rag_chatbot\core\embedding\embedding.py

```python
import os
import torch
import requests
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.embeddings.openai import OpenAIEmbedding
from transformers import AutoModel, AutoTokenizer
from ...setting import RAGSettings
from dotenv import load_dotenv


load_dotenv()


class LocalEmbedding:
    @staticmethod
    def set(setting: RAGSettings | None = None, **kwargs):
        setting = setting or RAGSettings()
        model_name = setting.ingestion.embed_llm
        if model_name != "text-embedding-ada-002":
            return HuggingFaceEmbedding(
                model=AutoModel.from_pretrained(
                    model_name,
                    torch_dtype=torch.float16,
                    trust_remote_code=True
                ),
                tokenizer=AutoTokenizer.from_pretrained(
                    model_name,
                    torch_dtype=torch.float16
                ),
                cache_folder=os.path.join(os.getcwd(), setting.ingestion.cache_folder),
                trust_remote_code=True,
                embed_batch_size=setting.ingestion.embed_batch_size
            )
        else:
            return OpenAIEmbedding()

    @staticmethod
    def pull(host: str, **kwargs):
        setting = RAGSettings()
        payload = {
            "name": setting.ingestion.embed_llm
        }
        return requests.post(f"http://{host}:11434/api/pull", json=payload, stream=True)

    @staticmethod
    def check_model_exist(host: str, **kwargs) -> bool:
        setting = RAGSettings()
        data = requests.get(f"http://{host}:11434/api/tags").json()
        list_model = [d["name"] for d in data["models"]]
```

```
        if setting.ingestion.embed_llm in list_model:
            return True
        return False
```

rag_chatbot\core\engine\engine.py

```
from llama_index.core.chat_engine import CondensePlusContextChatEngine, SimpleChatEngine
from llama_index.core.memory import ChatMemoryBuffer
from llama_index.core.llms.llm import LLM
from llama_index.core.schema import BaseNode
from typing import List
from .retriever import LocalRetriever
from ...setting import RAGSettings


class LocalChatEngine:
    def __init__(
        self,
        setting: RAGSettings | None = None,
        host: str = "host.docker.internal"
    ):
        super().__init__()
        self._setting = setting or RAGSettings()
        self._retriever = LocalRetriever(self._setting)
        self._host = host

    def set_engine(
        self,
        llm: LLM,
        nodes: List[BaseNode],
        language: str = "eng",
    ) -> CondensePlusContextChatEngine | SimpleChatEngine:

        # Normal chat engine
        if len(nodes) == 0:
            return SimpleChatEngine.from_defaults(
                llm=llm,
                memory=ChatMemoryBuffer(
                    token_limit=self._setting.ollama.chat_token_limit
                )
            )

        # Chat engine with documents
        retriever = self._retriever.get_retrievers(
            llm=llm,
            language=language,
            nodes=nodes
        )
        return CondensePlusContextChatEngine.from_defaults(
            retriever=retriever,
            llm=llm,
```

```
            memory=ChatMemoryBuffer(
                token_limit=self._setting.ollama.chat_token_limit
            )
        )
```

rag_chatbot\core\engine\retriever.py

```
from typing import List
from dotenv import load_dotenv
from llama_index.core.retrievers import (
    BaseRetriever,
    QueryFusionRetriever,
    VectorIndexRetriever,
    RouterRetriever
)
from llama_index.core.callbacks.base import CallbackManager
from llama_index.core.retrievers.fusion_retriever import FUSION_MODES
from llama_index.core.postprocessor import SentenceTransformerRerank
from llama_index.core.tools import RetrieverTool
from llama_index.core.selectors import LLMSingleSelector
from llama_index.core.schema import BaseNode, NodeWithScore, QueryBundle, IndexNode
from llama_index.core.llms.llm import LLM
from llama_index.retrievers.bm25 import BM25Retriever
from llama_index.core import Settings, VectorStoreIndex
from ..prompt import get_query_gen_prompt
from ...setting import RAGSettings

load_dotenv()


class TwoStageRetriever(QueryFusionRetriever):
    def __init__(
        self,
        retrievers: List[BaseRetriever],
        setting: RAGSettings | None = None,
        llm: str | None = None,
        query_gen_prompt: str | None = None,
        mode: FUSION_MODES = FUSION_MODES.SIMPLE,
        similarity_top_k: int = ...,
        num_queries: int = 4,
        use_async: bool = True,
        verbose: bool = False,
        callback_manager: CallbackManager | None = None,
        objects: List[IndexNode] | None = None,
        object_map: dict | None = None,
        retriever_weights: List[float] | None = None
    ) -> None:
        super().__init__(
            retrievers, llm, query_gen_prompt, mode, similarity_top_k, num_queries,
            use_async, verbose, callback_manager, objects, object_map, retriever_weights
        )
```

```
        self._setting = setting or RAGSettings()
        self._rerank_model = SentenceTransformerRerank(
            top_n=self._setting.retriever.top_k_rerank,
            model=self._setting.retriever.rerank_llm,
        )

    def _retrieve(self, query_bundle: QueryBundle) -> List[NodeWithScore]:
        queries: List[QueryBundle] = [query_bundle]
        if self.num_queries > 1:
            queries.extend(self._get_queries(query_bundle.query_str))

        if self.use_async:
            results = self._run_nested_async_queries(queries)
        else:
            results = self._run_sync_queries(queries)
        results = self._simple_fusion(results)
        return self._rerank_model.postprocess_nodes(results, query_bundle)

    async def _aretrieve(self, query_bundle: QueryBundle) -> List[NodeWithScore]:
        queries: List[QueryBundle] = [query_bundle]
        if self.num_queries > 1:
            queries.extend(self._get_queries(query_bundle.query_str))

        results = await self._run_async_queries(queries)
        results = self._simple_fusion(results)
