МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

**ЯВОРСЬКИЙ МАКСИМ ОЛЕКСАНДРОВИЧ**

Допускається до захисту:
В. о. завідувача кафедри
інформаційних технологій,
к. т. н, доцент
_____Оксана ЗЕЛІНСЬКА
«____»_____ 2024 р.

# РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ ДЛЯ МУЛЬТИХМАРНОЇ ПЛАТФОРМИ ЗЕРНОВИХ ЕЛЕВАТОРІВ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (магістерська) робота

Керівник:
Петро НІКОЛЮК,
професор кафедри інформаційних
технологій

Оцінка: ___/___/_____
(бали за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2024

## АНОТАЦІЯ

**Яворський М.О. Розробка серверної частини для мультихмарної платформи зернових елеваторів.** Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні технології обробки даних», Донецький національний університет імені Василя Стуса, Вінниця, 2024.

У кваліфікаційній (магістерській) роботі досліджено проблему оптимізації управління зерновими елеваторами шляхом використання мультихмарних платформ. Проаналізовано сучасні хмарні технології, алгоритми інтеграції та забезпечення безпеки, а також інструменти для розробки серверної частини на основі Java, Spring, Hibernate та інших технологій. Розроблено серверну частину платформи, яка забезпечує масштабованість, надійність і високу продуктивність у мультихмарному середовищі.

67 с., 3 рис., 1 дод., 25 джерел. Ключові слова: мультихмарність, зернові елеватори, серверна частина, Java, Spring, Postgres.

## ABSTRACT

**Yavorskyi M.O. Development of the server side for a multi-cloud platform for grain elevators.** Specialization 122 "Computer Science", educational program "Data Science", Vasyl' Stus Donetsk National University, Vinnytsia, 2024.

The qualification (master's) thesis investigates the problem of optimizing the management of grain elevators using multi-cloud platforms. The research analyzes modern cloud technologies, integration algorithms, security assurance, and development tools for server-side architecture based on Java, Spring, Hibernate, and other technologies. A server-side platform was developed to ensure scalability, reliability, and high performance in a multi-cloud environment.

67 p., 3 figures., 1 appendices., 25 references. Keywords: multi-cloud, grain elevators, server-side, Java, Spring, Postgres.

**ЗМІСТ**

ВСТУП ..... 6
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ..... 9
1.1 Історія аграрної промисловості ..... 9
1.1.1 Розвиток аграрної промисловості ..... 9
1.1.2 Виникнення та розвиток зернових елеваторів ..... 9
1.1.3 Еволюція ІТ-рішень у контексті управління зерновими елеваторами ..... 11
1.1.4 Розвиток зернових елеваторів: проблеми та рішення ..... 12
1.2 Використання інформаційних технологій у аграрній промисловості ... 13
1.2.1 Історія впровадження ІТ у аграрному секторі ..... 13
1.2.2 Сучасні ІТ-рішення для управління агропромисловими об'єктами 14
1.2.3 Інформаційні системи для зернових елеваторів ..... 15
1.3 Потреба у хмарних технологіях для зернових елеваторів ..... 15
1.3.1 Виклики традиційних ІТ-рішень для елеваторів ..... 15
1.3.2 Переваги хмарних технологій ..... 16
1.3.3 Впровадження мультихмарної платформи для зернових елеваторів ..... 17
1.4 Огляд існуючих рішень для управління зерновими елеваторами ..... 18
1.4.1 Традиційні ІТ-системи для елеваторів ..... 18
1.4.2 Хмарні платформи для аграрного сектору ..... 18
1.5 Висновки ..... 19
РОЗДІЛ 2 ОГЛЯД ІСНУЮЧИХ ПРОГРАМНИХ ЗАСОБІВ ДЛЯ РЕАЛІЗАЦІЇ ПОСТАВЛЕНОЇ ЗАДАЧІ ..... 20
2.1 Вибір та огляд Java(Spring + Hibernate) для реалізації серверної частини ..... 20
2.1.1 Огляд платформи Java ..... 20
2.1.2 Огляд Spring Framework ..... 21

2.1.3 Огляд Hibernate ..... 22
2.1.4 Архітектура серверної частини на Java (Spring + Hibernate) ..... 23
2.1.5 Реальні кейси використання Java, Spring і Hibernate ..... 23
2.1.6 Взаємодія Java (Spring + Hibernate) у мультихмарній платформі ... 24
2.2 Огляд технології Postgres ..... 25
2.2.1 Історія розвитку Postgres ..... 25
2.2.2 Архітектура та принципи роботи ..... 26
2.2.3 Особливості роботи та функціональність ..... 26
2.2.4 Переваги використання Postgres для мультихмарних платформ .... 28
2.2.5 Робота з великою кількістю з'єднань та обробка великих обсягів даних ..... 29
2.2.6 Особливості зберігання та оптимізації в Postgres ..... 30
2.2.7 Недоліки та обмеження Postgres ..... 31
2.2.8 Додаткові функції PostgreSQL для мультихмарних платформ ..... 31
2.3 Огляд технології Hazelkast ..... 32
2.3.1 Історія розвитку Hazelcast ..... 32
2.3.2 Основні компоненти та архітектура ..... 33
2.3.3 Функціональність Hazelcast ..... 34
2.3.4 Безпека, маштабованість та відмовостійкість в Hazelcast ..... 35
2.3.5 Інтеграція з іншими технологіями ..... 36
2.3.6 Переваги та недоліки Hazelcast ..... 37
2.3.7 Порівняння з іншими технологіями ..... 38
2.3.8 Використання в реальних проектах ..... 39
2.3.9 Висновок ..... 40
2.4 Огляд технології Grafana ..... 41
2.4.1 Загальні відомості про Grafana ..... 41
2.4.2 Основні компоненти архітектури Grafana ..... 42
2.4.3 Візуалізація даних у Grafana ..... 43

2.4.4 Алерти та сповіщення ..... 44
2.4.5 Використання Grafana в різних галузях ..... 45
2.4.6 Переваги та недоліки Grafana ..... 45
2.4.7 Висновок ..... 46
2.5 Огляд технології RabbitMq ..... 47
2.5.1 Введення в RabbitMQ ..... 47
2.5.2 Архітектура RabbitMQ ..... 48
2.5.3 Основні можливості RabbitMQ ..... 49
2.5.4 Інтеграція RabbitMQ з різними технологіями ..... 50
2.5.5 Використання RabbitMQ в реальних проєктах ..... 50
2.5.6 Переваги та недоліки RabbitMQ ..... 51
2.5.7 Висновок ..... 52
2.6 Висновок ..... 52
РОЗДІЛ 3 ЕТАПИ РОЗРОБКИ СЕРВЕРНОЇ ЧАСТИНИ МУЛЬТИХМАРНОЇ ПЛАТФОРМИ ДЛЯ ЗЕРНОВИХ ЕЛЕВАТОРІВ ..... 54
3.1 Аналіз вимог до системи ..... 54
3.2 Розробка архітектури та структури системи ..... 55
3.3 Реалізація основних модулів ..... 55
3.3.1 Підключення до бази даних PostgreSQL ..... 55
3.3.2 Реалізація обміну даними через RabbitMQ ..... 57
3.3.3 Інтеграція моніторингу через Grafana ..... 60
3.4 Тестування та оптимізація ..... 62
3.4.1 Види тестування ..... 62
3.4.2 Оптимізація ..... 63
3.5 Інтеграція з мультихмарним середовищем ..... 64
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ..... 66

# ВСТУП

У сучасному аграрному секторі, де точне управління запасами і оптимізація логістичних процесів відіграють ключову роль, зернові елеватори виступають важливою ланкою в ланцюзі постачання. З метою підвищення ефективності їх роботи і забезпечення надійного зберігання зерна, багато компаній впроваджують інформаційні технології, зокрема мультихмарні платформи. Такі платформи дозволяють керувати елеваторами на віддалених територіях, інтегрувати різні системи і забезпечувати доступ до даних в режимі реального часу.

Розробка серверної частини для мультихмарної платформи зернових елеваторів є важливим етапом цього процесу. Серверна частина відповідає за обробку запитів від користувачів, взаємодію з базами даних, управління інтерфейсами прикладного програмування (API) та забезпечення безпеки даних. Вона повинна бути масштабованою, надійною і здатною інтегруватися з різними хмарними сервісами.

**Актуальність проблеми**

Сучасні елеватори потребують систем, які здатні швидко реагувати на зміну умов, такі як коливання в урожайності, погодні умови та потреби ринку. Традиційні IT-системи часто не здатні забезпечити необхідну гнучкість та масштабованість. В цьому контексті мультихмарні платформи представляють собою перспективне рішення. Вони дозволяють використовувати ресурси кількох хмарних провайдерів, забезпечуючи тим самим високу доступність і відмовостійкість системи.

**Основні виклики**

При розробці серверної частини мультихмарної платформи для зернових елеваторів стоїть низка викликів:

1. Інтеграція з різними хмарними сервісами: Платформа повинна підтримувати інтеграцію з різними хмарними провайдерами (AWS, Google Cloud, Azure) для використання їхніх ресурсів.

2. Масштабованість: Система повинна мати можливість розширюватись у відповідь на збільшення обсягу даних та кількості користувачів.

3. Безпека: Забезпечення конфіденційності та цілісності даних, захист від несанкціонованого доступу.

4. Надійність: Висока доступність сервісу і мінімізація часу простою.

5. Продуктивність: Висока швидкість обробки запитів і мінімальні затримки в доступі до даних.

**Мета і завдання**

Метою розробки серверної частини є створення ефективної, надійної і безпечної платформи для управління зерновими елеваторами, яка здатна працювати в мультихмарному середовищі. Основними завданнями є:

- Розробка архітектури серверної частини, яка підтримує мультихмарність.
- Реалізація інтеграції з основними хмарними провайдерами.
- Забезпечення безпеки даних і захист від кібератак.
- Тестування і оптимізація продуктивності системи.
- Впровадження системи моніторингу та управління для підтримки стабільної роботи платформи.

Компанія "Інновіннпром" займається розробкою інноваційних рішень для агропромислового комплексу. У компанії я займаюся магістерською а зокрема розробкою серверної частини для мультихмарної платформи, яка оптимізує управління зерновими елеваторами. Я використовую Java для створення архітектури системи, інтегрую хмарні сервіси (AWS, Google Cloud, Azure) і застосовую контейнеризацію (Docker, Kubernetes) для легкого

розгортання та масштабування. Моя робота підвищує ефективність і продуктивність елеваторів.

Розробка серверної частини для мультихмарної платформи зернових елеваторів є складним, але необхідним кроком для модернізації аграрного сектору. Вона відкриває нові можливості для ефективного управління запасами і логістикою, сприяє підвищенню продуктивності та конкурентоспроможності аграрних підприємств. Успішна реалізація цього проекту потребує всебічного підходу, врахування сучасних технологічних тенденцій і викликів, а також ретельного планування і тестування всіх компонентів системи.

# РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

## 1.1 Історія аграрної промисловості

### 1.1.1 Розвиток аграрної промисловості

Аграрна промисловість є однією з найдавніших галузей людської діяльності, що бере свій початок з моменту переходу від мисливсько-збирального способу життя до землеробства. Перші цивілізації виникли саме завдяки розвитку сільського господарства, яке забезпечувало стабільний обсяг харчових продуктів.

З розвитком технологій, аграрна промисловість постійно еволюціонувала. Винахід плуга, колесо, зрошувальні системи, а також подальші інновації, такі як сівалки, жниварки і трактори, значно підвищили продуктивність землеробства. Протягом промислової революції XIX століття механізація процесів значно збільшила ефективність сільськогосподарського виробництва, сприяючи зростанню населення та урбанізації.

Сьогодні аграрна промисловість є високотехнологічною галуззю, що використовує найсучасніші досягнення науки і техніки, такі як біотехнології, генна інженерія, автоматизація та робототехніка. Важливу роль відіграє цифровізація і використання інформаційних технологій, що дозволяють оптимізувати процеси управління, моніторингу та аналізу даних.

### 1.1.2 Виникнення та розвиток зернових елеваторів

Зернові елеватори виникли в середині XIX століття як відповідь на потребу ефективного зберігання та обробки великих обсягів зерна. Перші елеватори з'явилися у США, де сільське господарство швидко розвивалося

завдяки широким просторам та сприятливим умовам для вирощування зернових культур.

Зерновий елеватор – це комплекс будівель та обладнання для зберігання, транспортування, очищення та сортування зерна. Основні функції елеватора включають прийом зерна з поля, його очищення від домішок, сушіння, зберігання та відправку на переробку або експорт. Основні етапи розвитку елеваторів включали механізацію процесів, автоматизацію та впровадження інформаційних технологій.

