GEMINI TELEGRAM BOT

```
# Import necessary modules

import os  # Provides functions for interacting with the operating system
import io  # Handles I/O operations (e.g., working with in-memory streams)
import docx  # Allows reading and writing Microsoft Word (.docx) files
import time  # Provides time-related functions (e.g., delays, timestamps)
import PyPDF2  # Enables reading and manipulating PDF files
import asyncio  # Supports asynchronous programming with coroutines
import logging  # Handles logging for debugging and monitoring
import PIL.Image
# Allows image processing using the Python Imaging Library (Pillow)
from datetime import datetime  # Deals with date and time operations
from textblob import TextBlob
# Performs NLP tasks like sentiment analysis
from pymongo import MongoClient  # Connects to and interacts with MongoDB
from serpapi import GoogleSearch
# Fetches search results from Google using SerpAPI
from pyrogram import Client, filters
# Handles Telegram bot interactions using Pyrogram
from pyrogram.types import Message, ReplyKeyboardMarkup, KeyboardButton  #
Telegram-specific UI elements
from pyrogram.enums import ParseMode  # Defines text formatting options
for Telegram (Markdown, HTML, etc.)
from pyrogram.errors import FloodWait
# Handles rate-limiting errors in Telegram bots
import google.generativeai as genai
# Provides access to Google's Generative AI models
from config import (
# Imports bot credentials and API keys from a configuration file
    API_ID, API_HASH, BOT_TOKEN, GOOGLE_API_KEY, MODEL_NAME, SERP_API_KEY
)

# Set up logging configuration to monitor bot activity

logging.basicConfig(level=logging.INFO)  # Set logging level to INFO
logger = logging.getLogger(__name__)  # Create a logger instance

# Initialize the Pyrogram Client for the Telegram bot

app = Client(
    "gemini_session",  # Name of the session
    api_id=API_ID,  # Telegram API ID (from config)
    api_hash=API_HASH,  # Telegram API Hash (from config)
```

```
    bot_token=BOT_TOKEN,  # Telegram Bot Token (from config)
    parse_mode=ParseMode.MARKDOWN  # Enables Markdown formatting in
messages
)

# Validate that the Google API key is set before proceeding

if not GOOGLE_API_KEY:
    logger.error("GOOGLE_API_KEY is missing! Please check your
config.")  # Log an error if API key is missing
    exit(1)  # Terminate the script

# Configure Google Generative AI with the provided API key

genai.configure(api_key=GOOGLE_API_KEY)

# Initialize the AI model with the specified model name

model = genai.GenerativeModel(MODEL_NAME)
```

```
# MongoDB Connection

MONGO_URI = "your-mongodb-url"
client = MongoClient(MONGO_URI, serverSelectionTimeoutMS=5000)  #
Initialize MongoDB client with a 5-second timeout

# Verify MongoDB connection

try:
    client.admin.command('ping')  # Sends a "ping" command to check if the
database is reachable
    logger.info("Connected to MongoDB")  # Logs successful connection
except Exception as e:
    logger.error(f"MongoDB connection failed: {e}")  # Logs error message
if connection fails
    exit(1)  # Exits the script if the database connection cannot be
established




# Define MongoDB database and collections
```

```
db = client["telegram_bot"]  # Select the 'telegram_bot' database
users_collection = db["users"]  # Collection for storing user details
chat_history_collection = db["chat_history"]
# Collection for storing chat history
file_metadata_collection = db["file_metadata"]
# Collection for storing metadata of uploaded files
websearch_history_collection = db["websearch_history"]
# Collection for storing web search history
sentiment_history = db["sentiment_history"]
# Collection for storing sentiment analysis results

# Bot Commands Help Text

HELP_TEXT = """
🤖 **Bot Commands:**

🚀 **/start** - Start the bot and register yourself.
💬 **/text <prompt>** - Generate AI-powered responses for any query.
🖼️ **/img** - Analyze and describe images using AI.
📄 **/file** - Upload a document to get a summarized version.
📊 **/sentiment <text>** - Analyze the sentiment of the given text.
🌍 **/websearch <query>** - Search the web and get top results
instantly.
ℹ️ **/help** - View this command list anytime.
"""

# Decorator function to append the help message after
executing a command

def append_help(func):
    async def wrapper(client: Client, message: Message):
        await func(client, message)  # Execute the original function
        await message.reply_text(HELP_TEXT)
      # Send the help text after executing the command
    return wrapper

# Function to analyze sentiment of a given text

def get_sentiment(text):
    analysis = TextBlob(text)  # Create a TextBlob object to analyze text
    polarity = analysis.sentiment.polarity  # Extract the polarity score
(-1 to 1)

    # Determine sentiment based on polarity score
    if polarity > 0:
```

```
        return "positive"
      # Positive sentiment if polarity is greater than 0
    elif polarity < 0:
        return "negative"  # Negative sentiment if polarity is less than 0
    else:
        return "neutral"  # Neutral sentiment if polarity is exactly 0


# Command handler for the /start command

@app.on_message(filters.command("start"))
@append_help  # Adds the help message after executing this command
async def start_handler(client: Client, message: Message):
    try:
        user_id = message.from_user.id # Extract the user ID
        user_data = users_collection.find_one({"chat_id": user_id})
      # Check if user exists in the database

        if not user_data:
      # If the user is not found, register them in the database
            users_collection.insert_one({
                "chat_id": user_id,
