ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«АВТОМАТИЗОВАНА ІНТЕЛЕКТУАЛЬНА СИСТЕМА «VIZORADATE» ДЛЯ УНІВЕРСАЛЬНОГО АНАЛІЗУ В РЕЖИМІ РЕАЛЬНОГО ЧАСУ »

на здобуття освітнього ступеня магістр

за спеціальності 126 Інформаційні системи та технології

(код,найменування спеціальності)

освітньо-професійної програми Інформаційні системи та технології

(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Ольга ЗУБАР

(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:

здобувач вищої освіти

група ІСДМ-61 Ольга ЗУБАР

(ім'я, ПРІЗВИЩЕ)

Керівник Ольга ПОЛОНЕВИЧ

к.т.н. *(ім'я, ПРІЗВИЩЕ)*

Рецензент:

(ім'я, ПРІЗВИЩЕ)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти магістр
Спеціальність 126 Інформаційні системи та технології
Освітньо-професійна прорама Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ
_____ Каміла СТОРЧАК
“____” _____2025 року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Зубар Ользі Олегівні
(прізвище, ім’я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Автоматизована інтелектуальна система "VizoraDate" для універсального аналізу в режимі реального часу.
керівник кваліфікаційної роботи: Ольга ПОЛОНЕВИЧ к.т.н., доцент
(ім’я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від “ 30 ” жовтня 2025 р. № 467

2. Строк подання кваліфікаційної роботи «26» грудня 2025 р.

3. Вихідні дані кваліфікаційної роботи:
1. Системи аналізу даних у режимі реального часу.
2. Архітектури real-time аналітичних систем.
3. Методи обробки потокових даних.
4. Інтелектуальні компоненти систем аналізу даних у реальному часі.
5. Науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):

1. Дослідження особливостей потокових даних та принципів їх обробки у режимі реального часу.
2. Огляд архітектурних підходів побудови систем аналізу даних у реальному часі.
3. Аналіз результатів реалізації прототипу автоматизованої інтелектуальної системи «VizoraDate».

5.Перелік ілюстраційного матеріалу: *презентація*

6. Дата видачі завдання « 30 » жовтня 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Підбір технічної літератури	30.10.2025 – 05.11.2025	
2.	Дослідження особливостей потокових даних та real-time аналізу	06.11.2025 – 12.11.2025	
3.	Дослідження архітектурних підходів побудови систем аналізу даних у реальному часі	13.11.2025 – 18.11.2025	
4.	Результати реалізації прототипу автоматизованої інтелектуальної системи «VizoraDate»	19.11.2025 – 26.11.2025	
5.	Висновки по роботі	26.11.2025	
6.	Розробка демонстраційних матеріалів, доповідь.	27.11.2025	
7.	Оформлення магістерської роботи	28.11.2025	

Здобувач вищої освіти _____ Ольга ЗУБАР

(підпис) *(ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Ольга ПОЛОНЕВИЧ

(підпис) *(ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття ступня магістр: 81 стор., 9 рис., 2 табл., 29 джерел.

Мета роботи - розробити прототип інтелектуальної системи аналізу даних у реальному часі..

Об'єкт дослідження - процес аналітичної обробки даних у режимі, наближеному до реального часу.

Предмет дослідження - методи, алгоритми та архітектурні підходи до інтелектуального аналізу даних.

Короткий зміст роботи. У першому розділі магістерської роботи проаналізовано особливості потокових даних та методи їх обробки у режимі реального часу. Розглянуто алгоритми потокової обробки та сучасні інструменти real-time аналітики.

У другому розділі виконано аналіз існуючих архітектурних підходів, типів real-time систем та практик їх підтримки. Проаналізовано функціональні ролі систем оперативної аналітики.

У третьому розділі описано архітектуру та програмну реалізацію прототипу інтелектуальної системи «VizoraDate». Представлено реалізацію модулів завантаження, аналізу та інтелектуального профілювання даних, а також результати роботи системи.

КЛЮЧОВІ СЛОВА: REAL-TIME АНАЛІЗ ДАНИХ, ПОТОКОВІ ДАНІ, ІНТЕЛЕКТУАЛЬНА СИСТЕМА, ПРОФІЛЮВАННЯ ДАНИХ, МОДУЛЬНА АРХІТЕКТУРА, ШТУЧНИЙ ІНТЕЛЕКТ, АВТОМАТИЗОВАНИЙ АНАЛІЗ, ОПЕРАТИВНА АНАЛІТИКА.

ABSTRACT

The text part of the qualifying work for obtaining a bachelor's degree: 81 pp., 9 fig., 2 tables, 29 sources.

Purpose of the study – to develop a prototype of an intelligent real-time data analysis system.

Object of the study – the process of analytical data processing in a near real-time mode.

Subject of the study – methods, algorithms, and architectural approaches to intelligent data analysis.

Brief summary of the work.

In the first chapter of the master's thesis, the characteristics of streaming data and methods of their processing in real-time are analyzed. Stream processing algorithms and modern real-time analytics tools are considered.

In the second chapter, an analysis of existing architectural approaches, types of real-time systems, and practices of their maintenance is carried out. The functional roles of operational analytics systems are analyzed.

In the third chapter, the architecture and software implementation of the prototype of the intelligent system "VizoraDate" are described. The implementation of data ingestion, analysis, and intelligent data profiling modules, as well as the system operation results, are presented.

KEYWORDS: REAL-TIME DATA ANALYSIS, STREAMING DATA, INTELLIGENT SYSTEM, DATA PROFILING, MODULAR ARCHITECTURE, ARTIFICIAL INTELLIGENCE, AUTOMATED ANALYSIS, OPERATIONAL ANALYTICS.

ЗМІСТ

ВСТУП

Актуальність теми. У роботі з даними нині є одна помітна тенденція, про яку вже майже не сперечаються: інформація рідко надходить у готовому, статичному вигляді. Вона рухається, накопичується пошарово, приходить фрагментами, і цей темп змушує аналітичні системи працювати інакше, ніж ще кілька років тому. Багато завдань, які раніше можна було виконувати після повного збору даних, тепер потребують швидкої реакції. Тому частина традиційних інструментів просто не встигає — вони були створені під зовсім інші сценарії.

Водночас зростає кількість ситуацій, коли перш ніж запускати складні моделі або довгі конвеєри, потрібно буквально “на око”, але впевнено, оцінити стан даних. Побачити, що всередині: які типи полів, чи не порушена структура, де можуть бути дублікати, які значення виглядають підозріло. Такі речі не завжди очевидні, коли працюєш з великими наборами, і тим більше, коли дані зібрані з різних джерел. І ось тут виникає прогалина: потужні системи потокової аналітики є, але вони громіздкі, а прості програми для перегляду даних не дають достатньої глибини.

Ще один момент, який підсилює актуальність цієї теми, пов’язаний з практикою. У багатьох командах аналітики зазвичай беруть дані “як є”, і перший етап роботи — це не аналіз, а пошук того, що не так: неправильні формати дат, дивні символи, нерівномірні колонки. Ручна перевірка забирає багато часу, і, чесно кажучи, часто пропускаються дрібні, але важливі деталі. Автоматизована система, яка може швидко розібрати структуру файлу, відтворити його ключові особливості та навіть пояснити їх, значно скорочує цей підготовчий етап.

Окремо варто згадати про інтелектуальні компоненти. З появою більш доступних мовних моделей стало можливим додавати до аналізу не лише “сухі” показники, а й змістові пояснення. У багатьох випадках саме вони допомагають зрозуміти, що приховується за колонкою: умовний код події, фрагмент тексту, або, скажімо, поле, яке виглядає як дата, але насправді змішане з іншими форматами. Без автоматичної підказки це доводиться з’ясовувати вручну.

Тому є велика потреба в інструменті, який дозволяє швидко й без зайвих налаштувань отримувати перше уявлення про дані, особливо коли ці дані змінюються або надходять у великій кількості. І саме тому розроблення інтелектуальної системи по типу «VizoraDate», яка поєднує структурну діагностику, статистику та AI-пояснення, має чітку й очевидну актуальність.

розробити прототип інтелектуальної системи універсального аналізу даних у реальному часі, здатної виконувати структурне, статистичне та змістове профілювання даних із використанням модульної архітектури та елементів штучного інтелекту.*Мета роботи* -

Для досягнення мети було визначено такі завдання:

1. проаналізувати теоретичні основи потокової аналітики та інструменти, що застосовуються у системах real-time;
2. дослідити існуючі архітектурні підходи, моделі та практики побудови систем для обробки даних у реальному часі;
3. реалізувати прототип системи VizoraDate з модульною структурою та компонентами інтелектуального аналізу;

Об'єкт дослідження - процес аналітичної обробки даних у режимі, наближеному до реального часу.

Предмет дослідження - методи, алгоритми та інструменти інтелектуального профілювання даних, а також архітектурні підходи до побудови модульних систем первинної аналітики.

Методи дослідження. У роботі застосовано методи теоретичного аналізу, системного підходу до проектування програмних архітектур, елементи моделювання потоків обробки даних, методи інтелектуального аналізу та сучасні техніки роботи з даними (профілювання, структурна валідація, автоматична типізація). На прикладному рівні використано інструменти Polars, Pandas, Streamlit, Dash та API-моделі LLM.

Наукова новизна одержаних результатів. Новизна роботи полягає у поєднанні декількох підходів у межах одного прототипу: інтерактивної аналітики,

автоматизованої структурної діагностики, статистичного аналізу та генеративних AI-пояснень. У межах VizoraDate реалізовано модульну архітектуру, яка допускає сценарії напівреального часу, а інтелектуальні компоненти здатні проводити семантичне інтерпретування колонок на основі реальних прикладів даних. Таким чином створено концептуально нову основу для гібридного Data Profiling — поєднання класичної прикладної аналітики та контекстних AI-описів.

Практична значущість одержаних результатів. Розроблений прототип може використовуватися як інструмент первинного аналізу даних у бізнес-аналітиці, Data Quality, побудові ETL/ELT-процесів та під час роботи з малоформалізованими наборами даних. Система дозволяє швидко оцінити структуру датасету, знайти проблемні поля, виявити дублікати, переглянути характер значень, а також отримати автоматичні змістові коментарі. Інструмент може бути інтегрований у робочі процеси аналітиків, розробників, інженерів даних чи команд, які працюють з даними спорадично і не мають важких платформ на рівні enterprise.

Апробація результатів магістерської роботи. Основні положення і результати магістерської роботи публікувались на науково практичних конференціях, що проходили на базі Державного університету інформаційно-комунікаційних технологій.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ТА МЕТОДОЛОГІЧНІ ОСНОВИ АНАЛІЗУ ДАНИХ В РЕАЛЬНОМУ ЧАСІ

Аналіз даних у реальному часі давно перестав бути вузьким напрямом для окремих компаній, які працюють з високими навантаженнями. Він поступово став частиною звичайних інформаційних систем, бо дані тепер з’являються не так, як колись - вони приходять безперервними хвилями, інколи нерівними, інколи надто швидкими, а часом навпаки, з короткими провалами. І все це разом вимагає не стільки швидкості, скільки здатності реагувати без затримки. Певною мірою це навіть змінює уявлення про сам процес обробки: вже не можна просто зібрати інформацію, відкласти її “на потім” і спокійно запускати аналітику, як це було колись у класичних системах.

Потоки даних взагалі рідко поводяться стабільно. Вони можуть бути нерівними, стрибати по обсягу, йти із запізненням, розриватися або приходити від різних джерел одночасно. І тут уже важко говорити про “ідеальні” умови обробки, бо сама природа потоків не надто передбачувана. У багатьох випадках навіть структура подій може змінюватися раптово: сьогодні джерело надсилає певний формат, а завтра з'являється додаткове поле чи змінюється тип одного з параметрів. Це вимагає від системи гнучкості, яка у звичайних підходах не була настільки необхідною.

Паралельно з цим зростає інтерес до моделей, що можуть працювати у реальному часі. Раніше машинне навчання було орієнтоване здебільшого на історичні дані та стабільні вибірки. Потоки ж вносять свою логіку: розподіл ознак змінюється, з'являються нові патерни, а моделі повинні реагувати швидко. Іноді моделі, що мають працювати на потоці, поводяться не так, як у статичному середовищі, і доводиться комбінувати різні підходи: щось реагує швидше, щось навпаки потребує більше часу. Це створює дивну ситуацію, коли не сам алгоритм визначає робочий темп, а вимушені обмеження системи. Через це інженеру доволі

часто доводиться переглядати вже налагоджені процеси і, по суті, переосмислювати спосіб, у який дані “зустрічаються” з моделлю в момент обробки.

1.1 Особливості потокових даних

Потокові дані здаються простими лише на перший погляд: подія за подією, якійсь нескінченний «рух» від джерела до системи. Але коли розібратися, стає зрозуміло, що саме їхня поведінка і створює всі труднощі в аналізі. Вони не схожі ні на класичні вибірки, ні на журнальні записи, ні навіть на live-реплікації баз. Потік буквально живе власним життям, і це життя не підкоряється правилам «чистих» наборів.

Однією з основних властивостей є безперервність. Потік не має чіткого старту чи моменту, коли можна зупинитися та видихнути. Йому байдуже, що в системі зараз високе навантаження або що потрібно обробити попередні записи. Події не зупиняються. Вони надходять і вдень, і вночі, та ще й нерівномірно. З боку, можливо, здається, що потік працює стабільно, але якщо подивитися ближче, то помітно, що темп постійно змінюється. Була секундна тиша, а потім - десятки або сотні повідомлень, без пояснень. Усе це створює додаткову напругу, бо система мусить реагувати не на абстрактні вимоги, а на реальні коливання трафіку, які важко передбачити.

Безперервність пов'язана і з тим, що дані не накопичуються в одному місці. Їхня природа - бути в русі. Це зовсім інший підхід до аналітики, бо в традиційних системах завжди є якась “точка збереження”, до якої можна повернутися. У потоках ви отримуєте лише фрагмент у той момент, коли він пролітає повз. Якщо система його не зловила або не зберегла -шанс втрачено. Тому безперервність автоматично породжує іншу важливу особливість - обмеженість доступу.

Поточні події доступні лише тоді, коли вони надходять. Раніше всі дані лежали в таблиці, умовно кажучи, під рукою. Тут такого немає. Потік має дуже коротку "пам'ять", і якщо розробник хоче щось зберегти для подальшого аналізу,

він повинен передбачити це заздалегідь: побудувати окремий буфер, кеш або вставити етап проміжного збереження. Інакше важлива інформація пройде як вода між пальців.

Висока швидкість, яка характерна для потоків, накладає ще більше вимог. І тут не йдеться про формальні тисячі подій за секунду. Йдеться про зміну темпу. Дві секунди все відносно нормально, а потім різкий стрибок. Так трапляється навіть у нескладних системах, не кажучи вже про фінансові транзакції чи телекомунікації. Потоки можуть бути хаотичними, і ця хаотичність - не помилка, а нормальна властивість. Через це доводиться враховувати, що швидкість надходження подій непередбачувана.

Звідси походить ще одна риса - нескінченність потоку. Це важко усвідомити інтуїтивно, бо ми звикли до вибірок, які мають початок і кінець. У потоках немає ніякого “останніх даних”. Якщо система працює постійно, потік не закінчується взагалі. Він триває, поки існує джерело. Як наслідок, аналітика постійно працює на частковій картині. Ви не бачите весь набір. Ви бачите лише те, що доступно зараз і в найближчий проміжок часу. Будь-яка «завершеність» потоку -штучна. Вона створюється вже на рівні обробки: наприклад, 10 секунд, 5 хвилин або година - це не природні межі, а технічна необхідність.

Оскільки потік нескінченний, його не можна «перечитати» разом з усіма подіями, як роблять у статистичному аналізі. Тому аналітика ґрунтується на обробці в реальному часі - тобто на тому, щоб приймати рішення за фактом надходження даних. Інших варіантів у потоків просто немає. Якщо система пропустить потрібний момент, подія вже не буде актуальною. Її сенс часто повністю прив’язаний до моменту. Наприклад, сигнал про відхилення з датчика має вагу тільки в той час, коли це відхилення реально відбулося, а не через хвилину.

Тут виникає ще один важливий аспект: залежність між подіями, яку потрібно "збирати" на ходу. Потокові дані завжди фрагментарні. Кожна подія показує лише частину того, що відбувається. У традиційних наборах ми маємо всю інформацію одразу. У потоках доводиться поступово накопичувати стан: останні значення,

останні дії користувача, попередні сигнали обладнання. Без цього аналіз буде уривчастим і втратить контекст.

І ще одна важлива властивість - неможливість виконувати довільні запити, як це роблять у звичайних базах. Потік не дозволяє сказати: “покажіть мені, що було годину тому”. Він не зберігає історію, якщо це не реалізовано окремим компонентом. Тому модель взаємодії в потоках інакша: тут не дані чекають на запит, а запити повинні бути готові в той момент, коли з'являються дані. Можна сказати, що аналітика працює «на зустріч» потоку, а не навпаки.

Через те, що потоки надходять із різних джерел, структура даних часто нестабільна. Це не помилка - це природний наслідок змін у джерелах. Сьогодні подія має один набір параметрів, завтра - інший. І система мусить це витримувати, не завалюючи весь процес. Важливо усвідомлювати, що нестабільність структури - не виняток, а норма для потокових систем. Те саме стосується і якості даних: шум, дублікати, пропуски - все це стандартні явища.

Не менш важливий момент - час. У потоках завжди є два різних типи часу: час, коли подія сталася, і час, коли вона потрапила в систему. Вони рідко збігаються. Навіть короткі затримки можуть змістити картину. І якщо система обробляє події без урахування цього зсуву, результат може бути неправильним.

У підсумку виходить, що потокові дані поєднують у собі цілу групу властивостей, які роблять їх складними для традиційної аналітики:

- вони не мають чіткої межі
- не повторюються
- надходять нерівномірно
- структурно нестабільні
- зникають одразу після появи
- не дають цілісної картини без збереження стану

І саме через це потребують спеціальних алгоритмів, архітектур і підходів до аналізу.

1.2. Методи та алгоритми обробки потокових даних

У роботі з потоками немає звичної послідовності кроків, де спочатку збирають дані, потім усе готують, а вже після цього запускають обчислення. Тут подія з’являється раптово, і система повинна одразу ж щось із нею робити, не відкладаючи на потім. Через це підхід до алгоритмів виходить більш оперативним: багато операцій виконуються на льоту, без проміжних етапів, які у звичайних наборах даних займають основну частину часу.. Але спрощеність у цьому контексті не означає примітивність. Навпаки: багато з цих алгоритмів доволі складні, просто їхня складність не в тому, як вони працюють "в середині", а в тому, як вони уживаються з постійним рухом подій.

Однією з центральних концепцій є віконування (windowing). У потоках використовують різні типи вікон, і кожне працює по-своєму. Є такі, що відкриваються через фіксовані проміжки часу; є інші, які поступово зсуваються і постійно захоплюють нові події; є й такі, що формуються залежно від поведінки джерела або довшої паузи між подіями. Ці варіанти відрізняються за логікою, і вибір залежить від конкретної задачі, бо в одному випадку важливі рівні інтервали, а в іншому - природні "сесії" дій. І кожен тип вирішує свою задачу. Наприклад, фіксовані вікна добре працюють там, де важливі регулярні інтервали (як у моніторингу мережі). Ковзні дозволяють бачити зміни якраз у момент, коли вони відбуваються. А сесійні корисні, коли події групуються не за часом, а за поведінкою певного джерела.

Існує також обробка в порядку надходження (out-of-order handling). Тут важливо не тільки прийняти подію, а й розв’язати питання: чи повинна вона вплинути на вже сформований результат? Залежно від логіки, запізніла подія може або коригувати результат, або бути відхиленою. В деяких системах допускають невеликі уточнення, які «доганяють» попередні статистики; в інших корекції заборонені, бо це порушує консистентність.

Наступна важлива група методів - це операції над потоком, тобто трансформації, які виконуються в реальному часі. Найпростіші - фільтрація, проєкція (відбір окремих полів), нормалізація. Але є й складніші: агрегації, об'єднання кількох потоків, злиття з довідковими даними, обчислення ковзних статистик. Усі ці операції повинні виконуватися швидко, бо кожна мілісекунда затримки накопичується в загальній обробці.

Саме через потребу працювати дуже швидко виникла потреба в stateful-алгоритмах, тобто таких, що зберігають певний стан між подіями. Це зовсім інша історія, ніж звичні цикли у пакетному аналізі. Стан у потоковій обробці може бути крихким, бо постійний рух подій та можливі збої створюють ризик його втрати. Тому stateful-алгоритми спираються на механізми резервування, періодичні збереження відміток (checkpointing) та відновлення робочого стану. Це забезпечує стабільність навіть під час перебоїв або зміни топології системи.

До таких алгоритмів належать, наприклад, підрахунок частоти (frequency counting), виявлення повторюваних шаблонів, оновлення ковзних середніх. З боку здається, що це просто статистика, але в потоках ці обчислення мають робити «на ходу», без доступу до всієї історії. Часто застосовують техніки наближення, наприклад, фільтри Блума чи HyperLogLog, бо точні обчислення можуть вимагати забагато пам’яті.

У потокових системах існує окремий пласт методів, який пов’язаний із тим, як система трактує послідовність подій. Бувають ситуації, коли важливим є не окрема подія, а те, що відбувається перед нею або після неї. Для цього застосовують підходи, які дозволяють «зловити» закономірність у динаміці. У різних середовищах їх реалізують по-своєму, але загальний принцип один: система намагається зрозуміти, чи формує низка подій певний зміст. Наприклад, у фінансових сервісах це може бути серія підозрілих транзакцій, у технічних системах - сукупність сигналів, що натякають на збої обладнання. Такі підходи не будують складних моделей, але дають можливість виявляти моменти, коли ситуація відхиляється від звичної.

