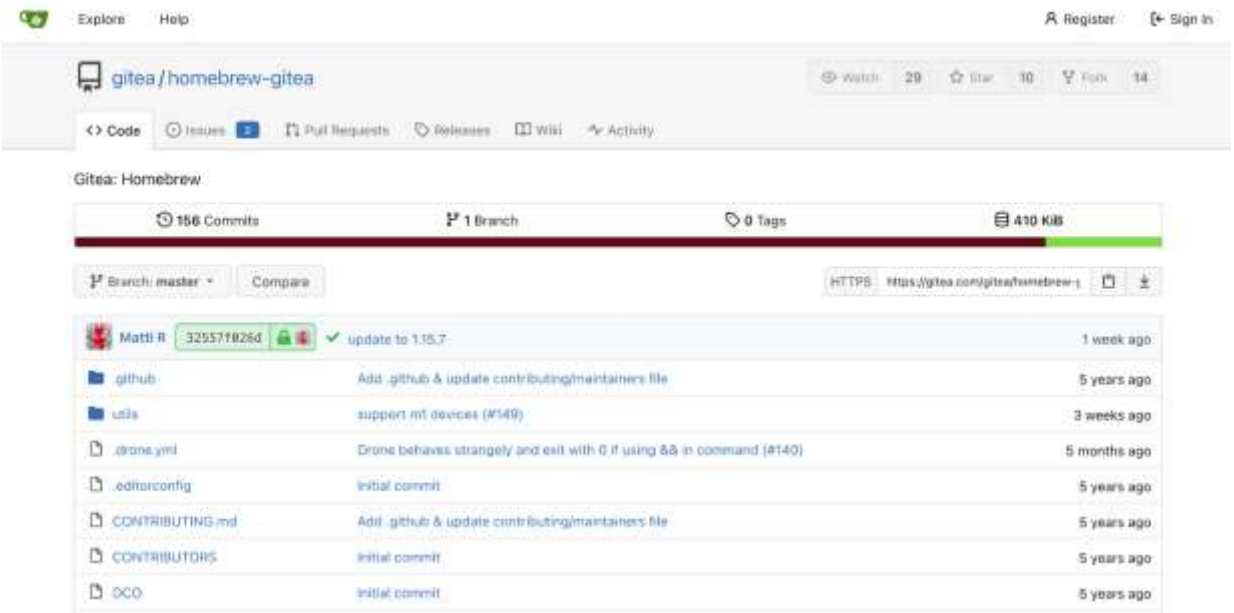


Рисунок 2.2 — головна сторінка репозиторію на Gitea

Але на великі навантаження Gitea не розрахований. У нього є певні обмеження по масштабуванню, і, що важливо, йому бракує розвинутих API та

системи прав, як у старших братів. Це все компенсується простотою й легкістю, але не підійде для середніх або великих команд.

2.2 Інструменти CI/CD

Автоматизація стала чимось більшим, ніж просто зручністю. Це вже звичка, рефлекс, іноді навіть — залежність. Усе, що можна не запускати вручну, має бути автоматизоване. І тут CI/CD — це не просто набір скриптів, а ціла філософія: перевірка, тестування, збірка, доставка — усе, бажано, без втручання людини. Але от що цікаво — інструментів для цього чимало, і в кожного своя філософія.

2.2.1 Jenkins

Jenkins — перше ім’я, яке спадає на думку, коли мова йде про CI/CD. Старий, потужний, майже безсмертний. Він як швейцарський ніж — має плагін на все. Хочеш нотифікацію в Slack? Є. Хочеш збирати Java на Windows і заливати в S3? Теж є.

Інтерфейс у Jenkins спартанський. Панелі, списки, текстові логи — усе функціонально, але без зайвих прикрас. Зате є деталі, які говорять самі за себе: build history, параметризовані job’и, граф виконання пайплайнів (рис. 2.3).

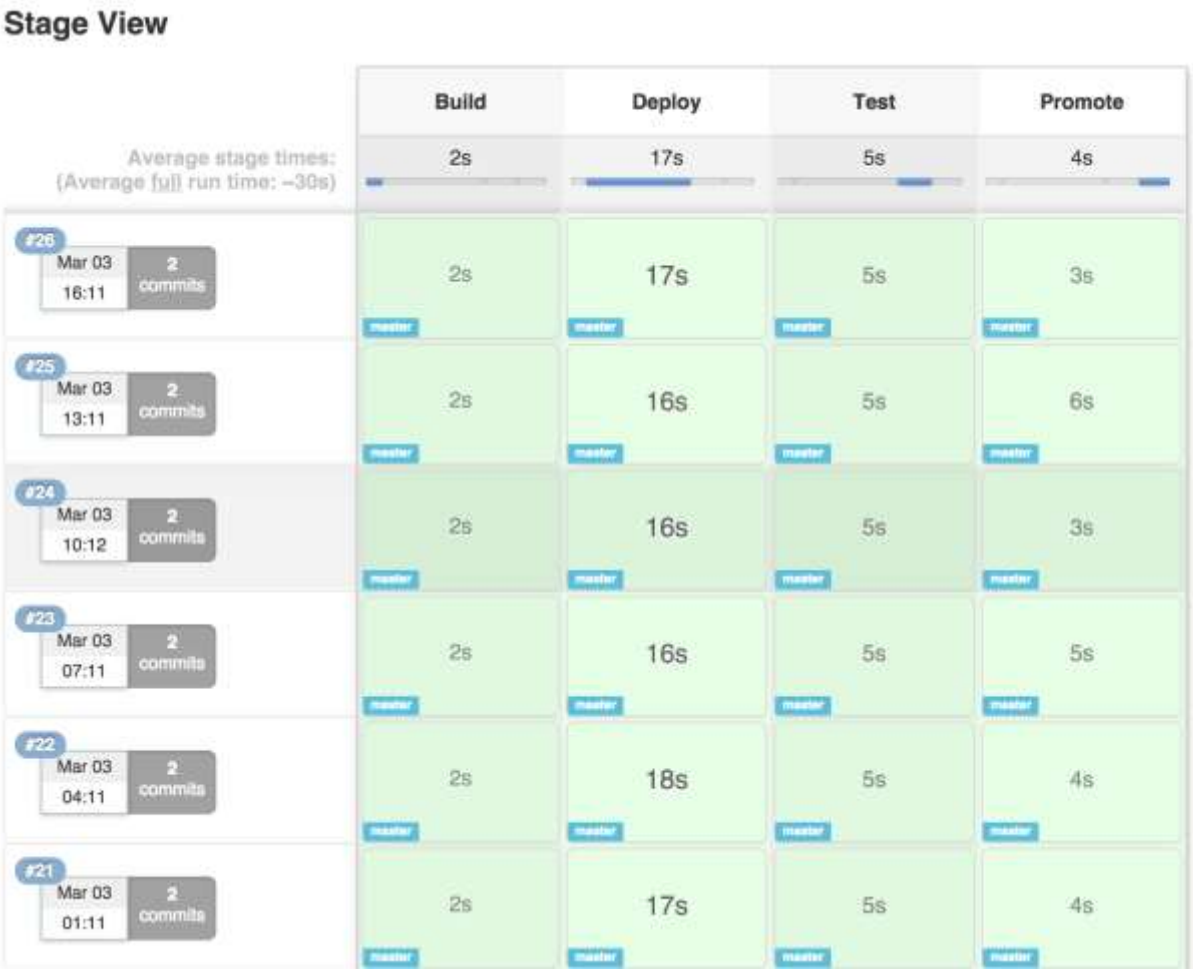


Рисунок 2.3 — сторінка пайплайну в Jenkins

Але пастка Jenkins в тому, що все розбите на плагіни. Кожен з них — окрема історія. Оновлення? Завжди ризик. Конфлікти між версіями? Будь ласка. Іноді налаштування Jenkins перетворюється на повноцінну роботу на пів ставки. Для невеликих команд це часто занадто. Але масштабованість у нього — фантастична.

2.2.2 GitHub Actions

І тут на арену виходить GitHub — зі своїм рішенням Actions. Дуже влучне ім’я: усе обертається навколо «дій». YAML-файли в репозиторії, в яких прописано: що, коли, як. Звучить знайомо? Так, бо вся логіка схожа на інші CI/CD системи, але тут усе інтегровано в GitHub. І це — як благословення, так і обмеження.

Почати дуже просто: вибрав шаблон, запушив — воно вже щось робить. А далі — або летиш на ракеті автоматизації, або впираєшся в нюанси: незрозумілий синтаксис, специфіка налаштування секретів, обмеження на ресурси. Наприклад, при роботі з великими артефактами чи нестандартними окруженнями — доведеться викручуватись.

У Actions є зручна візуалізація workflow. Можна прямо у вікні GitHub подивитись логіку виконання, побачити помилки й спробувати rerun.

2.2.3 CircleCI

CircleCI виглядає мінімалістично, але всередині — це серйозна інженерна платформа. Його люблять стартапи та продуктові команди за простоту конфігурації, швидкий старт, можливість паралельних збірок, кешування залежностей — тут усе швидко.

Він працює в хмарі, але має і варіант локального runner'а. Правда, його важче інтегрувати в ізольовані середовища. Зате в інтерфейсі можна побачити дуже детальну аналітику: яка стадія з'їдає найбільше часу, де найбільше фейлів, що можна оптимізувати.

У CircleCI YAML-файли доволі гнучкі. Можна описати цілу систему залежностей між джобами. Але все одно є обмеження, і кастомна логіка іноді потребує хаків.

Інтерфейс — чистий і сучасний. Все логічно: pipeline, job, step. На кожному рівні можна зайти в лог, перезапустити, побачити змінні (рис. 2.4).

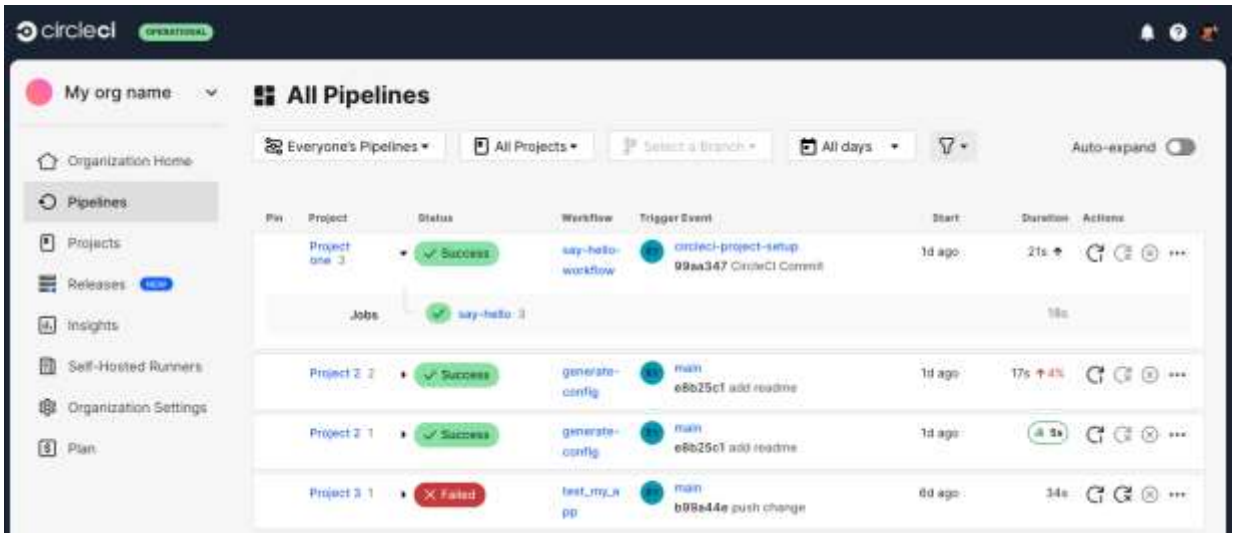


Рисунок 2.4 — сторінка пайплайнів в CircleCI

2.3 Системи моніторингу інфраструктури

Контейнери, віртуальні машини, пайплайни, сервіси — все це працює, поки працює. А коли щось ламається, потрібен моніторинг. Не просто графіки, а система, яка дозволить зрозуміти, що відбувається, де і чому. І зробити це швидко. В ідеалі — ще до того, як хтось встиг це помітити.

2.3.1 Zabbix

Zabbix — це класика. Той самий випадок, коли «надійно, бо перевірено часом». Він вміє багато: збирати метрики, будувати тригери, шукати аномалії, надсилати сповіщення. Причому не тільки по серверам, а й по базах даних, мережах, веб-сайтах. У нього — гігантська екосистема шаблонів.

Але от налаштування… іноді здається, що Zabbix розрахований на окрему людину, яка буде жити в ньому. Агент, сервер, проксі, правила — усе вручну. І якщо треба зібрати просту метрику з контейнера, доведеться пройти цілу війну. А ще — вебінтерфейс хоч і функціональний, але виглядає доволі старомодно (рис. 2.5).

Рисунок 2.5 — дашборд Zabbix

Zabbix — це стабільність, але не гнучкість. Йому комфортно у великих дата-центрах, із фізичними машинами. У сучасних DevOps-ландшафтах — уже не завжди.

2.3.2 Datadog

Зовсім інший підхід — Datadog. Він виглядає як SaaS-моніторинг майбутнього. Красиві графіки, зручна інтеграція з хмарними провайдерами, автоматичне виявлення сервісів, алерти по API. Усе максимально гнучко. І — найголовніше — дуже просто розгорнути. У контейнер кидаєш агент, і вже через хвилину бачиш перші метрики.

Для Kubernetes — просто знахідка. Можна виводити метрики по кожному поду, по лейблам, по сервісах. Плюс — логування, трасування, профілювання — все в одній панелі. Це величезний плюс для команд, які хочуть швидко бачити все й одразу.

Інтерфейс — дуже чистий. Все в drag-and-drop стилі. Можна скласти свій дашборд із метрик по CPU, пам’яті, запитам API, — і налаштувати тригер на будь-яке відхилення.

Але... це дорого. Дуже. Якщо в тебе сотні метрик, десятки сервісів — рахунок буде космічний. І залежність від хмари — теж не всім підходить. Секрети, приватність, політика компанії — усе це може стати причиною сказати «ні».

2.3.3 Nagios

Існує ціле покоління людей, які виросли з Nagios. Він був першим. Його вклад у моніторинг — беззаперечний. Але час минає. Сьогодні Nagios сприймається радше як low-level інструмент: простий, прямолінійний, консервативний.

Його філософія — «все через плагіни». І це працює, якщо ти хочеш сам усе прописати. Потрібно перевірити, чи працює SSH на хості — напиши скрипт. Хочеш знати, чи відповідь сайту не довша за 300 мс — знову скрипт. Гнучко, так. Але незручно. Особливо в епоху контейнерів.

Виглядає Nagios по-простому. І це не завжди мінус. На одному екрані можна побачити статус усіх хостів і сервісів: зелений — добре, червоний — погано.

Але інтеграції, візуалізація, API — усе це або старе, або дуже обмежене. І якщо проект розвивається, росте — Nagios швидко перестає встигати.

2.4 Централізоване логування та аналітика логів

Поки все працює — логи нікому не цікаві. Вони просто лежать собі десь у /var/log, і ніхто їх не чіпає. Але варто з’явитися дивному багу — і вони стають єдиним джерелом правди. Саме тому логування — це не просто журнал. Це засіб розслідування, аудиту, аналізу і навіть автоматичного реагування. Але знову ж — важливо не тільки писати логи, а й уміти з ними працювати.

2.4.1 ELK Stack

Одне з найвідоміших рішень для логінгу — ELK Stack: Elasticsearch + Logstash + Kibana. Кожен компонент — частина великої машини. Logstash збирає і обробляє, Elasticsearch індексує й зберігає, а Kibana — показує.

Сила ELK — у масштабованості. Хочеш обробляти терабайти логів на день? Без проблем. Потрібно фільтрувати по конкретному юзер ID, рівню помилки, типу запиту — усе це доступно в інтерфейсі. Kibana — це як пошуковик по всьому, що траплялося у твоїй інфраструктурі.

Інтерфейс Kibana схожий на щось між панеллю керування і системою аналітики. Графіки, гістограми, детальні фільтри, таймлайн — усе це на одному екрані (рис. 2.6).

