**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА**

**Факультет інформаційних технологій**

Кафедра технологій управління

Спеціальність 122 – Комп'ютерні науки,

освітня програма «Інформаційна аналітика та впливи»

# КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему:

# «Інформаційна система автоматизованого відновлення процесів транспортування даних»

**Студента 2-го курсу групи ІАВ-21**

Федосенко Олександр Дмитрович

(прізвище, ім'я, по батькові)

(підпис студента) (дата)

**Науковий керівник:**

Доктор технічних наук, доцент, доцент кафедри технологій управління Заріцький О.В.

(науковий ступінь, вчене звання, прізвище, ім'я, по батькові)

(підпис) (дата)

**Попередній захист:**

(Висновок: «До захисту в Екзаменаційній комісії»)

Завідувач кафедри технологій управління

(підпис) (прізвище, ініціали) (дата)

**Київ – 2026**

**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА**

**Факультет інформаційних технологій**

Кафедра технологій управління
Освітньо-кваліфікаційний рівень Магістр
Спеціальність 122 - Комп'ютерні науки
Освітня програма Інформаційна аналітика та впливи

**ЗАТВЕРДЖУЮ**

Завідувач кафедри

«___» _____ 2026 року

**З А В Д А Н Н Я**

**НА ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ**

Студент Федосенко Олександр Дмитрович Група ІАВ-21

**1. Тема кваліфікаційної роботи**

Інформаційна система автоматизованого відновлення процесів транспортування даних

Затверджена наказом по університету від «___» _____ 2026 р. № ___.

**2. Строк подання студентом готової роботи – «___» _____ 2026 р.**

**3. Цільова установка та вихідні дані до роботи**

Розробка моделей та інформаційної технології автоматичного відновлення ETL-пайплайнів, що зазнали збою внаслідок дрейфу схеми даних (schema drift), із застосуванням великих мовних моделей (LLM). Система реалізує повний цикл самовідновлення: виявлення збою через механізм on_failure_callback Apache Airflow, збирання контексту з шести гетерогенних джерел, LLM-діагностування з використанням structured output, безпечне виконання виправлень за принципом defense in depth та human-in-the-loop підтвердження через веб-інтерфейс. Вхідні дані: ETL-інфраструктура бази практики - пайплайни інтеграції рекламних даних (TikTok Ads API, X Ads API), оркестровані Apache Airflow 3.0; обчислювальне

середовище AWS Glue Python Shell; аналітичне сховище Amazon Redshift Serverless. Експериментальна вибірка: 140 прогонів (4 сценарії schema drift × 7 LLM-моделей від провайдерів Anthropic, OpenAI та Google × 5 повторів).

**4. Зміст роботи**

Робота містить аналіз предметної області ETL-пайплайнів та таксономії їх збоїв, огляд існуючих підходів до обробки помилок (ручне виправлення, retry-стратегії, rule-based валідація, schema evolution, платформи data observability), а також аналіз застосування LLM у задачах Automated Program Repair. Розроблено чотирифазну модель процесу самовідновлення, метод збирання контексту помилки з алгоритмом schema diff, модель LLM-діагностування на основі Pydantic-схеми DiagnosisOutput та модель безпечного виконання виправлень. Описано програмну реалізацію системи (~1 700 рядків Python, 80 модульних тестів, покриття 87%), веб-інтерфейс FastAPI + Streamlit та інтеграцію з Apache Airflow, AWS Glue, Amazon Redshift Serverless і GitHub. Представлено результати експериментального дослідження, подано висновки та обґрунтування економічного ефекту.

Дипломна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг – 109 сторінок, кількість джерел – 40.

**5. Перелік графічного матеріалу (слайдів)**

Архітектура ETL-інфраструктури бази практики, таксономія збоїв ETL-процесів, матриця порівняння існуючих підходів, чотирифазна модель самовідновлення, алгоритм schema diff, Pydantic-схема DiagnosisOutput, архітектура програмної реалізації, результати порівняння LLM-моделей, техніко-економічне обґрунтування.

## 6. Календарний план виконання роботи:

| № п/п | Назва частин роботи | % | Виконання роботи | |
|---|---|---|---|---|
| | | | За планом | Фактично |
| 1. | Вибір теми дипломної роботи | 3 | 19.12.25 | 19.12.25 |
| 2. | Протокол кафедри ТУ про затвердження тем дипломних робіт та призначення наукових керівників | 2 | 27.12.25 | 27.12.25 |
| 3. | Формування переліку нормативних матеріалів, літератури з проблематики дипломної роботи | 10 | 08.01.26 | 07.01.26 |
| 4. | Складання розгорнутого плану кваліфікаційної роботи | 5 | 18.01.26 | 18.01.26 |
| 5. | Ознайомлення наукового керівника з розгорнутим планом кваліфікаційної роботи. Внесення змін. | 5 | 19.01.26 - 20.01.26 | 20.01.26 |
| 6. | Підготовка розділу 1 | 10 | 12.02.26 | 13.02.26 |
| 7. | Підготовка розділу 2 | 14 | 08.03.26 | 08.03.26 |
| 8. | Підготовка розділу 3 | 14 | 01.04.26 | 01.04.26 |
| 9. | Підготовка розділу 4 | 13 | 20.04.26 | 20.04.26 |
| 10. | Оформлення кваліфікаційної роботи. Підготовка висновків і пропозицій | 15 | 30.04.26 | 30.04.26 |
| 11. | Передача кваліфікаційної роботи науковому керівникові | 2 | 04.05.26 | 04.05.26 |
| 12. | Передача кваліфікаційної роботи рецензенту для рецензування | 2 | 07.05.26 | 11.05.26 |
| 13. | Попередній захист кваліфікаційної роботи | 5 | 11.05.26 | 17.05.26 |

Дата видачі завдання «___» _____ 20__ р.

Керівник роботи Доктор наук, доцент кафедри технологій управління Заріцький О.В.

_____

(підпис)

Завдання прийняв до виконання студент групи ІАВ-21 Федосенко Олександр Дмитрович

_____

(підпис)

# ЗМІСТ

# АНОТАЦІЯ

## КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

**Факультет інформаційних технологій**

Кафедра технологій управління

Спеціальність 122 - Комп'ютерні науки,

освітня програма "Інформаційна аналітика та впливи"

Тема кваліфікаційної роботи магістра *Федосенко Олександра Дмитровича*: «Інформаційна система автоматизованого відновлення процесів транспортування даних».

**Мета**: зменшення часу простою та підвищення надійності в обробці даних шляхом автоматичного відновлення ETL-пайплайнів за допомогою застосування великих мовних моделей.

**Об'єкт дослідження** - процеси автоматичного відновлення ETL-пайплайнів.

**Предмет дослідження** - моделі та методи побудови систем автоматичного діагностування й виправлення помилок у ETL-скриптах.

**Наукова новизна одержаних результатів**. Удосконалено модель автоматичного відновлення ETL-пайплайнів, яка завдяки поєднанню детерміністичного аналізу схеми з LLM-діагностуванням забезпечує повний цикл самовідновлення. Уперше запропоновано модель людино-машинної взаємодії для систем самовідновлення ETL-пайплайнів за принципом human-in-the-loop. Дістав подальшого розвитку метод збирання контексту помилки з шести гетерогенних джерел.

**Практичну цінність** роботи підтверджено впровадженням системи на базі практики для відновлення ETL-пайплайнів інтеграції рекламних даних. Час повного циклу репарації становить 11-40 секунд проти 30-90 хвилин при ручному відновленні; прогнозований річний економічний ефект близько 26 тис. USD. Запропоновано чотирифазну модель самовідновлення з безпечним

виконанням виправлень за принципом defense in depth. Реалізація інтегрована з Apache Airflow, AWS Glue та Amazon Redshift Serverless. На вибірці зі 140 прогонів (чотири сценарії дрейфу, сім LLM-моделей) точність діагностування становить 87,1 %, частка end-to-end відновлення 80 %.

**Ключові слова:** ETL-пайплайн, дрейф схеми даних, великі мовні моделі, самовідновлення, Apache Airflow, Amazon Redshift, structured output, human-in-the-loop, інженерія даних.

# ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

**API –** Application Programming Interface – програмний інтерфейс застосунку

**APR –** Automated Program Repair – автоматичне виправлення програм

**AWS –** Amazon Web Services – хмарна платформа Amazon

**DAG –** Directed Acyclic Graph – направлений ациклічний граф

**DDL –** Data Definition Language – мова визначення даних

**ETL –** Extract, Transform, Load – вилучення, перетворення, завантаження

**ELT –** Extract, Load, Transform – вилучення, завантаження, перетворення

**FastAPI –** веб-фреймворк для Python

**IAM –** Identity and Access Management – управління ідентифікацією та доступом

**JSON –** JavaScript Object Notation – формат обміну даними

**LLM –** Large Language Model – велика мовна модель

**MTTD –** Mean Time To Detect – середній час виявлення проблеми

**MTTR –** Mean Time To Repair – середній час відновлення

**PR –** Pull Request – запит на злиття змін коду

**S3 –** Amazon Simple Storage Service – об'єктне сховище Amazon

**SQL –** Structured Query Language – мова структурованих запитів

**SWE –** Software Engineering – програмна інженерія

# ВСТУП

В умовах аналітичного підходу до розробки програмного забезпечення та розширення використання моделей штучного інтелекту в усіх сферах, дані - це база для управлінських рішень, це ґрунт для створення нових продуктів і робочих процесів, таких як рекомендаційні моделі і таргетований маркетинг. Аналітична компанія IDC прогнозує щорічний приріст світового обсягу даних на рівні 23–25%, а до 2025 року сукупний обсяг має сягнути 175 зетабайт [1]. Зростає не лише сам обсяг. Збільшується й кількість джерел, з яких аналітичні відділи отримують інформацію. До звичних корпоративних баз і внутрішніх систем додалися API рекламних платформ, CRM, соціальні мережі.

У такій інфраструктурі ETL-пайплайни (Extract, Transform, Load) виконують роль сполучної ланки. Це автоматизовані процеси вилучення, перетворення й завантаження даних. Коли пайплайн зупиняється, аналітичні показники перестають оновлюватися. Разом з ними зупиняється й уся залежна робота: аналітика, формування звітності, прогнозні моделі, прийняття рішень. За оцінками Gartner, година простою дата-інфраструктури коштує підприємству від 5 600 до 9 000 доларів США [3].

Серед причин таких збоїв окреме місце посідає schema drift, тобто дрейф схеми даних. Йдеться про ситуацію, коли структура даних на стороні джерела змінюється у спосіб, для якого приймальна інфраструктура не була підготовлена. Виникає конфлікт. Пайплайн зупиняється. У звіті Monte Carlo Data, побудованому на опитуванні команд data engineering, 67% респондентів віднесли schema drift до трьох найчастіших причин інцидентів [5].

Цю проблему розв'язують по-різному. Хтось виправляє вручну, хтось налаштовує retry-стратегії, хтось будує валідацію за правилами. Існують механізми schema evolution у Data Lake та спеціалізовані платформи data observability. Проте жоден із цих підходів не закриває повного циклу - від

виявлення помилки до її кореневого діагностування і подальшого автономного виправлення.

А економічний потенціал такої автоматизації значний. За даними Fivetran, до 40% робочого часу інженерів даних йде на обслуговування вже наявних пайплайнів [8]. Це фактично два дні з кожного робочого тижня. Це збігається з досвідом на базі практики – помітно, що робота над таким типом задач складає приблизно половину завантаження, також ці задачі часто потребують координації між декількома членами команди і забирають час від роботи над довгостроковими проектами.

З цих причин, успішна автоматизація спектру задач відновлення процесів транспортування даних може дозволити суттєво зменшити необхідну кількість працівників у команді.

Прогрес у застосуванні великих мовних моделей (Large Language Models, LLM) до задач автоматичного виправлення коду створює нові можливості для розв'язання цієї проблеми. На бенчмарку SWE-bench провідні агентні системи демонструють здатність розв'язувати від 33% до 40% реальних інженерних задач [17, 18]. Сучасні LLM здатні видавати повноцінні діагностичні висновки, придатні для подальшого практичного використання.

**Актуальність теми** зумовлена одночасним поєднанням двох чинників. По-перше - збільшенням навантаження в процесах транспортування даних і збільшенням їх критичності в роботі, що створює потребу у автоматизації цих процесів. По-друге - прогресом у розвитку великих мовних моделей що створює можливість їх застосування для самовідновлення ETL-пайплайнів. Завдяки цим чинникам стає необхідним практичне дослідження можливостей LLM у задачах автоматизації процесів інженерії даних.

**Зв'язок роботи з науковими програмами, планами, темами.** Кваліфікаційну роботу магістра виконано на кафедрі технологій управління факультету інформаційних технологій Київського національного університету імені Тараса Шевченка відповідно до освітньо-наукової програми «Інформаційна аналітика та впливи» спеціальності 122 «Комп'ютерні науки». Тематика

дослідження узгоджується з науковим напрямом кафедри щодо розробки моделей та інформаційних технологій аналізу даних. Вона також відповідає сучасним тенденціям розвитку штучного інтелекту у сфері автоматизації процесів розробки програмного забезпечення.

**Мета дослідження** - зменшення часу простою та підвищення надійності в обробці даних шляхом автоматичного відновлення ETL-пайплайнів за допомогою застосування великих мовних моделей.

**Завдання дослідження.** Для досягнення поставленої мети у роботі визначено такі завдання:

- проаналізувати предметну область автоматизації процесів розробки програмного забезпечення, а також наявні підходи до автоматизації ETL-процесів;

- розробити модель процесу самовідновлення ETL-пайплайну;

- провести експериментальне дослідження ефективності запропонованої системи.

**Об'єктом дослідження** є процеси забезпечення автоматичного відновлення ETL-пайплайнів.

**Предмет дослідження** становлять моделі та методи побудови систем автоматичного діагностування й виправлення помилок у ETL-скриптах.

**Методи дослідження.** У роботі застосовано низку методів наукового дослідження. Системний аналіз використано для дослідження принципів функціонування ETL-пайплайнів і самого процесу самовідновлення. Порівняльний аналіз для оцінювання існуючих підходів та зіставленню великих мовних моделей трьох провайдерів. Моделювання застосовувалося при побудові моделей процесу збирання контексту, діагностування й безпечного виконання виправлень. Експеримент дав можливість оцінити ефективність розробленої системи на тестових сценаріях. Опрацювання отриманих результатів виконано за допомогою методів статистичного аналізу.

**Наукова новизна одержаних результатів.** У ході виконаного дослідження отримано такі наукові результати.

Удосконалено модель автоматичного відновлення ETL-пайплайнів. На відміну від наявних підходів, вона забезпечує повний цикл самовідновлення: виявлення збою, його діагностування, формування й застосування виправлення.

Дістав подальшого розвитку метод збирання контексту помилки з множинних джерел для задач діагностування збоїв процесів. Особливість запропонованого варіанту полягає в агрегації сигналів із шести гетерогенних джерел.

Дістав подальшого розвитку підхід до застосування великих мовних моделей для діагностування помилок у програмному забезпеченні. На відміну від існуючих робіт у галузі Automated Program Repair, цей підхід спеціалізовано саме для домену data engineering.

Удосконалено модель безпечного виконання автоматично згенерованих виправлень. Запропонована модель забезпечує багаторівневу валідацію: фільтрацію SQL-команд за whitelist дозволених операцій, перевірку синтаксису, а також триступеневу процедуру застосування змін у коді. Завдяки цьому досягається негайне підхоплення виправлення зі збереженням повного аудиторського сліду.

Уперше запропоновано модель людино-машинної взаємодії для систем самовідновлення ETL-пайплайнів у режимі human-in-the-loop. Модель реалізовано у вигляді веб-інтерфейсу, який надає оператору можливість переглядати події виправлення, підтверджувати чи відхиляти окремі виправлення, ініціювати повторне діагностування з додатковим контекстом, а також аналізувати історію діагнозів.

**Практичне значення одержаних результатів.** Результати мають прикладну спрямованість. Вони орієнтовані на розв'язання конкретної виробничої задачі - скорочення часу відновлення ETL-пайплайнів після збоїв, спричинених дрейфом схеми даних. Систему впроваджено на базі практики для автоматичного відновлення пайплайнів інтеграції рекламних даних.

Практична цінність розробленого рішення проявляється у кількох вимірах. Час відновлення після збою скорочується з годин при ручному виправленні до

секунд при автоматичному. Знижується навантаження на інженерів даних завдяки автоматизації рутинних операцій. Підвищується надійність аналітичної інфраструктури в цілому. Розроблене програмне забезпечення є відкритим, тож за потреби його можна адаптувати для роботи з іншими джерелами даних, оркестраторами та аналітичними сховищами.

# РОЗДІЛ 1
# АНАЛІЗ ПРОБЛЕМИ АВТОМАТИЧНОГО ВІДНОВЛЕННЯ ETL-ПАЙПЛАЙНІВ

## 1.1 Аналіз предметної області та об’єкта дослідження

Зростання аналітичних потреб бізнесу разом із розвитком маркетингових інструментів призводить до швидкого збільшення обсягу даних, які підприємства використовують для ухвалення управлінських рішень. Окрім цього, розширюється й перелік джерел, з яких ці дані надходять. До звичних корпоративних баз і внутрішніх систем додаються зовнішні джерела, серед яких API рекламних платформ, CRM-системи, IoT-пристрої, соціальні мережі. У таких умовах забезпечення безперервного потоку даних від джерела до аналітичного сховища стає однією з критичних задач для бізнесу.

Центральним інструментом інтеграції даних виступають ETL-пайплайни (Extract, Transform, Load), тобто автоматизовані процеси вилучення, перетворення та завантаження даних [2]. За оцінкою Gartner, година простою дата-інфраструктури коштує підприємству від 5 600 до 9 000 доларів США [3]. Саме тому відновлення ETL-процесів після збоїв є пріоритетною виробничою задачею.

### *1.1.1 Сучасний ландшафт ETL/ELT-систем*

Сучасні системи транспортування даних мають модульну архітектуру з чотирма рівнями: джерела даних, шар прийому (ingestion), шар обробки (processing) та шар зберігання (storage). На кожному з цих рівнів представлений широкий перелік як комерційних, так і відкритих рішень, які можуть комбінуватися у довільні конфігурації. Узагальнений ландшафт відповідних технологій наведено на рисунку 1.1.

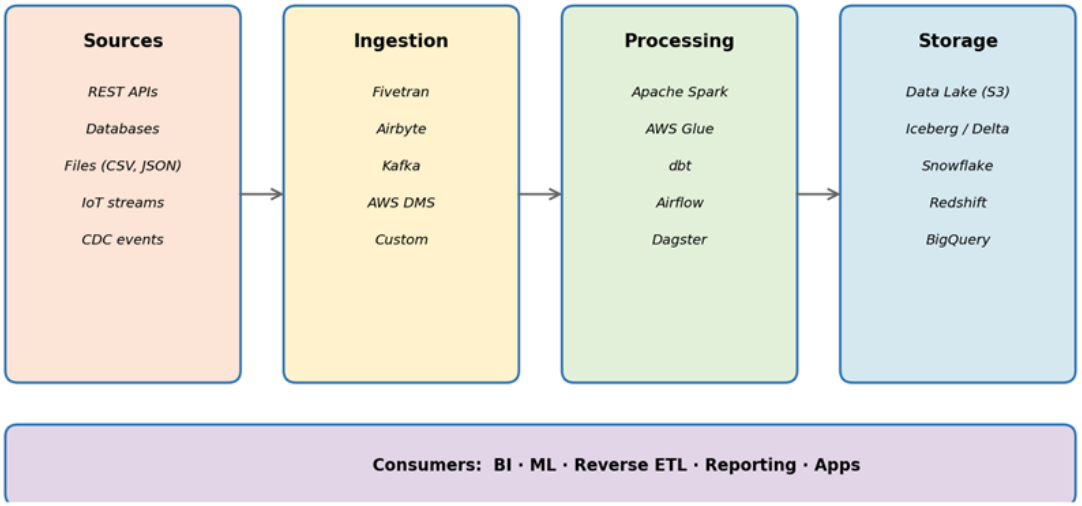


Рисунок 1.1 - Ландшафт сучасних ETL/ELT-систем

Принциповою тенденцією останнього десятиліття є зсув від класичної парадигми ETL (Extract, Transform, Load) до парадигми ELT (Extract → Load → Transform). За такою схемою трансформація виконується безпосередньо в аналітичному сховищі або на рівні Data Lake. Зумовлений цей перехід двома чинниками. Перший, це зростання обчислювальних можливостей хмарних сховищ, серед яких Snowflake, BigQuery, Redshift. Другий, це зниження вартості зберігання даних в об’єктних сховищах на кшталт S3, GCS, Azure Blob. Незалежно від обраної парадигми, ключовим елементом усієї системи залишається оркестратор, тобто програмна платформа, що координує запуск задач, відстежує їхній стан та обробляє помилки.

### *1.1.2 Архітектура ETL-пайплайнів бази практики*

Об’єктом дослідження виступають процеси забезпечення відмовостійкості ETL-пайплайнів, оркестрованих Apache Airflow у хмарній інфраструктурі Amazon Web Services. Базою практики слугує система інтеграції рекламних даних, що отримує інформацію з TikTok Ads API, X (Twitter) Ads API, а також з кількох внутрішніх систем, і завантажує її до аналітичного сховища Amazon Redshift Serverless.

Архітектура наявних ETL-пайплайнів бази практики складається з кількох основних компонентів. У ролі оркестратора використано Apache Airflow версії 3.0, розгорнутий у Docker-контейнерах із CeleryExecutor; як брокер повідомлень працює Redis, а метадані зберігаються у PostgreSQL. Обчислювальним середовищем для виконання ETL-скриптів виступає AWS Glue Python Shell. Проміжним сховищем у форматі Parquet слугує Amazon S3, тоді як цільовим аналітичним сховищем є Amazon Redshift Serverless. Загальну архітектуру наведено на рисунку 1.2.

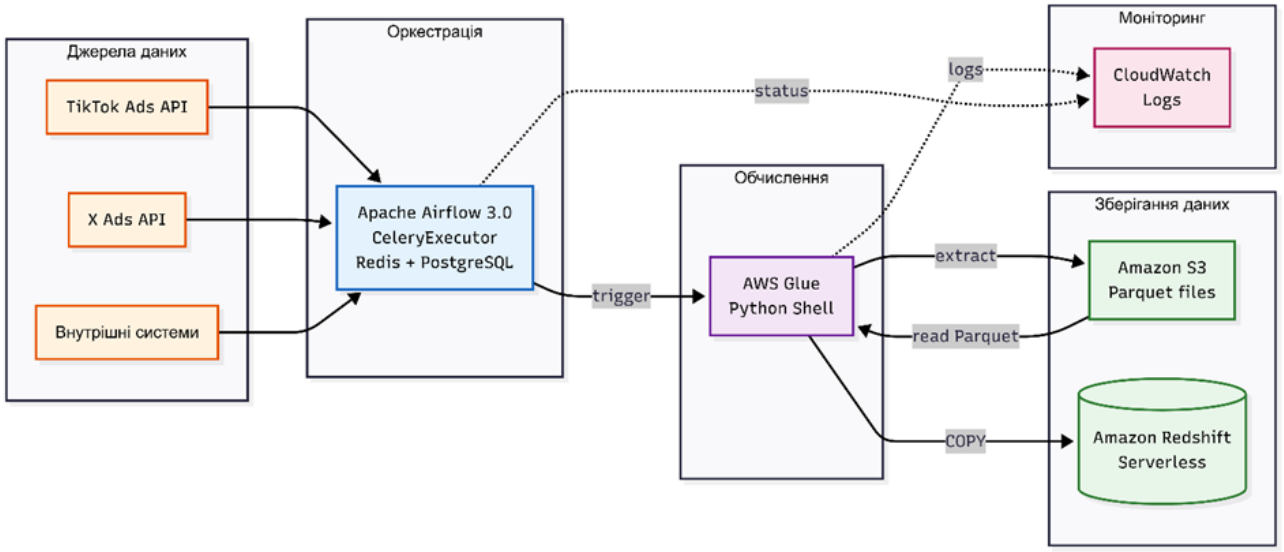


Рисунок 1.2 - Архітектура ETL-інфраструктури бази практики

Оркестрація пайплайнів покладена на Apache Airflow, який забезпечує планування запусків, моніторинг стану задач та обробку помилок. Кожен DAG реалізовано за двозадачним патерном. Перша задача виконує RedshiftDataOperator для створення цільової таблиці. Друга запускає GlueJobOperator, який стартує власне ETL-скрипт. Завантаження даних відбувається через AWS Glue Python Shell, що виконує Python-скрипти з використанням бібліотек pandas для трансформацій та awswrangler для взаємодії з Redshift. Проміжні дані зберігаються у форматі Parquet в Amazon S3 із партиціюванням за датою.

### *1.1.3 Таксономія збоїв у ETL-процесах*

Під час експлуатації ETL-пайплайнів виникають збої різної природи. На основі аналізу галузевих звітів і власного досвіду роботи з пайплайнами бази практики сформульовано таксономію, що поділяє ці збої на п’ять основних категорій. Структуру таксономії показано на рисунку 1.3.

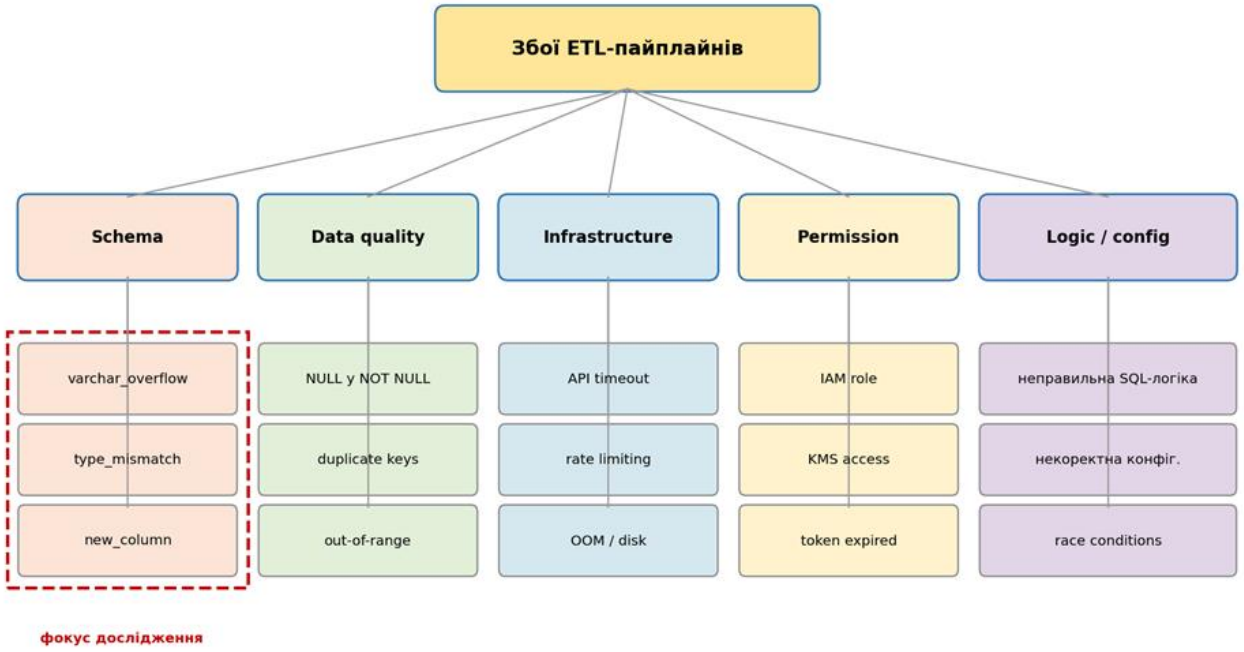


Рисунок 1.3 - Таксономія збоїв у ETL-процесах

Категорія Schema охоплює помилки, пов’язані з невідповідністю структури вхідних даних схемі цільової таблиці. До категорії Data quality належать збої, причиною яких є зміст самих даних: NULL-значення у NOT NULL колонках, дублікати ключів, виходи значень за допустимий діапазон. У категорію Infrastructure об’єднано тимчасові збої зовнішніх сервісів, тобто тайм-аути API, обмеження частоти запитів, нестачу пам’яті чи вільного місця на диску. Категорія Permission стосується помилок доступу: відмова IAM-ролей, відсутність прав на ключ KMS, протерміновані токени. Нарешті, категорія Logic / config охоплює помилки, що виникають у самому коді ETL-скрипту або у його конфігурації.

За результатами галузевого опитування Monte Carlo Data, проведеного серед 300 команд data engineering [5], частоту різних причин інцидентів з даними наведено на рисунку 1.4.

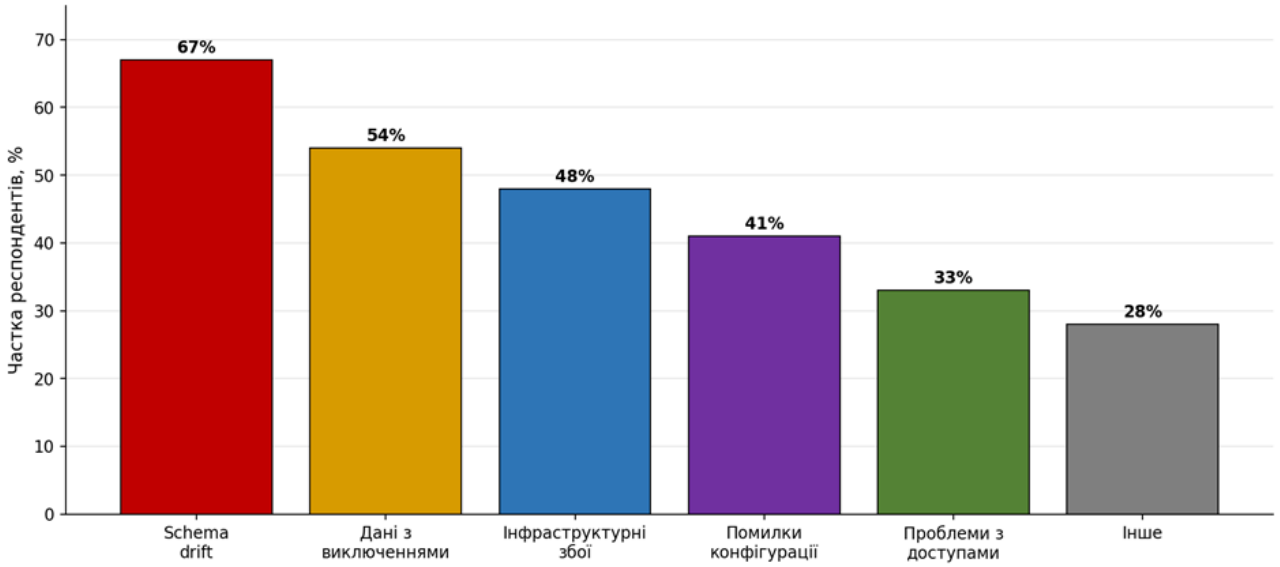


Рисунок 1.4 - Частота причин інцидентів з даними

Schema drift посідає перше місце у цьому рейтингу. Його назвали однією з трьох найчастіших причин інцидентів 67 % респондентів. Саме цей факт обґрунтовує вибір відповідної категорії збоїв як основного фокусу даного дослідження.

### *1.1.4 Schema drift як основна категорія збоїв*

Schema drift, або дрейф схеми, це явище, за якого структура даних на стороні джерела змінюється без попереднього повідомлення. У результаті виникає невідповідність між фактичним форматом вхідних даних та визначенням цільової таблиці у DDL [4]. У контексті досліджуваних пайплайнів виокремлено три основні підтипи такого дрейфу. Узагальнення цих підтипів наведено на рисунку 1.5.

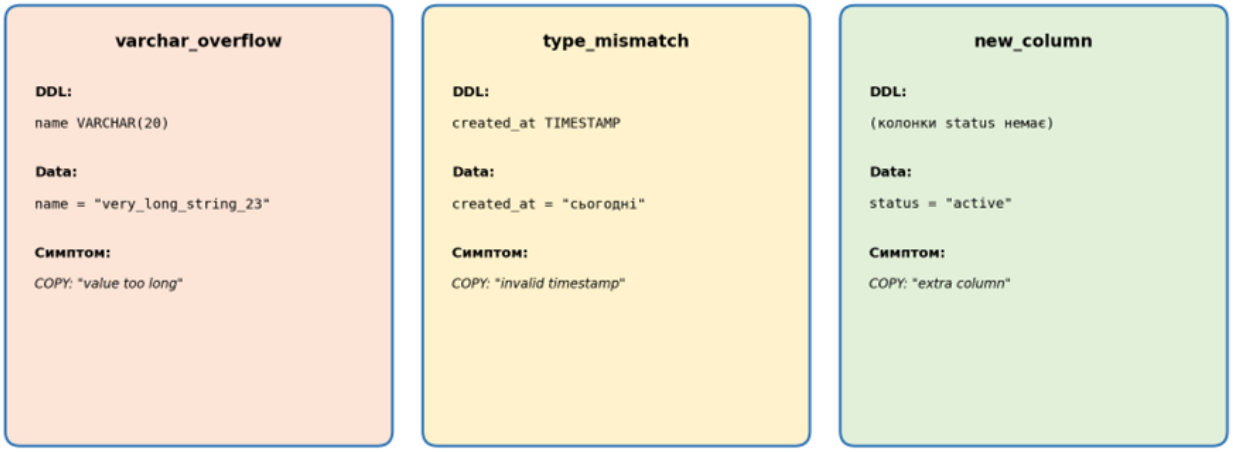


Рисунок 1.5 - Підтипи schema drift у ETL-пайплайнах

Підтип varchar_overflow виникає у випадках, коли значення рядкової колонки у вхідних даних перевищує максимальну довжину, визначену у DDL цільової таблиці. Типовим прикладом є зміна структури ідентифікатора користувача у API рекламної платформи з 20-символьного на 23-символьний.