Інший складний аспект - робота з потоками, що надходять із різних джерел. На практиці дані рідко бувають однорідними. Один сервіс надсилає події з однією частотою, інший - із затримкою, третій - у своєму форматі. І коли ці події потрібно об’єднати, виникає чимало нюансів. У статичних даних це не проблема: вся інформація зберігається й доступна одночасно. У потоках усе інакше - частина даних приходить у момент обробки, а частина ще у дорозі. Тому доводиться підтримувати проміжний стан, своєрідну тимчасову пам’ять, де зберігаються останні значення або фрагменти записів. Ці структури постійно змінюються, бо потік не стоїть на місці, і система підлаштовується під ритм кожного джерела.

Ще одна група методів пов’язана зі зменшенням обсягу потоків. На практиці обробити абсолютно всі події часто неможливо - потік може бути надто великим. Тому використовують різні техніки вибіркового відбору. Ідея проста: замість повної історії зберігається певна частина, яка приблизно відображає загальну картину. Є методи, де вибірка підтримується постійного розміру й поступово оновлюється - наприклад, коли нові події можуть замінювати старі. Так можна контролювати обсяг пам’яті та не втрачати загальний характер потоку.

У деяких випадках важливим стає не сам факт події, а її вплив на тенденцію. Деякі системи намагаються відстежити, як змінюється поведінка потоку, і тому використовують підходи, що працюють із поступовими оновленнями значень. Так моделі не перебудовуються заново, а лише коригують свої параметри за кожною новою частиною даних. Такий спосіб природний для потоків, бо повної історії немає, і доводиться працювати з тим, що надходить у момент.

Загалом, методи обробки потоку можна умовно поділити на декілька напрямків:

- ті, що забезпечують структуризацію потоку (віконування, впорядкування за часом)
- ті, що перетворюють події (фільтрація, агрегування)
- ті, що зберігають стан (stateful processing)
- ті, що працюють зі сценаріями (CEP)

- ті, що забезпечують масштабування і стійкість (checkpointing, відновлення, розподілена обробка)

Кожен напрям має власний набір алгоритмів, і вибір залежить від задачі. У потоках немає універсальних методів, бо кожен застосунок висуває свої вимоги до швидкості, точності, стабільності та ресурсів.

Потокові методи доводиться підлаштовувати під середовище, у якому дані надходять нерівно, інколи з пропусками, інколи зі зміщенням у часі. Алгоритми, що тут працюють, мають не лише виконати розрахунок, а й утримати загальний стан обробки, щоб події між собою узгоджувались. Це включає і реагування на різкі стрибки навантаження, і роботу з тими подіями, що приходять пізніше, ніж очікується.Саме через це потокова аналітика виглядає так, ніби рух і обчислення відбуваються одночасно - бо так воно і є.

1.3 Інструменти для аналізу даних у реальному часі

Інструменти, які використовують для аналізу даних у реальному часі, формують досить складну екосистему, і щоб зрозуміти, чому вона виглядає саме так, потрібно трохи зануритися в логіку їхнього розвитку. Більшість із них не створювалися як «пакет» для потоків. Кожен з'являвся для своїх задач, але з часом ці задачі почали з'єднуватися. Компанії, які працювали з великою кількістю подій, відчули, що їм потрібно не просто передавати повідомлення, а ще й зберігати їх, обробляти в той самий момент і передавати результат далі. Усе це привело до появи набору інструментів, що сьогодні фактично стали стандартом.

Почати варто з Apache Kafka, тому що саме вона, скоріше за все, буде присутня в будь-якій системі, що працює з потоком даних. Kafka побудована не за принципом черги, як могло б здатися з назви, а за принципом розподіленого журналу. Повідомлення не зникнуть одразу після того, як їх хтось прочитав. Вони лежать у логах, кожному присвоюється власний зсув, і споживач може читати їх у своєму темпі. Якщо система різко зростає, Kafka просто додає нові розділи або

брокери - її горизонтальне масштабування давно стало однією з причин, чому вона витримує навантаження глобальних аудиторій.

Ще важливішою є здатність Kafka повторно програвати події. Змінилася логіка обробки? Потрібно обчислити нову статистику з того, що вже було? Просто зрушують курсор читання і повторно проходять частину журналу. Така особливість здається дрібницею, але вона фактично дозволяє будувати системи, які живуть роками і не "ламаються" при зміні бізнесу. Дані в Kafka - не одноразова штука, а матеріал, який можна перегравати. Саме це робить Kafka популярною не лише у стримінгу, а й у логуванні, аналітиці клієнтської поведінки, телеметрії.

Рис.1.1. Логотип Apache Kafka

RabbitMQ, хоч і не позиціонується як конкурент Kafka, вирішує інші завдання. Він працює у сценаріях, де потрібно керувати тим, куди йде подія. Тут важлива маршрутизація: різні підсистеми можуть отримувати різні копії одного й того ж повідомлення. У деяких ситуаціях це незамінно. Наприклад, одна подія від застосунку повинна піти в аналітичний модуль, друга - у службу сповіщень, третя - у внутрішній журнал аудиту. RabbitMQ дає таку свободу за рахунок обмінів route-типу, fanout, direct тощо. Для систем, що побудовані на бізнес-процесах, а не на масивних потоках подій, це зручно, і тому RabbitMQ досі має величезну аудиторію.

Рис.1.2. Логотип RabbitMQ

NATS - інструмент іншого класу. Це мінімалістична платформа, зібрана так, щоб усі операції були максимально легкими та швидкими. Те, що NATS упорався би з тією ж логікою, що Kafka чи RabbitMQ, не значить, що він на це орієнтований. Його використовують там, де затримка вимірюється не мілісекундами, а мікросекундами, і де головне - миттєва передача повідомлень між сервісами. Часто це середовища FinTech, ринки торгівлі, внутрішні шини даних у високопродуктивних мікросервісах. У NATS немає складної логіки зберігання - події не призначені для довгострокового життя - але саме завдяки цій простоті він настільки швидкий.

Рис.1.3. Логотип NATS

Redpanda з’явилася вже пізніше і одразу отримала увагу через дві речі: сумісність із Kafka та відсутність JVM. Вона написана на C++, тому використовує ресурси ефективніше. Для компаній, які вже мають екосистему Kafka, перехід майже непомітний: ті самі клієнти, той самий протокол, але менше витрат на інфраструктуру й кращі показники швидкості.

У Redpanda інший дизайн: система розрахована на мінімум конфігурації, мінімум обслуговування, максимальну продуктивність. У деяких середовищах, де Kafka дає надто велику затримку через GC-паузи або має зайве навантаження на процесор, Redpanda може показувати кращі результати.

Рис.1.4 Логотип Redpanda

Як тільки дані доходять до платформи доставки, у гру вступають фреймворки обробки.

Apache Flink - один із найбільш технічно складних серед них. Він працює не з мікробатчами, а обробляє події буквально по одній. Найважливішою його особливістю є робота з часом події. Flink не просто бере часову позначку та сортує події; він робить дуже хитру річ - підтримує уявлення про "актуальний" час потоку на основі watermark, тобто умовних міток, за якими система визначає, чи можна вже рахувати певну статистику, чи потрібно ще почекати можливих затриманих подій. Це складно, але дає майже ідеальну точність у сценаріях, де події нерівні, приходять із запізненням або надходять паралельно з різних серверів. Flink ще й тримає state - тобто зберігає локальні дані між подіями. І робить це розподілено, з можливістю відновити все після збою.

Це робить його придатним для довготривалих робочих навантажень без втрати актуальності даних.

Рис.1.5. Логотиа Apache Flink

Spark Streaming працює за іншою моделлю: мікробатчі. Дані групуються маленькими частинами й обробляються разом. Це створює невелику затримку, але зате вся система виходить передбачуваною, стабільною та добре інтегрованою із загальною екосистемою Spark. Там, де потрібен потік і batch в одному середовищі, Spark Streaming - майже ідеальне рішення.

Особливо у випадках, де компанія вже має Spark і не хоче додавати нові складні фреймворки.

Рис.1.6. Логотип Spark Streaming

Apache Storm, хоча зараз здається більш “класичним”, досі зустрічається. Його гнучкість та можливість будувати довільні топології обробки робить його актуальним там, де інструменти нового покоління не дають достатньої кастомізації. Storm часто використовують для подій, які мають дуже конкретну логіку: маршрутизація між кількома етапами, нетривалі обчислення, збирання сигналів із розподілених компонентів.

Рис.1.7. Логотип Apache Storm

Beam та Dataflow взагалі намагаються відділити розробника від складності виконання. Beam створений як універсальна модель, де логіка обробки описується один раз, а виконання - обирається пізніше. Ця модель гнучка: їй однаково, де працювати - на Spark, на Flink чи на Dataflow.

Рис.1.8. Логотип Beam

Dataflow додає автоматичне масштабування: система сама розподіляє навантаження, сама розширюється, коли потік збільшується, і скорочується, коли

стає тихіше. Реально це дуже економить час, тому що налаштування кластерів завжди потребувало досвіду.

Рис.1.9. Логотип Dataflow

Хмарні інструменти, такі як AWS Kinesis, Google Pub/Sub або Azure Event Hubs, часто беруть на себе роль брокера, одночасно спрощуючи адміністрування. Вони з’явилися як відповідь на запит бізнесу: не кожній команді потрібен власний кластер Kafka чи Flink. Kinesis цікавий тим, що поєднує в собі кілька сервісів: один відповідає за стримінг, інший дозволяє писати SQL над потоком, а третій перекидає дані в сховища типу S3. Pub/Sub орієнтований на масштаби - його можна знайти в проєктах, де користувачів мільйони, а подій - мільярди. Azure Event Hubs має подібну модель, але інтегрується з аналітикою через Stream Analytics - сервіс, який дозволяє робити вікна й агрегації майже так само легко, як у SQL.

Сховища, оптимізовані для потоків, теж частина екосистеми. HBase - це база над HDFS, яка дозволяє писати величезну кількість записів постійно і без пауз. Вона працює як щось середнє між кешем і довготривалим сховищем. Для аналітики часто використовують ClickHouse - дуже швидку колоночну базу. У потокових системах його застосовують для запису проміжних агрегатів або оновлення статистики майже в реальному часі. Він токсично швидкий у читанні: якщо потрібно побудувати звіт на мільярдах рядків, то ClickHouse зробить це за секунди.

Time-series бази, такі як InfluxDB або TimescaleDB, створені для телеметрії. Вони оптимізують часові ряди, вміють стискати дані, підтримують великі обсяги вставки та зберігають дані з можливістю працювати через часові вікна, що особливо зручно для сенсорних систем.

Інструменти другого рівня - Redis, Zookeeper, etcd, Prometheus - забезпечують роботу всієї екосистеми. Redis - це ультрашвидкий кеш, який може тримати проміжний стан алгоритмів або використовуватися як сховище коротких часових вікон. Zookeeper та etcd керують конфігураціями, координують вузли, стежать за станами кластерів. Prometheus і Grafana дозволяють спостерігати за системою в реальному часі, що є критичним, бо будь-який збій може створити помітні затримки.

Усі ці рішення не конкурують між собою - вони взаємодіють. Типова архітектура виглядає як послідовність: потік надходить у Kafka або Pub/Sub, потім Flink чи Spark його обробляє, а результати записують у ClickHouse, HBase або спеціалізоване сховище. Моніторинг стежить за навантаженням, Redis тримає стан, а зовнішні компоненти використовують дані для аналітики або рекомендаційних систем. Саме ця взаємодія створює реальні системи, здатні реагувати на події миттєво.

РОЗДІЛ 2 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ, АРХІТЕКТУР ТА ПРАКТИК ПОБУДОВИ СИСТЕМ ДЛЯ ОБРОБКИ ДАНИХ У РЕАЛЬНОМУ ЧАСІ

У середовищі, де системи обробки даних ростуть швидше, ніж будь-які структури, що мали б їх утримувати в певних межах, усе здається рухливим. Часто навіть занадто рухливим. Реальні процеси в організаціях, які працюють із потоками, не схожі на рівні схеми з підручників. Буває так, що одна й та сама компанія користується одразу кількома підходами, ніби вони запроваджувалися в різні періоди, і ніхто до кінця не встиг їх об’єднати. Звідси й походить цікаве відчуття випадковості, яке супроводжує дослідження сучасних real-time систем: ти бачиш не модель, а своєрідний “зріз” еволюції, нерівномірної, трохи хаотичної, але живої. Через це реальні системи нагадують не витончену конструкцію, а механізм, який постійно ремонтують у русі. Іноді потоки поводяться як живий організм. Справді: то вони розширюються так, що ingestion шар ледве витримує; то з’являються несподівані повтори подій; то кількість користувацьких дій раптово злітає вгору і тримається на найвищій позначці довше, ніж очікувалося. Такі стрибки перевіряють архітектуру на міцність краще, ніж будь-яке тестування. А з ними приходять інші речі: дрібні мікрозатримки, проблеми з чергами, непомітні деградації, які проявляються лише під великим навантаженням. І тоді стає зрозуміло, що проєктувати real-time системи без досвіду реальної експлуатації майже неможливо.

Потокові системи не бувають завершеними. У них немає фінальної точки, після якої можна сказати, що робота виконана. Вони або змінюються, або старіють. І тому аналіз існуючих рішень завжди трохи запізнюється: поки ти описуєш одну модель, хтось уже тестує іншу, бо їхня система, з її дуже конкретними особливостями, підказала інше рішення. Саме через це реальний ландшафт real-time аналітики більше схожий на потокову карту, ніж на ідеальний план.

Усе це робить тему складною, але водночас дуже показовою. яка дає змогу побачити, як технології адаптуються до непередбачуваності, як інженери шукають рівновагу між точністю й швидкістю, а архітектурні моделі стають лише відправною точкою для практичної роботи.

2.1. Архітектурні моделі побудови систем аналізу даних у реальному часі

У реальних потокових системах архітектурні моделі завжди формуються трохи стихійно, навіть якщо зовні здається, що все побудовано строго за каноном. Немає такої команди, де архітектура з’являється одразу в готовому вигляді. Це завжди ланцюжок рішень, часто ситуативних, які згодом оформлюються у певний патерн. Через це й важко описати архітектури real-time систем у вигляді чіткої формули. Вони живі. Постійно рухаються. Іноді хаотично. Але саме в цій динаміці й лежить їхня сила.Подієво-орієнтована архітектура, насправді виростає з досвіду роботи з даними, які ніяк не хочуть чекати планових обробок. Подія має властивість виникати саме тоді, коли їй заманеться. Користувач натиснув кнопку, сенсор послав імпульс, система зафіксувала транзакцію. І ось уже подія летить у конвеєр, не питаючи дозволу. Тому EDA стала не стільки технологічним вибором, скільки природною відповіддю на характер потоків.

При цьому в EDA немає нічого магічного. Це радше спроба звести складність до керованого вигляду. Якщо кожна подія має чітку структуру й автономне значення, з нею можна працювати незалежно від того, що відбувається навколо. Але в реальності події рідко бувають ідеальними. Частина з них приходить без ключових полів, частина дублюється, частина надходить із запізненням. І тоді архітектура починає обростати додатковими шарами: фільтри, ретрай-логіка, схеми узгодження, компенсуючі механізми. Усе це ніби не прописано в EDA, але без цього EDA в промисловому середовищі просто не працює.

Streaming-first підхід, що набув популярності останніми роками, виріс з усвідомлення, що дані перестали бути чимось, що “надходить раз на добу”. Вони

течуть безперервно, і на них не можна реагувати одними лише партіонними процесами. Компанії, що намагалися поєднувати “старі” підходи з новими потоками, швидко зрозуміли, що така гібридність створює більше проблем, ніж вирішує. Простіше одразу будувати інфраструктуру навколо потоку, а вже потім (за потреби) додавати партіонні шматки.

Якщо ж порівнювати micro-batch і true streaming, то це, мабуть, найпоширеніша тема суперечок. Іноді ці режими подають як взаємовиключні, але це не зовсім так. У реальних компаніях вони співіснують абсолютно природно. Частина завдань не потребує обробки кожної окремої події -достатньо отримати порцію даних за хвилину. Так працюють великі агрегати, які об’єднують тисячі записів. Інші завдання, особливо пов’язані з реактивністю, не можуть чекати навіть цих кількох секунд. Наприклад, antifraud-сценарії: якщо транзакція підозріла, необхідно відреагувати негайно.

Через це micro-batch і true streaming беруть кожен своє. І добре, коли архітектура дозволяє перемикатися між ними без глобального рефакторингу. Бувають команди, які починали з micro-batch, а потім додали true streaming лише для критичних компонентів. Є й зворотні кейси -починали з потоків, а потім додавали micro-batch там, де було потрібно зменшити навантаження й отримувати більш точні результати.

Архітектури Lambda і Kappa -продовження цієї дискусії, але на рівні всієї системи. Lambda з її двома контурами здається водночас сильною і громіздкою. На старті все виглядає зрозуміло: швидкий контур дає оперативність, повільний гарантує точність. Але варто системі вирости хоча б до кількох десятків потоків -і підтримувати два паралельні шляхи обробки стає викликом. Змінюєш логику в одному -мусиш змінити й в іншому.

Kappa вирішила цю проблему одним махом: тільки один контур обробки, і всю історію можна відтворити через replay. Добре працює, коли є надійне логування і рівень обробки не падає. Утім, практики кажуть, що Kappa працює чудово, поки потік стабільний. Але якщо в системі є “дірки”, або події приходять

не в тому порядку, або черги трохи плавають -Kappa починає давати збої. Тому реальні компанії часто не дотримуються жодної моделі в чистому вигляді. Вони беруть частину з Lambda, частину з Kappa і будують щось адаптоване під власний характер даних.

Таблиця 2.1.

Порівняльна характеристика архітектурних моделей систем аналізу даних у реальному часі

Архітектурна модель	Базова ідея	Ключові переваги	Основні обмеження	Типові сценарії застосування
Подієво-орієнтована архітектура (EDA)	Обробка системи як сукупності незалежних подій	Висока реактивність, слабке зв'язування компонентів, гнучкість масштабування	Чутливість до неякісних подій, складність обробки запізнілих та дубльованих повідомлень	Транзакційні системи, IoT, користувацькі дії
Streaming-first	Потік як базовий примітив усієї системи	Природна робота з безперервними даними, мінімальна затримка	Високі вимоги до інфраструктури та моніторингу	Аналітика в реальному часі, поведінкові системи
Micro-batch processing	Обробка даних невеликими часовими порціями	Краще керування ресурсами, стабільність	Затримки між подією та результатом	Агрегації, звітність з малою критичністю до latency
True streaming	Обробка кожної події окремо	Мінімальна затримка, висока реактивність	Складність забезпечення консистентності	Antifraud, real-time алерти
Lambda-архітектура	Подвійний контур: швидкий + точний	Компроміс між швидкістю та точністю	Дублювання логіки, складність підтримки	Великі enterprise-системи

Kappa-архітектура	Єдиний потоковий контур з replay	Спрощення архітектури, менше коду	Чутливість до якості та порядку подій	Потокові системи зі стабільним логом

Processing layer -це серце системи, але інколи саме тут виникає найбільше організаційних труднощів. Наприклад, коли кілька команд одночасно працюють із потоком і кожна змінює свій фрагмент логіки. Або коли доводиться підтримувати одночасно кілька версій обробки, бо є старі події, які ще не пройшли через конвеєр. Це породжує дивні конфігурації, коли один і той самий pipeline існує в двох чи трьох варіантах залежно від часу, коли була згенерована подія.

Serving layer також має свої сюрпризи. Здавалося б, це просто шар, який зберігає результати й надає доступ до них. Але на практиці serving часто стає вузьким місцем, особливо якщо результати мають бути доступними майже миттєво. Або якщо вони потребують складної індексації. Або якщо є жорсткі SLO. І кожен бізнес-домен висуває свої вимоги, інколи навіть суперечливі між собою.

2.2. Типи існуючих real-time систем та їх функціональні ролі

Системи оперативної аналітики зазвичай виглядають не найефектніше, зате вони живуть найближче до щоденної рутини. На них дивляться менеджери, SRE, іноді навіть люди без технічного бекграунду, і хочуть побачити одну просту річ: що відбувається прямо зараз. Не через годину, не після нічного перерахунку. Зараз. Ці системи не обіцяють ідеальної точності до останнього десятого знаку, але дають картинку динаміки. Вони показують, що щось "повзе вниз", щось раптово зростає, а щось поводиться дивно, хоча формально всі пороги ще не порушені.

У багатьох компаніях системи оперативної аналітики стають першим кроком до більш глибокої real-time екосистеми. Спочатку з'являються панелі з "теперішніми" метриками, потім виникає вимога drill-down - пройти від загального показника до конкретних потоків, сервісів, клієнтів. Далі з'являються прості алерти, які запускають ланцюжок дій. І непомітно для себе організація переходить

від "подивитись, що там зараз", до "автоматично реагувати, коли щось йде не так". На цьому етапі системи оперативної аналітики вже перестають бути просто візуалізацією і починають впливати на процеси.

Зовсім інший темперамент у транзакційно-потокових систем. Тут реальний час звучить як жорстка вимога, а не як nice-to-have. Уявімо, що є фінансова операція, яка проходить через класичну OLTP базу, але паралельно для неї запускається потік подій, що перевіряється antifraud логікою. Якщо потік відстав на секунду, то рішення може бути вже пізнім. Якщо він спрацював двічі - можна заблокувати легальну активність. І це вже не лише технічна помилка, а репутаційний ризик.

У таких системах дуже тонко налаштований баланс між консистентністю і латентністю. Класичний підхід "зробимо все строго послідовно" тут не працює - буде занадто повільно. Повна асинхронність також небезпечна, бо створює divergence між тим, що записано у транзакційному сховищі, і тим, що побачили потокові компоненти. Тому команди будують гібридні схеми: частина подій обробляється прямо "на транзакції", частина - через стріми з дуже малими буферами. Звідси відома складність з ідемпотентністю, повторним програванням, менеджментом offset-ів. Це все не красиві теоретичні проблеми, а щоденні дрібні битви за те, щоб система не зламала власну модель даних.