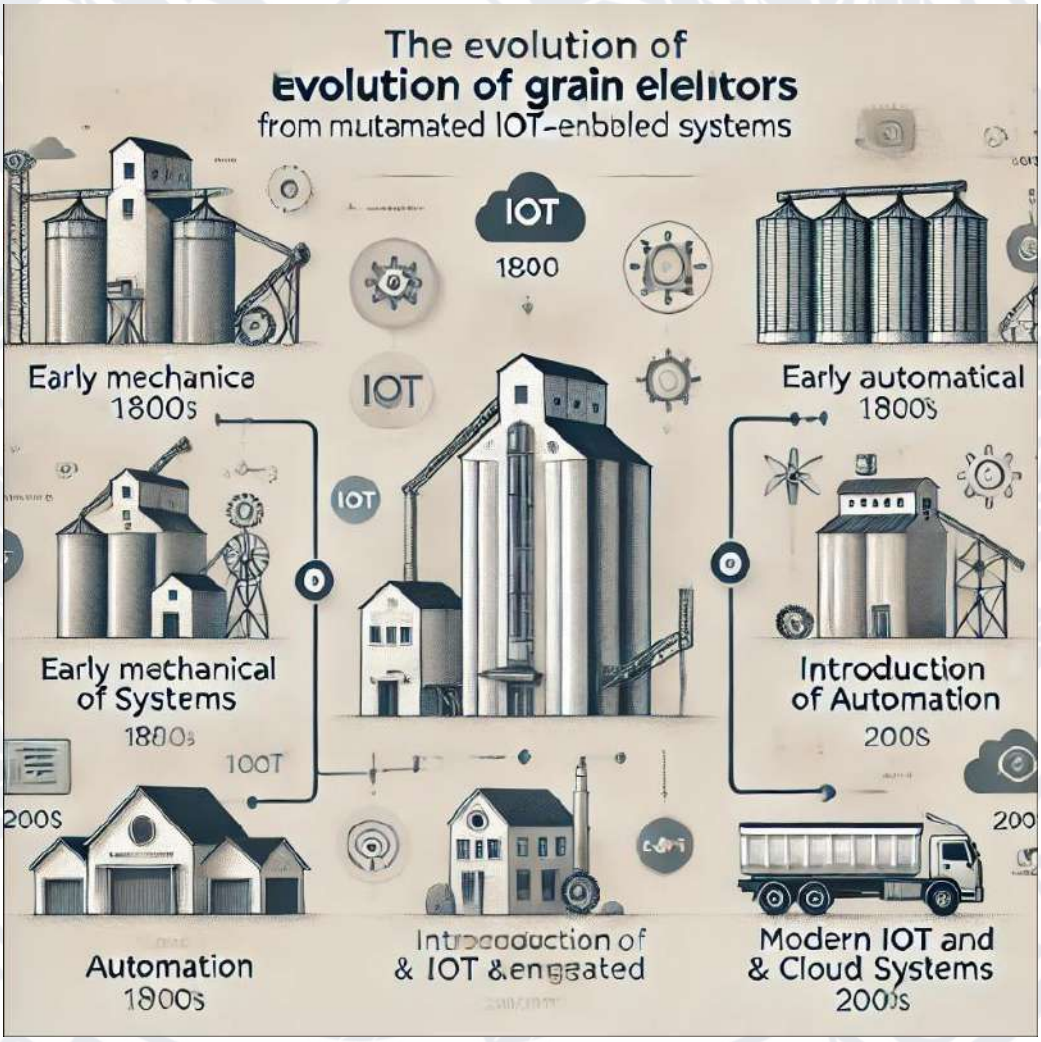


Рисунок 1.1 Еволюція зернових елеваторів: від механічних до автоматизованих IoT-систем.

Сучасні зернові елеватори використовують комп'ютеризовані системи управління для контролю за рівнем вологості, температури та станом зерна. Це дозволяє зберігати зерно в оптимальних умовах і мінімізувати втрати. Впровадження новітніх технологій, таких як Інтернет речей (IoT) та хмарні обчислення, відкриває нові можливості для підвищення ефективності та надійності елеваторів.

### 1.1.3 Еволюція ІТ-рішень у контексті управління зерновими елеваторами

Розвиток аграрної промисловості завжди супроводжувався пошуком рішень для підвищення ефективності зберігання та обробки врожаю. Зокрема, зернові елеватори історично були ключовим елементом ланцюга постачання зерна. З середини XIX століття, коли з'явилися перші механізовані елеватори в США, почався поступовий перехід від ручної праці до автоматизації.

Однак справжній прорив у функціонуванні елеваторів стався з впровадженням інформаційних технологій. У 1980-х роках почали з'являтися перші автоматизовані системи управління, які дозволяли здійснювати моніторинг і управління основними процесами (зберігання, очищення, сушіння). Згодом, у 2000-х роках, інтеграція сенсорів і систем SCADA дала можливість віддаленого контролю за станом зерна.

Сучасні рішення зосереджуються на цифровізації елеваторів, використовуючи хмарні технології, Інтернет речей (IoT) та машинне навчання. Ці технології дають змогу:

оперативно аналізувати великі обсяги даних про температуру, вологість і стан зерна;

автоматизувати прийняття рішень, таких як вентиляція чи переміщення зерна між силосами;

забезпечувати прозорість для всіх учасників ланцюга постачання.

Таким чином, еволюція ІТ-рішень для елеваторів стала основою для створення мультихмарних платформ, які пропонують гнучкість, масштабованість і інтеграцію з іншими бізнес-системами.

### 1.1.4 Розвиток зернових елеваторів: проблеми та рішення

Зернові елеватори виникли в середині XIX століття як відповідь на потребу ефективного зберігання та обробки великих обсягів зерна. Перші елеватори з'явилися у США, де сільське господарство швидко розвивалося завдяки широким просторам та сприятливим умовам для вирощування зернових культур.

Зерновий елеватор – це комплекс будівель та обладнання для зберігання, транспортування, очищення та сортування зерна. Основні функції елеватора включають прийом зерна з поля, його очищення від домішок, сушіння, зберігання та відправку на переробку або експорт. Основні етапи розвитку елеваторів включали механізацію процесів, автоматизацію та впровадження інформаційних технологій.

**Еволюція ІТ-рішень у функціонуванні елеваторів**

З впровадженням автоматизованих систем управління у 1980-х роках, зернові елеватори перейшли на якісно новий рівень. Ці системи забезпечували базовий моніторинг і управління процесами, такими як вентиляція та контроль температури зерна. У 2000-х роках поява сенсорних технологій і систем SCADA дозволила контролювати параметри зберігання в реальному часі.

Сучасні зернові елеватори оснащені IoT-сенсорами, хмарними обчисленнями та інструментами для аналітики великих даних. Це дозволяє

автоматизувати управління умовами зберігання, оптимізувати енерговитрати та прогнозувати можливі ризики псування зерна.

**Виклики у функціонуванні елеваторів:**

1. Відсутність інтегрованих платформ для обробки та зберігання даних.
2. Нестача рішень для оперативного контролю якості зерна.
3. Недостатня автоматизація енергоємних процесів, таких як вентиляція та сушіння.

Інтеграція мультихмарних платформ з використанням IoT і технологій машинного навчання стала логічним етапом у розвитку галузі. Такі платформи дозволяють ефективно поєднувати операційні процеси з аналітичними інструментами, забезпечуючи гнучкість та стабільність роботи елеваторів.

## 1.2 Використання інформаційних технологій у аграрній промисловості

### 1.2.1 Історія впровадження IT у аграрному секторі

Впровадження інформаційних технологій у аграрну промисловість розпочалося з появою перших комп'ютерних систем у середині XX століття.[1] Спочатку це були прості системи автоматизації обліку та управління, які допомагали фермерам контролювати витрати, планувати виробництво та управляти ресурсами.

З розвитком мікропроцесорної техніки та програмного забезпечення в 1980-1990-х роках з'явилися більш складні системи управління фермами, які включали функції управління врожайністю, планування сівозмін, моніторингу стану посівів та управління запасами. Використання супутникових систем GPS дозволило створити системи точного землеробства, які забезпечують точне внесення добрив та засобів захисту рослин.

У 2000-х роках розвиток Інтернету та бездротових технологій дозволив створювати мережі сенсорів для моніторингу стану полів у реальному часі, що значно підвищило ефективність управління фермерськими господарствами. Сьогодні аграрна промисловість активно використовує великі дані, машинне навчання та хмарні обчислення для оптимізації всіх етапів виробничого циклу. Мультихмарні технології дозволяють оптимізувати управління великими обсягами даних.[2]

### 1.2.2 Сучасні IT-рішення для управління агропромисловими об'єктами

Сучасні IT-рішення для управління агропромисловими об'єктами включають різноманітні платформи та програмні комплекси, що дозволяють оптимізувати управління фермерськими господарствами та елеваторами. До них належать системи точного землеробства, платформи для моніторингу та аналізу даних, системи управління ланцюгами постачань та платформи для управління агробізнесом.

Системи точного землеробства, такі як John Deere's Precision Agriculture та Trimble Agriculture, використовують GPS та сенсори для точного управління посівами, внесення добрив та засобів захисту рослин. Це дозволяє зменшити витрати на ресурси та підвищити врожайність.

Платформи для моніторингу та аналізу даних, такі як Climate FieldView та FarmLogs, збирають дані з різних джерел (сенсори, дрони, супутникові знімки) та надають аналітичні інструменти для прийняття рішень. Це дозволяє фермерам оперативно реагувати на зміни умов на полях та планувати роботу більш ефективно.[13]

Системи управління ланцюгами постачань, такі як AgriChain та AgriDigital, допомагають оптимізувати логістику та управління запасами, забезпечуючи прозорість та відстежуваність продукції від поля до споживача.

### 1.2.3 Інформаційні системи для зернових елеваторів

Інформаційні системи для зернових елеваторів, такі як GrainTrac та AGRIS, призначені для автоматизації та оптимізації процесів управління елеваторами.[19] Вони включають функції обліку зерна, управління запасами, моніторингу стану зерна, планування виробничих процесів та управління ланцюгами постачань.

GrainTrac забезпечує повний цикл управління зерновим елеватором від прийому зерна до його відправки. Система дозволяє автоматизувати процеси обліку та моніторингу, зменшуючи ризик помилок та підвищуючи ефективність роботи елеватора.

AGRIS надає інструменти для управління фінансами, продажами та логістикою, що дозволяє оптимізувати управління ланцюгами постачань та забезпечувати прозорість бізнес-процесів. Інтеграція з іншими системами дозволяє забезпечити комплексний підхід до управління елеватором.

## 1.3 Потреба у хмарних технологіях для зернових елеваторів

### 1.3.1 Виклики традиційних IT-рішень для елеваторів

Традиційні IT-рішення для управління зерновими елеваторами стикаються з низкою проблем та обмежень. Основні з них включають високу вартість впровадження та підтримки, складність масштабування та оновлення, а також обмеженість функціоналу.

Висока вартість впровадження та підтримки пов'язана з необхідністю встановлення та обслуговування локального серверного обладнання, а також з витратами на програмне забезпечення та IT-персонал. Це може бути особливо обтяжливим для малих та середніх підприємств.

Складність масштабування та оновлення традиційних IT-систем полягає в необхідності проведення складних та дорогих процедур при збільшенні обсягів даних або при впровадженні нових функціональних можливостей. Це може призводити до зниження ефективності роботи та зростання витрат.

Обмеженість функціоналу традиційних IT-рішень часто пов'язана з їх закритістю та недостатньою гнучкістю для інтеграції з іншими системами та платформами. Це ускладнює впровадження інноваційних технологій та зниження загальної ефективності управління елеватором.

### 1.3.2 Переваги хмарних технологій

Хмарні технології пропонують низку переваг, які роблять їх привабливими для впровадження у зернових елеваторах. До основних переваг належать:

**Масштабованість**: Хмарні сервіси дозволяють легко збільшувати або зменшувати ресурси залежно від потреб. Це особливо важливо для елеваторів, які можуть мати сезонні коливання у завантаженості.

**Доступність**: Хмарні платформи забезпечують доступ до даних та функцій з будь-якої точки світу, де є інтернет. Це дозволяє керувати елеватором дистанційно та в режимі реального часу.

**Безпека**: Провайдери хмарних послуг забезпечують високий рівень безпеки даних, включаючи резервне копіювання, захист від вірусів та атак, а також можливість швидкого відновлення даних у разі аварії.

**Зниження витрат**: Використання хмарних технологій дозволяє знизити витрати на IT-інфраструктуру та персонал, оскільки більшість ресурсів надається за підпискою або за принципом "оплата за використання".

**Інтеграція та гнучкість**: Хмарні платформи легко інтегруються з іншими системами та сервісами, що дозволяє створювати комплексні рішення для управління елеватором та агробізнесом в цілому.

### 1.3.3 Впровадження мультихмарної платформи для зернових елеваторів

Використання мультихмарних рішень надає додаткові переваги у порівнянні з однохмарними рішеннями. Основні з них включають:

**Відмовостійкість**: Розподіл даних та ресурсів між кількома хмарними провайдерами забезпечує високу стійкість до збоїв та аварій.

**Оптимізація витрат**: Мультихмарні рішення дозволяють вибирати оптимальні послуги від різних провайдерів, що може знизити загальні витрати на IT-інфраструктуру.

**Гнучкість та масштабованість**: Можливість вибору найбільш відповідних ресурсів для конкретних завдань забезпечує високу гнучкість та масштабованість системи.

**Безпека та відповідність**: Використання кількох провайдерів дозволяє забезпечити відповідність різноманітним стандартам та регуляторним вимогам у різних юрисдикціях.

Приклади успішного впровадження мультихмарних рішень можна знайти у багатьох галузях, включаючи фінансовий сектор, охорону здоров'я та виробництво. У аграрній промисловості мультихмарні платформи можуть забезпечити високу ефективність управління елеваторами, зменшення витрат та підвищення надійності систем.

## 1.4 Огляд існуючих рішень для управління зерновими елеваторами

### 1.4.1 Традиційні IT-системи для елеваторів

Багато елеваторів використовують застарілі IT-системи, які включають базове програмне забезпечення для ведення обліку, контролю запасів і простих операцій. Такі системи часто не інтегруються з сучасними технологіями, що створює низку проблем, зокрема:

Відсутність реального часу доступу до даних.

Складність у масштабуванні та інтеграції з іншими корпоративними системами.

Обмежена функціональність, що не дозволяє ефективно керувати всіма процесами елеватора.

### 1.4.2 Хмарні платформи для аграрного сектору

З розвитком технологій, все більше аграрних підприємств переходять на використання хмарних платформ. Хмарні рішення пропонують значні переваги:

**Масштабованість**: Хмарні платформи легко масштабуються в залежності від потреб підприємства.

**Доступність**: Дані доступні з будь-якої точки світу, що дозволяє керувати елеваторами віддалено.

**Безпека**: Сучасні хмарні платформи пропонують високий рівень захисту даних, використовуючи передові методи шифрування і аутентифікації.

## 1.5 Висновки

З аналізу предметної області стає очевидним, що аграрна промисловість, зокрема сектор зернових елеваторів, постійно еволюціонує, використовуючи передові технології для оптимізації виробничих процесів та підвищення продуктивності. Історично аграрна галузь перетворилася з простих методів обробки землі на високотехнологічну індустрію, що використовує біотехнології, автоматизацію та інформаційні технології.

Зокрема, управління зерновими елеваторами стикається з викликами традиційних IT-рішень, таких як високі витрати, складність масштабування та обмеженість функціоналу. Тут на допомогу приходять хмарні технології, які надають значні переваги у вигляді масштабованості, доступності, безпеки та зниження витрат.

Важливим кроком є впровадження мультихмарних платформ для зернових елеваторів, що дозволяє отримати додаткові переваги, такі як відмовостійкість, оптимізація витрат та гнучкість.

Отже, розвиток та використання інформаційних технологій у секторі агропромисловості, зокрема для управління зерновими елеваторами, є необхідним для підвищення продуктивності, ефективності та конкурентоспроможності галузі.

# РОЗДІЛ 2 ОГЛЯД ІСНУЮЧИХ ПРОГРАМНИХ ЗАСОБІВ ДЛЯ РЕАЛІЗАЦІЇ ПОСТАВЛЕНОЇ ЗАДАЧІ

## 2.1 Вибір та огляд Java(Spring + Hibernate) для реалізації серверної частини

### 2.1.1 Огляд платформи Java

Java була створена компанією Sun Microsystems у 1995 році. Спочатку її розробляли для інтерактивного телебачення, але вона швидко стала популярною завдяки своїй портативності через JVM (Java Virtual Machine). З моменту появи, Java еволюціонувала через численні версії та оновлення, зберігаючи свою репутацію як мова, що підтримує розробку багаторівневих додатків і серверних систем.

Java підтримує багатопотоковість, що дозволяє одночасне виконання багатьох операцій. Її кросплатформеність робить можливим розгортання додатків на будь-яких операційних системах. Java також має високу продуктивність завдяки JIT-компіляції (Just-in-Time). Крім того, у Java великий набір бібліотек і фреймворків для розробки серверних рішень.

Java має величезну екосистему, яка включає JVM для забезпечення кросплатформенності, бібліотеки для роботи з великими даними (наприклад, Apache Hadoop), інструменти для розробки мікросервісів (Spring Cloud, MicroProfile), а також підтримку хмарних рішень через інструменти для AWS, Azure і Google Cloud.

Java підходить як для малих, так і для великих проектів завдяки своїй масштабованій архітектурі. Вона дозволяє побудову систем з різним рівнем навантаження, що робить її незамінною для мультихмарних додатків, де потрібно обробляти великі обсяги запитів і даних.

### 2.1.2 Огляд Spring Framework

Spring — це фреймворк для розробки Java-додатків, що забезпечує підтримку модульності та спрощує розробку за рахунок вбудованих інструментів для роботи з базами даних, безпеки та веб-сервісів. Spring зменшує кількість «клейового коду» і забезпечує легкий доступ до різних технологій через різні модулі (Spring Core, Spring Boot, Spring MVC).

Spring підтримує масштабованість та мікросервісну архітектуру, що є ключовими для мультихмарних рішень. Модульність фреймворку дозволяє налаштувати його під конкретні потреби додатка, а його підтримка REST API дозволяє легко взаємодіяти з різними хмарними сервісами. Крім того, Spring Cloud пропонує інструменти для мультихмарного керування ресурсами.

Dependency Injection (DI) — це механізм, який дозволяє ін'єктувати залежності в компоненти, замість того, щоб вони створювали їх самостійно. IoC дозволяє краще організувати код, зменшивши залежність компонентів від конкретних реалізацій, що робить код більш тестованим і модульним.

Spring Boot — це мікрофреймворк, що спрощує налаштування та запуск Java-додатків. Він автоматично конфігурує додаток на основі залежностей, зменшуючи кількість ручного налаштування та дозволяючи швидко запустити прототип або повноцінний сервіс.

Spring WebFlux дозволяє створювати реактивні додатки, що можуть ефективно обробляти тисячі запитів одночасно. Це дуже корисно для мультихмарних систем, де потрібна швидка обробка даних від численних IoT-пристроїв в реальному часі.

Spring підтримує розподілені системи через мікросервіси та модулі Spring Cloud, що робить його ідеальним для побудови мультихмарних

додатків, забезпечуючи керування потоками даних та комунікацію між різними сервісами.

### 2.1.3 Огляд Hibernate

ORM — це методологія, що дозволяє працювати з базами даних як з об'єктами в програмному коді. Hibernate — один з найпопулярніших ORM-фреймворків для Java, який автоматизує трансформацію даних між об'єктами та реляційними базами даних.

Hibernate автоматично генерує SQL-запити та обробляє результат, що знижує необхідність ручного написання SQL-коду. Крім того, Hibernate має потужну підтримку кешування, що підвищує продуктивність системи. Його підтримка складних запитів (HQL) полегшує роботу з великими наборами даних.

Spring легко інтегрується з Hibernate через модуль Spring ORM. Це спрощує налаштування взаємодії з базами даних та забезпечує більш гнучке управління транзакціями. Використання Spring із Hibernate дозволяє швидко створювати та розгортати серверні додатки, які потребують взаємодії з базами даних.

Другий рівень кешування у Hibernate підвищує продуктивність системи, зберігаючи дані між сесіями користувачів. Це особливо корисно для мультихмарних додатків, оскільки зменшує навантаження на бази даних при великій кількості одночасних запитів.

Lazy Loading дозволяє завантажувати дані з бази лише тоді, коли це необхідно, що зменшує обсяги пам'яті та підвищує продуктивність у великих системах, таких як мультихмарні додатки з великою кількістю таблиць і даних.

### 2.1.4 Архітектура серверної частини на Java (Spring + Hibernate)

Серверні додатки зазвичай мають шарову архітектуру (Presentation Layer, Business Logic Layer, Data Access Layer). Spring забезпечує інтеграцію між цими рівнями за допомогою інструментів, таких як Spring MVC для керування запитами, і Hibernate для роботи з базами даних.

Spring Security — це потужний модуль Spring, який забезпечує захист додатків на рівні доступу та автентифікації. Він легко інтегрується з іншими модулями Spring і підтримує різні стратегії безпеки, такі як OAuth2, JWT (JSON Web Tokens), та інші.

Для мультихмарних платформ важливо мати правильну маршрутизацію запитів між мікросервісами. Використання API Gateway дозволяє централізовано управляти запитами, безпекою та балансуванням навантаження, що підвищує масштабованість та надійність системи.

Java-серверні додатки часто розгортаються за допомогою контейнеризації (Docker) і оркестрації (Kubernetes), що спрощує управління ресурсами та автоматизацію розгортання додатків у мультихмарних середовищах.

### 2.1.5 Реальні кейси використання Java, Spring і Hibernate

Java (Spring + Hibernate) використовується в численних компаніях для побудови масштабованих серверних додатків. Наприклад, Netflix використовує Spring для підтримки своєї масштабованої архітектури мікросервісів. Ця технологія дозволяє їм ефективно обслуговувати мільйони користувачів у глобальних масштабах.

Spring та Hibernate є ідеальними для мультихмарних платформ завдяки своїй масштабованості та інтеграції з різними хмарними провайдерами через Spring Cloud. Це дозволяє легко розподіляти навантаження між кількома хмарними середовищами, зберігаючи продуктивність і відмовостійкість системи.

Java активно використовується для роботи з великими обсягами даних в таких проектах як Hadoop або Apache Spark, що може бути корисним при розробці мультихмарних платформ для обробки IoT-даних у реальному часі.

Java-системи легко інтегруються з провайдерами хмарних послуг через SDK та бібліотеки для роботи з сервісами AWS, Azure, Google Cloud, що робить їх ідеальними для мультихмарних рішень.

### 2.1.6 Взаємодія Java (Spring + Hibernate) у мультихмарній платформі

Java, у поєднанні зі Spring Framework та Hibernate ORM, забезпечує інтеграцію бізнес-логіки із зберіганням і обробкою даних. Основна роль цих технологій полягає в управлінні потоками запитів, доступом до баз даних і забезпеченні масштабованості.

**Робота Spring Framework**

Spring Framework дозволяє будувати мікросервісну архітектуру, де кожен сервіс відповідає за конкретну функцію (наприклад, обробку даних сенсорів, зберігання або моніторинг). Основною перевагою Spring є його модульність: залежно від потреб системи можна додавати лише ті модулі, які необхідні, наприклад:

- **Spring Boot** для швидкого налаштування сервісів.
- **Spring Security** для забезпечення безпеки запитів.
- **Spring Data** для роботи з базами даних.

**Роль Hibernate ORM**

Hibernate ORM автоматизує перетворення даних із реляційної бази у вигляді об’єктів, що дозволяє уникнути ручного написання SQL-запитів. Це особливо важливо для систем, що працюють із великими обсягами даних, оскільки Hibernate оптимізує взаємодію з базою, зменшуючи затримки при запитах. Hibernate також підтримує кешування, що знижує навантаження на базу даних у випадках, коли одні й ті ж дані запитуються багаторазово.

**Переваги інтеграції у мультихмарній платформі**

Spring і Hibernate забезпечують масштабованість і швидкість обробки, дозволяючи створювати відмовостійкі системи. Наприклад, у мультихмарному середовищі Spring Cloud може розподіляти запити між кількома хмарними провайдерами, такими як AWS і Azure, залежно від доступності ресурсів.

## 2.2 Огляд технології Postgres

### 2.2.1 Історія розвитку Postgres

PostgreSQL бере свій початок із проєкту Ingres, розробленого у 1980-х роках у Каліфорнійському університеті Берклі. Його основною метою було створення потужної системи керування базами даних (СКБД), яка підтримує обробку транзакцій і забезпечує високу надійність. Пізніше проєкт Ingres еволюціонував у Postgres, з підтримкою сучасних можливостей, таких як ACID-транзакції, обробка великих обсягів даних, а також краща підтримка розширюваності, що дозволило створювати власні типи даних, методи індексації, функції.

PostgreSQL підтримується глобальною спільнотою розробників, що забезпечує швидке впровадження нових функцій, виправлення помилок та

багаторічну стабільність. Постійне оновлення та підтримка індустріальних стандартів роблять його одним із найбільш надійних інструментів для роботи з даними в корпоративних і хмарних середовищах.

### 2.2.2 Архітектура та принципи роботи

Postgres має клієнт-серверну архітектуру, де клієнти підключаються до сервера бази даних для виконання SQL-запитів. Сервер PostgreSQL відповідає за обробку запитів, транзакцій і збереження даних на диску. Він використовує багатопотокову обробку для одночасної роботи з кількома клієнтами, забезпечуючи високу ефективність і масштабованість. Основні компоненти архітектури PostgreSQL включають процеси керування буферами, фонову обробку, індексацію даних та журнал транзакцій (Write-Ahead Logging — WAL), що забезпечує відновлення даних у випадку збою.

Postgres використовує оптимізовану систему блокувань для управління паралельною роботою з даними, що знижує рівень конфліктів при одночасному доступі до бази. Він також підтримує розширення, що дозволяють розробникам інтегрувати нові функції без зміни ядра системи, такі як створення нових типів індексів або операторів.

### 2.2.3 Особливості роботи та функціональність

**ACID-транзакції:** PostgreSQL повністю підтримує принципи ACID (атомарність, консистентність, ізольованість, надійність), що забезпечує надійну обробку транзакцій. [4]

PostgreSQL підтримує паралельне виконання запитів, що дозволяє значно підвищити швидкість обробки складних запитів. Це особливо корисно,

коли потрібно виконати великі аналітичні операції, наприклад, розрахунок середньої температури за тиждень, який містить великі обсяги даних.

Крім того, PostgreSQL має розширену підтримку типів даних, таких як JSON і JSONB, що дозволяє працювати з напівструктурованими даними. Ця функція є важливою при роботі з IoT-даними, оскільки часто потрібно зберігати та аналізувати дані, що надходять із сенсорів у форматі JSON.

PostgreSQL підтримує тригери та процедури для автоматизації задач, що дозволяє значно знизити навантаження на користувачів і автоматизувати обробку даних. Наприклад, коли температура перевищує певний поріг, можна автоматично створювати записи в таблиці для подальшої обробки або аналізу.

**Масштабованість та продуктивність:** За рахунок підтримки індексів B-Tree, GiST, GIN та BRIN, а також паралельного виконання запитів, PostgreSQL забезпечує високу продуктивність навіть при великих обсягах даних.

**Типи даних та JSON:** Система має розширений набір типів даних, включаючи JSON і JSONB, що дозволяє використовувати PostgreSQL для обробки як реляційних, так і нереляційних даних, що стає особливо важливим у сучасних мультихмарних платформах.

**Транзакційні блоки:** PostgreSQL дозволяє виконувати складні транзакції в межах одного запиту або декількох запитів, що підвищує надійність обробки даних у мультихмарних архітектурах.

PostgreSQL підтримує реляційні таблиці, але також дозволяє зберігати дані в неструктурованих формах (JSON, JSONB), що робить його універсальним для обробки різних типів даних. Інші ключові функції включають підтримку складних типів даних, таких як масиви, можливість створення тригерів, функцій та збережених процедур, а також багаторівневу систему безпеки на основі ролей і доступів.

Завдяки підтримці JSON, PostgreSQL стає конкурентом багатьом NoSQL-рішенням, оскільки поєднує у собі реляційну структуру з можливостями зберігання неструктурованих даних. Це дозволяє спрощувати архітектуру системи, коли необхідно працювати з гібридними моделями даних.

### 2.2.4 Переваги використання Postgres для мультихмарних платформ

**Масштабованість**: Postgres має вбудовану підтримку горизонтального та вертикального масштабування, що дозволяє адаптувати систему до великих обсягів даних у мультихмарних середовищах.

PostgreSQL підтримує горизонтальне масштабування через партиціювання таблиць, що дозволяє розподіляти дані по різних серверах або вузлах. Це важливо для великих систем, де є потреба в обробці великих обсягів даних за мінімальний час.

Індексація, зокрема використання B-Tree, GiST, GIN та BRIN індексів, дозволяє ефективно працювати з даними різних типів. Для даних сенсорів це важливо, оскільки індекси допомагають прискорити пошук і фільтрацію даних за часовими інтервалами або за іншими критеріями.

Пули з'єднань також допомагають PostgreSQL ефективно обробляти велику кількість одночасних запитів, що є важливим для мультихмарних платформ, де кількість запитів може бути дуже великою.

**Реплікація даних**: Підтримка асинхронної та синхронної реплікації робить його ідеальним вибором для мультихмарних архітектур, де важливо забезпечити збереження та узгодженість даних між кількома дата-центрами.

**Підтримка кількох інстансів**: PostgreSQL дозволяє легко створювати кілька інстансів баз даних у різних хмарах або фізичних локаціях, що забезпечує відмовостійкість і оптимізує роботу з ресурсами.

**Зручність автоматизації**: Підтримка різних інструментів автоматизації (Ansible, Docker, Kubernetes) дозволяє легко інтегрувати Postgres у CI/CD-процеси, що підвищує ефективність управління даними в мультихмарних середовищах.

**Безпека**: Постгрес має розширені можливості для шифрування даних (як в стані спокою, так і при передачі) і автентифікації користувачів, що є важливим для хмарних рішень, де безпека є одним з ключових факторів.

Перевагою реплікації у PostgreSQL є можливість використовувати hot standby, де репліковані вузли можуть приймати запити на читання, що покращує продуктивність системи і знижує навантаження на основний вузол.

### 2.2.5 Робота з великою кількістю з'єднань та обробка великих обсягів даних

PostgreSQL ефективно працює з великою кількістю одночасних підключень завдяки використанню пулів з'єднань та підтримці процесів обробки запитів паралельно.[14] В умовах мультихмарних платформ, де кількість підключень може значно варіюватися, така функціональність допомагає ефективно розподіляти навантаження.

Для роботи з великими обсягами даних PostgreSQL використовує систему таблиць розділів (partitioning), що дозволяє розподіляти дані між кількома серверами або вузлами. Це підвищує ефективність запитів та швидкість обробки даних.

PostgreSQL пропонує різноманітні типи індексів, включаючи B-дерева, GiST, GIN та BRIN індекси, що дозволяє оптимізувати запити для різних типів

даних і випадків використання. Для великих обсягів даних це забезпечує ефективне сортування та пошук. Індекси на JSON-дані також підтримуються через GIN.

BRIN індекси є унікальними для PostgreSQL і призначені для дуже великих таблиць, де кожен індексний запис представляє діапазон значень, що забезпечує компроміс між швидкістю пошуку та збереженим обсягом даних.

### 2.2.6 Особливості зберігання та оптимізації в Postgres

Postgres підтримує зберігання неструктурованих даних, таких як JSON, що робить його потужним інструментом для обробки різноманітних типів даних.[3] Також система надає можливість оптимізації запитів завдяки вбудованому планувальнику запитів (query planner), що дозволяє аналізувати запити та вибирати найефективніший план виконання

PostgreSQL добре інтегрується з мультихмарними платформами, оскільки є незалежним від конкретного провайдера і може бути розгорнутим як на власних серверах, так і в публічних або приватних хмарах. Можливість інтеграції з Kubernetes, контейнеризації та автоматизованих інструментів управління інфраструктурою, таких як Terraform або Ansible, дозволяє гнучко налаштовувати та автоматизувати розгортання.

Завдяки тому, що PostgreSQL підтримує хмарні сервіси різних провайдерів (AWS RDS, Google Cloud SQL, Azure Database for PostgreSQL), можна легко створювати резервні копії та відновлювати базу на різних платформах. Крім того, багатоплатформна сумісність робить його вибором для складних мультихмарних систем.

PostgreSQL використовує Write-Ahead Logging (WAL), що дозволяє забезпечити цілісність даних навіть у разі збоїв системи. Це важливо для

збереження надійності в мультихмарних середовищах, де можуть виникати відмови окремих компонентів."

Також система має вбудований планувальник запитів (query planner), який дозволяє вибирати оптимальний шлях виконання запиту. Це забезпечує високу продуктивність при обробці складних аналітичних запитів, таких як агрегація або обчислення середніх значень за великими даними."

Кешування даних у PostgreSQL забезпечує швидкий доступ до часто використовуваних даних, що знижує навантаження на дискову систему та покращує продуктивність."

### 2.2.7 Недоліки та обмеження Postgres

Незважаючи на свою потужність, PostgreSQL має і свої недоліки. Наприклад, складність у налаштуванні для великомасштабних систем, порівняно нижча продуктивність при роботі з OLTP-запитами у порівнянні з іншими спеціалізованими СКБД, і обмеженість у підтримці деяких специфічних комерційних функцій (наприклад, автоматичний шардінг, що підтримується у NoSQL рішеннях). Однак, вибір PostgreSQL для мультихмарної платформи зумовлений його універсальністю, масштабованістю, активною спільнотою, і можливістю інтеграції з різними хмарними рішеннями, що робить його більш економічно вигідним та менш залежним від конкретного хмарного провайдера.[20]

### 2.2.8 Додаткові функції PostgreSQL для мультихмарних платформ

1. **Реплікація та резервне копіювання:**

PostgreSQL підтримує як синхронну, так і асинхронну реплікацію, що дозволяє зберігати копії даних у різних хмарних середовищах. Наприклад, дані можуть бути репліковані з основного вузла в AWS до вторинного вузла в Google Cloud.

2. **Підтримка кількох схем:**

PostgreSQL дозволяє створювати кілька схем в одній базі, що полегшує організацію даних для різних сервісів у мультихмарній системі.

3. **Розширення:**

PostgreSQL підтримує розширення, такі як PostGIS для геопросторових даних або Citus для масштабування горизонтального шардінгу, що робить його універсальним інструментом для різних завдань.

## 2.3 Огляд технології Hazelkast

### 2.3.1 Історія розвитку Hazelcast

Hazelcast була створена у 2008 році з метою побудови розподіленої кешуючої системи для Java-програм. Розроблена як рішення з відкритим вихідним кодом, Hazelcast швидко набула популярності завдяки своїй легкості та гнучкості. Система була розроблена для масштабування, підтримки високих навантажень та надання надійного механізму кешування і зберігання даних у пам'яті. Поступово Hazelcast розширила свої можливості, надаючи повноцінну підтримку розподілених обчислень, що дозволило їй перетворитися на одну з найпотужніших платформ для розподіленої обробки даних і обміну повідомленнями у реальному часі.

Hazelcast на початковому етапі орієнтувалася на кешування даних у пам'яті для Java-додатків, що дозволило досягнути високої швидкості доступу

до часто використовуваних даних без необхідності звертатися до традиційних баз даних. Це також дозволяло масштабувати систему без необхідності переписувати додатки.

### 2.3.2 Основні компоненти та архітектура

Hazelcast реалізує концепцію розподіленої платформи з високою відмовостійкістю. Одним з основних компонентів Hazelcast є **IMap** — розподілена реалізація структури даних «карта» (Map), яка використовується для кешування об'єктів у пам'яті між вузлами кластеру. Окрім IMap, Hazelcast підтримує інші розподілені структури даних, такі як **IQueue**, **ISet**, **IList**, а також **MultiMap** і **AtomicLong**, які використовуються для більш складних операцій і обчислень.

Ключовим аспектом архітектури Hazelcast є **кластеризація** — об'єднання кількох вузлів у кластер для обміну даними та забезпечення відмовостійкості. Hazelcast дозволяє вузлам вільно приєднуватися до кластеру або виходити з нього, зберігаючи при цьому цілісність даних завдяки автоматичному розподілу навантаження та реплікації даних між вузлами. Це дозволяє масштабувати систему відповідно до поточних вимог додатку без необхідності ручного втручання.

IMap: "IMap є розподіленою реалізацією стандартної структури даних 'Map', що зберігає дані в пам'яті. У розподіленій системі Hazelcast кожен вузол кластера зберігає частину даних, що дозволяє значно знизити затримки доступу та забезпечити високу доступність і відмовостійкість."

Розподілені структури даних: "Окрім IMap, Hazelcast підтримує інші структури даних, такі як IQueue (черга) і ISet (множина), що дозволяє зберігати дані на декількох вузлах і обробляти їх паралельно. Це дозволяє масштабувати систему відповідно до змінюваного навантаження."

Кластеризація: "Основним принципом роботи Hazelcast є кластеризація, що дозволяє автоматично реплікувати дані між вузлами кластера, забезпечуючи відмовостійкість і високий рівень доступності."

### 2.3.3 Функціональність Hazelcast

Hazelcast надає широкий набір функцій для розробників, серед яких:

Кешування даних у пам'яті — одна з основних функцій, яка забезпечує швидкий доступ до часто використовуваних даних. Hazelcast підтримує TTL (Time to Live) для даних, що дозволяє автоматично видаляти застарілі об'єкти з кешу.

Розподілені транзакції — підтримка транзакційного виконання операцій з можливістю відновлення стану після відмови одного з вузлів.

Розподілені події — система обміну повідомленнями та подіями, яка дозволяє вузлам реагувати на певні зміни в даних у режимі реального часу.

Потокова обробка — Hazelcast підтримує інтеграцію з технологіями обробки потоків, такими як Hazelcast Jet, що забезпечує ефективну обробку великих обсягів даних у режимі реального часу.

Системи черг та черги завдань — за допомогою структур даних, таких як IQueue, Hazelcast дозволяє реалізовувати черги завдань для обробки паралельних задач, що важливо для багатозадачних програм.

Кешування в пам'яті: Hazelcast підтримує кешування даних в пам'яті з автоматичним керуванням життєвим циклом елементів даних через параметр TTL (Time to Live). Це дозволяє автоматично видаляти застарілі об'єкти з кешу, підтримуючи актуальність даних у системі.

Розподілені події: Система обміну повідомленнями та подіями в Hazelcast дозволяє вузлам автоматично реагувати на зміни даних у реальному

часі. Наприклад, якщо температура сенсора змінюється, Hazelcast може сповістити інші частини системи або виконати автоматичні операції.

Потокова обробка: Завдяки інтеграції з Hazelcast Jet, система забезпечує обробку даних у реальному часі, що є важливим для застосувань, таких як моніторинг температури в реальному часі або обробка великих потоків даних від IoT-сенсорів.

### 2.3.4 Безпека, маштабованість та відмовостійкість в Hazelcast

Hazelcast має вбудовані механізми безпеки для захисту даних та взаємодії між вузлами. Система підтримує **шифрування даних** під час передачі між вузлами та клієнтами, а також надає можливість використання **сертифікатів SSL/TLS** для забезпечення конфіденційності інформації. Окрім цього, Hazelcast надає можливість налаштування **аутентифікації та авторизації** доступу до різних елементів кластера, що гарантує безпечний доступ до розподілених ресурсів.

Hazelcast відома своєю високою **масштабованістю** завдяки архітектурі, що підтримує горизонтальне масштабування. Додавання нових вузлів у кластер дозволяє розподіляти навантаження на обробку та зберігання даних без впливу на продуктивність. Завдяки **реплікації даних** між вузлами кластеру, Hazelcast забезпечує високий рівень **відмовостійкості**. Якщо один з вузлів виходить з ладу, інші вузли автоматично підхоплюють його роботу без втрати даних і переривання сервісу.

Hazelcast також дозволяє використовувати **сегментацію** даних, що підвищує ефективність зберігання і забезпечує оптимальний баланс між доступністю та продуктивністю. Система підтримує автоматичне відновлення даних після аварій, зменшуючи час простою і мінімізуючи ризик втрат інформації.

Масштабованість: Hazelcast підтримує горизонтальне масштабування, що дозволяє безперервно збільшувати потужність системи без зупинки роботи. Додавання нових вузлів до кластера дозволяє автоматично збільшувати кількість оброблюваних запитів.

Відмовостійкість: У разі відмови одного з вузлів кластера, інші вузли автоматично підхоплюють його функції завдяки механізмам реплікації даних. Це гарантує безперервну роботу системи без втрат інформації.

Сегментація даних: Hazelcast дозволяє здійснювати сегментацію даних між різними вузлами, що допомагає оптимізувати обробку запитів і підвищити ефективність зберігання.

### 2.3.5 Інтеграція з іншими технологіями

Однією з сильних сторін Hazelcast є її здатність легко інтегруватися з іншими технологіями та інфраструктурою. Hazelcast підтримує **Spring Framework**, що робить його використання інтуїтивним для Java-розробників, які вже знайомі з екосистемою Spring. Крім того, Hazelcast може бути інтегрований з популярними хмарними платформами, такими як **Amazon Web Services (AWS)**, **Microsoft Azure** та **Google Cloud**, що робить його ідеальним вибором для мультихмарних рішень.

Інтеграція з Spring: Hazelcast має зручну інтеграцію з Spring Framework, що дозволяє Java-розробникам використовувати його без додаткових зусиль для налаштування кешування та обробки даних. Інтеграція автоматично конфігурує Hazelcast у додатку, використовуючи конфігураційні файли Spring.

Інтеграція з хмарними платформами: Hazelcast підтримує інтеграцію з AWS, Azure, та Google Cloud, що дає можливість масштабувати систему та використовувати потужності цих хмарних провайдерів для зберігання та обробки даних.

**Hazelcast Jet**, додатковий продукт Hazelcast, спеціалізується на обробці потокових даних і легко інтегрується з основною платформою Hazelcast, що дозволяє поєднувати зберігання даних у пам'яті з потужними можливостями обробки в реальному часі.

### 2.3.6 Переваги та недоліки Hazelcast

Серед переваг Hazelcast можна виділити:

- Простота у використанні: Легка інтеграція з існуючими рішеннями та зрозумілий API.
- Висока продуктивність: Завдяки зберіганню даних у пам'яті та ефективній кластеризації, Hazelcast забезпечує низькі затримки при доступі до даних.
- Гнучкість і масштабованість: Можливість легко збільшувати або зменшувати кількість вузлів кластеру для задоволення потреб програми.

Проте Hazelcast має і кілька недоліків:

- Залежність від пам'яті: Використання великого обсягу оперативної пам'яті може стати проблемою при роботі з великими обсягами даних, що може вимагати оптимізації або використання зовнішнього сховища.
- Складність налаштувань: Для великих кластерів можуть знадобитися складні налаштування і конфігурації для досягнення оптимальної продуктивності.
- Вартість ліцензії: Деякі функції Hazelcast, особливо ті, що стосуються безпеки та адміністрування, доступні лише у комерційній версії.

Попри певні недоліки, Hazelcast залишається популярним вибором серед розробників, які працюють із мультихмарними платформами та розподіленими системами. Основними причинами його вибору є:

- Масштабованість: Hazelcast забезпечує стабільну продуктивність навіть при збільшенні кількості вузлів та обсягів даних.
- Продуктивність: Використання кешування в пам'яті дозволяє досягати високих швидкостей обробки даних.
- Інтеграція: Легка інтеграція з іншими технологіями, такими як Spring, Docker, Kubernetes та хмарними платформами, робить Hazelcast універсальним рішенням для широкого кола завдань.

**Переваги:**

Hazelcast забезпечує високий рівень продуктивності завдяки кешуванню даних у пам'яті та кластеризації. Це дозволяє обробляти дані з мінімальними затримками, що критично важливо для реального часу.

**Недоліки:**

Однак використання великої кількості оперативної пам'яті для зберігання даних може бути проблемним при обробці великих обсягів даних. Для таких випадків доцільно використовувати зовнішні сховища або оптимізувати кешування.

### 2.3.7 Порівняння з іншими технологіями

Для обґрунтування вибору Hazelcast варто провести порівняння з іншими технологіями, які реалізують подібні функції:

- Apache Ignite: Це ще одна система зберігання даних у пам'яті, яка надає можливості для обробки даних в реальному часі. Але Ignite має складнішу конфігурацію, а також вимагає більше ресурсів для запуску.
- Redis: Хоча Redis також відома як кешуюча технологія, вона не підтримує повноцінне функціонування кластерів з автоматичним відновленням. Redis зазвичай використовується для простіших задач, в той час як Hazelcast пропонує більш комплексні можливості обробки даних.

- Cassandra: Ця система оптимізована для роботи з великими обсягами даних у розподілених середовищах, але має вищі затримки доступу до даних порівняно з Hazelcast, особливо при випадкових запитах.

Hazelcast відрізняється легкістю використання, швидкістю та адаптивністю до потреб бізнесу, що робить її привабливою для компаній, які прагнуть до швидкого впровадження технологій.

На відміну від Apache Ignite, який має складнішу конфігурацію, Hazelcast пропонує більш просту інтеграцію та налаштування для розробників, що працюють з мікросервісами. Також Redis не підтримує автоматичне відновлення кластерів, а Hazelcast забезпечує стабільну роботу навіть при відмовах вузлів.

### 2.3.8 Використання в реальних проектах

Hazelcast знаходить широке застосування в різних галузях. Наприклад, у фінансових установах, де швидкість доступу до даних критично важлива, Hazelcast забезпечує миттєвий доступ до фінансової інформації, аналітики в реальному часі та підтримує транзакції. У сфері електронної комерції Hazelcast допомагає управляти обробкою замовлень та реєстрацією клієнтів, зберігаючи дані про сесії у пам'яті для миттєвого доступу.

Багато технологічних компаній також використовують Hazelcast для реалізації мікросервісної архітектури, де швидкість взаємодії між сервісами є критично важливою. Завдяки своїй гнучкості Hazelcast легко інтегрується з іншими компонентами екосистеми, такими як Kafka, що дозволяє будувати складні системи для обробки великих обсягів даних.

Hazelcast активно використовується в фінансових установах, де обробка транзакцій у реальному часі є критично важливою. Завдяки своїй високій продуктивності та масштабованості, Hazelcast дозволяє швидко обробляти

фінансові дані та забезпечує відмовостійкість навіть при великих навантаженнях.

У електронній комерції Hazelcast зберігає сесії користувачів у пам'яті для швидкого доступу до інформації, що дозволяє оптимізувати обробку замовлень і знижує затримки в обробці запитів.

### 2.3.9 Висновок

У зв’язку з постійним розвитком технологій, Hazelcast також активно вдосконалює свої рішення. Останні оновлення включають поліпшення продуктивності, нові функції безпеки, а також підтримку нових платформ і протоколів, таких як Kubernetes. Це дозволяє розробникам легко розгортати Hazelcast в контейнеризованих середовищах та знижує витрати на інфраструктуру.

Hazelcast є потужним інструментом для розробки розподілених додатків та управління даними в реальному часі. Його сильні сторони, такі як простота використання, висока продуктивність та здатність до масштабування, роблять його ідеальним вибором для реалізації серверної частини мультихмарних платформ. Навіть попри деякі недоліки, Hazelcast забезпечує надійні рішення для вирішення сучасних бізнес-викликів, що робить його популярним серед розробників.

Таким чином, обравши Hazelcast для своєї архітектури, можна значно покращити ефективність обробки даних, забезпечити їх безпеку і знизити ризик втрат, що особливо важливо для сучасних бізнес-інфраструктур.

## 2.4 Огляд технології Grafana

### 2.4.1 Загальні відомості про Grafana

Grafana є відкритим програмним забезпеченням, що спеціалізується на візуалізації та моніторингу даних.[5] Розроблена для того, щоб надавати користувачам можливість створювати інтерактивні дашборди, Grafana підтримує інтеграцію з безліччю джерел даних, включаючи популярні бази даних та системи моніторингу, такі як Prometheus, InfluxDB і Elasticsearch. Завдяки модульній архітектурі, Grafana дозволяє розробникам створювати плагіни, що розширюють її функціональність, надаючи можливість налаштування дашбордів під специфічні потреби користувачів. Ця технологія стала невід'ємною частиною екосистеми DevOps, оскільки дозволяє командам IT відстежувати продуктивність своїх систем, аналізувати метрики та швидко реагувати на аномалії. Важливість Grafana полягає в її здатності надавати реальний огляд даних в режимі реального часу, що є критично важливим для прийняття рішень у сучасному динамічному бізнес-середовищі.

Grafana надає потужні можливості для інтеграції з різноманітними джерелами даних, включаючи SQL бази даних (PostgreSQL, MySQL), NoSQL (Elasticsearch, MongoDB), сервіси моніторингу (Prometheus, InfluxDB), а також можливості для роботи з API для кастомних даних. Це робить Grafana універсальним інструментом для побудови централізованих дашбордів для різноманітних джерел даних в мультихмарних середовищах.

З інтеграцією IoT-сенсорів через REST API або MQTT, Grafana стає ключовим інструментом для моніторингу IoT-даних у реальному часі, що важливо для таких галузей, як агропромисловість, розподілені виробничі процеси та інші.

### 2.4.2 Основні компоненти архітектури Grafana

Архітектура Grafana складається з декількох ключових компонентів, які працюють разом, щоб забезпечити високу продуктивність та надійність:

- Сервер Grafana: Головний компонент, що відповідає за обробку запитів та з'єднання з джерелами даних. Він забезпечує авторизацію користувачів, обробляє запити на отримання даних та генерує HTML для дашбордів. Сервер може бути розгорнутий локально або у хмарі, що дозволяє організаціям гнучко управляти ресурсами.[15]
- Джерела даних: Grafana підтримує велику кількість джерел даних, включаючи SQL-бази даних, NoSQL-системи, системи моніторингу та API. Користувачі можуть налаштовувати підключення до різних джерел даних, що дозволяє інтегрувати інформацію з різних систем в одному дашборді. Це забезпечує можливість глибшого аналізу даних і візуалізації метрик.
- Клієнтський інтерфейс: Інтерфейс Grafana розроблений для зручності користувача. Користувачі можуть легко налаштовувати вигляд дашбордів, додавати та налаштовувати панелі, управляти фільтрами та періодами часу, що забезпечує динамічну взаємодію з даними. Клієнтський інтерфейс підтримує різні стилі візуалізації, від простих графіків до складних візуалізацій на основі географічних карт.

Сервер Grafana також підтримує кластеризацію, що дозволяє розподіляти навантаження на кілька серверів для високої доступності та зниження затримок при обробці запитів. Це забезпечує надійність системи в умовах високих навантажень, таких як численні запити від IoT-сенсорів у реальному часі.

Плагіни Grafana дозволяють розширювати можливості платформи, додаючи підтримку нових джерел даних, специфічних візуалізацій або інтеграцій. Наприклад, для агропромислових платформ можна створити

плагіни для інтеграції з конкретними сенсорами або підключення до спеціалізованих систем зберігання даних.

### 2.4.3 Візуалізація даних у Grafana

Grafana пропонує широкий спектр можливостей для візуалізації даних, що робить її незамінним інструментом для аналізу:

- Графіки та діаграми: Grafana має різноманітні типи графіків, включаючи лінійні, гістограми, кругові діаграми та теплові карти. Користувачі можуть налаштовувати кольори, розміри та інші елементи дизайну, щоб отримати найбільш інформативну та естетично привабливу візуалізацію. Це дозволяє легко виявляти тренди та аномалії в даних.
- Таблиці: Користувачі можуть створювати таблиці для детального представлення даних. Таблиці можуть містити числові, текстові та дата-значення, а також фільтри, що дозволяють гнучко аналізувати інформацію. Це особливо корисно для глибокого аналізу метрик та виявлення зв’язків між різними показниками.
- Спеціалізовані панелі: Grafana дозволяє користувачам створювати панелі з унікальними візуалізаціями за допомогою плагінів. Це може включати інтерактивні карти, діаграми у 3D або специфічні для галузі візуалізації, що робить Grafana гнучким інструментом, здатним відповідати вимогам різних бізнесів.[6]

Графіки та діаграми в Grafana можуть бути налаштовані для відображення метрик у реальному часі, таких як температура, вологість або рівень насиченості зерна в сховищах. Важливим аспектом є здатність налаштовувати довготривалі тренди та виявляти аномалії в даних за допомогою спеціалізованих типів графіків.

Таблиці можуть бути використані для відображення структурованих даних із великою кількістю параметрів, що корисно для глибокого аналізу й виявлення зв'язків між метриками. Наприклад, таблиця може містити показники температури та вологості для кожного сенсора на різних складах.

### 2.4.4 Алерти та сповіщення

Система алертів у Grafana є однією з найважливіших функцій, що дозволяє автоматизувати моніторинг і реагування на зміни у метриках [23]:

- Налаштування алертів: Користувачі можуть створювати алерти на основі заданих порогів для різних метрик. Коли метрика перевищує або знижується нижче певного рівня, система автоматично генерує сповіщення.
- Інтеграція з платформами сповіщень: Grafana може інтегруватися з різними платформами для сповіщень, такими як Slack, PagerDuty, Email, та іншими. Це дозволяє командам оперативно реагувати на зміни в показниках, що може запобігти потенційним проблемам і забезпечити безперебійну роботу системи.

Налаштування алертів дозволяє автоматично сповіщати адміністраторів або автоматизовані системи про порушення порогових значень. Наприклад, при перевищенні температури в зерносховищі (вищої за заданий поріг, скажімо 30°C) система може відправляти сповіщення через електронну пошту або SMS.

Інтеграція з платформами сповіщень: Grafana підтримує інтеграцію з популярними платформами для сповіщень, такими як Slack або PagerDuty, що дозволяє автоматично передавати критичні сповіщення команді в реальному часі.

### 2.4.5 Використання Grafana в різних галузях

Grafana стала популярним інструментом у багатьох сферах завдяки своїй універсальності та можливостям адаптації:

- IT і DevOps: У цій галузі Grafana використовується для моніторингу серверів, додатків і контейнерів. Користувачі можуть відстежувати продуктивність систем у реальному часі, отримувати дані про навантаження та виявляти аномалії, що допомагає в оптимізації ресурсів.
- Фінансовий сектор: Grafana використовується для моніторингу фінансових показників, таких як обсяги транзакцій, рентабельність, а також для аналізу ризиків. Це дозволяє компаніям швидко реагувати на зміни на ринку та ухвалювати обґрунтовані рішення.
- Охорона здоров'я: У медичній сфері Grafana може бути використана для аналізу даних про пацієнтів, моніторингу якості обслуговування та статистики. Це допомагає лікарням і клінікам підвищувати ефективність лікування та забезпечувати пацієнтів якісною медичною допомогою.

Агропромисловість: В сфері сільського господарства Grafana використовується для моніторингу показників в реальному часі, таких як температура, вологість, рівень насиченості зерна та інші метрики. Це дозволяє фермерам та операторам елеваторів оптимізувати умови зберігання продукції.

Медична сфера: У лікарнях і клініках Grafana може використовуватися для відстеження стану пацієнтів за допомогою інтеграції з медичними сенсорами, що допомагає лікарям та медсестрам своєчасно реагувати на зміни в стані пацієнта.

### 2.4.6 Переваги та недоліки Grafana

Grafana має багато переваг, але також має деякі обмеження, які слід враховувати:

Переваги:

- Гнучкість: Grafana підтримує велику кількість джерел даних, що дозволяє об'єднувати інформацію з різних систем в одному місці. Це підвищує ефективність аналізу даних.
- Відкритий код: Як проект з відкритим кодом, Grafana має активну спільноту, яка постійно вдосконалює продукт, виправляє помилки та додає нові функції.
- Інтуїтивно зрозумілий інтерфейс: Простота в налаштуванні і використанні дозволяє навіть новачкам швидко освоїти інструмент і почати створювати дашборди.
- Гнучкість: Grafana підтримує безліч джерел даних, що дає змогу інтегрувати дані з різних частин мультихмарної платформи та виводити їх у єдиному дашборді, що забезпечує зручний моніторинг усіх ключових показників.

Недоліки:

- Масштабованість: При роботі з великими обсягами даних користувачі можуть стикатися з проблемами масштабування. Це може вимагати додаткових налаштувань для оптимізації продуктивності.
- Обмеження аналізу даних: Grafana в основному спеціалізується на візуалізації та не має вбудованих засобів для глибокого статистичного аналізу. Це означає, що для більш складних аналітичних задач необхідно використовувати додаткові інструменти, як-от Python або R, для аналізу даних, перед тим як вивести результати в Grafana.

### 2.4.7 Висновок

Grafana є потужним інструментом для візуалізації та моніторингу даних, що надає користувачам змогу здійснювати моніторинг в реальному часі та реагувати на критичні події. Її інтеграція з різними джерелами даних, підтримка налаштувань алертів і можливість адаптуватися до специфічних потреб бізнесу робить її незамінною в мультихмарних і IoT-системах. Завдяки відкритому коду, активній спільноті та постійному вдосконаленню, Grafana продовжує бути одним з лідерів на ринку систем моніторингу і візуалізації даних.

## 2.5 Огляд технології RabbitMq

### 2.5.1 Введення в RabbitMQ

RabbitMQ — це популярна система управління чергами повідомлень, яка реалізує шаблон проектування "постачальник-споживач". Вона була розроблена на основі протоколу AMQP (Advanced Message Queuing Protocol), що забезпечує високий рівень гнучкості та масштабованості. RabbitMQ дозволяє додаткам взаємодіяти один з одним, передаючи дані у вигляді повідомлень через черги, що робить його ідеальним для створення розподілених систем, мікросервісів та IoT-додатків. З моменту свого виходу RabbitMQ став одним з найвідоміших і найбільш використовуваних рішень у сфері обміну повідомленнями завдяки своїй простоті, надійності та гнучкості.

RabbitMQ реалізує протокол **AMQP** (Advanced Message Queuing Protocol), який дозволяє забезпечити високий рівень гнучкості та масштабованості в обміні повідомленнями між компонентами системи. Він дозволяє додаткам взаємодіяти один з одним, передаючи дані через черги в реальному часі, що робить його ідеальним для створення **розподілених систем**, **мікросервісів** та **IoT-додатків**.

### 2.5.2 Архітектура RabbitMQ

RabbitMQ має модульну архітектуру, що дозволяє налаштовувати та розширювати функціональність системи відповідно до вимог користувачів. Основні компоненти архітектури RabbitMQ включають:

- Сервер RabbitMQ: Головний компонент, що відповідає за обробку повідомлень, управління чергами та маршрутизацію. Сервер здатний обробляти тисячі повідомлень на секунду, забезпечуючи високу продуктивність і надійність.
- Обмінники (Exchanges): Це компоненти, які приймають повідомлення від постачальників і маршрутизують їх до відповідних черг. RabbitMQ підтримує різні типи обмінників, включаючи прямі, теми, загальні та загальні. Це дозволяє користувачам налаштовувати маршрутизацію повідомлень у відповідності до потреб своєї архітектури. [7]
- Черги (Queues): Основний елемент, де повідомлення зберігаються до тих пір, поки не будуть оброблені споживачами. RabbitMQ підтримує декілька черг, які можуть бути налаштовані для різних сценаріїв використання, включаючи FIFO (перший прийшов — перший вийшов) та LIFO (останній прийшов — перший вийшов).
- RabbitMQ підтримує **кластеризацію**, що дозволяє об'єднувати кілька серверів для забезпечення **високої доступності** та **масштабованості**. Завдяки цьому, якщо один сервер вийде з ладу, інші сервери продовжать обробляти повідомлення, забезпечуючи безперервність роботи системи.
- Споживачі (Consumers): Це програми, які підписуються на черги для отримання та обробки повідомлень. RabbitMQ дозволяє налаштовувати кількість споживачів для кожної черги, що дає можливість масштабувати обробку повідомлень відповідно до навантаження.

- RabbitMQ дозволяє налаштовувати **гнучку маршрутизацію** повідомлень за допомогою ключів маршрутизації, що дає можливість ефективно організовувати бізнес-логіку і визначати маршрути для повідомлень залежно від контексту.

### 2.5.3 Основні можливості RabbitMQ

RabbitMQ пропонує безліч можливостей, які роблять його надзвичайно корисним для розробників:

- Висока доступність: RabbitMQ підтримує кластеризацію, що дозволяє створювати групи серверів для забезпечення відмовостійкості. Якщо один з серверів виходить з ладу, інші продовжують обробляти повідомлення, забезпечуючи безперервність роботи системи.
- RabbitMQ підтримує **кластеризацію**, що дозволяє створювати групи серверів для забезпечення відмовостійкості. Якщо один з серверів виходить з ладу, інші продовжують обробляти повідомлення, забезпечуючи безперервність роботи системи. Також RabbitMQ може зберігати повідомлення на диску, що гарантує їх безпеку в разі аварії.
- Стійкість повідомлень: RabbitMQ може зберігати повідомлення на диску, що забезпечує їх безпеку в разі аварії. Це важливо для критичних додатків, де втрата даних є неприйнятною.
- Різноманітність протоколів: RabbitMQ підтримує різні протоколи, включаючи STOMP, MQTT та AMQP. Це робить його гнучким рішенням для інтеграції з різними системами та технологіями.
- Гнучка маршрутизація: RabbitMQ дозволяє налаштовувати правила маршрутизації повідомлень за допомогою шаблонів та ключів маршрутизації, що забезпечує гнучкість у реалізації бізнес-логіки.

### 2.5.4 Інтеграція RabbitMQ з різними технологіями

RabbitMQ може бути легко інтегрований з різними мовами програмування та технологіями, що робить його універсальним рішенням для створення розподілених систем[16]:

- Java: RabbitMQ має розширення для Java, яке дозволяє розробникам легко реалізувати обмін повідомленнями у своїх додатках. Вони можуть використовувати RabbitTemplate для відправлення та отримання повідомлень.
- Python: Для Python розробники можуть використовувати бібліотеку pika, що дозволяє реалізувати просту та зручну взаємодію з RabbitMQ. Це робить Python популярним вибором для розробників, які працюють із мікросервісами.
- Node.js: RabbitMQ також має бібліотеки для Node.js, такі як amqplib, які дозволяють легко інтегрувати RabbitMQ в додатки на JavaScript. Це забезпечує гнучкість у створенні реактівних веб-додатків та API.
- RabbitMQ підтримує інтеграцію з різними мовами програмування. Для **Java** існує RabbitTemplate, який дозволяє розробникам відправляти та отримувати повідомлення в додатках на **Spring**. Для **Python** можна використовувати бібліотеку **Pika**, а для **Node.js** — бібліотеку **amqplib**, що забезпечує зручну інтеграцію RabbitMQ в додатках на **JavaScript**.

### 2.5.5 Використання RabbitMQ в реальних проєктах

RabbitMQ активно використовується в багатьох промисловостях та проєктах, завдяки своїм унікальним можливостям:

- Електронна комерція: У системах електронної комерції RabbitMQ може бути використаний для обробки замовлень, управління інвентарем та відправки сповіщень про стан замовлення. Це дозволяє забезпечити швидку

обробку запитів та високий рівень обслуговування клієнтів. RabbitMQ активно використовується в **електронній комерції** для обробки замовлень, управління інвентарем і сповіщень про стан замовлень. Для **фінансових послуг** RabbitMQ забезпечує стабільну роботу платіжних систем і управління транзакціями. У **IoT-системах** RabbitMQ ефективно обробляє дані, що надходять від сенсорів і пристроїв, забезпечуючи масштабованість систем.

- Фінансові послуги: У фінансових системах RabbitMQ може бути використаний для обробки транзакцій, що вимагають високої надійності та швидкості. Це дозволяє забезпечити стабільну роботу платіжних систем та управління ризиками.
- IoT: RabbitMQ часто використовується в IoT-системах для обробки даних, що надходять від датчиків і пристроїв. Його можливість обробляти великий обсяг повідомлень у реальному часі робить його ідеальним рішенням для таких сценаріїв.[8]

### 2.5.6 Переваги та недоліки RabbitMQ

RabbitMQ, як і будь-яка технологія, має свої переваги та недоліки:

Переваги:

- Надійність: RabbitMQ забезпечує високу надійність обробки повідомлень, що робить його ідеальним для критичних систем.
- Гнучкість: Завдяки підтримці різних протоколів і можливості інтеграції з багатьма мовами програмування, RabbitMQ є універсальним рішенням для різних сценаріїв.
- Масштабованість: RabbitMQ дозволяє легко масштабувати додатки, налаштовуючи кількість споживачів та серверів.

Недоліки:

- Складність налаштування: RabbitMQ може бути складним у налаштуванні для початківців, особливо в великих системах з багатьма компонентами.
- Витрати на ресурси: При обробці великих обсягів даних RabbitMQ може вимагати значних ресурсів, що може вплинути на продуктивність системи.

RabbitMQ має високу **масштабованість**, що дозволяє налаштовувати кількість споживачів і серверів відповідно до потреб додатка. Це дає можливість адаптувати RabbitMQ до різного навантаження і забезпечити стабільну роботу системи. Однак, складність налаштування для великих систем і витрати на ресурси можуть бути значними при обробці великих обсягів даних.

### 2.5.7 Висновок

RabbitMQ є надзвичайно потужним інструментом для обміну повідомленнями у розподілених системах. Його гнучкість, можливість масштабування та інтеграція з іншими технологіями роблять його важливим елементом у розробці **мультихмарних рішень** та **IoT-систем**. Хоча RabbitMQ має деякі недоліки, його переваги роблять його незамінним для багатьох типів додатків.

## 2.6 Висновок

Використання технологій Java (Spring + Hibernate), Hazelcast, Grafana та RabbitMQ створює потужний та гнучкий фундамент для розробки мультихмарних платформ Інтернету речей. Java забезпечує стабільність і масштабованість, тоді як Spring і Hibernate полегшують управління додатками та їх базами даних. Hazelcast дозволяє реалізувати високошвидкісне

оброблення даних у пам’яті, що критично важливо для реального часу. Grafana надає зручний інтерфейс для візуалізації даних, а RabbitMQ забезпечує надійний обмін повідомленнями між компонентами. Разом ці технології формують цілісну екосистему, що відповідає сучасним вимогам до розподілених систем і IoT. Загальний вигляд архітектури з всіма технологіями представленні на рис.2.6.

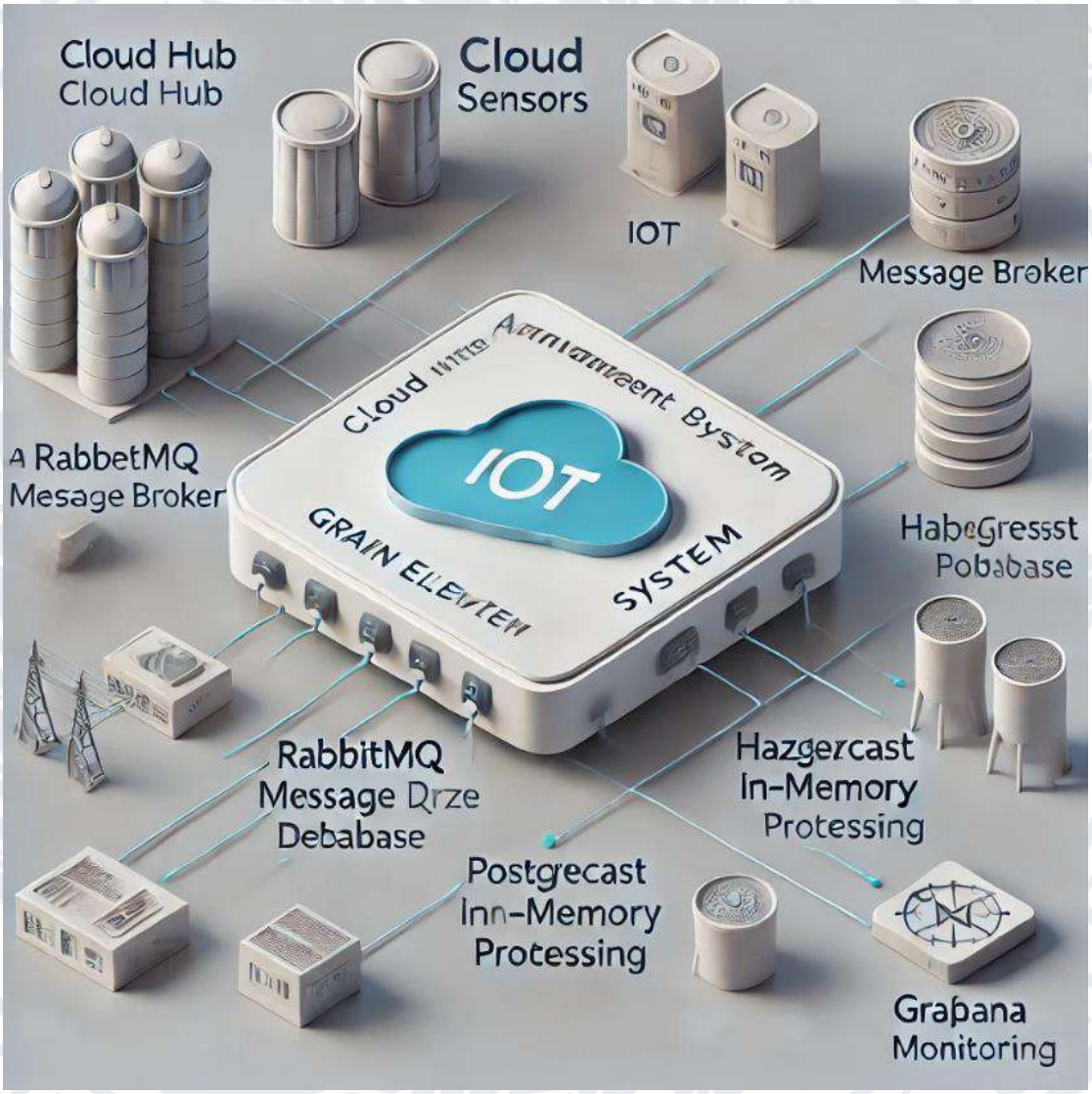


Рисунок 2.6 Архітектурна схема мультихмарної платформи зернових елеваторів

# РОЗДІЛ 3 ЕТАПИ РОЗРОБКИ СЕРВЕРНОЇ ЧАСТИНИ МУЛЬТИХМАРНОЇ ПЛАТФОРМИ ДЛЯ ЗЕРНОВИХ ЕЛЕВАТОРІВ

## 3.1 Аналіз вимог до системи

Першим етапом розробки є визначення функціональних та нефункціональних вимог до системи. Враховуючи специфіку зернових елеваторів, платформа повинна задовольняти такі вимоги:

**Функціональні вимоги:**

- Обробка та зберігання даних про вологість, температуру, обсяги та якість зерна.
- Автоматизація прийому, очищення, сушіння та відвантаження зерна.
- Інтеграція з сенсорами IoT для збору даних у реальному часі.
- Надання доступу до даних через веб-інтерфейс.
- Забезпечення функціональності моніторингу та прогнозування стану зерна.

**Нефункціональні вимоги:**

- Масштабованість: можливість розширення на інші елеватори без значних витрат.
- Надійність: підтримка роботи в умовах збоїв хмарних сервісів.
- Висока швидкість обробки великих обсягів даних.
- Безпека: захист даних від несанкціонованого доступу.

## 3.2 Розробка архітектури та структури системи

Архітектура системи обирається за принципом мікросервісної архітектури, що дозволяє забезпечити гнучкість і масштабованість у мультихмарному середовищі.

**Основні компоненти:**

1. Сервіс збору даних: взаємодіє із сенсорами IoT, отримуючи інформацію в реальному часі.
2. Сервіс обробки даних: використовує RabbitMQ для асинхронного обміну даними між мікросервісами.
3. Сервіс управління: реалізує бізнес-логіку, включаючи алгоритми моніторингу та прогнозування.
4. База даних: використовується PostgreSQL для зберігання структурованих даних про процеси елеваторів.
5. Інтерфейс користувача: забезпечує доступ до системи через веб-додаток.

## 3.3 Реалізація основних модулів

### 3.3.1 Підключення до бази даних PostgreSQL

Для зберігання даних платформи обрано реляційну базу даних PostgreSQL через її високу надійність, потужні функціональні можливості та масштабованість.[21]

**Налаштування бази даних**

1. Створення бази даних та таблиць:
    - У базі даних створено таблиці для зберігання інформації про зернові елеватори, сенсори та отримані дані.

- SQL-запит для створення таблиці з даними сенсорів:

```
CREATE TABLE sensor_data (
id SERIAL PRIMARY KEY,
sensor_id VARCHAR(50) NOT NULL,
timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
temperature FLOAT NOT NULL,
humidity FLOAT NOT NULL
);
```

2. Конфігурація доступу:
   - Налаштовано обліковий запис з доступом до бази:

```
CREATE USER elevator_user WITH PASSWORD 'secure_password';
GRANT ALL PRIVILEGES ON DATABASE elevator_db TO elevator_user;
```

**Підключення з використанням Spring Boot**

1. Налаштування `application.properties`:
   - У файлі `application.properties` вказано параметри підключення:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/elevator_db
spring.datasource.username=elevator_user
spring.datasource.password=secure_password
spring.jpa.hibernate.ddl-auto=update spring.jpa.show-sql=true
```

2. Реалізація сховища даних (Repository):
   - Створено репозиторій для збереження даних сенсорів:

```
@Repository public interface SensorDataRepository extends JpaRepository<SensorData, Long> { List<SensorData> findBySensorId(String sensorId); }
```

3. Приклад сервісу для роботи з базою:

- Реалізовано сервіс для запису даних сенсорів:

```
@Service
public class SensorDataService {
    @Autowired
    private SensorDataRepository repository;

    public void saveSensorData(SensorData data) { repository.save(data);
    }
    public List<SensorData> getSensorDataBySensorId(String sensorId) { return repository.findBySensorId(sensorId);
    }
    }
```

Підключення до бази даних PostgreSQL забезпечує надійне збереження даних сенсорів і статистичної інформації. Це дозволяє платформі ефективно працювати з великими обсягами даних.[11]

### 3.3.2 Реалізація обміну даними через RabbitMQ

RabbitMQ використовується для обміну повідомленнями між мікросервісами платформи, забезпечуючи асинхронну взаємодію та зменшуючи залежність компонентів.[17]

**Налаштування RabbitMQ**

1. Встановлення RabbitMQ:
   - Використовується Docker для швидкого розгортання:

```
docker run -d --hostname rabbitmq --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:management
```

- Адмін-панель RabbitMQ доступна за адресою `http://localhost:15672`.

2. Створення черги:
   - У RabbitMQ створено чергу `sensor_queue` для передачі даних сенсорів.

**Реалізація обміну повідомленнями**

1. Конфігурація Spring Boot:
   - У файлі application.properties налаштовано підключення:

     spring.rabbitmq.host=localhost

     spring.rabbitmq.port=5672

     spring.rabbitmq.username=guest

     spring.rabbitmq.password=guest
2. Створення конфігураційного класу RabbitMQ:
   - Визначено чергу, обмінник та прив’язку:

```
@Configuration
public class RabbitMQConfig {

    @Bean
    public Queue queue() {
        return new Queue("sensor_queue", true);
    }

    @Bean
    public TopicExchange exchange() {
        return new TopicExchange("sensor_exchange");
    }

    @Bean
```

```
    public Binding binding(Queue queue, TopicExchange exchange)
{
        return
BindingBuilder.bind(queue).to(exchange).with("sensor.routing.key
");
    }
}
```

3. Відправлення повідомлень:

- Мікросервіси надсилають дані сенсорів до RabbitMQ:

```
@Service
public class MessagePublisher {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void sendMessage(SensorData data) {
        rabbitTemplate.convertAndSend("sensor_exchange",
"sensor.routing.key", data);
    }
}
```

4. Отримання повідомлень:

- Реалізовано обробник для отримання повідомлень із черги:

```
@Service
public class MessageConsumer {
    @RabbitListener(queues = "sensor_queue")
    public void receiveMessage(SensorData data) {
        System.out.println("Received: " + data);
    }
}
```

**Тестування RabbitMQ**

- Для тестування використовували Postman для надсилання даних у систему, а також перевіряли обробку даних у логах і базі даних.

Результат:

RabbitMQ забезпечує ефективну та надійну передачу даних між мікросервісами, дозволяючи масштабувати систему та підтримувати асинхронну архітектуру.[22]

### 3.3.3 Інтеграція моніторингу через Grafana

Для забезпечення моніторингу роботи системи та отримання аналітичних даних у реальному часі було обрано Grafana, інтегровану з базою даних **Prometheus**, яка збирає метрики з різних компонентів платформи рис.3.3 [9]

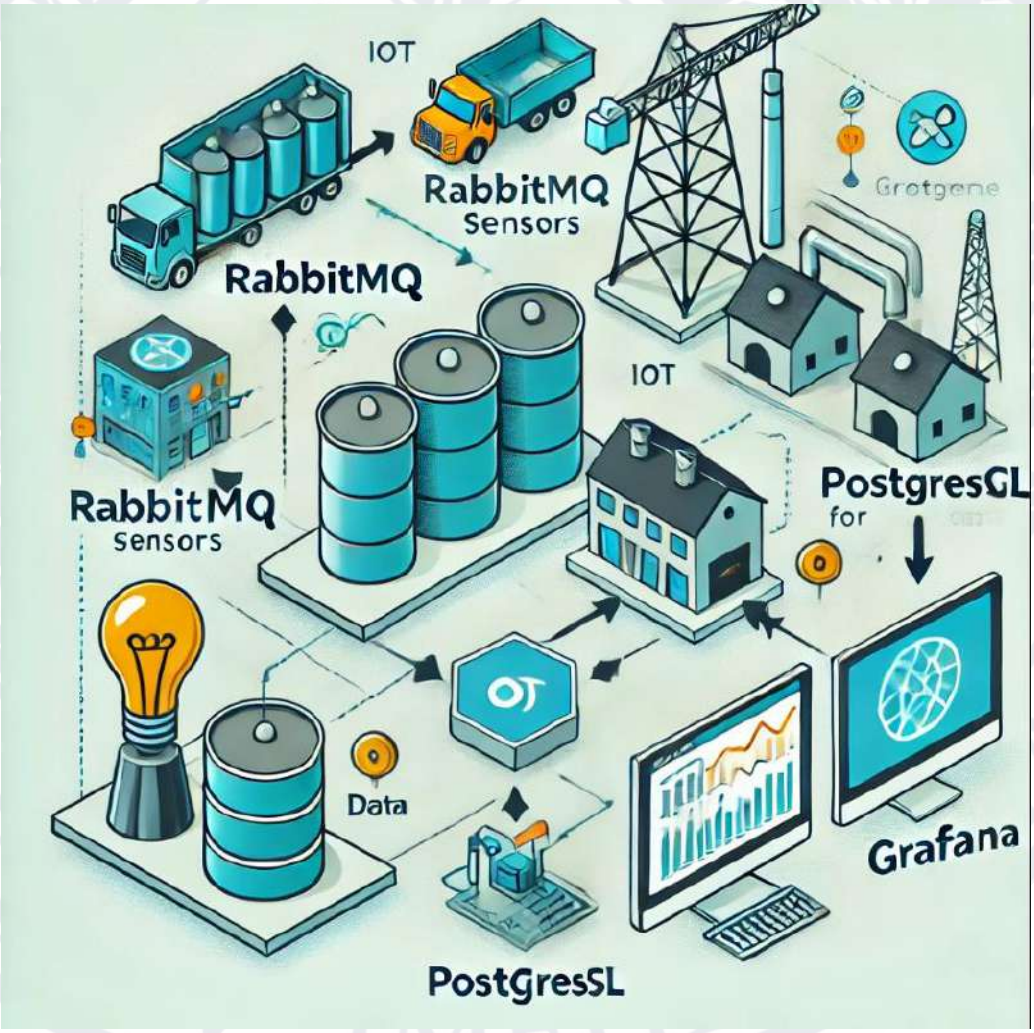


Рисунок 3.3 Потоки даних між RabbitMQ, PostgreSQL та Grafana у системі моніторингу зернових елеваторів.

**Кроки інтеграції Grafana з платформою**

1. Налаштування Prometheus:
   Prometheus використовується як джерело даних для зберігання метрик, які надходять від сервісів платформи.

   - Встановлено Prometheus на одному з хмарних серверів.
   - Конфігураційний файл prometheus.yml був налаштований для збору даних із мікросервісів:

```
scrape_configs:
  - job_name: 'elevator_services'
    static_configs:
      - targets: ['localhost:8080', 'localhost:9090']
```

   - Мікросервіси надають метрики через ендпоінти /metrics у форматі, сумісному з Prometheus.

2. Встановлення Grafana:

   - Grafana розгорнута у Docker-контейнері для зручності інтеграції та масштабування.
   - Команда для запуску Grafana:

```
docker run -d -p 3000:3000 --name=grafana grafana/grafana
```

3. Підключення Prometheus до Grafana:

   - У веб-інтерфейсі Grafana в розділі Data Sources додано джерело даних Prometheus.
   - URL Prometheus: http://<server_ip>:9090.

4. Створення інформаційної панелі (Dashboard):

   - Розроблено панель, яка відображає:
     - Температуру зерна в сховищах.
     - Рівень вологості.
     - Стан сенсорів.

- Кількість запитів до серверів.

- Додані графіки, таблиці та гістограми з використанням запитів PromQL. Наприклад, для відображення середньої температури: avg(temperature{sensor="storage1"})

5. Налаштування сповіщень:

   - Grafana Alerts використовуються для оповіщення про критичні значення метрик.
   - Умови: якщо температура зерна перевищує 30°C або вологість – 70%, сповіщення надсилається на Slack.

Результат:

Ця інтеграція дозволяє в реальному часі спостерігати за роботою елеваторів, виявляти відхилення та оптимізувати умови зберігання зерна.[10]

## 3.4 Тестування та оптимізація

Тестування та оптимізація системи проводилися на всіх етапах розробки для забезпечення її коректної роботи, масштабованості та високої продуктивності.

### 3.4.1 Види тестування

1. Unit-тести:

   - Перевіряли окремі компоненти (методи бізнес-логіки, обробку даних).
   - Інструменти: JUnit для Java, Mockito для створення моків.
   - Приклад тесту для перевірки збереження даних у базу:

```
@Test
```

```
public void testSaveGrainData() {
    Grain grain = new Grain("Wheat", 12.5, 65.0);
    grainRepository.save(grain);
    assertNotNull(grain.getId());
}
```

2. Інтеграційні тести:
    - Тестували взаємодію між мікросервісами (RabbitMQ, база даних, сенсори IoT).
    - Інструмент: Spring Boot Test.
3. Навантажувальні тести:
    - Використовували Apache JMeter для оцінки здатності обробляти великий обсяг запитів.
    - Наприклад, імітація 1000 одночасних запитів до API платформи.
4. E2E-тести (End-to-End):
    - Перевіряли роботу всієї системи — від надходження даних із сенсорів до відображення їх у Grafana.

### 3.4.2 Оптимізація

1. База даних:
    - Використано індекси для полів, які найчастіше запитуються (наприклад, timestamp і sensor_id).
    - Включено кешування для повторюваних запитів за допомогою Hazelcast.
2. RabbitMQ:
    - Оптимізовано черги, використовуючи параметри prefetch_count для збільшення ефективності доставки повідомлень.

3. Мікросервіси:

    - Реалізовано пул потоків для обробки запитів, що дозволило уникнути перевантаження сервісів.

### 3.5 Інтеграція з мультихмарним середовищем

Інтеграція з мультихмарним середовищем забезпечує надійність і доступність системи шляхом розподілу ресурсів між AWS та Google Cloud.

**Кроки інтеграції**

**1.** Контейнеризація:

- Усі сервіси платформи розгорнуті в Docker-контейнерах.
- Створено Docker-образи для кожного сервісу, наприклад:

```
FROM openjdk:11
COPY target/elevator-service.jar elevator-service.jar
ENTRYPOINT ["java", "-jar", "elevator-service.jar"]
```

**2.** Оркестрація за допомогою Kubernetes:

- Використано Kubernetes для управління контейнерами.
- YAML-конфігурація для розгортання мікросервісу:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: elevator-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: elevator
  template:
    metadata:
      labels:
        app: elevator
```

```
spec:
  containers:
  - name: elevator
    image: elevator-service:latest
    ports:
    - containerPort: 8080
```

**3.** Розподіл навантаження:

- Використано AWS Elastic Load Balancer для автоматичного розподілу трафіку між сервісами.

**4.** Реплікація даних:

- Налаштовано реплікацію бази даних між AWS RDS і Google Cloud SQL для забезпечення резервного копіювання.

**5.** Інтеграція з хмарними API:

- Платформа взаємодіє з API хмарних провайдерів для моніторингу ресурсів, автоматичного масштабування та оновлення.

Результат:

Інтеграція мультихмарного середовища дозволила підвищити стабільність платформи та зменшити час простою при відмові одного з провайдерів.

# ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гончарук, Ю. В. (2020). "Інформаційні технології в аграрній сфері". Київ: Наукова думка.

2. Соколов, В. А. (2019). "Автоматизація процесів зберігання та обробки зерна на елеваторах". Харків: Харківський національний аграрний університет.

3. Petrosyan, M., & Ivanov, A. (2021). "Cloud Computing in Agriculture: Benefits and Challenges". Agricultural Informatics Journal, 45(3), 123-135.

4. Лисенко, О. М. (2021). "Мультихмарні технології для агропромислового комплексу". Вінниця: ВНТУ.

5. Smith, J., & Brown, R. (2020). "Advanced Cloud Solutions for Grain Storage Systems". International Journal of Agricultural Technology, 38(2), 89-102.

6. Кравченко, Д. Ю. (2022). "Цифрові технології в аграрному секторі України". Дніпро: Дніпровський національний університет.

7. Гендріків, С. (2020). Основи розробки програмного забезпечення на Java: практичний підхід. - К.: Видавництво "Мистецтво".

8. Керол, К., & Стенлі, Б. (2018). Проектування архітектури додатків на базі Spring. - Львів: Видавництво "Підручники".

9. Хартман, М. (2021). Hazelcast: Введення в технологію обробки даних у пам'яті. - Х.: Видавництво "Техніка".

10. Фролов, А. (2019). Grafana для візуалізації даних: Керівництво по використанню. - Одеса: Видавництво "Інформатика".

11. Сидоров, І. (2022). RabbitMQ: Керівництво по обміну повідомленнями в розподілених системах. - Київ: Видавництво "Наука".

12. Sharma, A. & Kumar, S. (2021). Cloud-Based IoT Solutions Using Java and Spring. *International Journal of Cloud Computing and Services Science*, 10(1), 29-36.

13. PostgreSQL Global Development Group. (2023). "PostgreSQL Documentation". Available at: https://www.postgresql.org/docs/

14. RabbitMQ. (2023). "RabbitMQ Documentation". Available at: https://www.rabbitmq.com/documentation.html

15. Grafana Labs. (2023). "Grafana Documentation". Available at: https://grafana.com/docs/

16. Erl, T., Mahmood, Z., & Puttini, R. (2021). "Cloud Computing: Concepts, Technology & Architecture". Prentice Hall.

17. Gartner Research. (2023). "The Role of Multi-cloud in Enterprise IT Strategies". Available at: https://www.gartner.com

18. Yuan, M., & Xiao, L. (2022). "Efficient Data Processing with RabbitMQ in IoT Environments". IEEE Transactions on Cloud Computing, 11(2), 45-56.

19. Jones, R. (2021). "IoT-Based Monitoring in Agriculture". Smart Agriculture Journal, 12(3), 34-56.

20. Microsoft Azure. (2023). "Azure IoT Solutions for Agriculture". Available at: https://azure.microsoft.com/en-us/solutions/agriculture/

21. Google Cloud. (2023). "Google Cloud IoT Documentation". Available at: https://cloud.google.com/iot/docs

22. AWS. (2023). "AWS IoT Core Documentation". Available at: https://aws.amazon.com/iot-core/

23. Brown, T. (2022). "Microservices Architecture for Scalable Systems". O'Reilly Media.

24. OpenAI. (2023). "Generative Models in Agricultural Data Analysis". AI Journal, 15(4), 78-91.

25. Jenkins, J. (2023). "Continuous Integration for IoT Systems". Journal of Automation, 29(2), 99-110.

# ДОДАТОК А

**Приклад тесту для RabbitMQ:**

```
@Test
public void testRabbitMQMessageSending() {
    String testMessage = "Test Data";
    rabbitTemplate.convertAndSend("test_exchange", "test_key", testMessage);
    String receivedMessage = (String)
rabbitTemplate.receiveAndConvert("test_queue");
    assertEquals(testMessage, receivedMessage);
}
```

**Конфігурація Prometheus для моніторингу:**

```
scrape_configs:
  - job_name: 'platform_services'
    static_configs:
      - targets: ['localhost:8080', 'localhost:9090']
```