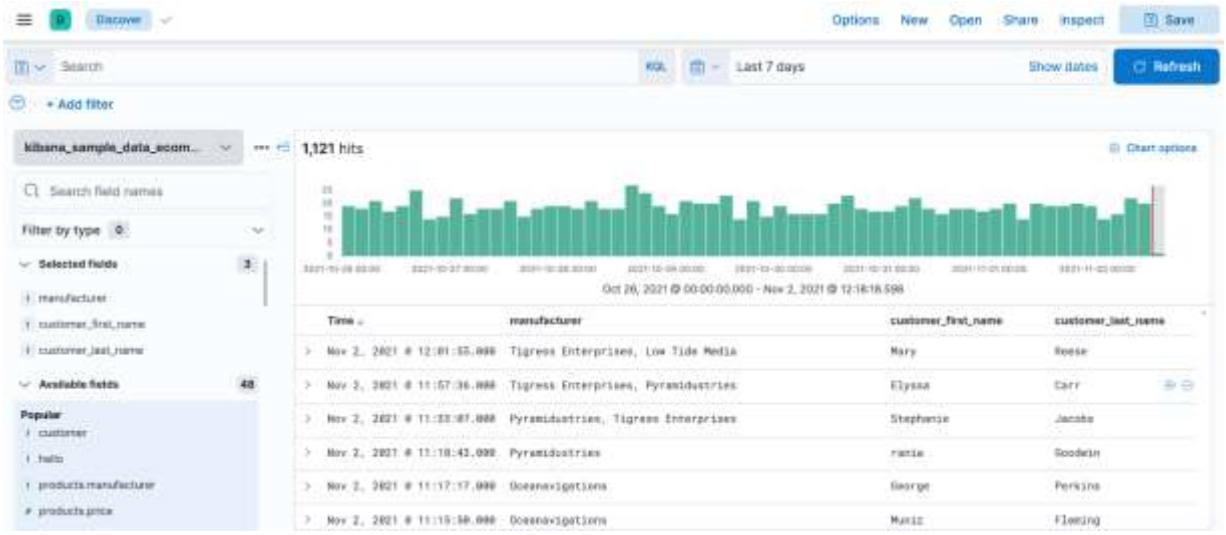


Рисунок 2.6 — інтерфейс Kibana

Попри велику популярність, ELK Stack має низку обмежень, які варто враховувати ще до впровадження. Найсуттєвіший із них — ліцензійна модель. Частина функціональності, зокрема контроль доступу, безпечне зберігання даних, алерти та інші розширення, доступні лише в комерційній версії. Це ставить проект у залежність від платного програмного забезпечення, що не завжди виправдано — особливо в умовах обмеженого бюджету або при побудові внутрішніх рішень.

2.4.2 Graylog

Graylog — альтернатива, яка намагається бути простішою. І, чесно кажучи, у багатьох це виходить. Він теж використовує Elasticsearch «під капотом», але додає свій шар логіки, інтерфейс і механізми обробки.

У Graylog все обертається навколо потоків (streams). Можна задати умови — і логи, що їм відповідають, підуть в окремий потік. Це зручно, якщо хочеш, наприклад, відділити системні логи від логів NGINX або бекенду.

Інтерфейс Graylog доволі зрозумілий. Візуально він нагадує суміш Kibana і старого Grafana: багато таблиць, фільтрів, швидкий drill-down.

Але знову — без недоліків не обійшлося. Безкоштовна версія має обмеження — наприклад, по алертах або ролях користувачів. Додаткові фічі — тільки в Enterprise. Іноді це стає критичним.

2.4.3 Loki + Grafana

Інший підхід демонструє Loki від розробників Grafana. Це не база даних у класичному розумінні, а time-series з логами. Ідея в тому, щоб поєднати метрики й логи в одному дашборді. Не треба перемикатися між Grafana і Kibana — усе поруч.

Loki не індексує повністю вміст логів, а тільки метадані. Це економить ресурси, але ускладнює пошук за значеннями. Наприклад, знайти всі записи з помилкою 500 — легко. Але знайти рядок із певним параметром в JSON — уже складніше.

Інтерфейс у Grafana — все та ж приємна знайома панель. Можна перемикатися між графіками і логами прямо на одному екрані. Натиснув на пік — побачив, які логи це спричинили (рис. 2.7).

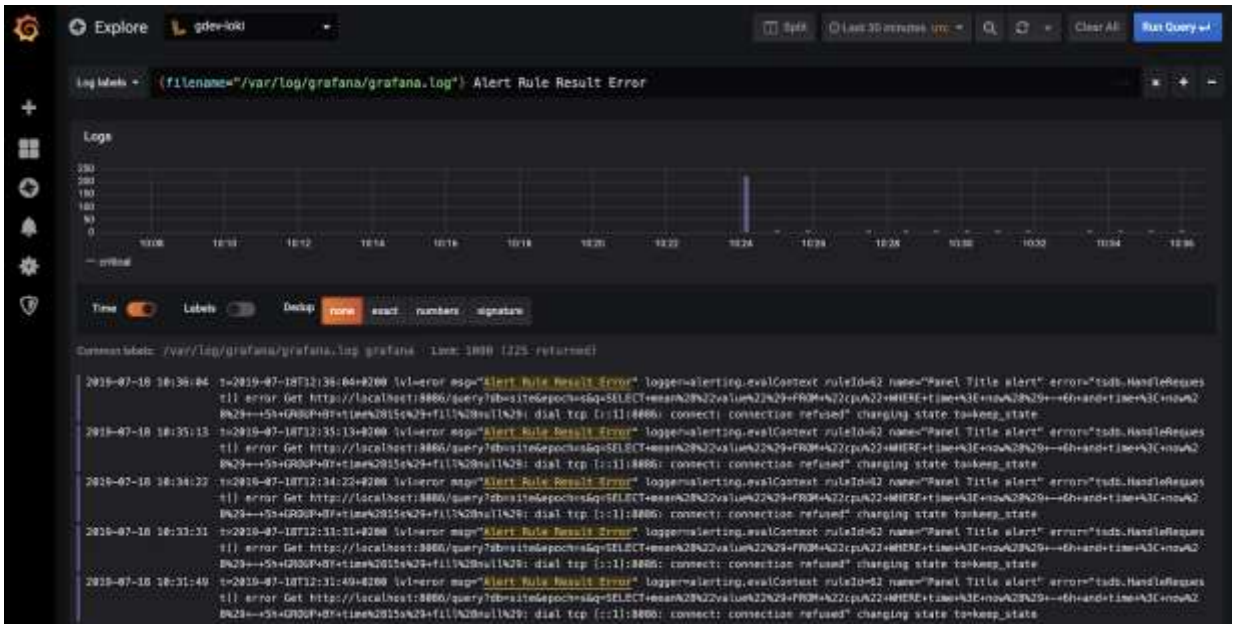


Рисунок 2.7 — Grafana Explore tab з результатами запиту по логах у Loki

Loki виглядає ідеально для сучасних Kubernetes-середовищ, але вимагає додаткових знань для кастомізації. І для великих обсягів — усе ж менш ефективний, ніж повноцінний Elasticsearch.

2.5 Аварійне відновлення та резервне копіювання

Бекапи — це те, що всі обіцяють зробити "після релізу". А потім… потім щось падає, і реліз уже нікому не цікавий. Відновлення — ось що стає головним. І коли система справді важлива, питання не в тому, чи вона зламається, а коли, і як швидко її можна повернути назад.

2.5.1 Veeam Backup & Replication

У корпоративному світі Veeam — це як синонім до слова "резервне копіювання". Він підтримує віртуальні машини, фізичні сервери, бази даних, хмарні сервіси. І все це — з єдиного інтерфейсу. Дуже зручно, якщо потрібно керувати тисячами копій і мати суворий RPO.

Інтерфейс Veeam — візуально приємний. Є календарі, дашборди, журнали копій, стан реплік, звіти про відновлення (рис. 2.8). Одним поглядом видно, що і коли зберігалося, і чи не зламалося щось під час копіювання.

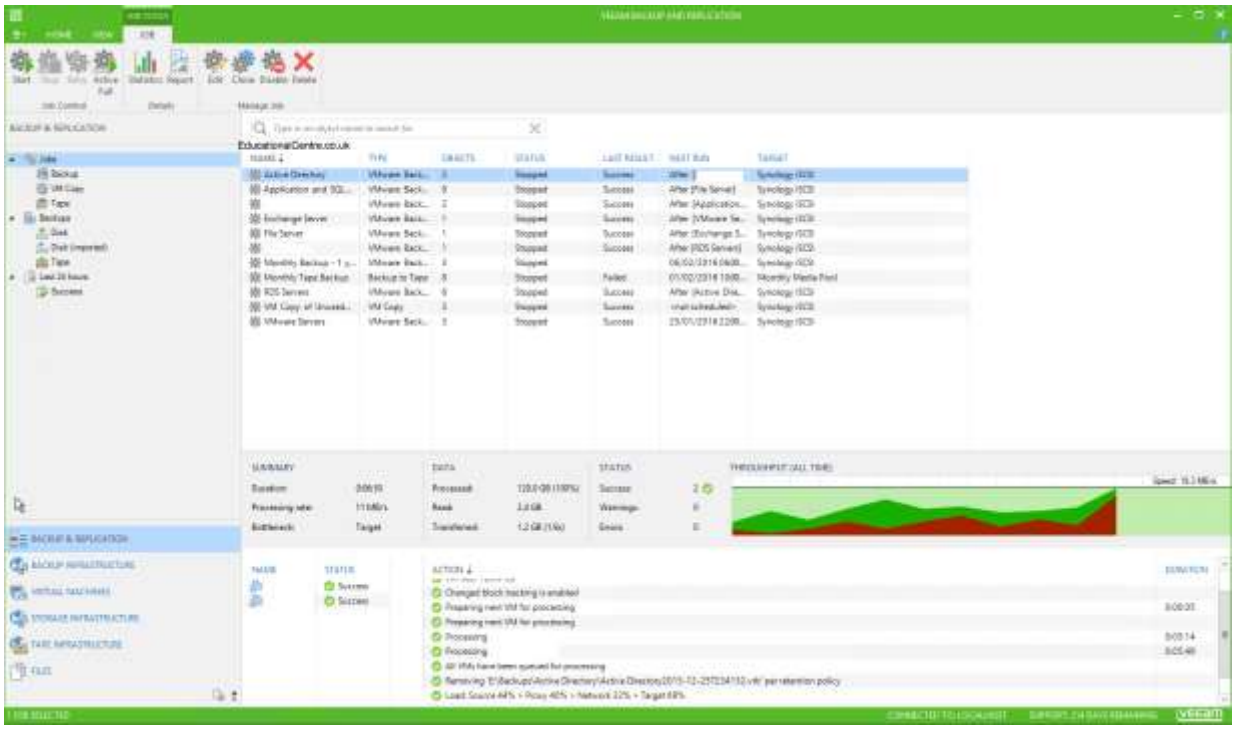


Рисунок 2.8 — головна панель Veeam

Але є нюанс — це рішення не з дешевих. Ліцензії, інфраструктура, підтримка. Крім того, воно переважно орієнтоване на віртуалізацію, хоча й підтримує інші сценарії. Тобто це вже не просто інструмент, а ціла стратегія, яка вимагає ресурсів — людських і фінансових.

2.5.2 Proxmox Backup Server

У середовищах з відкритим кодом часто обирають Proxmox Backup Server. Це легке рішення для резервного копіювання віртуалізованих систем, особливо в тандемі з Proxmox VE. Мінімум зайвого, максимум прямої користі.

Proxmox створює інкрементальні бекапи, шифрує їх, веде хешування для перевірки цілісності. Усе це відображається у простому, але функціональному вебінтерфейсі. Є змога запускати відновлення прямо з GUI, без жодного SSH.

Але тут є свій мінус — прив’язаність до Proxmox як основної платформи. Для VMware, Hyper-V чи хмари — доведеться шукати щось інше. Крім того, для повноцінної інтеграції в CI/CD або DR-процеси потрібні додаткові сценарії або API-зв’язки.

2.5.3 Bacula

Bacula — справжній динозавр світу резервного копіювання. Потужний, модульний, гнучкий. І дуже складний. Його не розгортають за пів години. Він потребує розуміння архітектури: директор, storage daemon, file daemon. Зате, якщо вже розгорнули — він здатен робити все. Архівування, дедублікація, шифрування, стрічки, каталоги — це про нього.

Його інтерфейс — не про дизайн. Більшість керування — через консоль. Є графічні оболонки, але виглядають вони доволі умовно.

Bacula — для ентузіастів і адміністраторів, які люблять повний контроль. Він не пробачає помилок, але дозволяє створити надзвичайно гнучку систему бекапів, яка буде працювати десятиліттями.

Висновки до другого розділу

Аналіз існуючих систем показав одне: ідеального рішення не існує. Усе залежить від контексту — розміру команди, вимог до безпеки, бюджету, технічного бекграунду. Те, що в одній компанії буде «срібною кулею», в іншій — обернеться обмеженнями або навіть ризиком.

У сфері контролю версій — GitHub давно задає стандарти. Але його переваги обертаються мінусами, щойно проект виходить за межі відкритого коду або стандартного workflow. Bitbucket інтегрується, але не завжди масштабно. А Gitea — ідеальний, коли потрібна легкість без компромісів щодо приватності.

CI/CD — це вже не просто автоматизація, а архітектурна дисципліна. І тут кожен інструмент — як окремий стиль. Jenkins вимагає глибоких налаштувань і досвіду. GitHub Actions — зручність, але за межами GitHub — пустота. CircleCI швидкий, але знову ж — не завжди контрольований.

Моніторинг і логування — це «сенсорна система» будь-якої інфраструктури. І якщо вона незбалансована, навіть найкраща CI/CD-система безсила. Zabbix — це про надійність, Datadog — про зручність, Nagios — про витримку. У логуванні — той самий принцип: хочеш гнучкість — бери ELK, хочеш простоту — дивись у бік Graylog або Loki.

А резервне копіювання — це про довіру. До себе в майбутньому. І тут важливо не тільки те, що система щось зберігає, а те, як швидко вона дозволить це відновити.

У підсумку, головна думка цього розділу не в тому, що щось «краще» чи «гірше». А в тому, що кожне рішення — це відповідь на конкретний набір запитань, і обирати його слід не за назвою, а за тим, наскільки воно відповідає архітектурним та операційним потребам конкретного проекту.

3 ЗАСОБИ РОЗРОБКИ СИСТЕМИ

3.1 Віртуалізація та хостинг: ESXi, vCenter, Ubuntu, Docker

Почати варто з основи, яка тримає на собі всю інфраструктуру. Це — платформа віртуалізації. Без неї не було б де запустити сервіси, протестувати конфігурації чи просто подивитися, як поводиться система в ізольованому середовищі.

Я обрав VMware ESXi у зв’язці з vCenter, і, якщо чесно, не пошкодував. Це класика в корпоративному середовищі, і вона повністю виправдовує свій статус. Гнучке керування ресурсами, стабільність роботи, підтримка шаблонів — усе це робить процес розгортання віртуальних машин не просто швидким, а приємним. Особливо в поєднанні з Terraform, але про це згодом.

У самому vCenter я створив окремий пул ресурсів для своєї CI/CD-інфраструктури. Це було важливо — відділити експерименти від основних сервісів. У шаблоні віртуальної машини — стандартний Ubuntu Server LTS, на якому одразу ж встановлювався Docker. Нічого зайвого. Лише базова ОС, контейнерний рушій і доступ через SSH.

На скріншоті нижче можна побачити, як виглядала одна з таких машин у vCenter (рис. 3.1). Там усе просто: 8 ядер, 16 ГБ оперативної пам’яті, SSD-диск і стандартна мережева карта. Нічого екстраординарного, зате стабільно і передбачувано.

