4. ANALISA DAN PEMBAHASAN

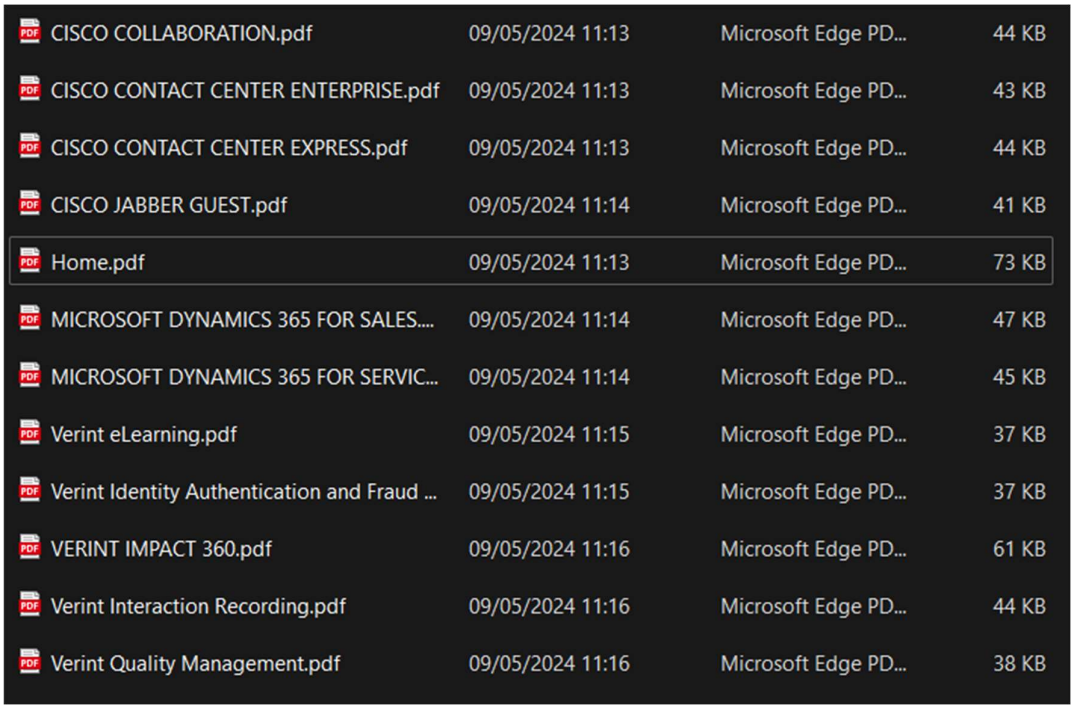
Bab ini membahas persiapan dan hasil implementasi sistem yang telah dibuat, sesuai dengan desain sistem. Pada bab ini, akan disajikan alur pengerjaan serta kode program yang digunakan beserta penjelasan terkait fungsionalitas dari setiap pengerjaan, bagian kode ,dan fitur yang ada. Berikut merupakan hasil implementasi yang telah digunakan.

4.1 Pengumpulan Data

Pengumpulan data adalah rangkaian proses yang paling pertama dilakukan dalam pengerjaan proyek. Kegiatan pengumpulan data meliputi pengumpulan data *text* dari *website* resmi PT. XYZ untuk pengujian *low volume use case*, dan pengumpulan data *text* berupa kumpulan buku dengan kuantitas sebanyak 5 buku dari *website* Otoritas Jasa Keuangan untuk pengujian *high volume use case.* Berikut adalah proses pengerjaan pengumpulan data, pada masing-masing *use case*:

a. *Low Volume Use Case*

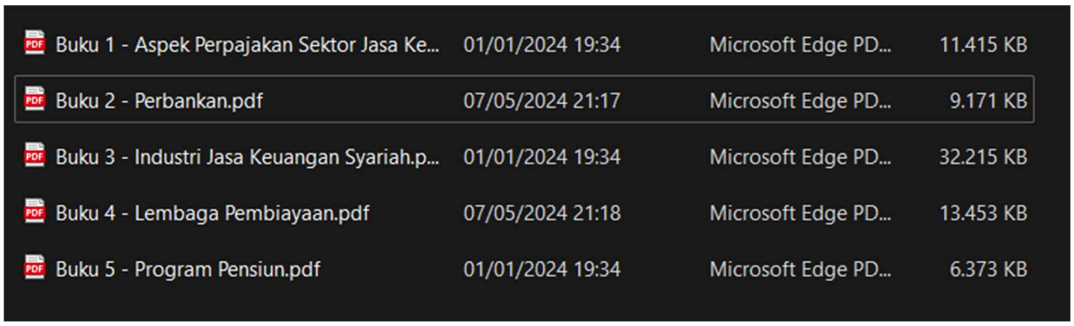
Pengumpulan data pada *low volume use case* dimulai dengan pengumpulan segala jenis bentuk *text* dari *website* PT. XYZ yang relevan untuk pengujian. Hasil pengumpulan data mencakup 12 *file*, masing-masing mewakili halaman web yang berisi penjelasan dan konteks yang cukup mendalam. File-file tersebut mencakup informasi tentang PT. XYZ, kumpulan kata-kata promosi perusahaan, dan layanan yang ditawarkan oleh PT. XYZ.



