Comprehensive Documentation for main17.py

1. Introduction

This document provides a comprehensive technical overview and detailed documentation for the `main17.py` Python application. This application serves as a robust backend service, primarily built using the FastAPI framework, designed to handle a variety of tasks including natural language processing (NLP), goal and task management, long-term memory, push notifications, research capabilities, and multimodal content analysis. It integrates with several external services and libraries such as Groq for advanced language model interactions, MongoDB for persistent data storage, FAISS for efficient similarity search, and Sentence Transformers for local embedding generation.

The primary objective of `main17.py` is to act as an intelligent assistant, capable of understanding user queries, managing personal goals, conducting research, and providing context-aware responses. It leverages a combination of cutting-edge AI models and efficient data handling mechanisms to deliver a responsive and personalized user experience. The architecture is designed to be scalable and maintainable, with clear separation of concerns for different functionalities.

This documentation will cover the following key areas:

- **Overall Architecture**: A high-level view of how different components of the application interact.
- **Environment and Dependencies**: Essential setup and external libraries used.
- **Data Models**: Pydantic models defining the structure of data exchanged within the API.
- **Database Interactions**: Details on MongoDB collections and data persistence.
- **Language Model Integrations**: How Groq and other LLMs are utilized for various tasks.
- **Embedding and Vector Search**: Implementation of FAISS and Sentence Transformers for semantic search.
- **File Processing and Multimodal Retrieval**: Handling of uploaded documents and code for contextual understanding.
- **Goal and Task Management**: Mechanisms for users to define, track, and manage their objectives.
- **Long-Term Memory**: How user interactions and goals are summarized and stored for personalized responses.

- **Push Notification System**: Implementation of web push notifications for reminders and proactive check-ins.
- **Research and Visualization**: Capabilities for internet browsing, subquery generation, and HTML visualization.
- **API Endpoints**: A detailed breakdown of all exposed API endpoints and their functionalities.
- **Error Handling and Logging**: Strategies for robust error management and operational monitoring.

Authored by: Manus AI

---

2. Overall Architecture

The `main17.py` application is structured as a FastAPI-based asynchronous web service, designed to handle concurrent requests efficiently. Its architecture can be broadly categorized into several interconnected layers and modules, each responsible for specific functionalities:

2.1. Core Service Layer (FastAPI)

At the heart of the application is FastAPI, which provides the web framework for building robust and high-performance APIs. It handles request routing, validation (using Pydantic), and serialization. Key features of this layer include:

- **Endpoint Management**: Defines various API endpoints for user interaction, data upload, goal management, and more.
- **Middleware**: Utilizes CORS middleware to enable cross-origin requests, ensuring the API can be accessed from different frontend applications.
- **Global Error Handling**: A centralized exception handler catches unhandled errors, providing consistent error responses and logging for debugging.

2.2. Data Persistence Layer (MongoDB)

MongoDB, a NoSQL database, is used for storing all application data. The `motor.motor_asyncio` driver facilitates asynchronous interactions with the database, aligning with FastAPI's asynchronous nature. Critical collections include:

- `chats_collection`: Stores conversation history for each user session.
- `memory_collection`: Holds long-term summaries and embeddings of user interactions for personalized responses.

- `uploads_collection` : Manages metadata and chunks of uploaded documents and code files.
- `goals_collection` : Stores user-defined goals and associated tasks.
- `users_collection` : Contains user subscription information for push notifications.
- `notifications_collection` : Schedules and tracks push notifications.
- `otp_collection` : Temporarily stores One-Time Passwords for email verification.

2.3. Language Model Integration Layer (Groq)

The application heavily relies on Groq's language models for various AI-driven tasks. To ensure reliability and manage API usage, this layer incorporates:

- **Multiple API Keys**: Different Groq API keys are used for distinct purposes (e.g., internet browsing, deep search, general generation, memory summarization, planning), enhancing rate limit management and fault tolerance.
- **Rate Limiting**: The `ratelimit` library is employed to enforce API call limits, preventing abuse and ensuring fair usage of external services.
- **Asynchronous Calls**: All interactions with Groq are performed asynchronously using `asyncio.to_thread` , preventing blocking of the main event loop.

2.4. Embedding and Vector Search Layer (Sentence Transformers & FAISS)

For efficient semantic search and retrieval of relevant context, the application integrates local embedding generation and a FAISS index:

- **`SentenceTransformer`** : Used to generate dense vector embeddings for text chunks from documents, code segments, and user queries. The `all-mpnet-base-v2` model is employed for this purpose.
- **FAISS (`faiss.IndexFlatL2`)**: Two FAISS indexes (`doc_index` and `code_index`) are maintained in memory to store and quickly search through document and code embeddings, respectively. This enables rapid retrieval of contextually similar information.

2.5. Document and Code Processing Layer

This layer is responsible for ingesting and preparing various file types for AI processing:

- **Text Extraction**: Supports PDF (`fitz`), DOCX (`docx2txt`), and plain text files. It extracts raw text content from these formats.

- **Chunking and Segmentation**: Extracted text is split into manageable chunks, and code files are segmented into functions and classes. This ensures that the content fits within the context windows of language models.
- **Embedding Storage**: Each chunk or segment is embedded and stored in MongoDB along with its metadata, and its embedding is added to the respective FAISS index.

2.6. Goal and Task Management Layer

This module provides the logic for users to interact with their goals and tasks. It parses specific commands from user messages (e.g., `[GOAL_SET]`, `[TASK_ADD]`) and updates the `goals_collection` accordingly. It also handles status changes, deadlines, and progress tracking.

2.7. Push Notification Layer

Leveraging web push technology, this layer manages user subscriptions and schedules notifications. It uses `pywebpush` to send notifications and `pytz` for timezone-aware scheduling of daily check-ins and proactive reminders.

2.8. Research and Visualization Layer

This advanced module enables the AI to conduct internet research and generate visualizations:

- **Subquery Generation**: Breaks down complex research queries into smaller, more manageable subqueries using an LLM.
- **Internet Browsing**: Uses Groq's internet browsing capabilities to fetch information relevant to subqueries.
- **Content Synthesis**: Synthesizes collected information into a coherent answer.
- **HTML Visualization**: Generates interactive HTML code to present research results visually, often incorporating charts and modern styling.
- **WebSocket Communication**: Utilizes WebSockets for real-time streaming of research and visualization progress to the client.

2.9. Utility Functions

A collection of helper functions supports various operations, including date/time formatting, OTP generation, email sending, message filtering, and MongoDB ObjectId conversion.

This modular design allows for clear separation of concerns, making the application easier to develop, test, and scale. The asynchronous nature, combined with intelligent caching

and external service integrations, contributes to a powerful and responsive AI assistant.

---

3. Environment and Dependencies

The `main17.py` application operates within a Python environment and relies on a specific set of libraries and external services. Proper setup of this environment is crucial for the application's correct functioning.

3.1. Core Technologies

- **Python**: The application is written in Python, leveraging its extensive ecosystem for AI and web development. The specific version used is typically Python 3.8+ for full `async` / `await` support.
- **FastAPI**: A modern, fast (high-performance) web framework for building APIs with Python 3.8+ based on standard Python type hints. It is the backbone of the application's web interface.
- **MongoDB**: A NoSQL document database used for flexible and scalable data storage. It stores all persistent application data, including chat histories, user goals, uploaded file metadata, and notification schedules.
- **Groq API**: An external service providing access to high-performance language models (LLMs) for various tasks such as content generation, summarization, internet browsing, and deep search. The application integrates with multiple Groq models.

3.2. Python Libraries

The following Python libraries are essential dependencies, managed via `pip`:

- **`fastapi`**: The web framework itself.
- **`uvicorn`**: An ASGI server that runs the FastAPI application.
- **`motor`**: An asynchronous Python driver for MongoDB, enabling non-blocking database operations.
- **`pydantic`**: Used for data validation and settings management, ensuring data integrity for API requests and configuration.
- **`python-dotenv`**: For loading environment variables from a `.env` file, crucial for managing API keys, database URIs, and other sensitive configurations.
- **`pytz`**: Provides timezone definitions, used for handling time-aware scheduling of notifications and daily check-ins.
- **`numpy`**: A fundamental package for numerical computing in Python, particularly used

here for handling vector embeddings.