                "first_name": message.from_user.first_name,
                 # Store user's first name
                "last_name": message.from_user.last_name,
                 # Store user's last name
                "username": message.from_user.username,
                 # Store Telegram username
                "phone_number": None
                 # Placeholder for phone number (to be updated later)
            })

            # Create a keyboard button to request the user's phone number
            keyboard = ReplyKeyboardMarkup(
                [[KeyboardButton("Share Phone Number",
request_contact=True)]],  # Button for phone sharing
                resize_keyboard=True
            # Resize the keyboard for better display
            )

            # Send a welcome message prompting the user to share their
phone number
            await message.reply_text("Welcome! Please share your phone
number.", reply_markup=keyboard)
        else:
```

```
            # If user is already registered, greet them
            await message.reply_text("Welcome back!")

    except Exception as e:
        logger.error(f"Error in start_handler: {e}")  # Log any errors
that occur


# Handler for when user shares their contact (phonenumber)

@app.on_message(filters.contact)
async def save_phone_number(client: Client, message: Message):
    try:
        user_id = message.from_user.id  # Get the user ID
        contact_number = message.contact.phone_number  # Extract phone
number from the contact message

        # Update the user's phone number in the database
        users_collection.update_one({"chat_id": user_id}, {"$set":
{"phone_number": contact_number}})

        # Confirm successful phone number storage
        await message.reply_text("✅ Your phone number has been saved
successfully!")

    except Exception as e:
        logger.error(f"Error in save_phone_number: {e}")  # Log any errors


# Handler for the /text command, which generates AI
responses

@app.on_message(filters.command("text"))
@append_help  # Adds help text after executing the command
async def gemi_handler(client: Client, message: Message):
    # Send a temporary loading message to inform the user
    loading_message = await message.reply_text("**Generating response,
please wait...**")

    try:
        # Ensure the user provided a prompt
        if len(message.text.strip()) <= 5:
            await message.reply_text("**Provide a prompt after the
command.**")
```

```
            return  # Exit the function if no valid prompt is provided

        # Extract the user prompt (text after the command)
        prompt = message.text.split(maxsplit=1)[1]

        # Generate a response using the AI model
        response = model.generate_content(prompt)

        # Extract the generated text response
        response_text = response.text

        # Perform sentiment analysis on the user's prompt
        sentiment = get_sentiment(prompt)

        # Store the conversation details in MongoDB
        chat_history_collection.insert_one({
            "chat_id": message.from_user.id,  # Store user ID
            "user_query": prompt,  # Store the user's query
            "bot_response": response_text,  # Store the generated response
            "sentiment": sentiment,  # Store the analyzed sentiment
            "timestamp": datetime.utcnow()  # Store the timestamp (UTC
format)
        })

        # Customize the bot's response based on the sentiment of the
prompt
        sentiment_responses = {
            "positive": "😊 Great! Here’s what I came up with:",
            "negative": "😔 I sense that things might be tough. Here's
something that might help:",
            "neutral": "🤔 Here’s the information you requested:"
        }
        response_text = f"{sentiment_responses.get(sentiment,
'')}\n\n{response_text}"

        # Ensure the message length does not exceed Telegram's 4096
character limit
        await message.reply_text(response_text if len(response_text) <=
4000 else response_text[:4000] + "...")

    except Exception as e:
        # Handle errors gracefully and inform the user
        await message.reply_text(f"**An error occurred: {str(e)}**")

    finally:
```

```
        # Delete the loading message once the response is generated
        await loading_message.delete()

# Handler for the /img command, which analyzes an image sent by the user

@app.on_message(filters.command("img") & filters.photo)
@append_help  # Adds help text after executing the command
async def analyze_image(client: Client, message: Message):
    # Send a temporary message to indicate that processing is happening
    processing_message = await message.reply_text("🖼️ **Analyzing image, please wait...**")

    try:
        # Download the image file as in-memory data
        img_data = await client.download_media(message.photo, in_memory=True)

        # Open the image using PIL (Pillow)
        img = PIL.Image.open(io.BytesIO(img_data.getbuffer()))

        # If the user provides a caption, use it as a prompt; otherwise, use a default prompt
        prompt = message.caption or "Describe this image."

        # Generate a response from the AI model using both the prompt and the image
        response = model.generate_content([prompt, img])

        # Extract the AI-generated text response
        response_text = response.text

        # Store metadata about the analyzed image in MongoDB
        file_metadata_collection.insert_one({
            "chat_id": message.from_user.id,  # Store user ID
            "file_name": "photo.jpg",  # Use a default name as Telegram doesn't provide file names for photos
            "file_type": "image",  # Indicate that it's an image file