Gambar 4.1 Kumpulan *list file* yang akan digunakan untuk pengujian *low volume use case*

b. *High Volume Use Case*

Pengumpulan data pada *high volume use case* dimulai dengan pengumpulan 5 buku yang telah ditetapkan. Buku yang ada tersedia secara *open source* pada *website* Otoritas Jasa Keuangan (OJK). Kumpulan buku yang ada berjumlah sebanyak 1008 halaman, dan telah berbentuk dalam format *Portable Document Format (PDF)* sehingga dapat segera digunakan dalam tahap pengujian *high volume use case*.



Gambar 4.2 Kumpulan *list file* buku yang akan digunakan untuk pengujian *high volume use case*

4.2 *Data Cleaning*

Proses *data cleaning* dimulai setelah mengumpulkan seluruh *file* dan data yang dibutuhkan selama pengujian berlangsung. *Data cleaning* yang dilakukan dalam proyek berfokus pada data pengujian *low volume use case*, dimana data tersebut masih memperlukan berbagai tahap proses agar dapat digunakan dalam tahap pengujian. Beberapa tahap proses tersebut merupakan:

a. *Data Masking*

Data Masking adalah proses *data cleaning* yang dilakukan untuk melindungi privasi PT. XYZ. Proses ini mengganti setiap nama asli PT. XYZ yang terdapat pada data dengan tulisan PT. XYZ. Proses ini juga menghilangkan segala informasi sensitif yang terdapat pada *website* yang telah diambil.

Gambar 4. 3 Contoh *data masking* pada *file home.pdf*

b. *Quality Control*

Quality control adalah proses *data cleaning* yang dilakukan untuk meminimalisir adanya *typo* pada hasil olahan data, guna memaksimalkan pengujian pada *low volume use case*.

c. *Language Translation Process*

Language translation process, adalah salah satu proses *data cleaning*, yang dilakukan untuk mengubah penggunaan Bahasa Inggris yang ada pada beberapa penjelasan *webpage* pada *website* PT. XYZ. Hal ini dilakukan mengingat adanya beberapa dokumen penjelasan produk yang menggunakan Bahasa Inggris.

4.3 Implementasi Sistem *Chatbot*

Pada proyek ini, pengembangan dan implementasi sistem *chatbot* dibangun berdasarkan *PrivateGPT* yang dilisensikan di bawah *Apache License 2.0*. Detail lengkap lisensi dapat ditemukan di: *http://www.apache.org/licenses/LICENSE-2.0. PrivateGPT* adalah sebuah proyek *open source* yang dikembangkan oleh Martínez Toro, Gallego Vico, dan Orgaz (2023), yang memungkinkan pengguna dapat memanfaatkan model *LLM* secara lokal tanpa interaksi dengan server eksternal.

PrivateGPT memiliki beberapa variasi sistem yang tersedia pada *repository* yang telah disediakan. Setelah berdiskusi dengan pihak PT. XYZ, pengerjaan proyek ini akan didasarkan pada seri *primordial* dari *repository PrivateGPT*. Seri *primordial* merujuk pada iterasi awal dari *repository* tersebut, yang berfungsi sebagai fondasi dasar dengan beberapa fitur dasar seperti menjalankan model *LLM*, pemuatan model, pemrosesan teks, dan kemampuan pengembangan sistem tanya jawab dengan penggunaan basis *library LangChain.*

Seri *primordial* dipilih karena tahap fondasi ini memungkinkan fleksibilitas dalam melakukan modifikasi sistem agar dapat disesuaikan dengan kebutuhan PT. XYZ. Dengan menggunakan versi ini, proyek ini dapat memanfaatkan teknologi *LLM* secara efektif, serta memberikan kemudahan dalam menyesuaikan dan mengembangkan fitur sesuai keinginan pengguna. Berikut adalah langkah-langkah yang telah diambil untuk mengembangkan dan melakukan implementasi terhadap sistem *chatbot* yang telah dibuat:

4.3.1 Pemasangan *PrivateGPT* Versi *Primordial*

Pengerjaan sistem *chatbot* diawali dengan melakukan instalasi sistem *primordial* dari *PrivateGPT* sebagai fondasi awal. Berikut adalah proses pemasangan yang dilakukan:

a. *Download PrivateGPT.zip* dari *installer. Extract zip file* pada *path* yang diinginkan. Pengguna juga dapat melakukan instalasi dengan melakukan *git clone* dengan *command* berikut:

Segmen program 4.1 Proses *git clone PrivateGPT* versi *primordial*

```
git clone https://github.com/zylon-ai/private-gpt.git --branch primordial
--single-branch
```

b. Lakukan *environment setup* dengan membuka *command prompt* dan lakukan *change directory* menuju *path folder PrivateGPT*. Setelah itu pastikan isi *requirements.txt* adalah sebagai berikut:

Segmen program 4.2 Kumpulan *library* yang dibutuhkan untuk menjalankan program

```
langchain==0.0.274
gpt4all==1.0.8
chromadb==0.4.12
llama-cpp-python==0.1.81
urllib3==2.0.4
PyMuPDF==1.24.1
python-dotenv==1.0.0
unstructured==0.10.8
extract-msg==0.45.0
tabulate==0.9.0
pandoc==2.3
pypandoc==1.11
tqdm==4.66.1
sentence_transformers==2.2.2
```

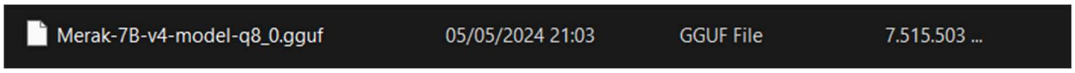
Setelah memastikan isi *requirements.txt* sama, *execute command* berikut:

Segmen program 4.3 Proses instalasi *requirements library* yang diperlukan

```
Cd change/to/path
pip3 install -r requirements.txt
```

c. Lakukan *download pre-trained Large Language Model (LLM)* yang diinginkan. Pada proyek ini, *LLM* yang akan digunakan adalah *Ichsan2895/Merak-7B-v4-GGUF*. Model Merak ini didapatkan melalui situs resmi *HuggingFace repository* berikut: https://huggingface.co/Ichsan2895/Merak-7B-v4-GGUF/blob/main/Merak-7B-v4-model-q8_0.gguf.

d. Setelah proses instalasi model *LLM* selesai, buat *folder* baru pada folder *PrivateGPT* dengan nama "*models*" dan letakkan model *LLM* Merak pada *folder* tersebut.