Вторинний ефект таких систем у тому, що вони змушують по-іншому мислити розробників прикладної логіки. Кожен новий бізнес-правило автоматично стає частиною потокового конвеєра. Не можна просто "дописати ще одну перевірку" - необхідно врахувати, як вона вплине на латентність, на порядок операцій, на можливість відкату. У певний момент команда починає відчувати, що працює не з окремими запитами, а з живим потоком, який має свій ритм. І це вже інший рівень відповідальності.

Поруч із цим живе свій світ платформ обробки користувацьких подій - clickstream і behavioural analytics. Тут дані поводяться майже як люди, від яких вони походять. У будні вдень один ритм, уночі інший, у вихідні третій, під час акцій -

ще якийсь окремий. Навіть незначна зміна в інтерфейсі продукту може повністю змінити картину потоків: люди почнуть натискати інші кнопки, іти іншим маршрутом, і система має це "побачити" не через місяць, а майже одразу.

Для таких платформ важлива не тільки швидкість, а й здатність розрізняти "звичну" поведінку і щось нетипове. Наприклад, у користувача раптом з'являється серія дуже швидких, майже автоматичних кліків - це може бути бот. Або навпаки, він довго зависає на одному екрані - можливо, там щось незрозуміло. Behavioral analytics намагається зібрати з цих фрагментів щось схоже на історію: як користувач зайшов, що спробував, що його зупинило. І все це в режимі, близькому до реального часу, щоб продуктова команда могла експериментувати.

Архітектурно такі системи часто відрізняються від "класичної" телеметрії тим, що тут більше високорівневих подій і менше низькорівневих метрик. Подія "користувач оформив замовлення" важить більше, ніж сотня дрібних кліків, але й цінується дорожче. Тому ці системи зазвичай тримають два шари: сирий clickstream, який можна переосмислити ще раз, і агреговані behavioral події, які вже використовуються для A/B тестів, персоналізації, рекомендацій. І між цими шарами постійно йде робота - змінюються правила, підправляються атрибути, додаються нові поля.

І нарешті, світ IoT та сенсорних систем. Тут реальний час прив'язаний не до логіки бізнесу, а до фізичних процесів. Якщо датчик температури в серверній повідомив про перегрів - у тебе є дуже обмежений час, щоб відреагувати. Якщо лічильник споживання електроенергії передає дані раз на кілька секунд, а потім раптово "замовкає", це теж сигнал. І проблема не тільки в об'ємі, хоча об'єми там часто колосальні. Проблема в тому, що дані приходять з різною точністю, із різними затримками, іноді зовсім не приходять.

У таких системах архітектура змушена враховувати фізичні обмеження: нестабільні канали, батареї на пристроях, локальні буфери. Часто сенсори можуть накопичувати події й відправляти їх пакетами, коли зв'язок з'являється - і тоді "реальний" час розтягується. Система бачить подію постфактум, але повинна якось

інтерпретувати її так, ніби вона все-таки сталася у свій момент. Звідси й складні механізми роботи з часовими мітками, локальними годинниками, компенсаційними алгоритмами. Усе це виглядає дуже технічно, але насправді впирається в одне просте питання: чи можемо ми довіряти цьому потоку настільки, щоб діяти на його основі.

Таблиця 2.2.

Класифікація real-time систем за функціональними ролями

Тип системи	**Функціональна роль**	**Архітектурні особливості**	**Ключові обмеження**
Системи оперативної аналітики	Поточне спостереження за станом бізнесу та сервісів	Streaming-first, агрегування в реальному часі	Компроміс між швидкістю та точністю
Транзакційно-потокові системи	Реакція на події в межах бізнес-операцій	Гібрид OLTP + streaming, ідемпотентність	Ризик неконсистентності
Behavioral / Clickstream системи	Аналіз поведінки користувачів	Подієва модель, сесійна обробка	Висока мінливість потоків
Системи телеметрії та моніторингу	Контроль технічного стану систем	Time-series обробка, алертинг	Високий обсяг і шум даних
IoT та сенсорні системи	Робота з фізичними процесами	Event-time, буферизація, компенсація	Нестабільність і затримки даних

Саме тому класифікація real-time систем за функціональними ролями виглядає більш чесною, ніж класифікація за технологіями. Фреймворки змінюються, з’являються нові назви, одні платформи стають модними, інші йдуть у тінь. А от ці ролі - оперативне спостереження, транзакційна точність,

телеметрична видимість, аналіз поведінки, робота з сенсорами - залишаються. І саме вони визначають, якими будуть архітектури завтра, незалежно від того, як називатиметься черговий стрімінг-движок.

2.3. Операційні та організаційні аспекти підтримки real-time систем

CI/CD у real-time -зовсім не CI/CD у класичному розумінні. У теорії це автоматичний процес, а в реальності -це цілий ритуал. Деплой стає майже подією: хтось прокручує логи, хтось тримає відкритими графіки затримок, хтось перевіряє freshness, хтось уже заздалегідь розуміє, які вузли можуть "заслабнути", якщо їх навантажити новою логікою. І те, що з боку здається "обережністю", насправді є стратегією виживання.

Потоки ж не мають пауз. Немає, умовно кажучи, "вікна", де можна щось оновити без ризику. Навіть о третій ночі вони можуть працювати так само інтенсивно, як удень. Тому потрібно навчитися змінювати колесо на ходу. Хотілося б сказати, що це робиться за допомогою hot-reload. Але hot-reload -це не магія. Це щось середнє між трюком і дуже добре спланованою авантюрою.

Складність у тому, що логіка обробки майже завжди має стан. Десь накопичуються події, десь тримаються проміжні результати. І все це сидить у пам'яті, у тих непомітних структурах, які самі вирішують, коли й що їм "зручно" вивільнити чи оновити. Змінити логіку без втрати цього стану -означає майже те саме, що вставити нову частину в механізм, який крутиться просто зараз. І якщо ти хоч на частку секунди помилишся, отримаєш дублікати, пропуски, або ще гірше - подвійну інтерпретацію однієї й тієї ж події.

Через це тіньові прогінки -shadow traffic -стають чи не найпопулярнішою технікою. На словах усе звучить просто: взяти копію потоку, прогнати її новою логікою і порівняти результат. Але на практиці shadow-режим часто викликає подвійний ефект: по-перше, він сам створює навантаження, яке змінює поведінку реального каналу; по-друге, копія потоку ніколи не поводиться абсолютно так

само, як оригінал. І інженери це знають. Тому shadow traffic -це не про "точне відтворення". Це про те, щоб хоча б побачити, чи не веде нова логіка до очевидного хаосу.

Ще один момент -це те, як сильно real-time системи залежать від узгодженості схем подій. Усі розуміють, що схема повинна еволюціонувати. Але коли вона еволюціонує в реальному часі, це означає, що продюсери й консьюмери можуть бути на різних етапах цієї еволюції протягом годин або навіть днів. Через це в потоці постійно змішуються події різних версій, і pipeline повинен бути готовим до цього. Ніякого єдиного моменту "перемикання" просто не буває.

Моніторинг у таких системах теж перестає бути набором красивих візуалізацій. Він перетворюється на спробу вловити зміни настрою системи. Латентність реагує на зовнішні зміни швидше за всі інші індикатори. Freshness інколи бреше, і її потрібно перевіряти непрямими ознаками. А throughput взагалі поводиться як жива істота: він росте й падає у хвилях, і в кожної хвилі є своя причина, хоч інколи вона неочевидна навіть фахівцям.

Якщо трафік раптом зростає вдвічі, система, яку хвалили за стійкість, може почати поводитися так, ніби її ніхто не проєктував. І ніхто й не проєктував на таке. Бо real-time не масштабується лінійно. Він масштабується через черги, які завжди були десь на межі. І коли одна черга зсувається хоч трохи, це створює тиск на інші. Поступово вся архітектура починає хитатися. І досвідчені інженери знають цей стан: ніби нічого критичного не сталося, але система вже не така, як учора.

У таких умовах тестування перетворюється на окрему інфраструктуру. Створюють дублікати конвеєрів, які, по суті, роблять те саме, що основний. Це дорого, це не завжди ефективно, але це єдиний спосіб побачити, як система поводиться у природних умовах. Емулятори тут майже не допомагають -вони не можуть передати характер реального трафіку. Лише справжні події, які рухаються своїми "хвилями", відкривають ті вразливі місця, про які ти навіть не думав.

З одного боку, це технічна робота, з конкретними задачами й чіткими інструментами. А з іншого -це майже ремісництво. Коли ти не просто пишеш код,

а доглядаєш систему. Хтось би назвав це підтримкою. Але ближче до суті те, що ти робиш, нагадує спостереження. Ти не стільки керуєш потоками, скільки стежиш, щоб вони не виходили з русла. І коли вони виходять, ти намагаєшся зрозуміти не "як виправити", а "чому це сталося саме зараз", бо в real-time дрібниць немає.

РОЗДІЛ 3 РОЗРОБЛЕННЯ ТА РЕАЛІЗАЦІЯ ПРОТОТИПУ СИСТЕМИ АНАЛІЗУ ДАНИХ У РЕАЛЬНОМУ ЧАСІ «VizoraData»

Система VizoraDate це інструмент, який має зняти з аналітика більшу частину рутинних дій, що супроводжують первинне знайомство з даними. У робочих потоках, де дані надходять фрагментами або партіями, особливо помітно, наскільки важливо швидко оцінити структуру набору, виявити неконсистентності та зрозуміти, з чим саме доведеться працювати далі. Саме на ці задачі орієнтована концепція прототипу: створити середовище, яке реагує на завантажений файл майже миттєво, формує перший аналітичний зріз і дає змогу рухатися від загального огляду до більш складних оцінок - статистичних, структурних і навіть інтелектуально сформульованих.

Рис.3.1 Логотип VizoraDate

У своїй основі VizoraDate поєднує два підходи. З одного боку, це Streamlit-додаток із виразною логікою інтерактивних сценаріїв: користувач завантажує файл, інтерфейс змінюється відповідно до його типу, а вся подальша навігація відбувається всередині однієї сесії без перезапуску та без втрати контексту. З іншого боку, поруч існує експериментальний Dash-модуль, який відпрацьовує ідею напівавтоматичного визначення структурних характеристик колонок, зокрема типів даних та часових патернів. Обидва шари працюють з одним і тим самим ядром, але по-різному розставляють акценти: Streamlit більше про швидкий огляд і візуалізацію, Dash - про гнучкішу логіку та контроль за кожною операцією.

Функціональність системи розширюється доволі широким набором можливостей. Після завантаження даних користувач отримує паспорт набору:

кодування, формат, розмір, кількість рядків і колонок, а також ознаки можливих дат. Для будь-якого CSV, JSON або NDJSON прототип будує структуру, автоматично визначає типи полів, підраховує пропуски, формує базову статистику, будує кореляції, теплові карти якості та витягує приклади значень. Поверх цього аналітичного шару додано модуль AI-описів колонок, який через зовнішню LLM генерує змістовні коментарі й пропонує інтерпретацію призначення полів. А блок пошуку унікального ключа автоматично перебирає комбінації колонок і виявляє структури, що можуть виступати природним об’єднувальним ідентифікатором.

В результаті VizoraDate виступає як прототип, який демонструє, як можна поєднати швидку реакцію інтерфейсу, структуровану обробку даних і інтелектуальні підказки в одному робочому середовищі. Це достатньо повний інструмент, що дає уявлення про архітектуру та принципи майбутнього комплексного рішення для аналізу даних у режимі, близькому до реального часу.

3.1. Архітектура та загальна концепція програмної реалізації VizoraDate

VizoraDate – це інструмент, що дозволяє працювати з даними швидко, без довгих підготовчих етапів і без потреби вручну "розкручувати" структуру файлу. У типовій ситуації аналітик отримує файл, часом від невідомого джерела, часом у форматі, який не зовсім стандартний, і його перший крок майже завжди однаковий: визначити структуру, оцінити якість, зрозуміти, чи є пропуски, чи збігаються типи значень, як виглядають категорії, чи є дублікати і чи можна визначити природний ключ. Саме на цей початковий контакт з даними й орієнтований прототип: він повинен формувати відчуття, що дані "розкриваються" одразу після завантаження, без зайвих дій з боку користувача.

У своїй основі VizoraDate працює як система, що реагує на подію завантаження файлу й одразу запускає ланцюжок автоматичного аналізу. Користувач натискає "Завантажити файл", передає CSV, JSON або NDJSON, і наступна сторінка вже містить паспорт набору: назву, розмір, кодування, кількість

рядків і колонок, ознаки колонок з датами, а також структуру (для JSON і NDJSON). Після цього інтерфейс пропонує перейти до огляду даних, подивитися типи полів, приклади значень, частку пропусків, статистику для кожної колонки, теплову карту якості чи структуру дублікатів. Усе це побудовано навколо одного датафрейму, що зберігається у внутрішньому стані сесії та не оновлюється без явного запиту користувача.

Прототип опирається одразу на два інтерфейсні рішення. Основний контур побудований на Streamlit - легкому фреймворку, який дозволяє формувати інтерфейс декларативно, реагуючи на зміни даних і стану. Streamlit відповідає за швидкі інтерактивні сценарії: перегляд структури таблиці, статистичний аналіз, AI-генерацію описів колонок, пошук унікальних ключів, побудову візуалізацій. Він працює в режимі, наближеному до real-time, у тому сенсі, що дії користувача відразу призводять до оновлення вмісту інтерфейсу без перезапуску всієї програми.

Паралельно існує Dash-прототип, який зосереджений на більш жорстко структурованих обчисленнях. Dash дозволяє будувати складніші інтерфейсні конструкції, такі як вкладки, розділені за типами колонок, або інструменти, де дані передаються між частинами інтерфейсу через окремі сховища (dcc.Store). Цей контур використовується як зона експериментів: аналіз дат, автоматичне визначення типів полів, побудова вкладених структур, сценарії, що вимагають колбеків Input/Output, яких у Streamlit немає.

Хоча системи працюють паралельно, вони поділяють однаковий концептуальний центр - модулі, розташовані в директорії core/. Там зібрана бізнес-логіка початкової обробки файлів: визначення кодування, аналіз структури CSV, розбір JSON і NDJSON, flatten-відображення вкладених об’єктів, підрахунок дат та формування службової інформації. Ці модулі не залежать від UI, отже інтерфейс може змінюватися чи доповнюватися, а ядро залишатиметься стабільним.

Сама модель роботи з даними в системі побудована навколо простого принципу: один файл = один внутрішній стан. Якщо файл завантажений, він зберігається у session_state разом із усіма характеристиками (кодування, розмір,

назва, кількість рядків, структура JSON, згенерована схема тощо). Далі ці дані читають різні модулі інтерфейсу -і завдяки цьому користувач перемикається між вкладками без додаткових обчислень. Така модель суттєво пришвидшує роботу, тому що кожна вкладка не виконує повторне читання файлу, а працює з уже сформованою структурою.

Побудова програмної архітектури VizoraDate почалася з потреби створити систему, яка може тримати в пам'яті одразу кілька рівнів обробки даних: від моменту отримання файлу до формування інтелектуального опису, пошуку ключів, виявлення аномалій та генерації підсумкових висновків. Розроблення цього прототипу пройшло через безліч ітерацій, і кожна з них показувала ту саму річ: якщо немає чіткої логічної структури, все інше швидко перетворюється на хаос. Тому архітектура VizoraDate - це про те, як система повинна працювати всередині.

З позиції користувача все виглядає як послідовність екранів, які розкривають різні грані одного й того ж набору. Спочатку - паспорт файлу. Потім - огляд структури. Далі - статистика. Якщо потрібно - AI-коментарі, що допомагають зрозуміти зміст даних. І вже окремим кроком - пошук унікального ключа та аналіз дублікованих записів. Оскільки кожний із цих модулів незалежний, користувач може повертатися назад, порівнювати результати, знову переглядати структуру, а потім переходити до візуалізацій, не перезавантажуючи файл.

Перевага такого підходу - у відділенні інтерфейсної логіки від змістовної, а також у тому, що система дає змогу працювати з даними поступово, не перевантажуючи користувача одномоментно великим обсягом інформації. Кожен модуль відкриває новий бік набору: структура показує форму, статистика - поведінку, AI - зміст, аналіз ключів - структурну цілісність. Усе це формує цілісний досвід роботи, але при цьому кожний сценарій може існувати самостійно.

Детальний огляд функціоналу VizoraDate:

Робота із VizoraDate виглядає як послідовна навігація між вкладками, що відкривають різні аспекти одного й того самого набору даних. У верхній частині інтерфейсу розміщений перемикач розділів, де в одному списку видно всі ключові

сценарії: від базового огляду до BI-дешборду й AI-висновків. Користувач у будь-який момент може змінити вибір, перейти в інший розділ і повернутися назад, при цьому дані не перезавантажуються, а весь аналіз відбувається в межах однієї сесії.

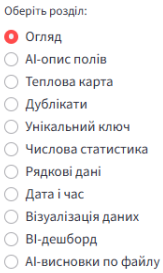


Рис. 3.2 Головне меню розділів VizoraDate

Після завантаження файлу система не поспішає занурювати користувача в таблицю з тисячами рядків. Першим з'являється компактний "паспорт" набору - блок із ключовими характеристиками, який дозволяє одним поглядом оцінити, з чим саме доведеться працювати. Тут же система показує назву, розмір, формат, кодування, кількість рядків і колонок, а також кількість полів, схожих на дати.

Ці параметри формуються автоматично в момент обробки файлу, тому користувач нічого не налаштовує вручну. Такий старт сильно спрощує навігацію: ще до відкриття таблиці зрозуміло, наскільки дані великі, чи можна очікувати часові ряди, чи є ризики з кодуванням.

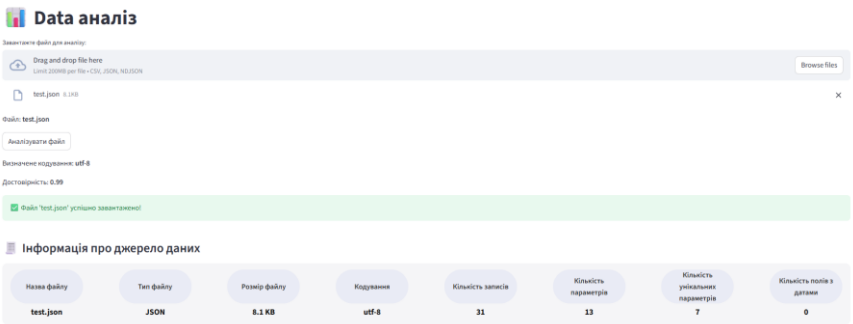


Рис.3.3 Паспорт набору: базові метадані й загальна інформація про джерело

Далі користувач переходить до основної таблиці - блоку "Огляд". Тут дані вперше показуються "вживу": зразки кількох рядків, перелік параметрів, приклади значень, приклад структури записів у файлі. Завдяки такому комбінованому

відображенню вдається одразу помітити, де дані мають змішані типи, де строкові значення виглядають як числа, або де частина значень є вкладеними JSON-структурами. Користувач може розгортати окремі частини інтерфейсу, відкривати експандери, переглядати схему JSON чи приклади значень - без зайвого перемикання між сторінками.

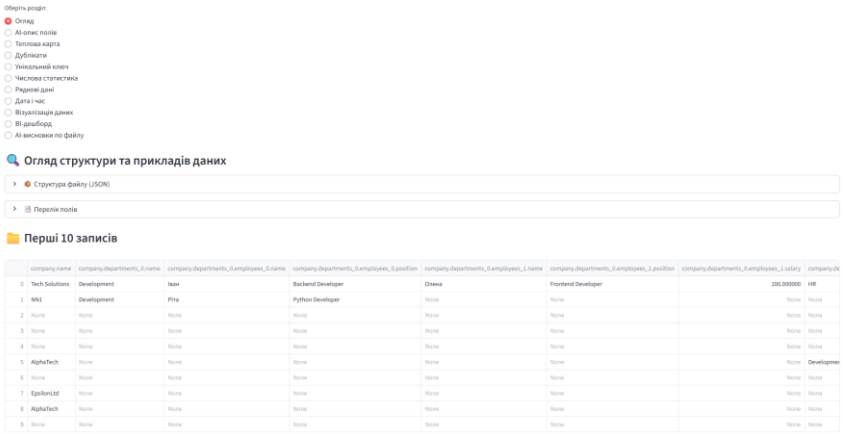


Рис. 3.4 Блок ‘Огляд’: структури колонок і приклади значень

В наступному розділі увага поступово зміщується від загального огляду до глибшого аналізу. “AI-опис полів” відкриває інтелектуальний шар: для кожної колонки система, використовуючи LLM, формує короткий текстовий опис, інтерпретує тип даних, типові значення, можливе призначення поля. Користувач бачить це у вигляді таблиці з описами, може щось поправити або переосмислити, якщо контекст йому відомий краще, ніж моделі. Такий екран часто стає ключовим, коли потрібно швидко розібратися в чужому наборі, де назви колонок не завжди промовисті. Для цього користувач самостійно спочатку обирає параметри, які потрібно описати. За замовчуванням, одразу доступний вибір всіх параметрів, які можна редагувати :

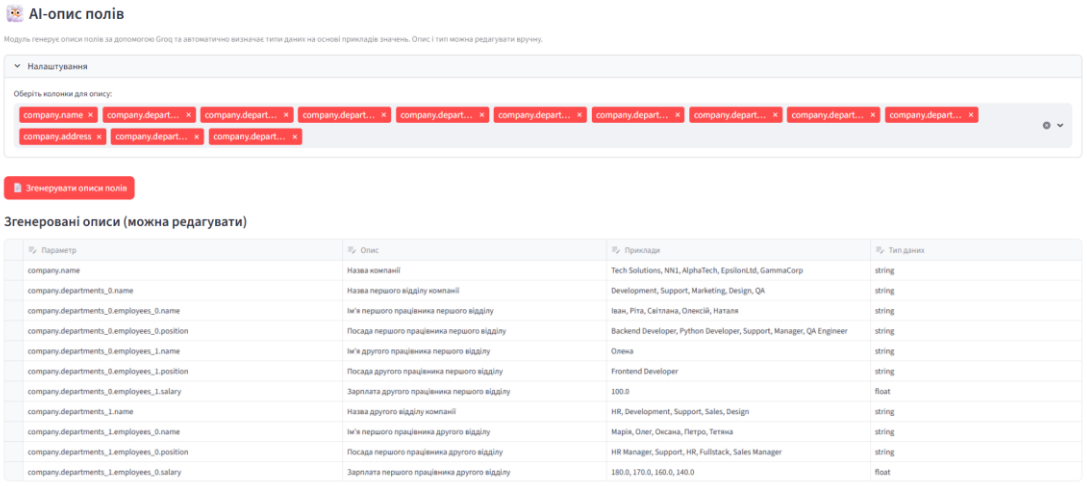


Рис. 3.5 Генерації AI-описів полів

Далі йде блок, пов’язаний з якістю даних та їх структурними аномаліями. Вкладка “Теплова карта” виводить агреговану картину пропусків, підозрілих значень і структурних проблем. Відсоткові колонки тут представлені як смуги з градієнтом, що показує частку порожніх та унікальних записів. При необхідності користувач може розгорнути окрему секцію й подивитися конкретні значення, які система вважає проблемними. Це дозволяє швидко локалізувати “вузькі місця” в наборі ще до того, як починати серйозну підготовку даних.

