Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота
за освітнім ступенем «бакалавр»
на тему:

ПРОГРАМНА СИСТЕМА ЗАБЕЗПЕЧЕННЯ ІНЖЕНЕРІЇ ДАНИХ ДЛЯ МАШИННОГО НАВЧАННЯ. ІНФРАСТРУКТУРА ДЛЯ ЗБОРУ, НАКОПИЧЕННЯ, КЕРУВАННЯ ТА ТРАНСФОРМАЦІЇ ДАНИХ

Виконав:
здобувач вищої освіти 4 курсу
спеціальності 121 Інженерія програмного забезпечення
заочної форми навчання
Герасимчук Марк Романович

Науковий керівник:
к.т.н., доцент Сяський В. А.

Рівне – 2026

ЗМІСТ

ВСТУП

Актуальність теми. Протягом останнього десятиліття центр ваги у прикладному машинному навчанні змістився з конструювання моделей на роботу з даними. У даноцентричній (*data-centric*) парадигмі якість, повнота та узгодженість даних впливають на кінцевий результат сильніше, ніж вибір конкретної архітектури моделі [5]. Як наслідок, навколо будь-якої моделі машинного навчання має існувати інженерна інфраструктура, що доводить сирі, різнорідні дані, які надходять нерівномірно, до впорядкованого, перевіреного й відтворюваного стану, придатного для навчання. Побудова такої інфраструктури — для збирання, накопичення, керування та перетворення даних — є самостійною інженерною задачею, актуальність якої зростає разом із поширенням систем, що ухвалюють рішення на основі даних. Окремо актуальним є застосування цього підходу до неструктурованої командної комунікації, яка наразі слабко охоплена наявними платформами інженерної аналітики. Викладене зумовлює актуальність теми кваліфікаційної роботи.

Метою кваліфікаційної роботи є проєктування та реалізація програмної системи інфраструктури інженерії даних для машинного навчання, що забезпечує наскрізний цикл збирання, накопичення, керування та перетворення даних із різнорідних неструктурованих джерел до стану, придатного для машинного навчання.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. проаналізувати предметну область інженерії даних і наявні рішення (інфраструктурні інструменти та платформи інженерної аналітики), виявити незаповнену нішу;
2. сформулювати вимоги та спроєктувати архітектуру системи (моделі даних, діаграми компонентів, потоків даних і взаємодії);
3. реалізувати шар збирання даних на основі розширюваних адаптерів-джерел зі стійкістю до збоїв;

4. реалізувати дворівневе накопичення з контролем часу життя записів і усуненням дублікатів;
5. реалізувати шар керування: конфігурування в базі даних, перевірку за описом-схемою, версіонування результатів і захищене зберігання облікових даних;
6. реалізувати конвеєр перетворення за медальйонною моделлю із залученням мовної моделі;
7. експериментально перевірити придатність отриманих даних для машинного навчання та провести тестування системи.

Об'єктом дослідження є процеси інженерії даних у конвеєрах машинного навчання.

Предметом дослідження є методи й програмні засоби збирання, накопичення, керування та перетворення даних у досліджуваній програмній системі.

Методи дослідження. Для розв'язання поставлених завдань використано такі методи: аналіз і порівняння наявних рішень — для дослідження предметної області; об'єктно-орієнтоване та компонентне проєктування, методи моделювання даних (моделювання «сутність — зв'язок», уніфіковану мову моделювання) — для побудови архітектури системи; методи розробки програмного забезпечення на платформі Node.js (мовою TypeScript) — для реалізації; методи машинного навчання (багатокласову логістичну регресію) та статистичного оцінювання якості класифікації — для експериментальної перевірки одержаних результатів.

Практичне значення одержаних результатів полягає в тому, що розроблено працездатну програмну систему, яка автоматизує збирання неструктурованої командної комунікації з кількох джерел та її перетворення у структуровані звіти з показниками діяльності. Систему побудовано на сучасному технологічному стеку й розгортають єдиними засобами контейнеризації. Закладені архітектурні рішення (розширювані адаптери, конфігурація як керований стан, перевірка за описом-схемою, медальйонне перетворення) придатні для повторного використання в інших задачах інженерії даних.

Структура та обсяг роботи. Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і додатків. У першому розділі проаналізовано предметну область і наявні рішення. У другому сформульовано вимоги та спроєктовано архітектуру системи. У третьому описано програмну реалізацію всіх чотирьох процесів. У четвертому наведено методику тестування та результати експериментальної перевірки придатності даних для машинного навчання. У додатках наведено фрагменти програмного коду системи та лістинг експерименту з машинного навчання.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1. Актуальність інженерії даних для машинного навчання

Протягом останнього десятиліття центр ваги у прикладному машинному навчанні змістився з конструювання моделей на роботу з даними. Якщо у класичній постановці дослідник фіксував набір даних і шукав найкращий алгоритм (так званий *модельноцентричний* підхід), то сучасна інженерна практика виходить із того, що якість, повнота та узгодженість даних впливають на кінцевий результат сильніше, ніж вибір конкретної архітектури моделі. Цей погляд дістав назву *даноцентричного* (*data-centric*) і означає, що систематичне покращення даних — їх збирання, очищення, упорядкування, версіонування та перетворення — стає окремою інженерною дисципліною [5].

Звідси випливає прямий наслідок для програмної інженерії: навколо моделі має існувати інфраструктура, яка перетворює сирі, різнорідні дані, що надходять нерівномірно, на впорядкований, перевірений і відтворюваний набір ознак (*features*), придатний для навчання. Саме така інфраструктура є предметом цієї роботи. Вона охоплює чотири процеси, винесені в назву теми: збирання, накопичення, керування та перетворення даних.

Слід розрізняти інфраструктуру даних і власне модель машинного навчання — це різні рівні відповідальності. Робота зосереджена на першому: на конвеєрі, що доводить дані до придатного для навчання стану. Це відповідає усталеному розумінню інженерії даних (*data engineering*) як дисципліни, яка відрізняється від науки про дані (*data science*) поділом праці: інженер даних будує й експлуатує конвеєри, а науковець даних споживає їхній результат [1]. За визначенням, інженерія даних — це розроблення, упровадження та підтримка систем і процесів, які приймають сирі дані та виробляють якісну, узгоджену інформацію для подальшого використання, зокрема для аналізу й машинного навчання [1].

Практична вага цього поділу зумовлена тим, що в реальних системах витрати на підготовку даних суттєво перевищують витрати на побудову самих моделей, а

більшість помилок прогнозування пояснюється не недосконалістю алгоритму, а вадами вхідних даних — пропусками, неузгодженістю одиниць виміру, дублікатами та зміщенням розподілу з часом. Тому інфраструктура даних розглядається не як допоміжний, а як визначальний складник системи машинного навчання, від якості якого безпосередньо залежить результат. Це формує вимогу до інфраструктури бути не лише функціональною, а й спостережуваною та керованою, що детальніше розглянуто в підрозділі 1.6.

1.2. Життєвий цикл даних і чотири процеси предметної області

Життєвий цикл даних у системі машинного навчання прийнято описувати як послідовність стадій: породження, збирання (*ingestion*), зберігання, перетворення та подача споживачеві [1]. Питання надійного, масштабованого зберігання й опрацювання великих обсягів даних докладно розглянуто в роботі [2]. Чотири процеси теми відображаються на цей цикл так:

1. Збирання — підключення до джерел даних (зовнішніх систем, програмних інтерфейсів, потоків подій, вебхуків) і приймання сирих записів; тут розв’язуються задачі автентифікації, обмеження частоти запитів (*rate limiting*) та стійкості до збоїв окремого джерела.
2. Накопичення — зберігання сирих і проміжних даних із контролем часу життя запису (*TTL*), усуненням дублікатів і поділом на зони за рівнем обробки.
3. Керування — конфігурування системи, перевірка вхідних даних за схемою, версіонування результатів, відстеження походження даних (*lineage*).
4. Перетворення — зведення накопичених даних до агрегованих, збагачених ознак та похідних артефактів, у тому числі із залученням моделі машинного навчання.

Ці процеси не утворюють лінійного «водоспаду»: у реальній системі вони замикаються в повторюваний конвеєр, де керування є наскрізним шаром над трьома іншими.

1.3. Парадигми ETL та ELT

Історично перетворення даних будували за схемою ETL (*Extract — Transform — Load*, видобування — перетворення — завантаження): дані видобувають із джерела, перетворюють у проміжному середовищі й лише потім завантажують у сховище. З появою недорогих масштабованих сховищ поширилася зворотна за порядком кроків схема ELT (*Extract — Load — Transform*, видобування — завантаження — перетворення): сирі дані спершу завантажують у сховище «як є», а перетворення виконують уже всередині сховища, часто декларативно [1].

Перевага схеми ELT полягає у збереженні сирих даних, що дозволяє повторно виконати перетворення за зміни вимог без повторного звернення до джерела. Це узгоджується з даноцентричним підходом [5]: сирий шар є джерелом істини, а похідні шари перебудовуються повторювано. У досліджуваній системі реалізовано саме ELT-подібну логіку — джерела віддають сирі зрізи, які зберігаються та згодом агрегуються.

Вибір між цими підходами тісно пов’язаний із моментом застосування схеми до даних. Підхід «схема під час запису» (*schema-on-write*) вимагає привести дані до строгої структури ще на вході, що забезпечує сувору цілісність, але ускладнює приймання різнорідних і слабоструктурованих джерел. Підхід «схема під час читання» (*schema-on-read*) дозволяє зберігати дані у первісному вигляді й накладати структуру вже під час їх використання, що є гнучкішим для неструктурованих даних [2]. Для предметної області цієї роботи, де основними джерелами є текстова комунікація різного формату, доцільнішим є збереження сирих даних із подальшим перетворенням, тобто підхід, ближчий до «схеми під час читання».

Узагальнене порівняння двох парадигм за ключовими критеріями наведено в табл. 1.1.

Таблиця 1.1

Порівняння парадигм ETL та ELT

Критерій	**ETL**	**ELT**
Момент перетворення	до завантаження, у проміжному середовищі	після завантаження, усередині сховища
Зберігання сирих даних	не зберігаються	зберігаються як джерело істини

Критерій	**ETL**	**ELT**
Повторне перетворення	потребує нового звернення до джерела	виконується повторно без звернення до джерела
Придатність до неструктурованих даних	низька (потрібна строга схема)	висока (схема під час читання)
Типове середовище	традиційні сховища даних	масштабовані хмарні сховища

Зіставлення показує, що визначальною перевагою схеми ELT для досліджуваної задачі є збереження незмінного сирого шару: воно дозволяє безкоштовно переграти будь-яке перетворення за зміни вимог і робить конвеєр стійким до помилок у логіці обробки. Саме тому в системі прийнято ELT-подібну організацію потоку даних, а строгу схему накладають лише на похідних, золотих рівнях.

1.4. Архітектури сховищ даних і медальйонна модель

За класичним означенням, сховище даних — це предметно-орієнтована, інтегрована, незмінна та підтримувана в часі сукупність даних, призначена для підтримки процесу прийняття управлінських рішень [2]. У свою чергу, озеро даних — це централізоване сховище, що дає змогу зберігати структуровані й неструктуровані дані будь-якого обсягу в їхньому первісному форматі. Спираючись на ці означення, можна виокремити три базові архітектури зберігання аналітичних даних [2, 6]:

- сховище даних (*data warehouse*) — структуроване сховище зі строгою схемою, оптимізоване під аналітичні запити, проте погано пристосоване до неструктурованих даних;
- озеро даних (*data lake*) — сховище сирих даних довільного формату; гнучке, але схильне до деградації в «болото даних» (*data swamp*) за відсутності керування;
- гібридне сховище (*lakehouse*) — поєднання дешевого зберігання сирих даних із транзакційністю та схемою поверх них [6].

