Handlo

VLM based RAG for caption & hashtag generator for Image

ABSTRACT

Handlo is an innovative social media co-pilot leveraging generative AI to simplify and enhance social media management. It addresses challenges like content creation, audience engagement, and trend adaptation by integrating advanced multimodal models like BLIP and TinyLlama for context retrieval and caption generation. Handlo employs Qdrant for vector storage, enabling personalized, trend-based hashtag recommendations. Beyond content optimization, it supports post-scheduling and engagement tracking to maximize impact. Future enhancements include music recommendations, optimal post scheduling insights, trend analysis, and cross-platform analytics. Designed for scalability and efficiency, Handlo empowers users to maintain an engaging and impactful online presence while streamlining repetitive tasks and boosting productivity.

***Keywords*:** Generative AI, Retrieval Augmented Generation, Large Language Model.

TABLE OF CONTENTS

Title **Page No.**

LIST OF ACRONYMS

ViT	Vision Transformer
VLM	Vision Language Model
API	Application Programming Interface
BLIP	Bootstrapping Language-Image Pre-training
GenAI	Generative Artificial Language
RAG	Retrieval Augmented Generation

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

Growing complexity of managing multiple platforms, audiences, and content strategies. Handling repetitive tasks such as captions and relevant hashtags for social media is kind of an hectic task which hampers productivity. This solution provides an edge for creating eye-catching captions and suggesting hashtags for boosting engagement.

In addition to this, this solution also provides scheduled posting, engagement tracking. Unlike general social media accelerators, it focuses specifically on enhancing post engagement through trending hashtag recommendation.

1.2 Problem Statement

Managing multiple social media platforms, audiences, and content strategies has become increasingly complex for individuals and businesses, leading to inefficiencies in maintaining an active and engaging online presence. Existing social media management solutions focus largely on automating posting and basic analytics, but they often fall short in addressing the core challenge of maximizing post engagement.

Current solutions fail to:

- Provide personalized and trending hashtag recommendations for boosting engagement.
- Efficiently assist in creating compelling captions that resonate with target audiences.
- Focus specifically on post engagement metrics beyond basic scheduling and analytics.

The absence of these key features hampers content performance, making it difficult to keep up with dynamic audience expectations and trends. This results in missed opportunities for social media growth and lower overall productivity for social media managers.

1.3 Objectives

- **Implement Efficient Image Context Retrieval**: Develop the ContextRetriever component to accurately analyze and extract contextual information from images using the BLIP model, ensuring a high level of precision and relevance in generated context.

- **Enhance Caption Generation**: Utilize the TinyLlama model to generate contextually appropriate captions based on the extracted image context and specified mood, aiming to improve user engagement and interactivity within the application.

- **Develop Hashtag Suggestion Mechanism**: Create the HashtagRetriever component to generate relevant hashtags from user-provided sentences by analyzing mood, objects, and locations, thereby enhancing content discoverability and user interaction.

- **Optimize Performance**: Ensure that both components leverage efficient processing techniques and optimize their integration with Qdrant for vector storage and Hugging Face for language model inference to support real-time applications.

Chapter 2

Literature Review

2.1 What are Visual Language Model

Vision language models are broadly defined as multimodal models that can learn from images and text. They are a type of generative models that take image and text inputs, and generate text outputs. Large vision language models have good zero-shot capabilities, generalize well, and can work with many types of images, including documents, web pages, and more. The use cases include chatting about images, image recognition via instructions, visual question answering, document understanding, image captioning, and others. Some vision language models can also capture spatial properties in an image. These models can output bounding boxes or segmentation masks when prompted to detect or segment a particular subject, or they can localize different entities or answer questions about their relative or absolute positions.

