Технічна документація

Аналітична web-платформа для збору, обробки та AI-аналізу даних

---

0. Огляд системи

Мета платформи — збирати дані з багатьох зовнішніх джерел (держреєстри, відкриті API, БД, скрейпінг), об'єднувати їх у єдиний структурований масив, виконувати класичний аналітичний розрахунок і формувати фінальний звіт за участю AI-модуля.

Система спроєктована як набір незалежних сервісів:

- шар інтеграцій із джерелами;
- нормалізація та очищення;
- сховище даних (операційне + аналітичне);
- аналітичний шар (класичні алгоритми);
- AI-шар (LLM, embeddings, RAG);
- backend API;
- web-інтерфейс.

Для демонстрації MVP можлива спрощена реалізація на одному сервісі (наприклад, FastAPI + Python), який працює з публічним безкоштовним API (`jsonplaceholder.typicode.com`) як мок-джерелом.

---

1. Модуль інтеграції із зовнішніми джерелами

1.1. Типи джерел

Платформа повинна вміти працювати з:

- HTTP API (REST/GraphQL, JSON/XML, різна швидкість відповіді);
- БД (PostgreSQL, MySQL, MSSQL тощо);
- файли (CSV, XLSX, XML, JSON у S3/object storage або FTP);
- джерела без API (HTML-сторінки, скрейпінг, headless-браузер);
- повільні або нестабільні джерела;
- джерела з rate limit та змінною структурою даних (schema drift).

1.2. Шар інтеграцій

Архітектурно інтеграції оформлюються як окремий шар **Data Ingestion Layer**:

- **Gateway/Facade**:
  - єдиний вхідний інтерфейс `IngestionService`, який приймає «тип джерела» + параметри;
  - внутрішньо викликає конкретний **адаптер** джерела.
- **Адаптери джерел**:
  - `ApiConnector`, `DbConnector`, `FileConnector`, `ScraperConnector`;
  - кожен адаптер інкапсулює специфіку авторизації, формату відповіді, пагінації тощо;
  - повертають уніфікований формат «raw events» (наприклад, список словників).

Технічно: окремий сервіс/модуль `ingestion` з чітким контрактом:

- вхід: запит на збір (`source_id`, `params`, `time_range`);
- вихід: потік нормалізованих raw-записів у чергу (Kafka/RabbitMQ) або в staging-таблиці.

1.3. Обробка помилок та змін структури

Основні кейси:

- тимчасові помилки мережі або 5xx → **retry з backoff**;
- невірний формат відповіді → логування + перехід у статус `failed_with_schema_error`;
- зміна структури даних:
  - адаптер **не падає**, а пропускає невідомі поля;
  - схема описується у внутрішньому форматі (schema registry);
  - додаються версії схем (v1, v2...), зберігається інформація про те, під якою схемою зберігалися конкретні дані.

1.4. Кешування

Рівні кешу:

- **Request-level cache**:
  - ключ: (джерело, параметри, часовий діапазон);
  - короткий TTL (хвилини/години);
  - зменшує кількість повторних звернень до повільних API.
- **Normalized entity cache**:
  - кеш уже нормалізованих сутностей (наприклад, контрагенти, компанії, документи);

- використовується для прискорення аналітики та AI-RAG.

- **Materialized snapshots**:
  - періодичне збереження агрегованих таблиць (summary за день/тиждень);
  - використання замість сирих даних для швидких відповідей.

1.5. Версійність даних

Версійність потрібна для:

- аудиту (що було відомо на конкретну дату);
- повторного запуску аналітики з тією ж вхідною базою;
- аналізу змін (deltas).

Підхід:

- **append-only модель**:
  - кожен запис має `valid_from`, `valid_to`, `version_id`;
  - оновлення — це новий запис із новим `version_id`, старий позначається `valid_to`.
- **schema versioning**:
  - у метаданих таблиці/запису зберігається `schema_version`;
  - логіка читання підтримує кілька версій схем одночасно (через мапінг до уніфікованої моделі).