Порівняння трьох архітектур за визначальними критеріями наведено в табл. 1.2.

Таблиця 1.2

Порівняння архітектур зберігання аналітичних даних

Критерій	**Сховище даних**	**Озеро даних**	**Гібридне сховище**
Структура даних	строга схема	довільний формат	сирі дані зі схемою поверх них
Неструктуровані дані	низька підтримка	висока підтримка	висока підтримка
Транзакційність	висока	відсутня	висока
Ризик деградації	низький	високий («болото даних»)	помірний
Вартість зберігання	висока	низька	низька

Зіставлення свідчить, що для системи, яка приймає різнорідну неструктуровану комунікацію, але потребує впорядкованого, придатного до аналізу результату, найкраще пасує гібридна модель: вона поєднує дешеве зберігання сирих даних із дисципліною схеми на похідних рівнях. Саме на цю модель спирається подальший опис потоку даних у системі.

Для впорядкування гібридного сховища широко застосовують медальйонну архітектуру. Медальйонна архітектура — це шаблон проєктування даних, що використовується для логічного впорядкування даних у гібридному сховищі з метою послідовного й поступового підвищення їхньої структури та якості під час проходження через шари архітектури [7]. Вона передбачає три логічні зони [7]: бронзову (*bronze*) — сирі дані «як надійшли»; срібною (*silver*) — очищені, дедупльовані, типізовані дані; золотою (*gold*) — агреговані, збагачені дані, готові до споживання, зокрема для машинного навчання. Ця модель прямо лягає на чотири процеси теми й використовується далі як каркас для опису потоку даних у системі.

Перевагою медальйонної моделі є чітке розмежування відповідальності між зонами та повторюваність кожного переходу. Оскільки бронзова зона зберігає незмінні сирі дані, будь-який дефект, виявлений у срібній чи золотій зоні, можна виправити повторним перетворенням без втрати вихідної інформації. Це безпосередньо реалізує принцип ідемпотентності похідних шарів і робить конвеєр стійким до помилок у логіці перетворення: на відміну від систем, які перезаписують дані «на місці», тут завжди є можливість відтворити кінцевий результат із первинного джерела істини.

1.5. Оркестрація конвеєрів даних

Конвеєр даних є графом залежних задач, які потрібно запускати за розкладом або подією, повторювати після збоїв і спостерігати за їх виконанням. Цим опікується оркестратор. Галузевими прикладами є Apache Airflow — зрілий оркестратор на основі орієнтованих ациклічних графів, де конвеєр описується програмним кодом [8]; Dagster — оркестратор, орієнтований на активи даних, із вбудованою типізацією та тестуванням [9]; Prefect — легша альтернатива з акцентом на динамічні потоки [10]. Стисле порівняння цих оркестраторів наведено в табл. 1.5.

Таблиця 1.5

Порівняння оркестраторів конвеєрів даних

Оркестратор	**Модель опису конвеєра**	**Визначальна особливість**
Apache Airflow	орієнтований ациклічний граф у програмному коді	зрілість і велика екосистема розширень
Dagster	орієнтація на активи даних	вбудована типізація й тестування
Prefect	динамічні потоки виконання	легкість і простота запуску

Спільним для всіх трьох є те, що вони розраховані на розгалужені конвеєри з багатьма взаємозалежними кроками та потребують окремого розгортання й супроводу.

У середовищах мови JavaScript (платформа Node.js) аналогічні задачі — черги фонових робіт, відкладене та періодичне виконання — розв’язують інакше: за допомогою черг на основі сховища Redis (бібліотека BullMQ [16]) або рушіїв довготривалих робочих процесів (Temporal [17]). У досліджуваній системі оркестрацію реалізовано мінімалістично: окремий планувальник виконує періодичні задачі за розкладом, що достатньо для поточного масштабу й уникає накладних витрат повноцінного оркестратора. Зазначені інструменти [8, 9, 10] наведено тут як галузевий контекст, відносно якого обґрунтовано спрощений вибір.

Вибір рівня складності оркестрації має відповідати масштабу й характеру задач. Повноцінні оркестратори [8, 9, 10] виправдані для розгалужених конвеєрів із десятками взаємозалежних кроків, складними умовами повторного запуску та потребою в розподіленому виконанні; вони, проте, потребують окремого

розгортання, супроводу та навчання команди. Для систем із невеликою кількістю періодичних, слабко пов’язаних між собою задач застосування такого інструмента призводить до невиправданого ускладнення архітектури. Тому в роботі застосовано власний планувальник, що виконує обмежений набір періодичних задач, гарантуючи їх послідовність і застосування змін розкладу без перезапуску; за зростання масштабу цей складник можна замінити на зовнішній оркестратор без зміни решти системи завдяки чіткому розмежуванню доменів.

1.6. Керування даними: якість, валідація, версіонування

Шар керування даними (*data governance*) забезпечує довіру до даних [1]. Його складовими є:

- перевірка якості та валідація даних — контроль вхідних записів на відповідність очікуваній схемі й обмеженням; галузевим інструментом є Great Expectations, що реалізує декларативні «очікування» щодо даних [13];
- версіонування даних — інструменти DVC (*Data Version Control*) [14] та lakeFS [15] дають змогу фіксувати версії наборів даних подібно до систем контролю версій, забезпечуючи відтворюваність експериментів;
- відстеження походження даних (*lineage*) — фіксація того, з яких джерел і через які перетворення отримано кожен артефакт, що є критичним для аудиту й налагодження;
- конфігурація як керований стан — зберігання параметрів системи в базі даних із валідацією та можливістю зміни без перезапуску.

Останній принцип безпосередньо застосовано у досліджуваній системі, де вся конфігурація поведінки централізована в базі даних і перевіряється під час виконання.

Перевірку якості даних доцільно проводити не одноразово, а на кожній межі між зонами конвеєра, де дані змінюють форму. Серед вимірів якості зазвичай виокремлюють повноту (відсутність пропусків в обов’язкових полях), узгодженість (несуперечність значень між собою та з визначеними обмеженнями), своєчасність (актуальність даних на момент використання) та унікальність (відсутність

небажаних дублікатів) [1]. Відображення цих вимірів на конкретні механізми контролю, реалізовані в системі, наведено в табл. 1.3.

Таблиця 1.3

Виміри якості даних і механізми їх контролю в системі

Вимір якості	**Зміст**	**Механізм контролю в системі**
Повнота	відсутність пропусків в обов'язкових полях	перевірка обов'язковості полів за описом-схемою
Узгодженість	несуперечність значень обмеженням	перевірка типу та регулярного виразу поля
Своєчасність	актуальність даних на момент використання	збирання за заданим часовим вікном
Унікальність	відсутність небажаних дублікатів	усунення дублікатів за складеним ключем

Невиконання будь-якого з цих вимірів на ранній стадії поширюється далі конвеєром і проявляється вже на рівні результатів моделі, де його значно важче діагностувати. Тому перевірку конфігурації та вхідних даних винесено на саму межу приймання, що зменшує вартість виявлення й усунення дефектів.

1.7. Перетворення даних і сховища ознак

Кінцевою метою конвеєра є ознаки, придатні для навчання моделі. Тут виокремлюють дві ключові практики. Перша — декларативне перетворення: інструмент dbt (*data build tool*) описує перетворення як версіоновані моделі мовою SQL із вбудованими тестами та документацією [11]. Друга — застосування сховища ознак (*feature store*), наприклад Feast, яке узгоджує ознаки між навчанням і використанням, усуваючи розбіжність між цими режимами (*training/serving skew*) [12].

У системах, де роль ознаки відіграє не числовий вектор, а семантично стиснений текст, частину перетворення бере на себе велика мовна модель: вона зводить масив сирих повідомлень до структурованої зведеної характеристики (переліку задач з оцінками). Це другий, відмінний від класичного, вузол машинного навчання, який також належить до шару перетворення.

Застосування мовної моделі як складника конвеєра перетворення має свою специфіку порівняно з традиційними декларативними перетвореннями. З одного боку, мовна модель здатна опрацьовувати неструктурований текст довільного

формату й виокремлювати з нього смисл без явного програмування правил, що є вирішальним для предметної області цієї роботи. З іншого боку, її вихід є недетермінованим і потребує додаткового впорядкування перед збереженням, а також контролю обсягу вхідних даних з огляду на обмеження на довжину запиту. Через це шар перетворення поєднує детерміновані операції (збирання, серіалізацію, агрегацію серій) із недетермінованим викликом мовної моделі, а проміжні результати кожного рівня зберігаються окремо, що дозволяє повторно використати їх без повторного звернення до моделі.

Окремо слід зважити на проблему розбіжності між режимами навчання та використання (*training/serving skew*), яку у класичних конвеєрах розв'язують сховища ознак. Її суть полягає в тому, що ознаки, обчислені під час навчання, можуть не збігатися з ознаками, обчисленими під час застосування моделі, якщо логіка їх формування дублюється у двох різних місцях. У досліджуваній системі цю проблему пом'якшено архітектурно: і для навчання, і для подальшого використання ознаки походять з одного й того самого впорядкованого золотого шару, сформованого єдиним конвеєром перетворення. Таким чином, замість окремого сховища ознак роль єдиного джерела істини відіграє сам медальйонний конвеєр, що для масштабу досліджуваної задачі є простішим і достатнім рішенням.

1.8. Огляд наявних рішень і виявлена прогалина

Перелічені вище інструменти (Airflow [8], Dagster [9], Prefect [10], dbt [11], Feast [12], Great Expectations [13], DVC [14]) є будівельними блоками інфраструктури даних, а не готовими цільовими системами. Вони універсальні й потребують інтеграції під конкретну предметну область, тому досліджувана система не конкурує з ними, а застосовує їхні ідеї до вузької задачі.

Найближчими за призначенням є платформи інженерної аналітики розробки програмного забезпечення (Software Engineering Intelligence) — Jellyfish, LinearB, Swarmia, Faros AI, — що збирають дані із систем контролю версій, трекерів задач і конвеєрів неперервної інтеграції та обчислюють метрики потоку доставки. Метричною основою таких платформ слугують дві визнані рамки: DORA, що

вводить чотири ключові метрики доставки (частоту розгортань, час від унесення зміни до її впровадження, частку невдалих змін і час відновлення) [3], та SPACE — багатовимірна рамка продуктивності розробників, яка свідомо застерігає від зведення продуктивності до однієї метрики [4].

Перелічені платформи орієнтовані на структуровані інженерні артефакти (унесення змін, запити на злиття, тікети), є закритими та дорогими. Поза їхньою увагою лишається неструктурована комунікація команди — повідомлення в месенджерах, обговорення, підсумки нарад, — де насправді фіксується значна частина робочого контексту. Саме в цю нішу спрямована досліджувана система: вона збирає неструктуровані текстові потоки з кількох джерел і за допомогою великої мовної моделі зводить їх до переліку задач із показниками (швидкість виконання, вплив на бізнес), формуючи метрики, споріднені з DORA та SPACE, але на якісно інших вхідних даних. Зіставлення платформ інженерної аналітики з пропонованою системою за ключовими ознаками наведено в табл. 1.4.

Таблиця 1.4

Зіставлення платформ інженерної аналітики з пропонованою системою

Ознака	**Платформи інженерної аналітики**	**Пропонована система**
Джерела даних	системи контролю версій, трекери задач, конвеєри інтеграції	месенджери, обговорення, підсумки нарад
Тип вхідних даних	структуровані артефакти	неструктурована текстова комунікація
Метод опрацювання	агрегація структурованих подій	семантичне стиснення мовною моделлю
Метрична основа	DORA, SPACE	показники, споріднені з DORA та SPACE
Доступність	закриті комерційні продукти	власна реалізація з відкритим стеком

Наведене зіставлення унаочнює нішу досліджуваної системи: вона працює з тим шаром даних, який лишається поза увагою наявних платформ, і застосовує до нього семантичне опрацювання, що донедавна було важко автоматизувати.

