Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра інформатики

РОЗРОБКА ФРЕЙМВОРКУ ДЛЯ МАСШТАБОВАНОЇ ETL-СИСТЕМИ

Текстова частина до магістерської роботи

За спеціальністю 121 «Інженерія програмного забезпечення»

Керівник магістерської роботи
доцент, к.ф-м.н. С.С. Гороховський
(прізвище та ініціали)

_____
(підпис)
“__”_____ 2024 р.

Виконав студент В.Г. Копійка
(прізвище та ініціали)

_____
(підпис)
“__”_____ 2024 р.

Київ-2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

доцент, к.ф-м.н.

_____ С.С. Гороховський

(підпис)

“__”_____ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на магістерську роботу

студенту Копійці Вадиму Геннадійовичу

Факультету інформатики 2 р.н. магістерської програми Інженерія програмного забезпечення

ТЕМА: Розробка фреймворку для масштабованої ETL-системи

Зміст ТЧ до курсової роботи:

Анотація
Вступ
Аналіз предметної області
Аналіз практик в етапах ETL, теоретичні відомості та вибір технлогій
Реалізація фреймворку для масштабованої ETL-системи
Висновки
Список використаних джерел
Додатки

Дата видачі “___” _____ 2023 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Розробка фреймворку для масштабованої ETL-системи

Календарний план виконання роботи:

№	Назва етапу курсової роботи	Термін виконання	Примітка
1.	Отримання завдання магістерської роботи	10.10.2023	
2.	Пошук джерел за тематикою	20.11.2023	
3.	Ознайомлення з відповідною літературою	20.12.2023	
4.	Огляд існуючих аналогів/пропозицій	20.02.2024	
5.	Планування структури розділів текстової частини роботи	27.02.2024	
6.	Проектування складових практичної частини	10.03.2024	
7.	Написання першого розділу роботи	15.03.2024	
8.	Початок написання другого розділу	10.04.2024	
9.	Створення компоненту середовища оркестратора та сервісів для ETL	20.04.2024	
10.	Аналіз літератури стосовно алгоритимів масштабування сервісів	30.04.2024	
11.	Створення компоненту мікросервісів для керування середовищем Swarm	08.05.2024	
12.	Завершення другого розділу та уточнення назви теми з керівником	09.05.2024	
	Передзахист виконання роботи	17.05.2024	
13.	Завершення написання фінального розділу роботи	31.05.2024	
14.	Надсилання розділів роботи керівнику	01.06.2024	
15.	Дооформлення роботи	02.06.2024	
16.	Створення презентації та здача роботи	03.06.2024	

Студент Копійка В.Г.

Керівник Гороховський С.С.

“__”_____ 2024 р.

ЗМІСТ

АНОТАЦІЯ

Дана дипломна робота описує процес розробки фреймворку для масштабованої ETL-системи, що і є її метою. Для досягнення мети роботи було проведено аналіз предметної області, що включає огляд особливостей та понять у системах ETL, наведено еволюцію таких систем та їхніх етапів.

Також здійснено покроковий аналіз практик в побудові кожного з комопонентів обробки даних, зроблено обгрунтований вибір технологій для реалізації етапів у процесі системи ETL.

Описана розробка трьох головних складових фреймворку, до яких входить мікросервісна компонента для побудови та керування станом системи, компонента середовища оркекстрації сервісів фреймворку та системи ETL, а також компонента графічного веб-інтерфейсу для безпосердньої роботи з фреймворком.

Як наслідок, результатом роботи став фреймворк, який здатний полегшити побудову масштабованої ETL-системи без надмірного втручання зі сторони користувача.

Ключові слова: ETL, ELT, EtLT, архітектура, видобування даних, перетворення даних, вивантаження даних, Data Lake, Data Lakehouse, обробка даних, система керування, мікросервіси, Docker, Swarm, контейнери, масштабування, середовище оркестрації.

ПЕРЕЛІК ЗАЛУЧЕНИХ СКОРОЧЕНЬ

ETL (Extract Transform Load) – архітектура системи, яка відповідає етапам видобування, перетворення та завантаження даних.

ELT (Extract Load Transform) - архітектура системи, яка відповідає поступовим етапам видобування, завантаження та перетворення даних.

EtLT (Extract tweak-transform Load Transform) - архітектура системи, що полягає в поступових етапах видобування, малих коригувань-перетворень, завантаження та перетворення даних.

REST (Representational State Transfer) - передача представницького стану.

API (Application Program Interface) – програмний інтерфейс для взаємодії

DL (Data Lake) – сховища неструктурованих даних.

DWH (Data Warehouse) – сховище структурованих даних.

DLH (Data Lakehouse) – сховище на базі поєднання якостей DWH та DL.

SQL (Structured Query Language) – мова для структурованих запитів.

PID (Proportional Integral Derivative) – пропорційна інтегральна похідна.

UI (User Interface) – користувацький інтерфейс.

DB (Database) – база даних.

ВСТУП

Актуальність теми та її практична значущість

Обробка даних є невід'ємною частиною більшості систем та процесів, а в епоху Big Data вона стала відігравати ще важливішу роль. Великі дані з самого початку були охарактеризовані трьома якостями, які часто позначають як 3Vs: volume (від англ. - обсяг), variety (від англ. - різноманітність) та velocity (від англ. - швидкість) [1]. Кількість цих якостей збільшується, а тому й викликів, з яким доводиться стикатися під час обробки даних, стає більше. Так зростає й кількість організацій різних типів і розмірів, які зацікавлені в тому, аби якомога ефективніше впоратися з Big Data в своїх галузях. Вони прагнуть обробляти все більше даних і отримувати від них ще більше користі, адже це дозволяє їм отримувати як перевагу над конкурентами, так і важливу інформацію, що стосується роботи їхніх систем у цілому.

Процес ETL як підхід до повного циклу обробки даних, що складався з фаз отримання, перетворення й збереження, з'явився ще до початку епохи Big Data як такої. Саме тому в той час, коли ресурси для збереження даних були дорожичими, початковий вигляд процесу строго базувався на трьох згаданих послідовних фазах [2], де будь-якому зберженню даних обов'язково передував етап перетворень, де відбувалися всі необхідні очищення та приведення їх до деякого потрібного вигляду. Це дозволяло заощадити коштовні ресурси сховища, але це так само сильно впливало на такі важливі якості отриманих даних, як актуальність та повноту. Більше того, традиційною складовою будь-якої ETL до пожвавлення розвитку Bid Data завжди був batch-processing (від англ. – обробка партіям), що передбачав достатньо велику затримку між появою і збереженням даних в залежності від обраної періодичності виконання процесу.

Очікувано, що в сучасному світі користь від отримання результатів, що з'являється лише з деякою періодичністю, зменшується. Збільшується потреба в

stream-processing (від англ. – потокова обробка) та безперевності ETL як такого, адже це дозволить зменшити проміжок часу між появою даних в джерелах та безпосередньою можливістю для фахівців взаємодіяти з ними. Таким чином, теперішні ETL-рішення, хоч часто й несуть таку абревіатуру або якось пов’язані з нею, значно змінилися в своїх архітектурах. Тепер змінюються не тільки сховища й технології, які є частиною процесу, а й самі етапи. Змінюється як порядок, так і їхня мета, що вже виходить за межі початкового перебігу ETL та його трьох фаз.

Як наслідок, теперішні рішення різняться одне від одного в багатьох аспектах, тож до тепер немає єдиного формалізованого або універсального підходу до побудови сучасного ETL [2]. Досить поширеними стають SaaS-сервіси, які виконують роль шаблонів для створення власних ETL-систем, зокрема до них можна віднести Aiven, Decodable, CloverDX. На жаль, ці сервіси не дають можливість слідкувати за використанням ресурсів на конкретних фазах процесу і в них відсутня прозорість використання даних користувачів. Білшість з них дозволяє здійснювати або batch-processing, або stream-processing, що не робить їх універсальними для використання для різних потреб. Більше того, через те, що ці сервіси виступають у ролі вже готових рішень із закритим кодом, у користувачів нема здатності легко розгорнути їхні сервіси локально для тестування або на хмарних потужностях за власним вибором, що могло б дозволити заощадити кошти та краще зрозуміти обсяг потрібних ресурсів.

Отже, оскільки вплив та поширення Big Data на важливість процесу обробки даних продовжує зростати, а сучасні рішення не мають достатньої прозорості й зручності, було прийнято рішення зробити метою цієї роботи розробку універсального як для batch-processing, так і для stream-processing, фреймворку для мастшабованої ETL-системи. Побудован система б відповідала сучасним дослідженням стосовно органазації етапів обробки даних і була б масштабованим рішенням, яке б задовольняло потребу в можливості розгортання всієї системи як локально, так і в хмарному середовищі.

Постановка задачі

1. Проаналізувати виклики, з якими стикалися протягом свого існування ETL-системи, та еволюцію їхньої аріхтектури з плином часу.

2. Розлянути сучасні практики в побудові етапів масштабованого ETL-процесу та відповідні технології, які використовується для цього, обгрунтувавши вибір кожної для використання під час розробки.

3. Здійснити реалізацію фреймворку для ETL-системи, який буде задовольняти потребу масштабування і полягатиме в наданні користувачу середовища для зручної побудови власного циклу обробки даних, залучайючи оптимальні практики, інструменти та технології, з відповідним розгортанням створеної системи локально та наведенням прикладу її використання.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Процес обробки даних в системі ETL та основні поняття

Перші згадки про процес, який би дозволив автоматизувати обробку необхідних даних, з’явилися ще на світанку цифровізації даних та мережі Інтернет як такої. Загальний розвиток інформаційних технологій неабияк вплинув на обсяги даних, які почали генеруватися (від англ. generate - створюватися) в усе більших і швидших темпах, що так само пришвидшило й розвиток підходів до їх обробки. Тож поява початкового ETL-процесу, основи якого ми знаємо сьогодні, сталася в сімдесятих-вісімдесятих роках минулого століття, коли сховища даних стали все більше нагадувати сьогоднішні бази даних. Одним з найважливіших чинників, які спровокували розвиток обробки й аналітики даних став як розквіт і поширення реляційних сховищ, так збільшення кількості даних які було можливо отримати й зберегти. Реляційні бази даних відокремили сутності й дозволили доступатися до екземплярів за межами транзакцій, що було неможливим в епоху транзакційних баз. Адже раніше відомості про користувача мали надсилатися з кожною транзакцією, а централізованого й уніфікованого способу доступатися до актуальних даних користувачів не існувало [3]. Саме тому, поява реляціних баз даних дозволила аналітикам отримувати більше користі від збереженого в сховищах. Коли кількість даних почала невпинно збільшуватися, а здійснення якісної аналітики цих даних ставало ще важливішою частиною успішного функціонування бізнесу, автоматизований й ефективний процес обробки таких даних став невід’ємною частиною в повсякденній роботі організацій. Більше того, зменшення ціни на доступний об’єм для збереження даних і розвиток постачальників хмарних ресурсів [2], які дозволили мати одночасно розподілену систему з майже необмеженим сховищем, максимальний обсяг якого залежить лише від потреб та можливостей

того чи іншого бізнесу, дозволили процесу ETL отримати новий виток розвитку та популяризації.

Починаючи від сімдесятих років минулого століття, повний цикл обробки даних, що включав збирання даних у початкову вигляді й збереження їх у потрібному для бізнесу форматі, почав поступово формалізовуватися. Сьогодні багатьом відома в галузі інформаційних технологій абревіатура ETL, що позначає систему повного процесу обробки даних, має три чітко визначені складові:

4. Extraction (від англ. – видобування, отримання) – знаменує першу літеру процесу і є незмінно першим етапом у ньому, адже саме він є початковою точкою всього pipeline (від англ. - конвеєру), без якого інші фази не здатні розпочатися. У цій фазі здійснюється отримання інформації зі вказаних джерел у вигляді, що безпосередньо притаманний цим джерелам.

5. Transformation (від англ. – перетворення, зміна) – знаменує другу літеру процесу, але при цьому не є обов’язково другим по порядку етапом у процесі, особливо якщо мова йде про більш сучасні підходи до побудови ETL, про що йтиметься пізніше. Ця фаза є важливим етапом всього циклу, на якому відбуваються перетворення у відповідності до правил і потреб конкретного бізнесу, що необхідні для цільового збереження вхідних даних, які вже будуть корисними для уніфікованої взаємодії зі сторони фахівців, зокрема аналітиків. Тут часто відбувається валідація даних, їхня очистка та приведення до конкретного вигляду [4].

6. Load (від англ. – вантаження) – знаменує останню літеру абревіатури, але, як і у випадку з попердньою фазою, не обов’язково має йти останньою в черзі виконання процесу. На цьому етапі відбувається збереження отриманих даних, рівень перетворення й форматування яких також залежить від конкретного підходу до імплементації ETL.

Усі ці фази, хоч і формалізувалися ще в минулому столітті, є невід’ємною частиною будь-якої сучасної ETL-системи, при цьому їхній порядок виконання в

конвеєрі вже давно почав змінюватися, але від цього кінцева мета процесу як такого, що полягає в переміщенні даних з різних неуніфікованих джерел до цільових сховищ у потрібному форматі, лишається незмінною.

1.2 Виклики у роботі систем ETL

Як і будь-який процес, а особливо той, який має багато складових, ETL стикнувся з викликами, що стосувалися різних аспектів. До них належить забезпечення якості даних, складність імлементації косплексних перетворень та якості підпроцесів в ETL в цілому. На сьогодні одним з найбільших викликів для ETL виступає потреба в можливості обробляти дані в наближеному до реального часі. Якщо раніше бізнесу було достатньо мати можливість аналізувати історичні дані, які були отримані кілька годин, днів або тижнів тому, то зараз цього вже не достатньо. З розвитком різних галузей та IT в цілому бізнеси змушені швидше реагувати на зміни та перебіг процесів, аби ті могли вчасно приймати важливі для їхнього становища рішення. До галузей, які потребують обробку в майже реальному часі відносять електронну комерцію, телекомунікацію, медицину та кібербезпеку, адже для всіх них критично важливо отримувати актуальні дані, які дозволяють їм ефективно та надійно функціонувати. Як наслідок, batch-processing все більше відходить на другий план, лишаючись обов’язковим, але ніяк не найактуальнішим типом обробки даних в сучасному світі. Він не здатний надати одну з найважливіших якостей будь-якої інформації, а саме - актуальність. Цей виклик «реального часу» полягає в кількох проблемах, які слід вирішити одночасно, а саме здатність забезпечувати таку обробку для даних, які швидко надходять і які мають великі обсяги. Тож маємо прямий вплив Big Data на появу нових викликів, з якими має стикатися сучасний процес ETL.

Основною задачею забезпечення обробки в реальному часі є надання доступу до даних з джерела інформації якомого ближче до часу її появи на вхідному ресурсі.

Для того, щоб впоратися з викликом «реального часу», часто виділяють такі ключові якості, які повинна мати ETL-система:

- Низька затримка – час між оновленням даних в джерелі, збиранням цих даних і збереженням їх у цільовому сховищі має бути зведений до мінміуму. Більше того, вплив на джерела даних також має бути мінімальним, аби не впливати на швидкодію всього процесу ще з самої початкової фази. Це дозволить зменшити вплив на працездатність джерала, який також має вчасно приймати й оновлення від своїх клієнтів, і підвищити пропускну здатність отримання даних в цілому.
- Висока доступність та відмовостійкість – забезпечити працездатність системи шляхом реплікацій проміжних сховищ та сервісів. Також здійснювати ефективне резервне копіювання для швидкого відновлення або переключання на інші проміжні сервіси. Наприклад, якщо ми ще не встигли обробити якісь дані, що вже не можна отримати безпосередньо з джерела, але при цьому вони ще були в черзі на виконання, то така черга має мати резервне сховище.
- Масштабованість та ефективність системи – забезпечити систему здатністю адаптуватися до змін в навантаженнях, аби вчасно збільшувати та зменшувати доступні для використання етапам ETL ресурси, що одночасно дозволить як підвищити пропускну здатність, так і заощаджувати, якщо буде можливість.

Згадані якості є невід’ємною частиною сучасних ETL, які мають впоратися з викликом доставки даних в наближеному до реального часі [5].

Оскільки дизайн процесу до сьогодні не має чіткої методології та визначених стандартів, побудова оптимальної архітектури всього ETL здебільшого базується на загальноприянятих практиках, які мають враховувати сучасні потреби та виклики. Таким чином, часто для визначення якості перебігу процесу використовуються метрики, що стосуються цілісності даних та ефективності роботи з ними, високодоступності системи та легкості в керуванні її ресурсами [2].

1.3 Еволюція в архітектурах та сховищах для систем ETL

Еволюція архітектури систем ETL супроводжувалися змінами в наявних типах централізованих сховищ, які були невід'мною частиною системи. Роль цих сховищ була настільки важливою, що саме вони й впливали на послідовність фаз всього процесу. Першим таким сховищем, навколо якого будувалися ETL-системи, був Data Warehouse (від англ. – сховище даних). Вперше така система для збереження даних була запропонована дослідниками з IBM наприкінці вісімдесятих років минулого століття, як рішення для суб'єктно орієнтованого, енергонезалежного й організованого сховища [6]. Незважаючи на те, що поняття DW з'явилося досить давно і стало традиційною складовою для збереження вже обробленої інформації, воно й досі лишається актуальним для різних імплементацій ETL. Data Warehouse відіграває роль централізованого місця збереження потрібної інформації з різних джерел для подальшого використання зі сторони бізнесу. Серед основних якостей такого сховища виділяють структурованість та якість даних, які зберігаються в ньому. Запис даних до такого сховища здійснюється за підходом schema-on-write (від англ. – схема на записі) [7], що передбачає визначення схеми даних заздалегідь запису. Відповідність до конктретної схеми має бути забезпечено шляхом приведення отриманих даних до потрібного вигляду. Часто такі сховища будуються навколо реляційних баз даних [8], аби наділити їх надійністю та строгістю, що забезпечується прицнипами ACID, що притаманні реляційним системам. Це дозовлило DW стати так званим single source of truth (від англ. – єдине джерело правди) для потреб бізнесу. Як наслідок, такий тип сховища був основною причиною початкої послідовності фаз в ETL, адже етап перетворень, який передує вантаженню, є невід'ємною частиною успішного збереження даних у DW [9]. Без нього підготовлення даних до визначеної в Data Warehouse схемі було б неможливим.

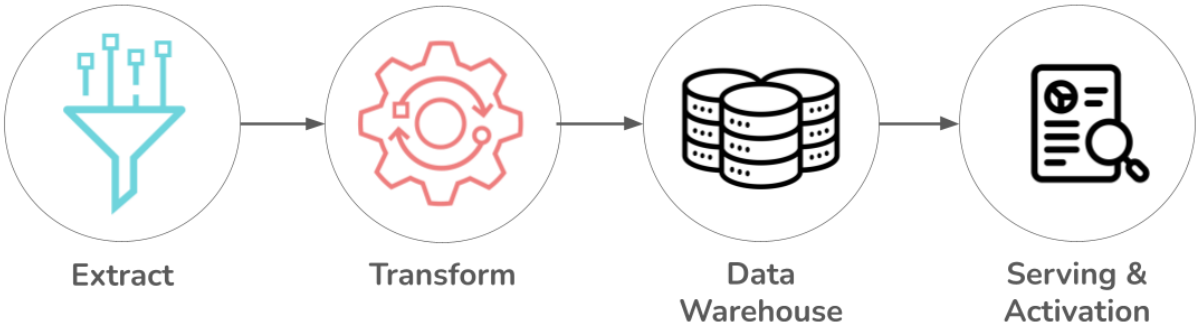


Рисунок 1.1 – Компоненти й етапи в ETL [10]

Вбачаючи особливості Data Warehouse, самотужки він вже не здатний задовольнити сучасні потреби в роботі з даними, які починають все більше ставати необхідними для конкурентноспроможного й ефективного функціонування бізнесу. До недоліків згаданого сховища можна віднести такі складнощі:

- Робота з різними неструктурованими даними, приведення яких до потрібного формату може призвести до погіршення швидкодії всього процесу ETL [9].
- Пристосування до змін вхідних даних, неправильний підхід в якому може призвести до пошкодження організованості даних та зв’язків між ними, на які було вже витрачено час [9].
- Робота з даними в реальному часі, що може неабияк впливнути на доступність і швидкодію сховища через непристосованість до великої кількості операцій зі вставки інформації внаслідок shema-on-write підходу [9].