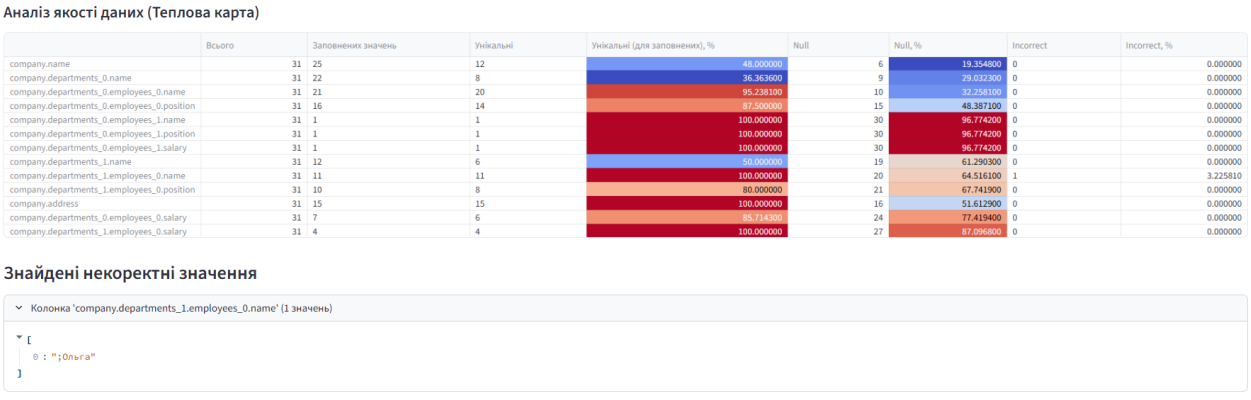


Рис. 3.6 Блок ‘Теплова карта’: огляд пропусків та аномальних значень

Окремий сценарій пов’язаний із дублями та унікальністю записів. У вкладці “Дублікати” показуються повні дублікати рядків, оцінюється їхня кількість.

Рис. 3.7 Розділ ‘Дублікати’

Поруч із цим працює вкладка “Унікальний ключ”: тут система перебирає окремі колонки та їх комбінації, підраховує частку унікальних значень і пропонує кандидати на природний ключ таблиці. Користувач бачить, які комбінації дають 100 % унікальності, а які лише наближаються до неї, і вже на цій основі може приймати рішення щодо структурування набору. Якщо ж комбінації, які дають 100% унікальність відсутні, система пропонує очистити повні дублікати, та заново визначити унікальну комбінацію.

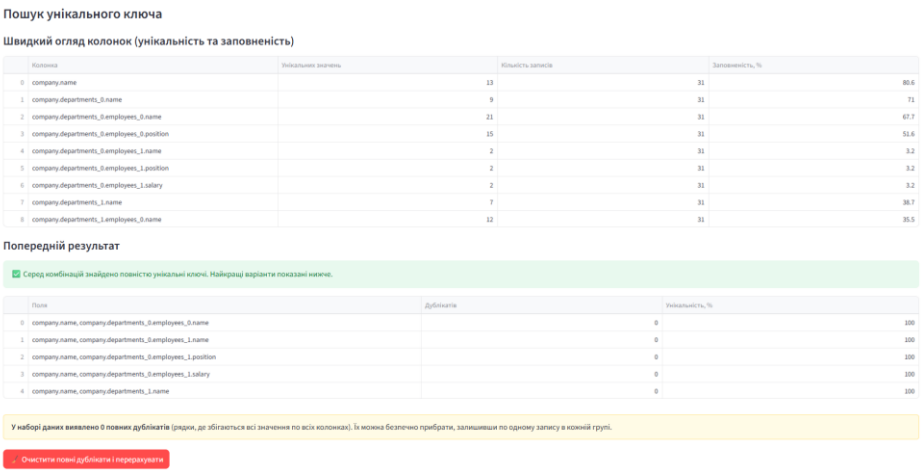


Рис. 3.8 Розділ ‘Унікальний ключ’

Далі аналіз розвивається в бік детальної статистики. Вкладки “Числова статистика”, “Рядкові дані” та “Дата і час” розділяють поля за типами й дають різні інструменти для кожного класу. У числовому блоці система рахує основні

дескриптивні показники, частку нульових та від'ємних значень, коефіцієнт варіації, дозволяє подивитися розподіл і, за потреби, рухомі агрегати.

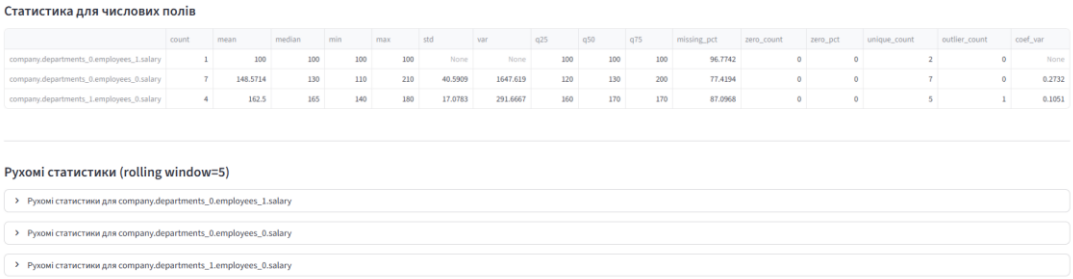


Рис. 3.9 Розділ 'Числова статистика'

Для рядкових колонок акцент робиться на кількості унікальних значень, довжині рядків, частці порожніх або "дивних" записів.

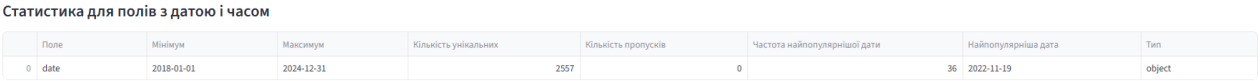
Рис. 3.10 Розділ 'Рядкові дані'

Розділ "Дата і час" окремо розглядає колонки із датами й часовими мітками: там можна побачити розподіл по періодах, щільність спостережень, стабільність часових інтервалів. Цей розділ став одним із найбільш насичених у всій системі, тому що саме часові дані зазвичай вимагають окремого підходу. Тут дані не просто подаються як значення у таблиці -система намагається "зловити" ритм набору: чи існує сезонність, чи є пікові періоди, як розподілені події у часі, чи є аномалії, і що взагалі відбувається з часовою компонентою.

Перший блок присвячений найпростішому, але критично важливому: загальним статистичним характеристикам часових колонок. Тут користувач бачить мінімальні та максимальні дати, діапазон охоплення, кількість спостережень,

частку пропусків, кількість унікальних часових значень і базові агрегації. На практиці цей блок дозволяє швидко зрозуміти, чи не “провисає” набір на великих часових відрізках, чи є дуплікація часових відміток, чи містить колонка “випадковий шум”.

У цьому ж блоці система автоматично визначає тип: дата, дата-час, timestamp, комбіновані формати. Для “сирих” дат, що мають різний формат у межах однієї колонки, VizoraDate намагається привести значення до єдиного стандарту, і якщо це неможливо -позначає колонку як “вимагає очищення”.


Статистика для полів з датою і часом

	Поле	Мінімум	Максимум	Кількість унікальних	Кількість пропусків	Частота найпопулярнішої дати	Найпопулярніша дата	Тип
0	date	2018-01-01	2024-12-31	2557	0	36	2022-11-19	object

Рис. 3.11 Статистика дат

Далі йде великий блок розподілів. Тут дані агрегуються по різних часових рівнях -системі байдуже, скільки у колонці точок, важливо, що вона містить часові ознаки. Залежно від доступних значень формується набір з кількох графіків:

- розподіл записів за роками,
- розподіл за кварталами,
- розподіл за місяцями,
- розподіл за тижнями,
- інколи -за днями тижня, якщо це релевантно.

Кожен графік візуально показує “щільність” даних у певний період. Дуже легко помітити провали, нерівномірність та піки: у продажах це можуть бути сезони, у веб-трафіку - дні з піковими навантаженнями, у фінансових транзакціях -характерні хвилі перед закриттям періодів.

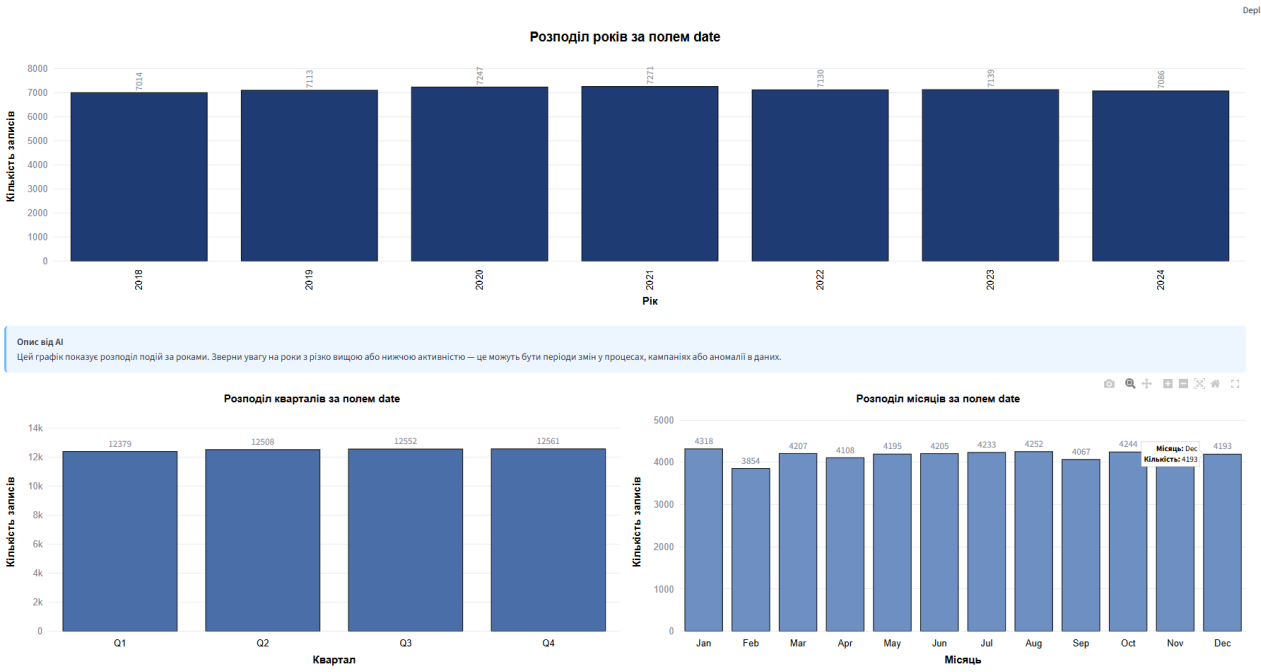


Рис.3.12 Розподіл дат за роками/кварталами/місяцями/тижнями

Після розподілів система переходить до динаміки. Тут дані збираються у часовий ряд: залежно від природи набору це може бути кількість записів на день, сумарне значення певної метрики, середнє за групою чи будь-який інший агрегат. Графік показує, як значення змінювалися з часом.

Для того, щоб зменшити шум, система автоматично додає лінію згладжування (moving average). Її параметри підбираються залежно від масштабу набору: якщо це денні дані за кілька років -вікно більше; якщо це годинні дані за кілька днів -менше. Такий графік дозволяє вловити тренд, не відволікаючись на випадкові коливання.

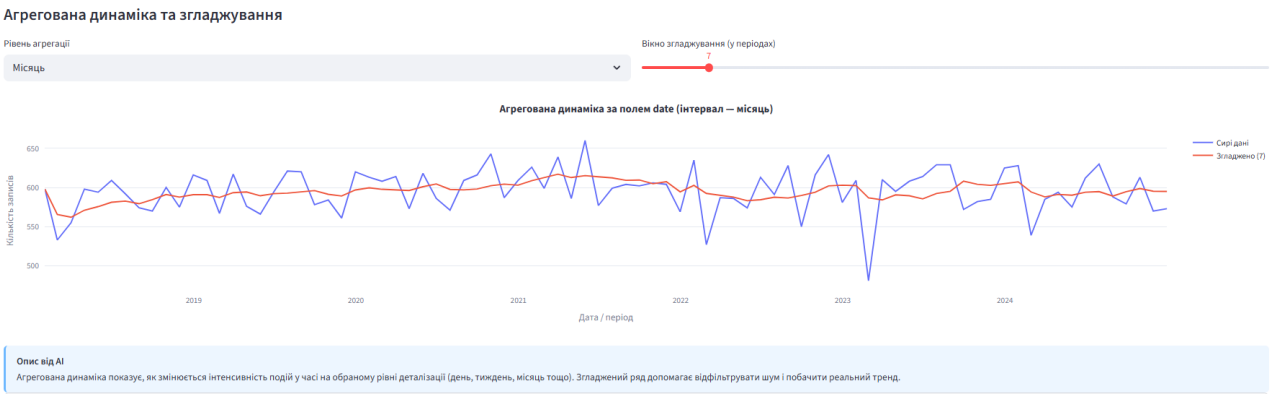


Рис. 3.13 Агрегована динаміка та згладжений часовий ряд

Далі вмикається блок теплових карт. Це окремий тип подання даних, який наочно показує “інтенсивність” подій у різних зонах календаря. Найчастіше це heatmap “дні vs місяці” або “години vs дні тижня”, залежно від того, які дані присутні. За таким графіком дуже легко побачити:

- періоди високої активності,
- календарні провали,
- стабільні шаблони,
- нюанси поведінки за годинами чи днями тижня.

Для наборів, де важлива сезонність, цей графік є фактично незамінним.

Поруч система формує перелік пікових дат -днів, коли кількість записів або значень суттєво відрізнялася від медіанного рівня. Це дозволяє швидко зорієнтуватися, чи є у вибірці “аномальні дні”.

Повноцінний часовий ряд: тренд, сезонність, аномалії (STL + ACF) . Це один із найскладніших блоків, і він якраз робить розділ “Дата і час” таким потужним. Система автоматично будує розкладення часової серії за допомогою STL (Seasonal-Trend Decomposition). Це дозволяє розділити ряд на три компоненти:

- тренд -довгостроковий напрям зміни,
- сезонність -повторювані цикли,
- залишки -шум і несподівані відхилення.

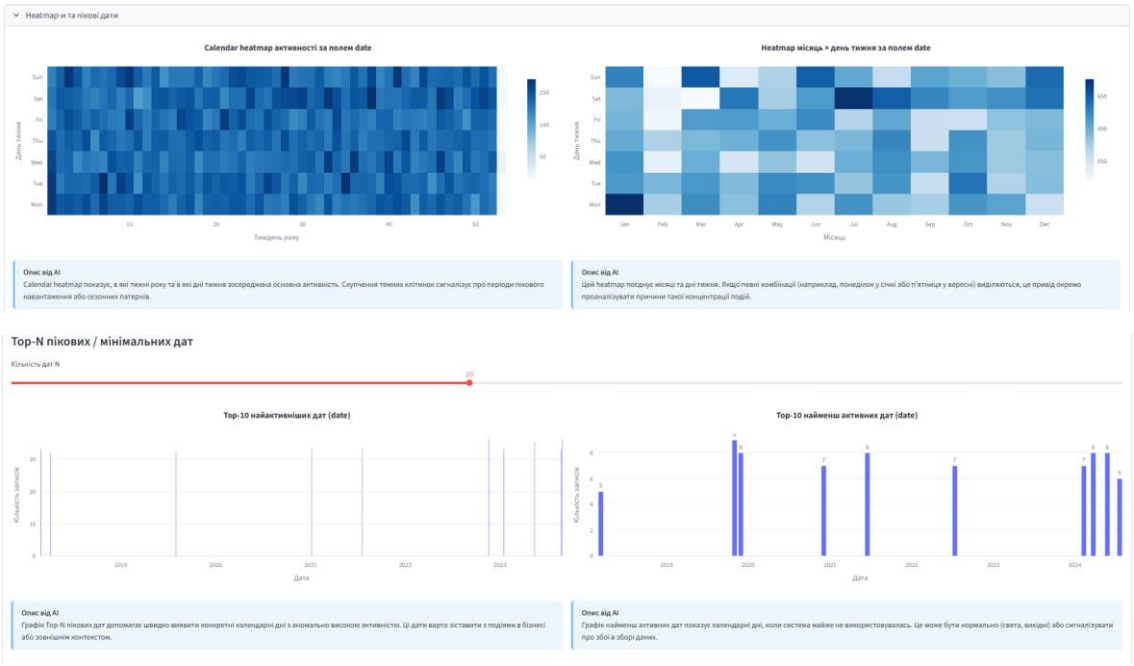


Рис. 3.14 Календарний heatmap та список пікових дат

Така декомпозиція робить видимими ті патерни, які важко сприймати на простому лінійному графіку.

Далі система автоматично обчислює ACF (autocorrelation function) - автокореляцію часового ряду. Графік ACF дозволяє побачити, чи є повторювані цикли, яка їхня довжина та чи можна припускати наявність довготривалої пам’яті у ряду. Це критично важливо для моделювання часових процесів.

Після цього система формує список потенційних аномалій: дні або періоди, де значення сильно вибиваються із загального патерну. Часто це операційні помилки, пікові навантаження, збої у системах збору даних або початок нових трендів.

Рис. 3.15 STL-декомпозиція: тренд, сезонність, залишки

Рис. 3.16 ACF-графік: автокореляції часового ряду

Після кожної візуалізації система генерує короткий текстовий висновок: що саме відбувається на графіку, які патерни помітні, чи є сезонність, чи присутній тренд, які потенційні причини піків, на що звернути увагу.

Фактично це автоматичний аналітичний супровід, який допомагає користувачу не лише бачити, але й інтерпретувати дані. В AI-описах система використовує контекст з графіка і приклади з реальних значень, тому коментарі отримуються достатньо точними й корисними.

Окремий пласт взаємодії забезпечують вкладки, орієнтовані на візуальне подання результатів.

У блоці “Візуалізації даних” користувач може будувати базові графіки й діаграми для вибраних колонок, експериментувати з різними поєднаннями ознак, дивитися, як зміни в параметрах впливають на графік.

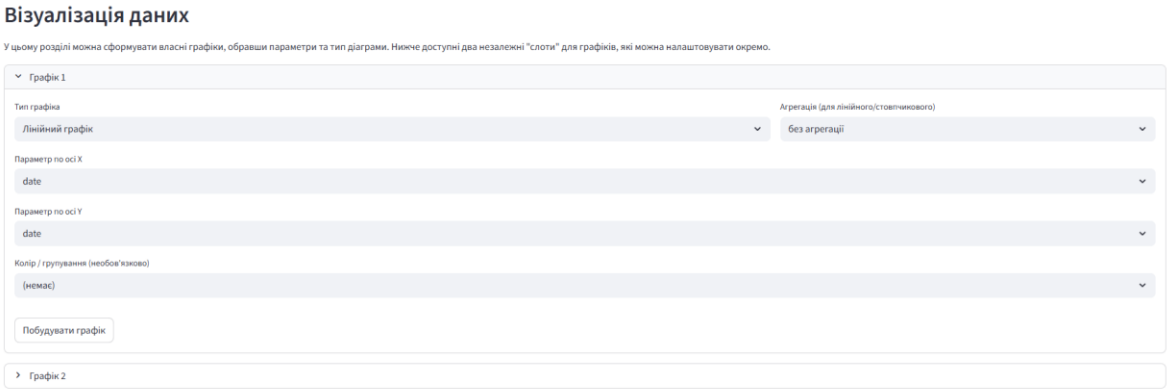


Рис. 3.17 Розділ ‘Візуалізація даних’

Вкладка “BI-дешборд” збирає найважливіші показники в одному місці: ключові агрегати, кілька репрезентативних графіків, узагальнені показники якості. До кожного графіку генеруються AI -описи. Це радше оглядовий блок, який зручно використати як вихідну точку для подальшої звітності.