- **httpx** : A fully featured HTTP client for Python 3, used for making asynchronous HTTP requests.
- **base64** : For encoding and decoding binary data, specifically used for handling image data in multimodal requests.
- **docx2txt** : A library for extracting text from .docx files.
- **PyMuPDF (fitz)**: A Python binding for MuPDF, used for extracting text from PDF documents.
- **loguru** : A library for robust and flexible logging, used for application-wide logging.
- **ratelimit** : A decorator for rate-limiting function calls, applied to external API interactions to prevent exceeding service limits.
- **sentence-transformers** : A Python framework for state-of-the-art sentence, text and image embeddings. Used to generate vector embeddings locally.
- **faiss-cpu** : A library for efficient similarity search and clustering of dense vectors. Used to create and manage in-memory vector indexes for document and code embeddings.
- **groq** : The official Python client library for interacting with the Groq API.
- **smtplib** : Python's standard library for sending emails, used for OTP delivery.
- **email.message** : For constructing email messages.
- **secrets** : For generating cryptographically strong random numbers, used for OTP generation.
- **html2image** : A library to convert HTML to images, potentially used for rendering visualizations.
- **pywebpush** : A Python library for sending Web Push Notifications.

3.3. Environment Variables

The application relies heavily on environment variables for configuration, loaded from a .env file using python-dotenv . Key environment variables include:

- MONGO_URI : The connection string for the MongoDB instance.
- VAPID_PUBLIC_KEY , VAPID_PRIVATE_KEY : Keys for Web Push Notification (VAPID protocol).
- SMTP_USERNAME , SMTP_PASSWORD : Credentials for the SMTP server used for sending emails.
- GROQ_API_KEY_* : Multiple Groq API keys (e.g., GROQ_API_KEY_INTERNET , GROQ_API_KEY_DEEPSEARCH , GROQ_API_KEY_MEMORY_SUMMARY ,

`GROQ_API_KEY_GENERATE_1`, `GROQ_API_KEY_PLANNING`) are used to manage different rate limits and access patterns for various LLM interactions.

3.4. External Services

- **MongoDB Atlas (or self-hosted MongoDB)**: Provides the database infrastructure.
- **Groq Cloud**: Hosts the language models used by the application.
- **SMTP Server**: An email server (e.g., `smtpout.secureserver.net`) for sending OTPs and other email communications.

3.5. Setup and Installation

To set up the development or production environment, the following general steps are required:

1. **Clone the repository**: Obtain the source code.
2. **Create a virtual environment**: `python -m venv venv && source venv/bin/activate`
3. **Install dependencies**: `pip install -r requirements.txt` (assuming a `requirements.txt` file lists all dependencies).
4. **Configure environment variables**: Create a `.env` file in the root directory with all necessary `MONGO_URI`, `VAPID` keys, `SMTP` credentials, and `GROQ_API_KEY`s.
5. **Run the application**: `uvicorn main17:app --host 0.0.0.0 --port 8000` (or similar command for production deployment).

This robust set of dependencies and a well-defined environment configuration ensure the application can perform its complex tasks reliably and efficiently.

---

4. Data Models (Pydantic)

Pydantic models are extensively used throughout `main17.py` to define the structure and validate the data exchanged via the FastAPI endpoints. This ensures type safety, provides clear API contracts, and automates data parsing and serialization.

4.1. Request Models

These models define the expected structure of incoming JSON payloads for various API endpoints.

- **`GenerateRequest`**
- **`BrowseRequest`**
- **`Subscription`**

- **OTPRequest**
- **VerifyOTPRequest**
- **Prompt**
- **DeepSearchRequest**
- **NLPRequest**
- **ResearchQuery**
- **RegenerateRequest**
- **PlanWeekRequest**
- **InstaPostRequest**
- **UserInput**

4.2. Response Models

These models define the structure of outgoing JSON responses from API endpoints.

- **GenerateResponse**
- **BrowseResponse**

4.3. Enum Models

- **PostGenOptions**

These Pydantic models are crucial for maintaining data consistency, enabling automatic documentation generation (via FastAPI's OpenAPI integration), and simplifying data handling throughout the application. They enforce contracts between the client and server, reducing the likelihood of errors due to malformed data.

---

5. Database Interactions (MongoDB)

The `main17.py` application leverages MongoDB as its primary data store, utilizing `motor.motor_asyncio` for asynchronous database operations. This section details the various collections used and their roles in maintaining the application's state and data.

5.1. MongoDB Client and Database Initialization

The connection to MongoDB is established through `AsyncIOMotorClient`, ensuring that database operations are non-blocking and compatible with FastAPI's asynchronous nature. The database instance is retrieved once and reused across the application.

```
Python
```

```
def get_database():
    client = AsyncIOMotorClient(os.getenv("MONGO_URI"))
    return client["stelle_db"]

db = get_database()
```

5.2. Collections Overview

The application uses several distinct collections, each designed to store specific types of data:

- **chats_collection**: Stores the conversational history for each user session. Each document in this collection represents a chat session and contains an array of messages, including user prompts and AI responses, along with their respective embeddings. This allows for context-aware interactions and regeneration of responses.
  - **Fields**: user_id, session_id, messages (array of role, content, embedding), last_updated.
- **memory_collection**: Dedicated to storing long-term memory summaries and their corresponding vector embeddings for each user. This collection enables the AI to maintain a persistent understanding of user interests, goals, and interaction style across sessions.
  - **Fields**: user_id, session_id, summary, vector (embedding), timestamp.
- **uploads_collection**: Manages metadata and content chunks of files uploaded by users. This includes documents (PDF, DOCX, TXT) and code files. Each chunk or code segment is stored with its embedding, allowing for multimodal retrieval.
  - **Fields**: user_id, session_id, filename, modality (e.g., 'document', 'code'), chunk_index or segment_name, text_snippet, embedding, timestamp, query_count.
- **goals_collection**: Stores user-defined goals and their associated tasks. This collection is central to the goal and task management system, tracking status, deadlines, and progress.
  - **Fields**: user_id, goal_id, session_id, title, description, status (e.g., 'active', 'in progress', 'completed'), created_at, updated_at, tasks (array of task_id, title, description, status, deadline, progress).
- **users_collection**: Stores user-specific information, primarily related to push notification subscriptions and timezone settings. This is crucial for delivering personalized and timely notifications.
  - **Fields**: user_id, push_subscription (dictionary), time_zone.

- `notifications_collection`: Used to schedule and track push notifications. Documents in this collection represent pending or sent notifications, including their scheduled time, message, and type.
  - **Fields**: `user_id`, `message`, `scheduled_time`, `type` (e.g., 'general', 'daily_checkin', 'reminder'), `sent`, `created_at`, `sent_at`, `status`.
- `otp_collection`: A temporary collection for storing One-Time Passwords (OTPs) used for email verification. It has a TTL (Time-To-Live) index to automatically expire documents after a set period (e.g., 300 seconds).
  - **Fields**: `email`, `otp`, `created_at`.

5.3. Data Management Operations

The application performs various CRUD (Create, Read, Update, Delete) operations on these collections:

- **Insertion**: New chat messages, user goals, uploaded file chunks, and notification schedules are inserted.
- **Retrieval**: Chat histories, user summaries, active goals, and pending notifications are frequently queried.
- **Updates**: Goal and task statuses, task deadlines, and progress updates are handled by updating existing documents. User push subscriptions and long-term memory summaries are also updated.
- **Deletion**: Completed goals, tasks, and expired OTPs are removed from their respective collections. Uploaded file chunks are removed when their `query_count` exceeds a threshold, indicating they are no longer actively relevant.