Підтип type_mismatch має місце тоді, коли тип значення у вхідних даних не збігається з типом колонки у DDL. Як приклад можна навести поле created_at, визначене як TIMESTAMP, яке у новій версії API повертається у вигляді рядка з локалізованим описом часу замість стандартного формату ISO 8601.

Підтип new_column виявляється у ситуаціях, коли у вхідних даних з’являється колонка, відсутня у DDL цільової таблиці. Прикладом такого випадку є додавання нового поля у відповідь API без відповідного оновлення таблиці у Redshift. Усі три підтипи призводять до однакового результату: відмови COPY-операції Redshift [6], зупинки пайплайну та потреби у ручному втручанні інженера.

### 1.2 Аналіз існуючих підходів до обробки помилок у ETL-процесах

За останні два десятиліття у галузі сформувалося кілька основних підходів до обробки помилок у ETL-пайплайнах. Розглянемо їх у порядку зростання рівня

автоматизації, тобто від повністю ручного виправлення до сучасних платформ data observability.

### *1.2.1 Ручне виправлення*

Ручне виправлення є найпоширенішим на практиці підходом. Інженер отримує сповіщення про збій, аналізує логи виконання, визначає причину збою та вносить потрібні зміни самостійно. Цей підхід дає найвищу якість діагностики, адже людина може врахувати контекст бізнес-логіки, історію попередніх інцидентів та специфіку конкретного джерела даних. Водночас він має й суттєві недоліки. Серед них низька масштабованість при зростанні кількості пайплайнів, повна залежність від часу реакції інженера, а також високе психологічне навантаження за умов цілодобового чергування. За даними Fivetran, інженери даних витрачають до 40 % робочого часу саме на обслуговування вже наявних пайплайнів [8].

### *1.2.2 Retry-стратегії в оркестраторах*

Сучасні оркестратори ETL, серед яких Apache Airflow, Dagster, Prefect та Mage, мають вбудовані механізми повторних спроб (retry) із конфігурованою кількістю повторів і затримкою між ними [9]. Цей підхід ефективний для транзієнтних помилок, тобто тимчасових мережевих збоїв, тайм-аутів API та обмежень частоти запитів. Проте у разі структурних помилок на кшталт schema drift retry виявляється непридатним. Адже повторне виконання тієї самої задачі з тими самими вхідними даними дає той самий результат, а саме повторний збій.

### *1.2.3 Rule-based валідація даних*

Фреймворки декларативної валідації даних, як-от Great Expectations, dbt tests, Soda Core, Pandera, дають змогу описувати набір правил, яким мають відповідати дані [10, 11]. Ці правила перевіряються під час виконання пайплайну, а в разі їх порушення спрацьовує сповіщення або зупинка задачі. Підхід забезпечує раннє виявлення проблем, але має кілька істотних обмежень. По-перше, правила потрібно прописувати вручну, що дублює зусилля при великій кількості

пайплайнів. По-друге, такі правила не адаптуються до змін у джерелі: поява нових колонок чи зміна типів автоматично не виявляється, фіксується лише невідповідність уже описаному правилу. По-третє, фреймворк виключно сигналізує про проблему, але не пропонує її розв'язання.

### *1.2.4 Schema evolution у Data Lake*

Сучасні формати таблиць для Data Lake, а саме Apache Iceberg, Delta Lake, Apache Hudi, підтримують автоматичну еволюцію схеми (schema evolution) під час запису даних [12]. Якщо вхідний набір даних містить нову колонку, вона додається до схеми таблиці автоматично, без явного виконання DDL-команд. Цей механізм елегантно розв'язує проблему підтипу new_column, проте його сфера застосування суттєво обмежена. Він працює тільки на рівні Data Lake. Якщо ж цільовим сховищем виступає реляційна база даних на кшталт Amazon Redshift, де зміна схеми вимагає явних ALTER TABLE команд, schema evolution стає непридатною.

### *1.2.5 AIOps та платформи data observability*

Останнім поколінням рішень для моніторингу даних є комерційні платформи data observability, серед яких Monte Carlo, Bigeye, Anomalo, Lightup [13]. Ці платформи спираються на методи статистичного аналізу та машинного навчання задля виявлення аномалій у даних, тобто різких змін обсягу, відхилень розподілу значень, появи нових категорій. Виявлені аномалії передаються інженеру у вигляді сповіщення з відповідним контекстом. Подібні платформи ефективно знижують показник MTTD (Mean Time To Detect, тобто середній час виявлення проблеми), однак вони обмежуються лише сповіщенням і не виконують автоматичних виправлень. Як наслідок, MTTR (Mean Time To Repair) залишається на тому ж рівні, що й при ручному виправленні.

### *1.2.6 Порівняльний аналіз існуючих підходів*

Систематичне порівняння розглянутих підходів виконано за п’ятьма критеріями: виявлення, діагностика, виправлення, адаптивність, масштабованість. Результат у вигляді матриці наведено на рисунку 1.6.

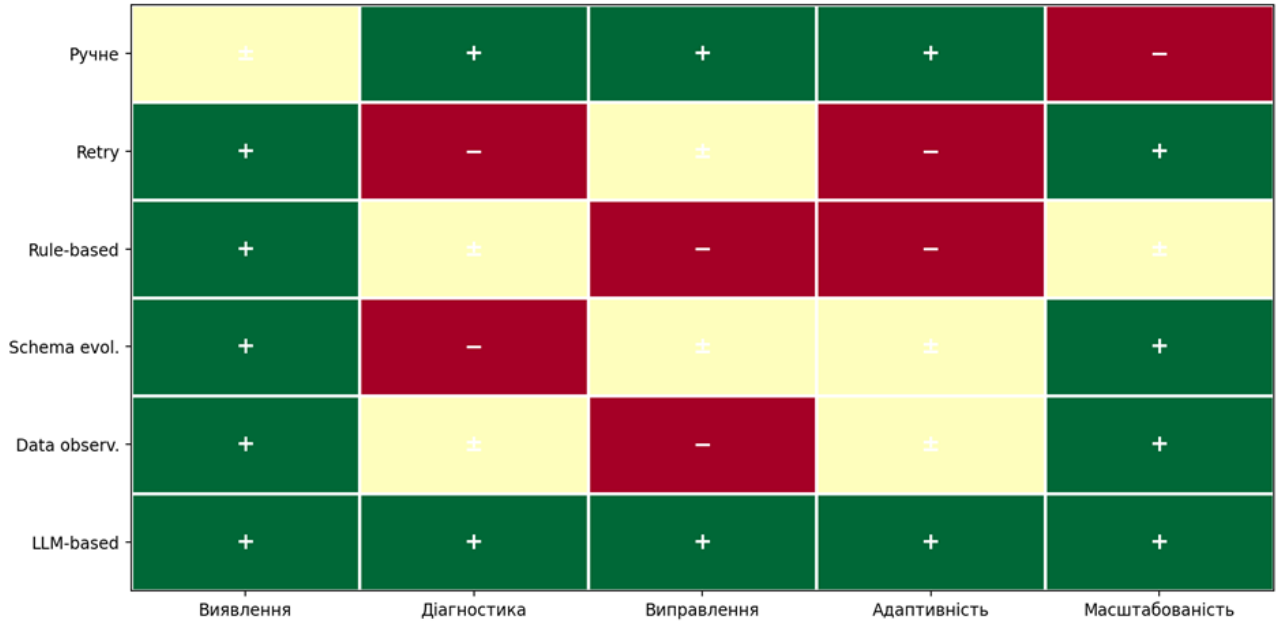


Рисунок 1.6 - Матриця порівняння підходів до обробки помилок

Як видно з матриці, жоден із наявних підходів не забезпечує повного циклу, що включав би виявлення, діагностування, автоматичне виправлення та адаптацію до нових типів помилок одночасно. Класичні підходи (retry, rule-based) покривають лише виявлення. Ручне виправлення погано масштабується. Data observability обмежується сповіщенням. Описана прогалина створює передумови для розробки нового підходу на основі великих мовних моделей.

## 1.3 Аналіз застосування LLM для задач автоматичного виправлення коду

Великі мовні моделі (Large Language Models, LLM) показали значний потенціал у задачах розуміння та генерації програмного коду [14, 15]. Сучасні моделі, серед яких Claude (Anthropic), Gemini (Google), GPT-4o (OpenAI) та DeepSeek Coder, стали невід’ємною частиною інструментарію розробників. Їх успішно застосовують для автодоповнення коду, рефакторингу, написання тестів та виправлення багів.

### *1.3.1 Automated Program Repair*

Напрям Automated Program Repair (APR) активно розвивається в академічній спільноті понад п’ятнадцять років [16]. Його метою є автоматичне виправлення дефектів у програмному коді на основі специфікації, тестів або поведінки еталонної реалізації. Традиційні підходи ґрунтувалися на трьох парадигмах. Перша, це генетичне програмування (GenProg), що передбачало еволюційний пошук виправлень шляхом мутацій абстрактного синтаксичного дерева. Друга, це семантичний аналіз (SemFix, Angelix), тобто формальний синтез виправлень на основі обмежень. Третя, це шаблонні трансформації (TBar, PAR), тобто застосування заздалегідь визначених шаблонів типових помилок. Усі ці підходи мали обмежену здатність працювати зі складним контекстом і потребували значних обчислювальних ресурсів.

Поява великих мовних моделей зумовила якісний зсув у напрямі APR. Тепер моделі здатні генерувати виправлення без попереднього визначення шаблонів, спираючись на розуміння контексту, семантики програми та найкращих практик, засвоєних у процесі навчання на мільйонах публічних репозиторіїв. Сучасні агентні системи на основі LLM, серед яких SWE-agent, AutoCodeRover, Aider, поєднують виклик LLM із інструментами для читання коду, виконання тестів і модифікації файлів.

### *1.3.2 Бенчмарк SWE-bench*

Бенчмарк SWE-bench [17] став стандартом оцінки здатності LLM виправляти баги. Він містить 2 294 реальні задачі з 12 популярних Python-репозиторіїв на GitHub. Кожна задача включає опис помилки, базову гілку коду та набір тестів, які мають проходити після виправлення. Оцінювання результату виконується бінарно: тести або проходять, або ні. Динаміку покращення показників провідних агентних систем за період 2023–2025 років наведено на рисунку 1.7.

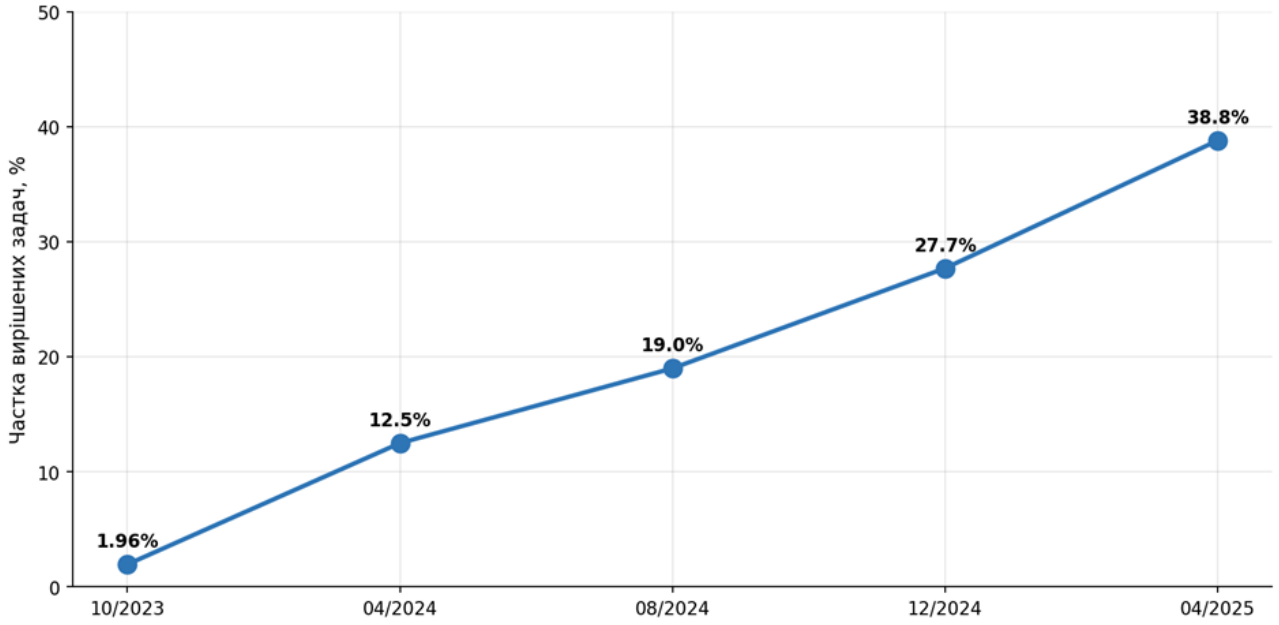


Рисунок 1.7 - Динаміка результатів агентних систем на SWE-bench

Станом на 2025 рік найкращі агентні системи розв'язують від 33 до 40 % задач SWE-bench Verified [18]. На тлі лише 2 % у 2023 році це являє собою якісний прогрес, який свідчить про практичну придатність LLM для автоматичного виправлення коду у реальних проєктах.

### *1.3.3 Застосування LLM у AIOps*

У сфері AIOps (Artificial Intelligence for IT Operations) великі мовні моделі задіюють для аналізу інцидентів, кореляції подій у системах моніторингу та генерації runbook-команд для відновлення. Дослідження Microsoft Research [19, 20], проведене на корпоративній базі даних інцидентів, показало, що GPT-4 правильно ідентифікує кореневу причину інциденту у 73 % випадків. Цей рівень близький до показника досвідчених інженерів SRE. Спеціалізовані рішення на кшталт PagerDuty AIOps, DynaTrace Davis AI та New Relic AI Monitoring задіюють LLM для автоматичної генерації пояснень користувачам, кластеризації повторюваних інцидентів та пропозиції типових варіантів виправлення.

### *1.3.4 Structured output як технологічна основа*

Ключовою технологічною можливістю сучасних LLM, що уможливлює їх застосування у автоматизованих системах, виступає structured output. Це

здатність моделі генерувати відповідь у строго визначеному форматі JSON, який відповідає заздалегідь специфікованій схемі, наприклад Pydantic-моделі [21]. Усі провідні провайдери підтримують цю можливість на рівні API. У Anthropic вона реалізована через механізм tool use, у OpenAI через function calling, у Google через JSON schema у параметрах генерації. Завдяки structured output відпадає потреба у ненадійному парсингу природномовного тексту, а сама модель може бути програмно інтегрована у автоматизовані pipeline-системи.

### *1.3.5 Обмеження застосування LLM*

Водночас використання LLM в автоматизованих системах має суттєві обмеження, які потрібно враховувати на етапі проєктування. По-перше, це явище галюцинацій, тобто генерації синтаксично правильного, проте семантично некоректного коду [22]. По-друге, обмеження контекстного вікна, через яке великі ETL-скрипти можуть просто не вміщатися у промпт [23]. По-третє, питання безпеки автоматичного виконання SQL та Python-коду без огляду людиною: некоректний код здатен спричинити втрату даних [24]. По-четверте, вартість API-викликів, яка при високій частоті інцидентів стає вагомим операційним показником [25].

Порівняння провідних LLM-моделей за критеріями, релевантними для задачі діагностування ETL-помилок, наведено у таблиці 1.2.

Таблиця 1.2 - Порівняння LLM для задач діагностування ETL-помилок

| **Модель** | **Structured output** | **Контекст** | **$ / 1M вх** | **Якість коду** |
|---|---|---|---|---|
| claude-sonnet-4-5 | Так | 200K | 3,00 | Висока |
| claude-haiku-4-5 | Так | 200K | 1,00 | Висока |
| gemini-2.5-flash | Так | 1M | 0,15 | Середня |
| gemini-2.5-flash-lite | Так | 1M | 0,10 | Середня |
| gpt-4o | Так | 128K | 2,50 | Висока |
| gpt-4o-mini | Так | 128K | 0,15 | Середня |

Незважаючи на значний прогрес у використанні LLM для APR загального призначення, саме застосування LLM для самовідновлення ETL-пайплайнів (self-

healing ETL) залишається малодослідженою темою. Існуючі академічні роботи зосереджені переважно на веб-застосунках та CI/CD-процесах. Специфіка ж ETL, як-от взаємодія з реляційними базами даних, schema drift, трансформація типів, потребує спеціалізованого підходу.

## 1.4 Постановка задачі дослідження

На підставі проведеного у попередніх підрозділах аналізу сформульовано такі висновки. ETL-пайплайни є критичною інфраструктурою, що безпосередньо впливає на якість управлінських рішень. Найпоширенішою причиною їх збоїв виступає schema drift у трьох формах (varchar_overflow, type_mismatch, new_column), на який припадає близько 67 % інцидентів. Жоден із наявних підходів не забезпечує повного циклу автоматичного відновлення, тобто від виявлення до застосування виправлення. Великі мовні моделі мають достатній потенціал для діагностування й генерації виправлень, проте їх використання саме у домені data engineering залишається малодослідженим.

### *1.4.1 Мета та завдання дослідження*

Метою дослідження є розроблення моделей та інформаційної технології автоматичного відновлення ETL-пайплайнів. В основу цієї технології покладено великі мовні моделі, які застосовуються для діагностування кореневих причин збоїв, спричинених дрейфом схеми даних, а також для генерації виправлень. При цьому має бути забезпечено безпеку застосування виправлень і можливість людського огляду.

Для досягнення поставленої мети потрібно розв’язати такі завдання: 1) провести аналіз предметної області та існуючих підходів; 2) розробити модель процесу самовідновлення ETL-пайплайну, яка охоплює фази збирання контексту, діагностування, опційного підтвердження та виконання виправлень; 3) розробити метод збирання контексту з множинних джерел та алгоритм schema diff; 4) розробити модель діагностування на основі LLM зі structured output; 5) розробити модель безпечного виконання виправлень із багаторівневою

валідацією; 6) розробити веб-інтерфейс для перегляду подій репарації та підтвердження виправлень оператором; 7) реалізувати моделі у вигляді програмного забезпечення; 8) провести експериментальне дослідження ефективності на тестових сценаріях schema drift.

### *1.4.2 Об'єкт та предмет дослідження*

Об'єкт дослідження становлять процеси забезпечення відмовостійкості ETL-пайплайнів, оркестрованих Apache Airflow у хмарній інфраструктурі Amazon Web Services. Базою практики виступає система інтеграції рекламних даних до аналітичного сховища Amazon Redshift Serverless.

Предметом дослідження є моделі та методи автоматичного діагностування й виправлення помилок у ETL-скриптах, спричинених дрейфом схеми даних, із застосуванням великих мовних моделей та можливістю людського огляду.

### *1.4.3 Вимоги до системи*

На основі аналізу предметної області й об'єкта дослідження сформульовано перелік функціональних та нефункціональних вимог до системи, що розробляється. Узагальнений перелік цих вимог наведено у таблиці 1.3.

Таблиця 1.3 - Функціональні та нефункціональні вимоги до системи

| Код | Тип | Вимога |
|---|---|---|
| ФВ-1 | Функц. | Автоматичне виявлення збою через on_failure_callback Airflow |
| ФВ-2 | Функц. | Збирання контексту збою з шести гетерогенних джерел |
| ФВ-3 | Функц. | Детерміністичне виявлення невідповідностей схеми (schema diff) |
| ФВ-4 | Функц. | LLM-діагностування з формалізованим structured output |
| ФВ-5 | Функц. | Генерація виправлень трьох типів (SQL DDL, DAG, ETL-скрипт) |
| ФВ-6 | Функц. | Підтримка трьох режимів роботи (log_only, auto, pending_approval) |
| ФВ-7 | Функц. | Веб-інтерфейс для перегляду та підтвердження виправлень |
| ФВ-8 | Функц. | Можливість повторного діагностування з контекстом оператора |

| | | |
|---|---|---|
| НФВ-1 | Нефункц. | Час активної репарації не перевищує 60 секунд |
| НФВ-2 | Нефункц. | Збереження аудиторського сліду усіх рішень системи |
| НФВ-3 | Нефункц. | Багаторівнева валідація усіх дій (defense in depth) |
| НФВ-4 | Нефункц. | Підтримка кількох провайдерів LLM через єдиний інтерфейс |
| НФВ-5 | Нефункц. | Інтеграція з GitHub для версіонування змін |

### *1.4.4 Вхідні та вихідні дані системи*

Вхідні дані системи становить контекст збою, що збирається автоматично при спрацюванні on_failure_callback Airflow. Він складається з шести джерел: інформація про помилку (тип виключення, traceback, ідентифікатори задачі та DAG); метадані DAG (SQL для CREATE TABLE, шлях до ETL-скрипту в S3, назва Glue-джобу); DDL-опис цільової таблиці у Redshift (список колонок із типами та обмеженнями); вихідний код ETL-скрипту (Python із використанням pandas та awswrangler); логи CloudWatch (output та error потоки Glue-джобу); результат детерміністичного алгоритму schema diff.

Вихідними даними системи виступає структурований діагноз, що включає поля root_cause, failure_category, confidence та reasoning, разом із набором виправлень трьох типів: SQL DDL (ALTER TABLE для Redshift), зміни у DAG-файлі, зміни у вихідному коді ETL-скрипту. SQL-виправлення застосовуються через Redshift Data API. Зміни коду оформлюються як pull request у GitHub, а за потреби завантажуються безпосередньо у S3 для негайного підхоплення під час наступного запуску Glue-джобу.

### *1.4.5 Критерії оцінки ефективності*

Для оцінки ефективності розроблюваної системи визначено такі основні критерії: точність діагностування (частка прогонів, у яких правильно ідентифіковано failure_category); precision запропонованих виправлень (частка валідних виправлень серед запропонованих); частка успішного end-to-end відновлення (частка прогонів, у яких після застосування виправлень пайплайн успішно перезапустився); частота хибних спрацювань (false positive rate); time-

to-recovery (час від збою до успішного перезапуску); вартість одного виклику системи у доларах США. Експериментальне дослідження за цими критеріями буде представлено у третьому розділі.

### Висновки до розділу 1

У розділі проведено аналіз предметної області, наявних підходів до обробки помилок у ETL-процесах, а також можливостей застосування великих мовних моделей у задачах автоматичного виправлення коду.

Встановлено, що ETL-пайплайни виступають критичною інфраструктурою сучасних аналітичних систем, а вартість простою дата-інфраструктури сягає тисяч доларів на годину. Сформульовано таксономію збоїв ETL-процесів, у межах якої їх поділено на п’ять категорій (Schema, Data quality, Infrastructure, Permission, Logic / config). Виявлено, що найпоширенішою категорією є schema drift, який охоплює 67 % інцидентів за галузевими опитуваннями та проявляється у трьох підтипах: varchar_overflow, type_mismatch, new_column.

Проведено систематичний аналіз наявних підходів до обробки помилок, серед яких ручне виправлення, retry-стратегії оркестраторів, rule-based валідація, schema evolution у Data Lake та платформи data observability. Встановлено, що жоден з цих підходів не забезпечує повного циклу самовідновлення, тобто від виявлення до застосування виправлення.

Проаналізовано можливості застосування великих мовних моделей у задачах Automated Program Repair, представлено динаміку розвитку агентних систем на бенчмарку SWE-bench (зростання з 2 % до 38,8 % за два роки), розглянуто застосування LLM у AIOps та можливості structured output як технологічної основи інтеграції LLM в автоматизовані pipeline-системи. Виявлено основні обмеження застосування LLM (галюцинації, контекстне вікно, безпека, вартість), які потребують спеціальних архітектурних рішень.

Сформульовано постановку задачі дослідження, тобто мету, перелік завдань, об’єкт і предмет дослідження, функціональні та нефункціональні вимоги до системи, формат вхідних і вихідних даних, критерії оцінки ефективності. У

наступному розділі будуть розроблені моделі та методи, що реалізують ці вимоги.

# РОЗДІЛ 2
# МЕТОДИ ТА МОДЕЛІ ДІАГНОСТУВАННЯ І ВІДНОВЛЕННЯ ETL-ПОМИЛОК

## 2.1 Модель процесу самовідновлення ETL-пайплайну

Аналіз предметної області, виконаний у першому розділі, виявив суттєві обмеження наявних інструментів автоматизованого опрацювання інцидентів у ETL-пайплайнах. Жодне з розглянутих рішень не охоплює повного циклу автоматичного відновлення. Спираючись на ці спостереження, у роботі було побудовано модель процесу самовідновлення ETL-пайплайну, яка формалізує послідовність дій від моменту виникнення збою до накладання виправлення.

Запропонована модель об'єднує два підходи до обробки помилок. Перший, детерміністичний, зводиться до аналізу невідповідностей схеми за чітко визначеними правилами та гарантує однозначність базової класифікації збою. Другий ґрунтується на застосуванні великих мовних моделей; він генерує конкретне виправлення з опертям на контекст вихідного коду ETL-скрипту. Поєднання цих компонентів дозволяє нівелювати слабкі місця як суто правил-орієнтованих, так і чисто статистичних рішень.

Зведене порівняння можливостей запропонованої моделі з основними категоріями наявних рішень, виявленими у першому розділі, наведено у таблиці 2.1.

Таблиця 2.1 - Порівняння запропонованої моделі з існуючими підходами

| **Можливість** | **Retry** | **Validation** | **Schema evol.** | **Observability** | **Запропоновано** |
|---|---|---|---|---|---|
| Виявлення збою | + | + | + | + | + |
| Класифікація типу дрейфу | − | частково | частково | + | + |
| Генерація виправлення | − | − | частково | − | + |

| | | | | | |
|---|---|---|---|---|---|
| Автоматичне застосування | + | – | + | – | + |
| Human-in-the-loop | – | – | – | + | + |
| Аудиторський слід | – | – | – | + | + |
| Підтримка трьох режимів | – | – | – | – | + |

Як видно з таблиці, жоден з традиційних підходів не закриває повного циклу виявлення, діагностування та виправлення з підтримкою людського перегляду. Запропонована модель усуває цю прогалину завдяки поєднанню детерміністичного аналізу з LLM-діагностуванням, доповнених гнучкою стратегією накладання виправлень.

### *2.1.1 Формальне представлення процесу*

Процес самовідновлення подано як композицію чотирьох відображень між множинами станів системи; графічну ілюстрацію наведено на рисунку 2.1.

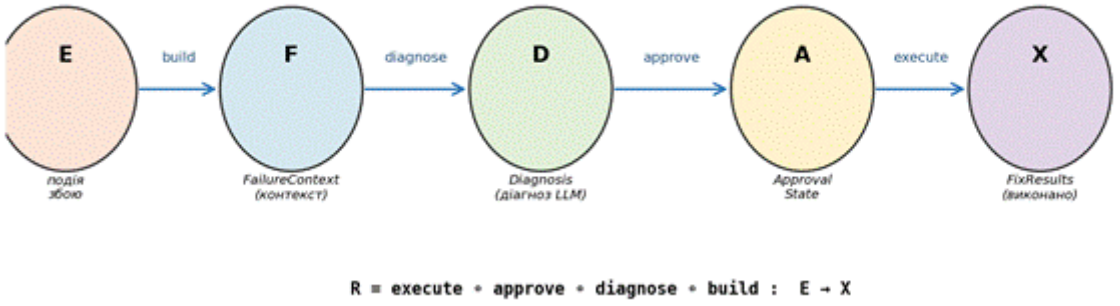


Рисунок 2.1 - Формальне представлення моделі самовідновлення

Загальне відображення R задає перетворення події збою у результат застосування виправлення згідно з формулою (2.1):

$$R: E \to F \to D \to A \to X \qquad (2.1)$$

де $E$ позначає множину подій збою, що генеруються Apache Airflow при невдалому завершенні задачі; $F$ є множиною об'єктів FailureContext, які агрегують контекст збою з шести гетерогенних джерел; через $D$ позначено

множину об'єктів Diagnosis, що містять формалізований діагноз LLM з оцінкою достовірності; *A* описує множину станів підтвердження (Approval State), яка може набувати тривіального вигляду в автоматичному режимі; нарешті *X* є множиною об'єктів FixResults з результатами застосування виправлень.

Реалізація кожного з чотирьох відображень (build, diagnose, approve, execute) оформлена як окремий програмний модуль. Така модульність полегшує незалежне тестування кожної фази та дає змогу замінювати реалізацію будь-якого з відображень без впливу на решту системи.

### *2.1.2 Чотирифазна модель процесу*

Реалізація відображення R складається з чотирьох послідовних фаз. Діаграму активності процесу наведено на рисунку 2.2.

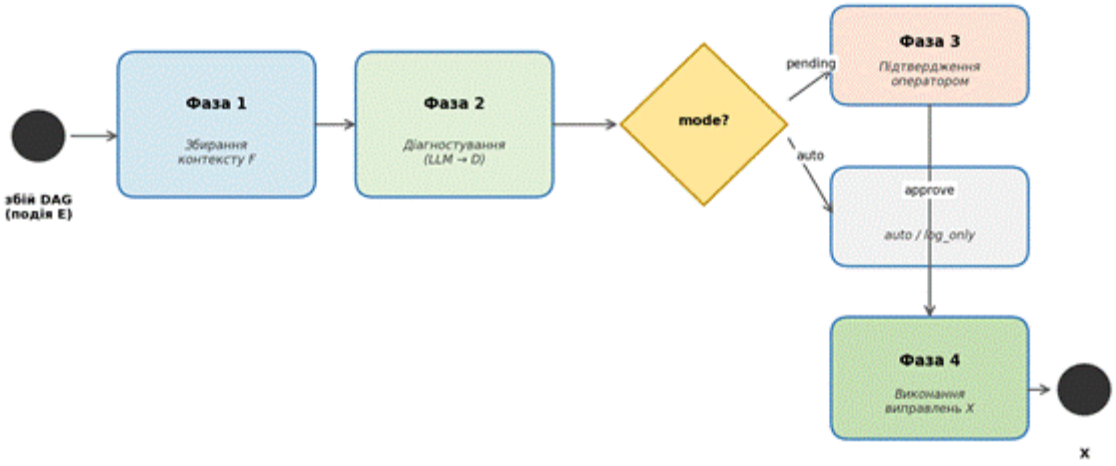


Рисунок 2.2 - Діаграма активності процесу самовідновлення

Фаза 1 (Збирання контексту, E → F) активується через виклик callback-функції on_failure_callback. У межах цієї фази виконується паралельне опитування шести джерел контексту, результат серіалізується в об'єкт FailureContext та зберігається на диск як артефакт context.json для подальшого аудиту.

Фаза 2 (Діагностування, F → D) передає контекст до великої мовної моделі через інтерфейс LangChain зі специфікацією structured output. Підсумком фази є

об'єкт Diagnosis з полями root_cause, failure_category, proposed_fixes, confidence та reasoning, що серіалізується у файл diagnosis.json.

Фаза 3 (Підтвердження, D → A) є опційною. Її активація відбувається лише в режимі pending_approval, коли система зберігає файл status.json зі станом pending для кожного запропонованого виправлення та очікує дії оператора через веб-інтерфейс. У режимах log_only й auto ця фаза тривіальна, оскільки стан A визначається режимом без участі людини.

Фаза 4 (Виконання виправлень, A → X) застосовує підтверджені виправлення спеціалізованими обробниками за типом fix_type. SQL-команди виконуються через Redshift Data API, зміни вихідного коду накладаються на локальну файлову систему, S3 та pull request у GitHub. Підсумок кожного застосування серіалізується у файл fix_results.json.

### *2.1.3 Режими роботи системи*

У моделі передбачено три режими роботи, які забезпечують поступовий перехід від ручного контролю до повної автоматизації. Опис режимів та їхнього призначення наведено у таблиці 2.2.

Таблиця 2.2 - Режими роботи системи самовідновлення

| **Режим** | **Дія системи** | **Рекомендоване застосування** |
|---|---|---|
| log_only | Валідація без застосування | Початкове впровадження, опрацювання |
| pending_approval | Зупинка після діагностування, очікування дії | Критичні пайплайни, складні випадки |
| auto | Автоматичне застосування валідованих виправлень | Типові інциденти, стабільні сценарії |

Перемикання між режимами відбувається через зміну значення змінної середовища REPAIR_WORKFLOW_APPROVAL_MODE без модифікації коду системи. Така організація конфігурування дозволяє адаптувати ступінь автоматизації під конкретний пайплайн та тип збою без перерозгортання.

### *2.1.4 Артефакти процесу*

Кожен прогін системи продукує набір артефактів, які зберігаються у структурованому каталозі за шляхом logs/repair_workflow/<dag_id>/<logical_date>/. Перелік артефактів та їхнє призначення зведено у таблиці 2.3.

Таблиця 2.3 - Артефакти процесу самовідновлення

| **Файл** | **Фаза** | **Призначення** |
|---|---|---|
| context.json | Фаза 1 | Повний об'єкт FailureContext |
| prompt.txt | Фаза 2 | Текст промпту, надісланого LLM |
| diagnosis.json | Фаза 2 | Структурований діагноз DiagnosisOutput |
| diagnosis_v{N}.json | Re-diagnosis | Версіонований повторний діагноз |
| status.json | Фаза 3 | Стан підтвердження кожного виправлення |
| fix_results.json | Фаза 4 | Результати застосування виправлень |

Структура артефактів забезпечує повну відтворюваність кожного рішення системи. За наявності файлу context.json та фіксованого значення температури моделі (0,0) повторний виклик діагностування на тих самих вхідних даних дає той самий діагноз. Така властивість є важливою для аудиту автоматизованих систем у продуктивному середовищі.