Рис.3.18 Розділ ‘BI-дешборд’

Завершує цей ланцюжок вкладка “AI-висновки по файлу”. На цьому екрані система намагається зібрати попередні кроки в більш цілісний наратив: на основі структури, статистики, якості та прикладів значень формується текстовий висновок

про стан набору, його сильні й слабкі сторони, потенційні ризики та напрямки подальшої обробки. Також, вкінці реалізано чат форму з AI-агентом, який зможе в реальному часі надати всі відповіді на питання щодо цього файлу. Фактично це спроба автоматизувати першу аналітичну записку, яку аналітик зазвичай робить для себе чи для команди після знайомства з новим файлом.

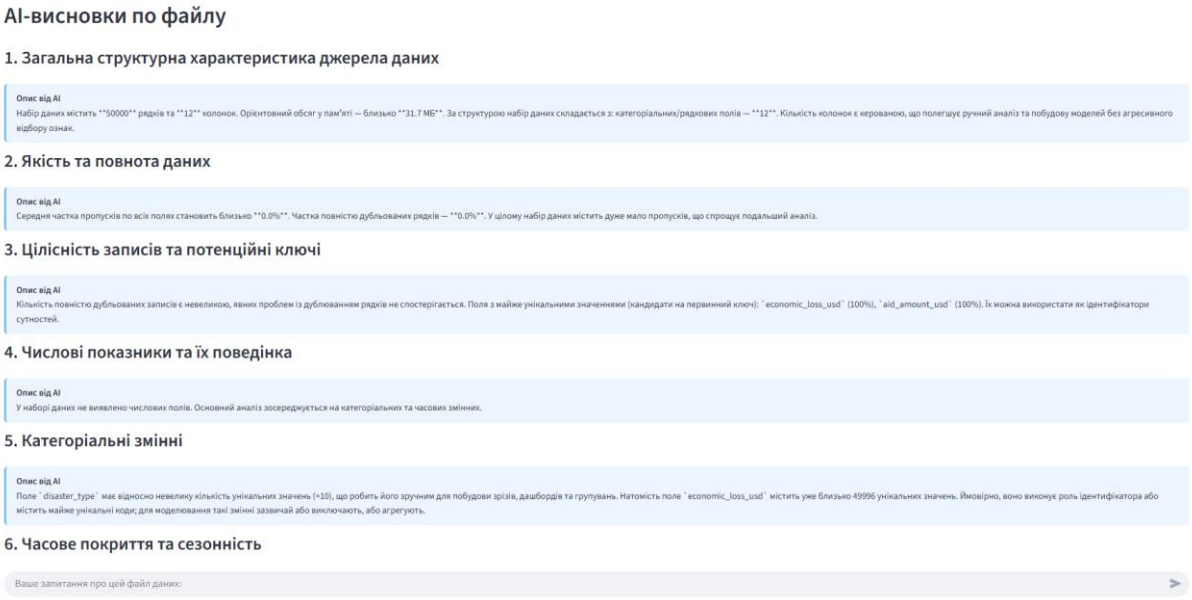


Рис. 3.19 Розділ ‘AI-висновки по файлу’

У підсумку користувач отримує набір розділів, які природно доповнюють один одного, і при цьому кожний із них досить самодостатній, щоб використовуватися окремо.

Структура проєкту та логіка організації директорій

Архітектура VizoraDate на перший погляд здається простою, проте всередині вона складається з кількох чітко розмежованих шарів, кожен з яких відповідає за свій аспект роботи системи. Структура директорій продумана так, щоб ізолювати бізнес-логіку від інтерфейсу, а інтелектуальні компоненти - від механізмів обробки файлів. Такий поділ дозволяє розвивати кожен шар окремо: UI може вдосконалюватися без втручання в логіку обчислень, а аналітичні модулі легко масштабуються, не зачіпаючи структуру Streamlit чи Dash.

Уся логіка системи зібрана у декількох основних директоріях:

- **core/** -ядро обробки даних, утиліти, логіка читання файлів;
- **ui/** -модулі, що відповідають за конкретні екрани Streamlit-додатку;
- **ai/** -інтелектуальний шар, генерація описів, робота з Groq API;
- **sections/** -модулі для Dash-прототипу;
- **.streamlit/** -конфігурації, секрети, кастомні налаштування;
- файли верхнього рівня (app.py, main.py, heatmap.py, stats.py, settings.py тощо).

Для розуміння загальної картини, структура проекту у вигляді багаторівневого дерева виглядає так:

```
VizoraDate/
│
├── app.py
├── main.py
├── settings.py
├── heatmap.py
├── stats.py
├── preprocessing.py
│
├── core/
│   ├── file_loader.py
│   ├── get_file_info.py
│   ├── read_csv.py
│   ├── read_json.py
│   ├── utils.py
│
├── ui/
│   ├── overview_view.py
```

```
│   ├── metadata_view.py
│   ├── ai_descriptions_view.py
│   ├── unique_key_view.py
│   ├── duplicates_view.py
│   ├── stats_views/
│   │       ├── numeric_stats_view.py
│   │       ├── string_stats_view.py
│   │       ├── datetime_stats_view.py
│   │
│   ├── visualization_view.py
│   ├── bi_dashboard_view.py
│   ├── ai_file_summary_view.py
│
├── ai/
│   ├── column_describer.py
│   ├── prompt_builder.py
│   ├── ai_client.py
│
├── sections/
│   ├── upload_section.py
│   ├── column_type_detection_section.py
│   ├── date_analysis_section.py
│
├── .streamlit/
│   ├── secrets.toml
│   ├── config.toml
│
└── chat_history.json
```

Такий розклад відображає реальний поділ відповідальностей. Весь вхідний конвеєр по роботі з файлами винесений у папку core і в окремі модулі get_file_info.py, read_csv.py, read_json.py на верхньому рівні. Інтерфейс Streamlit живе в ui і залежить від ядра, але не навпаки. Інтелектуальний шар сидить в ai, ізольований від решти. Dash-прототип зібраний у sections і main.py. А .streamlit тримає у собі ключ та конфігурації, потрібні під час запуску.

У самому ядрі майже весь трафік проходить через core/file_loader.py. Раніше ця логіка була прямо в app.py, але її потрібно було винести окремо, щоб Streamlit не "знаходився в курсі" деталей роботи з файлами. Функція process_file приймає шлях до тимчасового файлу та його оригінальну назву, визначає кодування, обирає правильний рідер і заповнює session_state. У коді це виглядає дуже прямо:

```
def process_file(file_path: str, original_name: str | None, st):
    try:
        encoding, confidence = get_file_info.detect_encoding(file_path)

        if file_path.lower().endswith(".csv"):
            delimiter = get_file_info.detect_delimiter(file_path, encoding)
            df = read_csv.read_csv(file_path, encoding=encoding, separator=delimiter)
            st.session_state['delimiter'] = delimiter
        elif file_path.lower().endswith(".json"):
            df, schema_json, parameters_list = read_json.read_json_file(file_path, encoding=encoding)
            st.session_state['schema_json'] = schema_json
            st.session_state['parameters_list'] = parameters_list
            st.session_state['json_type'] = "JSON"
        elif file_path.lower().endswith(".ndjson"):
            json_type, raw_json = read_json.load_raw_json(file_path, encoding)
            st.session_state['json_type'] = json_type
```

```
                schema_json = read_json.generate_json_schema(raw_json)
                st.session_state['schema_json'] = schema_json
                df, parameters_list = read_json.read_json(raw_json, schema_json=schema_json)
                st.session_state['parameters_list'] = parameters_list
            else:
                st.error("Непідтримуваний формат файлу. Підтримуються: CSV, JSON, NDJSON.")
                return

            st.session_state['df'] = df
            st.session_state['filename'] = original_name if original_name else os.path.basename(file_path)
            st.session_state['file_size'] = os.path.getsize(file_path)
            st.session_state['encoding'] = encoding
            st.session_state['date_cols'] = get_file_info.count_date_like_columns(df)
            st.session_state['main_tab'] = "Огляд"

        except Exception as e:
            print(f"Помилка під час аналізу файлу: {e}")
            st.error(f"Помилка під час аналізу файлу: {e}")
```

Тут добре видно, що file_loader нічого не знає про вкладки, графіки чи AI. Його завдання - зчитати CSV, JSON або NDJSON, покласти результат у стійкий стан і заодно наповнити службові поля: кодування, розмір, кількість колонок з датами, схему JSON, список параметрів. Streamlit отримує вже готову картинку світу і лише вирішує, як її показати. Важливий момент - format-специфічна логіка винесена в окремі модулі read_csv.py і read_json.py. Наприклад, читаючи CSV, система використовує polars і спеціальні налаштування, щоб не "проковтнути" проблемні рядки:

```
import polars as pl


def read_csv(path, encoding=None, separator=None) -> pl.DataFrame:
    return pl.read_csv(
        path,
        encoding=encoding,
        has_header=True,
        separator=separator,
        infer_schema_length=0,
        ignore_errors=False,
        truncate_ragged_lines=False
    )
```

Тут немає ніяких поблажок типу ignore_errors=True. Якщо в даних щось не так, то краще отримати явну помилку, а не напівзавантажений датасет. Для JSON та NDJSON, навпаки, логіка значно складніша і винесена в read_json.py разом із функціями flatten_dict, generate_json_schema, read_json, які будують плоский polars.DataFrame з вкладених структур. Допоміжний модуль get_file_info.py теж стоїть на боці ядра: він відповідає за detect_encoding, detect_delimiter, приблизний аналіз типів колонок, пошук кандидатів на унікальний ключ. Частина цього коду прямо використовується в Streamlit, частина - в Dash, але сам модуль нічого не знає про конкретний інтерфейс.

Вся інтерфейсна логіка Streamlit лежить в ui. Тут кожен файл - фактично окремий екран. metadata_view.py відповідає за "паспорт" файлу, overview_view.py за структурний огляд, ai_descriptions_view.py за AI-описи колонок, unique_key_view.py за пошук унікального ключа. Важливо, що ці модулі не читають файли напряму і майже не займаються важкими обчисленнями. Вони працюють з тим, що вже лежить у session_state. Наприклад, metadata_view будує компактні картки параметрів і малює їх через HTML/CSS, але всі значення бере з session_state, а не з диску.

UI концентрується на поданні: flex-блоки, картки, форматування розміру файлу через _format_size. Весь аналіз вже відбувся раніше, у core та допоміжних модулях верхнього рівня. Аналогічно overview_view.py оперує вже готовим df, schema_json, parameters_list. Там дані приводяться до pandas для зручної стилізації і показуються у вигляді таблиці з параметрами та прикладами значень, але сам модуль не відповідає за те, як ці дані потрапили в пам'ять.

Інтелектуальна частина ізольована в ai/column_describer.py. Саме тут відбувається виклик Groq API і формування описів колонок. Код побудований так, щоб отримати з DataFrame тільки найнеобхідніше - назву, тип і кілька прикладів - і перетворити це на промпт для LLM:

```
def collect_columns_info(df, max_examples: int = 5) -> List[Dict]:
    if hasattr(df, "to_pandas"):
        pdf = df.to_pandas()
    else:
        pdf = df if isinstance(df, pd.DataFrame) else pd.DataFrame(df)

    columns_info = []
    for col in pdf.columns:
        series = pdf[col]
        dtype = str(series.dtype)

        non_null = series.dropna()
        examples = non_null.unique()[:max_examples].tolist()
        examples_str = ", ".join(str(x) for x in examples)

        columns_info.append(
            {
                "name": str(col),
                "dtype": dtype,
```

```
                "examples": examples_str,
            }
        )
    return columns_info
```

Далі цей список перетворюється на markdown-таблицю і відправляється в Groq. Важливий момент - ai/ не знає нічого про Streamlit. Повертається просто текст, а вже ui/ai_descriptions_view.py вирішує, як його відобразити, у якому форматі показати таблицю, як дати користувачу можливість відредагувати типи.

Верхній рівень взаємодії "склеює" ядро, AI та UI: df приходить із session_state, AI працює через окремий модуль, а результат живе в таблиці, яку можна редагувати прямо в інтерфейсі.

Окремою лінією йде Dash-прототип у папці sections та файл main.py. Він живе ніби поруч із основним Streamlit-додатком, але повністю окремо. У main.py створюється Dash-додаток, в layout збираються три секції: завантаження, визначення типів, аналіз дат.

На фоні всього цього обертається кілька службових модулів верхнього рівня. stats.py реалізує функції numeric_stats, string_stats, datetime_stats, які працюють з polars DataFrame і повертають або словники зі статистикою, або списки записів для побудови таблиць. heatmap.py формує df_res з показниками якості і повертає стилізований pandas.DataFrame, готовий до показу в Streamlit. preprocessing.py займається нормалізацією дат і містить досить довгий список регулярних виразів і форматів у DATE_FORMATS, щоб привести різні строкові представлення дат до datetime. settings.py зберігає айдішники сховищ для Dash. requirements.txt фіксує все оточення - від streamlit, dash, polars і pandas до statsmodels, scipy і groq-клієнта.

Усе це разом реалізує просту, але правильну ідею. Ядро (core, get_file_info, read_* , preprocessing, stats, heatmap) робить важку роботу з даними. Інтерфейс Streamlit (ui та app.py) лише керує сценаріями і показує результати. Інтелектуальний шар (ai) додає надбудову у вигляді текстових описів і висновків. Dash-прототип (main.py та sections) існує як окрема гілка, яка експериментує з

іншим підходом до UI, але використовує ті самі службові модулі. За рахунок такого поділу систему можна розширювати в різні боки - додавати нові вкладки в Streamlit, підключати інші AI-моделі, виводити частину логіки в окремі сервіси - при цьому не ламати вже перевірене ядро обробки файлів.

Streamlit як основний UI: докладний аналіз app.py :

У Streamlit-реалізації VizoraDate вся логіка взаємодії з користувачем фактично зібрана в одному файлі -app.py. Це не "моноліт" у грубому сенсі, а швидше центральний диригент, який підтягує все, що потрібно, з core/, ui/, ai/ та службових модулів, і збирає це в послідовний сценарій. На початку файлу підключаються як внутрішні модулі, так і зовнішні залежності: Streamlit, polars, pandas, plotly, statsmodels, scipy, а також утиліти для роботи з файлами та аналітикою. Саме тут задається базова конфігурація сторінки через st.set_page_config -заголовок, іконка, розкладка "wide" для комфортної роботи з таблицями, а також перелік усіх вкладок, між якими користувач зможе перемикатися. Цей список фіксується в константі TAB_LABELS і вже тут видно, що застосунок задуманий як набір окремих аналітичних панелей: "Огляд", "AI-опис полів", "Теплова карта", "Дублікати", "Унікальний ключ", "Числова статистика", "Рядкові дані", "Дата і час", "Візуалізація даних", "BI-дешборд", "AI-висновки по файлу".

Власне Streamlit-життєвий цикл починається з функції main(). Вона задає заголовок вікна, малює блок завантаження і запускає первинний конвеєр обробки файлу. Сценарій для користувача виглядає дуже лінійно: він бачить заголовок " 📊 Data аналіз", кнопку завантаження і одразу розуміє, що без файлу система далі нікуди не піде. У коді це досить прозоро:

```
def main():
    st.title("📊 Data аналіз")


    uploaded_file = st.file_uploader(
        "Завантажте файл для аналізу:",
```

```
            type=["csv", "json", "ndjson"]
        )

        if uploaded_file is not None:
            st.write(f"Файл: **{uploaded_file.name}**")

            if st.button("Аналізувати файл"):
                suffix = os.path.splitext(uploaded_file.name)[1]
                with tempfile.NamedTemporaryFile(delete=False, suffix=suffix) as
tmp:
                    tmp.write(uploaded_file.getbuffer())
                    temp_path = tmp.name

                process_file(temp_path, uploaded_file.name, st)

        if 'df' in st.session_state:
            analyze_data()
```

Тут добре видно, як app.py не лізе всередину логіки читання файлу. Він просто створює тимчасовий файл, передає шлях і оригінальну назву в core.file_loader.process_file, а далі переключається у режим "аналізу", якщо в st.session_state з’явився ключ df. Умовно кажучи, main() тут виступає воротами в систему: поки завантаження не відбулося, жодна з вкладок не активується.

Функція process_file (вже в core/file_loader.py) забирає на себе всю "чорну роботу" -визначає кодування, роздільник, тип файлу, читає CSV/JSON/NDJSON, формує DataFrame і заповнює st.session_state основними параметрами. Після її виклику в session_state гарантовано будуть df, filename, file_size, encoding, date_cols, а для JSON/NDJSON ще й schema_json, parameters_list та json_type. Саме на це спирається наступний рівень -analyze_data().

Коли користувач натиснув кнопку аналізу, завантаження пройшло без помилок і df з’явився в стані сесії, main() передає керування у analyze_data(). Ця функція вже не займається файлами, вона працює з готовим набором і формує "робочий простір" для всіх вкладок. Тут же відображається повідомлення про успішне завантаження і викликається render_metadata, який малює "паспорт" файлу. Важливо, що на цьому етапі система вже не звертається до файлової системи - усе потрібне збережено у session_state. Саме це дає змогу далі вільно перемикатися між вкладками, не перевантажуючи дані і не гублячи контекст.

Усередині analyze_data() якраз і знаходиться головний навігаційний блок. Спочатку система дивиться, чи не виставив хтось прапор switch_to_unique (його, наприклад, може встановити модуль аналізу дублікатів, якщо користувач натиснув кнопку "перейти до унікального ключа"), і за потреби примусово переключає активну вкладку на "Унікальний ключ". Далі виводиться радіо-перемикач на основі TAB_LABELS, і вибір одразу синхронізується з st.session_state["main_tab"].

Після вибору вкладки tab analyze_data() працює як звичайний диспетчер: просто передає керування в потрібний модуль.

Весь перелік сценаріїв аналізу зводиться до однієї послідовності if/elif, і в цьому немає нічого зайвого -app.py не намагається "знати" подробиці того, що відбувається всередині кожного екрану, він тільки натискає правильні кнопки.

Кожна вкладка тут -окремий "світ". Огляд (render_overview) показує структуру датафрейму, частину рядків, типи колонок, схему JSON/NDJSON, якщо вона є. "AI-опис полів" викликає інтеграцію з Groq через ai.column_describer, збирає приклади значень, формує промпт, відправляє в LLM і потім показує результат у вигляді редагованої таблиці. "Теплова карта" працює через модуль heatmap і відображає якість даних -пропуски, аномалії, проблемні значення. "Дублікати" аналізують повтори, дозволяють оцінити їх кількість і при необхідності від них позбавитися. "Унікальний ключ" перебирає комбінації колонок, шукає варіанти, які дають 100 % унікальності, показує проміжні результати у вигляді таблиці. Три статистичні вкладки ("Числова статистика",

"Рядкові дані", "Дата і час") використовують функції з stats.py і виносять на екран розподіли, базові метрики, ковзні вікна, структурні показники. "Візуалізація даних" і "BI-дешборд" працюють вже з більш звичними графіками й дашбордами на базі plotly. "AI-висновки по файлу" збирають усе це в один текстовий аналітичний блок -фактично короткий звіт по набору.

Логіка st.session_state у цьому файлі не обмежується тільки зберіганням df і main_tab. Один із важливих компонентів -історія AI-чатів по файлу. У нижній частині app.py оголошується шлях до локального JSON-файлу chat_history.json і декілька службових функцій, які зв’язують конкретний датасет із власною історією запитів до моделі. Спочатку історія піднімається з диску в session_state:

```
CHAT_HISTORY_PATH = "chat_history.json"


def _load_chat_history_into_session():
    if "file_ai_chat_history" in st.session_state and isinstance(
        st.session_state["file_ai_chat_history"], dict
    ):
        return

    if os.path.exists(CHAT_HISTORY_PATH):
        try:
            with open(CHAT_HISTORY_PATH, "r", encoding="utf-8") as f:
                data = json.load(f)
            if isinstance(data, dict):
                st.session_state["file_ai_chat_history"] = data
                return
        except Exception:
            pass

    st.session_state["file_ai_chat_history"] = {}
```

Після цього будь-який AI-блок, який працює з файлом (зокрема "AI-висновки по файлу" і діалогові сценарії), може отримати і зберегти історію саме для цього датасету. Ідентифікація набору робиться через стабільний dataset_key, де у хеш згортається форма таблиці, назви колонок і перші 100 рядків.

Такий підхід дозволяє розглядати кожен файл як окремий "діалоговий об’єкт": якщо користувач повернеться до нього пізніше, AI побачить попередні запити й відповіді, а не починатиме все з нуля. Збереження назад у chat_history.json виконується функцією _save_chat_history_from_session(), яка просто вивантажує словник із st.session_state["file_ai_chat_history"] на диск.

Окремо варто згадати, як у app.py використовується HTML та кастомний CSS. Частина стилізації виноситься в окремі файли (style.css або шаблонні HTML-фрагменти для BI-звітів), частина вплітається безпосередньо в текст, який передається у st.markdown з unsafe_allow_html=True. Наприклад, для побудови HTML-версії BI-дешборду створюється шаблон з вбудованим <style>.

Цей HTML може бути показаний прямо в інтерфейсі або використаний для експорту/збереження звіту. Аналогічні прийоми з кастомним CSS застосовуються в модулях ui, зокрема у metadata_view, де картки метаданих оформлюються через власні класи з тінню, відступами й адаптивною сіткою.

У підсумку картина виглядає так. app.py відповідає за те, щоб у користувача був один безперервний сценарій: завантажити файл, побачити метадані, пролистати огляд, перейти до статистики, заглянути в теплову карту, подивитися на дублікати й унікальні ключі, спробувати AI-описи і AI-висновки, а потім, за потреби, відкрити BI-дешборд. Увесь шлях робиться без перезавантаження файлу, без "втрати пам’яті" між вкладками і без дублювання логіки -за це відповідає комбінація st.session_state, TAB_LABELS, прозорої диспетчеризації всередині analyze_data() і акуратно винесених модулів у ui/, core/ та ai/. Користувач бачить набір вкладок, які логічно продовжують одна одну, а app.py залишається непомітним, але критично важливим координатором цього процесу.