Опора лише на структуровані артефакти має принципове обмеження: значна частина рішень, домовленостей і змін контексту узгоджується саме в неформальному спілкуванні й ніколи не потрапляє до трекерів задач чи систем контролю версій. Як наслідок, метрики, обчислені виключно за структурованими

джерелами, дають неповну й подеколи викривлену картину діяльності команди. Опрацювання неструктурованої комунікації, навпаки, потребує етапу семантичного розуміння тексту, який донедавна було важко автоматизувати; поява доступних великих мовних моделей зробила такий етап практично здійсненним і відкрила можливість будувати показники діяльності на основі первинного, найбагатшого на контекст джерела. Підсумовуючи проведений аналіз, можна стверджувати, що побудова інфраструктури даних, яка систематично збирає, накопичує, керує та перетворює неструктуровану командну комунікацію, є актуальною та недостатньо охопленою наявними рішеннями задачею, розв’язанню якої й присвячено цю роботу.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

2.1. Вимоги до системи

2.1.1. Функціональні вимоги

На основі аналізу предметної області, проведеного в розділі 1, сформульовано функціональні вимоги до системи, наведені в табл. 2.1.

Таблиця 2.1

Функціональні вимоги до системи

Код	**Вимога**
ФВ-1	Підключення довільної кількості зовнішніх джерел даних (Telegram, Discord, GitHub, Fireflies) з конфігуруванням через програмний інтерфейс.
ФВ-2	Збирання сирих даних із джерел за часовим вікном на запит конвеєра (за моделлю опитування).
ФВ-3	Перевірка конфігурації кожної інтеграції за описом-схемою перед збереженням.
ФВ-4	Усунення дублікатів і хронологічне впорядкування зібраних записів.
ФВ-5	Зберігання конфігурації, шаблонів, каналів і результатів у реляційній базі даних.
ФВ-6	Перетворення сирих даних на трьох рівнях агрегації: коротке вікно, добовий і тижневий звіти.
ФВ-7	Виклик великої мовної моделі для семантичного стиснення даних у структурований звіт із показниками.
ФВ-8	Версіонування результатів за рівнем агрегації та читачем.
ФВ-9	Розсилання готових звітів у налаштовані канали виводу з посиланням на повну версію.
ФВ-10	Періодичний автоматичний запуск конвеєра за розкладом із застосуванням змін налаштувань без перезапуску.
ФВ-11	Публічний доступ до повного звіту за унікальним ідентифікатором.

2.1.2. Нефункціональні вимоги

Нефункціональні вимоги до системи наведено в табл. 2.2.

Таблиця 2.2

Нефункціональні вимоги до системи

Код	**Вимога**
НФВ-1	Стійкість до збоїв джерела: падіння або невалідні облікові дані одного джерела не зривають цикл, а проблемна інтеграція автоматично вимикається.
НФВ-2	Керованість: уся поведінка задається станом у базі даних, без функціональної конфігурації у змінних середовища; зміна налаштувань застосовується без перезапуску.
НФВ-3	Узгодженість запусків: важкі прогони конвеєра виконуються послідовно й не конкурують за спільні ресурси.

Код	Вимога
НФВ-4	Безпека облікових даних: ключі зберігаються лише в базі даних, доступ до них інкапсульовано в сервісному шарі.
НФВ-5	Розширюваність: нове джерело додається реалізацією єдиного інтерфейсу адаптера без зміни конвеєра.
НФВ-6	Відтворюваність розгортання: уся система піднімається одним описом середовища контейнеризації.
НФВ-7	Самодокументованість програмного інтерфейсу: кожну точку доступу описано за специфікацією OpenAPI.

2.2. Архітектурний стиль і принципи

Систему реалізовано як єдиний застосунок на каркасі NestJS (мова TypeScript) [19] із чіткими межами доменів. Замість мікросервісного розбиття обрано модульний моноліт: кожен домен є самодостатнім модулем із власним сервісом, контролером і описами даних, а взаємодія між доменами відбувається лише через явний експорт модуля та механізм впровадження залежностей (*dependency injection*). Такий вибір знижує операційну складність (один процес, одне розгортання), не втрачаючи модульності.

Вибір модульного моноліту замість мікросервісної архітектури обґрунтовано співвідношенням переваг і витрат для поточного масштабу системи. Мікросервісна архітектура виправдана, коли окремі складники мають різні профілі навантаження, розробляються незалежними командами та потребують самостійного масштабування; за це доводиться платити складністю мережевої взаємодії, розподілених транзакцій, розгортання й спостереження. Для системи з єдиним профілем навантаження та спільною моделлю даних ці витрати не компенсуються вигодами, тоді як модульний моноліт зберігає чіткі межі доменів усередині одного процесу й залишає можливість виокремити окремий домен у самостійну службу згодом, за потреби.

Усі домени дотримуються єдиного шаблону організації коду: один домен — один файл — один простір імен, усередині якого зібрано описи даних, сервіс, контролер і модуль. Це забезпечує вертикальну зв’язність домену та горизонтальну ізоляцію між доменами. Точка входу застосунку задає глобальний префікс маршрутів, вмикає наскрізну перевірку вхідних даних і публікує документацію інтерфейсу; кореневий модуль збирає граф із усіх доменних модулів.

2.2.1. Логічні шари

Домени системи групуються у три логічні шари, що відповідають процесам теми (рис. 2.1): шар збирання, наскрізний шар керування та шар перетворення; окремо стоїть шар накопичення.

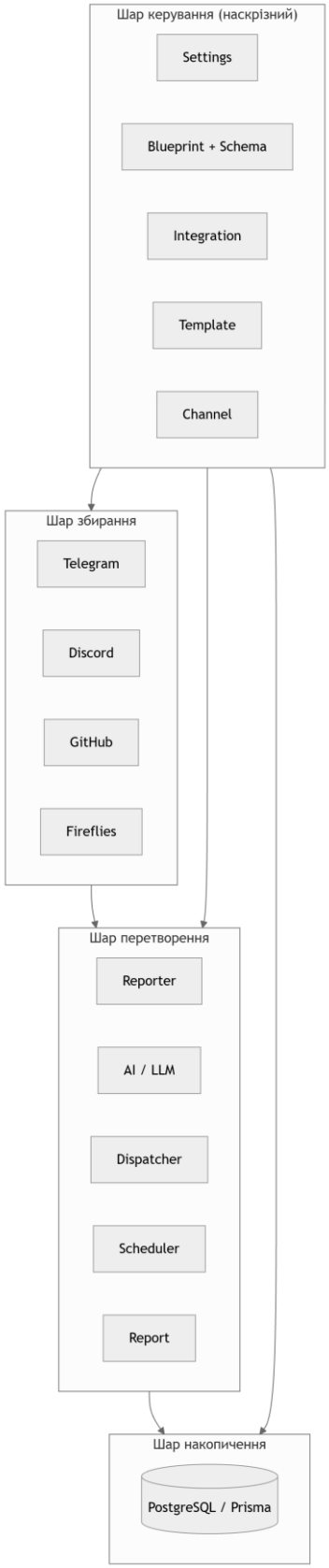

Рисунок 2.1. Логічні шари системи

2.3. Модель даних

Модель даних описано засобами об’єктно-реляційного відображення Prisma [20] і відображено на реляційну схему системи керування базами даних PostgreSQL [21]. Структуру даних подано на рис. 2.2 у вигляді діаграми «сутність — зв’язок».

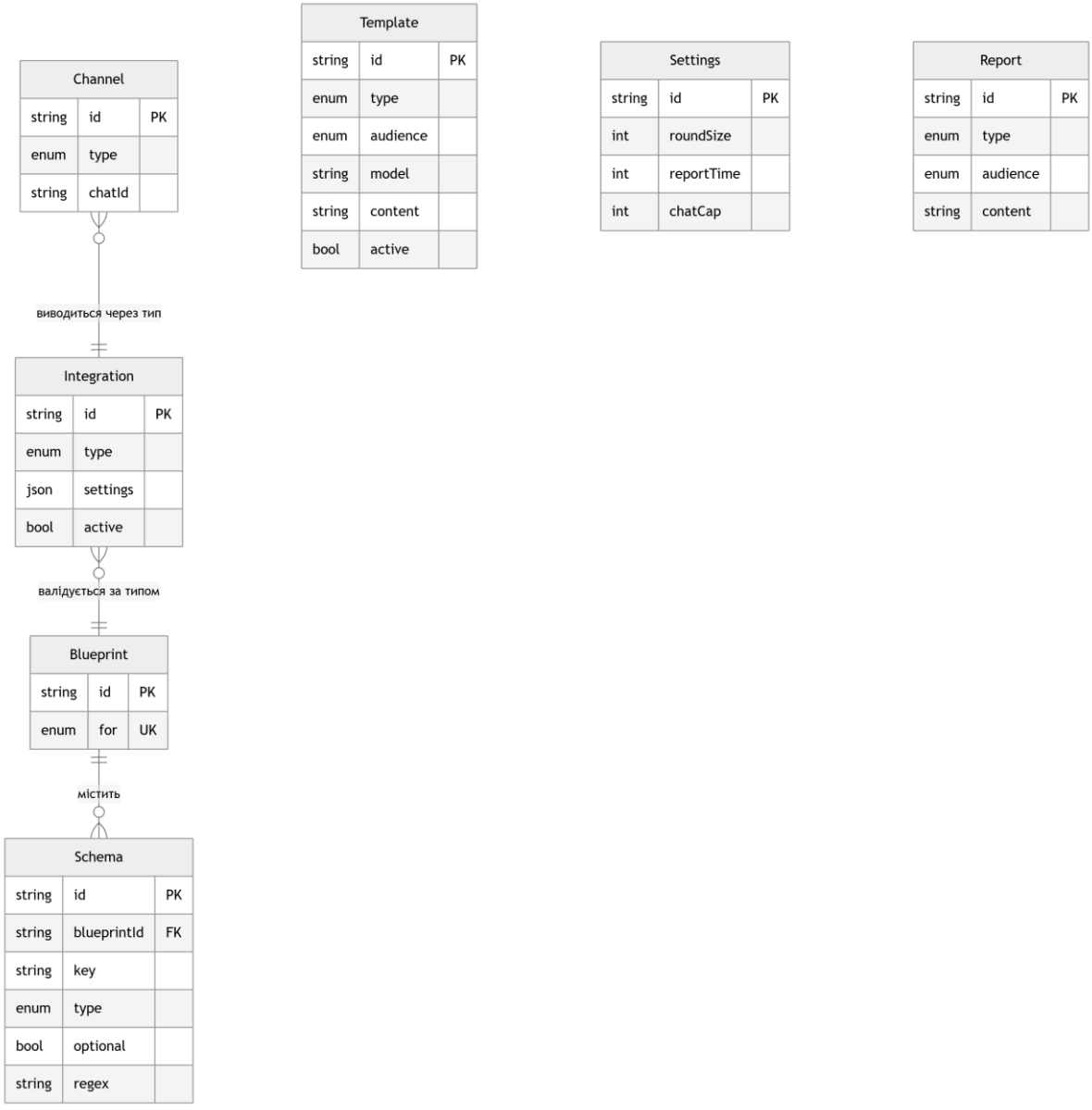

Рисунок 2.2. Модель даних (діаграма «сутність — зв’язок»)

Сутність опису-схеми (*Blueprint*) пов’язана з полями опису (*Schema*) фізичним зовнішнім ключем із каскадним видаленням. Зв’язки інтеграції з описом-схемою та каналу з інтеграцією є логічними (за типом) і перевіряються програмно. Сутність налаштувань є єдиним рядком, а результати (*Report*) розрізняються парою «рівень агрегації — читач», що дозволяє детерміновано добирати потрібну серію під час агрегації.