        return self._rerank_model.postprocess_nodes(results, query_bundle)


class LocalRetriever:
    def __init__(
        self,
        setting: RAGSettings | None = None,
        host: str = "host.docker.internal"
    ):
        super().__init__()
        self._setting = setting or RAGSettings()
        self._host = host

    def _get_normal_retriever(
        self,
        vector_index: VectorStoreIndex,
        llm: LLM | None = None,
        language: str = "eng",
    ):
        llm = llm or Settings.llm
        return VectorIndexRetriever(
            index=vector_index,
            similarity_top_k=self._setting.retriever.similarity_top_k,
            embed_model=Settings.embed_model,
            verbose=True
```

```
        )

    def _get_hybrid_retriever(
        self,
        vector_index: VectorStoreIndex,
        llm: LLM | None = None,
        language: str = "eng",
        gen_query: bool = True
    ):
        # VECTOR INDEX RETRIEVER
        vector_retriever = VectorIndexRetriever(
            index=vector_index,
            similarity_top_k=self._setting.retriever.similarity_top_k,
            embed_model=Settings.embed_model,
            verbose=True
        )

        bm25_retriever = BM25Retriever.from_defaults(
            index=vector_index,
            similarity_top_k=self._setting.retriever.similarity_top_k,
            verbose=True
        )

        # FUSION RETRIEVER
        if gen_query:
            hybrid_retriever = QueryFusionRetriever(
                retrievers=[bm25_retriever, vector_retriever],
                retriever_weights=self._setting.retriever.retriever_weights,
                llm=llm,
                query_gen_prompt=get_query_gen_prompt(language),
                similarity_top_k=self._setting.retriever.top_k_rerank,
                num_queries=self._setting.retriever.num_queries,
                mode=self._setting.retriever.fusion_mode,
                verbose=True
            )
        else:
            hybrid_retriever = TwoStageRetriever(
                retrievers=[bm25_retriever, vector_retriever],
                retriever_weights=self._setting.retriever.retriever_weights,
                llm=llm,
                query_gen_prompt=None,
                similarity_top_k=self._setting.retriever.similarity_top_k,
                num_queries=1,
                mode=self._setting.retriever.fusion_mode,
                verbose=True
            )

        return hybrid_retriever

    def _get_router_retriever(
```

```
        self,
        vector_index: VectorStoreIndex,
        llm: LLM | None = None,
        language: str = "eng",
    ):
        fusion_tool = RetrieverTool.from_defaults(
            retriever=self._get_hybrid_retriever(
                vector_index, llm, language, gen_query=True
            ),
            description="Use this tool when the user's query is ambiguous or unclear.",
            name="Fusion Retriever with BM25 and Vector Retriever and LLM Query
Generation."
        )
        two_stage_tool = RetrieverTool.from_defaults(
            retriever=self._get_hybrid_retriever(
                vector_index, llm, language, gen_query=False
            ),
            description="Use this tool when the user's query is clear and unambiguous.",
            name="Two Stage Retriever with BM25 and Vector Retriever and LLM Rerank."
        )

        return RouterRetriever.from_defaults(
            selector=LLMSingleSelector.from_defaults(llm=llm),
            retriever_tools=[fusion_tool, two_stage_tool],
            llm=llm
        )

    def get_retrievers(
        self,
        nodes: List[BaseNode],
        llm: LLM | None = None,
        language: str = "eng",

    ):
        vector_index = VectorStoreIndex(nodes=nodes)
        if len(nodes) > self._setting.retriever.top_k_rerank:
            retriever = self._get_router_retriever(vector_index, llm, language)
        else:
            retriever = self._get_normal_retriever(vector_index, llm, language)

        return retriever
```

rag_chatbot\core\ingestion\ingestion.py

```
import re
import fitz
from llama_index.core import Document, Settings
from llama_index.core.schema import BaseNode
from llama_index.core.node_parser import SentenceSplitter
from dotenv import load_dotenv
from typing import Any, List
```

```
from tqdm import tqdm
from ...setting import RAGSettings

load_dotenv()


class LocalDataIngestion:
    def __init__(self, setting: RAGSettings | None = None) -> None:
        self._setting = setting or RAGSettings()
        self._node_store = {}
        self._ingested_file = []

    def _filter_text(self, text):
        # Define the regex pattern.
        pattern = r'[a-zA-Z0-9
\u00C0-\u01B0\u1EA0-\u1EF9`~!@#$%^&*()_\-+=\[\]{}|\\;:\'",.<>/?]+'
        matches = re.findall(pattern, text)
        # Join all matched substrings into a single string
        filtered_text = ' '.join(matches)
        # Normalize the text by removing extra whitespaces
        normalized_text = re.sub(r'\s+', ' ', filtered_text.strip())

        return normalized_text

    def store_nodes(
        self,
        input_files: list[str],
        embed_nodes: bool = True,
        embed_model: Any | None = None
    ) -> List[BaseNode]:
        return_nodes = []
        self._ingested_file = []
        if len(input_files) == 0:
            return return_nodes
        splitter = SentenceSplitter.from_defaults(
            chunk_size=self._setting.ingestion.chunk_size,
            chunk_overlap=self._setting.ingestion.chunk_overlap,
            paragraph_separator=self._setting.ingestion.paragraph_sep,
            secondary_chunking_regex=self._setting.ingestion.chunking_regex
        )
        if embed_nodes:
            Settings.embed_model = embed_model or Settings.embed_model
        for input_file in tqdm(input_files, desc="Ingesting data"):
            file_name = input_file.strip().split('/')[-1]
            self._ingested_file.append(file_name)
            if file_name in self._node_store:
                return_nodes.extend(self._node_store[file_name])
            else:
                document = fitz.open(input_file)