Dash-прототип як модуль автоматизованої обробки полів:

У Dash-прототипі VizoraDate логіка організована вже не через Streamlit сесію, а через окремий застосунок на базі Dash, який живе у файлі main.py і кількох модулях у директорії sections. Якщо Streamlit у цьому проєкті відповідає за повноцінний інтерфейс аналітика, то Dash використовується скоріше як окрема лабораторія для "Системи автоматичної обробки полів", де тестуються сценарії автоматичного визначення типів колонок і початкового аналізу дат. Сам main.py завантажує Dash, створює застосунок з suppress_callback_exceptions=True, оголошує кілька dcc.Store для збереження стану та просто підшиває під себе секції з директорії sections. У розмітці це виглядає приблизно так: спочатку заголовок "Система автоматичної обробки полів", під ним три dcc.Store з ідентифікаторами на кшталт "df-store", "file-ext-store", "column-types-store", далі блок завантаження файлу з upload_section, блок для налаштування типів колонок з column_type_detection_section і під ним контейнер html.Div(id="tabs-container"), куди вже далі буде "підкладатися" контент аналізу дат. Власне Dash не зберігає DataFrame напряму - всі об'єкти, що рухаються між колбеками, серіалізуються в словники чи списки, тому dcc.Store у цьому контексті грає роль аналітичної пам'яті між кроками.

Секція upload_section у Dash повторює за змістом функцію завантаження у Streamlit, але робить це реактивно. Користувач бачить компонент dcc.Upload з підписом "Завантажте файл для аналізу", вибирає CSV або JSON, а далі в гру вступає колбек. Колбек прив’язаний до Input("file-upload", "contents") та State("file-upload", "filename") і повертає одразу два Output - в DATAFRAME_STORE_ID кладеться вже прочитаний набір даних, а в FILE_EXTENSION_FEATURE_STORE_ID зберігається службова інформація про розширення і деякі допоміжні ознаки. Всередині функції обробки здійснюється стандартна для проєкту операція: вміст, що прийшов з фронтенду, декодується з Base64, тимчасово кладеться в пам'ять у вигляді файлу, далі для нього

викликаються ті самі утиліти, що використовуються у Streamlit - модулі read_csv або read_json, а логіка визначення кодування чи роздільника делегується в get_file_info. Це важливий момент: ядро читання файлів не дублюється в Dash, воно спільне для обох інтерфейсів, тож Dash лише "натискає" той самий пайплайн, але з іншої UI-обгортки.

Коли дані вже опиняються в DATAFRAME_STORE_ID, їх підхоплює column_type_detection_section. Тут Dash-підхід відчувається найбільше, бо саме в цій секції для кожної колонки формується набір керуючих елементів: випадаючий список для типу поля, текстовий інпут для приміток або прикладів, додаткові контролери за потреби. У шаблоні використовуються ідентифікатори у форматі словників, наприклад {"type": "column-type", "index": col_name} та {"type": "column-input", "index": col_name}. Завдяки цьому Dash-колбеки можуть працювати не з жорстко прописаними айдішниками, а з групами компонентів через механізми ALL і MATCH. Збереження результатів користувацького мапінгу робиться в одному колбеку з кількома State, де список типів, список id селектів та список значень інпутів збираються докупи, після чого формується словник {column_name: detected_type}. Цей словник і потрапляє в COLUMN_TYPES_STORE_ID, який далі слугує вхідною точкою вже для аналізу дат. Така побудова дозволяє, з одного боку, автоматично поставити попередні типи (на базі евристик чи аналізу прикладів), а з іншого - не закривати можливість ручної корекції.

Секція date_analysis_section працює вже поверх двох сховищ - DATAFRAME_STORE_ID і COLUMN_TYPES_STORE_ID. В її колбеках у Input заходять обидва ці store, а в Output("tabs-container", "children") повертається розмітка, яка представляє результати аналізу дат. Тут Dash-прототип реалізує спрощену версію того, що у Streamlit винесено в окрему вкладку "Дата і час": спочатку виділяються колонки, які згідно з мапінгом користувача мають тип "date" або "datetime", далі з DataFrame формується кілька агрегованих серій, будується базова статистика (мінімальні та максимальні дати, кількість спостережень, вікно

охоплення), після чого генеруються кілька графіків. Для прикладу можна згадати типовий фрагмент, де для однієї часової колонки створюється розподіл по місяцях: полем дати виступає df[date_col], з нього обчислюється колонка month = df[date_col].dt.to_period("M"), далі побудовується value_counts по month, а на фронтенд повертається dcc.Graph з барчартом. У коді це виглядає як звичайний Dash-компонент з figure=go.Figure(...), де слоти даних заповнені згідно з підрахованими агрегатами.

Загальна архітектура колбеків у цьому прототипі дуже показова: upload_section пише дані в DATAFRAME_STORE_ID, column_type_detection_section читає цей store, виводить керуючі елементи для колонок і зберігає мапінг типів у COLUMN_TYPES_STORE_ID, а date_analysis_section читає обидва store і, вже маючи і структуру даних, і уявлення про типи, малює часову аналітику. Тобто замість глобального st.session_state тут використовується зв'язка dcc.Store + Input/Output. Усе, що рухається по цьому ланцюгу, має бути серіалізованим, тому DataFrame, як правило, перетворюється на щось на кшталт df.to_dict(orient="records"), і навпаки, на боці бекенда назад збирається через pd.DataFrame.from_records(...) або аналогічну конструкцію. Це накладає свої обмеження, але одночасно робить Dash-прототип доволі ізольованим від деталей зберігання стану на сервері.

Технічно обидва інтерфейси спираються на спільне середовище. У файлі requirements.txt поряд зі streamlit вказано dash, polars, pandas, plotly, statsmodels, scipy, клієнт для Groq та інші бібліотеки, пов'язані з обробкою даних і побудовою графіків. Віртуальне середовище створюється стандартним чином через venv, і цей самий набір залежностей використовується як для запуску Streamlit додатку (streamlit run app.py), так і для Dash (python main.py або через вбудований сервер). Конфігурація секретів зосереджена у .streamlit/secrets.toml: тут зберігається ключ для Groq API та інші параметри, які критично не можна тримати в коді. Streamlit отримує цей ключ через st.secrets["GROQ_API_KEY"] у модулі інтеграції з LLM, а

Dash-прототип теоретично може використовувати ті ж самі службові функції, хоч у поточній версії AI-компоненти явно прив'язані саме до Streamlit-частини.

Окремим елементом, який об'єднує архітектуру в цілому, виступає файл chat_history.json. Він використовується в основному UI для збереження історії AI-взаємодії з конкретними датасетами: в app.py існує логіка, яка при старті сесії завантажує цей файл у st.session_state["file_ai_chat_history"], а при завершенні або при зміні історії - знову скидає оновлений словник на диск. Ідентифікація набору, як уже згадувалося, робиться через стабільний ключ, що обчислюється за формою таблиці та першими рядками. Dash-прототип поки явно не підключений до цього механізму, але архітектурно нічого не заважає у майбутньому використовувати той самий файл як спільне сховище AI-контексту, просто через інший шлях доступу (наприклад, через утиліти з окремого модуля, який буде однаково доступний і Streamlit, і Dash).

Якщо дивитися на систему цілісно, видно, що Streamlit і Dash тут не конкурують, а доповнюють один одного. Основний сценарій аналізу даних, побудови теплових карт, дублікативного аналізу, AI-описів і BI-дешборду реалізований у app.py та модулях ui/ і спирається на st.session_state як на єдине джерело правди. Dash-прототип у main.py і sections/ працює як майданчик для відпрацювання більш вузького, але глибокого кейсу - "Система автоматичної обробки полів": тут опрацьовується механіка визначення типів колонок, передачі структур через dcc.Store, складання кількох залежних колбеків і побудови окремих графіків по датах. Обидві гілки тримаються на одному ядровому коді для читання та попередньої обробки файлів, на спільному наборі залежностей і, за необхідності, можуть обмінюватися результатами через прості, зрозумілі формати на диску (CSV, JSON). Саме за рахунок такого поділу VizoraDate виглядає не як набір випадкових скриптів, а як архітектурно продуманий прототип, в якому є основний "фронт" для аналітика і окрема зона, де можна тестувати складніші інтерфейсні та алгоритмічні рішення.

3.2 Модулі завантаження та первинної обробки даних

У VizoraDate перший контакт користувача з системою завжди починається однаково - з моменту, коли він піднімає локальний файл у інтерфейс. І хоч на поверхні це виглядає як простий “file_uploader”, за ним працює доволі акуратно побудований конвеєр. Streamlit у цьому сенсі поводиться як надійний диспетчер: він утримує файл у тимчасовому буфері, дозволяє прочитати його як байтовий потік, однак не бере на себе жодної відповідальності за те, що буде далі. Саме тому вся справжня логіка завантаження опиняється в модулі core/file_loader.py, а app.py реагує радше як фасад - він просто передає те, що отримав, у спеціалізоване ядро.

На початку app.py вікно пропонує користувачу завантажити CSV, JSON або NDJSON. Потім, коли файл потрапляє у пам’ять, Streamlit створює тимчасовий файл через стандартний механізм tempfile.NamedTemporaryFile. Це потрібно не так для зручності, як для стабільності: багато утиліт Polars і Pandas все ще працюють на основі файлових шляхів, а не потоків. У цей момент у розпорядженні системи опиняється реальний файл на диску, і він готовий для передачі в конвеєр.

Модуль file_loader.py - це вже повноцінна логіка. Усередині process_file система починає з простого, але критичного кроку - визначення формату та первинної інформації про файл. Саме для цього викликається get_file_info, який досліджує кодування, розмір, тип структури, а також, у випадку CSV, визначає потенційний роздільник. Тут усе побудовано навколо реальних викликів користувачів: файли приходять у Windows-кодуванні, із BOM-мітками, з dividers у вигляді “;”, іноді з десятками тисяч рядків. Конвеєр має витримати це, не впавши в першу ж помилку.

Сама логіка get_file_info виглядає приблизно так: спочатку відкривається файл у режимі читання байтів, з нього вирізаються перші кілька кілобайт, і по цим даним визначається кодування через chardet. Далі - окрема гілка для CSV: невеликий фрагмент файлу проганяється через набір евристик, які визначають роздільник. Логіка проста, але вона закриває більшість реальних кейсів.

Отримавши кодування, приблизний формат і розмір, process_file вирішує, у який із двох спеціалізованих модулів передавати дані - read_csv чи read_json. Розгалуження відбувається за розширенням файлу, і хоча це здається очевидним, у проєкті є також окремий обробник NDJSON, який прихований у read_json як режим “построчного читання”. Тут, у read_json, логіка побудована так, що на виході система отримує вирівняну таблицю, навіть якщо вхідний JSON історично не був таблицею, а скоріше масивом вкладених об’єктів. Такий підхід дозволяє працювати з даними з API, з лог-файлами й зі складними структурами, які складаються з масивів словників.

Після завершення читання файл перетворюється на DataFrame у Polars або Pandas, залежно від сценарію. І хоча Polars використовується для швидкості, усі UI-модулі у Streamlit у кінці все одно отримують Pandas-версію, тому file_loader додатково виконує .to_pandas() перед записом у сесію. У цьому місці він також виконує попереднє визначення дат - сканує кожну колонку й намагається знайти ті, значення яких можуть бути перетворені в datetime.

Зібравши цю інформацію, process_file переходить до найважливішого кроку - заповнення st.session_state. У цьому проєкті session_state стає єдиним джерелом правди, тому в нього записується не тільки сам DataFrame, а й:

- filename -справжня назва файлу;
- file_size -розмір у байтах;
- encoding -визначене кодування;
- delimiter -якщо файл CSV;
- json_type -якщо файл JSON, а структуру вдалося класифікувати;
- schema_json -схема вирівняного JSON;
- date_cols -список колонок, попередньо визначених як дати.

Саме ці дані згодом відображаються у модулі metadata_view. Він не виконує логіки, але виступає як "паспорт" набору, де картки з HTML/ CSS стилями акуратно підсвічують структуру файлу.

У момент, коли process_file завершується, система переходить у режим, який можна назвати "дані вже в пам’яті". Дані більше не читаються з диску, а всі UI-вкладки працюють поверх одного й того ж об'єкта в session_state. Така організація суттєво знижує навантаження на пам'ять і час відповіді: файл читають один раз, а далі просто звертаються до збережених параметрів. І хоч на рівні інтерфейсу користувач бачить лише коротке повідомлення "Файл успішно завантажено", насправді в цей момент у системі вже відбулося кілька критично важливих операцій - від ідентифікації структури до повноцінної превалідації.

Після того, як файл пройшов через первинні перевірки в get_file_info, система переходить до найважливішого -перетворення вмісту на коректний, стабільний DataFrame. У VizoraDate ключова особливість полягає в тому, що тут немає двох паралельних кодових баз для CSV і JSON -усе зведено до кількох чітких модулів (read_csv.py, read_json.py) і одного керуючого центру (file_loader.py). Саме там відбувається вибір, куди направити подальшу обробку.

У випадку з CSV все починається з того, що read_csv.py відкриває файл через Polars, оскільки ця бібліотека значно швидша і точніша у роботі з неочевидними типами.Polars намагається автоматично розпізнати дати або хоча б конвертувати формат у щось однорідне. Якщо Polars не впорається, у подальшому ці колонки будуть перечитані вже через Pandas або через власні утиліти. Але важливо, що CSV-обробник тут працює за принципом “спочатку швидко, потім точно”, і це дає гарний ефект: навіть дуже великі файли обробляються без помітних затримок.

Одразу після читання CSV-таблиця перетворюється на Pandas-версію, адже саме Pandas очікують модулі у Streamlit-UI, зокрема всі статистичні обчислення зроблені на ньому.

Одразу після конвертації декілька колонок проходять перевірку на datetime, і якщо формат хоч якось виглядає консистентно, вони відразу переводяться в datetime64[ns]. Саме тут формується список date_cols, який потім використовують як Streamlit-модулі статистики дат, так і AI-опис.

З JSON усе значно складніше, і в read_json.py це добре видно. Проблема JSON у тому, що він може бути таблицею лише “в теорії”. На практиці це або масиви вкладених словників, або лог-файли, або NDJSON із різними ключами в кожному рядку. Саме тому модуль починається з визначення “типу JSON”: він читає перші кілька рядків, аналізує їх, і вирішує, з чим має справу. У спрощеному вигляді це виглядає так:

- якщо структура - масив словників, тоді робимо flatten і збираємо таблицю;
- якщо це NDJSON (рядок = окремий JSON-об’єкт), читаємо построково, вирівнюємо кожен об’єкт під загальну схему;
- якщо всередині вкладені масиви - пробуємо розкривати тільки перший рівень.

Серце read_json.py - фаза flatten. Вона робить з вкладених JSON-структур щось, що можна подати у DataFrame, тобто будує ключі у вигляді field.subfield.subsubfield.

NDJSON - окремий випадок. Тут модуль читає файл рядок за рядком, кожен рядок пропускає через json.loads, flatten-ить і одразу додає до списку записів.

Так з'являється список рівних словників, з яких потім без проблем будується Pandas-таблиця. Саме такий підхід дозволяє працювати з napari-логами, telemetry events і всіма структурами “з реального світу”, де структурної чистоти зазвичай немає.

Коли JSON прочитаний, модуль формує так звану schema_json - просту, але дуже важливу структуру, де перераховані всі ключі та можливі типи значень. Її пізніше показує metadata_view як складову паспорту набору. Якщо JSON був глибоким або нерівномірним, саме схема показує користувачу “де болить”: які ключі холості, які колонки sparsely-populated, де вкладеність надмірна.

У CSV-обробнику і JSON-обробнику є одна спільна деталь - обидва виконують попередню валідацію: шукають порожні колонки, NaN, незрозумілі текстово-числові колонки, і хоча модуль не виправляє дані, він принаймні реєструє

все це в session_state, щоб потім UI міг підсвітити це у вигляді попередження. Зокрема get_file_info рахує кількість рядків і колонок, що дозволяє одразу відобразити обсяг даних.

І вже після цього система завершує всю обробку нормалізованою формою - DataFrame, список колонок, попередні типи, схему, формальну інформацію про файл.

3.3 Модулі структурного та статистичного аналізу даних

Перший шар аналітики у VizoraDate починається не зі статистики, а зі своєрідного “ознайомчого погляду”, який користувач отримує на вкладці, що будується модулем ui/overview_view.py. Це наче коротка зустріч із даними: система не намагається їх інтерпретувати, не оцінює складність, але показує достатньо, щоб сформувати первинне розуміння структури. Саме тому в інтерфейсі, відразу після метаданих, з’являється кілька ключових блоків: частина таблиці, типи колонок, приклади значень, попередня інформація про пропуски й оцінка однорідності.

У overview_view.py логіка побудована доволі прямо. Спочатку модуль дістає DataFrame зі st.session_state, і тут важливо, що він не використовує жодних проміжних копій — усе працює поверх того самого об’єкта, який було прочитано на етапі file_loader. Далі виводиться кілька перших рядків таблиці. Ця частина виглядає дуже просто, але за цією простотою прихована важлива деталь: огляд завжди показує саме ті типи колонок, які Pandas виявив після нормалізації. У випадку CSV Polars уже міг частково відпрацювати дату як datetime, але справжня типізація відбувається саме тут, тому користувач відразу бачить, де система визначила дату, де текст, де число.

Наступний блок —опис колонок. Він будується у вигляді таблички: ліворуч назва колонки, праворуч тип. Якщо значення у колонці змішані (наприклад, числа з рядками), модуль пробує самостійно виявити, що колонка непослідовна, і підсвічує це у вигляді помітки. І хоча в інтерфейсі цей момент виглядає доволі

буденно, він має велике значення для подальших етапів: модулі статистики вибирають лише чисто числові колонки, модулі аналізу рядків — лише string, а все, що не вписується в типову матрицю, ігнорується з попередженням.

Після цього overview_view переходить до блоку пропусків. Саме ці значення потім стають видимими як міні-таблиця зі стовпцями “Колонка”, “Пропусків”, “% від загальної кількості”. Така візуальна форма дозволяє дуже швидко оцінити, чи містить набір критичні проблеми, чи пропуски локальні, чи рівномірні. Більш того, якщо пропуски розподілені нерівномірно або в колонці спостерігається щось схоже на деградацію якості (наприклад, блок рядків із великою кількістю NaN), то ця інформація візуально зчитується одразу на етапі перегляду, ще до переходу у теплові карти.

Огляд включає також приклади значень. Для кожної колонки модуль вибирає кілька записів, що дозволяє зрозуміти контекст: наприклад, у колонці "city" це можуть бути реальні назви міст, у колонці з датою — приклади форматів, і так далі. Тут немає жодної складної логіки, але саме така “людська” підказка дозволяє швидко помітити випадки, де колонка, що виглядає як дата, насправді містить щось зовсім інше (наприклад, "0000-00-00", "-", "не вказано"). Такі нюанси AI і статистика помічають пізніше, але користувач бачить їх уже тут.

Одразу після перегляду структури користувач може перейти до теплової карти, яка будується модулем heatmap.py. У Streamlit це виглядає доволі скромно — ніби просто графік, але всередині використовується комбінація Pandas та Plotly, де пропуски ідентифікуються як бінарна матриця.

Функція підключається до Plotly Heatmap, де для пропусків задано одну палітру, а для заповнених значень — іншу. Результат — характерна шахматка, де одразу видно:

- колонки з великою кількістю порожніх значень;
- ділянки, де пропуски сконцентровані в певних рядах;
- можливі “провали” у структурі даних (наприклад, масове зникнення значень у певний період).

І хоча сама теплокарта — це лише частина модуля, саме вона найкраще підсвічує глибинні структурні проблеми.

Ще одна важлива деталь, яку формує overview_view, — це схема JSON, якщо вихідний файл був JSON або NDJSON. Тут система не реконструює JSON вручну, а використовує schema_json, попередньо підготовлену в read_json. У результаті в інтерфейсі з’являється форматований JSON-блок, де видно ключі, вкладеність, типи полів.

Це дуже важлива частина аналізу: користувач бачить, чи JSON був чистим, чи його довелося вирівнювати, чи є вкладені поля, які не були розкриті автоматично.

Усі ці елементи — огляд, приклади, пропуски, схема — разом формують першу ділянку аналітики у VizoraDate. Вони не виконують складних обчислень, але саме тут стає зрозуміло, з чим користувач працюватиме далі. Це фактично "аналітична розминка" перед тим, як перейти до статистики, кореляцій, часових аналізів і AI-модулів.

Після того, як користувач переглянув структуру набору й отримав первинне уявлення про дані, інтерфейс VizoraDate переходить на наступний рівень — глибший статистичний зріз. Уся ця частина побудована навколо модуля stats.py, який, по суті, є математичним ядром системи, і кількох великих UI-модулів у ui/stats_views/. Вони працюють разом: stats.py обчислює показники, а Streamlit уже відповідає за те, як ці дані виводяться користувачу.

У stats.py усе починається з того, що дані діляться на три великі групи: числові, рядкові та часові колонки. Це робиться не “магічно”, а через звичайну логіку Pandas: df.select_dtypes(include="number") або аналогічні фільтри. Таке розділення необхідне, бо статистика для кожної групи суттєво відрізняється, і на практиці це допомагає уникати змішування методів. Наприклад, числові поля відразу можуть бути проаналізовані через df.describe(), де доступні mean, median, std, мінімум, максимум, квартилі, тоді як для рядкових колонок це неможливо — там важливі зовсім інші показники.

Логіка числової статистики у VizoraDate виглядає просто, але під капотом вона охоплює досить багато задач. Для кожної числової колонки система формує табличку, де зведені базові характеристики:

- середнє значення;
- медіана;
- стандартне відхилення;
- мінімум і максимум;
- кількість унікальних значень;
- кількість пропусків.

