**Documentation**

**document\_reader.py**

**Purpose:**  
Handles reading and extracting text from a wide range of document formats, including DOCX, PDF, CSV, and Excel files (XLSX, XLS, XLSM).

**Key Features & Functions:**

* read\_document(file\_path): Main method for dispatching file reading based on extension.
* \_read\_pdf(file\_path): Uses PyPDF2 to extract text from PDF files, page by page.
* \_read\_docx(file\_path): Converts DOCX to PDF via docx2pdf, then reuses \_read\_pdf for text extraction to handle pages numbers.
* \_read\_tabular(file\_path): Parses CSVs and Excel files into chunks of readable text. Supports multiple sheets for Excel files.

**Notable Design Choices:**

* DOCX files are converted to PDF before reading to ensure handling page numbers.
* Tabular files are paginated into readable blocks (approx. 100 rows per “page”).
* Text are extracted from Tables.

-----------------------------------------------------------------------------------------------------------------------------

**publication\_chunker.py**

**Purpose:**  
Splits long publication documents into smaller, manageable text chunks using token count as a guide. This helps in preprocessing for summarization, translation, or embedding tasks.

**Key Features & Functions:**

* chunk\_publication(file\_path, max\_tokens):  
  Reads the document and splits it into token-based chunks, preserving metadata like page number and chunk index.
* chunk\_text(file\_path, max\_tokens):  
  Returns pure text chunks (no metadata), useful for streamlined summarization or LLM input.
* get\_token\_count(text):  
  Utility function for counting tokens using the tiktoken tokenizer (cl100k\_base).

**Notable Design Choices:**

* Integrates directly with the DocumentReader for seamless document loading.
* Uses a consistent tokenizer across the entire pipeline to ensure compatibility with LLM token limits.

-----------------------------------------------------------------------------------------------------------------------------

**vector\_db.py**

**Purpose:**  
Manages the creation, storage, and querying of a vector database for document embeddings. The database uses FAISS for fast similarity search and is designed to handle chunks of documents with metadata.

**Key Features & Functions:**

* **\_load\_db(self)**: Loads an existing FAISS index and its associated metadata from disk, if they exist. If the database does not exist, initializes empty structures.
* **\_save\_db(self)**: Saves the FAISS index and metadata to disk after updating the database.
* **generate\_embeddings(self, chunks: List[Dict])**: Generates text embeddings for a list of document chunks using the SentenceTransformer model. Tracks processing time and token count for performance metrics.
* **add\_to\_db(self, chunks: List[Dict])**: Adds document chunks and their embeddings to the FAISS index. If the index is empty, it initializes it with the correct dimension based on the embeddings.
* **get\_db\_size(self) -> int**: Returns the number of entries currently in the database (i.e., the number of document embeddings).

**Notable Design Choices:**

* **FAISS for Efficient Search**: The database uses FAISS (IndexFlatL2) for high-performance similarity search on document embeddings, enabling quick retrieval of relevant chunks.
* **Embedding Generation**: The nomic-ai/nomic-embed-text-v2-moe model is used to generate embeddings for text chunks, providing a semantic representation of the content for efficient search.
* **Metadata Storage**: Alongside the embeddings, metadata (source, page number, chunk number, and text) is stored, ensuring that each chunk can be traced back to its origin.

-----------------------------------------------------------------------------------------------------------------------------

**rag\_qa.py**

**Purpose:**  
Implements a Retrieval-Augmented Generation (RAG) system for question-answering using a vector database (FAISS) and a quantized Llama 3.1 model. The system retrieves relevant chunks of information and generates answers based on that context.

**Key Features & Functions:**

* **\_get\_relevant\_chunks(self, question: str, top\_k: int = 2)**: Retrieves the top-k most relevant document chunks from the vector database using FAISS. The chunks are selected based on the similarity of their embeddings to the query.
* **answer\_question(self, question: str)**: Accepts a question, retrieves relevant document chunks, and generates an answer using the Llama 3.1 model. The conversation history and performance metrics are also tracked.

**Notable Design Choices:**

* **Conversation History**: The system keeps track of the last two user-assistant message pairs to maintain context and improve answer relevance.
* **Chunk-based Retrieval**: Document chunks are retrieved from the vector database to provide context for generating answers, ensuring that the model works with up-to-date information.
* **Model Management**: The system uses the LLMManager for interacting with the Llama 3.1 model, abstracting away the complexity of LLM integration.

-----------------------------------------------------------------------------------------------------------------------------

**llm\_manager.py**

**Purpose:**  
Manages operations related to Large Language Models (LLMs), including model loading, tokenization, response generation, and translation. It supports both causal (text generation) and sequence-to-sequence (translation) models.