            "description": response_text,  # Store the generated description of the image
            "timestamp": datetime.utcnow()  # Store the timestamp (UTC format)
        })

        # Send the AI-generated description of the image to the user
```

```
        await message.reply_text(f"🖼️ **Image
Analysis:**\n{response_text}")

    except Exception as e:
        # Handle errors gracefully and notify the user
        await message.reply_text(f"❌ Error analyzing image: {str(e)}")

    finally:
        # Delete the loading message after processing is complete
        await processing_message.delete()


# Handler for the /file command, which analyzes a document
file sent by the user

@app.on_message(filters.command("file") & filters.document)
@append_help  # Adds help text after executing the command
async def analyze_file(client: Client, message: Message):
    # Send a temporary message to indicate that processing is happening
    processing_message = await message.reply_text("📂 **Analyzing file,
please wait...**")

    try:
        file = message.document  # Get file metadata from the message
        file_name = file.file_name  # Extract file name
        file_type = file.mime_type  # Extract file type (MIME type)

        # If the user provides a caption, use it as a prompt; otherwise,
use a default prompt
        prompt = message.caption if message.caption else "Summarize the
contents of this file."

        # Download the file as in-memory data
        file_data = await client.download_media(file, in_memory=True)

        # Initialize extracted_text as an empty string
        extracted_text = ""

        # Check the file type and extract text accordingly
        if file_type == "application/pdf":  # PDF Files
            pdf_reader =
PyPDF2.PdfReader(io.BytesIO(file_data.getbuffer()))
            extracted_text = " ".join([page.extract_text() for page in
pdf_reader.pages if page.extract_text()])
```

```
        elif file_type == "text/plain":  # TXT Files
            extracted_text = file_data.getvalue().decode("utf-8")

        elif file_type in ["application/vnd.openxmlformats-
officedocument.wordprocessingml.document", "application/msword"]:  # DOCX
Files
            doc = docx.Document(io.BytesIO(file_data.getbuffer()))
            extracted_text = "\n".join([para.text for para in
doc.paragraphs])

        else:
            # If the file format is unsupported, return an error message
            extracted_text = "⚠ Unsupported file format. Only PDF, TXT,
and DOCX are supported."

        # Ensure that extracted_text is not empty
        if not extracted_text.strip():
            extracted_text = "⚠ No readable text found in this file."

        # Generate AI response using the extracted text (limited to 4000
characters to avoid excessive length)
        response =
model.generate_content(f"{prompt}\n\n{extracted_text[:4000]}")
        response_text = response.text

        # Store metadata about the analyzed file in MongoDB
        file_metadata_collection.insert_one({
            "chat_id": message.from_user.id,  # Store user ID
            "file_name": file_name,  # Store the file name
            "file_type": file_type,  # Store the file type
            "description": response_text,  # Store the AI-generated
summary or analysis
            "timestamp": datetime.utcnow()  # Store the timestamp (UTC
format)
        })

        # Send the AI-generated summary of the file to the user (limited
to 4000 characters)
        await message.reply_text(f"🗁 **File
Analysis:**\n{response_text[:4000]}")

    except Exception as e:
        # Handle errors gracefully and notify the user
        await message.reply_text(f"❌ Error analyzing file: {str(e)}")
```

```
    finally:
        # Delete the loading message after processing is complete
        await processing_message.delete()
```

```
# Handler for the /sentiment command, which analyzes the sentiment of a given text

@app.on_message(filters.command("sentiment"))
@append_help  # Adds help text after executing the command
async def sentiment_handler(client: Client, message: Message):
    # Check if the user provided text after the command
    if len(message.text.split()) < 2:  # Ensures the command is followed by text input
        await message.reply_text("🔍 **Usage:** /sentiment <text>")  # Prompt user to provide input
        return  # Exit function if no text is provided

    # Extract the text input from the message, excluding the command itself
    text = message.text.split(maxsplit=1)[1]

    # Perform sentiment analysis using TextBlob
    blob = TextBlob(text)  # Create a TextBlob object with the given text
    sentiment = blob.sentiment.polarity  # Extract polarity score (-1 to 1)

    # Determine sentiment category based on polarity score
    if sentiment > 0:
        sentiment_result = "😊 Positive"  # Positive sentiment if polarity is greater than 0
    elif sentiment < 0:
        sentiment_result = "😞 Negative"  # Negative sentiment if polarity is less than 0
    else:
        sentiment_result = "😐 Neutral"  # Neutral sentiment if polarity is exactly 0

    # Store sentiment analysis result in MongoDB for future reference
    sentiment_history.insert_one({
        "chat_id": message.from_user.id,  # Store user ID
        "user_query": text,  # Store the analyzed text input
        "sentiment": sentiment_result,  # Store sentiment category (Positive/Negative/Neutral)
```

```
        "timestamp": datetime.utcnow()  # Store timestamp in UTC format
    })

    # Reply to the user with the sentiment analysis result
    await message.reply_text(f"📊 **Sentiment
Analysis:**\n{text}\n\nResult: {sentiment_result}")
```

