CSCA 5028 - NLP / Text Mining Dashboard for Tech Articles

Joan Kusuma

Contents

Project Description

This project is a dynamic dashboard for analyzing software engineering and data science articles. Its goal is to explore trends in the field and identify which topics or types of articles resonate most with readers. The dashboard includes an exploratory data analysis and text mining, which give insights through visualization and text.

Tech Stack

- Frontend: HTML, CSS, JavaScript
- Backend: Python, Django
- Database: PostgreSQL
- Orchestration and Messaging: Airflow, RabbitMQ
- Production Environment: Gunicorn, Nginx, Docker
- Version Control: Git, Github
- Deployment: AWS (EC2, RDS, Elastic IP, ECR)

Architecture

High-Level Architecture of the system:

- **Data Collection:**
  - An Airflow orchestrator is responsible for scheduling and running the tasks here
  - The first step is by scraping publicly available archive from the most popular tech publishers on Medium.com
  - The next step is to transform and clean the data to fit the schema in PostgreSQL
  - The data is then inserted to a **message queue** with RabbitMQ, which will store every row as a JSON queue
  - The Database Worker fetch the data from the queue and insert them into PostgreSQL
- **Data Persistence:**
  - Data is saved from the data collection step into a PostgreSQL database

- **Backend System:**
  - The backend system is written in Python using the Django framework, which includes:
    - **RESTful API endpoints** to accept GET requests from Frontend
    - **Data analysis** by querying the database from PostgreSQL and analyzing the data using data analysis and nlp libraries, such as NumPy, Pandas, NLTK, etc.
    - **Testing** includes unit tests, integration tests, and using mocks to test sample data
- **Web App Basic Form / Frontend:**
  - The frontend is written in HTML, CSS, and JavaScript. Javascript is used to fetch data from the backend API endpoints and charts are visualized with Chart.js
- **Product Environment:**
  - The app is containerized with Docker, with Gunicorn serving as the WSGI HTTP server and Nginx acting as a reverse proxy to handle incoming web traffic and distribute it to the application
  - The app is deployed on AWS running the following services:
    - AWS Elastic IP for static IP address
    - AWS EC2 running web app in Docker container
    - AWS ECR where the Docker images are stored
    - AWS RDS PostgreSQL running the PostgreSQL database

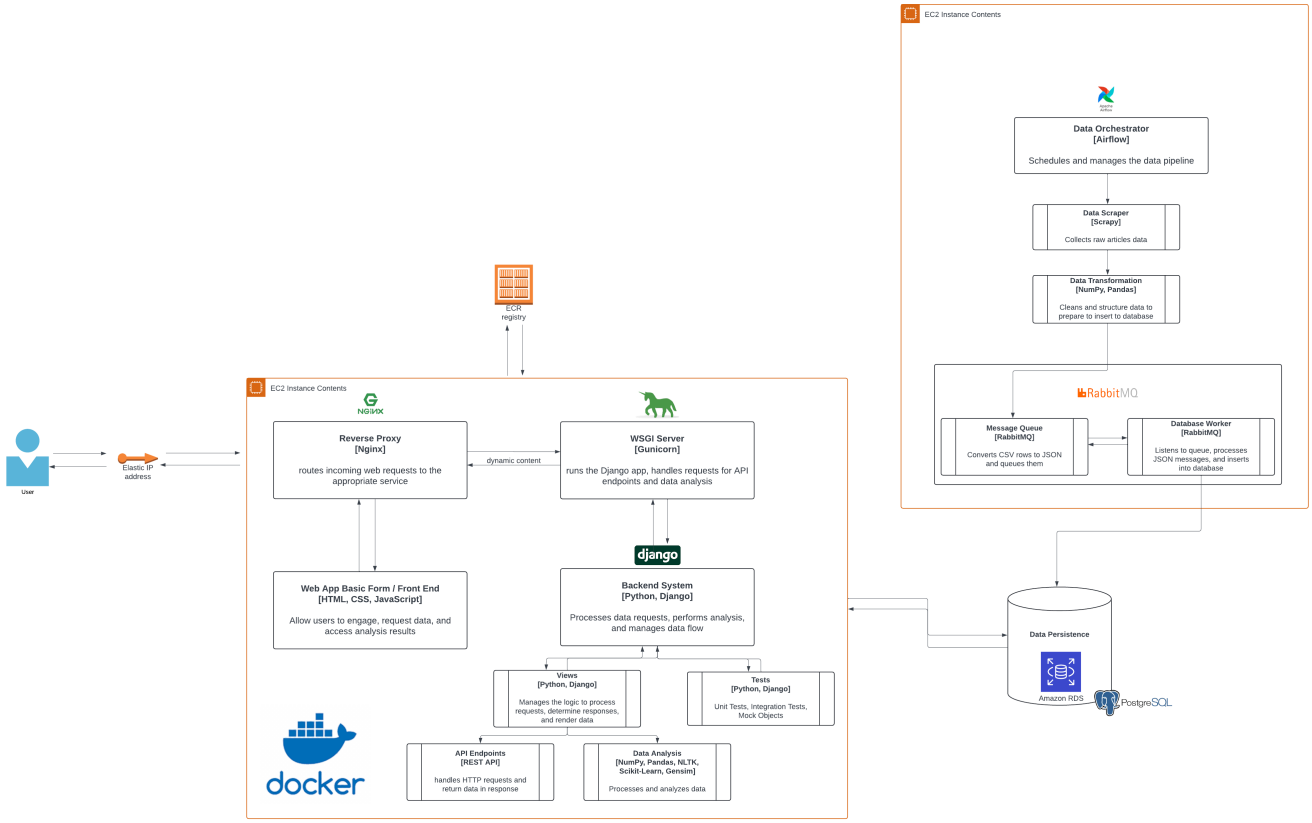


Figure: software architecture diagram of the web app

Web Application Basic Form/Reporting

To access the code to the frontend:

- `nlp-text-mining-dashboard/nlp_dashboard/nlp_dashboard/static/`
- `nlp-text-mining-dashboard/nlp_dashboardnlp_dashboard/templates/`

The frontend UI is built with HTML, CSS, and JavaScript. HTML provides the page structure, CSS styles the interface, and JavaScript fetches data from backend API endpoints, with Chart.js used to display charts.

Figure: screenshot of the frontend for data analysis dashboard

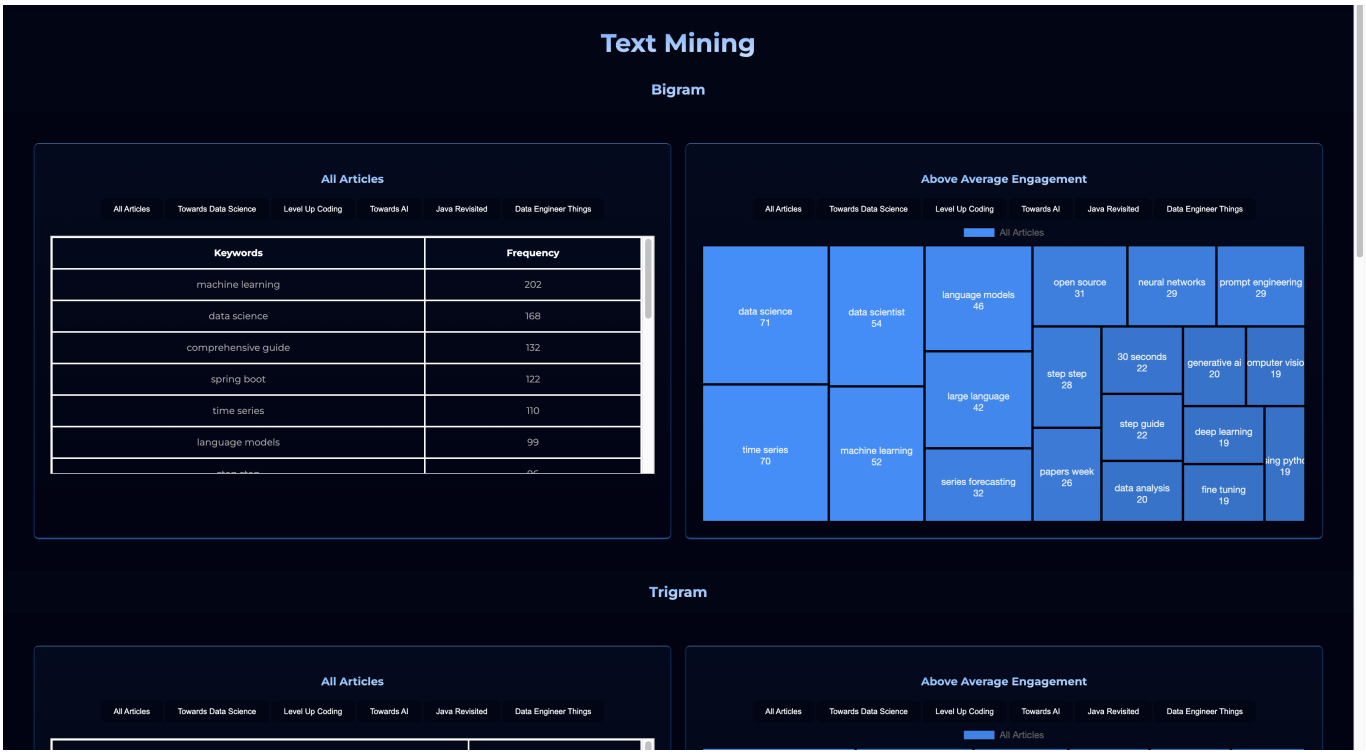


Figure: screenshot of the frontend for text mining dashboard

Data Collection

To run: see Quick Start Airflow

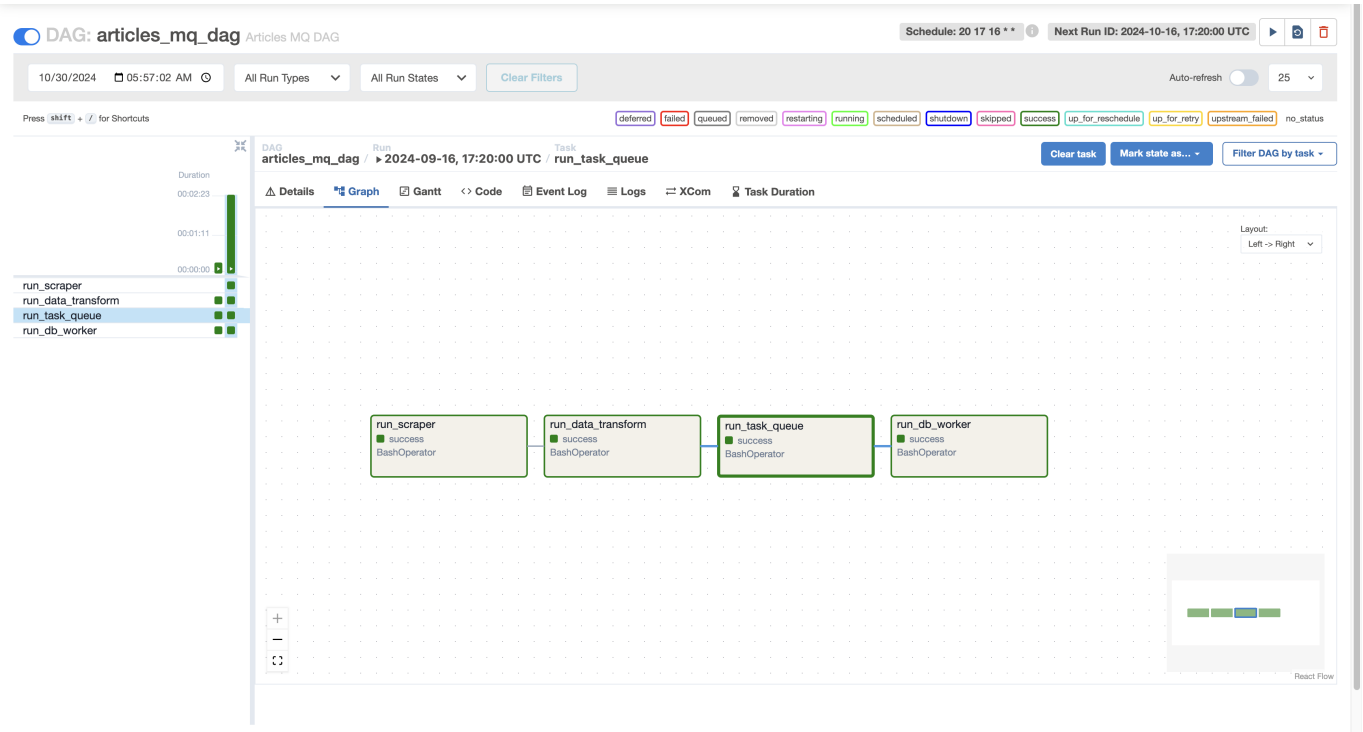
To access the code to the data collection workflow:

- DAG running scheduled bash operators on the scripts:
  - `airflow/dags/mq_articles_dag.py`
- Scraper to collect data from the website:
  - `airflow/scripts/articles_collector/articles_collector/`
- Data Transformation script to clean data:
  - `airflow/scripts/transform_data.py`
- Message Queue Publisher to queue every row as JSON
  - `airflow/scripts/message_queue/csv_task_queue.py`
- Message Queue Subscriber (worker) to consume data from the queue and insert them into PostgreSQL database
  - `airflow/scripts/message_queue/db_worker.py`

A Directed Acyclic Graph (DAG) in Airflow is a pipeline that defines the sequence and dependencies of tasks to be executed in a workflow. The DAG is scheduled to run on a monthly basis to fetch data from the previous month. In this project, the Airflow DAG manages the data collection workflow, which begins with a web scraper to collect raw data, followed by data transformation and cleaning, queues data rows in JSON, and ends with a message queue worker that inserts them into the database.

This pipeline is written in Python, using Bash operators in the DAG. The scraper is built with Scrapy and uses ProxyMesh as the proxy server. In the data transformation step, NumPy and Pandas are used to clean and structure the data to fit the database schema. RabbitMQ handles message queueing, with each cleaned data row queued in JSON. Finally, an MQ consumer (worker) listens to RabbitMQ, consumes the data, and inserts it into the PostgreSQL database.

I chose RabbitMQ because of the acknowledgment system, which confirms successful consumption of each message before deletion, reducing data loss.



Successful DAG run of the steps outlined in data collection step

Data Analyzer

To access the code to the data analyzer:

- `nlp-text-mining-dashboard/nlp_dashboard/nlp_dashboard/nlp_app/views.py`

The data analysis part reads data from the PostgreSQL database, it is written in Python and uses these libraries to perform analysis operations: NumPy, Pandas, Scikit-Learn, NLTK, Gensim, and PyLDAvis. It's split into 2 pages, the first page contains a general exploratory data analysis, and the second page contains the NLP/text mining analysis.

A quick rundown of the analysis performed is available on the report `data-text-analysis.pdf`

Testing and Mock Objects

For this project, I incorporated unit testing, integration testing, and used mock objects. I'm going to briefly go through some code for each testing type that I have used for my project.

For full code of the tests: `nlp-text-mining-dashboard/nlp_dashboard/nlp_dashboard/nlp_app/tests.py`

- **Unit Testing**

Unit tests are used to test individual components or functions of the code in isolation. In this example, the goal is to test the cache functionality of the `get_articles_data` function:

```
class GetArticlesDataCacheTest(TestCase):
    def setUp(self):
        # Clear cache before each test
```

```
        cached_articles_data.clear()
        ...

        # Call to check cache miss
        get_articles_data()
        self.assertIn("all", cached_articles_data)
```

- **Integration Testing**

Integration tests check how various components of the application work together. In this example, it sends an HTTP request to the `releases-claps-by-week` view and checks if it returns a successful response (status code 200). This verifies that multiple components (the client, view, and database) interact correctly to produce the expected result.

```
class GetReleasesClapsByWeekTests(TestCase):
    ...
    # Make a request to the view
    response = self.client.get(reverse("releases-claps-by-week"))

    # Verifying the response structure and data
    self.assertEqual(response.status_code, 200)
```

- **Mock Objects**

Mocking is used to simulate external dependencies, such as database calls or external APIs, to isolate the functionality being tested.

In this example, the `@patch("nlp_app.views.get_articles_data")` decorator is applied to replace the real `get_articles_data` function with a mock object. It is used to simulate the return value of `get_articles_data` without actually fetching any articles from the database or external service. The mock is configured to return a predefined `DataFrame` (with mock data) when called.

```
class PublisherCountTest(TestCase):
    @patch("nlp_app.views.get_articles_data")
    def test_get_publisher_count(self, mock_get_articles_data):
        mock_data = pd.DataFrame(
            [
                {"collection": "Towards Data Science"},
                {"collection": "Towards Data Science"},
                {"collection": "Towards AI"},
                {"collection": "Level Up Coding"},
            ]
        )
        mock_get_articles_data.return_value = mock_data
        ...
```

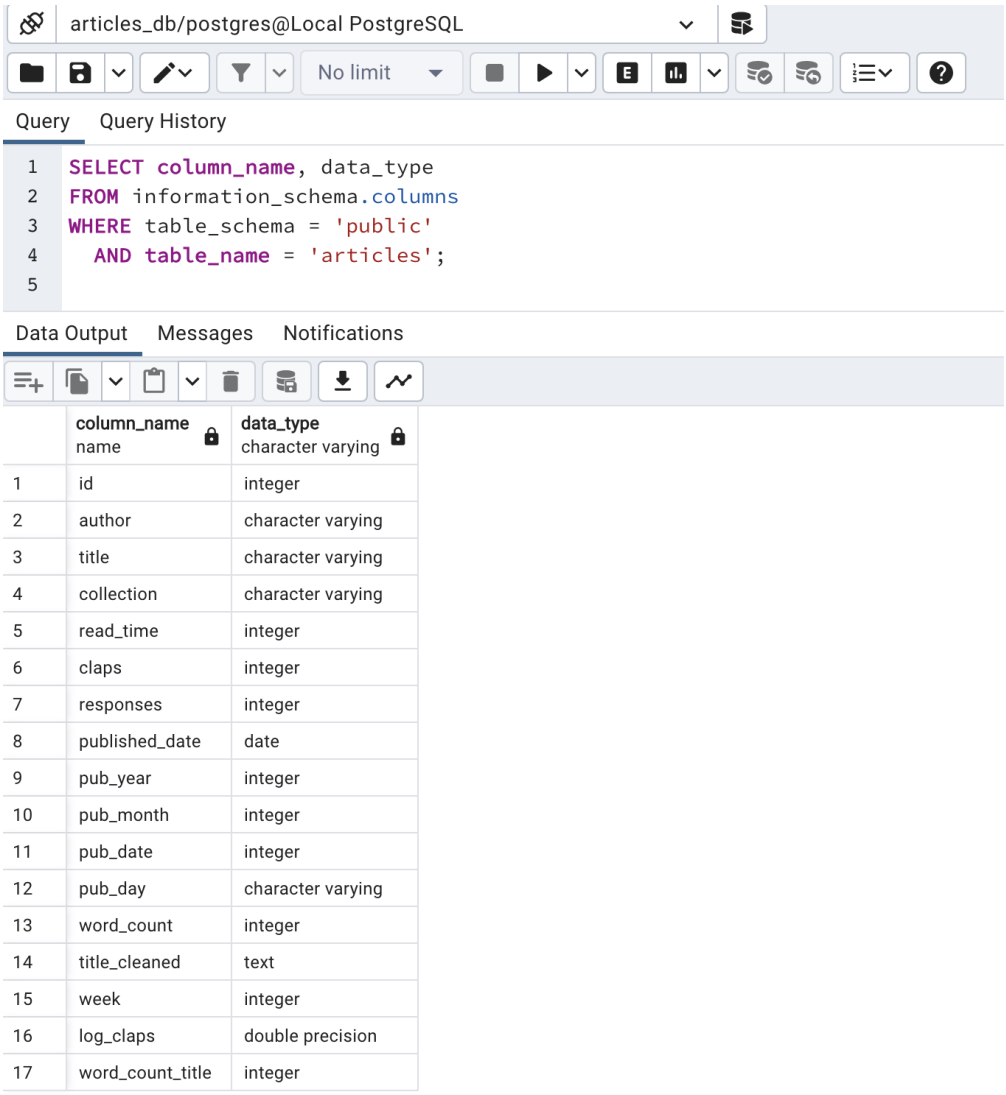
Data Persistence

The code to the data model is available at: `nlp-text-mining-dashboard/nlp_dashboard/nlp_dashboard/nlp_app/models.py`

In this project, the data is stored in **PostgreSQL** relational database. It is saved in an **articles** table with 17 columns, which captures key information such as the article's title, author, publication date, and various engagement metrics.

```
articles_db=# \d public.articles
                                  Table "public.articles"
      Column      |          Type          | Collation | Nullable |               Default
------------------+------------------------+-----------+----------+--------------------------------------
 id               | integer                |           | not null | nextval('articles_id_seq'::regclass)
 author           | character varying(255) |           | not null |
 title            | character varying(255) |           | not null |
 collection       | character varying(255) |           | not null |
 read_time        | integer                |           |          |
 claps            | integer                |           |          |
 responses        | integer                |           |          |
 published_date   | date                   |           |          |
 pub_year         | integer                |           |          |
 pub_month        | integer                |           |          |
 pub_date         | integer                |           |          |
 pub_day          | character varying(50)  |           | not null |
 word_count       | integer                |           |          |
 title_cleaned    | text                   |           |          |
 week             | integer                |           |          |
 log_claps        | double precision       |           |          |
 word_count_title | integer                |           |          |
:
```

Figure: articles table schema accessed through psql



	column_name name	data_type character varying
1	id	integer
2	author	character varying
3	title	character varying
4	collection	character varying
5	read_time	integer
6	claps	integer
7	responses	integer
8	published_date	date
9	pub_year	integer
10	pub_month	integer
11	pub_date	integer
12	pub_day	character varying
13	word_count	integer
14	title_cleaned	text
15	week	integer
16	log_claps	double precision
17	word_count_title	integer

Figure: articles table schema accessed through pgadmin

REST Collaboration Internal or API Endpoints

The code for the API endpoints is available at: `nlp-text-mining-dashboard/nlp_dashboard/nlp_dashboard/nlp_app/views.py`

In this project, I am implementing RESTful collaboration using Django's built-in API endpoints to fetch and process article data. These endpoints provide insights and analysis as discussed in the `data-text-analysis.pdf' file.

Each endpoint interacts with the `Articles` model, processes data, and returns the results in JSON format. For performance, results are cached to avoid repeated database queries. The APIs support GET requests and are designed to handle publisher-specific data or default to all available data.

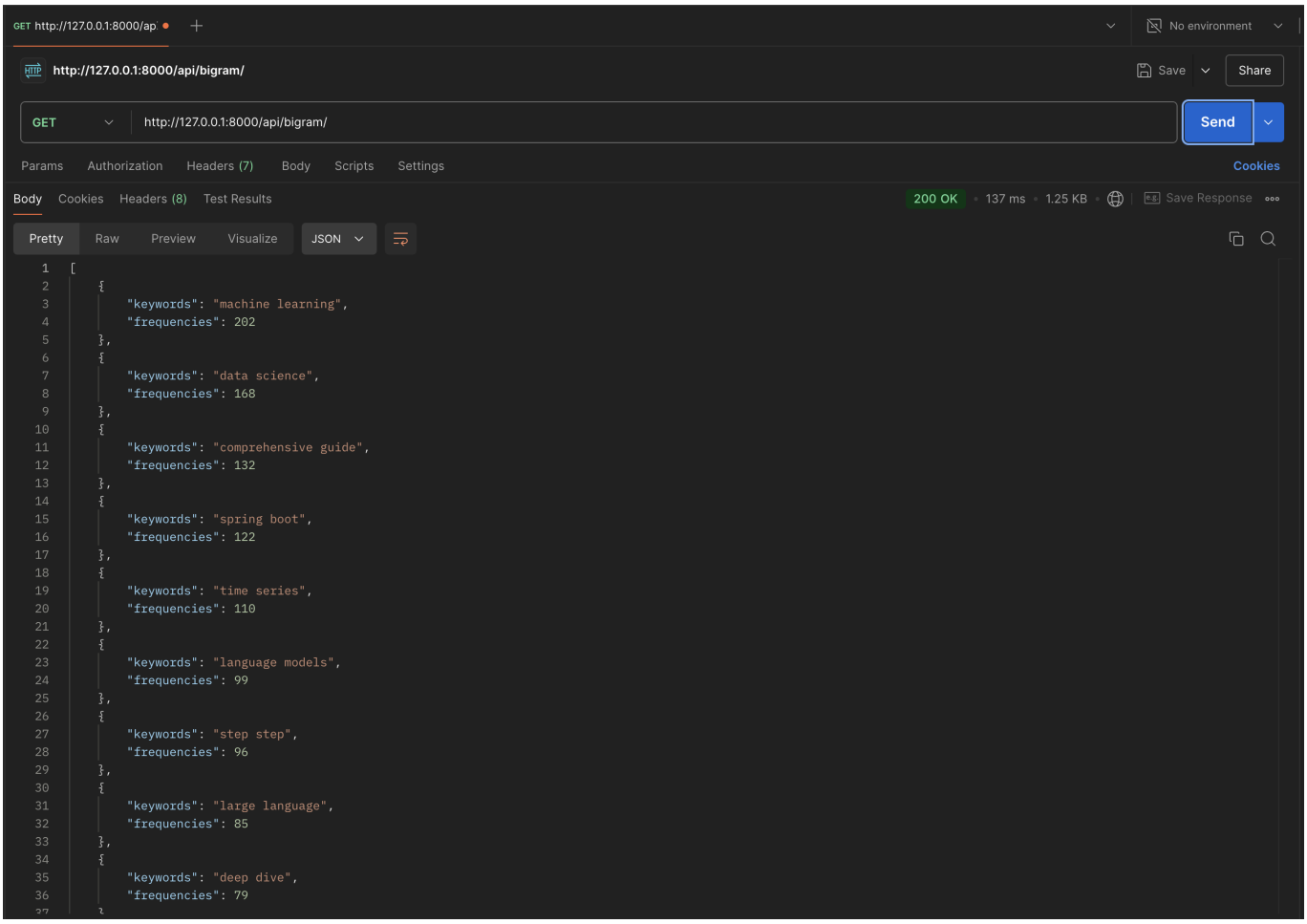


Figure: Testing the API Endpoint for a GET request via Postman

Product Environment

The web app is accessible through the domain name: https://teas.cafe

The following files contain the product environment code:

- ./nginx
- Dockerfile.prod
- docker-compose.prod.yml
- entrypoint.prod.sh

I used the following:

- AWS Cloud services for deployment
  - User connects via HTTPS to https://teas.cafe
  - the browser sends a DNS query to find the IP address of the domain and maps the domain to the AWS Elastic IP address
  - AWS Elastic IP provides the IP address, it is also connected to the EC2 resource
  - EC2 instance runs the web app, it is also connected to AWS RDS PostgreSQL to access data
- Nginx as web server and reverse proxy
  - It acts as the entry point for user requests when they connect to the web app
  - Nginx essentially receives the request first when the request hits EC2
  - It terminates the HTTPS connection (decrypts the connection using the SSL certificate)

    - It then checks if the request is for static content or dynamic, if dynamic, Nginx forwards it to Guniconr
- Gunicorn as application server
    - Gunicorn runs the Python code to query the database or perform backend logic
    - It then returns the response to Nginx
- Docker acts as container
    - The Docker images are stored in ECR
    - EC2 pulls images from ECR and build and run the image in the EC2 instance

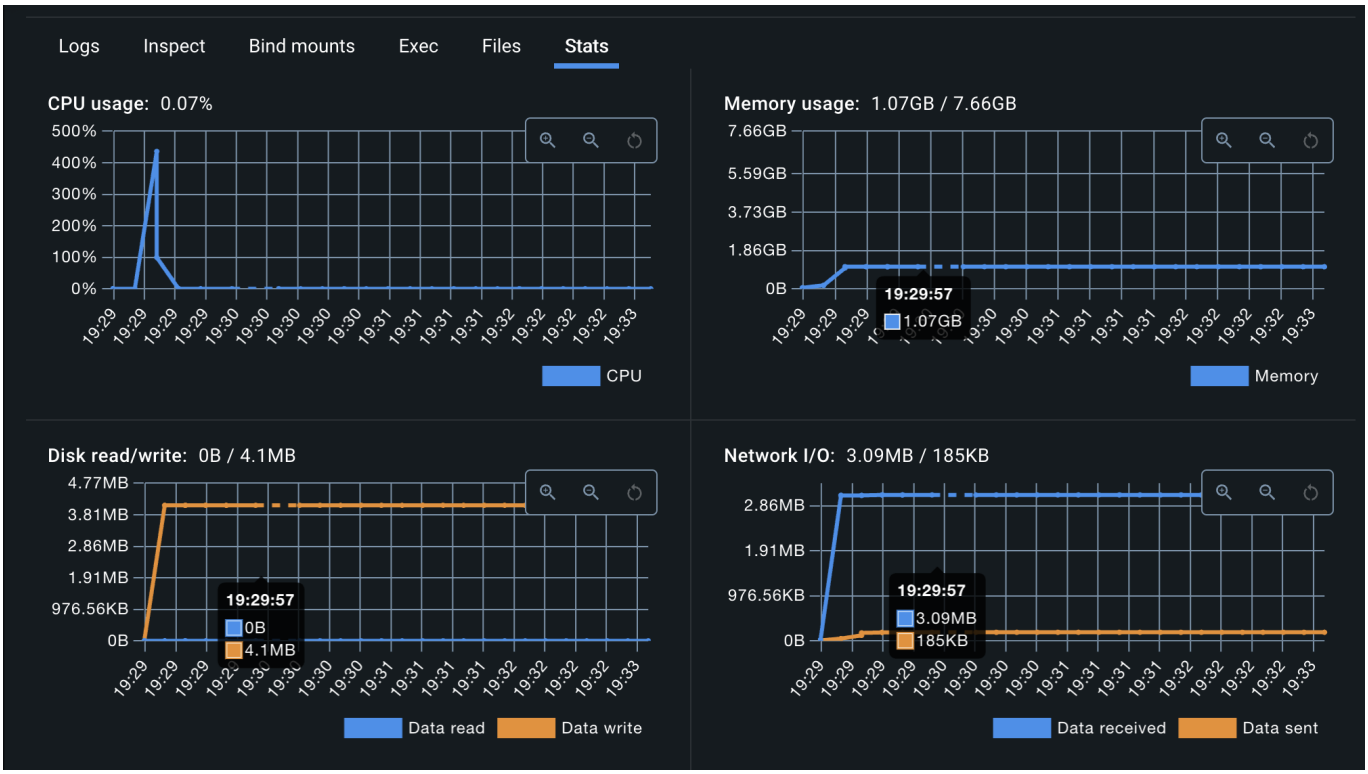


Figure: Test Run of the Docker stats for the web app

Event Collaboration Messaging

The code for the messaging queue's publisher and consumer are available at:

- `airflow/scripts/message_queue/csv_task_queue.py`
- `airflow/scripts/message_queue/db_worker.py`

In this project, event collaboration messaging is implemented using RabbitMQ to pass CSV data from a producer `csv_task_queue.py` to a consumer `db_worker.py`. The producer reads data from a CSV file, serializes each row as JSON, and sends it to a RabbitMQ queue. The consumer listens to this queue, processes the incoming messages, validates the data, and inserts it into a PostgreSQL database.

The consumer acknowledges each message after processing and checks if the queue is empty, stopping further consumption when all data is processed. This setup allows for efficient, decoupled communication between components in the system.

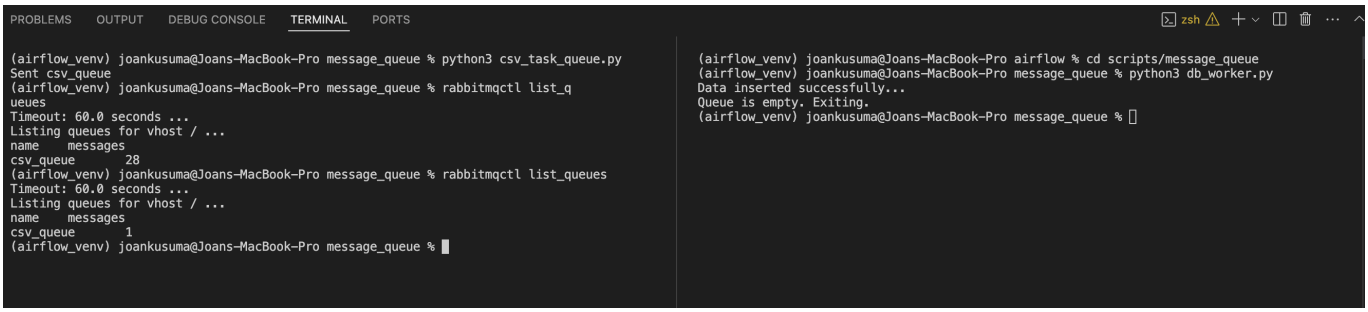


Figure: (left) message queue before emptying, and after emptying, (right) queue listener processing messages and inserting them into PostgreSQL

Quickstart Web App(without Docker)

Please make sure you have Postgres installed in your local computer before your start the following steps

You can set up the database following the Airflow steps below or use the backup.dump

Set up your Postgres Database:

```
pg_restore -U <your_username> -h localhost -d new_database_name -v
/path/to/backup.dump
```

Navigate into the project's directory:

```
cd nlp_dashboard/nlp_dashboard
```

Comment the following lines in settings.py as this is for production only:

```
DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"

SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTO", "https")

CSRF_TRUSTED_ORIGINS = os.environ.get("CSRF_TRUSTED_ORIGINS").split(" ")
```

Create a .env file and include the following information:

```
DEBUG=1 # leave as is
SECRET_KEY = 'insert-a-secret-key-here-can-be-anything' # you can leave as
is or change to your preference
DJANGO_ALLOWED_HOSTS=localhost,0.0.0.0,127.0.0.1 # leave as is

SQL_ENGINE = 'django.db.backends.postgresql' # leave as is
SQL_NAME = 'db-name-you-set-up-earlier'
SQL_USER = 'your-postgres-username'
SQL_PASSWORD = 'your-postgres-password'
SQL_HOST = 'localhost' # leave as is
```

```
SQL_PORT = '5432' # leave as is
DATABASE = 'postgres' # leave as is
```

Create and activate your virtual environment, install dependencies, and run

```
python -m venv venv_name
pip install -r requirements.txt
source venv_name/bin/activate

# inspect the database, copy the models over to models.py
python manage.py inspectdb
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

Quick Start Airflow (Data Collection and Message Queue)

Please make sure you have Postgres installed prior to the following steps

Airflow Postgres Set up

```
createdb -U postgres database_name
psql -U postgres
CREATE USER user_name with PASSWORD 'your-password-here';
GRANT ALL PRIVILEGES ON DATABASE database_name to user_name;
```

Create and activate your virtual environment

```
python -m venv airflow_venv
source airflow_venv/bin/activate
pip install -r requirements.txt
```

(if the above doesn't work) install airflow with postgres

```
pip install apache-airflow-providers-postgres
```

Change airflow.cfg (look for sql_alchemy_conn in the file)

```
sql_alchemy_conn = postgresql+psycopg2://<insert-your-pg-username>:<your-
postgres-password>@localhost:5432/<database-name>
# for example
#sql_alchemy_conn =
postgresql+psycopg2://user_name:password1234@localhost:5432/database_name
```

Go to your airflow folder and initialize

```
airflow db init
```

Create a new airflow user

```
airflow users create --username username_you_want --password
password_you_want --firstname your_first_name --lastname your_last_name --
role Admin --email example@email.com
```

List airflow users to make sure it's there

```
airflow users list
```

Run Airflow scheduler

```
airflow scheduler
```

In a separate (new) terminal, run the web server (remember to activate your airflow venv)

```
airflow webserver
```

Access the Airflow DAG UI through the link given, for example

```
http://localhost:8000/
```