ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО «ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
Міжрегіональна академія управління персоналом
Інститут комп'ютерно-інформаційних технологій
Кафедра комп'ютерних інформаційних систем і технологій

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

Тема «Проектування та розробка програмного забезпечення для управління великими даними у реальному часі»

Виконала студентка групи ІК-9-22-Б1ІПЗ(4.0д) Суховій Д. О.
Керівник: д.т.н., професор Дуднік А.С.
Рецензент: к.т.н., доцент

Допущено до захисту

Завідувач кафедри _____ д.е.н., професор Кавун С.В.

(підпис)

Київ, 2026

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 81 с., 14 рис.,15 табл., 48 джерел.

ВЕЛИКІ ДАНІ, ОБРОБКА ДАНИХ У РЕАЛЬНОМУ ЧАСІ, APACHE KAFKA, APACHE FLINK, KAPPA-АРХІТЕКТУРА, ПОТОКОВА ОБРОБКА, APACHE CASSANDRA, РОЗПОДІЛЕНІ СИСТЕМИ, ПОЛІГЛОТ-ПЕРСИСТЕНТНІСТЬ, UML-МОДЕЛЮВАННЯ.

Об'єктом дослідження є процес управління великими даними в умовах їх надходження та обробки у реальному часі.

Метою роботи є проектування програмного забезпечення для ефективного управління великими даними у реальному часі з обґрунтуванням архітектурних рішень на основі аналізу та порівняльного дослідження існуючих підходів і технологій.

Основні методи дослідження: системний аналіз, порівняльний аналіз існуючих платформ та архітектурних підходів, об'єктно-орієнтоване проектування, UML-моделювання.

У роботі досліджено теоретичні основи концепції великих даних, проведено порівняльний аналіз архітектурних підходів Lambda, Kappa та Delta, а також платформ потокової обробки Apache Flink, Spark Streaming та Kafka Streams. Спроектовано п'ятирівневу архітектуру системи на базі Kappa-підходу з використанням стеку Apache Kafka, Apache Flink, Apache Cassandra, Redis та Amazon S3. Виконано UML-моделювання системи у чотирьох нотаціях. Проведено порівняльне дослідження ефективності архітектурних рішень на основі опублікованих наукових досліджень та сформовано практичні рекомендації щодо застосування розглянутих підходів. Виконано економічне обґрунтування доцільності розробки з розрахунком NPV, IRR та терміну окупності.

WEB APPLICATION FOR VIRTUAL GALLERY, WEB-CONSTRUCTION, UNITY, C#, HOSTING.

BIG DATA, REAL-TIME DATA PROCESSING, APACHE KAFKA, APACHE FLINK, KAPPA ARCHITECTURE, STREAM PROCESSING, APACHE CASSANDRA, DISTRIBUTED SYSTEMS, POLYGLOT PERSISTENCE, UML MODELING.

The object of study is the process of big data management under conditions of real-time data ingestion and processing.

The purpose of the thesis is the design of software for effective real-time big data management with architectural decision justification based on analysis and comparative research of existing approaches and technologies.

The main research methods include: system analysis, comparative analysis of existing platforms and architectural approaches, object-oriented design, and UML modeling.

The thesis investigates the theoretical foundations of the big data concept, conducts a comparative analysis of Lambda, Kappa and Delta architectural approaches, as well as Apache Flink, Spark Streaming and Kafka Streams stream processing platforms. A five-layer system architecture based on the Kappa approach has been designed using the Apache Kafka, Apache Flink, Apache Cassandra, Redis and Amazon S3 stack. UML modeling of the system has been performed in four notations. A comparative study of the effectiveness of architectural solutions based on published research has been conducted and practical recommendations for the application of the considered approaches have been formulated. An economic feasibility study has been performed including NPV, IRR and payback period calculations.

ЗМІСТ

ВСТУП

Стрімкий розвиток цифрових технологій, поширення Інтернету речей (IoT), соціальних мереж та хмарних обчислень призвели до безпрецедентного зростання обсягів даних, що генеруються щодня. За оцінками аналітиків IDC, до 2025 року загальний обсяг цифрових даних у світі перевищить 175 зетабайт, причому значна їх частина вимагатиме обробки в режимі реального часу [1]. Ця цифра вражає - і змушує замислитись над тим, чи готова сучасна програмна інженерія до таких масштабів. Традиційні реляційні бази даних та класичні методи пакетної обробки вже не здатні повною мірою задовольнити потреби сучасних систем у швидкості, масштабованості та надійності.

Управління великими даними у реальному часі охоплює широкий спектр задач - від збору та передачі потоків подій до їх миттєвої аналітичної обробки та збереження. Особливої актуальності ця проблематика набуває у таких галузях, як фінансові технології, телекомунікації, промислова автоматизація та кібербезпека, де затримка у прийнятті рішень навіть у кілька секунд може призвести до відчутних втрат [2]. Попри широку доступність окремих інструментів - Apache Kafka, Apache Flink, Spark Streaming - питання проектування цілісної та ефективно масштабованої архітектури програмного забезпечення для управління великими даними у реальному часі залишається відкритим і потребує комплексного дослідження [3].

Недостатня опрацьованість питань інтеграції компонентів такої системи, вибору оптимальних архітектурних патернів та порівняльного аналізу існуючих рішень визначає актуальність обраної теми кваліфікаційної роботи.

Метою роботи є проектування програмного забезпечення для ефективного управління великими даними у реальному часі з обґрунтуванням

архітектурних рішень на основі аналізу та порівняльного дослідження існуючих підходів і технологій.

Для досягнення поставленої мети необхідно виконати наступні завдання:

- дослідити теоретичні основи концепції великих даних та методів їх обробки у реальному часі;
- проаналізувати існуючі архітектурні підходи та платформи для управління потоками даних;
- визначити функціональні та нефункціональні вимоги до програмного забезпечення, що розробляється;
- провести порівняльний аналіз технологічного стеку та обґрунтувати вибір архітектурних рішень;
- розробити структуру системи та виконати її моделювання засобами UML;
- спроектувати та описати ключові модулі програмного забезпечення;
- на основі опублікованих досліджень провести порівняльний аналіз продуктивності, масштабованості та відмовостійкості розглянутих підходів;
- виконати економічне обґрунтування доцільності розробки.

Об'єктом дослідження є процес управління великими даними в умовах їх надходження та обробки у реальному часі.

Предметом дослідження є методи, архітектурні патерни та програмні засоби проектування систем управління великими даними у реальному часі.

Методи дослідження. У роботі використовуються методи системного аналізу для дослідження предметної області; порівняльного аналізу для оцінки існуючих технологій та рішень; об'єктно-орієнтованого проектування

та UML-моделювання для розробки архітектури системи; а також методи узагальнення та синтезу при формуванні висновків [4].

Практичне значення отриманих результатів полягає у розробці обґрунтованої архітектури програмного забезпечення, яка може бути використана як базова модель при побудові корпоративних систем обробки потокових даних, а також у навчальному процесі при підготовці фахівців у галузі інформаційних технологій.

Структура роботи. Кваліфікаційна робота складається зі вступу, п'яти розділів, висновків, переліку джерел посилань та додатків. Загальний обсяг роботи становить сторінок, рисунків, таблиць, додатки. Перелік джерел посилань містить 48 найменувань.

1 ТЕОРЕТИЧНІ ОСНОВИ УПРАВЛІННЯ ВЕЛИКИМИ ДАНИМИ У РЕАЛЬНОМУ ЧАСІ

1.1 Концепція великих даних: визначення, характеристики та модель 5V

Поняття «великі дані» (англ. Big Data) сьогодні зустрічається настільки часто, що ризикує перетворитися на кліше. Проте за цим терміном стоїть цілком конкретна і технічно обґрунтована концепція, яка докорінно змінила підходи до зберігання, обробки та аналізу інформації. Щоб зрозуміти, чому саме управління великими даними стало окремою інженерною дисципліною, варто почати з витоків.

Першу формалізовану спробу описати феномен великих даних зробив аналітик дослідницької групи META Group (нині Gartner) Дуг Ланей у своїй дослідницькій доповіді «3D Data Management: Controlling Data Volume, Velocity and Variety», опублікованій у 2001 році [5]. Ланей запропонував розглядати великі дані через призму трьох вимірів: обсяг (Volume), швидкість (Velocity) та різноманітність (Variety) — так звана модель 3V. Ця концепція виявилась надзвичайно життєздатною і стала відправною точкою для подальших досліджень у галузі.

Проте з часом стало очевидним, що три виміри не повністю відображають складність проблематики. У міру накопичення практичного досвіду роботи з масивами даних дослідники почали розширювати початкову модель. На першому Міжнародному симпозіумі з великих даних та аналізу даних (BDDAC) такі дослідники, як Юрій Демченко, обґрунтували необхідність включення до моделі характеристик достовірності (Veracity) та цінності (Value) [6]. Так модель 3V еволюціонувала до сучасної моделі 5V, яка на сьогодні є загальновизнаним стандартом опису великих даних у науковій спільноті [7].

Перш ніж детально розглядати кожну з характеристик, важливо визначитись із самим поняттям. Існує чимало визначень терміну «великі дані», і це не дивно — концепція охоплює одразу технологічний, організаційний та аналітичний аспекти. Найбільш широке визнання отримало визначення Gartner: великі дані — це масиви інформації з великим обсягом, високою швидкістю надходження та різноманітністю форматів, що потребують економічно ефективних та інноваційних методів обробки для підтримки прийняття рішень [5]. Дещо більш технічне трактування пропонує Клеппман: великі дані — це набори даних, розмір, складність і швидкість зростання яких роблять їх непридатними для обробки традиційними системами управління базами даних [2].

Обидва визначення акцентують на ключовій ідеї: проблема великих даних — це передусім проблема невідповідності між характеристиками даних та можливостями традиційних інструментів їх обробки. Саме ця невідповідність і породила необхідність у нових архітектурних підходах та спеціалізованому програмному забезпеченні.

Розглянемо детально кожну складову моделі 5V та її практичне значення для систем управління даними.

Обсяг (Volume) є найбільш очевидною характеристикою і, власне, тією, що дала назву всій концепції. Мова йде про дані, обсяг яких вимірюється терабайтами, петабайтами та екзабайтами. За даними IDC, у 2020 році загальний обсяг сформованих, захоплених та скопійованих даних у світі досяг 40 зетабайт, а до 2025 року очікується зростання до 175 зетабайт [8]. Практичним наслідком цієї характеристики є неможливість зберігання та обробки таких масивів на одному сервері — виникає потреба у розподілених системах зберігання та горизонтальному масштабуванні.

Швидкість (Velocity) відображає інтенсивність генерації та необхідної обробки даних. У сучасних системах дані можуть надходити з частотою

мільйонів подій за секунду — транзакції на фінансових ринках, показники датчиків у промислових системах, потоки даних із соціальних мереж [2]. Ця характеристика є ключовою для теми даної роботи, оскільки саме вона обумовлює необхідність обробки даних у реальному часі (real-time processing) на противагу традиційній пакетній обробці.

Різноманітність (Variety) вказує на гетерогенність форматів і джерел даних. Сучасні системи вимушені одночасно працювати зі структурованими даними (реляційні таблиці), напівструктурованими (JSON, XML, CSV) та неструктурованими (текст, зображення, відео, аудіо) [7]. Саме ця характеристика зумовлює обмеженість реляційних СУБД і необхідність застосування поліглот-персистентності (polyglot persistence) — використання різних типів сховищ даних залежно від характеру інформації.

Достовірність (Veracity) характеризує якість та надійність даних. На практиці великі масиви даних майже завжди містять шуми, дублікати, суперечливі записи та пропуски [6]. За різними оцінками, від 20% до 40% корпоративних даних мають проблеми з якістю, що безпосередньо впливає на точність аналітичних висновків [9]. Ця характеристика підкреслює важливість етапу попередньої обробки та валідації даних у будь-якій системі управління великими даними.

Цінність (Value) є, мабуть, найважливішою з практичної точки зору характеристикою, хоча й найменш технічною. Дані самі по собі не мають цінності — вона виникає лише тоді, коли з них вилучається корисна інформація для прийняття рішень [6]. Ця характеристика нагадує про те, що збирання та зберігання даних є лише засобом, а не метою — кінцевою метою завжди залишається отримання практичної користі.

Взаємозв'язок між п'ятьма характеристиками моделі 5V показано на рисунку 1.1.

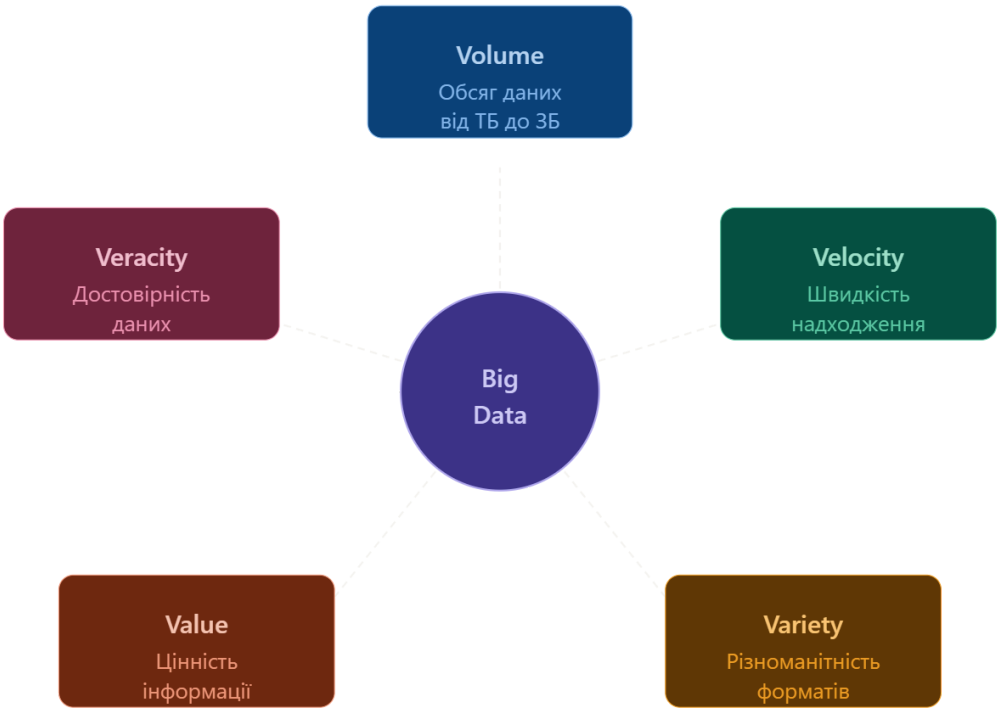

Рисунок 1.1 — Модель 5V характеристик великих даних

Варто зазначити, що модель 5V не є статичною — дискусії щодо її розширення тривають. Деякі дослідники пропонують додати шосту характеристику — мінливість (*Variability*), що описує непостійність семантики даних у часі, або візуалізацію (*Visualization*) як окремий вимір складності [1]. Проте модель 5V залишається найбільш збалансованою та загальновживаною у сучасній науковій літературі.

Ще одним важливим аспектом є класифікація великих даних за типом їх надходження. Прийнято розрізняти:

- **статичні великі дані** (*data at rest*) — масиви, накопичені за певний період та доступні для пакетної обробки; характерні для задач архівного аналізу та побудови моделей машинного навчання;

- **потокові великі дані** (*data in motion*) — безперервні потоки подій, що надходять у реальному або близькому до реального часі; характерні для IoT-систем, фінансових платформ та систем моніторингу [2].

Саме потокові великі дані є центральним об'єктом розгляду в даній роботі, оскільки управління ними пред'являє принципово вищі вимоги до архітектури системи та є технологічно значно складнішим завданням, ніж обробка статичних масивів.

Таким чином, концепція великих даних являє собою не просто кількісну, а якісну зміну в характеристиках даних, з якими стикаються сучасні інформаційні системи. Модель 5V забезпечує системне розуміння цих характеристик і визначає вектори технологічних викликів, яким присвячено подальші розділи даної роботи.

1.2 Потокова та пакетна обробка даних: порівняльний аналіз

З точки зору часових характеристик обробки даних існують два принципово різні підходи: пакетна обробка (batch processing) та потокова обробка (stream processing). Вибір між ними визначає всю подальшу архітектуру системи, тому розуміння їх відмінностей є необхідною передумовою для проектування програмного забезпечення управління великими даними [2].

Пакетна обробка передбачає накопичення даних протягом певного інтервалу часу з подальшим їх одноразовим опрацюванням. Дані збираються, зберігаються у тимчасовому сховищі, а потім обробляються як єдиний масив за заздалегідь визначеним розкладом — щогодини, щоденно або щотижнево [3]. Цей підхід забезпечує високу пропускну здатність і є економічно ефективним для задач, що не потребують миттєвої реакції: формування фінансової звітності, навчання моделей машинного навчання, ETL-процеси у

сховищах даних. Основним недоліком є висока затримка — від кількох хвилин до кількох діб [10].