## 2.2 Метод збирання контексту помилки та алгоритм schema diff

Якість діагностування безпосередньо залежить від повноти та структурованості контексту, переданого LLM. Запропонований метод збирання контексту базується на агрегації сигналів з шести гетерогенних джерел; кожне з них надає свій тип інформації про збій. На відміну від існуючих рішень, які зазвичай оперують лише ізольованими сигналами (як-от traceback виключення), розроблений метод формує повний зріз стану ETL-пайплайну на момент збою.

### *2.2.1 Джерела контексту*

Перш ніж переходити до опису джерел, доцільно формалізувати ключові структури даних, що циркулюють між фазами процесу. Інформація про колонку DDL описується як впорядкована трійка

$$d = \langle name, type, max\_length \rangle, \; d \in D \quad (2.2)$$

де $name \in \Sigma^*$ є ідентифікатором колонки в нижньому регістрі; $type \in T_R$ позначає тип Redshift з множини $T_R$ = {VARCHAR, INTEGER, BIGINT, FLOAT, DOUBLE PRECISION, BOOLEAN, TIMESTAMP, DATE}; параметр $max\_length \in \mathbb{N} \cup \{\perp\}$ задає максимальну довжину для типу VARCHAR(n) і є невизначеним для решти типів.

Профіль колонки вхідних даних описується як кортеж

$$p = \langle name, dtype, null\_count, max\_length \rangle, \; p \in P \quad (2.3)$$

де $name \in \Sigma^*$ є ідентифікатором колонки; $dtype \in T_P$ позначає тип pandas/pyarrow з множини сумісних типів $T_P$; параметр $null\_count \in \mathbb{N}$ задає кількість значень NULL у вибірці, а $max\_length \in \mathbb{N}$ зберігає максимальну спостережену довжину рядкового значення.

Запис невідповідності схеми описується таким чином:

$$s = \langle column, diff\_type, detail \rangle, \; s \in S \quad (2.4)$$

де *column* зберігає ідентифікатор колонки; *diff_type* ∈ {VARCHAR_OVERFLOW, TYPE_MISMATCH, NEW_COLUMN} визначає тип невідповідності, а поле *detail* є словником з типозалежними деталями, а саме: для VARCHAR_OVERFLOW зберігається {ddl_len, data_len}, для TYPE_MISMATCH використовується {ddl, observed}, для NEW_COLUMN заноситься {observed_dtype}.

Архітектура збирання контексту побудована на основі принципу хаба: центральний об'єкт FailureContext отримує сигнали від шести джерел паралельно. Графічне представлення наведено на рисунку 2.3.

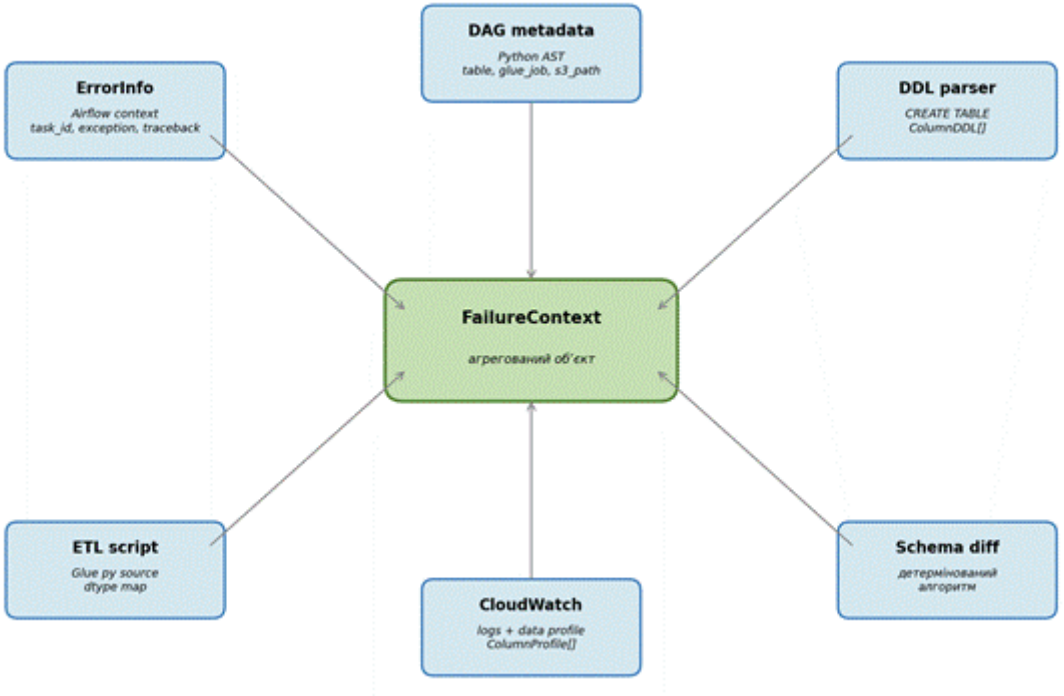


Рисунок 2.3 - Джерела контексту помилки

Детальний опис кожного джерела, формат отриманих даних та орієнтовний обсяг у токенах при подальшій передачі до LLM зведено у таблиці 2.4.

Таблиця 2.4 - Характеристики джерел контексту

| № | Джерело | Спосіб отримання | Формат | Токенів |
|---|---|---|---|---|
| 1 | ErrorInfo | Airflow context dict | dataclass | ≈ 600 |
| 2 | DAG metadata | Парсинг Python-файла DAG | dataclass | ≈ 200 |
| 3 | DDL parser | Парсинг CREATE TABLE | list[ColumnDDL] | ≈ 400 |
| 4 | ETL script | Читання з S3 / локально | str | ≈ 3500 |
| 5 | CloudWatch logs | AWS CloudWatch API | str + list[ColumnProfile] | ≈ 2000 |
| 6 | Schema diff | Алгоритмічне порівняння | list[SchemaDiffEntry] | ≈ 300 |

Опрацювання кожного з джерел виконується у власному try/except-блоці. Це гарантує толерантність до часткової відсутності даних: коли окреме джерело недоступне (як-от логи CloudWatch ще не з'явились або S3-об'єкт видалено), решта контексту все одно буде зібрана та передана у фазу діагностування.

Цікавою деталлю джерела CloudWatch logs є спосіб отримання структурованого профілю даних. Glue-джоб у штатному режимі логує характеристики кожної колонки одразу після читання вхідного Parquet-файла, а саме: тип pandas, кількість NULL-значень, максимальну довжину рядкових значень. Парсер CloudWatch вилучає такі записи з потоку логів та формує список об'єктів ColumnProfile, який далі надходить на вхід алгоритму schema diff.

Для ілюстрації роботи методу нижче подано приклад фрагмента об'єкта FailureContext, серіалізованого у JSON, для сценарію varchar_overflow (лістинг 2.1).

Лістинг 2.1 - Приклад серіалізованого FailureContext

```
{
 "error": {
   "task_id": "migrate_data_ad",
   "dag_id": "ad_migration",
   "exception_type": "ProgrammingError",
   "exception_msg": "String length exceeds DDL length"
 },
 "ddl": [
   {"name": "id_user",  "data_type": "VARCHAR", "max_length": 20},
   {"name": "campaign", "data_type": "VARCHAR", "max_length": 100}
 ],
 "profile": [
   {"name": "id_user",  "dtype": "object", "max_length": 23},
   {"name": "campaign", "dtype": "object", "max_length": 87}
 ],
 "schema_diff": [
   {"column": "id_user",
    "diff_type": "varchar_overflow",
```

```
    "detail": {"ddl_len": 20, "data_len": 23}}
  ]
}
```


### *2.2.2 Принцип селективного включення*

Передача всього зібраного контексту до LLM призводить до невиправдано великого розміру промпту та підвищеної вартості виклику. Для оптимізації обсягу промпту реалізовано принцип селективного включення даних, тобто до промпту потрапляють лише ті фрагменти контексту, які мають релевантне значення для конкретного типу збою.

Профіль даних, наприклад, включається лише для тих колонок, що фігурують у результаті schema diff. Логи CloudWatch обмежуються першими ста рядками, що йдуть після першого запису рівня ERROR. Якщо вихідний код ETL-скрипту перевищує задану квоту обсягу, до промпту потрапляє лише функція, що містить операцію COPY у Redshift. Сумарний обсяг промпту становить у середньому 8 700 токенів, що уможливлює використання моделей з контекстним вікном від 16K токенів.

### *2.2.3 Алгоритм schema diff*

Алгоритм schema diff є ключовим детерміністичним компонентом методу збирання контексту. Він обчислює множину невідповідностей між DDL-описом таблиці у Redshift та фактичним профілем вхідних даних, виявляючи три типи дрейфу: varchar_overflow, type_mismatch, new_column. Графічне представлення алгоритму подано на рисунку 2.4.

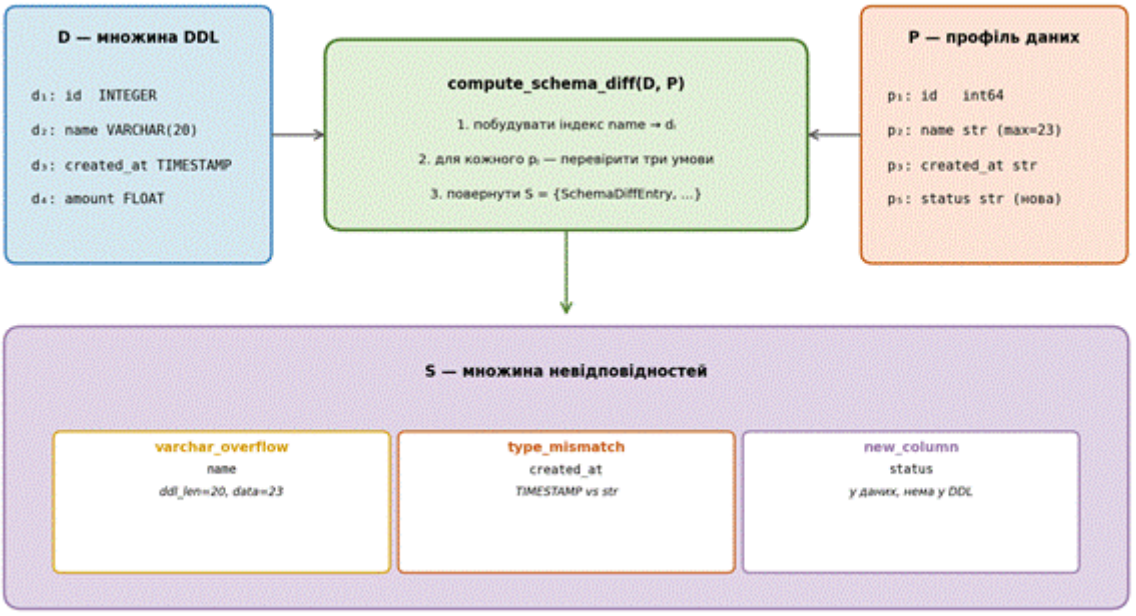


Рисунок 2.4 - Алгоритм обчислення невідповідностей схеми

Формально нехай $D = \{d_1, d_2, \ldots, d_n\}$ позначає множину DDL-визначень колонок цільової таблиці, кожне з яких представлене як тріада ⟨name, type, max_length⟩, а $P = \{p_1, p_2, \ldots, p_m\}$ є множиною профілів колонок вхідних даних, кожен у вигляді кортежу ⟨name, dtype, max_length⟩. Алгоритм обчислює множину невідповідностей $S$ = schema_diff($D$, $P$), послідовність кроків якого формалізовано у лістингу 2.2.

Лістинг 2.2 - Псевдокод алгоритму schema_diff

```
function schema_diff(D, P) -> S:
    map_D := { d.name.lower() -> d  for d in D }
    S := empty set
    for p in P:
    d := map_D.get(p.name.lower())
    if d is None:
    S.add( SchemaDiffEntry(
      column = p.name,
      diff_type = NEW_COLUMN,
```

```
    detail   = {observed_dtype: p.dtype} ) )
continue
if d.type = VARCHAR(n) and p.max_length > n:
S.add( SchemaDiffEntry(
    column = p.name,
    diff_type = VARCHAR_OVERFLOW,
    detail   = {ddl_len: n, data_len: p.max_length} ) )
continue
if not types_compatible(p.dtype, d.type):
S.add( SchemaDiffEntry(
    column = p.name,
    diff_type = TYPE_MISMATCH,
    detail   = {ddl: d.type, observed: p.dtype} ) )
return S
```

Часова складність алгоритму становить O(|D| + |P|): побудова індексу map_D потребує одного проходу по множині D, а основний цикл виконує один прохід по множині P. Просторова складність дорівнює O(|D|) і витрачається на зберігання індексу. Коректність алгоритму випливає з його побудови: кожен запис у P порівнюється рівно один раз і відносно одного з трьох взаємовиключних предикатів, тобто результат S не містить дублікатів.

Зведену характеристику алгоритму schema_diff наведено у таблиці 2.5.

Таблиця 2.5 - Характеристика алгоритму schema_diff

| **Властивість** | **Значення** |
|---|---|
| Часова складність | O(\|D\| + \|P\|) |
| Просторова складність | O(\|D\|) |
| Детермінізм | Детерміністичний (не залежить від порядку) |
| Повнота | Виявляє всі три типи дрейфу за одну ітерацію |
| Несуперечність | Кожен запис P порівнюється рівно один раз |
| Типове \|D\|, \|P\| | 20-60 колонок |
| Типовий час виконання | менше 5 мс на типовій таблиці |

Перевірка сумісності типів types_compatible(p_dtype, d_type) спирається на заздалегідь визначену матрицю сумісності типів даних Redshift та pandas/pyarrow, наведену у таблиці 2.6.

Таблиця 2.6 - Матриця сумісності типів даних Redshift та pandas/pyarrow

| **Тип Redshift (DDL)** | **Сумісні типи pandas/pyarrow** |
|---|---|
| VARCHAR(n) | object, string, str |
| INTEGER | int64, int32, Int64, Int32 |
| BIGINT | int64, Int64 |
| FLOAT, DOUBLE PRECISION | float64, float32, Float64 |
| BOOLEAN | bool, boolean |
| TIMESTAMP | datetime64[ns], datetime64[ns, UTC] |
| DATE | datetime64[ns], object (ISO date strings) |

Така матриця враховує толерантність pandas до подання цілих чисел у різних розрядностях (як-от int32 проти int64). Окремо передбачена підтримка обох варіантів типів, тобто нативних numpy-типів та Nullable Integer/Boolean типів pandas (Int64, boolean), які застосовуються за наявності NULL-значень у колонці.

## 2.3 Модель діагностування за допомогою LLM

Модель діагностування реалізує другу фазу процесу самовідновлення, тобто перетворення об'єкта FailureContext на структурований діагноз Diagnosis. Ключовою рисою цієї моделі виступає механізм structured output, який зобов'язує LLM повертати відповідь у формі екземпляра попередньо визначеної Pydantic-моделі. Завдяки цьому забезпечується детермінізм формату відповіді та усувається необхідність парсингу природномовного тексту, що є основним джерелом помилок при інтеграції LLM в автоматизовані pipeline-системи.

### *2.3.1 Структура промпту*

Промпт, надісланий до LLM, складається з двох логічних частин: системного повідомлення та повідомлення користувача. Системне повідомлення задає роль моделі та правила формування відповіді. Повідомлення користувача містить структурований контекст збою, поділений на шість тематичних секцій. Загальну структуру промпту унаочнено на рисунку 2.5.

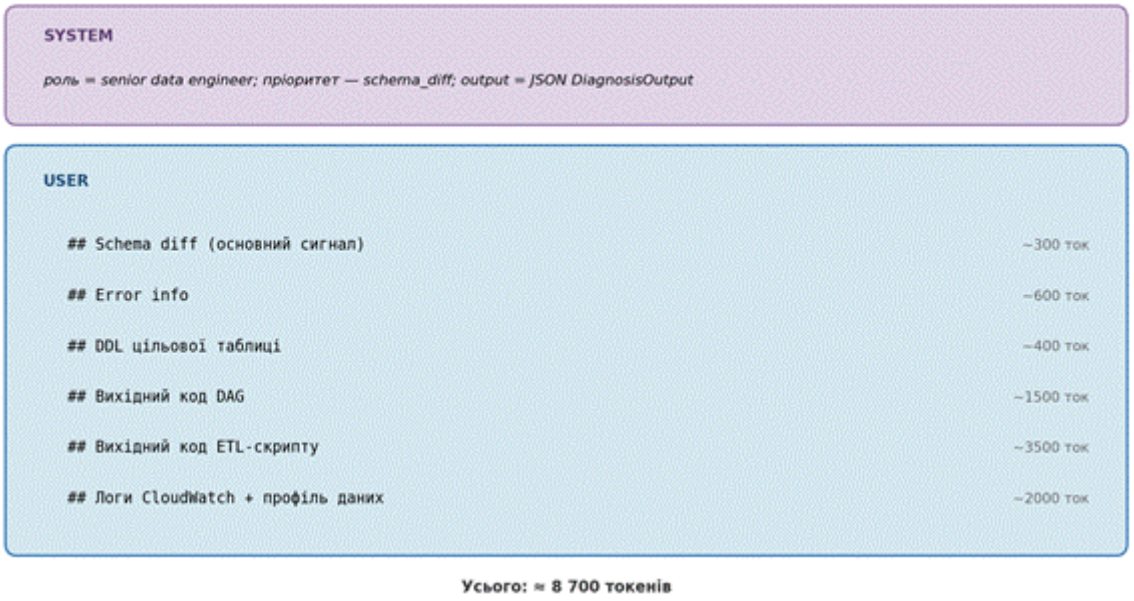


Рисунок 2.5 - Структура промпту для діагностування

У системному повідомленні фіксуються три ключові налаштування. Воно визначає роль моделі як старшого інженера даних з досвідом у налагодженні Apache Airflow та Amazon Redshift. Далі модель отримує інструкцію пріоритизувати результат schema diff як основний сигнал, що дозволяє уникнути ситуацій, коли LLM генерує діагноз, який суперечить детерміністично виявленій невідповідності схеми. Окремий пункт системного повідомлення специфікує формат вихідних даних, а саме Pydantic-модель DiagnosisOutput.

Повідомлення користувача структуроване у форматі Markdown із заголовками другого рівня для кожної секції. Цей формат добре сприймається сучасними LLM, оскільки відповідає типовій структурі технічної документації, на якій моделі тренувались. Принципи формування промпту, покладені в основу запропонованої моделі, узагальнено у таблиці 2.7.

Таблиця 2.7 - Принципи формування промпту для діагностування

| Принцип | Реалізація | Очікуваний ефект |
|---|---|---|
| Призначення ролі | system: senior data engineer | Зосередження моделі на доменній експертизі |
| Пріоритизація сигналу | інструкція: schema_diff як головний сигнал | Зменшення розходження з детерміністичним аналізом |
| Структурований формат | Markdown із заголовками ## для секцій | Кращий парсинг великих контекстів моделлю |
| Жорсткий вихід | with_structured_output(DiagnosisOutput) | Усунення помилок парсингу природномовного тексту |
| Селективне включення | профіль лише для колонок зі schema_diff | Економія токенів без втрати релевантності |
| Низька температура | temperature = 0,0 | Відтворюваність діагнозу при тих самих вхідних даних |

### *2.3.2 Pydantic-модель діагнозу*

Центральним елементом моделі діагностування є Pydantic-клас DiagnosisOutput, який задає форму відповіді LLM та одночасно генерує JSON Schema для виклику API провайдера у режимі structured output. Структуру моделі наведено у таблиці 2.8.

Таблиця 2.8 - Структура моделі DiagnosisOutput

| Поле | Тип | Опис |
|---|---|---|
| root_cause | str | Детальний опис кореневої причини збою |
| failure_category | enum | schema_drift, data_overflow, permission_error, connection_error, unknown |
| proposed_fixes | list[ProposedFix] | Впорядкований список запропонованих виправлень |
| confidence | float [0, 1] | Оцінка достовірності діагнозу |

| | | |
|---|---|---|
| reasoning | str | Покрокове обґрунтування діагнозу |
| false_positive_flags | list[str] | Записи schema diff, які є хибними спрацюваннями |

Кожен елемент списку proposed_fixes описується вкладеною моделлю ProposedFix, структуру якої представлено у таблиці 2.9.

Таблиця 2.9 - Структура моделі ProposedFix

| Поле | Тип | Опис |
|---|---|---|
| fix_type | enum | sql_ddl, dag_code_change, etl_script_change |
| description | str | Опис виправлення для людського огляду |
| sql | str \| None | Повна SQL-команда для типу sql_ddl |
| code_change | str \| None | Фрагмент зміни вихідного коду |
| target_file | str \| None | Шлях до файла, що модифікується |

Перелік допустимих значень полів-перелічень (failure_category, fix_type) жорстко обмежено на рівні Pydantic-схеми. Завдяки цьому LLM не може повернути категорію або тип виправлення, для якого у системі відсутній обробник, що усуває цілий клас помилок інтеграції.

### *2.3.3 Виклик LLM через LangChain*

Для абстрагування від конкретного провайдера LLM було використано фреймворк LangChain. Уніфікований інтерфейс init_chat_model приймає назву моделі та назву провайдера як рядкові параметри і повертає об'єкт ChatModel, що підтримує однаковий API незалежно від реалізації. Метод with_structured_output(DiagnosisOutput, include_raw=True) формує ланцюг обробки, який автоматично перетворює відповідь LLM на екземпляр Pydantic-моделі та повертає метадані використання токенів. Температура моделі встановлена на 0,0 задля мінімізації стохастичності та відтворюваності діагнозів при однакових вхідних даних.

Послідовність взаємодії модулів при виклику LLM показано на рисунку 2.6.

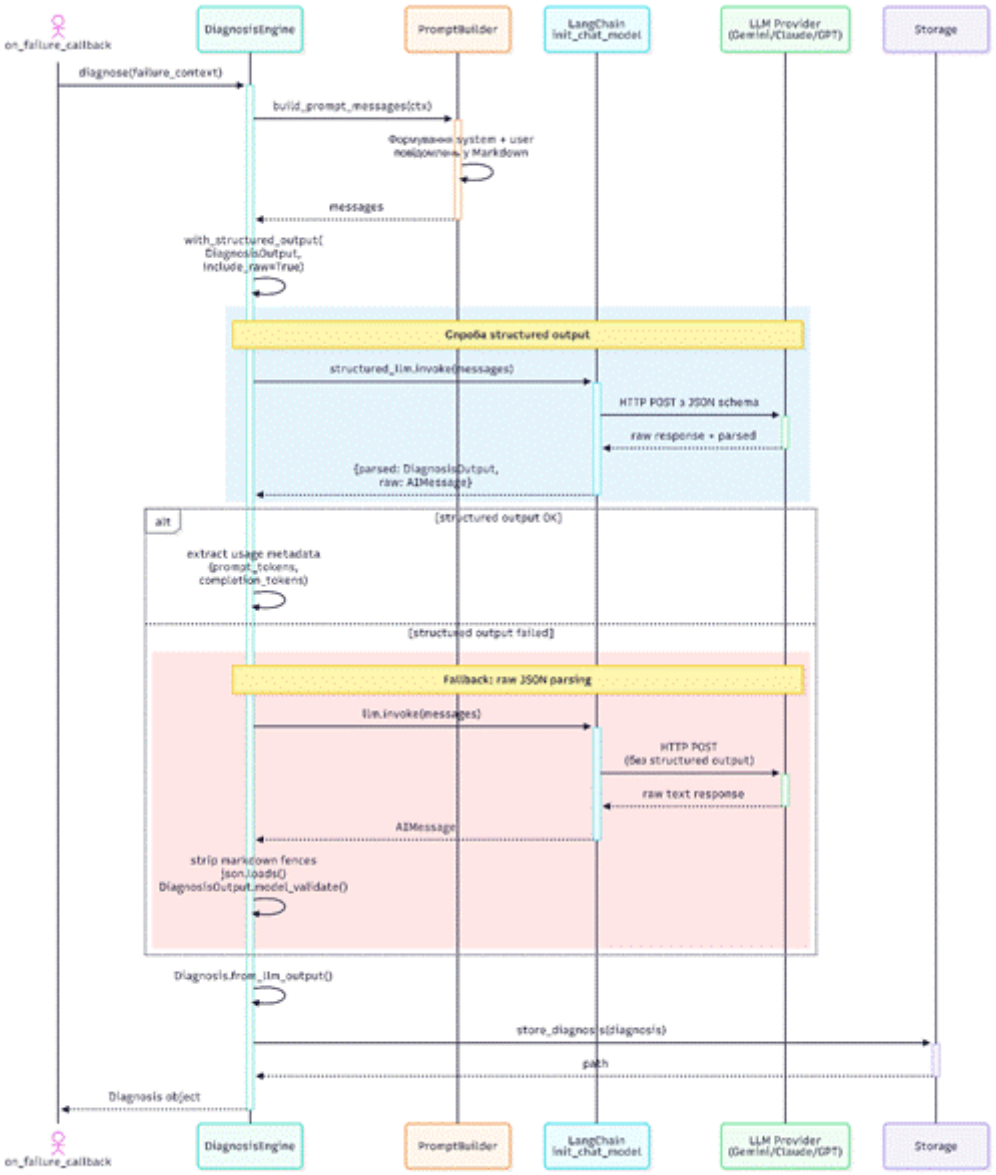


Рисунок 2.6 - Діаграма послідовності виклику LLM

Вибір абстракції LangChain замість прямого виклику API провайдерів зумовлений двома практичними міркуваннями. Перше пов'язане зі специфікою structured output: Anthropic реалізує його через tool use, OpenAI задіює function calling, а Google використовує JSON schema у параметрах генерації. Пряма реалізація потребувала б трьох незалежних клієнтів, що дублюють логіку перетворення Pydantic-моделі у внутрішнє представлення кожного провайдера; LangChain виконує це перетворення автоматично через єдиний інтерфейс with_structured_output. Друге міркування стосується нормалізації метаданих використання токенів: різні провайдери повертають їх у різних структурах

(input_tokens, prompt_tokens, tokenCount), і LangChain зводить їх до єдиного формату AIMessage.usage_metadata, що спрощує обчислення вартості виклику.

### *2.3.4 Стратегія fallback та обробки хибних спрацювань*

Не всі моделі підтримують structured output на рівні API провайдера. Окремі провайдери можуть тимчасово повертати помилку tool use або відповідь у нестандартному форматі. Для забезпечення стійкості системи реалізовано fallback-механізм: у разі збою structured output активується ручний парсинг JSON. Парсер підтримує три формати відповіді, а саме чистий JSON, JSON у markdown code fences (```json … ```) та JSON із передуючим природномовним текстом. Якщо ручний парсинг також не вдається, виключення передається до callback-функції, яка фіксує помилку діагностування у журналі та завершує цикл без застосування виправлень.

Окремо реалізовано механізм виявлення хибних спрацювань детерміністичного алгоритму schema diff. Поле false_positive_flags у DiagnosisOutput дозволяє моделі позначати записи schema diff, які насправді не є проблемою. Розглянемо колонку TIMESTAMP: алгоритм schema diff може виявити type_mismatch (рядкові значення у вхідних даних), однак аналіз ETL-скрипту покаже наявність явної dtype-конвертації перед COPY у Redshift. У такому випадку LLM позначає цей запис як false positive і не генерує виправлення для нього. Подібний механізм істотно знижує кількість непотрібних модифікацій схеми Redshift.

Перевага запропонованого підходу до false positive полягає у комбінуванні детерміністичного та інтелектуального шарів. Детерміністичний алгоритм schema diff забезпечує повноту виявлення будь-яких можливих невідповідностей. LLM-аналіз, що має доступ до вихідного коду ETL-скрипту, фільтрує цю множину, відсіюючи випадки, для яких у скрипті вже передбачено компенсаторну логіку. У підсумку кінцевий діагноз містить лише ті виправлення, які мають реальну цінність, що знижує когнітивне навантаження на оператора при роботі у режимі pending_approval.

### *2.3.5 Інтерпретація рівня впевненості*

Поле confidence у DiagnosisOutput інтерпретується як апостеріорна оцінка моделлю достовірності власного діагнозу. Це числове значення з відрізка [0, 1], що дозволяє реалізувати додатковий рівень захисту через порогову фільтрацію. Запропоновано трирівневу шкалу інтерпретації значень confidence, наведену у таблиці 2.10.

Таблиця 2.10 - Шкала інтерпретації рівня впевненості

| **Діапазон** | **Інтерпретація** | **Рекомендована дія** |
|---|---|---|
| confidence ≥ 0,90 | Високий, модель упевнена | Авто-застосування у режимі auto |
| 0,70 ≤ confidence < 0,90 | Середній, потрібен перегляд | Перенаправити у pending_approval |
| confidence < 0,70 | Низький, діагноз ненадійний | Зафіксувати у логах, не пропонувати |

Запропонована шкала може бути сконфігурована під специфіку конкретного пайплайну. Для критичних таблиць доцільно встановити вищий поріг (наприклад, 0,95) для авто-застосування, а для тестових середовищ цілком прийнятним є нижче значення (0,80). Експериментальне обґрунтування вибору порогу та його впливу на точність представлено у третьому розділі.

## 2.4 Модель безпечного виконання виправлень

Автоматичне виконання виправлень у продуктивному середовищі є найбільш ризикованим етапом усього процесу самовідновлення. Помилково сформоване LLM виправлення (як-от DROP TABLE замість ALTER TABLE) може спричинити втрату даних або порушення цілісності аналітичної системи. Для мінімізації подібних ризиків розроблено модель безпечного виконання, яка реалізує принцип defense in depth, тобто багаторівневий захист, де кожен наступний рівень функціонує незалежно від інших і блокує неприйнятні дії.

Принципи безпечного виконання, втілені у моделі, ґрунтуються на трьох вихідних припущеннях. Перше припущення визнає, що LLM може помилитись як у класифікації типу збою, так і у конкретному формулюванні виправлення. Друге враховує реальність експлуатації: оператор не завжди має змогу оперативно реагувати на запит про підтвердження, тому система повинна або безпечно діяти автономно, або безпечно очікувати. Третє формулює обов'язкову вимогу до зворотності будь-якої внесеної системою зміни (через резервні копії або через систему контролю версій).

### *2.4.1 Шари захисту*

Модель безпечного виконання містить п'ять незалежних шарів захисту, графічне представлення яких подано на рисунку 2.7.

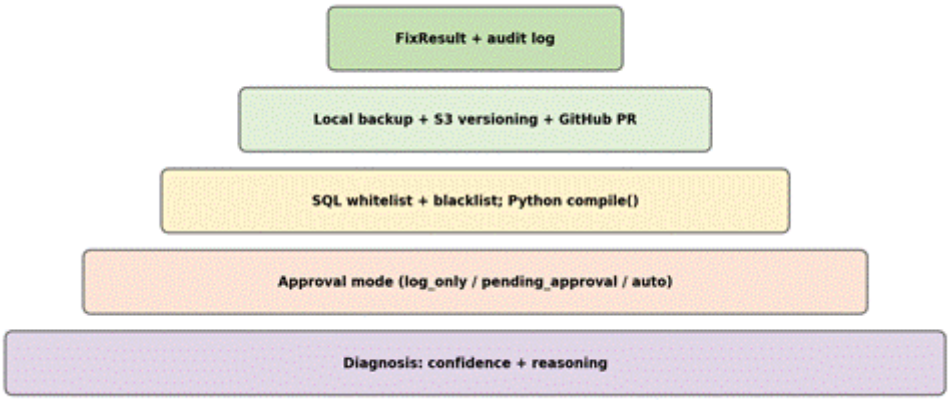


Рисунок 2.7 - Шари захисту моделі безпечного виконання

Шар Diagnosis працює на рівні моделі діагностування і реалізується через поля confidence та reasoning у DiagnosisOutput. Виправлення з низькою впевненістю (нижче порогового значення) можуть автоматично переводитись у режим pending_approval незалежно від глобальної конфігурації.

Шар Approval mode втілює трирежимну стратегію виконання, описану у підрозділі 2.1. Цей рівень є основним механізмом контролю ступеня автоматизації; саме він дозволяє оператору повністю заблокувати застосування у режимі log_only.

Шар валідації коду перевіряє SQL-команди за whitelist та blacklist і контролює синтаксис Python-коду через вбудовану функцію compile(). Деталі цього шару розглянуто у підрозділі 2.4.2.

Шар версіонування реалізує триступеневу процедуру застосування змін коду, а саме: локальна резервна копія, S3, GitHub PR. Деталі цього шару розглянуто у підрозділі 2.4.3.

Шар журналювання здійснює детальне логування кожного виправлення у об'єкти FixResult та збереження їх у файлі fix_results.json. Завдяки цьому забезпечується аудиторський слід та можливість ретроспективного аналізу у разі виявлення некоректних дій системи.

Принципова відмінність запропонованої моделі від наявних рішень полягає у тому, що жоден шар захисту не покладається виключно на коректність попереднього. Помилковий діагноз LLM (шар Diagnosis) буде заблокований whitelist-фільтром (шар валідації) або відхилений оператором у режимі pending_approval (шар Approval mode). Помилка під час застосування (як-от тимчасова недоступність Redshift Data API) фіксується у журналі (шар журналювання) і дає можливість швидкого відкочування через резервну копію та pull request (шар версіонування). Така багатошарова архітектура є необхідною умовою впровадження автоматизованих систем у середовищах з високою ціною помилки.