---

2. Архітектура всієї системи

2.1. Основні компоненти

1. **Data Ingestion Layer**
   - конектори до джерел;
   - черги для асинхронного збору.
2. **Cleaning & Normalization Layer**
   - очищення, нормалізація форматів;
   - мапінг у єдину модель даних.
3. **Data Storage**
   - **Operational Data Store (ODS)**: сирі/напівоброблені дані;
   - **Data Warehouse**: агрегації, факт-таблиці, вимірювання;

- **Data Lake / Object Storage**: логи, файли, snapshot-и;
- **Search / Vector Store**: індексація текстових/семантичних даних.

4. **Analytics Layer**

   - реалізація класичних алгоритмів: агрегації, скоринг, ризикові моделі;
   - пакет batch-job'ів і on-demand запитів.

5. **AI Module**

   - генерація текстових звітів;
   - RAG-pipeline (embeddings + vector store);
   - пост-обробка та валідація.

6. **Backend API**

   - FastAPI/аналог;
   - endpoints для запуску задач, отримання статусу, видачі результатів;
   - авторизація/аутентифікація (JWT/OAuth2).

7. **Web-інтерфейс**

   - SPA (React/Vue/Angular) або SSR;
   - форми запитів, сторінка процесингу, сторінка результатів, історія запитів.

2.2. Де використовуються черги (message queues)

Черги (Kafka/RabbitMQ/SQS/тощо) застосовуються для:

- **Data ingestion**:
  - кожне джерело пише «сирі івенти» у топік;
  - декілька воркерів паралельно забирають та обробляють.
- **Обробки запиту користувача**:
  - запит користувача = job;
  - job потрапляє в чергу;
  - workflow/worker-сервіс виконує pipeline: збір → нормалізація → аналітика → AI.
- **Long-running tasks**:
  - heavy-аналіз, побудова звітів, перерахунок моделей запускаються як background tasks;
  - API не блокується, користувач отримує лише `job_id`.

2.3. Відокремлення AI-логіки від бізнес-логіки

- AI-модуль — **окремий сервіс** або хоча б окремий модуль:
  - має контракт: `structured_context → prompt → LLM → draft_report`;
  - не містить бізнес-правил, тільки текстова генерація + RAG.
- Класична бізнес-логіка:
  - розрахунок метрик, ризикових індикаторів, агреговані дані;
  - працює без LLM, детерміновано, тестується звичайними юніт-тестами.
- AI працює **поверх** результатів класичної логіки:
  - LLM отримує вже пораховані цифри та структуру, а не сирі дані;
  - це знижує ризики «вигаданих» числових значень.

2.4. Горизонтальне масштабування

- Ingestion:
  - масштабування конекторів по кількості джерел і трафіку (більше воркерів на топік).
- Normalization/Analytics:
  - stateless-воркери, які можна масштабувати за кількістю задач.
- Backend API:
  - кілька інстансів за балансером;
  - sticky-sessions не потрібні, стан зберігається в БД/кеші.
- AI-модуль:
  - окремий пул воркерів, який масштабується по кількості LLM-запитів;
  - можливий шар rate limiting / throttling на рівні AI.

---

3. Логіка AI-аналізу

3.1. Pipeline витягнення фактів

1. **ETL/нормалізація**:
   - сирі дані приводяться до єдиної моделі;

- ідентифікуються об’єкти (компанії, особи, події).

2. **Fact extraction (NLP)**:
    - з текстових полів витягуються факти: дати, суми, події, зв’язки;
    - використовується NER, rule-based парсери або невеликі моделі.
3. **Entity linking / deduplication**:
    - склеювання однакових сутностей із різних джерел;
    - побудова графа зв’язків між сутностями.
4. **Формування логічних блоків**:
    - блок «компанія A»: всі факти + пов’язані суб’єкти;
    - блок «подія»: всі записи про конкретну транзакцію/кейс.
5. **Підготовка контексту для AI**:
    - конвертація блоків у стиснутий структурований формат (JSON/табличний вигляд);
    - побудова промпта із чіткими інструкціями.