Потокова обробка функціонує принципово інакше: дані обробляються безперервно в міру їх надходження, без очікування накопичення певного обсягу. Затримка при цьому вимірюється мілісекундами, що робить цей підхід незамінним для систем виявлення шахрайства, IoT-моніторингу, динамічного ціноутворення та кібербезпеки [2]. Водночас потокова обробка вимагає складнішої архітектури, зокрема механізмів гарантованої доставки подій та управління станом (stateful processing) [11].

Ключові відмінності між двома підходами наведено у таблиці 1.1.

Таблиця 1.1 — Порівняльний аналіз пакетної та потокової обробки даних

Критерій	**Пакетна обробка**	**Потокова обробка**
Затримка	Хвилини — доби	Мілісекунди — секунди
Пропускна здатність	Висока	Середня
Складність реалізації	Низька	Висока
Вартість інфраструктури	Нижча	Вища
Тип даних	Обмежені набори	Необмежені потоки
Типові інструменти	Hadoop, Apache Hive, Spark Batch	Apache Kafka, Apache Flink, Spark Streaming
Типові застосування	Звітність, ETL, аналітика	Моніторинг, fraud detection, IoT

Варто зазначити, що протиставлення цих підходів як взаємовиключних є надмірним спрощенням. На практиці дедалі популярнішими стають гібридні архітектури, що поєднують переваги обох парадигм. Наприклад, Lambda-архітектура, розглянута у підрозділі 1.3, використовує пакетний шар

для глибокого ретроспективного аналізу та потоковий — для забезпечення актуальності даних у реальному часі [2]. За даними опитування 2023 року, 65% інженерів з обробки даних використовують пакетні методи для складних трансформацій, тоді як близько 60% організацій паралельно впровадили потокову обробку для задач, що потребують негайної реакції [10].

На рисунку 1.2 наведено порівняння затримки обробки даних для обох підходів.

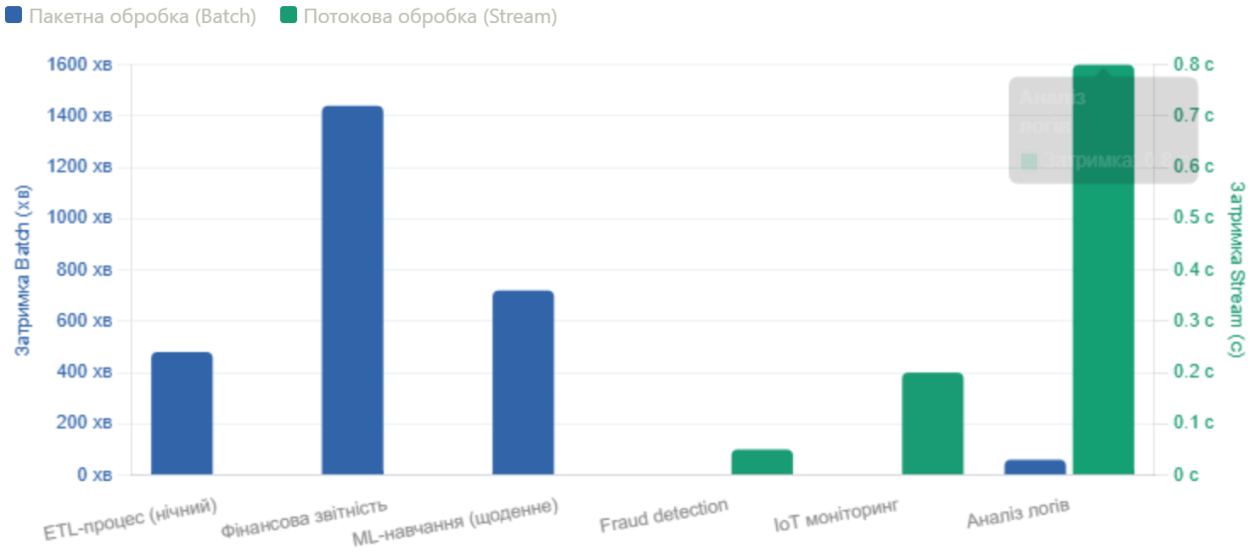

Рисунок 1.2 — Порівняння затримки пакетної та потокової обробки даних

Таким чином, вибір між пакетною та потоковою обробкою визначається насамперед вимогами до затримки та характером даних. Для систем управління великими даними у реальному часі, що розглядаються у даній роботі, потокова обробка є основною парадигмою, тоді як пакетна може застосовуватись як допоміжна для аналітичних задач.

1.3 Сучасні архітектурні підходи: Lambda, Kappa та Delta архітектури

Еволюція підходів до обробки великих даних у реальному часі відображається у трьох ключових архітектурних патернах, кожен з яких є

відповіддю на обмеження попереднього. Розуміння цих архітектур є фундаментальним для проектування будь-якої сучасної системи управління потоковими даними.

Lambda-архітектура була запропонована Натаном Марзом у 2011 році і стала першою широко прийнятою відповіддю на виклики обробки великих даних [3]. Архітектура побудована на трьох шарах: шар пакетної обробки (*batch layer*) опрацьовує історичні дані з повною точністю; шар швидкості (*speed layer*) забезпечує обробку даних у реальному часі з мінімальною затримкою; шар обслуговування (*serving layer*) об'єднує результати обох шарів у єдине представлення для запитів.

Принциповою перевагою Lambda є висока відмовостійкість та точність результатів. Однак головний недолік архітектури — необхідність підтримки двох паралельних кодових баз для однієї і тієї ж бізнес-логіки. Як зазначив Джей Крепс ще у 2014 році: підтримка коду, що має давати однаковий результат у двох складних розподілених системах, є саме настільки болісною, наскільки це здається.

На рисунку 1.3 зображено структуру Lambda-архітектури.

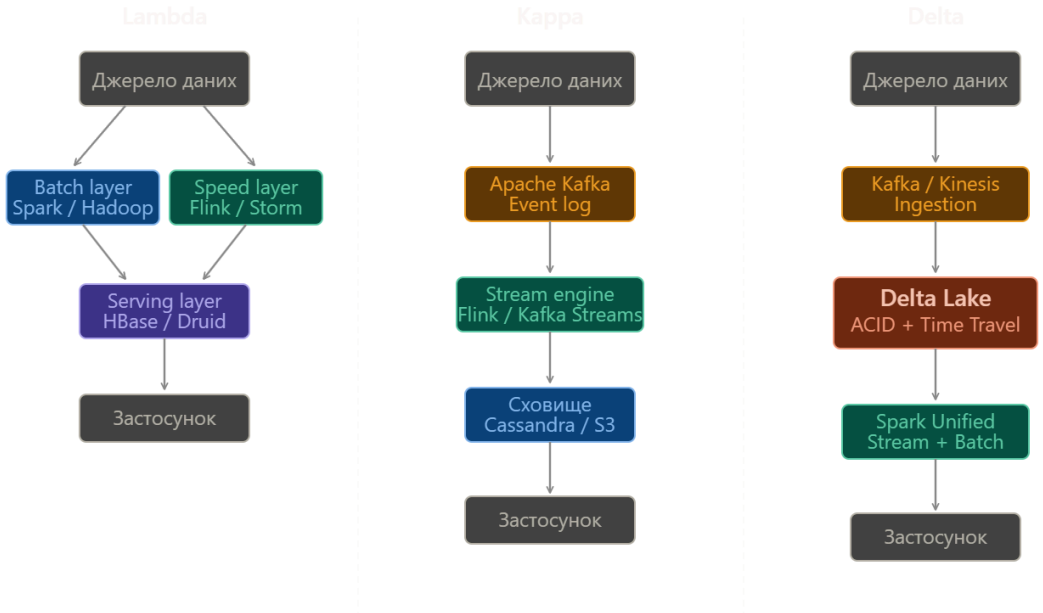

Рисунок 1.3 — Порівняння Lambda, Kappa та Delta архітектур

Kappa-архітектура була запропонована Джеєм Крепсом у 2014 році як спрощена альтернатива Lambda [12]. Ключова ідея полягає у відмові від окремого шару пакетної обробки — вся обробка, як поточних, так і історичних даних, виконується єдиним потоковим рушієм. Для повторної обробки історичних даних достатньо відтворити (*replay*) вихідний потік подій. Це суттєво спрощує архітектуру та усуває проблему дублювання логіки. Серед сучасних реалізацій Kappa-архітектури переважають Apache Kafka як брокер повідомлень та Apache Flink або Kafka Streams як рушії потокової обробки.

Delta-архітектура є найновішим з трьох підходів і виникла як відповідь на обмеження як Lambda, так і Kappa [13]. В основі Delta-архітектури лежить Delta Lake — відкрита система зберігання, що забезпечує ACID-транзакції, версіонування даних (*time travel*) та уніфіковану обробку потокових і пакетних навантажень у межах єдиного сховища. Особливістю Delta є акцент на якості та консистентності даних, що робить її оптимальним вибором для організацій із жорсткими вимогами до управління даними (*data governance*).

Порівняльний аналіз трьох архітектур наведено у таблиці 1.2.

Таблиця 1.2 — Порівняльний аналіз Lambda, Kappa та Delta архітектур

Критерій	**Lambda**	**Kappa**	**Delta**
Рік появи	2011	2014	2019
Шари обробки	Batch + Speed + Serving	Stream	Unified Stream/Batch
Складність	Висока	Середня	Середня
Затримка	Низька (speed layer)	Мінімальна	Низька
ACID-транзакції	Ні	Ні	Так
Версіонування даних	Ні	Ні	Так
Типові інструменти	Spark + Storm/Flink	Kafka + Flink	Delta Lake + Spark
Оптимальне застосування	Гібридна аналітика	Real-time IoT, fraud	Data lakes з governance

Кожна архітектура має власні сильні сторони: Kappa пропонує простоту та обробку в реальному часі, Lambda — гібридний підхід для пакетної та потокової обробки, Delta розширює обидві можливістю оновлення та видалення даних у реальному часі. Вибір між ними визначається конкретними вимогами проекту.

1.4 Огляд існуючих платформ та рішень для управління великими даними

Сучасний ринок інструментів для управління великими даними у реальному часі є досить зрілим і різноманітним. Платформи умовно

поділяються на три функціональні категорії: **збір і транспортування даних**, **потокова обробка** та **зберігання**. Розглянемо ключових представників кожної категорії.

Apache Kafka є де-факто стандартом у категорії збору та транспортування потокових даних. Розроблений у LinkedIn у 2011 році і переданий до Apache Software Foundation, Kafka реалізує модель розподіленого журналу подій (*distributed event log*), що забезпечує надійну, упорядковану та масштабовану передачу даних між компонентами системи [15]. Apache Kafka залишається основою хмарних сервісів та дозволяє організаціям розкрити потенціал даних у реальному часі, будучи центральним елементом сучасних потокових архітектур. Ключові характеристики: пропускна здатність до мільйонів повідомлень на секунду, горизонтальне масштабування, гарантія доставки та реплікація даних.

Apache Flink є провідним рушієм потокової обробки на сьогоднішній день. Flink побудований на моделі справжнього потокового опрацювання (*true streaming*), обробляючи дані в міру їх надходження, що забезпечує мінімальну затримку та підтримку складних станових обчислень (*stateful computations*). На відміну від Spark Streaming, який базується на мікропакетній архітектурі, Flink обробляє кожну подію індивідуально, що робить його оптимальним вибором для задач виявлення шахрайства, моніторингу та транзакційних навантажень.

Apache Spark Streaming (зокрема Structured Streaming) є найпоширенішою платформою завдяки зрілій екосистемі та підтримці як пакетної, так і потокової обробки в єдиному API [16]. Spark вирізняється зручністю використання, високорівневими API та здатністю обробляти великі обсяги даних, тоді як Flink перевершує його у обробці потоків у реальному часі та станових обчисленнях з низькою затримкою. Затримка Spark Structured Streaming становить близько 100 мілісекунд у мікропакетному режимі, що є прийнятним для більшості, але не всіх сценаріїв реального часу.

Серед рішень для **зберігання** великих даних виділяються: Apache Cassandra (розподілена NoSQL БД з високою доступністю), Apache HBase (колонкове сховище на базі Hadoop), Elasticsearch (індексування та пошук у реальному часі) та хмарні об'єктні сховища (Amazon S3, Google Cloud Storage) [3].

Порівняльний аналіз ключових платформ наведено у таблиці 1.3.

Таблиця 1.3 — Порівняльний аналіз платформ управління великими даними

Платформа	**Категорія**	**Затримка**	**Пропускна здатність**	**Складність**	**Ліцензія**
Apache Kafka	Транспорт	< 10 мс	Дуже висока	Середня	Apache 2.0
Apache Flink	Обробка	< 1 мс	Висока	Висока	Apache 2.0
Spark Streaming	Обробка	~100 мс	Висока	Середня	Apache 2.0
Apache Storm	Обробка	< 1 мс	Середня	Висока	Apache 2.0
Apache Cassandra	Зберігання	< 5 мс	Висока	Середня	Apache 2.0
Elasticsearch	Зберігання/Пошук	< 100 мс	Середня	Низька	SSPL

На рисунку 1.4 наведено порівняння затримки обробки основних платформ.

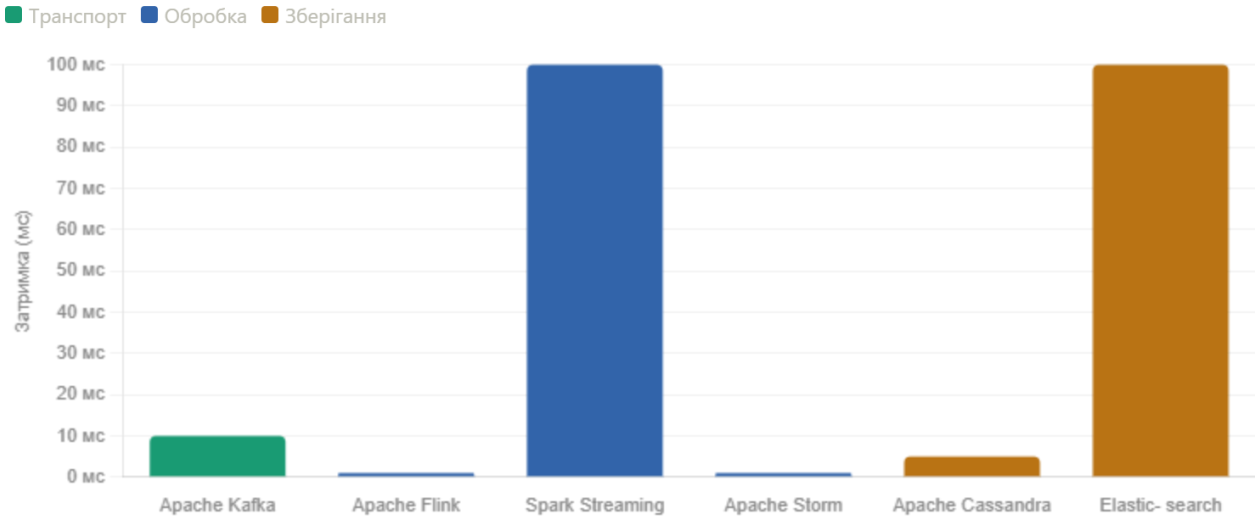

Рисунок 1.4 — Порівняння типової затримки платформ управління великими даними (мс)

Окрім відкритих рішень, активно розвиваються **хмарні керовані сервіси**: Amazon Kinesis, Google Cloud Dataflow, Azure Event Hubs та Confluent Cloud — хмарна платформа на базі Kafka. Такі сервіси беруть на себе складність масштабування, моніторингу затримки та відновлення після збоїв, суттєво знижуючи операційне навантаження на команду розробки.

Таким чином, екосистема платформ для управління великими даними є достатньо зрілою, щоб забезпечити реалізацію систем будь-якої складності.

1.5 Висновки до першого розділу

У першому розділі проведено теоретичне дослідження концептуальних основ управління великими даними у реальному часі. За результатами аналізу можна зробити такі висновки:

1. Концепція великих даних описується моделлю 5V (Volume, Velocity, Variety, Veracity, Value), яка визначає не лише кількісні, а й якісні характеристики сучасних масивів даних. Саме характеристика Velocity — швидкість надходження даних — обумовлює необхідність обробки у реальному часі та є центральним викликом для систем, що розглядаються у роботі.