Gambar 4.4 *LLM* Merak yang akan digunakan pada *folder models*

e. Setelah proses instalasi model *LLM* selesai, perlu adanya proses penyesuaian pada *environment PrivateGPT*. Proses penyesuaian *environment* dilakukan untuk menyesuaikan spesifikasi yang diinginkan pada program. Berbagai spesifikasi yang dapat dimodifikasi terletak pada *file example.env* dengan *content* sebagai berikut:

Segmen program 4.4 Pengaturan pada *environment file* yang digunakan pada sistem

```
PERSIST_DIRECTORY=db
MODEL_TYPE=LlamaCpp
MODEL_PATH=models/Merak-7B-v4-model-q8_0.gguf
EMBEDDINGS_MODEL_NAME=cahya/bert-base-indonesian-522M
MODEL_N_CTX=2048
MODEL_N_BATCH=8
TARGET_SOURCE_CHUNKS=4
```

Dalam proyek ini, "*example.env*" akan diubah menjadi "*.env*". *File* ini dibuat sebagai *file* konfigurasi sistem yang ada dan akan digunakan dalam sistem *chatbot*. Konfigurasi yang ada pada *file* ini meliputi:

- *PERSIST_DIRECTORY*: memberi indikasi lokasi *vector database.* Pada *file* yang ada, *vector database* tersimpan pada sebuah *folder* bernama db.
- *MODEL_TYPE*: memberi indikasi tipe *LLM* yang digunakan. Dalam proyek ini, penulis menggunakan tipe *LlamaCpp.*
- *MODEL_PATH*: menghubungkan program dengan *LLM* dengan memberi *path* menuju *folder models*.
- *EMBEDDINGS_MODEL_NAME*: memberi indikasi *embedding model* yang akan digunakan. Dalam proyek ini, *embedding model* yang akan digunakan bernama *cahya/bert-base-indonesian-522M.*
- *MODEL_N_CTX*: memberi indikasi jumlah *token* (kata maupun sub-kata) yang dapat diproses oleh model *LLM*. Semakin besar nominal yang diberikan maka semakin bagus peforma yang dapat dicapai oleh model, mengingat model dapat melihat *token* dengan jumlah yang lebih banyak dapat menghasilkan *output* yang lebih baik

secara *contextual*. Namun perlu diingat, semakin besar nominal yang diberikan, semakin besar pula *computing power* yang dibutuhkan. Nominal yang besar dapat berdampak baik terutama pada sistem yang memiliki data *text* dalam jumlah besar.

- *MODEL_N_BATCH*: memberi indikasi jumlah *batch* (kelompok data) yang akan diproses secara bersamaan dalam satu iterasi saat *model LLM* melakukan *inference*. Semakin besar nilai yang diberi, maka semakin banyak data yang dapat diproses secara bersamaan sehingga *inference time* dapat menjadi lebih cepat. Namun apabila nominal yang diberikan semakin besar, maka *computing power* yang dibutuhkan pun juga akan semakin besar.
- *TARGET_SOURCE_CHUNKS*: memberi indikasi jumlah potongan (*chunks*) dari sumber dokumen yang akan digunakan dalam proses pencarian dan *text generation*. Dalam sistem *RAG*, dokumen sering kali dipecah menjadi beberapa *chunks* atau potongan kecil untuk memudahkan pencarian dan pemrosesan. Dalam proyek ini, nominal *chunks* diatur pada nominal 4 yang mengindikasikan bahwa sistem akan menggunakan hingga 4 *chunks* yang paling relevan untuk menjawab sebuah pertanyaan.

Dengan sistem yang ada, terdapat 2 jenis *LLM* dapat digunakan pada *chatbot*. Pertama adalah penggunaan *LLM* berbasis *Llama* dan *LLM* berbasis *GPT4ALL*. Hal tersebut dapat dilihat dari *example.env original* milik *PrivateGPT* yang menggunakan *LLM* berbasis *GPT4ALL* untuk implementasi yang dilakukan. *PrivateGPT* juga menggunakan *embedding model* yang berbeda dari yang digunakan sistem dalam proyek ini.