                all_text = ""
```

```
                for doc_idx, page in enumerate(document):
                    page_text = page.get_text("text")
                    page_text = self._filter_text(page_text)
                    all_text += " " + page_text
                document = Document(
                    text=all_text.strip(),
                    metadata={
                        "file_name": file_name,
                    }
                )

                nodes = splitter([document], show_progress=True)
                if embed_nodes:
                    nodes = Settings.embed_model(nodes, show_progress=True)
                self._node_store[file_name] = nodes
                return_nodes.extend(nodes)
        return return_nodes

    def reset(self):
        self._node_store = {}
        self._ingested_file = []

    def check_nodes_exist(self):
        return len(self._node_store.values()) > 0

    def get_all_nodes(self):
        return_nodes = []
        for nodes in self._node_store.values():
            return_nodes.extend(nodes)
        return return_nodes

    def get_ingested_nodes(self):
        return_nodes = []
        for file in self._ingested_file:
            return_nodes.extend(self._node_store[file])
        return return_nodes
```

rag_chatbot\core\model\model.py

```
from llama_index.llms.ollama import Ollama
from llama_index.llms.openai import OpenAI
from ...setting import RAGSettings
from dotenv import load_dotenv
import requests

load_dotenv()

class LocalRAGModel:
    def __init__(self) -> None:
        pass
```

```python
    @staticmethod
    def set(
        model_name: str = "llama3:8b-instruct-q8_0",
        system_prompt: str | None = None,
        host: str = "host.docker.internal",
        setting: RAGSettings | None = None
    ):
        setting = setting or RAGSettings()
        if model_name in ["gpt-3.5-turbo", "gpt-4", "gpt-4o", "gpt-4-turbo"]:
            return OpenAI(
                model=model_name,
                temperature=setting.ollama.temperature
            )
        else:
            settings_kwargs = {
                "tfs_z": setting.ollama.tfs_z,
                "top_k": setting.ollama.top_k,
                "top_p": setting.ollama.top_p,
                "repeat_last_n": setting.ollama.repeat_last_n,
                "repeat_penalty": setting.ollama.repeat_penalty,
            }
            return Ollama(
                model=model_name,
                system_prompt=system_prompt,
                base_url=f"http://{host}:{setting.ollama.port}",
                temperature=setting.ollama.temperature,
                context_window=setting.ollama.context_window,
                request_timeout=setting.ollama.request_timeout,
                additional_kwargs=settings_kwargs
            )

    @staticmethod
    def pull(host: str, model_name: str):
        setting = RAGSettings()
        payload = {
            "name": model_name
        }
        return requests.post(
            f"http://{host}:{setting.ollama.port}/api/pull",
            json=payload, stream=True
        )

    @staticmethod
    def check_model_exist(host: str, model_name: str) -> bool:
        setting = RAGSettings()
        data = requests.get(
            f"http://{host}:{setting.ollama.port}/api/tags"
        ).json()
        if data["models"] is None:
            return False
```

```
        list_model = [d["name"] for d in data["models"]]
        if model_name in list_model:
            return True
        return False
```

rag_chatbot\core\prompt\qa_prompt.py

```
def get_context_prompt(language: str) -> str:
    if language == "vi":
        return CONTEXT_PROMPT_VI
    return CONTEXT_PROMPT_EN


def get_system_prompt(language: str, is_rag_prompt: bool = True) -> str:
    if language == "vi":
        return SYSTEM_PROMPT_RAG_VI if is_rag_prompt else SYSTEM_PROMPT_VI
    return SYSTEM_PROMPT_RAG_EN if is_rag_prompt else SYSTEM_PROMPT_EN


SYSTEM_PROMPT_EN = """\
This is a chat between a user and an artificial intelligence assistant. \
The assistant gives helpful, detailed, and polite answers to the user's questions based
on the context. \
The assistant should also indicate when the answer cannot be found in the context."""

SYSTEM_PROMPT_RAG_EN = """\
This is a chat between a user and an artificial intelligence assistant. \
The assistant gives helpful, detailed, and polite answers to the user's questions based
on the context. \
The assistant should also indicate when the answer cannot be found in the context."""

CONTEXT_PROMPT_EN = """\
Here are the relevant documents for the context:

{context_str}

Instruction: Based on the above documents, provide a detailed answer for the user
question below. \
Answer 'don't know' if not present in the document."""

CONDENSED_CONTEXT_PROMPT_EN = """\
Given the following conversation between a user and an AI assistant and a follow up
question from user,
rephrase the follow up question to be a standalone question.

Chat History:
{chat_history}
Follow Up Input: {question}
Standalone question:\
"""
```

```
SYSTEM_PROMPT_VI = """\
Đây là một cuộc trò chuyện giữa người dùng và một trợ lí trí tuệ nhân tạo. \
Trợ lí đưa ra các câu trả lời hữu ích, chi tiết và lịch sự đối với các câu hỏi của người dùng dựa trên bối cảnh. \
Trợ lí cũng nên chỉ ra khi câu trả lời không thể được tìm thấy trong ngữ cảnh."""

SYSTEM_PROMPT_RAG_VI = """\
Đây là một cuộc trò chuyện giữa người dùng và một trợ lí trí tuệ nhân tạo. \
Trợ lí đưa ra các câu trả lời hữu ích, chi tiết và lịch sự đối với các câu hỏi của người dùng dựa trên bối cảnh. \
Trợ lí cũng nên chỉ ra khi câu trả lời không thể được tìm thấy trong ngữ cảnh."""

CONTEXT_PROMPT_VI = """\
Dưới đây là các tài liệu liên quan cho ngữ cảnh:

{context_str}

Hướng dẫn: Dựa trên các tài liệu trên, cung cấp một câu trả lời chi tiết cho câu hỏi của người dùng dưới đây. \
Trả lời 'không biết' nếu không có trong tài liệu."""

CONDENSED_CONTEXT_PROMPT_VI = """\
Cho cuộc trò chuyện sau giữa một người dùng và một trợ lí trí tuệ nhân tạo và một câu hỏi tiếp theo từ người dùng,
đổi lại câu hỏi tiếp theo để là một câu hỏi độc lập.