Згадані вище аспекти не дозволяють повноцінно впоратися з викликами, що з’являються в епоху Big Data, використовуючи Data Warehouse. Більше того, DW обмежує повноту отриманих даних, що надходять з джерел. Внаслідок цього окремі частини інформації, які б могли стати цінними для аналітики чи машинного навчання, просто відсутні. Якщо мова йтиме про потокову інформацію або таку, що доступна лише протягом деякого проміжку часу у власних джерелах, то такі дані взагалі зникають назавжди, що неабияк може вплинути на можливість бізнесу вчасно приймати рішення.

Як насділок, з часом з’явилася потреба в новому рішенні, яке б могло допомогти впоратися зі складнощами, що були присутні під час використання Data Warehouse. Тож наступним кроком в розвитку залучених в системи ETL сховищ став так званий Data Lake. Вперше запропонований компанією Pentaho в 2011 році [11], цей тип сховища мав стати новим підходом до збереження даних, адже він не передбачав необхідність визначення структури та взаємозв’язків між даними. Таким чином, DL став відігравати роль, як і у випадку DW, централізованого сховища, але такого, що здатне зберігати в собі як струтктуровані, так напівструктуровані та неструктуровані дані різного розміру. Тож таке сховище зазвичай має пласку або ієрархічну архітектуру з файлів, але частіше першу. Воно передбачає збереження записів на одному рівні і надання кожному з них унікального ідентифікатора разом з допоміжними метаданими [6]. Часто вантаження даних в Data Lake здійснюється у специфічному форматі файлу на кшталт Apache Parquet, що дозволяє зберігати початковий стан інформації без будь-яких перетворень, які б впливали на її повноту [7]. Враховуючи цю особливість, даний тип сховища слідує підходу schema-on-read (від англ. - схема на читанні) [2], що полягає у відсутності визначення схеми даних, що передує запису, але передбачає здійснення перетворень даних вже під час читання, аби отримати користь від збереженого. У зв’язку з появою DL потреба у фазі перетворень після отримання даних в ETL процесі перестала бути обов’язковою. Тож нове сховище спровокувало одну з найбільший змін в архітектурі процесу ETL, яка призвела до появи вже новою абревіатури – ELT. Це зміна символізувала переміщення фази трансформації даних на останнє місце, дозволивши неабияк зменшити час між появою даних в джерелі з їхнім збереженням в цільовому сховищі, де вони вже будуть доступні бізнесу для використання.

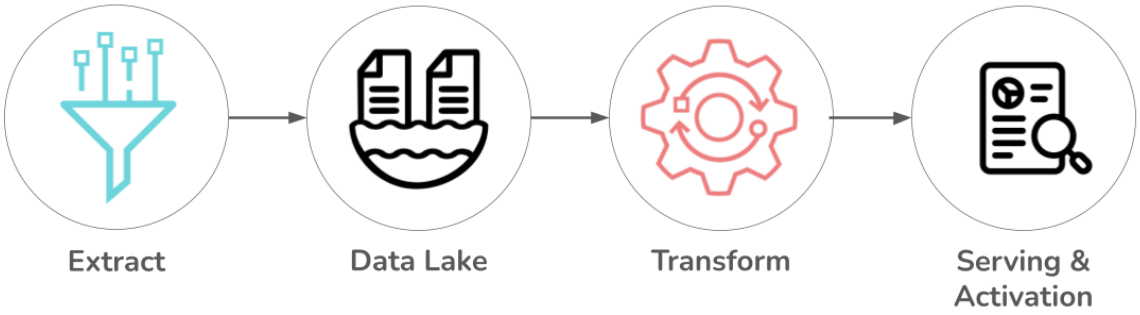


Рисунок 1.2 – Компоненти й етапи в ELT [10]

Як наслідок, якості, якими наділений Data Lake, дозволили вирішити складнощі з якими стикалися бізнеси під час використання Data Warehouse. Нове сховище зробило взаємодію з даними різної структури в реальному часі можливим, що полегшило адаптацію до Big Data в цілому. До того ж, процес перетворення з такого, що відбувається регулярно, перейшов у стан on-demand (від англ. – на вимогу), що неабияк зменшило навантаження на перебіг системи ETL та зробило можливим збирання більшої кількості даних у початковому форматі.

Незважаючи на те, який поштовх в розвитку ETL дала поява Data Lake, це сховище здебільшого через власні переваги так само має й недоліки. Найбільшим і найголовнішим недоліком є відсутність прямого контролю за даними, які зберігаються. Таким чином, почало з'являтися негативне поняття data swamp (від англ. – болото даних) [2], яке стало символізувати погіршену якість даних в тих Data Lake, де скупчується все і без найменшого контролю.

Переваги обох сховищ вказували на незамінність кожного в сучасних і універсальних ETL-рішеннях, а недоліки вказували на те, що необхідно їх взаємовирішити. Внаслідок цього почали з'являтися двошарові архітектурнів рішення, які включали два сховища одночасно, але при цьому цілком відокремлено. Це хоч і універсалізувало ETL-систему, де Data Lake та Data Warehouse були джерлами для різної аудиторії в межах бізнесу, ускладнило систему в цілому, поставивши під питання цілісність даних між двома сховищами, та змусило дублювати великий обсяг даних двічі. Як наслідок, новим поняттям, яке було

введено, як найсучасніше рішення для сховища в ETL, став Data Lakehouse. Він має бути цілісним сховищем, який об’єднує найкраще з двох запропонованих попередників, при цьому не дублюючи збережені дані. Попри те, що Data Lakehouse ще досить молоде рішення, вже існують чітко визначені характеристики такого сховища. Вони полягають в тому, що DLH – це централізована система керування даними, яка надає строгість DW на базі неструкторованого DL. Таким чином, на одному цілісному сховищі будуть здійснюватися зручні й оптимізовані запити, що притаманні DW, а сховище буде таким самим дешевиш і гнучким як і в DL [2]. Попри загальну визначеність того, що за Data Lakehouse буде майбутнє сховищ, навколо яких будуватимуться сучасні системи обробки й взаємодії з даними, не припиняються зміни в архітектурі процесу ETL. Так зовсім недавно все частіше стали з’являтися згадки про настпуника ELT, а саме - EtLT, який, окрім сучасного сховища та зручної взаємодії, яку надає Data Lakehouse, включає в собі ще підпроцес tweaking (від англ. – налаштування, коригування) [12]. На цьому етапі мають відбуватися легкі перетворення, але при цьому життєво необхідні для якості даних, і зміни, які б зняли наванатження безпосердньо зі сховища DLH. Ця фаза може бути використана бізнесом для мінімального виокремлення потрібної інформації, фільтрації окремих типів даних та легкого форматування. Тож загалом цей додатковий етап покликаний допомогти уникнути ще більше проблем появи непотрібних даних, які знижують загальну якість сховища даних.

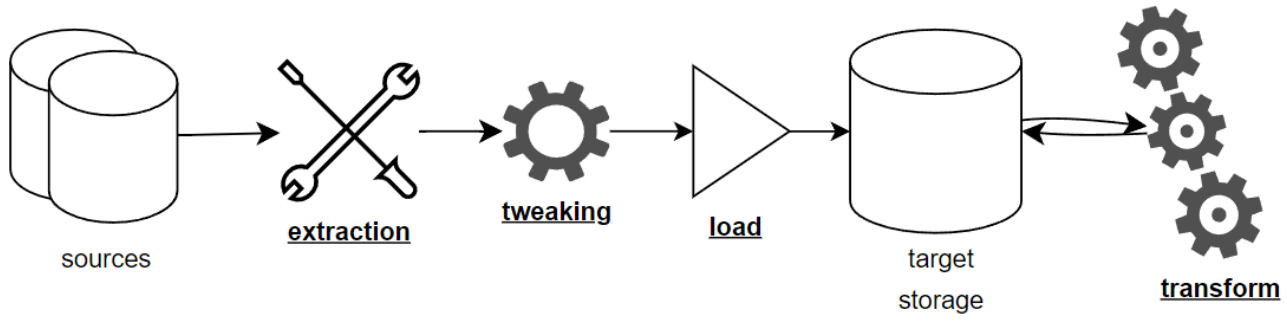


Рисунок 1.3 – Компоненти та етапи EtLT

Отже, розглянувши головні кроки в еволюції ETL-систем разом зі змінами й розвитком типів сховищ, які в них використовуються, можна виокремити найсучаснішу архітектуру EtLT разом із Data Lakehouse як достатньо універсальну й оптимальну. Вона вирішує більшість проблем, з якими можна стикнутися в сучасному світі, особливо коли мова йде про невпинне збільшення обсягу, різноманітності і швидкості появи нових даних, що є основними викликами Big Data. Незважаючи на це, кожна попередня архітектура може стати в пригоді для конкретних потреб. Наприклад, Data Warehouse і надалі може використовуватися як головне централізоване сховище в ETL системах, у якиї не потрібно забезпечувати обробку даних в реальному часі, особливо якщо вони не є надто різноманітними за своїм форматом. Так само й згадана архітектура ELT без допоможіних етапів разом з Data Lake може задовльнити потреби в галузях, де отримані з джерел дані є досить організованими, що не призведе до data swamp, про який згадувалося раніше, а користувачам цих збережених даних значно важливіша безпосередня швидкість готовності до взаємодії з ними у вихідному вигляді, аніж їхня структурованість.

1.4 Висновки до розділу 1

У цьому розділі було розглянуто першопричини появи поняття ETL як процесу повного циклу обробки даних, а також описані основні його складові, а саме - три етапи: видобування даних, їхнє перетворення та безпосереднє збереження. Також були згадані складнощі, з якими доводиться стикатися під час побудови таких систем, зокрема особлива увага була приділена до виклику пристосування до потреб обробки даних в наближеному до реального часі. Було здійснено аналіз еволюції процесу ETL, зокрема були розглянуті різні архітектури, які відірзняються різною послідовністю виконання згаданих етапів обробки даних. Також були розглянуті типи сховищ, що використовуються в тій чи іншій архітектурі процесу.

РОЗДІЛ 2. АНАЛІЗ ПРАКТИК В ЕТАПАХ ETL, ТЕОРЕТИЧНІ ВІДОМОСТІ ТА ВИБІР ТЕХНОЛОГІЙ

2.1 Аналіз пркатик у реалізації етапу видобування даних та супутніх технологій

Невпинний розвиток ETL призвів не тільки до зміни типів сховищ, але й до появи різних підходів, які використовуються для реалізації кожного з етапів процесу обробки даних.

Врахування існуючих підходів до організації етапу видобування даних є важливим кроком на шляху до побудови ефективного процесу ETL, адже саме він впливає на те, як ефективно та в якому вигляді будуть збиратися дані з джерел. Зараз існує кілька загальновизнаних методів для отримання даних, а саме – повне та інкрементальне видобування. Повне полягає в тому, що кожний раз, коли виконується перший етап ETL, дістаються абсолютно всі дані з відповідного джерела. Цей метод дозволяє уникнути потреби в узгодженні попердньо отриманої інформації з новими змінами, які відбулися і які слід відслідкувати. Це дозволяє уникнути проблеми загального порушення цілісності даних та їхньої можливої втрати в процесі виконання такого етапу. Інкрементальний метод навпаки полягає в тому, аби збирати лише зміни, що відбулися з моменту останнього видобування з джерела [13]. Таким чином, цей метод передбачає чітке узгодження кожного наступного процесу отримання даних з тим, що передував йому.

З цього можна зробити висновок, що перший метод є легшим та надійнішим в реалізації, але від цього він не стає найоптимальнішим на сьогодні рішенням. Повне видобування одразу провокує кілька проблем, з якими може стикнутися система ETL в ході виконання. Перша полягає в надмірному навантаженні на джерело, з якого здійснюється отримання даних. У залежності від періодичності повне копіювання може погіршити вже наступний процес видобування, але ще більшою проблемою може стати повна недоступність джерела як для системи ETL

та і для інших його клієнтів, які не будуть здатні вносити зміни в його сховищі. Тож може погіршитися загальний досвід роботи як для користувачів оброблених даних, так і тих, хто генерує першочергові дані для джерела. Друга не менш важлива проблема може вплинути на швидкодію етапів ETL як таких, адже для подальшої обробки необхідно в межах системи здійнювати порівняння новоотриманих даних з тими, що вже наявні в цільовому сховищі системи. У залежності від кількості і розмірів змін, цей процес може дуже сильно коливатися в часі, який буде витрачений на його успішне завершення, що вплине на наступні кроки в процесі.

Як наслідок, вбачаючи особливості інкрементального видобування, його популярність почала зростати. Більше того, лише він міг задовольнити потребу в обробці даних в реальному часі. До того ж, якщо у випадку повного видобування щось покращити було досить складно, особливо у випадку навантаження на джерело, то для інкрементального методу простір для покращення був значно ширший. Саме тому одночасно з появою поняття CDC почали розвиватися різні й технології, які б дозволяли якомога ефективніше й надійніше реалізувати такий підхід.

CDC має декілька різних імплементацій видобування інформації, що беруть за основу якийсь допоміжний аспект у схвоищі:

- На основі атрибуту часу зміни - це перша реалізація, що базується на додаванні спеціального атрибуту до кожної таблиці, отримання інформації з якої потрібно здійснити. Під цим атрибутом має зберігатися значення наближеного до точної дати та часу, коли рядок з’явився в таблиці або коли в ньому була здійснена будь-яка зміна. Незважаючи на те, що цей підхід до реалізації операції CDC є досить легким, він має достатню кількісь недоліків, яка ставить під питання його використання в сучасних системах. До найбільш вагомих проблем можна віднести неможливість відслідковування за рядками, які були видалені з таблиці. Для вирішення цієї проблеми слід залучати інші методи, які виходитимуть за межі зміни схеми даних у сховищі. Більше того, слід слідкувати за тим, що такий атрибут

обов’язково оновлюється в кожному рядку, адже інакше бути впевненим у працездатності такої реалізації буде важко. Цю проблему можна вирішити шляхом автоматизації встановлення нового значення в атрибуті, використовуючи, якщо доступні в базі даних, трігери (від англ. - провокатори). Саме вони б і забезпечили стовідсоткове оновлення значення для кожного рядка, який зазнає змін. Слід врахувати те, що використання цього інструменту сховища може значно вплинути на його працездатність, особливо якщо йдеться про велику кількість сутностей і змін у них, за якими слід постійно слідкувати [14]. Також використання такого підходу погіршує загальну самостійність і легкість компонентів ETL, адже для того, аби задовльнити такий обов'язковий етап як видобування, користувачам системи слід буде підлаштовувати всі свої сховища під таку реалізацію CDC. Це в принципі унеможливлює побудову ETL без втручання в схему та стан джерела, що може збільшити кількість проблем внаслідок людського фактору.

- На основі трігерів для допоміжних таблиць - це друга реалізація, яка полягає у використанні вже згаданого інструменту трігерів у сховищі разом із допоможними сутностями, які ще називають як shadow tables (від англ. – тіньові таблиці). Підходи до того, що саме зберігати в кожному рядку такої сутності залежить від виду копіювання, зокрема виділяють такі два:

1. Повне копіювання екзмпляру, що зазнав змін.
2. Копіювання лише ключа екземпляру та назви супутньої операції.

У тому випаку, якщо будуть зберігатися всі атрибути деякого екземпляру сутностей, за яким відбувається відслідковування, ми стикаємося з проблемою дублювання даних та загальним навантаженням на сховище знову ж таки через використання трігерів. Якщо ж скористатися другим видом, то ми стикаємося вже з іншою проблемою, яка стосуватиметься потреби в прямому звернені до таблиці для визначення змін, що відбулися. Це не тільки погіршує загальну швикодію першого етапу ETL, а й призводить до втрати розуміння проміжних змін екземпляру, якщо таких вже було кілька з моменту останнього видобування [14].

Більше того, будь-яка зміна в схемі даних сховища буде призводити до втрати даних без попереднього втручання в роботу трігерів та відповідної зміни атрибутів тіньової сустності. Це також неабияк впливає на універсальність ETL, адже, як і у випадку першої реалізації CDC, користувачам системи можливо доведеться втручатися в роботу сховища. Більше того, загальна особливість цієї імплементації не дуже підходить до систем, які матимуть забезпечувати обробку даних в реальному часі, адже занадто багато накладних витрат як у сенсі використання вільного місця, так і в швидкодії процесу.

- На основі журналу транзакцій сховища – це третя реалізація, що вже не вимагає втручання в схему чи стан даних сховища. Вона полягає в тому, аби користуватися існуючим в системі логуванням змін, які відбуваються у базі даних. Цим можливо скористатися в сховищах, що працюють на базі транзакцій, успішне виконання яких записується в спеціальних журнал у чіткій послідовності. Саме його вміст дозволяєм повністю відновити схему й стан сховища до останньої успішно виконанної операції [14]. Таким чином, під час роботи такої реалізації CDC відбувається слідкування безпосередньо за журналом, а не самим сховищем, що неабияк його розвантажує, оскільки ми не робимо прямих звернень до екземлярів сутностей. Незважаючи на це, імлементація такого варіанту видобування не була легкою. Найбільша проблема, з якою довелося стикнутися, не була пов’язана зі складнощами, що були притаманні попердньо згаданим реалізаціям CDC, а тому не стосувалися перенавантаження сховища або дублювання даних. Тут склідність полягала у пристосуванні до різних баз даних й їхніх відмінностей в тому, як саме вони здійснюються ведення журналу виокнаних операції. Саме цей аспект змушував перших розробників такого типу CDC витрачати багато часу на унікальну імплементацію для кожного конкретного сховища. Попри таку особливість, універсальність цього рішення затьмарила попередні імплементації, а тому недоліки не змогли вплинути на оптимальність такого рішення в сучасному світі. Справа в тому, що видобування даних на базі зчитування журналу операцій знову

ж таки дозволило впоратися з викликами, що стосувалися Big Data. Більше того, вміст журналу також включає зміни, що були здійснені на схемі сховища, що дозволить спостерігати за змінами в структурі його даних [15]. Тож завдяки цій реалізації CDC поява системи ETL, яка б могла забезпечити обробку даних в реальному часі, стала реальною перспективою, адже тепер перший етап видобування дозволив ефективно здійснювати відслідковування за всіма змінами в схвоищі без додаткових часових витрат та безпосереднього навантаження на сховища із забезпеченням цілісності даних, що гарантує журнал транзакцій.

Отже, третій розглянутий підхід до реалізації CDC, який в англомовній літературі носить назву Log-based CDC, став набувати все більшої популярності, внаслідок чого з’явилося багато готових інструментів, які забезпечували роботу з різними базами даних. До таких, що підтримують роботу з кількома сховищами і є популярними в спільноті належить багато розробок, зокрема й від великих компаній, а саме – Oracle Golden Gate та IBM IIDR. Такі рішення часто розпосюджуються лише за деяку плату і майже завжди мають закритий вихідних код, що дещо відштовхує бізнеси й розробників у цілому. Через це почали з’являтися інструменти з уже відкритим кодом, які дозовляли реалізувати те саме CDC, зокрема неабиякої популярності набув Debezium Kafka Connector в силу своєї легкості в налаштуванні та роботі з одним з найпопулярніших брокерів повідомлень Kafka [16].