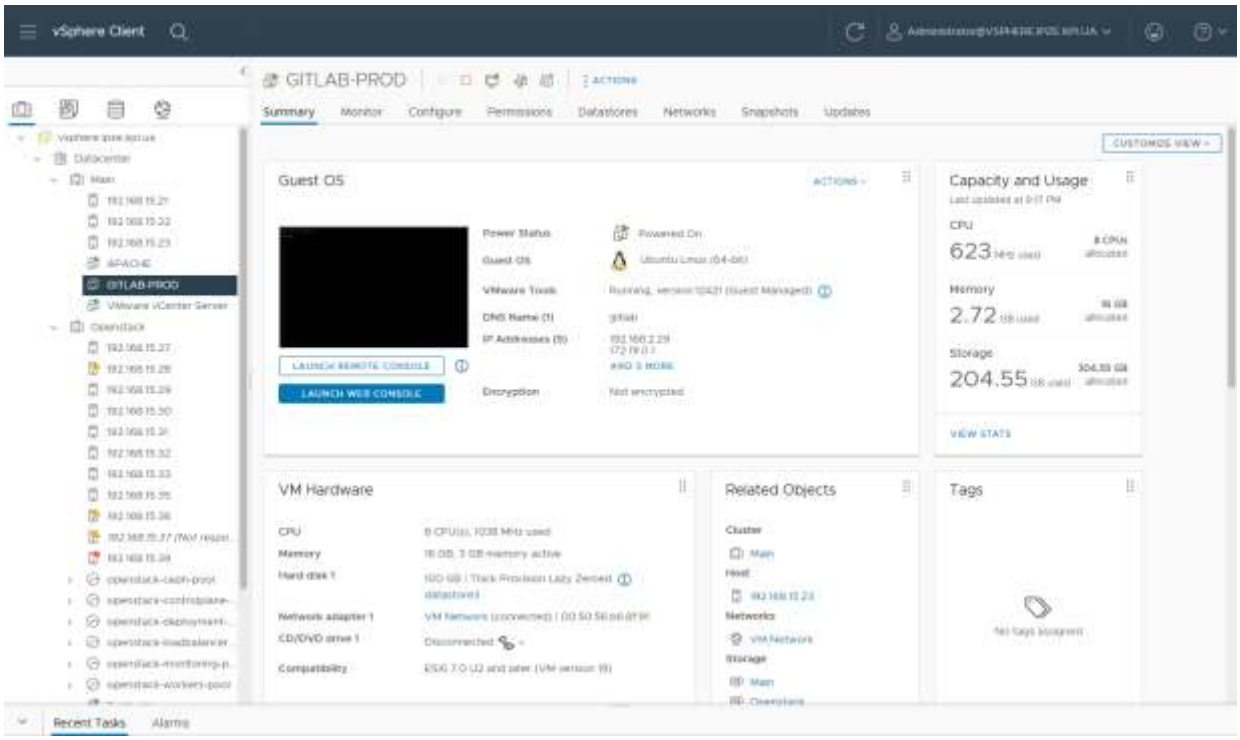


Рисунок 3.1 — Конфігурація VM у vCenter

Той факт, що все це працює на блейд-сервері, дозволяє гнучко масштабуватися в разі потреби. Додав ще одну віртуалку — і готово. Все централізовано. Можна в будь-який момент зазирнути до vCenter, побачити стан машин, змінити ресурси чи створити снапшот перед великими оновленнями.

Коли створюєш середовище з нуля, дуже важливо, щоб воно не було "сюрпризом". Docker дає саме це — передбачуваність. Усі сервіси — від GitLab до OpenSearch — запускалися в контейнерах. Це не тільки спрощувало управління, але й знімало купу головного болю з "а чому тут воно працює, а там ні".

Ще один плюс — ізоляція. Контейнери не лізуть один до одного в мережу, не конфліктують за порти, і при цьому легко підключаються до зовнішніх систем, якщо треба.

Звісно, таке середовище потребує належного моніторингу, але саме тому я почав з фундаменту. Надійна платформа — надійна система.

3.2 Система контролю версій: GitLab CE

GitLab CE у цій системі виконує ключову функцію — він став центром усього робочого процесу. Саме з нього починалося все, що стосувалося коду, змін і запуску автоматизації. Без GitLab CE цей проект не мав би цілісності.

Його локальна інсталяція дала мені максимум контролю. Усі дані, включно з історією комітів, файлами, артефактами CI/CD, токенами — зберігалися в межах локальної інфраструктури. Це важливо: не просто для безпеки, а для повного розуміння того, що відбувається "під капотом".

Платформа дозволяє створювати репозиторії, налаштовувати гілки, організовувати спільну роботу над кодом — і все це в одному середовищі. Від моменту першого коміту до моменту деплою — увесь шлях відбувається через GitLab. Не потрібно переключатись між сервісами, усе зібрано в єдиному інтерфейсі.

GitLab CE підтримує систему ролей і доступу, яку можна підлаштувати під команду. Це дозволяє чітко розмежувати права: хто створює pipeline, хто запускає, хто лише читає. Така гнучкість важлива в системі, де взаємодіє кілька компонентів одночасно.

Ще одна суттєва риса — автоматизація. GitLab CI/CD, інтегрований у GitLab CE, дозволяє будувати пайплайни практично з будь-якою логікою. Хочеш — тестуєш, хочеш — деплоїш, хочеш — просто перевіряєш YAML. Усе це доступне одразу, без встановлення сторонніх розширень. Це частина базової функціональності платформи.

У візуальному інтерфейсі (рис. 3.2) можна побачити, як GitLab показує історію виконання конвеєрів. Кожен рядок — це окремий запуск, зі своїм автором, гілкою, статусом і тривалістю. Це зручно, коли потрібно відслідковувати стабільність або аналізувати проблеми.

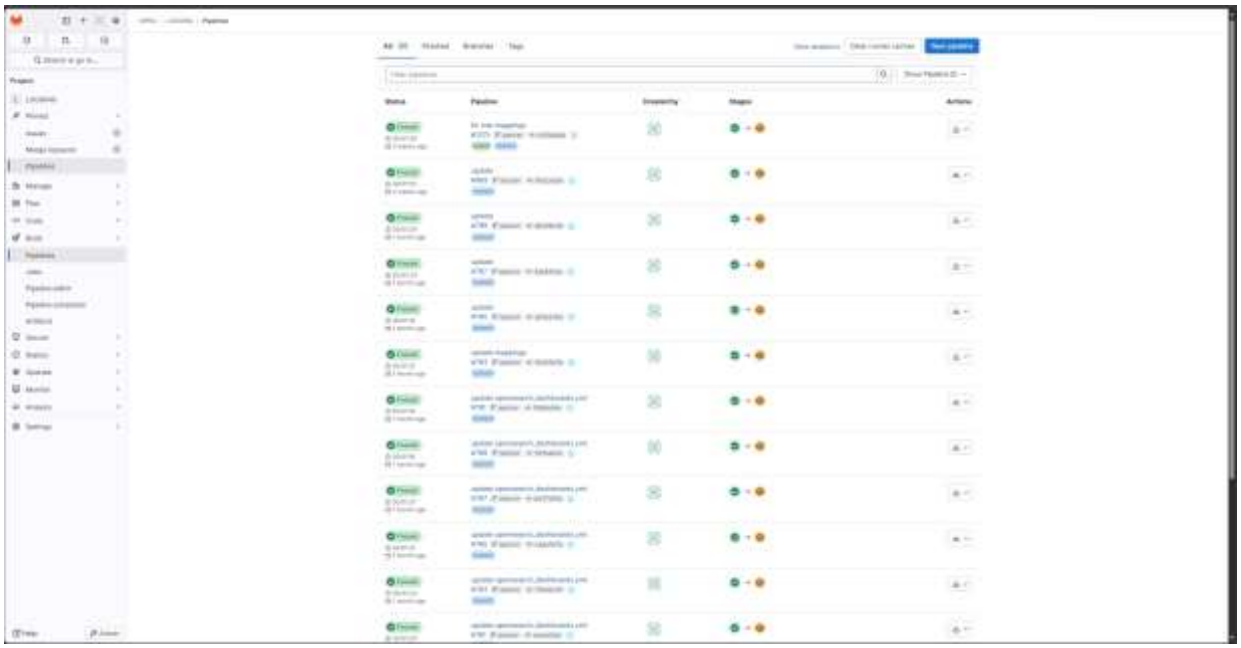


Рисунок 3.2 — Сторінка пайплайнів у GitLab

GitLab CE виконує ще одну, неочевидну, але дуже важливу роль — він зберігає контекст. Тут можна створити issue, пов’язати його з merge request, додати до milestone, обговорити зміни в коментарях, залишити опис до pipeline. І коли через місяць повертаєшся до тієї ж задачі — все на місці. Зв’язки, обговорення, хронологія — нічого не губиться.

Ця система стала точкою входу в CI/CD-інфраструктуру. Усе, що розгорталося, змінювалося або відновлювалося — починалося з GitLab. У певному сенсі це був не просто інструмент, а головний інтегратор усіх інших компонентів: Terraform, Docker, моніторинг, логування — усі вони взаємодіяли через пайплайни GitLab.

3.3 Інфраструктура як код: Terraform

Коли конфігурації стає багато — її неможливо втримати вручну. Один сервер, другий, з десяток мережевих параметрів, змінні, правила, доступи. І ось уже натискаєш “Apply”, не зовсім розуміючи, що саме міняєш. У цей момент починаєш цінувати декларативний підхід.

Terraform став відповіддю на все це. Він дозволив зробити те, що раніше доводилося тримати в голові — формалізувати. Інфраструктура більше не була набором команд у терміналі — вона стала зрозумілим кодом, який можна переглянути, зберегти, порівняти, відкотити.

Його логіка проста: ти описуєш бажаний стан, а Terraform сам вирішує, що потрібно змінити, аби цього стану досягти. І він це робить обережно — спочатку покаже план, а вже потім, якщо погодишся, внесе зміни.

Це особливо зручно, коли ти маєш справу з віртуальними машинами, хостами, DNS-записами або просто мережею. Замість налаштовувати вручну — ти додаєш блок у код. Коміт. Пайплайн. Готово.

Одна з найбільших переваг — повторюваність. Створив шаблон — і далі просто передаєш йому параметри. Неважливо, розгортаєш тестове середовище чи продакшн — ти впевнений, що конфігурації ідентичні. Ти їх бачиш. Вони в репозиторії і їх можна порівняти одна з одною.

На скріншоті (рис. 3.3) видно фрагмент структури Terraform-каталогу. Кожен модуль відповідає за свою частину — мережа, обчислювальні ресурси, моніторинг, логування. Це зручно і з точки зору підтримки, і з точки зору передачі проекту іншій людині.

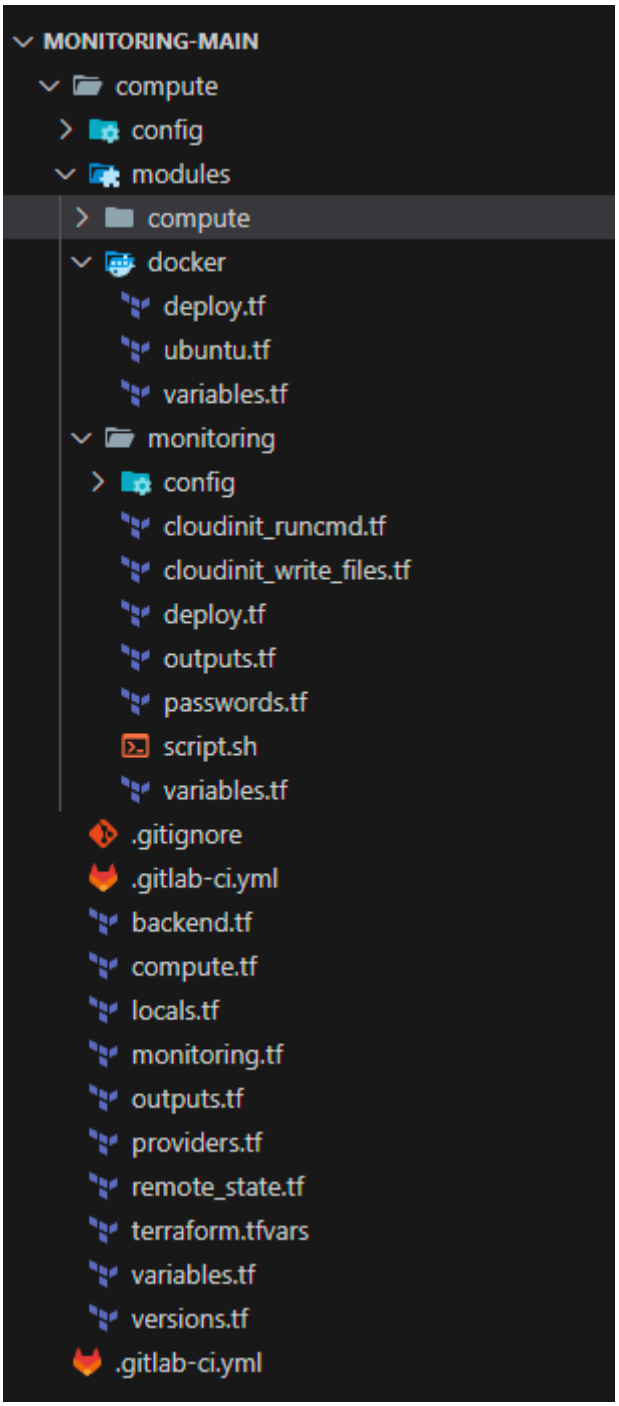


Рисунок 3.3 — Структура каталогу з Terraform-модулями

Terraform не просто економить час. Він дає впевненість. У тому, що система зібрана правильно. У тому, що можна відкотити зміни. У тому, що кожна дія зафіксована. А ще — що не потрібно пам’ятати всі опції з голови. Бо вся інфраструктура — вже в коді.

3.4 CI/CD пайплайни: GitLab Runner і gitlab-ci.yml

Автоматизація — це не розкіш. Це потреба. Особливо коли маєш справу з інфраструктурою, що складається з кількох компонентів, де кожен залежить від іншого. Запустити вручну? Перевірити вручну? Ні. Це не той шлях.

У цьому проекті CI/CD став серцем усіх змін. Змінив щось у коді — pipeline сам перевіряє, тестує, запускає, вносить. Порядок, послідовність, контроль. Саме таким я бачив і хотів бачити цей процес. І GitLab дав мені цю можливість.

Основна логіка роботи полягає у файлі .gitlab-ci.yml. Це, по суті, сценарій — тільки для машин. Він визначає, що відбувається на кожному етапі. Коли зберігається зміна, яка гілка, які джоби, яка послідовність. І що важливо — усе це фіксується прямо в репозиторії разом із кодом. Немає "документу десь там", усе прозоро.

GitLab Runner — це механізм, що виконує ці інструкції. Він бере YAML, читає, розуміє і запускає. І вже не важливо, де саме він розгорнутий — у контейнері, на віртуалці чи в окремому середовищі. Він просто працює.

Перевага такої моделі в тому, що все відтворюване. Один пайплайн у тестовому сереовищі — той самий у продакшн. Не потрібно думати, "а чи я там не забув змінну?" або "чи встановлений той пакет?" — все під контролем. Причому автоматичним.

На скріншоті (рис. 3.4) видно типовий вигляд пайплайну в GitLab: етапи, стани, логи. Навіть без глибокого аналізу видно, чи pipeline пройшов, на чому впав, скільки тривав. Це зручно. І чесно кажучи — заспокоює.

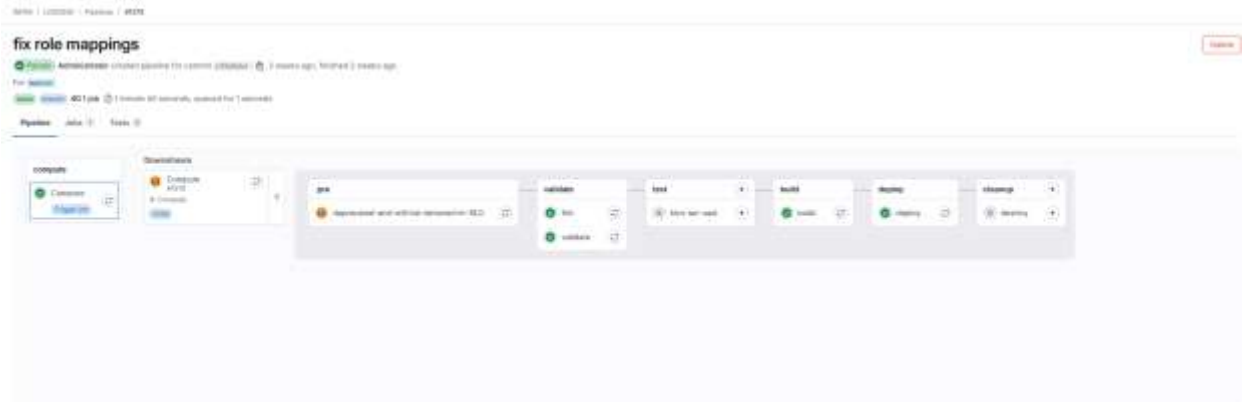


Рисунок 3.4 — Інтерфейс пайплайну в GitLab

Ще одна річ, яку дає пайплайн — це швидкий фідбек. Замість того щоб вручну запускати команди, створювати VM, деплоїти щось кудись, — достатньо одного коміту. І система сама зробить усе інше. Це не просто економія часу. Це зовсім інший ритм роботи.

І найголовніше — все це працює на кожному кроці: при створенні середовища, оновленні конфігурації, резервному відновленні, перевірках на змінність. Пайплайни стали не просто зручним інструментом — вони стали звичкою.

