Stream-Based Wiki Page Tracking System

1. Introduction

This project implements a real-time system for ingesting, processing, and querying Wikimedia page creation events. The primary goal is to support **ad-hoc analytical queries** (Category B) on page creation data, such as retrieving user activity, domain-specific statistics, and individual page metadata.

The system uses the **Wikimedia EventStream API** as the source of truth, processes events in real time with **Apache Spark**, stores structured data in **Apache Cassandra**, and exposes query capabilities via a **FastAPI-based REST API**.

Category B queries are the current focus of this report due to integration issues with the Spark–Cassandra connector version needed for batch aggregation (Category A). Nevertheless, the full data pipeline is functional and extensible.

2. System Architecture

The core pipeline includes the following stages:

2.1 Real-Time Ingestion

The pipeline starts with a custom-built Kafka producer that listens to the Wikimedia Event Stream. It continuously receives page creation events and sends them to the **input Kafka topic**.

Component:

producer.py

Output Topic: input

Purpose: Connects to Wikimedia stream and publishes raw JSON events into Kafka.

2.2 Stream Processing Layer

The real-time transformation pipeline is implemented using two Spark Streaming jobs, each focused on a distinct processing responsibility:

2.2.1 cleaner.py

This Spark job subscribes to the input topic, validates incoming data, drops incomplete records, extracts and normalizes relevant fields (such as user name, domain, timestamps), and adds derived metadata like date_hour. It writes the cleaned output to the processed Kafka topic.

Input Topic: input

Output Topic: processed

Component: cleaner.py

2.2.2 write_to_cassandra.py

This job reads the processed stream and writes the cleaned data into four Cassandra tables simultaneously:

- pages_by_id
- pages_by_user
- pages_by_domain
- pages_by_time

Each table is designed for specific downstream queries and REST API use cases. The writer uses foreachBatch logic to ensure consistency and fault tolerance.

Input Topic: processed

Component: write_to_cassandra.py

2.3 Data Storage Layer

All cleaned and transformed events are persisted in **Apache Cassandra**. The storage is denormalized and partitioned to ensure low-latency access to common queries such as:

- Retrieve all pages by a user
- Count pages per domain

- Fetch pages by ID
- Analyze page activity by hour

Each table schema is described in detail in **Section 3**.

2.4 Serving Layer (REST API)

Finally, a **FastAPI** application exposes a set of RESTful endpoints to retrieve both raw and aggregated data. The API is optimized for real-time responsiveness and accesses Cassandra directly using the official Python driver.

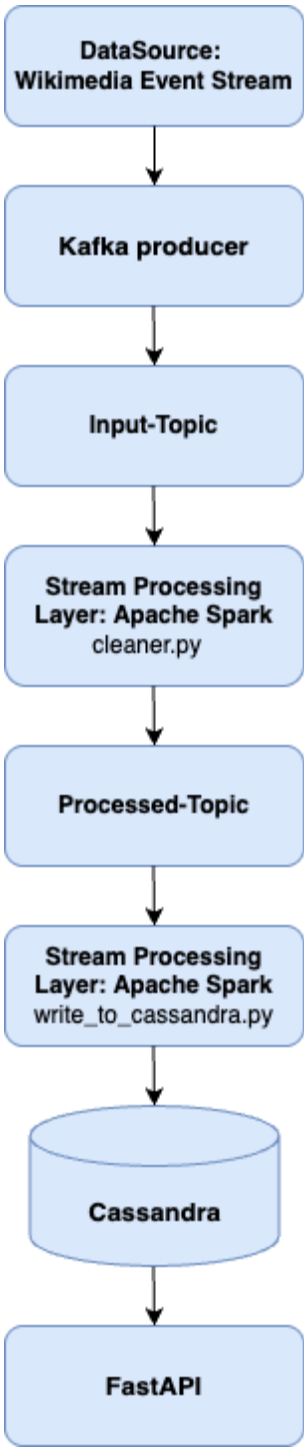
Endpoints support:

- Domain listing
- Page lookups by user, domain, or ID
- User activity within time windows
- Reports for top contributors and bot-generated content

Component: main.py (FastAPI), models.py, cassandra_client.py

2.5 Architecture Diagram

The following diagram summarizes the end-to-end flow of data through the system:



3. Data Modeling

The system uses **Apache Cassandra** as its primary storage. Apache Cassandra was selected as the primary data store due to its suitability for high-throughput, write-heavy workloads and its ability to scale horizontally across distributed

nodes. The system processes real-time events from a public stream (Wikimedia page creations) and must support fast insertions, efficient time-based filtering, and domain/user-specific aggregations.

The data model is designed in a **query-driven** fashion, meaning each table is optimized for a specific access pattern defined by API endpoints or reporting logic. Below we describe each table, its purpose, schema, and rationale behind its primary key design.

Category B: Tables Supporting Ad-hoc API Queries

3.1 pages_by_id

Schema:

```
CREATE TABLE IF NOT EXISTS pages_by_id (
  page_id TEXT PRIMARY KEY,
  user_id TEXT,
  user_name TEXT,
  domain TEXT,
  page_title TEXT,
  created_at TIMESTAMP
);
```

- **Use Case**:

  Used by the **`/pages/id/{page_id}`** endpoint to fetch all metadata for a given page.

- **Primary Key Design**:

  page_id is globally unique, allowing efficient single-row retrieval.

- **Why this design**:

  Simple direct-access table with no need for clustering keys. Ensures the lowest-latency lookups.

3.2 pages_by_user

Schema:

```
CREATE TABLE IF NOT EXISTS pages_by_user (
user_id TEXT,
created_at TIMESTAMP,
page_id TEXT,
user_name TEXT,
domain TEXT,
page_title TEXT,
PRIMARY KEY (user_id, created_at, page_id)
) WITH CLUSTERING ORDER BY (created_at DESC);
```

- **Use Case**:

  Supports the **/pages/user/{user_id}** endpoint to retrieve a user's recent page creations.

- **Primary Key Design**:
  - Partitioned by user_id
  - Clustered by created_at DESC, page_id as tie-breaker
- **Why this design**:

Supports efficient chronological access, especially for user dashboards or timeline views. The DESC ordering helps fetch recent activity without full partition scan.

3.3 pages_by_domain

Schema:

```
CREATE TABLE IF NOT EXISTS pages_by_domain (
domain TEXT,
created_at TIMESTAMP,
page_id TEXT,
user_id TEXT,
user_name TEXT,
page_title TEXT,
```

```
PRIMARY KEY (domain, created_at, page_id)
) WITH CLUSTERING ORDER BY (created_at DESC);
```

- **Use Case**:

  Used by **/pages/domain/{domain}** to count or browse domain-specific pages.

- **Primary Key Design**:
  - Partitioned by domain
  - Clustered by created_at DESC, page_id
- **Why this design**:

  Time-based ordering optimizes scans within a recent time range.

3.4 pages_by_time

Schema:

```
CREATE TABLE IF NOT EXISTS pages_by_time (
date_hour TEXT,
user_id TEXT,
user_name TEXT,
page_id TEXT,
domain TEXT,
page_title TEXT,
created_at TIMESTAMP,
PRIMARY KEY (date_hour, user_id, created_at, page_id)
) WITH CLUSTERING ORDER BY (user_id ASC, created_at DESC);
```

- **Use Case**:

  Enables /stats/time-range/ endpoint to return user activity across a time window (hourly resolution).

- **Primary Key Design**:
  - Partitioned by date_hour (e.g., “2025-05-18-13”)
  - Clustered by user_id, then created_at DESC

- **Why this design**:

  Limits partition size per hour while supporting fast per-user statistics for reporting dashboards. Clustering ensures time-bounded and user-aggregated filtering.

4. Data Ingestion & Processing

In this architecture, the system ingests live data from the Wikimedia page creation stream, processes it in real-time using Apache Spark, and stores the results in Apache Cassandra for downstream analytics and API access.

