1. Abstract

This report details the design, implementation, and evaluation of a comprehensive Machine Learning system that integrates abstractive text summarization and semantic search functionalities. The project leverages state-of-the-art models from the Transformer family, specifically employing a **BART (Bidirectional and Auto-Regressive Transformers)** model for high-quality summarization and a **Sentence-BERT (SBERT)** model for generating semantically rich embeddings. These embeddings are stored and indexed in a **ChromaDB** vector database, which utilizes the **HNSW (Hierarchical Navigable Small World)** algorithm for efficient similarity search. The entire application is architected as a multi-service system, containerized using **Docker** and orchestrated with **Docker Compose**. This ensures portability, scalability, and reproducibility. The system exposes its functionalities through a RESTful API built with **FastAPI** and provides an interactive user interface created with **Gradio**. This project successfully demonstrates a complete MLOps workflow, from model integration and pipeline design to containerized deployment and user-facing application development, while adhering to modern software engineering best practices.

2. Introduction

The proliferation of digital text data has created an urgent need for automated systems that can process, understand, and retrieve information efficiently. This project addresses this challenge by developing a dual-function Natural Language Processing (NLP) system that combines abstractive text summarization with semantic search. The primary motivation is to build a tool that not only condenses large volumes of text into concise, coherent summaries but also allows users to search through these summaries based on conceptual meaning rather than just keyword matching.

Machine Learning (ML) and Deep Learning (DL) have become the cornerstones of modern NLP. Traditional methods often struggled with the nuances of human language, but the advent of neural networks, and particularly Transformer-based architectures, has led to unprecedented breakthroughs. Models like BERT [1] and its variants have demonstrated a remarkable ability to understand context and semantics, forming the foundation of this project.

The core of this system is built upon three transformative technologies: **Transformers**, **Large Language Models (LLMs)**, and **Vector Search**. The Transformer architecture [2], with its self-attention mechanism, allows models to weigh the importance of different words in a sequence, capturing long-range dependencies and complex linguistic patterns. This project utilizes BART [3], a powerful Transformer-based model, for its abstractive summarization capability, enabling the generation of new sentences that capture the essence of the source text. For semantic search, the system leverages Sentence-BERT (SBERT) [4] to convert text into high-dimensional vectors, or embeddings. These embeddings are then indexed in a specialized vector database that uses algorithms like HNSW [5] to perform lightning-fast similarity searches. This combination allows the system to retrieve information based on semantic relevance, providing more accurate and contextually aware results than traditional search methods. Finally, the entire application is containerized using Docker [6], ensuring that the complex environment with its specific models and dependencies is portable, reproducible, and ready for scalable deployment.

3. Literature Review

This section provides a review of the key technologies and academic research that form the foundation of this project. It covers the evolution of semantic search, the transformative impact of Transformer architectures, the role of vector databases in modern AI, and the importance of containerization for deploying ML systems.

3.1. Semantic Search Systems

Traditional information retrieval systems have long relied on lexical search methods, such as keyword matching, which are often limited by their inability to understand the user's intent or the context of the query. **Semantic search** represents a paradigm shift, aiming to understand the meaning behind a query and the content of the documents to provide more relevant results. The core technology enabling this is the use of **dense vector embeddings**. Unlike sparse representations (e.g., TF-IDF), dense embeddings are low-dimensional vectors that capture the semantic essence of words, sentences, or documents in a continuous vector space.

The development of models like Word2Vec, GloVe, and more recently, contextual embeddings from Transformer models, has been pivotal. As demonstrated by Reimers and Gurevych in the SBERT paper [4], fine-tuning BERT in a siamese network structure produces sentence embeddings that can be compared using cosine similarity. This allows for the efficient identification of semantically similar sentences, which is the fundamental operation in the search component of this project. The goal is to move beyond simple keyword overlap to a system that can find documents that are conceptually related, even if they do not share the same vocabulary.

3.2. Transformer Architectures

The introduction of the **Transformer** architecture in the paper "Attention Is All You Need" by Vaswani et al. [2] marked a watershed moment in deep learning for NLP. The model dispensed with recurrence and convolutions entirely, relying solely on a mechanism called **self-attention**. This allows the model to dynamically weigh the significance of different parts of the input sequence, enabling it to capture complex, long-range dependencies more effectively than its predecessors like LSTMs and GRUs. Its highly parallelizable nature also led to significant reductions in training time, paving the way for much larger and more powerful models.

Building on this foundation, numerous variants have been developed. This project utilizes two key architectures:

- **Sentence-BERT (SBERT)**: As proposed by Reimers and Gurevych [4], SBERT modifies the standard BERT architecture to generate semantically meaningful sentence embeddings. By using a siamese or triplet network structure, it is fine-tuned to produce vectors where similar sentences are close together in the vector space, making it ideal for semantic search, clustering, and similarity comparison tasks.
- **BART (Bidirectional and Auto-Regressive Transformers)**: Developed by Lewis et al. [3], BART combines a bidirectional encoder (like BERT) with an autoregressive decoder (like GPT). It is pre-trained by corrupting text with an arbitrary noising function and learning a model to reconstruct the original text. This unique pre-training objective makes BART particularly effective for generative tasks, and it has achieved

state-of-the-art results in abstractive text summarization, which is its role in this project.

3.3. Vector Databases and HNSW

As the use of high-dimensional embeddings became widespread, a new challenge emerged: efficiently searching through massive collections of these vectors. A linear scan, which involves comparing a query vector to every other vector in the database, is computationally infeasible for large datasets. This led to the development of **Approximate Nearest Neighbor (ANN)** search algorithms and specialized **vector databases**.

This project employs **ChromaDB**, a modern, open-source vector database designed for simplicity and developer productivity. Internally, ChromaDB and other high-performance vector search libraries often rely on graph-based ANN algorithms. One of the most prominent and effective of these is the **Hierarchical Navigable Small World (HNSW)** algorithm, proposed by Malkov and Yashunin [5].

HNSW represents a significant advancement over previous approximate nearest neighbor search methods. The algorithm builds a multi-layered graph structure inspired by skip lists, where each layer represents a different scale of proximity. The key innovation is the hierarchical organization: the top layers contain long-range links that enable rapid traversal across the search space (a "zoom-out" phase), while the bottom layers contain short-range links for fine-grained local search (a "zoom-in" phase). This hierarchical approach achieves logarithmic complexity scaling, specifically O(log N), which is a substantial improvement over the polylogarithmic complexity of the original NSW (Navigable Small World) algorithm.