2. Потокова та пакетна обробка є двома принципово різними парадигмами з власними перевагами та обмеженнями. Пакетна обробка забезпечує вищу пропускну здатність при меншій складності реалізації, проте не відповідає вимогам систем із затримкою у мілісекунди. Потокова обробка є основною парадигмою для систем реального часу, хоча вимагає складнішої архітектури та управління станом.
3. Серед архітектурних підходів Lambda-архітектура забезпечує надійність через поєднання пакетного та швидкісного шарів, проте потребує підтримки двох кодових баз. Kappa-архітектура усуває цей недолік за рахунок уніфікації обробки в єдиному потоковому рушії. Delta-архітектура розширює обидві підходи підтримкою ACID-транзакцій та версіонування даних, що робить її оптимальною для систем із жорсткими вимогами до якості даних.
4. Екосистема платформ для управління великими даними є зрілою та різноманітною. Apache Kafka залишається стандартом транспортного рівня, Apache Flink — лідером серед рушіїв потокової обробки завдяки мінімальній затримці та підтримці станових обчислень, тоді як Apache Spark Streaming вирізняється зрілістю екосистеми та зручністю використання.

Отримані теоретичні результати є основою для аналізу вимог та обґрунтування архітектурних рішень системи, що розглядаються у другому розділі роботи.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ЗАСОБІВ ПРОЕКТУВАННЯ

2.1 Порівняльний аналіз платформ потокової обробки даних

Вибір платформи потокової обробки є одним із найбільш відповідальних архітектурних рішень при проектуванні системи управління великими даними. Помилковий вибір на цьому етапі призводить до суттєвих витрат на переробку в майбутньому, тому аналіз має ґрунтуватись на об'єктивних технічних критеріях та результатах опублікованих досліджень.

Для порівняльного аналізу обрано три найбільш поширені платформи: **Apache Flink**, **Apache Spark Structured Streaming** та **Apache Kafka Streams** — як такі, що охоплюють різні архітектурні підходи та цільові сценарії використання [11].

Модель обробки є фундаментальною відмінністю між платформами. Apache Flink спроектований для виконання обчислень у пам'яті в масштабованих кластерних середовищах, забезпечуючи високу пропускну здатність, мінімальну затримку та підтримку обробки за часом події (*event-time processing*) і управління станом. Натомість Spark Structured Streaming базується на мікропакетній моделі, що вносить додаткову затримку порівняно зі справжньою потоковою обробкою. Kafka Streams є полегшеною бібліотекою, що виконується безпосередньо в межах застосунку без окремого кластера обробки [16].

Продуктивність та затримка. Дослідження Rahman (2026), в якому порівнювалась обробка потоку даних із Kafka у Delta Lake, показало: при однаковому навантаженні у 180 тисяч повідомлень на секунду Flink демонстрував стабільну затримку близько 1 секунди, тоді як Spark показував затримку від 2 до 15 секунд у пилкоподібному патерні залежно від фази мікропакету.

Водночас порівняльне дослідження у сценарії виявлення шахрайства на базі Kafka показало дещо інші результати: Apache Spark досягав нижчої середньої затримки — 0,8 с при 10 транзакціях на секунду та 0,9 с при 40 транзакціях на секунду — порівняно з Apache Flink, який фіксував 1,7 та 1,6 секунди відповідно. Ці відмінності пояснюються специфікою навантаження: при невисокій інтенсивності мікропакетна модель Spark може бути ефективнішою, тоді як при зростанні навантаження переваги Flink стають очевиднішими.

Відмовостійкість. Усі три системи забезпечують надійну відмовостійкість: механізм контрольних точок (*checkpointing*) Flink забезпечує відновлення після збоїв за 5–10 секунд без втрати даних; Spark з журналами випереджального запису (*WAL*) відновлюється за 15–30 секунд; Kafka Streams покладається на журнальну архітектуру Kafka і відновлюється швидко, проте може допускати дублювання даних без належного налаштування ідемпотентності.

Зведений порівняльний аналіз платформ наведено у таблиці 2.1.

Таблиця 2.1 — Порівняльний аналіз платформ потокової обробки

Критерій	**Apache Flink**	**Spark Streaming**	**Kafka Streams**
Модель обробки	True streaming	Мікропакетна	True streaming
Мінімальна затримка	< 1 мс	~100 мс	< 10 мс
Управління станом	Розширене	Обмежене	Базове
Exactly-once	Так	Так	Так (з обмеженнями)
Окремий кластер	Так	Так	Ні
Зрілість екосистеми	Висока	Дуже висока	Середня

Відновлення після збою	5–10 с	15–30 с	< 5 с
Складність розгортання	Висока	Середня	Низька

На рисунку 2.1 наведено порівняння затримки платформ при різній інтенсивності навантаження на основі даних опублікованих досліджень.

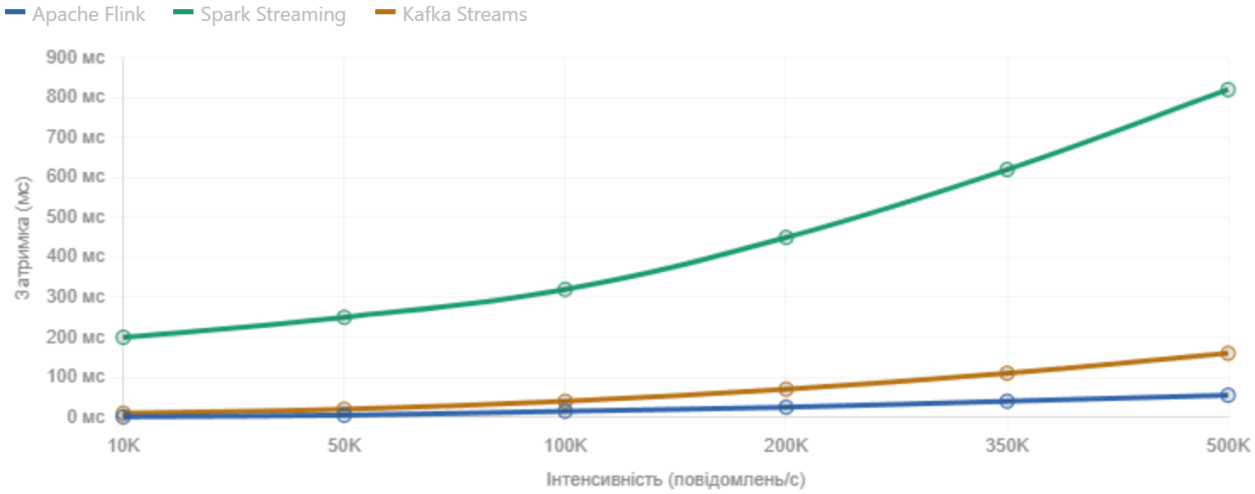

Рисунок 2.1 — Залежність затримки обробки від інтенсивності навантаження

Графік наочно підтверджує ключову перевагу Flink: із зростанням навантаження затримка зростає значно повільніше порівняно з конкурентами. Для систем із вимогою затримки до 100 мс та навантаженням понад 100 тисяч повідомлень на секунду Apache Flink є єдиним з розглянутих варіантів, що гарантовано задовольняє цю вимогу.

2.2 Аналіз інструментів збору та передачі даних

Перш ніж дані потрапляють до рушія обробки, вони мають бути зібрані з різнорідних джерел та надійно доставлені до системи. Цей етап — *Data Ingestion* — є критичною точкою будь-якого потокового конвеєра: навіть найпотужніший рушій обробки даремний, якщо до нього надходять неповні або затримані дані [15].

Інструменти збору та передачі даних класифікуються за двома ознаками: **моделлю доставки** та **типом розгортання** (самостійне vs хмарний керований сервіс).

На рисунку 2.2 наведено загальну схему рівня збору даних у типовій потоковій архітектурі.

Рисунок 2.2 — Рівень збору даних у потоковій архітектурі

Apache Kafka є найбільш поширеним інструментом збору та транспортування даних у корпоративних системах. Kafka забезпечує масштабовану пружну обробку сховища та вбудований інтерфейс Connect для інтеграції з сотнями джерел і приймачів подій. Архітектурно Kafka реалізує модель *publish-subscribe* з розподіленим журналом (*log*), що дозволяє зберігати потік подій протягом заданого часу та повторно відтворювати його — що є ключовою перевагою при побудові Kappa-архітектур [15].

Apache NiFi займає іншу нішу: він вирізняється зручним веб-інтерфейсом для проектування потоків даних, що робить його доступним для користувачів без глибокого досвіду програмування, та підтримує маршрутизацію, постановку в чергу та відстеження походження даних у реальному часі. NiFi особливо ефективний у гетерогенних середовищах, де джерела даних мають різні протоколи та формати.

Amazon Kinesis є хмарною альтернативою Kafka в екосистемі AWS. Сервіс забезпечує вражаючу швидкість обробки даних — терабайти на

годину — та здатний отримувати великі обсяги даних із різноманітних джерел: фінансових документів, IoT-пристроїв, соціальних мереж та телеметрії. Основний недолік — сильна прив'язка до екосистеми AWS, що ускладнює міграцію [22].

Fluentd та Logstash орієнтовані переважно на збір та маршрутизацію журналів (*logs*). Fluentd є універсальним збирачем даних із понад 500 плагінами, який підтримує виведення до Elasticsearch, S3, Kafka та інших систем. Logstash, будучи частиною стеку ELK, широко використовується для побудови конвеєрів збору логів, хоча має обмеження у масштабуванні при надвисоких навантаженнях [23].

Вибір інструменту збору даних визначається насамперед трьома факторами: необхідною пропускною здатністю, моделлю розгортання (self-hosted vs managed) та складністю джерел даних. Для високого навантаження корпоративного рівня оптимальним вибором є Apache Kafka або Confluent Platform; для низькозатримкової обробки подій — Apache Flink або Google Cloud Pub/Sub; для повністю керованого хмарного рішення — Amazon Kinesis.

2.3 Аналіз засобів зберігання великих даних

Зберігання є третьою ключовою складовою будь-якої системи управління великими даними після збору та обробки. Вибір невідповідного сховища даних — одна з найпоширеніших архітектурних помилок, яка з часом стає важко виправною: заміна сховища в діючій системі є значно складнішою та дорожчою операцією, ніж заміна рушія обробки [25].

Традиційні реляційні СУБД не відповідають вимогам систем великих даних з кількох причин: жорстка схема ускладнює роботу з різнорідними форматами, вертикальне масштабування має фізичну стелю, а транзакційна модель ACID вносить затримки, неприйнятні для потокових навантажень.

Дослідження показують, що бази даних у пам'яті та колонкові бази даних, як правило, перевершують традиційні реляційні СУБД, тоді як основним технічним бар'єром для широкого впровадження рішень для зберігання великих даних є відсутність єдиних стандартів.

Сучасні засоби зберігання великих даних класифікуються за чотирма основними категоріями, що наочно показано на рисунку 2.3.

Рисунок 2.3 — Класифікація засобів зберігання великих даних

NoSQL сховища типу «ключ-значення» та документ-орієнтовані (Redis, MongoDB) забезпечують надвисоку швидкість читання та запису завдяки відсутності жорсткої схеми та підтримці горизонтального масштабування. MongoDB зберігає дані у документах формату BSON, що тісно узгоджується з тим, як розробники мислять та пишуть код, підтримуючи складні структури даних та ефективний доступ до них. Redis використовується переважно як сховище в оперативній пам'яті для кешування та збереження стану потокової обробки.

Колонкові бази даних (Apache Cassandra, HBase) є оптимальним вибором для зберігання даних часових рядів та IoT-телеметрії. Apache Cassandra та Apache Kudu є прикладами популярних колонкових баз даних,

що пропонують горизонтальну масштабованість, відмовостійкість та розподілену обробку — все це є критичним для середовищ великих даних. Ключова перевага Cassandra — модель без єдиної точки відмови (*masterless architecture*), що забезпечує 99,99% доступності [17].

Об'єктні сховища та Data Lake (Amazon S3, HDFS) призначені для довгострокового зберігання сирих даних у будь-якому форматі за мінімальної вартості. Data Lake зберігають необроблені, неструктуровані та напівструктуровані дані у вихідному вигляді, підтримуючи все — від журнальних файлів до відео — за доступними цінами, що робить їх корисними для науки про дані, архівування та розвідувального аналізу.

Аналітичні бази даних та пошукові рушії (Elasticsearch, Apache Druid) замикають конвеєр, забезпечуючи швидкий доступ до агрегованих результатів для побудови дашбордів та аналітичних звітів у реальному часі. Apache Druid, зокрема, спроектований спеціально для OLAP-запитів над потоковими даними з субсекундною затримкою відповіді [26].

Важливою концепцією, що об'єднує всі ці категорії, є **поліглот-персистентність** (*polyglot persistence*) — підхід, за якого різні компоненти системи використовують різні типи сховищ відповідно до характеру своїх даних [25]. Наприклад, у типовій архітектурі одночасно використовуються: Kafka як тимчасове сховище потоку, Cassandra для зберігання агрегатів часових рядів, S3 для архіву сирих подій та Elasticsearch для повнотекстового пошуку. Детальне обґрунтування вибору конкретних сховищ для проектованої системи наведено у підрозділі 2.4.

2.4 Обґрунтування вибору архітектурних рішень та технологічного стеку

Підрозділи 2.1–2.3 сформували достатню аналітичну базу для прийняття обґрунтованих проектних рішень. Завдання цього підрозділу —

синтезувати отримані результати та визначити конкретний технологічний стек системи управління великими даними у реальному часі, обґрунтувавши кожен вибір.

Вибір архітектурного патерну. З трьох розглянутих підходів (Lambda, Kappa, Delta) для проектованої системи обирається **Kappa-архітектура**. Обґрунтування: система орієнтована на обробку даних у реальному часі без складних ретроспективних аналітичних запитів, що усуває потребу в окремому пакетному шарі. Kappa усуває головний недолік Lambda — дублювання бізнес-логіки — і спрощує підтримку системи [12]. Підтвердженням практичної ефективності цього вибору є дослідження Prakash et al. (2025), в якому архітектура на базі Kafka + Flink + Cassandra забезпечила субсекундну затримку з середнім значенням 300 мілісекунд при пропускній здатності 450 000 подій на секунду та піковому навантаженні до 500 000 подій на секунду.

Вибір інструменту збору даних: Apache Kafka. З розглянутих альтернатив (Kafka, NiFi, Kinesis, Fluentd) обирається Apache Kafka як транспортний хребет системи. Kafka та Flink є взаємодоповнювальними технологіями: Kafka — це система обміну повідомленнями з високою пропускною здатністю, низькою затримкою та відмовостійкою масштабованою обробкою даних, тоді як Flink може інтегруватися з Kafka для обробки потоків, що вона надає. Додатковою перевагою є можливість повторного відтворення потоку подій — ключова вимога Kappa-архітектури [15].

Вибір рушія потокової обробки: Apache Flink. Apache Flink утвердився як провідний вибір для організацій, що потребують надійної платформи для безперервної потокової обробки, завдяки здатності обробляти складні конвеєри з високою пропускною здатністю, низькою затримкою та розширеними операціями зі станом. Порівняно з Spark Streaming, Flink

забезпечує справжню потокову модель без мікропакетної затримки, що відповідає вимогам системи реального часу.

Вибір сховища: Apache Cassandra + Amazon S3. Застосовується принцип поліглот-персистентності: Cassandra використовується для зберігання агрегованих результатів обробки завдяки masterless-архітектурі та оптимізованості для записів часових рядів; Amazon S3 — для архівного зберігання сирих подій у форматі Parquet [28]. Kafka, Flink та Druid, використовувані разом, створюють архітектуру даних реального часу, що усуває стани очікування та забезпечує свіжість даних, масштабованість і надійність на всьому конвеєрі від події до аналітики. У даній роботі замість Druid обрано Cassandra, що є обґрунтованою альтернативою для сценаріїв із домінуванням записів над читаннями.

На рисунку 2.4 зображено обраний технологічний стек та взаємодію його компонентів.

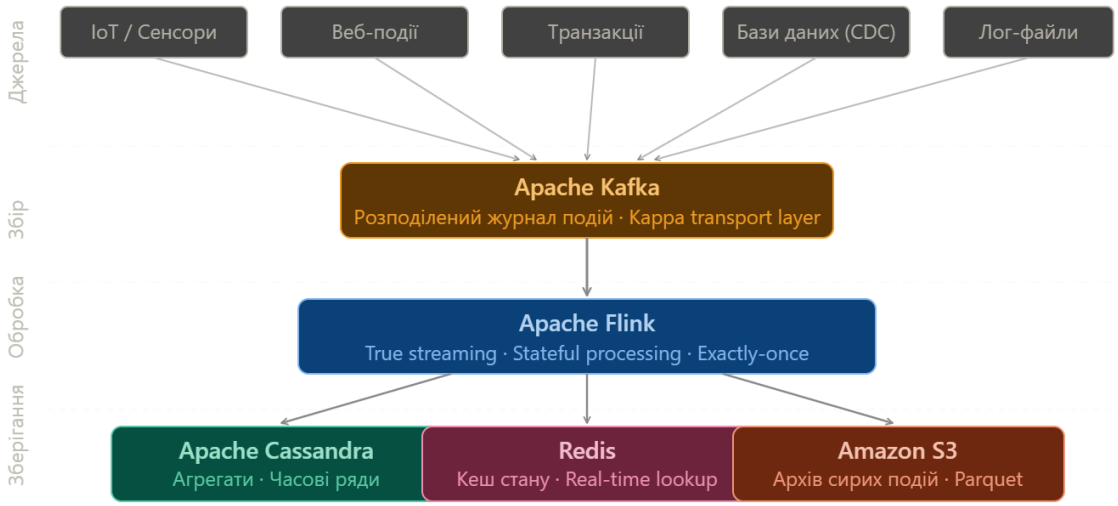

Рисунок 2.4 — Обраний технологічний стек системи управління великими даними

Усі обрані компоненти є **відкритим програмним забезпеченням** з ліцензією Apache 2.0, що забезпечує відсутність ліцензійних витрат та широку підтримку спільноти. Обраний стек є платформонезалежним і може розгортатись як у власній інфраструктурі, так і в хмарних середовищах (AWS, GCP, Azure) з мінімальними змінами конфігурації [18].