3.5 Моніторинг та аналітика: Prometheus, Grafana, Alertmanager

Щось ламається. Але проблема не в самій поломці. Проблема — коли про неї дізнаєшся надто пізно. Моніторинг у системі — це як відчуття болю в організмі: без нього немає шансу вчасно зреагувати.

У своїй інфраструктурі я заклав моніторинг у саме її серце. Він не був “додатком” після розгортання. Він був частиною архітектури з самого початку. І саме тому вся зв'язка Prometheus + Grafana + Alertmanager спрацювала як треба.

Prometheus відповідає за одне: збір метрик. Він постійно “опитує” сервіси, які я вказав: CI-сервери, раннери, контейнери. Усе, що дихає — під наглядом. І працює це дуже акуратно: pull-модель дозволяє уникати зайвого навантаження.

Зібрані метрики — це лише цифри. Щоб із ними працювати, потрібна візуалізація. Тут у гру вступає Grafana. Саме вона перетворює потік сирих даних на дашборди, які можна зрозуміти з першого погляду. CPU, RAM, час виконання job’ів, кількість активних пайплайнів — усе це показується у зручному вигляді (рис. 3.5).

Рисунок 3.5 — Дашборд в Grafana

Здається дрібницею? Але коли бачиш, як несподівано просідає навантаження або навпаки, різко росте, то ти розумієш: без цього ти наче всліпу. Моніторинг — це очі системи.

Але навіть очі — не завжди достатньо. Потрібна ще реакція. Автоматична. Тут з’являється Alertmanager. Він відстежує певні умови: навантаження вище порогу, відсутність відповідей, затримки. І як тільки щось не так — надсилає сповіщення. У нашому випадку — у Telegram. Але це вже не важливо. Головне — знати раніше, ніж це стане проблемою.

Уся ця трійка — Prometheus, Grafana, Alertmanager — не просто інструменти. Вони дають контроль. А контроль — це передбачуваність. А передбачуваність — це стабільність. І саме цього мені хотілося досягти у проекті.

3.6 Централізоване логування: OpenSearch, Logstash, Filebeat

Моніторинг дає цифри. Але коли щось іде не так — саме логи розповідають історію. Вони — мовчазні свідки кожного запиту, відповіді, винятку, попередження. Саме в них заховані підказки, що, де й чому відбулося.

У великій системі розподілених сервісів логів багато. І вони розкидані всюди: по контейнерах, віртуальних машинах, CI-агентах, системних процесах. Тримати їх окремо — це як мати тисячу сторінок з нотатками, розкиданими по квартирі. Тому — централізація.

Для цієї задачі я використав зв’язку OpenSearch, Logstash і Filebeat, і кожен із них має свою чітку роль.

Filebeat — мовчазний збирач. Він сидить на машині й чекає, поки в журналі з’явиться новий рядок. Як тільки щось відбувається — він пакує повідомлення й передає його далі. У нього немає аналітики чи логіки — лише збір. І це його сила.

Logstash — це мозок. Саме він читає, аналізує, розділяє, маркує, обробляє. Йому можна сказати: "якщо це GitLab — класифікуй як gitlab-*; якщо там ERROR — поміть червоним". Він розуміє структуру JSON, витягує ключові поля, додає мітки. Все, щоб потім можна було знайти потрібне за пару секунд.

А от OpenSearch — це база. Велика, гнучка, здатна працювати з потоками подій у реальному часі. Вона зберігає все: від системних логів до повідомлень про помилки. І не просто зберігає, а ще й дозволяє шукати, сортувати, фільтрувати, візуалізувати.

Використовуючи OpenSearch Dashboards, я отримав зручний інтерфейс для роботи з логами: фільтри, часові діапазони, групування. Усе це дозволяє

зосередитись не на пошуку файлу, а на пошуку сенсу. На скріншоті (рис. 3.6) — типовий приклад роботи з логами GitLab у Dashboards.

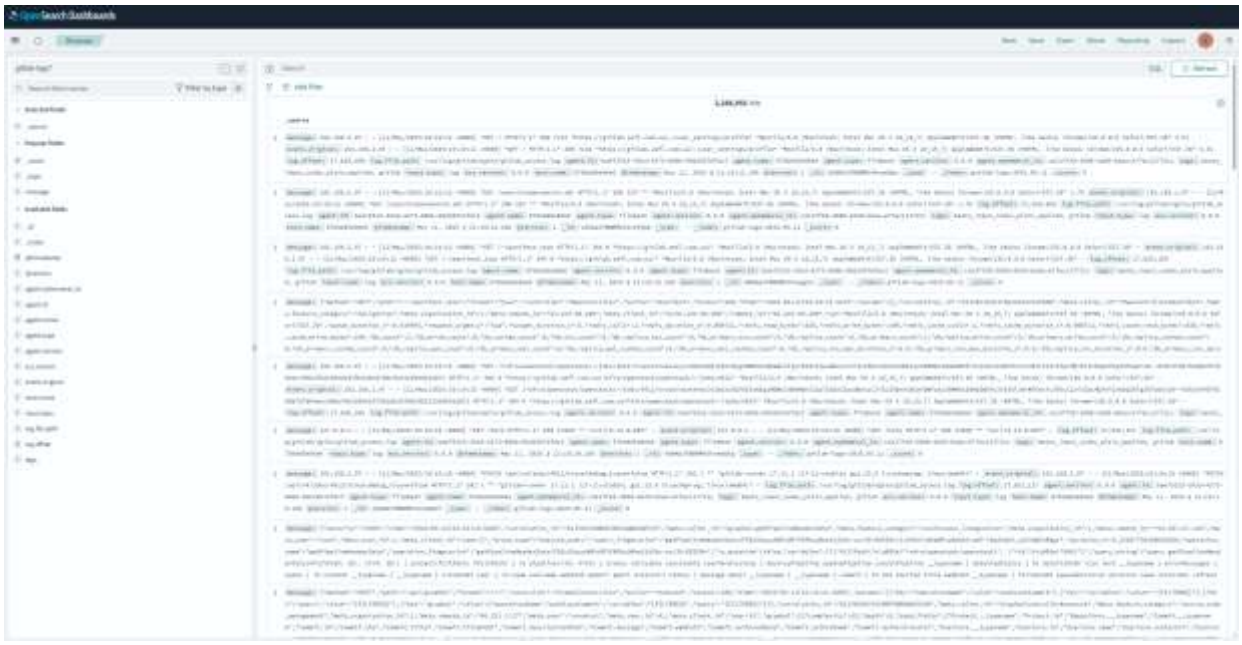


Рисунок 3.6 — OpenSearch Dashboards із прикладом логів GitLab

Логування — це не про "на всяк випадок", це про реакцію. Якщо сервіс впав — перше питання: що сталося? А друге — чому? І тільки третє — як це виправити. Без логів відповіді нема.

І ще: завдяки централізованому логуванню зникає потреба лізти на кожну машину. Усе видно в одному місці. Це не просто зручно, це рятує час і нерви.

3.7 Безпека та резервне копіювання інфраструктури

Найнадійніша система — та, яка вміє падати красиво. Не панічно, не фатально, а тихо — з можливістю швидко відкотитися. Без істерики. Саме на це я робив ставку під час побудови інфраструктури.

Сценарії збоїв — це не вигадка. Хтось випадково видалив конфіг, оновлення пішло не за планом, сервер завис — подібне трапляється. І саме тому я заклав основу — автоматичне резервне копіювання через vCenter.

Щотижневі снапшоти — просте, але потужне рішення. Вони створювались на рівні віртуальних машин без зупинки сервісів. Вночі. За розкладом. Усе автоматично. Ніяких скриптів, жодних ручних процедур. Просто запланована дія — і система має точку повернення.

Що особливо зручно — відновлення з такого снапшота займає менше 20 хвилин. Без відлагодження, без додаткових сценаріїв. Відкриваєш vCenter, обираєш потрібну дату, натискаєш "Restore" — і через кілька хвилин віртуальна машина повертається до життя у стабільному стані.

На скріншоті (рис. 3.7) видно приклад вікна з активними снапшотами GitLab-сервера. Кожен має дату, опис, розмір. Це надає впевненості. Система не боїться втратити контроль.

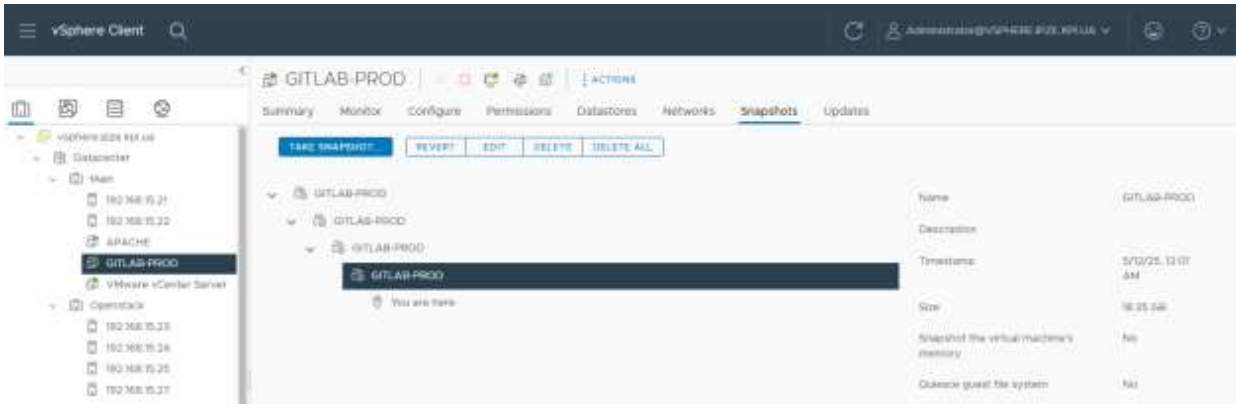


Рисунок 3.7 — список снапшотів для GitLab VM

Окрема перевага цього підходу — він не залежить від самої системи CI/CD. Навіть якщо GitLab “ляже”, снапшот повертає все — конфігурації, історію пайплайнів, ранери, артефакти. Це як страховка, яка працює в найгіршому сценарії.

Безпека — це не тільки про запобігання. Це ще й про здатність швидко відновитися. І в цьому сенсі регулярні снапшоти стали простим, але дуже ефективним рішенням, яке вдалося інтегрувати без зайвих складнощів.

Висновки до третього розділу

Жодна система не будується навмання. Кожен інструмент, кожна платформа, кожна технологія — це рішення. Прийняте або з технічних міркувань, або з практичних міркувань. А часто — з обох одразу.

У цьому розділі я пройшовся по основних "цеглинках", з яких складалася моя CI/CD-інфраструктура. Починаючи з віртуалізації через VMware ESXi — як стабільного фундаменту, який дозволяє легко масштабуватися, ізолювати сервіси й керувати ними централізовано. Це була база, на яку все спиралося.

Далі — GitLab CE, платформа, яка об'єднала в собі і систему контролю версій, і механізми автоматизації, і точку входу для всіх процесів. Вона стала центром, через який проходить кожен шматок коду, кожна зміна, кожна дія.

Terraform додав до цього логіку: “інфраструктура як код” — не модний термін, а реальний підхід, що зробив середовище прозорим і повторюваним. Внесення змін стало контрольованим процесом, а не ручною роботою без історії.

Автоматизація через CI/CD-пайплайни дала системі ритм: коміт — запуск — результат. Без ручної участі, без плутанини. Все — за сценарієм. Все — під контролем.

Інфраструктура не була б повноцінною без моніторингу та логування. Prometheus, Grafana, OpenSearch — разом вони дали змогу не просто спостерігати, а розуміти, що відбувається. І діяти тоді, коли це справді потрібно.

Наостанок — резервне копіювання. Воно не рятує кожного дня. Але коли стається збій, саме воно дозволяє зітхнути спокійно й відновити все без втрат. Це той невидимий механізм, який дає впевненість у завтрашньому дні.

У підсумку, засоби, які я використав, не були випадковими. Вони працювали разом — як частини єдиного механізму. І саме завдяки цьому вдалося створити стабільну, гнучку й передбачувану інфраструктуру, яка готова до змін, навантажень і навіть помилок.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У цьому розділі детально описано етапи реалізації інфраструктури: від автоматизованого створення віртуальних машин до впровадження систем моніторингу, логування і резервного відновлення. Усі дії виконувались з акцентом на автоматизацію та стабільність, використовуючи сучасні DevOps-практики.

4.1 Платформи віртуалізації: VMware ESXi та vCenter

Першим етапом було створення віртуального середовища для подальшого розгортання сервісів моніторингу та логування. Для цього я використав Terraform у зв'язці з провайдером vSphere, що дозволяє взаємодіяти безпосередньо з vCenter. У конфігураційному файлі, розміщеному в каталозі compute/, було вказано адресу сервера vCenter, облікові дані для підключення та налаштування, які ігнорують перевірку SSL-сертифікатів у тестовому середовищі. Далі визначався ресурс для клонування віртуальної машини з шаблону Ubuntu. Параметри включали ім'я віртуальної машини, кількість ядер процесора, об'єм оперативної пам'яті, розмір жорсткого диска та конфігурацію мережевого інтерфейсу (рис. 4.1).

```
module "logging_node" {
  source = "./modules/compute"

  vsphere_compute_vm_name = "LOGGING"

  vsphere_template_path = var.vsphere_template_path

  vsphere_datastore_cluster    = var.vsphere_datastore_cluster
  vsphere_compute_cluster_name = var.vsphere_compute_cluster_name

  vsphere_network_name = var.vsphere_network_name
  datacenter_name      = var.datacenter_name

  vphere_memory_size   = var.logging_nodes_memory
  vsphere_cpu_count    = var.logging_nodes_cpu
  num_cores_per_socket = var.logging_nodes_cpu_num_cores
  vsphere_disk_size    = var.logging_nodes_disk_size

  dns_zone = var.vsphere_dns_zone
}
```

Рисунок 4.1 — Конфігурація віртуальної машини, що використовується для OpenSearch

Після створення машини на неї автоматично встановлювався Docker, що дало змогу одразу переходити до розгортання сервісів — таких як Prometheus, Grafana, OpenSearch та Logstash. Управління Terraform-станом здійснювалося через GitLab-managed remote state, що дозволило уникнути застосування зовнішніх сховищ або ручного керування файлами стану, а також забезпечило контроль змін на рівні CI/CD.

4.2 Автоматизація пайплайнів у GitLab CI/CD

Наступним етапом стало налаштування конвеєрів у GitLab CI/CD. У кожному з репозиторіїв — monitoring та logging — основний файл .gitlab-ci.yml містив лише один job на стадії compute, який за допомогою директиви trigger запускав дочірній пайплайн із підкаталогу compute/ (рис. 4.2). Такий підхід дозволив уникнути дублювання конфігурацій Terraform та зробив структуру більш зручною для супроводу.

```yaml
workflow:
  rules:
    - if: $CI_COMMIT_REF_PROTECTED

stages:
  - compute
  # - configuration

variables:
  IGNORE_TF_DEPRECATION_WARNING: "true"

Compute:
  stage: compute
  trigger:
    include: 'compute/.gitlab-ci.yml'
    strategy: depend
  rules:
    - changes:
        - compute/
        - compute/**
        - compute/**/**
        - compute/**/**/**
```

Рисунок 4.2 — Кореневий .gitlab-ci.yml

У дочірньому пайплайні (рис. 4.3) імпортувався офіційний шаблон Terraform.latest.gitlab-ci.yml, що включає стандартні job’и для ініціалізації, планування та застосування змін.

```
include:
  - template: Terraform.latest.gitlab-ci.yml

default:
  image:
    name: $CI_REGISTRY/infra/tools/docker/openstack-ci-image:latest
    entrypoint: [""]
```

Рисунок 4.3 — Дочірній .gitlab-ci.yml в папці “compute”

Також конфігурація передбачала вказання Docker-образу з усіма необхідними інструментами, встановлення змінних середовища, які визначали назву середовища в залежності від гілки, шлях до конфігураційних файлів та ідентифікатор стану. Крім того, до секції before_script було додано налаштування SSH-ключа, який використовувався для безпечного підключення до vCenter. Додаткові job’и, як-от SAST-сканування за допомогою kics-iac-sast, ручне розгортання (deploy) та видалення ресурсів (destroy), виконувалися тільки на захищених гілках, що відповідало політиці безпеки.