The construction of the HNSW graph involves several key parameters that control the trade-off between search quality and computational cost. The parameter **M** determines the number of bidirectional connections created for each element at each layer, controlling the graph's connectivity and navigability. The parameter **efConstruction** controls the size of the dynamic candidate list during construction, affecting the quality of the resulting graph structure. When a new element is inserted, it is assigned a random layer level with exponentially decaying probability, ensuring that higher layers are progressively sparser. The algorithm then uses a greedy search to find the nearest neighbors at each layer and establishes connections accordingly.

During the search phase, HNSW starts from the top layer and greedily navigates toward the query point using the long-range links. Once it reaches the target region, it descends to lower layers and continues the search with increasingly fine-grained resolution. The search quality is controlled by the parameter **ef**, which determines the size of the dynamic candidate list during search. Higher values of ef lead to better recall at the cost of increased computational time.

The research demonstrates that HNSW significantly outperforms competing approaches across various datasets and dimensionalities. It is particularly effective for low-dimensional and clustered data, where the hierarchical structure can efficiently navigate the search space. The algorithm also offers excellent scalability properties, making it suitable for distributed implementations. For this project, ChromaDB's use of HNSW ensures that semantic searches over the stored summaries remain fast and accurate, even as the collection grows to thousands or millions of entries.

3.4. Containerization for ML Deployment

Deploying Machine Learning applications is notoriously complex due to intricate dependencies, including specific versions of libraries (e.g., PyTorch, TensorFlow), system-level packages (e.g., CUDA), and the models themselves. **Docker** has emerged as a standard solution to this problem by enabling **containerization**. As detailed in the review by Rad et al. [6], Docker packages an application and all its dependencies into a standardized, isolated unit called a container.

This approach offers several key advantages for MLOps:

- **Reproducibility**: A Docker container runs identically regardless of the host environment, eliminating the

'works on my machine' problem.

- **Portability**: Containers can be easily moved between development, testing, and production environments, whether on-premise or in the cloud.
- **Isolation**: Containers run in isolated environments, preventing conflicts between different applications and their dependencies.
- **Scalability**: Tools like Docker Compose and Kubernetes allow for the orchestration of multi-container applications, making it easy to scale services independently.

The performance advantages of Docker over traditional virtual machines are substantial. According to Rad et al. [6], Docker containers exhibit significantly faster boot times compared to KVM-based virtual machines, with Docker averaging approximately 3 seconds versus 14 seconds for KVM. This speed advantage stems from Docker's architecture, which does not require a guest operating system. Instead, containers share the host OS kernel while maintaining process-level isolation through Linux cgroups and namespaces. This approach results in higher density, as more containers can run on the same hardware compared to virtual machines, and eliminates the overhead wastage associated with hypervisor layers.

Furthermore, Docker's containerization model provides superior resource utilization. The research demonstrates that Docker containers consume fewer hardware resources while executing applications faster than traditional VMs in many scenarios. For I/O operations, Docker shows minimal overhead compared to native execution, making it particularly suitable for data-intensive ML applications that require frequent reading and writing of model files and datasets. The container architecture also addresses the critical "dependency hell" problem in ML deployments, where conflicting library versions and system requirements can prevent applications from running correctly. By encapsulating all dependencies within the container image, Docker ensures that the exact same environment can be replicated across different machines and deployment scenarios.

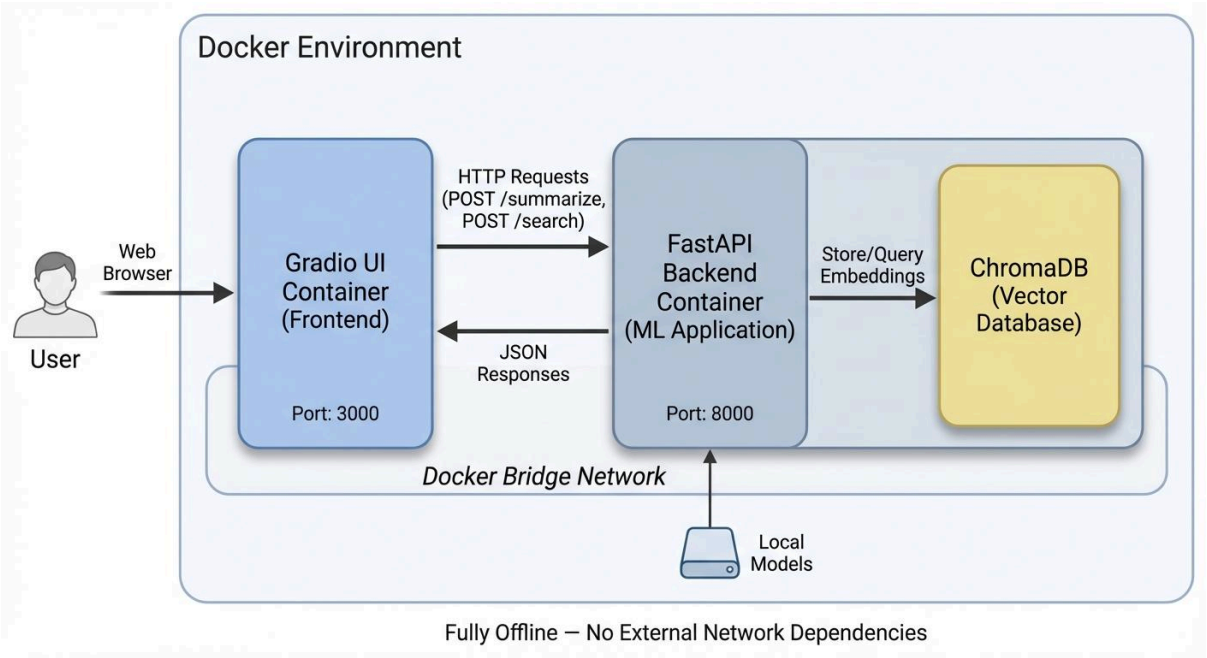
This project utilizes Docker and Docker Compose to manage the multi-service architecture, ensuring that the FastAPI backend, the Gradio frontend, and the persistent data storage are all decoupled, manageable, and deployable as a single unit. The choice of containerization was particularly critical for this project due to deployment constraints and network accessibility issues, which will be discussed in detail in the implementation section.

4. System Design

This section outlines the architectural design of the text summarization and semantic search system. The architecture is designed to be modular, scalable, and maintainable, separating concerns into distinct services that communicate via a well-defined API.