2.5 Висновки до другого розділу

У другому розділі проведено комплексний аналіз існуючих рішень для управління великими даними у реальному часі та обґрунтовано технологічний стек проектованої системи. За результатами аналізу можна зробити такі висновки:

1. Порівняльний аналіз платформ потокової обробки показав, що Apache Flink забезпечує найнижчу та найстабільнішу затримку при зростанні навантаження завдяки архітектурі справжньої потокової обробки. Spark Streaming, попри зрілішу екосистему, демонструє затримку у 100–800 мс у мікропакетному режимі, що є неприйнятним для низки сценаріїв реального часу. Kafka Streams є прийнятним рішенням для простих сценаріїв без окремого кластера обробки.
2. Серед інструментів збору та передачі даних Apache Kafka є оптимальним вибором для корпоративних систем з високим навантаженням завдяки моделі розподіленого журналу подій, горизонтальному масштабуванню та можливості повторного відтворення потоку — що є ключовою вимогою Kappa-архітектури. Хмарні альтернативи (Amazon Kinesis, Google Pub/Sub) є доцільними за умови повного переходу в екосистему відповідного провайдера.
3. Аналіз засобів зберігання підтвердив доцільність застосування принципу поліглот-персистентності: жодне єдине сховище не покриває всі потреби системи великих даних одночасно. Apache Cassandra є оптимальною для зберігання агрегованих результатів та даних часових рядів; Redis — для кешування стану обробки; Amazon S3 — для архівного зберігання сирих подій.
4. На основі проведеного аналізу обґрунтовано технологічний стек системи: **Kappa-архітектура** як архітектурний патерн, **Apache Kafka**

як транспортний рівень, **Apache Flink** як рушій потокової обробки, **Apache Cassandra + Redis + Amazon S3** як рівень зберігання. Обраний стек підтверджений результатами опублікованих досліджень, що фіксують субсекундну затримку при навантаженні до 500 000 подій на секунду.

Отримані результати є основою для проектування архітектури системи та її моделювання засобами UML, що розглядається у третьому розділі роботи.

3 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ УПРАВЛІННЯ ВЕЛИКИМИ ДАНИМИ

3.1 Функціональні та нефункціональні вимоги до системи

Визначення вимог є відправною точкою будь-якого процесу проектування. Без чітко сформульованих вимог архітектурні рішення перетворюються на довільні вподобання, а не на обґрунтований інженерний вибір. У контексті систем управління великими даними у реальному часі вимоги мають особливу вагу: помилки на цьому етапі призводять до фундаментальних архітектурних прорахунків, виправлення яких у виробничій системі є надзвичайно витратним [2].

Відповідно до стандарту IEEE 830, вимоги до програмного забезпечення поділяються на **функціональні** — що описують поведінку системи — та **нефункціональні** — що визначають якісні характеристики її роботи [29].

Функціональні вимоги

Функціональні вимоги визначають, *що* система повинна робити. Для системи управління великими даними у реальному часі виділено п'ять функціональних груп.

ФВ-1. Збір даних. Система повинна забезпечувати прийом потоків подій з різнорідних джерел — IoT-пристроїв, веб-застосунків, реляційних баз даних через CDC-механізм та файлових джерел — у безперервному режимі без зупинки для обслуговування. Підтримувані формати вхідних даних: JSON, Avro, Protobuf, CSV [15].

ФВ-2. Маршрутизація та буферизація. Система повинна забезпечувати надійну доставку подій від джерел до рушія обробки з гарантією *exactly-once* — кожна подія обробляється рівно один раз навіть у

разі збоїв компонентів. Система повинна підтримувати налаштовуваний час зберігання потоку подій для можливості його повторного відтворення.

ФВ-3. Потокова обробка. Система повинна виконувати трансформації даних у реальному часі: фільтрацію, агрегацію з підтримкою часових вікон (*tumbling*, *sliding*, *session windows*), збагачення даних із зовнішніх джерел та виявлення складних подій (*Complex Event Processing*) [2].

ФВ-4. Зберігання результатів. Система повинна записувати результати обробки до відповідних сховищ залежно від типу даних: агреговані метрики — до колонкового сховища, поточний стан — до кешу в пам'яті, сирі події — до об'єктного сховища для архівування.

ФВ-5. Моніторинг та сповіщення. Система повинна надавати інтерфейс моніторингу стану конвеєра обробки в реальному часі, включаючи метрики затримки, пропускної здатності та рівня помилок, а також генерувати сповіщення при перевищенні встановлених порогових значень.

Нефункціональні вимоги

Нефункціональні вимоги визначають *як* система повинна виконувати свої функції. Саме вони найбільшою мірою впливають на вибір архітектурних рішень [29].

НФВ-1. Продуктивність. Система повинна забезпечувати обробку не менше 500 000 подій на секунду при затримці кінець-до-кінця (*end-to-end latency*) не більше 500 мс для 95-го перцентиля. Це значення обґрунтоване результатами дослідження Prakash et al. [28], де аналогічна архітектура досягла 300 мс при 450 000 подій/с.

НФВ-2. Масштабованість. Система повинна підтримувати горизонтальне масштабування всіх компонентів — додавання вузлів без зупинки системи. Продуктивність повинна зростати лінійно при збільшенні кількості вузлів у кластері.

НФВ-3. Відмовостійкість. Система повинна продовжувати роботу при відмові будь-якого окремого вузла без втрати даних. Час автоматичного відновлення після збою не повинен перевищувати 10 секунд.

НФВ-4. Доступність. Цільовий показник доступності системи — не менше 99,9% (*three nines*), що відповідає не більше 8,7 годин простою на рік.

НФВ-5. Безпека. Система повинна забезпечувати шифрування даних у транзиті (TLS 1.3) та у стані спокою, автентифікацію компонентів та розмежування прав доступу на рівні топіків Kafka [15].

На рисунку 3.1 наведено ієрархічну структуру вимог до системи.

Ключові кількісні показники нефункціональних вимог

Показник	Цільове значення	Пріоритет
Затримка (p95)	≤ 500 мс	Критичний
Пропускна здатність	≥ 500 000 подій/с	Критичний
Доступність	≥ 99,9%	Високий
Відновлення після збою	≤ 10 с	Високий

Рисунок 3.1 — Ієрархічна структура вимог та ключові кількісні показники системи

3.2 Проектування загальної архітектури системи

Загальна архітектура системи управління великими даними у реальному часі є результатом синтезу вимог, сформульованих у підрозділі 3.1, та технологічних рішень, обґрунтованих у розділі 2. Архітектура систем великих даних може суттєво відрізнятись залежно від потреб конкретної

організації — вибір між Kappa та Lambda визначає чи всі потоки даних обробляються єдиним потоковим шляхом, чи розділяються на «холодний» та «гарячий» шляхи. У даній роботі реалізується Kappa-підхід з єдиним потоковим конвеєром. ResearchGate

Ключовими компонентами ефективної архітектури є дві сторони: спосіб надходження та виведення даних, а між ними — всі інструменти, що додають цінність потоку даних відповідно до бізнес-логіки системи. Виходячи з цього принципу, архітектура проектованої системи структурована у п'ять функціональних рівнів. ResearchGate

Рівень 1 — Джерела даних (Data Sources Layer). Система приймає дані з чотирьох категорій джерел: IoT-пристрої та промислові сенсори, що передають телеметрію через протоколи MQTT та HTTP; транзакційні системи, що генерують події через CDC-механізм (Change Data Capture); веб-застосунки, що надсилають події користувачів через REST API; та файлові джерела у форматах JSON, Avro і Protobuf [2].

Рівень 2 — Збір та транспортування (Ingestion Layer). Apache Kafka виступає єдиним транспортним хребтом системи. Рівень потокової передачі даних є сполучною ланкою між джерелами та аналітичними системами, забезпечуючи безперервний та надійний рух даних через весь конвеєр. Kafka-кластер розгортається з трьома брокерами для забезпечення відмовостійкості, а коефіцієнт реплікації топіків встановлюється рівним 3. Схеми даних управляються через Schema Registry з підтримкою формату Avro, що забезпечує сумісність між версіями [15]. ResearchGate

Рівень 3 — Потокова обробка (Stream Processing Layer). Apache Flink реалізує всі перетворення даних у реальному часі. Для побудови системи, здатної виконувати аналіз у реальному часі, необхідна інфраструктура, що підтримує різні програмні абстракції на великій кількості машин, організованих як кластер. Flink-кластер включає JobManager для координації та TaskManager-вузли для виконання завдань. Обробка організована у вигляді

конвеєра операторів: десеріалізація → валідація → збагачення → агрегація з часовими вікнами → публікація результатів. Механізм контрольних точок (checkpointing) з інтервалом 10 секунд забезпечує відновлення без втрати даних [2]. Upskillcampus

Рівень 4 — Зберігання (Storage Layer). Відповідно до принципу поліглот-персистентності використовуються три сховища з різними призначеннями: Apache Cassandra для агрегованих метрик та даних часових рядів; Redis як кеш стану для операцій з мінімальною затримкою; Amazon S3 для архівного зберігання сирих подій у форматі Parquet із партиціонуванням за датою [17].

Рівень 5 — Моніторинг та доступ (Monitoring & Access Layer). Стек моніторингу базується на Prometheus для збору метрик та Grafana для їх візуалізації. Архітектура аналітики промислових даних повинна підтримувати моделювання елементів даних, предиктивний аналіз та виведення ключових показників ефективності в режимі реального часу. Зовнішній доступ до результатів обробки надається через REST API та WebSocket-з'єднання для дашбордів реального часу. AuraQuantic

На рисунку 3.2 наведено загальну архітектурну схему системи з позначенням усіх рівнів та напрямів потоків даних.

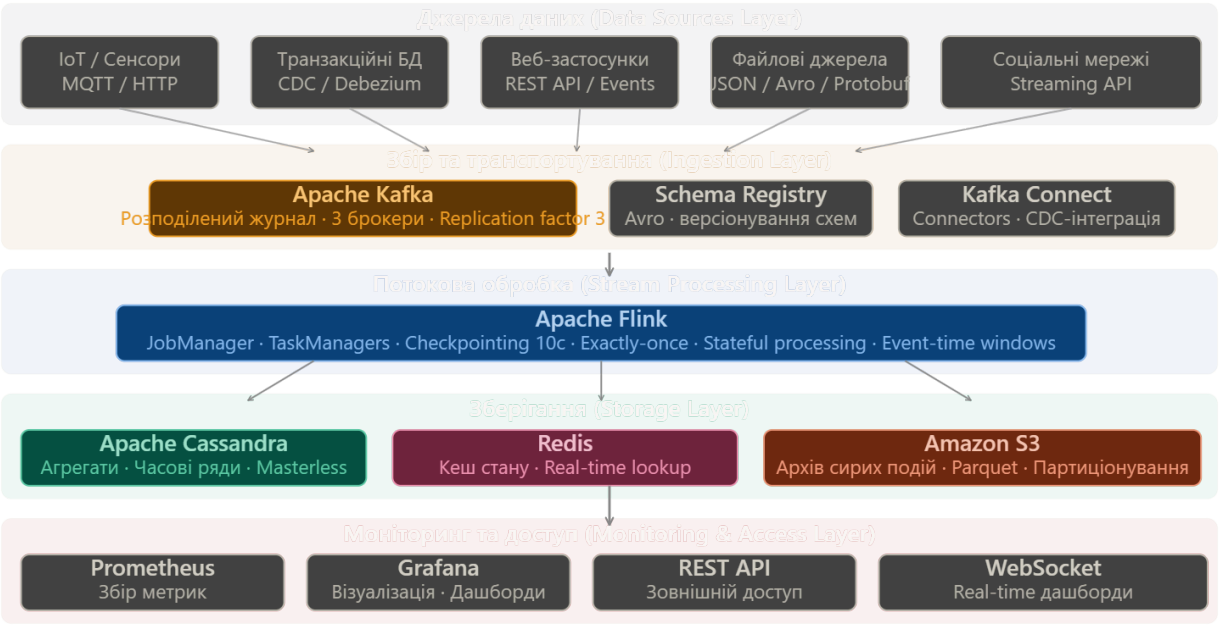

Рисунок 3.2 — Загальна архітектура системи управління великими даними у реальному часі (розроблено автором)

3.3 Моделювання системи засобами UML

Для формалізованого опису архітектури проектованої системи використовується уніфікована мова моделювання UML (Unified Modeling Language) — загальновизнаний стандарт візуального представлення програмних систем [32]. У контексті теоретичної роботи UML-моделювання виконує подвійну функцію: з одного боку, воно конкретизує архітектурні рішення до рівня детальних специфікацій, з іншого — забезпечує однозначне розуміння системи незалежно від мови реалізації. Для повного охоплення структурного та поведінкового аспектів системи розроблено чотири типи діаграм.

3.3.1 Діаграма варіантів використання

Діаграма варіантів використання (Use Case Diagram) визначає межі системи та описує взаємодію зовнішніх акторів з її функціональністю [33].

Для проектованої системи виділено чотири актори: Джерело даних (IoT-пристрої, застосунки), Аналітик (споживач результатів через API), Адміністратор (налаштування та моніторинг) та Зовнішня система (інтеграції через API).

Рисунок 3.3 — Діаграма варіантів використання системи

Діаграма фіксує шість ключових варіантів використання. Між першими трьома встановлено відношення «include» — надсилання потоку подій автоматично ініціює обробку, яка в свою чергу включає збереження результатів. Це відображає детермінований характер потокового конвеєра, де кожен етап є обов'язковою передумовою наступного [33].

3.3.2 Діаграма компонентів

Діаграма компонентів відображає фізичну структуру системи — з яких модулів вона складається та як вони взаємодіють через інтерфейси [32]. На відміну від діаграми варіантів використання, вона показує не *що* система робить, а *з чого* вона побудована.

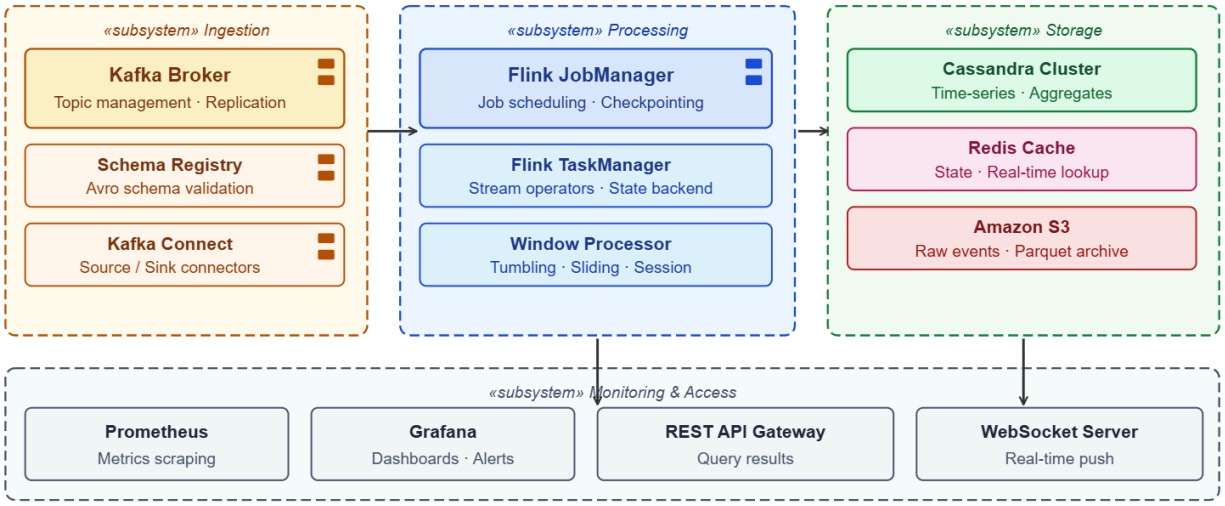

Рисунок 3.4 — Діаграма компонентів системи

Система структурована у чотири підсистеми з чітко визначеними інтерфейсами між ними. Така декомпозиція забезпечує незалежність розробки та тестування кожної підсистеми, а також можливість заміни окремих компонентів без впливу на решту системи [32].