5.4. Indexing

MongoDB indexes are utilized to optimize query performance. A notable example is the TTL index on `otp_collection`'s `created_at` field, which automatically handles the expiration of OTPs, ensuring security and data hygiene.

```python
otp_collection.create_index("created_at", expireAfterSeconds=300)
```

5.5. Object ID Conversion

A utility function `convert_object_ids` is provided to convert MongoDB's `ObjectId` instances to string representations, which is often necessary when returning data via JSON APIs.

Python

```
def convert_object_ids(document: dict) -> dict:
    for key, value in document.items():
        if key == "_id":
            document[key] = str(value)
        elif isinstance(value, dict):
            document[key] = convert_object_ids(value)
        elif isinstance(value, list):
            document[key] = [convert_object_ids(item) if isinstance(item,
dict) else item for item in value]
    return document
```

This comprehensive use of MongoDB allows `main17.py` to store and manage a rich variety of structured and unstructured data, supporting its complex AI-driven functionalities and maintaining a persistent, personalized experience for users.

---

6. Language Model Integrations

The `main17.py` application heavily relies on advanced Language Models (LLMs) provided by Groq to power its intelligent functionalities. The integration is designed to be robust, efficient, and capable of handling various types of AI tasks, from general conversational responses to specialized research and content generation.

6.1. Groq API Clients and Key Management

To optimize performance and manage API rate limits effectively, the application initializes multiple Groq clients, each potentially using a different API key and assigned to specific tasks. This strategy helps distribute the load and prevents a single bottleneck from impacting critical functionalities.

Python

```
# Step 1: Get all keys that start with 'GROQ_API_KEY_'
available_keys = [
    value for key, value in os.environ.items()
    if key.startswith("GROQ_API_KEY_")
]

# Step 2: Choose a random key for each client
internet_client = Groq(api_key=random.choice(available_keys))
deepsearch_client = Groq(api_key=random.choice(available_keys))
client = Groq(api_key=random.choice(available_keys))
```

- **internet_client** : Used for tasks requiring internet access, such as summarizing web pages or YouTube videos, and fetching trending hashtags or SEO keywords. It typically uses models optimized for browsing capabilities (e.g., compound-beta).
- **deepsearch_client** : Dedicated to more intensive search and research tasks, potentially using models capable of deeper analysis.
- **client** : A general-purpose client used for various conversational and generation tasks, often employing models like llama-3.3-70b-versatile or deepseek-r1-distill-llama-70b .
- **client_generate** : For the main /generate and /nlp endpoints, a random API key is selected from a pool of GROQ_API_KEY_GENERATE_* keys to further distribute requests and enhance reliability.

6.2. Rate Limiting Strategy

To comply with API usage policies and ensure stable operation, rate limiting is applied to critical Groq API calls using the ratelimit library. This prevents the application from overwhelming the Groq API with too many requests in a short period.

Python

```
CALLS_PER_MINUTE = 50
PERIOD = 60  # seconds

@sleep_and_retry
@limits(calls=CALLS_PER_MINUTE, period=PERIOD)
async def query_internet_via_groq(
    query: str,
    return_sources: bool = False
) -> Union[str, Tuple[str, List[dict]]]:
    # ... implementation ...
```

The @sleep_and_retry and @limits decorators ensure that calls to query_internet_via_groq (and similarly rate_limited_groq_call) adhere to a maximum of 50 calls per minute, automatically pausing and retrying if the limit is approached.

6.3. Core LLM Functions and Their Applications

The application defines several asynchronous functions that encapsulate specific interactions with Groq models:

- **query_internet_via_groq(query: str, return_sources: bool = False)** : This function is the primary interface for performing internet-aware queries. It sends a query to Groq and can optionally return a list of sources (title and URL) if the LLM uses a search tool. It's

used for tasks like summarizing web content, YouTube videos, and general information retrieval.

- **content_for_website(content: str)** : Summarizes provided content concisely, listing key themes and providing a brief final summary. This is useful for processing external content (e.g., from URLs) before feeding it into the main generation pipeline.
- **detailed_explanation(content: str)** : Provides a more detailed explanation or summary of given content, often used for deeper analysis of external information.
- **classify_prompt(prompt: str)** : Classifies a user prompt to determine if it requires internet research. This helps the application decide whether to engage the browsing capabilities of the LLM.
- **browse_and_generate(query: str)** : Executes an internet search based on a query and synthesizes the findings into a coherent response. This function is used when a user's query explicitly or implicitly requires external knowledge.
- **generate_seed_keywords(query: str)** : Generates 3 seed keywords from a user's query, which are then used to fetch trending hashtags and SEO keywords.
- **fetch_trending_hashtags(seed_keywords: list)** : Browses social media platforms (Instagram, X, Reddit) to find up to 30 trending hashtags related to the provided seed keywords, aiming to boost SEO.
- **fetch_seo_keywords(seed_keywords: list)** : Identifies up to 15 SEO keywords by analyzing top blogs and posts related to the seed keywords.
- **generate_caption(query: str, seed_keywords: list, trending_hashtags: list, seo_keywords: list)** : Crafts a social media caption with a strong hook, incorporating the user's query, seed keywords, trending hashtags, and SEO terms.
- **efficient_summarize(previous_summary: str, new_messages: list, user_id: str)** : Used for long-term memory, this function generates a concise summary of user interactions, interests, and goals. It takes into account previous summaries and new messages to create an updated, personalized overview.
- **clarify_query(query: str)** : Refines and optimizes a user's original query into a more effective and concise search prompt, improving the quality of subsequent LLM interactions.
- **clarify_response(query: str)** : Understands the user's query and outlines the next steps to address it, often formatted in Slack Markdown.
- **generate_subqueries(main_query: str, num_subqueries: int = 3)** : Breaks down a main research query into several distinct subqueries, facilitating a more structured and comprehensive research process.

- **visual_generate_subqueries(main_query: str, num_subqueries: int = 3)** : Similar to generate_subqueries but tailored for visualization topics.
- **visual_synthesize_result(main_query: str, contents: list, max_context: int = 4000)** : Synthesizes collected content into a concise, accurate, and well-structured answer for a visualization prompt.
- **generate_html_visualization(content: str)** : Generates complete HTML code for a professional and modern data visualization based on synthesized research results, including interactive elements and dark theme styling.

6.4. Asynchronous Execution

All interactions with Groq API are wrapped in asyncio.to_thread . This is a crucial design choice for a FastAPI application, as it allows CPU-bound (or I/O-bound but synchronous) operations like calling external APIs to run in a separate thread pool, preventing them from blocking the main asynchronous event loop. This ensures the application remains responsive even during lengthy LLM calls.

By strategically utilizing multiple Groq clients, implementing robust rate limiting, and carefully designing asynchronous interaction patterns, main17.py achieves high performance and reliability in its AI-driven functionalities.

---

7. Embedding and Vector Search

To enable efficient retrieval of relevant information from uploaded documents and code, main17.py implements a sophisticated embedding and vector search system. This system allows the application to understand the semantic meaning of user queries and find contextually similar pieces of information, significantly enhancing the AI's ability to provide accurate and informed responses.

7.1. Local Embedding Generation with Sentence Transformers

The core of the semantic search capability is the generation of vector embeddings. These numerical representations capture the semantic meaning of text, allowing for mathematical comparisons of similarity. The application uses the SentenceTransformer library for this purpose.

Python

```
local_embedding_model = SentenceTransformer('all-mpnet-base-v2')

async def generate_text_embedding(text: str | None) -> list:
    if not text:
```

```
        logging.warning("generate_text_embedding called with empty or None
text; returning empty embedding.")
        return []
    try:
        embedding = await asyncio.to_thread(
            local_embedding_model.encode,
            text,
            convert_to_numpy=True
        )
        embedding_list = embedding.tolist()
        if len(embedding_list) != 768:
            logging.error(f"Embedding has unexpected length:
{len(embedding_list)}")
            return []
        return embedding_list
    except Exception as e:
        logging.error(f"Local embedding generation error: {e}",
exc_info=True)
        return []
```

- **Model Choice**: The `all-mpnet-base-v2` model is selected for its balance of performance and accuracy in generating sentence embeddings. This model produces 768-dimensional vectors.
- **Asynchronous Execution**: The `local_embedding_model.encode` method, which can be CPU-intensive, is executed within `asyncio.to_thread`. This ensures that the embedding generation process does not block the main asynchronous event loop of the FastAPI application, maintaining responsiveness.
- **Input Handling**: The function gracefully handles `None` or empty input text, returning an empty list and logging a warning. It also validates the output embedding dimension, logging an error if it deviates from the expected 768 dimensions.