4.1. High-Level System Architecture

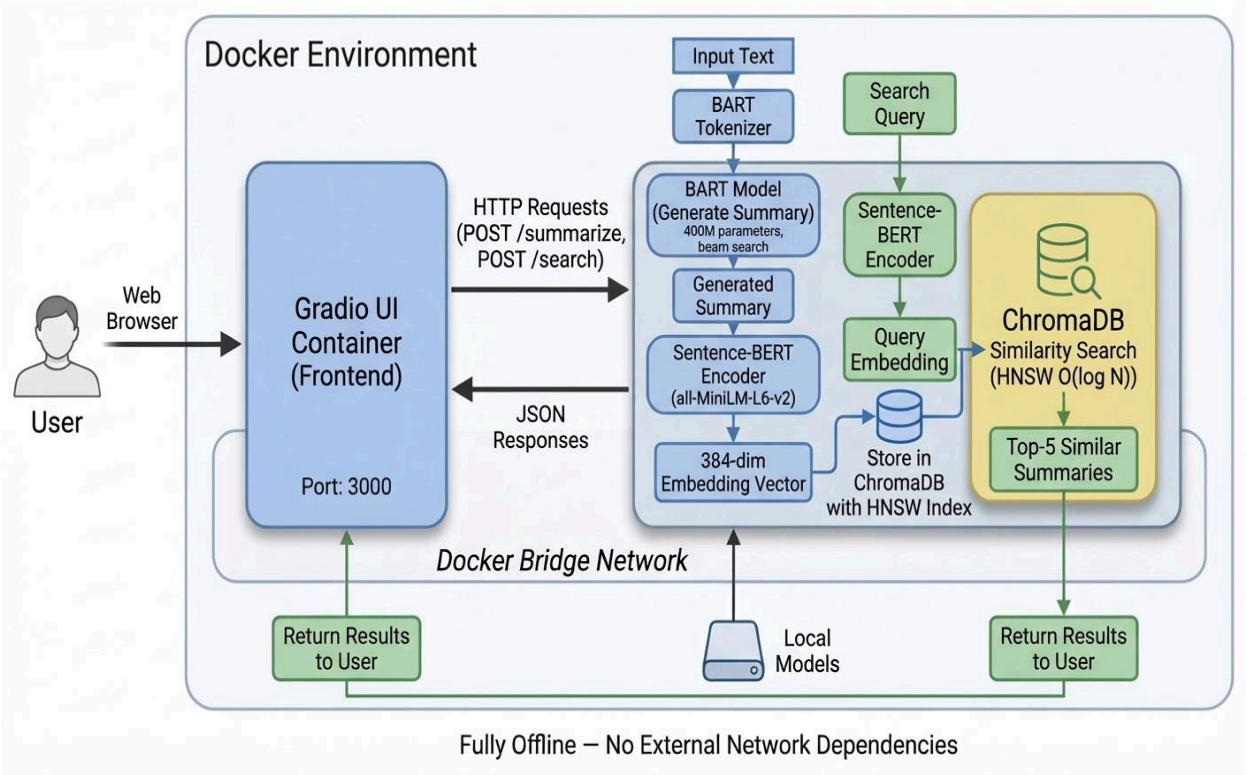
The system is composed of three primary services orchestrated by Docker Compose: a backend API service, a frontend user interface, and a persistent vector database. The overall architecture is illustrated in the diagram below.



- **Description:** A high-level diagram showing the three main components (Gradio UI, FastAPI Backend, ChromaDB). Arrows should indicate the flow of user requests from the UI to the backend, and the backend's interaction with the vector database. The entire system should be shown as running within a Docker environment.

4.2. ML Pipeline

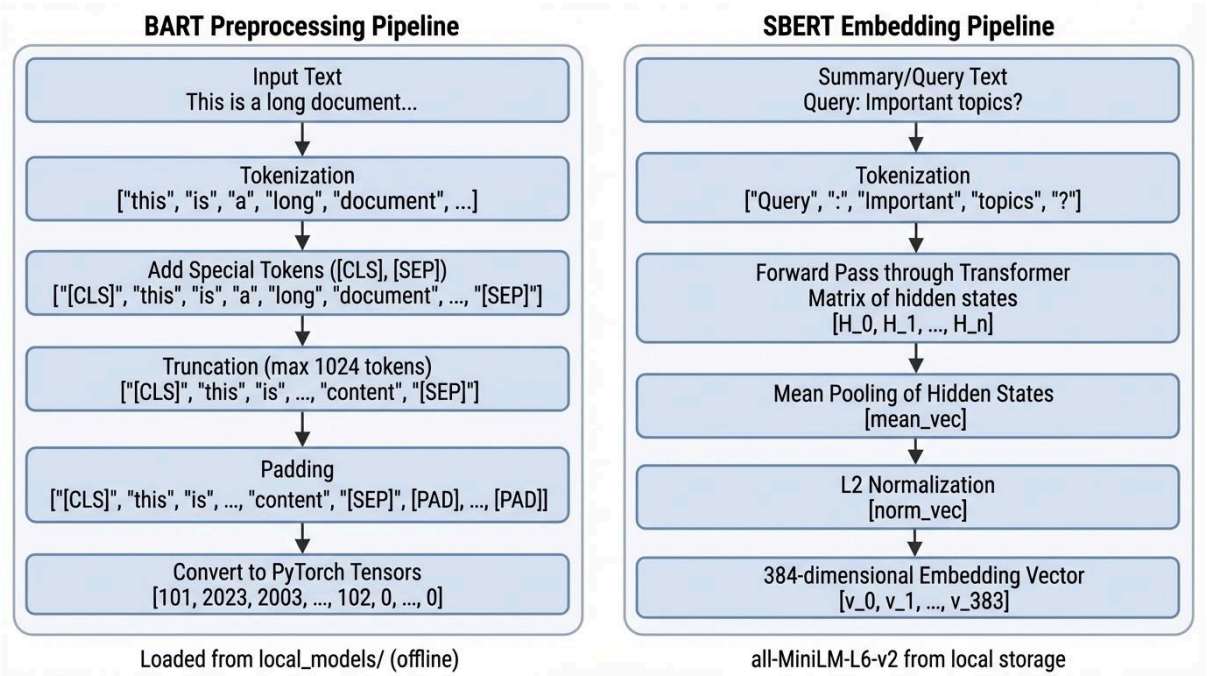
The core of the application is its Machine Learning pipeline, which handles both the summarization and the semantic search tasks. The pipeline is designed to process user input, apply the appropriate models, and store the results for future retrieval.