4.3.2 Pemanfaatan *GPU*

Pada dasarnya, sistem yang disediakan oleh *PrivateGPT* seri *primordial* tidak memanfaatkan penggunaan komponen *GPU* sama sekali. Hal ini dapat berdampak pada peforma *inference time*, mengingat sistem hanya dapat menggunakan kapasitas *CPU* yang ada. Maka dari itu untuk memastikan sistem dapat memanfaatkan kapasitas setiap *hardware* yang digunakan sebaik mungkin, fitur *GPU Acceleration* dapat digunakan untuk mencapai hal tersebut. Penambahan fitur *GPU Acceleration* dapat dilakukan dengan langkah berikut:

a. Dalam proyek ini, pemanfaatan *GPU* mengharuskan sebuah *operating system* memiliki sistem *CUDA* yang telah dikembangkan oleh *NVIDIA developer* (2024). apabila tidak, pengembang dapat melakukan instalasi dengan mengikuti panduan resmi cara instalasi sistem *CUDA* berikut: https://developer.nvidia.com/cuda-downloads. Setelah melakukan instalasi, pengembang dapat memastikan bahwa *GPU driver* dan sistem *CUDA* telah terpasang dengan baik dengan menjalankan *command nvidia-smi* dan *nvcc --version* pada terminal *command prompt.*

b. Hapus *llama-cpp-python* dan *install* kembali dengan comand berikut:

Segmen program 4.5 Instalasi *llama-cpp-python*

```
set LLAMA_CUBLAS=1
set CMAKE_ARGS=-DLLAMA_CUBLAS=on
set FORCE_CMAKE=1
pip install llama-cpp-python --no-cache-dir --verbose
```

c. Tambahkan penggunaan *GPU layer* pada *PrivateGPT*:

Segmen program 4.6 Penambahan *GPU layer* pada *model_type*

```
match model_type:
        case "LlamaCpp":
                llm = LlamaCpp(model_path=model_path, n_ctx=model_n_ctx,
callbacks=callbacks, verbose=True, n_gpu_layers=25)
        case "GPT4All":
                 llm = GPT4All(model=model_path, max_tokens=model_n_ctx,
backend='gptj', n_batch=model_n_batch, callbacks=callbacks, verbose=False)
        case _default:
            # raise exception if model_type is not supported
              raise Exception(f"Model type {model_type} is not supported.
Please choose one of the following: LlamaCpp, GPT4All")
```

4.3.3 Pengaturan *Chroma Vector Database*

Chroma vector database digunakan untuk mengelola penyimpanan dan pencarian dokumen berdasarkan *embedding* yang dimiliki oleh model *LLM*. Maka dari itu dalam sistem perlu dibuat sebuah pengaturan untuk melakukan inisialisasi sebuah *chroma vector database* sebagai media penyimpanan dan pengambilan kumpulan vektor dokumen.

Segmen program 4.7 Konfigurasi dan inisialisasi *chroma vector database*

```
import os
from dotenv import load_dotenv
from chromadb.config import Settings

load_dotenv()

# Define the folder for storing the database
PERSIST_DIRECTORY = os.environ.get('PERSIST_DIRECTORY')
if PERSIST_DIRECTORY is None:
    raise Exception("Please set the PERSIST_DIRECTORY environment variable")

# Define the Chroma settings
CHROMA_SETTINGS = Settings(
    persist_directory=PERSIST_DIRECTORY,
    anonymized_telemetry=False
)
```

Terdapat beberapa langkah yang diambil dalam proses konfigurasi dan inisialisasi *chroma vector database* sebagai berikut:

- *Load_dotenv*():
    - menggunakan *module dotenv* untuk memuat variabel yang ada pada *environment file.*
- *PERSIST_DIRECTORY*:
    - adalah sebuah variabel dari *environment file* yang menyimpan lokasi direktori tempat penyimpanan data vektor yang akan dihasilkan.

- *CHROMA_SETTINGS*:
    - mengkonfigurasi pengaturan untuk *chroma*, termasuk direktori penyimpanan dan opsi telemetri (fitur pengumpulan dan pengiriman data dari satu sistem menuju server pusat untuk pemantauan).

4.3.4 *Storing Task*

Dalam proyek ini*, storing task* adalah tahap penyimpanan data dokumen ke dalam *vector database*. Tahap ini sangat penting karena memastikan bahwa semua dokumen yang akan digunakan oleh sistem untuk pencarian informasi telah diproses dan disimpan dalam format yang efisien. Proses *storing task* melibatkan beberapa langkah kritis, mulai dari *imports* dan inisialisasi modul, penggunaan *custom document loader*, pemrosesan dokumen, hingga penambahan *embedding* vektor ke dalam *vector database*. Setiap tahap ini dirancang untuk memastikan bahwa dokumen dapat diakses dan diambil dengan cepat dan akurat oleh sistem *retrieval augmented generation (RAG)*.