2.4. Декомпозиція на компоненти та граф залежностей

Залежності між доменними модулями за впровадженням сервісів утворюють ациклічний граф (рис. 2.3). Планувальник запускає формування звіту; модуль звіту звертається до конвеєра перетворення та розсилача; конвеєр спирається на сервіс збирання, шаблони, налаштування й посередника до мовної моделі; сервіс збирання, своєю чергою, об’єднує адаптери окремих джерел.

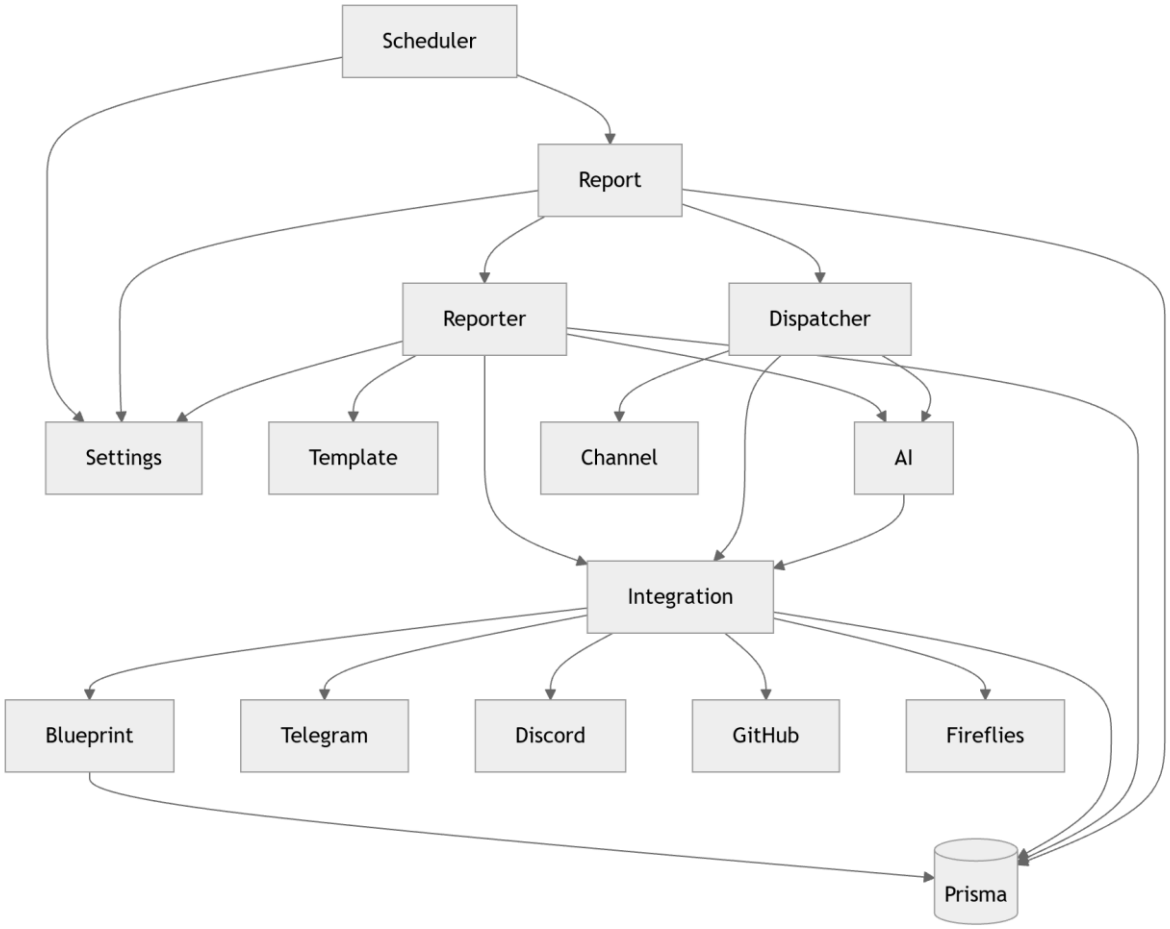

Рисунок 2.3. Граф залежностей компонентів

Адаптери джерел впроваджуються в сервіс збирання як масив реалізацій єдиного інтерфейсу, що дозволяє додавати нові джерела без зміни конвеєра й безпосередньо реалізує вимогу розширюваності (НФВ-5).

2.5. Динаміка: конвеєр генерації звіту

На рис. 2.4 наведено діаграму послідовності повного циклу формування добового звіту — від спрацювання планувальника до розсилання в канали. Цикл короткого вікна є підмножиною цього процесу без агрегації серій і без розсилання.

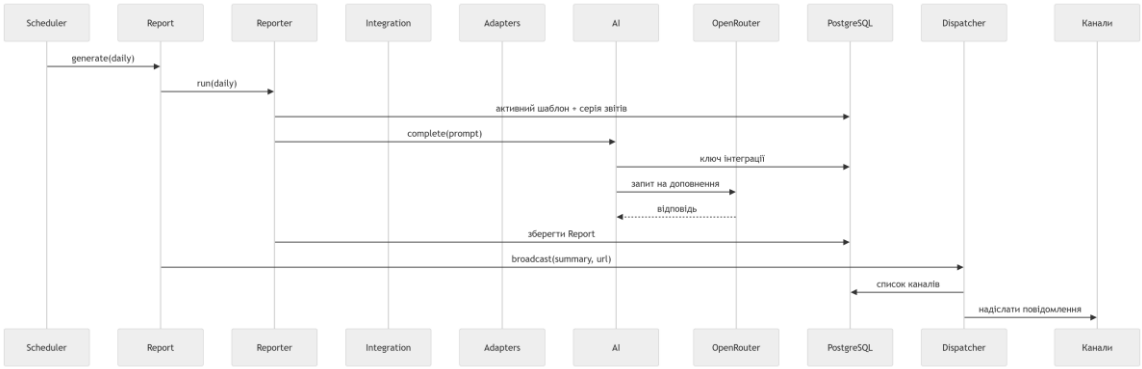

Рисунок 2.4. Діаграма послідовності генерації добового звіту

Для циклу короткого вікна збирання сирих даних виконує сервіс збирання, який паралельно опитує всі адаптери, склеює та усуває дублікати в зібраному наборі. Часове вікно визначається трирівневим правилом: явно заданим значенням, інтервалом від останнього звіту короткого рівня або значенням за замовчуванням із налаштувань.

2.6. Проєкція на медальйонну архітектуру

Потік даних відображається на зони медальйонної моделі (рис. 2.5).

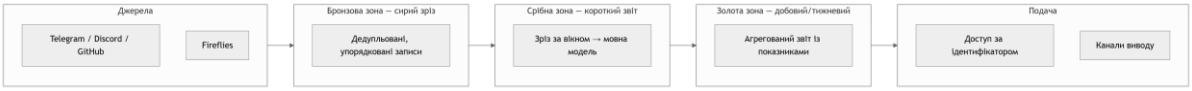

Рисунок 2.5. Проєкція потоку даних на медальйонну архітектуру

Бронзовою зоною є набір сирих, але дедупльованих і хронологічно впорядкованих записів. Срібною зоною є звіт за коротким вікном — сирий зріз, перетворений мовною моделлю на первинну зведену характеристику. Золотою зоною є добові й тижневі звіти, що агрегують серію нижчого рівня з урахуванням попереднього звіту як «пам’яті» та виокремлюють показники діяльності (швидкість виконання, вплив на бізнес). Саме золота зона є придатним для машинного навчання артефактом.

2.7. Обґрунтування технологічного стеку

Обраний технологічний стек і обґрунтування кожного шару наведено в табл. 2.3.

Таблиця 2.3

Технологічний стек системи та його обґрунтування

Шар	**Технологія**	**Обґрунтування**
Мова	TypeScript	Статична типізація поверх платформи Node.js; єдина мова з клієнтською частиною.
Каркас бекенду	NestJS [19]	Впровадження залежностей, модульність, вбудована підтримка специфікації OpenAPI та валідації.
Сховище	PostgreSQL [21]	Реляційна цілісність і поля типу JSON для гнучкої конфігурації.
Відображення даних	Prisma [20]	Декларативна схема як джерело істини, типобезпечний клієнт, міграції.
Гаряча зона	Redis [22]	Зберігання сирих записів з обмеженням часу життя за потреби потокового збирання.
Шлюз до мовних моделей	OpenRouter [18]	Єдиний інтерфейс до багатьох моделей із вибором моделі на рівні шаблону.
Документація інтерфейсу	OpenAPI	Автоматичне формування опису з анотацій коду.
Розгортання	Docker Compose [26]	Відтворюване локальне та промислове середовище.
Клієнтська частина	Next.js	Адміністративна панель керування інтеграціями, шаблонами та налаштуваннями.

Відносно галузевих стандартів у роботі прийнято два свідомі спрощення, обґрунтовані масштабом задачі. Перше — застосування власного планувальника на періодичних таймерах замість повноцінного зовнішнього оркестратора (Airflow [8], Dagster [9]) чи черги фонових робіт (BullMQ [16]), що усуває залежність від важкого зовнішнього компонента для поточного обсягу задач. Друге — застосування моделі опитування замість постійного потокового передавання в чергу: адаптери віддають зріз на запит конвеєра, що спрощує модель взаємодії та знижує накладні витрати.

2.8. Розгортання

Систему розгортають засобами Docker Compose [26]; цілісне середовище піднімає зворотний проксі, застосунок, клієнтську частину та базу даних (рис. 2.6). Уся функціональна конфігурація зберігається в базі даних, а середовище містить лише рядки підключення та параметри запуску.