Хоча більшість із цих даних дає df.describe(), модуль додає кілька додаткових параметрів, що корисні для оцінки “розкиданості” і стабільності колонки: наприклад, коефіцієнт варіації (CV), який розраховується як std / mean. Для колонок, де CV наближається до нуля, значення майже не змінюються, а якщо він великий — варіація суттєва, і такі поля мають знати аналітики.

У Streamlit-інтерфейсі ці таблиці виводяться через st.dataframe, але з певною стилізацією: найчастіше у верхній частині показано короткий опис колонки (відомий тип, кількість пропусків, відсоток заповненості), після чого йде сам блок статистики. Виглядає це як послідовність компактних секцій, кожна з яких містить міні-заголовок, таблицю і маленькі примітки, якщо дані виявили аномалії (наприклад, “значення у верхніх 0.1% сильно відрізняються від решти”).

Другий важливий аспект — кореляції. У stats.py передбачено обчислення кореляційної матриці через df.corr(), але обчислення робиться тільки для числових колонок, і це принципове рішення: метод Пірсона працює виключно з числовими значеннями, і VizoraDate дотримується цієї методологічної чистоти. Відповідно, якщо у наборі кілька десятків числових колонок, результат являє собою матрицю k × k, де кожен елемент — це значення кореляції між двома змінними. Для більш загальних випадків (наприклад, коли розподіли асиметричні або містять повільні тренди) модуль може використовувати коефіцієнт Спірмена, але він підключається

лише тоді, коли дані явно не підходять для Пірсона — ця логіка живе в окремій гілці функцій.

Кореляційна матриця у візуальному представленні — це вже робота heatmap.py. Теплокарта будується через Plotly, і хоча код виглядає лаконічно, він дає дуже якісну картинку. Загальний принцип такий: модуль приймає матрицю значень та список колонок, після чого формує:

```
fig = px.imshow(
    corr_matrix,
    labels=dict(color="Кореляція"),
    x=columns,
    y=columns,
    color_continuous_scale="RdBu_r"
)
```

Палітра “RdBu_r” обрана не випадково — вона яскраво підсвічує негативні та позитивні зв’язки і дозволяє швидко зловити аномальну поведінку даних: два поля, що змінюються синхронно; колонки, які виявилися майже лінійними копіями одна одної; слабкі, але стабільні залежності, помітні в часі. Для аналітика це дуже цінний елемент, бо він дозволяє оцінити, чи дані структуровані природним чином, чи в них є приховані артефакти.

Ще один важливий блок статистики — це розподіли. У VizoraDate окремо генеруються гістограми та boxplot-и для числових колонок.

Boxplot будується через px.box, і тут особливо важливо, що він чітко позначає викиди. У даних, які приходять у реальних кейсах, викиди зустрічаються набагато частіше, ніж хочеться, і правильно побудований boxplot дає змогу моментально побачити ці “сполохи”.

Для рядкових колонок логіка інша — тут найбільшу цінність дають показники частоти. У stats.py реалізовано аналіз топ-n значень: по суті, це серія value_counts().head(10). Далі ui/string_stats_view.py перетворює це на компактну

таблицю, де можна побачити “розподіл популярності” значень. Це один із найважливіших елементів, якщо користувач аналізує категоріальні поля: місто, категорія товару, тип події тощо.

Усі ці статистичні блоки об’єднані в одну логічну систему. Користувач переходить на відповідну вкладку — наприклад, "Числова статистика" — і отримує повний портрет усіх числових колонок: від базових характеристик до розподілів та аномалій. Аналогічно для рядкових даних. Що важливо — усі модулі виводять інформацію у “людинозрозумілому” форматі. Наприклад, якщо виявлена надмірна кількість унікальних значень у колонці, яка має бути категоріальною, система не мовчить — у вигляді м’якого попередження вона натякає, що колонка може бути неправильно інтерпретована або потребує ручного перегляду.

У цьому ж контексті працює і модуль теплової карти пропусків. Він не просто позначає NaN, а дає змогу побачити патерни: суцільні вертикальні лінії (коли вся колонка має пропуски), суцільні горизонтальні (коли певні групи рядків неповні), або “розсіяні” пропуски, які зазвичай з’являються через SES/відтворення даних із нестабільних джерел.

Уся статистика, що виходить із stats.py, — це не абстрактні числа, а практично корисний матеріал, який стає основою для AI-висновків і подальших модулів аналізу. Тому цей розділ архітектури критично важливий: він закриває питання якості, дає змогу “промацати” дані, відчути їхню текстуру і структуру, а не просто побачити таблицю.

У VizoraDate модуль структурного аналізу має ще один ключовий компонент — це ui/metadata_view.py. Його роль часто недооцінюють, але насправді саме з нього починається «розуміння» набору даних у широкому сенсі. Якщо overview_view показує структуру таблиці, а stats.py формує числову сутність даних, то metadata_view — це своєрідний паспорт, довідник, базова документація. Він відповідає за те, щоб користувач зрозумів не стільки зміст таблиці, скільки її природу: звідки вона взялася, який вона має формат, наскільки вона велика, наскільки чиста, чи містить вкладені структури.

Сам модуль працює дуже просто: він дістає зі st.session_state усе, що було зібрано на попередніх етапах конвеєра завантаження, і перетворює це на набір чітко оформлених карток. Там немає жодної логіки обчислень — усі значення вже були визначені у file_loader.py та get_file_info.py. Але саме оформлення робить цей екран важливим: дані показані не як абстрактні ключі, а як окремі блоки, які легко прочитати.

Саме ці картки формують весь «паспорт». Окремо показана назва файлу, під нею — формат. Потім кодування, яке система визначила автоматично; розмір у мегабайтах чи кілобайтах; кількість колонок і рядків (причому в реальних даних це має величезне значення, бо саме тут користувач часто помічає, що завантажив файл з мільйонами рядків, і йому варто очікувати триваліших обчислень); список колонок-дат, якщо такі були знайдені на вході; і, нарешті, схема JSON, якщо набір мав саме таку структуру.

Схема JSON у цьому контексті — надзвичайно важлива частина. Коли дані прийшли у форматі JSON або NDJSON, metadata_view відображає schema_json у форматованому вигляді. Це робиться за допомогою st.json, що дозволяє згорнути/розгорнути розгалужені структури і фактично побачити всю логічну карту даних: які ключі вкладені, які типи значень відповідають кожному ключу, чи структура рівномірна або містить пропуски. Для багатьох користувачів саме ця частина інтерфейсу стає першим справжнім “відкриттям”, бо JSON у сирому вигляді виглядає вкрай нечитабельно, а тут його перетворюють на акуратне дерево.

За сценою цей модуль працює у тісній зв'язці з session_state: сам metadata_view нічого не розраховує, він просто дістає значення filename, file_size, encoding, date_cols, delimiter, json_type, schema_json і викладає їх послідовно. Завдяки тому, що усе це було заздалегідь сформовано у file_loader.py, відображення метаданих не спричиняє жодних обчислювальних навантажень і працює моментально. Паралельно це гарантує консистентність: усі вкладки системи читають один і той самий набір параметрів, ніхто нічого не переобчислює, і структура не “пливе” між різними екранами.

Візуальна частина також має значення. У metadata_view функція st.markdown використовується не просто як метод для виводу тексту, а як спосіб створення стильної сітки карток. Блоки розташовані у ряд або плитково (залежить від ширини екрана), усі вони мають однакові відступи, рамку, фон, і в результаті користувач отримує дуже чисту, структуровану панель. Зовні вона виглядає схоже на аналітичні інструменти корпоративного рівня — саме через цю міні-сітку метаданих, яка справляє враження завершеного, продуманого продукту.

Деякі з параметрів метаданих відіграють технічну роль у подальших модулях. Наприклад, date_cols використовується статистичним блоком для формування часових розподілів; encoding може бути потрібним у модулях, що стосуються валідації символів; delimiter — у візуалізації і в AI-описі; json_type впливає на те, як AI формує огляд структури. Але в інтерфейсі користувач бачить їх як окремі деталі — невеликі, але необхідні фрагменти загальної картини.

Саме тому метадані — це більше, ніж вступ. Це основа, яка дає користувачу впевненість, що система коректно “зрозуміла” файл. Якщо тут щось виглядає неправильно — неправильний формат, невірне кодування, підозріла кількість колонок — користувач може відреагувати ще до того, як перейде до глибоких модулів. І в цьому сенсі metadata_view виконує майже діагностичну функцію: він зчитує, чи вхідний конвеєр відпрацював коректно, і чи набір у тому вигляді, в якому він подається в аналіз, відповідає реальному стану даних.

Разом із overview_view і блоками статистики metadata_view формує цілісну систему. Це три опорні точки, через які користувач послідовно переходить від технічної інформації → до структурної → до статистичної. Система не примушує його одразу заглиблюватися у графіки чи AI-висновки — вона дозволяє спочатку “прочитати” набір через чисті, прості характеристики. Саме така поступова архітектура робить VizoraDate зрозумілою навіть тоді, коли дані великі, неоднорідні або складні.

3.4 Інтелектуальні компоненти VizoraDate: AI-генерація описів та пошук унікальних ключів

У VizoraDate модуль AI-описів колонок став тим елементом, який суттєво змінив характер системи: дані перестають бути просто структурою, і набувають рис осмисленого об’єкта. Звичайно, користувач бачить таблиці, розподіли, статистику, але справжню інтерпретацію — що саме означає колонка, чи можна вважати її категоріальною, яка в неї семантична роль — забезпечує саме AI. І те, як це реалізовано в ai/column_describer.py, доволі показово: модуль побудований дуже практично, без зайвих фантазій, як утиліта, що бере приклади і перетворює їх на якісний природний опис.

У центрі логіки — інтеграція з Groq API, зокрема з моделлю llama-3.1-8b-instant. Проєкт використовує швидку модифікацію моделі, оскільки ця задача не потребує глибинного reasoning, але потребує стабільних, компактних і семантично точних відповідей. API-ключ зберігається у .streamlit/secrets.toml, тому ні у коді, ні у репозиторії він не лежить відкритим. Сам виклик моделі формує payload зі system_prompt і user_prompt. Тут особливо цікаво те, як формується саме user_prompt. Модуль бере колонку з Pandas DataFrame, вибирає з неї невеликий набір значень — не всі, не занадто багато, а невеликий дистилят, зазвичай 8–12 рядків, — і передає їх у модель як контекст. Обрізання прикладів зроблено дуже продумано: щоб уникнути шуму, модуль вибирає унікальні, різні за формою значення, і намагається представити модельові той діапазон, з яким реально доведеться працювати.

Після цього формується user_prompt, який виглядає як компактний, але змістовний запит. Він пояснює моделі: “ось колонка, ось її приклади, дай короткий огляд змісту, типу, можливого призначення”. І саме цей момент визначає успішність результату: VizoraDate не просить модель вигадувати зайве, не вимагає прогнозів чи розлогих статей — лише коротку, практичну характеристику.

У system_prompt прописана роль моделі: бути технічною, стислою, але при цьому корисною для аналітика. І модель дотримується цього напряму. Завдяки цьому відповіді виглядають живими, але не перевантаженими. Наприклад, якщо колонка — список міст, AI сформулює щось на кшталт: “Колонка містить назви населених пунктів; можна розглядати як категоріальне поле. Висока різноманітність значень.” Якщо це дата — модель підкаже, що значення мають часову структуру, а якщо це ідентифікатор — AI відзначить стабільність формату і відсутність дублювання в прикладах.

Уже в ui/ai_descriptions_view.py AI-відповідь перетворюється на акуратну таблицю. Структура блоку така: ліворуч — назва колонки, праворуч — короткий опис. Якщо модуль автодетекції типів (detect_type_from_samples) визначив тип, користувач бачить два результати поруч: “AI-опис” і “Автоматично визначений тип”. Це дає змогу порівняти і побачити, чи обидва ці механізми узгоджуються. Саме так формується задумана синергія: модель описує зміст, а внутрішня логіка — структуру, і якщо вони сходяться, це сигнал, що колонка зрозуміла системі.

В інтерфейсі блок виглядає доволі елегантно: кожен опис виділений у підкреслений контейнер, а колонка і опис розташовані поруч. Тут використовується st.markdown з кастомним HTML, і візуально це нагадує аналітичний звіт чи аудит даних.

До речі, модуль автодетекції типів, хоч і працює просто, але виконує важливу функцію. Він бере приклади значень, проганяє їх через кілька перевірок (isnumeric? datetime? string pattern?) і відносить колонку до категоріальних, числових, datetime або текстових. Це дозволяє системі оцінювати, чи AI правильно зрозумів зміст, і позначати потенційні невідповідності. Наприклад, якщо AI описує поле як числовий код, але автодетектор бачить там змішані строки, це сигнал, що колонка, можливо, містить артефакти або має помилки введення.

Усе це разом формує дуже точний, завершений механізм. Важливо, що він працює statefully: результати AI-запитів зберігаються у session_state, тому переходи між вкладками не викликають повторних звернень до API. Це знижує затримку і

навантаження, а користувач у будь-який момент може переглянути AI-опис для всіх колонок одразу, без повторного завантаження чи перезапуску.

У підсумку цей модуль перетворює VizoraDate на систему, яка не просто “показує” дані, а допомагає розібратися з їхнім змістом. Тобто він фактично створює інтелектуальну документацію, яку інакше довелося б писати вручну. За рахунок цього користувач отримує унікальну можливість швидко зрозуміти набір навіть тоді, коли він великий, нечітко структурований або взагалі незнайомий.

Пошук унікальних ключів, аналіз дублікованих записів :

У системах аналізу даних унікальний ключ — це не просто технічна деталь, а фактично «хребет» набору. Якщо він є і працює коректно, дані поводяться передбачувано; якщо його немає або він зіпсований дублями, будь-який подальший аналіз починає хитатися. Саме тому у VizoraDate модуль ui/unique_key_view.py посідає окреме місце: він не лише знаходить унікальні колонки, а й працює як діагностичний інструмент, що дозволяє оцінити чистоту набору.

На вході модуль отримує Pandas DataFrame, збережений у st.session_state["df"]. І хоча всі попередні модулі могли працювати з Polars або частково трансформованими структурами, тут завжди працює саме Pandas, бо саме він дає просту і надійну апроксимацію унікальності через nunique().

Перший крок — перевірка поодиноких колонок. Модуль проходить усі стовпці таблиці і для кожного обчислює:

```
df[col].nunique() == len(df)
```

Але реальний набір рідко має один-єдиний простий ключ. Саме тому друга частина логіки — пошук комбінацій. Модуль перебирає пари колонок, а у разі потреби — трійки і тд. Саме ця операція виявляє приховані ключі, які неочевидні візуально. У таблицях з подіями це може бути комбінація event_id + timestamp, у таблицях продажів — order_id + line_number. Модуль будує всі можливі пари, перевіряє їх, і якщо комбінація справді унікальна, додає її в список кандидатів. Для

невеликих датасетів він може піти і на рівень трійок. Для великих — це обмежено, щоб не створювати вибуху обчислень.

Важливо, що алгоритм працює максимально «обережно». Він не намагається вгадувати логіку таблиці, не робить припущень, а перевіряє математичну властивість: чи може ця комбінація забезпечити унікальність рядка. Таке суворе правило дає дуже практичний результат: користувач бачить тільки ті комбінації, які справді можуть бути первинним ключем, а не які «виглядають підозріло». І це кардинально відрізняє VizoraDate від інструментів, що працюють на евристиках.

У модулі є ще один важливий крок — аналіз повних дублікатів. Під “повними дублями” тут маються на увазі рядки, де збігаються абсолютно всі значення по всіх колонках. Після цього система визначає, скільки саме дубльованих рядків існує, і виводить коротку аналітику у вигляді стилізованого блоку.

Потім з’являється можливість видалити дублікати. Тут модуль поводиться напрочуд обережно: він не змінює набір без підтвердження, і після видалення одразу НЕ перезаписує df на диску; замість цього він оновлює session_state.

Оновлення у session_state автоматично провокує перерахунок усіх залежних модулів — статистичних, AI-описових, структурних. Це дає дуже логічний ефект: якщо користувач видаляє дублікати, система «перепрочитує» весь набір як новий стан і перебудовує всі метрики відповідно. Такі моменти критично важливі у аналітичних інструментах, які працюють statefully, оскільки гарантують цілісність даних.

Коли дублікати усунено, модуль повторно обчислює кандидатів на ключ. Це особливо корисно в наборах, де дублікати були просто артефактами злиття даних чи логування. Буває так, що колонка не була унікальною до очищення, але стає унікальною після — unique_key_view одразу це покаже.

Фінальна частина логіки — інтерфейс. Модуль подає інформацію дуже делікатно: він не перевантажує таблицями; навпаки, показує ті аспекти, які корисні для прийняття рішення. У верхній частині — кількість кандидатів на ключ. Нижче — список колонок, які можуть бути унікальними. Далі — пари і трійки. І нарешті

— блоки з дублями та кнопкою “очистити”. Виходить така собі інтерактивна карта унікальності, де користувач крок за кроком розуміє, чи набір «веде себе» як цілісний об’єкт, чи потребує ручного втручання.

У результаті unique_key_view.py працює як інструмент контролю якості. Він визначає не просто первинний ключ, а загальний стан структурної цілісності. А головне — він пов’язаний з AI-модулем: якщо AI визначає колонку як "ідентифікатор", але модуль унікальності бачить дублікати, це дуже цінний сигнал, який змушує переглянути природу цих даних.

Інтелектуальні модулі VizoraDate — це не просто окремі блоки, що виконують свої завдання ізольовано. У реальному використанні вони створюють доволі синхронну систему, у якій змістові описи, типізація, статистика і пошук унікальних ключів підсилюють одне одного. У цьому сенсі прототип працює не як набір розрізнених функцій, а як невеликий консультативний механізм, що допомагає користувачу зрозуміти дані на різних рівнях.

Найцікавіше те, що AI-модуль і модуль унікальних ключів, хоч і побудовані на різних принципах, у реальності часто звертають увагу на одні й ті самі семантичні зони. Наприклад, якщо AI описує колонку як “унікальний код ” або “ідентифікатор запису”, це майже завжди перший сигнал перейти у unique_key_view і перевірити, чи справді значення унікальні. Якщо ж AI помічає, що колонка має «невпорядковані текстові значення, що повторюються», і водночас модуль унікальності виявляє дублікати або навіть повтори по всіх колонках (повні дублікати), це вказує на проблеми іншого рівня — структура таблиці могла бути порушена ще на етапі генерації або експорту. Таким чином, змістові описи AI стають своєрідними “коментарями до діагностики”.

Так само добре видно зв’язок між автоматичною типізацією і статистичними модулями. Коли column_describer.py на основі прикладів визначає, що значення схожі на дати, Streamlit-панелі автоматично вмикають сценарії, орієнтовані на часовий аналіз — розподіли за роками, кварталами, сезонністю. Якщо ж тип визначений неправильно (таке буває, коли у прикладах змішані формати),

користувач бачить це вже на етапі статистики: розподіл не будується, кореляція не рахується, модуль сигналізує про проблеми з типом даних. Така взаємодія забезпечує самокорекцію: система не просто механічно застосовує AI-висновки, а дозволяє людині перевірити їх у реальних графіках і метриках.

Крім того, результати AI-описів стають корисними при виявленні проблемних полів. Наприклад, коли AI описує колонку як “ймовірний код події, що має слабку інформативність” або “категоріальне поле з великим числом можливих значень”, користувач часто одразу переходить до статистики рядкових колонок, щоб переконатися, чи не перевантажена ця категорія. Якщо ж AI вказує на “можливий timestamp”, а типізація автоматично не визначила дату — це майже напевно означає наявність структурних артефактів: змішані формати, зайві символи, локальні маркери. І хоча VizoraDate не перетворює афектовані колонки автоматично, вона дає користувачу чітку підказку — де варто провести ручну очистку.

Ще одна точка інтеграції — поведінка всієї системи після очищення даних. Якщо користувач видаляє повні дублікати у unique_key_view, автоматично перераховуються не тільки кандидати на ключ, а й результати AI-описів та всі статистичні блоки. Це відбувається непомітно, через оновлення session_state. Імпліцитно виходить, що VizoraDate працює як невелика pipeline-система: один крок змінює стан даних, інші кроки реагують, оновлюючи результати без повторного завантаження файлу. Саме така поведінка робить інтелектуальні модулі не статичним блоком, а інтерактивною екосистемою.

Якщо подивитися ширше, обидва інтелектуальні модулі — AI-опис та аналіз унікальності — виконують роль рефлексивного шару системи. Один дивиться на дані "зсередини" (через зміст і приклади), другий — "зі структури" (через унікальність, повтори, комбінації). Разом вони дають дуже повну картину. Навіть якщо користувач не знає природи набору, кілька хвилин у цих вкладках дозволяють зрозуміти найважливіше: що означають колонки, які з них мають стабільну

структуру, чи є дублювання, чи має таблиця природну первинну структуру, і які поля можуть бути опорними для моделей або подальшого аналізу.

У підсумку інтелектуальні можливості VizoraDate не є декоративним доповненням. Вони змінюють користувацьку взаємодію на глибшому рівні: система не просто відтворює обчислення, а допомагає інтерпретувати дані як об’єкт, що має зміст, форму і внутрішню логіку. Саме це робить прототип ближчим до повноцінних аналітичних платформ нового покоління, де взаємодія AI та ручних перевірок уже стала нормою, а не експериментом.

ВИСНОВКИ

У цій роботі було послідовно розглянуто як теоретичні основи аналізу даних у режимі реального часу, так і практичні аспекти розроблення інтелектуальної системи, здатної виконувати повний цикл первинної аналітики. У ході дослідження вдалося поєднати базові концепції потокових обчислень, поширені архітектурні підходи та реальні вимоги, які виникають під час роботи з динамічними даними у сучасних аналітичних процесах.