3.2. Embeddings, vector storage, RAG

- **Embeddings**:
    - для кожного текстового документу/факту створюється векторне представлення;
    - використовується для пошуку схожих кейсів, документів, попередніх звітів.
- **Vector storage (Weaviate, PGVector, Qdrant, Pinecone)**:
    - індексація всіх релевантних текстових блоків;
    - швидке отримання контексту за запитом.
- **RAG-підхід**:
    - на основі запиту і підготовлених метрик вибираються релевантні документи;
    - ці документи + структуровані дані потрапляють у prompt;
    - LLM генерує звіт лише на основі цього контексту.

3.3. Механізми контролю якості та верифікації

- **Cross-source validation**:
    - цифри та ключові факти перевіряються по кількох джерелах;
    - якщо джерела суперечать одне одному, у звіті явно вказується «є розбіжності».

- **Rule-based checks**:
    - діапазони значень (не може бути від'ємної кількості, дата в майбутньому тощо);
    - консистентність сум (subtotal, total, податки).
- **Post-processing LLM output**:
    - парсинг звіту в структурований формат (JSON schema);
    - перевірка: кожне число, яке використовує LLM, повинно існувати в вихідних даних/контексті;
    - якщо виявлена невідповідність — маркер `suspected_hallucination`, можливий автозапуск повторного аналізу з більш жорсткими інструкціями.

---

4. Користувацький сценарій (user-flow)

4.1. Ввід запиту

1. Користувач заходить на **головну сторінку**.
2. Вводить:
    - тип аналізу (наприклад, компанія, об'єкт, кейс);
    - ідентифікатори (назва, код, інші параметри);
    - часовий діапазон та додаткові опції.
3. Натискає «Запустити аналіз».

4.2. Запуск процесу обробки

1. Backend API створює **Job**:
    - `job_id`, тип аналізу, параметри, статус `pending`.
2. Job відправляється в чергу.
3. Користувач отримує `job_id` і одразу перенаправляється на «сторінку процесингу».

4.3. Інформування про прогрес та статуси

Статуси:

- `pending` — завдання поставлене в чергу;
- `collecting` — триває збір даних із джерел;
- `normalizing` — нормалізація й очищення;

- `analyzing` — класичний аналітичний розрахунок;
- `ai_generating` — робота AI-модуля;
- `ready` — звіт готовий;
- `error` — критична помилка;
- додатково можна мати `partial_ready`, якщо є частковий результат.

Комунікація:

- фронтенд робить **polling** або використовує WebSocket/EventSource для оновлень;
- разом зі статусом повертається короткий progress summary (наприклад, `3/5 sources processed`).

4.4. Вигляд готового звіту

Формати:

- web-сторінка (HTML);
- PDF (згенерований на бекенді);
- JSON (для інтеграцій).

Структура:

- заголовок (ідентифікатор об'єкта запиту, дата формування);
- резюме (executive summary) — 3–7 ключових висновків;
- секції:
  - «Дані з джерел» (які джерела, що знайдено);
  - «Ключові показники» (метрики, ризики, індикатори);
  - «AI-аналіз» (текстовий опис, інтерпретація);
  - «Обмеження даних» (що не вдалося отримати, де є конфлікти).

4.5. Історія запитів

- для кожного користувача:
  - `job_id`, тип запиту, дата/час запуску, статус, дата завершення;
  - посилання на результат (web/PDF/JSON).
- зберігається в окремій таблиці (наприклад, `user_queries`, `reports`);
- підтримка фільтрації за датою, типом запиту, статусом.