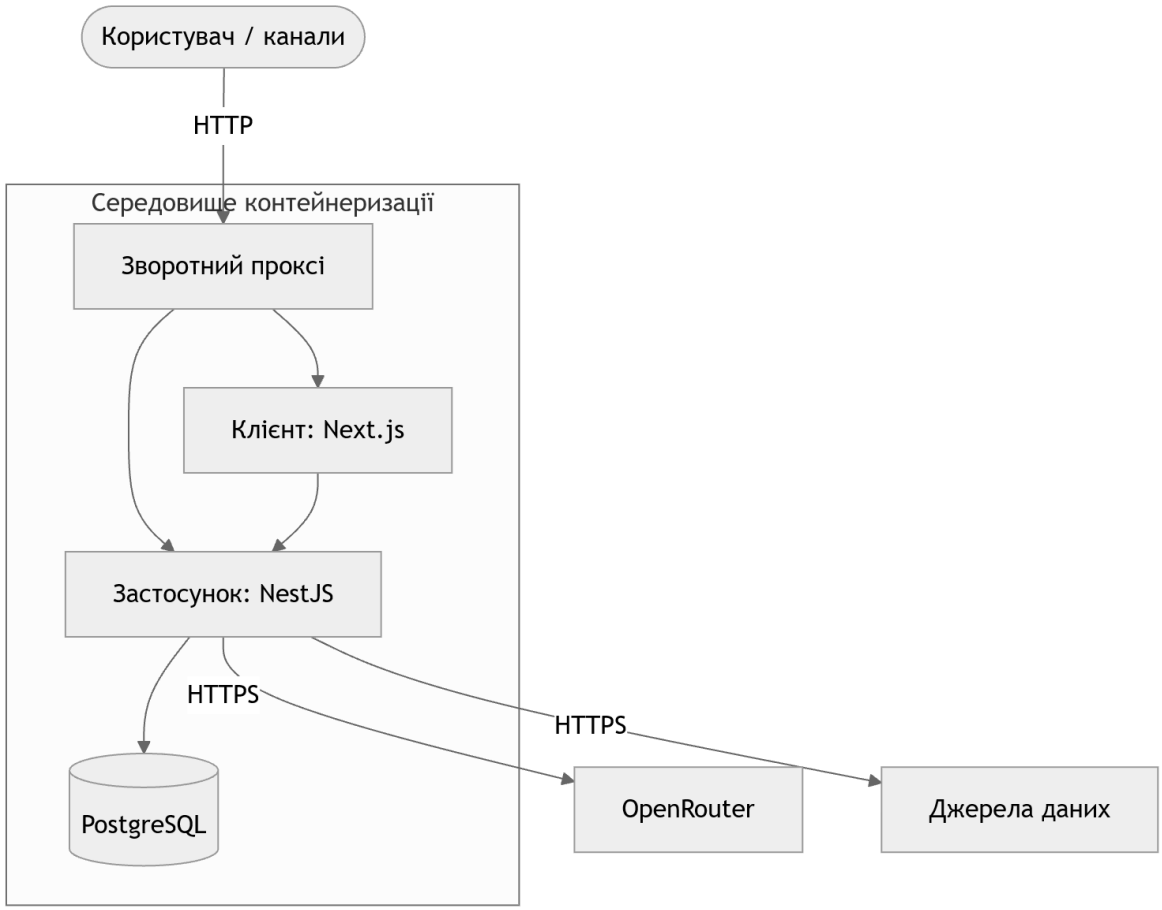

Рисунок 2.6. Схема розгортання системи

2.9. Сценарій використання системи

Щоб пов’язати спроєктовані компоненти з практикою, розглянемо типовий наскрізний сценарій використання системи з погляду адміністратора. Спершу адміністратор реєструє джерело даних: надсилає до програмного інтерфейсу опис інтеграції із зазначенням її типу та параметрів підключення. Перед збереженням система звіряє надані параметри з описом-схемою відповідного типу — за відсутності обов’язкового поля, невідповідності типу чи порушення формату запис відхиляється з діагностичним повідомленням, тож завідомо непрацездатна конфігурація не потрапляє до системи.

Після реєстрації принаймні одного джерела та налаштування активного шаблону запиту до мовної моделі система працює без втручання людини. Планувальник у визначені моменти запускає конвеєр: сервіс збирання опитує всі активні джерела за часовим вікном, зводить отримані записи, усуває дублікати та

впорядковує їх за часом; конвеєр перетворення серіалізує підготовлений зріз, підставляє його в активний шаблон і звертається до мовної моделі, яка зводить масив повідомлень до структурованого звіту з показниками. Готовий звіт зберігається у базі даних із позначенням рівня агрегації та читача, після чого розсилач надсилає його стислу версію в налаштовані канали виводу, додаючи посилання на повну версію.

Якщо в процесі роботи облікові дані певного джерела стають недійсними, система автоматично вимикає проблемну інтеграцію, не перериваючи опрацювання решти джерел; для відновлення достатньо оновити облікові дані та повторно ввімкнути запис. Зміна параметрів роботи — наприклад, періодичності формування звітів — набуває чинності негайно, без перезапуску застосунку. Описаний сценарій демонструє, що спроєктована архітектура забезпечує наскрізний цикл опрацювання даних із мінімальним адмініструванням і коректним реагуванням на позаштатні ситуації.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

Розділ описує програмну реалізацію спроєктованої в розділі 2 архітектури. Виклад структуровано за чотирма процесами теми — збирання, накопичення, керування та перетворення даних, — з наскрізними підрозділами щодо структури проєкту, планування й розсилання. Для кожного процесу наведено опис застосованих механізмів і показано, як вони задовольняють вимоги, сформульовані в розділі 2.

3.1. Структура проєкту та технологічна основа

Систему реалізовано як монорепозиторій із двома застосунками: серверним, на каркасі NestJS [19], і клієнтським, на каркасі Next.js. Монорепозиторій обрано для того, щоб серверна та клієнтська частини спільно використовували описи типів даних і єдині домовленості про формат обміну, а зміни в обох частинах могли узгоджуватися в межах однієї ревізії. Доменну логіку зосереджено в серверному застосунку, де кожному домену відповідає окремий файл — простір імен.

Кожен домен дотримується єдиного шаблону організації коду: простір імен містить описи даних із декларативними правилами перевірки, сервіс із прикладною логікою, контролер, що публікує точки доступу, і модуль, який оголошує залежності домену. Такий шаблон забезпечує вертикальну зв’язність кожного домену (усе, що стосується домену, зосереджено в одному місці) та горизонтальну ізоляцію між доменами (взаємодія можлива лише через явний експорт модуля). Доступ між доменами здійснюється виключно через механізм впровадження залежностей, що усуває приховані зв’язки й полегшує тестування з підставними реалізаціями.

Точка входу серверного застосунку встановлює глобальний префікс маршрутів, вмикає наскрізну перевірку вхідних даних і публікує документацію інтерфейсу за специфікацією OpenAPI. Режим суворої перевірки налаштовано так, що до сервісів не потрапляє жодне поле, не оголошене в описі вхідних даних: невідомі поля відхиляються, а типи приводяться до оголошених. Це є базовим

рубежем контролю якості вхідної інформації та реалізує вимогу самодокументованості інтерфейсу (НФВ-7), а автоматично сформована за анотаціями коду документація гарантує, що опис кожної точки доступу відповідає її фактичній поведінці.

3.2. Реалізація шару збирання

3.2.1. Контракт адаптера джерела

Усі джерела даних реалізують єдиний інтерфейс адаптера, який оголошує тип джерела та метод отримання зрізу даних від заданого моменту часу. Зріз подається як набір із трьох колекцій — користувачів, чатів і повідомлень, — що утворюють спільну структуру, незалежну від конкретного джерела. Адаптери впроваджуються в сервіс збирання як масив реалізацій через окремий постачальник залежностей, тому сервіс збирання не залежить від конкретного переліку джерел. Це безпосередньо реалізує вимогу розширюваності (НФВ-5): додавання нового джерела зводиться до реалізації інтерфейсу та реєстрації ще однієї залежності, без жодних змін у конвеєрі. Контракт адаптера та обгортку запиту зі стійкістю до збоїв наведено в додатку А.

3.2.2. Збирання та усунення дублікатів

Сервіс збирання опитує всі зареєстровані адаптери паралельно, ізолюючи можливий збій кожного з них. Якщо окремий адаптер завершується помилкою, він повертає порожній зріз, помилку фіксують у журналі, а опитування решти джерел триває. Завдяки цьому реалізовано вимогу стійкості до збоїв (НФВ-1): недоступність або несправність одного джерела не зриває цикл збирання загалом.

Зведені від усіх адаптерів записи об’єднуються в єдиний набір, з якого усуваються дублікати. Ключ дедуплікації формується складеним: для користувачів і чатів — з ідентифікатора джерела та запису, а для повідомлень — додатково з ідентифікатора чату, оскільки той самий ідентифікатор повідомлення може траплятися в різних чатах. Після усунення дублікатів повідомлення впорядковуються за позначкою часу, що дає конвеєрові перетворення

хронологічно послідовний вхід. Усунення дублікатів виконує узагальнена допоміжна функція з налаштовуваним правилом формування ключа, яку повторно використовують різні частини системи.

3.2.3. Стійкість на рівні протоколу передавання

Адаптери, що працюють за протоколом передавання гіпертексту, використовують спільну допоміжну функцію запиту. Вона інкапсулює типові сценарії взаємодії із зовнішніми службами: у відповідь на код перевищення дозволеної частоти запитів функція зчитує службовий заголовок із рекомендованою затримкою й повторює запит після неї, а відповіді про відмову в доступі перетворює на окремий виняток автентифікації. Цей виняток обробляється на рівні джерела й призводить до автоматичного вимкнення інтеграції з недійсними обліковими даними. Завдяки цьому один недійсний ключ не спричиняє повторюваних помилок у наступних циклах, а відновлення відбувається після ручного оновлення облікових даних.

3.2.4. Адаптер джерела Telegram

Найскладнішим є адаптер джерела Telegram. На відміну від ботового підходу, який обмежений повідомленнями, надісланими безпосередньо боту, цей адаптер під’єднується як користувацький клієнт за протоколом MTProto [25]. Це дає змогу читати історію діалогів, до яких має доступ обліковий запис, що відповідає прикладному призначенню системи — збиранню вже наявної командної комунікації. У реалізації виокремлено два аспекти: авторизацію сесії та власне збирання повідомлень.

Авторизація сесії є процесом скінченного автомата з кількома кроками. Спершу за номером телефону ініціюється вхід і запитується код підтвердження; після надсилання коду, за наявності двофакторного захисту, додатково запитується пароль. Тимчасова сесія входу тримається в пам’яті за згенерованим квитком з обмеженим часом життя та періодичним витісненням прострочених записів, аби не накопичувати незавершені спроби. Інтерфейс на кожному кроці повідомляє клієнтові, який саме ввід потрібен далі, не розкриваючи внутрішнього стану

процесу входу. Після успішної авторизації отриманий рядок сесії зберігається в конфігурації інтеграції й надалі використовується збирачем без повторного проходження всієї процедури.

Збирання повідомлень виконується пакетами. Для кожного активного рядка інтеграції клієнт під’єднується, послідовно читає діалоги й накопичує повідомлення в межах заданого часового вікна, припиняючи читання діалогу, щойно натрапляє на повідомлення, старіше за нижню межу вікна. У реалізації враховано низку прикладних особливостей: обхід тематичних розділів супергруп із формуванням складеного ідентифікатора чату, вилучення імен вкладень із метаданих без їх завантаження, фільтрування занадто великих чатів за відповідним налаштуванням, пропуск повідомлень від ботів, а також автоматичне вимкнення інтеграції в разі характерних для протоколу помилок автентифікації. Інші адаптери (Discord, GitHub, Fireflies) реалізують той самий інтерфейс простіше, оскільки відповідні служби надають структурованіший доступ до даних.

3.3. Реалізація шару накопичення

3.3.1. Реляційне сховище

Тривале зберігання забезпечує система керування базами даних PostgreSQL [21] під керуванням засобу об’єктно-реляційного відображення Prisma [20]. Засіб відображення обрано тому, що він використовує декларативний опис схеми як єдине джерело істини: за цим описом автоматично формуються типобезпечний клієнт доступу та міграції схеми. Сервіс доступу до бази даних впроваджується в будь-який доменний сервіс і керує життєвим циклом з’єднання, що централізує роботу зі сховищем. Для гарячої зони сирих даних за потреби потокового збирання використовується сховище Redis [22] з обмеженням часу життя записів, що дозволяє тимчасово утримувати великі обсяги сирих даних без перевантаження тривалого сховища.

3.3.2. Версіонування результатів

Кожен результат конвеєра зберігається як окремий запис із композитним ключем, що фіксує рівень агрегації (короткий, добовий або тижневий) та читача (для подальшого оброблення мовною моделлю чи для людини). Такий ключ дозволяє конвеєрові вищого рівня детерміновано добирати потрібну серію результатів нижчого рівня за часовим вікном, не вдаючись до додаткових позначок. Це водночас забезпечує відтворюваність перетворень: за збереженими результатами можна простежити, з яких саме вхідних серій сформовано кожен агрегований звіт.

3.3.3. Серіалізація у проміжний формат

Перед подачею до мовної моделі зрізи даних серіалізуються у формат значень, розділених комами. Узагальнена допоміжна функція самостійно визначає набір стовпців за наявними полями та екранує службові символи, забезпечуючи коректність формату навіть для повідомлень довільного змісту. Цей формат обрано як компактний і передбачуваний для мовної моделі: він має менші накладні витрати на лексеми порівняно з форматом обміну об’єктами, що при обмеженнях на довжину запиту дозволяє вмістити більший обсяг корисних даних.