Segmen program 4.8 Tahap *storing task*

```
import os
import glob
from typing import List
from dotenv import load_dotenv
from multiprocessing import Pool
from tqdm import tqdm
import streamlit as st

from langchain.document_loaders import (
    CSVLoader,
    EverNoteLoader,
    PyMuPDFLoader,
    TextLoader,
    UnstructuredEmailLoader,
    UnstructuredEPubLoader,
    UnstructuredHTMLLoader,
    UnstructuredMarkdownLoader,
    UnstructuredODTLoader,
```

```
    UnstructuredPowerPointLoader,
    UnstructuredWordDocumentLoader,
)

from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.docstore.document import Document

if not load_dotenv():
    st.write("Could not load .env file or it is empty. Please check if it
exists and is readable.")

from constants import CHROMA_SETTINGS
import chromadb
from chromadb.api import API

# Load environment variables with defaults
persist_directory = os.environ.get('PERSIST_DIRECTORY', 'chroma_persistence')
source_directory = os.environ.get('SOURCE_DIRECTORY', 'source_documents')
embeddings_model_name = os.environ.get('EMBEDDINGS_MODEL_NAME', 'all-MiniLM-
L6-v2')
chunk_size = int(os.environ.get('CHUNK_SIZE', 500))
chunk_overlap = int(os.environ.get('CHUNK_OVERLAP', 50))

# Ensure source_documents directory exists
if not os.path.exists(source_directory):
    os.makedirs(source_directory)

# Custom document loaders
class MyElmLoader(UnstructuredEmailLoader):
    """Wrapper to fallback to text/plain when default does not work"""

    def load(self) -> List[Document]:
        try:
            try:
                doc = super().load()
```

```
            except ValueError as e:
                if 'text/html content not found in email' in str(e):
                    self.unstructured_kwargs["content_source"] = "text/plain"
                    doc = super().load()
                else:
                    raise
        except Exception as e:
            raise type(e)(f"{self.file_path}: {e}") from e
        return doc

# Map file extensions to document loaders and their arguments
LOADER_MAPPING = {
    ".csv": (CSVLoader, {}),
    ".doc": (UnstructuredWordDocumentLoader, {}),
    ".docx": (UnstructuredWordDocumentLoader, {}),
    ".enex": (EverNoteLoader, {}),
    ".eml": (MyElmLoader, {}),
    ".epub": (UnstructuredEPubLoader, {}),
    ".html": (UnstructuredHTMLLoader, {}),
    ".md": (UnstructuredMarkdownLoader, {}),
    ".odt": (UnstructuredODTLoader, {}),
    ".pdf": (PyMuPDFLoader, {}),
    ".ppt": (UnstructuredPowerPointLoader, {}),
    ".pptx": (UnstructuredPowerPointLoader, {}),
    ".txt": (TextLoader, {"encoding": "utf8"}),
}

def load_single_document(file_path: str) -> List[Document]:
    ext = "." + file_path.rsplit(".", 1)[-1].lower()
    if ext in LOADER_MAPPING:
        loader_class, loader_args = LOADER_MAPPING[ext]
        loader = loader_class(file_path, **loader_args)
        return loader.load()
    raise ValueError(f"Unsupported file extension '{ext}'")

def load_documents(source_dir: str, ignored_files: List[str] = []) ->
List[Document]:
```

```
    all_files = []
    for ext in LOADER_MAPPING:
        all_files.extend(glob.glob(os.path.join(source_dir,
f"**/*{ext.lower()}"), recursive=True))
        all_files.extend(glob.glob(os.path.join(source_dir,
f"**/*{ext.upper()}"), recursive=True))
    filtered_files = [file_path for file_path in all_files if file_path not
in ignored_files]

    documents = []
    with Pool(processes=os.cpu_count()) as pool:
        with tqdm(total=len(filtered_files), desc='Loading new documents',
ncols=80) as pbar:
            for docs in pool.imap_unordered(load_single_document,
filtered_files):
                documents.extend(docs)
                pbar.update()
    return documents

def process_documents(ignored_files: List[str] = []) -> List[Document]:
    st.write(f"Loading documents from {source_directory}")
    documents = load_documents(source_directory, ignored_files)
    if not documents:
        st.write("No new documents to load")
        return []
    st.write(f"Loaded {len(documents)} new documents from
{source_directory}")
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size,
chunk_overlap=chunk_overlap)
    documents = text_splitter.split_documents(documents)
    st.write(f"Split into {len(documents)} chunks of text (max. {chunk_size}
tokens each)")
    return documents

def batch_chromadb_insertions(chroma_client: API, documents: List[Document]):
    max_batch_size = chroma_client.max_batch_size
    for i in range(0, len(documents), max_batch_size):
```

```
        yield documents[i:i + max_batch_size]

def does_vectorstore_exist(persist_directory: str, embeddings:
HuggingFaceEmbeddings) -> bool:
    db = Chroma(persist_directory=persist_directory,
embedding_function=embeddings)
    return bool(db.get()['documents'])

def main():
    embeddings = HuggingFaceEmbeddings(model_name=embeddings_model_name)
    chroma_client = chromadb.PersistentClient(settings=CHROMA_SETTINGS,
path=persist_directory)

    try:
        if does_vectorstore_exist(persist_directory, embeddings):
            st.write(f"Appending to existing vectorstore at
{persist_directory}")
            db = Chroma(persist_directory=persist_directory,
embedding_function=embeddings, client_settings=CHROMA_SETTINGS,
client=chroma_client)
            collection = db.get()
            documents = process_documents([metadata['source'] for metadata in
collection['metadatas']])
            if documents:
                st.write(f"Creating embeddings. May take some minutes...")
                for batch in batch_chromadb_insertions(chroma_client,
documents):
                    db.add_documents(batch)
        else:
            st.write("Creating new vectorstore")
            documents = process_documents()
            if documents:
                st.write(f"Creating embeddings. May take some minutes...")
                first_batch = next(batch_chromadb_insertions(chroma_client,
documents))
                db = Chroma.from_documents(first_batch, embeddings,
persist_directory=persist_directory, client_settings=CHROMA_SETTINGS,
```

```
client=chroma_client)
                for batch in batch_chromadb_insertions(chroma_client,
documents):
                    db.add_documents(batch)
        st.write(f"Process complete! You can now run your system to query
your documents")
    except Exception as e:
        st.error(f"An error occurred: {e}")

if __name__ == "__main__":
    main()
```

Berikut merupakan langkah-langkah yang diambil dalam tahap *storing task* :