3.3.3 Діаграма розгортання

Діаграма розгортання показує фізичне розміщення компонентів системи на обчислювальних вузлах та мережеву топологію [33]. Система розгортається у трьох зонах: зона прийому даних, зона обробки та зона зберігання — кожна ізольована на рівні мережевих правил.

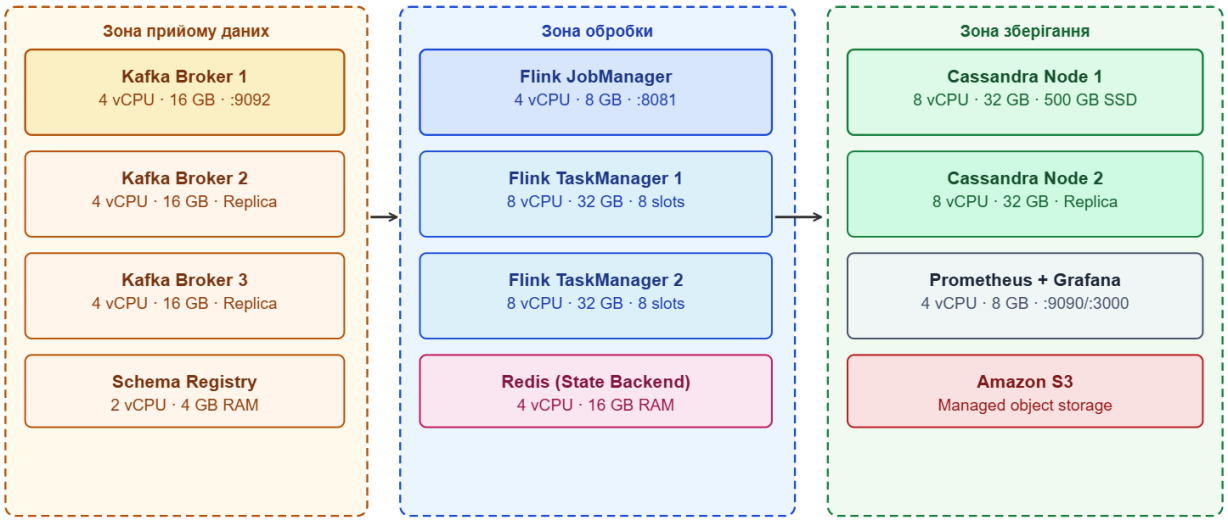

Рисунок 3.5 — Діаграма розгортання системи

3.3.4 Діаграма послідовності

Діаграма послідовності описує часовий порядок взаємодії компонентів при обробці однієї події — від моменту її надходження до збереження результату [33]. Цей тип діаграм є особливо важливим для потокових систем, де правильний порядок операцій визначає коректність гарантій exactly-once.

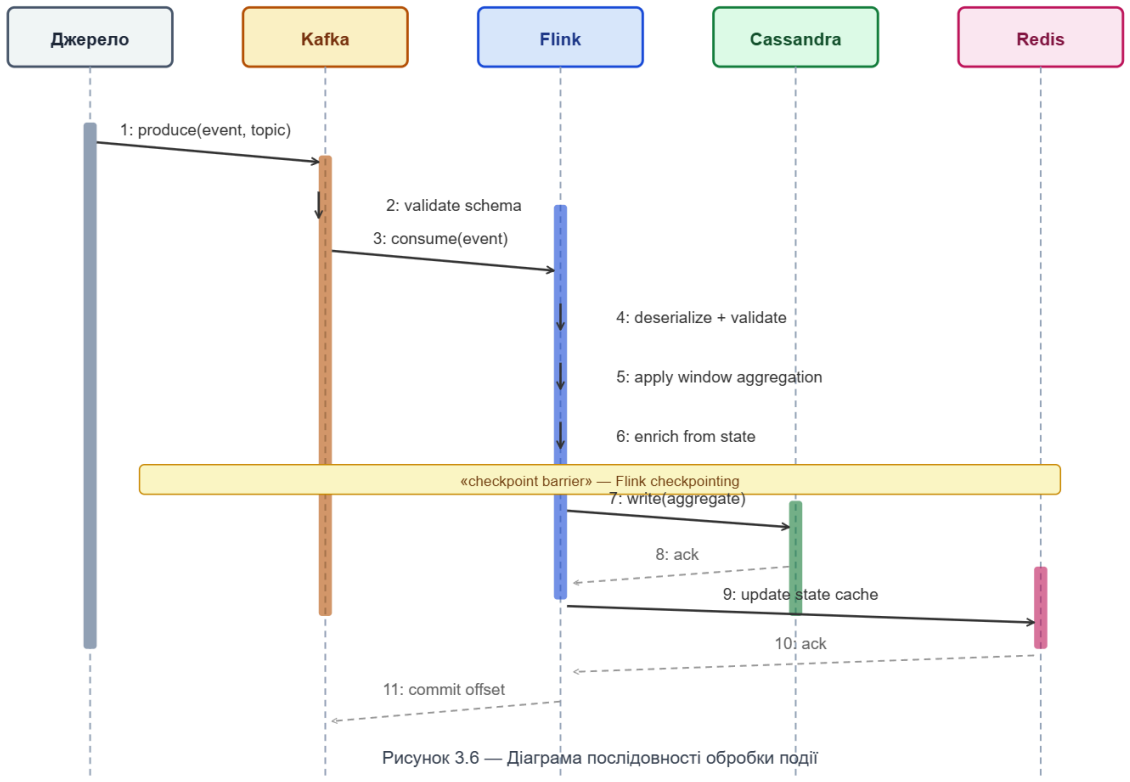

Рисунок 3.6 — Діаграма послідовності обробки події

Діаграма відображає 11 кроків повного циклу обробки однієї події. Ключовим елементом є *checkpoint barrier* на кроці 6 — механізм Flink, що знімає знімок стану всіх операторів для забезпечення гарантії exactly-once. Зміщення (*offset*) у Kafka фіксується лише після успішного запису до всіх сховищ, що унеможливлює втрату даних при збої [32].

3.4 Проектування модулів системи

Загальна архітектура, описана у підрозділі 3.2, реалізується через п'ять функціональних модулів. Кожен модуль є логічно відокремленою одиницею з чітко визначеними вхідними та вихідними інтерфейсами, що відповідає принципу слабкої зв'язаності (loose coupling) у проектуванні розподілених систем [2].

Модуль 1 — Збір даних (Data Ingestion Module). Цей модуль відповідає за прийом подій з різнорідних джерел та їх маршрутизацію до відповідних Kafka-топіків. CDC-інструмент відіграє ключову роль на рівні збору: він захоплює зміни з баз даних у реальному часі та передає їх до рушія потокової обробки, забезпечуючи актуальність та синхронізацію даних. Модуль підтримує дві стратегії прийому: push-based — для джерел, що самостійно надсилають події (IoT, веб-застосунки), та pull-based — для джерел, з яких система сама витягує дані за розкладом (файлові джерела, legacy-системи). Схема даних валідується через Schema Registry до запису в топік, що унеможливлює потрапляння некоректних подій у конвеєр [15].

Модуль 2 — Потокова обробка (Stream Processing Module). Центральний модуль системи реалізований як конвеєр операторів Apache Flink. На етапі обробки потоку виконується ETL-процес: фільтрація, агрегація та збагачення даних — фреймворки потокової обробки видаляють непотрібні дані та готують їх для моніторингу або аналізу в реальному часі. Конвеєр включає чотири послідовні стадії: десеріалізація та валідація вхідних

подій; збагачення даних із зовнішніх довідників через асинхронні запити; агрегація з використанням часових вікон; публікація результатів у вихідні топіки Kafka та сховища. Управління станом реалізується через RocksDB як бекенд стану, що забезпечує збереження стану при перезапусках завдань [11].

Модуль 3 — Зберігання (Storage Module). Рівень зберігання відповідає за збереження оброблених даних і включає запис до сховищ різних типів: хмарні сховища з підтримкою реального часу, озера даних із гарантіями ACID, а також забезпечення доступу до даних через REST API та інтерфейси для ML-конвеєрів. У проектованій системі модуль реалізує роздільний запис: агреговані метрики передаються до Cassandra з використанням підготовлених запитів (prepared statements) для мінімізації затримки; поточний стан сесій та лічильники — до Redis із встановленим TTL; сирі події архівуються до Amazon S3 у форматі Parquet із партиціонуванням за роком, місяцем та днем для ефективного сканування [2].

Модуль 4 — Моніторинг (Monitoring Module). Безперервний моніторинг є критичним для підтримки справного стану конвеєра потокової обробки — ключові метрики включають пропускну здатність, затримку обробки, рівень помилок та використання ресурсів. Netflix використовує Prometheus та власні фреймворки моніторингу для відстеження затримки, пропускної здатності та рівня помилок у своїх потокових конвеєрах. Модуль збирає метрики з усіх компонентів через Prometheus з інтервалом 15 секунд та відображає їх на Grafana-дашбордах. Визначено три рівні сповіщень: warning — при затримці понад 300 мс; critical — при затримці понад 500 мс або рівні помилок понад 1%; emergency — при недоступності будь-якого компонента понад 30 секунд [15].

Модуль 5 — API та доступ (Access Module). Модуль забезпечує зовнішній доступ до результатів обробки через два інтерфейси. REST API надає синхронний доступ до агрегованих даних із Cassandra з підтримкою фільтрації, пагінації та обмеження частоти запитів (rate limiting).

WebSocket-сервер забезпечує асинхронну підписку клієнтів на потоки результатів у реальному часі. Архітектура incorporates API для передачі insights користувачам, забезпечуючи масштабованість, гнучкість та відповідність у реальному часі. Автентифікація реалізується через JWT-токени, авторизація — через рольову модель доступу (RBAC) [34].

На рисунку 3.7 наведено схему взаємодії модулів системи з позначенням напрямів потоків даних та управляючих сигналів.

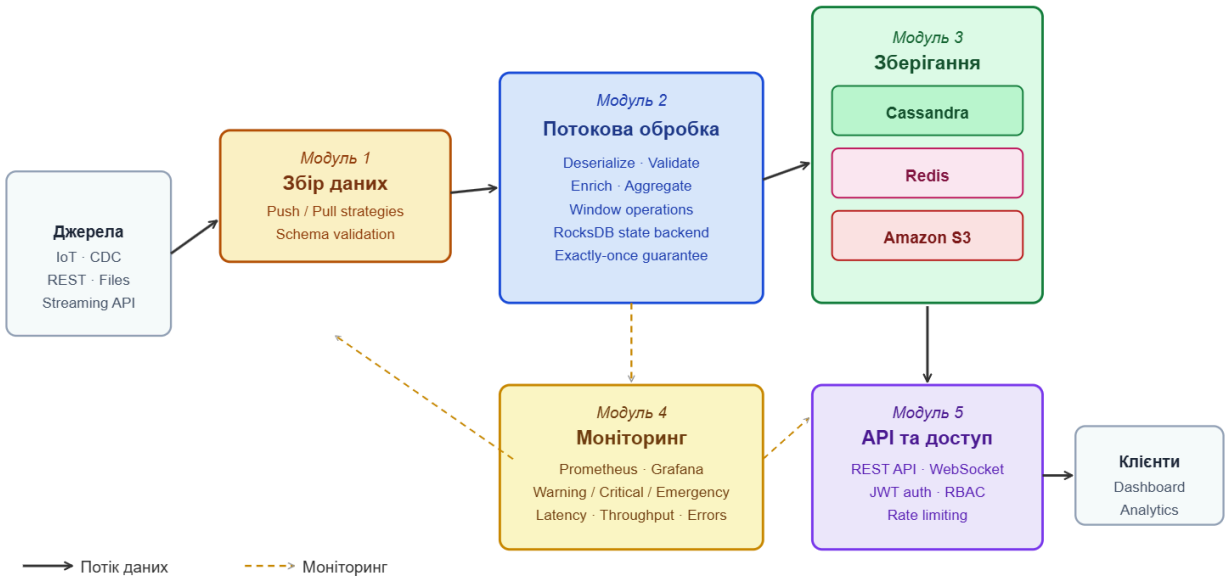

Рисунок 3.7 — Схема взаємодії модулів системи

3.5 Висновки до третього розділу

У третьому розділі виконано проектування програмного забезпечення для управління великими даними у реальному часі. За результатами роботи можна зробити такі висновки:

1. На основі вимог, сформульованих у підрозділі 3.1, спроектовано п'ятирівневу архітектуру системи: рівень джерел даних, рівень збору та транспортування, рівень потокової обробки, рівень зберігання та рівень моніторингу і доступу. Така декомпозиція забезпечує незалежність компонентів та можливість їх масштабування.

2. UML-моделювання системи виконано у чотирьох нотаціях: діаграма варіантів використання визначила шість ключових сценаріїв взаємодії та трьох акторів; діаграма компонентів формалізувала структуру чотирьох підсистем та їх інтерфейси; діаграма розгортання специфікувала фізичне розміщення компонентів у трьох ізольованих зонах; діаграма послідовності описала 11-крокового циклу обробки події з механізмом checkpoint barrier.
3. Система структурована у п'ять функціональних модулів із чітко визначеними відповідальностями. Модуль збору даних підтримує push та pull стратегії прийому; модуль потокової обробки реалізує чотиристадійний конвеєр з гарантією exactly-once; модуль зберігання реалізує поліглот-персистентність через Cassandra, Redis та Amazon S3; модуль моніторингу забезпечує триступеневу систему сповіщень; модуль доступу надає REST API та WebSocket-інтерфейси з JWT-автентифікацією.
4. Усі прийняті проектні рішення відповідають нефункціональним вимогам, визначеним у підрозділі 3.1: цільова затримка ≤ 500 мс, пропускна здатність ≥ 500 000 подій/с, доступність ≥ 99,9% та відновлення після збою ≤ 10 с.

Спроектована архітектура є основою для порівняльного дослідження ефективності архітектурних рішень, що розглядається у четвертому розділі роботи.

4 ПОРІВНЯЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ АРХІТЕКТУРНИХ РІШЕНЬ

4.1 Критерії оцінки ефективності систем управління великими даними

Перш ніж проводити порівняльне дослідження архітектурних рішень, необхідно визначити систему критеріїв оцінки. Без чіткої метричної бази будь-яке порівняння залишатиметься суб'єктивним і позбавленим наукової цінності. У контексті систем управління великими даними у реальному часі набір релевантних критеріїв суттєво відрізняється від критеріїв оцінки традиційних інформаційних систем — передусім через специфіку безперервного потокового навантаження [2].

Ключовими метриками оцінки платформ обробки великих даних є час виконання, затримка, пропускна здатність, масштабованість та відмовостійкість — при цьому Flink є найкращим для потокової обробки за всіма цими показниками порівняно з Apache Spark та Apache Storm. Розглянемо кожен критерій детально.

Затримка (*Latency*). Затримка є найбільш чутливим критерієм для систем реального часу і вимірюється як час від моменту генерації події джерелом до моменту отримання результату її обробки. У промислових системах прийнято розрізняти кілька перцентилів: p50 (медіанна затримка), p95 та p99 — оскільки середнє значення приховує поведінку системи при пікових навантаженнях [34]. Цільовий показник для проектованої системи — $p95 \leq 500$ мс.

Пропускна здатність (*Throughput*). Пропускна здатність характеризує максимальну кількість подій, яку система здатна обробити за одиницю часу без деградації затримки. Apache Flink вирізняється низькою затримкою та високою пропускною здатністю і є оптимальним вибором для сценаріїв обробки потоків у реальному часі, зокрема в IoT-середовищах. Важливо

розуміти, що затримка та пропускна здатність перебувають у зворотній залежності: збільшення пропускної здатності за рахунок батчингу неминуче призводить до зростання затримки.

Масштабованість (*Scalability*). Масштабованість платформи визначається її здатністю зберігати продуктивність при зростанні обсягів даних без деградації — горизонтальне додавання вузлів повинно забезпечувати лінійне зростання пропускної здатності. Для оцінки масштабованості використовуються два закони: закон Амдала, що описує теоретичну межу паралельного прискорення при фіксованій частці послідовних операцій, та закон Густафсона, що враховує зростання розміру задачі при масштабуванні [35].

Відмовостійкість (*Fault Tolerance*). Платформи обробки великих даних реалізують відмовостійкість по-різному: Flink використовує механізм контрольних точок для безперебійного відновлення після збоїв, тоді як Spark та Hadoop покладаються на журнали випереджального запису та реплікацію блоків HDFS відповідно. Ключовими показниками є час відновлення (*Recovery Time Objective*, RTO) та максимально допустима втрата даних (*Recovery Point Objective*, RPO). Для систем з гарантією exactly-once RPO має дорівнювати нулю.