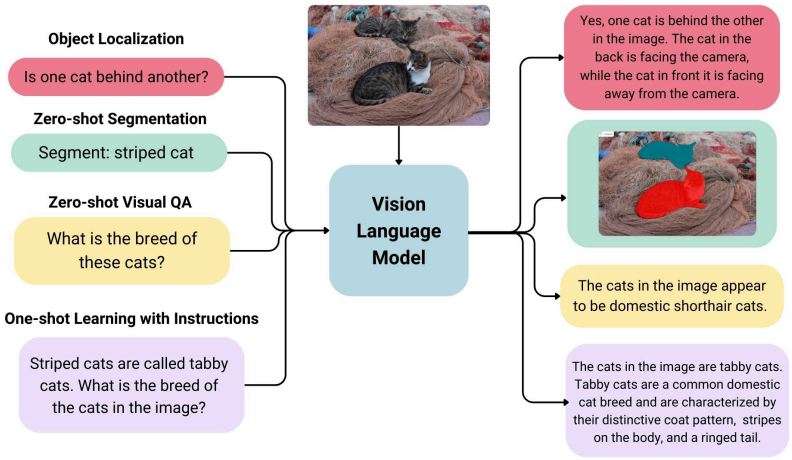

Figure 2.1: Diagramatic Understanding of VLM

2.1.1 Vision Transformer (ViT)

Vision Transformers (ViTs) are deep learning models for computer vision, adapting the transformer architecture from NLP. They divide images into patches, embed them, and

process them as sequences. Using Multi-Head Self-Attention, ViTs capture global relationships between image regions. A learnable classification token summarizes the image for tasks like classification. While highly effective on large datasets, ViTs require significant data and computational resources compared to convolutional neural networks (CNNs).

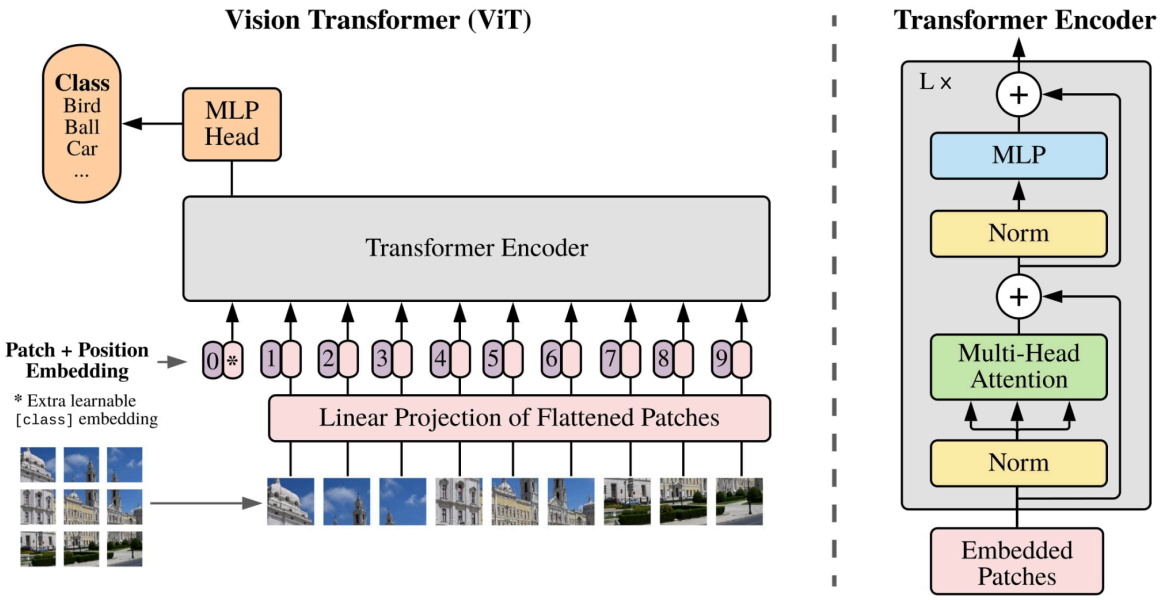

Figure 2.2: Architecture of Vision Transformer.