- *Imports* dan inisialisasi modul:
    - Melakukan *import* seluruh *library* yang diperlukan.
- Memuat *environment file*:
    - Menggunakan *load_dotenv*() untuk memuat variabel-variabel dari *environment file*.
- Pemeriksaan *directory file*:
    - Memastikan keberadaan *directory file source_documents*. Apabila belum ada, *directory* tersebut akan dibuat.
- *Custom document loader*:
    - Menggunakan *MyElmLoader* untuk menangani pemrosesan dokumen *email*, dengan *fallback* ke *text/plain* jika *text/html* tidak ditemukan.
- *Mapping* ekstensi *file* ke *loader*:
    - Mendefinisikan *LOADER_MAPPING* yang memetakan ekstensi *file* ke *loader* yang sesuai.
- *Load single document*:
    - Memuat dokumen satu per satu berdasarkan ekstensi *file* dengan menggunakan *loader* yang sesuai. Apabila *format file* tidak sesuai dengan *loader* yang ada, fungsi ini akan mengeluarkan *error*.

- *Load documents*:
    - Memuat semua dokumen dari *file directory* menggunakan *multiprocessing* untuk efisiensi. Fungsi ini menggunakan *tqdm* untuk menampilkan *progress bar* selama pemuatan dokumen.
- *Process documents*:
    - Memproses dokumen dengan memecahnya menjadi *chunks* atau potongan dokumen yang lebih kecil menggunakan *RecursiveCharacterTextSplitter*.
- *Batch ChromaDB insertions*:
    - Mengelola *batch insertions* ke *chroma* untuk efisiensi penyimpanan.
- *Does vectorstore exist*:
    - Memeriksa apakah *vectorstore* sudah ada dengan memeriksa dokumen yang tersimpan.
- *Main*:
    - Menginisialisasi *embedding* dengan *HuggingFaceEmbeddings*.
    - Membuat *client chroma* dengan pengaturan *CHROMA_SETTINGS*.
    - Memeriksa keberadaan *vectorstore*; apabila belum ada, *vectorstore* akan dibuat.
    - Memproses dokumen untuk membuat *embedding vector*.
    - Menyimpan *embedding* dokumen ke dalam *chroma vector database* dalam *batch* untuk efisiensi.
    - Proses konversi *text* menjadi vektor dilakukan dengan *model embedding HuggingFaceEmbeddings.* Dokumen dipecah menjadi *chunks* yang lebih kecil dan setiap *chunks* dikonversi menjadi *embedding vector* sebelum disimpan ke dalam *chroma vector database*.

4.3.5 *Search Task*

Search task adalah bagian dari *chatbot* yang bertugas untuk mencari jawaban dari *vector database* dan menyerahkan hasil pencarian tersebut atau hasil *similarity search* kepada *model LLM* untuk memberi *model LLM* konteks pemahaman jawaban dari pertanyaan yang diberikan. Oleh karena itu *search task* memanfaatkan baik *embedding model* dan *model LLM* dalam upaya menjawab pertanyaan yang dilontarkan pengguna *chatbot*.

Segmen program 4.9 Tahap inisialisasi *embedding model, LLM,* dan *chroma vector database*

```
import os
import logging
from dotenv import load_dotenv
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import Chroma
from langchain.llms import LlamaCpp
import chromadb
from response_handler import get_response

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

load_dotenv()

embeddings_model_name = os.environ.get("EMBEDDINGS_MODEL_NAME", "all-MiniLM-
L6-v2")
persist_directory = os.environ.get('PERSIST_DIRECTORY', 'chroma_persistence')
model_type = os.environ.get('MODEL_TYPE', 'LlamaCpp')
model_path = os.environ.get('MODEL_PATH', 'models/llama-7b')
model_n_ctx = int(os.environ.get('MODEL_N_CTX', 2048))
model_n_batch = int(os.environ.get('MODEL_N_BATCH', 8))
target_source_chunks = int(os.environ.get('TARGET_SOURCE_CHUNKS', 4))

from constants import CHROMA_SETTINGS

def initialize_embeddings(model_name):
    try:
        embeddings = HuggingFaceEmbeddings(model_name=model_name)
        logger.info(f"Embeddings initialized for model: {model_name}")
        return embeddings
    except Exception as e:
        logger.error(f"Failed to initialize embeddings: {e}")
        raise
```

```
def initialize_llm(model_type, model_path, model_n_ctx, model_n_batch):
    try:
        if model_type == "LlamaCpp":
            llm = LlamaCpp(model_path=model_path, n_ctx=model_n_ctx,
verbose=True, n_gpu_layers=25)
        else:
            raise ValueError(f"Model type '{model_type}' is not supported.")
        logger.info(f"Model {model_type} loaded successfully from path:
{model_path}")
        return llm
    except Exception as e:
        logger.error(f"Failed to load the model: {e}")
        raise

if __name__ == "__main__":
    embeddings = initialize_embeddings(embeddings_model_name)
    llm = initialize_llm(model_type, model_path, model_n_ctx, model_n_batch)

    chroma_client = chromadb.PersistentClient(settings=CHROMA_SETTINGS,
path=persist_directory)
    db = Chroma(persist_directory=persist_directory,
embedding_function=embeddings, client_settings=CHROMA_SETTINGS,
client=chroma_client)
    retriever = db.as_retriever(search_kwargs={"k": target_source_chunks})

    query = "Apa itu PT. XYZ?"
    response, response_time = get_response(query, llm, retriever,
model_n_ctx, embeddings)
    print(f"Response: {response}")
    print(f"Response time: {response_time:.2f} seconds")
```

Berikut merupakan langkah-langkah yang diambil dalam tahap inisialisasi *model* dan *chroma vector database*:

- *Import* modul dan konfigurasi *logging*:
    - *Import* seluruh *library* yang diperlukan.
    - Lakukan *logging* dikonfigurasi untuk mencatat informasi dan kesalahan yang terjadi selama eksekusi.