4.3 Впровадження моніторингу з Prometheus та Grafana

Система моніторингу була реалізована як два окремі Terraform-модулі. Перший модуль відповідав за налаштування Prometheus. У конфігураційному шаблоні prometheus.yml.tpl я визначив список таргетів для збору метрик, зокрема — GitLab CE, GitLab Runner, а також кастомні експортери для Docker-контейнерів. Інтервали опитування та імена job’ів були підібрані таким чином, щоб не перевантажувати систему й одночасно зберігати достатній рівень деталізації. Окремим файлом rules.yml були визначені умови для алертів (рис. 4.4), наприклад

— якщо протягом певного часу не надходять метрики від Runner. Alertmanager був налаштований для відправки повідомлень у Telegram-чат команди через webhook.

```yaml
- name: Sidekiq Alerts
  rules:
    - alert: SidekiqHighQueueLatency
      expr: sidekiq_queue_latency_seconds{name="default"} > 1
      for: 1m
      labels:
        severity: warning
      annotations:
        summary: "Sidekiq default queue latency high"
        description: "Latency >1s on default queue (value={{ $value }}s)."
    - alert: SidekiqFailedJobsSpike
      expr: increase(sidekiq_jobs_failed_total[5m]) > 50
      for: 5m
      labels:
        severity: critical
      annotations:
        summary: "Spike in Sidekiq failed jobs"
        description: "More than 5 failures in 5m (count={{ $value }})."
    - alert: SidekiqDeadJobsHigh
      expr: sidekiq_jobs_dead_size > 5000
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "High number of dead Sidekiq jobs"
        description: "Dead jobs >100 (value={{ $value }})."
```

Рисунок 4.4 — Приклад правила алертів

Другий модуль, modules/grafana, забезпечував автоматичне створення контейнера Grafana з попередньо прописаними datasources для Prometheus (рис. 4.5) та наборами дашбордів у форматі JSON.

```yaml
apiVersion: 1
prune: true
datasources:
  - name: prometheus
    type: prometheus
    access: proxy
    orgId: 1
    uid: "default_prometheus_ds"
    url: http://prometheus:9090
    basicAuth: false
    isDefault: true
    jsonData:
      timeInterval: 15s
      tlsAuth: false
      tlsAuthWithCACert: false
```

Рисунок 4.5 — Налаштування datasources для Grafana

Після запуску всі потрібні візуалізації — моніторинг процесорного навантаження, використання пам’яті, активність job’ів у GitLab CI — автоматично підвантажувалися й були доступні в інтерфейсі Grafana. Це дозволило отримати наочну картину стану всієї інфраструктури без додаткових ручних налаштувань.

4.4 Централізоване логування через OpenSearch та Logstash

Для збору логів я розгорнув два окремі Terraform-модулі — один для OpenSearch, другий для Logstash. Модуль OpenSearch відповідав за створення кластера із трьома нодами, які запускалися через Docker Compose (рис. 4.6). У шаблонах конфігурацій задавались параметри доступу, розподіл ролей та політики безпеки. Після старту кластер автоматично індексував системні, прикладні логи та логи GitLab, використовуючи оптимізовані шаблони з шардінгом і реплікацією.

```
opensearch:
  image: opensearchproject/opensearch:latest
  container_name: opensearch
  restart: always
  environment:
    - discovery.type=single-node
    - logger.level=INFO
    - plugins.security.ssl.http.enabled=false
    - "OPENSEARCH_JAVA_OPTS=-Xms2048m -Xmx2048m"
    - OPENSEARCH_INITIAL_ADMIN_PASSWORD=${opensearch_admin_password}
  ulimits:
    memlock:
      soft: -1
      hard: -1
    nofile:
      soft: 65536
      hard: 65536
  volumes:
    - ./opensearch-data:/usr/share/opensearch/data
    - ./config/config.yml:/usr/share/opensearch/config/opensearch-security/config.yml
    #- ./config/internal_users.yml:/usr/share/opensearch/plugins/opensearch-security/securityconfig/internal_users.yml
    - ./config/roles_mapping.yml:/usr/share/opensearch/config/opensearch-security/roles_mapping.yml
  ports:
    - 9200:9200
    - 9600:9600
  networks:
    - opensearch-net
```

Рисунок 4.6 — Конфігурація сервісу OpenSearch в docker-compose

Logstash був налаштований на прийом логів через порт 5044, зокрема від Beats або Filebeat-агентів. У конфігураційному файлі logstash.conf.tpl визначалась

логіка обробки JSON-логів від GitLab: логи парсилися, класифікувалися за рівнем важливості та записувалися у відповідні індекси формату «gitlab-logs-YYYY.MM.dd.» (рис. 4.7)

```
# ---------------------------------------------------------------------------
# FILTERS
# ---------------------------------------------------------------------------
filter {
  ### GitLab logs (filebeat → logstash)
  if [log][file][path] =~ /^\/var\/log\/gitlab\// {
    # tag for downstream
    mutate { add_tag => ["gitlab"] }
    mutate {
      add_field => {
        "[@metadata][target_index]" => "gitlab-logs-%%{+YYYY.MM.dd}"
      }
    }
  }
```

Рисунок 4.7 — Обробка логів від GitLab

Потім ці логи ставали доступними для перегляду через OpenSearch Dashboards, де я створив візуалізації для фільтрації подій рівня ERROR. Такий підхід дозволив швидко виявляти повторювані проблеми та пришвидшив процес їх усунення.

4.5 Організація аварійного відновлення за допомогою щотижневих снапшотів у vCenter

З метою забезпечення безперервної роботи системи я реалізував щотижневе створення снапшотів основних віртуальних машин через вбудовані засоби vCenter. Автоматичне резервне копіювання запускалося за графіком, заданим у планувальнику vCenter. Усі створені снапшоти фіксувалися в системному журналі, що дозволяло легко відстежувати їхню актуальність і стан.

Під час перевірки цього механізму я навмисно викликав збої у GitLab-сервері і тестував відновлення з останнього снапшоту. Процедура відкату до збереженого стану займала менше 20 хвилин, що підтвердило ефективність обраного підходу.

Висновки до четвертого розділу

У четвертому розділі була безпосередньо реалізована вся спроектована інфраструктура, що раніше існувала лише у вигляді концепцій і схем. Основні компоненти — GitLab CI/CD, моніторинг, логування, резервне копіювання — набули практичного втілення у конкретних інструментах, які були налаштовані, протестовані та інтегровані в єдину систему.

Робота з платформою віртуалізації VMware ESXi разом із vCenter продемонструвала, що навіть у локальному середовищі можна досягти високого рівня керованості та масштабованості. Усі сервіси були ізольовані у власних віртуальних машинах, що дозволило уникнути конфліктів і підвищити стабільність системи.

CI/CD процеси, реалізовані через GitLab, забезпечили автоматичне розгортання та перевірку змін. Було налаштовано кілька конвеєрів, які охоплюють повний цикл — від моменту коміту до готового до використання сервісу. Завдяки цьому зменшилася кількість ручних дій, а ризики, пов’язані з людським фактором, було мінімізовано.

Моніторинг за допомогою Prometheus і Grafana дав змогу фіксувати критичні параметри в реальному часі, аналізувати динаміку навантаження та створювати візуалізації, що полегшують діагностику. А система логування на базі OpenSearch і Logstash забезпечила централізований збір подій та зручний пошук інцидентів.

Особливу увагу приділено організації процесу аварійного відновлення: через регулярні снапшоти у vCenter вдалося досягти можливості відновити працездатність усієї інфраструктури за короткий проміжок часу. Це створює відчуття надійності та стійкості до збоїв.

Загалом реалізація підтвердила життєздатність запропонованої архітектури. Інфраструктура не лише функціонує як єдине ціле, а й дозволяє гнучко реагувати на зміни, масштабуватись і швидко відновлюватись у разі непередбачуваних ситуацій.

5 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

У цьому розділі детально описано взаємодію користувача з CI/CD-інфраструктурою, побудованою на базі GitLab, а також з додатковими сервісами моніторингу, логування та аварійного відновлення. Розглянуто основні сценарії, які виконуються розробником або DevOps-інженером — від перегляду стану конвеєрів до аналізу помилок і відновлення роботи після збою.

5.1 Авторизація та огляд конвеєрів у GitLab

Після авторизації в GitLab користувач потрапляє на головну сторінку, де одразу бачить список доступних проектів і загальний статус останніх CI/CD-конвеєрів. Для зручності, система використовує кольорові індикатори: зелений сигналізує про успішне завершення, жовтий — про попередження, а червоний — про наявність помилок. Кожен конвеєр містить інформацію про час запуску, автора змін та тривалість виконання. Через розділ «CI/CD → Pipelines» можна перейти до розширеного списку запусків, де доступна деталізація кожного з них. (рис. 5.1)

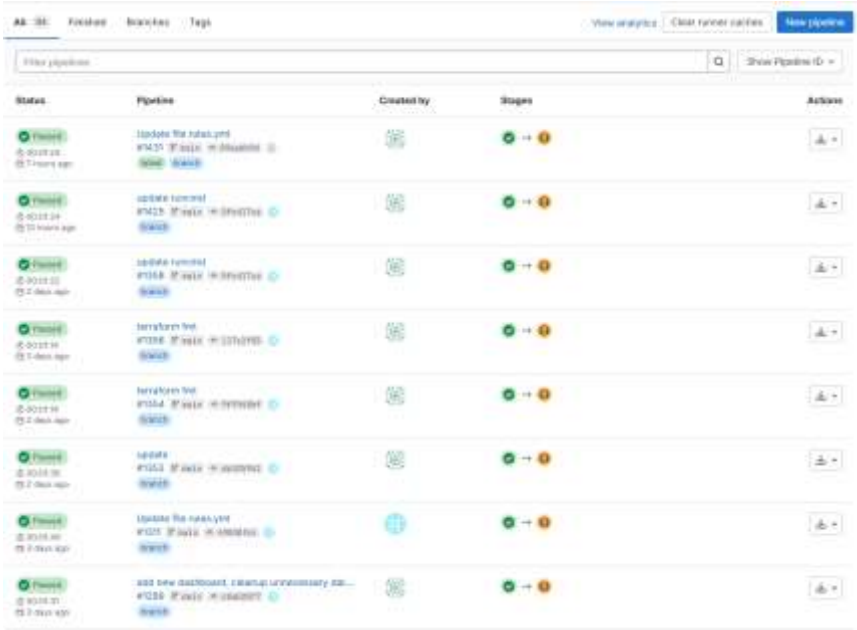


Рисунок 5.1 — Список конвеєрів у GitLab зі сторінки «Pipelines»

Це дає можливість швидко оцінити стабільність процесу розгортання та вчасно виявити слабкі місця.

5.2 Запуск конвеєрів та моніторинг виконання джобів

Конвеєри в GitLab зазвичай запускаються автоматично після створення merge request або коміту до захищеної гілки. GitLab Runner отримує інформацію про нове завдання, завантажує потрібні артефакти й послідовно виконує всі етапи збірки. У процесі виконання користувач може у реальному часі спостерігати за статусом кожного етапу, переглядаючи консольні логи через інтерфейс GitLab. (рис. 5.2)

Рисунок 5.2 — Сторінка обраної Job

У разі помилки система автоматично підсвічує проблемні рядки, що значно пришвидшує аналіз і усунення проблем. Додатково доступна функція завантаження повного лог-файлу або архіву з артефактами, що дозволяє проводити більш глибоку діагностику.

5.3 Перегляд і аналіз метрик продуктивності

Контроль стану системи здійснюється за допомогою зв’язки Prometheus і Grafana. Prometheus зберігає часові ряди метрик, і за потреби користувач може формувати запити вручну, використовуючи мову PromQL. Це дозволяє отримати, наприклад, точні показники використання ресурсів на Runner-хостах або відстежити затримки у роботі контейнерів. Інтерфейс Grafana надає зручні

дашборди, які відображають ключові показники — середній час виконання джобів, кількість запланованих задач, навантаження на хости та інші метрики (рис. 5.3).

Рисунок 5.3 — Дашборд Grafana із метриками GitLab

Існує можливість змінювати часові інтервали, порівнювати дані з історією, а також створювати індивідуальні панелі моніторингу відповідно до потреб користувача.

5.4 Сповіщення про критичні інциденти до Telegram

Інформування команди про критичні події здійснюється через Alertmanager, налаштований у зв'язці з Prometheus. Коли спрацьовує умова для алерту (наприклад, Runner перестав надсилати метрики або перевищено поріг використання CPU), система одразу надсилає повідомлення до Telegram-каналу (рис. 5.4).

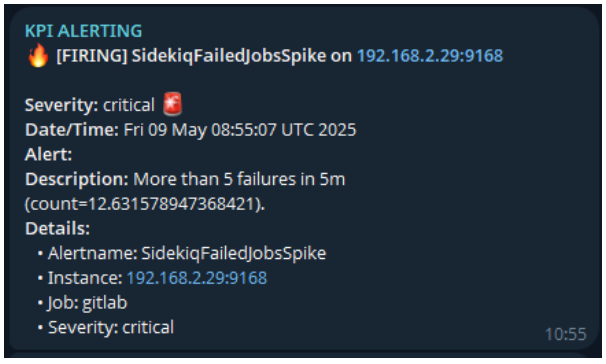


Рисунок 5.4 — Telegram-сповіщення з деталями про помилку

У сповіщенні міститься назва джобу, короткий опис проблеми та її статус, що дозволяє оперативно зреагувати на інцидент без потреби постійного моніторингу графіків. Крім того, всі правила для сповіщень можна переглянути та змінити безпосередньо в інтерфейсі Grafana (рис. 5.5).

Рисунок 5.5 — Перегляд правил сповіщень у Grafana

Це значно спрощує підтримку й налаштування системи алертів, адже навіть людина, що не має попереднього досвіду з подібними системами, може швидко та зручно адаптувати правила сповіщень під свої потреби.

5.5 Централізоване логування та детальний пошук подій

Усі журнали подій — як системні, так і прикладні — централізовано збираються у кластері OpenSearch. Користувач може здійснювати пошук за ключовими словами (рис. 5.6), фільтрувати записи за рівнем важливості (наприклад, ERROR або CRITICAL), а також переглядати логи в розрізі часу.

Рисунок 5.6 — інтерфейс OpenSearch Dashboards з переліком GitLab логів

Завдяки можливостям агрегації, можна виявляти повторювані помилки, аналізувати частоту виникнення збоїв або порівнювати динаміку інцидентів. Інструменти drill-down дають змогу переходити від загальних графіків до конкретних логів за job ID або часовою міткою (рис. 5.7).

Рисунок 5.7 — Інформація про заданий лог в OpenSearch Dashboards

Спеціально створені дашборди в OpenSearch Dashboards дозволяють візуалізувати критичні події, що значно прискорює процес розслідування інцидентів та пошуку кореневої причини проблем.

5.6 Аварійне відновлення за допомогою снапшотів віртуальних машин

Щоб забезпечити безперервність роботи CI/CD-інфраструктури, було налаштовано автоматичне створення снапшотів усіх ключових віртуальних машин у vCenter (рис. 5.8).

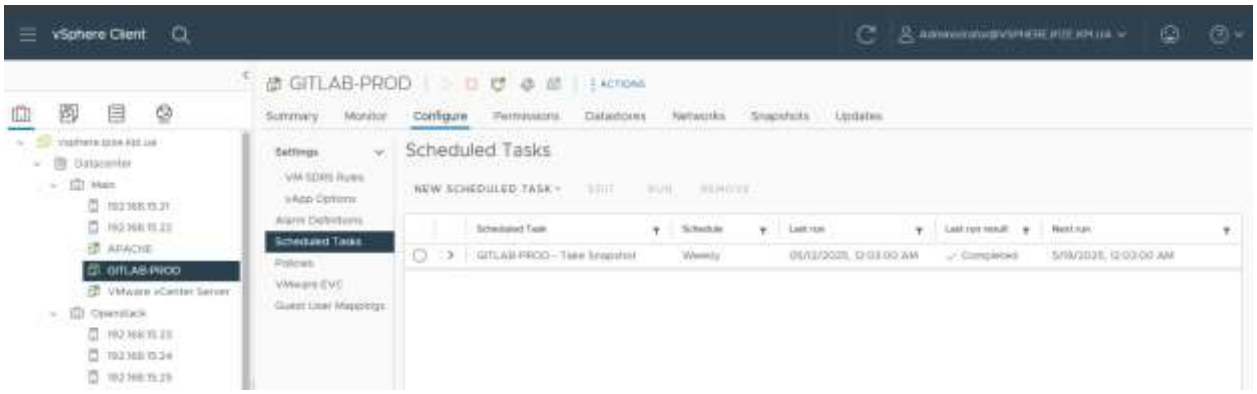


Рисунок 5.8 — Scheduled Task для автоматичного створення снапшоту GitLab VM

Снапшоти створюються щотижня у нічний час — це дозволяє зберігати актуальні копії стану без навантаження на інфраструктуру. У випадку збою адміністратор заходить до vCenter, відкриває розділ «VMs and Templates», обирає відповідну віртуальну машину та повертає її до одного з наявних знімків через вкладку «Snapshots» (рис. 5.9).

