# Technical Documentation for Arogo AI Assistant

## Table of Contents



---

## Overview

The Arogo AI Assistant is a multi-functional AI system developed as part of the Arogo AI LLM Engineer Intern Assignment. It integrates several large language models (LLMs) and NLP modules to provide functionalities such as:

- Text Summarization
- Sentiment Analysis
- Named Entity Recognition (NER)
- Question Answering
- Code Generation and Code Review
- Retrieval-Augmented Generation (RAG) for document search and context augmentation
- Content Moderation to prevent harmful or biased outputs

The assistant is accessible via a Streamlit-based user interface, supports multi-turn conversations, and allows dynamic switching between various LLM providers (OpenAI, Gemini, HuggingFace).

---

# System Architecture

The system follows a modular architecture, making it easy to add or modify individual components. The key layers include:

1. **LLM Abstraction Layer:**
   Provides a unified interface (`LLMWrapper`) for interacting with different LLMs (OpenAI, Gemini, and HuggingFace).

2. **NLP Task Modules:**
   Separate modules for each NLP functionality such as summarization, sentiment analysis, NER, question answering, code generation, and code review.

3. **RAG & Vector Store:**
   A retrieval module built using FAISS and Sentence Transformers to index and search document content. This augments LLM responses with context from uploaded documents.

4. **Content Moderation:**
   Implements text classification using models like `martin-ha/toxic-comment-model` (with optional fallback to Gemini moderation) to filter harmful or biased outputs.

5. **Caching & Logging:**

   - **Caching:** Uses a custom cache manager to store responses and reduce redundant LLM calls.
   - **Logging:** Captures metrics (response time, prompt lengths, errors) in a log file, facilitating monitoring and debugging.

6. **Multi-Turn Context Management:**
   A context manager (using a deque) that maintains conversation history to support multi-turn dialogues.

---

# Modules and Components

## LLM Integration

- **LLMWrapper:**
  - Abstracts different LLM providers.
  - Manages content moderation, caching, logging, and context integration.
  - Providers supported: OpenAI, Gemini, and (optionally) HuggingFace.

## NLP Tasks

Each functionality is implemented in a dedicated module:

- **Summarization:** (`summarization.py`) Generates concise summaries.
- **Sentiment Analysis:** (`sentiment_analysis.py`) Classifies text sentiment.
- **Named Entity Recognition (NER):** (`ner.py`) Extracts entities from text.
- **Question Answering:** (`question_answering.py`) Answers questions using provided context.
- **Code Generation & Code Review:** (`code_generation.py`, `code_review.py`) Generate code snippets and review them.

## Retrieval-Augmented Generation (RAG)

- **VectorStore:**
  - Utilizes FAISS and SentenceTransformer ("all-MiniLM-L6-v2") to encode and index document content.
  - Supports various file formats (TXT, PDF, CSV, JSON) via dedicated ingestion methods.
  - Provides semantic search for relevant documents to augment LLM prompts.

## Content Moderation

- **ContentModerator:**
  - Uses a toxic text classifier (e.g., `martin-ha/toxic-comment-model` or alternatives).
  - Implements chunking to process long texts and falls back to Gemini moderation if necessary.
  - Ensures harmful or biased outputs are filtered before responses are generated.

## Caching and Logging

- **CacheManager:**
  - Caches LLM responses based on prompt hash to reduce latency and API costs.

- **AILogger:** (`logger.py`)
    - Logs request and response metrics (timestamp, provider, prompt length, response time, errors).
    - Uses OS-independent paths to store logs in a designated logs folder.

### Multi-Turn Context Management

- **ContextManager:**
    - Maintains conversation history using a fixed-size deque.
    - Allows the assistant to provide contextually relevant responses over multiple turns.

---

# Data Flow and Processing

1. **Input:**
    - Users interact via a Streamlit UI.
    - Inputs include plain text, documents (TXT, PDF, CSV, JSON), and questions with context.
2. **Processing:**
    - For chat tasks, user input is processed by the LLMWrapper, which first checks for toxicity, applies caching, and then uses the selected LLM to generate a response.
    - RAG integrates relevant document excerpts into the prompt to enhance responses.
    - Multi-turn context is maintained to ensure continuity.
3. **Output:**
    - The assistant outputs responses in the UI.
    - Evaluation results can be generated using pre-defined test cases, and performance metrics are logged for analysis.

---

# Installation & Setup

1. **Prerequisites:**
    - Python 3.8+

Install dependencies:
pip install -r requirements.txt

    -
2. **Environment Variables:**

Create a `.env` file in the project root with:
OPENAI_API_KEY=your_openai_api_key
GEMINI_API_KEY=your_gemini_api_key

    -
3. **Running the Application:**

Start the Streamlit app:
streamlit run src/interface/app.py

    -
4. **Document Upload:**
    - For RAG, upload files via the sidebar in the Chat tab.