The main component in ingestion pipeline is **Apache Kafka**, which allows producers and consumers to evolve independently and provides message persistence, enabling reprocessing and fault tolerance.

The ingestion and processing workflow is organized into modular stages:

- **Kafka Producer**: A lightweight Python client connects to the Wikimedia EventStream and pushes JSON-formatted page creation events to Kafka.
- **Kafka Topics**:
  - input: the raw entry point for all events.
  - processed: the topic that stores cleaned and enriched data, ready to be persisted.
- **Stream Processing**: Two Apache Spark jobs consume these topics:
  - The **Cleaner job** reads from input, validates and enriches the data, and emits the result to processed.
  - The **Writer job** consumes from processed and inserts records into Cassandra tables optimized for various queries.
- **Cassandra** acts as the system's long-term, query-optimized storage, supporting API queries and precomputed reports.

Kafka decouples ingestion from processing, improves system robustness through backpressure handling and replay capabilities, and enables parallel consumers for scaling or extended analytics.

In the subsections below, we describe each component in detail.

4.1 Kafka-Based Event Ingestion

The system ingests real-time events from the public Wikimedia Event Stream — specifically, the page-create channel which emits structured data whenever a new Wikipedia page is created.

To stream these events into our system, we implemented a **custom Kafka producer** using Python and the kafka-python and sseclient libraries. The producer connects to the SSE stream and publishes incoming events to the Kafka **input topic**.

Kafka was chosen for its:

- **High throughput** and **low latency** guarantees.
- **Persistence**: messages are retained even if consumers are offline.
- **Scalability**: multiple consumers can process the stream in parallel.
- **Resilience**: decouples upstream (producers) from downstream (Spark jobs and storage).

Topics Used:

- input: receives raw Wikimedia JSON events.
- processed: stores validated, enriched, and formatted data suitable for Cassandra ingestion.

Kafka ensures that each stage in the pipeline operates independently and can be scaled, debugged, or restarted in isolation.

4.2 Spark Stream Processing – Cleaning Layer (cleaner.py)

The **first Spark job**, defined in cleaner.py, consumes from the input topic. Its responsibilities include:

- **Schema enforcement**: converts loosely-structured JSON into a strict schema.
- **Filtering**: removes events with missing critical fields (e.g., user_id, domain).
- **Enrichment**: derives fields like date_hour from the timestamp, which are later used for partitioning in Cassandra.
- **Transformation**: re-serializes the data into compact Kafka-compatible format.

This job outputs the cleaned dataset to the **processed topic**. Additionally, for observability, it logs samples of processed records to the console.

4.3 Spark Stream Processing – Storage Layer (write_to_cassandra.py)

The **second Spark job**, defined in write_to_cassandra.py, reads from the processed topic and writes the events into four Cassandra tables:

- pages_by_id
- pages_by_user
- pages_by_domain
- pages_by_time

This job leverages foreachBatch to:

- Log per-batch sizes and targets.
- Handle retries and failures gracefully.
- Split logic for time-based vs. common aggregations (e.g., pages_by_time includes date_hour for partitioning).

5. API Design & Results

The system also exposes a RESTful API built with **FastAPI**. The API is optimized for data access patterns defined by the underlying Cassandra tables. Each endpoint corresponds directly to a predefined query, enabling predictable latency and efficient execution.

Category B Endpoints (Ad-Hoc Queries)

These endpoints provide direct lookups or aggregations over raw data tables.