 # Handler for the /websearch command, which performs a web search using the SERP API and generates a summary.

```
@app.on_message(filters.command("websearch"))
@append_help  # Adds help text after executing the command
async def web_search(client: Client, message: Message):
    # Check if the user provided a query after the command
    if len(message.text.split()) < 2:  # Ensures the command is followed
by a query
        await message.reply_text("🔍 **Usage:** /websearch
your_query")  # Prompt user to provide a query
        return  # Exit function if no query is provided

    query = message.text.split(maxsplit=1)[1]
      # Extract the query from the message
    loading_msg = await message.reply_text("🔎 Searching the web...")
      # Send a loading message while searching

    # Perform sentiment analysis on the search query using the
get_sentiment function
    sentiment = get_sentiment(query)

    try:
        # Initialize the SERP API client with the provided query, API key
        search = GoogleSearch({"q": query, "api_key": SERP_API_KEY})

        # Perform the search and retrieve top 3 organic results
        results = search.get_dict().get("organic_results", [])[:3]
      # Get top 3 search results

        # Check if results are found
        if not results:
            await message.reply_text("❌ No results found.")
      # Inform the user if no results were found
            return  # Exit the function if no results are found
```

```
        # Construct the response text with titles and links for the top 3
results
        response_text = "\n".join([f"🔷
[{r['title']}]({r['link']})\n{r['snippet']}" for r in results])

        # Generate a summary of the search results using Gemini AI model
        summary_prompt = "Summarize the following web search results:\n" +
response_text  # Prepare prompt for AI summary
        ai_summary = model.generate_content([summary_prompt])  # Generate
content using the AI model
        summary_text = ai_summary.text.strip()  # Extract the generated
summary text

        # Construct the final response with search results and AI-
generated summary
        response = f"🌍 **Top Search
Results:**\n\n{response_text}\n\n**Gemini AI Summary:**\n{summary_text}"

        # Store the query, bot response, and sentiment in MongoDB for
future reference
        websearch_history_collection.insert_one({
            "chat_id": message.from_user.id,  # Store user ID
            "user_query": query,  # Store the search query
            "bot_response": response,  # Store the bot's response (search
results and summary)
            "sentiment": sentiment,  # Store sentiment of the query
            "timestamp": datetime.utcnow()  # Store the timestamp of the
query
        })

        # Customize the response based on sentiment analysis (positive,
negative, or neutral)
        sentiment_responses = {
            "positive": "🎉 Great! Here are some useful links:",  #
Positive sentiment response
            "negative": "😟 It seems like you have concerns. These
results might help:",  # Negative sentiment response
            "neutral": "🔍 Here are the search results you requested:"  #
Neutral sentiment response
        }

        # Final response combining sentiment and search results
        response = f"{sentiment_responses.get(sentiment,
'')}\n\n{response_text}\n\n**Gemini AI Summary:**\n{summary_text}"
```

```
        # Send the final response to the user with search results and
summary
        await message.reply_text(response)

    except Exception as e:
        # Handle any exceptions (e.g., API issues, connection errors)
        await message.reply_text(f"⚠ Error: {str(e)}")  # Inform the
user about the error

    finally:
        # Delete the loading message after the processing is complete
        await loading_msg.delete()


# Run the bot if this script is executed directly

if __name__ == "__main__":
    logger.info("Starting the bot...")  # Log message to indicate the bot
is starting
    app.run()  # Start the Pyrogram bot
```

