ABSTRACT

The **Multi-Modal Query Processing and Knowledge Retrieval System** is a comprehensive information aggregation platform designed to enhance user engagement with diverse sources of knowledge and multimedia content. This system integrates multiple query inputs, including textbased and voice-activated commands, to facilitate seamless access to information from popular sources such as YouTube, Wikipedia, Google, and News APIs. By leveraging natural language processing, speech recognition, and multimedia retrieval techniques, the system provides an adaptive and user-friendly interface for efficient, multi-channel searches.

This project aims to provide a centralized tool that allows users to gather data, summarize information, and interact with multimedia content. Users can retrieve concise or detailed summaries from Wikipedia in various languages, explore relevant videos on YouTube, perform Google searches for web content, and access news articles within specific time frames. A text-tospeech module enables users to listen to retrieved summaries, while PDF export functionality allows saving chat histories for future reference.

The platform's multi-modal capabilities streamline the search process, making it highly useful for students, researchers, and professionals who require quick access to reliable information across multiple formats. The project's architecture emphasizes scalability, efficiency, and usability, offering a robust framework that supports personalized and context-aware information retrieval. The **Multi-Modal Query Processing and Knowledge Retrieval System** thus addresses the growing need for a dynamic and unified search tool, tailored to meet the requirements of diverse user groups in academia, research, and daily information-seeking tasks.

Table of Contents

Table of Contents

Chapter-1 Introduction

1.1 Overview of the Project

The Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS) is an innovative and comprehensive platform designed to revolutionize the way users access and interact with information. In today's digital era, data is scattered across various sources and formats, including text, audio, video, and images. The MMQPKRS addresses this fragmentation by integrating **multiple data source**s and **modalities** into a single, unified system.

The system leverages **APIs**, **AI models**, and **web scraping techniques** to provide users with seamless access to information from diverse sources such as:

- ✓ **Wikipedia:** For detailed summaries and knowledge retrieval.
- ✓ **Google:** For web search results.
- ✓ **YouTube:** For video search and playback.
- ✓ **News Articles:** For up-to-date information on current events.
- ✓ **Social Media Platforms:** For data extraction from LinkedIn and Instagram.
- ✓ **NASA Databases:** For space exploration and astronomy data.

The MMQPKRS is designed to simplify the process of accessing and processing multimodal data, making it an invaluable tool for a wide range of users, including students, researchers, educators, and general enthusiasts. By combining text, voice, and image-based queries, the system offers a rich and interactive user experience.

Key Features of MMQPKRS:

1. **Unified Interface:** A single platform to access information from multiple sources.
2. **Multimodal Query Processing:** Support for text, voice, and image-based queries.
3. **AI-Powered Knowledge Retrieval:** Intelligent responses and summarization using AI models like Mistral.
4. **Custom Shortcuts:** Users can add and manage their favorite resources.
5. **Real-Time Results:** Instant access to up-to-date information from various APIs.
6. **Data Visualization:** Tools for analyzing and visualizing retrieved data.
7. **Downloadable Outputs:** Users can save results in formats like PDF, MP3, and ZIP.

Applications:

The MMQPKRS is a versatile tool with applications in:

- ➢ **Education:** Students and educators can access academic content, generate summaries, and watch educational videos.

- **Research:** Researchers can retrieve and analyze data from multiple sources, including news articles and social media.
- **Entertainment:** Users can explore movies, music, and trending videos on YouTube.
- **Business:** Professionals can generate logos, images, and stay updated with the latest news.
- **Personal Use:** Individuals can explore their interests, from space exploration to social media trends, in a single platform.

1.2 Problem Statement

In today's digital age, the demand for accessing information from multiple sources and in various formats—such as **text, audio, video, and images**—has grown exponentially. Users, including students, researchers, professionals, and general enthusiasts, often need to retrieve and process information from diverse platforms like **Wikipedia, Google, YouTube, news websites, social media, and specialized databases** (e.g., NASA). However, existing systems and tools are typically limited in the following ways:

- **Single Modality Focus:** Most platforms are designed to handle only one type of data (e.g., text-based search engines or video platforms like YouTube). This forces users to switch between multiple tools to access different types of information, leading to inefficiency and a fragmented user experience.
- **Lack of Integration:** There is no unified system that integrates multiple data sources and modalities into a single platform. Users must manually aggregate information from different sources, which is time-consuming and error-prone.
- **Limited Customization:** Existing tools often lack the flexibility to allow users to customize their search preferences or save frequently accessed resources for quick retrieval.
- **Poor Accessibility:** Users with specific needs, such as voice-based queries or text-to-speech capabilities, often find it challenging to interact with traditional systems that do not support multimodal inputs.
- **Inefficient Knowledge Retrieval:** Without AI-powered summarization or intelligent processing, users are often overwhelmed by the volume of information retrieved, making it difficult to extract meaningful insights.

The Challenge:

The fragmentation of tools and platforms creates a significant barrier to efficient information retrieval and processing. Users are forced to navigate multiple interfaces, learn different workflows, and manually compile data from various sources. This not only **wastes time** but also **reduces productivity** and **frustrates users.**

How MMQPKRS Addresses the Problem?

The Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS) is designed to address these challenges by providing a unified multimodal platform that integrates diverse data sources and processing capabilities into a single system. Key solutions offered by MMQPKRS include:

1. **Integration of Multiple Data Sources:** Combines information from Wikipedia, Google, YouTube, news APIs, social media platforms, and NASA databases into one platform.
2. **Support for Multimodal Queries:** Allows users to interact with the system using text, voice, and image-based inputs.
3. **AI-Powered Knowledge Retrieval:** Utilizes advanced AI models (e.g., Mistral) to provide intelligent responses, summarization, and data analysis.
4. **Customizable Interface:** Enables users to add custom shortcuts and manage their preferences for quick access to frequently used resources.
5. **Efficient Data Processing:** Streamlines the retrieval and processing of information, reducing the time and effort required by users.
6. **Enhanced User Experience:** Provides a seamless and intuitive interface that simplifies the process of accessing and interacting with multimodal data.