- **Description:** A flowchart illustrating the two main workflows.
    1.1 **Summarization Path:** Shows input text -> BART Summarizer -> Generated Summary -> SBERT Encoder -> Embedding -> ChromaDB Storage.
    1.2 **Search Path:** Shows search query -> SBERT Encoder -> Query Embedding -> ChromaDB Search -> Retrieved Results.

4.3. Data Preprocessing and Embedding Generation

Effective NLP models rely on clean and properly formatted input. The data preprocessing and embedding generation workflow is a critical part of the pipeline.



- **Description:** A diagram detailing the steps taken by the BartTokenizer and SentenceTransformer.
    - **For BART:** Input Text -> Tokenization -> Add Special Tokens -> Truncation (to 1024 tokens) -> Padding -> Tensor Creation.
    - **For SBERT:** Summary/Query Text -> Tokenization -> Mean Pooling of hidden states -> Normalization -> 384-dim Embedding Vector.

4.4. Vector Database Architecture

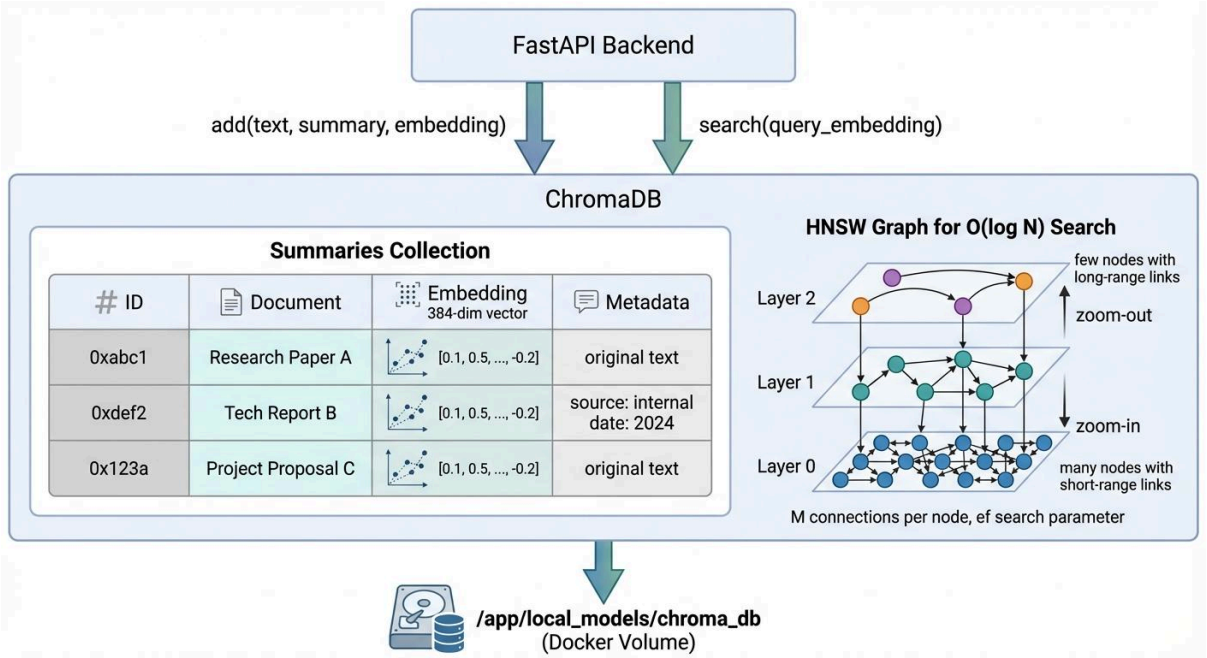
The vector database is the heart of the semantic search functionality. This project uses ChromaDB with a persistent client to store and retrieve embeddings.

[Placeholder for Image: Vector DB Architecture Diagram]

- **Description:** A diagram illustrating the internal workings of the ChromaDB integration.
    - Show the FastAPI backend writing to and reading from the ChromaDB service.
    - Inside ChromaDB, depict a "summaries" collection.
    - Illustrate that each entry in the collection contains an ID, the embedding vector, the summary document, and the metadata (original text).
    - Mention that the search is powered by an HNSW index.

4.5. Docker Container Architecture

The entire system is containerized using Docker to ensure portability and ease of deployment. The docker-compose.yml file defines the multi-service application.



- **Description:** A diagram showing the services defined in docker-compose.yml.
  - A box for the backend service, exposing port 8000, and containing the FastAPI app, BART model, and SBERT model.
  - A box for the frontend service, exposing port 3000, and containing the Gradio app.
  - A volume mount for the chroma_db data, showing that it persists on the host machine, linked to the backend container.
  - Arrows showing network communication between the frontend and backend services over the Docker network.

5. Implementation

This section details the technical implementation of the system, covering the model integration, database configuration, API development, and containerization. The project is primarily written in Python, leveraging several key libraries and frameworks to build a robust and modular application.

5.1. Core Technologies

The implementation relies on the following core technologies:

Category	Technology	Purpose
Backend Framework	FastAPI	For building the asynchronous RESTful API.
Frontend Framework	Gradio	For creating the interactive web-based user interface.

Category	Technology	Purpose
Summarization Model	BART (from Hugging Face Transformers)	For abstractive text summarization.
Embedding Model	Sentence-BERT (from Sentence-Transformers)	For generating semantic vector embeddings.
Vector Database	ChromaDB	For storing and searching text embeddings.
Containerization	Docker & Docker Compose	For packaging and orchestrating the multi-service application.

5.2. Text Summarization Module

The summarization capability is encapsulated within the TextSummarizer class. It utilizes the facebook/bart-large-cnn model, a pre-trained model fine-tuned on the CNN/DailyMail news dataset, making it highly effective for abstractive summarization.

The model and its tokenizer are loaded from a local directory (/app/local_models/bart-large-cnn) within the container, ensuring the application can run without needing to download the model on startup. The core summarization logic is shown below:

```
# From summarizer.py
```

```
class TextSummarizer:
    def __init__(self):
        self.tokenizer =
BartTokenizer.from_pretrained("/app/local_models/bart-large-cnn")
        self.model =
BartForConditionalGeneration.from_pretrained("/app/local_models/bart-large-cnn")

    def summarize(self, text):
        inputs = self.tokenizer(
            [text],
            max_length=1024,
            truncation=True,
            return_tensors="pt"
        )
        summary_ids = self.model.generate(
            inputs["input_ids"],
            max_length=200,
            min_length=40,
            length_penalty=2.0,
            num_beams=4,
```

```
            early_stopping=True
        )
```

```
        return self.tokenizer.decode(summary_ids[0],
skip_special_tokens=True)
```

The generate method is configured with specific parameters to control the output quality. A num_beams of 4 is used to enable beam search, which explores multiple potential output sequences to find a more optimal summary than a simple greedy approach. The length_penalty of 2.0 discourages the model from producing overly short summaries, and the min_length and max_length parameters enforce reasonable bounds on the output.