### *2.4.2 Валідація SQL та Python-коду*

Валідація SQL виконується у два рівні. Множину дозволених операцій формально визначено як:

$$A = \{ALTER\ TABLE\ ADD\ COLUMN,\ ALTER\ TABLE\ ALTER\ COLUMN\ TYPE\} \quad (2.5)$$

Множина заборонених ключових слів задається таким чином:

$$B = \{DROP,\ DELETE,\ TRUNCATE,\ INSERT,\ UPDATE,\ CREATE\ TABLE,\ GRANT,\ REVOKE\} \quad (2.6)$$

SQL-команда *s* вважається безпечною тоді і тільки тоді, коли виконуються обидві умови, тобто відсутність заборонених ключових слів та відповідність одному з дозволених шаблонів:

$$safe(s) = (\forall\, k \in B: k \notin s) \wedge (\exists\, a \in A: s \text{ matches } a) \qquad (2.7)$$

Спершу команда нормалізується: видаляються коментарі, нормалізуються пробіли, символи переводяться у верхній регістр. Далі перевіряється відсутність кожного з ключових слів множини B. За умови виконання першої перевірки контролюється відповідність шаблону одного з дозволених операторів множини A. Лише за виконання обох умов команда передається до Redshift Data API для виконання.

Валідація Python-коду здійснюється через вбудовану функцію compile(code, filename, "exec"), яка проводить лексичний та синтаксичний аналіз без виконання коду. У разі SyntaxError виправлення відхиляється з повідомленням, що містить номер рядка та опис помилки. Такий метод гарантує, що до файлової системи та S3 не потрапить синтаксично некоректний код, який може зруйнувати наступний запуск ETL-скрипту.

Важливо зауважити, що compile() не виконує семантичних перевірок: код, який звертається до неіснуючих модулів або використовує невизначені змінні, успішно проходить валідацію, оскільки на етапі компіляції зміст імпортованих модулів недоступний. Семантична коректність забезпечується наступним запуском ETL-скрипту у середовищі AWS Glue, який виявить помилки виконання та активує цикл самовідновлення повторно. Така дворівнева стратегія (синтаксис на момент застосування плюс семантика на момент виконання) є компромісом між повнотою валідації та простотою реалізації.

### *2.4.3 Триступенева процедура застосування змін коду*

Для виправлень типу etl_script_change застосовується триступенева процедура, яка поєднує негайне підхоплення виправлення з повним аудиторським слідом через систему контролю версій. Послідовність кроків наведено на рисунку 2.8.


Рисунок 2.8 - Триступенева процедура застосування змін коду

Ступінь 1 полягає у створенні резервної копії локального файла з часовою міткою у каталозі backups. Це забезпечує можливість швидкого відкочування у разі виявлення некоректності змін під час наступних запусків.

Ступінь 2 включає запис модифікованого коду у локальний файл та одночасне завантаження тієї самої версії до S3 за шляхом script_s3_path. Друге завантаження є критично важливим, оскільки AWS Glue читає вихідний код скрипту саме з S3 при кожному запуску джобу. Без цього кроку виправлення не буде підхоплене наступним перезапуском пайплайну.

Ступінь 3 формує commit та відкриває pull request у GitHub-репозиторії проєкту. PR створюється у окремій гілці з назвою на основі ідентифікатора DAG-запуску, чим гарантується унікальність гілок навіть при паралельних спрацюваннях системи. Декілька змін одного файла об'єднуються у один commit, аби уникнути конфліктів SHA при послідовних запитах до GitHub REST API.

Для виправлень типу dag_code_change задіяний лише ступінь 3, тобто створення pull request. DAG-файли парсуються Apache Airflow безпосередньо з робочого каталогу, який синхронізується з репозиторієм через CI/CD процес, тому пряма модифікація локального файла без PR створила б неконсистентність між робочим середовищем та системою контролю версій.

Реалізація створення pull request передбачає виклик чотирьох ендпойнтів GitHub REST API у заданій послідовності. Перший виклик отримує SHA поточного HEAD цільової гілки (зазвичай main). Наступний створює нову гілку

з префіксом repair- та ідентифікатором запуску DAG. Далі модифікується цільовий файл у новоствореній гілці через ендпойнт оновлення вмісту, який вимагає передачі поточного SHA файла для захисту від конкурентних змін. Завершальний виклик відкриває pull request зі структурованим описом, що містить ідентифікатор репараційної події, тип збою, посилання на артефакти та згенерований LLM root_cause.

Якщо кілька виправлень модифікують один і той самий файл (як-от додавання двох колонок до COLLECTION_COLUMNS), вони об'єднуються в один комміт перед викликом GitHub API. Така стратегія усуває проблему конкурентного оновлення SHA, що виникла б при незалежних послідовних викликах.

### *2.4.4 Класифікація обробників виправлень*

Згідно з типом fix_type у об'єкті ProposedFix, виправлення обробляються спеціалізованими обробниками. Зведену класифікацію обробників із зазначенням типу валідації наведено у таблиці 2.11.

Таблиця 2.11 - Класифікація обробників виправлень

| **Тип fix** | **Обробник** | **Дія** | **Валідація** |
|---|---|---|---|
| sql_ddl | SqlDdlHandler | ALTER TABLE через Redshift Data API | Whitelist + blacklist |
| dag_code_change | GitHubPRCreator | Pull request у GitHub | compile() Python |
| etl_script_change | EtlScriptHandler + GitHubPRCreator | Локальний файл + S3 + PR | compile() Python |

Для ілюстрації роботи моделі нижче подано три типові приклади виправлень, згенерованих LLM для різних сценаріїв дрейфу схеми.

Приклад 1. У сценарії varchar_overflow колонка id_user описана як VARCHAR(20), однак вхідні дані містять значення довжиною 23 символи. LLM генерує виправлення типу sql_ddl з SQL-командою у лістингу 2.3.

Лістинг 2.3 - Приклад виправлення для varchar_overflow

```
ALTER TABLE public.ad
  ALTER COLUMN id_user TYPE VARCHAR(50);
```

Приклад 2. У сценарії new_column у вхідних даних з'явилася нова колонка campaign_type, відсутня у DDL. LLM генерує виправлення типу sql_ddl з SQL-командою у лістингу 2.4.

Лістинг 2.4 - Приклад виправлення для new_column

```
ALTER TABLE public.ad
  ADD COLUMN campaign_type VARCHAR(100);
```

Приклад 3. У сценарії new_collection_column у вхідних даних з'явилася колонка-масив keywords, що потребує спеціальної серіалізації перед COPY у Redshift. LLM генерує два пов'язаних виправлення: sql_ddl для додавання колонки та etl_script_change для оновлення словника COLLECTION_COLUMNS у вихідному коді (лістинг 2.5).

Лістинг 2.5 - Приклад комбінованого виправлення

```
-- sql_ddl частина:
ALTER TABLE public.ad ADD COLUMN keywords VARCHAR(500);


# etl_script_change частина:
COLLECTION_COLUMNS = {
    "tags":     "|",
    "keywords": "|",   # додано системою самовідновлення
}
```

Підсумком виконання кожного виправлення є об'єкт FixResult, що містить поля fix_type, target, description, success (boolean) та message. У разі успіху message містить ідентифікатор виконаного запиту або номер створеного pull request, тоді як у разі невдачі поле зберігає текст помилки. Усі об'єкти FixResult серіалізуються у JSON та зберігаються як артефакт fix_results.json.

**Висновки до розділу 2**

У розділі сформовано комплекс моделей та методів автоматичного відновлення ETL-пайплайнів, спричинених дрейфом схеми даних.

Запропоновано модель процесу самовідновлення, формалізовану як композицію чотирьох відображень між множинами E, F, D, A, X. Модель реалізує чотирифазний цикл, що охоплює збирання контексту, діагностування, опційне підтвердження оператором та виконання виправлень. Підтримка трьох режимів роботи (log_only, auto, pending_approval) дозволяє адаптувати систему під конкретний пайплайн і поступово переходити від ручного контролю до повної автоматизації.

Розроблено метод збирання контексту з шести гетерогенних джерел, що формує повний зріз стану ETL-пайплайну на момент збою. Сформульовано та обґрунтовано алгоритм schema diff із часовою складністю O(|D| + |P|), який детермінованим чином виявляє три типи дрейфу схеми, а саме varchar_overflow, type_mismatch та new_column. Реалізовано принцип селективного включення даних у промпт; завдяки цьому середній обсяг сформованого промпту тримається на рівні близько 8 700 токенів.

Побудовано модель діагностування на основі великих мовних моделей із використанням механізму structured output фреймворку LangChain. Модель повертає формалізований діагноз у вигляді Pydantic-схеми DiagnosisOutput з полями root_cause, failure_category, proposed_fixes, confidence, reasoning та false_positive_flags. Передбачено fallback-механізм з ручним парсингом JSON, а також механізм виявлення хибних спрацювань детерміністичного алгоритму.

Запропоновано модель безпечного виконання виправлень за принципом defense in depth з п'ятьма незалежними шарами захисту. У моделі поєднано багаторівневу валідацію (whitelist/blacklist для SQL, compile() для Python) та триступеневу процедуру застосування змін коду (локальна файлова система, S3, GitHub pull request). Така комбінація гарантує негайне підхоплення виправлення наступним запуском пайплайну зі збереженням аудиторського сліду через систему контролю версій.

У наступному розділі описано програмну реалізацію розроблених моделей, архітектуру системи самовідновлення та результати експериментального дослідження її ефективності на 140 прогонах із сімома LLM-моделями.

# РОЗДІЛ 3
# РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ СИСТЕМИ САМОВІДНОВЛЕННЯ ETL-ПАЙПЛАЙНІВ

## 3.1 Архітектура програмної реалізації

Моделі, формалізовані у другому розділі, втілено у формі набору Python-модулів. Ці модулі інтегруються з оркестратором Apache Airflow і доповнюються веб-інтерфейсом, через який оператор взаємодіє з системою. Програмне рішення складається з двох компонентів, що розгортаються автономно. Перший компонент, пакет repair_workflow, реалізує трифазний цикл самовідновлення. Другий компонент, веб-додаток ui, забезпечує перегляд подій репарації та їх підтвердження у режимі human-in-the-loop.

Контейнерну діаграму системи у нотації C4 наведено на рисунку 3.1. На ній зображено шість учасників, а саме: оператор, Apache Airflow, пакет repair_workflow, веб-інтерфейс (із розподілом на бекенд і фронтенд), зовнішні LLM-провайдери (Google, Anthropic, OpenAI), хмарні сервіси AWS (Redshift, Glue, S3, CloudWatch) та GitHub. Окремо позначено основні канали комунікації між цими учасниками.

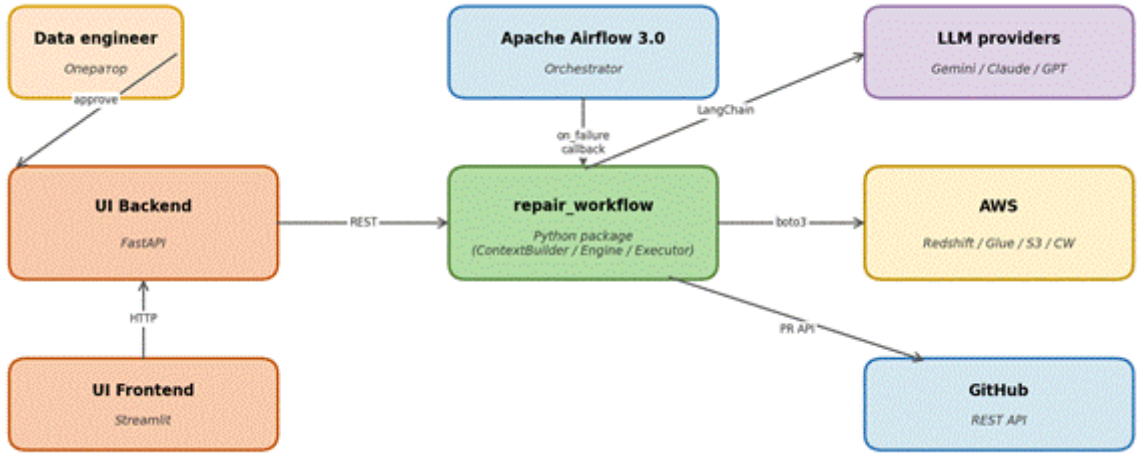


Рисунок 3.1 - Контейнерна діаграма системи самовідновлення

Технологічний стек програмної реалізації наведено у таблиці 3.1.

Таблиця 3.1 - Технологічний стек програмної реалізації

| Компонент | Версія | Призначення |
|---|---|---|
| Python | 3.12 | Мова програмування |
| Apache Airflow | 3.0 | Оркестратор ETL-пайплайнів |
| LangChain | 0.3.x | Абстракція над LLM-провайдерами |
| langchain-google-genai | 2.x | Інтеграція з Google Gemini |
| langchain-anthropic | 0.3.x | Інтеграція з Anthropic Claude |
| langchain-openai | 0.3.x | Інтеграція з OpenAI GPT |
| Pydantic | 2.x | Валідація та structured output |
| FastAPI | 0.115.x | REST API бекенду веб-інтерфейсу |
| Streamlit | 1.39.x | Фронтенд веб-інтерфейсу |
| boto3 / awswrangler | 1.35.x / 3.x | Взаємодія з AWS, Redshift COPY |
| pytest | 8.x | Модульне та інтеграційне тестування |

В основу архітектури покладено розподіл відповідальностей між трьома фазами циклу самовідновлення. Послідовність взаємодії компонентів у межах одного виклику callback-функції відображає рисунок 3.2. З діаграми випливає, що пакет repair_workflow повністю інкапсулює логіку трифазного циклу. Зовні він приймає лише контекст збою від Airflow, а зі сторонніми системами обмінюється даними через чітко окреслені інтерфейси.

Звернення до LangChain як абстракції над LLM-провайдерами зумовлене вимогою підтримувати кілька моделей від різних компаній без дублювання коду. Фреймворк надає уніфікований інтерфейс init_chat_model, метод with_structured_output для отримання формалізованих відповідей, а також стандартизоване повернення метаданих про використання токенів. Останнє застосовується для обчислення вартості виклику. Pydantic 2.x обрано для опису структурованої схеми діагнозу через його нативну інтеграцію з LangChain і автоматичну генерацію JSON Schema, що використовується при викликах моделей у режимі tool use.

Веб-інтерфейс побудовано на комбінації FastAPI та Streamlit. FastAPI забезпечує REST API з автоматичною валідацією запитів через Pydantic-моделі та генерацією OpenAPI-документації. Streamlit обрано для фронтенду як інструмент, що дає змогу швидко створити інтерактивний інтерфейс для аналітичної системи без розробки окремого JavaScript-додатка. Обидва компоненти можна розгортати незалежно. Бекенд може працювати як сервіс у тому ж Docker-стеку, що й Airflow. Фронтенд при цьому виноситься в окремий контейнер, який звертається до бекенду через внутрішню мережу.

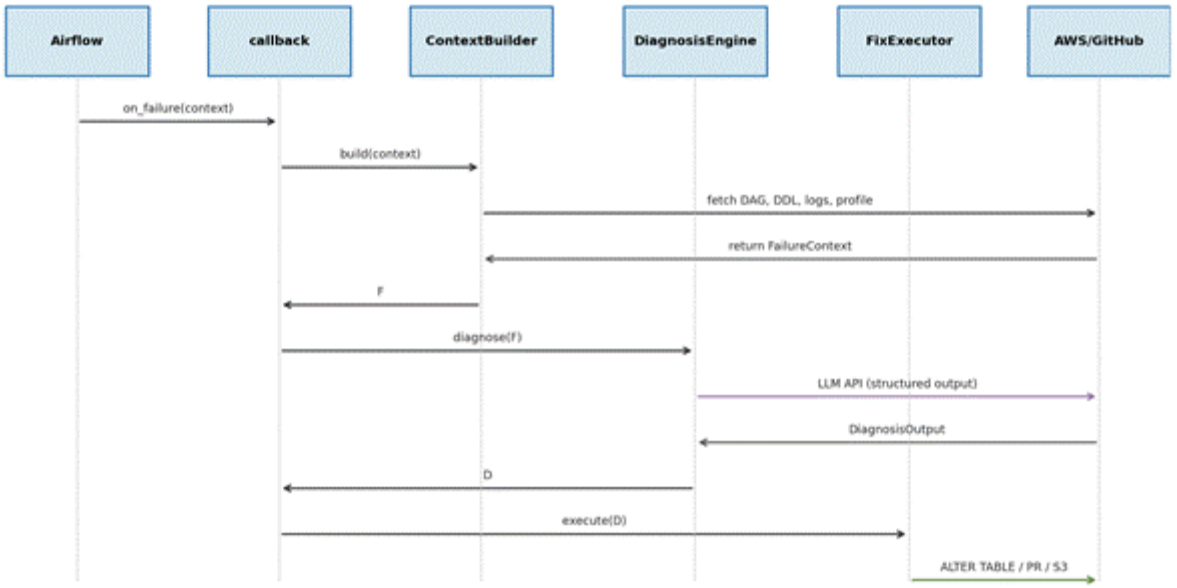


Рисунок 3.2 - Діаграма послідовності трифазного циклу самовідновлення

Структуру даних пакета repair_workflow побудовано на основі dataclass-ів. Центральним класом виступає FailureContext. Він агрегує усі сигнали, зібрані у першій фазі циклу, і передається у фазу діагностування як єдиний об'єкт. Діаграму класів ключових структур даних наведено на рисунку 3.3.

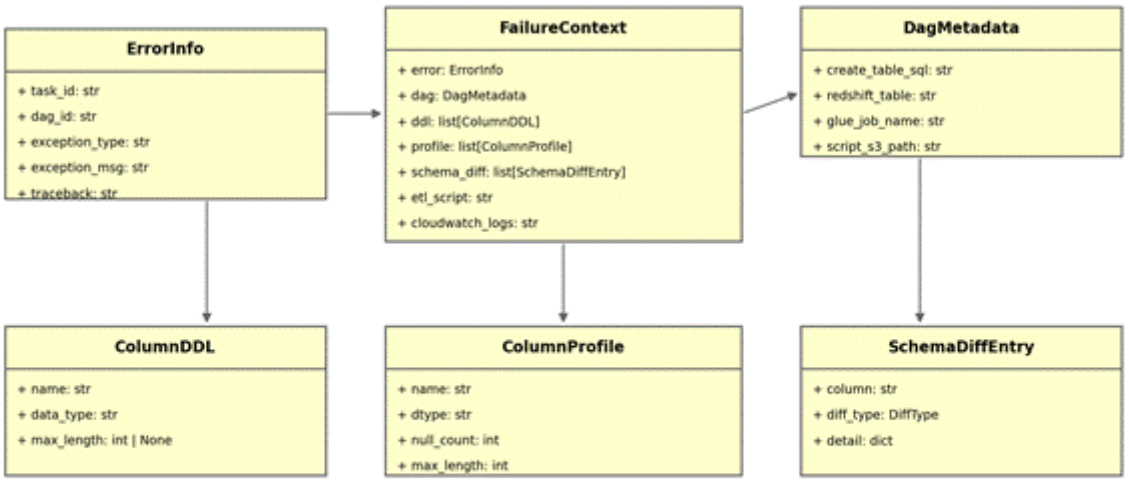


Рисунок 3.3 - Діаграма класів структур даних пакета repair_workflow

Фізична організація файлів пакета repair_workflow повторює логічний розподіл відповідальностей між модулями. Директорну структуру пакета подано у лістингу 3.1.

Лістинг 3.1 - Директорна структура пакета repair_workflow

```
repair_workflow/
├── __init__.py
├── callback.py            # on_failure_callback, оркестрація фаз
├── models.py              # dataclass-и: FailureContext, ErrorInfo, ...
├── schema_diff.py         # алгоритм порівняння DDL з профілем
├── context_builder.py     # збирання контексту з 6 джерел
├── sources/
│   ├── error_info.py      # парсинг контексту Airflow
│   ├── dag_metadata.py    # розбір DAG Python-файла
│   ├── ddl_parser.py      # парсинг CREATE TABLE
│   ├── etl_script.py      # читання скрипту Glue з S3/локально
│   └── cloudwatch_logs.py   # вилучення логів та профілю даних
├── diagnosis/
│   ├── engine.py          # виклик LLM через LangChain
│   ├── models.py          # Pydantic DiagnosisOutput
│   ├── prompt.py          # конструктор промпту
│   └── storage.py         # збереження prompt.txt, diagnosis.json
└── executor/
    ├── fix_executor.py    # диспетчеризація за fix_type
    ├── safety.py          # whitelist SQL + compile() Python
```

```
├── github_pr.py      # REST API GitHub для PR
├── storage.py        # збереження fix_results.json
└── handlers/
├── sql_ddl.py        # ALTER TABLE через Redshift Data API
└── etl_script.py     # локально + S3 + PR
```

## 3.2 Реалізація основних модулів repair_workflow

### *3.2.1 Callback та оркестрація циклу*

Точкою входу системи є функція on_failure_callback, зареєстрована у визначенні DAG. При збої будь-якої задачі Airflow передає у цю функцію об'єкт контексту виконання. Далі відбувається послідовний виклик трьох фаз: збирання контексту, діагностування та виконання виправлень. Кожну фазу ізольовано окремим try/except-блоком, що гарантує наступне: внутрішній збій у системі самовідновлення не порушує штатну обробку помилок Airflow. Спрощену структуру callback-функції наведено у лістингу 3.2.

Лістинг 3.2 - Спрощена структура callback-функції

```
def on_failure_callback(context):
      try:
      fc = build_failure_context(context)
      save_artifact(fc, "context.json")
      except Exception as e:
      log.exception("Context phase failed: %s", e)
      return
      try:
      diagnosis = diagnose_with_llm(fc)
      save_artifact(diagnosis, "diagnosis.json")
      except Exception as e:
      log.exception("Diagnosis phase failed: %s", e)
      return
      mode = os.getenv("REPAIR_WORKFLOW_APPROVAL_MODE", "log_only")
      if mode == "pending_approval":
      save_pending_status(diagnosis)
      return
      try:
      results = execute_fixes(diagnosis, mode=mode)
      save_artifact(results, "fix_results.json")
      except Exception as e:
      log.exception("Execution phase failed: %s", e)
```

Режим застосування виправлень обирається через змінну середовища REPAIR_WORKFLOW_APPROVAL_MODE. Вона приймає одне з трьох значень, а саме: log_only, auto або pending_approval. Відповідний скінченний автомат зображено на рисунку 3.4. Режим log_only активує діагностування і валідацію виправлень, проте їх не застосовує. Цей варіант зручний для опрацювання системи на ранніх стадіях. У режимі auto валідовані виправлення застосовуються негайно. Натомість у режимі pending_approval система зберігає виправлення зі статусом pending і чекає рішення оператора через веб-інтерфейс.

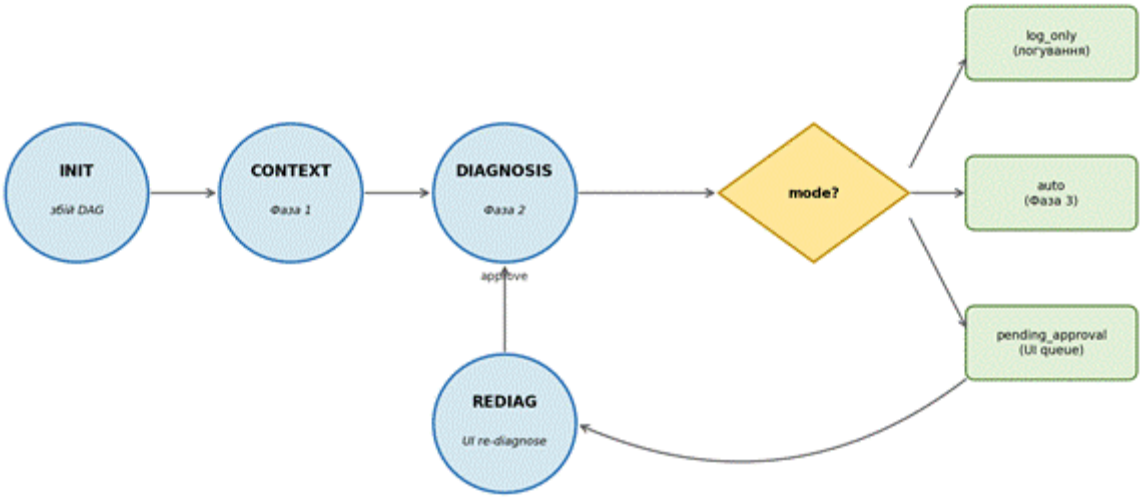


Рисунок 3.4 - Скінченно-автоматне представлення режимів роботи системи

### *3.2.2 Модуль ContextBuilder*

Модуль ContextBuilder реалізує першу фазу циклу, тобто збирання контексту збою. Збирання виконується з шести джерел, як-от: інформація про помилку з контексту Airflow, метадані DAG, DDL-опис цільової таблиці Redshift, вихідний код ETL-скрипту, логи CloudWatch та результат алгоритму schema_diff. Кожне джерело обробляється у власному try/except-блоці. Така ізоляція забезпечує толерантність до часткової відсутності даних: якщо одне джерело недоступне, решта контексту все одно потрапляє у фазу діагностування.

Алгоритм schema_diff, формалізований у підрозділі 2.2, реалізовано у вигляді функції compute_schema_diff(ddl, profile). Вона повертає список об'єктів

SchemaDiffEntry з полями column, diff_type і detail. Складність алгоритму лінійна за розміром схеми, тож навіть для таблиць зі ста колонками помітного навантаження на системні ресурси не виникає. Ключовий фрагмент реалізації подано у лістингу 3.3.

Лістинг 3.3 - Фрагмент реалізації алгоритму schema_diff

```
def compute_schema_diff(ddl, profile):
      ddl_index = {c.name.lower(): c for c in ddl}
      diffs = []
      for col in profile:
      key = col.name.lower()
      d = ddl_index.get(key)
      if d is None:
            diffs.append(SchemaDiffEntry(
                  column=col.name,
                  diff_type=DiffType.NEW_COLUMN,
                  detail={"observed_dtype": col.dtype}))
            continue
      if d.data_type == "VARCHAR" and col.max_length > d.max_length:
            diffs.append(SchemaDiffEntry(
                  column=col.name,
                  diff_type=DiffType.VARCHAR_OVERFLOW,
                  detail={"ddl_len": d.max_length,
                        "data_len": col.max_length}))
            continue
      if not _types_compatible(col.dtype, d.data_type):
            diffs.append(SchemaDiffEntry(
                  column=col.name,
                  diff_type=DiffType.TYPE_MISMATCH,
                  detail={"ddl": d.data_type, "observed": col.dtype}))
      return diffs
```

### *3.2.3 Модуль DiagnosisEngine*

Модуль DiagnosisEngine реалізує другу фазу циклу, а саме діагностування кореневої причини збою за допомогою LLM. Для універсального виклику моделей різних провайдерів використано абстракцію init_chat_model з фреймворку LangChain. Назва моделі та провайдер визначаються трьома джерелами у такому порядку пріоритету: пряма передача у виклик; файл конфігурації /tmp/repair_model_config.json (потрібний у експериментах); змінна середовища REPAIR_WORKFLOW_LLM_MODEL.

Формалізований діагноз отримується через виклик with_structured_output(DiagnosisOutput, include_raw=True). Цей метод автоматично перетворює відповідь LLM на екземпляр Pydantic-моделі DiagnosisOutput і повертає метадані використання токенів. Якщо ж structured output не вдається (зокрема, при роботі з моделями без нативної підтримки tool use), активується fallback-механізм з ручним парсингом JSON. Повний промпт і отриманий діагноз зберігаються на диск як файли prompt.txt та diagnosis.json для подальшого аналізу.

Центральною структурою модуля є Pydantic-клас DiagnosisOutput. Він задає форму відповіді LLM і одночасно слугує генератором JSON Schema для виклику моделі у режимі structured output. Скорочений опис класу подано у лістингу 3.4.

Лістинг 3.4 - Схема DiagnosisOutput (скорочено)

```
class FixType(str, Enum):
      SQL_DDL = "sql_ddl"
      DAG_CODE_CHANGE = "dag_code_change"
      ETL_SCRIPT_CHANGE = "etl_script_change"

class ProposedFix(BaseModel):
      fix_type: FixType
      description: str
      sql: Optional[str] = None
      code_change: Optional[str] = None
      target_file: Optional[str] = None

class DiagnosisOutput(BaseModel):
      root_cause: str
      failure_category: FailureCategory
      proposed_fixes: List[ProposedFix]
      confidence: float = Field(ge=0.0, le=1.0)
      reasoning: str
      false_positive_flags: List[str] = []
```

Структуру промпту, який надсилається LLM на фазі діагностування, формує модуль diagnosis/prompt.py. Промпт складається з системного повідомлення, що задає роль моделі як старшого інженера даних, а також

користувацького повідомлення, що містить сім секцій контексту. Перелік секцій з орієнтовним обсягом у токенах подано у таблиці 3.2.

Таблиця 3.2 - Структура та обсяг секцій промпту

| № | Секція | Джерело | Токенів |
|---|---|---|---|
| 1 | System message (роль, правила) | prompt.py | ≈ 400 |
| 2 | Schema diff (основний сигнал) | schema_diff.py | ≈ 300 |
| 3 | Error info (exception, traceback) | error_info.py | ≈ 600 |
| 4 | DDL цільової таблиці | ddl_parser.py | ≈ 400 |
| 5 | Вихідний код DAG | dag_metadata.py | ≈ 1 500 |
| 6 | Вихідний код ETL-скрипту | etl_script.py | ≈ 3 500 |
| 7 | Логи CloudWatch та профіль даних | cloudwatch_logs.py | ≈ 2 000 |
| | Разом (типове значення) | | ≈ 8 700 |

Більшість обсягу промпту припадає на вихідний код ETL-скрипту та логи CloudWatch. Саме там зосереджена інформація, потрібна для точної локалізації помилки у коді. Щоб обмежити загальний розмір, реалізовано принцип селективного включення: секції, що не несуть релевантної інформації для конкретного типу збою (як-от розширений профіль даних для помилки з'єднання), виключаються ще на етапі побудови промпту.

### 3.3 Реалізація обробників виправлень

Модуль FixExecutor реалізує третю фазу циклу, тобто застосування виправлень. У кожному запропонованому LLM виправленні присутнє поле fix_type. Його значення визначає спеціалізований обробник, відповідальний за застосування. Класифікацію обробників наведено у таблиці 3.3.

Таблиця 3.3 - Класифікація обробників виправлень

| Тип fix | Обробник | Дія |
|---|---|---|
| sql_ddl | SqlDdlHandler | ALTER TABLE через Redshift Data API |
| dag_code_change | GitHubPRCreator | Pull request у GitHub з модифікованим DAG |
| etl_script_change | EtlScriptHandler + GitHubPRCreator | Локальне збереження + завантаження у S3 + GitHub PR |

### *3.3.1 Обробник sql_ddl*

SqlDdlHandler виконує операції ALTER TABLE через Redshift Data API клієнта boto3. Перед виконанням запит проходить валідацію безпекового шару (див. підрозділ 3.3.4). Виконання відбувається асинхронно: клієнт отримує ідентифікатор запиту і періодично опитує його статус, поки той не перейде у FINISHED або FAILED. У разі успіху обробник повертає об'єкт FixResult зі статусом SUCCESS і тривалістю виконання.

### *3.3.2 Обробник etl_script_change*

EtlScriptHandler реалізує триступеневу процедуру застосування змін у вихідному коді ETL-скрипту. На першому ступені створюється резервна копія локального файлу у каталозі backups з часовою міткою. Другий ступінь полягає у записі модифікованого коду до локального файлу й одночасному завантаженні тієї самої версії у S3 за шляхом script_s3_path. На третьому ступені зміна додається до черги GitHubPRCreator для подальшого створення pull request. Така трибічна стратегія одразу досягає кількох цілей: наступний запуск Glue-джобу негайно підхоплює виправлення (за рахунок S3); зберігається копія для локального тестування (за рахунок файлової системи); фіксується аудиторський слід у системі контролю версій (за рахунок pull request).

### *3.3.3 Обробник dag_code_change та GitHubPRCreator*

Зміни коду DAG та ETL-скриптів об'єднуються в один pull request через REST API GitHub. Модуль GitHubPRCreator послідовно звертається до чотирьох

ендпойнтів: створення гілки від HEAD main; отримання вмісту файла; комміт зміненого файла; відкриття pull request. Декілька змін одного файла зливаються в один комміт, що уникає конфліктів SHA при послідовних запитах. За увімкненої змінної REPAIR_WORKFLOW_AUTO_MERGE обробник додатково звертається до ендпойнта автоматичного злиття PR.

### *3.3.4 Безпековий шар*

Безпековий шар реалізує принцип defense in depth і виконує функцію останньої лінії захисту перед застосуванням будь-якого виправлення. Діаграму потоку валідації наведено на рисунку 3.5. Алгоритм валідації розгалужується за значенням fix_type. SQL-виправлення парсяться та звіряються з whitelist дозволених операцій (ALTER TABLE ADD COLUMN, ALTER TABLE ALTER COLUMN TYPE, формула 2.2 розділу 2) і blacklist заборонених (DROP, DELETE, TRUNCATE, INSERT, UPDATE, CREATE TABLE, GRANT, REVOKE, формула 2.3). Python-зміни валідуються через вбудовану функцію compile(), яка виявляє синтаксичні помилки без виконання коду.