3.4. Реалізація шару керування

3.4.1. Конфігурація як керований стан

Налаштування системи зберігаються єдиним записом, який автоматично створюється з усталеними значеннями за першого звернення й перевіряється на допустимі діапазони на рівні опису вхідних даних. Таке рішення робить базу даних єдиним місцем зберігання поведінкових параметрів і усуває потребу в конфігураційних змінних середовища, що спрощує супровід і зменшує ризик неузгодженості налаштувань між складниками системи.

Ключовою реалізаційною деталлю є механізм підписки. Під час оновлення налаштувань сервіс сповіщає всіх підписників про зміну, завдяки чому залежні складники — насамперед планувальник — перенастроюються негайно й без перезапуску застосунку. Це реалізує вимогу керованості (НФВ-2): адміністратор

може змінити, наприклад, період формування звітів, і нова періодичність набуде чинності одразу.

3.4.2. Перевірка конфігурації за описом-схемою

Для кожного типу інтеграції зберігається опис допустимих полів її конфігурації. Описи завантажуються з конфігураційних файлів під час запуску: процедура звіряє файли з умістом бази даних — створює відсутній опис, оновлює наявні поля та видаляє ті, яких уже немає у файлі, — тож файли залишаються джерелом істини. Перевірка конфігурації конкретної інтеграції контролює для кожного поля наявність (з огляду на ознаку обов’язковості), відповідність оголошеному типу та, за наявності, відповідність регулярному виразу.

Істотною особливістю є те, що перевірка виконується під час будь-якої зміни запису інтеграції, а не лише під час його створення. Це запобігає прихованому дрейфу конфігурації: якщо опис-схема з часом набув нового обов’язкового поля, повторне ввімкнення раніше створеного запису, у якому цього поля немає, завершиться відмовою, а не мовчазною активацією неповної конфігурації. Такий підхід підвищує довіру до даних на межі їх приймання.

3.4.3. Захищене зберігання облікових даних

Облікові дані зовнішніх служб зберігаються лише у відповідних записах бази даних і зчитуються через локалізовану допоміжну функцію, яка інкапсулює доступ до них і не допускає поширення небезпечних приведень типів кодом системи. Облікові дані для звернення до мовної моделі беруться виключно з активного запису відповідного типу й ніколи — зі змінних середовища. Це реалізує вимогу безпеки облікових даних (НФВ-4) і водночас спрощує їх ротацію, оскільки заміна ключа не потребує перерозгортання системи.

3.4.4. Керування шаблонами

Шаблони запитів до мовної моделі групуються за рівнем агрегації та читачем; у межах кожної групи активним може бути рівно один шаблон, саме його використовує конвеєр. Активація шаблону виконується атомарно: в одній

транзакції скидається ознака активності в інших шаблонах групи й установлюється в обраному, що унеможливлює стан із кількома активними шаблонами. Ідентифікатор моделі, зазначений у шаблоні, перевіряється проти переліку доступних моделей шлюзу OpenRouter [18] ще під час запису шаблону, тож помилковий або застарілий ідентифікатор відхиляється завчасно, а не під час формування звіту.

3.5. Реалізація шару перетворення

3.5.1. Конвеєр формування звіту

Ядром перетворення є конвеєр формування звіту, який реалізує медальйонну логіку трьох рівнів. Конвеєр обирає активний шаблон відповідного рівня й читача та будує запит до мовної моделі. Для короткого вікна запит будується із сирого зрізу: сервіс збирання повертає набір записів за вікном, конвеєр відокремлює записи аудіо- та відеодзвінків від текстових повідомлень, серіалізує дані й підставляє їх у заповнювачі шаблону. Для добового та тижневого рівнів запит будується з агрегації: конвеєр добирає серію звітів нижчого рівня за відповідним вікном і додає попередній звіт того ж рівня як «пам’ять», що забезпечує наступність зведень. Підстановку значень у заповнювачі шаблону виконує окрема функція з безпечним пропуском невідомих заповнювачів, тож помилка в шаблоні не призводить до аварійного завершення. Сформований мовною моделлю результат зберігається як окремий запис із відповідним рівнем і читачем.

Трирівнева побудова конвеєра безпосередньо втілює медальйонну логіку накопичення смислу. На короткому рівні мовна модель уперше перетворює сирий зріз повідомлень на первинну зведену характеристику — це перехід від бронзової до срібної зони. На добовому рівні конвеєр уже не звертається до сирих повідомлень, а агрегує серію коротких звітів за добу, отримуючи ущільнене, очищене від шуму зведення. На тижневому рівні так само агрегуються добові звіти. Завдяки тому, що кожен вищий рівень спирається на вже опрацьований результат нижчого, а не на сирі дані, обсяг тексту, який подається до мовної моделі, лишається обмеженим навіть для тривалих періодів, а підсумкове зведення

поступово очищується від несуттєвих деталей — це і є практичним виявом золотої зони медальйонної моделі.

3.5.2. Вузол мовної моделі

Звернення до мовної моделі інкапсульовано в окремому посереднику, що працює зі шлюзом OpenRouter [18], який надає єдиний інтерфейс до багатьох моделей. Посередник обробляє типові ситуації: помилки автентифікації (з автоматичним вимкненням відповідної інтеграції), порожні відповіді та обгорткове форматування, яке модель іноді додає навколо корисного змісту й яке знімається перед збереженням. Окремий режим посередника стискає людиночитний звіт у компактну форму, придатну для надсилання в канали з обмеженням на довжину повідомлення; це другий перехід через мовну модель, потрібний саме для розсилання, тоді як основний звіт зберігається повністю.

3.6. Реалізація планування та розсилання

3.6.1. Планувальник

Планувальник реалізовано не через зовнішній засіб періодичного запуску, а на вирівняних таймерах без накопичення похибки: кожне спрацювання обчислює час до наступного вирівняного слоту, завдяки чому моменти запуску не «спливають» із часом. Звіт короткого вікна формується через задану в налаштуваннях кількість хвилин, добовий — щодня у визначений момент доби, тижневий — наприкінці тижня, причому тижневий запускається після добового, щоб урахувати щойно сформований добовий звіт.

Усі прогони конвеєра виконуються послідовно. Це принципово, оскільки формування звіту є ресурсомістким і звертається до спільної зовнішньої служби мовної моделі; послідовне виконання запобігає конкуренції паралельних прогонів за спільні ресурси (НФВ-3). На старті та за кожної зміни налаштувань таймери перевзводяться, що в поєднанні з механізмом підписки (підрозділ 3.4.1) забезпечує застосування змін розкладу без перезапуску (НФВ-10).

3.6.2. Зведення кроків і розсилання

Окремий сервіс зводить кроки формування звіту воєдино. Цикл короткого вікна лише обчислює часове вікно за трирівневим правилом і повертає результат; добовий і тижневий цикли додатково формують людиночитний звіт, виокремлюють із нього зведення та передають його на розсилання разом із посиланням на повну версію. Розсилач проходить усіма налаштованими каналами виводу; збій надсилання в окремий канал фіксується в журналі й не зупиняє оброблення інших каналів, що підвищує надійність доставки. Для одних каналів звіт додатково стискається у компактну форму, для інших надсилається як є; помилки автентифікації каналу так само вимикають відповідну інтеграцію, узгоджуючись із загальним підходом до недійсних облікових даних.

3.7. Розгортання

Складання й запуск системи виконуються засобами менеджера пакетів; цілісне середовище піднімається засобами контейнеризації Docker Compose [26] і охоплює зворотний проксі, серверний застосунок, клієнтську частину та базу даних. Зворотний проксі спрямовує зовнішні запити до серверного застосунку та клієнтської частини, серверний застосунок звертається до бази даних і до зовнішніх служб. Уся функціональна конфігурація лишається в базі даних, а середовище містить лише рядки підключення та параметри запуску, що робить розгортання відтворюваним (НФВ-6) і переносним між середовищами.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА

Розділ присвячено перевірці системи на двох рівнях: перший — тестування коректності самої інфраструктури даних; другий — експериментальна демонстрація її кінцевої мети, тобто придатності отриманих даних золотої зони для машинного навчання. Другий рівень реалізовано як самодостатній відтворюваний експеримент із базовою моделлю.

4.1. Методика тестування

Прийнято трирівневу піраміду тестування, у якій кількість тестів зменшується від модульного рівня до системного, а вартість і час їх виконання, навпаки, зростають.

На модульному рівні ізольовано перевіряють окремі функції та сервіси, поведінка яких не залежить від зовнішнього середовища: серіалізацію даних у проміжний формат, усунення дублікатів за складеним ключем, обчислення наступного вирівняного слоту розкладу, перевірку конфігурації за описом-схемою. Такі тести виконуються швидко й покривають крайові випадки логіки.

На інтеграційному рівні перевіряють взаємодію кількох доменів через граф залежностей із підставними реалізаціями зовнішніх служб. Тут контролюють, що, наприклад, конвеєр формування звіту правильно добирає активний шаблон, звертається до посередника мовної моделі та зберігає результат, а збій окремого джерела коректно ізолюється. Підставні реалізації дозволяють відтворити рідкісні умови (відмову в доступі, перевищення частоти запитів) без звернення до реальних служб.

На системному рівні виконують наскрізні сценарії через програмний інтерфейс із дійсною базою даних і щонайменше однією налаштованою інтеграцією, що перевіряє систему як єдине ціле — від приймання запиту до збереження та видачі результату. Інструментальною основою слугує стандартний для застосунків NestJS набір засобів модульного та системного тестування.

Важливою передумовою відтворюваності тестування є ізоляція від зовнішніх служб. Звернення до месенджерів, систем контролю версій та мовної моделі замінюються підставними реалізаціями, які повертають наперед визначені відповіді, зокрема й аварійні. Це дозволяє, по-перше, виконувати тести без реальних облікових даних і мережевих звернень, а отже швидко й повторювано; по-друге, відтворювати рідкісні позаштатні умови — відмову в доступі, перевищення частоти запитів, порожню відповідь моделі, — які в реальному середовищі виникають нерегулярно й погано піддаються спостереженню. Тестові дані формуються безпосередньо в коді тесту або завантажуються з фіксованих наборів, що усуває залежність результату від стану зовнішніх систем і робить кожен прогін детермінованим.

4.2. Тестові сценарії інфраструктури

Нижче наведено план тестування ключових інваріантів системи. Очікувані результати випливають із реалізації, описаної в розділі 3, і слугують специфікацією для автоматизованого набору тестів.

Тестові сценарії шару керування наведено в табл. 4.1.

Таблиця 4.1

Тестові сценарії шару керування

№	Сценарій	Очікуваний результат
1	Створення інтеграції з відсутнім обов’язковим полем конфігурації	Відмова з повідомленням про відсутнє поле
2	Створення інтеграції зі значенням невідповідного типу	Відмова з повідомленням про невідповідність типу
3	Значення, що не задовольняє регулярний вираз поля опису-схеми	Відмова з повідомленням про порушення формату
4	Повторне ввімкнення рядка, чий опис-схема набув нового обов’язкового поля	Відмова замість мовчазної активації неповної конфігурації
5	Тип інтеграції без зареєстрованого опису-схеми	Відмова з повідомленням про відсутній опис
6	Активація шаблону в межах його групи	Рівно один активний шаблон; інші скинуто в одній транзакції
7	Зміна параметра налаштувань поза допустимим діапазоном	Відмова на рівні перевірки вхідних даних
8	Зміна періоду запуску під час роботи	Планувальник перевзводить таймери без перезапуску

№	Сценарій	Очікуваний результат
9	Поле, не оголошене в описі вхідних даних, у тілі запиту	Відмова через сувору перевірку вхідних даних

Тестові сценарії шару збирання наведено в табл. 4.2.