Lịch sử Trò chuyện:
{chat_history}
Đầu vào Tiếp Theo: {question}
Câu hỏi độc lập:\
"""
```

rag_chatbot\core\prompt\query_gen_prompt.py

```
from llama_index.core import PromptTemplate


def get_query_gen_prompt(language: str):
    if language == "vi":
        return query_gen_prompt_vi
    return query_gen_prompt_en


query_gen_prompt_vi = PromptTemplate(
    "Bạn là một người tạo truy vấn tìm kiếm tài năng, cam kết cung cấp các truy vấn tìm kiếm chính xác và liên quan, ngắn gọn, cụ thể và không mơ hồ.\n"
    "Tạo ra {num_queries} truy vấn tìm kiếm độc đáo và đa dạng, mỗi truy vấn trên một dòng, liên quan đến truy vấn đầu vào sau đây:\n"
    "### Truy vấn Gốc: {query}\n"
    "### Vui lòng cung cấp các truy vấn tìm kiếm mà:\n"
    "- Liên quan đến truy vấn gốc\n"
```

```
    "- Được xác định rõ ràng và cụ thể\n"
    "- Không mơ hồ và không thể hiểu sai\n"
    "- Hữu ích để lấy kết quả tìm kiếm chính xác và liên quan\n"
    "### Các Truy Vấn Được Tạo Ra:\n"
)

query_gen_prompt_en = PromptTemplate(
    "You are a skilled search query generator, dedicated to providing accurate and
relevant search queries that are concise, specific, and unambiguous.\n"
    "Generate {num_queries} unique and diverse search queries, one on each line, related
to the following input query:\n"
    "### Original Query: {query}\n"
    "### Please provide search queries that are:\n"
    "- Relevant to the original query\n"
    "- Well-defined and specific\n"
    "- Free of ambiguity and vagueness\n"
    "- Useful for retrieving accurate and relevant search results\n"
    "### Generated Queries:\n"
)
```

rag_chatbot\core\prompt\select_prompt.py

```
def get_single_select_prompt(language: str):
    if language == "vi":
        return single_select_prompt_vi
    return single_select_prompt_en


single_select_prompt_en = (
    "Some choices are given below. It is provided in a numbered list "
    "(1 to {num_choices}), "
    "where each item in the list corresponds to a summary.\n"
    "---------------------\n"
    "{context_list}"
    "\n---------------------\n"
    "Using only the choices above and not prior knowledge, return "
    "ONE AND ONLY ONE choice that is most relevant to the query: '{query_str}'\n"
)

single_select_prompt_vi = (
    "Dưới đây là một số lựa chọn được đưa ra, được cung cấp trong một danh sách có số thứ
tự "
    "(từ 1 đến {num_choices}), "
    "trong đó mỗi mục trong danh sách tương ứng với một tóm tắt.\n"
    "---------------------\n"
    "{context_list}"
    "\n---------------------\n"
    "Chỉ sử dụng các lựa chọn ở trên và không dùng kiến thức trước đó, hãy chọn "
    "1 và chỉ 1 lựa chọn mà liên quan nhất đến câu truy vấn: '{query_str}'\n"
```

```
)
```

rag_chatbot\core\vector_store\vector_store.py

```python
from llama_index.core import VectorStoreIndex
from dotenv import load_dotenv
from ...setting import RAGSettings

load_dotenv()


class LocalVectorStore:
    def __init__(
        self,
        host: str = "host.docker.internal",
        setting: RAGSettings | None = None,
    ) -> None:
        # TODO
        # CHROMA VECTOR STORE
        self._setting = setting or RAGSettings()

    def get_index(self, nodes):
        if len(nodes) == 0:
            return None
        index = VectorStoreIndex(nodes=nodes)
        return index
```

rag_chatbot\eval\qa_generator.py

```python
import random
import re
import os
import uuid
from typing import List
from tqdm import tqdm
from llama_index.core.llms.utils import LLM
from llama_index.core.schema import MetadataMode, TextNode
from llama_index.core.storage.docstore import DocumentStore
from llama_index.core.evaluation import EmbeddingQAFinetuneDataset
from ..core.model import LocalRAGModel
from ..core.embedding import LocalEmbedding
from ..core.ingestion import LocalDataIngestion
from ..setting import RAGSettings


DEFAULT_QA_GENERATE_PROMPT_TMPL = """\
Context information is below.

---------------------
{context_str}
---------------------
```

```
Given the context information and not prior knowledge.
generate only questions based on the below query.

You are a Teacher/ Professor. Your task is to setup \
{num_questions_per_chunk} questions for an upcoming \
quiz/examination. The questions should be diverse in nature \
across the document. Restrict the questions to the context information provided. \
Only provide the questions, not the answers.\"
"""


# generate queries as a convenience function
def generate_question_context_pairs(
    nodes: List[TextNode],
    llm: LLM,
    qa_generate_prompt_tmpl: str = DEFAULT_QA_GENERATE_PROMPT_TMPL,
    num_questions_per_chunk: int = 2,
) -> EmbeddingQAFinetuneDataset:
    """Generate examples given a set of nodes."""
    node_dict = {
        node.node_id: node.get_content(metadata_mode=MetadataMode.NONE)
        for node in nodes
    }

    queries = {}
    relevant_docs = {}
    for node_id, text in tqdm(node_dict.items()):
        query = qa_generate_prompt_tmpl.format(
            context_str=text, num_questions_per_chunk=num_questions_per_chunk
        )
        response = llm.complete(query)

        result = str(response).strip().split("\n")
        questions = [
            re.sub(r"^\d+[\).\s]", "", question).strip()
            for question in result
        ]
        questions = [question for question in questions if len(question) > 0]

        for question in questions:
            question_id = str(uuid.uuid4())
            queries[question_id] = question
            relevant_docs[question_id] = [node_id]

    # construct dataset
    return EmbeddingQAFinetuneDataset(
        queries=queries, corpus=node_dict, relevant_docs=relevant_docs
    )
```

```python
class QAGenerator:
    def __init__(
        self,
        embed_model: str | None = None,
        llm: str | None = None,
        host: str = "host.docker.internal",
    ) -> None:
        setting = RAGSettings()
        setting.ingestion.embed_llm = embed_model or setting.ingestion.embed_llm
        self._embed_model = LocalEmbedding.set(setting)
        self._llm = LocalRAGModel.set(model_name=llm or setting.ollama.llm, host=host)
        self._ingestion = LocalDataIngestion()

    def generate(
        self,
        input_files: list[str],
        output_dir: str = "val_dataset",
        max_nodes: int = 100,
        num_questions_per_chunk=2,
    ) -> None:
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)

        if os.path.exists(os.path.join(output_dir, "docstore.json")):
            print("Docstore already exist! Skip ingestion.")