1.3 Objectives of the Project

The Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS) is designed with a clear set of objectives to address the challenges of fragmented information retrieval and provide users with a seamless, efficient, and intelligent platform. The primary objectives of the project are:

1. Integration of Multiple Data Sources

Objective: Combine information from diverse and reliable sources into a single platform.
Details:

- Integrate data from **Wikipedia** for detailed summaries and knowledge retrieval.
- Incorporate **Google Search** for comprehensive web search results.
- Enable **YouTube** video search and playback for multimedia content.
- Fetch up-to-date information from **news APIs.**
- Extract data from **social media platforms** like LinkedIn and Instagram.
- Explore **NASA databases** for space and astronomy-related information.

Outcome: Users can access information from multiple sources without switching between platforms.

2. Multimodal Query Processing

Objective: Support multiple input modalities for enhanced user interaction.
Details:

- Allow **text-based queries** for traditional search.
- Enable **voice-based queries** for hands-free interaction using speech recognition.
- Support **image-based queries** for visual search and analysis.

Outcome: Users can interact with the system in the way that is most convenient for them, improving accessibility and usability.

3. AI-Powered Knowledge Retrieval

Objective: Utilize AI models to provide intelligent responses and summarization.
Details:

- Implement **Mistral AI** for conversational AI and intelligent responses.
- Use **summarization techniques** to condense large volumes of information into concise summaries.
- Provide **context-aware responses** based on user queries.

Outcome: Users receive accurate, relevant, and concise information, reducing the time spent processing data.

4. User-Friendly Interface

Objective: Develop an intuitive and responsive interface for seamless user interaction.
Details:

- Use **Streamlit** to create a web-based interface that is easy to navigate.
- Design a clean and visually appealing layout.
- Ensure responsiveness across devices (desktop, tablet, mobile).

Outcome: Users can interact with the system effortlessly, regardless of their technical expertise.

5. Customization and Extensibility

Objective: Allow users to personalize the system to suit their needs.
Details:
Enable users to add custom shortcuts for frequently accessed resources.
Provide options to manage preferencZes, such as language, search filters, and output formats.
Allow users to save and organize their search results.
Outcome: Users can tailor the system to their specific requirements, enhancing productivity and user satisfaction.

6. Scalability and Performance

Objective: Ensure the system can handle large volumes of data and user requests efficiently.
Details:
Optimize the system for high performance and low latency.
Design the architecture to scale horizontally to accommodate increasing user demand.
Implement caching mechanisms to reduce redundant API calls and improve response times.
Outcome: The system remains reliable and responsive, even under heavy usage.

Summary of Objectives

The MMQPKRS aims to:

1. **Integrate multiple data sources** into a single platform.
2. **Support multimodal queries** (text, voice, and image) for enhanced interaction.

3. **Leverage AI model**s for intelligent knowledge retrieval and summarization.
4. **Provide a user-friendly interface** for seamless navigation.
5. **Enable customization** to meet individual user needs.
6. **Ensure scalability and performance** for handling large-scale data and user requests.

1.4 Scope of the Project

The **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is designed to provide a comprehensive and versatile platform for accessing, processing, and retrieving information from multiple sources and modalities. The scope of the project encompasses the following key functionalities and features:

1. Real-Time Search Results from Multiple Sources

- The system integrates data from diverse sources, including:
- **Wikipedia:** For detailed summaries and knowledge retrieval.
- **Google:** For web search results.
- **YouTube:** For video search and playback.
- **News APIs:** For up-to-date information on current events.
- **Social Media Platforms:** For data extraction from LinkedIn and Instagram.
- **NASA Databases:** For space exploration and astronomy-related information.
- **Outcome:** Users can access real-time information from multiple sources in a single platform.

2. Support for Multilingual Queries

- The system supports **multilingual queries**, allowing users to retrieve information in their preferred language.
- Example: Wikipedia summaries can be generated in languages such as English, Spanish, Hindi, Chinese, and Telugu.
- **Outcome:** Enhanced accessibility for users across different regions and language preferences.

3. Voice-Based Queries for Hands-Free Interaction

- The system enables **voice-based queries** using speech recognition technology.
- Users can interact with the system without typing, making it ideal for hands-free scenarios.
- **Outcome:** Improved accessibility and convenience for users.

4. AI-Powered Image and Logo Generation

- The system leverages AI models to generate **custom images and logos** based on user prompts.
- Example: Users can input a description like "a futuristic cityscape" or "a logo for a tech startup" to generate visual content.
- **Outcome:** Users can create high-quality visuals for personal, educational, or business purposes.

5. Social Media Data Extraction and Analysis

- The system extracts and analyzes data from **social media platforms** such as LinkedIn and Instagram.
- Features include:
  - Profile data extraction.
  - Image and link extraction.
  - Data visualization and analysis.
- **Outcome:** Users can gain insights from social media data for research, marketing, or personal use.

6. Exploration of Space Data Using NASA APIs

- The system integrates NASA APIs to provide access to space-related data, including:
  - Astronomy Picture of the Day (APOD).
  - Mars Rover photos.
  - Near-Earth Object (NEO) data.
  - Earth imagery.
- **Outcome:** Users can explore and learn about space and astronomy in an interactive way.

7. Downloadable Results in Various Formats

- The system allows users to **download and save results** in multiple formats, including:
  - **PDF:** For text-based summaries and reports.
  - **MP3:** For audio outputs (e.g., text-to-speech conversions).
  - **ZIP:** For bulk downloads of images or other files.
- **Outcome:** Users can save and share retrieved information in their preferred format.

8. Target Users

The MMQPKRS is designed for a wide range of users, including:

- **Students:** For academic research, learning, and project work.
- **Researchers:** For data collection, analysis, and visualization.
- **Educators:** For creating teaching materials and accessing educational resources.
- **General Users:** For personal exploration, entertainment, and staying informed.

Summary of Scope

The MMQPKRS is a **versatile and user-centric platform** that provides real-time access to multimodal information from diverse sources. Its scope includes:

- Real-time search results.
- Multilingual and voice-based queries.
- AI-powered image and logo generation.
- Social media data extraction and analysis.
- Space data exploration using NASA APIs.

- Downloadable results in various formats.