---

5. Основні інженерні ризики та рішення

5.1. Ризики

1. Нестабільні джерела даних.
2. «Брудні» або неповні дані.
3. Довгі часи відповіді.
4. Відсутність частини даних.
5. Помилки AI (галюцинації, некоректні висновки).
6. Високе навантаження (багато одночасних запитів).

5.2. Рішення

1. **Нестабільні джерела**
    - retry з exponential backoff;
    - fallback на кеш/історичні дані;
    - окремі health-чекери по кожному джерелу.
2. **Брудні/неповні дані**
    - валідація на рівні нормалізації;
    - використання словників/довідників для нормалізації значень;
    - маркування записів як «partial»/«low_quality».
3. **Довгі часи відповіді**
    - асинхронний ingestion;
    - черги та воркери замість синхронних API-викликів;
    - кешування результатів (особливо для часто повторюваних запитів).
4. **Відсутність частини даних**
    - fallback-джерела;
    - позначення у звіті, що висновки зроблені на неповній базі;
    - опція «дочекайся всіх джерел» vs «поверни частковий результат».
5. **AI-помилки (галюцинації)**
    - RAG: AI працює тільки з переданим контекстом;
    - контроль відповідності чисел і фактів даним;
    - чіткі системні інструкції в prompt (не вигадувати, зазначати відсутність даних).
6. **Навантаження**
    - горизонтальне масштабування воркерів та API;
    - rate limiting per user/tenant;

- пріоритизація задач (внутрішні/VIP-запити, SLA).

---

6. Міні-структура інтерфейсу

6.1. Головна сторінка (форма запиту)

Блоки:

- поле для запиту (тип аналізу, ідентифікатор, дата/період);
- розширені параметри (фільтри, глибина аналізу);
- кнопка «Запустити аналіз»;
- коротка історія останніх запитів (опціонально).

6.2. Сторінка збору та обробки даних

Блоки:

- статус job'a:
  - поточний етап (collecting / analyzing / ai_generating);
  - прогрес по джерелах;
- таймлайн:
  - timestamps основних етапів;
- лог/alert-панель:
  - попередження про проблемні джерела, частковий аналіз.

6.3. Сторінка результату (готовий звіт)

Блоки:

- загальний заголовок + базова інформація по запиту;
- блок «Резюме» (AI-summary);
- блок «Ключові показники» (класична аналітика);
- блок «Деталі по джерелах»;
- блок «Обмеження та якість даних»;
- кнопки:
  - «Завантажити PDF»;
  - «Експорт JSON»;
  - «Поділитися посиланням».

6.4. Сторінка історії запитів

Блоки:

- таблиця:
  - дата, тип запиту, статус, час виконання;
  - посилання на звіт;
- фільтри:
  - період, статус, тип аналізу;
- можливість повторного запуску аналізу з тими ж параметрами.

6.5. Базові налаштування

Блоки:

- мови інтерфейсу / звітів;
- формат звіту (PDF/HTML/JSON);
- налаштування нотифікацій (email/веб-push);
- параметри глибини аналізу (light/full).

---

7. Reference MVP-реалізація (для тестового)

Для демонстраційного MVP:

- Backend: FastAPI (Python 3.14, Pydantic v2).
- Endpoint `/analyze`:
  - `query` — назва dataset (наприклад, `posts`);
  - `options.max_sample_items` — обмеження кількості записів у sample.
- Зовнішнє джерело:
  - **бесплатний тестовий API** `https://jsonplaceholder.typicode.com/{query}`.
- Логіка:
  - отримання списку об'єктів;
  - нормалізація ключів (snake_case), фільтр пустих значень;
  - кешування результату по `query` в in-memory TTL кеші;
  - побудова текстового summary (мок AI-аналізу);
  - повернення структури:
    - `status`, `source_items`, `items_count`, `summary`, `sample`.

Цей MVP є спрощеним зрізом повної архітектури: показує pipeline «збір → нормалізація → кеш → аналітичний summary», і може бути розширений до повноцінної платформи за описаною вище схемою.