        nodes = self._ingestion.store_nodes(input_files, embed_nodes=True)
        random.shuffle(nodes)
        dataset = generate_question_context_pairs(
            nodes=nodes[:max_nodes],
            llm=self._llm,
            num_questions_per_chunk=num_questions_per_chunk,
        )

        # save dataset
        dataset.save_json(os.path.join(output_dir, "dataset.json"))

        # save nodes
        docstore = DocumentStore()
        docstore.add_documents(nodes)
        docstore.persist(persist_path=os.path.join(output_dir, "docstore.json"))
```

rag_chatbot\setting\setting.py

```python
from pydantic import BaseModel, Field
from typing import List


class OllamaSettings(BaseModel):
    llm: str = Field(
```

```
        default="llama3:8b-instruct-q8_0", description="LLM model"
    )
    keep_alive: str = Field(
        default="1h", description="Keep alive time for the server"
    )
    tfs_z: float = Field(
        default=1.0, description="TFS normalization factor"
    )
    top_k: int = Field(
        default=40, description="Top k sampling"
    )
    top_p: float = Field(
        default=0.9, description="Top p sampling"
    )
    repeat_last_n: int = Field(
        default=64, description="Repeat last n tokens"
    )
    repeat_penalty: float = Field(
        default=1.1, description="Repeat penalty"
    )
    request_timeout: float = Field(
        default=300, description="Request timeout"
    )
    port: int = Field(
        default=11434, description="Port number"
    )
    context_window: int = Field(
        default=8000, description="Context window size"
    )
    temperature: float = Field(
        default=0.1, description="Temperature"
    )
    chat_token_limit: int = Field(
        default=4000, description="Chat memory limit"
    )


class RetrieverSettings(BaseModel):
    num_queries: int = Field(
        default=5, description="Number of generated queries"
    )
    similarity_top_k: int = Field(
        default=20, description="Top k documents"
    )
    retriever_weights: List[float] = Field(
        default=[0.4, 0.6], description="Weights for retriever"
    )
    top_k_rerank: int = Field(
        default=6, description="Top k rerank"
    )
```

```
    rerank_llm: str = Field(
        default="BAAI/bge-reranker-large", description="Rerank LLM model"
    )
    fusion_mode: str = Field(
        default="dist_based_score", description="Fusion mode"
    )


class IngestionSettings(BaseModel):
    embed_llm: str = Field(
        default="BAAI/bge-large-en-v1.5", description="Embedding LLM model"
    )
    embed_batch_size: int = Field(
        default=8, description="Embedding batch size"
    )
    cache_folder: str = Field(
        default="data/huggingface", description="Cache folder"
    )
    chunk_size: int = Field(
        default=512, description="Document chunk size"
    )
    chunk_overlap: int = Field(
        default=32, description="Document chunk overlap"
    )
    chunking_regex: str = Field(
        default="[^,.;。？！]+[,.;。？！]?", description="Chunking regex"
    )
    paragraph_sep: str = Field(
        default="\n \n", description="Paragraph separator"
    )
    num_workers: int = Field(
        default=0, description="Number of workers"
    )


class StorageSettings(BaseModel):
    persist_dir_chroma: str = Field(
        default="data/chroma", description="Chroma directory"
    )
    persist_dir_storage: str = Field(
        default="data/storage", description="Storage directory"
    )
    collection_name: str = Field(
        default="collection", description="Collection name"
    )
    port: int = Field(
        default=8000, description="Port number"
    )
```

```
class RAGSettings(BaseModel):
    ollama: OllamaSettings = OllamaSettings()
    retriever: RetrieverSettings = RetrieverSettings()
    ingestion: IngestionSettings = IngestionSettings()
    storage: StorageSettings = StorageSettings()
```

rag_chatbot\ui\ui.py

```
import os
import shutil
import json
import sys
import time
import gradio as gr
from dataclasses import dataclass
from typing import ClassVar
from llama_index.core.chat_engine.types import StreamingAgentChatResponse
from .theme import JS_LIGHT_THEME, CSS
from ..pipeline import LocalRAGPipeline
from ..logger import Logger


@dataclass
class DefaultElement:
    DEFAULT_MESSAGE: ClassVar[dict] = {"text": ""}
    DEFAULT_MODEL: str = ""
    DEFAULT_HISTORY: ClassVar[list] = []
    DEFAULT_DOCUMENT: ClassVar[list] = []

    HELLO_MESSAGE: str = "Hi 👋, how can I help you today?"
    SET_MODEL_MESSAGE: str = "You need to choose LLM model 🤖 first!"
    EMPTY_MESSAGE: str = "You need to enter your message!"
    DEFAULT_STATUS: str = "Ready!"
    CONFIRM_PULL_MODEL_STATUS: str = "Confirm Pull Model!"
    PULL_MODEL_SCUCCESS_STATUS: str = "Pulling model 🤖 completed!"
    PULL_MODEL_FAIL_STATUS: str = "Pulling model 🤖 failed!"
    MODEL_NOT_EXIST_STATUS: str = "Model doesn't exist!"
    PROCESS_DOCUMENT_SUCCESS_STATUS: str = "Processing documents 📄 completed!"
    PROCESS_DOCUMENT_EMPTY_STATUS: str = "Empty documents!"
    ANSWERING_STATUS: str = "Answering!"
    COMPLETED_STATUS: str = "Completed!"