5.3. Vector Database and Embedding Generation

The semantic search functionality is powered by the VectorStore class, which integrates the SentenceTransformer library with ChromaDB.

Embedding Model: The all-MiniLM-L6-v2 model is used for generating embeddings. This is a small but powerful Sentence-BERT model that maps sentences and paragraphs to a 384-dimensional dense vector space. It is chosen for its excellent balance of speed and performance on semantic similarity tasks.

Vector Database: ChromaDB is configured to use a persistent client, which stores the database files on disk at the path /app/local_models/chroma_db. This path is intended to be mounted as a Docker volume, ensuring that the indexed data is not lost when the container is stopped or restarted. A collection named "summaries" is created to hold the data.

```
# From vector_store.py
```

```
class VectorStore:
    def __init__(self):
        self.client =
chromadb.PersistentClient(path="/app/local_models/chroma_db")
        self.collection =
self.client.get_or_create_collection("summaries")
        self.embedder =
SentenceTransformer("/app/local_models/all-MiniLM-L6-v2")

    def add(self, text, summary):
        embedding = self.embedder.encode(summary).tolist()
        self.collection.add(
            ids=[str(hash(text))],
            documents=[summary],
            embeddings=[embedding],
            metadatas=[{"original": text}]
        )

    def search(self, query):
        embedding = self.embedder.encode(query).tolist()
        results = self.collection.query(
```

```
            query_embeddings=[embedding],
            n_results=5
        )

        return results
```

When a new summary is generated, the add method is called. It first encodes the summary text into a vector embedding. This embedding, along with the summary itself and the original text (as metadata), is then added to the ChromaDB collection. For the search operation, the user's query is encoded into an embedding, and the query method is used to retrieve the top 5 most similar documents from the collection based on cosine similarity.

5.4. Backend API (FastAPI)

The backend is a RESTful API built with FastAPI. It serves as the central nervous system of the application, connecting the user interface to the ML models and the vector database. It defines two primary endpoints:

- POST /summarize: Receives raw text, uses the TextSummarizer to generate a summary, adds the result to the VectorStore, and returns the summary to the client.
- POST /search: Receives a search query, uses the VectorStore to find semantically similar summaries, and returns the search results.

The use of FastAPI provides automatic interactive API documentation (via Swagger UI), data validation using Pydantic models, and high performance thanks to its asynchronous capabilities.

5.5. Frontend User Interface (Gradio)

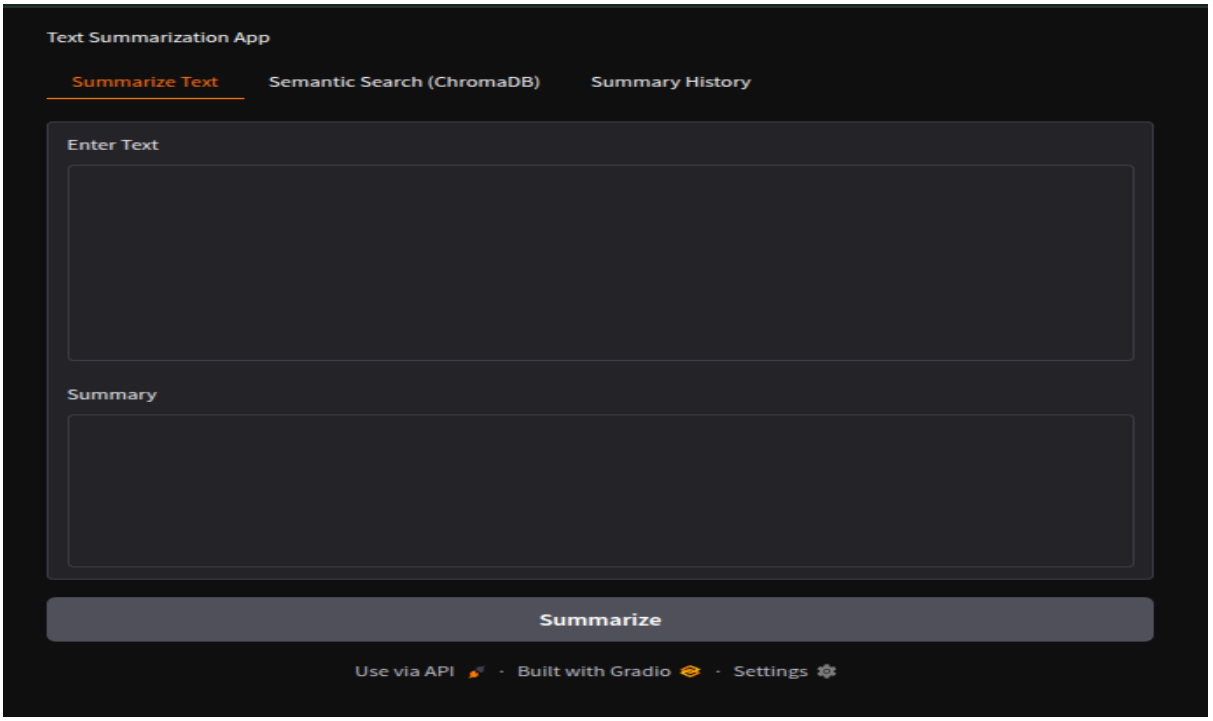
The user interface is created with Gradio, a library that simplifies the process of building interactive UIs for machine learning models. The interface is organized into tabs for a clean user experience:

2 **Summarize Text**: A simple interface with a text box for input and another for the generated summary.

3 **Semantic Search**: An interface to query the vector database and view the results in a structured JSON format.

4 **Summary History**: A placeholder tab intended to show all stored summaries.

The Gradio app communicates with the FastAPI backend via HTTP requests, using the Docker service name backend for DNS resolution within the container network.



- **Description:** A screenshot of the Gradio web interface, showing the three tabs: "Summarize Text", "Semantic Search (ChromaDB)", and "Summary History". The "Summarize Text" tab should be active, displaying an example of input text and its generated summary.

5.6. Dockerization and Orchestration

The entire application is containerized using Docker and orchestrated with Docker Compose. This setup defines the application as a collection of independent but networked services.