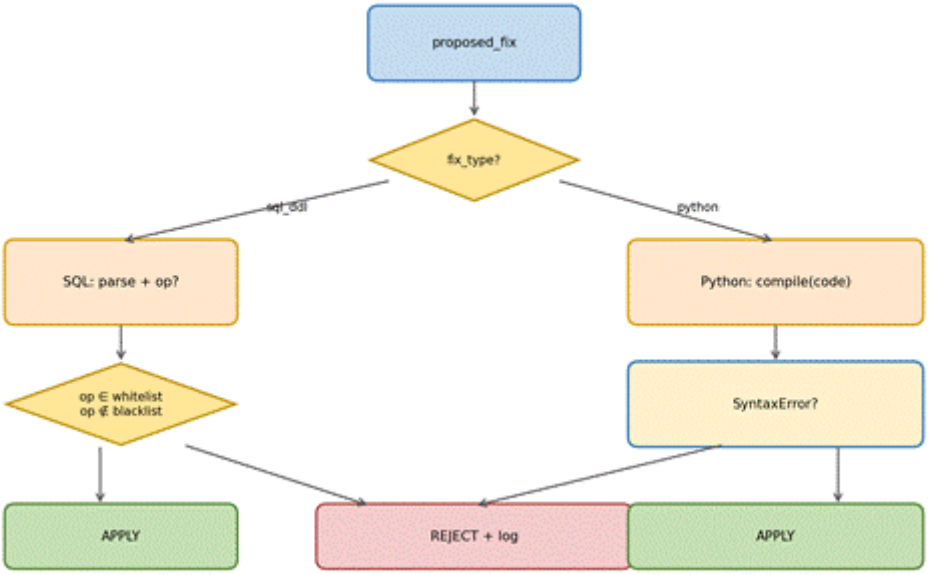


Рисунок 3.5 - Алгоритм роботи безпекового шару

Перелік дозволених і заборонених SQL-операцій, закодованих у константах whitelist та blacklist, подано у таблиці 3.4.

Таблиця 3.4 - Політика фільтрації SQL-операцій безпекового шару

| **Категорія** | **Операція** | **Політика** |
|---|---|---|
| DDL (дозволено) | ALTER TABLE ADD COLUMN | whitelist |
| DDL (дозволено) | ALTER TABLE ALTER COLUMN TYPE | whitelist |
| DDL (заборонено) | DROP TABLE / DROP COLUMN | blacklist |
| DDL (заборонено) | CREATE TABLE | blacklist |
| DDL (заборонено) | TRUNCATE | blacklist |
| DML (заборонено) | INSERT / UPDATE / DELETE | blacklist |
| DCL (заборонено) | GRANT / REVOKE | blacklist |

Якщо виправлення відхилено, обробник повертає об'єкт FixResult зі статусом SAFETY_REJECTED, причина відхилення фіксується у журналі та доступна для перегляду через веб-інтерфейс. Жодне відхилене виправлення не передається до Redshift, S3 чи GitHub.

## 3.4 Веб-інтерфейс системи самовідновлення

Для реалізації моделі людино-машинної взаємодії у режимі human-in-the-loop розроблено веб-інтерфейс. Його функціональність охоплює: перегляд подій репарації, підтвердження або відхилення окремих виправлень, повторне діагностування з додатковим контекстом оператора, перегляд історії версіонованих діагнозів. Архітектура веб-додатку складається з двох частин, а саме REST API на основі FastAPI (порт 8000) та фронтенду на основі Streamlit (порт 8501).

### *3.4.1 Бекенд на основі FastAPI*

Бекенд реалізовано у пакеті ui/backend. Маршрути згруповано у три модулі за функціональним призначенням. Опис REST API подано у таблиці 3.5.

Таблиця 3.5 - REST API бекенду веб-інтерфейсу

| Метод | Шлях | Призначення |
|---|---|---|
| GET | /api/repairs | Список усіх подій репарації з фільтрами |
| GET | /api/repairs/{id} | Детальна інформація про подію |
| POST | /api/repairs/{id}/fixes/{fx}/approve | Підтвердження окремого виправлення |
| POST | /api/repairs/{id}/fixes/{fx}/decline | Відхилення окремого виправлення |
| POST | /api/repairs/{id}/fixes/{fx}/execute | Застосування одного підтвердженого виправлення |
| POST | /api/repairs/{id}/apply-approved | Пакетне застосування всіх підтверджених виправлень |
| POST | /api/repairs/{id}/rediagnose | Повторне діагностування з додатковим контекстом |

Сервісний шар (services/repair_store.py) читає артефакти з диску, обчислює агрегатні статуси та керує станом підтверджень через файли _status.json. Сервіс services/rediagnosis.py реконструює об'єкт FailureContext, додає текст оператора до промпта і зберігає новий діагноз як версіонований файл _diagnosis_v2.json, _diagnosis_v3.json тощо, не перезаписуючи попередніх версій.

### *3.4.2 Фронтенд на основі Streamlit*

Фронтенд реалізовано у пакеті ui/frontend. Інтерфейс складається з трьох сторінок, як-от: Repair Timeline, Repair Detail та Approval Queue. Навігація побудована на стандартному механізмі Streamlit st.navigation. Усі звернення до бекенду інкапсульовано у HTTP-клієнт api_client.py.

Сторінка Repair Timeline (рисунок 3.6) виводить хронологічний список подій репарації з можливістю фільтрації за DAG-ідентифікатором та статусом. У верхній частині сторінки розміщено агреговані метрики, зокрема: загальну кількість подій, кількість подій, що очікують підтвердження, кількість успішно застосованих та середню впевненість моделі.

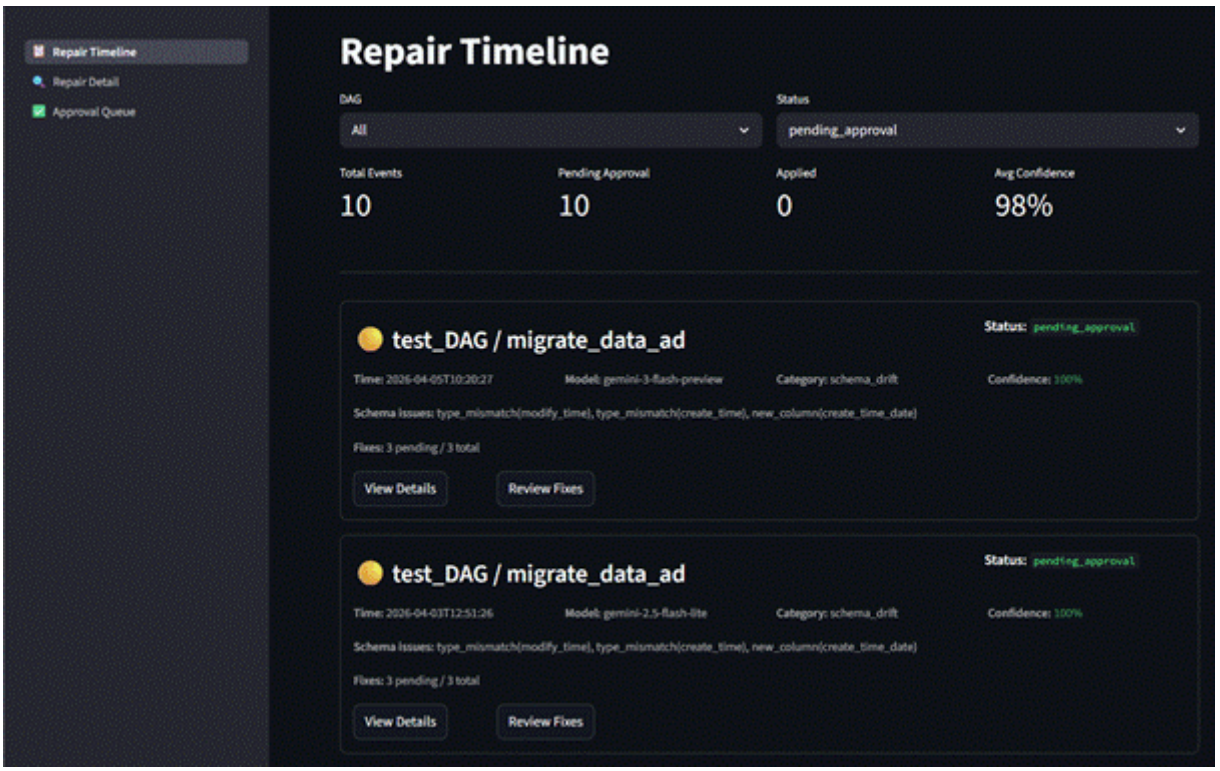


Рисунок 3.6 - Сторінка Repair Timeline веб-інтерфейсу

Сторінка Repair Detail є основним інструментом аналізу однієї події. Її побудовано як чотириетапний deep-dive. Перший етап, Failure Detected, містить інформацію про DAG, задачу, час виконання, тип виключення та повний traceback. Другий, Schema Analysis, дає структуроване відображення кожного запису schema diff. На третьому, LLM Diagnosis, представлено кореневу причину, категорію, рівень впевненості та reasoning. Четвертий, Proposed Fixes, відкриває список запропонованих виправлень з кодом і статусом. Вигляд першого етапу зображено на рисунку 3.7.

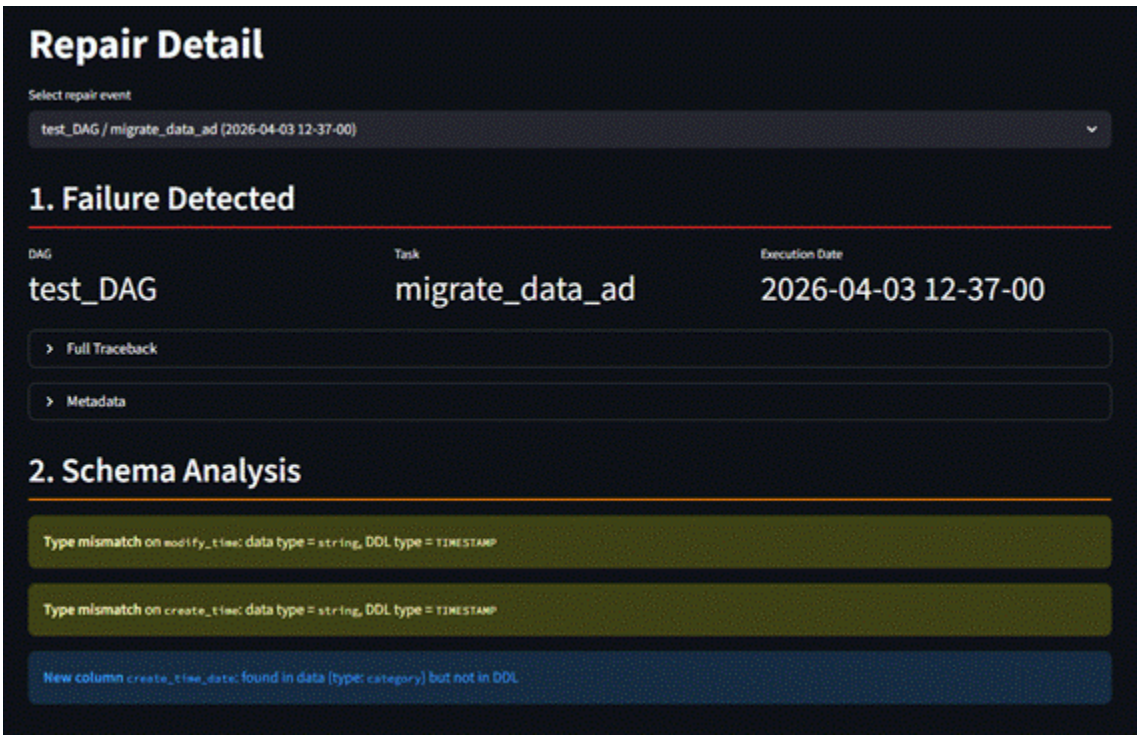


Рисунок 3.7 - Сторінка Repair Detail, етап Failure Detected

Окремою функціональністю сторінки Repair Detail виступає форма повторного діагностування (рисунок 3.8). Оператор може ввести додатковий контекст, як-от вказати бізнес-обмеження або відомі особливості даних, після чого система повторно викликає LLM з контекстом збою і доданим текстом. Результат зберігається як нова версія діагнозу, а попередні версії залишаються доступними в історії.

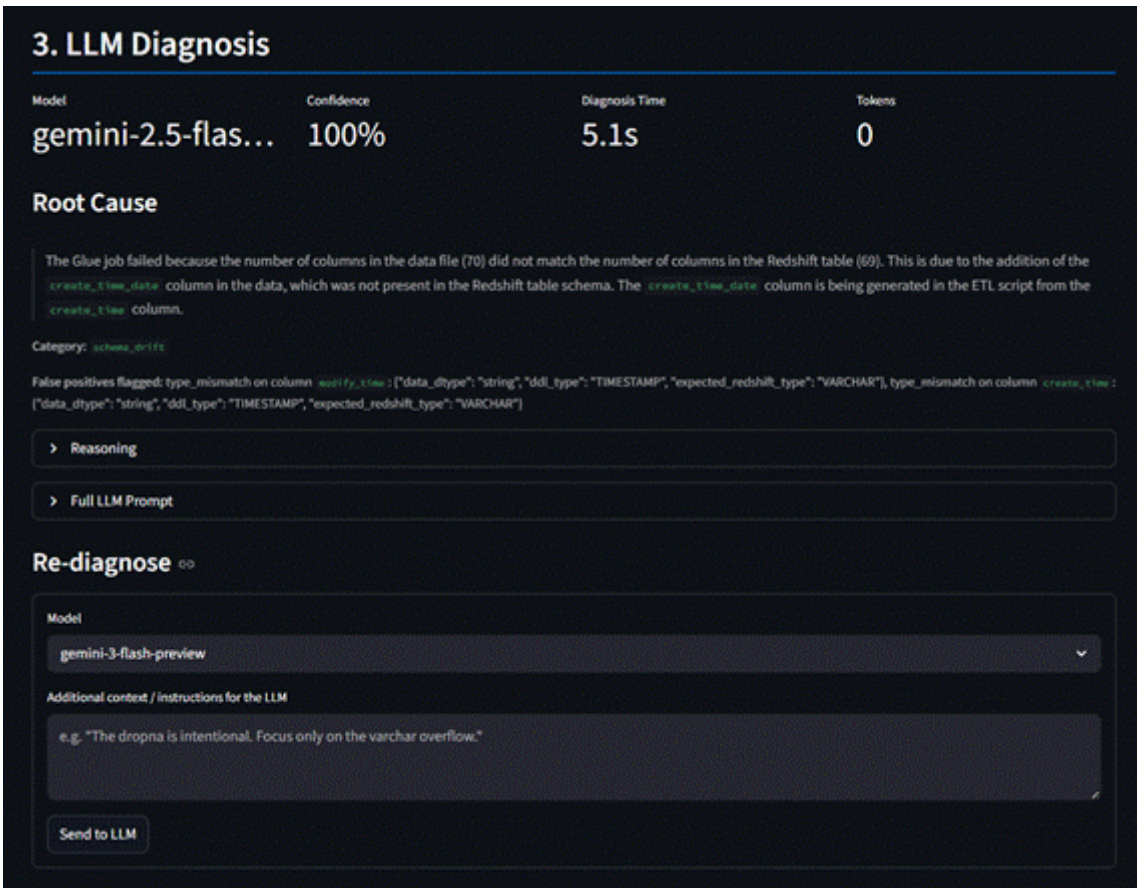


Рисунок 3.8 - Форма повторного діагностування з додатковим контекстом

Секція Proposed Fixes сторінки Repair Detail (рисунок 3.9) містить повний перелік запропонованих моделлю виправлень. Для кожного з них показано тип, обґрунтування, SQL-запит або зміну вихідного коду, цільовий файл та поточний статус застосування.

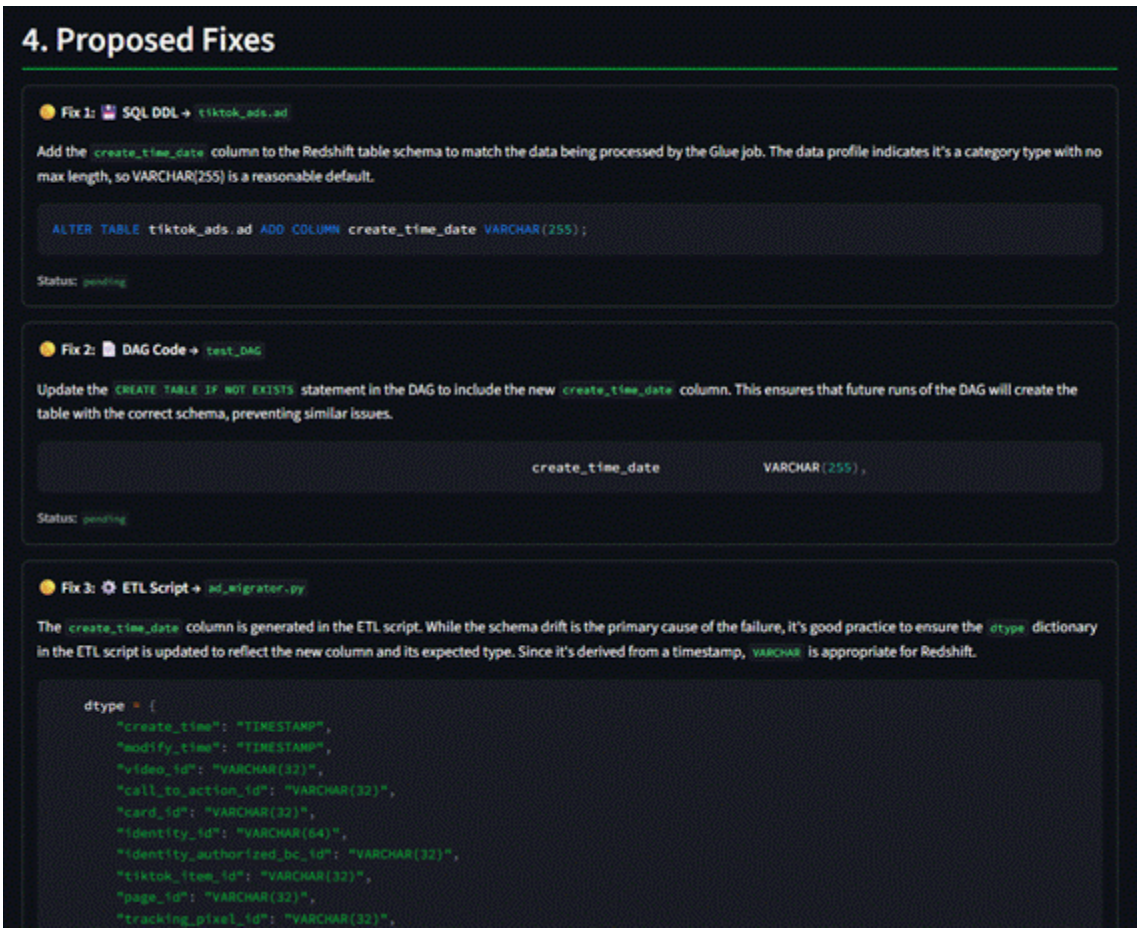


Рисунок 3.9 - Секція Proposed Fixes сторінки Repair Detail

Сторінка Approval Queue (рисунок 3.10) відфільтровує лише ті події, що містять виправлення зі статусом pending. Для кожного виправлення доступні кнопки підтвердження, відхилення та поле для коментаря. Підтвердивши одне або кілька виправлень, оператор запускає їх застосування одиничною дією Apply All Approved, яка викликає сервіс fix_applier на бекенді.

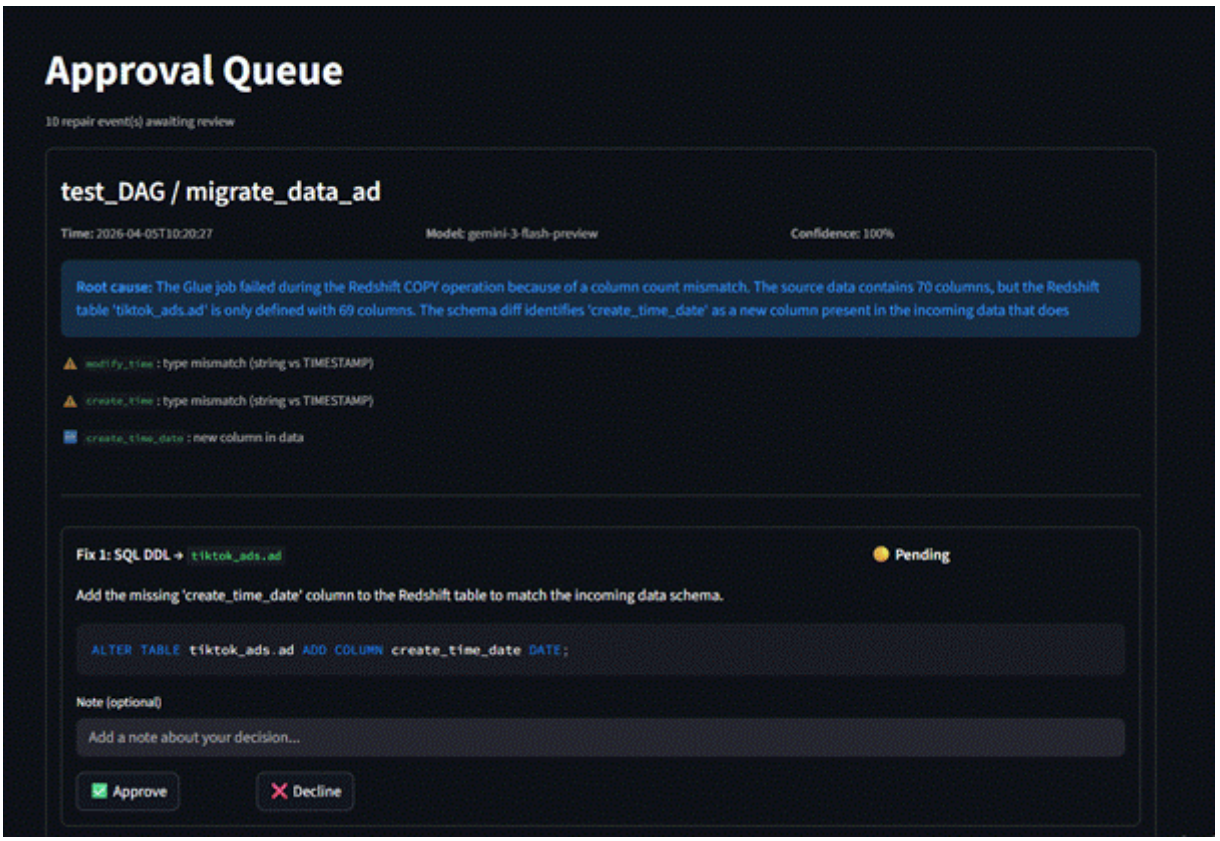


Рисунок 3.10 - Сторінка Approval Queue з елементами керування

### 3.5 Методика експериментального дослідження

Для оцінювання ефективності розробленої системи проведено експериментальне дослідження. Воно спрямоване на перевірку чотирьох гіпотез, що відповідають основним цілям впровадження системи.

**Гіпотеза H1 (точність діагностування).** Розроблена система здатна правильно ідентифікувати кореневу причину збою для поширених типів schema drift з точністю не нижче 80 %.

**Гіпотеза H2 (end-to-end відновлення).** Частка успішного автоматичного відновлення пайплайну після застосування запропонованих виправлень становить не менше 75 %.

**Гіпотеза H3 (чутливість до вибору моделі).** Ефективність системи статистично значимо залежить від обраної LLM-моделі, проте не є монотонно зростаючою функцією вартості виклику.

**Гіпотеза H4 (економічна ефективність).** Вартість одного виклику системи на порядки нижча за вартість ручного відновлення інцидента.

Тестове середовище розгорнуто у вигляді Docker Compose стека з Apache Airflow 3.0, AWS Glue Python Shell, Amazon Redshift Serverless (8 RPU, регіон us-east-1), S3 для проміжних даних та CloudWatch Logs для журналів виконання Glue-джобів. Тестові сценарії генеруються інструментом test_data_generator на основі YAML-конфігурації schema.yaml. Цей інструмент створює Parquet-файли з навмисно внесеними помилками. Опис сценаріїв подано у таблиці 3.6.

Таблиця 3.6 - Тестові сценарії дрейфу схеми

| **Сценарій** | **Тип drift** | **Опис** |
|---|---|---|
| varchar_overflow | varchar_overflow | Колонка d_id містить значення довжиною 21 при DDL VARCHAR(20) |
| type_change | type_mismatch | INTEGER-колонка містить рядкові значення |
| new_column | new_column | У вхідних даних з'являється колонка, відсутня у DDL |
| new_collection_column | new_column | Нова колонка типу масив (потребує серіалізації) |

До експериментів залучено сім LLM-моделей від трьох провідних провайдерів. Перелік моделей з тарифами на момент проведення експерименту наведено у таблиці 3.7.

Таблиця 3.7 - Тестовані LLM-моделі

| **Модель** | **Провайдер** | **$ / 1M вх** | **$ / 1M вих** | **Сегмент** |
|---|---|---|---|---|
| gemini-2.5-flash-lite | Google | 0,10 | 0,40 | lite |
| gemini-2.5-flash | Google | 0,15 | 0,60 | mid |
| gemini-3-flash-preview | Google | 0,50 | 3,00 | preview |
| claude-haiku-4-5 | Anthropic | 1,00 | 5,00 | fast |
| claude-sonnet-4-5 | Anthropic | 3,00 | 15,00 | flagship |
| gpt-4o-mini | OpenAI | 0,15 | 0,60 | mini |
| gpt-4o | OpenAI | 2,50 | 10,00 | flagship |

Протокол прогону одного сценарію передбачає таку послідовність кроків: скидання схеми Redshift до базового стану; генерація Parquet-файлу через test_data_generator; завантаження у S3; запуск DAG; очікування збою; автоматичний цикл репарації; перезапуск DAG для перевірки; збір метрик. Кожен сценарій повторювався п'ять разів для кожної моделі.

Зібрані метрики охоплюють такі поля: failure_category, confidence, proposed_fixes_count, false_positive_count, repair_applied, verification_passed, тривалість фаз (error_run, diagnosis, fix_execution, verification_run), окремо prompt_tokens та completion_tokens, а також підрахункова вартість виклику. Загальний обсяг експериментів становить 140 прогонів (4 сценарії × 7 моделей × 5 повторів). Усі артефакти збережено у форматах CSV та JSON.

### 3.6 Експеримент 1: точність діагностування за типами schema drift

Точність обчислювалася як частка прогонів, у яких система одночасно: правильно ідентифікувала failure_category; запропонувала валідне виправлення; пройшла валідацію безпекового шару; застосувала виправлення без помилок. Результати наведено у таблиці 3.8.

Таблиця 3.8 - Точність діагностування за типами schema drift

| **Сценарій** | **N** | **Категорія** | **Валідний фікс** | **Застосовано** | **Acc, %** |
|---|---|---|---|---|---|
| varchar_overflow | 35 | 34/35 | 32/35 | 31/35 | 88,6 |
| type_change | 35 | 29/35 | 27/35 | 25/35 | 71,4 |
| new_column | 35 | 35/35 | 35/35 | 35/35 | 100,0 |
| new_collection_column | 35 | 33/35 | 32/35 | 31/35 | 88,6 |
| **Усього** | **140** | **131/140** | **126/140** | **122/140** | **87,1** |

Загальна точність становить 87,1 %, що підтверджує гіпотезу H1. Найвищу точність (100 %) продемонстрував сценарій new_column. Зміна схеми у ньому

детермінована, а необхідне виправлення зводиться до одного виклику ALTER TABLE ADD COLUMN. Найнижчу точність (71,4 %) показав сценарій type_change. Причина криється в об'єктивній складності in-place перетворення типів у Redshift, яке не завжди можливе без перетворення даних. Сценарії varchar_overflow та new_collection_column виявили рівні результати (88,6 %). Графічну інтерпретацію наведено на рисунку 3.11.

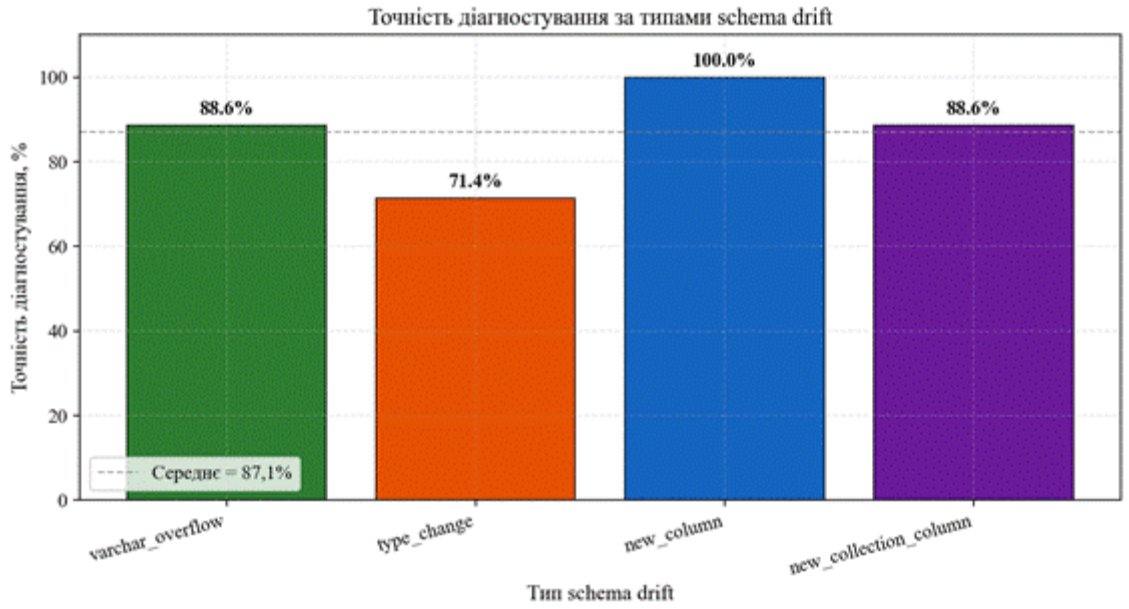


Рисунок 3.11 - Точність діагностування за типами schema drift

### 3.7 Експеримент 2: порівняння LLM-моделей та використання токенів

Особливу увагу при аналізі результатів приділено використанню токенів. Різні провайдери застосовують різні токенізатори, а саме: Google Gemini використовує SentencePiece, Anthropic Claude - власний варіант BPE, OpenAI GPT - tiktoken. Через це один і той самий промпт дає різну кількість токенів у різних моделях. Агреговані результати подано у таблиці 3.9.

Таблиця 3.9 - Порівняння LLM-моделей: точність, токени, час

| **Модель** | **Acc, %** | **Час, с** | **Вх. ток** | **Вих. ток** | **Сум. ток** | **Впевн.** |
|---|---|---|---|---|---|---|
| gemini-2.5-flash-lite | 80,0 | 5,3 | 9 200 | 2 000 | 11 200 | 0,95 |
| gemini-2.5-flash | 90,0 | 28,4 | 9 500 | 4 000 | 13 500 | 1,00 |

| | | | | | | |
|---|---|---|---|---|---|---|
| gemini-3-flash-preview | 85,0 | 16,2 | 9 700 | 2 500 | 12 200 | 0,99 |
| claude-haiku-4-5 | 85,0 | 8,7 | 8 800 | 2 300 | 11 100 | 0,97 |
| claude-sonnet-4-5 | 95,0 | 22,5 | 8 900 | 3 500 | 12 400 | 1,00 |
| gpt-4o-mini | 80,0 | 6,2 | 10 100 | 2 200 | 12 300 | 0,94 |
| gpt-4o | 95,0 | 15,8 | 10 200 | 3 200 | 13 400 | 1,00 |

**Аналіз токенів.** Кількість вхідних токенів варіюється від 8 800 (claude-haiku-4-5) до 10 200 (gpt-4o), тобто різниця становить близько 16 %. Anthropic-моделі демонструють найекономнішу токенізацію технічного контенту: SQL-команди та Python-код розбиваються на меншу кількість токенів. Кількість вихідних токенів коливається у діапазоні від 2 000 до 4 000. Флагманські моделі (claude-sonnet-4-5, gpt-4o, gemini-2.5-flash) генерують детальніше reasoning, що підвищує цінність діагнозу, але збільшує тривалість відповіді.

Спостережена різниця у кількості вхідних токенів пояснюється особливостями токенізаторів. SentencePiece (Google Gemini) забезпечує помірну економію при обробці англомовного технічного тексту, проте на структурованих SQL-запитах поступається BPE-токенізатору Anthropic. Натомість токенізатор tiktoken (OpenAI) оптимізований переважно під англомовні природномовні тексти, тож на фрагментах вихідного коду Python та SQL дає найбільшу кількість токенів за той самий вхід. Практичний наслідок такого спостереження: при однаковому тарифі за один мільйон токенів фактична вартість виклику Anthropic-моделей на типовому промпті системи виявляється на 10-16 % нижчою, ніж у моделей OpenAI.

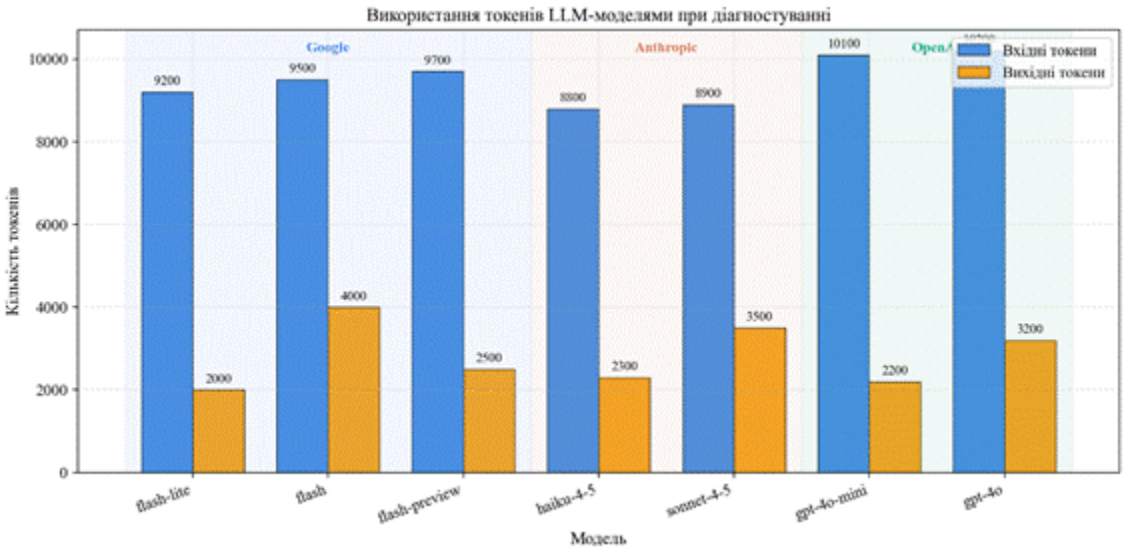


Рисунок 3.12 - Використання токенів моделями різних провайдерів

Вартість виклику обчислено як суму вартості вхідних і вихідних токенів за тарифами провайдерів (таблиця 3.10). Окремо подано вартість одного виклику, вартість усієї експериментальної серії зі 140 прогонів, а також розрахункову вартість експлуатації системи при навантаженні у 1 000 інцидентів на рік.