class LLMResponse:
    def __init__(self) -> None:
        pass

    def _yield_string(self, message: str):
        for i in range(len(message)):
            time.sleep(0.01)
```

```
            yield (
                DefaultElement.DEFAULT_MESSAGE,
                [[None, message[: i + 1]]],
                DefaultElement.DEFAULT_STATUS,
            )

    def welcome(self):
        yield from self._yield_string(DefaultElement.HELLO_MESSAGE)

    def set_model(self):
        yield from self._yield_string(DefaultElement.SET_MODEL_MESSAGE)

    def empty_message(self):
        yield from self._yield_string(DefaultElement.EMPTY_MESSAGE)

    def stream_response(
        self,
        message: str,
        history: list[list[str]],
        response: StreamingAgentChatResponse,
    ):
        answer = []
        for text in response.response_gen:
            answer.append(text)
            yield (
                DefaultElement.DEFAULT_MESSAGE,
                history + [[message, "".join(answer)]],
                DefaultElement.ANSWERING_STATUS,
            )
        yield (
            DefaultElement.DEFAULT_MESSAGE,
            history + [[message, "".join(answer)]],
            DefaultElement.COMPLETED_STATUS,
        )


class LocalChatbotUI:
    def __init__(
        self,
        pipeline: LocalRAGPipeline,
        logger: Logger,
        host: str = "host.docker.internal",
        data_dir: str = "data/data",
        avatar_images: list[str] = ["./assets/user.png", "./assets/bot.png"],
    ):
        self._pipeline = pipeline
        self._logger = logger
        self._host = host
        self._data_dir = os.path.join(os.getcwd(), data_dir)
        if not os.path.exists(self._data_dir):
```

```
            os.makedirs(self._data_dir, exist_ok=True)
        self._avatar_images = [
            os.path.join(os.getcwd(), image) for image in avatar_images
        ]
        self._variant = "panel"
        self._llm_response = LLMResponse()

    def _get_respone(
        self,
        chat_mode: str,
        message: dict[str, str],
        chatbot: list[list[str, str]],
        progress=gr.Progress(track_tqdm=True),
    ):
        if self._pipeline.get_model_name() in [None, ""]:
            for m in self._llm_response.set_model():
                yield m
        elif message["text"] in [None, ""]:
            for m in self._llm_response.empty_message():
                yield m
        else:
            console = sys.stdout
            sys.stdout = self._logger
            response = self._pipeline.query(chat_mode, message["text"], chatbot)
            for m in self._llm_response.stream_response(
                message["text"], chatbot, response
            ):
                yield m
            sys.stdout = console

    def _get_confirm_pull_model(self, model: str):
        if (model in ["gpt-3.5-turbo", "gpt-4"]) or (self._pipeline.check_exist(model)):
            self._change_model(model)
            return (
                gr.update(visible=False),
                gr.update(visible=False),
                DefaultElement.DEFAULT_STATUS,
            )
        return (
            gr.update(visible=True),
            gr.update(visible=True),
            DefaultElement.CONFIRM_PULL_MODEL_STATUS,
        )

    def _pull_model(self, model: str, progress=gr.Progress(track_tqdm=True)):
        if (model not in ["gpt-3.5-turbo", "gpt-4"]) and not (
            self._pipeline.check_exist(model)
        ):
            response = self._pipeline.pull_model(model)
            if response.status_code == 200:
```

```
            gr.Info(f"Pulling {model}!")
            for data in response.iter_lines(chunk_size=1):
                data = json.loads(data)
                if "completed" in data.keys() and "total" in data.keys():
                    progress(data["completed"] / data["total"], desc="Downloading")
                else:
                    progress(0.0)
        else:
            gr.Warning(f"Model {model} doesn't exist!")
            return (
                DefaultElement.DEFAULT_MESSAGE,
                DefaultElement.DEFAULT_HISTORY,
                DefaultElement.PULL_MODEL_FAIL_STATUS,
                DefaultElement.DEFAULT_MODEL,
            )

    return (
        DefaultElement.DEFAULT_MESSAGE,
        DefaultElement.DEFAULT_HISTORY,
        DefaultElement.PULL_MODEL_SCUCCESS_STATUS,
        model,
    )

def _change_model(self, model: str):
    if model not in [None, ""]:
        self._pipeline.set_model_name(model)
        self._pipeline.set_model()
        self._pipeline.set_engine()
        gr.Info(f"Change model to {model}!")
    return DefaultElement.DEFAULT_STATUS

def _upload_document(self, document: list[str], list_files: list[str] | dict):
    if document in [None, []]:
        if isinstance(list_files, list):
            return (list_files, DefaultElement.DEFAULT_DOCUMENT)
        else:
            if list_files.get("files", None):
                return list_files.get("files")
            return document
    else:
        if isinstance(list_files, list):
            return (document + list_files, DefaultElement.DEFAULT_DOCUMENT)
        else:
            if list_files.get("files", None):
                return document + list_files.get("files")
            return document

def _reset_document(self):
    self._pipeline.reset_documents()
    gr.Info("Reset all documents!")
```

```
        return (
            DefaultElement.DEFAULT_DOCUMENT,
            gr.update(visible=False),
            gr.update(visible=False),
        )

    def _show_document_btn(self, document: list[str]):
        visible = False if document in [None, []] else True
        return (gr.update(visible=visible), gr.update(visible=visible))

    def _processing_document(
        self, document: list[str], progress=gr.Progress(track_tqdm=True)
    ):
        document = document or []
        if self._host == "host.docker.internal":
            input_files = []
            for file_path in document:
                dest = os.path.join(self._data_dir, file_path.split("/")[-1])
                shutil.move(src=file_path, dst=dest)
                input_files.append(dest)
            self._pipeline.store_nodes(input_files=input_files)
        else:
            self._pipeline.store_nodes(input_files=document)
        self._pipeline.set_chat_mode()
        gr.Info("Processing Completed!")
        return (self._pipeline.get_system_prompt(), DefaultElement.COMPLETED_STATUS)

    def _change_system_prompt(self, sys_prompt: str):
        self._pipeline.set_system_prompt(sys_prompt)
        self._pipeline.set_chat_mode()
        gr.Info("System prompt updated!")

    def _change_language(self, language: str):
        self._pipeline.set_language(language)
        self._pipeline.set_chat_mode()
        gr.Info(f"Change language to {language}")

    def _undo_chat(self, history: list[list[str, str]]):
        if len(history) > 0:
            history.pop(-1)
            return history
        return DefaultElement.DEFAULT_HISTORY

    def _reset_chat(self):
        self._pipeline.reset_conversation()
        gr.Info("Reset chat!")
        return (
            DefaultElement.DEFAULT_MESSAGE,
            DefaultElement.DEFAULT_HISTORY,
            DefaultElement.DEFAULT_DOCUMENT,
```

```
            DefaultElement.DEFAULT_STATUS,
        )

    def _clear_chat(self):
        self._pipeline.clear_conversation()
        gr.Info("Clear chat!")