2.2 Інструменти для сполучення Extraction з іншими етапами

Перш ніж остаточно спиратися на якийсь конкректний інструмент для етапу видобування в ETL, слід попередньо обрати ще один, який часто називають брокером повідомлень. Брокер повідомлень – це програмне забезпечення, що дозволяє різним системам обмінюватися інформацією одне з одним за певним стандартом [17]. Для успішної роботи брокер має мати такі три складові: producer (від англ. - виробник), що є створювачем повідомлень, queue (від англ. - черга) або ще називають topic (від англ. - тематика), що є сховищем повідомлень у порядку їхнього надхлдження, та consumer (від англ. - споживач), що є отримувачем повідомлень з черги. Тож брокер дозволить розділити в системі ETL його компоненти, а саме – зчитувач джерела даних та безпосерднього отримувача, які в свою чергу не матимуть підлаштовуватися одне під одного в разі зміни іншого. Окрім згаданого забезпечення самостійності компонентів, використання брокера впливає й на інші не менш важливі аспекти. По-перше, під час роботи брокер не вимагає наявності або активності компонентів з обох його сторін. Тож відсутність або помилка в роботі створювача повідомлень не впливає на можливість споживача їх отримувати і відповідно навпаки, що покращує стійкість системи до непередбачуваних проблем. По-друге, у разі помилки отримання поідомлення, але за умови, що споживач активний, брокер здатний здійснювати повторону відправку цьому споживачу того самого повідомлення одразу чи через деякий проміжок часу. Це підвищує впевненість у доставці повідомлень. По-третє, черга може відігравати роль повноцінного сховища, яке зберігатиме навіть уже спожиті повідомлення протягом певного проміжку часу. Таким чином, брокер повідомлень забезпечує загальну надійність збереження даних [18]. Більше того, екземплярів, які б змогли взяти на себе відповідальності брокеру повідомлення, може бути кілька, що дозволить зняти навантаження зі споживача або ж навпаки розподілити його, якщо отримувачів відповідно буде кілька. Як наслідок, використання такого брокера

здатне вирішити багато проблем із надійністю, масштабованістю і гнучкістю процесу, з якими б довелося стикатися розробникам ETL системи власноруч для успішної й ефективної роботи процесу у проміжку між видобуванням та зберіганням даних у цільовому місці.

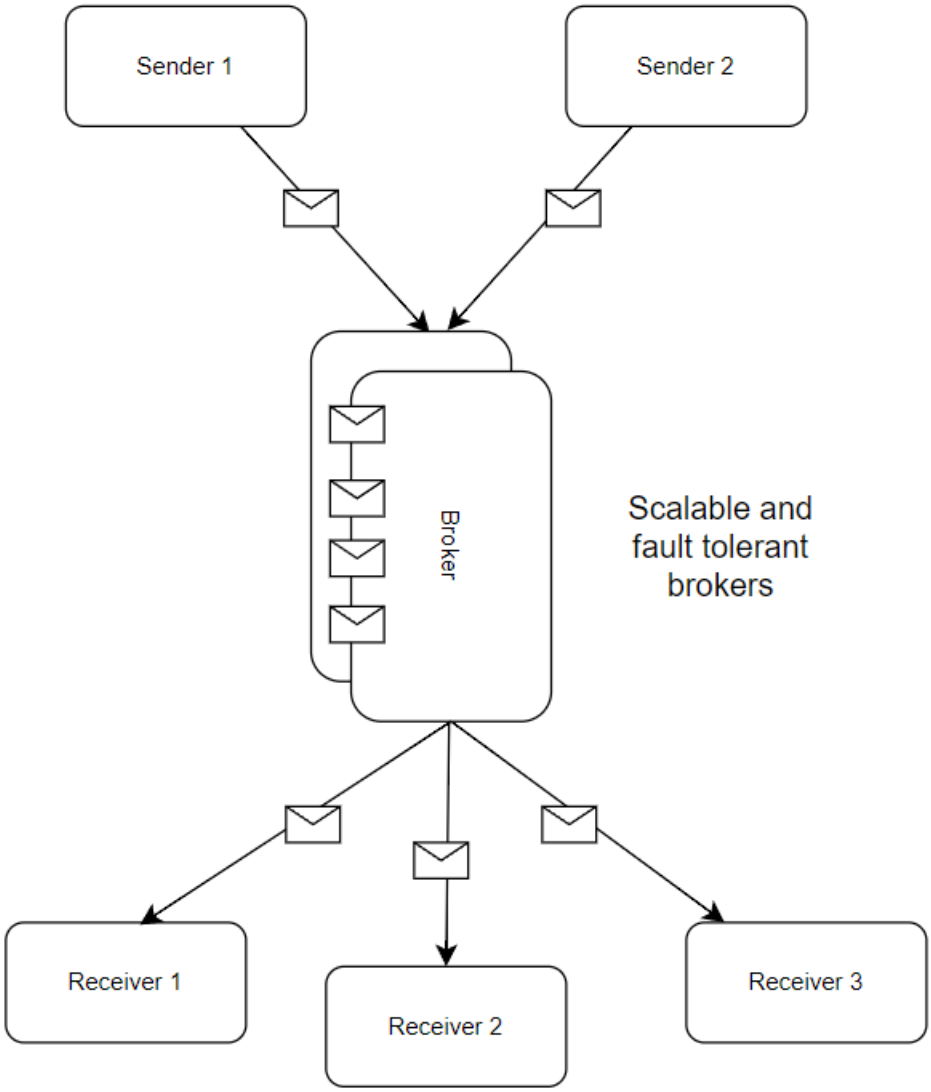


Рисунок 2.1 – Візуалізація роботи з брокерами повідомлень

Оскільки використання брокерів повідомлень стало досить поширеним в багатьох галузях, навіть за межами ETL, розвиток і збільшення конкретних реалізацій був очікуваним. Серед найпопулярніших виділяють такі рішення з

відкрити кодом, як RabbitMQ, Redis, Apache Kafka та імплементації на базі Active MQ, а також хмарні IBM MQ, Amazon SQS та Google Pub/Sub [19]. Незважаючи на різноманіття рішень, передусім цікавлять інструменти з відкритим кодом, які необмежують нас у використанні конкретного хмарного постачальника та дозволяють розгорнути себе локально за межами хмарних платформ. Тож передусім цікавлять можливості та загальна працездатність таких брокерів повідомлень.

Завдяки дослідженню, що включає в собі результати різних тестувань роботи таких брокерів, як RabbitMQ, Redis, Apache Kafka та Artemis на базі Active MQ, що були проведені в однакових умовах за рахунок використання однієї і тієї самої системи [20], можемо виокремити з ниї найоптимальніші рішення. Двома основними показниками, які використовувалися для аналізу роботи брокерів у згаденій роботі, були пропускна здатність та загальна затримка обробки повідомлень. Усі тестові повідомлення генерувалися за допомогою спеціального інструменту OpenMessaging Benchmark, який надає можливість тестувальнику налаштовувати різні параметри, зокрема кількість повідомлень в секунду та їхню вагу, що дозволило автору експериментів здійснити різні навантаження на брокери. Більше того, вбачаючи те, що для першого показника було здійснено 18 експериментів по 6 разів кожний для окремого брокера, а для перевірки другого – 15 експериментів по 5 разів кожний, можна бути впевненим у достовірності отриманих результатів.

Якщо мова йде про пропускну здатність, то найкраще себе показав брокер Apache Kafka, який здатний обробити близько 255 тисяч повідомлень в секунду, за умови, що вага такого повідомлення близько ста кілобайт. Під час експериментів з повідомленням меншої ваги результати були ще вищими, як можна побачити в таблиці нижче. Тож результат показника пропускої здатності Kafka для найбільшого повідомлення сильно відривається від того, на що здатні інші рішення, зокрема більше як у 14 разів у порівнняні з наступним переможцем в ролі RabbitMQ.

	10B	100B	1KB	10KB	50KB	100KB
Artemis	7652.99	8006.72	5872.98	1400.05	389.26	21.34
RabbitMQ	92,658	91,326	83,400	56,692	25,342	18,221
Redis	34,347	29,111	27,734	12,624	3298.67	2157.69
Kafka	666,517	599,794	427,361	333,145	298,590	255,285

Рисунок 2.2 – Порівняльна таблиця пропускної здатності брокерів [19]

У випадку показника затримки, найкращим виявився брокер Redis. Його особливість, що полягає в збереженні оброблюваних даних в оперативній пам’яті, на відміну від інших рішень, які використовують звичайне дискове сховище, відіграла тут найбільшу роль. Це дозволяє йому уникнути накладних витрат у сенсі швидкості доступу до оброблюваних даних, що і стало наслідком такого високого результату.

	1MBps	3MBps	6MBps	10MBps	15MBps
Artemis	44.0	87.0	111.0	225.0	3894.01
RabbitMQ	984.0	990.0	990.0	990.0	990.0
Redis	2.0	2.0	2.0	2.0	2.0
Kafka	5.0	5.0	6.0	6.0	6.0

Рисунок 2.3 – Порівняльна таблиця затримки в обробці повідомлень брокерами [19]

За порівняльною таблицею вище, можна побачити затримку в мілісекундах, яка була потрібна для того, аби обробити 99% всіх повідомлень з конкретною вагою. Таким чином, у брокера Redis тут відрив він попереднього переможця у ролі Kafka прблизно в три рази.

Незважаючи на друге місце Apache Kafka в останньому експерименті, його беззаперечна перевага у першому затьмарює цю поразку. Як наслідок, навіть автор дослідження виокремлює Kafka як найоптимальніше рішення серед усіх розглянутих, якщо для системи, в якій буде використовуватися цей брокер,

одночасно важливою є як висока пропускна здатність, так і досить низька затримка обробки.

Отже, повертаючись до вибору інструменту для здійснення етапу видобування в ETL за допомогою методу CDC, згаданий раніше Debezium також стане оптимальним рішенням для вибору через можливість його локального розгортання і прямою призначеністю для роботи з Kafka, адже побувадований на базі Kafka Connect і є його програмним розширенням.

2.3 Аналіз практик у побудові етапів петеворень та супутніх технологій

Після того, як вибір інструментів і технологій для видобування даних та брокеру повідомлень зроблено, слід забезпечити наступний в черзі виконання етап обробки даних у відповідності до сучасної архітектури під абревіатурою EtLT, яка розглядалася в першому розділі роботи. Другу фазу в цій архітерктурі, що носить назву tweaking, можна забезпечити використанням різного роду інстурментів, які здатні без зайвих часових витрати здійснювати різні опреації над повідомленнями. Часто для цього використовують інстременти для потокової обробки даних. Завдяки тому, що брокером повідомлень був обраний Apache Kafka, можемо скористатися супутніми технологіями з його широкої екосистеми для задоволення тако потреби.

Одним із перших інструментів, до яких варто звернутися, є безпоседньо Kafka Connect, який вже згадувався і на якому побудований Debezium, та доступні в ньому функції. Він надає можливість здійснювати так звані SMTs (Single Message Transforamtions), які здатні впливати на кожне окреме повідомлення. Процес такого перетворення можливий як для коннекторів (від англ. – під’єднувачів), які збирають інформацію з джерел для подальшої відправки до кластеру з брокерів повідомлень Kafka, так і для тих, що безпосередньо виводять дані з брокерів в напрямку цільового сховища або наступного етапу. У першому випадку вплив на

повідомлення здійснюється попередньо тому, коли вони передаються в чергу брокера, а в другому – попредньо вивантаженню з неї. Більше того, Kafka Connect розгортається як окремий сервер, що дозволяє розвантажити самі брокери й забезпечити внутрішню відмовостійкість та масштабованість завдяки режиму розподіленості, в якому конектор може складатися з кількох учасників, як займатимуться зчитуванням та трансформацією даних незалежно один від одного. Тож саме через ці можливості доречно використовувати конектори з обох вихідних сторін брокерів повідомлень, аби не брати на себе відповідлаьність за відмовостійкість роботи під'єднувачів як для видобування, так і завантаження даних. Як наслідок, питання того, який інструмент використовувати для етапу збереження даних, також відпадає, адже вибір Kafka Connect є оптимальним рішенням для успішного виконання обох раніше згаданих процесів по роботі з даними перед і після їхнього перебування в черзі брокеру.

У Kafka Connect реалізовані різні типи перетворень, якими можна одразу скористатися без власноручного написання функції. До доступних належать такі, як заміна, обнулення, приведення до конкретного типу полів у повідомленні, декомпресія з архівного формату gzip та переведення значень часових полів у різні формати. Окрім того, що перелічені функції не є повним списком з тих, що вже імплементовані в системі Kafka Connect, існує можливість власноручного створення функцій для більш специфічних перетворень [20]. Внаслідок того, що SMT працює з даними лише на рівні кожного повідомлення окремо, немає можливості здійснювати більш складні трансформації, зокрема групування повідомлень за деякими ознаками або їхнє сортування. Хоча такі операції можуть вже виходити за межі tweaking, групування даних часто необхідне навіть у тих ETL-системах, які вимагають мінімальної затримки в обробці повідомлень. Такі трансформації здатні збілшити користь від даних, що зберігатимуться в Data Lake, мінімзуючи їхню розпорошеність і надмірну кількість малих файлів.

Тож для забезпечення таких перетворень перед етапом вивантаження даних в цільове сховище, слід звернутися до інших інструментів, зокрема рекомендованими та одними з найпоширеніших для роботи саме з Kafka є такі потокові обробники, як Kafka Streams та ksqlDB, обидва з яких були створенні дотичними до Kafka розробниками [22]. Ці два обробники сильно відрізняються у тому, як вони дозволяють здіснюватися перетворення над даним. Kafka Streams дозволяє впливати на потоки повідомлень в брокерах за допомогою програмного втручання на базі таких JVM-мов як Java та Scala, тобто цей обробник виступає в ролі бібліотеки, яка побудована на базі роботи з інтерфейсом створювачів та споживачів Kafka. Виклик достпних функцій по роботі з потоками даних такої бібліотки дозволяє вливати на їхній вигляд в реальному часі. ksqlDB у свою чергу виступає спеціальною базою даних потокових подій, яка надає абстракційний рівень для взаємодії з поідомленнями, де можна безпосередньо здійснювати SQL-запити, не втручаючись у код [22]. У свою чергу також слід зазначити, що ksqlDB реалізована на базі Kafka Streams, а тому виступає більше високорівневим і готовим інструментом в порівнянні. Незважаючи на дочірність інструменту, він обмежений у можливостях роботи з даними внаслідок SQL-подібних запитів, а також має менший перелік форматів видобутих даних, з якими здатний працювати. Тому ksqlDB сприймає лише Avro, JSON та CSV, тоді як в Kafka Streams може бути реалізована робота з будь-яким форматом [23].

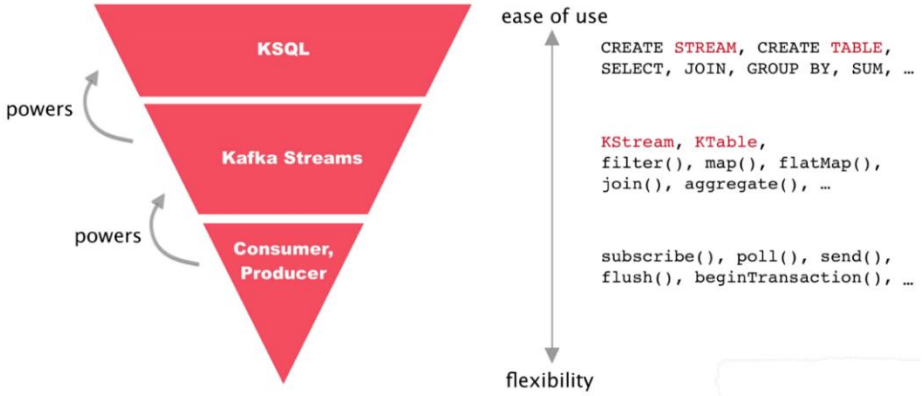


Рисунок 2.4 – Ієрархія залежності між Kafka Streams та ksqlDB

Вбачаючи їхні особливості й спорідненість, обидва рішення не є прямими конкурентами, а скоріше різними підходами до роботи з потоковими даними, де один є низькорівневим, а інший - високорівневим. Через це вони обидва утримують своє місце в обробці й широко використовуються для різних бізнес-потреб, адже вибір залежить саме від цього. Якщо необхідна легкість у роботі без додаткового програмування, то ksqlDB буде оптимальним рішенням, але якщо потрібні більш специфічні перетворення, які виходять за межі можливостей SQL-подібних запитів, то тут доведеться попрацювати з Kafka Streams і реалізувати власну програму. Тож залучення будь-якого зі згаданих обробників може бути доречним у межах ETL-системи, якщо необхідно здійснювати операції групування та сортування даних в наближеному до реального часу темпі одразу після їхнього видобування з джерел.

2.4 Огляд практик в організації збереження даних та сховищ в ETL

Наступним кроком після етапу вивантаження даних буде створення цільового сховища, яка відіграватиме ключову роль в наданні видобутих даних. За висновком у першому розділі, де розглядалися типи сховищ, навколо яких будуються ETL-системи, сучасним і універсальним рішенням є саме Data Lakehouse. Для його побудови передусів слід мати деяку основу, а саме сховище Data Lake, яке виступатиме кістяком всього DLH. Для відігравання ролі DL можна обрати різні сховища, але вони, як згадувалося раніше, мають надавати простір для даних різного типу, які будуть як структуровані, так і навпаки. Частіше за все в їхній ролі обирають об’єктні сховища на кшталт S3, який не має ієрархічної структури файлів й виступає сховищем плаского простору імен. Також досить розпосідженим є використання розподіленої файлової системи HDFS, яка навпаки є ієрархічною і складається з тек та супутніх файлів у них, які розташовуються по блоках в межах системи [24]. Вибір кожного із сховищ для основи Data Lake залежить від потреб і можливостей бізнесу, але не менш важливим є й підхід до збереження даних в них, зокрема йдеться про їхній формат та розподіл.

Формат файлів важливий через те, що їхнє збереження в оригінальному було б занадто витратним, особливо коли мова йде про великі обсяги. Більше того, початковий формат здатний негативно вплинути на швидкодію роботи з файлами. Через це з’явилися так звані формати для Bid Data, до яких передусім відносять Parquet, ORC та Avro. Саме ці формати дозволяють зменшувати вагу кожного оригінального файлу шляхом його стискання. Більше того, ці формати дозволяють розбивати один файл між кількома дисками або середовищами сховища, що дозволяє паралельно його обробляти, що неабияк пришвидшує запити до сховища та роботу з даними [25]. Тож використання таких форматів є важливим кроком на шляху до імплементації ефективного і швидкого сховища, яке б змогло впоратися з викликами Big Data. Незважаючи на спільні якості, згадані формати мають і вагомі відмінності.

Наприклад, Parquet та ORC передбачають збереження даних в стовпчиковому вигляді, що дозволяє уникнути зчитування всієї інформації файлу й звертатися лище до тих полів, які необхідні. Окрім того, що така особливість зменшує час затрачений на запити, такі формати передбачають сильніше стиснення інформації, що позитивно впливає на використання доступного місця в сховищі. Незважаючи на ці переваги, саме ці формати не пристосовані до того, щоб втручатися в структуру їхніх файлів, а тому не є найоптимальнішими рішеннями у роботі з даними, які потребуватимуть змін у майбутньому. Саме тут на допомогу прийде формат, який записує дані в рядковому вигляді, а саме – Avro. Він використовує JSON як основу для опису даних у файлі, який в свою чергу особливим чином бінаризований, чим і досягається значне зменшення ваги [25].

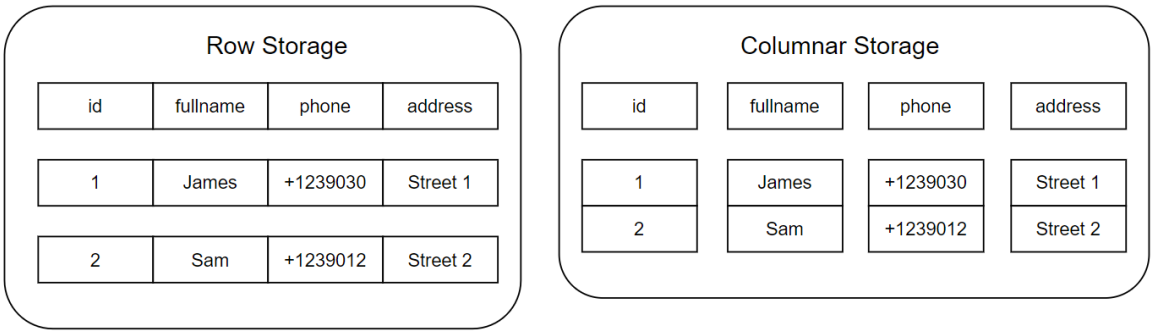


Рисунок 2.5 – порівняння збереження даних в різних типах форматів

Avro має недоліки, які випливають з його особливостей, зокрема він передбачає менші можливості для стиснення ваги файлів у порівнянні з іншими форматіами і не дозволяє ваємодіяти з окремими полями, але при цьому є більш гнучкою альтернативою зі змогою швидко записувати файли внаслідок своєї рядковості. Незважаючи на неможливість зчитування окремих стовпчиків, цей формат може бути досить доречним для використання у випадках, коли потрібен і використовується майже весь вміст збережних файлів. Тож такий формати може стати в пригоді для специфічних бізнес-потреб, де швидкість вибіркового зчитування і кінцевий розмір файлу не настільки важливі, як гнучкість та запис.

