How to build a real-time News Search Engine using Vector DBs

Razvant Alexandru · Following

Published in Decoding ML · 19 min read · Apr 13, 2024

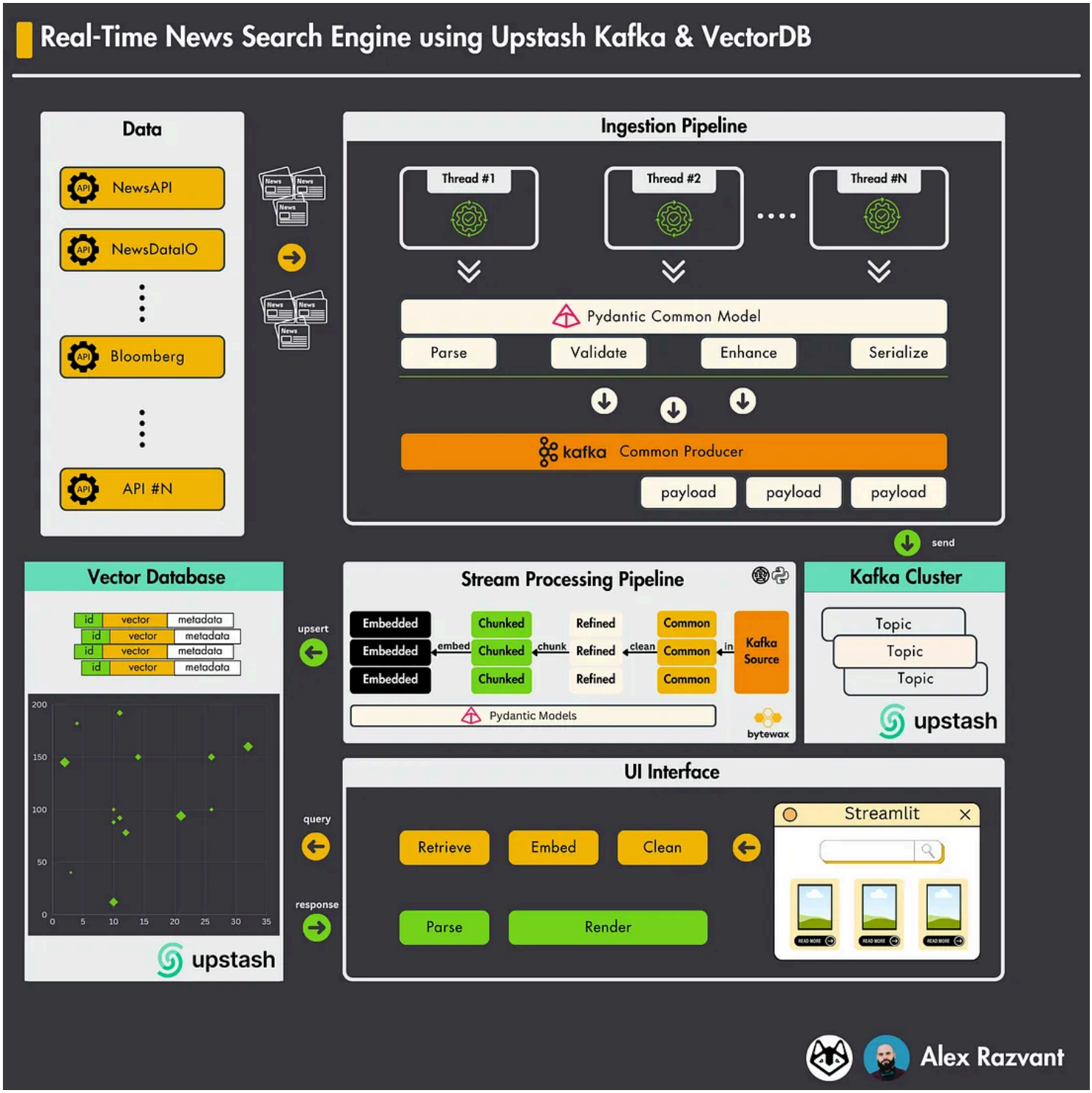
A hands-on guide to implementing a live news aggregating streaming pipeline with Apache Kafka, Bytewax, and Upstash Vector Database.

According to a research study done by earthweb.com, the **daily** influx of news articles, both online and offline, lies between **2 to 3 million!**

With so much news coming at us from all sides, it's tough to keep up. That's why there's a big push for shorter, faster ways to get the news we care about.

In this article, we're aiming to tackle just that!
More precisely, we will build a system that can ingest from multiple sources and then narrow down the information channel to an endpoint tailored to your interests — **A News Search Engine**!

We will not just talk about theory, or tell you how to implement such a system — we'll tell and show step-by-step!

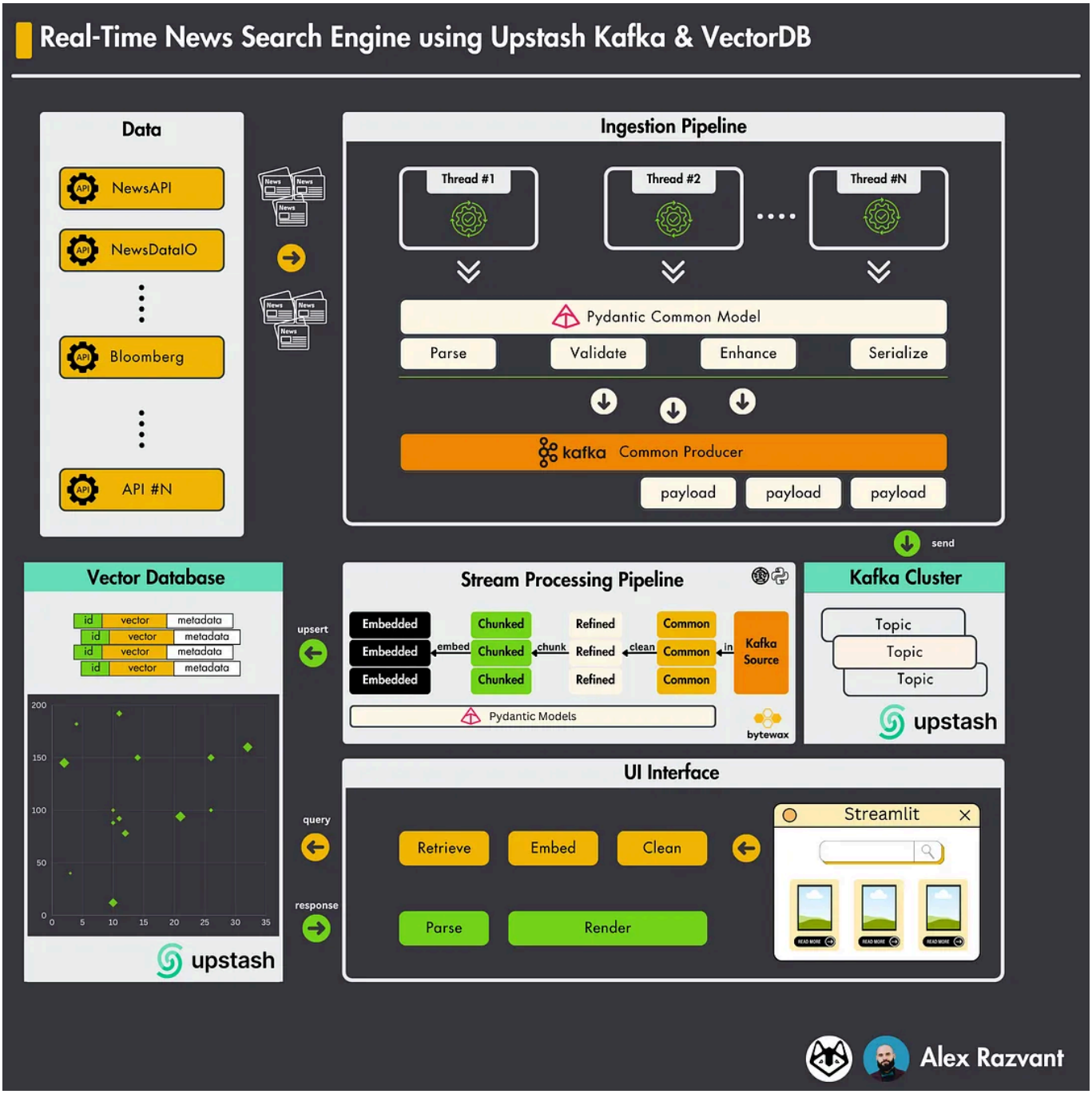


Full Architecture (Image by Author)

Before diving in, note that everything in this article has full code support on the **Decoding ML Articles GitHub Repository** [1].

Table of Contents

Architecture Overview



Full Architecture (Image by Author)

As a summary, we're going to build a **live system** that fetches news articles from live news APIs, parses and formats the raw data into our defined formats then serializes and streams messages to a Kafka Topic as a first stage.

The second stage will use Bytewax to streamline the messages from our Kafka Topic by further cleaning, parsing, chunking, embedding, and upserting vectors to a Vector Database, ending with a UI from which we can interact with our database.

Tooling Considerations

Selecting the right tools was key to achieving high performance, scalability, and ease of implementation. Let's iterate over the tooling used throughout the project:

- **Upstash Serverless Kafka**: for robust, scalable event streaming without worrying about infrastructure management.
- **Python Threading**: for concurrently fetching data from multiple news APIs, while sharing the thread-safe Kafka Producer instance, optimizing memory footprint and performance.
- **Pydantic Models**: to ensure consistent and validated data exchange structures.
- **Bytewax**: for its simplicity and speed in processing and transforming streaming data.
- **Upstash Vector Database**: Serverless, easy to set up, easy to integrate within Python, JS, and GO. A big plus is the rich navigation options from the UI console dashboard and the real-time status metrics.

As per hardware requirements, *no GPU is required.*

How much does it cost? — **FREE.**
I've tailored this guide ***to use the free-tier plans only*** *— so you* ***won't have to pay*** *anything for the platforms used!*

Prerequisites

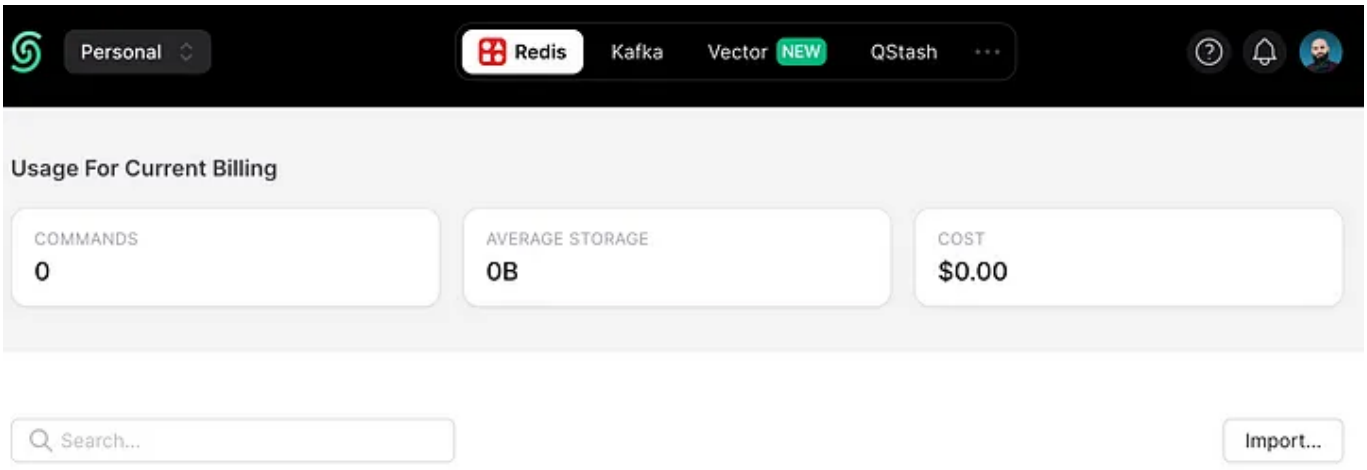
Before implementing anything, we have to make sure we can access each service, meaning we'll have to set up the following:

- *A new Upstash Kafka Cluster*
- *A new Upstash Vector Index*
- *Registering to News APIs*
- *Install environment*

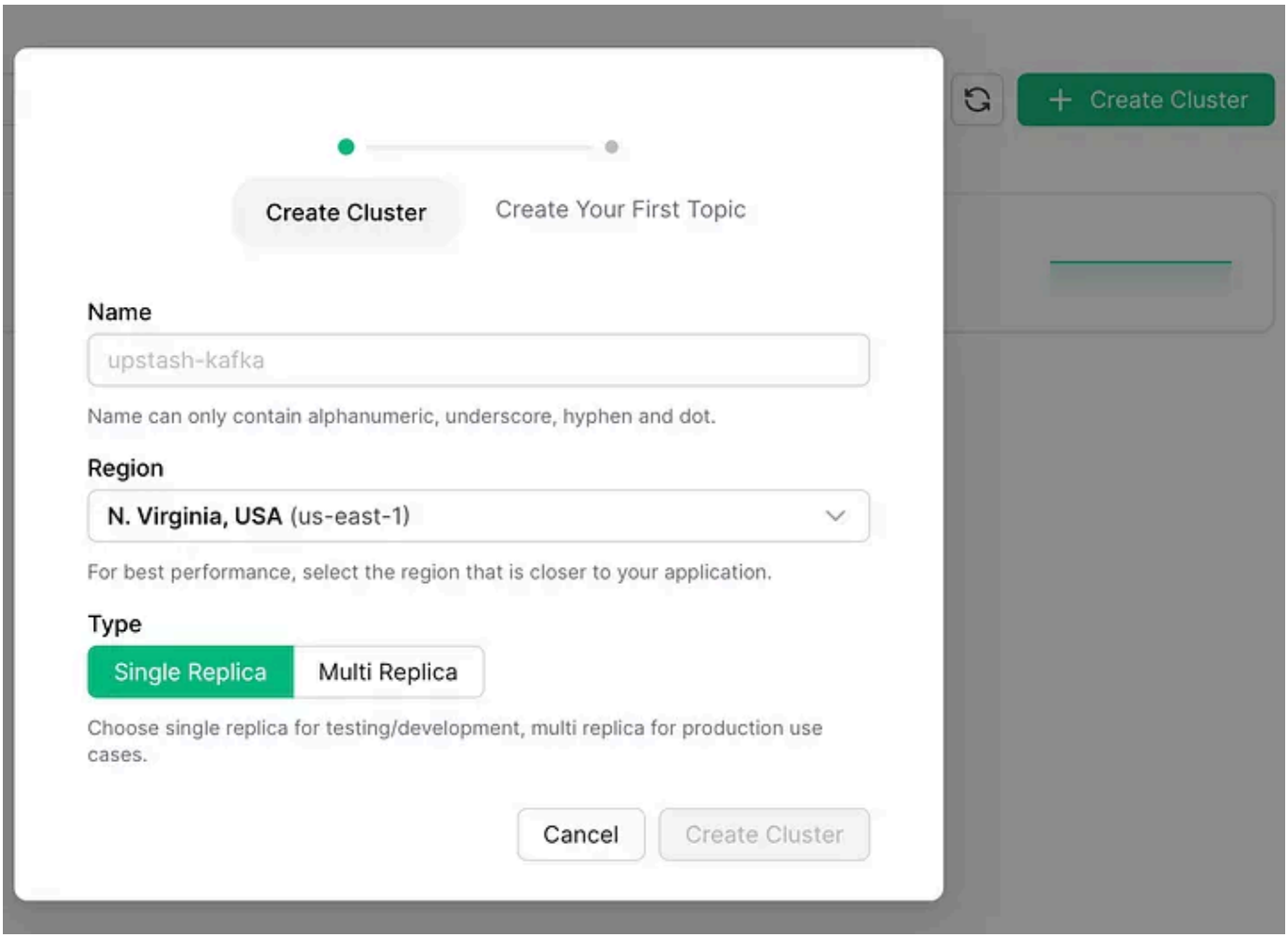
It won't take long — around 5 minutes on a fresh start.

Creating a new Upstash Kafka Cluster

First, we'll have to register to Upstash, and after logging in, a dashboard like this will appear:

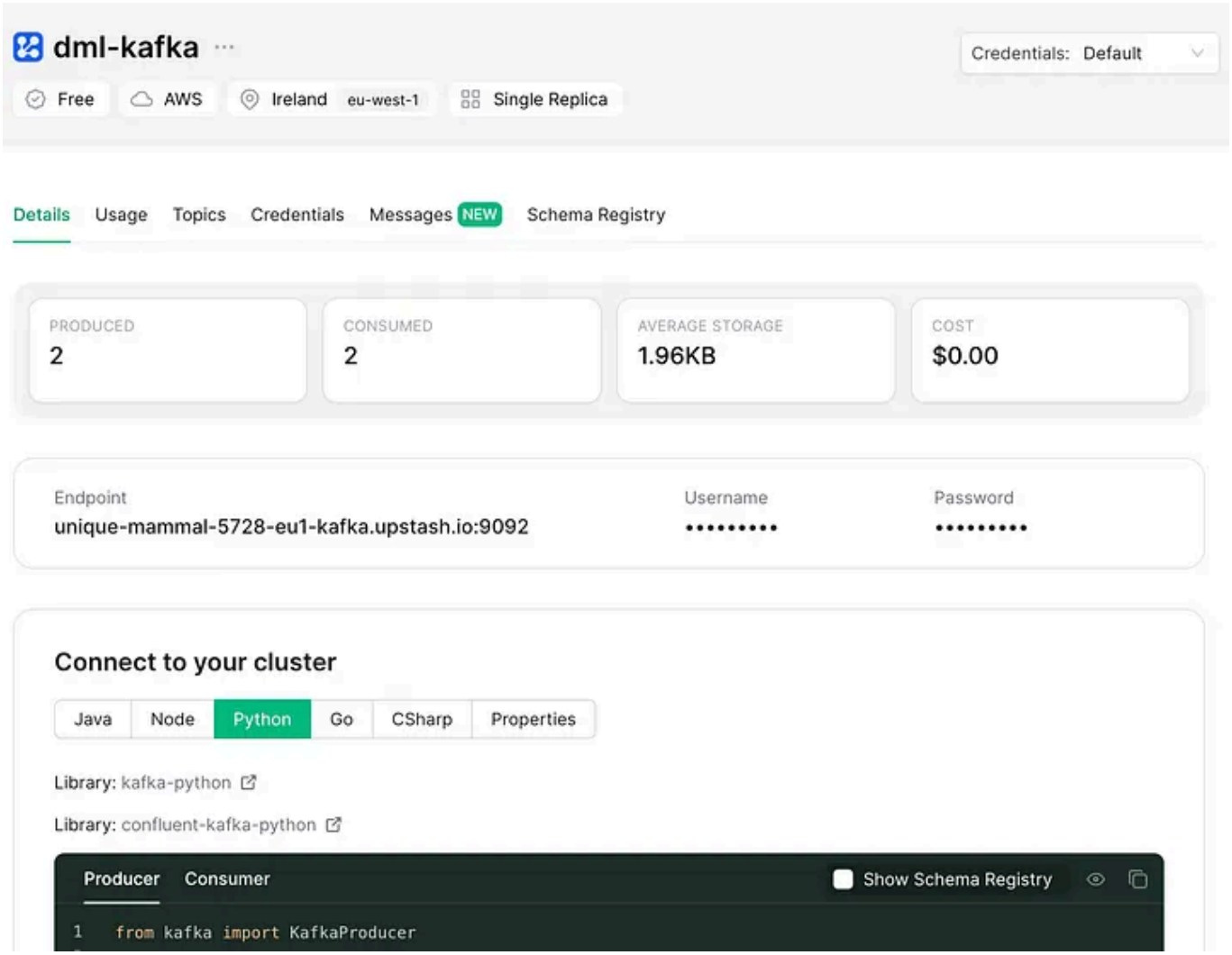


Next, we would have to select **Kafka** from the top bar and create a new cluster using the + *Create Cluster* button. Once clicked, you'll be prompted to this modal:



Screenshot from Upstash -> Kafka

Give a name to your cluster, select a region closest to your location, and end by clicking *Create Cluster*. Once done, your new Kafka cluster will appear below. Go ahead and select your new Kafka cluster and you'll be prompted to this view:

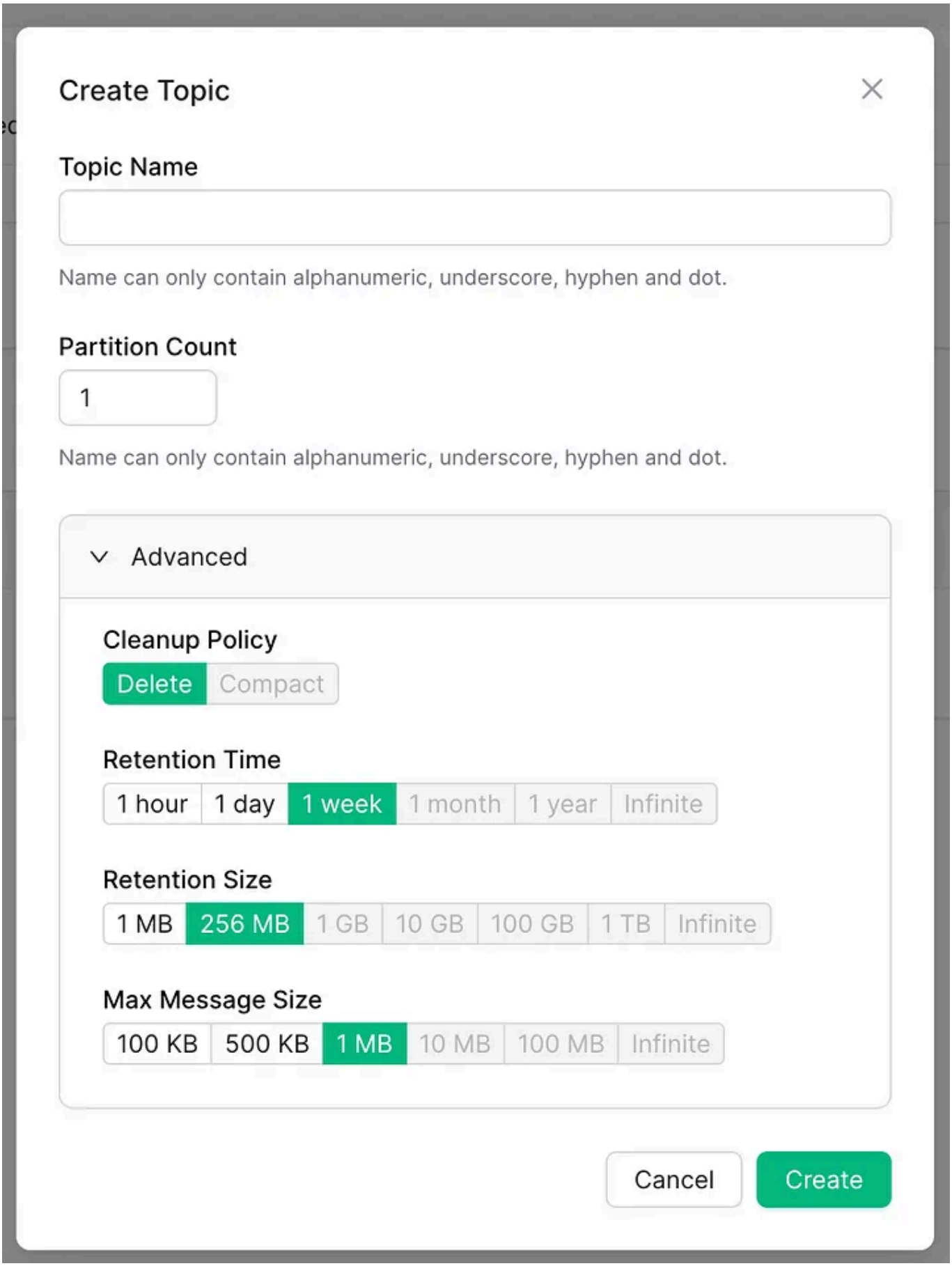


Let's unpack key elements in this view:

- **Details:** shows you an overview of your cluster and functionalities provided by Upstash
- **Usage:** shows you charts on how many messages have been produced/consumed, cost implications, etc.
- **Topics:** this tab will allow you to create and monitor details on created Kafka topics.

The next step after creating a cluster is to define a topic to which we can **produce** (send) and **consume** (get) messages.

Under the **Topics** tab, select *Create Topic* and you'll be prompted to this view:


Create Topic
Topic Name
Name can only contain alphanumeric, underscore, hyphen and dot.
Partition Count
1
Name can only contain alphanumeric, underscore, hyphen and dot.
Advanced
Cleanup Policy
Delete
Compact
Retention Time
1 hour
1 day
1 week
1 month
1 year
Infinite
Retention Size
1 MB
256 MB
1 GB
10 GB
100 GB
1 TB
Infinite
Max Message Size
100 KB
500 KB
1 MB
10 MB
100 MB
Infinite
Cancel
Create

Assign a topic name, you can leave the rest on default and end by clicking *Create.* ***It's that easy***

> *Under the advanced tab, you can customize what will happen with the messages on the topic. In the default case, one message cannot exceed 1MB in size, the max size of the stored messages on the topic is 256MB (after which it starts to delete older ones) and the topic is cleared after one week.*

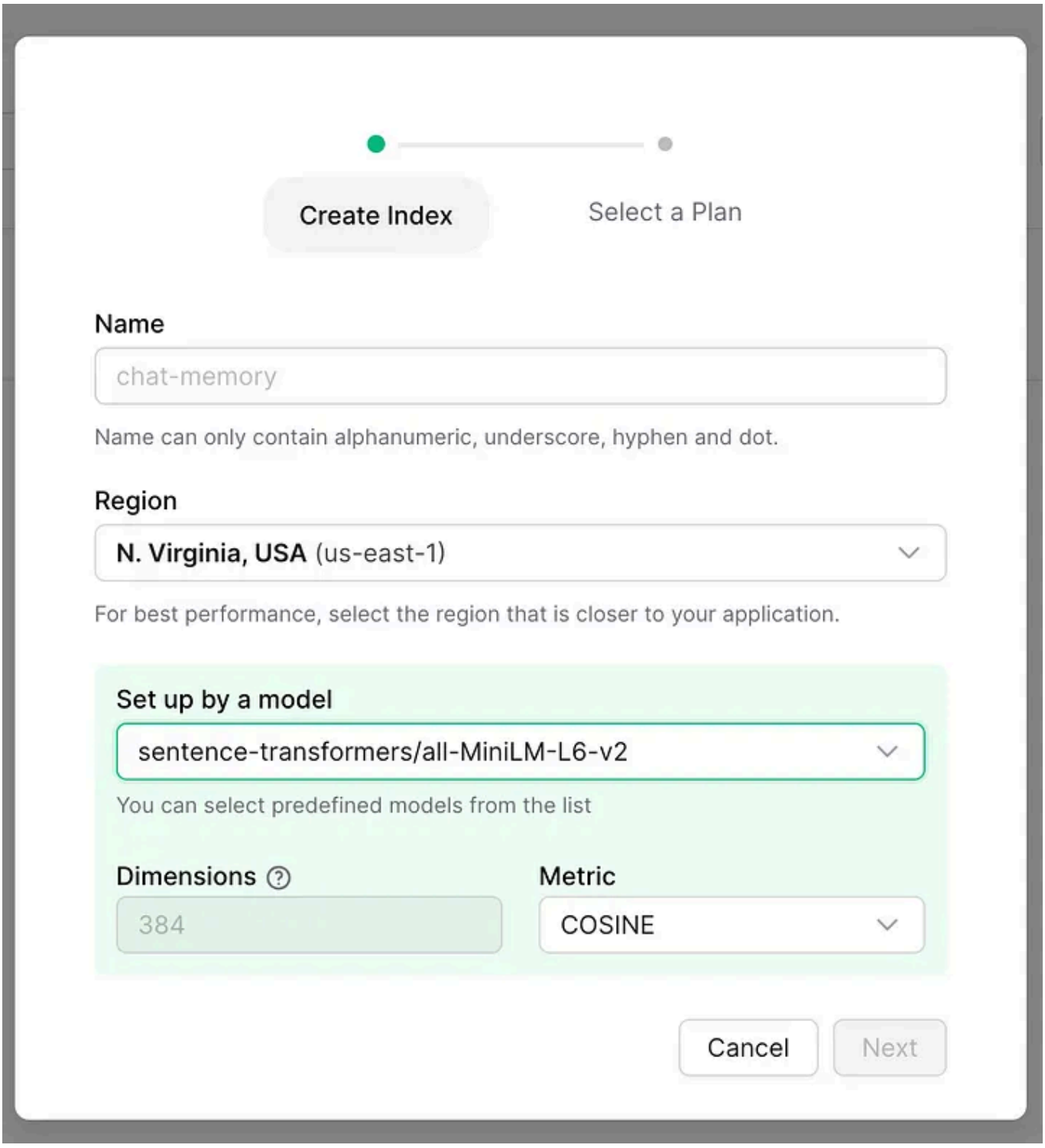
We've successfully created a Kafka Cluster, now we only need to copy and set the env variables that will help us connect to our cluster. To do that, go to your cluster dashboard, at the **Details** tab copy the *endpoint, username,* and *password*, and paste them into your `.env` file.
After that, go to **Topics** and copy your Kafka topic name.

This is how your `.env` file should look up until now:

```
UPSTASH_KAFKA_UNAME="[USERNAME HERE]"
UPSTASH_KAFKA_PASS="[PASSWORD HERE]"
UPSTASH_KAFKA_ENDPOINT="[ENDPOINT HERE]"
UPSTASH_KAFKA_TOPIC="[TOPIC NAME HERE]"
```

Creating a new Upstash Vector Index

Now, let's go ahead and create a new Vector Database. For that, select **Vector** from the main dashboard and then + ***Create Index***, you'll be prompted to this view:



Assign a name to your vector database, then select a region that is closest to your location and under `Set up by a model` select `sentence-transformers/all-MiniLM-L6-v2` as that's the model we're going to use when generating embedding for our news articles and the `cosine` metric for vector-distance comparison.

> *You could also select another model, but in case the model used when embedding articles differs in dimensions compared to the Upstash's created vector — the dims will have to be padded and that's not recommended.*

After you've created a new Vector Index, you can follow the same workflow as for the Kafka Cluster. Copy the *Index Name, Endpoint,* and *Token* and paste them into our `.env` file.

This is how your `.env` file should look up until now:

```
UPSTASH_KAFKA_UNAME="[USERNAME HERE]"
UPSTASH_KAFKA_PASS="[PASSWORD HERE]"
UPSTASH_KAFKA_ENDPOINT="[ENDPOINT HERE]"
UPSTASH_KAFKA_TOPIC="[TOPIC NAME HERE]"

UPSTASH_VECTOR_ENDPOINT="[VECTOR ENDPOINT HERE]"
UPSTASH_VECTOR_TOPIC="[VECTOR NAME HERE]"
UPSTASH_VECTOR_KEY="[VECTOR TOKEN HERE]"
```

Registering to News APIs

We will be using the following APIs from which to fetch articles:

1. 🔗 NewsAPI

Provides a free **developer** plan where we can call their API **100** times a day.

2. 🔗 NewsData

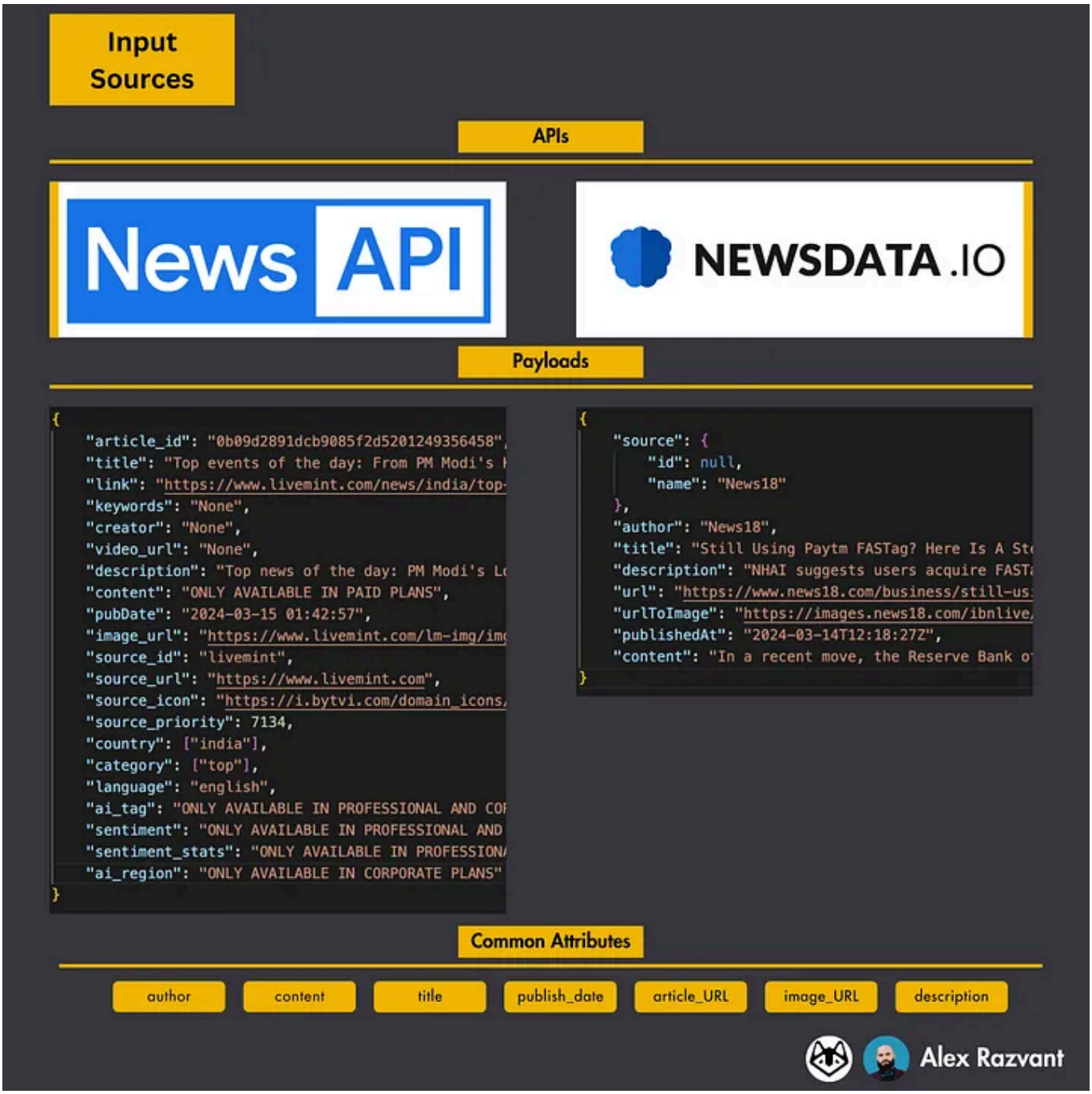
Provides a **free** plan where we get 200 credits/day, and each credit equals 10 articles, this means that we can fetch a total of **2000** articles per day.

For our current use case, these APIs provide enough functionality to implement and validate the news search engine we're building while leaving room for improvements and scaling since the underlying workflow will be the same.
The only constraint that bounds to the free plan, is that we can't perform a timed-batch fetch, meaning we cannot use `from_date, to_date` when querying these APIs, but that's not a problem.
We'll mimic this behavior by using a `wait time` between fetch calls.

The next step is to register on both platforms — don't worry, it's as straightforward as possible.

1. After you've registered to **NewsAPI**, head over to **/account** and you'll see a `API_KEY` field, copy and paste it into our `.env` at `NEWSAPI_KEY`.
2. After you've registered to **NewsData**, head over to **/api-key**, copy the `API KEY` and paste it into our `.env` file at `NEWSDATAIO_KEY`.

The **boring part is done**, we now have access to these APIs and can fetch articles. Here's how a payload looks like from both APIs:



Payload examples from both APIs. (Image by author)

Prerequisites Recap

After all these 3 steps, creating a Kafka Cluster, creating a Vector Index, and registering to the News APIs, our `.env` file should look like this:

```
UPSTASH_KAFKA_UNAME="[USERNAME HERE]"
UPSTASH_KAFKA_PASS="[PASSWORD HERE]"
UPSTASH_KAFKA_ENDPOINT="[ENDPOINT HERE]"
```

```
UPSTASH_KAFKA_TOPIC="[TOPIC NAME HERE]"

UPSTASH_VECTOR_ENDPOINT="[VECTOR ENDPOINT HERE]"
UPSTASH_VECTOR_TOPIC="[VECTOR NAME HERE]"
UPSTASH_VECTOR_KEY="[VECTOR TOKEN HERE]"

NEWSAPI_KEY="[NEWSAPI KEY HERE]"
NEWSDATAIO_KEY="[NEWSDATA KEY HERE]"
NEWS_TOPIC = "news" # this is the category of articles we're going to fetch
```

The next step, before diving into the implementation details is to install the environment and the required packages.
Here's how the Makefile install step looks like:

```
# Makefile
...
install:
 @echo "$(GREEN) [CONDA] Creating [$(ENV_NAME)] python env $(RESET)"
 conda create --name $(ENV_NAME) python=3.9 -y
 @echo "Activating the environment..."
 @bash -c "source $$(conda info --base)/etc/profile.d/conda.sh && conda activate
   && pip install poetry \
   poetry env use $(which python)"
 @echo "Installing Packages"
 @echo "Changing to pyproject.toml location..."
 @bash -c " PYTHON_KEYRING_BACKEND=keyring.backends.fail.Keyring poetry install"
...
```

To prepare the environment, run `make install .`

Next, let's investigate the handler implementation that fetches articles from these sources.

Data Gathering

The purpose of this module is to encapsulate the functionality of querying both APIs, parse the payloads, format them to a common document format using attributes present in both payloads, and use a shared KafkaProducer instance to send messages to our cluster.

In detail, we'll cover the following sub-modules:

- Articles Fetching Manager Class
- How to send messages to our Kafka Cluster
- Pydantic Data Models
- Running the pipeline

Articles Fetching Manager Class

Let's start by diving into the implementation:

```
import datetime
import functools
import logging
from typing import Callable, Dict, List

from newsapi import NewsApiClient
from newsdataapi import NewsDataApiClient
from pydantic import ValidationError

from models import NewsAPIModel, NewsDataIOModel
from settings import settings

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)


def handle_article_fetching(func: Callable) -> Callable:
    """
    Decorator to handle exceptions for article fetching functions.
```

```
        This decorator wraps article fetching functions to catch and log any excepti
        that occur during the fetching process. If an error occurs, it logs the erro
```


```
        Returns:
            Callable: The wrapped function.
        """

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            try:
                return func(*args, **kwargs)
            except ValidationError as e:
                logger.error(f"Validation error while processing articles: {e}")
            except Exception as e:
                logger.error(f"Error fetching data from source: {e}")
                logger.exception(e)
            return []

        return wrapper

    class NewsFetcher:
        """
        A class for fetching news articles from various APIs.

        Attributes:
            _newsapi (NewsApiClient): Client for the NewsAPI.
            _newsdataapi (NewsDataApiClient): Client for the NewsDataAPI.

        Methods:
            fetch_from_newsapi(): Fetches articles from NewsAPI.
            fetch_from_newsdataapi(): Fetches articles from NewsDataAPI.
            sources: Returns a list of callable fetch functions.
        """

        def __init__(self):
            self._newsapi = NewsApiClient(api_key=settings.NEWSAPI_KEY)
            self._newsdataapi = NewsDataApiClient(apikey=settings.NEWSDATAIO_KEY)

        @handle_article_fetching
        def fetch_from_newsapi(self) -> List[Dict]:
            """Fetch top headlines from NewsAPI."""
            response = self._newsapi.get_everything(
                q=settings.NEWS_TOPIC,
                language="en",
                page=settings.ARTICLES_BATCH_SIZE,
                page_size=settings.ARTICLES_BATCH_SIZE,
```

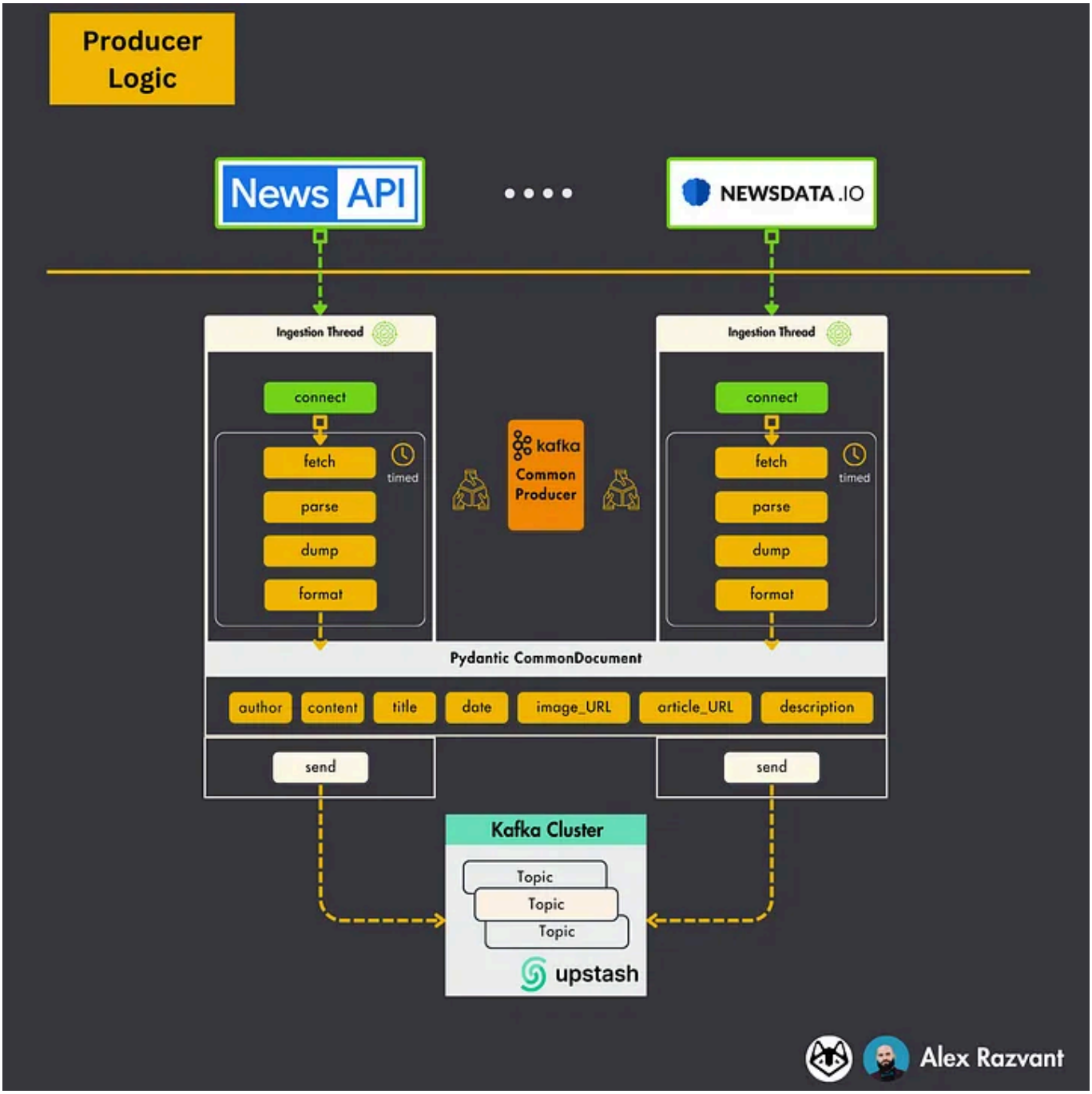
```
            )
            return [
                NewsAPIModel(**article).to_common()
                for article in response.get("articles", [])
            ]

        @handle_article_fetching
        def fetch_from_newsdataapi(self) -> List[Dict]:
            """Fetch news data from NewsDataAPI."""
            response = self._newsdataapi.news_api(
                q=settings.NEWS_TOPIC,
                language="en",
                size=settings.ARTICLES_BATCH_SIZE,
            )
            return [
                NewsDataIOModel(**article).to_common()
                for article in response.get("results", [])
            ]

        @property
        def sources(self) -> List[callable]:
            """List of news fetching functions."""
            return [self.fetch_from_newsapi, self.fetch_from_newsdataapi]
```

Here are a few **key considerations** in this implementation:

- `NewsAPIModel` and `NewsDataIOModel` are Pydantic models accustomed to specific payload formats.
- We're using the `handle_article_fetching` decorator to catch validation errors or broader exceptions when dumping raw payloads into Pydantic models.
- We have a property called `sources` that returns the callable methods for querying the APIs. This will be used within the data gathering module which spawns multiple producer threads to send messages to our Kafka Cluster. **We'll get into that next.**

Producing Kafka Messages

Here's the workflow we're going to implement:



Producing Kafka Messages (Image by author)

Key points from here:

- *We use separate threads to fetch from the APIs*
- *We share a common Kafka Producer instance to send the messages*

- *We use Pydantic Models to ensure data exchange.*

Using separate threads to fetch articles, and a single Kafka Producer instance to send messages to the cluster is a recommended approach in our use case. Here's why:

- **Efficiency and Performance:** KafkaProducer is thread-safe. A new instance involves creating a network connection, and some configuration. Sharing one single instance across threads reduces the overhead associated with these operations.
- **Throughput:** A single producer instance batches messages together before sending them to the Kafka cluster.
- **Resources:** Although not fully applicable to our use case, since we have only 2 producer threads, limiting the number of instances optimizes the system resource utilization.

Here's the main functionality that handles the messaging to **Kafka** [2]:

```
def run(self) -> NoReturn:
        """Continuously fetch and send messages to a Kafka topic."""
        while self.running.is_set():
            try:
                messages: List[CommonDocument] = self.fetch_function()
                if messages:
                    messages = [msg.to_kafka_payload() for msg in messages]
                    self.producer.send(self.topic, value=messages)
                    self.producer.flush()
                logger.info(
                    f"Producer : {self.producer_id} sent: {len(messages)} msgs."
                )
                time.sleep(self.wait_window_sec)
            except Exception as e:
```

```
                logger.error(f"Error in producer worker {self.producer_id}: {e}"
                self.running.clear()  # Stop the thread on error
```

Key considerations from the implementation:

- We're spawning as many `KafkaProducerThread` instances as there are fetch sources.
- We wrap all these threads under `KafkaProducerSwarm` .
- We share a single `KafkaProducer` instance across all threads, which will communicate with our cluster.
- We're following a **Singleton Design Pattern** [5] as we can scale to N fetching threads but still keep a single KafkaProducer instance.

Data exchange models with Pydantic

From the code snippet implementations presented above, you might have observed the use of `*Document`, `*Model` objects that were not explained before. Let's dive into what they are within this section.

These are Pydantic models for data exchange and within the application we're building, these models are:

- **NewsDataIOModel** : wraps and formats a raw payload from the NewsData API.
- **NewsAPIModel:** wraps and formats a raw payload from the NewsAPI API.
- **CommonDocument**: establishes a common format between the different News formats mentioned above.

- **RefinedDocument:** filters the common format by grouping helpful fields under metadata and emphasizing key fields like `article description text` .
- **ChunkedDocument:** chunks the text and ensures lineage between chunk_id and document_id.
- **EmbeddedDocument:** embeds chunks ensuring lineage between chunk_id and document_id.

For example, this is what the CommonDocument model looks like since it represents the **bridge** between different news payload formats:

```
class CommonDocument(BaseModel):
    article_id: str = Field(default_factory=lambda: str(uuid4()))
    title: str = Field(default_factory=lambda: "N/A")
    url: str = Field(default_factory=lambda: "N/A")
    published_at: str = Field(
        default_factory=lambda: datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    )
    source_name: str = Field(default_factory=lambda: "Unknown")
    image_url: Optional[str] = Field(default_factory=lambda: None)
    author: Optional[str] = Field(default_factory=lambda: "Unknown")
    description: Optional[str] = Field(default_factory=lambda: None)
    content: Optional[str] = Field(default_factory=lambda: None)

    @field_validator("title", "description", "content")
    def clean_text_fields(cls, v):
        if v is None or v == "":
            return "N/A"
        return clean_full(v)

    @field_validator("url", "image_url")
    def clean_url_fields(cls, v):
        if v is None:
            return "N/A"
        v = remove_html_tags(v)
        v = normalize_whitespace(v)
        return v
```

```
    @field_validator("published_at")
    def clean_date_field(cls, v):
        try:
            parsed_date = parser.parse(v)
            return parsed_date.strftime("%Y-%m-%d %H:%M:%S")
        except (ValueError, TypeError):
            logger.error(f"Error parsing date: {v}, using current date instead."

    @classmethod
    def from_json(cls, data: dict) -> "CommonDocument":
        """Create a CommonDocument from a JSON object."""
        return cls(**data)

    def to_kafka_payload(self) -> dict:
        """Prepare the common representation for Kafka payload."""
        return self.model_dump(exclude_none=False)
```

Let's unpack it:

- It contains a series of common attributes for both news article formats.
- Validates each field cleans or assigns a default value using the `field_validator` decorator.
- The `to_kafka_payload` method ensures message serialization before sending it to the Kafka Cluster.

Text field cleaning process

The cleaning process is a simple one, we're using methods to clean text and ensure we:

- Remove trailing spaces or \n, \t.
- Remove ul/li bullet points
- Remove HTML tags if present within the text.

We're using the **unstructured** [7] Python library to streamline these transformations.

> 💡 *Check more unstructured examples here: cleaning examples.*

Running KafkaProducers

Up until this point, we've done/implemented the following modules:

- Registered to all the required services
- Created Kafka Cluster and Vector Database
- Implemented the NewsArticle fetch handler.
- Implemented the Pydantic models for data exchange.
- Implemented the KafkaProducer logic.

With that being done, we can now safely run the `produce` stage of our pipeline and check for messages in our KafkaCluster on Upstash.

Let's do just that!

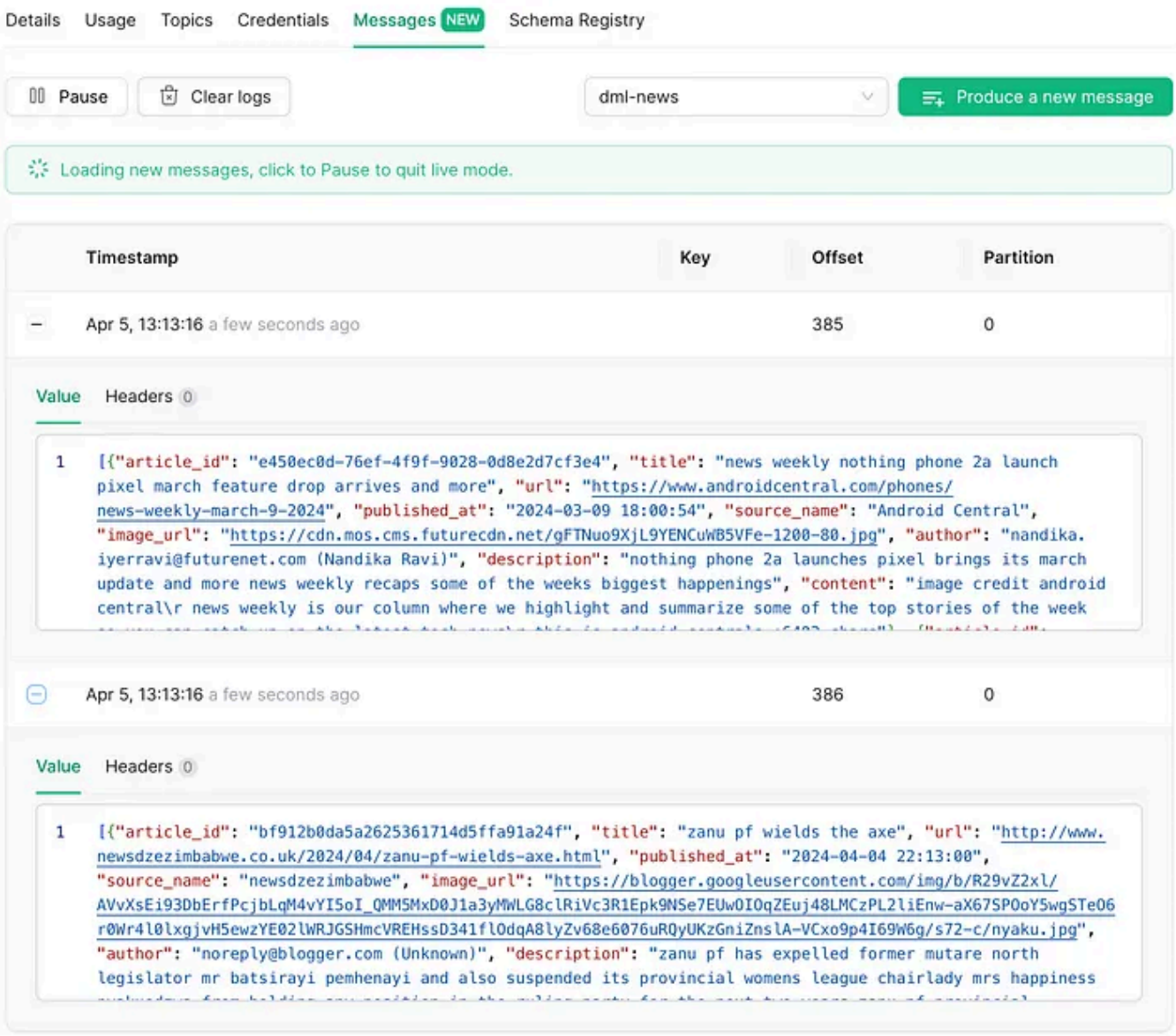
At the root of our project, in the `Makefile` we have a command that runs the **data gathering**:

```
....

run_producers:
 @echo "$(GREEN) [RUNNING] Data Gathering Pipeline Kafka Producers $(RESET)"
 @bash -c "poetry run python -m src.producer"

...
```

This 🔗 ***Makefile*** contains useful commands to interact with the solution we're building, in this use case we would need to execute the `run_producers` using `make run_producers .`
This will start the `KafkaSwarm` and handle threads that fetch articles from the NewsAPIs, format them, and push them to our Kafka Cluster.

CLI logs output after executing the command. (Image by author)

From the logs, we've seen that both Producer threads have sent **5msgs** each. To check if our messages have reached the cluster, go to *Upstash Console→ KafkaCluster → Messages*. You should see a view like this:



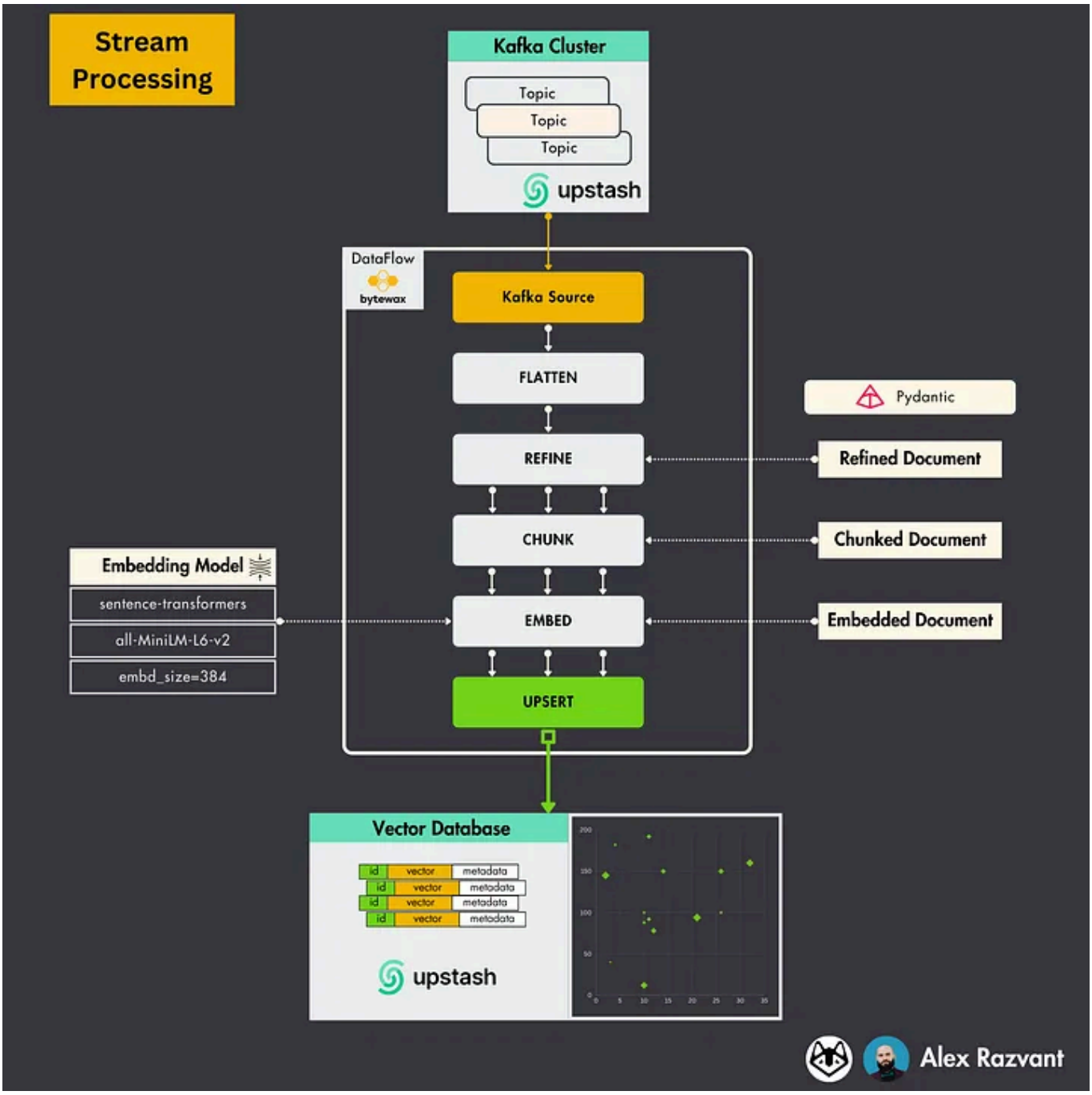
Upstash Kafka Messages (Image by author)

At this point, we're done with implementing and testing our Data Gathering Pipeline, where we fetch news articles from APIs, format them, and send messages to Kafka.
Next, we'll implement the "consumer" or Ingestion Pipeline to process the new messages from Kafka.

The Ingestion Pipeline

After we've validated that we've got messages on our Kafka Topic, we have to implement the "**consumer**" pipeline, meaning we'll:

1. Read messages from our Kafka Topic
2. Parse, format, chunk, and generate embeddings
3. Generate Vector objects and upsert them to **Upstash Vector Index** [3].



For this, we're going to use **Bytewax** [4] to define a DataFlow that chains these steps in the correct order.

Let’s dive straight into the implementation and explain key concepts!

1. **Defining the Kafka Source as input for our Bytewax Flow.**

```
import json
from typing import List

from bytewax.connectors.kafka import KafkaSinkMessage, KafkaSource

from logger import get_logger
from models import CommonDocument
from settings import settings

logger = get_logger(__name__)


def build_kafka_stream_client():
    """
    Build a Kafka stream client to read messages from the Upstash Kafka topic us
    """
    kafka_config = {
        "bootstrap.servers": settings.UPSTASH_KAFKA_ENDPOINT,
        "security.protocol": "SASL_SSL",
        "sasl.mechanisms": "SCRAM-SHA-256",
        "sasl.username": settings.UPSTASH_KAFKA_UNAME,
        "sasl.password": settings.UPSTASH_KAFKA_PASS,
        "auto.offset.reset": "earliest",  # Start reading at the earliest messag
    }
    kafka_input = KafkaSource(
        topics=[settings.UPSTASH_KAFKA_TOPIC],
        brokers=[settings.UPSTASH_KAFKA_ENDPOINT],
        add_config=kafka_config,
    )
    logger.info("KafkaSource client created successfully.")
    return kafka_input


def process_message(message: KafkaSinkMessage):
    """
    On a Kafka message, process the message and return a list of CommonDocuments
    - message: KafkaSinkMessage(key, value) where value is the message payload.
    """
    documents: List[CommonDocument] = []
    try:
        json_str = message.value.decode("utf-8")
```

```
            data = json.loads(json_str)
            documents = [CommonDocument.from_json(obj) for obj in data]
            logger.info(f"Decoded into {len(documents)} CommonDocuments")
            return documents
        except StopIteration:
            logger.info("No more documents to fetch from the client.")
        except KeyError as e:
            logger.error(f"Key error in processing document batch: {e}")
        except json.JSONDecodeError as e:
            logger.error(f"Error decoding JSON from message: {e}")
            raise
        except Exception as e:
            logger.exception(f"Unexpected error in next_batch: {e}")
```

Key points from this implementation:

- `build_kafka_stream_client` : creates an instance of a KafkaConsumer using the predefined Bytewax KafkaSource connector.
- `process_message` : a callback that will process the message from our Kafka Topic.

2. Defining the Upstash Vector (Index) as the output of our Bytewax flow.

```
from typing import Optional, List

from bytewax.outputs import DynamicSink, StatelessSinkPartition
from upstash_vector import Index, Vector
from models import EmbeddedDocument
from settings import settings
from logger import get_logger


logger = get_logger(__name__)


class UpstashVectorOutput(DynamicSink):
    """A class representing a Upstash vector output.
```

```
    This class is used to create a Upstash vector output, which is a type of dyn
    at-least-once processing. Messages from the resume epoch will be duplicated

    Args:
        vector_size (int): The size of the vector.
        collection_name (str, optional): The name of the collection.
            Defaults to constants.VECTOR_DB_OUTPUT_COLLECTION_NAME.
        client (Optional[UpstashClient], optional): The Upstash client. Defaults
    """

    def __init__(
        self,
        vector_size: int = settings.EMBEDDING_MODEL_MAX_INPUT_LENGTH,
        collection_name: str = settings.UPSTASH_VECTOR_TOPIC,
        client: Optional[Index] = None,
    ):
        self._collection_name = collection_name
        self._vector_size = vector_size

        if client:
            self.client = client
        else:
            self.client = Index(
                url=settings.UPSTASH_VECTOR_ENDPOINT,
                token=settings.UPSTASH_VECTOR_KEY,
                retries=settings.UPSTASH_VECTOR_RETRIES,
                retry_interval=settings.UPSTASH_VECTOR_WAIT_INTERVAL,
            )

    def build(
        self, step_id: str, worker_index: int, worker_count: int
    ) -> StatelessSinkPartition:
        return UpstashVectorSink(self.client, self._collection_name)


class UpstashVectorSink(StatelessSinkPartition):
    """
    A sink that writes document embeddings to an Upstash Vector database collect
    This implementation enhances error handling and logging, utilizes batch upse
    and follows Pythonic best practices for readability and maintainability.

    Args:
        client (Index): The Upstash Vector client to use for writing.
        collection_name (str, optional): The name of the collection to write to.
            Defaults to the value of the UPSTASH_VECTOR_TOPIC environment variab
    """

    def __init__(
        self,
```

```
        client: Index,
        collection_name: str = None,
    ):
        self._client = client
        self._collection_name = collection_name
        self._upsert_batch_size = settings.UPSTASH_VECTOR_UPSERT_BATCH_SIZE

    def write_batch(self, documents: List[EmbeddedDocument]):
        """
        Writes a batch of document embeddings to the configured Upstash Vector d

        Args:
            documents (List[EmbeddedDocument]): The documents to write.
        """
        vectors = [
            Vector(id=doc.doc_id, vector=doc.embeddings, metadata=doc.metadata)
            for doc in documents
        ]

        # Batch upsert for efficiency
        for i in range(0, len(vectors), self._upsert_batch_size):
            batch_vectors = vectors[i : i + self._upsert_batch_size]
            try:
                self._client.upsert(vectors=batch_vectors)
            except Exception as e:
                logger.error(f"Caught an exception during batch upsert {e}")
```

Key points from this implementation:

- `UpstashVectorOutput` : instantiates the Bytewax DynamicSink abstraction designed to route data to various destinations. In our case, this will wrap over the Upstash Vector Index client connection.
- `UpstashVectorSink` : wraps over our DynamicSink and handles the functionality of upserting vectors to our VectorDatabase. This `StatelessSinkPartition` means the DynamicSink would not hold any state and whatever inputs to our Sink is handled according to the `write_batch` functionality implementation.

Building the rest of Bytewax Flow

Here's the full implementation of our DataFlow that fetches messages from Upstash Kafka Topic, cleans, refines, chunks, embeds, and upserts vectors to Upstash Vector Index.

```
"""
    This script defines the ByteWax dataflow implementation for the Upstash use
    The dataflow contains these steps:
        1. Input: Read data from a Kafka stream.
        2. Refine: Transform the input data into a common format.
        3. Chunkenize: Split the input data into smaller chunks.
        4. Embed: Generate embeddings for the input data.
        5. Output: Write the output data to the Upstash vector database.
"""

from pathlib import Path
from typing import Optional

import bytewax.operators as op
from vector import UpstashVectorOutput
from consumer import process_message, build_kafka_stream_client
from bytewax.connectors.kafka import KafkaSource
from bytewax.dataflow import Dataflow
from bytewax.outputs import DynamicSink
from embeddings import TextEmbedder
from models import ChunkedDocument, EmbeddedDocument, RefinedDocument
from logger import get_logger

logger = get_logger(__name__)


def build(
    model_cache_dir: Optional[Path] = None,
) -> Dataflow:
    """
    Build the ByteWax dataflow for the Upstash use case.
    Follows this dataflow:
        * 1. Tag: ['kafka_input']   = The input data is read from a KafkaSource
        * 2. Tag: ['map_kinp']      = Process message from KafkaSource to Common
            * 2.1 [Optional] Tag ['dbg_map_kinp'] = Debugging after ['map_kinp']
        * 3. Tag: ['refine']        = Convert the message to a refined document
            * 3.1 [Optional] Tag ['dbg_refine'] = Debugging after ['refine']
        * 4. Tag: ['chunkenize']    = Split the refined document into smaller ch
            * 4.1 [Optional] Tag ['dbg_chunkenize'] = Debugging after ['chunkeni
```

```
        * 5. Tag: ['embed']         = Generate embeddings for the chunks
            * 5.1 [Optional] Tag ['dbg_embed'] = Debugging after ['embed']
        * 6. Tag: ['output']        = Write the embeddings to the Upstash vector
    Note:
        Each Optional Tag is a debugging step that can be enabled for troublesho
    """
    model = TextEmbedder(cache_dir=model_cache_dir)

    dataflow = Dataflow(flow_id="news-to-upstash")
    stream = op.input(
        step_id="kafka_input",
        flow=dataflow,
        source=_build_input(),
    )
    stream = op.flat_map("map_kinp", stream, process_message)
    # _ = op.inspect("dbg_map_kinp", stream)
    stream = op.map("refine", stream, RefinedDocument.from_common)
    # _ = op.inspect("dbg_refine", stream)
    stream = op.flat_map(
        "chunkenize",
        stream,
        lambda refined_doc: ChunkedDocument.from_refined(refined_doc, model),
    )
    # _ = op.inspect("dbg_chunkenize", stream)
    stream = op.map(
        "embed",
        stream,
        lambda chunked_doc: EmbeddedDocument.from_chunked(chunked_doc, model),
    )
    # _ = op.inspect("dbg_embed", stream)
    stream = op.output("output", stream, _build_output())
    logger.info("Successfully created bytewax dataflow.")
    logger.info(
        "\tStages: Kafka Input -> Map -> Refine -> Chunkenize -> Embed -> Upsert
    )
    return dataflow


def _build_input() -> KafkaSource:
    return build_kafka_stream_client()


def _build_output() -> DynamicSink:
    return UpstashVectorOutput()
```

Key points from this implementation:

- A **TextEmbedder** instance which is a singleton wrapper over our embedding model `sentence-transformers/all-MiniLM-L6-v2` [6] which will be used to compute embeddings from the article's text.
- A **stream** variable is used to define and control the Bytewax DataFlow.
- Various debugging steps across different stages of the DataFlow using Bytewax's `op.inspect` operator.
- A **_build_input()** method that wraps the **KafkaSource** client, for simplicity.
- A **_build_output()** method that wraps the **UpstashVector** client, for simplicity.

In this workflow, we have defined the following stage names:

1. `kafka_input` : stage of fetching and converting Kafka messages to CommonDocument Pydantic format.
2. `map_kinp` : meaning Map Kafka Input, which applies a flat map over the received messages, yielding a ***List[CommonDocument]*** Pydantic objects.
3. `refine` : which iterates over ***List[CommonDocument]*** and creates RefinedDocument instances.
4. `chunkenize` : which iterates over ***List[RefinedDocument]*** and creates ChunkedDocument instances.
5. `embed` : which iterates over ***List[ChunkedDocuments]*** and creates EmbeddedDocument instances.

6. `output` : which iterates over ***List[EmbeddedDocument],*** creates Vector objects, and upserts them to our Upstash Vector Index.

Starting the pipelines

Up until this point, we have implemented the:

- **Data Gathering Pipeline**: which fetches, at a given interval raw payloads from NewsAPIs, formats them, and sends messages to our **Kafka** [2] Topic.
- **Ingestion Pipeline**: This is a Bytewax DataFlow that connects to our Kafka Topic and consumes messages, ending with upserting vectors to our **Vector Database** [3].

We can start both of these services using the pre-defined commands in our **Makefile,** found at the root of the project:

```
# To run the Data Gathering Pipeline for producing Kafka Messages
make run_producers

# To run the Ingestion Pipeline for consuming Kafka Messages and upserting Vecto
make run_pipeline
```

And... **done**!
We've successfully started our producer/consumer services.
The only module left is the UI to interact with our Vector Database and search for news articles.

User Interface

The UI is a basic **Streamlit** [8] application that has the following functionalities:

- *A text search bar*
- *A div section that populates cards with the fetched articles from our vector database.*

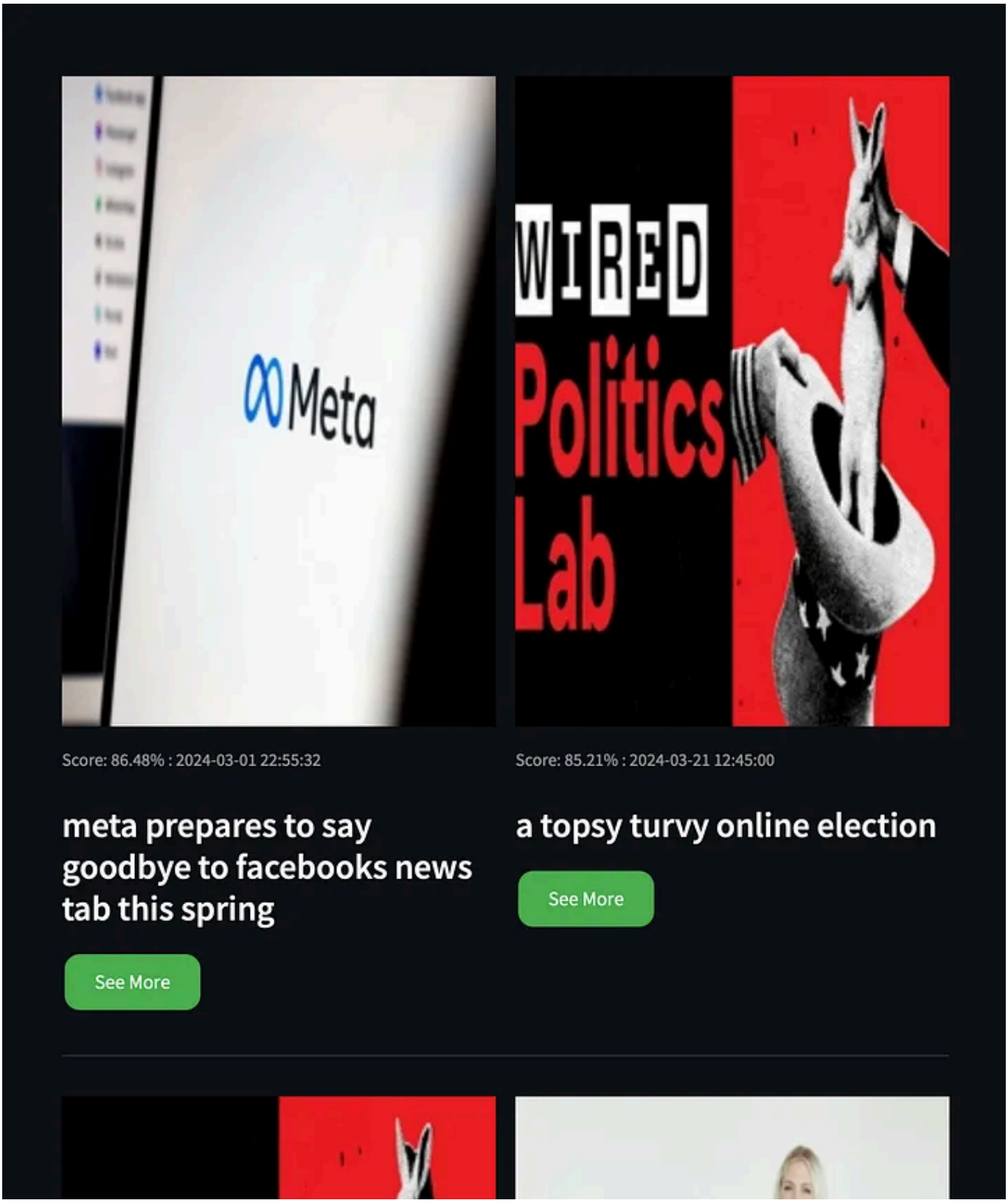
A card contains the following data fields:

- Date Published
- Similarity Score
- Article Image
- A **SeeMore** button that once clicked, sends you to the original article URL.

Once you've entered a message/question into the text bar — the input is cleaned (lowercase, remove non-ASCII, etc) and then embedded.
Using the new embeddings, we're querying the Vector Database to fetch the most similar entries — the outputs of which will be constructed and rendered.

Here's an example:

UI Text Input Field (Image by author)



UI Cards of fetched Articles. (Image by author)

Conclusion

Congratulations!

You've done it! You've built a News Search Engine that's not just a cool project but is ready to go live. We didn't just throw things together; we've also followed the best software development practices.

We used Pydantic to make sure our data played nice, wrote unit tests, took advantage of threading to speed things up, and brought in Upstash's serverless Kafka and Vector Database to not only easily set up our pipelines, but also make them fast, scalable and fail-safe.

You've now got the know-how to apply this blueprint to pretty much any data-driven idea you can think of. It's a big win, not just for this project, but for all the cool things you'll build in the future.

References

[1] News Search Engine using Upstash Vector — Decoding ML Github (2024)
[2] Upstash Serverless Kafka
[3] Upstash Serverless Vector Database
[4] Bytewax Stream Processing with Python
[5] Singleton Pattern
[6] sentence-transformers/all-MiniLM-L6-v2
[7] unstructured Python Library
[8] Streamlit Python

Further Reading

Sorted based on relevance to this article.

medium.com

> *In this article, you'll learn about the cool features of Pydantic and why I recommend using Pydantic Models whenever you have to structure class models for data exchange. The example showcased, is directly linked with the current article. You'll learn about the key attributes you must know, advantages and features as well as how to integrate best practices in your data engineering workflows.*

A Real-time Retrieval System for RAG on Social Media Data

Use a streaming engine to populate a vector DB in real-time. Improve RAG accuracy using rerank & UMAP.

medium.com

> *In this article, you'll learn how to build a Real-Time Retrieval system using Social Media Posts (e.g LinkedIn). It is tightly linked with the current article, providing an walkthrough vector re-ranking and visualization with UMAP.*

MI System Design | Large Language Models | Data Engineering

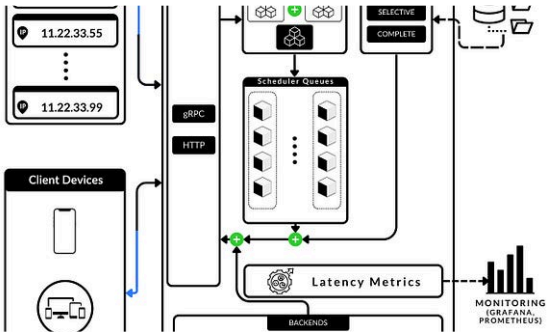
Written by Razvant Alexandru

Following

156 Followers · Editor for Decoding ML

Senior ML Engineer | Generative AI | MLOps. Leveraging AI systems to production! | 🔗
Join 4.5k+ engineers at Decoding ML https://decodingml.substack.com

More from Razvant Alexandru and Decoding ML



Razvant Alexandru in Decoding ML

Deploying Deep Learning Models at Scale — Triton Inference Server ...

Prepare and deploy an Image Classification model at scale using the Triton Inference...

✦ · 9 min read · Feb 7, 2024

430 3

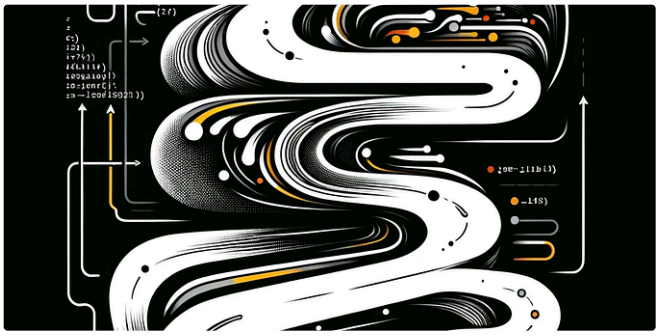
Paul Iusztin in Decoding ML

An End-to-End Framework for Production-Ready LLM Systems b...

From data gathering to productionizing LLMs using LLMOps good practices.

16 min read · Mar 16, 2024

1.5K 10



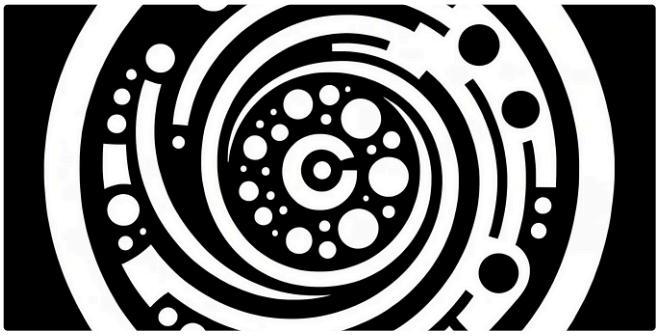
Paul Iusztin in Decoding ML

A Real-time Retrieval System for RAG on Social Media Data

Use a streaming engine to populate a vector DB in real-time. Improve RAG accuracy usin...

12 min read · Mar 30, 2024

318



Razvant Alexandru in Decoding ML

Why you need to pay attention to LLM Prompt Templates ?

On November 30 2022, ChatGPT was launched and everyone had a chance to test,...

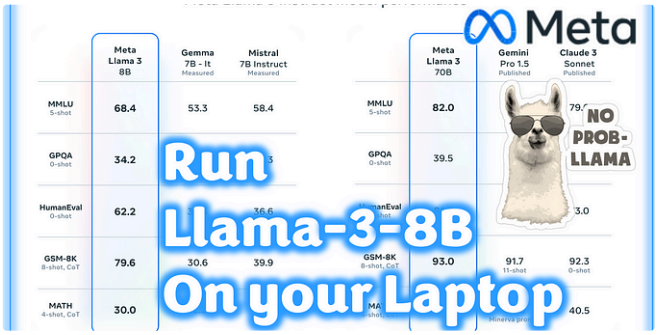
9 min read · Jan 12, 2024

202

See all from Razvant Alexandru

See all from Decoding ML

Recommended from Medium



Vatsal Saglani in Towards AI

Llama 3 + Groq is the AI Heaven

Llama 3 shines on Groq with blazing generation

✦ · 9 min read · Apr 21, 2024

862 6

Fabio Matricardi in Generative AI

Llama3 is out and you can run it on your Computer!

After only 1 day from the release, here is how you can run even on your Laptop with CPU...

✦ · 8 min read · Apr 19, 2024

1.6K 18

Lists

Natural Language Processing

1411 stories · 909 saves

ChatGPT prompts

47 stories · 1490 saves

AI Regulation

6 stories · 430 saves

Generative AI Recommended Reading

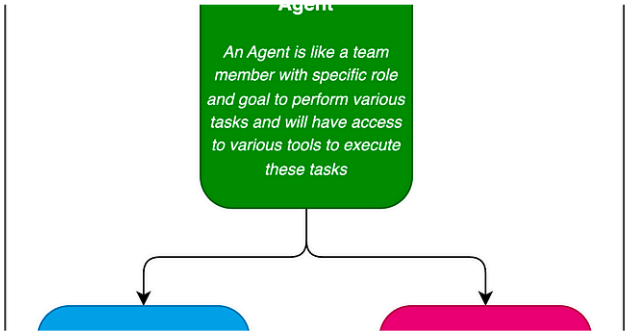
52 stories · 982 saves

Gavin Li in AI Advances

Run the strongest open-source LLM model: Llama3 70B with just ...

The strongest open source LLM model Llama3 has been released, Here is how you...

4 min read · Apr 21, 2024



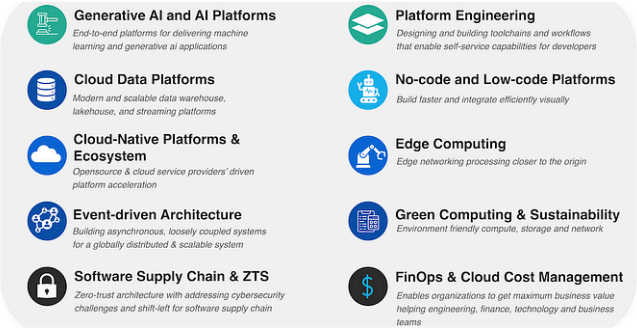
A B Vijay Kumar

Multi-Agent System — Crew.AI

Multi-Agent systems are LLM applications that are changing the automation landscape...

7 min read · Apr 17, 2024

1.2K 3



Ankur Kumar in Vedcraft

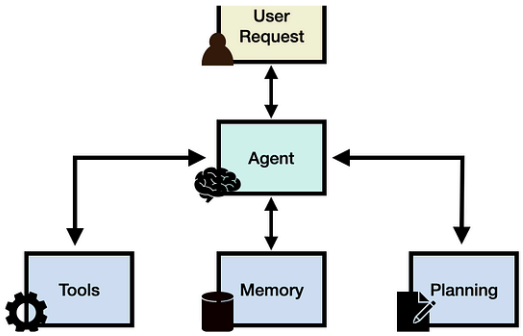
Top Ten Technology Trends for 2024

Observing technology trends by analysts, research companies, and thought leaders...

· 9 min read · Apr 14, 2024

244 8

218 5



Vishal Rajput in AIGuys

AI Agents Are All You Need

AI Agents, Understanding the role of Tools, Memory, and Planning in making them work.

· 14 min read · Apr 22, 2024

323 1

See more recommendations