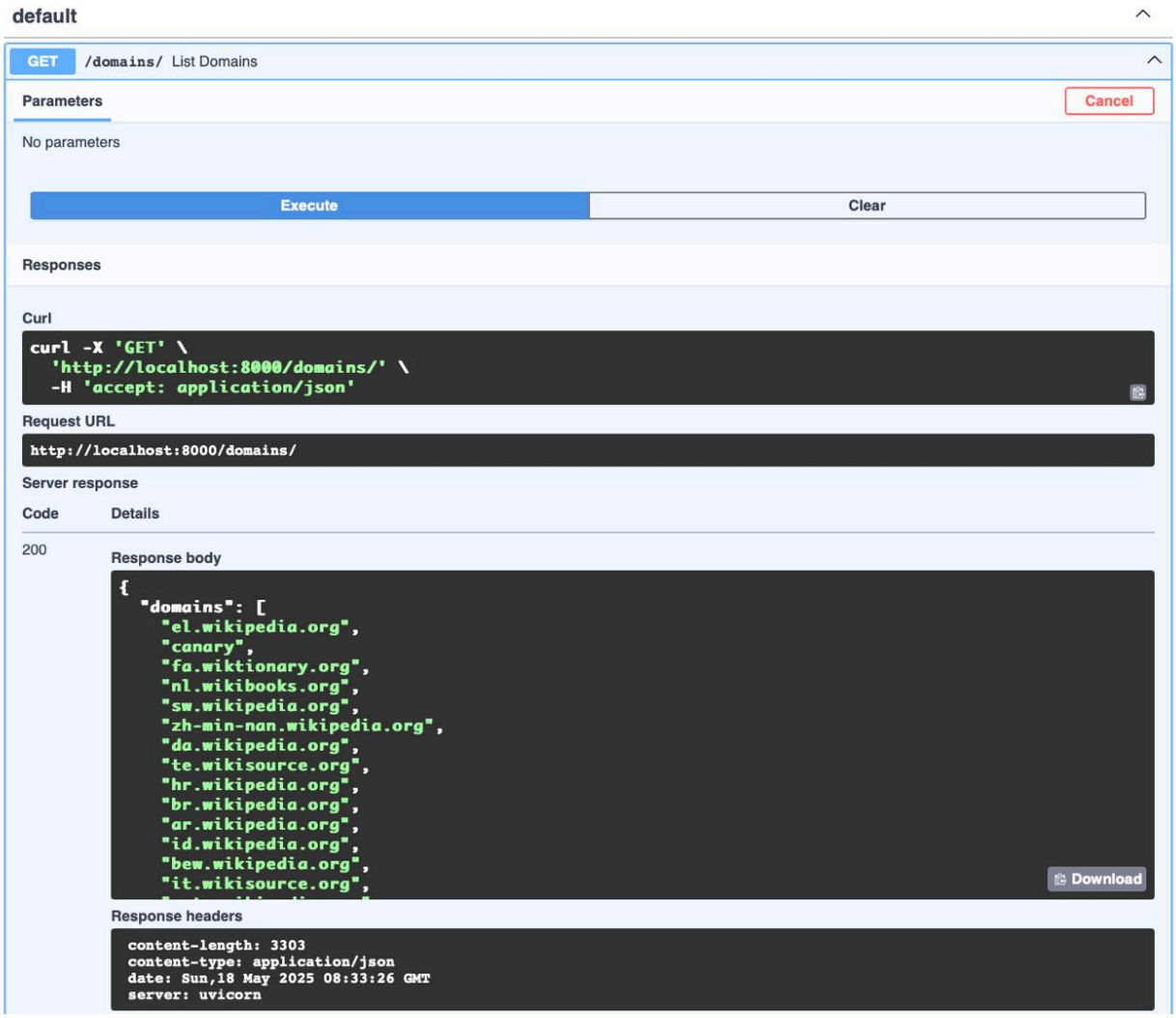
GET /domains/

Query:

```
SELECT DISTINCT domain FROM pages_by_domain;
```

Purpose: *Return the full list of domains for which pages were created.*
GET /pages/user/{user_id}

Result:



