Technical Documentation

Analytical Web Platform for Multi-Source Data Collection, Processing, and AI-Driven Reporting

---

0. System Overview

The platform is designed to collect data from multiple external sources (government registries, public APIs, databases, scrapers), unify it into a structured dataset, perform classical analytical processing, and generate a final AI-augmented report.

The system is architected as a set of independent components:

- integration layer with external sources;
- data cleaning and normalization;
- data storage (operational + analytical);
- analytics layer (classical algorithms);
- AI module (LLM, embeddings, RAG);
- backend API;
- web interface.

For the MVP demonstration, a simplified single-service implementation can be used (e.g., FastAPI + Python) working with a **free public test API (`jsonplaceholder.typicode.com`)** as a mock external source.

---

1. External Data Integration Module

1.1. Types of supported data sources

The platform must handle the following types of sources:

- HTTP APIs (REST/GraphQL, JSON/XML, varying response speeds);
- Databases (PostgreSQL, MySQL, MSSQL, etc.);
- Files (CSV, XLSX, XML, JSON via S3/Object Storage or FTP);
- Sources without API (HTML pages, scraping, headless browser);
- Unstable or slow sources;
- Rate-limited or schema-drifting APIs.

1.2. Integration layer architecture

The integration layer (**Data Ingestion Layer**) provides:

- **Unified Gateway / Facade**:
    - a single entry point `IngestionService`;
    - internally delegates to appropriate adapters.
- **Source Adapters**:
    - `ApiConnector`, `DbConnector`, `FileConnector`, `ScraperConnector`;
    - each adapter handles auth, pagination, response parsing, throttling;
    - returns a unified “raw events” format (list of dictionaries).

Technically this is a standalone module/service `ingestion` with a strict contract:

- input: request for data collection (`source_id`, params, time_range);
- output: raw normalized events pushed to a queue (Kafka/RabbitMQ) or staging storage.

1.3. Error handling & schema changes

Key scenarios:

- temporary network errors / 5xx → **retry with exponential backoff**;
- invalid response → error logging + mark as `schema_error`;
- schema drift:
    - adapters ignore unknown fields instead of failing;
    - use **schema registry** with versioning (v1, v2, v3);
    - old data is mapped to a unified internal model before further processing.

1.4. Caching strategy

Caching is used at several levels:

- **Request-level cache**:
    - key: (source, parameters, time range);
    - short TTL;
    - reduces load on slow/expensive APIs.

- **Normalized entity cache**:
    - store normalized entities (company profiles, documents);
    - speeds up analytics and RAG context generation.
- **Materialized snapshots**:
    - periodic daily/weekly aggregated views for fast queries;
    - used instead of raw data where possible.

1.5. Data versioning

Needed for:

- auditability (what was known at a given time);
- reproducibility of analysis results;
- tracking changes over time.

Approach:

- **append-only model**:
    - each record has `valid_from`, `valid_to`, `version_id`;
    - updates create a new version, old one becomes closed.
- **schema versioning**:
    - `schema_version` stored with each record;
    - readers support multiple schema versions via mapping rules.

---

2. System Architecture

2.1. Core components

1. **Data Ingestion Layer**
    - connectors, scrapers, parsers;
    - queues for async ingestion.
2. **Cleaning & Normalization Layer**
    - data cleansing, unification, validation;
    - mapping into internal canonical model.
3. **Data Storage**
    - **ODS (Operational Data Store)** for raw/processed events;
    - **Data Warehouse** for metrics, analytics tables;
    - **Object Storage / Data Lake** for logs, snapshots, documents;
    - **Vector Store** for semantic indexing (embeddings).
4. **Analytics Layer**
    - classical metrics, risk indicators, aggregates;
    - scheduled and on-demand jobs.
5. **AI Module**
    - report generation;
    - RAG (vector search + context selection);
    - hallucination detection + post-processing.
6. **Backend API**
    - FastAPI (or similar);
    - job orchestration: start job, fetch status, fetch results;
    - authentication/authorization layer.
7. **Web Interface**
    - SPA (React/Vue/Angular) or server-rendered UI;
    - request form, processing screen, result screen, query history.

2.2. Where message queues are used

Queues (Kafka/RabbitMQ/SQS/etc.) are used for:

- **Parallel data ingestion**:
    - source adapters push raw events into topics;
    - multiple workers consume and process them concurrently.
- **User-triggered jobs**:

    - each analysis request → enqueued job;
    - worker service executes full pipeline.

- **Long-running operations**:
    - AI processing, heavy analytics, cross-source validation.