Таблиця 4.2

Тестові сценарії шару збирання

№	Сценарій	Очікуваний результат
10	Окремий адаптер завершується винятком під час збирання	Повертається порожній результат для нього, інші джерела збираються
11	Дублікати записів від різних адаптерів	Усунення дублікатів за складеним ключем
12	Відповідь джерела про перевищення частоти запитів	Повтор запиту після зазначеної затримки
13	Відповідь джерела про відмову в доступі	Інтеграцію автоматично вимкнено
14	Чат із кількістю учасників понад установлену межу	Чат пропущено під час збирання
15	Повідомлення від бота	Пропущено

Тестові сценарії шару перетворення та розсилання наведено в табл. 4.3.

Таблиця 4.3

Тестові сценарії шару перетворення та розсилання

№	Сценарій	Очікуваний результат
16	Запуск формування звіту без активного шаблону	Відмова з повідомленням про відсутній активний шаблон
17	Відсутність налаштованої інтеграції з мовною моделлю	Відмова з повідомленням про відсутність інтеграції
18	Невідомий ідентифікатор моделі у шаблоні під час створення	Відмова з повідомленням про невідому модель
19	Відповідь мовної моделі з обгортковим форматуванням	Обгортку знято, зміст збережено
20	Збій надсилання в один із каналів	Помилку зафіксовано, інші канали отримують звіт
21	Формування звітів у день тижневого підсумку	Тижневий звіт формується послідовно після добового

Наведені сценарії безпосередньо відповідають реалізації, описаній у розділі 3, а їх детермінована природа робить їх придатними для автоматизації.

4.3. Експериментальна перевірка придатності даних для машинного навчання

4.3.1. Постановка експерименту

Кінцевою метою інфраструктури є дані золотої зони, придатні для машинного навчання. Щоб перевірити це кількісно, поставлено задачу багатокласової класифікації — передбачити клас впливу задачі на бізнес (низький, середній або високий) за ознаками, які реально витягуються конвеєром із командної комунікації.

Задачу класифікації обрано тому, що вона безпосередньо відповідає прикладному призначенню системи: золота зона конвеєра містить перелік задач із показниками, а оцінювання впливу задачі на бізнес є типовим запитом, на який має відповідати система підтримки прийняття рішень. Класифікація на впорядковані класи (низький, середній, високий) дозволяє водночас перевірити, чи постачає інфраструктура ознаки, достатні для відтворення не лише наявності сигналу, а й його впорядкованості.

Оскільки реальні дані задач є чутливими (містять персональну та комерційну інформацію), для демонстрації використано синтетичний набір, згенерований під схему системи. Це водночас ілюструє принцип керування даними: модель навчається на знеособлених даних, що відтворюють статистику предметної області, без розкриття реальних повідомлень. Такий підхід узгоджується з вимогами захисту персональних і комерційних даних і є типовою практикою за неможливості використати реальні дані. Експеримент є самодостатнім і відтворюваним: за фіксованого початкового стану генератора та навчання повторний запуск дає ідентичні результати, що дозволяє незалежно перевірити наведені нижче показники.

4.3.2. Набір даних

Генератор є детермінованим (із фіксованим початковим станом) і створює 4000 прикладів із вісьмома ознаками, перелік яких наведено в табл. 4.4.

Таблиця 4.4

Ознаки синтетичного набору даних

Ознака	**Зміст**
Кількість повідомлень	кількість повідомлень, що згадують задачу

Ознака	**Зміст**
Кількість учасників	кількість учасників обговорення
Кількість джерел	кількість різних джерел (від одного до чотирьох)
Середня довжина	середня довжина повідомлення, символів
Блокери	згадки блокерів та залежностей
Пріоритет	оцінка пріоритетних ключових слів (від 0 до 1)
Обсяг коду	обсяг змін коду (0, якщо задача не стосується коду)
Затримка	середня затримка відповіді, годин

Мітку призначено за квантильними порогами (тертилями) латентного балу — лінійної комбінації стандартизованих ознак із доданим гауссовим шумом. Це забезпечує збалансовані класи (приблизно по 1333 приклади) і нетривіальні межі: шум робить задачу такою, що не розв’язується тривіально, тож модель має відновити приховану залежність, а не запам’ятати дані.

4.3.3. Модель і протокол

Як базову обрано модель багатокласової логістичної регресії з регуляризацією, реалізовану самостійно. Теоретичні засади цього методу класифікації викладено в праці [24], а його реалізацію в галузевих бібліотеках машинного навчання описано в роботі [23]. Протокол експерименту передбачав: стратифікований розподіл вибірки у співвідношенні 75 до 25 (2998 прикладів для навчання, 1002 для тестування); стандартизацію ознак за статистикою лише навчальної вибірки (без витоку інформації з тестової); навчання методом градієнтного спуску; порівняння з базовою лінією — прогнозуванням найчастішого класу. Ядро навчання моделі наведено в додатку Б.

4.3.4. Результати

Підсумкові метрики на тестовій вибірці наведено в табл. 4.5.

Таблиця 4.5

Підсумкові метрики моделі на тестовій вибірці

Метрика	**Значення**
Частка правильних відповідей	73,25 %
Базова лінія (найчастіший клас)	33,33 %
Макроусереднена точність	73,33 %
Макроусереднена повнота	73,25 %

Метрика	Значення
Макроусереднена міра F1	73,29 %

Показники класифікації за класами наведено в табл. 4.6.

Таблиця 4.6

Показники класифікації за класами

Клас	Точність	Повнота	Міра F1	Обсяг
Низький	77,18 %	76,95 %	77,06 %	334
Середній	60,65 %	61,38 %	61,01 %	334
Високий	82,18 %	81,44 %	81,80 %	334

Матрицю невідповідностей наведено в табл. 4.7 (рядок — фактичний клас, стовпець — прогнозований).

Таблиця 4.7

Матриця невідповідностей класифікатора

Факт / прогноз	Низький	Середній	Високий
Низький	257	75	2
Середній	72	205	57
Високий	4	58	272

Важливість ознак за модулем коефіцієнтів моделі наведено в табл. 4.8.

Таблиця 4.8

Важливість ознак моделі

Ознака	Вага
Пріоритет	1,236
Кількість учасників	0,866
Обсяг коду	0,675
Кількість джерел	0,458
Затримка	0,426
Блокери	0,362
Кількість повідомлень	0,271
Середня довжина	0,143

4.3.5. Аналіз результатів

Модель суттєво перевершує базову лінію (73,25 % проти 33,33 %), отже у даних наявний відновлюваний сигнал, а інфраструктура постачає інформативні ознаки. Структура помилок є змістовною: низький і високий класи майже не плутаються (лише 2 та 4 випадки), а плутанина зосереджена на суміжних межах із

середнім класом. Це очікувано для порядкової цільової змінної й свідчить, що модель уловила саме впорядкованість впливу, а не випадковий шум. Найскладнішим є середній клас (міра F1 близько 61 %), що типово для серединного класу між двома впорядкованими, межі якого з обох боків розмиті доданим шумом. Важливість ознак відтворює приховану залежність: найвпливовішими є пріоритет, кількість учасників та обсяг коду, що збігається з вагами, закладеними в генератор, і підтверджує коректність як даних, так і навчання.

Окремо варто наголосити на практичному значенні структури матриці невідповідностей. Той факт, що низький і високий класи майже не плутаються між собою, означає, що інфраструктура надійно відрізняє задачі з полярним впливом на бізнес — а саме така відмінність є найважливішою для підтримки прийняття рішень. Помилки ж зосереджені на межах із середнім класом, тобто там, де навіть для людини віднесення задачі до середнього чи суміжного рівня є неоднозначним. Це свідчить, що похибка моделі має переважно «сусідній» характер і рідко призводить до грубих помилок класифікації на дві позиції. Такий профіль помилок є прийнятним для прикладної задачі ранжування задач за впливом і додатково підтверджує, що ознаки, які постачає інфраструктура, несуть впорядкований, а не випадковий сигнал.

4.3.6. Обмеження та напрями подальшої роботи

Демонстрацію виконано на синтетичних даних; на реальних знеособлених задачах точність очікувано буде дещо нижчою через додаткові джерела шуму, як-от неоднорідність стилю спілкування, неповноту окремих ознак і зміщення розподілу з часом. Базову лінійну модель обрано навмисно простою, щоб результат відображав саме якість ознак, а не потужність моделі; застосування нелінійних методів (зокрема ансамблевих) імовірно підвищить якість, особливо для середнього класу, межі якого розмиті. Окремим напрямом є розширення набору ознак за рахунок глибших характеристик комунікації, що їх здатна виокремити мовна модель.

Другий вузол машинного навчання у системі — семантичне стиснення мовною моделлю — у цій роботі оцінено якісно, за зв’язністю й повнотою зведення. Його кількісне оцінювання є окремою методичною задачею, оскільки вихід мовної моделі не має єдиної правильної відповіді; її розв’язання потребує формування еталонних анотацій та застосування метрик відповідності, що винесено в напрями подальшої роботи. Загалом проведений експеримент підтверджує, що спроєктована й реалізована інфраструктура виконує своє призначення: вона доводить неструктуровані дані до стану, у якому з них відновлюється змістовний, придатний для машинного навчання сигнал.

ВИСНОВКИ

У роботі розв’язано задачу проєктування та реалізації програмної системи інфраструктури інженерії даних для машинного навчання, що забезпечує наскрізний цикл збирання, накопичення, керування та перетворення даних. Отримано такі основні результати.

Проаналізовано предметну область інженерії даних: розглянуто даноцентричну парадигму, життєвий цикл даних, парадигми перетворення даних, архітектури сховищ і медальйонну модель, засоби оркестрації, керування якістю та версіонування даних, а також сховища ознак. Оглянуто наявні рішення — інфраструктурні інструменти та платформи інженерної аналітики з метричними рамками доставки — і виявлено незаповнену нішу: автоматизоване опрацювання неструктурованої командної комунікації, у яку спрямовано розроблену систему.

Сформульовано вимоги та спроєктовано архітектуру системи: визначено одинадцять функціональних і сім нефункціональних вимог, обрано архітектурний стиль модульного моноліту, побудовано модель даних із семи сутностей, граф залежностей компонентів, діаграму послідовності конвеєра та проєкцію потоку даних на медальйонну архітектуру, обґрунтовано технологічний стек і свідомі спрощення відносно галузевих стандартів.

Реалізовано всі чотири процеси теми. Шар збирання побудовано на розширюваному контракті адаптерів зі стійкістю до збоїв джерел, з повноцінним адаптером джерела Telegram. Накопичення реалізовано на реляційному сховищі з версіонуванням результатів. Шар керування охоплює конфігурацію як керований стан із застосуванням змін без перезапуску, перевірку за описом-схемою, захищене зберігання облікових даних і атомарне керування шаблонами. Шар перетворення реалізує медальйонну агрегацію трьох рівнів із залученням мовної моделі.

Експериментально підтверджено придатність отриманих даних для машинного навчання: на синтетичному наборі під схему системи базова модель класифікації впливу задач досягла частки правильних відповідей 73,25 % і макроусередненої міри F1 73,29 % проти базової лінії 33,33 %, з осмисленою

структурою помилок та важливістю ознак, що відповідає закладеній залежності. Сформульовано трирівневу методику тестування й план із двадцяти одного детермінованого сценарію перевірки інваріантів системи.