Окрім вибору формату, в Data Lake слід передбачити можливість розподілу збережених файлів за деякими показниками. Це необхідно для того, аби зробити сховище легшим у користуванні й підлаштувати його під вимоги поноцінного Data Lakehouse, в якому буде передбачена пряма робота з даними, які вже будуть корисні аналітикам та бізнесу в цілому. Через те, що Data Lakehouse буде єдиним сховищем в системі, слід забезпечити розмежування так званих початкових даних, які отримують напряму з брокерів повідомлень, та інших вже більш опрацьованих. Внаслідок такої потреби з’явився новий підхід до організації сховища під назвою Medallion Architecture, який ще згадується як multi-hop. Він полягає в логічному розподілі даних з метою поступового покращення їхньої якості та забезпечення різних цільових потреб. Слід зазначити, що часто така архітектура передбачає

наявність трьох окремих прошарків даних, які називаються у відповідності до рівня підготоволеності даних, а саме - бронзовий, срібний та золотий [26].

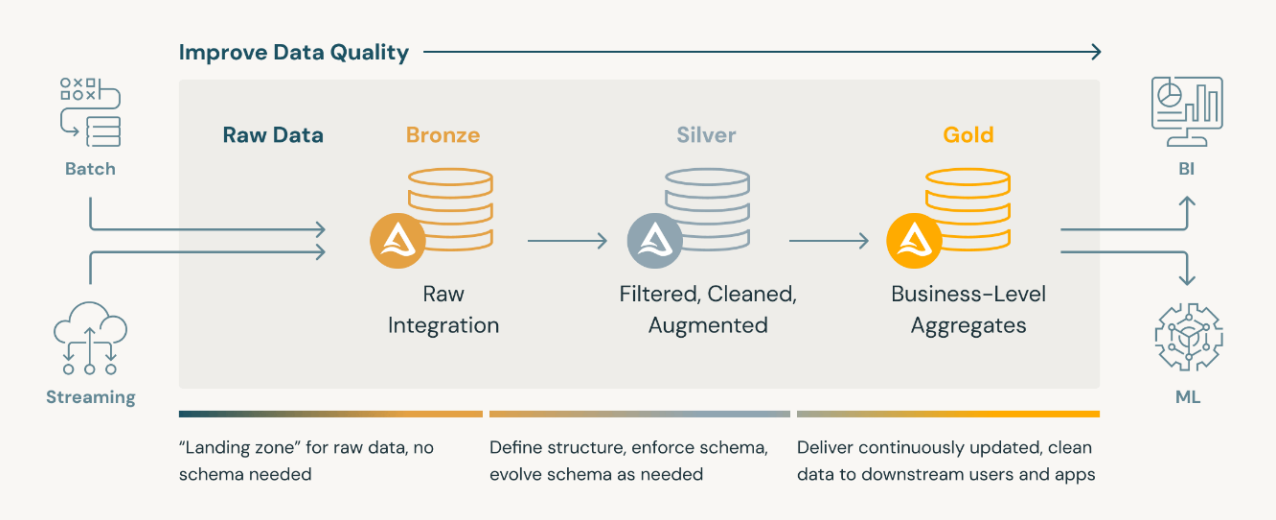


Рисунок 2.6 – Зображення компонентів процесу в побудові Medallion Architecture [26]

Найперший зі згаданих прошарків зберігає в собі ті файли, які були отримані одразу після завантаження в Data Lake, тобто це дані, які не втратили свого початкового вигляду з джерела. Ці дані можуть бути певною мірою профільтровані або доповнені, зокрема можуть бути додані специфічні поля, які б відігравали роль метаданих, але без серйозних втручань, адже це вагомий аспект у їхній корисності для системи. Тож основною метою цього прошарку є надання якомога більш швидкого доступу до видобутих даних, які важливі своєю актуальністю і повнотою, які можуть бути використані для швидкого прийняття рішень та перевикористання. Перевикористовуватися такі дані можуть задля доповнення даних на срібному та золотому рівнях або для їхнього відновлення до деякого попереднього вигляду. Більше того, цей прошарок часто є джерелом для історії оброблених даних, внаслідок чого його ще часто назвивають рівнем для Cold storage (від англ. – холодного сховища) [26]. Тож можна сказати, що бронзовий рівень – це внутрішнє для Data Lakehouse джерело, до якого можна звернутися в разі потреби отримання мінімально перетворених видобутих даних.

Другий прошарок виступає місцем для збереження даних, які пройшли вже досить вагомі кроки на шляху до надання вибраної для бізнесу і якісної для його потреб інформації. Тож перед появою на срібному рівні, найчастіше здійснюється різного роду фільтрація та доопрацювання з бронзового прошарку. Таким чином на бронзовому прошарку запускаються автоматизовні процеси з обробки даних, а успішно отриманий результат вже вивдиться на срібний. У цьому процесі обробки передусім застосовується об’єднання незалежних між собою даних, виконується приведення до конкретних форматів і стандартизованих значень, а також виконуються операції з виправлень й видалень пошкоджених або повторюваних даних. Часто срібний рівень виступає сховищем, яким уже здатні користуватися різні фахівці в межах бізнесу, не боючись за рівень їхньої якості, адже дані в ньому вже мають належний вигляд для задоволення окремих потреб розробників та аналітиків [27].

Фінальний прошарок складається з даних, які вже сильно відрізняються від того, що було отримано бронзовим рівнем. Тут знаходятся дані, які були організовані таким чином, щоб задовольняти специфічні потреби бізнесу, що не дозволяє використовувати їх для різних цілей як це було у випадку зі срібним. Таким чином тут дані настільки денормалізовані шляхом різних об’єднань та доповнень, які в свою чергу можуть здійснюватися як автоматично, так і за допомогою прямого втручання, що вони стають корисні лише для конкретної задачі ведення звітів, які б було легко сприймати бізнесу [26]. Саме через свою опрацьованість і специфічність цей прошарок носить назву золотий, адже він несе найвищу цінність для бізнесу.

Отже, з огляду на логічність розподілу даних, Medallion Architecture може стати в нагоді для організації центрального сховища в ETL-системі, дані в якій можуть бути корисними для різної аудиторії в межах одного бізнесу. Ця архітктура дозволить мінімізувати витрати на утримання кількох різних сховищ для

розмежування потреб та загалом зазбезпечити ефективне поєдання EtLT з Data Lakehouse.

Для того, щоб реалізувати архітектуру Medallion та загалом надати сховищу Data Lake якості, які притаманні Data Warehouse, почали з’влятися спеціалізовні інструменти, а саме – Apache Hudi, Delta Lake та Apache Iceberg. Вони були створенні для забезпечення відповідності принципам ACID та можливість ствоерння SQL-подібних запитів на великих обсягах структурованих і неструкторованих даних. Тож їхнього головною метою є перетворення звичайного Data Lake у Data Lakehouse [28]. Усі загдані інструменти переслідують схожу мету і для роботи кожного з них дані в сховищі мають бути приведені до відповідного табличного формату. Робота кожного з них також базується на використанні раніше згаданого BigData-формату Parquet для оптимізації роботи з великими обсягами даних [28], але при цьому вони мають і відмінності, зокрема у підтримуваних двигунах запитів, зокрема таких як Apache Spark, Apache Hive та Presto [29], з якими вони здатні працювати, та різних підходах до оптимізації запитів до таблиць. Тож перш ніж обрати кокретний інстрмент для побудови Data Lakehouse, слід проаналізувати загальну ефективність у їхній роботі та можливості підтримуваних двигунів для запитів.

Робота двигунів для запитів полягає у тому, аби надати якомога наближені можливості SQL-операцій над даними, але при цьому відокремити середовища виконання цих запитів від безпосереднього місця збереження. Це дозволяє у випадку роботи з великими даними розвантажити їхнє сховище та надати можливість виконуваним запитам масштабуватися. Процес виконання запитів починається з моменту їхнього надсилання деякій керівній програмі, яка у свою чергу розподіляє виконання цього запиту на різних підпорядкованих програмах-працівниках [29]. Часто усі учасники процесу розміщенні на окремих машинах для забезпечення розподіленості системи двигуна та забезпечення паралельного виконання на працівниках. Саме завдяки цим двигунам і будуть виконуватися

запити, які перетворюватимуть дані у потрібний для Data Lakehouse інстурментів табличний формат, який потім дозволить звертатися до даних таблиць SQL-подібним чином. У свою чергу відповідні таблиці, вміст яких був досягнутий шляхом певних перетворень, будуть розташовані на окремих прошарках архітектури Medallion. Двиуни Apache Spark, Apache Hive і Presto демонструють схожі показники під час роботи з даними, вімінності полягаються в тому, що вони різняться в тому, як вони працюють запитами в яких задіюється різний обсяг даних. Так, наприклад автор досліджень ефективності згаданих інстурментів виділяє гарну роботу Apache Spark для запитів, виконання яких значною мірою залежать розрахункових потужностей процесора, тоді як Apache Hive гарне підійде для запитів, які значною мірою залежать від обсягу вхідних даних, а Presto оптимальний для малих запитів [30]. Тож використання кожного залежить від конкретної галузів та потреб.

Тоді як вибір двигуна завершено, слід розглянути ефективність роботи Data Lakehouse інстурментів. Проведений в наближених до однакових умов TPC-DS експеримент для Delta Lake, Apache Hudi та Apache Iceberg, який полягає у оцінці працездатності сховищ для BigData на базі набору різних запитів, що впливають на його стан [31], здатний продемонструвати відмінність у ефективності операцій зчитування, запису та інших. Тож за таблицею видно, що показники Delta Lake та Apache Hudi досить наближені, тоді як Apache Iceberg значною мірою відстає. Як наслідок, хоч ефективність і не є єдиною метрикою для обрання інструменту, але перевага перших двох є досить вагомою для того, аби здійснити вибір на їхню користь. Вбаючи популярність та підтримуваність зі сторони розробників, що має Data Lake [32], його інтегрованість з двигуном Apache Spark [28] та легший в налаштуванні процес перетворення даних [33], можна сказати, що він буде оптимальним для задоволення потреб у побудові і організації сховища Data Lakehouse.

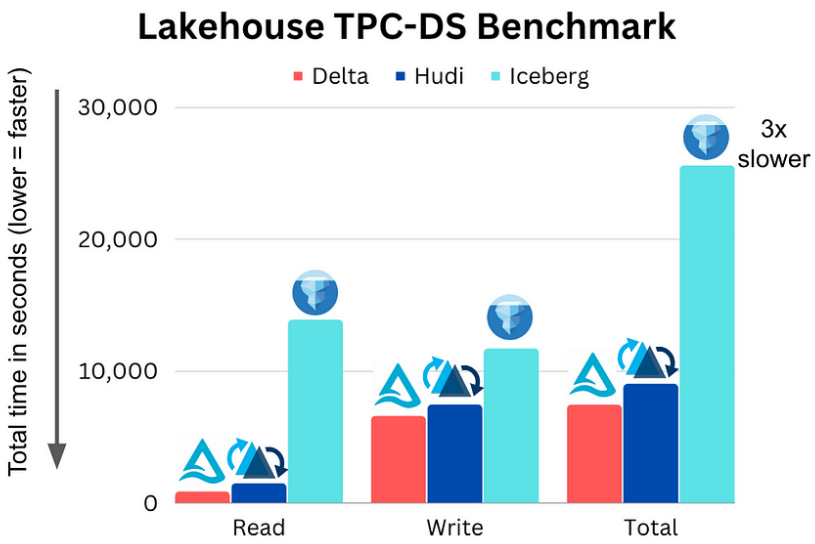


Рисунок 2.7 – Порівняння швидкодії інструментів для організації Data Lakehouse [33]

2.4 Розгляд технологій для керування та розгортання ETL системи

Тепер коли набір технологій та підходів для реалізації компонентів та етапів системи ETL розглянуто і вибір кожної став зрозумілим, слід звернутися до інструменту, який дозволить нам їх усіх розгорнути як локально, так і в хмарному середовищі. Для того, щоб розгортання системи було уніфікованим, а його компоненти напряму не впливали на середовище операційної системи, скористаємося контейнеризацією компонентів за допомогою Docker. Контейнеризація вже досить тривалий час є стандартом для розгортання як власного написаного коду, так і готових застосунків. Суть контейнеру полягає в тому, що це спеціальна програмна обгортка для іншого ПЗ, яка повністю відокремлює його виконання, конфігурацію та сховище від системи кінцевого користувача, але при цьому без залучення гостьової операційної системи у власну структуру як це відбувається у віртуальних машиах. Таким чином, контейнеризація вирішує одразу кілька проблем під час розробки та запуску системи з компонентів, а саме – загальне покращення безпеки шляхом ізоляції виконуваних в контейнерах програм, збільшення ефективності у використанні ресурсів в порівнняні з віртуальними машинами, а також загальна гнучкість у розгортанні в різних середовищах і підтримці їхніх оновлень [34].

З поступовим збільшенням кількості контейнерів, які відповідають за виконання певних сервісів у системі, слід якось організовувати та автоматизувати цикл їхнього життя. Інструменти, які використовуються для цього, називають оркестраторами контейнерів. До найпопулярніших представників такого типу інстрементів відносять Kubernetes, Docker Swarm та Apache Mesos [35]. Оркестрація в першу чергу допомагає керувати версіями сервісів контейнерів, масштабувати їхню кількість та налагоджувати мережеву комунікацію між собою. Більшість оркестраторів, зокрема й згадані, працюють на базі заздалегідь визначеної конфігурації, в якій описані всі необхідні елементи для успішного розгортання

сервісів у контейнерах. На інструменти оркестрації також часто може бути покладена така відповідальність, як відновлення контейнерів у разі відмови, балансування вхідного навантаження, а також масштабування запущених сервісів на доступних машинах [36]. Слід зазначати, що хоч всі згадані інструменти мають як переваги, так і недоліки, більшість IT-спільноти все ж визнає Kubernetes як найбільш повний і гнучкий у налаштуванні оркестратор, який має функціонал для вирішення більшості проблем. Через це він уже тривалий час є виробничим стандартом у великих системах, коли підтримуваність різних типів контейнерів, автоматизована відомовостійкість та масштабування є важливими аспектами в роботі. Незважаючи на явну перевагу Kubernetes, Docker Swarm теж лишається стандартом для розгортання невеликих систем, де кількість сервісів не виходить за межі кількох десятків. Більше того, багато хто також визнає, що Docker Swarm є одним з найшвдших у роботі оркестраторів, який не вимагає прискіпливої конфігурації [37].

Отже, враховуючи те, що Kubernetes значно повільніше розгортає нові контейнери, ніж Docker Swarm, швидкість якого інколи в 5 разів вища у відповідності до продемонстрованих результатів у дослідженні [38], а використання ресурсів при цьому значно нижче [39], оркестратор від Docker може стати гарним вибором для використання в системі, яка має обмежену кількість сервісів і які мають бути швидко відновлені в разі відмови. Більше того, сумісність конфігурації Docker Compose третьої версії, яка використовується для локального запуску набору контейнерів, з можливістю розгорнутися в середовищі оркестратора Swarm неабияк привертає увагу [40], адже одна конфігурація з мінімальним втручанням у деякі налаштування буде єдиним джерелом для обох розгортань.

Також слід зазначити, що система Swarm складається зі значно меншої кількості компонентів у порівнні з Kubernetes та Mesos, що робить розуміння його роботи більш прозорим та гнучким. Основними учасникам Swarm є відкоремлені вузли, які відповідають окремим машинам відповідно, кожен з цих вузлів може

мати якусь з двох ролей або дві одночасно, а саме – менеджера та працівника. Наявність менеджерів є невід'ємною частиною системи оркестратора, адже без них не вдасться призначати завдання та ребалансувати їх. Через це бажано мати кілька менеджерів, зокрема непарну кількість, аби забезпечити максмимальне значення толерованих відмов серед них. Причина полягає в тому, що вони працюють за принципом консенсуного алгоритму Raft, а тому для забезпечення цілісності стану системи більше 50% від усіх менеджерних вузлів повинні мати однакове бачення стану системи до моменту недоступності головного менеджера [41].

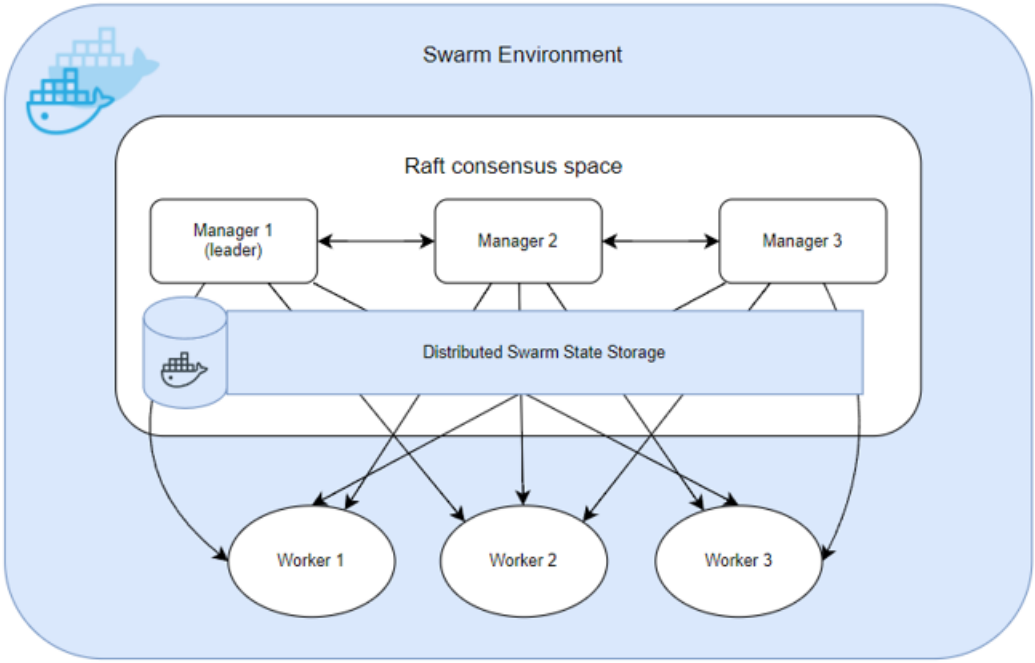


Рисунок 2.8 – Docker Swarm середовище та його складові

Тож саме менеджери в Swarm відповідають за успішність оркестрування як такого і тому займаються делегуванням сервісів за окремими показниками відповідним працівникам, які мають їх виконувати. Тож, наприклад, якщо якийсь працівник із певних причини буде недоступний, то оркестратор зніме з нього завдання розгортання контейнеру й передасть іншому доступному в межах системи Swarm [42].

2.5 Висновки до розділу 2

У даному розділі були розглянуті підходи до реалізації кожного компоненту системи обробки даних ETL у відповідності до одного з найсучасніших типів архітектури, що був згаданий у першому розділі, а саме – EtLT. Були проаналізовані існуючі технології, які використовуються для забезпечення кожного з етапів та підетапів, а також зроблено обгрунтований вибір на користь окремих рішень у відповідності до можливостей та порівняльних результатів з конкурентами. Розглянуті стратегії з організації централізованого сховища даних на базі сучасних принципів Date Lakehouse та архітектури Medallion. Були наведені популярні технології для оркестрації контейнеризованих сервісів, розглянуті принципи їхньої роботи та зроблений вибір на користь найоптимальнішого у відповідності до вимог і потреб системи.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ФРЕЙМВОРКУ ДЛЯ МАСШТАБОВАНОЇ ETL-СИСТЕМИ

3.1 Аналіз технічного завдання

Розробка фреймворку полягає в тому, аби надати каркас для побудови деякої системи, що враховуватиме певні принципи, підходи та структуру [43]. Тож з огляду на обрані практики та технології для побудови процесу інтеграції даних, можемо перейти до реалізації фреймворку для забезпечення масштабованої ETL-системи. Цей фреймворк надаватиме каркас, який буде скеровувати користувача для успішної побудови перебігу процесу обробки даних у відповідності до архітектури EtLT. У свою чергу, на кінцевого користувача буде покладено лише мінамільну відповідальність, яка полягатиме у підключенні окремих компонентів, зокрема джерел даних та цільового сховища, визначення запитів для перетворення даних, а також вказання деяких параметрів для масштабування за потреби. При цьому вся структура системи, вибір інструментів, які забезпечуватимуть виконання етапів інтеграції даних, моніторинг роботи сервісів системи та здійснення їхнього масштабування у відповідності до навантаження буде покладено безпосередньо на розроблений фреймворк. Таким чином, він покликаний полегшити побудову й розгортання комплексної системи ETL, де для функціонування компонентів не потрібно буде втручатися в її код та конфігурацію основних компонентів.