- Memuat *environment variable*:
    - Menggunakan *dotenv* untuk memuat variabel pada *environment file.*
- Konfigurasi *chroma settings*:
    - Melakukan *import* pengaturan *chroma* dari *file constants*.
- Inisialisasi *embeddings*:
    - Menggunakan fungsi *initialize_embeddings* untuk menginisialisasi *embedding model* menggunakan *HuggingFaceEmbeddings.*
- Inisialisasi *LLM*:
    - Menggunakan *initialize_llm* untuk inisialisasi model *LLM* berdasarkan tipe *model* yang ditentukan.
- *Main*:
    - Menjalankan inisialisasi *embeddings* dan *model LLM*. Kemudian inisialisasi *chroma client* dan *retriever* dilakukan untuk menyiapkan sistem agar dapat mengambil dokumen yang relevan.

Segmen program 4.10 Tahap *response generation*

```
import time
import logging
from langchain.chains import RetrievalQA
from langchain.llms import LlamaCpp
from langchain.vectorstores import Chroma

logger = logging.getLogger(__name__)

def get_response(query, llm: LlamaCpp, retriever: Chroma, model_n_ctx: int,
embedding_model) -> tuple:
    start_time = time.time()
    try:
        qa = RetrievalQA.from_chain_type(llm=llm, chain_type="stuff",
retriever=retriever, return_source_documents=True)
        logger.info(f"Received query: {query}")
        if len(query) > model_n_ctx:
            query = query[:model_n_ctx]
```

```
            logger.info(f"Truncated query to {model_n_ctx} characters.")

        result = qa(query)
        response_time = time.time() - start_time

        response = result['result']

        logger.info("Generated response successfully.")
        return response, response_time
    except Exception as e:
        response_time = time.time() - start_time
        logger.error(f"Error generating response: {e}")
        return f"Error generating response: {e}", response_time
```

Berikut merupakan langkah-langkah yang diambil dalam tahap *response generation* dan *handler*:

- *Import* modul dan konfigurasi *logging*:
    - *Import* seluruh *library* yang diperlukan.
    - Lakukan *logging* dikonfigurasi untuk mencatat informasi dan kesalahan yang terjadi selama eksekusi.
- *Response generator*:
    - Menggunakan *get_response* untuk mengambil dokumen yang relevan menggunakan *retriever* dan menghasilkan respons dari *model LLM*.
    - Mencatat waktu yang dibutuhkan untuk menghasilkan respons.

4.3.6 *User Interface*

Pengerjaan proyek menggunakan desain *user interface* berbasis *Streamlit*. *User interface* yang dibuat merupakan sebuah *webpage* dengan segala keperluan dari sistem *chatbot* dan sistem *retrieval*. Beberapa fitur yang dapat diakses *via user interface* meliputi sarana *upload* dokumen, aktivasi *retrieval system*, *chatbox* untuk menampilkan *respon, input field* untuk membuat *query* pertanyaan dari pengguna, dan fungsi *clear* untuk mengosongkan *chatbox*.