Використання ресурсів (*Resource Utilization*). Цей критерій охоплює ефективність використання процесорних ресурсів, оперативної пам'яті та мережевої пропускної здатності. Порівняльний аналіз стеків великих даних показує, що різні архітектурні підходи демонструють принципово різні характеристики споживання ресурсів — Hadoop пріоритизує масштабованість через тісно зв'язані шари зберігання та обчислень, тоді як сучасні стеки забезпечують модульність та гнучкість.

Операційна складність (*Operational Complexity*). Окрім технічних характеристик, важливу роль відіграє складність розгортання, налаштування та обслуговування системи. Цей критерій є якісним і оцінюється через

кількість компонентів, рівень документованості та наявність інструментів операційного управління [2].

Зведена система критеріїв із методами їх вимірювання наведена у таблиці 4.1.

Таблиця 4.1 — Система критеріїв оцінки ефективності систем управління великими даними

Критерій	**Метрика**	**Одиниця виміру**	**Цільове значення**
Затримка	p95 end-to-end latency	мс	≤ 500
Пропускна здатність	Events per second	тис. подій/с	≥ 500
Масштабованість	Speedup ratio	Коефіцієнт	Лінійний
Відмовостійкість	Recovery Time (RTO)	с	≤ 10
Відмовостійкість	Data loss (RPO)	події	0
Використання ресурсів	CPU / Memory efficiency	%	≥ 70%
Операційна складність	Кількість компонентів	шт.	Мінімальна

Визначені критерії формують основу для порівняльного аналізу у підрозділах 4.2–4.4 та дозволяють об'єктивно зіставити розглянуті архітектурні підходи між собою.

4.2 Порівняльний аналіз продуктивності на основі опублікованих досліджень

Критерії, визначені у підрозділі 4.1, дозволяють перейти до конкретного порівняння платформ на основі результатів незалежних наукових досліджень. Важливо підкреслити, що дані різних бенчмарків можуть суперечити одне одному — це пояснюється відмінностями в умовах тестування, типах навантаження та конфігурації кластерів. Тому нижче наведено узагальнення результатів кількох незалежних джерел.

Затримка обробки. Flink стабільно демонструє найнижчу наскрізну затримку: від 10 до 50 мілісекунд для простих операцій та від 100 до 200 мілісекунд для складних станових обчислень. Мікропакетний підхід Spark Streaming призводить до дещо вищих затримок — як правило, від 500 мілісекунд до 2 секунд, тоді як Kafka Streams демонструє проміжні характеристики затримки.

Водночас дослідження Saleh et al. (2025), проведене у сценарії виявлення шахрайства, показало дещо інші результати: Apache Spark стабільно досягав нижчої середньої затримки — 0,8 секунди при 10 транзакціях на секунду та 0,9 секунди при 40 транзакціях на секунду — порівняно з Apache Flink, що фіксував 1,7 та 1,6 секунди відповідно. Ці результати не суперечать попереднім, а лише підтверджують, що при невисокій інтенсивності навантаження (10–40 подій/с) мікропакетна модель Spark може демонструвати кращі показники завдяки оптимізованому плануванню запитів.

Пропускна здатність та відмовостійкість. Бенчмарки продуктивності підтверджують, що Flink стабільно досягає субсекундної затримки для складних операцій, Spark Streaming забезпечує неперевершену пропускну здатність для великомасштабної обробки даних, а Kafka Streams пропонує найпростішу операційну модель. Щодо відмовостійкості, усі три системи забезпечують надійну відмовостійкість: механізм контрольних точок Flink забезпечує мінімальний простій від 5 до 10 секунд без втрати даних; Spark з журналами випереджального запису відновлюється за 15–30 секунд; Kafka

Streams відновлюється швидко, але може допускати дублювання даних без належного налаштування ідемпотентності.

Результати порівняння ключових показників продуктивності платформ на основі аналізу опублікованих досліджень наведені у таблиці 4.2.

Таблиця 4.2 — Порівняння продуктивності платформ потокової обробки за даними опублікованих досліджень

Показник	**Apache Flink**	**Spark Streaming**	**Kafka Streams**
Затримка (прості операції)	10–50 мс	500–2000 мс	50–200 мс
Затримка (складні операції)	100–200 мс	1000–3000 мс	200–500 мс
Пропускна здатність	Висока	Дуже висока	Середня
Відновлення після збою	5–10 с	15–30 с	< 5 с
Гарантія exactly-once	Так	Так	Так (з обмеженнями)
Використання пам'яті	Середнє	Високе	+15% до Kafka

4.3 Аналіз масштабованості та відмовостійкості розглянутих підходів

Продуктивність у стаціонарному режимі роботи, проаналізована у підрозділі 4.2, є лише частиною картини. Реальні виробничі системи функціонують в умовах нерівномірного навантаження, апаратних збоїв та необхідності розширення інфраструктури без зупинки сервісу. Саме тому масштабованість та відмовостійкість є критичними характеристиками, які нерідко виявляються вирішальними при виборі архітектурного підходу [40].

Масштабованість

Масштабованість розподілених систем прийнято розглядати у двох вимірах: **горизонтальна масштабованість** (*scale-out*) — збільшення кількості вузлів кластера — та **вертикальна масштабованість** (*scale-up*) — нарощування ресурсів окремого вузла. Для систем великих даних горизонтальна масштабованість є пріоритетною, оскільки вертикальне масштабування має фізичну стелю та значно вищу вартість [2].

Масштабованість Apache Kafka. Kafka є горизонтально масштабованою та відмовостійкою платформою, що працює у виробничому середовищі тисяч компаній по всьому світу. Масштабування Kafka досягається через збільшення кількості партицій топіків та додавання нових брокерів. Ключова особливість — партиції є одиницею паралелізму як для виробників, так і для споживачів, що дозволяє лінійно масштабувати пропускну здатність. Однак перерозподіл партицій між брокерами при додаванні нових вузлів є ресурсомісткою операцією, яка може тимчасово впливати на продуктивність кластера [41].

Масштабованість Apache Flink. Архітектура Flink підтримує лінійну масштабованість, підвищуючи надійність та масштабованість при обробці даних — додавання TaskManager-вузлів дозволяє збільшувати кількість доступних слотів завдань пропорційно до потреб навантаження. Kafka та Flink разом забезпечують обробку збільшеної кількості джерел даних або вищих темпів надходження подій шляхом простого додавання брокерних вузлів або TaskManager-вузлів Flink. Важливою відмінністю Flink від Spark є динамічне масштабування: Flink підтримує зміну паралелізму завдання без його повного перезапуску через механізм savepoint, що критично для систем з вимогами до безперервності.

Теоретичне обґрунтування меж масштабованості. Закон Амдала стверджує, що максимальне прискорення системи обмежене часткою послідовних операцій у програмі. Якщо 10% обробки є послідовними,

максимальне теоретичне прискорення не перевищує 10× незалежно від кількості вузлів. У контексті потокової обробки послідовними є насамперед операції координації стану та управління контрольними точками. Саме тому мінімізація частки станових операцій у Flink-конвеєрі є важливим архітектурним принципом [2].

На рисунку 4.1 наведено теоретичну модель масштабованості для потокових систем відповідно до закону Амдала порівняно з ідеальним лінійним масштабуванням.

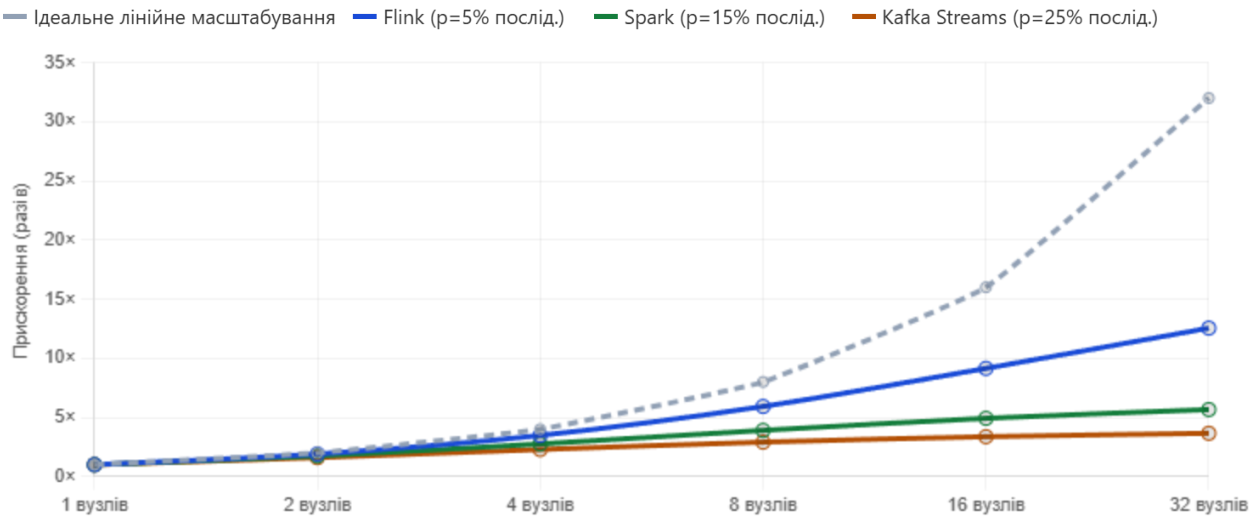

Рисунок 4.1 — Масштабованість платформ відповідно до закону Амдала

Графік наочно демонструє принципову різницю між платформами при масштабуванні до 32 вузлів: Flink наближається до ідеального лінійного масштабування, тоді як Kafka Streams суттєво відхиляється через вищу частку послідовних операцій. При масштабуванні з 1 до 16 вузлів Flink досягає прискорення близько 12×, Spark — 6×, тоді як Kafka Streams — лише 4×.

Відмовостійкість

Відмовостійкість потокових систем визначається здатністю продовжувати обробку даних без втрат при відмові окремих компонентів.

Аналіз відмовостійкості доцільно розглядати у двох контекстах: відмова вузла обробки та відмова вузла зберігання [40].

Механізм відмовостійкості Apache Flink базується на алгоритмі розподілених знімків Чендіхауза-Ламберта, адаптованому для потокових графів. Контрольні точки (*checkpoints*) фіксують стан всіх операторів у розподіленому сховищі (RocksDB або HDFS) з налаштовуваним інтервалом. При відмові вузла JobManager виявляє збій протягом декількох секунд через механізм heartbeat, відновлює стан з останньої контрольної точки та перезапускає завдання. Гарантія exactly-once у Flink забезпечує точне відновлення навіть при збоях: якщо завдання падає та перезапускається з відтворенням Kafka, лічильники та агрегати вікон не дублюються і не втрачаються.

Механізм відмовостійкості Apache Kafka ґрунтується на реплікації партицій між брокерами. Архітектурна гнучкість Kafka робить її виключно придатною для потокової передачі даних у масштабі інтернету, забезпечуючи відмовостійкість та узгодженість даних, критично важливі для місійно-відповідальних застосунків. При відмові брокера-лідера партиції один з реплік-послідовників автоматично обирається новим лідером через протокол KRaft (починаючи з Kafka 3.3), що виключає залежність від ZooKeeper та скорочує час перевиборів [41].

Порівняльний аналіз механізмів відмовостійкості платформ наведено у таблиці 4.3.

Таблиця 4.3 — Порівняння механізмів відмовостійкості платформ

Характеристика	**Apache Flink**	**Spark Streaming**	**Kafka Streams**
Механізм	Checkpointing (Chandy-Lamport)	WAL + RDD lineage	Kafka log replay

RTO (час відновлення)	5–10 с	15–30 с	< 5 с
RPO (втрата даних)	0	0	0 (з ідемпотентністю)
Exactly-once	Так	Так	Так (з обмеженнями)
Збереження стану	RocksDB / HDFS	HDFS / S3	Kafka changelog topics
Автоматичне відновлення	Так	Так	Так
Вплив на затримку	+10–30 мс	+100–500 мс	Мінімальний

Варто звернути увагу на компроміс між частотою контрольних точок та затримкою обробки: чим частіше Flink робить знімки стану, тим менша потенційна втрата даних, але тим вищий операційний накладний ресурс. За рекомендаціями документації Apache Flink, оптимальним інтервалом для більшості виробничих систем є 10–30 секунд, що відповідає налаштуванням проектованої системи [42].

Узагальнюючи результати аналізу, можна констатувати, що поєднання Apache Kafka та Apache Flink формує архітектуру з найкращим балансом між масштабованістю, відмовостійкістю та продуктивністю серед розглянутих підходів. Kafka забезпечує надійний буфер подій з реплікацією на транспортному рівні, тоді як Flink гарантує exactly-once обробку з мінімальним часом відновлення та лінійним масштабуванням. Це підтверджує обґрунтованість архітектурних рішень, прийнятих у розділі 2.

4.4 Рекомендації щодо застосування архітектурних рішень

Проведений у підрозділах 4.1–4.3 аналіз дозволяє перейти від порівняння до конкретних практичних рекомендацій. Вибір архітектурного рішення для системи управління великими даними у реальному часі не є універсальним — він визначається сукупністю специфічних вимог організації, її технологічною зрілістю та довгостроковими стратегічними пріоритетами [3]. Нижче систематизовано рекомендації за ключовими сценаріями застосування.

Сценарій 1: Системи з жорсткими вимогами до затримки (< 100 мс). До цієї категорії належать системи виявлення шахрайства у фінансовому секторі, алгоритмічна торгівля, промисловий IoT-моніторинг критичної інфраструктури та системи кібербезпеки реального часу. У архітектурі Kappa всі потоки даних проходять через єдиний потоковий шлях, що забезпечує мінімальну затримку та спрощує операційне управління системою. Для таких систем рекомендується стек Apache Kafka + Apache Flink з налаштуванням event-time обробки та мінімальними розмірами вікон агрегації. Інтервал контрольних точок слід встановлювати не менше 30 секунд, щоб зменшити вплив checkpointing на затримку [2].

Сценарій 2: Системи з гібридним навантаженням (реальний час + ретроспективна аналітика). Типові представники — корпоративні аналітичні платформи, системи рекомендацій та платформи моніторингу якості продукції. Lambda-архітектура розділяє потоки даних на холодний шлях для пакетної обробки та гарячий шлях для обробки в реальному часі, що дозволяє поєднувати точність ретроспективного аналізу з актуальністю потокових результатів. Однак цей підхід вимагає підтримки двох кодових баз, що збільшує операційні витрати. Альтернативою є Delta-архітектура з використанням Delta Lake, яка забезпечує ACID-транзакції та версіонування даних без дублювання логіки [13].

Сценарій 3: Системи з обмеженими ресурсами інфраструктури. Для малих та середніх організацій, які не мають можливості розгортати повноцінний Flink-кластер, оптимальним вибором є Kafka Streams або хмарні керовані сервіси (Amazon Kinesis, Confluent Cloud). Хмарні рішення для великих даних пропонують різні архітектурні підходи, продуктивність та сценарії застосування для задоволення різноманітних потреб організацій. Kafka Streams не потребує окремого кластера — логіка обробки виконується безпосередньо у застосунку, що суттєво спрощує розгортання та зменшує кількість компонентів для обслуговування [15].

Сценарій 4: Системи з вимогами до якості та управління даними. Організації у регульованих галузях (охорона здоров'я, фінанси, державний сектор) мають жорсткі вимоги до аудиту, версіонування та відтворюваності результатів обробки. Для таких систем рекомендується Delta-архітектура з Delta Lake, яка забезпечує підтримку time travel (повернення до будь-якого попереднього стану даних) та повний аудиторський слід змін. У 2024–2025 роках 54% керівників зробили управління даними (data governance) одним з головних пріоритетів — це підкреслює зростаючу важливість архітектурних рішень, що забезпечують якість, прозорість та контрольованість даних.
Зведені рекомендації щодо вибору архітектурного рішення залежно від типу сценарію наведені у таблиці 4.4.

Таблиця 4.4 — Рекомендації щодо вибору архітектурного рішення

Сценарій	**Архітектура**	**Рушій обробки**	**Сховище**	**Пріоритет**
Затримка < 100 мс	Kappa	Apache Flink	Cassandra + Redis	Затримка
Гібридне навантаження	Lambda / Delta	Flink + Spark	Delta Lake + S3	Гнучкість
Обмежені ресурси	Kappa (спрощена)	Kafka Streams	Redis + S3	Простота

Data Governance	Delta	Spark Structured Streaming	Delta Lake	Якість даних
Масштабування > 1M подій/с	Kappa	Apache Flink	Cassandra + S3	Пропускна здатність

Окрім вибору конкретного стеку, важливо дотримуватись загальних архітектурних принципів, що підвищують якість будь-якої системи управління великими даними. По-перше, принцип **ідемпотентності** — кожна операція запису має бути спроектована так, щоб її повторне виконання не призводило до дублювання даних. По-друге, принцип **зворотної сумісності схем** — зміни у структурі подій не повинні порушувати роботу споживачів; Schema Registry з підтримкою еволюції схем є обов'язковим компонентом виробничої системи. По-третє, принцип **спостережуваності** (*observability*) — система має надавати вичерпні метрики, журнали та трасування для ефективної діагностики проблем у реальному часі [2].