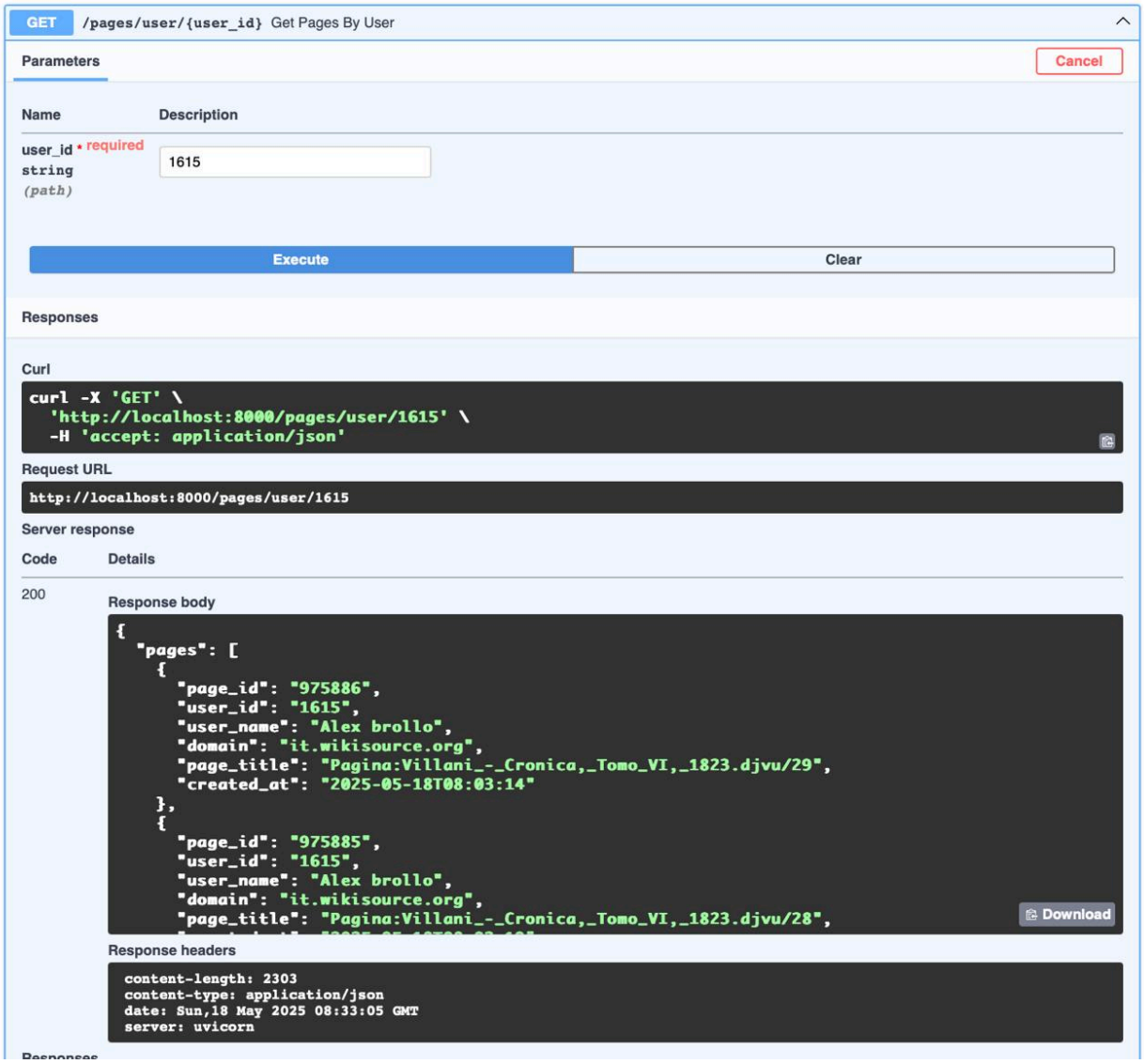
GET /pages/user/{user_id}

Query:

```
SELECT * FROM pages_by_user WHERE user_id = ?;
```

Purpose: Return **all pages** created by a given user, ordered by created_at DESC.