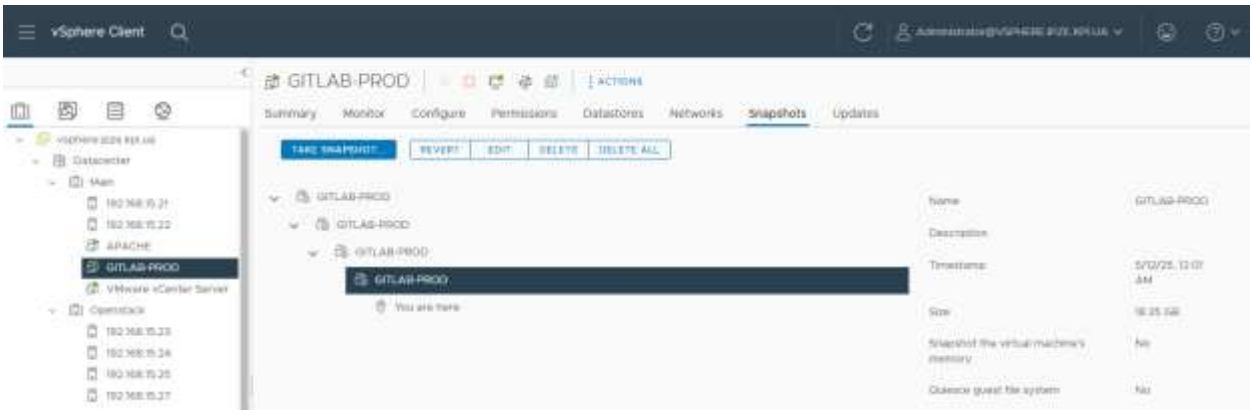


Рисунок 5.9 — консоль vCenter зі списком снапшотів та опцією Restore

Після завершення процедури всі сервіси автоматично перезапускаються у стабільному стані. Такий підхід дозволяє мінімізувати час простою, не вдаючись до складних сценаріїв відновлення або залучення сторонніх інструментів.

Висновки до п’ятого розділу

У п’ятому розділі було детально розглянуто взаємодію користувача з побудованою CI/CD-інфраструктурою. Цей етап дозволив переконатися, що система не лише технічно працездатна, а й зручна у повсякденному використанні — незалежно від того, чи йдеться про запуск пайплайнів, аналіз логів чи моніторинг продуктивності.

Окрему увагу було приділено доступності та простоті. Авторизація в GitLab, перегляд конвеєрів, контроль виконання джобів — усі ці дії реалізовані без надлишкової складності. Це важливо, адже користувач не має витрачати зусилля на боротьбу з інтерфейсом — він має працювати із системою, а не проти неї.

Також у розділі описано сценарії реакції на події. Наприклад, Telegram-нотифікації у випадку критичних збоїв дозволяють оперативно реагувати на інциденти. Це не просто зручність — це частина культури DevOps, де швидкість та прозорість відіграють ключову роль.

Не менш вагомим є блок із централізованим логуванням. Користувач має змогу швидко отримати потрібну інформацію про події в системі — без необхідності вручну перевіряти кожен окремий вузол. Це значно пришвидшує аналіз, особливо в умовах реального інциденту.

Аварійне відновлення із снапшотів — ще один компонент, важливість якого складно переоцінити. Коли щось іде не так, можливість повернутися до стабільного стану буквально за кілька кліків — це і про безпеку, і про впевненість у власній інфраструктурі.

У результаті стало очевидно: користувацька частина реалізованої системи не тільки відповідає технічним вимогам, але й дозволяє ефективно працювати з усіма доступними інструментами. Це свідчить про продуману архітектуру, в якій зручність використання стала логічним продовженням технічного підходу.

ВИСНОВКИ

У процесі виконання дипломної роботи було поглиблено практичні навички у проектуванні архітектури CI/CD-інфраструктури, а також удосконалено теоретичні знання щодо сучасних методів автоматизації життєвого циклу розробки програмного забезпечення. Робота охоплювала повний цикл створення та налаштування системи автоматичної інтеграції, доставки, моніторингу, централізованого логування та резервного копіювання компонентів інфраструктури.

Результатом дослідження стало побудоване комплексне рішення на основі платформи GitLab CE з інтеграцією інструментів Terraform, Prometheus, Grafana, OpenSearch та інших сервісів, що дозволяє реалізувати стабільну та масштабовану DevOps-інфраструктуру. У ході реалізації було застосовано підхід "інфраструктура як код", що забезпечив прозорість, відтворюваність і контрольованість змін конфігурацій середовища.

Окрему увагу приділено реалізації системи моніторингу продуктивності та стану компонентів за допомогою Prometheus і Grafana, що дозволяє оперативно відстежувати ключові метрики. Централізоване логування, організоване на базі OpenSearch, забезпечує ефективний аналіз подій і пришвидшує процес локалізації інцидентів. Крім того, налаштування регулярного резервного копіювання через інструменти VMware vCenter надало можливість швидкого відновлення працездатності системи у разі збоїв або критичних відмов.

Розроблена система може бути адаптована для використання в різних організаціях, де є потреба в безперервному розгортанні, контролі змін та стабільному функціонуванні програмного забезпечення. Виконана робота підтвердила актуальність автоматизованого підходу до керування інфраструктурою та продемонструвала ефективність сучасних open-source інструментів у побудові DevOps-рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. DevOps handbook, second edition: how to create world-class agility, reliability, and security in technology organizations / P. Debois та ін. IT Revolution Press, 2021.
2. Shahin M., Ali Babar M., Zhu L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*. 2017. Т. 5. С. 3909–3943. URL: https://doi.org/10.1109/access.2017.2685629 (дата звернення: 21.05.2025).
3. Lotz N., Timberlake C. Automating DevOps with GitLab CI/CD Pipelines: Build powerful CI/CD pipelines to verify, secure, and deploy your code using real-life examples. Packt Publishing - ebooks Account, 2023. 315 с.
4. Канівець, І. Infrastructure as a Code: найкращі практики для DevOps-інженерів. [Електронний ресурс]. Режим доступу: https://dou.ua/forums/topic/40274/
5. Brikman Y. Terraform : up & running: writing infrastructure as code. O'Reilly Media, 2019. 368 с.
6. Morris, K. Infrastructure as Code: Managing Servers in the Cloud. 2nd ed. Sebastopol: O’Reilly Media, 2020. 350 p.
7. Pivotto J. Prometheus : up and running: infrastructure and application performance monitoring. O'Reilly Media, Incorporated, 2023.
8. Bajpai M. Automating monitoring and incident management with prometheus, grafana, and google cloud pub/sub. *International journal of science and research (IJSR)*. 2022. Т. 11, № 1. С. 1673–1675. URL: https://doi.org/10.21275/sr24829151754 (дата звернення: 22.05.2025).

9. Fallwell, C. What is log management in DevOps? [Електронний ресурс]. Режим доступу: https://www.sumologic.com/blog/log-management-devops
10. Kulikova, D. Become The Master Of Disaster: Disaster Recovery Plan for DevOps. [Електронний ресурс]. Режим доступу: https://gitprotect.io/blog/become-the-master-of-disaster-disaster-recovery-plan-for-devops/

ДОДАТОК А

Впровадження CI/CD інфраструктури на базі GitLab з автоматизованими механізмами моніторингу, логування та аварійного відновлення

Програмний код

Аркушів 11

2025

Logging/modules/logging/config/docker-compose.yaml.tpl

```
services:
  opensearch:
    image: opensearchproject/opensearch:latest
    container_name: opensearch
    restart: always
    environment:
      - discovery.type=single-node
      - logger.level=INFO
      - plugins.security.ssl.http.enabled=false
      - "OPENSEARCH_JAVA_OPTS=-Xms2048m -Xmx2048m"
      - OPENSEARCH_INITIAL_ADMIN_PASSWORD=${opensearch_admin_password}
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
    volumes:
      - ./opensearch-data:/usr/share/opensearch/data
      - ./config/config.yml:/usr/share/opensearch/config/opensearch-
security/config.yml
      #- ./config/internal_users.yml:/usr/share/opensearch/plugins/opensearch-
security/securityconfig/internal_users.yml
      - ./config/roles_mapping.yml:/usr/share/opensearch/config/opensearch-
security/roles_mapping.yml
    ports:
      - 9200:9200
      - 9600:9600
    networks:
      - opensearch-net

  opensearch-dashboards:
    image: opensearchproject/opensearch-dashboards:latest
    container_name: opensearch-dashboards
    restart: always
    ports:
      - 5601:5601
    expose:
      - "5601"
    environment:
      OPENSEARCH_HOSTS: '["http://opensearch:9200"]'
      OPENSEARCH_USERNAME: admin
      OPENSEARCH_PASSWORD: ${opensearch_admin_password}
    volumes:
      - ./config/opensearch_dashboards.yml:/usr/share/opensearch-
dashboards/config/opensearch_dashboards.yml
    networks:
      - opensearch-net

  logstash:
    image: opensearchproject/logstash-oss-with-opensearch-output-plugin:latest
    container_name: logstash
    restart: always
    volumes:
      - ./logstash/pipeline:/usr/share/logstash/pipeline:ro
    ports:
      - "5044:5044"
      - "24224:24224"
```

```
    environment:
      LS_JAVA_OPTS: "-Xmx256m -Xms256m"
    networks:
      - opensearch-net
    depends_on:
      - opensearch

networks:
  opensearch-net:
```

Logging/modules/logging/config/logstash/pipeline/logstash.conf.tpl

```
# -----------------------------------------------------------------------------
# INPUTS
# -----------------------------------------------------------------------------
input {
  # GitLab → Beats on 5044
  beats {
    port => 5044
  }

  http {
    port  => 24224
    codec => json
  }
}

# -----------------------------------------------------------------------------
# FILTERS
# -----------------------------------------------------------------------------
filter {
  ### GitLab logs (filebeat → logstash)
  if [log][file][path] =~ /^\/var\/log\/gitlab\// {
    # tag for downstream
    mutate { add_tag => ["gitlab"] }
    mutate {
      add_field => {
        "[@metadata][target_index]" => "gitlab-logs-%{+YYYY.MM.dd}"
      }
    }
  }

  ### OpenStack service logs (fluentd or beats)
  # match either the beats-style path or the fluentd source field
  if (
       [log][file][path]   =~
/^\/var\/log\/(nova|neutron|keystone|glance|cinder|placement|ceilometer|heat)\/.*\
.log$/
    or [source]           =~
/^\/var\/log\/(nova|neutron|keystone|glance|cinder|placement|ceilometer|heat)\/.*\
.log$/
    or [programname]
    or [python_module]    =~ /^neutron\.plugins\.ml2\.drivers\.openvswitch/
  ) {
    mutate { add_tag => ["openstack"] }
    mutate {
      add_field => {
        "[@metadata][target_index]" => "openstack-logs-%{+YYYY.MM.dd}"
      }
    }
  }
```

```
}


# ---------------------------------------------------------------------------
# OUTPUT
# ---------------------------------------------------------------------------
output {
  opensearch {
    hosts    => ["http://opensearch:9200"]
    index    => "%%{[@metadata][target_index]}"
    user     => ${opensearch_admin_username}
    password => ${opensearch_admin_password}
    sniffing => false
    ssl      => false
  }
}
```

Logging/modules/compute/main.tf

```
resource "vsphere_virtual_machine" "vm" {
  name                 = var.vsphere_compute_vm_name
  resource_pool_id     = data.vsphere_compute_cluster.cluster.resource_pool_id
  datastore_cluster_id = data.vsphere_datastore_cluster.datastore_cluster.id
  num_cpus             = var.vsphere_cpu_count
  num_cores_per_socket = var.num_cores_per_socket
  memory               = var.vphere_memory_size
  guest_id             = "ubuntu64Guest"

  nested_hv_enabled = true
  ept_rvi_mode      = "on"
  hv_mode           = "hvOn"

  network_interface {
    network_id = data.vsphere_network.network.id
  }

  clone {
    template_uuid = data.vsphere_virtual_machine.template.id

    customize {
      linux_options {
        host_name = var.vsphere_compute_vm_name
        domain    = "${var.vsphere_compute_vm_name}.${var.dns_zone}"
      }
      network_interface {}
    }
  }

  disk {
    label = "Main"
    size  = var.vsphere_disk_size
  }

  lifecycle {
    ignore_changes = [
      clone
    ]
  }
}
```

Logging/.gitlab.yaml

```
workflow:
  rules:
    - if: $CI_COMMIT_REF_PROTECTED

stages:
  - compute
  # - configuration

variables:
  IGNORE_TF_DEPRECATION_WARNING: "true"

Compute:
  stage: compute
  trigger:
    include: 'compute/.gitlab-ci.yml'
    strategy: depend
  rules:
    - changes:
        - compute/
        - compute/**
        - compute/**/**
        - compute/**/**/**


# Configuration:
#   stage: configuration
#   trigger:
#     include: 'configuration/.gitlab-ci.yml'
#     strategy: depend
#   rules:
#     - changes:
#         - configuration/
#         - configuration/**
#         - configuration/**/**
#         - configuration/**/**/**
```

Monitoring/modules/monitoring/config/docker-compose.yaml.tpl

```
services:
  prometheus:
    image: prom/prometheus:v2.45.3
    restart: unless-stopped
    volumes:
      - ./prometheus/:/etc/prometheus/
      - ./prometheus_data:/var/lib/prometheus
    command:
      - --web.enable-remote-write-receiver
      - --config.file=/etc/prometheus/prometheus.yml
      - --storage.tsdb.path=/var/lib/prometheus
    ports:
      - "9090:9090"

  node-exporter:
    image: prom/node-exporter:v1.7.0
    restart: unless-stopped
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
```

```
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.rootfs=/rootfs'
      - '--path.sysfs=/host/sys'
      - '--collector.filesystem.mount-points-
exclude=^/(sys|proc|dev|host|etc)($$|/)'
    ports:
      - "9100:9100"

  alertmanager:
    image: prom/alertmanager:v0.27.0
    restart: unless-stopped
    volumes:
      - ./alertmanager/alertmanager.yml:/etc/alertmanager/alertmanager.yml
      - ./alertmanager_data:/alertmanager
    command:
      - --config.file=/etc/alertmanager/alertmanager.yml
      - --storage.path=/alertmanager
    ports:
      - "9093:9093"

  grafana:
    image: grafana/grafana:10.4.0
    restart: unless-stopped
    ports:
      - "3000:3000"
    environment:
      GF_AUTH_BASIC_ENABLED: true
      GF_SECURITY_ADMIN_USER: admin
      GF_SECURITY_ADMIN_PASSWORD: ${grafana_admin_password}
      GF_SERVER_DOMAIN: grafana.sefl.com.ua
      GF_SERVER_ROOT_URL: https://grafana.sefl.com.ua
    volumes:
      - ./grafana:/etc/grafana
      - ./grafana_data:/var/lib/grafana
```

Monitoring/modules/monitoring/config/alertmanager/alertmanager.yml.tpl

```
route:
  receiver: telegram-alerts
  group_by: ['alertname', 'severity']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 1h
  routes:
  - receiver: 'scaling-webhook'
    match:
      alertname: 'CpuUsageHighContinuously'

receivers:
- name: telegram-alerts
  telegram_configs:
  - api_url: https://api.telegram.org
    bot_token: ${bot_token}
    chat_id: ${chat_id}
    parse_mode: HTML
    message: |-
      {{ if eq .Status "firing" }}🔥 {{ end }}{{ if eq .Status "resolved" }}☑ {{
end }}<b>[{{ .Status | toUpper }}] {{ .CommonLabels.alertname }} on {{
.CommonLabels.instance }}</b>
```

```
      {{ range .Alerts }}
      {{$severity := .Labels.severity -}}
      {{ if eq $severity "warning" -}}
      <b>Severity:</b> {{$severity}} ⚠
      {{ else if eq $severity "critical" -}}
      <b>Severity:</b> {{$severity}} 🚨
      {{ else -}}
      <b>Severity:</b> {{$severity}} 💀
      {{ end -}}
      {{$status := .Status -}}
      {{ if eq $status "firing" -}}
      <b>Date/Time:</b> {{ .StartsAt.Format "Mon 02 Jan 15:04:05 MST 2006" }}
      {{ else -}}
      <b>Date/Time:</b> {{ .EndsAt.Format "Mon 02 Jan 15:04:05 MST 2006" }}
      {{ end -}}
      <b>Alert:</b> {{ .Annotations.title }}
      <b>Description:</b> {{ .Annotations.description }}
      <b>Details:</b>
        {{ range .Labels.SortedPairs }} • {{ .Name | title }}: {{ .Value }}
        {{ end }}
      {{ end }}
- name: scaling-webhook
  webhook_configs:
  - url: http://127.0.0.1:8080
```