Для надання якісного досвіду користування фреймворком буде надано два інтерфейси для взаємодії, а саме REST API та графічний інтерфейс, де перший може стати в нагоді для наглядного розуміння особливостей роботи каркасу, а другий – для зручного візуального сприйняття та керування. Таким чином, для повноцінної роботи фреймворку, слід реалзіувати такі три основні компоненти:

1. Систему масштабованих сервісів, які забезпечуватимуть повний цикл обробки даних в аріхтектурі EtLT, яка буде розгорнута у середовищі оркестратора Docker Swarm.

2. Групу мікросервісів, які відповідатимуть за виконання завдань по роботі з сервісами в Swarm та надання REST API.
3. Веб-платформу, яка відповідатиме за надання графічного інтерфейсу користувача на базі використання REST API.

Основна взаємодія між комонентами буде реалізована за допомогою HTTP запитів, а взаємодія між мікросервісами, які відповідатимуть за серверні завдання, буде реалізована на базі згаданої раніше черги повідомлень Redis, яка має гарні показники у загальній швидкодії, зокрема в сенсі потрібних ресурсів, пропускній здатності та затримці. Використання саме мікросервісної архітектури передусім обумовлено тим, що це дозволить системі сервісів-учасників більш гнучко масштабуватися, забезпечити загальну модульність системі, де буде можливість легко замінити компонент за потреби, а ще зменшити залежність від самої реалізації кожного сервісу, адже взаємодія між ними відбуватиметься за деяким взаємопідтримуваним інтерфейсом [44], яким у нашому випадку виступатиме Redis. Слід також зазначити, що мікросервіси будуть написані за допомогою фреймворка для Node.js, а саме Nest.js, який має зручну струкутру визначення взаємодії для мікросервісів і реалізації REST API. Тож тепер, коли зрозуімла основна структура системи та завдання компонентів, перейдемо до безпосереднього огляду реалізації кожного з них.

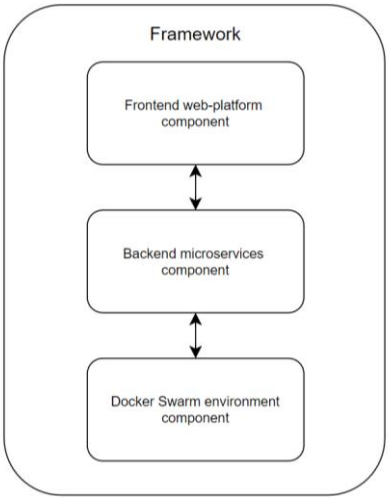


Рисунок 3.1 – Головні компоненти, які забезпечують роботу фреймворку

3.2 Компонента середовища Docker Swarm та супутні сервіси

Основою компоненту для розгортання сервісів у Docker Swarm є безпосередньо його вузли-учасники. Оскільки Swarm передбачає розгортання на окремих фізичних або віртуальних машинах, нам слід передусім забезпечити локальне розгортання системи працівників та менеджерів в оркестраторі, яке б дозволило користуватися учасниками навіть на одній фізичній машині. Це можливо досягнути різними способами, зокрема підняттям віртуальних машині на базі таких існуючих та популярних платформ для віртуалізації, як VMware або VirtualBox [45], але такий підхід передбачатиме багато втручань зі сторони користувача та залучення значної кількості ресурсів його операційної системи, а тому тут також буде оптимальніше скористатися саме контейнеризацією. Основним викликом такої задачі буде те, що слід у відповідних контейнерах розгорнути двигун контейнеризації, наприклад, Docker або Podman, аби в них також можна було піднімати контейнеризовані сервіси. Через зростання популярності конейтеризації з часом саме в екосистемі Docker з’явилося поняття вкладеної контейнеризації, яка досягалася шляхом використання спеціального образу DinD (Docker Inside Docker) для контейнеру. Такий контейнер здатний розгортати інші контейнери в середині свого середовища [46], де б він мав повноцінний двигун для надання всіх тих самих можливостей Docker, що й на повноцінній операційній системі. Використання цього рішення б дозволило заощадити значну кількість ресурсів і так само автоматизувати підняття всієї системи учасників у Docker Swarm аналогічно тому, як це робиться з усіма іншими контейнерами. Тож для реалізації такого рішення користувачу слід мати лише встановлений Docker на операційній системі його машини та доступ до мережі Інтернет, аби була можливість завантажити необхідні образи для контейнерів.

Таким чином для підняття працівників та менеджерів скористаємося звичною для Docker конфігурацією, а саме - docker-compose в форматі yaml, де ми зможемо

декларативно описати бажаний набір контейнеризованих сервісів, які в нашому випадку будуть учасниками системи Swarm. У відповідності до модифікованої до більш сучасних реалій конфігурації [47], зможемо підняти одразу всіх потрібних учасників. Для цього нам потрібно створити привілейованого менеджера за допомогою допоміжного контейнеру й проініціалізувати середовище Swarm, під час цього будуть збережені у спільному для всіх контейнерів сховищі необхідні ключі підключення, що дозволять їм під'єднуватися до середовища оркестратора.

```
volumes:
  swarm:

services:
  manager:
    container_name: manager-1
    image: docker:18-dind
    networks:
      - swarm
    ports:
      - "2375:2375/tcp"
      - "2377:2377/tcp"
    privileged: true
    volumes:
      - "/lib/modules:/lib/modules:ro"

  init:
    command: sh -e -c '(docker swarm init || true) &&
     docker swarm join-token manager --quiet > /swarm/manager-token &&
     docker swarm join-token worker --quiet > /swarm/worker-token'
    container_name: swarm-manager-1-init
    environment:
      DOCKER_HOST: "tcp://docker:2375"
    image: docker:18
    links:
      - manager:docker
    networks:
      - swarm
    volumes:
      - "swarm:/swarm"
```

Рисунок 3.2 – Конфігурація для підняття локального лідера Docker Swarm

Після того, як контейнер головного менеджера буде успішно піднято, до нього можна буде приєднати інших учасників, які зможуть мати будь-яку з двох ролей, яка визначатиметься тільки обраним ключем під час підключення.

```
entrypoint: ["sh", "-c", "
  sleep 2;
  dockerd-entrypoint.sh &
  while ! docker info > /dev/null 2>&1; do
    sleep 1;
  done;
  docker swarm join --token $$(cat /swarm/manager-token) manager1:2377 &&
  tail -f /dev/null"]
```

Рисунок 3.3 – Команда для під'єднання в ролі менеджера

Тож згідно з практиками відмовостійкості в алгоритмі Raft, який згадувався в попредньому розділі, маємо створити якомога більше менеджерів, сума з яких дорівнюватиме непарному числу, аби у працюючих менеджерів було більше шансів досягти консенсусу та продовжувати успішно керувати середовищем Swarm навіть якщо якісь з них стануть недоступними. Часто пропонують всіх учасників робити менеджерами, аби кожен міг перебрати на себе роль головного оркестратора і, як наслідок, підвищити відмовостійкість, але через це з'являються й інші проблеми, зокрема збільшується використання ресурсів на відміну від звичайних працівників і знижується рівень безпечності системи, адже втручання в роботу будь-якого учасника здатне надати доступ до даних всього Swarm.

Оскільки треба врахувати ці аспекти та перспективу заощадження коштів у межах розгортання в хмарі, важко запрпонувати ідеальний набір вузлів, але за більш менш сприятливих умов бажано підняти щонайменше 5 менеджерів. Один з них, а саме – лідер, буде або drained, або ж призначений лише для певних моніторингових завдань, тобто не прийматиме участь у роботі головних сервісів, що дозволить мінімізувати вплив на роботу оркестрації. Що ж стосовно працівників, то їх можна підняти деяку парну кількість, наприклад, 2. Така кількість зумовлена тим, що ми матимемо 6 повноцінних учасників, де буде чотири приймаючих менеджера та два звичайних вузли, які будуть здатні виконувати роботу сервісів і забезпечувати відмовостійкість однакового рівня для всіх основних процесів у системі EtLT. Тож по два учасники буде залучено для Kafka Connect, тобто для видобування та вивантаження, а також для роботи брокерів Kafka та етапу перетворень для забезпечення сховища Data Lakhouse. Таким чином, ми будемо здатні толерувати 2 недоступних менеджери й забезпечувати мінімальну відмовостійкість для виконання кожного зі згаданих процесів у системі, тобто толерувати по одній відмові вузла-працівника.

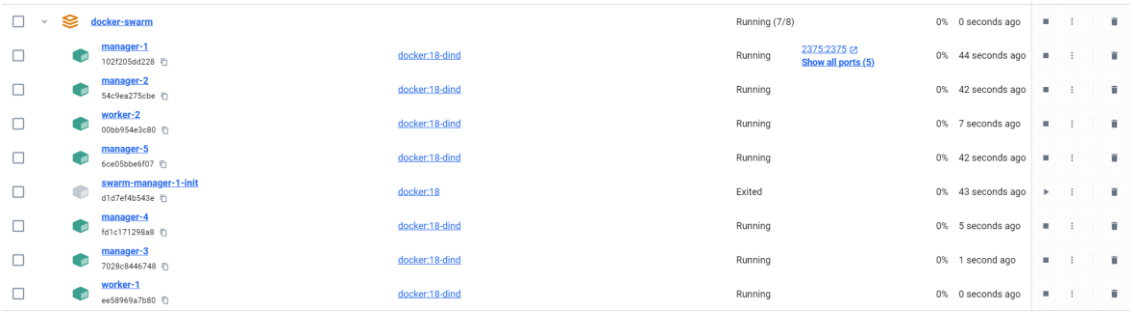


Рисунок 3.4 – Учасники Docker Swarm, які були розгорнуті шляхом DinD

Незважаючи на можливу сталість числа учасників у Swarm, це не свідчить про незмінність кількості контейнерів на кожному з них, адже масштабування Docker Swarm передусім полягає в зміні кількості контейнерів, що виконуть сервіси і не призначене додавати нових учасників. Тож у даному випадку масштабування в межах оркестраторів є обмеженим не стільки кількістю доступних вузлів, скільки наявними в них ресурсами, адже кожен учасник середовища здатний виконувати більше одного сервісу і відповідно мати різну кількість запущених в собі контейнерів. Про безпосередній алгоритм масштабування сервісів та його організації у середовищі Swarm буде згадано пізніше, коли розглядатиметься компонент мікросервісів, де окермі з них і займатимуться цією задачею.

Після того, як ми вже маємо розгорнуте середовище оркестратора Swarm, а його вузли здатні виконувати функції учасників, слід перейти до визначення конфігурації сервісів, що безпосередньо забезпечуватимуть виконання повного цилку обробки даних в межах архітектури EtLT. Тож перерахуємо основні сервіси системи, враховуючи як основні, так і допоміжні. Блок основних сервісів, які є обов’язковими для базової роботи фреймворку для обробки даних, складається з таких:

- source-connector – відовідає за видобування даних з джерел за допомогою інструменту Debezium на базі Kafka Connect із залученням CDC, використаний Docker-образ – debezium/connect.

- kafka-broker – відповідає самодостаньому брокеру повідомлень, який працює без Apache Zookeeper і яких може бути кілька, на базі Apache Kafka, який є сховищем повідомлень і сполучною ланкою між видобуванням та вивантаженням, використаний Docker-образ – debezium/kafka.
- sink-connector – відповідає за вивантаження даних на базі Kafka Connect з брокеру повідомлень в цільові сховища, які відіграватимуть роль основи у Data Lakehouse, використаний Docker-образ – confluentinc/cp-kafka
- schema-registry – відповідає за взаєсуміність схеми даних, що створюється в source-connector з тим, що отримується в sink-connector, використаний Docker-образ – confluentinc/cp-schema-registry.
- spark-master – відповідає за делегування виконання запиту працівниками, що надійшов до двигуна Apache Spark для перетворення даних в цільовому сховищі Data Lakehouse, використаний Docker-образ – bitnami/spark.
- spark-worker – відповідає працівнику, яких може бути кілька, в межах системи виконання запитів Apache Spark, , використаний Docker-образ той самий, що і spark-master.
- cadvisor – відповідає за відслідковування роботи та ресурсів учасників Docker Swarm, має бути розгорнутий глобально, тобто на кожному з вузлів оркестратора, дані з яких будуть використані для масштабування сервісів у системі, використаний Docker-образ - google/cadvisor.

Блок допоміжних сервісів, які не є обов’язковими для успішної роботи системи EtLT, а скоріше відіграють роль додаткових можливостей та візуалізації окремих аспектів, включає такі:

- kafka-ui – відповідає за надання візуалізації роботи складових системи Apache Kafka, а саме – доступних брокерів і черг у них, створючів та отримувачів повідомлень з черги в ролі компонентів source-connector та sink-connector, використаний Docker-образ - provectuslabs/kafka-ui.

- influx – відповідає за сховище для часових рядів, яке дозволить ефективно збирати інформацію з cadvisor-monitoring для подальшої візуалізації, використаний Docker-образ – influxdb.
- grafana – відповідає за моніторинг та візуалізацію стану сховища, зокрема в даному випадку виступатиме в ролі графічного представлення даних роботи вузлів Docker Swarm, що були зібрані в influx cadvisor, використаний Docker-образ – grafana/grafana.
- ksqlDb – відповідає за надання SQL-подібного інтерфейсу для здійснення потокових перетворень, який розгортатиметься за вимогою користувача, в межах брокерів Kafka, використаний Docker-образ – confluentinc/ksqldb-server.

Конфігурація з сервісів, що належить обом блокам, вміщена в окремому файлі docker-compose третьої версії, який можна розгорнути в межах середовища Swarm, виконавши відповідну команду для розгортання визначеного набору сервісів з вузла-менеджера. Для кожного сервісу мають бути вказані відповідні змінні оточення, значення налаштувань та особливості розгортання, які можуть включати вузли, на яких можливий запуск сервісів, а також кількість реплік та обсяг максимально доступних для використання ресурсів на кожному з учасників. Таким чином, якщо надати учасникам певні лейбли (від англ. label - позначки), за якими процес розгортання Swarm буде здатний розрізняти вузли один від одного не тільки за допомогою ролей. Як наслідок, можемо вказати кожному зі згаданих сервісів відповідні позначки вузлів для розгортання, аби забезпечити згадану раніше рівність і відмовостійкість кожного з етапів процесу.

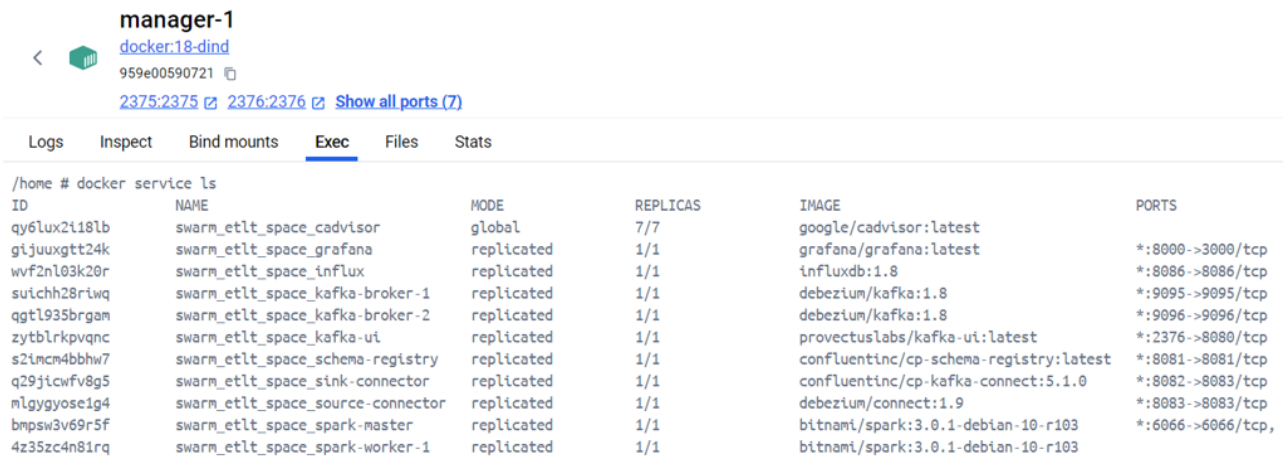


Рисунок 3.5 – Група основних і допоміжних сервісів, що розгорнуті на учасниках Docker Swarm, для забезпечення ETL-процесу

3.3 Компонента мікросервісів та супутні алгоритми

3.3.1 Складові системи мікросервісів та їхні особливості

Маємо підняте середовище Docker Swarm та всі запущені сервіси, що нам потрібні для забезпечення процесу обробки даних, на його відповідних учасниках. Самі по собі сервіси на вузлах не здатні надати зручний інтерфейс для керування перебігом процесу, тому тут приходить черга наступного компоненту в загальній системі фреймворку, а саме – мікросервіси, що задовольнятимуть різні серверні потреби. Тож розглянемо ці мікросервіси та їхнє призначення:

- swarm-space-handler – відповідає за надання можливостей керування середовищем Docker Swarm, зокрема він дозволятиме перевіряти стан оркестратора та його учасників, надавати можливість розгортання нових сервісів на відповідних вузлах та видаляти їх. Тож саме за його допомоги буде здійнюватися масштабування сервісів в середовищі. Також саме він дозволятме втручатися в процес tweaking на рівні Kafka брокеру, адже надаватиме інтерфейс для розгортання власного образу для забезпечення роботи Kafka Streams та ksqlDB, якщо користувач фреймворку матиме таку потребу. Більше того, він буде важливим гравцем в забезпеченні Medallion Architecture в сховищі, адже саме через нього буде можливість надсилати файли з кодом мови Python для виконання робіт Apache Spark, який і забезпечуватиме основне перетворення даних в системі.
- kafka-configurator – відповідає за надання інтерфейсу для підключення джерел сервісу source-connector, з яких будуть видубуватися дані, і сховищ sink-connector, до яких дані потраплятимуть з брокеру повідомлень. Також за допомогою нього можна буде впливати на конфігурацію цих підключень, зокрема вказувати граничний розмір вивантаження в сховища, виключати з переліку видобування окремі таблиці та докементи в джерелах.
- system-rest-api – відповатиме за надання REST API інтерфейсу для впливу на роботу згаданих вище мікросервісів, надаватиме візуалізацію інтерфейсних

ендпоінтів (від англ. endpoint – кінцева точка) за допомогою інструменту Swagger, що може стати в нагоді для майбутнього користувача фреймворку.

Окрім цих мікросервісів, будуть ще три не менш важливих, які матимуть спільну мету і схожі задачі, але при цьому будуть спиратися на різні компоненти в системі. Мова йде про мікросервіси, які будуть приймати рішення стосовно масштабування сервісів в середовищі Docker Swarm для окремих блоків аріхтектури EtLT, а тому зокрема слідкуватимуть за такими метриками, як завантаження процесора та оперативної пам’яті контейнером на учасниках. До цих мікросервісів входять такі: kafka-connectors-scaler – відповідатиме за прийняття рішень у масштабуванні сервісів, що стосуються видобування та вивантаження даних, kafka-brokers-scaler – за брокери та сервіси, що дотичні до брокера, зокрема Kafka Streams або ksqlDB, що можуть бути додані користувачем, spark-spaces-scaler – за масштабування сервісів-учасників у системі Spark відповідно. Таке розбиття зумовлено тим, аби була можливість розподілити моніторинг і масштабування окремих блоків системи, а також більш специфічно для роботи кожного приймати відповідні рішення.