---

# Evaluation & Testing

- **Evaluation Framework:**
    - Implemented in `src/evaluation/evaluator.py` using ROUGE for summarization, code execution for code generation, and exact match for question answering.
    - Sample test cases are provided in JSON format and can be uploaded in the Evaluation tab.
- **Test Suite:**
    - Unit tests are provided in the `tests/` directory (`test_app.py`, `test_evaluation.py`, `test_nlp_tasks.py`) covering core functionalities.

---

# Known Issues & Future Improvements

## Known Issues

- **Token Length Warnings:**
  Some models (especially Hugging Face) generate warnings if inputs exceed token limits.
- **Clustering Warnings:**
  FAISS clustering issues if the training data is insufficient.
- **Multiple Logger Instances:**
  Repeated instantiation of `AILogger` might lead to duplicate log entries.

## Future Improvements

- **Enhanced UI Features:**
  - Real-time token usage monitoring.
  - Chat history search and detailed logging within the UI.
- **Robust Error Handling:**
  - Improved exception handling for file uploads and LLM calls.
- **Scalability:**
  - Moving the vector store to a persistent database.
  - Optimizing caching for high-volume usage.
- **Advanced Moderation:**
  - Integration with alternative or custom moderation models for more nuanced content filtering.

---

# System Requirements & Considerations*

- **Memory & Compute:**
  - **RAM:** At least 16 GB recommended, especially when loading large models even with quantization.
  - **GPU:** A dedicated GPU will significantly speed up inference, though the model can run on CPU, it will be slower.
- **Storage:** Ensure sufficient disk space for model files and caching.
- **Environment:** The model is designed to run in an environment with PyTorch, Transformers, and BitsAndBytes installed.

*for Running hugging face model locally

---

# Sample Input Examples for UI Operations

## General Chat

- **Input:**
  "Hello, how are you today?"

---

## Summarization

- **Input:**
  "Climate change is a long-term shift in temperatures and weather patterns. While natural factors play a role, since the 1800s, human activities such as burning fossil fuels have been the main driver of climate change. This results in phenomena like rising sea levels, extreme weather events, and shifts in biodiversity."

---

## Sentiment Analysis

- **Input (Positive):**
  "I absolutely love this product! It has exceeded all my expectations."
- 
- **Input (Negative):**
  "This is the worst service I have ever experienced."

---

## Named Entity Recognition (NER)

- **Input:**
  "Barack Obama served as the President of the United States and lives in Washington, D.C."

---

## Question Answering

- **Context:**
  "The Eiffel Tower is an iconic landmark located in Paris, France, and was constructed in 1889."
- **Question:**
  "Where is the Eiffel Tower located?"

---

## Code Generation

- **Input:**
  "Write a Python function to add two numbers."

---

## Code Review

**Input:**

```
def add(a, b):

  return a + b
```

- 

---

## Retrieval-Augmented Generation (RAG)

- **Operation:**
  Upload a document (e.g., a text file containing an article on climate change) through the sidebar.
- **Chat Query:**
  "Summarize the uploaded document."
- **Expected Output:**
  A summary that captures the key points of the document, with the answer augmented by the retrieved context from the document.

---

## Evaluation

**Test Cases (via JSON Upload):**

Use a JSON file with sample test cases similar to the example below:

```
{
 "summarization": [
  {
   "input": "A long article about climate change...",
   "generated": "Climate change impacts global temperatures.",
   "reference": "Global warming is caused by human activities."
  }
 ],
 "code_generation": [
  {
   "input": "Write Python code to add two numbers",
   "generated": "def add(a, b): return a+b",
   "test_cases": [
    {"assertion": "add(2,3)", "expected": 5},
    {"assertion": "add(-1,1)", "expected": 0}
   ]
  }
 ],
 "question_answering": [
  {
   "context": "The Eiffel Tower is in Paris.",
   "question": "Where is the Eiffel Tower located?",
```

```
    "generated": "The Eiffel Tower is in Paris.",
    "reference": "Paris"
  }
 ]
}
```


---

## Conclusion

The Arogo AI Assistant demonstrates a comprehensive integration of LLMs with various NLP functionalities, retrieval augmentation, and content moderation, all wrapped within a user-friendly Streamlit interface. This documentation serves as a guide to understand the system's architecture, functionality, and potential areas for improvement. Future iterations can focus on enhancing scalability, error handling, and user experience.