**Key Features & Functions:**

* **generate\_response(self, prompt: str, ...)**: Generates a text response based on a prompt using the causal language model. Supports parameters like max\_new\_tokens, temperature, and top\_p to control generation behavior.
* **generate\_chat\_response(self, messages: List[Dict[str, str]], ...)**: Generates a response using a chat format, accepting a list of message dictionaries with 'role' and 'content'. Useful for interactive or conversational AI systems.
* **generate\_translation(self, text: str, target\_lang\_code: str, ...)**: Translates text to a target language using a seq2seq model like NLLB. Requires the model to be initialized with the type 'seq2seq' for translation tasks.

**Notable Design Choices:**

* **Quantization for Efficiency**: The manager uses the BitsAndBytesConfig for quantizing models, optimizing memory usage and computation while maintaining performance.
* **Model Flexibility**: Supports both causal and seq2seq models, making it versatile for both text generation and translation tasks.
* **Device Handling**: Automatically selects GPU or CPU based on availability, enabling efficient model inference.

-----------------------------------------------------------------------------------------------------------------------------

**publication\_translator.py**

**Purpose:**  
Handles translation of publication documents into English or Arabic using the NLLB model. It preserves document structure across different formats (PDF, DOCX) and measures performance metrics like token usage and translation time.

**Key Features & Functions:**

* **translate\_document(file\_path, target\_language)**  
  Determines the file type, chunks content if necessary and translates the document to the specified language using the NLLB model.
* **\_translate\_docx(file\_path, target\_lang\_code)**  
  Preserves paragraph and table structure while translating DOCX files.
* **\_translate\_text(text, target\_lang\_code)**  
  Translates raw text using the LLMManager and logs performance metrics.
* **\_translate\_pdf\_via\_docx(pdf\_path, target\_lang\_code)**  
  Converts PDF to DOCX for translation, then back to PDF, maintaining layout fidelity.
* **save\_translated\_document(translated\_content, output\_path)**  
  Saves the translated output as DOCX or PDF, handling structure and cleanup of temporary files.

**Techniques Supported:**

* **Direct Translation:**  
  For raw text or DOCX files, translated content is preserved with formatting and structure.
* **PDF Round-Trip Translation:**  
  Converts PDF → DOCX → translated DOCX → PDF to preserve formatting while enabling translation.

**Notable Design Choices:**

* **Model-Agnostic Layer:**  
  Integrates the LLMManager to support any future changes in underlying translation models (currently using facebook/nllb-200-distilled-600M).
* **Flexible Format Handling:**  
  Supports .pdf, and .docx inputs with dynamic processing strategies for each.
* **Structure Preservation:**  
  Uses docx and pdf2docx to retain tables, paragraphs, and layout integrity in translations.

-----------------------------------------------------------------------------------------------------------------------------

**publication\_summarizer.py**

**Purpose:**  
Handles summarization of long documents using different techniques (extractive, abstractive, hybrid). Using **Llama 3.1** for robust and context-aware document processing. and evaluates the summaries using ROUGE scores.

**Key Features & Functions:**

* summarize\_document(file\_path, technique):  
  Loads and chunks the document, then summarizes each chunk using a specified method.
* evaluate\_summary(reference\_summary, generated\_summary):  
  Uses ROUGE-1, ROUGE-2, and ROUGE-L to compare a generated summary to a reference.
* save\_summary(summary, output\_path):  
  Exports the final summary into a .docx file.

**Techniques Supported:**

* **Extractive:** Selects and copies the most important original sentences.
* **Abstractive:** Rewrites the content in new words.
* **Hybrid:** Mixes both methods for a comprehensive yet concise result.

**Notable Design Choices:**

* Modular prompt generation for each technique.
* Seamless integration with the LLMManager, PerformanceMetrics, and PublicationChunker for end-to-end functionality.

-----------------------------------------------------------------------------------------------------------------------------

**performance\_metrics.py**

**Purpose:**  
This class tracks and calculates performance metrics for NLP operations, particularly those involving text chunk processing and embedding generation.

**Key Features & Functions:**

* **reset(self)**: Resets all performance metrics to their initial values. Metrics include tokens processed, total time spent, and tokens per second.
* **start\_tracking(self)**: Starts tracking the total processing time by recording the current time.
* **add\_chunk\_metrics(self, tokens: int, processing\_time: float)**: Adds metrics for a processed chunk, including the number of tokens in the chunk and the processing time it took to handle that chunk. It also updates the tokens per second metric.
* **stop\_tracking(self)**: Stops tracking the processing time and calculates the final time elapsed.
* **get\_metrics(self) -> Dict**: Returns the current performance metrics as a dictionary, excluding the start time.
* **print\_summary(self)**: Prints a summary of the performance metrics, including the total tokens processed, total time taken, and tokens per second.

**Notable Design Choices:**

* **Metric Tracking**: The class tracks essential performance indicators like the number of tokens processed, total processing time, and tokens per second to gauge the efficiency of the NLP operations.
* **Granular Metrics**: By updating metrics for each chunk processed, the system provides a detailed view of how different parts of the workload contribute to overall performance.