        return (
            DefaultElement.DEFAULT_MESSAGE,
            DefaultElement.DEFAULT_HISTORY,
            DefaultElement.DEFAULT_STATUS,
        )

    def _show_hide_setting(self, state):
        state = not state
        label = "Hide Setting" if state else "Show Setting"
        return (label, gr.update(visible=state), state)

    def _welcome(self):
        for m in self._llm_response.welcome():
            yield m

    def build(self):
        with gr.Blocks(
            theme=gr.themes.Soft(primary_hue="slate"),
            js=JS_LIGHT_THEME,
            css=CSS,
        ) as demo:
            gr.Markdown("## Local RAG Chatbot 🤖")
            with gr.Tab("Interface"):
                sidebar_state = gr.State(True)
                with gr.Row(variant=self._variant, equal_height=False):
                    with gr.Column(
                        variant=self._variant, scale=10, visible=sidebar_state.value
                    ) as setting:
                        with gr.Column():
                            status = gr.Textbox(
                                label="Status", value="Ready!", interactive=False
                            )
                            language = gr.Radio(
                                label="Language",
                                choices=["vi", "eng"],
                                value="eng",
                                interactive=True,
                            )
                            model = gr.Dropdown(
                                label="Choose Model:",
                                choices=[
                                    "llama3.1:8b-instruct-q8_0",
                                ],
                                value=None,
```

```
                    interactive=True,
                    allow_custom_value=True,
                )
                with gr.Row():
                    pull_btn = gr.Button(
                        value="Pull Model", visible=False, min_width=50
                    )
                    cancel_btn = gr.Button(
                        value="Cancel", visible=False, min_width=50
                    )

                documents = gr.Files(
                    label="Add Documents",
                    value=[],
                    file_types=[".txt", ".pdf", ".csv"],
                    file_count="multiple",
                    height=150,
                    interactive=True,
                )
                with gr.Row():
                    upload_doc_btn = gr.UploadButton(
                        label="Upload",
                        value=[],
                        file_types=[".txt", ".pdf", ".csv"],
                        file_count="multiple",
                        min_width=20,
                        visible=False,
                    )
                    reset_doc_btn = gr.Button(
                        "Reset", min_width=20, visible=False
                    )

            with gr.Column(scale=30, variant=self._variant):
                chatbot = gr.Chatbot(
                    layout="bubble",
                    value=[],
                    height=550,
                    scale=2,
                    show_copy_button=True,
                    bubble_full_width=False,
                    avatar_images=self._avatar_images,
                )

                with gr.Row(variant=self._variant):
                    chat_mode = gr.Dropdown(
                        choices=["chat", "QA"],
                        value="QA",
                        min_width=50,
                        show_label=False,
                        interactive=True,
```

```
                        allow_custom_value=False,
                    )
                    message = gr.MultimodalTextbox(
                        value=DefaultElement.DEFAULT_MESSAGE,
                        placeholder="Enter you message:",
                        file_types=[".txt", ".pdf", ".csv"],
                        show_label=False,
                        scale=6,
                        lines=1,
                    )
                with gr.Row(variant=self._variant):
                    ui_btn = gr.Button(
                        value="Hide Setting"
                        if sidebar_state.value
                        else "Show Setting",
                        min_width=20,
                    )
                    undo_btn = gr.Button(value="Undo", min_width=20)
                    clear_btn = gr.Button(value="Clear", min_width=20)
                    reset_btn = gr.Button(value="Reset", min_width=20)

        with gr.Tab("Setting"):
            with gr.Row(variant=self._variant, equal_height=False):
                with gr.Column():
                    system_prompt = gr.Textbox(
                        label="System Prompt",
                        value=self._pipeline.get_system_prompt(),
                        interactive=True,
                        lines=10,
                        max_lines=50,
                    )
                    sys_prompt_btn = gr.Button(value="Set System Prompt")

        with gr.Tab("Output"):
            with gr.Row(variant=self._variant):
                log = gr.Code(
                    label="", language="markdown", interactive=False, lines=30
                )
                demo.load(
                    self._logger.read_logs,
                    outputs=[log],
                    every=1,
                    show_progress="hidden",
                    # scroll_to_output=True,
                )

        clear_btn.click(self._clear_chat, outputs=[message, chatbot, status])
        cancel_btn.click(
            lambda: (gr.update(visible=False), gr.update(visible=False), None),
            outputs=[pull_btn, cancel_btn, model],
```

```
        )
        undo_btn.click(self._undo_chat, inputs=[chatbot], outputs=[chatbot])
        reset_btn.click(
            self._reset_chat, outputs=[message, chatbot, documents, status]
        )
        pull_btn.click(
            lambda: (gr.update(visible=False), gr.update(visible=False)),
            outputs=[pull_btn, cancel_btn],
        ).then(
            self._pull_model,
            inputs=[model],
            outputs=[message, chatbot, status, model],
        ).then(self._change_model, inputs=[model], outputs=[status])
        message.submit(
            self._upload_document, inputs=[documents, message], outputs=[documents]
        ).then(
            self._get_respone,
            inputs=[chat_mode, message, chatbot],
            outputs=[message, chatbot, status],
        )
        language.change(self._change_language, inputs=[language])
        model.change(
            self._get_confirm_pull_model,
            inputs=[model],
            outputs=[pull_btn, cancel_btn, status],
        )
        documents.change(
            self._processing_document,
            inputs=[documents],
            outputs=[system_prompt, status],
        ).then(
            self._show_document_btn,
            inputs=[documents],
            outputs=[upload_doc_btn, reset_doc_btn],
        )

        sys_prompt_btn.click(self._change_system_prompt, inputs=[system_prompt])
        ui_btn.click(
            self._show_hide_setting,
            inputs=[sidebar_state],
            outputs=[ui_btn, setting, sidebar_state],
        )
        upload_doc_btn.upload(
            self._upload_document,
            inputs=[documents, upload_doc_btn],
            outputs=[documents, upload_doc_btn],
        )
        reset_doc_btn.click(