7.2. FAISS for Efficient Vector Search

Once text chunks and code segments are converted into embeddings, an efficient mechanism is needed to search through these vectors. Facebook AI Similarity Search (FAISS) is employed for this purpose, providing highly optimized algorithms for similarity search in large vector spaces.

Python

```
doc_index = faiss.IndexFlatL2(768)
code_index = faiss.IndexFlatL2(768)

user_memory_map = {}
```

```
file_doc_memory_map = {}
code_memory_map = {}
```

- **faiss.IndexFlatL2(768)** : Two `IndexFlatL2` indexes are initialized, one for documents (`doc_index`) and one for code (`code_index`). `FlatL2` is a simple yet effective index that performs an exhaustive search using L2 (Euclidean) distance, suitable for moderate-sized datasets where high recall is paramount. The `768` indicates the dimensionality of the vectors it will store.
- **Memory Maps**: Python dictionaries (`user_memory_map` , `file_doc_memory_map` , `code_memory_map`) are used to map FAISS internal indices back to the original metadata (e.g., `user_id` , `session_id` , `filename` , `chunk_index` , `text_snippet`). This allows the application to retrieve the actual text content and context once a similar embedding is found.

7.3. Multimodal Context Retrieval (`retrieve_multimodal_context`)

The `retrieve_multimodal_context` function is responsible for orchestrating the semantic search process, combining document and code context based on a user's query.

Python

```
async def retrieve_multimodal_context(query: str, session_id: str,
filenames: list[str]) -> Tuple[str, set]:
    try:
        embedding = await generate_text_embedding(query)
        contexts = []
        used_filenames = set()

        # Document Index Search
        if embedding and doc_index.ntotal > 0:
            query_vector = np.array(embedding, dtype="float32").reshape(1,
-1)
            k = 10  # Retrieve top 10 document chunks
            distances, indices = doc_index.search(query_vector, k)
            # ... (logic to filter and select top 2 chunks per file) ...

        # Code Index Search
        if embedding and code_index.ntotal > 0:
            query_vector = np.array(embedding, dtype="float32").reshape(1,
-1)
            k = 3  # Retrieve top 3 code segments
            distances, indices = code_index.search(query_vector, k)
            # ... (logic to filter and select code segments) ...

        return "\n\n".join(contexts), used_filenames
```

```
    except Exception as e:
        logging.error(f"Error during multimodal retrieval: {e}",
exc_info=True)
        return "", set()
```

- **Query Embedding**: The user's query is first converted into an embedding vector.
- **Document Search**: If the `doc_index` contains entries, a similarity search is performed to find the top `k` (e.g., 10) most relevant document chunks. The results are then further refined to select the top 2 chunks per relevant file, preventing a single large file from dominating the context.
- **Code Search**: Similarly, if the `code_index` has entries, a search is performed to find the top `k` (e.g., 3) most relevant code segments.
- **Context Filtering**: Retrieved chunks and segments are filtered based on the current `session_id` and optionally by specific `filenames` provided in the request. A `query_count` mechanism is also in place to limit how many times a specific chunk can be used, preventing over-reliance on stale or frequently used information.
- **Snippet Generation**: For each retrieved piece of context, a snippet (e.g., `"From {filename} (Chunk {index}):\n{text_snippet}"`) is constructed and added to the overall `contexts` list.
- **Return Value**: The function returns a concatenated string of all retrieved contexts and a set of filenames that contributed to the context.

7.4. Integration with File Processing

When files are uploaded via the `/uploadfile` endpoint, their content is processed, chunked/segmented, embedded, and then added to the respective FAISS indexes and MongoDB `uploads_collection`. This ensures that all ingested data is immediately available for semantic search.

This robust embedding and vector search system is critical for the AI's ability to provide highly relevant and context-aware responses, drawing directly from the user's provided documents and codebases.

---

8. File Processing and Multimodal Retrieval

The `main17.py` application is designed to ingest and process various types of user-uploaded files, including documents and code, to enrich the context available to the AI. This multimodal capability allows the AI to provide more accurate and relevant responses by drawing information directly from the user's specific data. The process involves text extraction, content chunking/segmentation, embedding generation, and storage in both MongoDB and FAISS indexes.

8.1. Document Extraction Functions

The application supports extracting text from common document formats:

- **`extract_text_from_pdf(file: UploadFile) -> str`**:
  This asynchronous function handles PDF files. It uses `fitz` (PyMuPDF) to open the PDF stream and extract text page by page. Robust error handling is included to log any parsing failures.
- **`extract_text_from_docx(file: UploadFile) -> str`**:
  For Microsoft Word documents (`.docx`), the `docx2txt` library is used. The file content is read into a `BytesIO` object, which `docx2txt.process` then uses to extract the text. Similar to PDF extraction, it includes error logging.
- **`extract_text_from_txt(file: UploadFile) -> str`**:
  Plain text files (`.txt`) are handled by simply decoding their contents using UTF-8. This is the simplest extraction method.

8.2. Content Chunking and Code Segmentation

To effectively utilize the extracted text with language models, which often have token limits, the content is broken down into smaller, manageable pieces:

- **`split_text_into_chunks(text: str, chunk_size: int = 500, overlap: int = 100) -> list[str]`**:
  This utility function splits long text into overlapping chunks. It processes text by words, creating chunks of a specified `chunk_size` (default 500 words) with a defined `overlap` (default 100 words). Overlapping chunks help preserve context across chunk boundaries during retrieval.
- **`extract_code_segments(code: str) -> list`**:
  For Python code files, this function attempts to parse the code using Python's `ast` (Abstract Syntax Tree) module. It identifies top-level function and class definitions and extracts their code as individual segments. If AST parsing fails (e.g., due to syntax errors), it falls back to a simpler line-based chunking mechanism.

8.3. File Upload Endpoint (`/uploadfile`)

The `/uploadfile` endpoint is a POST request handler that accepts multiple files from a user. It orchestrates the entire file processing pipeline:

Python

```
@app.post("/uploadfile")
async def upload_file(
    user_id: str = Form(...),
    session_id: str = Form(...),
```

```
    modality: str = Form(...),
    files: List[UploadFile] = File(...)
):
    # ... implementation ...
```

- **Input Parameters**: It expects `user_id`, `session_id`, `modality` (e.g.,

9. Goal and Task Management

One of the core functionalities of `main17.py` is to assist users in managing their goals and tasks. This system allows users to define objectives, break them down into actionable tasks, track their progress, set deadlines, and receive proactive reminders. The implementation is deeply integrated with the AI's conversational capabilities, allowing users to manage their goals through natural language commands.

9.1. Goal and Task Data Structure

Goals and tasks are stored in the `goals_collection` in MongoDB. Each goal is a document that can contain an array of tasks. This hierarchical structure allows for detailed tracking.

Goal Document Structure:

JSON

```
{
  "user_id": "string",
  "goal_id": "string" (UUID),
  "session_id": "string",
  "title": "string",
  "description": "string",
  "status": "active" | "in progress" | "completed" | "cancelled",
  "created_at": "datetime",
  "updated_at": "datetime",
  "tasks": [
    {
      "task_id": "string" (UUID),
      "title": "string",
      "description": "string",
      "status": "not started" | "in progress" | "completed" | "cancelled" |
"Deadline Exceeded",
      "created_at": "datetime",
      "updated_at": "datetime",
      "deadline": "datetime" | null,
      "progress": [
        {
          "timestamp": "datetime",
```

```
            "description": "string"
          }
        ]
      }
    ]
}
```

9.2. AI-Driven Goal and Task Commands

The AI processes user messages and identifies specific commands embedded within the text to manage goals and tasks. These commands follow a predefined Markdown-like syntax, enabling the AI to parse user intent and update the database accordingly. The `generate_response` and `nlp_websocket_endpoint` functions contain the logic for parsing these commands.

Command Syntax and Functionality:

- **`[GOAL_SET: <goal_title>]`** : Creates a new goal. The AI extracts the goal title from the user's message. If tasks are also specified using `[TASK: <task_desc>]` within the same message, they are added to the newly created goal.
  - *Example*: "My new goal is to learn Python. `[GOAL_SET: Learn Python]` `[TASK: Complete Python Basics course]` `[TASK: Build a simple web app]` "
- **`[TASK: <task_desc>]`** : Defines a task associated with the most recently set or active goal. This command is typically used in conjunction with `GOAL_SET` .
- **`[GOAL_DELETE: <goal_id>]`** : Deletes an existing goal based on its unique `goal_id` . The AI will attempt to find and remove the corresponding goal document from `goals_collection` .
  - *Example*: "Delete my old project goal. `[GOAL_DELETE: <uuid-of-goal>]` "
- **`[TASK_DELETE: <task_id>]`** : Removes a specific task from its parent goal using its `task_id` . This involves updating the `tasks` array within the relevant goal document.
  - *Example*: "I've decided not to do that task. `[TASK_DELETE: <uuid-of-task>]` "
- **`[TASK_ADD: <goal_id>: <task_description>]`** : Adds a new task to an existing goal. The `goal_id` specifies which goal the task should be added to.
  - *Example*: "Add a new task to my 'Learn Python' goal: `[TASK_ADD: <uuid-of-goal>: Explore FastAPI]` "
- **`[TASK_MODIFY: <task_id>: <new_title_or_description>]`** : Updates the title or description of an existing task. The AI finds the task by `task_id` and modifies its `title` field.
  - *Example*: "Update my 'Build a simple web app' task: `[TASK_MODIFY: <uuid-of-task>: Build a FastAPI backend with MongoDB]` "
- **`[GOAL_START: <goal_id>]`** : Changes the status of a goal to "in progress".

    - *Example*: "I'm starting my Python learning journey! `[GOAL_START: <uuid-of-goal>]` "
- **`[TASK_START: <task_id>]`** : Changes the status of a specific task to "in progress".
    - *Example*: "Starting the FastAPI course. `[TASK_START: <uuid-of-task>]` "
- **`[GOAL_COMPLETE: <goal_id>]`** : Marks a goal as "completed".
    - *Example*: "I've finished my Python goal! `[GOAL_COMPLETE: <uuid-of-goal>]` "
- **`[TASK_COMPLETE: <task_id>]`** : Marks a specific task as "completed".
    - *Example*: "Finished the FastAPI course. `[TASK_COMPLETE: <uuid-of-task>]` "
- **`[TASK_DEADLINE: <task_id>: <YYYY-MM-DD HH:MM>]`** : Sets a deadline for a task. The AI also schedules a reminder notification one day before the deadline.
    - *Example*: "Set a deadline for the web app task: `[TASK_DEADLINE: <uuid-of-task>: 2025-10-31 17:00]` "
- **`[TASK_PROGRESS: <task_id>: <progress_description>]`** : Adds a progress update to a task. This allows users to log their efforts and milestones.
    - *Example*: "Progress on web app: `[TASK_PROGRESS: <uuid-of-task>: Implemented user authentication]` "

9.3. Integration with AI Response Generation

When a user interacts with the `/generate` or `/nlp` endpoints, the AI first processes the user's message to identify any goal or task management commands. It then updates the `goals_collection` in MongoDB based on these commands. After processing, the commands are typically stripped from the message before being sent to the LLM for generating a conversational response, ensuring the LLM focuses on the core query rather than the command syntax.

The `goals_context` (a summary of active goals and tasks) is also included in the system prompt for the LLM, allowing the AI to be aware of the user's ongoing objectives and tailor its responses accordingly. This context helps the AI provide more relevant advice, suggestions, and support related to the user's personal and professional development.

9.4. Weekly Planning (`/Plan_my_week`)

The `/Plan_my_week` endpoint provides a sophisticated feature for generating a structured weekly plan based on the user's active goals and tasks. This endpoint leverages an LLM to act as an "expert project manager and personal coach."

- **Process**:
    1. **Timezone Retrieval**: Fetches the user's timezone to localize dates and times.

2. **Goal Retrieval**: Gathers all active and in-progress goals and their associated tasks from `goals_collection`.
3. **Context Preparation**: Formats the goals and tasks, including their statuses and deadlines (noting "Deadline Exceeded" tasks), into a comprehensive context for the LLM.
4. **AI Prompt Construction**: A detailed prompt is constructed, instructing the LLM to generate a weekly plan in a specific JSON format. Critical rules are enforced, such as prioritizing overdue tasks, providing detailed "how-to" descriptions for tasks and sub-tasks, and ensuring a full 7-day plan.
5. **LLM Call**: An asynchronous call is made to a Groq LLM (e.g., `llama-3.3-70b-versatile`) with a `response_format` set to `json_object` to ensure structured output.
6. **Plan Storage**: The generated JSON plan is then stored in the `weekly_plans_collection` in MongoDB, either as a new entry or by updating an existing plan for the current week.

This goal and task management system, combined with AI-driven planning, empowers users to effectively organize their work and personal objectives, with the AI acting as a proactive and intelligent assistant.

---

10. Long-Term Memory

The `main17.py` application incorporates a sophisticated long-term memory system that allows the AI to retain context, user preferences, and ongoing goals across multiple sessions. This personalization is crucial for providing a consistent and highly relevant user experience. The system relies on summarizing past interactions and storing these summaries as vector embeddings for efficient retrieval.

10.1. Memory Data Structure

Long-term memory is stored in the `memory_collection` in MongoDB. Each document in this collection represents a user's summarized memory.

Memory Document Structure:

JSON

```
{
  "user_id": "string",
  "session_id": "string",
  "summary": "string" (concise text summary),
  "vector": "array of floats" (embedding of the summary),
```

```
  "timestamp": "datetime"
}
```

10.2. Efficient Summarization (`efficient_summarize`)

The core of the long-term memory system is the `efficient_summarize` function, which is responsible for creating a concise summary of user interactions. This summary captures the user's interests, style, and ongoing goals.

Python

```
async def efficient_summarize(previous_summary: str, new_messages: list,
user_id: str, max_summary_length: int = 500) -> str:
    user_queries = "\n".join([msg["content"] for msg in new_messages if
msg["role"] == "user"])
    context_text = f"User ID: {user_id}\n"
    if previous_summary:
        context_text += f"Previous Summary:\n{previous_summary}\n\n"
    context_text += f"New User Queries:\n{user_queries}\n\n"
    # ... (incorporates active goals context) ...

    summary_prompt = (
        f"Based on the following context, generate a concise summary (max
{max_summary_length} characters) "
        f"that captures the user's interests, style, and ongoing
goals:\n\n{context_text}"
    )
    try:
        client = Groq(api_key=os.getenv("GROQ_API_KEY_MEMORY_SUMMARY"))
        response = await asyncio.to_thread(
            client.chat.completions.create,
            messages=[{"role": "system", "content": "You are an AI that
creates personalized conversation summaries."},
                      {"role": "user", "content": summary_prompt}],
            model="llama3-70b-8192",
            max_tokens=150,
            temperature=0.7
        )
        return response.choices[0].message.content.strip()
    except Exception as e:
        logging.error(f"Long-term memory summarization error: {e}",
exc_info=True)
        return previous_summary if previous_summary else "Summary
unavailable."
```

- **Contextual Input**: The summarization process takes into account the `previous_summary` (if any), `new_messages` from the current interaction, and the user's `active_goals`. This rich context allows the LLM to generate a highly relevant and evolving summary.
- **LLM Role**: A Groq LLM (e.g., `llama3-70b-8192`) is instructed to act as an "AI that creates personalized conversation summaries," ensuring the output is tailored to the application's needs.
- **Conciseness**: The prompt explicitly requests a concise summary, typically limited to a `max_summary_length` (default 500 characters), to keep the memory manageable and efficient for future LLM contexts.
- **Asynchronous Execution**: The LLM call is made asynchronously using `asyncio.to_thread` to prevent blocking.

10.3. Storing Long-Term Memory (`store_long_term_memory`)

The `store_long_term_memory` function orchestrates the update of a user's long-term memory. It is typically called in the background after a certain number of messages have accumulated in a chat session (e.g., every 10 messages).

Python

```
async def store_long_term_memory(user_id: str, session_id: str,
new_messages: list):
    try:
        mem_entry = await memory_collection.find_one({"user_id": user_id})
        previous_summary = mem_entry.get("summary", "") if mem_entry else ""
        new_summary = await efficient_summarize(previous_summary,
new_messages, user_id)
        new_vector = await generate_text_embedding(new_summary)
        new_vector_np = np.array(new_vector, dtype="float32").reshape(1, -1)

        if mem_entry:
            # Update existing memory entry
            await memory_collection.update_one(
                {"user_id": user_id},
                {"$set": {
                    "summary": new_summary,
                    "session_id": session_id,
                    "vector": new_vector,
                    "timestamp": datetime.datetime.now(datetime.timezone.utc)
                }}
            )
            # Update FAISS index
            if user_id in user_memory_map:
                doc_index.remove_ids(np.array([user_memory_map[user_id]],
```

```
dtype="int64"))
            idx = doc_index.ntotal
            doc_index.add(new_vector_np)
            user_memory_map[user_id] = idx
        else:
            # Insert new memory entry
            await memory_collection.insert_one({
                "user_id": user_id,
                "session_id": session_id,
                "summary": new_summary,
                "vector": new_vector,
                "timestamp": datetime.datetime.now(datetime.timezone.utc)
            })
            # Add to FAISS index
            idx = doc_index.ntotal
            doc_index.add(new_vector_np)
            user_memory_map[user_id] = idx
        logging.info(f"Long-term memory updated for user {user_id}")
    except Exception as e:
        logging.error(f"Error storing long-term memory: {e}", exc_info=True)
```

- **Retrieval of Previous Memory**: It first attempts to retrieve any existing long-term memory for the `user_id` from `memory_collection`.
- **Summarization**: The `efficient_summarize` function is called to generate an updated summary.
- **Embedding Generation**: The `new_summary` is then converted into a vector embedding using `generate_text_embedding`.
- **MongoDB Update/Insert**: The `memory_collection` is either updated with the new summary and embedding (if an entry exists) or a new entry is inserted.
- **FAISS Integration**: The embedding of the new summary is also added to the `doc_index` (or a dedicated user memory index if one were created). If an old summary existed, its corresponding entry in the FAISS index is removed before adding the new one. This ensures that the FAISS index always reflects the most current user memory.

10.4. Usage in AI Response Generation

When generating a response for a user, the AI retrieves the user's long-term memory summary and includes it in the system prompt. This allows the LLM to generate responses that are consistent with the user's past interactions, preferences, and ongoing goals, leading to a more personalized and coherent conversational flow.

Python

```
# ... inside generate_response or nlp_websocket_endpoint ...
long_term_memory = ""
mem_entry = await memory_collection.find_one({"user_id": user_id})
if mem_entry and "summary" in mem_entry:
    long_term_memory = mem_entry["summary"]

# ... then added to messages for LLM ...
if long_term_memory:
    messages.append({"role": "system", "content": f"Long-term memory:
{long_term_memory}"})
```

This robust long-term memory system is a key component in enabling `main17.py` to function as a truly intelligent and personalized AI assistant, capable of learning and adapting to individual user needs over time.

---

11. Push Notification System

The `main17.py` application includes a comprehensive push notification system, enabling proactive communication with users for reminders, daily check-ins, and other important updates. This system leverages Web Push technology, allowing notifications to be delivered even when the user is not actively using the application. It integrates with VAPID (Voluntary Application Server Identification) for secure and authenticated messaging.

11.1. VAPID Configuration

For secure push notifications, the application uses VAPID keys, which are loaded from environment variables. These keys are essential for identifying the application server to the push service.

Python

```
VAPID_PUBLIC_KEY = os.getenv("VAPID_PUBLIC_KEY")
VAPID_PRIVATE_KEY = os.getenv("VAPID_PRIVATE_KEY")
VAPID_CLAIMS = {"sub": "mailto:info@stelle.world"}
```

- **`VAPID_PUBLIC_KEY`**: The public key used by the client (browser) to subscribe to push notifications.
- **`VAPID_PRIVATE_KEY`**: The private key used by the server to sign push requests, ensuring their authenticity.
- **`VAPID_CLAIMS`**: A dictionary containing information about the sender, typically an email address or URL, used by the push service for contact in case of issues.

11.2. User Subscription Endpoint (`/subscribe`)

Users register their browser's push subscription details with the backend via the `/subscribe` endpoint. This endpoint receives the subscription object generated by the browser's Push API, along with the user's ID and timezone.

Python

```
@app.post("/subscribe")
async def subscribe(subscription: Subscription):
    try:
        user_filter = {"user_id": subscription.user_id}
        update_data = {
            "$set": {
                "push_subscription": subscription.subscription,
                "time_zone": subscription.time_zone
            }
        }
        await users_collection.update_one(user_filter, update_data,
upsert=True)
        return {"success": True, "message": "Subscription stored
successfully."}
    except Exception as e:
        logging.error(f"Error in subscription: {e}", exc_info=True)
        raise HTTPException(status_code=500, detail="Failed to store
subscription.")
```

- **`Subscription` Model**: The incoming data is validated using the `Subscription` Pydantic model, ensuring the presence of `user_id` , `subscription` object, and `time_zone` .
- **Database Storage**: The subscription information is stored in the `users_collection` in MongoDB. An `upsert=True` operation ensures that if a user already exists, their subscription is updated; otherwise, a new entry is created.

11.3. Scheduling Notifications (`schedule_notification`)

Notifications are not sent immediately but are scheduled for a specific future time. This allows for flexible and time-sensitive delivery, such as daily check-ins or deadline reminders.

Python

```
async def schedule_notification(user_id: str, message: str, scheduled_time:
datetime.datetime, notif_type: str = "general"):
    notif = {
        "user_id": user_id,
        "message": message,
```

```
        "scheduled_time": scheduled_time,
        "type": notif_type,
        "sent": False,
        "created_at": datetime.datetime.now(datetime.timezone.utc)
    }
    await notifications_collection.insert_one(notif)
    logging.info(f"Notification scheduled for user {user_id} at
{scheduled_time} with type \'{notif_type}\'.")
```

- **`notifications_collection`**: Scheduled notifications are stored in this MongoDB collection, marked with a `scheduled_time` and a `sent` flag (initially `False`).
- **Notification Types**: Notifications can be categorized by `notif_type` (e.g., "general", "daily_checkin", "reminder"), allowing for different handling or display on the client side.

11.4. Notification Checker (`notification_checker`)

A background task continuously monitors the `notifications_collection` for notifications that are due to be sent. This asynchronous loop ensures timely delivery.

Python

```
async def notification_checker():
    while True:
        now = datetime.datetime.now(datetime.timezone.utc)
        cursor = notifications_collection.find({"scheduled_time": {"$lte":
now}, "sent": False})
        async for notif in cursor:
            # ... (retrieves user subscription, sends webpush, updates
notification status) ...
            try:
                webpush(
                    subscription_info,
                    data=payload,
                    vapid_private_key=VAPID_PRIVATE_KEY,
                    vapid_claims=VAPID_CLAIMS
                )
                await notifications_collection.update_one(
                    {"_id": notif["_id"]},
                    {"$set": {"sent": True, "sent_at":
datetime.datetime.now(datetime.timezone.utc)}}
                )
                logging.info(f"Push notification sent to user {user_id}")
            except WebPushException as ex:
```

```
                # ... (error handling, e.g., if subscription is invalid) ...
        await asyncio.sleep(60) # Checks every minute
```

- **Polling Mechanism**: The `notification_checker` runs in an infinite loop, querying the database every 60 seconds for unsent notifications whose `scheduled_time` is in the past or present.
- **`webpush`**: For each due notification, it retrieves the user's `push_subscription` from `users_collection` and uses the `pywebpush.webpush` function to send the notification. The payload includes a title, body, icon, and data URL.
- **Status Update**: After a successful send, the notification's `sent` status is updated to `True` in the database. Error handling is included for `WebPushException` (e.g., if a subscription has expired).

11.5. Proactive Scheduling

The system includes several proactive scheduling mechanisms to engage users and help them stay on track with their goals:

- **`daily_checkin_scheduler`**:
  This scheduler runs daily and sends a morning check-in notification to users. It identifies active goals and includes them in the message, encouraging users to focus on their objectives. Timezone awareness (`pytz`) ensures the notification is sent at 9 AM local time for each user.
- **`proactive_checkin_scheduler`**:
  This scheduler sends proactive check-in notifications at multiple points throughout the day (e.g., 9 AM, 2 PM, 7 PM local time). It checks on the user's progress with active goals and offers support.
- **`schedule_immediate_reminder(user_id: str, reminder_text: str)`**:
  Allows the AI to schedule an immediate reminder (e.g., 1 minute in the future) based on user commands or internal logic. This is used for dynamic, context-specific reminders.

11.6. Integration with Goal Management

The push notification system is tightly integrated with the goal and task management features. When a user sets a deadline for a task using `[TASK_DEADLINE]`, the system automatically schedules a reminder notification (e.g., one day before the deadline) to help the user stay organized.

This robust push notification system significantly enhances the user experience by providing timely, personalized, and proactive assistance, reinforcing the AI's role as an intelligent and supportive assistant.

---

12. Research and Visualization

The `main17.py` application features an advanced research and visualization module that allows the AI to conduct in-depth internet research and present the findings in a structured and visually appealing format. This capability is primarily managed through WebSocket connections, providing real-time feedback to the user as the research process unfolds.

12.1. Research Process (`research_process`)

The research process is a multi-step workflow designed to break down a complex query, gather information from the internet, and synthesize it into a coherent answer.

Python

```
async def research_process(main_query: str, websocket: WebSocket):
    try:
        await websocket.send_json({"step": "start", "message": "Starting
research..."})

        # 1. Generate Subqueries
        await websocket.send_json({"step": "generating_subqueries",
"message": "Generating subqueries..."})
        subqueries = await generate_subqueries(main_query)
        # ...

        # 2. Query Internet for Each Subquery
        all_contents = []
        for subquery in subqueries:
            await websocket.send_json({"step": "querying", "subquery":
subquery, "message": f"Querying for ‘{subquery}’..."})
            content = await query_internet_via_groq(subquery)
            # ...

        # 3. Synthesize Final Result
        await websocket.send_json({"step": "synthesizing", "message":
"Synthesizing final result..."})
        final_result = await synthesize_result(main_query, all_contents)
        await websocket.send_json({"step": "final_result", "result":
final_result})

        await websocket.send_json({"step": "end", "message": "Research
complete."})
    except WebSocketDisconnect:
        # ...
```

```
    except Exception as e:
        # ...
```

- **Step 1: Subquery Generation**: The `generate_subqueries` function uses an LLM to break down the `main_query` into several distinct, focused search queries. This allows for a more comprehensive and structured investigation.
- **Step 2: Internet Browsing**: For each `subquery`, the `query_internet_via_groq` function is called to fetch relevant information from the web. The progress is streamed to the user via WebSocket.
- **Step 3: Content Synthesis**: After gathering content from all subqueries, the `synthesize_result` function uses an LLM to combine the trimmed contents into a single, well-structured answer to the original `main_query`.

12.2. Visualization Process (`visualization_process`)

The visualization process extends the research workflow by adding a final step to generate an interactive HTML visualization of the synthesized results.

Python

```
async def visualization_process(main_query: str, websocket: WebSocket):
    try:
        # ... (Steps 1 & 2 are similar to research_process, using
visual_generate_subqueries and visual_synthesize_result) ...

        # 3. Synthesize Visualization Result
        await websocket.send_json({"step": "synthesizing", "message":
"Synthesizing research result..."})
        synthesized = await visual_synthesize_result(main_query,
all_contents)
        await websocket.send_json({"step": "synthesized", "result":
synthesized})

        # 4. Generate HTML Visualization
        await websocket.send_json({"step": "generating_html", "message":
"Generating HTML visualization..."})
        html_code = await generate_html_visualization(synthesized)
        await websocket.send_json({"step": "html_generated", "html":
html_code})

        await websocket.send_json({"step": "end", "message": "Visualization
complete."})
    except WebSocketDisconnect:
        # ...
```

```
    except Exception as e:
        # ...
```

- **`generate_html_visualization(content: str)`**: This function is the core of the visualization capability. It takes the synthesized research result and uses a powerful LLM (e.g., `deepseek-r1-distill-llama-70b`) to generate a complete HTML document. The prompt instructs the LLM to create a professional, modern layout with a dark theme, brand color accents, and interactive visualizations (e.g., pie charts, bar graphs) using modern CSS and JavaScript. The goal is to produce a self-contained HTML file that visually represents the data and insights from the research.

12.3. WebSocket Endpoints

To manage these long-running research and visualization processes, the application uses WebSockets, providing a persistent, bidirectional communication channel between the client and server.

- **`/start_research` and `/start_visualization` (POST)**: These endpoints initiate the process. They accept a query, generate a unique `query_id`, and store the query text in a dictionary. This `query_id` is returned to the client, who then uses it to establish a WebSocket connection.
- **`/ws/research/{query_id}` and `/ws/visualization/{query_id}` (WebSocket)**: These are the main WebSocket endpoints. Once a client connects, the corresponding `research_process` or `visualization_process` is invoked. Throughout the process, JSON messages are sent to the client, indicating the current step (e.g., `generating_subqueries`, `searching`, `synthesizing`, `html_generated`) and providing intermediate results. This allows the client to display a real-time progress indicator to the user.

12.4. Deep Search (`/start_deepsearch` and `/ws/deepsearch/{query_id}`)

The deep search functionality follows a similar pattern to the research process but is designed for more in-depth, multi-step investigations. It involves clarifying the user's query, generating subqueries, browsing the internet, and synthesizing the results, all while streaming progress over a WebSocket.

This combination of subquery-based research, LLM-powered synthesis, and real-time WebSocket communication enables `main17.py` to perform powerful, on-demand research tasks and present the findings in a highly engaging and informative manner.

---

13. API Endpoints

The `main17.py` application exposes a comprehensive set of API endpoints using the FastAPI framework. These endpoints serve as the primary interface for client applications to interact with the AI's various functionalities. This section provides a detailed breakdown of each endpoint, its purpose, and its request/response structure.

13.1. Core Interaction Endpoints

- **`POST /generate`** : This is a primary endpoint for generating AI responses. It accepts a `GenerateRequest` containing the user's prompt, session information, and optional filenames for context. It returns a streaming response of the AI-generated text. This endpoint orchestrates context retrieval, LLM interaction, and goal/task command processing.
- **`POST /regenerate`** : Allows a client to request a new response for the last user message in a given session. It takes a `RegenerateRequest` and uses the existing chat history to generate a new answer, providing an alternative to the previous response.
- **`POST /uploadfile`** : Handles the uploading of files (PDF, DOCX, TXT, Python). It processes the files by extracting text, chunking/segmenting, generating embeddings, and storing them in MongoDB and FAISS. This makes the file content available for multimodal retrieval in subsequent interactions.