Segmen program 4.11 *User interface* sistem *chatbot*

```
import streamlit as st
from privateGPT import initialize_embeddings, initialize_llm
from ingest import process_documents
from response_handler import get_response
from dotenv import load_dotenv
import os
import logging

load_dotenv()

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

st.title('PT. XYZ Customer Service')

if 'user_input' not in st.session_state:
    st.session_state['user_input'] = ''
if 'response' not in st.session_state:
    st.session_state['response'] = ''
if 'response_time' not in st.session_state:
    st.session_state['response_time'] = 0.0
if 'llm' not in st.session_state:
    st.session_state['llm'] = None
if 'embeddings' not in st.session_state:
    st.session_state['embeddings'] = None
if 'retriever' not in st.session_state:
    st.session_state['retriever'] = None
if 'reset' not in st.session_state:
    st.session_state['reset'] = False

def initialize_components():
    try:
        embeddings_model_name = os.environ.get("EMBEDDINGS_MODEL_NAME", "all-
MiniLM-L6-v2")
        model_type = os.environ.get('MODEL_TYPE', 'LlamaCpp')
```

```
        model_path = os.environ.get('MODEL_PATH', 'models/llama-7b')
        model_n_ctx = int(os.environ.get('MODEL_N_CTX', 2048))
        model_n_batch = int(os.environ.get('MODEL_N_BATCH', 8))
        target_source_chunks = int(os.environ.get('TARGET_SOURCE_CHUNKS', 4))
        persist_directory = os.environ.get('PERSIST_DIRECTORY',
'chroma_persistence')
        from constants import CHROMA_SETTINGS
        import chromadb
        from langchain.vectorstores import Chroma

        st.session_state['embeddings'] =
initialize_embeddings(embeddings_model_name)
        chroma_client = chromadb.PersistentClient(settings=CHROMA_SETTINGS,
path=persist_directory)
        db = Chroma(persist_directory=persist_directory,
embedding_function=st.session_state['embeddings'],
client_settings=CHROMA_SETTINGS, client=chroma_client)
        st.session_state['retriever'] = db.as_retriever(search_kwargs={"k":
target_source_chunks})
        st.session_state['llm'] = initialize_llm(model_type, model_path,
model_n_ctx, model_n_batch)
        st.session_state['model_n_ctx'] = model_n_ctx
    except Exception as e:
        st.error(f"Error initializing model or embeddings: {e}")
        st.stop()

uploaded_file = st.file_uploader("Choose a file", key='file_uploader')
if uploaded_file is not None:
    with open(os.path.join('source_documents', uploaded_file.name), 'wb') as
f:
        f.write(uploaded_file.getvalue())
    st.success('File uploaded successfully.')

if st.button('Process Files'):
    try:
        documents = process_documents()
        if documents:
```

```
            st.success('Documents processed successfully.')
        else:
            st.warning('No documents to process.')
    except Exception as e:
        st.error(f"Error processing documents: {e}")

chat_display = f"User: {st.session_state.get('user_input', '')}\nBot:
{st.session_state.get('response', '')}\n\nResponse Time:
{st.session_state.get('response_time', 0.0):.2f} seconds"

st.markdown("""
    <style>
    .chatbox {
        background-color: #0e1117;
        color: #f0f0f0;
        padding: 10px;
        border-radius: 5px;
        height: 200px;
        overflow-y: auto;
        white-space: pre-wrap;
        font-family: monospace;
    }
    .chatbox::selection {
        background: rgba(255, 255, 255, 0.2);
    }
    </style>
    """, unsafe_allow_html=True)

st.markdown(f'<div class="chatbox">{chat_display}</div>',
unsafe_allow_html=True)

user_input_temp = st.text_input("Enter your message:", key='user_input_temp')

if st.button('Submit'):
    if user_input_temp:
        if st.session_state['llm'] is None or st.session_state['embeddings']
is None or st.session_state['retriever'] is None:
```

```
            initialize_components()

        try:
            with st.spinner('Generating response...'):
                response, response_time = get_response(user_input_temp,
st.session_state['llm'], st.session_state['retriever'],
st.session_state['model_n_ctx'], st.session_state['embeddings'])
                st.session_state['response'] = response
                st.session_state['response_time'] = response_time
                st.session_state['user_input'] = user_input_temp
                st.session_state['reset'] = False
                st.rerun()
        except Exception as e:
            st.error(f"Error generating response: {e}")

if st.button('Clear'):
    st.session_state['response'] = ''
    st.session_state['response_time'] = 0.0
    st.session_state['user_input'] = ''
    st.session_state['reset'] = True
    st.rerun()
```

Berikut merupakan fitur yang disediakan pada *user interface* segmen program 4.11:

- *Import* modul dan konfigurasi *logging*:
    - Melakukan *import* semua *library* yang diperlukan.
    - Lakukan *logging* dikonfigurasi untuk mencatat informasi dan kesalahan yang terjadi selama eksekusi.
- Inisialisasi *Streamlit* dan *session variable*:
    - Menetapkan judul aplikasi.
    - Menginisialisasi *session variable*.
- Inisialisasi komponen:
    - Menggunakan *initialize_components* untuk menginisialisasikan *embeddings, LLM,* dan *retriever* jika belum diinisialisasi.
- *Upload file* dan *document processing*:
    - Menyediakan *interface* untuk *upload file* dan melakukan *document processing*.

- *Chat response*:
    - Menampilkan *chatbox* dan menerima *input* dari pengguna.
    - Menyediakan tombol *Submit* untuk mengirim *input* pengguna.
    - Menyediakan tombol *clear* untuk mengosongkan *chatbox.*

d. *Script* Evaluasi *Repetitive Based Question*

Segmen program 4.12 Perhitungan repetitive based question

```
from google.colab import files
import pandas as pd


# Unggah file terlebih dahulu
uploaded = files.upload()

def load_data(file_path):
    # Load the Excel file
    df = pd.read_excel(file_path, skiprows=3)

    # Clean and rename columns
    df.columns = ['No', 'Pertanyaan', 'Jawaban']
    df['Jawaban'] = df['Jawaban'].fillna('')
    df['Jumlah Kata'] = df['Jawaban'].apply(lambda x: len(x.split()))

    return df

def analyze_variability(df):
    # Analyze variability in word counts
    results = df.groupby('Pertanyaan')['Jumlah Kata'].describe()
    return results


def main(file_path):
    df = load_data(file_path)
    variability_results = analyze_variability(df)
    print(variability_results)


# Path to the uploaded file
file_path = '/content/Book1.xlsx'
main(file_path)
```

Berikut merupakan fitur yang tersedia dalam perhitungan *repetitive based question* pada segmen 4. 12:

- *Import* modul:
    - *From* google.colab *import files* digunakan untuk melakukan *load file* hasil *testing repetitive based question* yang telah dilakukan.

    - Modul *pandas* digunakan untuk melakukan akses pada *file excel* hasil *testing repetitive based question*.
- *Load data*:
    - *Load data* adalah *function* yang digunakan untuk melakukan *pre-proccessing* data pada hasil *testing repetitive based question*.
    - df['Jumlah Kata'] = df['Jawaban'].apply(lambda x: len(x.split())) digunakan dalam *code* untuk menambahkan kolom baru berisi jumlah kata dalam setiap jawaban.
- *Analyze variability*:
    - Melakukan kalkulasi dengan menghitung statistik deskriptif meliputi *mean, std, min,* dan *max.*
- *Main function*:
    - Melakukan *print* hasil kalkulasi.