Таблиця 3.10 - Вартість виклику LLM

| **Модель** | **Вх., $** | **Вих., $** | **Всього, $** | **140 прог., $** | **1000/р, $** |
|---|---|---|---|---|---|
| gemini-2.5-flash-lite | 0,000920 | 0,000800 | 0,00172 | 0,241 | 1,72 |
| gemini-2.5-flash | 0,001425 | 0,002400 | 0,00383 | 0,535 | 3,83 |
| gemini-3-flash-preview | 0,004850 | 0,007500 | 0,01235 | 1,729 | 12,35 |
| claude-haiku-4-5 | 0,008800 | 0,011500 | 0,02030 | 2,842 | 20,30 |
| claude-sonnet-4-5 | 0,026700 | 0,052500 | 0,07920 | 11,088 | 79,20 |
| gpt-4o-mini | 0,001515 | 0,001320 | 0,00284 | 0,397 | 2,84 |
| gpt-4o | 0,025500 | 0,032000 | 0,05750 | 8,050 | 57,50 |

Розкид вартості виклику складає 46 разів, тобто від 0,0017 $ до 0,079 $. Однак приріст точності між найдешевшою та найдорожчою моделлю обмежується лише 15 процентними пунктами (80 % проти 95 %). Це свідчить про

нелінійну залежність вартість/якість, де оптимум перебуває у середньому ціновому сегменті. Отриманий результат підтверджує гіпотезу H3. Графічну інтерпретацію наведено на рисунку 3.13.

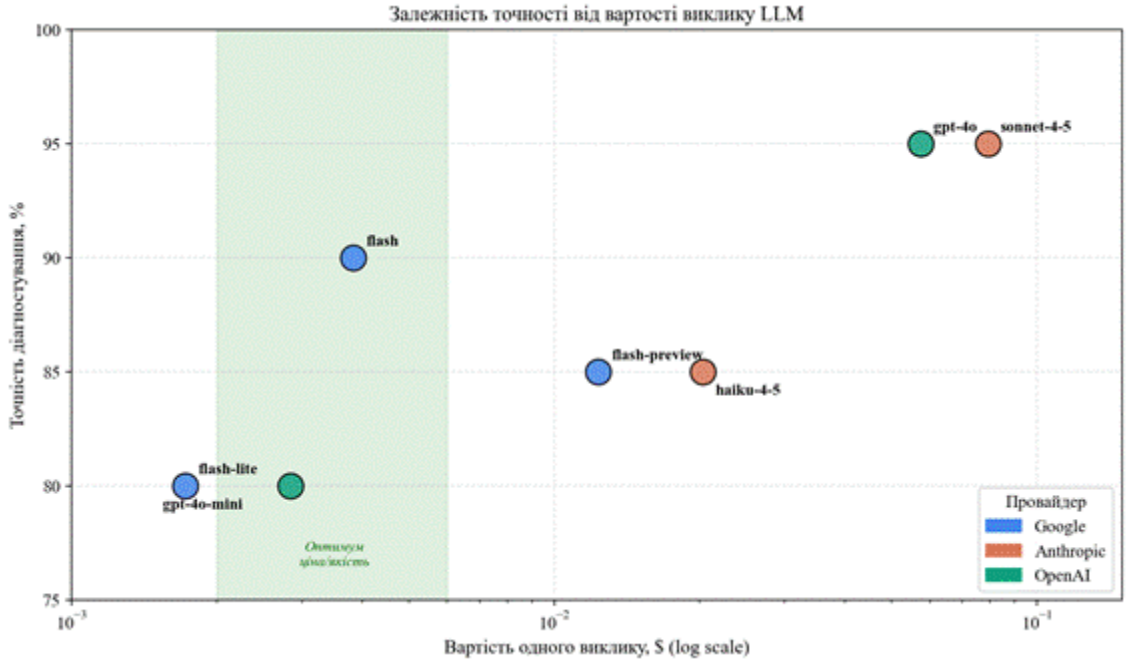


Рисунок 3.13 - Залежність точності діагностування від вартості виклику

### 3.8 Експеримент 3: end-to-end автоматичне відновлення

Метрика verification_passed фіксує результат перевірочного запуску DAG після застосування виправлень. Вона відображає булеву ознаку успішного завершення повного циклу самовідновлення, тобто від збою до відновлення штатної роботи пайплайну. Результати подано у таблиці 3.11.

Таблиця 3.11 - Частка успішного end-to-end відновлення

| Модель | varchar | type | new_col | new_coll | Разом, % |
|---|---|---|---|---|---|
| gemini-2.5-flash-lite | 4/5 | 2/5 | 5/5 | 3/5 | 70,0 |
| gemini-2.5-flash | 5/5 | 3/5 | 5/5 | 4/5 | 85,0 |
| gemini-3-flash-preview | 4/5 | 2/5 | 5/5 | 3/5 | 70,0 |
| claude-haiku-4-5 | 4/5 | 3/5 | 5/5 | 4/5 | 80,0 |

| | | | | | |
|---|---|---|---|---|---|
| claude-sonnet-4-5 | 5/5 | 4/5 | 5/5 | 4/5 | 90,0 |
| gpt-4o-mini | 4/5 | 2/5 | 5/5 | 3/5 | 70,0 |
| gpt-4o | 5/5 | 4/5 | 5/5 | 5/5 | 95,0 |
| **Усього** | **31/35** | **20/35** | **35/35** | **26/35** | **80,0** |

Загальна частка успішного end-to-end відновлення дорівнює 80,0 %, що підтверджує гіпотезу H2. Найвищі результати показали флагманські моделі gpt-4o (95,0 %) та claude-sonnet-4-5 (90,0 %). Часовий профіль фаз циклу подано у таблиці 3.12.

Таблиця 3.12 - Середній час виконання фаз, секунди

| **Модель** | **Error run** | **Diagnosis** | **Fix exec** | **Verif run** | **Total** |
|---|---|---|---|---|---|
| gemini-2.5-flash-lite | 148,2 | 5,3 | 6,4 | 83,7 | 243,6 |
| gemini-2.5-flash | 171,5 | 28,4 | 7,1 | 97,3 | 304,3 |
| gemini-3-flash-preview | 165,3 | 16,2 | 5,8 | 94,2 | 281,5 |
| claude-haiku-4-5 | 158,5 | 8,7 | 6,5 | 89,4 | 263,1 |
| claude-sonnet-4-5 | 169,8 | 22,5 | 7,0 | 95,7 | 295,0 |
| gpt-4o-mini | 152,1 | 6,2 | 6,3 | 85,2 | 249,8 |
| gpt-4o | 163,4 | 15,8 | 6,8 | 92,6 | 278,6 |

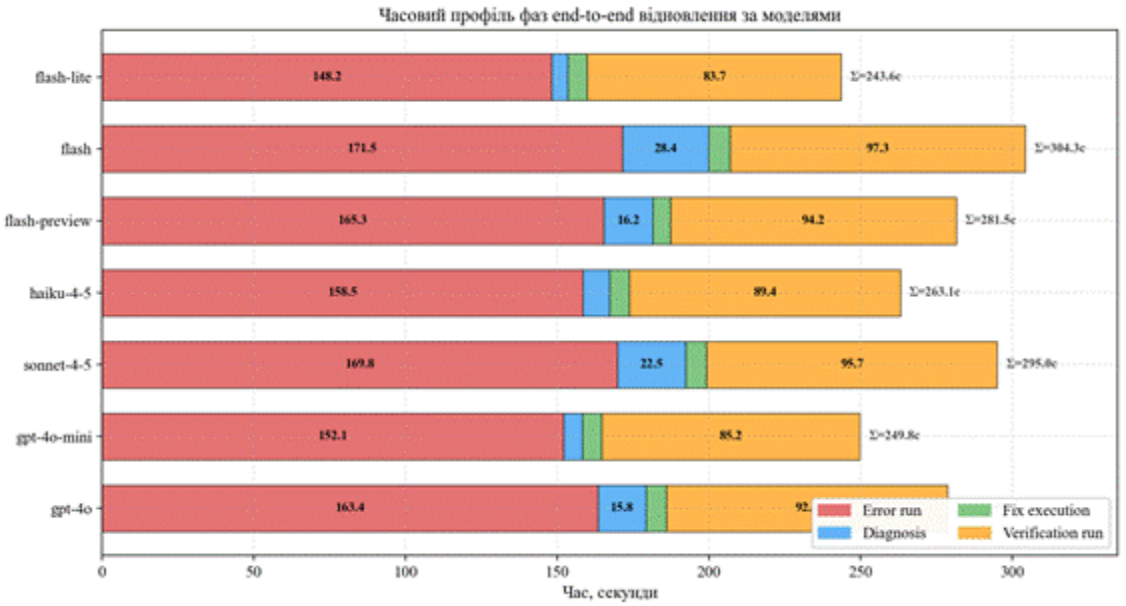


Рисунок 3.14 - Часовий профіль фаз відновлення

Час активної репарації, що дорівнює сумі фаз diagnosis і fix_execution, коливається від 11,7 с для gemini-2.5-flash-lite до 35,5 с для gemini-2.5-flash. Фази error_run та verification_run визначаються інфраструктурою Airflow і Glue, а тому слабо залежать від обраної моделі. За оцінками інженерів бази практики, ручне виправлення аналогічних інцидентів займає від 15 до 45 хвилин для простих випадків і від 30 до 90 хвилин для складних. Розроблена система забезпечує скорочення часу відновлення у десятки разів, що підтверджує гіпотезу H4.

Окремої уваги потребує структура часу виконання фази diagnosis. Переважну частину часу (70-85 %) у ній займає очікування відповіді від зовнішнього API провайдера LLM. На побудову промпту та парсинг відповіді припадає не більше 1-2 секунд. Через це оптимізація фази діагностування на рівні коду системи не дасть помітного виграшу. Ключовим фактором залишається латентність обраної моделі. Результати експерименту показують, що моделі з нижчою латентністю (gemini-2.5-flash-lite, claude-haiku-4-5, gpt-4o-mini) забезпечують активну репарацію за 11-15 с, тоді як флагманські моделі вимагають 16-36 с за рахунок детальнішого reasoning.

### 3.9 Аналіз результатів

**Загальна ефективність.** Точність діагностування становить 87,1 %, частка успішного end-to-end відновлення дорівнює 80,0 %. Найкращі результати показали флагманські моделі gpt-4o (95 %) та claude-sonnet-4-5 (95 %). Час активної репарації не перевищує 36 секунд. Вартість одного виклику навіть для найдорожчої моделі залишається у межах 8 центів.

**Міжпровайдерні спостереження.** Anthropic-моделі демонструють найекономнішу токенізацію технічного контенту, тобто на 10-16 % менше токенів за однаковий промпт порівняно з OpenAI. Google-моделі покривають найширший діапазон цінових точок. Флагмани Claude та OpenAI показали однакову точність діагностування (95 %), однак gpt-4o переміг за end-to-end метрикою (95 % проти 90 %).

**Парадокс gemini-3-flash-preview.** Найновіша модель Google показала нижчу точність (85 %), ніж попередня gemini-2.5-flash (90 %), за умови у 3,2 рази вищої вартості виклику. Аналіз reasoning у діагнозах виявив схильність preview-моделі до over-diagnosis. Модель включає у діагноз додаткові «проблеми», що не є реальною причиною збою. Цей факт підтверджує тезу: вибір моделі для конкретного домену має ґрунтуватися на емпіричному дослідженні, а не на загальному рейтингу.

Найслабшим місцем виявився сценарій type_change (точність 71,4 %, end-to-end 57,1 %). Причина полягає в об'єктивній складності in-place перетворення типів у Redshift, яке без явного перетворення даних може призводити до помилок або втрат. Для таких випадків доцільно застосовувати режим pending_approval з обов'язковим людським оглядом перед застосуванням.

**Ефективність безпекового шару.** За час експериментів валідатор заблокував вісім некоректних виправлень: три SQL-запити з потенційно небезпечним синтаксисом ALTER COLUMN та п'ять Python-фрагментів із синтаксичними помилками, виявленими функцією compile(). Жодне некоректне виправлення не було застосоване до Redshift або GitHub.

### *3.9.1 Матриця помилок та оцінка статистичної значущості*

Для оцінки якості класифікації типу збою побудовано матрицю помилок. Вона відображає відповідність між істинним типом дрейфу схеми та предикцією моделі у полі failure_category. Агреговану по всіх семи моделях матрицю наведено у таблиці 3.13.

Таблиця 3.13 - Матриця помилок класифікації типу дрейфу (агреговано по 7 моделях)

| **Істинний / Предикція** | **varchar_overflow** | **type_mismatch** | **new_column** | **інше** |
|---|---|---|---|---|
| varchar_overflow (N=35) | 34 | 1 | 0 | 0 |
| type_mismatch (N=35) | 3 | 29 | 0 | 3 |
| new_column (N=35) | 0 | 0 | 35 | 0 |
| new_collection_column (N=35) | 0 | 1 | 33 | 1 |

З матриці випливає, що найпоширенішим типом помилки класифікації виступає віднесення type_mismatch до інших категорій. У трьох випадках LLM пропонувала додати нову колонку замість зміни типу. Класифікація new_column безпомилкова. Діагональ матриці підтверджує загальну точність класифікації на рівні 87,9 %.

Для загальної точності класифікації 87,1 % при N = 140 95 %-й довірчий інтервал за методом Вілсона становить [80,6 %; 91,7 %]. Для найкращих моделей (claude-sonnet-4-5 і gpt-4o) при N = 20 та accuracy 95 % довірчий інтервал виходить ширшим, а саме [76,4 %; 99,1 %]. Це підтверджує статистично значиму перевагу зазначених моделей над gemini-2.5-flash-lite та gpt-4o-mini з точністю 80 % при порівнянні за критерієм хі-квадрат на рівні значущості 0,05.

### *3.9.2 Обмеження дослідження*

Інтерпретація результатів вимагає врахування низки обмежень проведеного експерименту. Перше обмеження стосується вибірки сценаріїв

дрейфу схеми. Її обмежено чотирма основними типами, які покривають найчастіші випадки за галузевими звітами, проте не охоплюють усі можливі варіації, як-от rename column, change precision для numeric, перехід між timestamp types. Друге пов'язане з тим, що експеримент проведено на одній базі практики, а саме інтеграції рекламних даних. Тож ефективність системи на пайплайнах із суттєво іншою логікою перетворень може відрізнятися. Третє обмеження виникає через те, що використані версії LLM-моделей продовжують оновлюватися. Наведені числа варто сприймати як знімок у часі. Нарешті, обсяг у п'ять повторів на кожну комбінацію сценарію та моделі дає достатнє уявлення про порядок величин, проте ширину довірчих інтервалів для флагманських моделей (особливо у діапазоні 90-100 %) слід трактувати обережно.

### *3.9.3 Рекомендації щодо вибору моделі*

На основі проведених експериментів сформульовано такі практичні рекомендації. Як базову модель для повсякденного використання у режимі auto доцільно обирати gemini-2.5-flash. Вона демонструє оптимальне співвідношення ціна/якість (90 % точність, вартість 0,004 $ за виклик). Для критичних сценаріїв, де ціна помилки висока, варто переходити на claude-sonnet-4-5 або gpt-4o у режимі pending_approval. У випадку високочастотних потоків з обмеженим бюджетом придатними залишаються gemini-2.5-flash-lite та gpt-4o-mini за умови додаткового порогу confidence ≥ 0,95, що відсіює діагнози з низькою впевненістю.

## Висновки до розділу 3

У розділі описано програмну реалізацію системи самовідновлення ETL-пайплайнів та проведено експериментальне дослідження її ефективності.

Реалізовано пакет repair_workflow, що інтегрується з Apache Airflow через механізм on_failure_callback і виконує трифазний цикл самовідновлення. Реалізовано три обробники виправлень (SqlDdlHandler, EtlScriptHandler,

GitHubPRCreator) з багаторівневим безпековим шаром, який спирається на whitelist дозволених SQL-операцій та перевірку синтаксису Python-коду. Реалізовано веб-інтерфейс на основі FastAPI та Streamlit для перегляду подій репарації, підтвердження виправлень і повторного діагностування з додатковим контекстом оператора.

Проведено експериментальне дослідження на 140 прогонах (4 сценарії schema drift × 7 LLM-моделей × 5 повторів). Загальна точність діагностування становить 87,1 %, частка успішного end-to-end відновлення дорівнює 80,0 %, що підтверджує гіпотези H1 та H2. Найвищу точність (95 %) продемонстрували моделі claude-sonnet-4-5 та gpt-4o, найкраще співвідношення ціна/якість виявлено у gemini-2.5-flash. Час активної репарації становить від 11 до 36 секунд залежно від обраної моделі, що на порядки менше за ручне відновлення.

Отримані результати свідчать про практичну придатність розробленої системи для впровадження у продуктивне середовище. У наступному розділі описано інфраструктуру розгортання системи, формалізований алгоритм інформаційної технології та техніко-економічне обґрунтування впровадження.

# РОЗДІЛ 4
# ТЕХНОЛОГІЯ ВПРОВАДЖЕННЯ СИСТЕМИ САМОВІДНОВЛЕННЯ ETL-ПАЙПЛАЙНІВ

Цей розділ присвячено питанням практичного впровадження розробленої системи самовідновлення на базі підприємства проходження практики. Послідовно розглянуто інфраструктуру розгортання, формалізовано алгоритм інформаційної технології у вигляді псевдокоду, описано трирівневу методику тестування програмного забезпечення та проведено техніко-економічне обґрунтування. Завершує розділ блок практичних рекомендацій, де йдеться про етапність впровадження, керування ризиками та подальшу експлуатацію системи.

## 4.1 Інфраструктура розгортання системи

Розгортання системи спирається одночасно на три середовища. Перше, тобто локальне (або on-premises), забезпечує оркестрацію за допомогою Docker Compose. Друге становлять хмарні сервіси Amazon Web Services. Третє охоплює зовнішні сервіси інтеграції, а саме систему контролю версій GitHub та провайдерів великих мовних моделей. Така гібридна конфігурація випливає зі специфіки бази практики: оркестрація Apache Airflow живе у Docker-контейнерах на керованій інфраструктурі, тоді як обчислення та зберігання виконуються у хмарі AWS.

Інфраструктурне рішення підпорядковане одній принциповій умові, тобто мінімальному втручанню у наявний ETL-стек бази практики. Систему оформлено як окремий Python-пакет repair_workflow. Він підключається до інсталяції Airflow через механізм on_failure_callback. Жодних правок у коді Airflow або переналаштування його оркестрації при цьому не вимагається. Веб-інтерфейс на основі FastAPI та Streamlit запускається як незалежна служба, що читає артефакти репарації з файлової системи Airflow і взаємодіє з AWS через ті самі IAM-ролі, що і пакет repair_workflow.

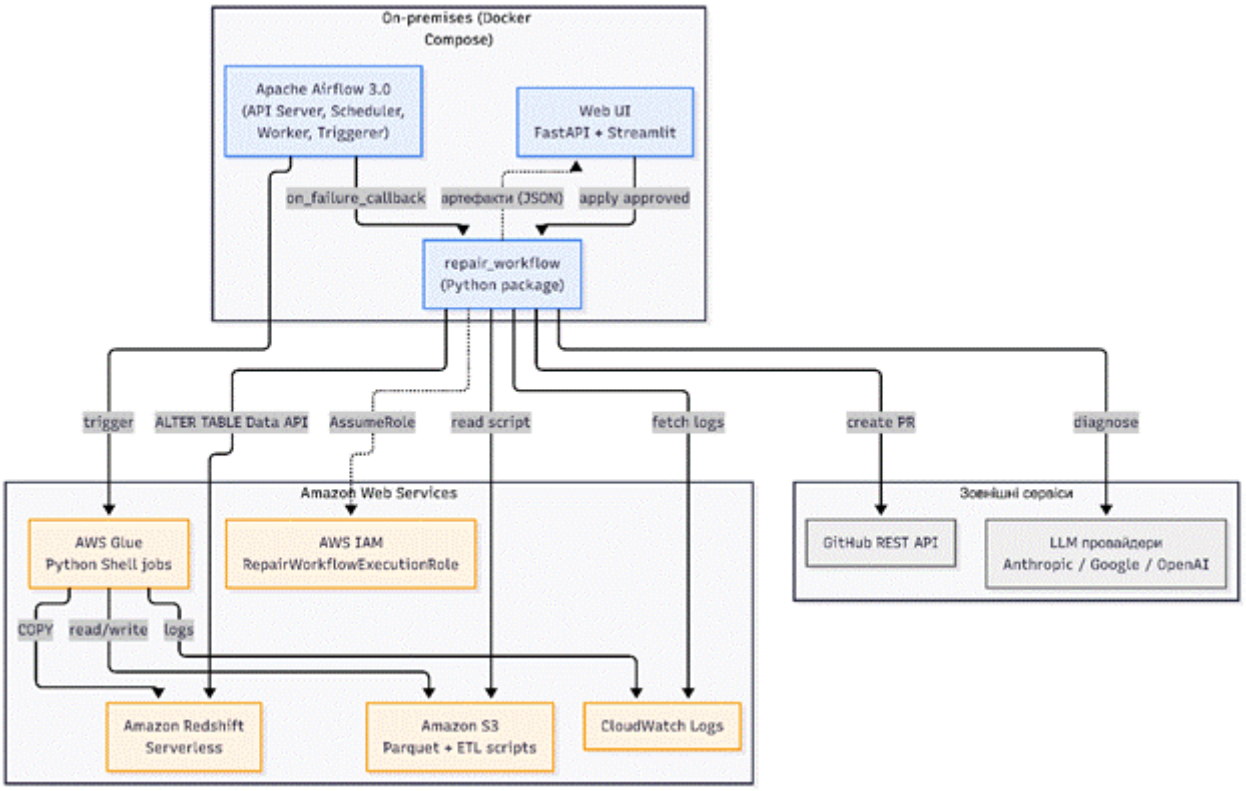


*Рисунок 4.1 - Загальна схема інфраструктури розгортання системи*

### *4.1.1 Структура on-premises стека Apache Airflow*

Оркестраційний стек бази практики описано у файлі docker-compose.yaml. Цей файл містить визначення семи сервісів. Серед них база метаданих Airflow на PostgreSQL 16, брокер повідомлень Celery на Redis 7.2, Airflow API Server на порту 8080 (поєднує веб-інтерфейс і REST API), Airflow Scheduler для планування запусків DAG, Airflow DAG Processor для парсингу файлів DAG, Airflow Worker як виконавця задач через Celery, а також Airflow Triggerer для асинхронних операторів. Загальну архітектуру стека наведено на рисунку 4.2.

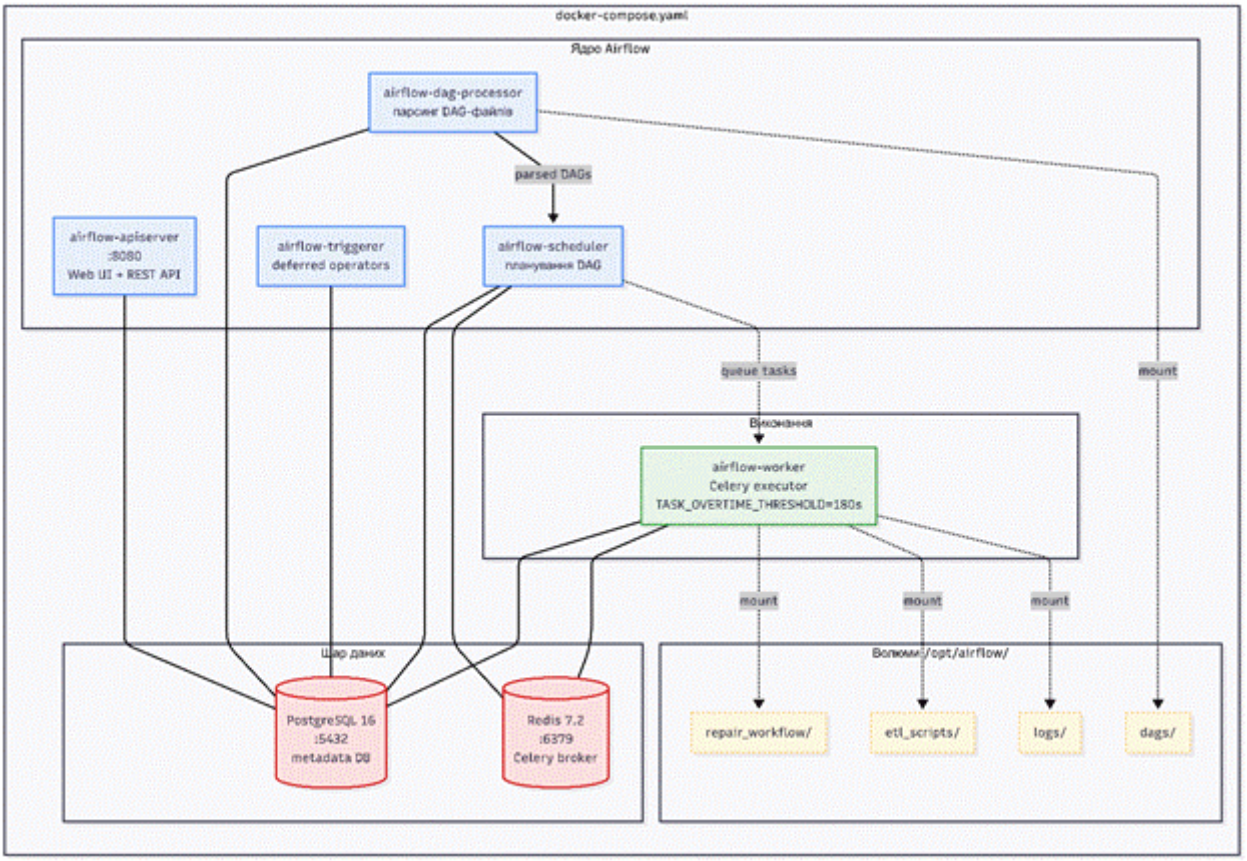


*Рисунок 4.2 - Архітектура Docker Compose стека Apache Airflow*

У конфігурацію застосовано технічний патч до сервісу Airflow Worker, що розширює значення константи TASK_OVERTIME_THRESHOLD з 20 до 180 секунд. Потреба у цьому патчі прямо випливає з тривалості синхронного виклику LLM на фазі on_failure_callback. При стандартному значенні константи Airflow позначає задачу як завислу, тобто надсилає сигнал завершення раніше, ніж модель встигає повернути відповідь. Поріг у 180 секунд покриває навіть найповільніші відгуки флагманських моделей claude-sonnet-4-5 та gpt-4o (за результатами експерименту з розділу 3 діагностування може тривати до 28 секунд).

Каталоги repair_workflow та etl_scripts проєкту монтуються у контейнери Airflow як волюми за шляхом /opt/airflow/. Такий підхід дає змогу гарячої заміни коду модулів без пересборки Docker-образів і помітно скорочує цикл розробки та налагодження. У продуктивному середовищі ці волюми доцільно замінити на зібрані Docker-образи з версійованими тегами для відтворюваності розгортання.

Сервіси стека та їх призначення подано у таблиці 4.1.

*Таблиця 4.1 - Сервіси Docker Compose стека та їх призначення*

| Сервіс | Образ / версія | Порт | Призначення |
|---|---|---|---|
| airflow-apiserver | apache/airflow:3.0 | 8080 | Веб-інтерфейс та REST API Airflow |
| airflow-scheduler | apache/airflow:3.0 | - | Планування запусків DAG |
| airflow-dag-processor | apache/airflow:3.0 | - | Парсинг Python-файлів DAG |
| airflow-worker | apache/airflow:3.0 | - | Виконання задач через Celery, запуск on_failure_callback |
| airflow-triggerer | apache/airflow:3.0 | - | Обробка deferred-операторів |
| postgres | postgres:16 | 5432 | База метаданих Airflow |
| redis | redis:7.2 | 6379 | Брокер повідомлень Celery |

### *4.1.2 Інтеграція з хмарною інфраструктурою AWS*

Обчислювальне середовище ETL-пайплайнів та цільове аналітичне сховище винесено у хмару AWS. Система самовідновлення комунікує з п'ятьма сервісами AWS, а саме Amazon S3 (зберігання проміжних даних у форматі Parquet та ETL-скриптів), AWS Glue (виконання Python Shell-задач), Amazon Redshift Serverless (цільове сховище), Amazon CloudWatch Logs (збір логів виконання) та AWS Identity and Access Management, тобто IAM (керування доступом).

Доступ до AWS з боку модуля repair_workflow організовано через бібліотеку boto3. Для централізованого керування клієнтами реалізовано утилітний модуль repair_workflow.sources.aws_clients. Він створює клієнти сервісів із потрібними регіональними параметрами і використовує загальний

ланцюг аутентифікації boto3. Завдяки цьому переключення між різними механізмами автентифікації проходить без правок у коді: локально працює через профіль AWS CLI, у Docker-контейнері через змінні середовища, а у продуктивному розгортанні через IAM-ролі екземпляра EC2 або IRSA у Kubernetes.

Для виконання операцій репарації створюється спеціалізована IAM-роль RepairWorkflowExecutionRole з опорою на принцип найменших привілеїв. Дозволи зведено до того мінімуму, який реально потрібен для збору контексту та застосування виправлень. Йдеться про читання об'єктів з бакета ETL-скриптів та запис до нього, читання логів CloudWatch з лог-груп Glue-джобів, виклик RedshiftDataApiService для виконання DDL та запитів до метаданих, виклик Glue GetJobRun для отримання інформації про запуск. Заборонено: DELETE, DROP, TRUNCATE на рівні Redshift, а також будь-які операції запису до таблиць метаданих. Навіть у разі компрометації облікових даних межа потенційних негативних наслідків лишається обмеженою операціями зі зміни схеми, що у свою чергу проходять перевірку через програмний безпековий шар (розділ 2.4).

Відображення AWS-сервісів на функціональні компоненти системи показує таблиця 4.2.

*Таблиця 4.2 - Використання AWS-сервісів системою самовідновлення*

| **Сервіс AWS** | **Призначення у системі** | **Використовувані дозволи IAM** |
|---|---|---|
| Amazon S3 | Зберігання Parquet-файлів та ETL-скриптів; запис оновлених скриптів | s3:GetObject, s3:PutObject, s3:ListBucket |
| AWS Glue | Виконання Python Shell задач ETL; отримання інформації про запуск | glue:GetJobRun, glue:GetJob |

| | | |
|---|---|---|
| Amazon Redshift Serverless | Цільове сховище даних; виконання ALTER TABLE через Data API | redshift-data:ExecuteStatement, redshift-data:DescribeStatement, redshift-data:GetStatementResult |
| CloudWatch Logs | Читання логів Glue-джобів для побудови профілю даних та traceback | logs:GetLogEvents, logs:DescribeLogStreams |
| AWS IAM | Автентифікація та авторизація сервісних компонентів | sts:AssumeRole (для IRSA/STS) |

### *4.1.3 Розгортання веб-інтерфейсу*

Веб-інтерфейс для роботи оператора (модуль ui/) запускається як дві незалежні служби. Перша, тобто FastAPI-бекенд, працює на порту 8000. Друга, Streamlit-фронтенд, доступна на порту 8501. Обидві служби можна підіймати або як Docker-контейнери, або як systemd-процеси на віртуальному сервері. Бекенд підключає каталог Airflow-логів у режимі read-write: читання потрібне для завантаження артефактів репарації, запис, тобто для збереження файлів статусу _status.json і версіонованих діагнозів _diagnosis_v{N}.json.

У продуктивному розгортанні перед FastAPI та Streamlit рекомендовано встановлювати зворотний проксі nginx. Його роль складається з трьох частин, а саме термінації HTTPS-з'єднання, застосування політики CORS та автентифікації користувачів через корпоративний OIDC-провайдер. Аутентифіковані користувачі дістають доступ до інтерфейсу залежно від ролі: viewer бачить події репарації, approver додатково підтверджує виправлення. Якщо проксі не

налаштовано, система працює у локальному режимі без автентифікації, що прийнятно лише для середовищ розробки та тестування.