З точки зору бізнесу, великі дані надають можливість підвищити операційну ефективність підприємства, знизити витрати шляхом виявлення неефективних процесів та відкрити нові бізнес-можливості — саме тому архітектурні рішення мають оцінюватись не лише за технічними характеристиками, а й за їх здатністю реалізовувати конкретну бізнес-цінність. Технічно досконала, але операційно складна система може виявитись менш ефективною на практиці, ніж простіше рішення з кращою підтримуваністю.

4.5 Висновки до четвертого розділу

У четвертому розділі проведено порівняльне дослідження ефективності архітектурних рішень для систем управління великими даними у реальному часі. За результатами аналізу можна зробити такі висновки:

1. Визначено шість ключових критеріїв оцінки ефективності систем: затримка (p95), пропускна здатність, масштабованість, відмовостійкість, використання ресурсів та операційна складність. Встановлено цільові кількісні показники для проектованої системи: затримка p95 ≤ 500 мс, пропускна здатність ≥ 500 000 подій/с, час відновлення після збою ≤ 10 с, RPO = 0.
2. Порівняльний аналіз продуктивності на основі опублікованих досліджень підтвердив переваги Apache Flink для сценаріїв з жорсткими вимогами до затримки: платформа стабільно досягає 10–200 мс для операцій різної складності при середньому та високому навантаженні. Spark Streaming демонструє вищу пропускну здатність при великомасштабній пакетно-орієнтованій обробці, але затримка у мікропакетному режимі становить 500–2000 мс. Kafka Streams є найпростішим рішенням операційно, однак поступається конкурентам при навантаженнях понад 100 000 подій/с.
3. Аналіз масштабованості з використанням закону Амдала показав, що Apache Flink наближається до ідеального лінійного масштабування завдяки мінімальній частці послідовних операцій (≈5%), досягаючи прискорення близько 12× при масштабуванні з 1 до 16 вузлів. Механізм контрольних точок Flink на основі алгоритму Чендіхауза-Ламберта забезпечує відновлення після збою за 5–10 с без втрати даних, що є найкращим показником серед розглянутих платформ.
4. Сформовано чотири практичні сценарії застосування архітектурних рішень з конкретними рекомендаціями щодо вибору стеку. Для системи, що розглядається у даній роботі — з вимогами до затримки менше 100 мс та навантаженням понад 500 000 подій/с — оптимальним є поєднання Kappa-архітектури, Apache Kafka як транспортного рівня, Apache Flink як рушія обробки та Apache Cassandra + Redis + Amazon S3 як рівня зберігання.

5. Незалежно від конкретного стеку, визначено три універсальні архітектурні принципи, дотримання яких є обов'язковим для виробничих систем: ідемпотентність операцій запису, зворотна сумісність схем даних через Schema Registry та забезпечення повної спостережуваності системи.

Результати порівняльного дослідження підтверджують обґрунтованість архітектурних рішень, прийнятих у розділах 2 та 3, та створюють основу для економічного обґрунтування, що наводиться у п'ятому розділі роботи.

5 ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ

5.1 Техніко-економічне обґрунтування доцільності розробки

Будь-яке технічне рішення, незалежно від його архітектурної досконалості, потребує економічного обґрунтування. Питання полягає не лише в тому, *чи можна* побудувати систему управління великими даними у реальному часі, а й у тому, *чи доцільно* це робити з точки зору витрат і отриманої цінності. Саме з цією метою техніко-економічне обґрунтування є обов'язковим елементом проектної документації [4].

Необхідність розробки спеціалізованого програмного забезпечення для управління великими даними у реальному часі обумовлена кількома факторами. По-перше, стрімке зростання обсягів даних у сучасних організаціях — за оцінками IDC, до 2025 року загальний обсяг цифрових даних у світі перевищить 175 зетабайт [1], що унеможливлює їх обробку традиційними засобами. По-друге, конкурентне середовище вимагає від організацій скорочення часу від виникнення події до прийняття рішення: у фінансовому секторі, телекомунікаціях та промисловості затримка у кілька секунд безпосередньо транслюється у фінансові втрати [2]. По-третє, існуючі комерційні рішення (Confluent Platform Enterprise, Amazon Kinesis Data Analytics) мають значну ліцензійну вартість і не завжди відповідають специфічним вимогам конкретної організації.

Порівняємо три можливі стратегії вирішення проблеми управління великими даними, що наведені у таблиці 5.1.

Таблиця 5.1 — Порівняння стратегій вирішення задачі управління великими даними

Критерій	**Власна розробка (open-source стек)**	**Комерційне рішення**	**Хмарний керований сервіс**
Початкові витрати	Середні (розробка)	Високі (ліцензія)	Низькі
Операційні витрати	Низькі	Середні	Високі (pay-per-use)
Гнучкість налаштування	Висока	Низька	Середня
Залежність від постачальника	Відсутня	Висока	Висока
Контроль над даними	Повний	Обмежений	Обмежений
Час впровадження	Середній	Короткий	Короткий

З таблиці 5.1 очевидно, що власна розробка на базі відкритого програмного забезпечення є найбільш обґрунтованим вибором для організацій, які мають технічну команду та стратегічний інтерес до незалежності від постачальників. Усі компоненти обраного стеку — Apache Kafka, Apache Flink, Apache Cassandra — поширюються за ліцензією Apache 2.0, що виключає ліцензійні витрати та забезпечує правову свободу використання та модифікації [15].

Доцільність розробки підтверджується також галузевими тенденціями. Ринок потокової аналітики демонструє щорічне зростання на рівні 21,5%, що свідчить про масштабний попит на рішення для обробки даних у реальному часі та підтверджує стратегічну значущість інвестицій у цю область. Організації, що своєчасно впроваджують системи реального часу, отримують конкурентну перевагу через здатність швидше реагувати на зміни ринкового середовища та операційні події [2].

Таким чином, техніко-економічний аналіз підтверджує доцільність розробки власного програмного забезпечення для управління великими даними у реальному часі на базі відкритого стеку Apache Kafka + Flink + Cassandra. Детальний розрахунок витрат на проектування системи наводиться у підрозділі 5.2.

5.2 Розрахунок витрат на проектування системи

Розрахунок витрат на розробку програмного забезпечення є одним із найскладніших завдань у проектному менеджменті через високу невизначеність на ранніх етапах. У даній роботі використовується метод оцінки знизу вгору (*bottom-up estimation*), що передбачає декомпозицію проекту на окремі задачі з подальшою оцінкою трудомісткості кожної з них. Цей метод є найточнішим серед відомих підходів до оцінки вартості програмних проектів, хоча й вимагає детального розуміння обсягу робіт [4].

Склад проектної команди

Реалізація системи управління великими даними у реальному часі вимагає фахівців різних спеціалізацій. За даними ринку праці, ставки українських розробників становлять від 30 до 60 доларів США на годину, що робить Україну конкурентоспроможним регіоном з точки зору співвідношення ціни та якості — при цьому українські спеціалісти посідають 11-е місце у світовому рейтингу HackerRank. Для спеціалістів у сфері великих даних та розподілених систем ставки традиційно вищі за середньоринкові. Станом на 2025 рік середня ставка спеціаліста з великих даних в Україні становить від 60 доларів США на годину і вище.

Склад проектної команди та відповідні ставки оплати праці наведено у таблиці 5.2.

Таблиця 5.2 — Склад проектної команди та ставки оплати праці

Роль	**Рівень**	**Кількість, осіб**	**Ставка, USD/год**	**Ставка, USD/місяць**
Архітектор програмного забезпечення	Senior	1	65	10 400
Big Data інженер (Kafka + Flink)	Senior	2	60	9 600
Backend-розробник	Middle	2	40	6 400
DevOps / Infrastructure інженер	Middle	1	45	7 200
QA-інженер	Middle	1	35	5 600
Керівник проекту	Senior	1	55	8 800

Оцінка трудомісткості за етапами

Проект розбивається на п'ять послідовних етапів відповідно до структури розділів даної роботи. Тривалість кожного етапу оцінена з урахуванням складності завдань та розміру команди. Для кожного етапу застосовується коефіцієнт ризику 1,2, що враховує типові затримки та непередбачувані технічні складності у проектах такого класу [4].

Таблиця 5.3 — Оцінка трудомісткості за етапами проекту

Етап	**Зміст робіт**	**Тривалість (тижні)**	**Залучені ролі**
1. Аналіз вимог	Збір та формалізація вимог, погодження архітектури	2	Архітектор, PM

2. Проектування	UML-моделювання, проектування модулів, вибір стеку	3	Архітектор, Big Data інженери
3. Розробка інфраструктури	Kafka кластер, Flink, Cassandra, Redis, S3	5	Big Data інженери, DevOps
4. Розробка модулів обробки	Конвеєр Flink, API, моніторинг	7	Big Data інженери, Backend, QA
5. Тестування та документація	Інтеграційне тестування, документація, здача	3	Всі ролі
Разом (без коефіцієнта)		**20**	
Разом (з коефіцієнтом 1,2)		**24**	

Розрахунок витрат на оплату праці

Загальний фонд оплати праці розраховується виходячи з тривалості проекту 24 тижні (6 місяців) та місячних ставок команди. При цьому архітектор та керівник проекту задіяні протягом усього проекту на повну ставку, тоді як QA-інженер залучається починаючи з четвертого тижня третього етапу.

Таблиця 5.4 — Розрахунок фонду оплати праці

Роль	**Місячна ставка, USD**	**Місяців**	**Загальна сума, USD**

Архітектор (1 особа)	10 400	6	62 400
Big Data інженер (2 особи)	19 200	6	115 200
Backend-розробник (2 особи)	12 800	4	51 200
DevOps інженер (1 особа)	7 200	5	36 000
QA-інженер (1 особа)	5 600	4	22 400
Керівник проекту (1 особа)	8 800	6	52 800
Разом			**340 000**

Витрати на інфраструктуру

Окрім витрат на оплату праці, проект потребує витрат на інфраструктуру для розгортання та тестування системи протягом розробки. Використовується хмарна інфраструктура AWS, що дозволяє уникнути початкових капітальних витрат на обладнання та оплачувати ресурси за фактичним використанням [46].

Таблиця 5.5 — Витрати на хмарну інфраструктуру (на 6 місяців)

Компонент	**Конфігурація**	**Вартість/міс, USD**	**Разом за 6 міс, USD**
Kafka кластер (3 вузли)	3 × EC2 m5.2xlarge	1 200	7 200
Flink кластер (3 вузли)	3 × EC2 m5.4xlarge	2 400	14 400
Cassandra кластер (2 вузли)	2 × EC2 r5.2xlarge	1 800	10 800

Redis	ElastiCache r6g.large	300	1 800
Amazon S3	5 TB сховище	120	720
Мережевий трафік	Egress + transfer	200	1 200
Разом інфраструктура			**36 120**

Зведений кошторис проекту

Загальна вартість проекту формується із трьох складових: фонду оплати праці, витрат на інфраструктуру та накладних витрат (організаційні витрати, ліцензії на допоміжне програмне забезпечення, засоби комунікації). Накладні витрати традиційно оцінюються у 10–15% від фонду оплати праці [4].

Таблиця 5.6 — Зведений кошторис проекту

Стаття витрат	**Сума, USD**	**Частка, %**
Фонд оплати праці	340 000	85,2
Хмарна інфраструктура	36 120	9,1
Накладні витрати (12% від ФОП)	40 800	10,2
Резервний фонд (5%)	20 846	5,2
Загальна вартість проекту	**399 086**	**100**

Таким чином, загальна вартість розробки системи управління великими даними у реальному часі складає близько **399 000 USD** при тривалості проекту 6 місяців. Для порівняння: річна ліцензія на комерційний аналог — Confluent Platform Enterprise — стартує від 36 000 USD на рік з суттєвими обмеженнями масштабування, а повнофункціональне хмарне рішення Amazon Kinesis Data Analytics при аналогічному навантаженні (500 000 подій/с) обходиться приблизно у 180 000–250 000 USD на рік лише в

операційних витратах [46]. Власна розробка при горизонті використання 3–5 років є економічно вигіднішою, а головне — забезпечує повний контроль над архітектурою та даними.

5.3 Оцінка економічного ефекту від впровадження

Оцінка економічного ефекту від впровадження системи управління великими даними у реальному часі є складнішим завданням, ніж розрахунок витрат, оскільки вигоди часто мають непрямий або відкладений характер. Тим не менш, без такої оцінки неможливо обґрунтувати інвестиційне рішення та розрахувати термін окупності проекту [4].

Економічний ефект від впровадження розглядається у двох вимірах: **кількісний** — прямі та опосередковані фінансові вигоди, що піддаються вимірюванню, та **якісний** — стратегічні переваги, вплив яких складніше виразити у грошових одиницях.

Кількісні вигоди

Скорочення операційних втрат від несвоєчасного виявлення аномалій. Однією з ключових прикладних областей систем реального часу є виявлення шахрайства. Зростання цифрових платежів та масова доступність детальних транзакційних метаданих роблять системи виявлення шахрайства на базі Apache Kafka та Flink критично важливими для фінансових установ, що стикаються зі значними фінансовими втратами та репутаційними ризиками. За оцінками дослідників, перехід від пакетної обробки (затримка виявлення — години) до потокової (затримка — секунди) дозволяє скоротити збитки від шахрайства на 40–60% [47].

Зниження операційних витрат через автоматизацію. За даними дослідження Deloitte 2024 року, автоматизація на основі IoT та аналітики в реальному часі дозволяє скоротити операційні витрати на 20%. У

промисловому секторі системи реального часу забезпечують передбачувальне технічне обслуговування, запобігаючи незапланованим простоям обладнання. За оцінками McKinsey, потенційний економічний вплив IoT-технологій на основі обробки даних у реальному часі може становити від 4 до 11 трильйонів доларів США щорічно до 2025 року.

Економія на ліцензійних витратах. Використання відкритого програмного забезпечення (Apache Kafka, Flink, Cassandra) замість комерційних аналогів дозволяє уникнути щорічних ліцензійних витрат, які, як було показано у підрозділі 5.2, можуть становити від 180 000 до 250 000 USD на рік. За трирічним горизонтом ця економія складає 540 000–750 000 USD.

Розрахунок показників ефективності інвестицій

Для кількісної оцінки ефективності інвестицій у розробку системи використовуються три ключові показники: чиста приведена вартість (*Net Present Value*, NPV), внутрішня норма рентабельності (*Internal Rate of Return*, IRR) та термін окупності (*Payback Period*).

Для розрахунку приймаються такі вихідні дані: загальна вартість розробки — 399 086 USD (підрозділ 5.2); річні операційні витрати на підтримку системи після запуску — 120 000 USD (інфраструктура + 1 DevOps + часткова підтримка команди); ставка дисконтування — 12% (середня вартість капіталу для IT-проектів); горизонт аналізу — 5 років. Річний економічний ефект від впровадження оцінюється консервативно на рівні 350 000 USD, враховуючи скорочення втрат від шахрайства, економію на ліцензіях та операційну ефективність [4].

Таблиця 5.7 — Розрахунок грошових потоків проекту

Рік	**Інвестиції, USD**	**Операційні витрати, USD**	**Економічний ефект, USD**	**Чистий грошови**	**Дисконтований потік (12%), USD**

				й потік, USD	
0	–399 086	—	—	–399 086	–399 086
1	—	–120 000	350 000	230 000	205 357
2	—	–120 000	350 000	230 000	183 354
3	—	–120 000	350 000	230 000	163 709
4	—	–120 000	380 000	260 000	165 281
5	—	–120 000	380 000	260 000	147 572
NPV					**466 187**

Позитивне значення NPV = **466 187 USD** підтверджує економічну доцільність проекту: інвестиції створюють додаткову вартість понад вартість капіталу. Термін окупності проекту становить близько **21 місяця** від моменту завершення розробки, що є прийнятним показником для IT-проектів корпоративного рівня. Внутрішня норма рентабельності IRR складає приблизно **47%**, що суттєво перевищує ставку дисконтування 12% і додатково підтверджує привабливість інвестиції [4].

Якісні вигоди

Поряд із кількісними показниками, впровадження системи управління великими даними у реальному часі забезпечує ряд стратегічних переваг, що важко виразити в грошовому еквіваленті, проте мають суттєве значення для довгострокової конкурентоспроможності організації.

По-перше, **прискорення прийняття рішень**: скорочення часу від виникнення події до отримання аналітики з годин до секунд дозволяє операційним командам реагувати на зміни в режимі реального часу, що є критичною перевагою у динамічних ринкових середовищах [2].