Завдяки цим мікросервісам можна визначити систему фреймворку як agent-based, що відповідно залежить від роботи деяких програм-агентів. Для більшого розуміння цього поняття та проведення паралелей зі згаданими мікросервісами для масштабування, розглянемо такі якості агента:

- Автономність – агент здатний самостійно приймати рішення в деякому середовищі, без прямого втручання зі сторони іншої програми або безпосередньо людини [48], що відповідає можливостям наших мікросервісів, адже вони також власноручно займаються вирішенням питання масштабування.
- Реактивність – агент реагує на зміни в середовищі, завдяки можливості сприймати його окремі показники [48], що також присутнє у наших мікросервісах, адже вони реагуватимуть на зміни використання ресурсів контейнерами.

- Проактивність – агент не просто приймає рішення у відповідності до зміни стану середовища, а якому той знаходиться, а переслідує деяку мету, виконуючи ті чи інші дії [48], що також закладено в наших мікросервісах, адже їхньою метою буде забезпечення певного відсоткового використання ресурсів.
- Соціалізованість – здатність агента взаємодіяти з іншими агентами за допомогою визначених комунікаціних повідомлень [48], а це також присутнє в наших мікросервісах, адже кожен із них буде здатний взаємодіяти з іншими для отримання і надсилання певних даних.

3.3.2 Техніки для масштабування та пропозиція гібридного алгоритму

Перш ніж приступити до реалізації алгоритму масштабування, який буде використаний мікросервісами, слід спочатку розглянути найпоширеніші на сьогодні техніки для цього. Техніки – це одна з семи категорій у підході до автоматичного масштабування [44], яка полягає саме в тому, який програмний алгоритм використовується для його здійснення.

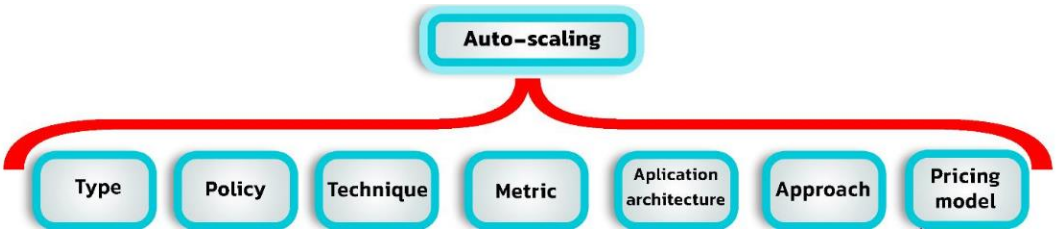


Рисунок 3.6 – Існуючі категорії для організації масштабування сервісів [44]

До поширених варіантів, що використовуються з категорії технік, часто відносять такі, що залежать від аспектів: порогових значень, нечітких правил, машинного навчання, теорії масового обслуговування, теорії контролю, часових рядів та профілювання виконуваної програми [44]. Цікаво, що в двох достатньо великих дослідженнях, одне з яких датоване весною 2018-го року [49], а друге - літом 2023-

го року [44], серед розглянутих і згаданих в посиланнях на інші роботи рішеннях відсутня комбінована техніка з використанням теорії контролю та нечітких правил. Незважаючи на це, переваги в такому поєднанні наводились вже досить давно [50]. Тож розглянемо його дещо докладніше, аби зрозуміти користь використання такого гібридного рішення для масштабування сервісів у нашій системі.

Почнемо з теорії контролю, вона полягає в постійному сприйнятті відгуків системи деяким контролером (від англ. controller - керівник) на його вказівки, які він генерує внаслідок порівняння цих самих відгуків із вхідними запитами, які той може отримувати від іншої системи або користувача. Для визначення цієї вказівки використовуються різні підходи до організації роботи контролера, одним з найпоширеніших і ефективніших є PID (Proportional Integral Derivative) [50].

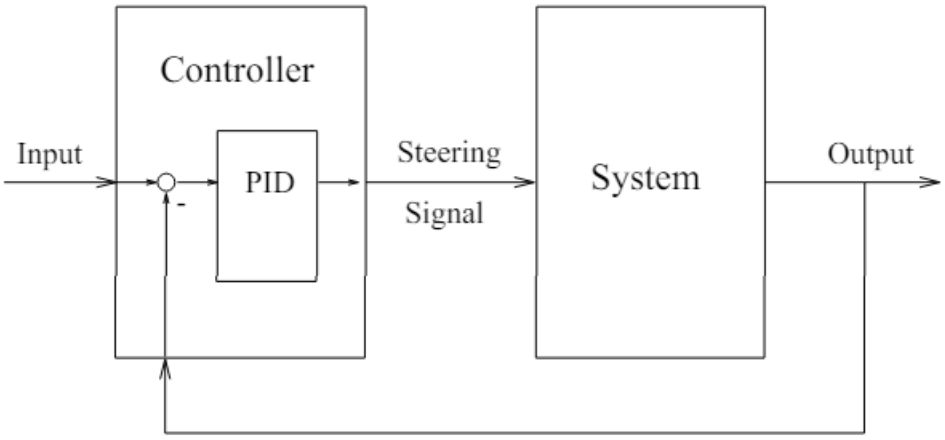


Рисунок 3.7 – Схема роботи за системи за теорією контролю [50]

Робота контролера на базі PID полягає в тому, аби впливати на фінальний вивід системи таким чином, аби наближати його до деякого бажаного стану, базуючись на значенні різниці, яку ще називають похибкою, між бажаним та існуючим станом системи. Незважаючи на складність теорії контролю як такої, саме підхід PID дозволяє якомога прозоріше зрозуміти її концепт, зокрема завдяки компонентам, які стоять за його абревіатурою, а саме - компонентів пропорційного, інтегрального та похідного коригування, сума значень роботи яких і дозволяє

генерувати сигнал для системи, аби наближати її до бажаного стану [50]. Розглянемо окремо кожний з компонентів:

- Пропорційний компонент - полягає у впливі на сигнал для системи шляхом залучення корегування, значення якого базується на похибці між бажаним результатом та теперішнім станом системи, тож значення цієї різниці напряму керує значенням корекції, адже чим вища різниця, тим значніше корегування [51].
- Інтегральний компонент – впливає на сигнал для системи шляхом залучення історії відслідкованих значень похибки, що були отримані внаслідок роботи вище згаданого компоненту. Більше того, особливість саме цього компоненту дозволяє контролеру продовжувати впливати на значення сигналу для системи навіть якщо пропорційний компонент повернув нульову різницю [51].
- Похідний компонент – полягає у визначенні можливих у майбутньому змін в похибці між бажаним і явним станом системи шляхом взяття похідної від швидкості змін цієї різниці відносно часу. Основною його задачею є загальне зменшення рівня та вібрацій змін у сигналі, що буде надісланий системі контролером, аби не було надмірного впливу на її стан в разі різкого збільшення або зменшення значення отриманого з інтегрального компоненту [52].

Також слід зазначити, що кожний зі згаданих компонентів супроводжується деякими коефіцієнтами: KP, KI та KD відповідно, значення яких можуть і мають бути адаптовані до потреб і результатів роботи контролера для конкретної системи [51].

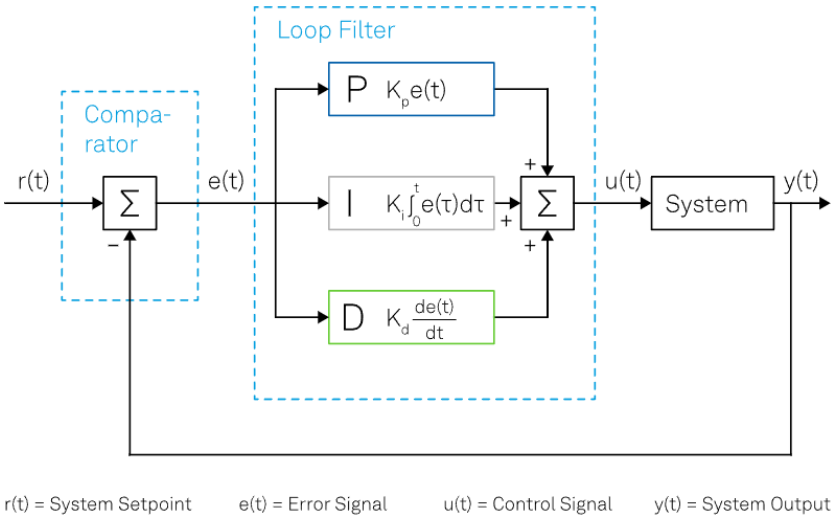


Рисунок 3.8 – Зображення роботи складових PID-контролера [53]

Уже під час безпосередньої роботи контролера і спостеріганням за значеннями сигналів, можна почати підлаштовувати його коефіцієнти для досягання окремих ефектів у роботі системи. Для допомоги в цьому слід скористатися запропонованою таблицею в дослідженні [54], в якій зазначено те, як саме зміни кожного з коефіцієнтів впливають на роботу системи, але майже завжди радять починати зі значень, що рівні 1, а вже потім корегувати.

Parameters	Rise time	Overshoot	Settling time	Steady state error
k_p	Decrease	Increase	Small change	Decrease
k_i	Decrease	Increase	Increase	Eliminate
k_d	Small Decrease	Decrease	Decrease	None

Рисунок 3.9 – Зображення таблиці ефектів зміни коефіцієнтів PID [54]

Тож враховуючи всі складові контролеру PID, можна дійти до висновку, що це досить прозора і гнучка система для реагування на зміни в системі, яка здатна наближати її до деякого бажаного стану. Більше того, для отримання сигналу не потрібно здійснювати значних обрахунків, які б могли вплинути на швидкодію

прийнтяття рішень. Незважаючи на це, ця техніка вимагає додаткових зусиль для її адаптації до конкретного випадку.

Незважаючи на ефективність і прозорість в роботі PID-контролеру, використання лише цієї техніки в масштабуванні сервісів у Docker Swarm може не надати бажаних результатів. Це обумовлено його лінійністю, адже контролер PID не зможе надати бажаної гнучкості в реагуванні на стан системи, яка не завжди свідчить про однозначність і рівень бажаної зміни. Так, наприклад, раптове підвищене використання процесора деяким контейнером сервісу не каже про те, що його одразу слід масштабувати, адже є ще й інші метрики, показники яких здатні краще й стабільніше відобразити наванатеження, зокрема показники використання пам’яті. Тож для того, аби була можливість адаптуватися під середовище сервісів та певним чином урізномаїтнювати варіанти значень, слід скористатися допомжіною технікою нечіткої логіки.

Особливістю цієї техніки є те, що вона дозовляє реалізувати можливість отримання нечітких результатів, потреба яких так часто зустрічається в циклі життя різних комплексних системах. Вона належить до багатозначних логік, в яких можливі відовіді розподілені по рівнях правди, тобто від найменшого рівня значенням 0 і до найбільшого – 1. Тож така техніка чудово відходить для систем, відповідь на перебіг процесів у яких не можна однозначно обрати лише з двох варіантів булевої логіки, а скоріше в проміжку між ними [55].

Імплементація такої техніки полягає у забезпеченні роботи таких чотирьох компонентів:

- Fuzzification (від англ. - розмивання) – компонент, робота якого полягає в перетворенні деяких конкретних значень в нечіткі [56], саме тут набір з двох згаданих значень буде розкладатися на групу, яка часто складається з кількох послідовних значень, що й описують кроки від 0 до 1 [57].
- Бази правил – компонент, який відповідає за визначені умови-правила, що відповідають на питання - якщо-то [57].

- Fuzzy Inference Process (від англ. – процес нечіткого висновку) – компонент, який відповідає за поєднання бази правил з отриманими, але вже розмитими значеннями [56].
- Defuzzification (від англ. – зворотній процес розмивання) – компонент, який здатний перетворити отриманий висновок в сприятливі для системи значення, якими та вже зможе скористатися [57].

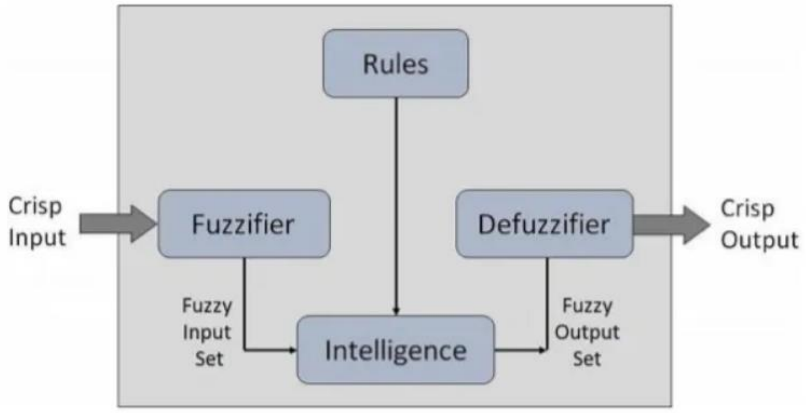


Рисунок 3.10 – Зображення повного процесу виконання нечіткої логіки [57]

Важливою частиною цієї техніки є також розуміння такого аспекту, як Fuzzy Set (від англ. - набір) та функцій членства (належності). Fuzzy Set – це набір елементів, що мають різні ступені членства, яке відповідає деякому числу в межах від 0 до 1 і тому здатне відображати різний рівень членства кожного [58]. Функції членства відіграють роль графічного відображення Fuzzy Set, визначаючи те, якому ступеню членства кожна конкретна точка відповідатиме [59]. Слід зазначити, що існують різні типи цих функцій, зокрема часто виділяють так, як триангулярні, трапезоїдні та Гауса [60]. У відповідності до досліджень, зазначених у роботі [60], часто виділяють саме першу та третю зі згаданих, але в силу того, що триангулярна значно легша в імплементації й використовує менше ресурсів для обрахування [60][61], тож вибір на її користь можна досить впевнено назвати оптимальним.

$$\mu_A(x) = \begin{cases} 0 & if\ x \le a \\ \dfrac{x-a}{b-a} & if\ a \le x \le b \\ \dfrac{c-x}{c-b} & if\ b \le x \le c \\ 0 & if\ x \ge c \end{cases}$$

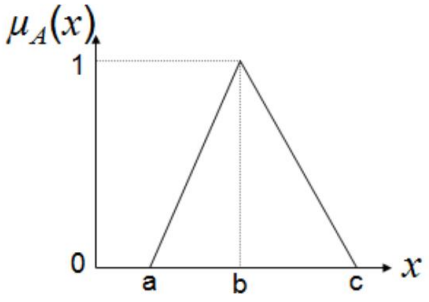


Рисунок 3.11 – Система рівнянь та графік триангулярної функції [62]

Для кращого розуміння цих аспектів, скористаємося прикладом зі статті Baeldung [63], де наведено застосування нечіткої логіки для визначення швидкості вентилятора за рівнем температури за Фаренгейтом. Хоч точні значення кожної з точок, які були залучені для створення функцій членства, не були наведіні, нехай вони будуть такими для трьох вхідних та вихідних наборів Fuzzy Set, що позначають рівні температури та швидкість вентилятора відповідно.

Вхідні набори:

1. cold: [0, 30, 50]
2. normal: [40, 55, 75]
3. hot: [65, 85, 100]

Вихідні набори:

1. slow: [0, 20, 40]
2. medium: [20, 40, 60]
3. fast: [40, 60, 80]

Внаслідок використання значень з кожних наборів, ми здатні отримати триангулярні функції членства, які матимуть такі графіки, як відображено на рисунку нижче. Слід зазнчити, що для легшого відображення, вони зобразили ступінь членства в межах від 0 до 100, а не до 1. Дивлячись на перший графік, бачимо, що число 30 з першого набору має найвищій ступінь членства для cold, 55 – для normal, а 85 – для hot відповідно. Тож коли на вхід прийде число 45, його слід

буде розмити шляхом використаннях функції членства, що була наведена вище, для кожного з наборів. Внаслідок чого отримаємо ступінь того, наскільки 45 належить cold, normal та hot. Після цього вже почнеться Inference Processing, який на базі деяких Fuzzy-правил визначить те, наскільки число 45 належить кожному з вихідних наборів. Отримання ступені членства пізніше будуть використання для отримання вже фінального значення, що відповідатиме швидкості вентилятора.

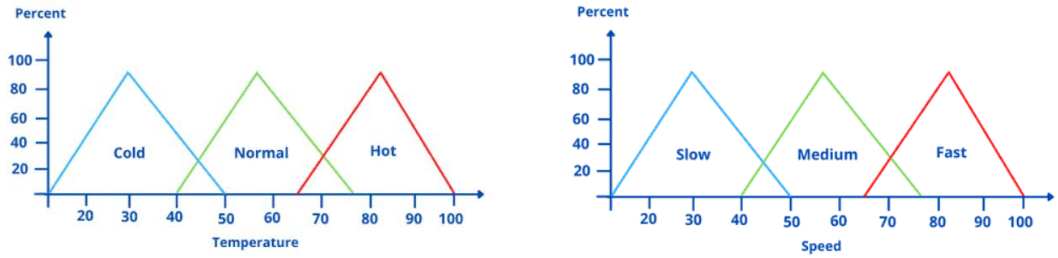


Рисунок 3.12 – Графіки триангулярних функцій для вхідних і вихідних наборів Fuzzy Set з прикладу на Baeldung [63]

Тож тепер, коли є розуміння особливостей обох технік, можемо приступити до побудови гібридного рішення, яке дозволить приймати рішення для масштабування сервісів в Docker Swarm. Враховуючи переваги нечіткої логіки, зробимо її основною в сенсі визначення дії для масштабування, адже ми здатні об’єднати кілька показників для того, аби отримувати деякий результат. У нашому випадку цими показниками буде відповідно відсоток використання процесору і пам’яті контейнером сервісу. Для цього наведемо три Fuzzy Set, два з яких відповідатимуть за вхідні набори, а один для вихідних відповідно. Слід зазначити, що значення за замовченням в цих наборах і коефіцієнтів для PID буде можливо змінити зі сторони користувача через панель налаштування за допомогою REST API або UI системи.

```
private readonly inputFuzzySetUsageOfCPU = {
  low: [0, 25, 50],
  average: [25, 50, 75],
  high: [50, 75, 100],
};

private readonly inputFuzzySetUsageOfMemory = {
  low: [0, 25, 50],
  average: [25, 50, 75],
  high: [50, 75, 100],
};

private readonly outputFuzzySetReplicasScaling = {
  downscale: [-2, -1, 0],
  stale: [-1, 0, 1],
  upscale: [0, 1, 2],
};
```

Рисунок 3.13 – Вхідні і вихідні набори Fuzzy Set для нашого алгоритму

Після цього можемо визначити функцію членства на базі триангуляції, аби була змога отримати відповідні ступені членства для показників low, average та high, де параметрами будуть виступати вхідне число відсотку використання процесору або пам’яті, а також значення, які відповідають деякому з Fuzzy Set наборів.