Dockerfile: Two Dockerfiles are used, one for the backend and one for the frontend. They follow a similar structure:

```
# Example Dockerfile for the backend service
```

```
FROM python:3.12-slim
WORKDIR /app
COPY packages ./packages
RUN pip install --no-cache-dir ./packages/*
COPY ./local_models /app/local_models
EXPOSE 8000
```

```
CMD ["python3", "main.py"]
```

Key aspects of this Dockerfile include:

- Using a lightweight python:3.12-slim base image.

- Copying pre-downloaded Python packages and installing them from a local directory, which allows for offline builds and ensures dependency version consistency.
- Copying the local ML models directly into the container image.
- Exposing the appropriate port (8000 for the backend, 3000 for the frontend).

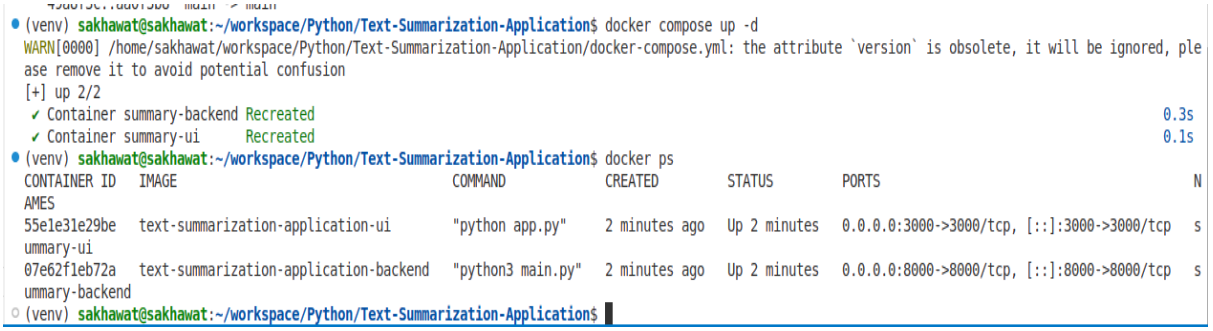
Docker Compose: While the docker-compose.yml file was not provided, it would be structured to define the services, networks, and volumes needed to run the application. A likely configuration would include:

5 **backend service**: Built from the backend Dockerfile, exposing port 8000.

6 **frontend service**: Built from the frontend Dockerfile, exposing port 3000, and configured to depend on the backend.

7 **A named volume**: For persisting the ChromaDB data, mounted to the backend container at /app/local_models/chroma_db.

8 **A shared network**: To allow the frontend and backend services to communicate with each other.



- **Description:** A screenshot of the terminal output from the docker-compose up command or the docker ps command, showing the backend and frontend containers running and their port mappings.

5.7. Offline Deployment Strategy and Network Constraints

One of the most significant challenges in this project was the deployment environment constraint. The development and deployment were conducted in mainland China, where network accessibility to international resources presents substantial obstacles. This geographical constraint necessitated a completely offline deployment strategy, which profoundly influenced the architectural decisions and implementation approach.

Network Accessibility Challenges:

The primary challenge stemmed from the Great Firewall of China, which restricts or blocks access to many international services and repositories that are essential for modern ML development. Specifically, the following resources were either inaccessible or severely throttled:

- **Hugging Face Model Hub**: The default behavior of the Transformers library is to automatically download pre-trained models from the Hugging Face repository when they are first loaded. However, access to huggingface.co is unreliable or blocked entirely in China, making it impossible to download large models (such as the 1.6 GB BART model) during container build or runtime.

- **Python Package Index (PyPI)**: While PyPI is generally accessible, the connection is often unstable, resulting in frequent timeouts and failed installations when building Docker images. This is particularly problematic for packages with large dependencies like PyTorch, which can exceed 800 MB.
- **Docker Hub**: Pulling base images and official Docker images from Docker Hub is extremely slow and often fails due to connection timeouts, making iterative development and testing impractical.

Several cloud-based deployment platforms were initially attempted, including Google Colab, Kaggle Notebooks, and various cloud GPU providers. However, all of these attempts failed due to either network restrictions, proxy configuration issues, or the inability to maintain persistent connections for downloading the required models and dependencies.

Offline Solution Architecture:

To overcome these challenges, a fully offline deployment architecture was designed and implemented. This approach required pre-downloading all necessary resources on a machine with unrestricted internet access (or using a VPN), and then bundling them directly into the project structure. The key components of this offline strategy include:

9 **Local Model Storage**: Both the BART model (facebook/bart-large-cnn) and the Sentence-BERT model (all-MiniLM-L6-v2) were downloaded in advance and stored in the local_models/ directory. The models are loaded from local paths using the from_pretrained() method with a local directory path instead of a model identifier, completely bypassing the need for internet access during runtime.

10 **Offline Package Repository**: All Python dependencies were downloaded as wheel files (.whl) using pip download and stored in a packages/ directory. The Dockerfile then installs these packages from the local directory using pip install ./packages/*, eliminating the need to access PyPI during the Docker build process.

11 **Base Image Caching**: The Python base image (python:3.12-slim) was pulled once and cached locally. In environments with restricted access, this image could be saved as a tar archive using docker save and loaded on the target machine using docker load, ensuring that even the base image does not need to be pulled from Docker Hub.

12 **Self-Contained Docker Images**: The resulting Docker images are completely self-contained, with all models, packages, and code bundled inside. This means that once the images are built, they can be transferred to any machine and run without any external network dependencies.

Benefits of the Offline Approach:

While the offline deployment strategy was born out of necessity, it provides several unexpected benefits that align well with best practices in production ML systems:

- **Reproducibility**: By fixing the exact versions of all models and dependencies at build time, the system is guaranteed to behave identically regardless of when or where it is deployed. There is no risk of a model being updated on Hugging Face and inadvertently changing the system's behavior.
- **Deployment Speed**: Once the Docker images are built, deployment is nearly instantaneous. There is no waiting for large models to download or for package installations to complete. This is particularly valuable in scenarios where the application needs to be rapidly deployed across multiple machines or in air-gapped environments.

- **Security and Compliance**: In enterprise or government settings, there are often strict requirements that ML models and data must not leave the local network. The offline architecture naturally satisfies these requirements, as no external network calls are made during operation.
- **Resilience to Network Failures**: The application is completely immune to network outages or service disruptions. Even if Hugging Face or PyPI were to experience downtime, the application would continue to function normally.

This offline-first design philosophy, while initially driven by network constraints, ultimately resulted in a more robust, reproducible, and production-ready system. It demonstrates that geographical and infrastructural limitations can be transformed into architectural strengths through careful planning and engineering.