            self._reset_document, outputs=[documents, upload_doc_btn, reset_doc_btn]
        )
```

```
            demo.load(self._welcome, outputs=[message, chatbot, status])

        return demo
```

rag_chatbot\logger.py

```
import os
import sys
import re


class Logger:
    def __init__(self, filename):

        self.filename = os.path.join(os.getcwd(), filename)
        self.terminal = sys.stdout
        self.reset_logs()
        self.log = open(self.filename, "w")
        self.flush()

    def write(self, message):
        self.terminal.write(message)
        self.log.write(message)

    def flush(self):
        self.terminal.flush()
        self.log.flush()

    def isatty(self):
        return False

    def reset_logs(self):
        with open(self.filename, 'w') as file:
            file.truncate(0)

    def read_logs(self):
        sys.stdout.flush()

        # Read the entire content of the log file
        with open(self.filename, "r") as f:
            log_content = f.readlines()

        # Filter out lines containing null characters
        log_content = [line for line in log_content if '\x00' not in line]

        # Define the regex pattern for the progress bar
        progress_pattern = re.compile(r'\[.*\] \d+\.\d+%')

        # Find lines matching the progress bar pattern
        progress_lines = [line for line in log_content if
```

```
                        progress_pattern.search(line) and " - Completed!\n" not in
line]

        # If there are multiple progress bars, keep only the last one in recent_lines
        if progress_lines:
            valid_content = [line for line in log_content if line not in progress_lines]
            if log_content[-1] == progress_lines[-1]:
                valid_content.append(progress_lines[-1].strip("\n"))
        else:
            valid_content = log_content

        # Get the latest 30 lines
        recent_lines = valid_content[-300:]

        # Return the joined recent lines
        return ''.join(recent_lines)
```

rag_chatbot\ollama.py

```
import asyncio
import threading
import socket


def run_ollama_server():
    async def run_process(cmd):
        print('>>> starting', *cmd)
        process = await asyncio.create_subprocess_exec(
            *cmd,
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE,
            # env={**os.environ, 'OLLAMA_NUM_PARALLEL': '8', 'OLLAMA_MAX_LOADED_MODELS':
'1'}
        )

        # define an async pipe function
        async def pipe(lines):
            async for line in lines:
                print(line.decode().strip())

            await asyncio.gather(
                pipe(process.stdout),
                pipe(process.stderr),
            )

        # call it
        await asyncio.gather(pipe(process.stdout), pipe(process.stderr))

    async def start_ollama_serve():
        await run_process(['ollama', 'serve'])
```

```
    def run_async_in_thread(loop, coro):
        asyncio.set_event_loop(loop)
        loop.run_until_complete(coro)
        loop.close()

    # Create a new event loop that will run in a new thread
    new_loop = asyncio.new_event_loop()

    # Start ollama serve in a separate thread so the cell won't block execution
    thread = threading.Thread(target=run_async_in_thread, args=(new_loop,
start_ollama_serve()))
    thread.start()


def is_port_open(port):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        try:
            s.connect(('localhost', port))
            return True
        except ConnectionRefusedError:
            return False
```

rag_chatbot\pipeline.py

```
from .core import (
    LocalChatEngine,
    LocalDataIngestion,
    LocalRAGModel,
    LocalEmbedding,
    LocalVectorStore,
    get_system_prompt,
)
from llama_index.core import Settings
from llama_index.core.chat_engine.types import StreamingAgentChatResponse
from llama_index.core.prompts import ChatMessage, MessageRole


class LocalRAGPipeline:
    def __init__(self, host: str = "host.docker.internal") -> None:
        self._host = host
        self._language = "eng"
        self._model_name = ""
        self._system_prompt = get_system_prompt("eng", is_rag_prompt=False)
        self._engine = LocalChatEngine(host=host)
        self._default_model = LocalRAGModel.set(self._model_name, host=host)
        self._query_engine = None
        self._ingestion = LocalDataIngestion()
        self._vector_store = LocalVectorStore(host=host)
        Settings.llm = LocalRAGModel.set(host=host)
        Settings.embed_model = LocalEmbedding.set(host=host)
```

```
def get_model_name(self):
    return self._model_name

def set_model_name(self, model_name: str):
    self._model_name = model_name

def get_language(self):
    return self._language

def set_language(self, language: str):
    self._language = language

def get_system_prompt(self):
    return self._system_prompt

def set_system_prompt(self, system_prompt: str | None = None):
    self._system_prompt = system_prompt or get_system_prompt(
        language=self._language, is_rag_prompt=self._ingestion.check_nodes_exist()
    )

def set_model(self):
    Settings.llm = LocalRAGModel.set(
        model_name=self._model_name,
        system_prompt=self._system_prompt,
        host=self._host,
    )
    self._default_model = Settings.llm

def reset_engine(self):
    self._query_engine = self._engine.set_engine(
        llm=self._default_model, nodes=[], language=self._language
    )

def reset_documents(self):
    self._ingestion.reset()

def clear_conversation(self):
    self._query_engine.reset()

def reset_conversation(self):
    self.reset_engine()
    self.set_system_prompt(
        get_system_prompt(language=self._language, is_rag_prompt=False)
    )

def set_embed_model(self, model_name: str):
    Settings.embed_model = LocalEmbedding.set(model_name, self._host)

def pull_model(self, model_name: str):
```

```
        return LocalRAGModel.pull(self._host, model_name)

    def pull_embed_model(self, model_name: str):
        return LocalEmbedding.pull(self._host, model_name)

    def check_exist(self, model_name: str) -> bool:
        return LocalRAGModel.check_model_exist(self._host, model_name)

    def check_exist_embed(self, model_name: str) -> bool:
        return LocalEmbedding.check_model_exist(self._host, model_name)

    def store_nodes(self, input_files: list[str] = None) -> None:
        self._ingestion.store_nodes(input_files=input_files)

    def set_chat_mode(self, system_prompt: str | None = None):
        self.set_language(self._language)
        self.set_system_prompt(system_prompt)
        self.set_model()
        self.set_engine()

    def set_engine(self):
        self._query_engine = self._engine.set_engine(
            llm=self._default_model,
            nodes=self._ingestion.get_ingested_nodes(),
            language=self._language,
        )

    def get_history(self, chatbot: list[list[str]]):
        history = []
        for chat in chatbot:
            if chat[0]:
                history.append(ChatMessage(role=MessageRole.USER, content=chat[0]))
                history.append(ChatMessage(role=MessageRole.ASSISTANT, content=chat[1]))
        return history

    def query(
        self, mode: str, message: str, chatbot: list[list[str]]
    ) -> StreamingAgentChatResponse:
        if mode == "chat":
            history = self.get_history(chatbot)
            return self._query_engine.stream_chat(message, history)
        else:
            self._query_engine.reset()
            return self._query_engine.stream_chat(message)
```