Після того, як ступені членства отримані для кожного з вхідних наборів, можемо приступати до Inference Processing, який на базі правил дозволить нам отримати якийсь із вихідних наборів для downscale, stale або upscale. Для цього визначимо список правил, в якому буде здійснено перебір всіх комбінацій рівня використання процесора разом з пам’яттю. Через те, що для обох показників у нас по три набори, маємо загалом дев'ять правил, виконання кожного з яких відповідає певному вихідному набору. Визначити назву вихідного набору не достатньо для того, аби продовжити прийняття рішення, а тому нам слід також визначити ступінь членства для кожного з них. Для цього скористаємося взяттям найменшого зі значень степня членства для вхідних наборів у відповідності до правил, а саме – min(ступінь членства відповідному набору для показника процесору, ступінь членства відповідному набору для показника пам’яті)

```
{
  if: (
    cpuFuzzification: FuzzificationResponse,
    memoryFuzzification: FuzzificationResponse
  ) => cpuFuzzification.high && memoryFuzzification.high,
  then: "upscale",
  evaluatedValue: (
    cpuFuzzification: FuzzificationResponse,
    memoryFuzzification: FuzzificationResponse
  ) => Math.min(cpuFuzzification.high, memoryFuzzification.high),
},
```

Рисунок 3.14 – Визначення правил у відповідності до отриманих ступенів членства для вхідних наборів

Внаслідок застосування такої операції до кожного з правила, ми зможемо визначити те, який саме ступінь членства матиме відповідна комбінація показників використання процесору та пам’яті. Оскільки в нас багато правил і ввідповідність деякому набору зустрічається по кілька разів, нам слід буде взяти максимальне значення з визначених ступенів для кожного вихідного набору.

```
return {
  downscaleMax: Math.max(...inferenceResult.downscale),
  staleMax: Math.max(...inferenceResult.stale),
  upscaleMax: Math.max(...inferenceResult.upscale),
};
```

Рисунок 3.15 – Вибір максимального значення ступеню для кожного з вихідних наборів у відповідності до правил

Після цього матимемо здійснити процес знаходження деякого значення для дії масштабування, який вже є частиною Defuzzification, спираючись на значення вихідних наборів та попердньо отриманих степінів членства для кожного з них. Для знаходження цього значення використовують різні методи, зокрема до найпоширеніших та точніших відносять COG (Center of Gravity) або ж Centroid, COA (Center of Area) та MOM (Mean of Maximum). Усі вони надають схожі результати, тож вибір залежатиме скоріше від швидкості обрахунків, ніж від чогось

іншого [64]. Часто виокремлюють саме Centroid як найоптимальніший [65]. Тож отримавши значення COG для кожного вихідних наборів, можемо обрахувати зважений результат, який приведе нас до найбільш сприятливого значення одного з наборів, яке слід здійснити для масштабування контейнерів.

Після того, як ми успішно отримали рішення масштабування за технікою нечіткої логіки, можемо перейти до залучення PID-контролеру. Отримане значення в результаті нечіткої логіки для зміни кількості контейнерів дозовлить нам створити значення мети, яку матиме переслідувати техніка PID. Таким чином, якщо для деякого контейнеру ми отримали значення 1.3, що свідчить про збільшення кількості реплік контейнеру, вирахуємо значення мети для PID. Для цього скористаємося теперішнім число реплік та додамо відповідне число, заздалегідь його округливши, якщо воно не було цілим. Маючи бажану та теперішню кількість реплік, можемо приступити до обрахунку сигналу за відповідною формулою для PID.

```
calculatePidSignalForReplicasChange(currentNumberOfReplicas: number, desiredNumberOfReplicas: number) {
    const currentDateTime = Date.now()

    // Пропорційна компонента PID, e(t)
    const currentErrorVal = desiredNumberOfReplicas - currentNumberOfReplicas;
    //Δt
    const timeInterval = (currentDateTime - this.previousDateTime) / this.msInMins;
    //Δe(t)=e(t)-e(t-1)
    const changeInError = currentErrorVal - this.previousErrorVal;

    // Похідна компонента, Δe(t)/Δt
    const rateOfChangeOfError = changeInError / timeInterval;
    this.sumOfErrorVals += currentErrorVal;

    // Інтегральна компонента, ∫e(t)dt
    const integralOfErrors = this.sumOfErrorVals * timeInterval;

    this.previousErrorVal = currentErrorVal;
    this.previousDateTime = currentDateTime;

    return this.KP * currentErrorVal + this.KI * integralOfErrors + this.KD * rateOfChangeOfError;
}
```

Рисунок 3.16 – Підрахування значення сигналу для зміни кількості реплік контейнеру за PID

Після успішного виконання контролеру, отримаємо фінальне значення реплік для контейнеру деякого сервісу, яке зможемо надіслати середовищу Docker Swarm за допомогою мікросервісу swarm-handler, який розгорне додаткові репліки у

відповідності до ідетифікаторів контейнерів. Описаний алгоритм викликається з деяким інтервалом, який в коді можна буде змінити. Тож як тільки пройде деякий проміжок часу, такий масштабувальник звернеться до swarm-handler, звідки отримає потрібні значення завантаженості процесору та пам’яті контейнерами, що відповідають зоні відповідальності конкретного масштабувальника. Завдяки тому, що кожен розгорнутий раніше контейнер сервісу має відповідну приставку в назві, яка описує його блок приналежності, ми здатні з легкістю дістати всі необхідні дан.

Отже, має гібридну техніку прийняття рішення для масштабування, залучаючи техніки нечіткої логіки та теорії контролю. Така комбінація дозволила нам впоратися з нелінійністю прийняття рішення щодо масштабування, базуючись на кількох показниках, та відповідно отримати згладжене значення для зміни кількості реплік, яке буде можливо коригувати під час роботи системи за допомогою відповідних коефіцієнтів PID.

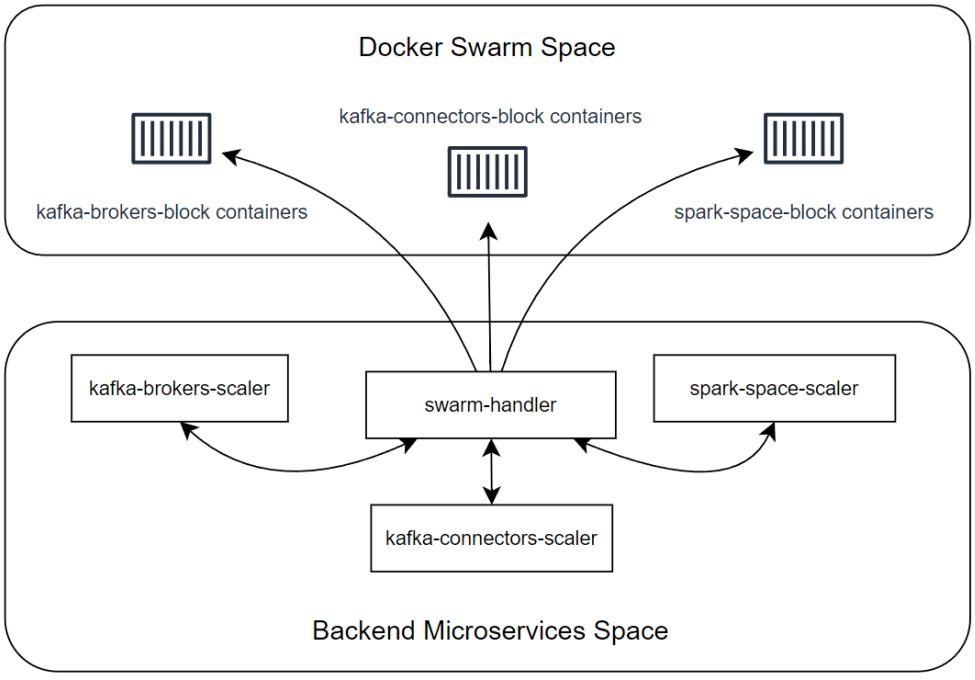


Рисунок 3.17 – Взаємодія масштабувальників з компоненти мікросервісів з компонентою середовища Docker Swarm

3.4 Компонента веб-плафторми та сценарій використання

Тепер коли описана основа фреймворку, перейдемо до фінального компоненту, який відповідає за його UI. Для кращого розуміння інтерфейсу, відтворимо повний цикл його використання для побудови ETL-системи. У разі успішного розгортання сервісів в межах Docker Swarm, користувач бачитиме таку візуалізацію компонентів EtLT-архітектури (див. додаток А). З самого початку користувачу слід додати джерела для видобування даних. Для цього, йому слід натиснути на кнопку додавання Connected sources, де йому буде представлена можливість обрати підтримувану базу даних, зокрема PostgreSQL та MongoDB, де буде можливість заповнити потрібні поля для налаштування підключення.

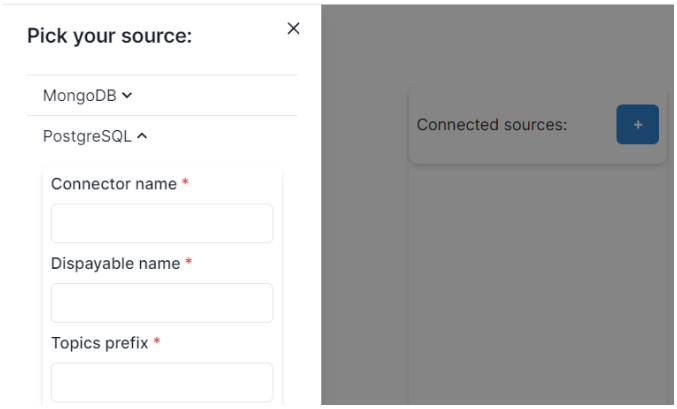


Рисунок 3.18 – Наслідок натиснення на кнопку додавання нового джерела

У разі успішного підключення, користувачу буде сповіщено про це у вигляді тимчасового повідомлення, після цього з’явиться відповідна картка конектора для джерела. На ній буде видно статус видобування, а саме – running або failing, а також розташовані допоміжні кнопки для видалення підключення та оновлення налаштувань. До оновлення налаштувань зокрема входить можливість зміни префіксів для топіків, в які видобуваються дані з конкретного джерела.

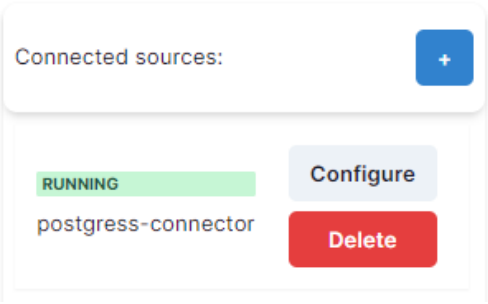


Рисунок 3.19 – Картка успішного підключення джерела

Також у користувача буде можливість залучити потокову обробку повідомлень, він зможе як розгорнути ksqlDB, так і власноруч написаний код для Kafka Streams на будь-якій мові, яка має відповідну підтримку. Для використання цієї програми треба буде створити Docker-образ та завантажити його на публічний Docker Hub, для можливості автоматизованого розгортання як контейнера в середовищі Docker Swarm.

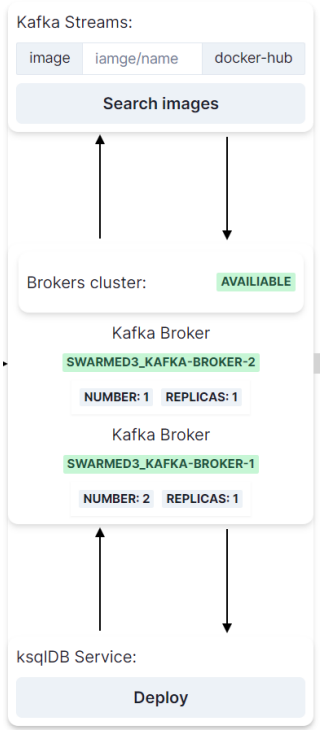


Рисунок 3.20 – Блок брокерів Kafka та допоміжних сервісів для етапу tweaking

Передбачена можливість глобального пошуку за назвою образу або ж створювачем на Docker Hub, для цього слід почати відповідних ввід, після чого одразу буде запропонований деякий перелік з образів, які підібрала пошукова система Docker.

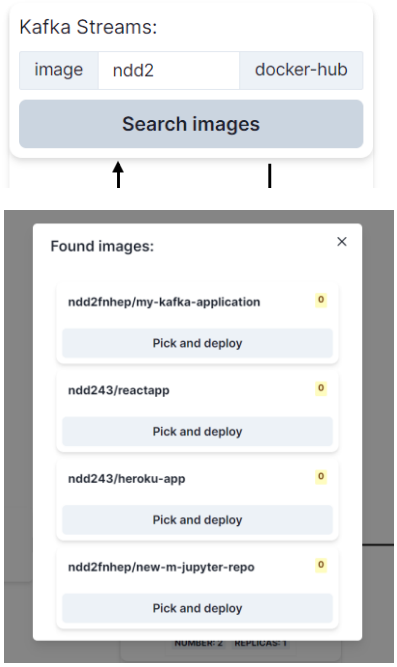


Рисунок 3.21 – Картка пошуку Docker-образу для Kafka Streams

У разі успішного підняття обраного образу для забезпечння Kafka Streams або ksqlDB, про це також буде повідомлено користувачу, після чого з’являться відповідні картки, на яких буде видно такі показники, як статус сервісу та кількість реплік його контейнеру. Слід зазначити, що їх також можна буде прибрати з Docker Swarm за потреби.

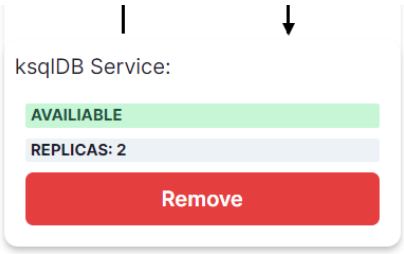


Рисунок 3.22 – Картка успішного розгортання сервісу ksqlDB

Тож тепер черга етапу вивантаження даних з брокерів Kafka, для цього користувачу слід буде відтворити ті самі кроки, що той здійснював для підключення джерел. Картка буде виглядати так само, що дозволить користувачу легше орієнтуватися в підключеннях.

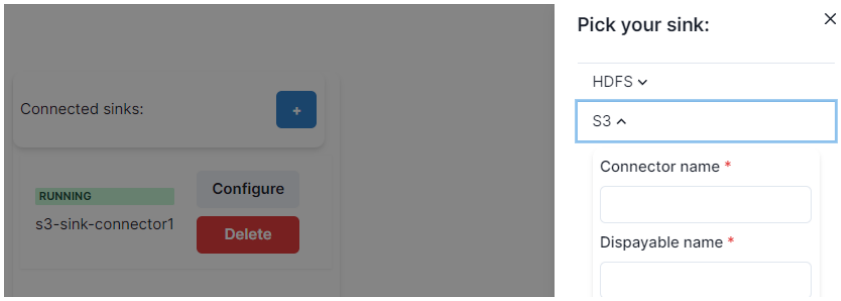


Рисунок 3.23 – Аналогічна до джерел картка під’єднання цільового сховища

Маючи підключені джерела, визначені додаткові перетоврення для брокерів Kafka та підключене цільове сховище, користувач здатний зайнятись фазою перетворень вже безпосердньо в Data Lake, зокрема для того, аби забезпечити архітектуру Medallion. Тож для цього йому надається візуалізований блок кластеру Spark, де він здатний побачити його доступність, кількість працівників та поля для завантеження spark-job файлів. Таким чином, людина здатна реалізувати власну логіку для потрібного перетворення даних у формати, що задовольнятимуть його бачення таких рівнів Medallion, як срібний та золотий відповідно. Він може з легкістю обрати доступний на його системі файл шляхов його перетягування або прямого пошуку файловій системі його машини, файл повинен містит Python-код, в якому буде реалізована логіка перетворень даних з Data Lake за допомогою PySpark та використанням бібліотеки Delta Lake. Слід зауважити, що в разі неуспішного виконання якого з надісланих файлів, користувачу про це буде повідомлено.

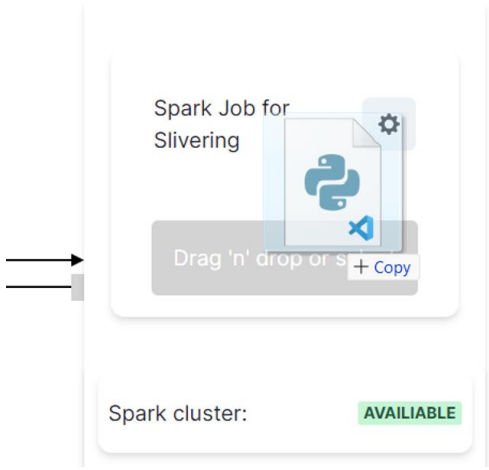


Рисунок 3.24 – Пересування файлу з Python-кодом для Spark

Як наслідок, весь шлях базового сценарію використання фреймворку продемонстровано. Незважаючи на це, це ще не всі можливості, якими може скористатися користувач, зокрема у сенсі візуалізації компонентів процесу обробки даних. Він також здатний звернутися до розгорнутих сервісів, що здатні показати докладнішу інформацію перебіг роботи сервісів. Наприклад, за потреби подивитися докладнішу інформацію по складовим Kafka, він здатний в браузері звернутися до інтерфейсу раніше згаданого сервісу kafka-ui за відповідним портом. Порти кожного з сервісів можливо побачити за допомогою команд перегляду складових Docker Swarm, зокрема на вузлі менеджера виконати команду docker service ls, де у виведеній табличці в найправішій колонці будуть вказати всі порти.

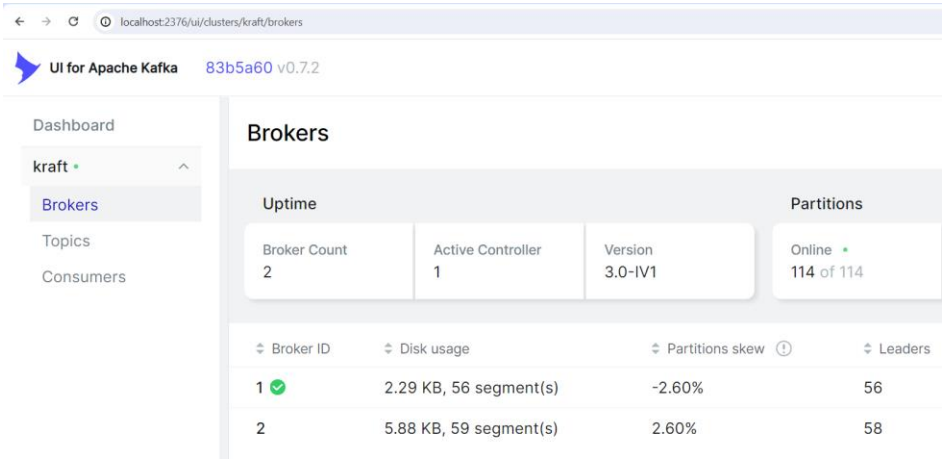


Рисунок 3.25 – Панель стану брокерів Kafka

Якщо ж буде потреба наглядно побачити використання ресурсів на вузлах-учасниках середовища Docker Swarm, то користувач також може звернутися до інтерфейсу Grafana, яка за допомогою сервісів influx та cadvisor, який розташований на кожному з контейнерів, здатна докладно візуалізувати навантаження. Єдине що, користувачу потрібно буде додати джерело формації графіків, вказавши порт сховища influx і відповідну назву бази даних, яка буде cadvisor. Потім перейшовши на створення Dashboard (див. додаток Б), у користувача буде одразу можливість вказати деякий запит, який діставатиме з influx дані по конкретному учаснику з середовища Swarm та надаватиме візуалізацію використання всіх доступних ресурсів вузлів, зокрема наванатежння процесору, використання пам’яті та пропускної здатності. Для цього слід заздалегідь вказати відповідний ідентифікатор учасника, якого користувач бажає переглянути. Цей ідентифікатор контейнеру сервісу cadvisor можна обрати із доступного переліку, який надасть схема influx, як показано на рисунку нижче.

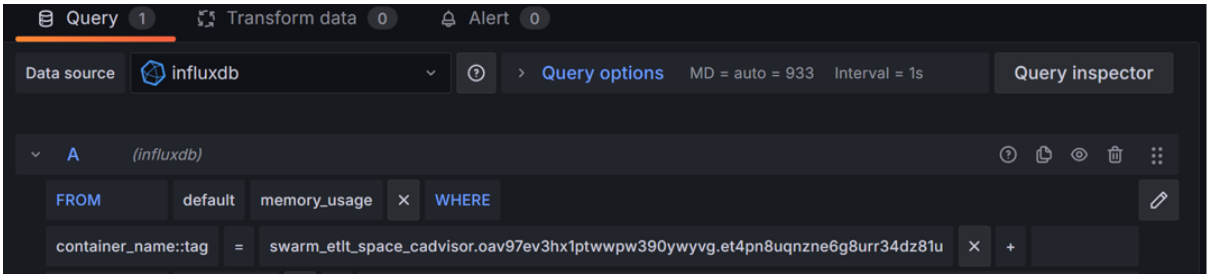


Рисунок 3.26 – Панель створення запиту Grafana

3.5 Висновки до розділу 3

У цьому розділі здійснено аналіз технічного завдання, поступовий огляд та його реалізацію шляхом опису компонентів фреймворку для масштабованої ETL-системи. Описано взаємодію кожного з компонентів між собою та їхні особливості. Тут наведено основні кроки по розгортанню локального середовища Docker Swarm разом з усіма сервісами для забезпечення роботи фреймворку. Здійснено огляд складових мікросервісної архітектури, яка відповідає за виконання серверних завдань, зокрема розгортання, масштабування та налаштування, а також надання REST API для взаємодії з системою користувачем та веб-платформою. Наведена реалізація гібридного алгоритму із залученням технік нечіткої логік та теорії контролю, для здійснення рішень щодо масштабування. Показані основні елементи веб-плафторми фреймворку для побудови та керування повним циклом обробки даних у відповідності до архітектури EtLT, а також зазначені додаткові можливості візуалізації окремих сервісів та складових системи, зокрема таких, як візуалізатор kafka-ui та моніторингова система Grafana.

ВИСНОВКИ

У підсумку виконання роботи розроблено фреймворк для масштабованої ETL-системи, з врахуванням викликів BigData. Таким чином, були поступово виконані всі пункти поставленої задачі, що дозволило реалізувати фреймворк, який може стати в пригоді користувачу для побудови ETL-системи у відповідності до сучасної архітектури EtLT. Розроблений фреймворк здатний надати гнучкість в побудові повного циклу обробки даних та забезпечити очікувані потреби в ефективності такого процесу, адже враховує сучасні вимоги в обробці даних. Перед безпосереднім етапом реалізації фреймворку були розглянуті компоненти та етапи, що притамані системам ETL. Також зроблено обгрунтований вибір відповідних рішень та практик в їхній реалізації. У фінальному розділі роботи, який відповідає за опис реалізації фреймворку, було наведено основні архітектурні рішення та алгоритми, зокрема гібридної техніки для масштабування, а також продемонстровано основні елементи взаємодії з системою через графічний інтерфейс веб-платформи.

Підбиваючи підсумки, треба зазначити вектори поліпшення розробленого фреймворку, зокрема в сенсі його централізованості та масштабування. У випадку першого аспекту гарним кроком на шляху покращень була б поява редактору для написання запитів для ksqlDb та Spark, що в теперішній реалізації можливо лише через прямі звернення за протоколом HTTP для першого та готові файли з Python-кодом для другого відповідно. У випадку другого аспекту слід розглянути залучення машинного навчання в процес прийняття рішень стосовно масштабування контейнерів сервісів. Наприклад, можна скористатися досить популярною сьогодні технікою Reinforcement Learning [66] для покращення ефективності та адаптованості алгоритму масштабування, який є основною успішності роботи системи обробки даних. Тож можливостей для покращень

вдосталь, а тому все залежить від зростаючих потреб користувачів, які все більше зацікавлені у залучені машинного навчання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Thabet N., Soomro T. Big Data Challenges. Journal of Computer Engineering & Information Technology. 2015. Т. 4, № 3. С. 1–2.
2. Simitsis A., Skiadopoulos S., Vassiliadis P. The History, Present, and Future of ETL Technology. Test-of-Time Award - Invited Talk. 2023. С. 3–12.
3. What is ETL (extract, transform, load)?. IBM. [Електронний ресурс]. URL: https://www.ibm.com/topics/etl (дата звернення: 15.03.2024).
4. Zode M. The Evolution of ETL -From Hand-coded ETL to Tool-based ETL. Cognizant Technology Solutions. С. 2–5.
5. Gorhe S. ETL in Near-real-time Environment: A Review of Challenges and Possible Solutions : Research. Auckland, 2020.
6. Nambiar A., Mundra D. An Overview of DataWarehouse and Data Lake in Modern Enterprise Data Management. Big data and cognitive computing. 2022. Т. 6, № 132. С. 2–24. URL: https://doi.org/10.3390/bdcc6040132 (дата звернення: 15.03.2024).
7. Salqvist P. A comparative study of the Data Warehouse and Data Lakehouse architecture : Degree Project in the Field of Technology Information Technology and the Main Field of Study Computer Science and Engineering. Stockholm, 2023. 147 с.
8. What is a data warehouse?. IBM. [Електронний ресурс]. URL: https://www.ibm.com/topics/data-warehouse (дата звернення: 15.03.2024).
9. Techmagic. Data Lake VS. Data Warehouse: Which Effective For Data Management. Medium. [Електронний ресурс]. URL: https://medium.com/techmagic/data-lake-vs-data-warehouse-which-effective-for-data-management-60b3e3b2a75a (дата звернення: 15.03.2024).

10. Ascend.io. Data warehouse, data lake, and the features of ETL and ELT. Linkedin. [Електронний ресурс]. URL: https://www.linkedin.com/pulse/data-warehouse-lake-features-etl-elt-ascend-io/ (дата звернення: 15.03.2024).
11. QuantumBlack AI by McKinsey. Lakes? Warehouses? Lakehouses? A short history of Data Architecture. Medium. [Електронний ресурс]. URL: https://medium.com/quantumblack/lakes-warehouses-lakehouses-a-short-history-of-data-architecture-bc942b0ed463 (дата звернення: 15.03.2024).
12. Ascend.io. The ETL to ELT to EtLT Evolution, and data pipelines. Linkedin. [Електронний ресурс]. URL: https://www.linkedin.com/pulse/etl-elt-etlt-evolution-data-pipelines-ascend-io/ (дата звернення: 15.03.2024).
13. Kakish K., Kraft T. A. ETL Evolution for Real-Time Data Warehousing. м. New Orleans Louisiana, листоп. 2012 р.
14. deBuzna J. Four Methods of Change Data Capture. Dataversity. [Електронний ресурс]. URL: https://www.dataversity.net/four-methods-of-change-data-capture/ (дата звернення: 15.03.2024).
15. ByteHouse. 7 advantages of using log-based CDC vs other methods. Medium. [Електронний ресурс]. URL: https://medium.com/@bytehousecloud/7-advantages-of-using-log-based-cdc-vs-other-methods-89a83f124529 (дата звернення: 15.03.2024).
16. Buckenhofer A. Log-based Change Data Capture – lessons learnt. Medium. [Електронний ресурс]. URL: https://medium.com/mercedes-benz-techinnovation-blog/change-data-capture-lessons-learnt-7976391cf78d (дата звернення: 15.03.2024).
17. What is a message broker?. IBM. [Електронний ресурс]. URL: https://www.ibm.com/topics/message-brokers (дата звернення: 15.03.2024).
18. Łuczak B. Message brokers – use cases & model AWS implementation using SNS & SQS. The Software House. [Електронний ресурс]. URL: https://tsh.io/blog/message-broker/ (дата звернення: 15.03.2024).

19. Tolety K. Popular Message Broker Platforms for 2024. Hevo. [Електронний ресурс]. URL: https://hevodata.com/learn/popular-message-broker/ (дата звернення: 15.03.2024).
20. Benchmarking Message Queues / R. Maharjan та ін. Telecom. 2023. Т. 4, № 2. URL: https://doi.org/10.3390/telecom4020018 (дата звернення: 15.03.2024).
21. Confluent. Kafka Connect Single Message Transform Reference for Confluent Platform. Confluent Documentation. [Електронний ресурс]. URL: https://docs.confluent.io/platform/current/connect/transforms/overview.html (дата звернення: 15.03.2024).
22. Traphagen D. Kafka Streams and ksqlDB Compared – How to Choose. Confluent Blog. [Електронний ресурс]. URL: https://www.confluent.io/blog/kafka-streams-vs-ksqldb-compared/ (дата звернення: 15.03.2024).
23. Confluent. KSQL and Kafka Streams. Confluent Documentation. [Електронний ресурс]. URL: https://docs.confluent.io/legacy/platform/5.1.4/ksql/docs/concepts/ksql-and-kafka-streams.html (дата звернення: 15.06.2024).
24. Sachin D. N. Distinguishing HDFS from Cloud Data Lakes: ADLS Gen2 and Amazon S3. Linkedin. [Електронний ресурс]. URL: https://www.linkedin.com/pulse/distinguishing-hdfs-from-cloud-data-lakes-adls-gen2-amazon-d-n--7e2fc/ (дата звернення: 15.03.2024).
25. BryteFlow. How to choose between Parquet, ORC and AVRO for S3, Redshift and Snowflake?. BryteFlow. [Електронний ресурс]. URL: https://bryteflow.com/how-to-choose-between-parquet-orc-and-avro/ (дата звернення: 15.03.2024).
26. Databricks. Medallion Architecture. Databricks. [Електронний ресурс]. URL: https://www.databricks.com/glossary/medallion-architecture (дата звернення: 15.03.2024).

27. Abiola A D. Medallion Architecture Layers in Microsoft Fabric Lakehouse. Linkedin. [Електронний ресурс]. URL: https://www.linkedin.com/pulse/medallion-architecture-layers-microsoft-fabric-a-david-msc-mvp-xkcue/ (дата звернення: 15.03.2024).
28. Gusarov A. Delta, Hudi, and Iceberg: The Data Lakehouse Trifecta. DZone. [Електронний ресурс]. URL: https://dzone.com/articles/delta-hudi-and-iceberg-the-data-lakehouse-trifecta (дата звернення: 15.03.2024).
29. Joffe T. SQL Query Engines – Intro and Benchmark. Medium | Nielsen-TLV-Tech-Blog. [Електронний ресурс]. URL: https://medium.com/nmc-techblog/sql-query-engines-intro-and-benchmark-44a658b47810 (дата звернення: 15.03.2024).
30. Kim D. Hive, Presto, and Spark on TPC-DS benchmark. SlideShare Presentation. [Електронний ресурс]. URL: https://www.slideshare.net/slideshow/hive-presto-and-spark-on-tpcds-benchmark/71534941 (дата звернення: 15.03.2024).
31. TPC Benchmarks Overview. TPC ORG. [Електронний ресурс]. URL: https://www.tpc.org/information/benchmarks5.asp (дата звернення: 15.03.2024).
32. Weller K. Delta, Hudi, Iceberg – Which is most popular?. Medium. [Електронний ресурс]. URL: https://medium.com/@kywe665/delta-hudi-iceberg-which-is-most-popular-29ca56767199 (дата звернення: 15.03.2024).
33. Weller K. Delta, Hudi, Iceberg – A Benchmark Compilation. Medium. [Електронний ресурс]. URL: https://medium.com/@kywe665/delta-hudi-iceberg-a-benchmark-compilation-a5630c69cffc (дата звернення: 15.03.2024).
34. Gandhi R., Szmrecsanyi P. The Benefits of Containerization and What It Means for You. IBM. [Електронний ресурс]. URL: https://www.ibm.com/blog/the-benefits-of-containerization-and-what-it-means-for-you/ (дата звернення: 15.03.2024).
35. What is container orchestration?. Red Hat. [Електронний ресурс]. URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration. (дата звернення: 15.03.2024).

36. What is container orchestration?. VMWare. [Електронний ресурс]. URL: https://www.vmware.com/topics/glossary/content/container-orchestration.html (дата звернення: 15.03.2024).
37. Vauban V. Orchestrator Comparison: Docker Swarm vs Kubernetes vs Apache Mesos. Linkedin. [Електронний ресурс]. URL: https://www.linkedin.com/pulse/orchestrator-comparison-docker-swarm-vs-kubernetes-apache-vauban/ (дата звернення: 15.03.2024).
38. Nickoloff J. Evaluating Container Platforms at Scale. Medium. [Електронний ресурс]. URL: https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c#.pbxx2w5ai (дата звернення: 15.03.2024).
39. Kronis. Docker Swarm over Kubernetes. Kronis Blog. [Електронний ресурс]. URL: https://blog.kronis.dev/articles/docker-swarm-over-kubernetes (дата звернення: 15.03.2024).
40. Deploy to Swarm Swarm.docs. [Електронний ресурс]. URL: https://docs.docker.com/get-started/swarm-deploy/ (дата звернення: 15.03.2024).
41. Raft consensus in swarm mode. Swarm.docs. [Електронний ресурс]. URL: https://docs.docker.com/engine/swarm/raft/ (дата звернення: 15.03.2024).
42. Swarm mode key concepts. Swarm.docs. [Електронний ресурс]. URL: https://docs.docker.com/engine/swarm/key-concepts/ (дата звернення: 15.03.2024).
43. What is a software framework?. Educative. [Електронний ресурс]. URL: https://www.educative.io/answers/what-is-a-software-framework (дата звернення: 15.05.2024).
44. ZargarAzad M., Ashtiani M. An Auto-Scaling Approach for Microservices in Cloud Computing Environments. Journal of Grid Computing. 2023. Т. 21, № 73. URL: https://doi.org/10.21203/rs.3.rs-3020374/v1 (дата звернення: 15.03.2024).

45. Pickavance M. Best virtual machine software of 2024. TechRadar. [Електронний ресурс]. URL: https://www.techradar.com/best/best-virtual-machine-software (дата звернення: 15.03.2024)
46. Kushwah S. Docker Inside Docker. Medium. [Електронний ресурс]. URL: https://medium.com/@shivam77kushwah/docker-inside-docker-e0483c51cc2c (дата звернення: 15.03.2024).
47. Christen J., Delahunt J. Docker Swarm Local. GitHub. [Електронний ресурс]. URL: https://github.com/dweomer/docker-swarm-local/blob/master/docker-compose.yml (дата звернення: 15.03.2024).
48. Wooldridge M. Agent-Based Software Engineering. 1997. С. 2–3.
49. Taherizadeh S., Stankovski V. Dynamic Multi-level Auto-scaling Rules for Containerized Applications. The Computer Journal. 2018. Т. 62, № 2. URL: https://doi.org/10.1093/comjnl/bxy043 (дата звернення: 15.03.2024).
50. Riedl R. How Fuzzy Control can Benefit from Classical Control Theory: The Fuzzy State Controller. м. Zurich, 12 лют. 1997 р.
51. Introduction: PID Controller Design. Control Tutorials For Matlab ans Simulink. [Електронний ресурс]. URL: https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID (дата звернення: 15.03.2024).
52. Proportional Integral Derivative (PID). Apmonitor | Dynamics and Control. [Електронний ресурс]. URL: https://apmonitor.com/pdc/index.php/Main/ProportionalIntegralDerivative (дата звернення: 15.03.2024).
53. Principles of PID Controllers. Zurich Instruments. [Електронний ресурс]. URL: https://www.zhinst.com/europe/de/resources/principles-of-pid-controllers (дата звернення: 15.03.2024).
54. Thiru R. HARDWARE IMPLEMENTATION OF MODEL PREDICTIVE CONTROL BASED FUZZY LOGIC CONTROLLER FOR TEMPERATURE

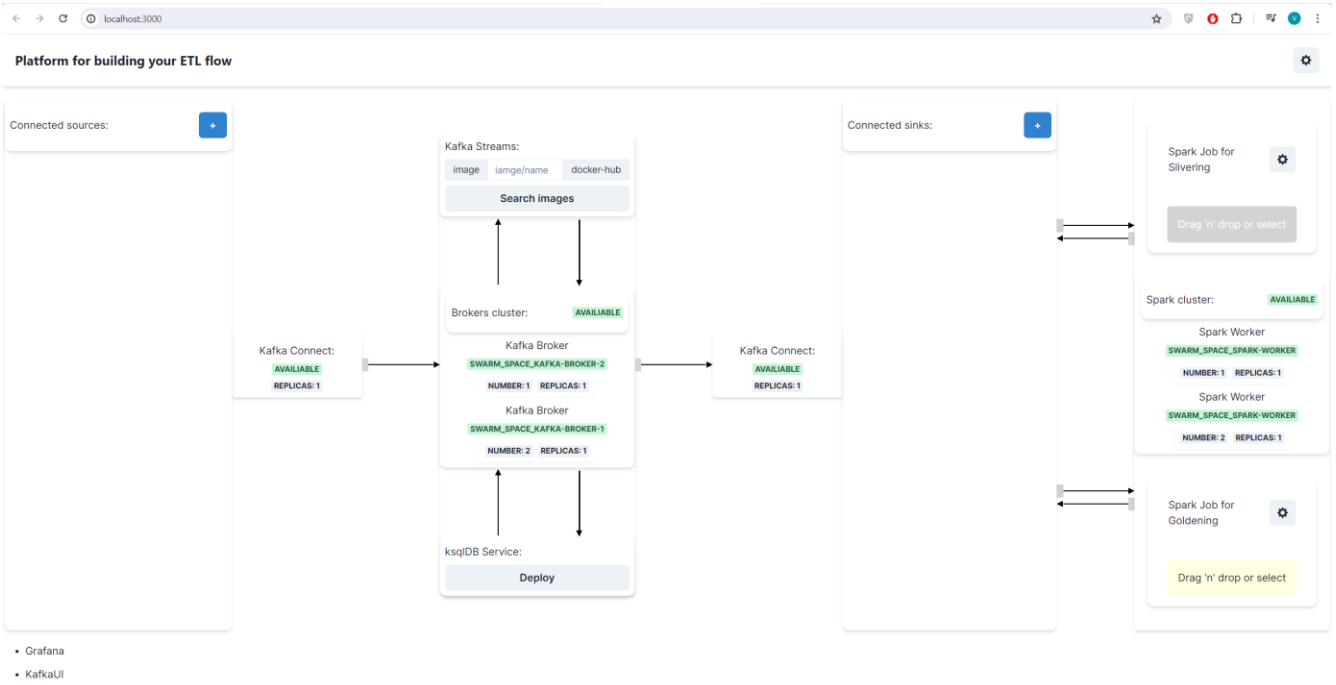
PROCESS STATION. International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering. 2013. Т. 2, № 5.

55. Fuzzy Logic. Stanford Encyclopedia of Philosophy. [Електронний ресурс]. URL: https://plato.stanford.edu/entries/logic-fuzzy/ (дата звернення: 15.03.2024).
56. Advanced Fuzzy Logic Technologies in Industrial Applications / ред.: Y. Bai, H. Zhuang, D. Wang. London : Springer London, 2006. URL: https://doi.org/10.1007/978-1-84628-469-4 (дата звернення: 02.06.2024).
57. Madathil N. FUZZY PID CONTROLLERS. Medium. [Електронний ресурс]. URL: https://medium.com/k-r-i-s-s/fuzzy-pid-controllers-26e1e1d800c5 (дата звернення: 22.03.2024).
58. Walker S. What is a fuzzy set?. Klu.ai. [Електронний ресурс]. URL: https://klu.ai/glossary/fuzzy-set (дата звернення: 22.03.2024).
59. Fuchs E. F., Masoum M. A. Power Quality in Power Systems, Electrical Machines, and Power-Electronic Drives. 3-тє вид. 2023.
60. Fuzzy Logic Based in Optimization Methods and Control Systems and its Applications / ред. A. Sadollah. InTech, 2018. URL: https://doi.org/10.5772/intechopen.73112 (дата звернення: 02.06.2024).
61. Mendel J. Fuzzy logic systems for engineering: a tutorial. Proceedings of the IEEE. 1995. Т. 83, № 3. С. 345–377. URL: https://doi.org/10.1109/5.364485.
62. Nesrine Ben Yahia, Henda Ben Ghezala, Narjès Bellamine Ben Saoud. Integrating fuzzy case-based reasoning and particle swarm optimization to support decision making. International Journal of Computer Science. 2012. Т. 9, № 3.
63. Bouguezzi S. Fuzzy Logic Explained. Baeldung CS. [Електронний ресурс]. URL: https://www.baeldung.com/cs/fuzzy-logic (дата звернення: 15.03.2024).
64. Mogharreban N., DiLalla L. F. Comparison of Defuzzification Techniques for Analysis of Non-interval Data. 2006 Annual Meeting of the North American Fuzzy Information Processing Society, м. Montreal, QC, Canada, 3–6 черв. 2006 р. 2006. URL: https://doi.org/10.1109/nafips.2006.365418 (дата звернення: 02.06.2024).

65. Defuzzification Methods- MATLAB & Simulink. MathWorks - Makers of MATLAB and Simulink - MATLAB & Simulink. [Електронний ресурс]. URL: https://www.mathworks.com/help/fuzzy/defuzzification-methods.html (дата звернення: 15.03.2024).
66. Bach F., Sutton R. S., Barto A. G. Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning series) second edition Edition. 2-ге вид. A Bradford Book, 2018.

ДОДАТКИ

Додаток А – Вигляд UI в разі успішного розгортання сервісів у Docker Swarm



Додаток Б – Панель Grafana для перегляду використання пам’яті на деякому з вузлів Docker Swarm