Result:



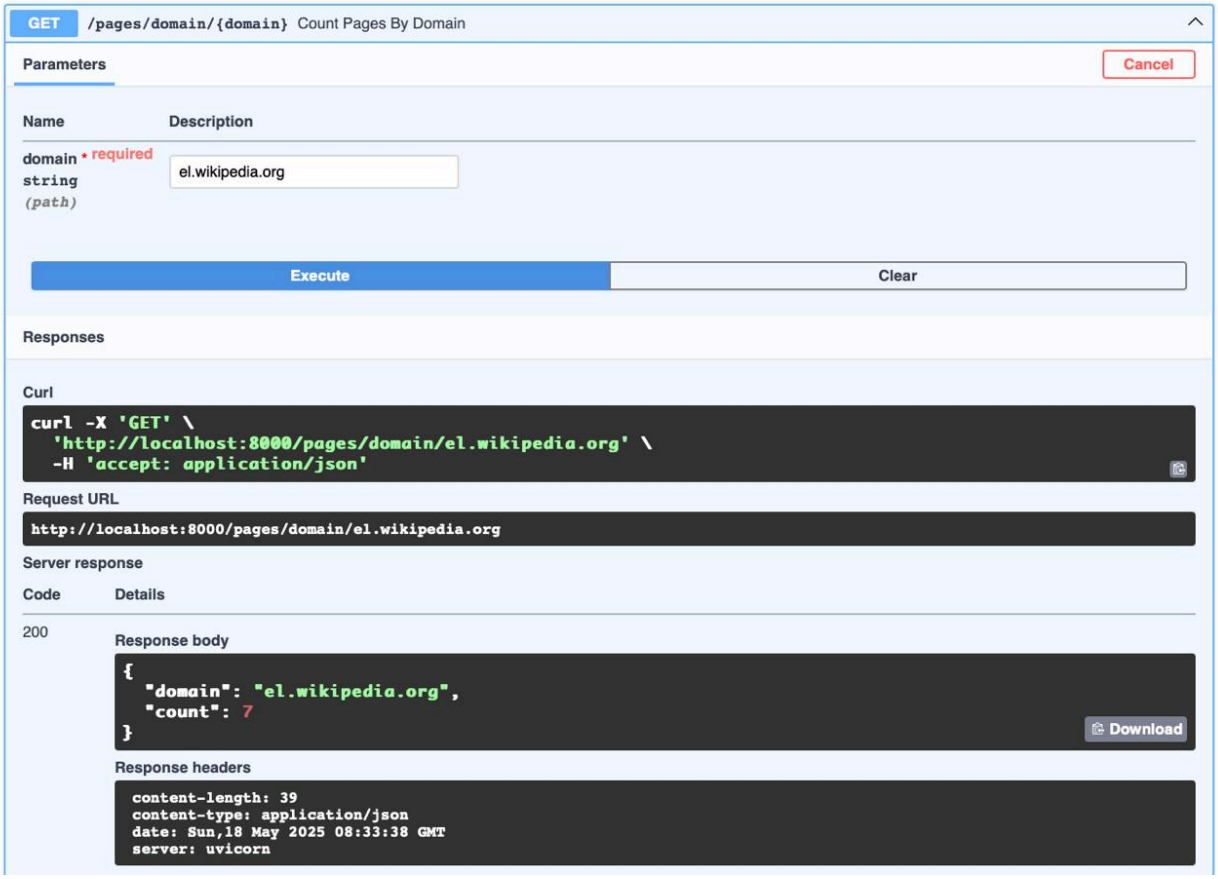
GET /pages/domain/{domain}

Query:

```
SELECT COUNT(*) FROM pages_by_domain WHERE domain = ?;
```

Purpose: Return the number of pages created for a specific domain.