Спроєктована й реалізована система виконує своє призначення — доводить різнорідні неструктуровані дані до стану, придатного для машинного навчання, — і задовольняє всі сформульовані вимоги. Мету роботи досягнуто, поставлені завдання виконано.

Напрямами подальшої роботи є реалізація автоматизованого набору тестів, застосування нелінійних методів навчання для підвищення якості класифікації, кількісне оцінювання семантичного стиснення мовною моделлю за еталонними анотаціями та розширення переліку джерел-адаптерів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Reis J., Housley M. Fundamentals of Data Engineering: Plan and Build Robust Data Systems. Sebastopol : O'Reilly Media, 2022. 447 p.
2. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 590 p.
3. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps. Portland : IT Revolution Press, 2018. 288 p.
4. Forsgren N., Storey M.-A., Maddila C., Zimmermann T., Houck B., Butler J. The SPACE of Developer Productivity. ACM Queue. 2021. Vol. 19, № 1. P. 20–48.
5. Ng A. Data-Centric AI: Real World Challenges and Research Opportunities. Stanford : Stanford University, 2022. URL: https://datacentricai.org (дата звернення: 02.06.2026).
6. Zaharia M., Ghodsi A., Xin R., Armbrust M. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. Proceedings of CIDR. 2021. 8 p.
7. Medallion Architecture: Bronze, Silver, and Gold Layers. Databricks Documentation. URL: https://docs.databricks.com/lakehouse/medallion.html (дата звернення: 02.06.2026).
8. Apache Airflow Documentation. The Apache Software Foundation. URL: https://airflow.apache.org/docs/ (дата звернення: 03.06.2026).
9. Dagster Documentation: The Data Orchestration Platform. Dagster Labs. URL: https://docs.dagster.io (дата звернення: 03.06.2026).
10. Prefect Documentation. Prefect Technologies. URL: https://docs.prefect.io (дата звернення: 03.06.2026).
11. dbt (data build tool) Documentation. dbt Labs. URL: https://docs.getdbt.com (дата звернення: 03.06.2026).
12. Feast: The Open Source Feature Store for Machine Learning. URL: https://docs.feast.dev (дата звернення: 03.06.2026).

13. Great Expectations Documentation: Data Quality Validation. URL: https://docs.greatexpectations.io (дата звернення: 04.06.2026).
14. Data Version Control (DVC) Documentation. Iterative. URL: https://dvc.org/doc (дата звернення: 04.06.2026).
15. lakeFS Documentation: Data Version Control. Treeverse. URL: https://docs.lakefs.io (дата звернення: 04.06.2026).
16. BullMQ: Premium Message Queue for NodeJS. URL: https://docs.bullmq.io (дата звернення: 04.06.2026).
17. Temporal Documentation: Durable Execution Platform. URL: https://docs.temporal.io (дата звернення: 04.06.2026).
18. OpenRouter API Documentation. URL: https://openrouter.ai/docs (дата звернення: 05.06.2026).
19. NestJS Documentation: A Progressive Node.js Framework. URL: https://docs.nestjs.com (дата звернення: 05.06.2026).
20. Prisma ORM Documentation. Prisma Data. URL: https://www.prisma.io/docs (дата звернення: 05.06.2026).
21. PostgreSQL 18 Documentation. The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/ (дата звернення: 05.06.2026).
22. Redis Documentation. Redis Ltd. URL: https://redis.io/docs/ (дата звернення: 05.06.2026).
23. Pedregosa F. et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research. 2011. Vol. 12. P. 2825–2830.
24. Bishop C. M. Pattern Recognition and Machine Learning. New York : Springer, 2006. 738 p.
25. Telegram API and MTProto Protocol Documentation. Telegram Messenger. URL: https://core.telegram.org/api (дата звернення: 06.06.2026).
26. Docker Compose Documentation. Docker Inc. URL: https://docs.docker.com/compose/ (дата звернення: 06.06.2026).

ДОДАТОК А. ФРАГМЕНТИ ПРОГРАМНОГО КОДУ СИСТЕМИ

У додатку наведено ключові фрагменти програмного коду системи мовою TypeScript, що ілюструють реалізаційні рішення, описані в розділі 3: контракт адаптера джерела зі стійкістю до збоїв, збирання з усуненням дублікатів, перевірку конфігурації за описом-схемою та планувальник на вирівняних таймерах.

Лістинг А.1. Контракт адаптера джерела та HTTP-обгортка зі стійкістю до збоїв

```
// Контракт адаптера джерела: тип джерела і метод отримання зрізу від моменту часу.
export interface Adapter {
  readonly type: IntegrationType;
  recent(frame: number, options?: Settings.Entity): Promise<Bundle>;
}


// Виняток автентифікації (HTTP 401/403). Джерело-викликач має вимкнути
// проблемну інтеграцію, а не прокидати помилку далі за конвеєром.
export class AuthError extends Error {
  constructor(public readonly status: number, public readonly url: string) {
    super(`AUTH_${status}:${url}`);
  }
}


// HTTP-запит з автоматичним урахуванням заголовка Retry-After на код 429;
// коди 401/403 → AuthError; інші не-2xx відповіді → помилка.
export async function fetchJson<T>(
  url: string, headers: Record<string, string>
): Promise<T> {
  const response = await fetch(url, { headers });
  if (response.status === 429) {
    const retry = Number(response.headers.get('retry-after') ?? '1');
    await sleep(retry * Constraints.Time.SECOND);
    return fetchJson<T>(url, headers);
  }
  if (response.status === 401 || response.status === 403) {
    throw new AuthError(response.status, url);
  }
  if (!response.ok) throw new Error(`HTTP_${response.status}:${url}`);
  return (await response.json()) as T;
}
```

Лістинг А.2. Збирання зрізу від усіх адаптерів з усуненням дублікатів

```
// Опитує кожен адаптер паралельно, склеює зрізи та усуває дублікати;
// падіння одного адаптера не зриває решту — він повертає порожній зріз.
async gather(frame: number, options?: Settings.Entity): Promise<Bundle> {
  const bundles = await Promise.all(
    this.adapters.map(adapter =>
      adapter.recent(frame, options).catch((error) => {
        Service.logger.error(`GATHER_FAILED:${adapter.type}:${error}`);
        return { users: [], chats: [], messages: [] } as Bundle;
      })
    )
  );
  const users = bundles.flatMap(b => b.users);
  const chats = bundles.flatMap(b => b.chats);
  const messages = bundles.flatMap(b => b.messages);
  return {
    users: Mutator.deduplicateBy(users, i => `${i.source}:${i.id}`),
```

```
    chats: Mutator.deduplicateBy(chats, i => `${i.source}:${i.id}`),
    messages: Mutator.deduplicateBy(messages, i => `${i.source}:${i.chatId}:${i.id}`)
      .sort((a, b) => a.timestamp.localeCompare(b.timestamp)),
  };
}

// Узагальнене усунення дублікатів за складеним ключем (поле або
// функція-екстрактор) зі збереженням порядку елементів.
export const deduplicateBy = <T>(
  arr: readonly T[], key: keyof T | ((item: T) => unknown)
): T[] => {
  const extract = typeof key === 'function' ? key : (item: T) => item[key];
  const seen = new Set<unknown>();
  const result: T[] = [];
  for (const item of arr) {
    const k = extract(item);
    if (seen.has(k)) continue;
    seen.add(k);
    result.push(item);
  }
  return result;
};
```

Лістинг А.3. Перевірка конфігурації інтеграції за описом-схемою

```
// Перевіряє об’єкт налаштувань за описом-схемою, зареєстрованим для типу
// інтеграції: наявність обов’язкових полів, тип і відповідність регулярному виразу.
async validate(type: IntegrationType, settings: Record<string, unknown>): Promise<void> {
  const blueprint = await this.findByType(type);
  if (!blueprint)
    throw new ConflictException(`MISSING_BLUEPRINT_CONFIGURATION:${type}`);
  for (const s of blueprint.schemas) {
    const value = settings[s.key];
    if (value === undefined || value === null) {
      if (s.optional) continue;
      throw new BadRequestException(`SETTINGS_MISSING:${s.key}`);
    }
    if (!matches(value, s.type))
      throw new BadRequestException(`SETTINGS_TYPE:${s.key}:${s.type}`);
    if (s.regex && !new RegExp(s.regex).test(value as string))
      throw new BadRequestException(`SETTINGS_REGEX:${s.key}`);
  }
}

const matches = (value: unknown, type: SchemaType): boolean => {
  if (type === SchemaType.string) return typeof value === 'string';
  if (type === SchemaType.number) return typeof value === 'number';
  if (type === SchemaType.boolean) return typeof value === 'boolean';
  return false;
};
```

Лістинг А.4. Планувальник на вирівняних таймерах без накопичення похибки

```
// Час (мс) до наступного слоту intervalMs, вирівняного на 00:00 UTC.
// Кожне спрацювання обчислює наступний слот наново — без накопичення похибки.
private static align(intervalMs: number): number {
  const now = Date.now();
  const dayStart = Math.floor(now / Constraints.Time.DAY) * Constraints.Time.DAY;
  const elapsed = now - dayStart;
  const slots = Math.floor(elapsed / intervalMs) + 1;
  return dayStart + slots * intervalMs - now;
}

// Час (мс) до 00:00 UTC + reportTimeMinutes; якщо слот доби вже минув — наступна доба.
```

```
private static next(reportTimeMinutes: number): number {
  const now = new Date();
  const utcMidnight = Date.UTC(now.getUTCFullYear(), now.getUTCMonth(), now.getUTCDate());
  let target = utcMidnight + reportTimeMinutes * Constraints.Time.MINUTE;
  if (target <= now.getTime()) target += Constraints.Time.DAY;
  return target - now.getTime();
}

// Послідовне виконання прогонів: повільний раунд не накладається на добовий цикл.
private async serialize(fn: () => Promise<void>) {
  const prev = this.running;
  this.running = prev.then(fn, fn);
  await this.running;
}
```

ДОДАТОК Б. ЛІСТИНГ ЕКСПЕРИМЕНТУ З МАШИННОГО НАВЧАННЯ

У додатку наведено ядро відтворюваного експерименту з розділу 4 — реалізовану з нуля багатокласову логістичну регресію (softmax) з L2-регуляризацією, навчану методом градієнтного спуску. Експеримент виконується середовищем Node.js без зовнішніх залежностей; за фіксованого початкового стану генератора повторний запуск дає наведені в підрозділі 4.3 показники.

Лістинг Б.1. Ядро навчання багатокласової логістичної регресії (softmax)

```
function softmax(z) {
  const m = Math.max(...z);
  const e = z.map(v => Math.exp(v - m));
  const s = e.reduce((a, v) => a + v, 0);
  return e.map(v => v / s);
}
function logits(x) {
  return Wt.map((wk, k) => x.reduce((a, v, j) => a + v * wk[j], b[k]));
}

// Градієнтний спуск з L2-регуляризацією (повний батч на 400 епох).
for (let epoch = 0; epoch < EPOCHS; epoch++) {
  const gW = Array.from({ length: K }, () => new Array(D).fill(0));
  const gb = new Array(K).fill(0);
  for (let i = 0; i < Ntr; i++) {
    const p = softmax(logits(Xtr[i]));
    for (let k = 0; k < K; k++) {
      const err = p[k] - (Ytr[i] === k ? 1 : 0);
      gb[k] += err;
      for (let j = 0; j < D; j++) gW[k][j] += err * Xtr[i][j];
    }
  }
  for (let k = 0; k < K; k++) {
    b[k] -= (LR * gb[k]) / Ntr;
    for (let j = 0; j < D; j++) {
      gW[k][j] = gW[k][j] / Ntr + L2 * Wt[k][j];
      Wt[k][j] -= LR * gW[k][j];
    }
  }
}
```