Теоретичний аналіз продемонстрував, що потоки даних мають власні закономірності та обмеження, які суттєво впливають на вибір методів їх обробки. Вивчені алгоритми і технічні інструменти дозволили сформувати узагальнену картину того, як працюють системи real-time analytics, де саме виникають ризики втрати даних, затримок або структурних помилок. Значну увагу приділено машинному навчанню, оскільки саме воно є ключем до адаптивної обробки та інтелектуального аналізу потоків. Разом з тим було виявлено й низку обмежень, пов’язаних з різнорідністю форматів, неповними структурами та складністю інтеграції ML-моделей у динамічні конвеєри.

Також, робота була зосереджена на оцінці існуючих підходів і архітектур. Це дало можливість не просто класифікувати системи, а й побачити їхні реальні сильні і слабкі сторони. Детальний огляд функціональних ролей різних типів real-time систем, а також аналіз операційних практик показав, що, попри значний розвиток інструментів, залишається потреба в гнучких, легких і водночас інтелектуальних рішеннях, здатних працювати з "неідеальними" даними. Такі спостереження стали основою для розроблення концепції VizoraDate.

Практична частина роботи завершилася створенням повноцінної програмної системи , яка поєднує структурну, статистичну та семантичну аналітику. VizoraDate реалізована у вигляді модульної системи з чітким поділом бізнес-логіки та інтерфейсних компонентів. Механізми завантаження даних, визначення типів, обробки пропусків, виявлення дублікатів, аналізу кореляцій та формування AI-

описів забезпечують комплексне профілювання наборів без необхідності масштабної інфраструктури. Значним результатом є інтеграція генеративної моделі для пояснення змісту колонок, що суттєво розширює можливості класичного профілювання.

Розроблена система продемонструвала стабільність та гнучкість у роботі з різними форматами, включаючи CSV, JSON і NDJSON. Важливо, що VizoraDate не залежить від конкретного домену і може бути застосована в різних середовищах: від дослідницьких завдань до процесів контролю якості даних чи підготовки інформації для подальшої аналітики. Отримані результати підтверджують, що створена архітектура є придатною до масштабування і подальшого розвитку.

Узагальнюючи все виконане, можна зазначити, що поставлена мета досягнута. В роботі сформовано цілісне бачення real-time аналізу даних, визначено ключові виклики і тенденції, а також створено інтелектуальну систему, яка не лише демонструє можливості реалізації, але й може бути використана як практичний інструмент. Подальший розвиток VizoraDate відкриває перспективи розширення функціональності, впровадження потокових обчислень, глибших AI-модулів та інтеграції з професійними аналітичними платформами.

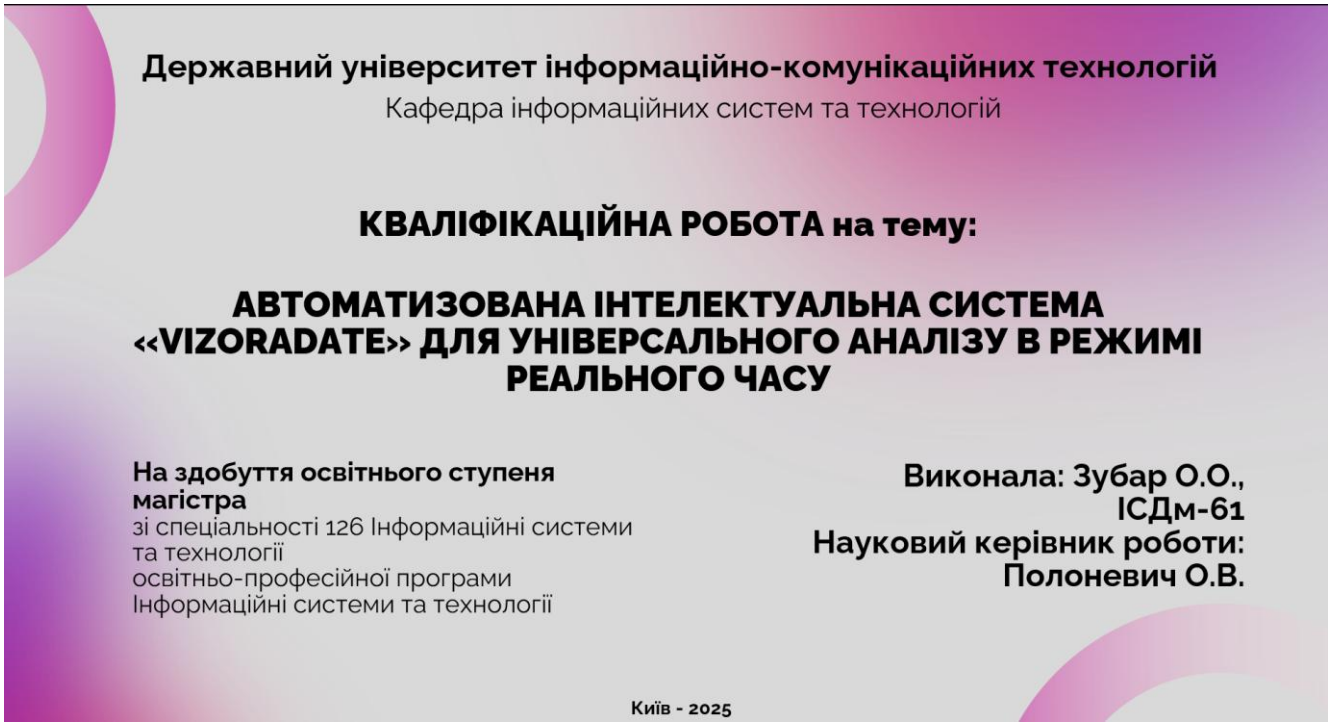
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

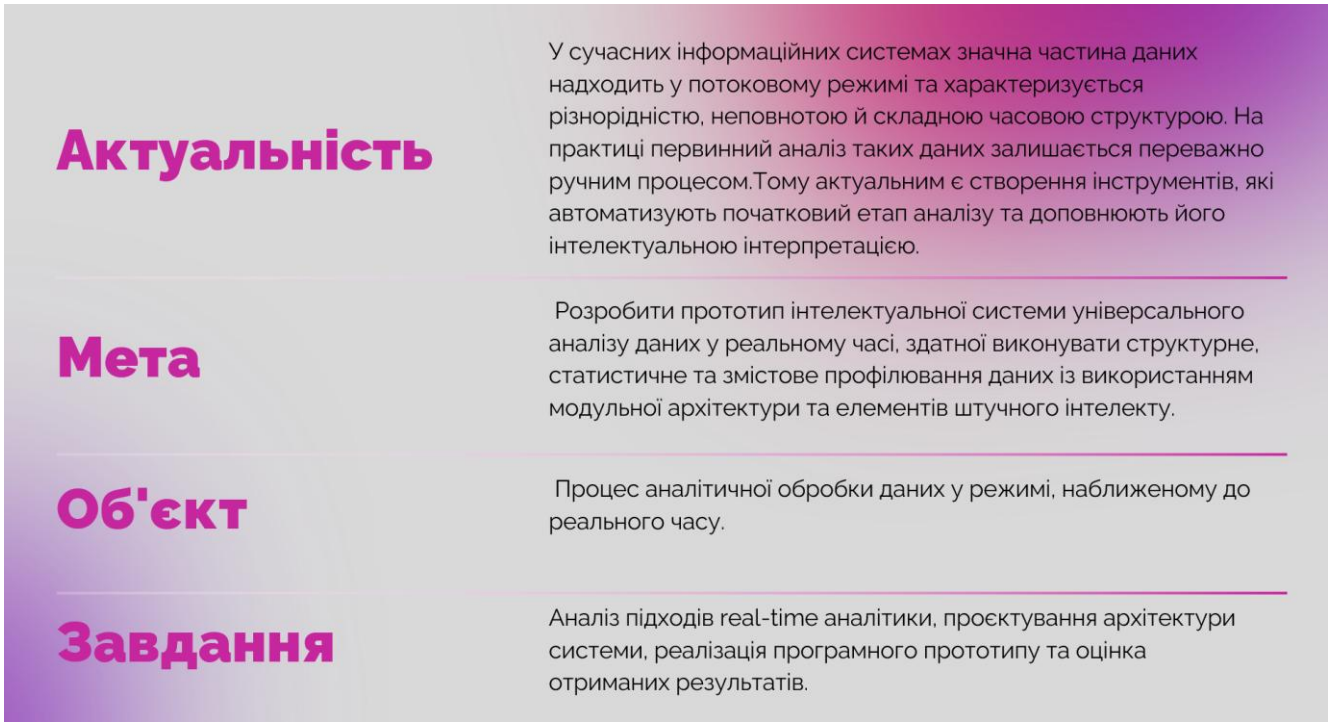
1. Akidau T., Chernyak S., Lax R. Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing. – Sebastopol : O'Reilly Media, 2018. – 312 p. – URL: https://www.oreilly.com/library/view/streaming-systems/9781491983867/
2. Apache Beam Documentation. – Apache Software Foundation, 2024. – URL: https://beam.apache.org/documentation/
3. Apache Flink Documentation. – Apache Software Foundation, 2024. – URL: https://flink.apache.org/docs/
4. Apache Kafka Documentation. – Apache Software Foundation, 2024. – URL: https://kafka.apache.org/documentation/
5. Apache Spark Structured Streaming Guide. – Apache Software Foundation, 2024. – URL: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
6. AWS Kinesis Developer Guide. – Amazon Web Services, 2024. – URL: https://docs.aws.amazon.com/kinesis/
7. Bifet A., Gavaldà R. Learning from Time-Changing Data with Adaptive Windowing. – SIAM, 2018. – URL: https://epubs.siam.org/doi/10.1137/1.9781611975673
8. Carbone P. et al. State Management in Apache Flink. – Proceedings of VLDB, 2019. – URL: https://www.vldb.org/pvldb/vol10/p1718-carbone.pdf
9. Clever Cloud. Managed RabbitMQ. – 2024. – URL: https://www.clever.cloud/product/managed-rabbitmq/
10. Databricks Documentation: Real-Time Analytics. – Databricks Inc., 2024. – URL: https://docs.databricks.com/en/streaming/index.html
11. Google Cloud Dataflow Documentation. – Google Cloud, 2024. – URL: https://cloud.google.com/dataflow/docs
12. Hohpe G., Woolf B. Enterprise Integration Patterns. – Boston : Addison-Wesley, 2019. – URL: https://www.enterpriseintegrationpatterns.com/

13. IBM Streaming Analytics Overview. – IBM Corporation, 2024. – URL: https://www.ibm.com/docs/en/streaming-analytics
14. Kleppmann M. Designing Data-Intensive Applications. – Sebastopol : O'Reilly Media, 2018. – 616 p. – URL: https://dataintensive.net/
15. Locus IT. Apache Flink for Data Science. – 2023. – URL: https://locusit.se/techpost/technology/apache-flink-for-data-science/
16. Medium. Message Ordering in Kafka. – 2022. – URL: https://medium.com/techverito/message-ordering-in-kafka-08ad06b73cf3
17. Medium. Calculate Real-Time Daily GMV with Apache Beam. – 2022. – URL: https://medium.com/trendyol-tech/calculate-real-time-daily-gmv-with-apache-beam-48d3763edb5b
18. Microsoft Azure Stream Analytics Documentation. – Microsoft, 2024. – URL: https://learn.microsoft.com/azure/stream-analytics/
19. MongoDB Atlas Stream Processing Documentation. – MongoDB Inc., 2024. – URL: https://www.mongodb.com/docs/atlas/atlas-stream-processing/
20. Oracle Stream Analytics Documentation. – Oracle Corporation, 2024. – URL: https://docs.oracle.com/en/middleware/osa/
21. Parse.ly. Apache Storm Overview. – 2021. – URL: https://www.parse.ly/storm/
22. Pääkkönen P., Pakkala D. Reference Architecture for Streaming Data Systems. – Journal of Big Data, 2018. – URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-018-0124-2
23. Redpanda. Event Streaming Platform. – 2024. – URL: https://www.redpanda.com/
24. Redpanda Documentation. – Redpanda Data Inc., 2024. – URL: https://docs.redpanda.com/
25. Shahrivari S. Beyond Batch Processing: Towards Real-Time Analytics. – Future Generation Computer Systems, 2019. – URL: https://www.sciencedirect.com/science/article/pii/S0167739X18308275
26. Snowflake Streaming Data Guide. – Snowflake Inc., 2024. – URL: https://docs.snowflake.com/en/user-guide/data-streaming

27. Stonebraker M. et al. The 8 Requirements of Real-Time Stream Processing. – ACM SIGMOD, 2018. – URL: https://dl.acm.org/doi/10.1145/3183713.3190664
28. Tableau. Real-Time Analytics Overview. – Salesforce, 2024. – URL: https://www.tableau.com/learn/articles/real-time-analytics
29. Zhang C., Chen C. Data Profiling and Metadata Management in Analytics Systems. – IEEE Access, 2020. – URL: https://ieeexplore.ieee.org/document/9090146

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (ПРЕЗЕНТАЦІЯ)





Загальна ідея та логіка рішення VizoraDate

Логотип системи VizoraDate

Система VizoraDate розроблена як інструмент автоматизованого первинного аналізу структурованих даних, орієнтований на швидке формування цілісного уявлення про набір даних одразу після його отримання.

На відміну від класичних BI-інструментів, система не потребує попереднього моделювання даних або налаштування схем і може працювати з довільними файлами без додаткової підготовки.

Ключова логіка рішення полягає у поєднанні статистичного аналізу, часових методів і інтелектуальної інтерпретації результатів у єдиному середовищі, що дозволяє скоротити час переходу від «сирих даних» до аналітичних висновків.

Місце VizoraDate в аналітичному процесі

VizoraDate позиціонується як початковий етап аналітичного конвеєра, який передує побудові моделей машинного навчання, розробці ETL-процесів або створенню бізнес-звітів.
Система вирішує завдання, які зазвичай виконуються вручну: перевірка структури файлу, виявлення проблем якості, аналіз часової динаміки. Це дозволяє уникнути помилок на пізніших етапах та приймати обґрунтовані рішення щодо подальшої обробки даних.

Data
Ingestion

VizoraDate
Structural Analysis

VizoraDate
Data Quality & Time Analysis

Data Preparation
ETL / Feature Engineering

Advanced Analytics
Reporting

Архітектурні принципи побудови системи

Архітектура VizoraDate побудована з урахуванням принципів модульності, масштабованості та ізоляції компонентів.

Основна обробка даних винесена в окремий аналітичний шар, який не залежить від способу візуалізації або інтерфейсу користувача.

Такий підхід дозволяє незалежно розвивати окремі модулі системи, зокрема додавати нові методи аналізу або змінювати інтерфейс без переписування базової логіки.

Modularity

Архітектура системи розділена на незалежні функціональні модулі.

Scalability

Можливість розширення системи без зміни базової логіки.

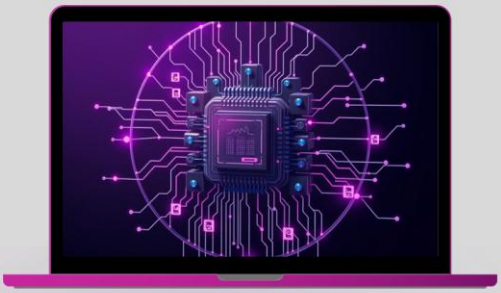
Isolation of Components

Аналітичне ядро не залежить від UI та способів візуалізації.

Extensibility

Додавання нових методів аналізу без перепроєктування системи.

Реалізація ядра обробки даних



Ядро системи VizoraDate відповідає за завантаження файлів, нормалізацію структури даних та виконання основних обчислювальних операцій.

Для реалізації використано бібліотеки Pandas і Polars, що дозволяє ефективно обробляти як невеликі, так і великі набори даних.

Окрему увагу приділено оптимізації агрегацій і групувань, які є критичними при роботі з часовими рядами.

Інтерфейс користувача та взаємодія

Інтерфейс системи реалізований у вигляді інтерактивного веб-додатку, який не потребує складного розгортання. Користувач взаємодіє із системою через покроковий сценарій: завантаження файлу, перегляд структурної інформації, аналіз якості та часових характеристик.

Такий формат дозволяє використовувати систему як технічними спеціалістами, так і користувачами з мінімальним аналітичним досвідом.

USER INTERACTION FLOW

Покроковий сценарій роботи: завантаження файлу → аналіз → результати.

Interactive Visualization

Інтерактивний веб-інтерфейс без складного розгортання.

Підтримка форматів і типів даних

VizoraDate підтримує роботу з поширеними форматами структурованих даних, зокрема CSV, JSON та NDJSON. Під час завантаження система автоматично визначає типи колонок, включаючи числові, категоріальні, текстові та часові поля. Це забезпечує універсальність рішення та можливість застосування в різних прикладних сценаріях без ручного налаштування.

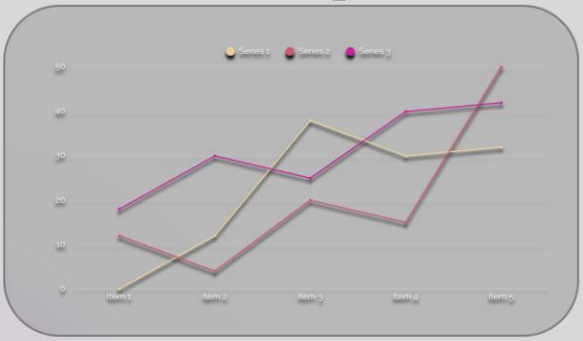
Supported Formats

CSV, JSON, NDJSON
Робота з поширеними форматами структурованих даних.

Data Types Detection

Числові, категоріальні, текстові та часові колонки.

Паспорт набору даних



Dataset Summary

Кількість рядків і колонок, формат, кодування.

Structural Metadata

Типи колонок, вкладені структури, часові поля.

Паспорт даних є базовим результатом роботи системи та містить узагальнену інформацію про набір: кількість рядків і колонок, формат, кодування, типи даних і наявність вкладених структур. Формування паспорта дозволяє швидко оцінити складність даних і прийняти рішення щодо доцільності подальшого аналізу або необхідності попереднього очищення.

Аналіз структури та якості даних

Система автоматично аналізує наявність пропусків, дублікатів та потенційних ідентифікаторів. Результати подаються у вигляді узагальнених показників і візуальних представлень, що дозволяє швидко ідентифікувати проблемні зони.

Цей етап є критичним для запобігання помилкам при подальшій аналітичній або машинній обробці даних.

Structure Analysis

Аналіз типів колонок і їх однорідності.

Data Quality Checks

Виявлення пропусків, дублікатів і потенційних ключів.

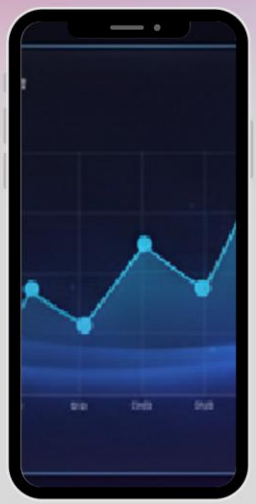
Часовий аналіз даних

Time Aggregation

Агрегація даних за різними часовими інтервалами.

Trend & Seasonality Analysis

Виявлення трендів і сезонних компонент.



Для колонок, що містять часові значення, VizoraDate автоматично будує часові ряди та агрегує дані за різними інтервалами. Застосовуються методи згладжування та декомпозиції, що дозволяє виділяти тренди, сезонні компоненти та випадкові коливання. Це забезпечує глибше розуміння динаміки даних і дозволяє виявляти нетипові зміни.

Виявлення закономірностей та аномалій

Система реалізує аналіз автокореляції та пошук пікових значень у часових рядах. Це дозволяє виявляти повторювані шаблони, циклічні процеси та потенційні аномалії, які можуть свідчити про помилки збору даних або важливі події. Отримані результати використовуються для формування подальших аналітичних гіпотез.

Autocorrelation

Аналіз повторюваних часових залежностей.

Patterns

Виявлення циклічних процесів.

Anomalies

Пікові значення та нетипові відхилення.

Інтелектуальна інтерпретація результатів

Інтелектуальний модуль системи забезпечує генерацію текстових описів окремих колонок і загальних висновків по всьому набору даних. Мовна модель аналізує статистичні характеристики та приклади значень, формуючи пояснення у зрозумілій для користувача формі.

Це дозволяє зменшити когнітивне навантаження та прискорити процес прийняття рішень

Висновки

У межах магістерської роботи було спроєктовано тареалізовано повноцінний програмний прототип автоматизованої інтелектуальної системи VizoraDate, призначеної для універсального первинного аналізу структурованих даних. Розроблена система поєднує автоматизований аналіз структури наборів даних, оцінку їх якості, дослідження часових характеристик, а також інтелектуальну інтерпретацію отриманих результатів, що дозволяє сформувати цілісне уявлення про дані на ранньому етапі аналітичного процесу.
Система VizoraDate демонструє практичну придатність для використання у реальних аналітичних задачах, зокрема у процесах підготовки даних, контролю їх якості, дослідницької аналітики та як допоміжний інструмент перед побудовою моделей машинного навчання або формуванням бізнес-звітів.

АПРОБАЦІЇ

III Всеукраїнська науково-технічна конференція "Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу"

Тема : ВИКОРИСТАННЯ МАШИННОГО НАВЧАННЯ ДЛЯ ДИНАМІЧНОЇ ОБРОБКИ ПОТОВИХ ДАНИХ

Дата : 18 листопада 2025 року

VII МІЖНАРОДНА НАУКОВО-ТЕХНІЧНА КОНФЕРЕНЦІЯ «СУЧАСНИЙ СТАН ТА ПЕРСПЕКТИВИ РОЗВИТКУ IOT»

Тема : АНАЛІЗ СУЧАСНИХ ТРЕНДІВ У СФЕРІ ВІЗУАЛІЗАЦІЇ ВЕЛИКИХ ДАНИХ

Дата : 15 квітня 2025 року

ДЯКУЮ ЗА УВАГУ !