Documenting How ChatGPT Was Used to Develop the Project

1. **Idea Generation and Brainstorming**

- ChatGPT was used to generate and refine ideas for various features and components of the Telegram bot. As the bot was designed to perform multiple tasks (such as text generation, sentiment analysis, image description, and file summarization), ChatGPT helped provide detailed concepts for each feature.

- **Feature Planning:** ChatGPT helped define and refine the bot's capabilities based on user input, providing insights on how to organize the command structure and what AI functionalities to prioritize.
- **Bot Command Design:** The chatbot commands (like /start, /text, /img, /file, /sentiment, /websearch) were brainstormed with the help of ChatGPT, which also suggested ways to structure the commands for better user experience.

2. **Pseudocode and Algorithm Design**

- Whenever complex features needed to be implemented, ChatGPT assisted in developing pseudocode and high-level algorithms.

- **Clear Structure:** ChatGPT provided clear, logical steps for implementing algorithms, which helped in breaking down the tasks into manageable chunks.
- **Efficient Code Implementation:** By suggesting how to structure loops, conditionals, and function calls, ChatGPT made it easier to implement the necessary code efficiently, reducing development time.

3. **Code Snippets and Examples**

- ChatGPT was actively consulted for specific code snippets, such as how to implement text generation using the Gemini model, extract text from a PDF using PyPDF2, or perform sentiment analysis using TextBlob.

- **Code Writing:** ChatGPT suggested optimal ways to write functions and integrate third-party libraries (like PIL, TextBlob, and PyPDF2).
- **Debugging Help:** Whenever there was a bug or issue with a specific function, ChatGPT helped by offering debugging strategies and code fixes, which accelerated the process of making the project functional.

4. Documentation and Commenting

- ChatGPT was used to generate clear and concise documentation for the codebase, helping to explain the purpose of different functions, commands, and integrations.
- It provided documentation on how each feature works, such as the interaction flow for each bot command, the AI model's functionality, or the sentiment analysis pipeline.

- **Automated Documentation:** ChatGPT produced helpful comments and docstrings for functions, reducing the manual effort needed for documenting the code.

- **Consistency in Documentation:** By using a consistent format for commenting the code, ChatGPT helped make the project more maintainable and readable.

6. Testing and Edge Case Handling

- When it came to testing the features of the bot, ChatGPT helped identify possible edge cases and provided testing strategies for different functionalities (e.g., testing the sentiment analysis feature with various types of input, testing file upload capabilities with different formats, or handling long text inputs in queries).
- Additionally, ChatGPT helped suggest test cases for validating the AI model's responses and ensuring that the web search feature returned accurate and relevant results.

- **Edge Case Identification:** ChatGPT assisted in thinking through potential edge cases that could cause issues, such as malformed input, file types that could break the bot, or extreme edge cases in sentiment analysis.
- **Testing Framework:** ChatGPT provided guidance on setting up a basic testing framework for validating different functionalities, which ensured that the bot remained robust during development.

Summary of ChatGPT's Contributions:

1. **Idea Generation & Brainstorming:** Assisted in generating the core ideas and features for the project, ensuring the bot was user-friendly and comprehensive.
2. **Pseudocode & Algorithm Design:** Helped with designing high-level algorithms, turning conceptual ideas into manageable steps for implementation.
3. **Code Snippets & Examples:** Provided practical code snippets for various integrations and features, speeding up development.
4. **Documentation & Commenting:** Streamlined the documentation process by generating useful comments and explanations for the codebase.
5. **Testing & Edge Case Handling:** Contributed to identifying potential edge cases and providing testing strategies to ensure the bot's robustness.
6. **Deployment Strategy & Best Practices:** Assisted in preparing the project for deployment and ensuring it followed industry best practices.

Overall, ChatGPT was an essential tool in accelerating development, streamlining workflows, and ensuring the project was both functional and user-friendly.

❖ MAJOR HIGHLIGHTS

1. Google Generative AI (Gemini Model)

Purpose:

- **Content Generation and Summarization:** The Gemini AI model is used throughout the project to generate content or summaries based on user queries, uploaded images, and files. It adds an intelligent, conversational element to the bot, allowing users to receive meaningful responses, summaries, and descriptions.

Use Cases in the Project:

- **Text Generation for Queries:**
  - When users input a prompt (via `/text`), the Gemini AI model generates a relevant response, creating meaningful content from the provided input. For example, if a user asks a question or provides a prompt, Gemini generates a detailed response based on the query.
- **Image Analysis and Descriptions:**
  - The model generates descriptions for images shared by users. For example, in the `/img` command, when a user sends a photo, the AI analyzes the image and generates a response, describing what is visible in the image.
- **File Content Summarization:**
  - When users upload documents (via `/file`), the AI model is used to generate summaries of the document content. Whether it's a PDF, DOCX, or text file, Gemini processes the document and creates a concise summary for the user.
- **Web Search Result Summarization:**
  - After performing a web search (via `/websearch`), the model generates a summary of the top search results. The AI provides a brief, understandable summary of web pages returned by the Google Search API.

How It Contributes:

- **Enhances User Experience:** The Gemini model's ability to generate human-like responses makes interactions with the bot more engaging and informative.
- **AI-Driven Analysis:** By incorporating generative AI, the bot can provide personalized, context-aware responses, even summarizing complex information like web search results and document contents.
- **Time Efficiency:** The use of AI ensures that users receive quick, accurate answers and summaries without having to manually read through long documents or search results.

2. TextBlob (Sentiment Analysis)

Purpose:

- **Sentiment Analysis:** TextBlob is used to perform sentiment analysis on user-provided text (queries or messages). It helps determine whether the sentiment of the input is positive, negative, or neutral.