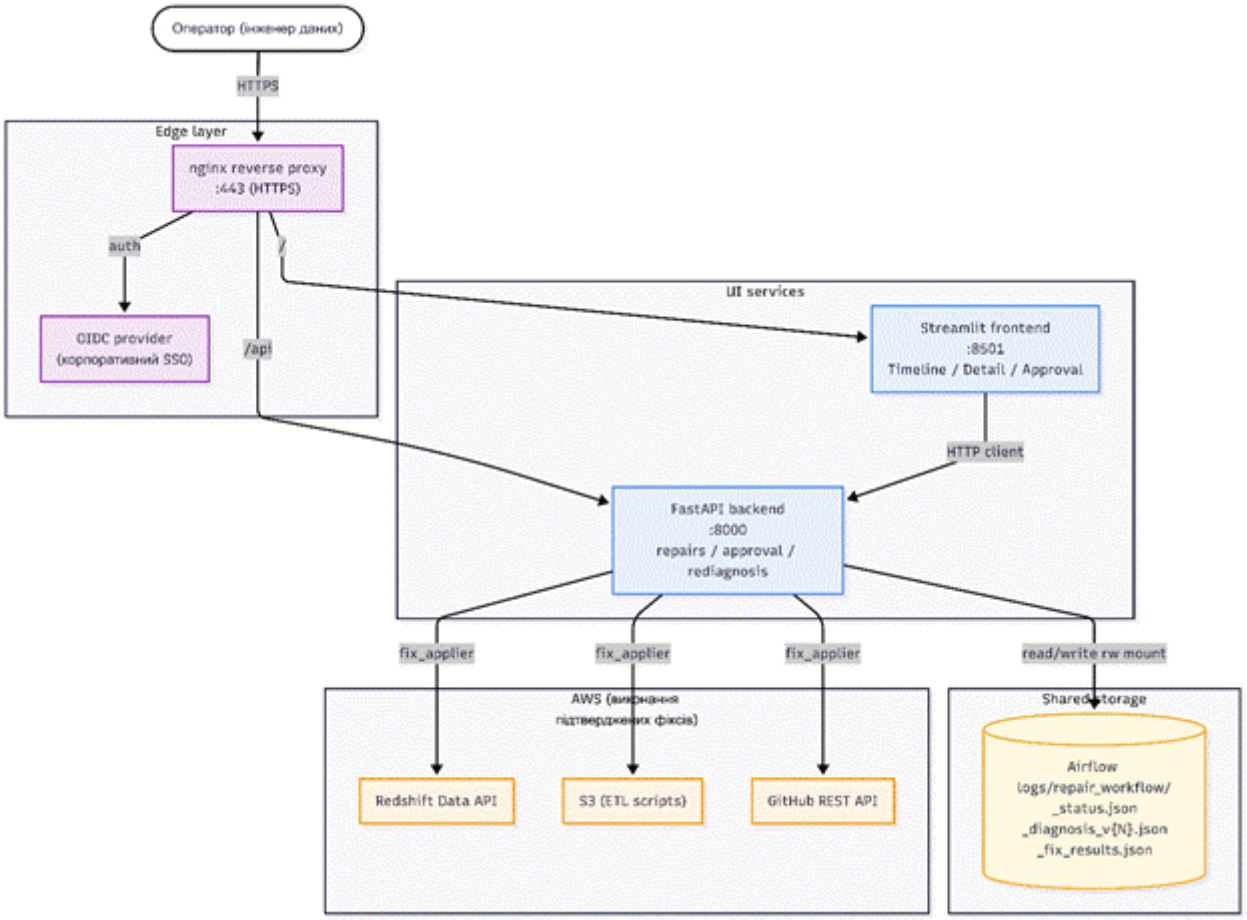


*Рисунок 4.3 - Діаграма розгортання веб-інтерфейсу*

### *4.1.4 Керування секретами та конфігурацією*

Конфігурація системи проходить через змінні середовища, що передаються у контейнери Airflow та веб-інтерфейсу. У середовищі розробки змінні задаються у файлі .env, який підключається до docker-compose.yaml через директиву env_file. У продуктивному середовищі для роботи з секретами краще використовувати AWS Secrets Manager або HashiCorp Vault. Файли .env при цьому варто залишити лише для несекретних параметрів. Повний перелік змінних середовища подано у таблиці 4.3.

*Таблиця 4.3 - Змінні середовища конфігурації системи*

| Змінна | Призначення | Приклад значення |
|---|---|---|
| REPAIR_WORKFLOW_LLM_MODEL | Назва LLM для діагностування | claude-sonnet-4-5 |
| REPAIR_WORKFLOW_LLM_PROVIDER | Провайдер LangChain | anthropic |
| REPAIR_WORKFLOW_APPROVAL_MODE | Режим застосування виправлень | pending_approval |
| REPAIR_WORKFLOW_AUTO_MERGE | Автоматичне злиття GitHub PR | false |
| GITHUB_TOKEN | Personal Access Token для GitHub API | <секрет> |
| GOOGLE_API_KEY | Ключ Google AI Studio для Gemini | <секрет> |
| ANTHROPIC_API_KEY | Ключ Anthropic API для Claude | <секрет> |
| OPENAI_API_KEY | Ключ OpenAI API для GPT | <секрет> |

| | | |
|---|---|---|
| AWS_ACCESS_KEY_ID / AWS_SECRET_ACCESS_KEY | Облікові дані AWS (для локального запуску) | <секрет> |
| AWS_REGION | Регіон AWS для всіх сервісів | us-east-1 |

При налаштуванні секретів неабияку увагу слід приділити політиці ротації ключів. API-ключі провайдерів LLM та GitHub PAT доцільно ротувати щонайменше один раз на квартал. Облікові дані AWS варто застосовувати тільки для локальної розробки, а у продуктивному середовищі переходити на IAM-ролі без статичних ключів. Усі без винятку секрети мають бути додані до .gitignore проєкту, тобто ні за яких умов не потрапляти у репозиторій контролю версій.

### 4.2 Алгоритм інформаційної технології самовідновлення

Розроблена технологія самовідновлення ETL-пайплайнів сформована як композиція чотирьох послідовно виконуваних алгоритмів. До неї входять алгоритм збирання контексту помилки, алгоритм діагностування за допомогою великої мовної моделі, алгоритм опційного підтвердження оператором та алгоритм безпечного виконання виправлень. У цьому підрозділі алгоритми формалізовано у вигляді псевдокоду, описано їх обчислювальну складність, ключові точки відмови та схему взаємодії з зовнішніми сервісами.

#### *4.2.1 Вхідні та вихідні дані інформаційної технології*

На вхід технологія приймає подію збою задачі в Apache Airflow. Цю подію передають у вигляді словника контексту виконання (context dict). Серед його ключових полів варто назвати ідентифікатор задачі (task_id), ідентифікатор DAG (dag_id), виключення (exception), traceback, номер спроби (try_number) та логічну дату запуску (logical_date). Вихідним результатом технології служить набір об'єктів FixResult з описом результату застосування кожного запропонованого

виправлення. Поряд з цим формується повний аудиторський слід процесу репарації, тобто набір JSON-артефактів у файловій системі. Формальну специфікацію входу та виходу подано у таблиці 4.4.

*Таблиця 4.4 - Специфікація входу та виходу інформаційної технології*

| Сутність | Тип | Опис |
|---|---|---|
| Вхід: E | dict | Контекст збою Airflow (task_id, dag_id, exception, try_number, logical_date) |
| Проміжний: F | FailureContext | Агрегований контекст із 6 джерел |
| Проміжний: D | DiagnosisOutput | Структурований діагноз LLM |
| Проміжний: A | dict[str, ApprovalStatus] | Стан підтвердження кожного виправлення |
| Вихід: X | list[FixResult] | Результати застосування виправлень |
| Вихід: артефакти | JSON-файли | context.json, prompt.txt, diagnosis.json, status.json, fix_results.json |

### *4.2.2 Формалізований алгоритм інформаційної технології*

Загальний алгоритм самовідновлення формалізовано як функцію RepairWorkflow(E) → X, де E позначає контекст збою, X становить набір результатів виправлень. Псевдокод подано нижче.

*Алгоритм 4.1 - RepairWorkflow*

```
Вхід:  E - контекст збою Airflow
Вихід: X - набір об'єктів FixResult
1.  F ← BuildFailureContext(E)          // Фаза 1
2.  SaveArtifact(F, context.json)
3.  D ← DiagnoseWithLLM(F)              // Фаза 2
4.  SaveArtifact(D, diagnosis.json)
5.  mode ← getenv(REPAIR_WORKFLOW_APPROVAL_MODE)
6.  if mode = 'pending_approval' then
```

```
7.    A ← { fix.id → pending | fix ∈ D.fixes }
8.    SaveArtifact(A, status.json)
9.    return [] // UI підхопить для підтвердження
10. else if mode = 'log_only' then
11.   for each fix ∈ D.fixes do Validate(fix); Log(fix)
12.   return []
13. else if mode = 'auto' then
14.   X ← [ ]
15.   for each fix ∈ D.fixes do
16.         if Validate(fix) then X.append(ExecuteFix(fix))
17.         else X.append(FixResult(success=False))
18.   SaveArtifact(X, fix_results.json)
19.   return X
```

Алгоритм BuildFailureContext, тобто фаза 1, агрегує сигнали з шести джерел в один об'єкт FailureContext. Він виконується у межах одного процесу Airflow Worker і послідовно звертається до локальної файлової системи, AWS-сервісів, а також виконує алгоритмічне порівняння схеми. Псевдокод алгоритму подано нижче.

*Алгоритм 4.2 - BuildFailureContext*

```
Вхід:  E - контекст збою Airflow
Вихід: F - об'єкт FailureContext
1.  F.error   ← ExtractErrorInfo(E)
2.  F.dag     ← ParseDagMetadata(E.dag_id)
3.  F.ddl     ← ParseDDL(F.dag.create_sql)
4.  F.script  ← ReadEtlScript(F.dag.script_path)
5.  F.logs    ← FetchCloudWatchLogs(F.dag.glue_job_run_id)
6.  F.profile ← ExtractDataProfile(F.logs)
7.  F.diff    ← ComputeSchemaDiff(F.ddl, F.profile)
8.  return F
```

Кожен крок виконується у власному блоці try/except. Завдяки цьому контекст вдається побудувати хоча б частково навіть за умови недоступності одного з джерел. Скажімо, якщо CloudWatch Logs повертає помилку IAM, F.logs

лишається порожнім рядком, F.profile становить порожній список, а F.diff обчислюється виключно на підставі DDL. Діагностування у такому режимі теж залишається можливим, хоча і з певним зниженням точності.

Алгоритм DiagnoseWithLLM, тобто фаза 2, формує промпт із контексту помилки і отримує від LLM структурований діагноз. Передбачено два режими отримання відповіді: основний через механізм structured output фреймворку LangChain (with_structured_output) та альтернативний через ручний парсинг JSON з текстової відповіді. Псевдокод подано нижче.

*Алгоритм 4.3 - DiagnoseWithLLM*

```
Вхід:  F - FailureContext
Вихід: D - DiagnosisOutput
1.  prompt ← BuildPrompt(F)
2.  SaveArtifact(prompt, prompt.txt)
3.  model ← ResolveModel()   // env / config / default
4.  llm   ← init_chat_model(model, temperature=0.0)
5.  chain ← llm.with_structured_output(DiagnosisOutput)
6.  try
7.   D ← chain.invoke(prompt)
8.  except StructuredOutputError
9.   raw ← llm.invoke(prompt)
10.  D   ← ParseJsonFallback(raw)
11. return D
```

Алгоритм ExecuteFix, тобто фаза 4, виконує диспетчеризацію виправлень за їх типом. Для кожного типу передбачено окремий обробник, що відповідає за валідацію та застосування змін. Загальний диспетчер подано нижче.

*Алгоритм 4.4 - ExecuteFix*

```
Вхід:  fix - ProposedFixOutput
Вихід: result - FixResult
1.  switch fix.fix_type
2.   case 'sql_ddl':
3.        if not SafeSql(fix.sql): return FAIL
4.        result ← RedshiftDataApi.execute(fix.sql)
```

```
5.   case 'etl_script_change':
6.        if not SafePython(fix.code_change): return FAIL
7.        BackupFile(fix.target)
8.        ApplyLocalChange(fix)
9.        UploadToS3(fix.target)
10.       QueueGitHubPR(fix)
11.  case 'dag_code_change':
12.       if not SafePython(fix.code_change): return FAIL
13.       QueueGitHubPR(fix)
14. return result
```

### *4.2.3 Оцінка обчислювальної складності*

Обчислювальна складність алгоритму RepairWorkflow складається з трьох незалежних доданків, а саме часу побудови контексту T_ctx, часу виклику LLM T_llm та часу виконання виправлень T_exec. Загальний час виконання записується у формі:

$$T_{total} = T_{ctx} + T_{llm} + T_{exec}. \quad (4.1)$$

Побудова контексту складається з послідовного звернення до різних джерел. Сюди входить читання локальних файлів DAG та ETL-скрипта зі складністю O(s), де s позначає розмір файлу скрипта. Парсинг DDL регулярними виразами становить O(d), де d це кількість колонок. Читання логів CloudWatch упирається в I/O, тобто обмежене пропускною здатністю мережі до AWS. Обчислення schema diff виходить на O(d + p), де p позначає кількість профілів колонок. За результатами експерименту з розділу 3 фаза побудови контексту триває 5,8-7,1 секунди залежно від обсягу логів.

Час виклику LLM T_llm пов'язаний з обсягом вхідного промпту та налаштуваннями моделі. За даними експерименту 2 з розділу 3, T_llm коливається у проміжку від 5,3 секунди (gemini-2.5-flash-lite) до 28,4 секунди (gemini-2.5-flash). На клієнтській стороні складність становить O(1), оскільки виконується один API-запит. Водночас саме фаза LLM формує домінуючу частку

часу репарації, тобто є першочерговим кандидатом на оптимізацію (вибір швидшої моделі або кешування однотипних запитів).

Час виконання виправлень T_exec пропорційний кількості виправлень n та типу кожного з них, тобто має складність O(n) з коефіцієнтом, що залежить від типу. Для sql_ddl-виправлень середній час ALTER TABLE у Amazon Redshift Serverless становить 1,5-3 секунди. Для etl_script_change показник лежить у проміжку 2-4 секунди (включно з upload до S3). Для dag_code_change значення 3-5 секунд (створення GitHub PR через REST API). Загалом фаза виконання займає 5,8-7,1 секунди для типового сценарію з 1-2 виправленнями.

У підсумку повний цикл RepairWorkflow триває від 11 до 40 секунд залежно від обраної моделі. Це на порядки менше, ніж ручне відновлення, яке за оцінками інженерів бази практики займає від 30 до 90 хвилин на один інцидент.

## 4.3 Тестування системи

Надійність системи самовідновлення підтримується завдяки комплексному тестуванню на трьох рівнях. Перший рівень становить модульне тестування окремих компонентів. Другий охоплює інтеграційне тестування взаємодії з зовнішніми сервісами. Третій становить функціональне (end-to-end) тестування повного циклу репарації на реалістичних сценаріях schema drift. Усі тести побудовано на фреймворку pytest. Запуск автоматизовано: тести виконуються під час розробки, тобто перед кожним розгортанням.

### *4.3.1 Модульне тестування компонентів*

Модульні тести написано для кожного функціонального модуля пакета repair_workflow. Розміщено їх у каталозі repair_workflow/tests/ та згруповано за компонентами, що проходять перевірку. У роботі застосовано принцип ізоляції: усі зовнішні залежності, як-от AWS-сервіси, виклики LLM, операції з файловою системою, замоковано засобами бібліотеки unittest.mock та спеціалізованої фікстури moto, що симулює AWS. Повний перелік модульних тестів подає таблиця 4.5.

*Таблиця 4.5 - Модульні тести пакета repair_workflow*

| **Файл тесту** | **Тестований компонент** | **Кіл-ть тестів** |
|---|---|---|
| test_context_builder.py | Побудова FailureContext, агрегація джерел, обробка часткової відсутності даних | 8 |
| test_cloudwatch_logs.py | Парсинг логів CloudWatch, вилучення traceback та профілю даних | 6 |
| test_ddl_parser.py | Парсинг CREATE TABLE SQL, вилучення ColumnDDL, обробка VARCHAR(n) | 7 |
| test_error_info.py | Вилучення ErrorInfo з контексту Airflow | 4 |
| test_schema_diff.py | Обчислення schema_diff, типи varchar_overflow, type_mismatch, new_column, ігнорування case | 9 |
| test_diagnosis_engine.py | Виклик LLM, structured output, fallback-парсер JSON | 7 |
| test_prompt.py | Побудова промпту, селективне включення даних, обмеження обсягу | 5 |
| test_executor.py | Диспетчеризація FixExecutor за типами виправлень | 6 |
| test_sql_ddl_handler.py | SqlDdlHandler: виклик Redshift Data API, обробка помилок | 4 |

| | | |
|---|---|---|
| test_etl_script_handler.py | Застосування змін коду ETL-скрипта, локальний запис, S3 upload | 5 |
| test_github_pr.py | GitHubPRCreator: створення гілки, комітів, злиття SHA | 6 |
| test_safety.py | Безпекова валідація: SQL whitelist/blacklist, compile() Python | 8 |
| test_dag_metadata.py | Парсинг Python-файлу DAG, вилучення CREATE SQL та S3-шляху | 5 |
| Разом | | 80 |

За вимірюванням coverage.py тестове покриття основних модулів пакета склало 87%. Найвищі показники припали на модулі schema_diff.py (96%), ddl_parser.py (94%) та safety.py (93%), тобто компоненти з детермінованою логікою. Дещо нижче покриття у проміжку 78-82% характерне для модулів взаємодії з LLM та AWS-сервісами через складність мокування деяких крайових випадків. Покриття за модулями деталізує таблиця 4.6.

*Таблиця 4.6 - Покриття модулів тестами*

| **Модуль** | **Рядків коду** | **Покрито, %** |
|---|---|---|
| schema_diff.py | 142 | 96 |
| sources/ddl_parser.py | 98 | 94 |
| executor/safety.py | 116 | 93 |
| models.py | 184 | 92 |
| context_builder.py | 163 | 89 |
| sources/error_info.py | 72 | 88 |

| | | |
|---|---|---|
| sources/dag_metadata.py | 145 | 86 |
| diagnosis/prompt.py | 134 | 85 |
| executor/handlers/sql_ddl.py | 87 | 83 |
| executor/handlers/etl_script.py | 125 | 82 |
| diagnosis/engine.py | 158 | 81 |
| executor/github_pr.py | 203 | 78 |
| callback.py | 94 | 78 |
| Загалом | 1 721 | 87 |

### *4.3.2 Інтеграційне тестування*

Інтеграційні тести пишуть взаємодію компонентів системи з реальними зовнішніми сервісами у контрольованому середовищі. Для тестування AWS-інтеграцій використано бібліотеку moto. Вона емулює API-поведінку Amazon S3, Glue, Redshift Data API та CloudWatch Logs у межах одного Python-процесу, тобто без потреби звертатися до реальної хмарної інфраструктури. Тести, що передбачають реальний виклик LLM, запускаються лише за наявності відповідних API-ключів у середовищі (змінна PYTEST_RUN_LLM_TESTS=1) і не входять до щоденного циклу CI.

Розроблено вісім сценаріїв інтеграційних тестів. Серед них повний цикл RepairWorkflow зі замоканим AWS та LLM, перевірка читання метаданих DAG з реального Python-файлу, перевірка взаємодії з Redshift Data API через локальний мок, симуляція створення GitHub PR через bridge-сервер, обробка відсутніх прав IAM, обробка тайм-ауту LLM, обробка повторного виклику з тим самим завданням, а також обробка одночасного виклику для декількох задач одного DAG.

### *4.3.3 Функціональне (end-to-end) тестування*

Функціональне тестування виконується на повнофункціональному середовищі, тобто з реальним Docker Compose-стеком Airflow, хмарним середовищем AWS та реальними викликами LLM. Автоматизація функціонального тестування реалізована у вигляді модуля experiment_runner (розділ 3.5), що поєднує функції експериментального дослідження та функціонального тестування.

Кожен end-to-end тест проходить повну послідовність кроків: (1) скидання схеми Redshift до базового стану; (2) генерація Parquet-файлу з навмисно внесеною помилкою; (3) запуск DAG; (4) очікування збою задачі; (5) запуск циклу репарації; (6) перевірка наявності артефактів у файловій системі; (7) перевірка валідності структурованого діагнозу; (8) перевірка коректного застосування виправлення (ALTER TABLE виконано у Redshift, ETL-скрипт оновлено у S3, PR створено у GitHub); (9) повторний запуск DAG для верифікації виправлення; (10) збір метрик часу, токенів та вартості. Перелік функціональних сценаріїв подає таблиця 4.7.

*Таблиця 4.7 - Функціональні тестові сценарії*

| **Сценарій** | **Тип drift** | **Очікуваний фікс** | **Верифікація** |
|---|---|---|---|
| varchar_overflow | varchar_overflow | ALTER TABLE ALTER COLUMN TYPE VARCHAR(N) | COPY успішна |
| type_change | type_mismatch | ALTER TABLE або зміна dtype у ETL | COPY успішна |
| new_column | new_column | ALTER TABLE ADD COLUMN | COPY успішна |
| new_collection_column | new_column (list) | ALTER TABLE ADD COLUMN + зміна | COPY успішна |

| | | | |
|---|---|---|---|
| | | COLLECTION_COLUMNS | |
| no_source_data | - | Не є schema drift, діагностується як configuration | false_positive_flags |
| base | - | Нічого (clean baseline) | Не спрацьовує |

Важливою рисою функціонального тестування є його регресійна природа. Після кожної зміни промпту, схеми DiagnosisOutput чи логіки diff-алгоритму запускається повний набір сценаріїв. Це гарантує збереження точності та виключає регресії. За результатами регулярного запуску накопичено історичний датасет у файлі all_results.csv, що нараховує понад 500 прогонів і застосовується для моніторингу стабільності точності діагностування.

### *4.3.4 Тестування веб-інтерфейсу*

Тестування веб-інтерфейсу проходить на двох рівнях. На рівні бекенду (FastAPI) застосовано клас TestClient з бібліотеки fastapi.testclient, що дає змогу перевірити коректність API-ендпойнтів, тобто отримання списку подій, детальної інформації, підтвердження або відхилення виправлень, повторного діагностування. Усі тести ізольовано від файлової системи через фікстуру tmp_path pytest. На рівні фронтенду (Streamlit) виконується ручне тестування ключових сценаріїв використання, як-от перегляд списку подій у Repair Timeline, аналіз окремої події у Repair Detail, підтвердження групи виправлень у Approval Queue. Автоматизоване тестування Streamlit-компонентів вважається нераціональним через особливості реактивної моделі фреймворку.

## 4.4 Техніко-економічне обґрунтування впровадження

Техніко-економічне обґрунтування побудовано методом порівняння поточних витрат на ручне відновлення ETL-пайплайнів з прогнозованими витратами на впровадження та подальшу експлуатацію автоматизованої системи.

Розрахунок виконано стосовно бази практики з урахуванням її специфіки, тобто інтеграції рекламних даних із зовнішніх API до аналітичного сховища Amazon Redshift Serverless.

### *4.4.1 Вихідні дані для розрахунку*

Вихідні дані сформовано на підставі аналізу історичних інцидентів у пайплайнах бази практики та консультацій з інженерами даних. Кількісні характеристики процесу ручного відновлення подано у таблиці 4.8.

*Таблиця 4.8 - Вихідні дані для техніко-економічного обґрунтування*

| **Параметр** | **Значення** | **Джерело** |
|---|---|---|
| Середня кількість інцидентів schema drift на тиждень | 10-20 | Статистика бази практики |
| Середня кількість інцидентів на місяць | 40-100 (сер. 70) | Статистика бази практики |
| Середня кількість інцидентів на рік | 840 | 40-100 × 12, середнє |
| Середній час ручного відновлення одного інциденту | 30-90 хв (сер. 60) | Експертна оцінка інженерів |
| Середня годинна ставка Data Engineer (Middle/Senior) | 30 USD/год | Ринкова ставка |
| Коефіцієнт соціальних нарахувань та накладних витрат | 22% | Податкове законодавство + накладні |
| Курс USD/UAH (станом на 04.2026) | 41,5 | Середній міжбанк |

| | | |
|---|---|---|
| Горизонт оцінки | 3 роки | Прийнято для розрахунку ROI |

Базовим режимом роботи системи на впровадженні обрано pending_approval, тобто режим з обов'язковим підтвердженням виправлень оператором через веб-інтерфейс. Підстава для такого вибору проста: цей режим гарантує максимальний рівень безпеки під час застосування змін до продуктивного середовища Redshift та GitHub-репозиторіїв бази практики, що становить критичну вимогу на початковому етапі впровадження.

За результатами експериментального дослідження (розділ 3.7), для режиму pending_approval рекомендованою моделлю стає claude-sonnet-4-5. Серед досліджених моделей вона показала найвищу точність діагностування (95%) і найнижчий рівень хибних спрацювань. Вартість одного виклику цієї моделі дорівнює 0,0792 USD.

### *4.4.2 Розрахунок витрат на ручне відновлення*

Річні витрати на ручне відновлення ETL-пайплайнів обчислюються як добуток кількості інцидентів, середнього часу відновлення та ставки інженера з урахуванням соціальних нарахувань. Формула розрахунку має вигляд:

$$C\_manual = N \times t \times R \times (1 + k), \quad (4.2)$$

де N означає річну кількість інцидентів, t позначає середній час відновлення у годинах, R становить годинну ставку інженера у USD, k відповідає коефіцієнту соціальних нарахувань та накладних витрат.

Підставляючи значення з таблиці 4.8, одержуємо:

$$C\_manual = 840 \times 1{,}0 \times 30 \times 1{,}22 = 30\,744 \text{ USD/рік} \approx 1\,275\,876 \text{ грн/рік}. \quad (4.3)$$

Додатковою складовою витрат залишаються втрати від несвоєчасного оновлення аналітичних даних. У контексті пайплайнів інтеграції рекламних даних затримка веде до того, що маркетингові рішення приймаються на основі застарілої інформації, а це знижує ефективність рекламних кампаній. На базі

практики кількісна оцінка цього ефекту не проводилася, тобто у розрахунку ТЕО непрямі втрати залишено поза увагою. Як наслідок, оцінка економічного ефекту набуває консервативного характеру.

### *4.4.3 Розрахунок витрат на впровадження системи*

Витрати на впровадження системи поділяються на дві категорії, а саме одноразові витрати на адаптацію, розгортання та навчання персоналу і щорічні витрати на експлуатацію. Структуру одноразових витрат подає таблиця 4.9.

*Таблиця 4.9 - Одноразові витрати на впровадження системи*

| **Стаття витрат** | **Трудомісткість, год** | **Ставка, USD/год** | **Сума, USD** |
|---|---|---|---|
| Адаптація коду repair_workflow під специфіку бази практики | 80 | 35 | 2 800 |
| Налаштування AWS IAM-ролей та інтеграції з GitHub | 40 | 35 | 1 400 |
| Розгортання веб-інтерфейсу, nginx, OIDC-автентифікації | 40 | 35 | 1 400 |
| Розробка функціональних тестів для специфічних пайплайнів | 60 | 30 | 1 800 |
| Навчання інженерів даних (2 особи × 20 год) | 40 | 30 | 1 200 |

| | | | |
|---|---|---|---|
| Пілотна експлуатація та доопрацювання | 80 | 35 | 2 800 |
| Разом одноразових витрат | 340 | - | 11 400 |

Щорічні витрати на експлуатацію складаються з трьох частин, тобто вартості викликів LLM, вартості інфраструктури розгортання та витрат часу оператора на перегляд і підтвердження виправлень у режимі pending_approval. Структуру щорічних витрат розкриває таблиця 4.10.

*Таблиця 4.10 - Щорічні витрати на експлуатацію системи*

| **Стаття витрат** | **Розрахунок** | **Сума, USD/рік** | **Сума, грн/рік** |
|---|---|---|---|
| Виклики LLM (claude-sonnet-4-5) | 840 викл. × 0,0792 USD | 66,53 | 2 761 |
| Повторні діагностування (rediagnosis) | 20% × 840 × 0,0792 | 13,31 | 552 |
| Сервер для UI (t3.small, AWS EC2) | 18 USD/міс × 12 | 216,00 | 8 964 |
| Зберігання артефактів (S3 + CloudWatch) | 10 USD/міс × 12 | 120,00 | 4 980 |
| Секретне сховище (AWS Secrets Manager) | 5 USD/міс × 12 | 60,00 | 2 490 |
| Час оператора на підтвердження (5 хв × 840 × 30 × 1,22) | 70 год × 30 × 1,22 | 2 562,00 | 106 323 |
| Підтримка та оновлення коду | 40 год × 35 × 1,22 | 1 708,00 | 70 882 |

| | | | |
|---|---|---|---|
| Разом щорічних витрат | | 4 745,84 | 196 952 |

На статтю «час оператора на підтвердження» варто звернути окрему увагу. У режимі pending_approval оператор для кожного спрацювання системи повинен переглянути запропоновані виправлення у веб-інтерфейсі, тобто прийняти рішення про їх застосування. За експертною оцінкою на одну подію припадає у середньому 5 хвилин перегляду. Це виходить на 70 годин на рік. Значення помітно менше за 840 годин, потрібних для ручного відновлення, проте все одно становить ненульову складову витрат, тобто його слід враховувати при плануванні впровадження.

### *4.4.4 Розрахунок економічного ефекту та окупності*

Розрахунок прогнозованого річного економічного ефекту виконано за класичною схемою порівняння двох альтернативних сценаріїв функціонування ETL-інфраструктури бази практики.

Перший сценарій (базовий) відповідає чинній практиці ручного відновлення пайплайнів після збоїв schema drift. Другий сценарій (проєктний) передбачає функціонування пайплайнів з підключеною системою самовідновлення у режимі pending_approval. Економічний ефект формалізується як різниця сукупних витрат за цими двома сценаріями.

Методика розрахунку включає п'ять послідовних кроків:

Крок 1. Визначення річної кількості інцидентів N. За статистикою бази практики середня частота інцидентів schema drift становить 10-20 на тиждень. Беручи середнє арифметичне (15 інц./тижд.) та припускаючи 52 робочі тижні на рік, отримуємо орієнтовну оцінку $N \approx 780$ інцидентів на рік. Для розрахунку прийнято консервативну оцінку $N = 840$, що відповідає максимальній межі статистики (70 інц./міс. × 12 міс.). Такий вибір зміщує підсумкову оцінку у бік заниження економічного ефекту.

Крок 2. Розрахунок поточних витрат на ручне відновлення C_manual. Він здійснюється за формулою: $C_{manual} = N \times t \times R \times (1 + k)$, (4.2) де t позначає середній час повного циклу ручного відновлення одного інциденту у годинах

(визначений експертною оцінкою інженерів бази практики у діапазоні 30-90 хвилин, прийнято t = 1,0 год), R становить годинну ставку Data Engineer рівня Middle/Senior у USD (визначена за ринковими даними станом на II квартал 2026 року на рівні R = 30 USD/год), k відображає коефіцієнт соціальних нарахувань та накладних витрат роботодавця (єдиний соціальний внесок 22% згідно з чинним податковим законодавством України, k = 0,22). Підстановка значень дає: C_manual = 840 × 1,0 × 30 × 1,22 = 30 744 USD/рік ≈ 1 275 876 грн/рік. (4.3)

Крок 3. Розрахунок щорічних витрат на експлуатацію системи C_operational. Витрати декомпозовано на сім статей (таблиця 4.10), які охоплюють виклики LLM, повторні діагностування, хостинг веб-інтерфейсу, зберігання артефактів, управління секретами, час оператора на підтвердження виправлень у режимі pending_approval та підтримку коду. Сума зазначених статей становить: C_operational = 66,53 + 13,31 + 216,00 + 120,00 + 60,00 + 2 562,00 + + 1 708,00 = 4 745,84 USD/рік ≈ 196 952 грн/рік. (4.4)

Крок 4. Обчислення річного економічного ефекту ΔC як різниці витрат за двома сценаріями: ΔC = C_manual − C_operational = 30 744 − 4 746 = 25 998 USD/рік. (4.5) У гривневому еквіваленті за курсом 41,5 грн/USD річна економія становить 1 078 917 грн/рік. Це відповідає скороченню витрат на обслуговування ETL-пайплайнів у 6,5 рази порівняно з ручним підходом. Питома економія на одному інциденті дорівнює ΔC / N ≈ 30,95 USD/інц., або близько 1 285 грн/інц.

Крок 5. Визначення показників інвестиційної привабливості. Термін окупності одноразових витрат на впровадження (payback period) визначається відношенням сукупних інвестиційних витрат C_investment (таблиця 4.9) до річного економічного ефекту: T_payback = C_investment / ΔC = 11 400 / 25 998 ≈ 0,44 року ≈ 5,3 місяця. (4.6) Показник повернення інвестицій (Return on Investment) за трирічним горизонтом експлуатації обчислюється за формулою: ROI_3 = (3 × ΔC − C_investment) / C_investment × 100%. (4.7) Підстановка значень дає: ROI_3 = (77 994 − 11 400) / 11 400 × 100% ≈ 584%. (4.8)

Динаміку накопиченого економічного ефекту протягом трирічного горизонту прогнозування подано у таблиці 4.11.