Result:



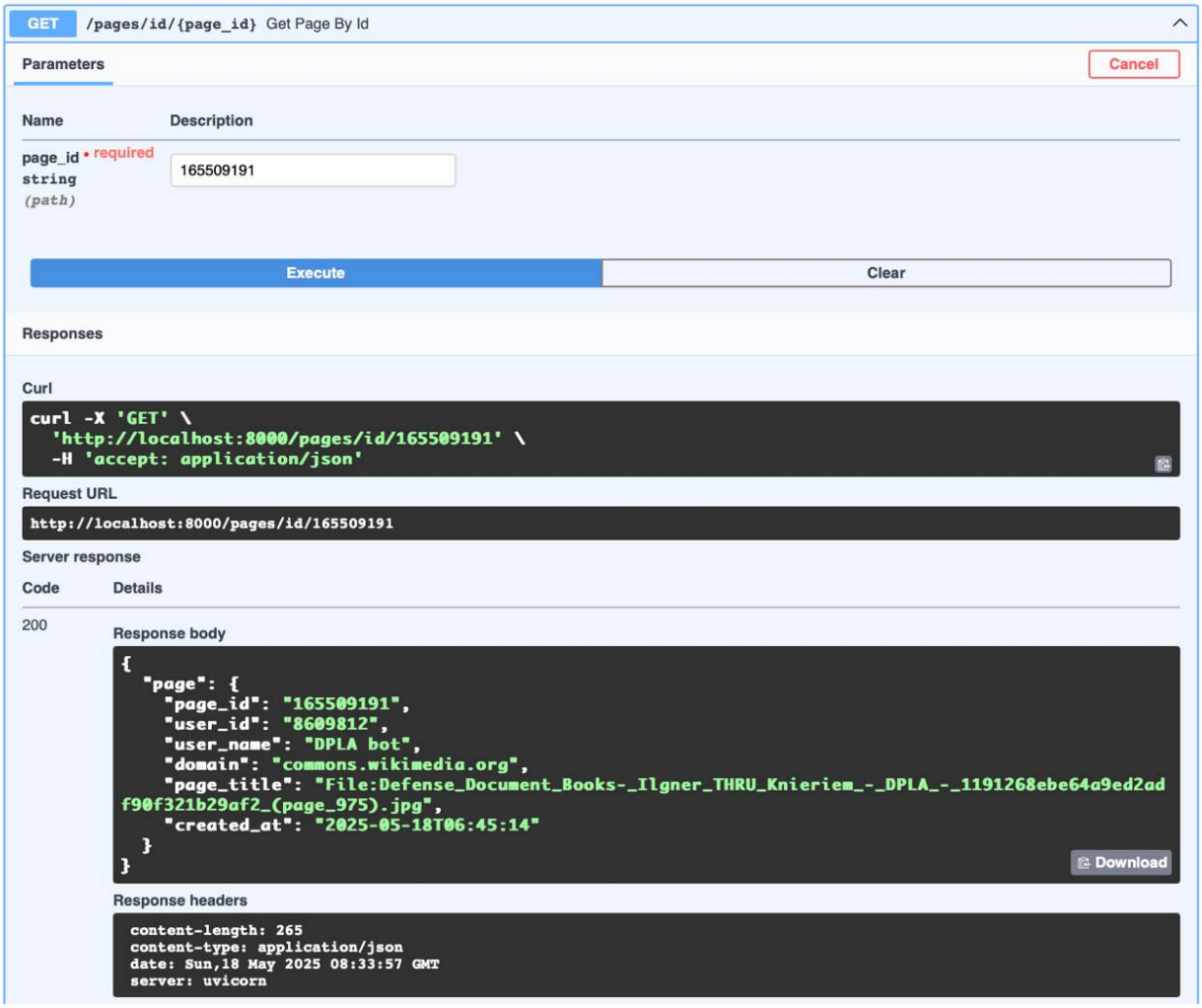
GET /pages/id/{page_id}

Query:

```
SELECT * FROM pages_by_id WHERE page_id = ?;
```

Purpose: Perform a fast, single-row lookup of page metadata by unique page ID.

Result:



GET /stats/time-range/

Query:

```
SELECT date_hour, user_id, user_name FROM pages_by_time WHERE date_hour
```

Purpose: Return users who created at least one page in a given range of full hours.

Execution Logic:

- The client specifies from_hour and to_hour (e.g., “2024-12-01-10”).
- The system computes the list of relevant date_hour partitions and queries them.

- Results are grouped by user.

Result: