



universität
wien

Bachelorarbeit

Efficient Text Embedding Inference in Apache Kafka

Verfasser

Raphael Lachtner

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2025

Studienkennzahl lt. Studienblatt: UA 033 521

Fachrichtung: Informatik - Allgemein

Betreuer: Dipl.-Ing. Dr.techn. Marian Lux

Abstract

Integrating text embedding models into data streaming platforms like Apache Kafka is important to support workflows such as retrieval-augmented generation. However, incorporating these computationally intensive text embedding models into Kafka pipelines brings performance, scalability, and resource utilization challenges. Existing solutions to this problem often rely on elaborate setups that lack standardized methods and are complex to integrate. This thesis addresses these challenges by investigating current approaches and identifying their limitations. Based on these findings, the design and implementation of the Java-based framework `kafka-text-embedding-inference` is presented. It uses Kafka's native Java APIs and an optimized inference solution to embed text at scale. The framework provides a modular architecture, that allows for the customization of chunking strategies, serialization formats, and integration with different embedding models. In addition, it uses a sophisticated batching approach that asynchronously embeds multiple messages at once. The proposed framework is evaluated against several alternatives, including a Python-based implementation using a co-located embedding model. The results show that the framework achieves higher throughput and resource efficiency than the other tested approaches. Compared to the widely used Kafka Streams client library, the framework benefits from optimized processing and achieves an approximately 33% higher throughput. This renders the `kafka-text-embedding-inference` framework an ideal solution for efficiently integrating text embedding inference into Apache Kafka pipelines.

Contents

Abstract	i
1 Introduction	1
1.1 Problem Statement	1
1.2 Gap	1
1.3 Thesis Objectives and Structure	1
2 Foundation	3
2.1 Related Work	3
2.1.1 Machine Learning Inference in Stream Processing Systems	3
2.1.2 Kafka Integration Patterns	3
2.1.3 Retrieval-Augmented Generation	3
2.2 Fundamentals of Apache Kafka	4
2.2.1 Message Delivery Semantics	4
2.2.2 Scaling and Performance Considerations	4
2.2.3 Interacting with Kafka	4
2.3 Text Embeddings	5
2.4 Retrieval-Augmented Generation Pipeline	5
2.4.1 General RAG overview	5
2.4.2 RAG Data Preparation Step in Kafka	6
2.5 Hugging Face Inference Service	7
2.6 Co-Located and External Service	8
3 Design and Idea	9
3.1 Test Implementations	9
3.1.1 Test Implementation using Python Client	9
3.1.2 Test Implementation using Kafka Streams	10
3.1.3 Test Implementation using Kafka Producer and Consumer API	10
4 Architecture and Implementation	11
4.1 kafka-text-embedding-inference: Overview	11
4.1.1 Introduction	11
4.1.2 High-Level Overview	11
4.1.3 Data Flow	11
4.2 System Design and Implementation	11
4.2.1 Core implementation	11
4.2.2 Models	13
4.2.3 Chunking	13
4.2.4 Serialization and Deserialization	13
4.2.5 Inference API	14
4.2.6 Configuration	14
4.2.7 Error Handling	15
4.2.8 Technical Implementation and Tools	15
4.3 Testing Strategy	16

4.4 Using and Extending the Framework	18
4.4.1 Building an Application	18
4.4.2 Demo Deployment and Configuration	19
4.4.3 Build and Containerisation Setup	20
4.4.4 Customise Processing Logic	21
5 Results and Evaluation	22
5.1 Evaluation Methodology	22
5.1.1 Performance Metrics	22
5.1.2 Test Environment	22
5.1.3 Software Specifications	22
5.1.4 Test Dataset Characteristics	23
5.1.5 Test Configuration	23
5.1.6 Embedding Model	23
5.2 Benchmarking Results	24
5.2.1 Approaches	24
5.2.2 Results	24
5.3 Discussion	26
5.3.1 Analysis of Results	26
5.3.2 Lessons Learned	26
5.3.3 Unexpected Findings	26
6 Conclusion and Future Work	27
6.1 Summary	27
6.2 Assumptions	27
6.3 Limitations	27
6.4 Future Work	27
6.4.1 Error Handling	27
6.4.2 Batching	28
6.4.3 Large-Language-Model Integration	28
6.4.4 Concurrent API Calls	28
6.4.5 Auto-Scaling and Scale to Zero	28
6.4.6 Quantization	28
Bibliography	29
7 Appendix	31

1 Introduction

Text embedding models have become important tools in modern software systems. They transform text into numerical vectors that encapsulate meaning and relationships between different pieces of text. These vectors are used in semantic searches, recommendation systems, and document classification. With the ever-increasing integration of machine learning (ML) models, specifically large language models (LLM), in all parts of software ecosystems, supporting them becomes increasingly important. Data-streaming solutions, like the most popular open-source stream-processing software Apache Kafka [1], are no exception. Such a streaming solution could be used to feed context to an LLM based on knowledge flowing through Kafka.

1.1 Problem Statement

Combining computationally intensive text embedding inference with Kafka pipelines introduces performance, scalability, and resource utilization challenges. Addressing these challenges to support such models is important to fully utilize the advantages that text embedding models provide within data streaming applications. In such scenarios, where data is continuously transferred and processed in (near-) real-time, it can be challenging to provide machine learning inference efficiently to generate embeddings economically. There are currently no standardized methods for such inference tasks in Kafka. A real-world example of an application discussed in this thesis is a data pipeline that processes research papers and creates text embedding vectors. It works by reading messages from Kafka and writing them back to Kafka after processing them. This aids in later finding and feeding relevant papers as context to a large language model in a process known as retrieval augmented generation (RAG).

1.2 Gap

Current approaches to providing text embedding inference in Apache Kafka typically rely on complex setups, which introduce latency and complexity. These approaches often lack standardized methods for handling machine learning inference within such pipelines. This results in inconsistent performance and scalability issues across implementations. Existing solutions do not adequately handle the economic and resource-efficient generation of embeddings. A software solution that operates in a Kafka streaming environment must provide support for a few important requirements. These include scaling capabilities, guaranteed message delivery semantics, robust error handling, and various serialization formats. Thus, a sophisticated Kafka implementation that reliably supports these essential features is required. By investigating and analyzing existing approaches for integrating streaming systems and model-serving strategies, the aim is to find reliable solutions that support these requirements.

1.3 Thesis Objectives and Structure

This thesis aims to explore current approaches to creating text embedding pipelines specifically tailored to RAG applications in Apache Kafka and to assess the requirements and features needed for those advanced data pipelines. The end goal is to create an efficient and practical software solution that handles text embedding inference in a streaming system while properly utilizing resources such as graphics processing units (GPU). In [Section 2](#) of this work, state-of-the-art pipeline technologies and text embedding models are explored. The aim is to find and review current ways of machine learning inference. In this process, the features and requirements of such a pipeline are investigated, and light is shed on the shortcomings of different approaches. Those results are

then used to design and implement some of those deemed reasonable approaches to gain insights and benchmark them in a real-world scenario. With these findings in mind, a software library is proposed in [Section 4](#) . First, the architecture and engineering are explained. Additionally, an example implementation is presented that illustrates the simplicity of constructing a robust data pipeline using the proposed solution. It explains how the pipeline can be managed and modified. After that, the focus is shifted to how testing is done for the library itself and how implementations can be tested. In [Section 5](#) , results will be evaluated. Finally, a summary of how the implemented solution differs from other discussed approaches is presented. The framework, including a demo application and further documentation, is publicly available on GitHub [2].

2 Foundation

2.1 Related Work

2.1.1 Machine Learning Inference in Stream Processing Systems

A recent work by Horchidan et al. introduces Crayfish [3], a benchmarking framework for the evaluation of machine learning inference in stream processing systems. In their work, they provide a study of different serving approaches and highlight their trade-offs. While they focus on general model serving, this work specifically addresses the challenges of text embedding generation in Kafka pipelines while focusing on retrieval augmented generation (RAG). Their findings about external and embedded serving informed some of the design decisions for this project. By combining machine learning models within streaming pipelines, data can be processed and enriched immediately as it flows through the system. This enables real-time analytics and decision-making in high-throughput environments. Research by Farki and Noughabi demonstrates the effectiveness of a pipeline like this [4]. In their work on real-time blood pressure prediction, they integrated ML models into a streaming framework using Apache Kafka and Apache Spark. This allowed them to infer large volumes of incoming data from wearable devices efficiently. Their pipeline collects data through Kafka, processes it using Spark, and then writes the results to a web-based platform. Their architecture uses Apache Spark as the primary processing engine, which provides powerful distributed computing capabilities but introduces additional complexity and resource overhead. The proposed framework instead focuses on lightweight, native Kafka processing.

2.1.2 Kafka Integration Patterns

An important work on integrating machine learning inference into data streaming platforms is the open-source framework Kafka-ML, as presented by Christian Martín et al. [5]. Kafka-ML aims to connect machine learning and artificial intelligence frameworks with Apache Kafka by providing an accessible and user-friendly web user interface that allows users to define, train, evaluate, and deploy ML models. In their work, they describe how they use containerization technologies and novel approaches to manage and reuse data streams. All this aims to ensure portability, easy distribution, fault tolerance, and high availability and potentially reduce the need for persistent data storage. However, Kafka-ML focuses on general ML model integration and does not specifically address the efficient integration of text embedding models within the stream processing system. Their framework uses a client written in Python in order to communicate with the Kafka broker. These clients run within the same process as the ML model. To run the models, their framework uses TensorFlow and PyTorch. This paper directly influenced this work by testing a similar approach that is specifically tailored to find the best way forward. The framework proposed in this work aims to simplify the creation of efficient text embedding pipelines in Kafka. It exclusively covers the inference with pre-trained models, allowing optimizations that might not be possible with general-purpose machine learning libraries.

2.1.3 Retrieval-Augmented Generation

An important contribution to RAG is presented by Wang et al. in their paper “Searching for Best Practice in Retrieval-Augmented Generation” [6]. The authors explore various components of RAG workflows to identify and implement optimal RAG practices. Their work aims to enhance large language models (LLM) by integrating retrieval mechanisms that access up-to-date and domain-specific knowledge. They address challenges like reducing hallucinations and improving response quality. Their work provides valuable insights into the optimization of RAG systems, such as the choice and workings of chunking strategies. In contrast, the focus of this

work is on integrating parts of the RAG workflow into a streaming system. This specialization allows to focus on the efficient generation of text embeddings in high throughput environments while benefiting from their findings on the improvement of RAG.

2.2 Fundamentals of Apache Kafka

Apache Kafka is a distributed event streaming platform that is designed to process large volumes of data in real time. It is the most popular open-source stream-processing software [1]. Common usages include high-performance data pipelines, streaming analytics, data integration, and mission-critical applications [1]. It operates on a publish-subscribe model basis that decouples applications from each other. **Producers** are applications that send data to Kafka, while **Consumers** read data from Kafka. They write to or read from a so-called **topic**, which can be seen as a log of events. Consumers can subscribe to and thus read from topics at any time. A topic is usually split into multiple, so-called **partitions** that enable data distribution and parallel processing.

As mentioned, Kafka pipelines are widely used across many industries, and as the usage of text embedding models and large language models grows, so does the need for robust inference solutions in Kafka. Machine learning inference involves using pre-trained models to make predictions about new data. The focus of this thesis is on building a reliable solution to streaming text embedding vectors that can later be used for retrieval augmented generation.

2.2.1 Message Delivery Semantics

Apache Kafka offers powerful semantic delivery guarantees. This allows for the building of sophisticated distributed and highly scalable data pipelines. Message delivery semantics can be categorized into three main types: **exactly-once** processing ensures strict consistency with no duplicates through idempotent producers and transactions, while **at-least-once** allows potential duplicates, and **at-most-once** accepts possible data loss but prevents duplicates.

2.2.2 Scaling and Performance Considerations

Kafka can be used as a highly scalable solution. When building inference into streaming processing systems, keeping up with other parts of the system and scaling pipelines based on current demand is necessary. This not only allows for swift message processing but also saves on processing resources by scaling the pipeline down when demand recedes. When it comes to developing sophisticated inference pipelines, challenges such as network latency between components, model loading times, and batching are important factors to consider. In [Section 2.3](#), these considerations are further explored in the context of text embedding models.

2.2.3 Interacting with Kafka

Interaction with Apache Kafka mainly occurs through producer and consumer clients. For programmatic approaches, various libraries and programming languages can be chosen. The option for client libraries primarily consists of the official Apache Kafka Java application programming interfaces (API), if using a Java virtual machine (JVM) language, or a library that is built around a C/C++ client library known under the name `librdkafka` [7]. The choice of Kafka client library affects system resilience through its handling of message ordering, delivery guarantees, and fault-tolerance mechanisms. Performance characteristics such as throughput and resource utilization vary between client implementations. This means that the programming language ecosystem and support for relevant Kafka protocols are essential factors [8].

Kafka Streams

Kafka Streams is part of the official set of Java APIs provided by Apache. It combines a consumer and producer approach and provides a high-level domain-specific language (DSL) for stream processing. It is generally a powerful solution but only supports record-by-record operations. It operates within the JVM ecosystem, which can add complexity when integrating with machine learning models.

Producer and Consumer API

The Producer and Consumer API are the core interfaces for interacting with Kafka. Kafka Streams, for example, is built on top of producer and consumer APIs [9]. Those libraries provide low-level access to Kafka, providing comprehensive configuration options and features like batch processing. They also run within a JVM environment, which might complicate certain machine learning tasks.

Python Client from Confluent

confluent-kafka-python is a wrapper library of `librdkafka`. It provides high-level producer and consumer access to Kafka. Conveniently, it also provides support for batch-processing messages from Kafka. As the name implies, it is based on Python, which makes it quite suitable to work with, as “Python continues to be the most preferred language for scientific computing, data science and machine learning” [10]. Python has a large machine learning ecosystem featuring many Python modules for text embedding, like sentence-transformers [11].

2.3 Text Embeddings

Text embeddings are numerical vector representations of text. They capture the context and relationships between different pieces of text. In their work about text embedding models, Merrick et al. describe that these models “represent queries like ‘How tall is Tom Cruise?’ and ‘Height of the actor who plays Maverick in Top Gun’ closely despite having no common words” [12]. This makes those models ideal in their use in a retrieval system, where finding similar documents is key. Embedding models are commonly used for semantic searches, recommendation systems, and document classification. The vector representations of the embedded text, for instance, enable the computation of similarity between vectors. This is very useful in the application of implementing part of a RAG system in Kafka, where text is read from Kafka and embeddings are produced back to Kafka. The serving of such models can benefit significantly from batching, impacting cost and performance [13].

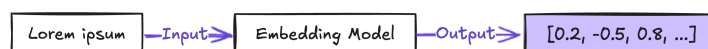


Figure 1: Text Embedding Transformation

2.4 Retrieval-Augmented Generation Pipeline

2.4.1 General RAG overview

A retrieval augmented generation (RAG) pipeline combines document retrieval with AI text generation to produce responses that are based on a knowledge base.

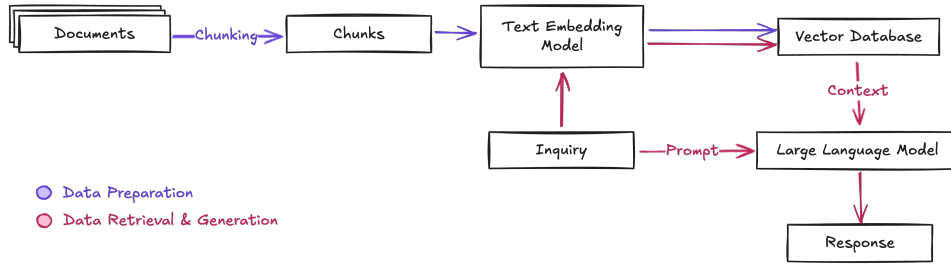


Figure 2: A simplified retrieval augmented generation pipeline

Figure 2 depicts a simple retrieval augmented generation pipeline. In the first step, data is processed by first chunking it. Then embeddings are generated from chunks and stored in a vector database along with additional relevant information about the document. Later on, the same embedding model is used to create a vector from an inquiry. This vector is then used to perform a similarity search inside the vector database. The returned context can be passed to a large language model that uses that information as a knowledge base to generate responses. In this work, we are especially interested in the depicted data preparation step, where incoming data is embedded. We want the Kafka pipeline to efficiently generate text embeddings. RAG can be used with documents so that the LLM can base its responses on them, reducing hallucinations and improving response accuracy in the process [6].

2.4.2 RAG Data Preparation Step in Kafka

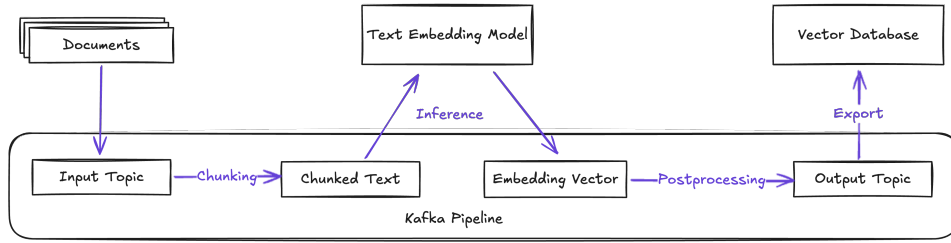


Figure 3: Part of a RAG pipeline in Apache Kafka

In Figure 3, we see the data preparation step applied to the workings of a Kafka pipeline. The input topic can be any Kafka topic containing data that needs to be embedded. The flow of the pipeline works by first chunking the input. In the next step, the Kafka data pipeline performs inference in order to retrieve embedding vectors. After receiving the embeddings, additional information from the source topic has to be included. This combined data is then written to an output topic, where it can eventually be exported to a vector database. Such an operation could be performed with a Kafka Sink Connector that automatically writes data from a topic to a downstream database. The most challenging part of this procedure is to send the chunked text to the text embedding model and retrieve the embedding vector.

2.5 Hugging Face Inference Service

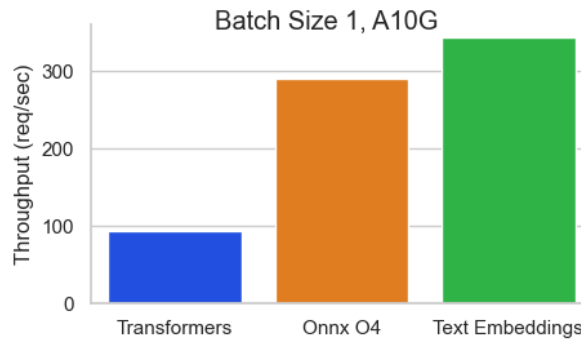


Figure 4: Comparison of inference throughput with batch size 1. Source: Hugging Face (2024), GitHub. [14]

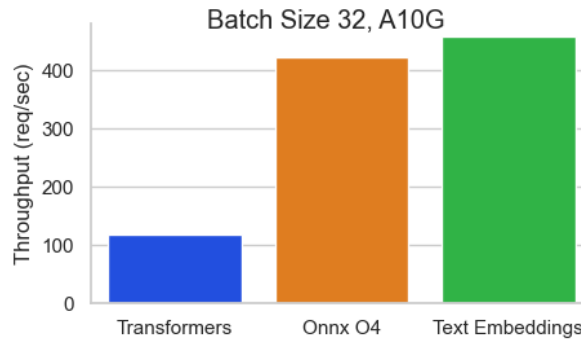


Figure 5: Comparison of inference throughput with batch size 32. Source: Hugging Face (2024), GitHub. [14]

Text-embeddings-inference (TEI) is an optimized inference service developed by Hugging Face that aims to be a “blazing fast inference solution for text embedding models” [14]. In essence, it is a piece of software that runs and serves text embedding models through an API. It supports many open source models, including BAAI/bge-large-en-v1.5, which is used in the evaluation of this work. The service is written in Rust and uses the Rust-based machine learning framework candle [14]. It utilizes various techniques to speed up inference, for instance, Flash Attention which can significantly improve model speed and memory usage [15]. Figure 4 and Figure 5 depict the throughput of the framework under different batch sizes. The inference service can be deployed as a common inference solution for not just the Kafka pipeline, but the entire machine learning infrastructure. This makes it easier to provide consistent embeddings across different applications while simplifying model management. It implements server-sided batching strategies, that allow the calling applications to send data to the API without the need to consider specific model batch size requirements. The service exposes an HTTP API in addition to a gRPC API. For this work, the focus is on the gRPC API, as it provides better performance compared to the REST API due to its usage of a binary protocol [16]. It accepts text inputs and returns their vector representations based on a configurable embedding model. The combination of the performance benefits, GPU acceleration support, and optimized attention mechanisms makes it an ideal candidate for the use-case of developing an efficient Kafka text embedding pipeline, where it is crucial to stream-process high-volume text embeddings. This approach, however, means that the model can’t be co-located in the Kafka streaming system, which causes reliance on network communication. Despite these drawbacks, calling external services is occasionally a necessary compromise when integrating such services [17].

2.6 Co-Located and External Service

There are typically two ways a machine learning model is served in a data streaming system. The first one is to co-locate and thus embed the model in the same process as the stream processing system. The second option is to externally run the model and then communicate with it over a network [18]. This choice was one of the most influential motivations to test a Python client approach.

- **Co-located service**

The streaming processing system and ML model run in the same process. A big advantage is reduced latency [19] because the inference is performed locally. Depending on the usage, this might speed up inference. A disadvantage of this approach is the potentially excessive use of resources on a processing node that might be shared with other streaming applications when deployed in a Kubernetes environment. If GPU acceleration is desired, each instance of the co-located application allocates a GPU node. The deployment and monitoring of the models can get complex if embedded. Model life cycle management can also become rather complicated when multiple models run in different applications and must be updated. Model loading time impacts pipeline flexibility. With GPU-accelerated co-location, containers must include all necessary drivers and wait for GPU provisioning during startup. Another important aspect is that an interoperability library has to be used if the model is not natively compatible with the stream processing system [3].

- **External service**

The model serving step is outsourced to an externally running inference service. This could be a commercially available service (e.g., OpenAI API) or running in the same environment as the stream processing system. Usually, such a service is accessed via a REST API, where input data is shared and eventually returned over the network. Network latency and availability become a concern when running the model externally, separated from the data pipeline. A big advantage of this approach is that model management is simpler, as there might only be the need for one inference service that provides inference with the same model for a wide range of applications. Integration in Kafka pipelines is simpler, but using such a service would introduce tight coupling between the application and the model server [19]. Both would also need to be scaled independently from each other since the ML inference part is decoupled from the Kafka pipeline. By using an external service, resource allocation can be optimized to serve the model without compromising the streaming application.

3 Design and Idea

Building upon previous chapters, the scope of this thesis includes implementing multiple test applications that are then benchmarked to find the most suitable approach for the Kafka text embedding pipeline. Each implementation in this section is described, and their differences are examined.

3.1 Test Implementations

To properly evaluate the direction in which the project should expand, multiple approaches, that were initially deemed reasonable were tested and compared. The results of these tests are discussed in [Section 5](#).

3.1.1 Test Implementation using Python Client

At first, we look into implementing solutions that utilize a Kafka Python client. This has the advantage of providing direct support for a large assortment of Python machine learning libraries that can be used to generate text embeddings. The application uses the Python library `confluent-kafka-python` developed by Confluent. A framework was developed using consumer and producer classes to read from and write to Kafka topics. With their `DeserializingConsumer` and `SerializingProducer` classes, the library provides a way to read and write serialized data from Kafka directly. However, the `DeserializingConsumer` does not support the consumption of more than one message at a time. To implement this behavior, a custom consumer approach was used by automatically reading multiple messages and deserializing them in the process. This allows the continuation of the batched inference approach. This work specifically focuses on JSON serialized data, which provides an uncomplicated way to exchange data in Kafka while providing support for proper schema evaluation. Thus, two JSON schema files were provided to the application. One is the input format that is expected to be read from Kafka. The other is the output format, which could be used in a downstream database. This includes all relevant information from the initial input and the inferred vector data. A naive chunking approach was used to stay consistent across implementations. The chunking stage of the pipeline splits input text based on a character limit. In light of these findings, two Python applications with similar internal structures were implemented and benchmarked. One uses `FastEmbed` for ML inference, while the other calls the TEI inference service API.

FastEmbed

`FastEmbed` is a “lightweight, fast, Python library built for embedding generation” [20] according to the authors. After researching and testing several suitable Python text embedding libraries, it became clear that `FastEmbed` was the best option. It fulfills all the requirements to use it in a co-located streaming application approach. It uses the ONNX runtime [20], making it perfectly suitable for a Kubernetes environment because it does not rely on PyTorch dependencies that inflate container size. Since GPU support is crucial for the use case, large dependencies like CUDA and cuDNN are required already. In containerized environments, the bigger container size increases start-up time. `FastEmbed` supports a range of models, including the model `BAAI/bge-large-en-v1.5` that was used in all of the benchmarks. As shown in [Figure 6](#), according to their own testing, `FastEmbed` performs at a higher number of characters per second rate than Hugging Face Transformers.

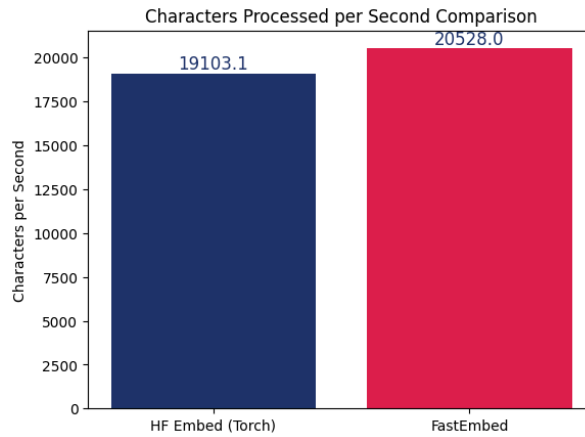


Figure 6: Embedding performance comparison. Source: Qdrant (2024), FastEmbed, GitHub. [20]

The FastEmbed library was used in the application by giving it a batched list of chunked input strings. As a result, it returns the desired list of vector data, which is eventually written back to Kafka. GPU acceleration with the appropriate FastEmbed GPU dependency was used, to represent a real-world scenario.

text-embeddings-inference

Another Python application works in a similar way, but it uses the TEI inference service, discussed in [Section 2.5](#), instead of FastEmbed. Rather than directly running the machine learning model in the same application, this approach calls a gRPC API, that returns the desired vector data in a similar fashion.

3.1.2 Test Implementation using Kafka Streams

Another implementation that was tested and seen as the baseline comparison, uses Kafka Streams. It is often used in Kafka pipelines due to its excellent integration with Kafka's ecosystem and its renowned high-throughput and scalability [21]. This approach, again, uses gRPC to call the Hugging Face text-embeddings-inference service to send and receive embedded vector data. The key difference is that Kafka Streams does not support batching. It merely supports record-by-record operations. It uses the same schemas as the previous Python applications, making it compatible with all other tested variations, and also uses the same naive character-based chunking approach.

3.1.3 Test Implementation using Kafka Producer and Consumer API

Another approach that was implemented and tested is the use of the official Kafka Producer and Consumer APIs. This approach is similar to the Kafka Streams approach, but implementation is more involved, as these APIs provide low-level access to consumers and producers. The implementation details are discussed in [Section 4](#).

4 Architecture and Implementation

4.1 kafka-text-embedding-inference: Overview

4.1.1 Introduction

Kafka-text-embedding-inference is a Java-based framework for building efficient text embedding pipelines in Kafka. The library uses Hugging Face's text-embedding-inference service and Kafka's native Java client APIs to provide a robust solution for inferring text embeddings at scale. The library takes over the entire data preparation step of a typical RAG pipeline, as depicted in [Figure 3](#). The implementation focuses on performance, reliability, and ease of use while providing flexibility for many different pipeline needs.

4.1.2 High-Level Overview

The framework is designed around a modular architecture that separates message processing, embedding generation, and general Kafka interaction. It utilizes Kafka's official Consumer and Producer APIs for low-level and reliable interaction with Kafka brokers, combined with a gRPC client for efficient communication with the TEI inference service. Configuration is handled through command-line interface (CLI) options. The library is flexible and can accommodate many different use cases by supporting the creation of data pipelines with customizable chunking strategies, message formats, and serialization.

4.1.3 Data Flow

The data flowing through the system follows a pipeline pattern, as depicted in [Figure 3](#):

- **Message Polling:** Records are consumed from a configured Kafka input topic using a configured batch size.
- **Processing Records:** Consumed messages are prepared for embedding, including chunking or text preprocessing.
- **Batch Embedding:** Prepared text chunks are batched and asynchronously sent to the inference service for optimal performance
- **Result Processing:** Generated embeddings are post-processed and combined with original messages
- **Message Production:** Enriched messages are produced to a configured output topic

4.2 System Design and Implementation

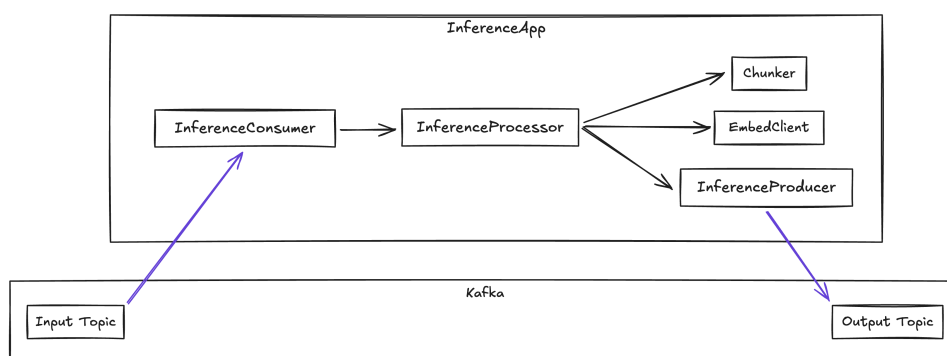


Figure 7: A high-level overview of the proposed framework

4.2.1 Core implementation

[Figure 7](#) depicts a high-level overview of the inner workings of the framework. `InferenceApp` is an abstract class that is supposed to be inherited in framework-based applications. The class provides powerful generics that

allow the specification of a key class, an input value class, and an output value class. They can be freely chosen to be any class that matches the requirements of the underlying data streaming infrastructure. For the library to know what part of the input object needs to be chunked, the `InputValue` generic has to extend `Chunkable`, an interface that provides a method returning a text to be chunked. Key and `OutputValue` can be freely chosen as long as their serialization configuration is specified correctly. Those configurations can be passed to the framework in the constructor of the `InferenceApp` class as a `SerializationConfig` class. This class is a simple Java Record that takes four objects of type `Class`: Deserialisers for keys and values and Serialisers for keys and values. This approach makes it possible to implement arbitrary serializers, e.g. Protocol Buffers or Avro.

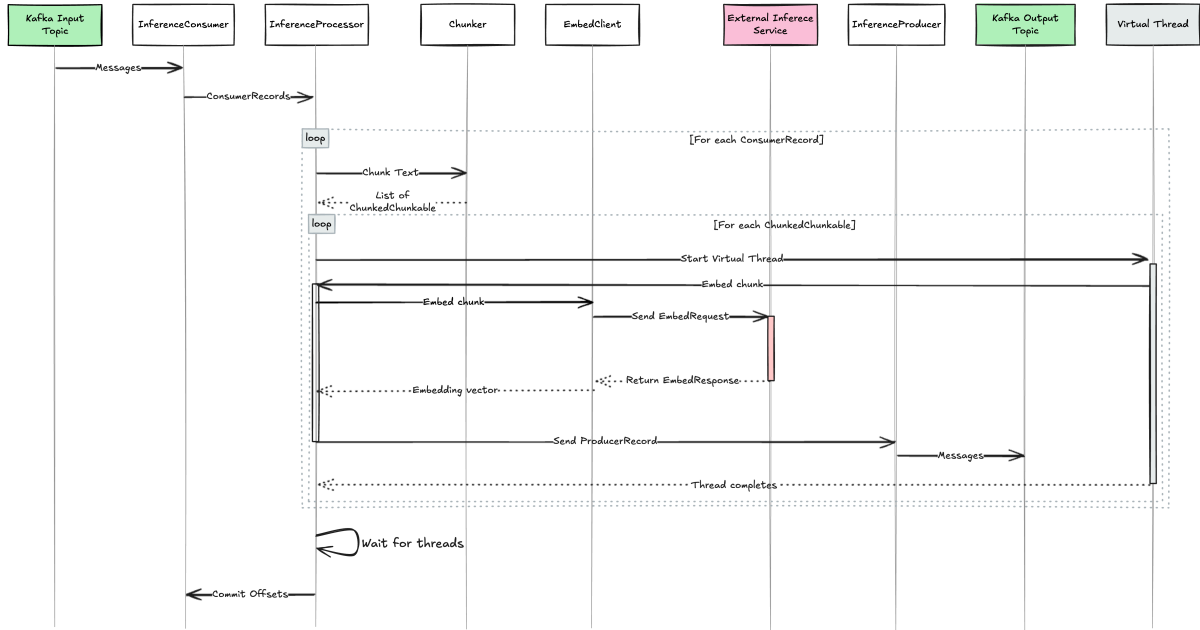


Figure 8: A high-level overview of the proposed framework

In [Figure 8](#) a sequence diagram showing the framework's basic operational sequence can be seen. It starts by receiving messages from the Kafka input topic. The `InferenceConsumer` class handles the consumption of messages from Kafka. The `InferenceProcessor` class is responsible for the core processing logic. This promotes cohesion and makes the code base more straightforward to understand and maintain. It then passes the received messages, as `Chunkable` records to the `Chunker` class, which splits the text and returns a list of text chunks, represented as `ChunkedChunkable`. To fully utilize the inference service, as discussed before, requests are sent to the service in batches. In essence, this works by spawning a new virtual thread for every chunk that needs to be embedded. The external inference service is called via a gRPC request in this thread. The resulting vector is then combined with the contents of the original message. This combined message, represented as a record called `EmbeddedChunkable`, is sent to the `InferenceProducer` class, which produces messages to Kafka. After the messages were successfully sent to the Kafka output topic, each individual thread completes its execution. By waiting for all threads to finish in the `InferenceProcessor` class, it is ensured, that all messages in the current batch are processed successfully. After that, offsets are sent back to the Kafka cluster using the `InferenceConsumer` class. These offsets are used to track the position of the last message processed by a consumer in Kafka. Subsequently, the next batch can be processed.

4.2.2 Models

- `Chunkable`

This functional interface is the base that all input values processed by the framework have to implement. It serves as a common object that provides text for chunking. The implementing application has to provide the abstract method, `getText()`, which returns a string used to chunk text later on.

- `ChunkedChunkable`

`ChunkedChunkable` is a simple Java record that holds a `Chunkable` object and a text chunk as a `String`. This object is yielded by the chunking stage that stores the text chunk with the original input object.

- `EmbeddedChunkable`

This Java record holds a `ChunkedChunkable` alongside an embedding vector. It is the result of the framework's embedding stage and is later used to create the output object.

4.2.3 Chunking

Chunking allows for the embedding of large documents by breaking down text into smaller segments, making it a crucial part of enhancing RAG precision [6]. This has the benefit of retrieving more precise information that aligns closely with the search query instead of returning the entire document. It also helps with token limits in the embedding model and, later on, the large language model. Depending on the chunking strategy, the size, overlap, and method play a significant role in the performance of the retrieval system. Chunks must be small enough to be specific but large enough to capture enough context and meaning. In the framework, chunking is left to the implementing application. A naive character-based chunk approach with maximum length and overlap was used for the test application. This was mainly chosen so that it becomes easier to compare different approaches on different platforms without having to consider things like tokenization. The framework provides a functional interface, `Chunker`. It should be implemented to create a list of `ChunkedChunkable` objects from an input `Chunkable`.

4.2.4 Serialization and Deserialization

Serializers and Deserializers are used in Kafka client applications to materialize data when necessary. They convert from Java objects to bytes and vice versa. In stream processing systems, various data formats are used, the most common ones being `String`, `Avro`, `Protocol Buffers`, and `JSON`, as well as primitive data types like `byte` or `int`. The framework allows specifying any combination of serializers and deserializers. This is done via a class called `SerializationConfig` that is passed to the constructor of the abstract `InferenceApp` class.

```
record SerializationConfig(Class keyDeserializerClass,  
                           Class valueDeserializerClass,  
                           Class keySerializerClass,  
                           Class valueSerializerClass)
```

Additionally, the framework expects the following abstract functions to be implemented in order to work with custom serialization formats properly:

```
public abstract Deserializer<Key> getKeyDeserializer();  
public abstract Deserializer<InputValue> getInputValueDeserializer();  
public abstract Serializer<Key> getKeySerializer();  
public abstract Serializer<OutputValue> getOutputValueSerializer();
```

`SerdeUtils`

The library provides a helper class called `SerdeUtils`. It provides static utility methods that can be used to simplify the creation of `Serializers`, `Deserializers`, and `Serdes`. They take a supplier and a map with configuration properties and return the fully configured object.

```
public static <T> Serde<T> getConfiguredSerde(  
    final Supplier<? extends Serde<T>> serdeSupplier,  
    final Map<String, Object> config)
```

Returns a configured `Serde` instance.

```
public static <T> Serializer<T> getConfiguredSerializer(  
    final Supplier<? extends Serializer<T>> supplier,  
    final Map<String, Object> config)
```

Returns a configured `Serializer` instance.

```
public static <T> Deserializer<T> getConfiguredDeserializer(  
    final Supplier<? extends Deserializer<T>> supplier,  
    final Map<String, Object> config)
```

Returns a configured `Deserializer` instance.

4.2.5 Inference API

As discussed in [Section 2.5](#), the framework uses a gRPC call to the Hugging Face text-embeddings-inference service. The library uses the `Embed` remote procedure call, provided by the TEI service, to generate embeddings. The framework sends an `EmbedRequest` that includes the text to be embedded. The TEI service then returns an `EmbedResponse` containing the embedding vector and metadata. This integration allows the framework to offload the intensive computations to the GPU-accelerated service, keeping the processing load on the stream processing application low.

4.2.6 Configuration

The framework uses `picocli` to provide command-line interface (CLI) options to configure the application. `Picocli` describes itself as it “aims to be the easiest way to create rich command line applications that can run on and off the JVM” [22]. They offer an explicit, flexible, and user-friendly way to pass arguments to the application. The proposed framework provides the following CLI options:

- `batch-size`

The value that is passed to the Kafka consumer to specify the maximum number of records returned in a single poll call [23].

- `bootstrap-server`

Represents the host and port of the Kafka broker, the application should connect and communicate with.

- `schema-registry`

This argument is optional. If specified, it connects the consumer and producer to a schema registry. This is used to manage and validate schemas for messages in Kafka topics. It is also used in serialization and deserialization.

- `input-topic`

Data for processing is read from the specified Kafka topic. The messages in this topic must conform to the specified format in the serialization configuration.

- `output-topic`

The output topic is used to write embedded messages. When implementing the framework, the output format must be specified as well.

- `tei-host`

This is the host of the text-embeddings-inference service that is used to generate embeddings.

- `tei-port`

Configures the port of the TEI service.

4.2.7 Error Handling

As is common in stream processing systems, processing should not be interrupted in case of an error. Instead, errors are caught early and are appropriately logged. The producer, for instance, has a callback that checks if a message was successfully produced. If that is not the case, it creates an error log and processes the following records. This approach to error handling enhances the system's resilience by preventing errors in individual messages from affecting the overall system's workflow.

4.2.8 Technical Implementation and Tools

Before developing the current framework, extensive testing was performed to identify important optimization factors. One of the most important ones was the choice of the inference model runtime and framework.

Batching Strategy

As seen in [Figure 5](#) by Hugging Face, batching can improve performance by up to 35 percent. The framework implements it by reading multiple records from Kafka simultaneously. Those records are then chunked, increasing the individual message batch size. After that, they are sent to the inference service at once.

Thread Management

To properly utilize the inference service, virtual threads are used to send the current batch of messages without having to wait for responses. The gRPC API of the TEI service does not provide the option to send multiple texts in one request. This was implemented by spawning a new virtual thread for each text in the current batch of chunked messages. Each thread is responsible for sending a request to the TEI service and waiting for a response. After all threads are spawned, the main processing handler waits for all of them to finish execution, ensuring the proper processing of all messages.

gRPC

When choosing a way to communicate with the TEI service, gRPC was used to gain an additional bit of performance benefit, especially when considering that embed requests are sent over a network, resulting in increased latency. While a standard REST API typically uses text-based formats like JSON, this approach uses the binary-based smaller and faster Protocol Buffers by default. gRPC is reported to reduce the size of messages sent over the network by up to 30 percent [24]. Additional research shows a 77.42 percent faster data transmission for small data sets when using gRPC compared to REST [25].

Gradle

Gradle is a build automation tool that was used in the development of the framework. It manages dependencies, compiles code, runs tests, and creates deployable artifacts. Gradle was used for dependency management in the project by utilizing a common version catalog for both the framework and the demo implementation in a multi-module setup. Another use-case is the container image creation, allowing, for instance, the deployment of the application in a Kubernetes cluster. Gradle also generates Protocol Buffer classes that were used to interact with the API of the TEI service based on a definition file provided by Hugging Face.

Jib

Jib is an open-source container image builder developed by Google. It is used to containerize the demo application so that it can run on Docker. Jib handles this containerization process directly in the build tool. It is configured in Gradle and fully integrated into the build life cycle.

Jackson

Jackson is a popular Java library for JSON processing. It is used in the example project to serialize and deserialize Kafka messages. It provides annotations and configuration options that make converting Java objects to and from the JSON format more straightforward. In the demo implementation, Jackson handles the conversion of Paper objects and their embedded counterparts to the JSON format.

Docker Compose

Docker Compose is used in the example project to orchestrate the complete demo environment. It provides a full production-like setup for testing and demonstrating the application. The compose configuration defines and manages multiple services that are part of the processing pipeline and configures how they interact with each other.

4.3 Testing Strategy

The framework is thoroughly tested to ensure reliability and catch potential bugs early in development. This includes writing tests for individual components, using mock objects to simulate some external dependencies, and verifying expected behavior with assertions.

JUnit Jupiter

JUnit provides the foundation for writing and executing unit tests in Java. Test classes and methods are annotated to indicate their role in the testing process. For instance, in the `InferenceConsumerTest` class, annotations are used to structure the tests.

```
class InferenceConsumerTest {

    @BeforeEach
    void setup() {
        // setup mock consumer and inference consumer
    }

    @Test
    void shouldPollRecord() {
        // test logic
    }
    // ...
}
```

Here, `@BeforeEach` sets up the necessary environment before each test, and `@Test` marks an individual test case.

AssertJ

AssertJ is used to write fluent and expressive assertions. This enhances test readability and maintainability.

In the `InferenceProducerTest` class, assertions are made using the fluent API of AssertJ:

```
assertThat(history).hasSize(1);
assertThat(record).isEqualTo(producerRecord);
```

These assertions state the expected conditions that have to be met in order for the test to succeed. In case of a failure it provides a clear overview of what went wrong.

Mockito

Mockito allows the creation of mock objects and defines their behavior. This allows for the components to be tested in isolation from their dependencies.

In `EmbedClientTest`, the gRPC stub is mocked to simulate a response from the external service.

```
@Mock
private EmbedGrpc.EmbedBlockingStub blockingStub = mock(EmbedGrpc.EmbedBlockingStub.class);

@Test
void testEmbed() {
    // define expected request and mock response
    when(blockingStub.embed(expectedRequest)).thenReturn(mockResponse);

    // call the method that is tested
    List<Float> result = embedClient.embed(chunkedChunkable);

    // verify the result and interaction with the mock
    assertThat(result).isEqualTo(mockEmbeddings);
    verify(blockingStub).embed(expectedRequest);
}
```

By mocking `blockingStub`, real network calls can be avoided. This allows the test to run quickly and reliably without having to setup and depend on another service.

MockConsumer and MockProducer

The framework utilizes Kafka's official `MockConsumer` and `MockProducer` classes to simulate interactions with a Kafka broker in the tests. `MockConsumer` allows to mimic consumer behavior by manually assigning partitions, setting offsets, and injecting records. Similarly, the `MockProducer` captures produced records and allows to verify that they are sent correctly.

Integration of Testing Tools

The combination of these tools creates a robust testing environment. In `InferenceAppTest`, this integration can be seen in action. Here is a simplified example:

```
@Test
void shouldProcessRecords() throws InterruptedException {
    // add records to the mock consumer
    mockConsumer.addRecord(record1);
    mockConsumer.addRecord(record2);

    // allow time for processing
    Thread.sleep(1000);

    // assert that the records were processed correctly
    assertThat(producedRecords)
        .extracting(record -> tuple(record.topic(), record.key(), record.value()))
        .containsExactlyInAnyOrder(
            tuple(OUTPUT_TOPIC, record1.key(), record1.value().chunk()),
            tuple(OUTPUT_TOPIC, record2.key(), record2.value().chunk())
        );
}
```

This test verifies that the application processes input records and produces the expected output, without relying on a real Kafka broker or inference service.

4.4 Using and Extending the Framework

This section showcases how the library can be used to build stream processing pipelines on the basis of the demo application `paper-inference-app`, which is available in the same GitHub repository as the framework [2].

4.4.1 Building an Application

This chapter discusses the details of how to implement a custom text embedding application using the provided framework.

```
public class PaperInferenceApp extends InferenceApp<String, Paper, EmbeddedPaper> {  
    // Implementation  
}
```

The first step is to extend the base `InferenceApp` class with specific types. The first generic type parameter is chosen to be a `String` and is used for both the input and output keys of messages in Kafka. The object `Paper`, which represents the input value, was chosen as the second type parameter. This object implements `Chunkable`. For the third parameter, which represents the output value, the `EmbeddedPaper` object was used.

Constructor

```
public PaperInferenceApp() {  
    super(new SerializationConfig(StringDeserializer.class, KafkaJsonSerializer.class,  
        StringSerializer.class,  
        KafkaJsonSerializer.class));  
}
```

Next, a `SerializationConfig` instance containing `Class` objects for Serializers and Deserializers is passed to the superclass constructor. The demo app uses string serialization for the message key and JSON serialization for the message value.

Implement Methods

```
@Override  
public Chunker createChunker() {  
    return new NaiveCharacterChunker(this.chunkSize, this.chunkOverlap);  
}
```

The implementation returns a new instance of the custom chunker, as will be discussed in detail later.

```
@Override  
public String getGroupId() {  
    return "paper-inference";  
}
```

This method returns an arbitrary string that the Kafka broker uses to identify the Kafka consumer.

```
@Override  
public EmbeddedPaper transformToOutputMessage(final EmbeddedChunkable embeddedChunkable) {  
    return EmbeddedPaper.fromEmbeddedChunkable(embeddedChunkable);  
}
```

The purpose of this method is to convert the embedded contents to the defined output object.

```
@Override  
public Deserializer<String> getKeyDeserializer() {  
    return new StringDeserializer();  
}
```

```

@Override
public Serializer<String> getKeySerializer() {
    return new StringSerializer();
}

```

In the demo application, the message key is a Java String. Serialization is handled through classes provided by the Kafka API. Similarly, the serialization for input and output value is done by using `KafkaJsonSerializer` and `KafkaJsonDeserializer` from the Confluent `kafka-json-serializer` library.

The input value `Paper` and the output value `EmbeddedPaper` are simple Java records. They use Jackson's `@JsonProperty` annotation to explicitly map JSON files, providing proper serialization and deserialization. `Paper` is the message that is read from Kafka, and `EmbeddedPaper` is written to Kafka from the application.

```

public class NaiveCharacterChunker implements Chunker {

```

The `NaiveCharacterChunker` class implements the `Chunker` interface, providing a basic text chunking strategy. As the name suggests, this implementation for the demo pipeline takes a naive approach by splitting text into chunks, based on character count, without considering sentences or paragraphs.

4.4.2 Demo Deployment and Configuration

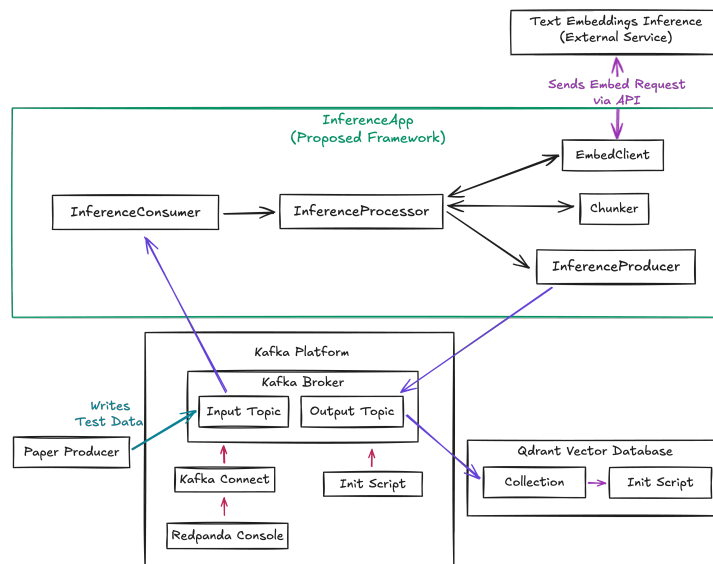


Figure 9: A high-level overview of the proposed framework and how it interacts with the demo setup

The GitHub repository [2] already contains a fully functional demo setup that can be used to showcase the demo application. This multi-container setup is managed with Docker and Docker Compose. An overview of the application stack is depicted in Figure 9. The compose file contains and configures numerous services needed to properly simulate and run the demo application in a production-like environment. The setup includes the following services:

Kafka

A single-node Kafka cluster that handles the message streaming. It is configured with two topics: raw input papers and fully processed output papers.

Kafka Connect

Manages data integration with external systems, specifically configured to write the embedded papers to the Qdrant vector database using a so-called sink connector.

Redpanda Console

A web-based user interface for monitoring Kafka. It provides access into topics, messages, and consumer groups, making debugging and monitoring the pipeline easy.

Hugging Face text-embeddings-inference

It runs the text embedding model service. For demo purposes, it's configured to use the lightweight sentence-transformers/all-MiniLM-L6-v2.

Qdrant

A vector database that stores the generated embeddings. It is configured with a collection that contains the embedded papers.

Paper Producer

A simple service that fetches papers from the Europe PMC API and produces them to Kafka, simulating a real-world data source. The source code for this application is also available on GitHub [2].

Inference App

The main application, that was implemented, demonstrates the framework's capabilities. It consumes papers, generates embeddings, and returns them to Kafka in a format compatible with the Qdrant connector.

The configuration options were already discussed in [Section 4.2.6](#). The demo application takes the following command line arguments:

```
--batch-size=1
--bootstrap-server=broker:29092
--input-topic=inference-test-paper
--output-topic=inference-test-embedded-paper
--tei-host=text-embeddings-inference
--tei-port=50051
```

4.4.3 Build and Containerisation Setup

Gradle, along with JIB, was used to automate the build process, manage dependencies, and containerize the application.

```
dependencies {
    implementation(project(":inference-pipeline"))
    // ...
}
```

In order to use the library, the inference-pipeline module is added as a local dependency.

```
// ...
jib {
    container {
        mainClass = "at.raphael.inference.PaperInferenceApp"
    }
    from {
        image = "eclipse-temurin:23-jre"
    }
    to {
        image = "paper-inference-app:" + project.version
    }
}
// ...
```

In addition to specifying the main class, the current Java 23 version of `eclipse-temurin` is used as the base image in the JIB configuration. The resulting container image is tagged with the project's version number.

4.4.4 Customise Processing Logic

- **Chunking**

The framework provides a simple interface that takes a `Chunkable` object and returns a list of chunked objects. The chunking strategy can be freely specified by implementing the `Chunker` interface and defining a custom `chunkText` method.

- **Embedding Model Selection**

The framework is set up to only need the TEI inference service's address and port. The embedding model and inference settings can be configured in the inference service, which provides a common embedding strategy across all applications that use the service.

- **Custom Serialization Schema**

In many cases and existing pipelines, we see the usage of advanced serialization schemas (e.g., Protocol Buffers, Avro). To implement those, the framework only needs the input and output format type parameters specified, along with providing serializers and deserializers.

5 Results and Evaluation

5.1 Evaluation Methodology

5.1.1 Performance Metrics

The primary metric through which performance was evaluated is message throughput in a Kafka cluster. These metrics were recorded and provide valuable insights into the performance of a Kafka pipeline:

- **Writes per Second**

In the case of the evaluation pipeline, writes per second represent the number of papers the pipeline processes and writes to Kafka per second.

- **Reads per Second**

This metric is the number of messages read from Kafka per second.

- **Processing Time**

This is the time it took to process and fully embed 20.000 papers.

5.1.2 Test Environment

Hardware Specifications

The benchmarking setup uses a Google Kubernetes Engine (GKE) cluster with two node pools: a GPU-enabled pool and a general-purpose worker pool. The Kubernetes version is 1.28.12-gke.10520002. Both pools operate within the same network subnet and run Container-Optimized OS from Google with containerd as the container runtime. The embedding model in the text embedding pipeline is GPU-accelerated in all test cases to represent a real-world scenario. CUDA version 12 was used in all tests.

GPU Node Pool

The GPU node pool utilizes a G2 machine with the following specifications:

- Nvidia L4 Tensor Core GPU
- 4 vCPUs
- 16 GB memory

Worker Node Pool

The worker node pool uses an E2 machine with these specifications:

- 8 vCPUs
- 32 GB memory

5.1.3 Software Specifications

Kafka

The Kafka cluster runs on said GKE environment with three brokers of Kafka version 3.8.

Model Serving

The text-embeddings-inference service from Hugging Face and applications with a co-located model were deployed on the same Kubernetes cluster.

```
resources:
  requests:
    cpu: "1"
```

```

memory: "2Gi"
nvidia.com/gpu: "1"
limits:
  cpu: "3"
  memory: "8Gi"
  nvidia.com/gpu: "1"
nodeSelector:
  cloud.google.com/gke-accelerator: nvidia-l4

```

The model serving container was configured to request 1 CPU core and 2 GB of memory and was limited to 3 CPU cores and 8 GB of memory. It also used one GPU node to accelerate the embedding model using the Nvidia L4 Tensor Core GPU.

Programming Languages

All Java applications and the presented framework uses the most current version of Java, Java 23. To comply with dependency requirements, Python 3.12.0 was used in all Python applications.

5.1.4 Test Dataset Characteristics

In order to test the pipeline in a realistic scenario, the benchmark setup was performed with real-world data that would also be used in a production pipeline. Europe PMC (PubMed Central) is a life-science literature database that provides free access to biomedical research publications. The data was collected using the REST API of Europe PMC [26]. The database contains over 33 million publications from various sources. It includes 10.2 million full-text articles and 6.5 million open-access articles. For the test dataset, 20,000 papers were randomly selected and downloaded. The composition of titles and abstracts is shown in [Table 2](#), including word, character, and token counts. For the inference task, only the paper abstracts were used. Other metadata, like the paper's title and DOI were included in the output to represent a real-world RAG workflow that performs retrieval based on the abstract text. This enables a perfect scenario to test and benchmark the framework.

type	title	abstract
mean word count	14.4	192.2
mean character count	112.1	1377.28
mean token count	24.2	304.44
max word count	84	916
max character count	588	5961
max token count	203	1246

Table 2: Metrics of the papers in the dataset.

5.1.5 Test Configuration

The input and output topics in the Kafka cluster were configured with one partition and three replicas. All test runs used a schema registry to validate the JSON schema. The JSON schema for input and output values is identical across all applications and tests. For the sake of gaining the most insights, numerous batching sizes were tested, starting at 1, and incrementing up to 100.

5.1.6 Embedding Model

As for the embedding model, BAAI/bge-large-en-v1.5 introduced by Xiao et al. was used across all tests [27]. The model was exclusively run GPU-accelerated. It was selected due to its excellent performance in the Massive Text Embedding Benchmark (MTEB), performed by Muenninghoff et al. [28]. It has a model size of 335 million

parameters and uses 1.25 GB of memory. The resulting embedding vector has 1024 dimensions and supports up to 512 tokens per embedding task.

5.2 Benchmarking Results

5.2.1 Approaches

Here is a summary of the test approaches that were compared to find the most suitable one. A detailed explanation of each approach is given in [Section 3](#)

- **Python-FastEmbed**

This approach uses the Python library FastEmbed to generate embeddings co-located with the Python Kafka client.

- **Python-TEI**

This approach is similar to the Python-FastEmbed approach, which uses a Python Kafka client, but the embedding model runs in an external service.

- **Kafka-Streams-TEI**

This, again, is using and calling an external service, using the Java client Kafka Streams to communicate with the Kafka broker.

- **Kafka-Text-Embedding-Inference**

This is the chosen framework that is proposed in this thesis.

5.2.2 Results

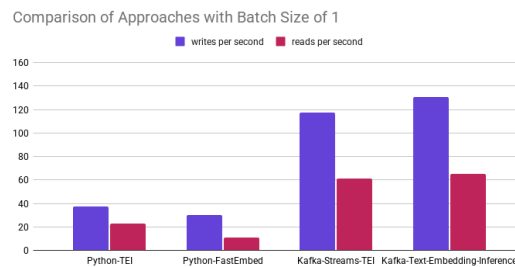


Figure 10: Comparison of Approaches with Batch Size of 1

[Figure 10](#) shows the read and write metrics for all four variants with batch size 1. We can clearly see, that the Python approaches are significantly slower than their Java counterparts.

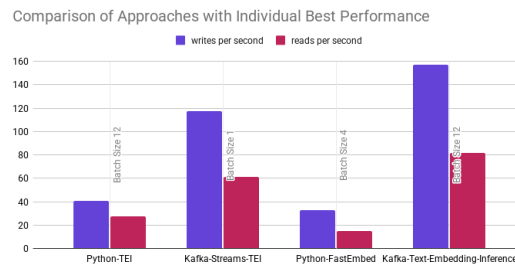


Figure 11: Comparison of Approaches with Individual Best Performance

[Figure 11](#) shows a comparison of each approach with their individual best performance. The Kafka-Text-Embedding-Inference framework profits from batching and outperforms all other solutions.

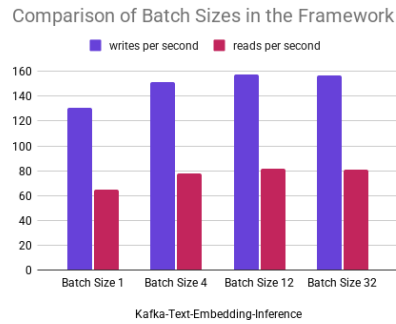


Figure 12: Comparison of Approaches with Individual Best Performance

Different batch sizes in the framework are outlined in [Figure 12](#) . When comparing batch sizes 1 to 32, an improvement of about 20 percent can be observed.

Another consideration when building flexible streaming pipelines is startup time. The co-located model has to wait for GPU node provision and initialization. The other solutions rely on an external service and thus do not need to wait for additional resources. Additionally, they can keep container size significantly lower because they do not require GPU drivers and model runtime. However, it is important to note that the external inference solution also has a considerable startup time.

5.3 Discussion

5.3.1 Analysis of Results

Maximum throughput performance across all four tested applications was achieved using the proposed framework. If this approach is compared to the embedded FastEmbed application, which is similar to the one proposed in ML-Kafka [5], a much higher throughput when it comes to generating text embeddings was achieved.

It can also be observed that both Python implementations are much slower than their Java counterparts. When both Python applications are compared, the one that uses the external service still performs better than the embedded one, even though there is an additional latency component when calling the inference service. This is an interesting takeaway about the abilities of the Python Kafka client and the process of embedding the model in the Kafka application. While Kafka Streams demonstrated strong performance without batching, the proposed Kafka-text-embedding-inference achieved a roughly 33% higher throughput through optimized batch processing.

5.3.2 Lessons Learned

The importance of proper batch size configuration became evident during testing. Increasing batch sizes initially improved performance, but a drop-off was observed past a certain threshold, indicating an optimal range for batch processing. The performance analysis revealed significant differences between client implementations. The official Java Kafka clients performed really well, when compared to their Python counterparts. Another notable finding was the performance of Hugging Face’s text-embeddings-inference service, which uses a combination of different techniques, discussed in [Section 2.5](#), to speed up inference. The advantages and disadvantages of external and co-located model serving became evident during research and later on during evaluation. Having a co-located model in a big streaming environment, potentially running several inference applications, can complicate the setup. This is especially true if GPU acceleration is desired. The decision was made to abandon the co-located approach and primarily focus on externally serving the model. The excellent performance of the TEI service further reinforced this decision. Even though there are drawbacks to using an external serving approach, the advantages of this approach cannot be denied.

5.3.3 Unexpected Findings

A surprising finding was the excellent performance of the Kafka Streams implementation, which uses record-by-record operations to call the gRPC API. Increasing the batch size by switching to a different Java API for the proposed solution achieved approximately 33 percent higher throughput. The baseline performance of the more straightforward record-by-record approach vastly exceeded initial expectations. An important finding that underlines the necessity of an appropriate error-handling solution was how machine learning libraries failed. They tended to fail unpredictably, and tracing errors was not always straightforward, especially when deployed on a remote server. This reinforces the importance of comprehensive error handling and monitoring around the inference part to ensure proper pipeline stability.

6 Conclusion and Future Work

6.1 Summary

In conclusion, this work presented the design and implementation of a framework for efficient text embedding generation in Apache Kafka stream processing systems. The framework addresses the challenge of integrating machine learning inference into data streaming applications while maintaining high throughput and resource efficiency. The main contributions include the development of an optimized library that simplifies the creation of such streaming pipelines. It also provides an optimized batching strategy for embedding generation and efficient integration of a gRPC inference service. Performance evaluations showed the effectiveness of this approach compared to other possible ways of implementing such a system. Key insights of this work are the importance of batching strategies, the significant performance differences of the used Kafka client, and the differences and obstacles of various model-serving environments. The implementation revealed that external model serving can provide better performance and flexibility than co-locating the model in the stream processing system. The `kafka-text-embedding-inference` framework provides a foundation for building efficient text embedding pipelines in Kafka while supporting important requirements for production pipelines. It offers a balance between performance and usability. Its modular design allows for complete customization and extension, making it adaptable to different use cases and characteristics.

6.2 Assumptions

The framework assumes that the input messages are text-based and suitable for common text embedding models. Although the framework does work with CPU-based inference, its primary use case and testing were performed with GPU acceleration in mind. Furthermore, the input messages can fit into memory for batch processing. Given the token limitations of current embedding models, this is feasible in most cases. For the framework to function correctly, the `text-embeddings-inference` service needs to be configured appropriately and accessible through a reliable and low-latency network connection.

6.3 Limitations

The current implementation of the framework has a few limitations that should be acknowledged. The primary one is its reliance on the `text-embeddings-inference` service, which brings many advantages compared to the embedded approach. A downside, however, is that it still faces network latencies and depends on the availability of the network and inference solution. Another limitation is that the framework relies on a fixed batch size and chunking strategy. This makes it ideal for tailoring the pipeline to specific needs, but it cannot dynamically adapt to changing needs. The inference service, Kafka, and the framework must be manually configured and adequately deployed. Another limitation is the lack of a retry approach in case of an error during processing. Usually, this would be implemented with a retry mechanism and a so-called dead-letter topic, where erroneous records are sent and can be dealt with later. As of now, the solution relies on a robust logging approach.

6.4 Future Work

6.4.1 Error Handling

Error handling has proven to be a critical aspect of implementing the framework, especially when considering the complexity of combining streaming systems with machine learning inference. Errors can happen at multiple

stages and have to be dealt with accordingly. The machine learning framework might return an error or silently fail. Similarly, network-related errors can lead to failed inference tasks. Of course, errors may also occur outside the inference part and inside the actual stream processing part. The framework can be vastly improved with solid retry mechanisms and the implementation of a service that handles and stores messages that were not processed successfully.

6.4.2 Batching

The current implementation uses a fixed batch size, which must be tested with the data at hand to find the optimal configurations. More research and testing under various settings might lead to new findings that can improve the batching strategy.

6.4.3 Large-Language-Model Integration

A direction to consider is the incorporation of large language models in evaluation to properly assess the RAG system's performance.

6.4.4 Concurrent API Calls

Another topic for further research would be to examine the best conditions for communicating with the inference service. Additional benchmarking could identify pipeline bottlenecks and guide scaling decisions. This would help determine whether performance improvements are better achieved by scaling the inference service or the Kafka application. It also reveals the potential benefits of running multiple instances of the framework against a single inference service instance, providing insights into optimal deployment configurations.

6.4.5 Auto-Scaling and Scale to Zero

Scaling is an important consideration in stream processing systems. A robust autoscaling solution can be established by using autoscalers like Kubernetes Event-driven Autoscaling (KEDA). By observing metrics in Kafka, KEDA can scale the proposed application depending on the current need. However, because the framework currently relies on an external inference service, scaling becomes more complex, primarily if multiple actors use the inference service. More research can be put into designing a flexible and economic scaling solution that scales the Kafka application and the inference service, to reduce GPU cost.

6.4.6 Quantization

Further research should investigate different quantization strategies in the pipeline. The current solution relies on passing the raw embedding from the ML model downstream. Bringing quantization into a data streaming system would require a stateful operation for most quantization techniques. An alternative approach that could be considered is to allow the downstream database to manage this step.

Bibliography

- [1] “Apache Kafka.” Accessed: Jan. 22, 2025. [Online]. Available: <https://kafka.apache.org/powered-by>
- [2] R. Lachtner, “Raphala/Kafka-Text-Embedding-Inference.” [Online]. Available: <https://github.com/raphala/kafka-text-embedding-inference>
- [3] S. Horchidan, P. H. Chen, E. Kritharakis, P. Carbone, and V. Kalavri, *Crayfish: Navigating the Labyrinth of Machine Learning Inference in Stream Processing Systems*. (2024). OpenProceedings.org. doi: 10.48786/EDBT.2024.58.
- [4] A. Farki and E. A. Noughabi, “Real-Time Blood Pressure Prediction Using Apache Spark and Kafka Machine Learning,” in *2023 9th International Conference on Web Research (ICWR)*, Tehran, Iran, Islamic Republic of: IEEE, May 2023, pp. 161–166. doi: 10.1109/ICWR57742.2023.10138962.
- [5] C. Martín, P. Langendoerfer, P. S. Zarrin, M. Díaz, and B. Rubio, “Kafka-ML: Connecting the Data Stream with ML/AI Frameworks,” *Future Generation Computer Systems*, vol. 126, pp. 15–33, Jan. 2022, doi: 10.1016/j.future.2021.07.037.
- [6] X. Wang *et al.*, “Searching for Best Practices in Retrieval-Augmented Generation,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Miami, Florida, USA: Association for Computational Linguistics, 2024, pp. 17716–17736. doi: 10.18653/v1/2024.emnlp-main.981.
- [7] “Confluentinc/Librdkafka.” Accessed: Jan. 22, 2025. [Online]. Available: <https://github.com/confluentinc/librdkafka>
- [8] “(21) Kafka Client Library Comparison | LinkedIn.” Accessed: Jan. 22, 2025. [Online]. Available: <https://www.linkedin.com/pulse/kafka-client-library-comparison-rob-golder/>
- [9] “Kafka Streams Architecture for Confluent Platform | Confluent Documentation.” Accessed: Jan. 30, 2025. [Online]. Available: <https://docs.confluent.io/platform/current/streams/architecture.html>
- [10] S. Raschka, J. Patterson, and C. Nolet, “Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence.” Accessed: Jan. 22, 2025. [Online]. Available: <http://arxiv.org/abs/2002.04803>
- [11] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks.” Accessed: Jan. 22, 2025. [Online]. Available: <http://arxiv.org/abs/1908.10084>
- [12] L. Merrick, D. Xu, G. Nuti, and D. Campos, “Arctic-Embed: Scalable, Efficient, and Accurate Text Embedding Models.” Accessed: Jan. 20, 2025. [Online]. Available: <http://arxiv.org/abs/2405.05374>
- [13] C. Zhang, M. Yu, W. Wang, and F. Yan, “MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving.”
- [14] “Huggingface/Text-Embeddings-Inference.” Accessed: Jan. 22, 2025. [Online]. Available: <https://github.com/huggingface/text-embeddings-inference>
- [15] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness.” Accessed: Jan. 21, 2025. [Online]. Available: <http://arxiv.org/abs/2205.14135>
- [16] “Introduction to gRPC.” Accessed: Jan. 22, 2025. [Online]. Available: <https://grpc.io/docs/what-is-grpc/introduction/>
- [17] “Timely Auto-Scaling of Kafka Streams Pipelines with Remotely Connected APIs with Torben Meyer.” Accessed: Jan. 22, 2025. [Online]. Available: <https://www.slideshare.net/slideshow/timely-autoscaling-of-kafka-streams-pipelines-with-remotely-connected-apis-with-torben-meyer/258209037>

- [18] S. Horchidan, E. Kritharakis, V. Kalavri, and P. Carbone, “Evaluating Model Serving Strategies over Streaming Data,” in *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning*, Philadelphia Pennsylvania: ACM, Jun. 2022, pp. 1–5. doi: 10.1145/3533028.3533308.
- [19] “Real-Time Model Inference with Apache Kafka and Flink for Predictive AI and GenAI - Kai Waehner.” Accessed: Jan. 22, 2025. [Online]. Available: <https://www.kai-waehner.de/blog/2024/10/01/real-time-model-inference-with-apache-kafka-and-flink-for-predictive-ai-and-genai/>
- [20] “Qdrant/Fastembed.” Accessed: Jan. 22, 2025. [Online]. Available: <https://github.com/qdrant/fastembed>
- [21] “How Kafka Streams Work and Their Key Benefits.” Accessed: Jan. 22, 2025. [Online]. Available: <https://double.cloud/blog/posts/2024/05/kafka-streams/>
- [22] “Picocli - a Mighty Tiny Command Line Interface.” Accessed: Jan. 22, 2025. [Online]. Available: <https://picocli.info/>
- [23] “Kafka Consumer Configuration Reference for Confluent Platform | Confluent Documentation.” Accessed: Jan. 22, 2025. [Online]. Available: <https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html#max-poll-records>
- [24] Ritu, S. Arora, A. Bhardwaj, A. Kukkar, and S. Kaur, “A Comparative Analysis of Communication Efficiency: REST vs. gRPC in Microservice- Based Ecosystems,” in *2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*, Sonipat, India: IEEE, May 2024, pp. 621–626. doi: 10.1109/INNOCOMP63224.2024.00107.
- [25] I. Olivos and M. Johansson, “Comparative Study of REST and gRPC for Microservices in Established Software Architectures.”
- [26] “Europe PMC.” Accessed: Jan. 22, 2025. [Online]. Available: <https://europepmc.org/RestfulWebService>
- [27] S. Xiao, Z. Liu, P. Zhang, N. Muennighoff, D. Lian, and J.-Y. Nie, “C-Pack: Packed Resources For General Chinese Embeddings.” Accessed: Jan. 21, 2025. [Online]. Available: <http://arxiv.org/abs/2309.07597>
- [28] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, “MTEB: Massive Text Embedding Benchmark.” Accessed: Jan. 21, 2025. [Online]. Available: <http://arxiv.org/abs/2210.07316>

7 Appendix

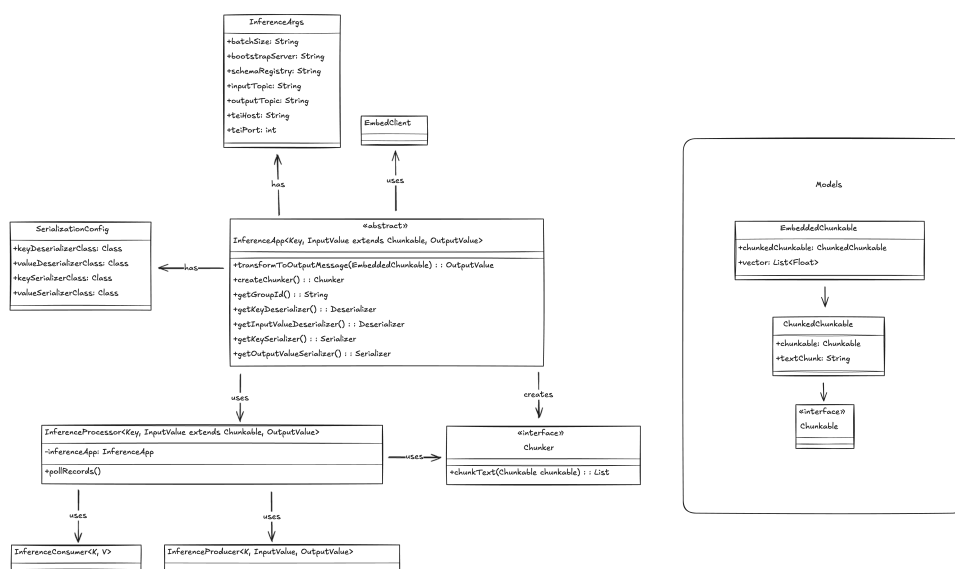


Figure 13: A class diagram of the proposed framework

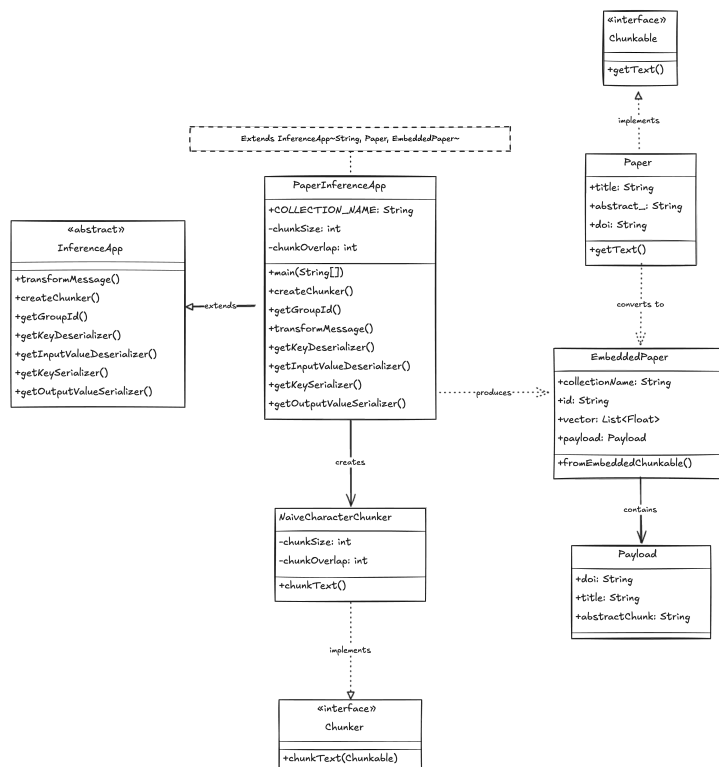


Figure 14: Class diagram of the demo application PaperInferenceApp

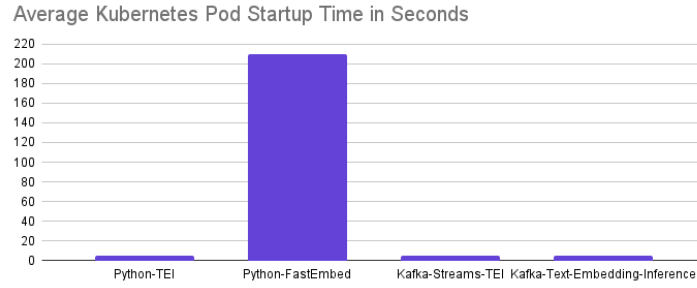


Figure 15: Average Kubernetes Pod Startup Time in Seconds

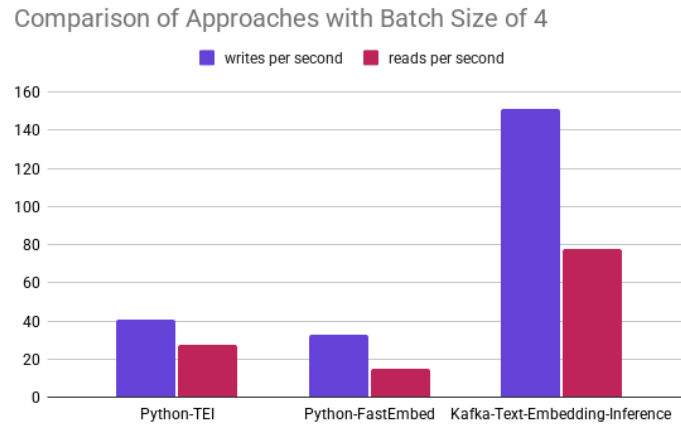


Figure 16: Comparison of Approaches with Batch Size of 4

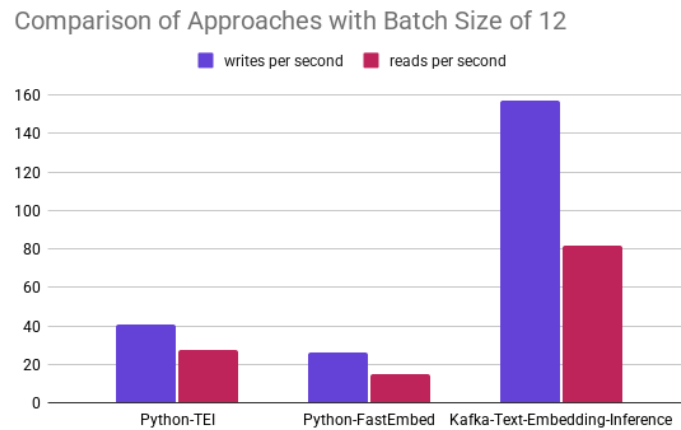


Figure 17: Comparison of Approaches with Batch Size of 12
































 demo	Running (5/9)	386.47%		
 broker 2928262e19c7 	confluentinc/cp-kafka:7.8.0	Running	9.19%	9091:9091  Show all ports (2)
 console-1 2917ab90e183  	docker.redpanda.com/redpandadata/	Created	0%	8080:8080 
 init-kafka-1 376fec4b5f09 	confluentinc/cp-kafka:7.8.0	Exited	0%	
 init-qdrant-1 6634efa88502  	curlimages/curl:7.87.0	Exited	0%	
 paper-inference-app-1 b9d0c7c981eb  	ghcr.io/raphala/kafka:text-embedding	Running	19.35%	
 paper-producer-1 2e29f66479cb  	ghcr.io/raphala/kafka:text-embedding	Exited	0%	
 qdrant-1 4233c9f58d6e  	qdrant/qdrant:v1.12.6	Running	0.15%	6333:6333  Show all ports (2)
 text-embeddings-inference 4215fc2a06c1  	ghcr.io/huggingface/text-embeddings	Running	196.63%	50051:50051 
 kafka-connect 608378fb19e  	confluentinc/cp-kafka-connect:7.8.0	Running	161.15%	8083:8083 

Figure 18: Screenshot of the docker services running in the demo setup, when fully initialized

inference-test-paper [📄](#) [🔍](#) [🔄](#)

Size: 7.75 MiB [🔍](#) Estimated messages: 5000 [🔍](#) Cleanup Policy: Delete [🔍](#) Retention: ~7 days or Infinite

[Produce Record](#) [Delete Records](#)

Messages Consumers Partitions Configuration ACL Documentation

START OFFSET: Newest - 50 MAX RESULTS: 50 [Add filter](#)

Filter table content ... [🔍](#) 84.4 kiB 53ms

TIMESTAMP	KEY	VALUE
26/01/2025, 17:39:43	10.28944/preprints202501.1727.v1 TEXT - 32 B	{"title": "Comparative Study for Community Engagement Appro...

Partition	Offset	Key	Value	Headers	Compression	Transactional
0	1142	Text (32 B)	Json (1.87 kiB)	No headers set	uncompressed	false

Key (32 B) Value (1.87 kiB) Headers

```

1 {
2   "title": "Comparative Study for Community Engagement Approach in Heritage Sites Through Two Frameworks For E
3   "abstract": "Community engagement has been revealed as a significant approach to heritage sites sustainable
4   "doi": "10.28944/preprints202501.1727.v1"
5 }

```

Figure 19: Screenshot of Redpanda Console showing a message in the input topic

inference-test-embedded-paper [📄](#) [🔍](#) [🔄](#)

Size: 14.9 MiB [🔍](#) Estimated messages: 4799 [🔍](#) Cleanup Policy: Delete [🔍](#) Retention: ~7 days or Infinite

[Produce Record](#) [Delete Records](#)

Messages Consumers Partitions Configuration ACL Documentation

START OFFSET: Newest - 50 MAX RESULTS: 50 [Add filter](#)

Filter table content ... [🔍](#) 388 kiB 83ms

TIMESTAMP	KEY	VALUE
26/01/2025, 17:46:49	10.28944/preprints202501.1781.v1 TEXT - 32 B	{"collection_name": "embeddings", "id": "66e9fc31-308f-4500-a...

Partition	Offset	Key	Value	Headers	Compression	Transactional
4	779	Text (32 B)	Json (6.98 kiB)	No headers set	gzip	false

Key (32 B) Value (6.98 kiB) Headers

```

1 {
2   "collection_name": "embeddings",
3   "id": "66e9fc31-308f-4500-ab30-2d916739d093",
4   "vector": [
5     -0.08568254,
6     -0.08740434,
7     -0.112690024,
8     0.073717386,
9     0.07931043,
10    0.018085828,
11    -0.09053083,
12    0.06724442,
13    ...

```

Figure 20: Screenshot of Redpanda Console showing a message in the output topic