1.5 Applications and Use Cases

The **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is a versatile platform with a wide range of applications across various domains. Its ability to integrate multiple data sources and modalities makes it a valuable tool for **education, research, entertainment, business, and personal use.** Below are the key applications and use cases of the MMQPKRS:

1. Education

➢ **Students:**

- Search for **academic content** from Wikipedia, Google, and other sources.
- Watch **educational videos** on YouTube for better understanding of complex topics.
- Generate **summaries** of lengthy articles or research papers using AI-powered tools.
- Use text-to-speech features to listen to study materials.

➢ **Teachers:**

- Create custom **shortcuts** for frequently used educational resources.
- Prepare teaching materials by retrieving and organizing information from multiple sources.
- Use **AI-generated visuals** to enhance classroom presentations.

2. Research

➢ **Researchers:**

- Retrieve and analyze data from **multiple sources**, including news articles, social media, and academic databases.
- Use **AI-powered summarization** to quickly extract key insights from large volumes of text.
- Explore **space data** using NASA APIs for astronomy and space-related research.
- Generate **visualizations** and reports for presenting research findings.

➢ **Data Analysts:**

- Extract and analyze **social media data** (e.g., LinkedIn, Instagram) for trends and patterns.
- Use **real-time image search** to gather visual data for analysis.

3. Entertainment

➢ **General Users:**

- Search for **movies, music, and trending videos** on YouTube.
- Use the **AI chat feature** for interactive and engaging conversations.
- Explore **space data** and astronomy-related content for personal interest.
- Generate **AI-powered images** for creative projects or personal use.

➢ **Content Creators:**

- Use the system to find inspiration and resources for creating content.
- Generate **logos and visuals** for YouTube channels, blogs, or social media profiles.

4. Business

➢ **Professionals:**

- Generate **logos and images** for marketing and branding purposes using AI-powered tools.
- Stay updated with the **latest trends** and news in their industry using the news search feature.
- Use **social media data extraction** to monitor brand presence and customer sentiment.

➢ **Entrepreneurs:**

- Research market trends and competitor analysis using data from multiple sources.
- Create **custom visuals** for presentations, pitches, and promotional materials.

5. Personal Use

➢ **General Users:**

- Explore personal interests, such as **space exploration, technology, or social media trends**, in a single platform.
- Use **voice-based queries** for hands-free interaction while multitasking.
- Download and save information in preferred formats (e.g., PDF, MP3) for offline use.

➢ **Hobbyists:**

- Use the system to learn new skills, such as cooking, photography, or DIY projects, by accessing tutorials and videos.
- Generate **custom images** for personal projects or creative hobbies.

Summary of Applications and Use Cases

The MMQPKRS is designed to cater to a diverse audience, including:

- **Students and Educators:** For academic research, learning, and teaching.
- **Researchers and Analysts:** For data retrieval, analysis, and visualization.
- **Entertainment Enthusiasts:** For exploring movies, music, and trending content.
- **Business Professionals:** For marketing, branding, and staying updated with industry trends.
- **General Users:** For personal exploration, hobbies, and creative projects.

1.6 Tools and Technologies Used

The **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is built using a robust stack of tools and technologies to ensure seamless functionality, scalability, and user-friendliness. Below is a detailed breakdown of the tools and technologies used in the development of the MMQPKRS:

1. Programming Language

Python:

- The primary programming language used for developing the MMQPKRS.
- Chosen for its simplicity, extensive libraries, and strong support for AI and data processing.

2. Web Framework

Streamlit:

- Used to create the user interface for the MMQPKRS.

- Enables rapid development of interactive and responsive web applications.
- Provides built-in support for data visualization and real-time updates.

3. APIs

YouTube Data API v3:
- Used for searching and retrieving YouTube videos and metadata.

Google Custom Search JSON API:
- Enables web search functionality within the system.

NewsAPI:
- Provides access to news articles from various sources.

Hugging Face API:
- Used for integrating AI models (e.g., Mistral) for conversational AI and summarization.

NASA APIs:
- Used for accessing space-related data, including astronomy pictures, Mars Rover photos, and near-Earth object data.

RapidAPI:
Provides various APIs for:
- YouTube MP3 conversion.
- AI-powered image and logo generation.
- Real-time image search.

4. Libraries

SpeechRecognition:
- Used for **voice search** functionality.

gTTS (Google Text-to-Speech):
- Converts text into speech for audio outputs.

Wikipedia-API:
- Retrieves summaries and data from Wikipedia.

Requests:
- Handles HTTP requests for API calls.

Pandas:
- Used for **data manipulation** and analysis.

Altair:
- Enables **data visualization** for presenting insights.

ReportLab:
- Used for **PDF generation** of search results and summaries.

5. AI Models

Mistral:
- A conversational AI model used for **intelligent responses** and summarization.

Stable Diffusion:
- An AI model used for **image generation** based on user prompts.

6. Deployment
Streamlit Sharing:
- Used for **cloud deployment** of the MMQPKRS.

Docker:
- Used for **containerization** to ensure consistent deployment across environments.

7. Version Control
Git and GitHub:
- Used for **version control** and collaborative development.
- Enables tracking of code changes and collaboration among team members.

8. Environment Management
.env Files:
- Used for managing **API keys** and sensitive configuration data.
- Ensures secure handling of credentials and environment-specific settings.

Summary of Tools and Technologies

The MMQPKRS leverages a combination of **programming languages, frameworks, APIs, libraries, AI models, and deployment tools** to deliver a powerful and user-friendly platform. The key components include:

- **Python** for development.
- **Streamlit** for the user interface.
- **APIs** for integrating diverse data sources.
- **Libraries** for voice search, text-to-speech, data manipulation, and visualization.
- **AI Models** for intelligent responses and image generation.
- **Deployment Tools** like Streamlit Sharing and Docker for cloud deployment.
- **Version Control** with Git and GitHub.
- **Environment Management** using .env files.

Chapter-2 Literature Survey

2.1 Existing Systems and Their Limitations

This section provides a detailed examination of the current systems and tools available for information retrieval, highlighting their key features and limitations. Understanding these limitations is crucial for identifying the gaps that the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** aims to address.