2.3. Separating AI logic from business logic

- AI module must be **isolated** as a standalone service or isolated module.
- Input/output strictly defined:
    - `structured_context → prompt → LLM → draft_report` .
- Business logic must:
    - compute metrics deterministically;
    - avoid embedding AI in core logic;
    - remain fully testable without AI.

AI augments **interpretation**, not **calculation**.

2.4. Horizontal scalability

- Ingestion:
    - scale workers per source or throughput.
- Normalization & Analytics:
    - stateless workers, replicated horizontally.
- Backend API:
    - behind a load balancer;
    - no local state (DB/cache only).
- AI Module:
    - dedicated worker pool, scaling based on LLM throughput;
    - rate limiting and concurrency control.

---

3. AI Analysis Logic

3.1. Fact extraction pipeline

1. **ETL & normalization** → uniform event model.
2. **NLP Fact extraction**:
    - NER, entity extraction, date/amount detection.
3. **Entity linking & deduplication**:
    - grouping identical entities across sources.
4. **Logical block building**:
    - “Company A block”, “Event block”, “Transaction block”.
5. **Context building for AI**:
    - compressed JSON-like structure + instructions.

3.2. Embeddings, vector store & RAG

- **Embeddings**:
    - semantic representation of documents/facts;
    - used for similar record search.
- **Vector store (Weaviate / Qdrant / PGVector / Pinecone)**:
    - fast context retrieval.
- **RAG pipeline**:
    - find relevant facts;
    - combine with structured metrics;
    - pass into LLM with strict prompt instructions.

3.3. Quality control & validation

- **Cross-source validation**:
    - verify numbers through multiple independent datasets.
- **Rule-based validation** :
    - numeric sanity checks;
    - consistency across fields.
- **Post-processing / hallucination detection**:

- parse LLM output back to JSON schema;
- detect values not present in source data;
- re-run with stricter constraints if inconsistencies found.

---

4. User Flow

4.1. Query input

User selects:

- analysis type;
- identifiers (company name, ID, etc.);
- optional filters or time ranges.

Then presses **Run Analysis**.

4.2. Processing flow

1. API creates a **Job** with `job_id`.
2. Job is sent to message queue.
3. User is redirected to the processing screen.

4.3. Progress tracking & statuses

Statuses:

- `pending`
- `collecting`
- `normalizing`
- `analyzing`
- `ai_generating`
- `ready`
- `error`
- (optional) `partial_ready`

Frontend uses polling or WebSockets to update progress.

4.4. Final report format

Formats:

- Web page (HTML);
- PDF;
- JSON.

Structure:

- title + metadata;
- executive summary (AI-generated);
- key metrics (classical analytics);
- per-source findings;
- data quality & limitations section.

4.5. Query history

Each user sees:

- job type, timestamps, status;
- link to report;
- ability to rerun the job.

Stored in separate tables (`user_queries`, `reports`).

---

5. Engineering Risks & Solutions

5.1. Key risks

1. Unstable data sources
2. Dirty or incomplete data
3. Long response times
4. Missing required data
5. AI hallucinations and inconsistencies
6. High load / high concurrency

5.2. Mitigation strategies

1. **Unstable sources**
    - retry/backoff
    - fallback to cached data
    - per-source health checks
2. **Dirty / incomplete data**
    - validation & normalization
    - deduplication
    - quality scoring
3. **Slow responses**
    - async ingestion
    - queues for heavy tasks
    - aggressive caching
4. **Missing data**
    - fallback secondary sources
    - mark incomplete results explicitly
5. **AI hallucinations**
    - RAG with ground-truth context
    - output validation
    - strict prompting
6. **High load**
    - horizontal stateless scaling
    - sharded ingestion
    - rate limiting per user

---

6. Minimal Interface Structure

6.1. Home page – Request Form

- input fields (type, ID, filters);
- submit button;
- recent history preview.

6.2. Processing page

- job status and progress bars;
- timeline of completed steps;
- warnings for missing/unstable sources.

6.3. Results page

- summary;
- metrics;
- per-source breakdown;
- download buttons: PDF / JSON.

6.4. Query history page

- table: date, type, status, execution time;
- filters;
- rerun button.

6.5. Settings page

- language;
- report format (PDF/HTML/JSON);
- notifications.

---

7. Reference MVP Implementation (for the test task)

For demonstration:

- Backend: FastAPI (Python 3.14, Pydantic v2).
- Endpoint `/analyze` :

    - `query` = dataset name (e.g., `posts`);
    - `options.max_sample_items` for sample size.
- External source:
    - **free public test API** `https://jsonplaceholder.typicode.com/{query}`.
- Processing flow:
    - fetch data → normalize → cache → generate summary → return sample.

This MVP illustrates the core pipeline (“ingest → normalize → cache → summarize”) and can be extended to the full architecture described above.