13.2. WebSocket Endpoints

- **`WS /nlp`** : A WebSocket endpoint for real-time, bidirectional natural language processing. It is similar to `/generate` but operates over a persistent connection, allowing for a more interactive and stateful conversational experience. It handles streaming responses and goal/task management commands.
- **`WS /ws/research/{query_id}`** : The WebSocket endpoint for the research process. After a research task is initiated via `/start_research`, the client connects to this endpoint to receive real-time updates on subquery generation, internet browsing, and content synthesis.
- **`WS /ws/visualization/{query_id}`** : Similar to the research endpoint, this WebSocket is used for the visualization process. It streams progress updates and delivers the final generated HTML code for the visualization.
- **`WS /ws/deepsearch/{query_id}`** : The WebSocket for the deep search functionality, providing real-time feedback as the AI conducts a multi-step investigation.

13.3. Research and Visualization Initiation

- **`POST /start_research`** : Initiates a research task. It accepts a `ResearchQuery` and returns a unique `query_id` that the client uses to connect to the `/ws/research/{query_id}`

WebSocket.

- **POST /start_visualization** : Kicks off a visualization task. It takes a `Prompt` and returns a `query_id` for the `/ws/visualization/{query_id}` WebSocket.
- **POST /start_deepsearch** : Starts a deep search task, accepting a `DeepSearchRequest` and returning a `query_id` for the corresponding WebSocket.

13.4. User and Goal Management

- **POST /subscribe** : Registers a user for push notifications. It accepts a `Subscription` object containing the user's ID, push subscription details, and timezone.
- **POST /Plan_my_week** : Generates a detailed weekly plan for a user based on their active goals and tasks. It takes a `PlanWeekRequest` and returns a structured JSON object representing the plan.
- **GET /get-goals** : Retrieves all goals and tasks for a specific `user_id` . It returns a list of goal objects with their nested tasks.
- **GET /get-session-history** : Fetches the chat history for a given `user_id` and `session_id` .

13.5. Content and Utility Endpoints

- **POST /browse** : A simple endpoint for performing internet searches. It takes a `BrowseRequest` and returns the search results.
- **GET /get-quote** : Generates a personalized, single-line quote for a user based on their long-term memory summary.
- **GET /recommended-content** : Provides a list of recommended content (summaries of relevant topics) for a user, derived from their long-term memory.
- **POST /send-otp** : Sends a One-Time Password to a user's email address for verification purposes.
- **POST /verify-otp** : Verifies an OTP submitted by a user.
- **POST /instapost** : Generates content for an Instagram post based on a prompt, including seed keywords, trending hashtags, SEO keywords, and a caption.
- **POST /analyze-images** : An endpoint for multimodal analysis. It accepts a text query and up to three images, using a vision-capable LLM to provide a response based on the combined text and image input.

13.6. Global Exception Handler

- **@app.exception_handler(Exception)** : This is not an endpoint but a global handler that catches any unhandled exceptions that occur during request processing. It logs the

error and returns a generic `500 Internal Server Error` response, ensuring that the application does not crash and provides a consistent error message to clients.

This well-defined set of API endpoints provides a clear and powerful interface for clients to leverage the full range of the AI assistant's capabilities, from simple conversational queries to complex, long-running research and visualization tasks.

---

14. Error Handling and Logging

Robust error handling and comprehensive logging are critical for the stability, maintainability, and debugging of any complex application. `main17.py` incorporates several mechanisms to manage errors gracefully and provide insightful logs.

14.1. Global Exception Handler

FastAPI provides a powerful way to handle exceptions globally. The application defines a global exception handler that catches any unhandled `Exception` that occurs during request processing. This prevents the server from crashing and ensures a consistent error response is returned to the client.

Python

```
@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    logging.error(f"Unhandled error: {exc}", exc_info=True)
    return JSONResponse(status_code=500, content={"detail": "Internal server
error, please try again later."})
```

- **Purpose**: To catch any unexpected errors that are not handled by specific `try-except` blocks within the application logic.
- **Logging**: When an unhandled exception occurs, it is logged at the `ERROR` level, including traceback information (`exc_info=True`), which is invaluable for diagnosing the root cause.
- **Client Response**: A generic `JSONResponse` with a `500 Internal Server Error` status code and a user-friendly message is returned. This prevents sensitive internal error details from being exposed to the client.

14.2. HTTP Exception Handling

For anticipated error conditions, FastAPI's `HTTPException` is used. This allows the application to explicitly raise HTTP errors with specific status codes and detail messages,

which are then automatically handled by FastAPI and converted into appropriate HTTP responses.

- **Example**: In the `/send-otp` endpoint, if email sending fails, an `HTTPException` with status code 500 is raised:
- **Example**: In the `/verify-otp` endpoint, if an OTP is invalid or expired, an `HTTPException` with status code 400 is raised:

14.3. Logging Strategy

The application employs a consistent logging strategy to record events, debug information, warnings, and errors. Both Python's standard `logging` module and `loguru` are utilized.

- **Basic Configuration**: At the application startup, basic logging is configured to output `INFO` level messages and above to the console.
- **`loguru` Integration**: The `loguru` library is imported and used in several places (e.g., `logger.info`, `logger.warning`, `logger.error`). `loguru` offers enhanced logging capabilities, including colored output, easy configuration, and structured logging, which can be beneficial for production environments.
- **Contextual Logging**: Log messages often include contextual information such as `user_id`, `session_id`, `filename`, and function names. This helps in tracing issues back to specific user interactions or code paths.
- **Error Details**: When logging exceptions, `exc_info=True` is frequently used to include the full traceback, which is crucial for detailed error analysis.
- **Warning Messages**: `logging.warning` is used for non-critical issues that might indicate potential problems but do not halt execution, such as AST parsing failures or unexpected LLM responses.

14.4. Specific Error Handling Examples

- **File Processing**: During file uploads, if text extraction fails for a specific file, an error is logged, and the response for that file indicates failure, allowing other files in the batch to be processed.
- **LLM API Calls**: All interactions with Groq API are wrapped in `try-except` blocks to catch potential `GroqError` or other network/API-related exceptions. If an error occurs, it's logged, and a fallback message or behavior is often provided.
- **Database Operations**: MongoDB operations are also enclosed in `try-except` blocks to handle potential database connection issues or query failures.
- **WebSocket Disconnections**: The WebSocket endpoints specifically handle `WebSocketDisconnect` exceptions, logging when a client disconnects gracefully or

unexpectedly.

By implementing these error handling and logging practices, `main17.py` aims to be a resilient application that can gracefully recover from errors, provide clear insights into its operation, and facilitate quick debugging when issues arise.

---

15. Conclusion

The `main17.py` application represents a sophisticated and multifaceted AI assistant built upon the FastAPI framework. It seamlessly integrates a wide array of advanced technologies, including Groq's powerful language models, MongoDB for robust data persistence, FAISS for efficient vector search, and Web Push for proactive user engagement. The modular architecture, characterized by distinct layers for data handling, AI integration, file processing, and user interaction, ensures scalability, maintainability, and high performance.

Key strengths of this application include its ability to:

- **Understand and Respond Contextually**: By leveraging multimodal retrieval from user-uploaded documents and code, coupled with a long-term memory system, the AI provides highly personalized and context-aware responses.
- **Automate Goal and Task Management**: Users can define, track, and manage their objectives through natural language commands, with the AI intelligently assisting in planning and progress monitoring.
- **Conduct Intelligent Research**: The research and visualization modules enable the AI to autonomously gather information from the internet, synthesize findings, and present them in interactive HTML formats, offering real-time updates via WebSockets.
- **Proactive User Engagement**: The push notification system ensures timely reminders, daily check-ins, and proactive support, enhancing user productivity and goal attainment.
- **Robust and Resilient Operation**: Comprehensive error handling, detailed logging, and strategic rate-limiting for external API calls contribute to the application's stability and reliability.

In summary, `main17.py` is a powerful example of how modern AI and web technologies can be combined to create an intelligent, empathetic, and highly functional assistant. Its design principles prioritize efficiency, user experience, and the ability to adapt to complex user needs, making it a valuable tool for personal and professional productivity.