2.1.2 BLIP Model

BLIP: Bootstrapping LanguageImage Pre-training for unified vision-language understanding and generation. BLIP is a new VLP framework which enables a wider range of downstream tasks than existing methods. It introduces two contributions from the model and data perspective, respectively:

(a) Multimodal mixture of Encoder-Decoder (MED): a new model architecture for effective multi-task pre-training and flexible transfer learning. An MED can operate either as a unimodal encoder, or an image-grounded text encoder, or an image-grounded text decoder. The model is jointly pre-trained with three vision-language objectives: imagetext contrastive learning, image-text matching, and image conditioned language modeling.

(b) Captioning and Filtering (CapFilt): a new dataset bootstrapping method for learning from noisy image-text pairs. We finetune a pre-trained MED into two modules: a captioner to produce synthetic captions given web images, and a filter to remove noisy captions from both the original web texts and the synthetic texts.

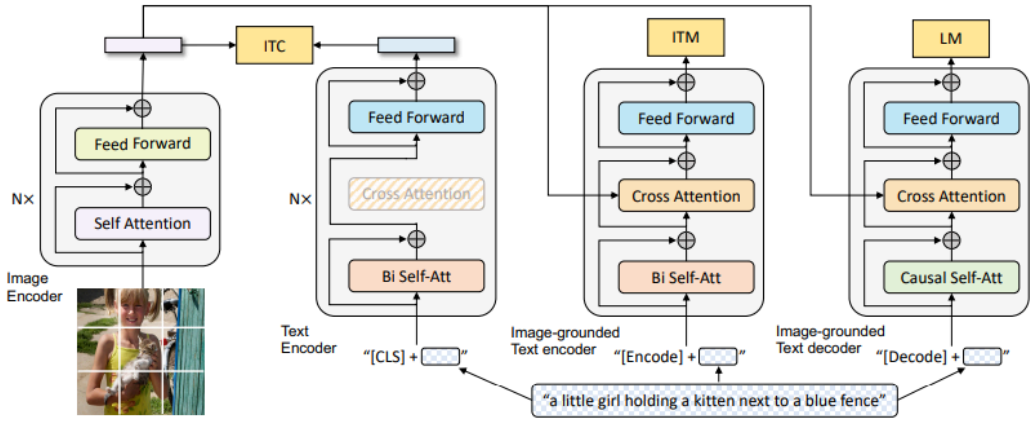

Figure 2.3: Architecture of BLIP Model

2.2 What are Retrieval Augmented Generation

Retrieval-augmented generation (RAG) is a technique for enhancing the accuracy and reliability of generative AI models with facts fetched from external sources.In other words, it fills a gap in how LLMs work. Under the hood, LLMs are neural networks, typically measured by how many parameters they contain. An LLM's parameters essentially represent the general patterns of how humans use words to form sentences.

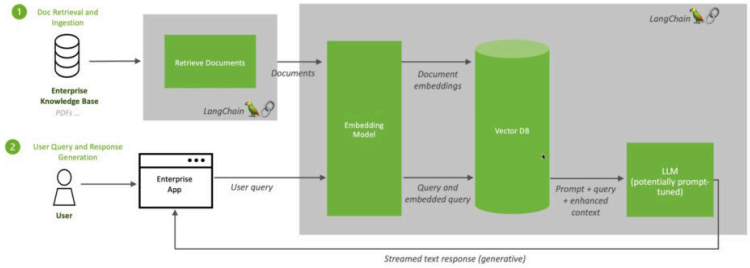

Figure 2.4: Diagramatic view of RAG

2.2.1 Llama Index

LlamaIndex is a framework for building context-augmented generative AI applications with LLMs including agents and workflows. LLMs offer a natural language interface between

humans and data. LLMs come pre-trained on huge amounts of publicly available data, but they are not trained on your data. Your data may be private or specific to the problem you're trying to solve. It's behind APIs, in SQL databases, or trapped in PDFs and slide decks. Context augmentation makes your data available to the LLM to solve the problem at hand. LlamaIndex provides the tools to build any context-augmentation use case, from prototype to production. Our tools allow you to ingest, parse, index and process your data and quickly implement complex query workflows combining data access with LLM prompting. The most popular example of context-augmentation is Retrieval-Augmented Generation or RAG, which combines context with LLMs at inference time.

2.2.2 TinyLlama

The TinyLlama project aims to pre-train a 1.1B Llama model on 3 trillion tokens. With some proper optimization, we can achieve this within a span of "just" 90 days using 16 A100-40G GPUs. The training started on 2023-09-01. We adopted exactly the same architecture and tokenizer as Llama 2. This means TinyLlama can be plugged and played in many open-source projects built upon Llama. Besides, TinyLlama is compact with only 1.1B parameters. This compactness allows it to cater to a multitude of applications demanding a restricted computation and memory footprint.

2.2.3 Mistral

Mistral AI is an innovative company in the field of artificial intelligence, focusing on creating advanced open-weight large language models (LLMs). Founded in 2023 by former researchers from leading AI organizations, Mistral aims to push the boundaries of what LLMs can achieve, with a strong emphasis on open-source contributions. The company is based in Europe and has quickly gained attention for its expertise and ambitious projects.

Mistral-7B-Instruct-v0.3 is a fine-tuned version of the Mistral 7B model, optimized specifically for instruction-following tasks. This model is designed to excel in conversational and assistant-like interactions, making it suitable for a variety of applications where clarity, accuracy, and responsiveness are key.

2.3 What are Vector Databases

A vector database is a collection of data stored as mathematical representations. Vector databases make it easier for machine learning models to remember previous inputs,

allowing machine learning to be used to power search, recommendations, and text generation use-cases. Data can be identified based on similarity metrics instead of exact matches, making it possible for a computer model to understand data contextually.

When one visits a shoe store, a salesperson may suggest shoes that are similar to the pair one prefers. Likewise, when shopping in an ecommerce store, the store may suggest similar items under a header like "Customers also bought..." Vector databases enable machine learning models to identify similar objects, just as the salesperson can find comparable shoes and the ecommerce store can suggest related products. (In fact, the ecommerce store may use such a machine learning model for doing so.)

To summarize, vector databases make it possible for computer programs to draw comparisons, identify relationships, and understand context. This enables the creation of advanced artificial intelligence (AI) programs like large language models (LLMs).

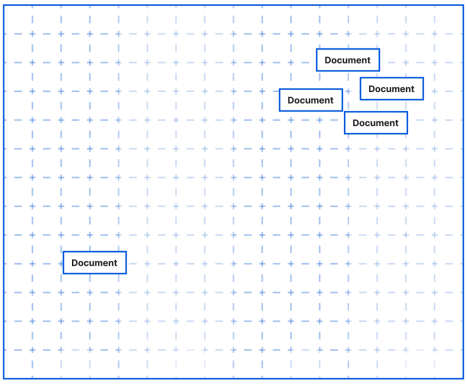

Figure 2.5: Graphical representation for Vectors

2.3.1 Quadrant Vector Database

Qdrant “is a vector similarity search engine that provides a production-ready service with a convenient API to store, search, and manage points (i.e. vectors) with an additional

payload." You can think of the payloads as additional pieces of information that can help you hone in on your search and also receive useful information that you can give to your users. You can get started using Qdrant with the Python qdrant-client, by pulling the latest docker image of qdrant and connecting to it locally, or by trying out Qdrant's Cloud free tier option until you are ready to make the full switch.

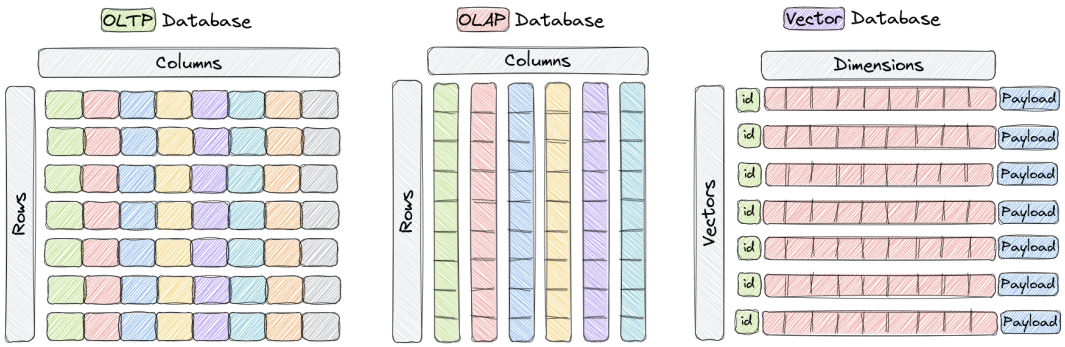

Figure 2.6: Vector Database Structures

2.3.2 Quadrant VectorStore

Vector stores are specialized data stores that enable indexing and retrieving information based on vector representations. These vectors, called embeddings, capture the semantic meaning of data that has been embedded. Vector stores are frequently used to search over unstructured data, such as text, images, and audio, to retrieve relevant information based on semantic similarity rather than exact keyword matches.

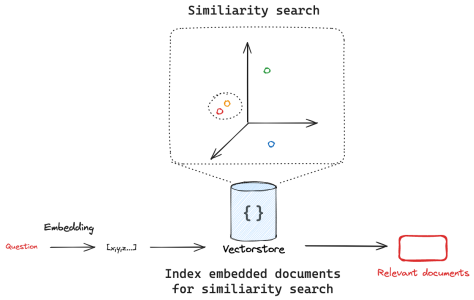

Figure 2.7: Vector Store in working

Chapter 3

Implementation and Result

3.1 System Architecture

3.1.1 Tools and Technology

Machine Learning and Natural Language Processing (NLP): Understanding of machine learning concepts and experience with NLP techniques, especially using models like BLIP and TinyLlama, is necessary for context retrieval and caption generation.

Data Handling and Processing: Skills in manipulating and processing images (using libraries like PIL) and text data for efficient input handling.

Database Management: Familiarity with vector databases (e.g., Qdrant) for efficient storage and retrieval of embeddings and context.

APIs and Web Services: Experience with RESTful APIs, especially for interacting with services like Hugging Face and Qdrant, is important for data retrieval and processing.

3.1.2 Design and Architecture

Components:

1. User Input (User Image): Users upload an image as input to the system.
2. UI Service: The central interface that handles the interaction between the user and the backend services.
Passes the uploaded image to the backend for further processing.
3. Backend Services:
Image Context Service: Extracts contextual information from the user-provided image. Leverages the BLIP Model for understanding and generating contextual data
Caption Generator: Uses the image context to generate a descriptive caption for the image.
Communicates with the Image Context Service for input.
Hashtag Recommendation Engine: Generates relevant hashtags based on the extracted image context and/or generated caption.
Both recommendation engines use image context as a primary input.

Data Flow:

1. The user uploads an image through the UI Service.
2. The UI Service forwards the image to the Image Context Service.
3. The Image Context Service processes the image using the BLIP Model and provides the context.
4. This context is: Used by the Caption Generator to create captions. Passed to the Hashtag Recommendation Engine for hashtag suggestions.
5. Outputs (captions, hashtags) are sent back to the UI Service for display to the user.

This architecture emphasizes modular design, where each service performs a specific task, leveraging machine learning models for intelligent output generation.

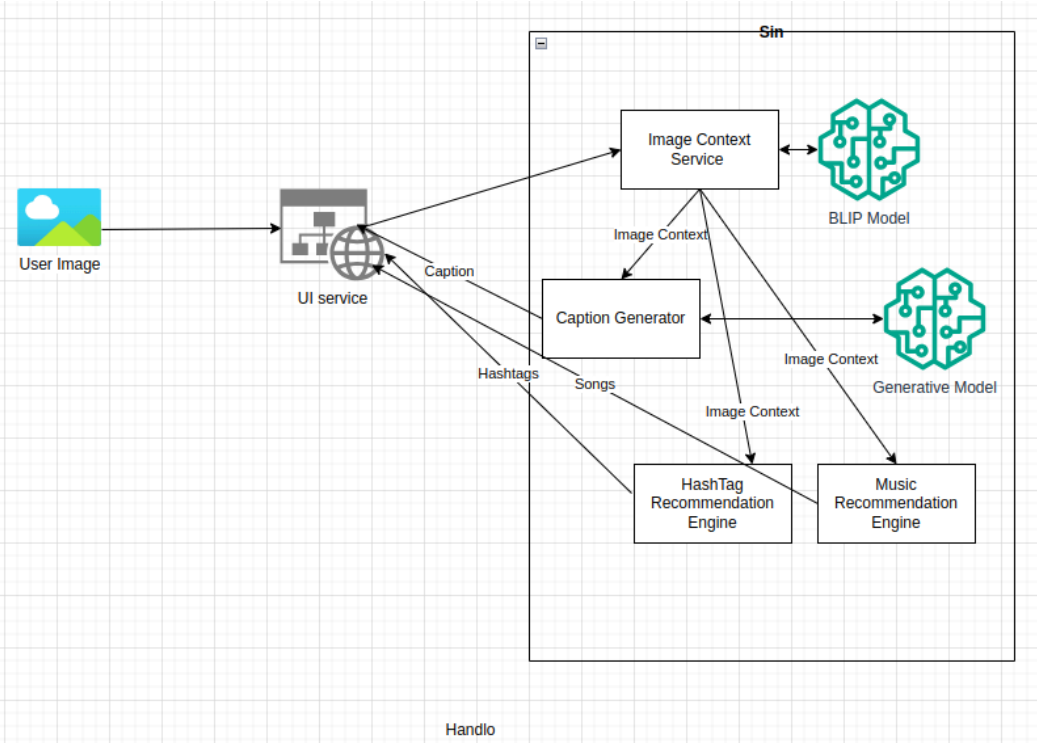

Figure 3.1: Architecture of Handlo

Chapter 4

Conclusion

4.1 Key Findings

Handlo addresses the growing complexity of managing social media by leveraging generative AI for caption creation and trending hashtag recommendations. It employs a multimodal ML pipeline with BLIP for image context retrieval, TinyLlama for caption generation, and Qdrant for vector storage. Handlo enhances post engagement through personalized, context-aware hashtags, efficient API integration, and scalability, filling gaps left by traditional social media management tools.

4.2 Future Contribution

1. Music Recommendation for Posts: Integrate AI-driven music recommendation to suggest background tracks for videos or reels based on post mood, context, and trends, enhancing content appeal and audience engagement.

2. Post Scheduling Recommendations: Employ analytics to determine optimal posting times based on audience activity, engagement trends, and platform-specific data, boosting reach and visibility.

3. Trend Analysis Integration: Add features to analyze trending hashtags, captions, and content formats, enabling users to stay ahead in social media trends.

4. Audience Sentiment Analysis: Introduce sentiment tracking to analyze audience reactions to posts, offering insights for tailoring future content strategies.

5. Collaboration Tools: Include multi-user management and collaboration features for teams to co-manage content calendars and post approvals.

References

[1] Salesforce, “BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation,” Hugging Face Research Paper.

[2] Twitter, “Twitter hashtag dataset” Hugging Face.

[3] Qdrant and LlamaIndex, “Combining Qdrant and LlamaIndex to keep Q&A systems up-to-date” Github.

[4] LlamaIndex, “Llama Index Documentation,” Documentation.

[5] zafercavdar, “Language Detection,” fasttext-langdetect.

[6] Qdrant, “Qdrant Client,” Documentation

Appendices

Appendix A

Code Attachments

The following is the partial / subset of the code. Code of some module(s) have been wilfully supressed.

A.1 Code

./app.py

```
import uvicorn
import traceback
from fastapi import FastAPI, Body
from fastapi.middleware.cors import CORSMiddleware
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.sdk.resources import Resource
from opentelemetry.semconv.resource import ResourceAttributes
from context_ret import ContextRetriever
from caption import Caption
from hashtag_rag.query import HashtagRetriever


resource = Resource(attributes={
   ResourceAttributes.SERVICE_NAME: "fastapi-users"
})

jaeger_exporter = JaegerExporter(
   agent_host_name="localhost",
   agent_port=6831,
)

provider = TracerProvider(resource=resource)
```

```
processor = BatchSpanProcessor(jaeger_exporter)
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)

tracer = trace.get_tracer(__name__)

app = FastAPI()

origins = [
    "http://localhost.tiangolo.com",
    "https://localhost.tiangolo.com",
    "http://localhost",
    "http://localhost:3000",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


FastAPIInstrumentor.instrument_app(app)

@app.post("/get-context")
async def get_context(response: dict = Body(...)):
    with tracer.start_as_current_span("get_context"):
        try:
            img_bytes = response['img_bytes']
            context_retriever = ContextRetriever()
            context = context_retriever.get_context(img_bytes)
            return {"context": context}
        except Exception as e:
            print(traceback.format_exc())
            return {"error": str(e)}

@app.post("/get-caption")
async def get_caption(response: dict = Body(...)):
```

```
    with tracer.start_as_current_span("get_caption"):
        try:
            context = response['context']
            mood = response['mood']
            caption = Caption(context, mood)
            captions = caption.generate_captions()
            return {"captions": captions}
        except Exception as e:
            print(traceback.format_exc())
            return {"error": str(e)}


@app.post("/get-hashtags")
async def get_hashtags(response: dict = Body(...)):
    with tracer.start_as_current_span("get_hashtags"):
        try:
            sentence = response['sentence']
            hashtag_retriever = HashtagRetriever()
            hashtags = hashtag_retriever.get_hashtags(sentence)
            return {"hashtags": hashtags}
        except Exception as e:
            print(traceback.format_exc())
            return {"error": str(e)}

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

./query.py

```
import os
from pathlib import Path
from dotenv import load_dotenv
from qdrant_client import QdrantClient
from llama_index.vector_stores.qdrant import QdrantVectorStore
from llama_index.core import Settings, SimpleDirectoryReader,
StorageContext, VectorStoreIndex, load_index_from_storage
from llama_index.vector_stores.qdrant import QdrantVectorStore
from llama_index.embeddings.fastembed import FastEmbedEmbedding
```

```
from llama_index.llms.huggingface_api import HuggingFaceInferenceAPI
from llama_index.core import PromptTemplate

load_dotenv(dotenv_path = './src/configurations/.env')

HF_TOKEN = os.getenv("HF_TOKEN")

Settings.embed_model = FastEmbedEmbedding()
Settings.llm = HuggingFaceInferenceAPI(
   model_name=os.getenv("MODEL_NAME"), token=HF_TOKEN, temperature=0.1
)


class HashtagRetriever():
   def __init__(self):
       self.query = """
                   You are a helpful AI assistant that suggests
hashtags from the sentence: {sentence}.
                   On the basis of mood, object, and place from the
provided sentence and use them to generate relevant hashtags.
                   If the sentence doesn't contain enough information,
just say 'Not enough information'.
               """
       self.BASE_PATH = os.getenv("BASE_PATH")

   def get_query_engine(self):
       persist_dir =
Path(f"{self.BASE_PATH}/data/storage_context").expanduser()
       client =
QdrantClient(path=f"{self.BASE_PATH}/vectorDB/hashtags.db")
       vector_store = QdrantVectorStore(client=client,
collection_name="collection-v1", prefer_grpc=True)
       storage_context = StorageContext.from_defaults(
                           vector_stores={'default': vector_store},
                           persist_dir=persist_dir
                       )
       index = load_index_from_storage(storage_context)
       query_engine = index.as_query_engine()
       return query_engine
```

```
   def get_hashtags(self, sentence):
       response =
self.get_query_engine().query(self.query.format(sentence=sentence))
       return response
```

A.2 Directory Structure

'~':
handlo

data:
hash_dir hashtags him.jpg phot.py snow_img.jpg storage_context test1.jpeg test2.jpeg

__pycache__:
context_ret.cpython-310.pyc llama_index.cpython-310.pyc

src:
app.py caption.py configurations context_ret.py hashtag_rag __init__.py __pycache__
requirements

vectorDB:
collection hashtags.db