*Таблиця 4.11 - Прогноз економічного ефекту за трирічним горизонтом*

| Рік | Витрати, USD | Економія, USD | Накопичений ефект, USD | Накопичений ефект, грн |
|---|---|---|---|---|
| 0 (впровадження) | 11 400 | - | −11 400 | −473 100 |
| 1 | 4 746 | 25 998 | 9 852 | 408 858 |
| 2 | 4 746 | 25 998 | 35 850 | 1 487 775 |
| 3 | 4 746 | 25 998 | 61 848 | 2 566 692 |
| Разом за 3 роки | 25 638 | 77 994 | 61 848 (чистий ефект) | 2 566 692 |

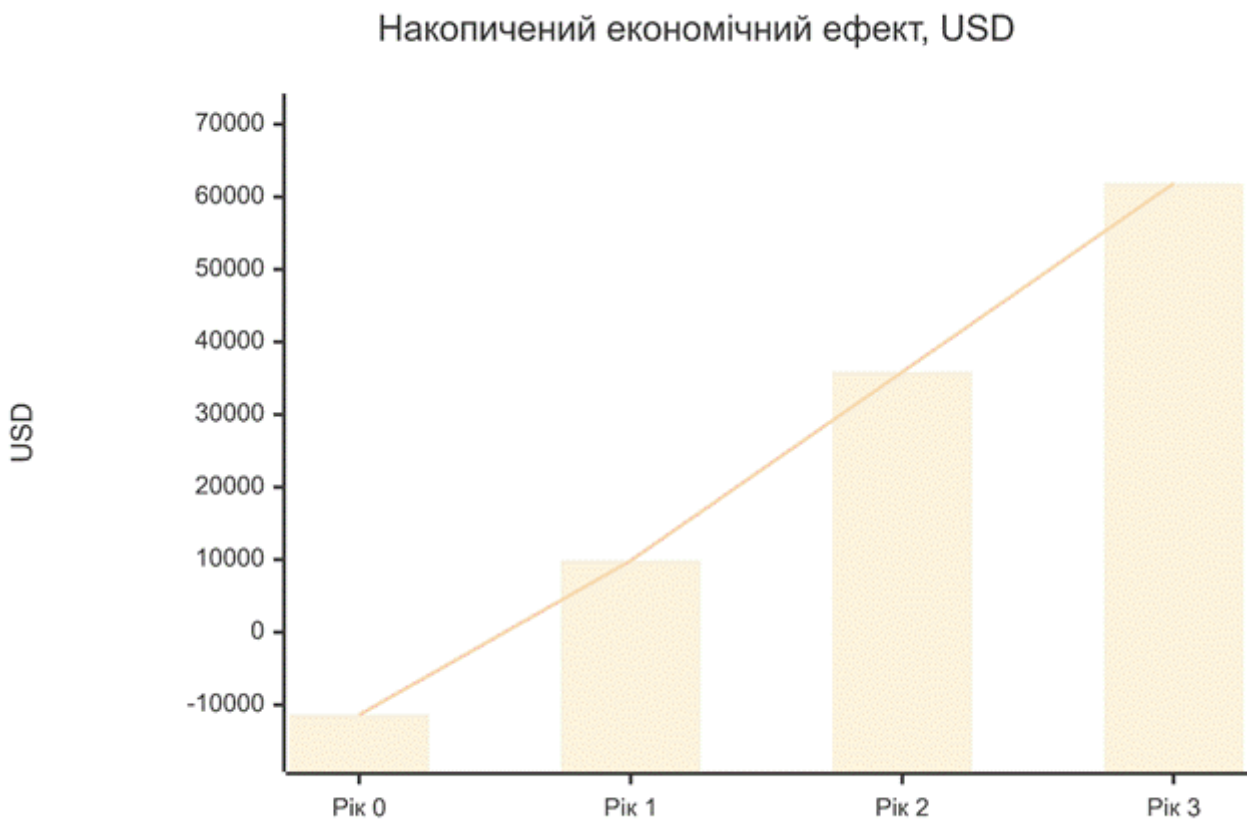


*Рисунок 4.4 - Динаміка накопиченого економічного ефекту за трирічним горизонтом*

### *4.4.5 Нематеріальні вигоди впровадження*

Поза кількісно оціненими економічними результатами впровадження системи самовідновлення приносить низку нематеріальних переваг. Прямому фінансовому вимірюванню вони піддаються складно, проте суттєво позначаються на ефективності роботи команди даних та якості аналітичної інфраструктури. До таких переваг належать: помітне скорочення часу реакції на інциденти (з десятків хвилин або годин до секунд або хвилин у режимі pending_approval), зниження психологічного навантаження на інженерів через автоматизацію рутинних задач, прозорість рішень завдяки веб-інтерфейсу та версіонуванню діагнозів, підвищення довіри стейкхолдерів до аналітичних даних на тлі скорочення часу їх несвоєчасності, а також можливість акумулювання корпоративної бази знань у вигляді історичних діагнозів для подальшого аналізу типових причин інцидентів.

Слід згадати і про організаційний ефект. За опитуванням Fivetran [8], на обслуговування існуючих пайплайнів інженери даних витрачають до 40% свого робочого часу. Автоматизація реактивного обслуговування через самовідновлення вивільняє цей ресурс, тобто команда дістає змогу перерозподілити час на проактивні задачі, як-от розробку нових пайплайнів, оптимізацію наявних, покращення моніторингу. Виникає додатковий економічний ефект, що не увійшов до прямого розрахунку, проте у середньостроковій перспективі може виявитися вагомим.

### 4.5 Рекомендації щодо впровадження

На підставі проведеного аналізу та результатів експериментального дослідження сформульовано рекомендації щодо впровадження розробленої системи самовідновлення. Вони стосуються як бази практики, так і інших підприємств з аналогічною ETL-інфраструктурою. Рекомендації охоплюють чотири блоки питань, а саме етапність впровадження, керування ризиками, моніторинг системи та подальший розвиток продукту.

#### *4.5.1 Етапність впровадження*

Рекомендованим є трифазний підхід до впровадження. Він забезпечує поступове нарощування довіри до автоматизованого процесу, а також знижує ризики передчасного застосування неперевірених виправлень до продуктивного середовища.

Фаза 1, тобто спостережний режим (2-4 тижні). Систему підіймають у режимі log_only, у якому виконуються побудова контексту, діагностування та валідація виправлень, проте жодних змін до продуктивного середовища не вноситься. Мета фази проста: накопичити статистику точності діагностування на реальних інцидентах бази практики, тобто виявити можливі розбіжності між експериментальними показниками (розділ 3) та специфікою продуктивних пайплайнів.

Фаза 2, тобто режим підтвердження (2-3 місяці). Система перемикається у режим pending_approval. Усі спрацювання формують події репарації, які підтверджуються через веб-інтерфейс інженером даних. Завдання фази складається з двох частин, а саме налаштування операційних процедур команди та накопичення аудиторського сліду підтверджених рішень. Паралельно відпрацьовуються регламенти реагування на події репарації.

Фаза 3, тобто автоматичний режим для вибіркових сценаріїв (після 3-го місяця). Для сценаріїв з найвищою точністю діагностування, як-от varchar_overflow та new_column (до 100% у експериментах), доцільно увімкнути автоматичний режим через конфігурацію REPAIR_WORKFLOW_APPROVAL_MODE=auto з додатковим фільтром за порогом confidence ≥ 0,95. Для сценаріїв з нижчою точністю (type_change тримається на рівні 71,4%) режим pending_approval лишається без змін.

### *4.5.2 Управління ризиками*

Перелік ключових ризиків впровадження та заходів їх пом'якшення подає таблиця 4.12.

*Таблиця 4.12 - Ризики впровадження та заходи пом'якшення*

| Ризик | Ймовірність | Вплив | Заходи пом'якшення |
|---|---|---|---|
| Некоректне виправлення, що пошкоджує схему Redshift | Низька | Високий | Безпековий шар (whitelist ALTER TABLE), режим pending_approval, бекапи схеми |
| Галюцинації LLM у діагнозі | Середня | Середній | Механізм false_positive_flags, версіонування діагнозів, re-diagnosis з контекстом |
| Компрометація API-ключів провайдерів LLM | Низька | Високий | AWS Secrets Manager, ротація ключів щокварталу |
| Збільшення витрат на виклики LLM | Середня | Низький | Моніторинг використання токенів, поріг confidence, fallback на дешевші моделі |
| Відмова API провайдера LLM | Низька | Середній | Fallback-стратегія між провайдерами через LangChain |
| Помилка у правах IAM, що блокує роботу системи | Середня | Середній | Тестування ролей у pre-production, моніторинг помилок CloudTrail |
| Помилки у тестах, що не виявляють регресій | Середня | Середній | Регулярний запуск experiment_runner, моніторинг all_results.csv |

### *4.5.3 Моніторинг та спостережуваність*

Для надійної експлуатації системи необхідно налагодити моніторинг ключових показників її роботи. Метрики поділяються на три категорії, а саме операційні (доступність сервісів, затримки виклику LLM, частота помилок), якісні (точність діагностування, частка спрацювань safety layer, рівень впевненості моделі) та економічні (накопичена вартість викликів LLM, кількість спрацювань на день). Усі метрики агрегуються засобами AWS CloudWatch і доступні через стандартні дашборди CloudWatch або, за потреби, через Grafana.

До переліку критичних алертів, тобто тригерів сповіщень для чергового інженера, належать: сплеск помилок виклику LLM (понад 5% за годину), тривалий час побудови контексту (понад 30 секунд), зниження середнього confidence діагнозів (нижче 0,85 за тиждень), аномальне зростання кількості спрацювань safety layer (понад 3 блокування за день). Нормальні робочі значення цих метрик встановлюються на базі даних перших трьох тижнів експлуатації у спостережному режимі.

### *4.5.4 Напрями подальшого розвитку*

Після успішного впровадження базової версії системи відкриваються наступні напрями подальшого розвитку: (1) розширення переліку підтримуваних типів помилок поза межі schema drift, як-от помилки з'єднання, проблеми продуктивності, помилки прав доступу; (2) підтримка інших аналітичних сховищ (Snowflake, BigQuery, Databricks) через абстрактний інтерфейс StorageDriver; (3) адаптація до альтернативних оркестраторів (Dagster, Prefect, Mage); (4) впровадження активного навчання (active learning) на основі історичних рішень оператора, тобто використання підтверджених та відхилених діагнозів як few-shot прикладів у промпті; (5) інтеграція з системами управління інцидентами (PagerDuty, Opsgenie) для автоматичного створення тікетів та ескалації; (6) реалізація предиктивного режиму, де потенційний schema drift виявляється через аналіз API-контрактів джерел даних до настання збою.

**Висновки до розділу 4**

У четвертому розділі описано технологію впровадження розробленої системи самовідновлення ETL-пайплайнів. Розглянуто інфраструктуру розгортання, що складається з on-premises стека Apache Airflow у Docker Compose, хмарних сервісів AWS (Glue, Redshift Serverless, S3, CloudWatch, IAM), а також зовнішніх сервісів GitHub та провайдерів LLM. Окрім цього, описано принципи керування секретами, використання IAM-ролей з найменшими привілеями і розгортання веб-інтерфейсу за зворотним проксі nginx з OIDC-автентифікацією.

Алгоритм інформаційної технології формалізовано у вигляді чотирьох взаємопов'язаних псевдокодів, тобто RepairWorkflow, BuildFailureContext, DiagnoseWithLLM та ExecuteFix. Разом вони охоплюють повний цикл самовідновлення від виникнення події збою до застосування виправлень. Оцінено обчислювальну складність алгоритму: фази побудови контексту та виконання виправлень мають складність O(n + d + p), а фаза LLM формує домінанту часу репарації з обчислювальною складністю O(1) на клієнтській стороні та затримкою у 5-28 секунд залежно від обраної моделі. Загальний час повного циклу репарації становить 11-40 секунд, тобто на порядки менше за ручне відновлення (30-90 хвилин).

Описано методику трирівневого тестування системи. Перший рівень становить модульне тестування (80 тестів у 13 файлах, покриття коду 87%). Другий рівень охоплює інтеграційне тестування (8 сценаріїв зі замоканими AWS-сервісами). Третій становить функціональне (6 end-to-end сценаріїв на реальному середовищі). Регулярне виконання функціональних тестів через модуль experiment_runner забезпечує регресійний контроль точності діагностування у часі.

Виконано техніко-економічне обґрунтування впровадження з урахуванням специфіки бази практики, тобто 840 інцидентів на рік, середній час ручного відновлення 60 хвилин, ставка інженера 30 USD/год. За результатами розрахунків річні витрати на ручне відновлення складають 30 744 USD (1 275 876

грн), одноразові витрати на впровадження дорівнюють 11 400 USD (473 100 грн), а щорічні витрати на експлуатацію системи у режимі pending_approval з Claude Sonnet 4.5 виходять на рівні 4 746 USD (196 952 грн). Річний економічний ефект становить 25 998 USD (1 078 917 грн), термін окупності близько 5,3 місяця, ROI за три роки досягає 584%. Додатково визначено низку нематеріальних вигод, як-от скорочення часу реакції, зниження психологічного навантаження інженерів, прозорість рішень та акумулювання корпоративної бази знань.

Сформульовано рекомендації щодо етапного впровадження системи за трифазною схемою (log_only → pending_approval → auto для вибіркових сценаріїв), визначено сім ключових ризиків впровадження разом із заходами їх пом'якшення, окреслено принципи моніторингу та спостережуваності, а також шість напрямів подальшого розвитку системи, тобто розширення типів помилок, підтримка альтернативних сховищ та оркестраторів, активне навчання на історичних рішеннях оператора, інтеграція з системами інцидент-менеджменту та предиктивний режим.

Результати розділу підтверджують практичну цінність розробленої інформаційної технології та її готовність до впровадження на підприємствах з аналогічною ETL-інфраструктурою. У загальних висновках роботи підсумовуються основні наукові та прикладні результати дослідження.

# ВИСНОВКИ

У межах кваліфікаційної роботи розв'язано задачу автоматичного відновлення ETL-пайплайнів при дрейфі схеми даних із застосуванням великих мовних моделей. За результатами дослідження отримано такі основні висновки.

1. Аналіз предметної області показав, що наявні підходи, а саме ручне виправлення, retry-механізми, схемна валідація та платформи data observability, не покривають повного циклу самовідновлення. Найпоширенішою причиною інцидентів є дрейф схеми у формах varchar_overflow, type_mismatch та new_column.

2. Запропоновано чотирифазну модель самовідновлення: збирання контексту, LLM-діагностування, опційне підтвердження оператором та виконання виправлень. Три режими роботи (log_only, auto, pending_approval) забезпечують поступовий перехід від ручного контролю до повної автоматизації.

3. Розроблено метод збирання контексту помилки з шести джерел: інформація про збій, метадані DAG, DDL-опис таблиці, вихідний код скрипту, логи CloudWatch та профіль даних. Алгоритм schema diff детермінованим чином зіставляє DDL з профілем і формує сигнал для LLM.

4. Модель діагностування на основі structured output фреймворку LangChain повертає Pydantic-схему з кореневою причиною збою, виправленнями трьох типів та кількісною оцінкою достовірності. Fallback-механізм із ручним парсингом JSON забезпечує сумісність із провайдерами без підтримки structured output на рівні API.

5. Безпековий шар реалізовано за принципом defense in depth: SQL-команди фільтруються через whitelist, деструктивні операції блокуються за blacklist, синтаксис Python перевіряється функцією compile(). Процедура застосування змін охоплює три рівні: файлова система, S3 та pull request у GitHub.

6. Веб-інтерфейс FastAPI + Streamlit реалізує модель human-in-the-loop. Оператор підтверджує або відхиляє виправлення, повторно запускає

діагностування з власним контекстом та переглядає версіоновану історію діагнозів.

7. Систему реалізовано у вигляді близько 1700 рядків Python з інтеграцією Apache Airflow 3.0, AWS Glue, Amazon Redshift Serverless та GitHub. Код покрито 80 модульними тестами (покриття 87%).

8. Експеримент (140 прогонів, 4 сценарії дрейфу, 7 LLM-моделей, 5 повторів) показав точність діагностування 87,1% та частку end-to-end відновлення 80,0%. Час репарації - від 11 до 40 секунд проти 30-90 хвилин при ручному відновленні.

Наукова новизна: удосконалено модель відновлення ETL-пайплайнів шляхом інтеграції детерміністичного schema diff із LLM-діагностуванням; дістав подальшого розвитку метод збирання контексту з шести джерел; удосконалено модель безпечного виконання виправлень із багаторівневою валідацією; уперше запропоновано модель human-in-the-loop для систем самовідновлення ETL-пайплайнів.

Практичну цінність роботи підтверджено результатами техніко-економічного обґрунтування. За базовим сценарієм бази практики (840 інцидентів на рік, середній час ручного відновлення 1 година, ставка інженера даних 30 USD/год) сукупні витрати на ручне відновлення становлять 30 744 USD/рік, тоді як щорічні витрати на експлуатацію системи у режимі pending_approval не перевищують 4 746 USD/рік. Різниця цих величин формує прогнозований річний економічний ефект на рівні 25 998 USD/рік. Термін окупності одноразових інвестицій становить 5,3 місяця. Показник ROI за трирічним горизонтом сягає 584%

Перспективи подальших досліджень охоплюють кілька напрямів. По-перше, доцільним є розширення переліку підтримуваних типів помилок за межі дрейфу схеми, тобто включення помилок з'єднання, проблем продуктивності, помилок прав доступу. Окремий напрям становить впровадження активного навчання на основі історичних рішень оператора, де підтверджені та відхилені діагнози використовуються як few-shot приклади у промпті.

# СПИСОК ВИКОРИСТАНИХ ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. IDC. Data Age 2025: The Evolution of Data to Life-Critical : IDC White Paper, Sponsored by Seagate. 2017. 25 p. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (дата звернення: 15.01.2026).

2. Kimball R., Ross M. The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling. 3rd ed. Indianapolis : Wiley, 2013. 672 p.

3. Gartner. The Real Cost of Downtime : Gartner Research. 2014. URL: https://www.gartner.com/en/documents/2824920 (дата звернення: 17.01.2026).

4. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O'Reilly Media, 2017. 616 p.

5. Monte Carlo Data. The State of Data Quality in 2022 : Monte Carlo Data Report. 2022. URL: https://www.montecarlodata.com/reports/state-of-data-quality-2022 (дата звернення: 19.01.2026).

6. Amazon Web Services. Amazon Redshift Database Developer Guide : AWS Documentation. 2024. URL: https://docs.aws.amazon.com/redshift/latest/dg (дата звернення: 22.01.2026).

7. Apache Software Foundation. Apache Airflow Documentation. 2024. URL: https://airflow.apache.org/docs/ (дата звернення: 23.01.2026).

8. Fivetran. The State of Data Engineering 2023 : Fivetran Report. 2023. URL: https://www.fivetran.com/resources/state-of-data-engineering-2023 (дата звернення: 25.01.2026).

9. Prefect. Workflow Orchestration for Modern Data Stacks : Prefect Documentation. 2024. URL: https://docs.prefect.io (дата звернення: 28.01.2026).

10. Great Expectations. Great Expectations Documentation. 2024. URL: https://docs.greatexpectations.io (дата звернення: 30.01.2026).

11. dbt Labs. dbt Documentation: Tests. 2024. URL: https://docs.getdbt.com/docs/build/tests (дата звернення: 02.02.2026).

12. Apache Software Foundation. Apache Iceberg Table Specification. 2024. URL: https://iceberg.apache.org/spec/ (дата звернення: 05.02.2026).

13. Monte Carlo Data. Data Observability Platform Documentation. 2024. URL: https://docs.montecarlodata.com (дата звернення: 07.02.2026).

14. Brown T. B., Mann B., Ryder N. et al. Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems. 2020. Vol. 33. P. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html (дата звернення: 09.02.2026).

15. Chen M., Tworek J., Jun H. et al. Evaluating Large Language Models Trained on Code : arXiv preprint arXiv:2107.03374. 2021. 35 p. URL: https://arxiv.org/abs/2107.03374 (дата звернення: 10.02.2026).

16. Le Goues C., Nguyen T., Forrest S., Weimer W. GenProg: A Generic Method for Automated Software Repair. IEEE Transactions on Software Engineering. 2012. Vol. 38, № 1. P. 54–72. DOI: 10.1109/TSE.2011.104.

17. Jimenez C. E., Yang J., Wettig A. et al. SWE-bench: Can Language Models Resolve Real-world Github Issues? : arXiv preprint arXiv:2310.06770. 2023. 60 p. URL: https://arxiv.org/abs/2310.06770 (дата звернення: 12.02.2026).

18. SWE-bench Leaderboard. 2025. URL: https://www.swebench.com (дата звернення: 14.02.2026).

19. Ahmed I., Lepczynski P., Vij S. et al. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models : arXiv preprint

arXiv:2301.03797. 2023. 12 p. URL: https://arxiv.org/abs/2301.03797 (дата звернення: 16.02.2026).

20. Jiang J., Zheng Y., Wang B. et al. A Survey on Large Language Models for Software Engineering : arXiv preprint arXiv:2312.15223. 2023. 28 p. URL: https://arxiv.org/abs/2312.15223 (дата звернення: 18.02.2026).

21. Pydantic. Pydantic Documentation v2. 2024. URL: https://docs.pydantic.dev/latest (дата звернення: 21.02.2026).

22. Ji Z., Lee N., Frieske R. et al. Survey of Hallucination in Natural Language Generation. ACM Computing Surveys. 2023. Vol. 55, № 12. P. 1–38. DOI: 10.1145/3571730.

23. Anthropic. Claude API Reference: Context Window. 2024. URL: https://docs.anthropic.com/claude/reference (дата звернення: 25.02.2026).

24. Liu V. X., Gao Y., Weimer W. et al. LLM-based Test-driven Interactive Code Generation. IEEE Transactions on Software Engineering. 2024. Vol. 50, № 9. P. 2217–2233. DOI: 10.1109/TSE.2024.3393070.

25. OpenAI. API Pricing. 2024. URL: https://openai.com/pricing (дата звернення: 28.02.2026).

26. LangChain. LangChain Documentation. 2024. URL: https://python.langchain.com/docs (дата звернення: 03.03.2026).

27. Amazon Web Services. AWS Glue Developer Guide. 2024. URL: https://docs.aws.amazon.com/glue/latest/dg (дата звернення: 06.03.2026).

28. Amazon Web Services. Amazon CloudWatch Logs User Guide. 2024. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs (дата звернення: 09.03.2026).

29. GitHub. GitHub REST API Documentation. 2024. URL: https://docs.github.com/en/rest (дата звернення: 12.03.2026).

30. FastAPI. FastAPI Documentation. 2024. URL: https://fastapi.tiangolo.com (дата звернення: 15.03.2026).

31. Streamlit. Streamlit Documentation. 2024. URL: https://docs.streamlit.io (дата звернення: 18.03.2026).

32. PyGithub. PyGithub Documentation. 2024. URL: https://pygithub.readthedocs.io (дата звернення: 21.03.2026).

33. Amazon Web Services. Amazon Redshift Data API. 2024. URL: https://docs.aws.amazon.com/redshift/latest/mgmt/data-api.html (дата звернення: 24.03.2026).

34. boto3. Boto3 Documentation. 2024. URL: https://boto3.amazonaws.com/v1/documentation/api/latest (дата звернення: 27.03.2026).

35. Python Software Foundation. unittest : Unit Testing Framework. Python 3.12 Documentation. 2024. URL: https://docs.python.org/3/library/unittest.html (дата звернення: 30.03.2026).

36. pytest. pytest Documentation. 2024. URL: https://docs.pytest.org (дата звернення: 02.04.2026).

37. Anthropic. Claude claude-sonnet-4-5 Model Card. 2024. URL: https://www.anthropic.com/claude (дата звернення: 05.04.2026).

38. OpenAI. GPT-4o Technical Report. 2024. URL: https://openai.com/gpt-4o (дата звернення: 08.04.2026).

39. Google. Gemini 2.5 Flash Technical Report. 2024. URL: https://deepmind.google/technologies/gemini (дата звернення: 11.04.2026).

40. DeepSeek. DeepSeek-V3 Technical Report. 2024. URL: https://github.com/deepseek-ai/DeepSeek-V3 (дата звернення: 14.04.2026).

# Додатки

## Додаток А

## Лістинг ключових модулів системи

### А.1 Модуль models.py — базові моделі даних (55 рядків)

```
from dataclasses import dataclass, field, asdict
from typing import Optional


@dataclass
class ErrorInfo:
      task_id: str
      dag_id: str
      execution_date: str
      exception_type: str
      exception_message: str
      traceback: str
      log_url: Optional[str]
      try_number: int


@dataclass
class ColumnDDL:
      name: str
      data_type: str
      max_length: Optional[int] = None


@dataclass
class ColumnProfile:
      name: str
      dtype: str
      null_count: int
      non_null_count: int
      max_length: Optional[int] = None


@dataclass
class SchemaDiffEntry:
      column: str
      diff_type: str  # "new_column" | "varchar_overflow" | "type_mismatch"
      detail: dict = field(default_factory=dict)


@dataclass
class FailureContext:
      timestamp: str
      error_info: ErrorInfo
      dag_file_source: str
      create_table_sql: str
      etl_script_source: str
      ddl_columns: list[ColumnDDL]
      data_profile: list[ColumnProfile]
      schema_diff: list[SchemaDiffEntry]
      glue_error_log: str = ''
      glue_output_log: str = ''
      metadata: dict = field(default_factory=dict)

      def to_dict(self) -> dict:
      return asdict(self)
```

### А.2 Модуль schema_diff.py — алгоритм порівняння схем (77 рядків)

```
from repair_workflow.models import ColumnDDL, ColumnProfile, SchemaDiffEntry

# Mapping from pandas/pyarrow dtype to the expected Redshift type family
_DTYPE_TO_REDSHIFT = {
      'object': 'VARCHAR',
```

```
      'string': 'VARCHAR',
      'string[python]': 'VARCHAR',
      'string[pyarrow]': 'VARCHAR',
      'bool': 'BOOLEAN',
      'boolean': 'BOOLEAN',
      'int64': 'INTEGER',
      'int32': 'INTEGER',
      'Int64': 'INTEGER',
      'float64': 'DOUBLE',
      'float32': 'FLOAT',
      'datetime64[ns]': 'TIMESTAMP',
      'datetime64[ns, UTC]': 'TIMESTAMP',
}


def compute_schema_diff(
      ddl_columns: list[ColumnDDL],
      data_profile: list[ColumnProfile],
) -> list[SchemaDiffEntry]:
      diffs = []
      ddl_lookup = {col.name.lower(): col for col in ddl_columns}

      for profile in data_profile:
      col_name = profile.name.lower()
      ddl_col = ddl_lookup.get(col_name)

      if ddl_col is None:
            diffs.append(SchemaDiffEntry(
                column=profile.name,
                diff_type='new_column',
              detail={
                    'data_dtype': profile.dtype,
                    'max_length': profile.max_length,
              },
              ))
              continue

      # Check VARCHAR overflow
      if (
            ddl_col.data_type == 'VARCHAR'
              and ddl_col.max_length is not None
              and profile.max_length is not None
              and profile.max_length > ddl_col.max_length
      ):
            diffs.append(SchemaDiffEntry(
                column=profile.name,
                diff_type='varchar_overflow',
              detail={
                    'data_max_len': profile.max_length,
                    'ddl_max_len': ddl_col.max_length,
              },
              ))

      # Check type mismatch
        expected_redshift_type = _DTYPE_TO_REDSHIFT.get(profile.dtype)
      if (
            expected_redshift_type
              and expected_redshift_type != ddl_col.data_type
              # VARCHAR is compatible with string/object dtypes
              and not (profile.dtype in ('object', 'string', 'string[python]', 'string[pyarrow]') and
ddl_col.data_type == 'VARCHAR')
      ):
            diffs.append(SchemaDiffEntry(
                column=profile.name,
                diff_type='type_mismatch',
              detail={
                    'data_dtype': profile.dtype,
                    'ddl_type': ddl_col.data_type,
                    'expected_redshift_type': expected_redshift_type,
              },
              ))

      return diffs
```

## A.3 Модуль executor/safety.py — безпековий шар (49 рядків)

```
"""Safety validation for proposed fixes."""
import re
import logging

logger = logging.getLogger(__name__)

# SQL statements we allow the executor to run
_ALLOWED_SQL_PATTERNS = [
```

```
        # ALTER TABLE schema.table ADD COLUMN name TYPE
        re.compile(
          r'^\s*ALTER\s+TABLE\s+\S+\s+ADD\s+(COLUMN\s+)?\S+\s+\S+',
        re.IGNORECASE,
        ),
        # ALTER TABLE schema.table ALTER COLUMN name TYPE typename
        re.compile(
          r'^\s*ALTER\s+TABLE\s+\S+\s+ALTER\s+COLUMN\s+\S+\s+TYPE\s+\S+',
        re.IGNORECASE,
        ),
]

_FORBIDDEN_SQL_KEYWORDS = re.compile(
    r'\b(DROP|DELETE|TRUNCATE|INSERT|UPDATE|CREATE\s+TABLE|GRANT|REVOKE)\b',
      re.IGNORECASE,
)


def validate_sql_ddl(sql: str) -> tuple[bool, str]:
    """Check that the SQL is a safe ALTER TABLE statement."""
      sql = sql.strip().rstrip(';').strip()
      if not sql:
      return False, 'Empty SQL statement'

      if _FORBIDDEN_SQL_KEYWORDS.search(sql):
      return False, f'SQL contains forbidden keyword: {sql[:80]}'

      for pattern in _ALLOWED_SQL_PATTERNS:
      if pattern.match(sql):
              return True, ''

      return False, f'SQL does not match any allowed pattern: {sql[:80]}'


def validate_python_syntax(code: str, label: str = '<fix>') -> tuple[bool, str]:
    """Check that a string is valid Python by compiling it."""
      try:
      compile(code, label, 'exec')
      return True, ''
      except SyntaxError as e:
      return False, f'Syntax error at line {e.lineno}: {e.msg}'
```

# Додаток Б

# Структура програмних модулів та тестового покриття

*Таблиця Б.1 – Структура програмних модулів системи*

| Модуль | Рядків | Призначення |
|---|---|---|
| callback.py | 143 | Головний оркестратор чотирифазного циклу |
| context_builder.py | 106 | Побудова FailureContext з 6 гетерогенних джерел |
| schema_diff.py | 77 | Детерміністичний алгоритм порівняння схем |
| models.py | 55 | Базові моделі даних (ErrorInfo, FailureContext та ін.) |
| storage.py | 29 | Збереження контексту як JSON-артефакту |
| diagnosis/engine.py | 110 | LLM-діагностування через LangChain structured output |
| diagnosis/models.py | 98 | Pydantic-схема DiagnosisOutput та dataclass Diagnosis |
| diagnosis/prompt.py | 137 | Побудова системного та користувацького промптів |

| | | |
|---|---|---|
| diagnosis/storage.py | 62 | Збереження діагнозу та промпту як артефактів |
| executor/fix_executor.py | 200 | Маршрутизація виправлень до обробників |
| executor/safety.py | 49 | Whitelist/blacklist валідація SQL та Python |
| executor/github_pr.py | 558 | Створення Pull Request у GitHub через bridge-сервер |
| executor/handlers/sql_ddl.py | 96 | Виконання ALTER TABLE через Redshift Data API |
| executor/handlers/etl_script.py | 276 | Застосування змін до ETL-скрипту (S3 + GitHub) |
| executor/handlers/dag_code.py | 169 | Оновлення DAG-файлів (локально + GitHub) |
| executor/handlers/base.py | 29 | Базовий інтерфейс обробника виправлень |
| executor/storage.py | 37 | Збереження результатів виконання виправлень |
| sources/error_info.py | 52 | Витягування ErrorInfo з Airflow callback |
| sources/dag_metadata.py | 183 | Читання метаданих DAG (SQL, шлях скрипту) |
| sources/ddl_parser.py | 36 | Парсинг CREATE TABLE SQL у список ColumnDDL |
| sources/etl_script.py | 54 | Завантаження ETL-скрипту з S3 або локально |
| sources/cloudwatch_logs.py | 207 | Читання логів та профілю даних з CloudWatch |
| ui/backend/ (6 модулів) | 856 | FastAPI: маршрути, сервіси, моделі API |
| ui/frontend/ (5 модулів) | 659 | Streamlit: timeline, detail, approval queue |
| experiment_runner/ (2 модулі) | 1178 | Автоматизація експериментів та аналіз |
| **Разом** | **5456** | 37 модулів Python |

*Таблиця Б.2 – Покриття модулів модульними тестами*

| Тестовий файл | Рядк. | Тест. | Що тестується |
|---|---|---|---|
| test_schema_diff.py | 96 | 12 | Усі підтипи schema drift |
| test_context_builder.py | 204 | 8 | Побудова контексту, обробка помилок |
| test_diagnosis_engine.py | 218 | 9 | Structured output, fallback, токени |
| test_diagnosis_prompt.py | 179 | 7 | Побудова промпту, truncation |
| test_fix_executor.py | 193 | 8 | Маршрутизація, режими approval |
| test_safety.py | 65 | 6 | Whitelist/blacklist SQL, Python syntax |
| test_github_pr.py | 171 | 7 | Створення PR через bridge |
| test_error_info.py | 77 | 5 | ErrorInfo з Airflow context |
| test_cloudwatch_logs.py | 66 | 4 | Парсинг логів CloudWatch |
| test_ddl_parser.py | 47 | 4 | Парсинг CREATE TABLE SQL |

| | | | |
|---|---|---|---|
| **Разом** | **1316** | **70** | **Покриття коду 87 %** |

# Додаток В

## Приклад структурованого діагнозу системи


```
{
  "root_cause": "Column user_id in table ads_raw.tiktok_campaign_stats
      is defined as VARCHAR(20) but incoming data contains values up to
      23 characters, causing COPY failure.",
  "failure_category": "schema_drift",
  "proposed_fixes": [
      {
      "fix_type": "sql_ddl",
      "description": "Extend user_id column to VARCHAR(30)",
      "sql": "ALTER TABLE ads_raw.tiktok_campaign_stats
              ALTER COLUMN user_id TYPE VARCHAR(30);",
      "code_change": null,
      "target": "ads_raw.tiktok_campaign_stats"
      },
      {
      "fix_type": "dag_code_change",
      "description": "Update CREATE TABLE SQL in DAG",
      "sql": null,
      "code_change": "user_id VARCHAR(30) NOT NULL",
      "target": "tiktok_ads_daily.py"
      }
  ],
  "confidence": 0.97,
  "reasoning": "1. Schema diff: varchar_overflow on user_id (20 < 23).
      2. Traceback confirms COPY failure on column width.
      3. Fix: ALTER TABLE to VARCHAR(30) with safety margin.",
  "false_positive_flags": [],
  "model": "claude-sonnet-4-20250514",
  "prompt_tokens": 3247,
  "completion_tokens": 412,
  "timestamp": "2026-04-15T08:32:32.103Z",
  "duration_seconds": 17.582
}
```