Existing Systems

1. Search Engines (e.g., Google, Bing):

Key Features:
- Provide **text-based search results** from web pages.
- Offer advanced filtering options (e.g., date, region, file type).

Limitations:
- Limited to textual content and do not effectively integrate multimedia (e.g., videos, images).
- Lack support for **voice or image-based** queries.
- Do not provide **AI-powered summarization** or intelligent responses.

2. Video Platforms (e.g., YouTube):

Key Features:
- Focus on **video conten**t and provide a vast library of videos.
- Support video playback, recommendations, and user interactions (e.g., likes, comments).

Limitations:
- Limited to **video content** and do not integrate other data sources like text or images.
- Lack tools for **text-based search** or summarization of video content.
- Do not support **multimodal queries** (e.g., voice or image-based searches).

3. Knowledge Bases (e.g., Wikipedia):

Key Features:
- Offer **detailed textual information** on a wide range of topics.
- Provide multilingual support and references to external sources.

Limitations:
- Limited to **text-based content** and do not support multimedia integration.
- Do not provide **voice or image-based query support.**

- Lack **AI-powered summarization** or conversational capabilities.

4. News Aggregators (e.g., Google News):

Key Features:
Aggregate **news articles** from various sources.
Provide real-time updates and personalized news feeds.

Limitations:
Limited to **text-based news articles** and do not integrate multimedia (e.g., videos, images).
Lack support for **social media data** or other non-news sources.
Do not provide **AI-powered summarization** or intelligent filtering.

5. Social Media Platforms (e.g., LinkedIn, Instagram):

Key Features:
Focus on **user-generated content** (e.g., posts, images, videos).
Provide tools for networking, sharing, and engagement.

Limitations:
Lack data **extraction and analysis tools** for external use.
Limited to **platform-specific content** and do not integrate with other data sources.
Do not support **multimodal queries** or AI-powered features.

6. AI Chatbots (e.g., ChatGPT):

Key Features:
Provide **conversational AI capabilities** for text-based interactions.
Offer intelligent responses and context-aware conversations.

Limitations:
Limited to **text-based interactions** and do not support voice or image-based queries.
Lack integration with **multimedia content** or external data sources.
Do not provide tools for **data visualization** or summarization of external content.

Summary of Limitations

- **Fragmentation:** Users must switch between multiple platforms to access different types of information (text, video, images).
- **Single Modality:** Most systems focus on a single type of data (e.g., text, video) and do not support multimodal queries.
- **Lack of Integration:** No unified platform combines multiple data sources and modalities.
- **Poor Customization:** Limited options for users to personalize their search experience.
- **Inefficient Knowledge Retrieval:** Users are often overwhelmed by the volume of information retrieved, with no tools for summarization or intelligent filtering.

How MMQPKRS Addresses These Limitations

The MMQPKRS is designed to overcome these limitations by:

1. **Integrating Multiple Data Sources:** Combining information from Wikipedia, Google, YouTube, news APIs, social media, and NASA databases.
2. **Supporting Multimodal Queries:** Allowing text, voice, and image-based queries for enhanced interaction.
3. Providing AI-Powered Features: Using AI models for summarization, intelligent responses, and image generation.
4. **Enabling Customization:** Allowing users to create custom shortcuts and manage preferences.
5. **Offering a Unified Platform:** Providing a single interface for accessing and processing multimodal information.

2.2 Research Papers and References

This section reviews the key research papers and references that have significantly contributed to the field of **multimodal systems and knowledge retrieval.** These resources provide the theoretical foundation and practical insights necessary for developing the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).**

Key Research Papers

1. **"Multimodal Machine Learning: A Survey and Taxonomy"** by T. Baltrušaitis, C. Ahuja, and L. Morency.

Summary:
- Discusses the challenges and opportunities in multimodal machine learning.
- Provides a taxonomy for understanding multimodal systems, including data representation, fusion, and alignment.

Relevance to MMQPKRS:
- Helps in designing the system's architecture for integrating multiple modalities (text, voice, images).
- Guides the implementation of multimodal data fusion techniques.

2. **"Attention Is All You Need"** by A. Vaswani et al.

Summary:
- Introduces the Transformer architecture, which revolutionized natural language processing (NLP).
- Focuses on the self-attention mechanism for capturing contextual relationships in data.

Relevance to MMQPKRS:

- Forms the basis for AI-powered components like Mistral, used for conversational AI and summarization.

- Enhances the system's ability to process and generate context-aware responses.

3. **"Deep Speech: Scaling up end-to-end speech recognition"** by A. Hannun et al.

Summary:
- Explores advancements in speech recognition technology.
- Introduces an end-to-end deep learning model for accurate and efficient speech-to-text conversion.

Relevance to MMQPKRS:
- Supports the implementation of voice search functionality.
- Enables hands-free interaction with the system.

4. **"Generative Adversarial Networks (GANs)"** by I. Goodfellow et al.

Summary:
- Introduces Generative Adversarial Networks (GANs), a framework for generating realistic data (e.g., images, audio).
- Consists of a generator and discriminator that compete to improve the quality of generated outputs.

Relevance to MMQPKRS:
- Forms the basis for AI-powered image and logo generation.
- Enables the system to create high-quality visuals based on user prompts.

5. **"A Survey of Knowledge Retrieval Techniques"** by J. Zhang, Y. Li, and Y. Zhang.

Summary:
- Provides an overview of knowledge retrieval methods.
- Highlights the importance of integrating multiple data sources and modalities for efficient information retrieval.

Relevance to MMQPKRS:
- Guides the design of the system's knowledge retrieval framework.
- Emphasizes the need for multimodal integration and AI-powered summarization.

References

1. API Documentation:

YouTube Data API v3:
- https://developers.google.com/youtube/v3

Used for searching and retrieving YouTube videos and metadata.

Google Custom Search JSON API:
- https://developers.google.com/custom-search/v1

Enables web search functionality within the system.

NewsAPI:
- https://newsapi.org/docs

Provides access to news articles from various sources.

Hugging Face API:
- https://huggingface.co/docs

Used for integrating AI models (e.g., Mistral) for conversational AI and summarization.

NASA APIs:
- https://api.nasa.gov/

Used for accessing space-related data, including astronomy pictures, Mars Rover photos, and near-Earth object data.

2. Libraries and Frameworks:

Streamlit:
- https://docs.streamlit.io/

Used for building the user interface of the MMQPKRS.

SpeechRecognition:
- https://pypi.org/project/SpeechRecognition/

Enables voice search functionality.

gTTS (Google Text-to-Speech):
- https://gtts.readthedocs.io/

Converts text into speech for audio outputs.

Wikipedia-API:
- https://pypi.org/project/Wikipedia-API/

Retrieves summaries and data from Wikipedia.

Requests:
- https://docs.python-requests.org/

Handles HTTP requests for API calls.

Pandas:
- https://pandas.pydata.org/docs/

Used for data manipulation and analysis.

Altair:
- https://altair-viz.github.io/

Enables data visualization for presenting insights.

ReportLab:
- https://www.reportlab.com/docs/reportlab-userguide.pdf

Used for PDF generation of search results and summaries.

3. AI Models:

➢ **Mistral:**
A conversational AI model used for intelligent responses and summarization.

➢ **Stable Diffusion:**
An AI model used for image generation based on user prompts.

4. Deployment Tools:

Streamlit Sharing:
- https://streamlit.io/cloud

Used for cloud deployment of the MMQPKRS.

Docker:
- https://www.docker.com/

Used for containerization to ensure consistent deployment across environments.

Version Control:

➢ **Git and GitHub:**
- https://github.com/

Used for version control and collaborative development.

2.3 Comparative Analysis of Similar Systems

This section provides a **comparative analysis of the Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** with existing systems, such as **Google, YouTube, Wikipedia,** and **ChatGPT.** The comparison highlights the unique **advantages** of the MMQPKRS and demonstrates how it addresses the limitations of existing systems.

Comparison Table

Feature	MMQPKRS	Google	YouTube	Wikipedia	ChatGPT
Multimodal Queries	Yes (text, voice, image)	Yes	No (video only)	No (text only)	No (text only)
Integration of Sources	Yes (Wikipedia, Google, YouTube, News, Social Media, NASA)	No (web pages only)	No (videos only)	No (text only)	No (text only)
AI-Powered Features	Yes (summarization, image generation, chat)	Yes	Yes	Yes	Yes (text-based chat)
Customization	Yes (custom shortcuts, preferences)	Limited (basic filters)	No	No	No
Real-Time Results	Yes	Yes	Yes	Yes	Yes

Advantages of MMQPKRS

1. **Unified Platform:**
- Combines **multiple data sources** (Wikipedia, Google, YouTube, news APIs, social media, NASA) into a single system.
- Eliminates the need to switch between different platforms for accessing diverse types of information.

2. **Multimodal Interaction:**
- Supports **text, voice, and image-based queries** for enhanced user interaction.
- Provides a more **intuitive and flexible user experience** compared to systems limited to a single modality.

3. **AI-Powered Features:**
- Offers **intelligent responses** and **summarization** using AI models like Mistral.
- Includes **image generation** capabilities using models like Stable Diffusion.
- Provides **context-aware conversations** and **data analysis** tools.

4. **Customization:**
- Allows users to create **custom shortcuts** for frequently accessed resources.
- Enables users to **manage preferences** (e.g., language, search filters, output formats).
- Provides a **personalized experience** tailored to individual needs.

5. **Versatility:**

Suitable for a wide range of applications, including:
Education: Students and educators can access academic content and generate summaries.
Research: Researchers can retrieve and analyze data from multiple sources.
Entertainment: Users can explore movies, music, and trending videos.
Business: Professionals can generate logos and stay updated with industry trends.
Personal Use: Individuals can explore their interests in a single platform.

6. **Real-Time Results:**

Provides **real-time access** to up-to-date information from various sources.
Ensures that users receive the **latest and most relevant data.**

Comparison with Existing Systems

1. **Google:**

Strengths: Comprehensive web search results, advanced filtering options.
Limitations: Limited to text-based queries and web pages; lacks integration with multimedia or other data sources.
MMQPKRS Advantage: Integrates multiple data sources and supports multimodal queries.

2. **YouTube:**

Strengths: Extensive library of video content, user-friendly interface.
Limitations: Limited to video content; lacks integration with text or image-based data.

MMQPKRS Advantage: Combines video search with text and image-based queries.

3. **Wikipedia:**

Strengths: Detailed textual information on a wide range of topics.
Limitations: Limited to text-based content; lacks support for voice or image-based queries.
MMQPKRS Advantage: Provides AI-powered summarization and supports multimodal queries.

4. **ChatGPT:**

Strengths: Conversational AI capabilities, context-aware responses.
Limitations: Limited to text-based interactions; lacks integration with multimedia or external data sources.
MMQPKRS Advantage: Integrates AI-powered chat with multimodal data sources and features.

2.4 Need for a Multimodal Approach

This section justifies the **need for a multimodal approach** in information retrieval systems and explains how the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** addresses the challenges faced by existing systems. The multimodal approach is essential for meeting the growing demand for **efficient, flexible, and user-friendly** information retrieval.

Challenges Addressed by MMQPKRS

1. **Fragmentation of Tools:**

Current Scenario:
- Users rely on **multiple platforms** (e.g., Google for web search, YouTube for videos, Wikipedia for text) to access different types of information.
- Switching between platforms is **time-consuming** and **inefficient.**

MMQPKRS Solution:
- Integrates **multiple data sources** (Wikipedia, Google, YouTube, news APIs, social media, NASA) into a **single platform.**
- Provides a **unified interface** for accessing diverse types of information.

2. **Limited Interaction Modalities:**

Current Scenario:
- Most systems support only **text-based queries** (e.g., Google, Wikipedia).
- Users cannot interact with systems using **voice or image-based inputs.**

MMQPKRS Solution:
- Supports **multimodal queries** (text, voice, and image-based).
- Enables **hands-free interaction** through voice search.
- Allows users to upload images for **visual search** and analysis.

3. **Inefficient Knowledge Retrieval:**

Current Scenario:
- Users are often **overwhelmed** by the volume of information retrieved.
- Lack of tools for **summarization** or **intelligent filtering.**

MMQPKRS Solution:
- Uses **AI-powered summarization** to provide concise and relevant results.
- Offers **context-aware responses** and **data visualization** tools.
- Reduces information overload by presenting **key insights** in an organized manner.

4. Lack of Customization:

Current Scenario:
- Existing systems offer **limited options** for personalization.
- Users cannot customize their search preferences or save frequently accessed resources.

MMQPKRS Solution:
- Allows users to create **custom shortcuts** for quick access to frequently used resources.
- Provides options to manage preferences (e.g., language, search filters, output formats).
- Enhances **user satisfaction** and **productivity.**

5. Growing Demand for Multimodal Systems:

Current Scenario:
- The increasing availability of **multimodal data** (text, audio, video, images) requires systems that can process and integrate these modalities.
- Existing systems are **not equipped** to handle multimodal data effectively.

MMQPKRS Solution:
- Provides a **unified platform** for processing and retrieving multimodal information.
- Supports **real-time integration** of diverse data sources and modalities.
- Meets the growing demand for **flexible and versatile** information retrieval systems.

Why a Multimodal Approach is Essential

1. **Enhanced User Experience:**
- A multimodal approach allows users to interact with the system in the way that is most convenient for them (e.g., typing, speaking, or uploading images).
- Provides a **more intuitive and engaging** user experience.

2. **Improved Accessibility:**
- Supports **voice-based queries** for users who prefer hands-free interaction.
- Enables **image-based queries** for visual search and analysis.
- Makes the system accessible to a **wider audience**, including users with disabilities.

3. **Efficient Information Processing:**
- Combines **text, voice, and image-based inputs** to provide comprehensive and accurate results.
- Uses **AI-powered tools** for summarization, filtering, and visualization, reducing the time and effort required by users.

4. **Versatility:**
- Suitable for a wide range of applications, including **education, research, entertainment, business, and personal use.**
- Adapts to the needs of different users and use cases.

5. **Future-Proofing:**
- As the volume and variety of data continue to grow, a multimodal approach ensures that the system remains **relevant and scalable.**
- Prepares the system to handle **emerging technologies** and data formats.

Chapter-3 System Design

3.1 System Architecture

The **system architecture** of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is designed to integrate **multiple data sources** and **modalities** while providing a **seamless user experience**. The architecture is modular and scalable, ensuring that the system can handle diverse user queries and deliver accurate, relevant results. Below is a detailed breakdown of the architecture components:

1. User Interface (UI)

Technology: Built using **Streamlit**, a powerful framework for creating interactive web applications.

Features:
- **Responsive Design:** Works seamlessly on desktops, tablets, and mobile devices.
- **Multimodal Input:** Supports **text, voice, and image-based queries.**
- **Interactive Elements**: Includes buttons, sliders, and dropdowns for user interaction.
- **Real-Time Updates:** Displays results dynamically as they are processed.

2. Application Layer

Purpose: Handles user queries and processes them using appropriate modules.

Modules:

a) **Wikipedia Search:**
Retrieves summaries and detailed information from Wikipedia.
Supports multilingual queries.

b) **Google Search:**
Provides web search results using Google Custom Search JSON API.

c) **YouTube Integration:**
Searches and retrieves YouTube videos.
Supports video playback and YouTube-to-MP3 conversion.

d) **News Search:**
Fetches news articles from various sources using NewsAPI.
Allows filtering by date and topic.

e) **AI Chat:**
Provides conversational AI capabilities using Mistral.
Generates intelligent responses and summaries.
Image and Logo Generation:
Generates AI-powered images and logos using Stable Diffusion.

f) **Social Media Integration:**
Extracts and analyzes data from LinkedIn and Instagram.

g) **NASA Space Data Explorer:**
Provides access to space-related data (e.g., APOD, Mars Rover photos) using NASA APIs.

3. API Integration Layer

Purpose: Connects with external APIs for data retrieval and processing.

Integrated APIs:

a) **YouTube Data API v3:**
Used for searching and retrieving YouTube videos.

b) **Google Custom Search JSON API:**
Enables web search functionality.

c) **NewsAPI:**
Provides access to news articles.

d) **Hugging Face API:**
Used for AI-powered summarization and conversational AI.

e) **NASA APIs:**
Provides space-related data (e.g., APOD, Mars Rover photos).

f) **RapidAPI:**
Used for YouTube MP3 conversion, image generation, and real-time image search.

4. Data Processing Layer
Purpose: Processes and analyzes data retrieved from APIs.

Components:

a) **AI-Powered Summarization:**
Uses **Mistral** to generate concise summaries of retrieved data.

b) **Image Generation:**
Uses **Stable Diffusion** to create AI-powered images and logos.

c) **Data Visualization:**
Uses **Altair** to create interactive visualizations of data.

d) **Text-to-Speech Conversion:**
Uses **gTTS** to convert text into speech for audio outputs.

5. Database Layer
Purpose: Stores user preferences, custom shortcuts, and session data.

Technology:
- Uses lightweight databases or local storage for simplicity.
- Example: SQLite or JSON files for storing user data.

Data Stored:
- User preferences (e.g., language, search filters).
- Custom shortcuts (e.g., frequently accessed resources).
- Session data (e.g., recent searches, chat history).

6. Deployment Layer
Purpose: Ensures the system is accessible to users via the cloud.

Technology:

a) **Streamlit Sharing:**
Used for cloud deployment of the MMQPKRS.
Provides a public URL for accessing the application.

b) **Docker:**
Used for containerization to ensure consistency across environments.
Simplifies deployment and scaling.

Key Features of the Architecture
✓ **Modular Design:**
Each component (e.g., Wikipedia Search, YouTube Integration) is independent and can be updated or replaced without affecting the entire system.

✓ **Scalability:**
The architecture is designed to handle increasing user demand by scaling horizontally (e.g., adding more servers).

✓ **Flexibility:**
Supports multiple data sources and modalities, making it adaptable to different use cases.

✓ **User-Centric:**
Focuses on providing a seamless and intuitive user experience.

3.2 Workflow of the System

The workflow of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is designed to provide a **smooth and efficient user experience.** It ensures that user queries are processed accurately and that the results are presented in a clear and interactive manner. Below is a step-by-step breakdown of the workflow:

1. User Input
Step: The user enters a query using **text, voice, or image.**

Details:
Text Input: The user types a query in the search bar (e.g., "What is AI?").
Voice Input: The user speaks the query, which is converted to text using **speech recognition.**
Image Input: The user uploads an image for visual search (e.g., a logo or a picture of an object).

2. Query Processing
Step: The system identifies the type of query and routes it to the appropriate module.

Details:
- The system analyzes the query to determine its type (e.g., text, voice, image) and context.
- Based on the query, the system selects the relevant module:

✓ **Wikipedia Search:** For general knowledge queries.
✓ **Google Search:** For web search results.
✓ **YouTube Integration:** For video-related queries.
✓ **News Search:** For current events and news articles.
✓ **AI Chat:** For conversational queries and summarization.
✓ **Image and Logo Generation:** For generating AI-powered images.
✓ **Social Media Integration:** For extracting data from LinkedIn or Instagram.
✓ **NASA Space Data Explorer:** For space-related queries.

3. API Calls
Step: The system makes API calls to retrieve data from external sources.

Details:
The selected module interacts with the relevant APIs to fetch data:
✓ **YouTube Data API v3:** For video search and metadata.
✓ **Google Custom Search JSON API:** For web search results.
✓ **NewsAPI:** For news articles.
✓ **Hugging Face API:** For AI-powered summarization and chat.
✓ **NASA APIs:** For space-related data.
✓ **RapidAPI:** For YouTube MP3 conversion, image generation, and real-time image search.

The retrieved data is passed to the **Data Processing Layer** for further analysis.

4. Data Processing
Step: The retrieved data is processed using AI models for summarization, image generation, or visualization.

Details:
AI-Powered Summarization:
✓ Uses Mistral to generate concise summaries of articles, videos, or other content.
Image Generation:
✓ Uses Stable Diffusion to create AI-powered images or logos based on user prompts.
Data Visualization:
✓ Uses Altair to create interactive charts and graphs for data analysis.
Text-to-Speech Conversion:
✓ Uses **gTTS** to convert text into speech for audio outputs.

5. Output Generation
Step: The system generates outputs in various formats based on the processed data.

Details:
Text Summaries:
✓ Concise summaries of articles, videos, or other content.
Videos or Images:
✓ Retrieved videos from YouTube or generated images using AI
Interactive Visualizations:
✓ Charts, graphs, or maps for data analysis.
Downloadable Files:
✓ Results can be downloaded in formats like **PDF, MP3**, or **ZIP.**

6. User Interaction
Step: The user interacts with the results.

Details:
- The results are displayed on the user interface in an organized and interactive manner.
- The user can:

Play Videos: Watch retrieved YouTube videos directly on the platform.
Download Files: Save results as PDFs, MP3s, or ZIP files.
Ask Follow-Up Questions: Continue the conversation with the AI chat feature.
Explore Visualizations: Interact with charts and graphs for deeper insights.

Key Features of the Workflow

i. **Efficiency:**
The workflow ensures that user queries are processed quickly and accurately.

ii. Flexibility:
Supports multiple input modalities (text, voice, image) and output formats (text, video, image, PDF, MP3, ZIP).

iii. **Interactivity:**
Provides interactive elements (e.g., video playback, data visualizations) for enhanced user engagement.

iv. **Scalability:**
The modular design allows for easy addition of new features or data sources.

3.3 Data Flow Diagrams (DFDs)

The **Data Flow Diagrams (DFDs)** provide a visual representation of how data moves through the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)**. They help in understanding the flow of information between different components of the system and external entities. The DFDs are organized into three levels: **Level 0 (Context Diagram), Level 1 (High-Level DFD), and Level 2 (Detailed DFD).**

Level 0 (Context Diagram)

- **Purpose:** Shows the interaction between the system and external entities.

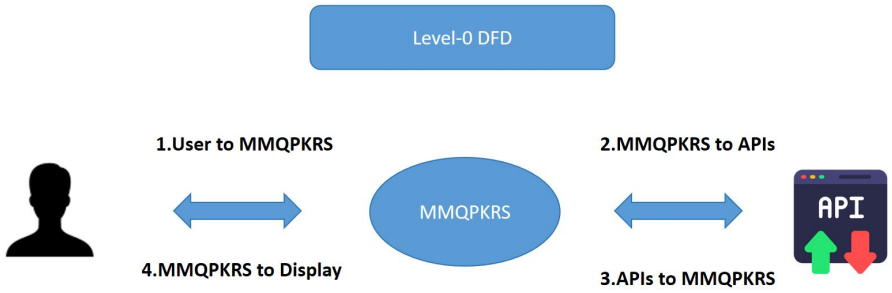


Fig1: Level-0 Data Flow Diagram

- **Components:**

a) **User:**
✓ Interacts with the system by entering queries (text, voice, or image).
✓ Receives results in various formats (text, video, image, PDF, MP3, ZIP).

b) **External APIs:**
✓ Provide data to the system (e.g., YouTube, Google, NASA, Hugging Face).

c) **MMQPKRS:**
✓ Processes user queries and retrieves data from external APIs.
✓ Generates outputs and presents them to the user.

- **Data Flow:**

➢ User → MMQPKRS: Queries (text, voice, image).
➢ MMQPKRS → External APIs: API requests.
➢ External APIs → MMQPKRS: Data (text, video, images, etc.).
➢ MMQPKRS → User: Results (summaries, videos, images, downloadable files).

Level 1 (High-Level DFD)

- **Purpose:** Breaks down the system into major processes.