Monitoring/modules/monitoring/config/grafana/provisioning/datasources/datasource.yaml.tpl

```
# For configuration options, see
#   https://grafana.com/docs/grafana/latest/administration/provisioning/#example-data-source-config-file

apiVersion: 1
prune: true
datasources:
  - name: prometheus
    type: prometheus
    access: proxy
    orgId: 1
    uid: "default_prometheus_ds"
    url: http://prometheus:9090
    basicAuth: false
    isDefault: true
    jsonData:
      timeInterval: 15s
      tlsAuth: false
      tlsAuthWithCACert: false
```

Monitoring/modules/monitoring/config/grafana/provisioning/dashboards/definitions/gitlab.json

```
{
  "id": null,
  "title": "Gitlab Dashboard",
  "tags": ["ruby", "sidekiq", "gitlab", "ci"],
  "timezone": "browser",
  "schemaVersion": 36,
  "version": 2,
  "refresh": "1m",
```

```
  "panels": [
    { "id": 1, "title": "Ruby GC Count (rate)", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "rate(ruby_gc_stat_count[1m])",
"legendFormat": "GC Count rate" } ], "gridPos": { "x": 0, "y": 0, "w": 8, "h": 8 }
},
    { "id": 2, "title": "Ruby GC Time (rate)", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "rate(ruby_gc_stat_time[1m])",
"legendFormat": "GC Time rate (ms)" } ], "gridPos": { "x": 8, "y": 0, "w": 8, "h":
8 } },
    { "id": 3, "title": "Heap Live Slots", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_heap_live_slots",
"legendFormat": "Live Slots" } ], "gridPos": { "x": 16, "y": 0, "w": 8, "h": 8 }
},
    { "id": 4, "title": "Sidekiq Jobs Processed vs Failed", "type": "graph",
"datasource": "prometheus", "targets": [ { "expr":
"increase(sidekiq_jobs_processed_total[1h])", "legendFormat": "Processed (1h)" },
{ "expr": "increase(sidekiq_jobs_failed_total[1h])", "legendFormat": "Failed (1h)"
} ], "gridPos": { "x": 0, "y": 8, "w": 12, "h": 8 } },
    { "id": 5, "title": "Sidekiq Default Queue Latency", "type": "graph",
"datasource": "prometheus", "targets": [ { "expr":
"sidekiq_default_queue_latency_seconds", "legendFormat": "Latency (s)" } ],
"gridPos": { "x": 12, "y": 8, "w": 12, "h": 8 } },
    { "id": 6, "title": "GitLab Projects Count", "type": "stat", "datasource":
"prometheus", "targets": [ { "expr":
"gitlab_database_rows{query_name=\"projects\"}", "legendFormat": "Projects" } ],
"gridPos": { "x": 0, "y": 16, "w": 12, "h": 8 } },
    { "id": 7, "title": "Heap Free Slots", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_heap_free_slots",
"legendFormat": "Free Slots" } ], "gridPos": { "x": 12, "y": 16, "w": 6, "h": 8 }
},
    { "id": 8, "title": "Old Objects", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_old_objects", "legendFormat":
"Old Objects" } ], "gridPos": { "x": 18, "y": 16, "w": 6, "h": 8 } },
    { "id": 9, "title": "Minor vs Major GC (1h rate)", "type": "graph",
"datasource": "prometheus", "targets": [ { "expr":
"increase(ruby_gc_stat_minor_gc_count[1h])", "legendFormat": "Minor GC" }, {
"expr": "increase(ruby_gc_stat_major_gc_count[1h])", "legendFormat": "Major GC" }
], "gridPos": { "x": 0, "y": 24, "w": 8, "h": 8 } },
    { "id": 10, "title": "Sidekiq Queue Sizes", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "sidekiq_queue_size{name=\"default\"}",
"legendFormat": "default" }, { "expr": "sidekiq_queue_size{name=\"mailers\"}",
"legendFormat": "mailers" } ], "gridPos": { "x": 8, "y": 24, "w": 8, "h": 8 } },
    { "id": 11, "title": "Workers vs Processes", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "sidekiq_workers_size", "legendFormat":
"Workers" }, { "expr": "sidekiq_processes_size", "legendFormat": "Processes" } ],
"gridPos": { "x": 16, "y": 24, "w": 8, "h": 8 } },
    { "id": 12, "title": "GitLab Uploads", "type": "stat", "datasource":
"prometheus", "targets": [ { "expr":
"gitlab_database_rows{query_name=\"uploads\"}", "legendFormat": "Uploads" } ],
"gridPos": { "x": 0, "y": 32, "w": 12, "h": 8 } },
    { "id": 13, "title": "Active Users", "type": "stat", "datasource":
"prometheus", "targets": [ { "expr":
"sum(gitlab_database_rows{query_name=\"users\",state=\"active\"})",
"legendFormat": "Active Users" } ], "gridPos": { "x": 12, "y": 32, "w": 12, "h": 8
} },
    { "id": 14, "title": "Heap Allocatable Pages", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_heap_allocatable_pages",
"legendFormat": "Allocatable Pages" } ], "gridPos": { "x": 0, "y": 40, "w": 12,
"h": 8 } },
    { "id": 15, "title": "Total Allocated vs Freed Objects", "type": "graph",
"datasource": "prometheus", "targets": [ { "expr":
```

```
"ruby_gc_stat_total_allocated_objects", "legendFormat": "Allocated Objects" }, {
"expr": "ruby_gc_stat_total_freed_objects", "legendFormat": "Freed Objects" } ],
"gridPos": { "x": 12, "y": 40, "w": 12, "h": 8 } },
    { "id": 16, "title": "Malloc Increase vs Limit (rate)", "type": "graph",
"datasource": "prometheus", "targets": [ { "expr":
"rate(ruby_gc_stat_malloc_increase_bytes[1m])", "legendFormat": "Malloc Increase
rate" }, { "expr": "ruby_gc_stat_malloc_increase_bytes_limit", "legendFormat":
"Limit" } ], "gridPos": { "x": 0, "y": 48, "w": 12, "h": 8 } },
    { "id": 17, "title": "Old Objects Utilization", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_old_objects /
ruby_gc_stat_old_objects_limit", "legendFormat": "Utilization" } ], "gridPos": {
"x": 12, "y": 48, "w": 12, "h": 8 } },
    { "id": 18, "title": "CI Stale Builds & Unarchived Traces", "type": "graph",
"datasource": "prometheus", "targets": [ { "expr": "ci_stale_builds",
"legendFormat": "Stale Builds" }, { "expr": "ci_unarchived_traces",
"legendFormat": "Unarchived Traces" } ], "gridPos": { "x": 0, "y": 56, "w": 12,
"h": 8 } },
    { "id": 19, "title": "Sidekiq Job Status Counts", "type": "graph",
"datasource": "prometheus", "targets": [ { "expr": "sidekiq_jobs_enqueued_size",
"legendFormat": "Enqueued" }, { "expr": "sidekiq_jobs_scheduled_size",
"legendFormat": "Scheduled" }, { "expr": "sidekiq_jobs_retry_size",
"legendFormat": "Retry" }, { "expr": "sidekiq_jobs_dead_size", "legendFormat":
"Dead" } ], "gridPos": { "x": 12, "y": 56, "w": 12, "h": 8 } },
    { "id": 20, "title": "Heap Sorted Length", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_heap_sorted_length",
"legendFormat": "Sorted Length" } ], "gridPos": { "x": 0, "y": 64, "w": 6, "h": 8
} },
    { "id": 21, "title": "Heap Eden Pages", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_heap_eden_pages",
"legendFormat": "Eden Pages" } ], "gridPos": { "x": 6, "y": 64, "w": 6, "h": 8 }
},
    { "id": 22, "title": "Heap Tomb Pages", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_heap_tomb_pages",
"legendFormat": "Tomb Pages" } ], "gridPos": { "x": 12, "y": 64, "w": 6, "h": 8 }
},
    { "id": 23, "title": "Total Allocated Pages", "type": "stat", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_total_allocated_pages",
"legendFormat": "Allocated Pages" } ], "gridPos": { "x": 18, "y": 64, "w": 6, "h":
8 } },
    { "id": 24, "title": "Total Freed Pages", "type": "stat", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_total_freed_pages",
"legendFormat": "Freed Pages" } ], "gridPos": { "x": 0, "y": 72, "w": 6, "h": 8 }
},
    { "id": 25, "title": "Read Barrier Faults", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "ruby_gc_stat_read_barrier_faults",
"legendFormat": "Read Barrier Faults" } ], "gridPos": { "x": 6, "y": 72, "w": 6,
"h": 8 } },
    { "id": 26, "title": "WB Unprotected Objects", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr":
"ruby_gc_stat_remembered_wb_unprotected_objects", "legendFormat": "WB Unprotected"
} ], "gridPos": { "x": 12, "y": 72, "w": 6, "h": 8 } },
    { "id": 27, "title": "Sidekiq Queue Paused", "type": "graph", "datasource":
"prometheus", "targets": [ { "expr": "sidekiq_queue_paused{name=\"default\"}",
"legendFormat": "Default Paused" }, { "expr":
"sidekiq_queue_paused{name=\"mailers\"}", "legendFormat": "Mailers Paused" } ],
"gridPos": { "x": 18, "y": 72, "w": 6, "h": 8 } }
  ]
}
```

Monitoring/modules/monitoring/config/prometheus/prometheus.yml.tpl

```
global:
  scrape_interval: 15s
  evaluation_interval: 10s
rule_files:
  - rules.yml
alerting:
  alertmanagers:
   - static_configs:
     - targets:
        - alertmanager:9093
scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']
  - job_name: gitlab
    static_configs:
      - targets: ['192.168.2.29:9168']
  - job_name: gitlab-runner
    static_configs:
      - targets: ['192.168.2.29:9252']
  - job_name: 'openstack'
    scheme: https
    honor_labels: true
    metrics_path: /federate
    static_configs:
      - targets: ['192.168.1.80:9091']
    basic_auth:
      username: ${openstack_prometheus_username}
      password: ${openstack_prometheus_password}
    tls_config:
      insecure_skip_verify: true
    params:
      'match[]':
        - '{service="neutron"}'
        - '{service="nova"}'
        - '{service="masakari"}'
        - '{service="placement"}'
        - '{service="venus"}'
        - '{service="keystone"}'
        - '{service="heat"}'
        - '{service="blazar"}'
        - '{service="horizon"}'
        - '{service="cinder"}'
        - '{service="skyline_console"}'
        - '{service="glance"}'
        - '{service="skyline_apiserver"}'
        - '{service="rabbitmq_openstack-controlplane-1"}'
        - '{service="rabbitmq_openstack-controlplane-2"}'
        - '{service="aodh"}'
        - '{service="heat_cfn"}'
        - '{service="trove"}'
```

Monitoring/modules/monitoring/config/prometheus/rules.yml

```
groups:
  - name: User count alerts
    rules:
    - alert: UserCountGreaterThan30
```

```
      expr: gitlab_database_rows{query_name="users",state="active"} > 30
      for: 5m
      labels:
        severity: critical
      annotations:
        summary: "Active user count > 30"
        description: "There are {{ $value }} active users (threshold: 30)."
  - name: Ruby GC Alerts
    rules:
      - alert: RubyHeapFreeSlotsLow
        expr: ruby_gc_stat_heap_free_slots < 10000
        for: 1m
        labels:
          severity: warning
        annotations:
          summary: "Ruby heap free slots low"
          description: "Less than 10k free slots remain on the heap (value={{
$value }})."
      - alert: RubyOldObjectsRatioHigh
        expr: (ruby_gc_stat_old_objects / ruby_gc_stat_heap_live_slots) > 5
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High proportion of old objects"
          description: "Old objects exceed 50% of live slots (ratio={{ printf
\"%.2f\" $value }})."

  - name: Sidekiq Alerts
    rules:
      - alert: SidekiqHighQueueLatency
        expr: sidekiq_queue_latency_seconds{name="default"} > 1
        for: 1m
        labels:
          severity: warning
        annotations:
          summary: "Sidekiq default queue latency high"
          description: "Latency >1s on default queue (value={{ $value }}s)."
      - alert: SidekiqFailedJobsSpike
        expr: increase(sidekiq_jobs_failed_total[5m]) > 50
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "Spike in Sidekiq failed jobs"
          description: "More than 5 failures in 5m (count={{ $value }})."
      - alert: SidekiqDeadJobsHigh
        expr: sidekiq_jobs_dead_size > 5000
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High number of dead Sidekiq jobs"
          description: "Dead jobs >100 (value={{ $value }})."

  - name: GitLab Database Alerts
    rules:
      - alert: GitlabOrphanedProjectsExist
        expr: gitlab_database_rows{query_name="orphaned_projects"} > 0
        for: 10m
        labels:
```

```
          severity: warning
        annotations:
          summary: "Orphaned projects detected"
          description: "There are {{ $value }} orphaned projects in the DB."
      - alert: GitlabUploadsCountHigh
        expr: gitlab_database_rows{query_name="uploads"} > 100
        for: 15m
        labels:
          severity: warning
        annotations:
          summary: "High uploads count"
          description: "{{ $value }} rows in uploads check for unexpected growth."

  - name: CI Health Alerts
    rules:
      - alert: CIStaleBuildsDetected
        expr: ci_stale_builds > 0
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Stale CI builds present"
          description: "{{ $value }} stale builds detected investigate runners."
      - alert: CIUnarchivedTracesAccumulating
        expr: ci_unarchived_traces > 10
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Unarchived CI traces accumulating"
          description: "{{ $value }} unarchived traces storage may fill up."

  - name: Workers Load Alerts
    rules:
      - alert: CpuUsageHighContinuously
        expr: (sum(openstack_nova_vcpus_used) / sum(openstack_nova_vcpus) * 100) >
50
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "OpenStack workers have high CPU usage continuously"
          description: "More than 50% of CPU used by OpenStack workers (value={{
$value }})."
```

ДОДАТОК Б

Впровадження CI/CD інфраструктури на базі GitLab з автоматизованими механізмами моніторингу, логування та аварійного відновлення

Тези

Аркушів 3

2025

Впровадження CI/CD інфраструктури на базі GitLab з автоматизованими механізмами моніторингу, логування та аварійного відновлення

У сучасному цифровому світі вимоги до швидкості випуску оновлень і одночасного забезпечення високої якості програмного забезпечення постійно зростають. Компанії прагнуть скоротити час від написання коду до попадання його в робоче середовище, водночас мінімізувати ризики та збої в роботі кінцевого продукту. Саме із цим завданням успішно справляється підхід DevOps, який передбачає тісну взаємодію розробників, тестувальників та фахівців з експлуатації інфраструктури. Завдяки єдиному потоку процесів — від планування й написання коду до його доставки і підтримки — досягається синергія між командами, знижується кількість конфліктів і прихованих помилок, а також підвищується прозорість робочих процесів.

Однією з ключових практик DevOps є безперервна інтеграція (CI) та безперервна доставка (CD). При їх впровадженні кожен внесений розробником коміт одразу відправляється у централізовану систему контролю версій, де активується автоматизований конвеєр. Цей конвеєр виконує низку послідовних кроків: збірку програмних артефактів, запуск юніт- та інтеграційних тестів, статичний аналіз коду, а також перевірку на відповідність внутрішнім стандартам безпеки та якості. Лише після успішного проходження всіх перевірок артефакти стають доступними для подальшої доставки в підготовлене середовище тестування або безпосередньо в продакшен, якщо цього вимагає політика випуску. Такий підхід дозволяє значно скоротити час на виявлення та усунення дефектів, адже проблеми локалізуються одразу, не накопичуючись у кілька релізів.

Проте автоматизовані збірки та деплой — це лише основа стабільності. Для підтримки роботи системи на високому рівні необхідно впроваджувати рішення для моніторингу показників працездатності та централізованого логування. Збір

часових метрик — навантаження на сервіси, використання ресурсів, час відгуку — дозволяє формувати точні запити на виявлення аномалій і запускати оповіщення при перевищенні встановлених порогів. Водночас система агрегації логів збирає події з різних компонентів, класифікує їх за рівнем критичності та надає інтерфейс для швидкого пошуку й аналізу причин збоїв. Інтерактивні панелі з графіками та гістограмами допомагають інженерам реагувати на інциденти в режимі реального часу та оптимізувати продуктивність додатків.