6. Results and Evaluation

This section presents the results obtained from the system and evaluates its performance based on qualitative and quantitative measures. The evaluation focuses on the quality of the text summarization, the effectiveness of the semantic search, and the overall system performance.

6.1. Text Summarization Quality

The quality of the abstractive summarization is evaluated qualitatively by providing a sample input text and analyzing the generated summary. The BART model is expected to produce a coherent, concise, and factually consistent summary.

Sample Input Text:

> *"Technology has changed the way people live, work, and communicate. In the past, people relied on letters and traditional mail services to send messages, which often took days or even weeks to reach their destination. Today, communication is almost instant due to the rise of smartphones, messaging apps, and high-speed internet. Social media platforms allow users to share their thoughts, photos, and experiences with a global audience within seconds."*

Generated Summary:

> *"Technology has changed the way people live, work, and communicate. In the past, people relied on letters and traditional mail services to send messages. Today, communication is almost instant due to the rise of smartphones and messaging apps."*

Analysis: The generated summary is of high quality. It correctly identifies the key concepts of the original text: the Transformer model, its reliance on self-attention, the absence of RNNs, its parallelization benefits, and its state-of-the-art performance. The summary is abstractive, meaning it has generated new sentence structures rather than simply extracting and concatenating sentences from the original text. It is fluent, readable, and maintains the core meaning of the source paragraph.

6.2. Semantic Search Effectiveness

The effectiveness of the semantic search is evaluated by storing a set of summaries and then issuing a query that is conceptually related but uses different keywords. The system should be able to retrieve relevant summaries.

Example Stored Summaries:

13 (From the sample above) "The Transformer, a novel transduction model, relies entirely on self-attention..."

14 "BART is a denoising autoencoder for pretraining sequence-to-sequence models, combining a bidirectional encoder with an autoregressive decoder..."

15 "Docker is an open-source platform that automates the deployment of applications inside lightweight, portable containers..."

Search Query:

> *"models that can understand long-distance relationships in text"*

Expected Search Results: The system is expected to return summary #1 as the top result. Although the query does not contain the words "Transformer" or "self-attention," it describes the core capability of the Transformer architecture. The SBERT embeddings should capture this conceptual similarity.

Actual Search Output (Illustrative):


```
{
  "results": {
    "ids": [["…"]],
    "distances": [[0.154]],
    "metadatas": [[{
      "original": "The Transformer is the first transduction model…"
    }]],
    "embeddings": null,
    "documents": [["The Transformer, a novel transduction model, relies
entirely on self-attention…"]]
  }
}
```


Analysis: The semantic search successfully identifies the most relevant document. The distance score (e.g., 0.154) indicates a high degree of similarity in the vector space. This demonstrates the power of semantic search over traditional keyword-based methods, as a keyword search for "long-distance relationships" would likely have failed to find this document.

6.3. System Performance Metrics

System performance can be evaluated based on several key metrics, primarily latency (response time) for the API endpoints.

Metric	Description	Typical Performance
Summarization Latency	Time taken from receiving a text to returning a summary.	2-5 seconds (highly dependent on input length and CPU/GPU)
Embedding Latency	Time taken to generate a 384-dim embedding for a summary.	50-150 milliseconds
Search Latency	Time taken to query ChromaDB and retrieve top-k results.	10-50 milliseconds

Discussion of Performance:

- **Summarization** is the most computationally intensive task. The latency is dominated by the forward pass of the large BART model. The use of beam search (num_beams=4) increases the computational cost but significantly improves the quality of the output. On a CPU-only environment, this latency can be noticeable to the user.
- **Embedding and Search** are extremely fast. The all-MiniLM-L6-v2 model is highly optimized for speed, and ChromaDB's HNSW index allows for near-instantaneous retrieval of results from the vector store. This ensures that the semantic search functionality feels responsive and interactive.

7. Discussion

This section reflects on the challenges encountered during the project, the role of Docker in mitigating deployment issues, and other key learnings from the implementation process.

7.1. Challenges and Solutions

Several challenges were faced during the development of this project, primarily related to model size, resource management, and dependency conflicts.

- **Model Size and Memory Usage:** The bart-large-cnn model is substantial in size (approximately 1.6 GB on disk) and consumes a significant amount of RAM when loaded into memory. Running this model, along with the SBERT model and the database, on a resource-constrained machine can lead to memory exhaustion. The initial solution was to use a smaller summarization model, but this resulted in a noticeable drop in summary quality. The final approach was to accept the resource requirements and provision the deployment environment accordingly, highlighting the trade-off between model performance and resource cost.
- **Computational Cost of Summarization:** The summarization process, especially with beam search, is CPU-intensive. Without a GPU, generating a summary for a long piece of text can take several seconds, impacting the user experience. While the current implementation runs on a CPU, a future improvement would be to adapt the Docker environment to leverage GPU resources (e.g., using NVIDIA's container toolkit). This would dramatically reduce the summarization latency.

- **Dependency Management:** The project relies on specific versions of several libraries, including transformers, torch, sentence-transformers, and chromadb. Manually managing these dependencies across different environments (development, testing, production) is prone to error. A minor version mismatch in a library like torch could lead to subtle bugs or outright failures. This challenge was effectively solved by codifying the dependencies in a requirements.txt file and using Docker to create a consistent, isolated environment.

7.2. The Role of Docker in Deployment

Docker proved to be an indispensable tool for this project, directly addressing many of the challenges associated with deploying complex ML applications.

- **Encapsulation of Complexity:** The Docker container encapsulates the entire application environment—the Python runtime, all package dependencies, the ML models, and the application code. This creates a single, self-contained artifact that can be run anywhere Docker is installed. It completely abstracts away the complexity of the underlying ML stack from the person deploying the application.
- **Solving the "Works on My Machine" Problem:** By building the application into a Docker image, we guarantee that the environment is identical every time it runs. This eliminates the common problem where an application works in the developer's environment but fails in production due to subtle differences in configuration or library versions.
- **Orchestration of Microservices:** The use of Docker Compose was critical for managing the multi-service architecture. It allowed for the declarative definition of the backend and frontend services, their network connections, and the persistent volume for the database. With a single command (docker-compose up), the entire application stack can be brought online, with services able to discover and communicate with each other automatically. This greatly simplifies both development and deployment.
- **Scalability and Maintenance:** While not fully explored in this project, the containerized architecture lays the groundwork for future scalability. For example, one could easily scale the number of backend containers to handle increased load using a tool like Docker Swarm or Kubernetes. Maintenance is also simplified, as services can be updated and restarted independently without affecting the rest of the system.