Use Cases in the Project:

- **Sentiment Detection for Text Queries:**
  - In commands like `/sentiment`, TextBlob analyzes the sentiment of a given text. For example, if a user provides a statement or a piece of text, the bot analyzes it to determine whether the sentiment is positive, negative, or neutral.
- **Sentiment Analysis for Web Search Queries:**
  - When users perform a web search, TextBlob analyzes the sentiment of the search query. This sentiment analysis helps to customize the response by adding a personalized touch, such as providing a positive tone for a happy query or offering helpful links for a negative sentiment query.
- **Sentiment Analysis of AI-Generated Content:**
  - The bot also analyzes the sentiment of generated content to adjust the tone of responses. For example, it can adapt its responses to be more empathetic when dealing with negative sentiments, or offer a more enthusiastic tone for positive queries.

How It Contributes:

- **Personalized Interactions:** Sentiment analysis allows the bot to provide responses that align with the user's emotional state, making interactions feel more human and personalized.
- **Improved User Engagement:** By adjusting the tone of responses based on sentiment, the bot can ensure that the user experience is more appropriate, empathetic, and effective.
- **Better Content Moderation:** Sentiment analysis can help the bot filter or adjust content that may not be appropriate or may need a more cautious response.

3. SERP API (Google Search API)

Purpose:

- **Web Search Integration:** The SERP API is used to perform real-time searches on the web based on the user's query. It retrieves organic search results, providing the bot with links, titles, and snippets from the top web pages.

Use Cases in the Project:

- **Searching for Information:**
  - When a user sends a query using the `/websearch` command, the bot uses the SERP API to fetch relevant search results. The search results are then presented to the user, along with an AI-generated summary of the findings.

How It Contributes:

- **Real-Time Data Retrieval:** The SERP API provides the bot with up-to-date web data, which makes the bot capable of responding to user queries with the most recent and relevant information available.
- **Enhances the Scope of the Bot:** By integrating web search functionality, the bot can provide answers to a wider range of questions, especially those that go beyond the bot's in-built knowledge.

4. PyPDF2 (PDF Text Extraction)

Purpose:

- **Text Extraction from PDF Files:** PyPDF2 is used to extract text from PDF documents that are uploaded by users.

Use Cases in the Project:

- **PDF Document Processing:**
  - When users upload PDF files through the `/file` command, PyPDF2 reads and extracts text from the file to allow the bot to summarize the content or perform further analysis.

How It Contributes:

- **File Handling Capabilities:** The bot can handle various file types, especially PDFs, which are common for documents. This allows users to interact with the bot without worrying about file formats, ensuring ease of use.
- **Content Extraction:** By extracting text from PDFs, the bot can analyze and summarize documents, providing valuable insights without requiring users to manually read lengthy content.

5. Pillow (Image Processing)

Purpose:

- **Image Analysis and Handling:** Pillow (PIL) is used to process and manipulate images sent by users. The library helps in opening, resizing, and handling image files.

Use Cases in the Project:

- **Image Analysis:**
  - The bot uses Pillow to handle images that users send in the `/img` command. The images are processed and analyzed to generate descriptions, summaries, or insights.

How It Contributes:

- **Image Recognition and Description:** By integrating image processing, the bot can provide detailed descriptions of images, making it more versatile in handling diverse input from users (e.g., text and images).

Summary of AI Tool Contributions to the Project:

- **Generative AI (Gemini Model)** enhances the bot's ability to understand and generate human-like responses, summaries, and content across a variety of inputs (text, images, files, web searches).
- **TextBlob** ensures the bot can analyze the sentiment of both user queries and generated content, allowing it to respond appropriately to the user's emotional tone.
- **SERP API** empowers the bot to perform real-time web searches and deliver current, relevant information, enhancing its knowledge base.
- **PyPDF2** provides the ability to extract and analyze text from PDF files, allowing the bot to summarize and understand documents.
- **Pillow (PIL)** enables the bot to process and describe images, expanding its capabilities beyond just text interactions.

By combining these AI tools, the project leverages a powerful and flexible approach to delivering interactive, intelligent, and personalized user experiences.