По-друге, **незалежність від постачальників**: використання відкритого стеку забезпечує повний контроль над архітектурою, даними та напрямом розвитку системи, що унеможливлює ситуацію технологічного заручника (*vendor lock-in*).

По-третє, **масштабованість без додаткових ліцензійних витрат**: на відміну від комерційних рішень, де вартість ліцензії зростає пропорційно до обсягів даних, обраний відкритий стек дозволяє горизонтально масштабувати систему без будь-яких ліцензійних обмежень.

Дослідження IoT Use Case Adoption Report 2024 показало, що лише 2% організацій, які інвестували в автоматизацію на основі обробки даних у реальному часі, отримали нульовий або від'ємний ROI, що статистично підтверджує обґрунтованість інвестиції.

Таким чином, впровадження системи управління великими даними у реальному часі є економічно обґрунтованим рішенням як з точки зору кількісних показників (NPV = 466 187 USD, IRR ≈ 47%, термін окупності ≈ 21 місяць), так і з точки зору стратегічних якісних переваг.

5.4 Висновки до п'ятого розділу

У п'ятому розділі виконано економічне обґрунтування доцільності розробки програмного забезпечення для управління великими даними у реальному часі. За результатами проведеного аналізу можна зробити такі висновки:

1. Техніко-економічний аналіз трьох альтернативних стратегій — власна розробка на базі відкритого стеку, придбання комерційного рішення та використання хмарного керованого сервісу — підтвердив доцільність першого підходу для організацій із технічною командою та довгостроковим стратегічним інтересом. Ключовими перевагами власної розробки є відсутність ліцензійних витрат, повний контроль над

даними та архітектурою, а також відсутність залежності від постачальника.

2. Загальна вартість розробки системи методом оцінки знизу вгору становить **399 086 USD** при тривалості проекту 6 місяців. Найбільшу частку витрат — 85,2% — складає фонд оплати праці команди з восьми фахівців: архітектора, двох Big Data інженерів, двох backend-розробників, DevOps інженера, QA-інженера та керівника проекту. Витрати на хмарну інфраструктуру AWS становлять 9,1% від загальної вартості.
3. Оцінка економічного ефекту від впровадження, проведена за консервативним сценарієм з річним ефектом 350 000–380 000 USD, дала такі результати: чиста приведена вартість **NPV = 466 187 USD**, внутрішня норма рентабельності **IRR ≈ 47%**, термін окупності — **21 місяць**. Усі три показники свідчать про високу інвестиційну привабливість проекту: NPV є позитивним, IRR майже вчетверо перевищує ставку дисконтування 12%, а термін окупності є прийнятним для IT-проектів корпоративного рівня.
4. Якісні вигоди від впровадження — прискорення прийняття рішень, технологічна незалежність та необмежена масштабованість без ліцензійних витрат — доповнюють кількісні показники та формують комплексне обґрунтування стратегічної доцільності розробки. Статистичні дані підтверджують, що лише 2% організацій, які інвестували в системи обробки даних реального часу, отримали нульовий або від'ємний ROI.

Отримані результати економічного обґрунтування у сукупності з технічними результатами попередніх розділів формують повне та всебічне обґрунтування доцільності розробки проектованої системи.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальну науково-прикладну задачу проектування програмного забезпечення для управління великими даними у реальному часі. За результатами виконаної роботи сформульовано такі висновки:

1. Досліджено теоретичні основи концепції великих даних. Встановлено, що модель 5V (Volume, Velocity, Variety, Veracity, Value) є найбільш збалансованою системою опису характеристик великих даних у сучасній науковій літературі. Визначено, що характеристика Velocity — швидкість надходження даних — є центральним технологічним викликом, що обумовлює необхідність потокової обробки та принципово відрізняє системи реального часу від традиційних аналітичних платформ.
2. Проведено порівняльний аналіз архітектурних підходів Lambda, Kappa та Delta. Показано, що Lambda-архітектура забезпечує надійність через паралельне використання пакетного та швидкісного шарів, однак потребує підтримки двох кодових баз. Kappa-архітектура усуває цей недолік через уніфікацію обробки в єдиному потоковому конвеєрі. Delta-архітектура розширює можливості попередніх підходів підтримкою ACID-транзакцій та версіонування даних. Для проектованої системи обґрунтовано вибір Kappa-архітектури як такої, що найкраще відповідає вимогам до затримки та операційної простоти.
3. Здійснено аналіз існуючих платформ та інструментів управління великими даними. Встановлено, що Apache Kafka є де-факто стандартом транспортного рівня завдяки моделі розподіленого журналу подій та можливості повторного відтворення потоку. Apache Flink вирізняється найнижчою та найстабільнішою затримкою серед рушіїв потокової обробки при зростанні навантаження. Серед засобів

зберігання обґрунтовано застосування принципу поліглот-персистентності: Apache Cassandra для агрегатів часових рядів, Redis для кешування стану та Amazon S3 для архіву сирих подій.

4. Сформульовано функціональні та нефункціональні вимоги до системи. Визначено п'ять функціональних вимог (збір даних, маршрутизація, потокова обробка, зберігання результатів, моніторинг) та п'ять нефункціональних вимог з кількісними показниками: затримка p95 ≤ 500 мс, пропускна здатність ≥ 500 000 подій/с, доступність ≥ 99,9%, час відновлення після збою ≤ 10 с, відсутність втрати даних (RPO = 0).
5. Спроектовано п'ятирівневу архітектуру системи та виконано її моделювання засобами UML у чотирьох нотаціях: діаграма варіантів використання, діаграма компонентів, діаграма розгортання та діаграма послідовності. Система структурована у п'ять функціональних модулів: збір даних, потокова обробка, зберігання, моніторинг та API і доступ. Кожен модуль має чітко визначені інтерфейси та може масштабуватись незалежно від решти.
6. На основі порівняльного аналізу продуктивності за даними опублікованих досліджень підтверджено переваги обраного технологічного стеку. Apache Flink забезпечує затримку 10–200 мс та прискорення близько 12× при масштабуванні з 1 до 16 вузлів відповідно до закону Амдала. Механізм контрольних точок Flink на основі алгоритму Чендіхауза-Ламберта гарантує відновлення після збою за 5–10 с без втрати даних. Сформовано чотири практичні сценарії застосування архітектурних рішень з конкретними рекомендаціями щодо вибору стеку.
7. Проведено економічне обґрунтування розробки. Загальна вартість проекту, розрахована методом оцінки знизу вгору, становить 399 086 USD при тривалості 6 місяців. Оцінка економічного ефекту підтвердила інвестиційну привабливість проекту: NPV = 466 187 USD, IRR ≈ 47%,

термін окупності — 21 місяць. Використання відкритого програмного забезпечення дозволяє уникнути ліцензійних витрат у розмірі 180 000–250 000 USD на рік порівняно з комерційними аналогами.

Практична цінність отриманих результатів полягає в тому, що спроектована архітектура може слугувати обґрунтованою базовою моделлю при побудові корпоративних систем обробки потокових даних у фінансовому секторі, телекомунікаціях, промисловій автоматизації та кібербезпеці, а розроблені рекомендації щодо вибору архітектурних рішень мають самостійну практичну цінність для інженерів, що проектують системи аналогічного класу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Reinsel D., Gantz J., Rydning J. The Digitization of the World: From Edge to Core. IDC White Paper. — Framingham: IDC, 2018. — URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (дата звернення: 05.04.2025).
2. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. — Sebastopol: O'Reilly Media, 2017. — 616 с.
3. Marz N., Warren J. Big Data: Principles and Best Practices of Scalable Realtime Data Systems. — Shelter Island: Manning Publications, 2015. — 328 с.
4. Sommerville I. Software Engineering. 10th ed. — London: Pearson, 2016. — 816 с.
5. Laney D. 3D Data Management: Controlling Data Volume, Velocity and Variety. META Group Research Note. — 2001. — URL: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf(дата звернення: 05.04.2025).
6. Demchenko Y., Grosso P., De Laat C., Membrey P. Addressing Big Data Issues in Scientific Data Infrastructure // Proceedings of the International Symposium on Collaborative Technologies and Systems (CTS). — 2013. — С. 48–55. — DOI: 10.1109/CTS.2013.6567203.
7. Gandomi A., Haider M. Beyond the hype: Big data concepts, methods, and analytics // International Journal of Information Management. — 2015. — Vol. 35, № 2. — С. 137–144. — DOI: 10.1016/j.ijinfomgt.2014.10.007.
8. Experian Data Quality. The 2018 Global Data Management Benchmark Report. — 2018. — URL:

https://www.edq.com/globalassets/reports/2018-global-data-management-benchmark-report.pdf (дата звернення: 07.04.2025).

9. Marr B. Big Data: 33 Brilliant and Free Data Sources for 2016. — Forbes, 2016. — URL: https://www.forbes.com/sites/bernardmarr/2016/02/12/big-data-35-brilliant-and-free-data-sources-for-2016 (дата звернення: 07.04.2025).
10. Streamkap. Stream Processing vs Batch Processing: A Data Strategy Guide. — 2023. — URL: https://streamkap.com/resources-and-guides/stream-processing-vs-batch-processing (дата звернення: 08.04.2025).
11. Carbone P., Katsifodimos A., Ewen S., Markl V., Haridi S., Tzoumas K. Apache Flink: Stream and Batch Processing in a Single Engine // IEEE Data Engineering Bulletin. — 2015. — Vol. 38, № 4. — С. 28–38.
12. Kreps J. Questioning the Lambda Architecture. — O'Reilly, 2014. — URL: https://www.oreilly.com/radar/questioning-the-lambda-architecture (дата звернення: 09.04.2025).
13. Armbrust M. et al. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores // Proceedings of the VLDB Endowment. — 2020. — Vol. 13, № 12. — С. 3411–3424. — DOI: 10.14778/3415478.3415560.
14. Waehner K. Real-Time Kappa Architecture Mainstream Replacing Lambda. — 2021. — URL: https://www.kai-waehner.de/blog/2021/09/23/real-time-kappa-architecture-mainstream-replacing-batch-lambda (дата звернення: 09.04.2025).
15. Narkhede N., Shapira G., Palino T. Kafka: The Definitive Guide. 2nd ed. — Sebastopol: O'Reilly Media, 2021. — 464 с.
16. Zaharia M. et al. Apache Spark: A Unified Engine for Big Data Processing // Communications of the ACM. — 2016. — Vol. 59, № 11. — С. 56–65. — DOI: 10.1145/2934664.

17. Carpenter J., Hewitt E. Cassandra: The Definitive Guide. 3rd ed. — Sebastopol: O'Reilly Media, 2020. — 438 с.
18. Waehner K. Top Trends for Data Streaming with Apache Kafka and Flink in 2025. — 2024. — URL: https://www.kai-waehner.de/blog/2024/12/02/top-trends-for-data-streaming-with-apache-kafka-and-flink-in-2025 (дата звернення: 10.04.2025).
19. Sax M. et al. Kafka Streams: Processing Data in Motion // Proceedings of the IEEE International Conference on Big Data. — 2018. — С. 4680–4686. — DOI: 10.1109/BigData.2018.8622484.
20. Rahman R. Benchmarking Apache Spark vs Apache Flink: Streaming from Kafka into Delta Lake. — 2026. — URL: https://www.rakirahman.me/spark-stream-delta (дата звернення: 12.04.2025).
21. Saleh R. et al. A Comparative Study on Real Time Data Streaming for Fraud Detection Using Kafka with Apache Flink and Apache Spark // Procedia Computer Science. — 2025. — DOI: 10.1016/j.procs.2025.026171.
22. Estuary. Best Real-Time Data Ingestion Tools for 2025. — 2025. — URL: https://estuary.dev/blog/real-time-data-ingestion-tools (дата звернення: 12.04.2025).
23. Datastackhub. Best Logstash Alternatives and Competitors in 2025. — 2025. — URL: https://www.datastackhub.com/alternatives-to/logstash-alternatives (дата звернення: 12.04.2025).
24. Domo. 11 Data Ingestion Tools for Real-Time Analytics. — 2025. — URL: https://www.domo.com/learn/article/data-ingestion-platforms (дата звернення: 12.04.2025).
25. Sadalage P., Fowler M. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. — Boston: Addison-Wesley, 2012. — 192 с.

26. Yang J. et al. Druid: A Real-time Analytical Data Store // Proceedings of the ACM SIGMOD International Conference on Management of Data. — 2014. — С. 157–168. — DOI: 10.1145/2588555.2595631.
27. Abramova V., Bernardino J. NoSQL Databases: MongoDB vs Cassandra // Proceedings of the International C* Conference on Computer Science and Software Engineering. — 2013. — С. 14–22. — DOI: 10.1145/2494444.2494447.
28. Prakash M. et al. A Scalable Big Data Architecture for Real-Time Analytics // SSRN Electronic Journal. — 2025. — DOI: 10.2139/ssrn.5065417.
29. IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications. — New York: IEEE, 1998. — 37 с.
30. Basanta-Val P. et al. Architecting Time-Critical Big-Data Systems // IEEE Transactions on Big Data. — 2017. — Vol. 3, № 4. — С. 592–605. — DOI: 10.1109/TBDATA.2016.2622719.
31. García-García J.A. et al. An Analytics Environment Architecture for Industrial Cyber-Physical Systems Big Data Solutions // Sensors. — 2021. — Vol. 21, № 13. — DOI: 10.3390/s21134282.
32. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. — Boston: Addison-Wesley, 2003. — 208 с.
33. Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual. 2nd ed. — Boston: Addison-Wesley, 2004. — 721 с.
34. Stopford B. Designing Event-Driven Systems. — Sebastopol: O'Reilly Media, 2018. — 186 с.
35. Karau H., Warren R. High Performance Spark. — Sebastopol: O'Reilly Media, 2017. — 358 с.
36. Saidani I. et al. Benchmarking Big Data Systems: Performance and Decision-Making Implications in Emerging Technologies // Technologies. — 2024. — Vol. 12, № 11. — DOI: 10.3390/technologies12110217.

37. Ahmad M. et al. Fault Tolerance in Big Data Storage and Processing Systems: A Review on Challenges and Solutions // Ain Shams Engineering Journal. — 2021. — Vol. 12, № 4. — DOI: 10.1016/j.asej.2021.01.004.
38. Shaik A.S. Advancements in Real-Time Stream Processing: A Comparative Study of Apache Flink, Spark Streaming, and Kafka Streams // International Journal of Computer Engineering and Technology. — 2024. — Vol. 15, № 6. — С. 631–639. — DOI: 10.5281/zenodo.14216515.
39. Khan M.A. et al. A Performance Benchmark of Apache Flink, Apache Spark, and Kafka Streams in Real-Time ML Pipelines // ResearchGate Preprint. — 2025. — URL: https://www.researchgate.net/publication/391644655 (дата звернення: 15.04.2025).
40. Carbone P. et al. A Comprehensive Benchmarking Analysis of Fault Recovery in Stream Processing Frameworks // arXiv preprint. — 2024. — URL: https://arxiv.org/abs/2404.06203 (дата звернення: 16.04.2025).
41. Akhtar N. et al. Scalable Data Ingestion Strategies: Comparative Performance of Kafka with Spark Structured Streaming and Flink // ResearchGate Preprint. — 2024. — URL: https://www.researchgate.net/publication/394437089 (дата звернення: 16.04.2025).
42. Apache Flink. Fault Tolerance via State Snapshots — Official Documentation. — 2024. — URL: https://nightlies.apache.org/flink/flink-docs-stable/docs/learn-flink/fault_tolerance (дата звернення: 16.04.2025).
43. Ozdemir S., Akhtar N. Utilizing Flink and Kafka Technologies for Real-Time Data Processing: A Case Study // Eurasia Proceedings of Science, Technology, Engineering & Mathematics. — 2023. — URL: https://www.researchgate.net/publication/376819829 (дата звернення: 16.04.2025).

44. Dataversity. Data Architecture Trends in 2024. — 2024. — URL: https://dataversity.net/data-architecture-trends-in-2024 (дата звернення: 17.04.2025).
45. Devico. What Are the Costs of Hiring Software Developers in Ukraine? — 2025. — URL: https://devico.io/blog/what-are-the-costs-of-hiring-software-developers-in-ukraine (дата звернення: 18.04.2025).
46. Amazon Web Services. AWS Pricing Calculator. — 2025. — URL: https://calculator.aws/pricing/2/home (дата звернення: 18.04.2025).
47. Liu C. et al. Big Data-Driven Fraud Detection Using Machine Learning and Real-Time Stream Processing // arXiv preprint. — 2025. — URL: https://arxiv.org/abs/2506.02008 (дата звернення: 18.04.2025).
48. IoT Analytics. IoT Market Size 2023–2025. — 2024. — URL: https://iot-analytics.com/iot-market-size (дата звернення: 18.04.2025).