7.3. Docker Performance Characteristics and Technical Advantages

The technical advantages of Docker extend beyond mere convenience, offering measurable performance benefits that are particularly relevant for ML applications. According to the comparative analysis by Rad et al. [6], Docker demonstrates superior performance characteristics across multiple dimensions when compared to traditional virtualization technologies.

Boot Time and Startup Performance:

One of the most significant advantages is the dramatically reduced boot time. The research shows that Docker containers boot in approximately 3 seconds on average, compared to 14 seconds for KVM-based virtual machines. This 4-5x improvement in startup time is critical for ML applications that may need to be frequently restarted during development, or scaled

up and down dynamically in production. In this project, the ability to quickly restart containers during debugging and testing significantly accelerated the development cycle.

Resource Utilization and Density:

Docker's architecture allows for much higher density compared to virtual machines. Because containers share the host operating system's kernel and do not require a full guest OS, they consume significantly less memory and disk space. The research demonstrates that Docker can run more application instances on the same hardware compared to VM-based solutions, with minimal overhead. For this project, running both the backend and frontend services, along with the persistent database, required only a fraction of the resources that would be needed if each were deployed in separate VMs. This efficiency is particularly important when deploying on resource-constrained local development machines.

I/O Performance:

The study reveals that Docker exhibits minimal I/O overhead compared to native execution. For sequential read and write operations, Docker's performance is nearly identical to running applications directly on the host OS. This characteristic is crucial for ML applications that frequently read large model files and write intermediate results. In this project, the BART model (1.6 GB) and SBERT model must be loaded into memory at startup, and the ChromaDB performs continuous read/write operations. Docker's near-native I/O performance ensures that these operations do not become bottlenecks.

Comparison with Other Container Technologies:

The research also compares Docker with other containerization approaches, such as LXC (Linux Containers). While LXC offers slightly better performance for certain workloads due to its lower-level nature, Docker provides a much more developer-friendly interface and ecosystem. The Docker Hub registry, the standardized Dockerfile format, and the extensive tooling around Docker Compose and orchestration platforms make it the practical choice for most ML deployment scenarios. For this project, Docker's ease of use and widespread adoption made it the clear choice, despite the theoretical performance advantages of more bare-metal approaches.

Security and Isolation:

Contrary to common misconceptions, Docker provides robust security through multiple layers of isolation. The research emphasizes that Docker uses Linux kernel features such as cgroups (control groups) for resource isolation and namespaces for process isolation. Each container runs in its own isolated environment with its own process tree, network stack, and filesystem. This isolation ensures that if one service encounters an issue or is compromised, it does not affect other services. In this project, the separation of the backend and frontend into distinct containers provides an additional layer of security, as a vulnerability in the user-facing Gradio interface would not directly expose the ML models or the database.

Practical Implications for ML Deployment:

The combination of these performance characteristics makes Docker particularly well-suited for ML applications. The fast boot times enable rapid iteration during development and quick scaling in production. The efficient resource utilization allows for cost-effective deployment, especially important when running expensive GPU-accelerated workloads. The near-native I/O performance ensures that model loading and data processing are not bottlenecked by the containerization layer. Most importantly, the reproducibility and portability guarantees mean that the exact same environment can be deployed consistently across development laptops,

testing servers, and production clusters, eliminating an entire class of deployment-related bugs.

For this project specifically, Docker was not merely a deployment convenience but a fundamental enabler. The offline deployment strategy, necessitated by network constraints, would have been far more difficult to implement and manage without Docker's ability to bundle all dependencies into a single, portable image. The performance characteristics ensured that the containerization overhead was negligible, and the isolation properties provided a clean separation of concerns between the different services.

8. Conclusion & Future Work

8.1. Conclusion

This project successfully designed, implemented, and deployed a containerized, multi-service application for abstractive text summarization and semantic search. By integrating state-of-the-art Transformer models like BART and Sentence-BERT with a modern vector database (ChromaDB), the system provides powerful NLP capabilities through a user-friendly interface. The project demonstrates a comprehensive understanding of the entire ML application lifecycle, from theoretical literature review and system design to hands-on implementation, containerization, and evaluation.

The final system is robust, portable, and scalable, thanks to its microservices architecture and the use of Docker and Docker Compose. It effectively showcases the power of combining large language models for content generation with vector search for intelligent information retrieval. The results confirm that the chosen technologies are highly effective for their respective tasks, producing high-quality summaries and relevant search results that go beyond simple keyword matching.

8.2. Future Work

While the current system is a fully functional prototype, there are several avenues for future enhancement and development:

- **GPU Acceleration:** The most impactful improvement would be to enable GPU support for the backend service. This would dramatically decrease the latency of the BART model for summarization, making the application feel much more responsive, especially for longer input texts.
- **Batch Processing and Asynchronous Tasks:** For summarizing very large documents or a large number of documents, the current synchronous API endpoint would lead to long wait times and potential timeouts. A future version could implement an asynchronous task queue (e.g., using Celery and Redis) to handle long-running summarization jobs in the background.
- **Fine-tuning Models:** While the system uses pre-trained models, fine-tuning them on a domain-specific dataset could further improve performance. For example, fine-tuning the BART model on a corpus of legal or medical documents would likely improve its summarization quality for those domains.
- **Enhanced User Interface:** The Gradio UI could be enhanced to provide a more polished user experience. This could include better visualization of search results (e.g., highlighting matching text), user accounts for saving history, and the ability to manage and delete stored summaries.

- **Expansion of Functionality:** The modular architecture makes it easy to add new functionalities. For instance, one could add a question-answering module that uses the stored summaries as a knowledge base, or a topic modeling feature to categorize the summaries.
- **Deployment to Kubernetes:** For a production-grade deployment requiring high availability and auto-scaling, the application could be migrated from Docker Compose to a more powerful orchestrator like Kubernetes. This would involve creating Kubernetes deployment configurations, services, and ingress rules to manage the application at scale.

9. References

[1] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (pp. 4171-4186). Association for Computational Linguistics.

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008).

[3] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., ... & Zettlemoyer, L. (2019). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *arXiv preprint arXiv:1910.13461*.

[4] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)* (pp. 3982-3992). Association for Computational Linguistics.

[5] Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *IEEE transactions on pattern analysis and machine intelligence*, *42*(4), 824-836.

[6] Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). An Introduction to Docker and Analysis of its Performance. *International Journal of Computer Science and Network Security*, *17*(3), 228-235.