Не менш важливим етапом є забезпечення безперервності бізнес-процесів у випадку критичних збоїв: впровадження стратегій резервного копіювання й аварійного відновлення. Регулярне створення знімків віртуальних середовищ або бекапів ключових сервісів дозволяє швидко повернути систему до працездатного стану після апаратних чи програмних відмов. Детально описані процедури відновлення та періодичне тестування «відкатів» гарантують, що навіть у найскладніших сценаріях простої будуть мінімальними, а втрата даних — непомітною для кінцевого користувача.

Поєднуючи безперервну інтеграцію та доставку, цілодобовий моніторинг і централізоване логування з продуманими механізмами відновлення, організації отримують змогу не лише швидко випускати нові релізи, а й підтримувати високу якість і доступність сервісів. Такий комплексний підхід лежить в основі сучасних DevOps-культур і стає запорукою успіху будь-якого програмного проекту.

Попри всі переваги, DevOps не є універсальним рішенням, позбавленим викликів і обмежень. Одним із ключових бар’єрів на шляху до ефективного впровадження цієї методології є культурні зміни в межах організації. Традиційна модель, де кожна команда працює ізольовано у своїй функціональній "капсулі", часто вступає в конфлікт із принципами відкритої взаємодії, прозорості й спільної відповідальності, які закладено в основу DevOps. Перебудова мислення працівників, подолання недовіри між командами та формування нового підходу до розв'язання проблем вимагає часу, внутрішніх ініціатив і активної підтримки з боку керівництва.

Також варто враховувати питання безпеки. Інфраструктура, побудована за принципами DevOps, зазвичай передбачає постійні зміни та доступ до великої кількості сервісів і середовищ. Якщо DevOps не супроводжується інтеграцією безпеки на кожному етапі життєвого циклу розробки, то система стає вразливою до кібератак.

Не менш важливим викликом є зростання складності підтримки інфраструктури. З розвитком мікросервісної архітектури, контейнеризації та гібридних хмарних середовищ обсяг інструментів, які необхідно моніторити, налаштовувати й оновлювати, значно збільшується. У таких умовах зростає навантаження на інженерів платформи та DevOps-фахівців, що може призвести до вигорання та втрати концентрації на стратегічних задачах.

Отже, хоча DevOps значно покращує швидкість розгортання та якість програмного продукту, його впровадження супроводжується низкою організаційних і технічних викликів. Для досягнення стабільного результату необхідно поєднувати технологічні рішення з чіткою внутрішньою стратегією, навчанням команди, послідовною трансформацією культури та безперервним удосконаленням процесів.

ДОДАТОК В

Впровадження CI/CD інфраструктури на базі GitLab з автоматизованими механізмами моніторингу, логування та аварійного відновлення

Презентація

Аркушів 9

2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

«Впровадження CI/CD інфраструктури на базі GitLab з автоматизованими механізмами моніторингу, логування та аварійного відновлення»

Крутиголова Володимир Олегович, ТВ-12
Голець Владислав Олександрович, асистент

Київ - 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Актуальність теми

- В умовах цифрової трансформації зростає потреба в стабільних, автоматизованих та самовідновлюваних IT-системах.
- DevOps-підхід, зокрема CI/CD, дозволяє скоротити час між написанням коду та його впровадженням у продакшн, забезпечуючи якість і стабільність.
- Водночас важливо мати централізоване логування, систему моніторингу та механізми резервного копіювання для мінімізації впливу збоїв.
- Побудова комплексної інфраструктури з урахуванням цих аспектів – ключ до надійної та гнучкої розробки ПЗ.

Мета та основні завдання

Мета роботи - Побудова CI/CD-інфраструктури на базі GitLab з вбудованими засобами моніторингу, логування та аварійного відновлення..

Основні завдання:

- Проаналізувати існуючі інструменти для CI/CD, моніторингу та логування.
- Створити інфраструктуру за принципами «Infrastructure as Code» з використанням Terraform.
- Налаштувати пайплайни автоматичної збірки та деплою в GitLab CI/CD.
- Реалізувати моніторинг через Prometheus і Grafana.
- Побудувати систему логування на базі OpenSearch та Filebeat.
- Впровадити автоматичне резервне копіювання та відновлення через vCenter.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Огляд аналогів

Системи контролю версій

- GitHub – зручний, популярний, підтримка GitHub Actions, але лише хмарна версія, складно інтегрувати у приватну інфраструктуру
- Bitbucket – інтеграція з Jira, трекінг задач, але обмежена гнучкість, менш зручний інтерфейс
- Gitea – легкий, self-hosted, просте розгортання, але обмежена масштабованість, спрощений функціонал

CI/CD інструменти

- Jenkins – потужний, величезна екосистема плагінів, але складне налаштування, оновлення можуть викликати конфлікти
- GitHub Actions – простий старт, інтеграція з GitHub, але обмеження в кастомізації, складно з великими проєктами
- CircleCI – швидкий старт, гнучке кешування, але обмеження у self-hosted варіантах, залежність від хмари

Огляд аналогів

Системи моніторингу

- Zabbix – гнучкий, перевірений часом, але складний у налаштуванні, морально застарілий інтерфейс
- Nagios – мінімалістичний, сценарний контроль, але низька інтеграція з сучасними DevOps-інструментами
- Datadog – красивий інтерфейс, автоматичне виявлення сервісів, але дорогий, залежність від хмарних сервісів

Логування

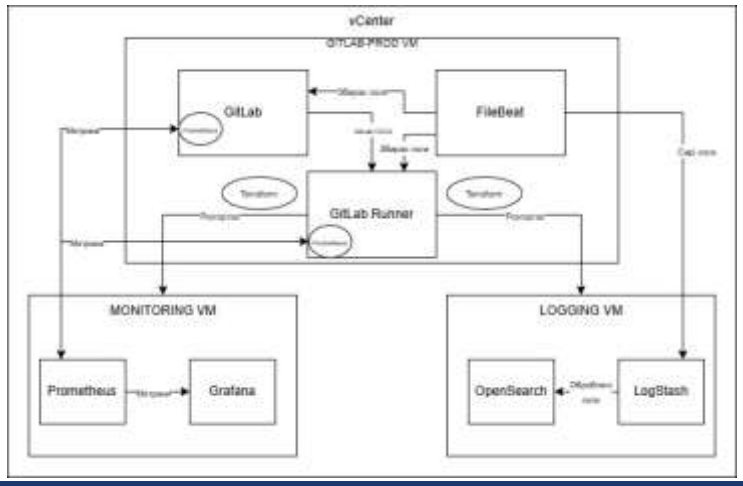
- ELK Stack – потужний пошук і аналітика, але частково комерційний, ресурсоємний
- Graylog – зрозумілий інтерфейс, потокова фільтрація, але обмеження в безкоштовній версії
- Loki + Grafana – логування + метрики на одному дашборді, але обмежений пошук, менш ефективний при великих обсягах логів

Аварійне відновлення

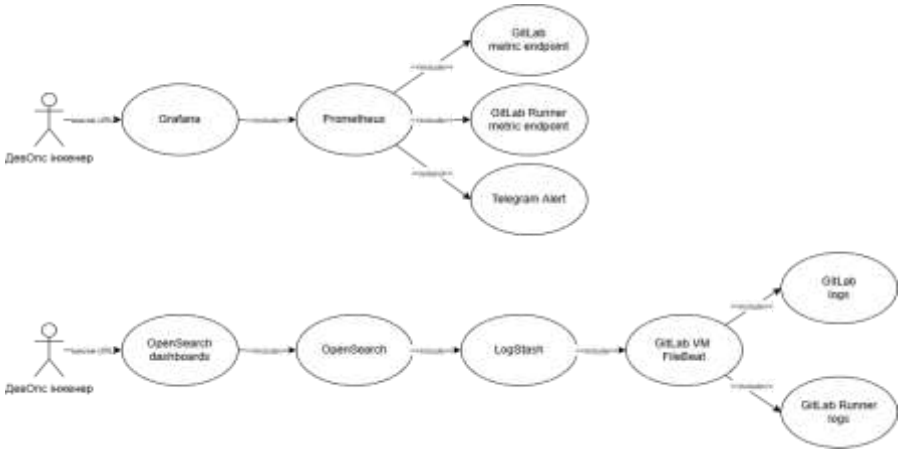
- Veeam Backup & Replication – корпоративний стандарт для резервного копіювання, але потребує платної ліцензії, складне налаштування у невеликих інфраструктурах
- Proxmox Backup Server – оптимізований для середовища Proxmox, але вузька спеціалізація, не підходить для VMware
- Bacula – потужне рішення з відкритим кодом, але складне конфігурування, висока крива навчання

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Архітектура системи



Опис функціональності системи



Недоліки та переваги, розробленої системи

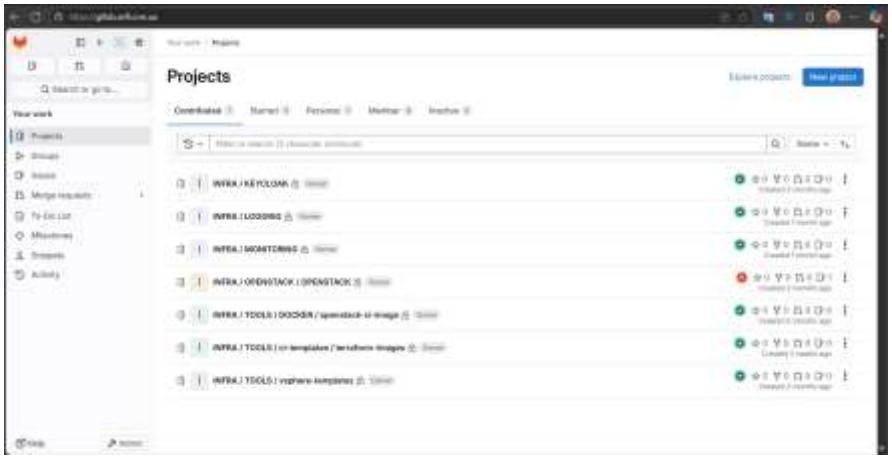
Переваги

- Повна автоматизація CI/CD-процесів через GitLab
- Реалізація моніторингу з алертами в Telegram
- Централізоване логування з гнучким пошуком і фільтрацією
- Візуалізація метрик у Grafana з реального часу
- Резервне копіювання та сценарії аварійного відновлення
- Побудова інфраструктури за підходом Infrastructure as Code
- Мінімальна участь адміністратора у щоденних процесах

Недоліки

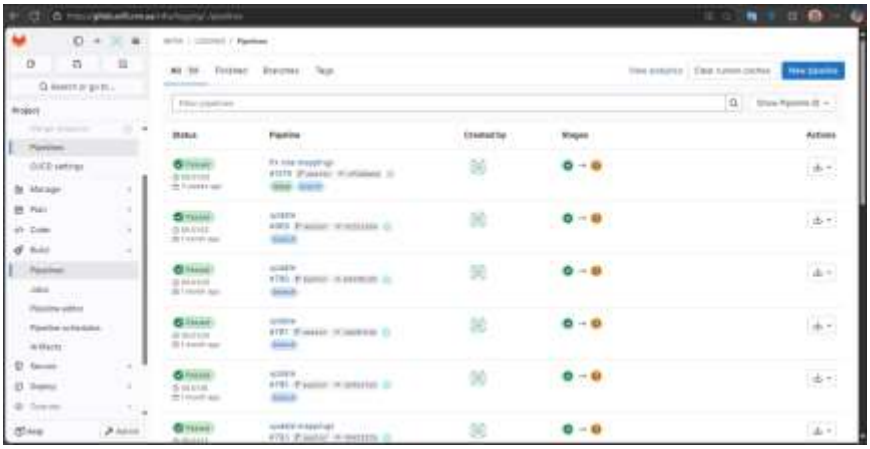
- Підвищені вимоги до ресурсів для запуску всіх сервісів одночасно
- Складність первинного налаштування інструментів (Prometheus, OpenSearch)
- Потреба в базових знаннях DevOps для підтримки системи

Клієнтський інтерфейс. Головна сторінка GitLab



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

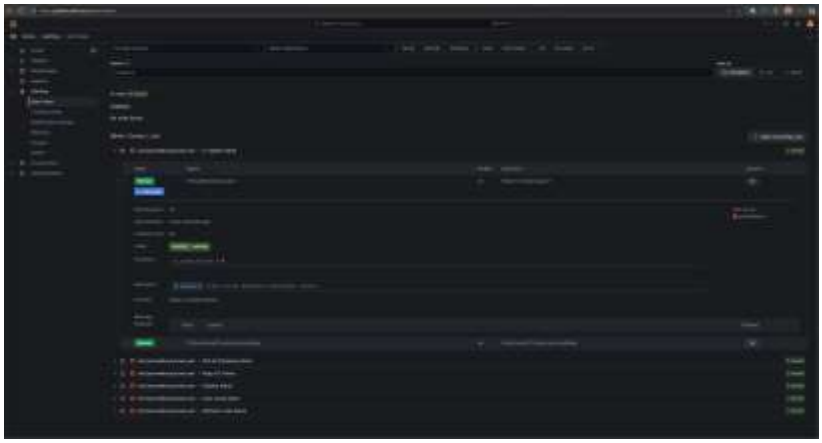
Клієнтський інтерфейс. Сторінка пайплайнів у GitLab



Клієнтський інтерфейс. Дашборд GitLab у Grafana

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

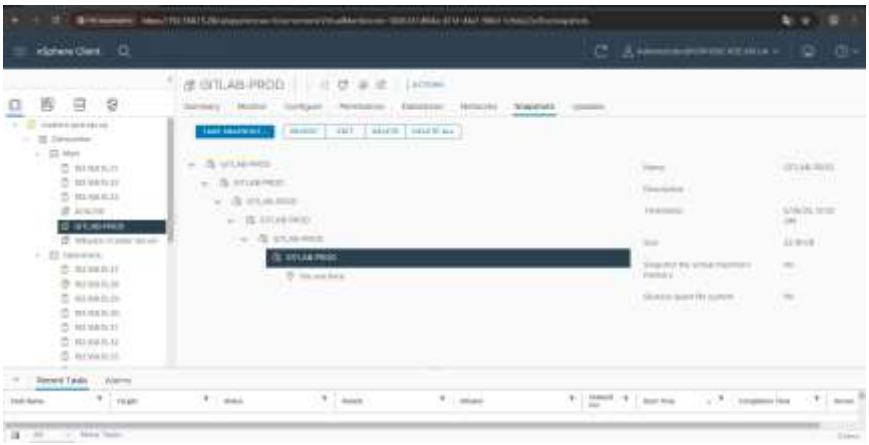
Клієнтський інтерфейс. Правила алертів у Grafana



Клієнтський інтерфейс. Перегляд GitLab логів в OpenSearch Dashboards

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Клієнтський інтерфейс. Перегляд снапшотів GitLab VM у vCenter



Використані технології

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Тестування та супровід

Тестування розробленої системи проводилося виключно в ручну, а саме, було протестовано:

- Запуск і виконання пайплайнів у GitLab CI/CD
- Роботу GitLab Runner
- Розгортання інфраструктури через Terraform
- Збір метрик через Prometheus
- Побудову графіків в Grafana та їх актуальність
- Спрацьовування алертів та сповіщення в Telegram
- Збір логів Filebeat з GitLab та Runner
- Обробку логів через Logstash
- Пошук і фільтрацію логів в OpenSearch
- Автоматичне створення снапшотів у vCenter
- Відновлення віртуалок із створених снапшотів

Висновки

- Розроблено CI/CD-інфраструктуру на базі GitLab, яка автоматизує процеси збірки, тестування та розгортання. Система побудована за принципами DevOps і Infrastructure as Code, що забезпечує її гнучкість та відтворюваність.
- Інтегровано моніторинг і логування, які дозволяють у реальному часі відстежувати стан системи, аналізувати події, а також оперативно реагувати на збої через алерти в Telegram.
- Забезпечено стійкість інфраструктури шляхом впровадження механізмів резервного копіювання та аварійного відновлення через vCenter, що дозволяє швидко повертати систему до працездатного стану.
- Результати роботи довели ефективність комплексного підходу, що поєднує автоматизацію, контроль і самовідновлення. Запропоноване рішення може бути масштабоване та адаптоване до потреб різних організацій.