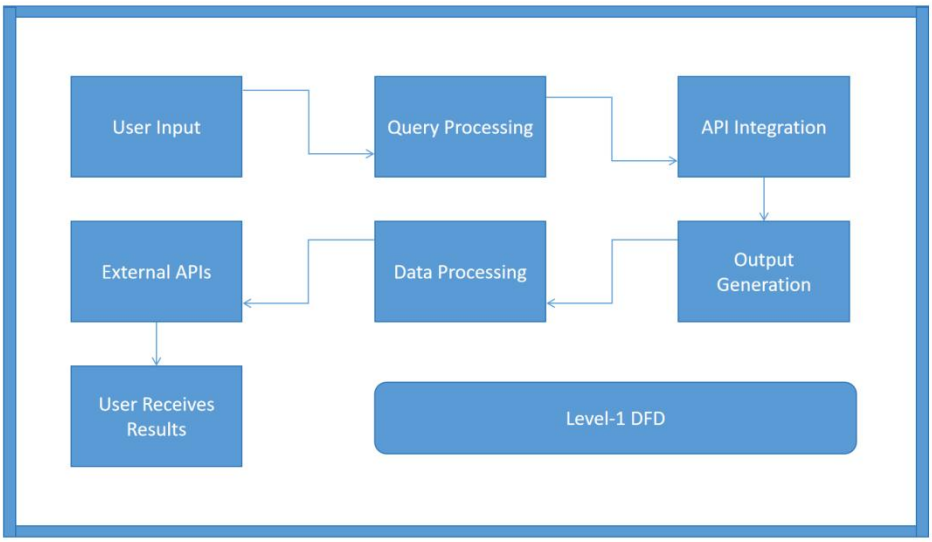


Fig2: Level-1 Data Flow Diagram

- **Processes:**

a) **Query Processing:**
✓ Receives user queries (text, voice, image).
✓ Identifies the type of query and routes it to the appropriate module.
b) **API Integration:**
✓ Makes API calls to external sources (e.g., YouTube, Google, NASA).
✓ Retrieves data (e.g., videos, web pages, space data).
c) **Data Processing:**
✓ Processes retrieved data using AI models (e.g., summarization, image generation).
✓ Converts data into user-friendly formats (e.g., text summaries, visualizations).
d) **Output Generation:**
✓ Generates outputs (e.g., text, videos, images, downloadable files).
✓ Presents results to the user in an interactive and organized manner.

- **Data Flow:**

➢ User → Query Processing: Queries.
➢ Query Processing → API Integration: Query details.
➢ API Integration → External APIs: API requests.
➢ External APIs → API Integration: Retrieved data.
➢ API Integration → Data Processing: Raw data.

- Data Processing → Output Generation: Processed data.
- Output Generation → User: Results.

Level 2 (Detailed DFD)

Purpose: Provides a detailed view of each process, including how text, voice, and image queries are handled, and how data is retrieved, processed, and presented to the user.

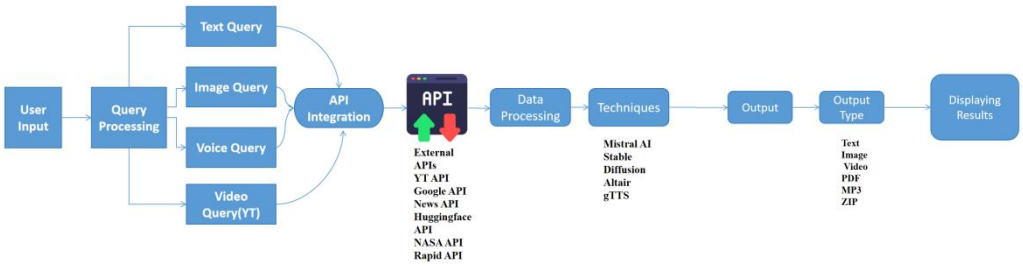


Fig3: Level-2 Data Flow Diagram

- **Query Processing:**

a) **Text Query:**
✓ The user enters a text query (e.g., "What is AI?").
✓ The system identifies the query type and routes it to the appropriate module (e.g., Wikipedia, Google).
b) **Voice Query:**
✓ The user speaks the query, which is converted to text using speech recognition.
✓ The text is then processed like a regular text query.
c) **Image Query:**
✓ The user uploads an image for visual search.
✓ The system uses image processing techniques to analyze the image and retrieve relevant data.

- **API Integration:**

a) **YouTube Data API v3:**
✓ Searches for videos based on the query.
✓ Retrieves video metadata (e.g., title, URL, thumbnail).
b) **Google Custom Search JSON API:**
✓ Fetches web search results.
c) **NewsAPI:**
✓ Retrieves news articles based on the query.
d) **Hugging Face API:**
✓ Provides AI-powered summarization and conversational responses.
e) **NASA APIs:**
✓ Fetches space-related data (e.g., APOD, Mars Rover photos).

f) **RapidAPI:**
✓ Handles YouTube MP3 conversion, image generation, and real-time image search.

- **Data Processing:**

a) **AI-Powered Summarization:**
✓ Uses Mistral to generate concise summaries of retrieved data.
b) **Image Generation:**
✓ Uses Stable Diffusion to create AI-powered images or logos.
c) **Data Visualization:**
✓ Uses Altair to create interactive charts and graphs.
d) **Text-to-Speech Conversion:**
✓ Uses gTTS to convert text into speech for audio outputs.

- **Output Generation:**

a) **Text Summaries:**
✓ Displays concise summaries of articles, videos, or other content.
b) **Videos or Images:**
✓ Displays retrieved videos or generated images.
c) **Interactive Visualizations:**
✓ Displays charts, graphs, or maps for data analysis.
d) **Downloadable Files:**
✓ Allows users to download results as PDFs, MP3s, or ZIP files.

Key Features of the DFDs

i. **Clarity:**
Provides a clear and visual representation of how data flows through the system.
ii. **Modularity:**
Breaks down the system into manageable processes, making it easier to understand and implement.
iii. **Comprehensiveness:**
Covers all major processes, from query processing to output generation.
iv. **Scalability:**
The modular design allows for easy addition of new features or data sources.

3.4 Use Case Diagrams

The **Use Case Diagrams** describe the interactions between users and the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).** They provide a high-level overview of the system's functionality and the key actions that users can perform. Below are the **key use cases** and their descriptions:

1. Search for Information

Description:
✓ Users can search for information using **text, voice, or image-based queries.**

Steps:
- ✓ The user enters a query (text, voice, or image).
- ✓ The system processes the query and retrieves relevant data from external sources (e.g., Wikipedia, Google, YouTube).
- ✓ The system presents the results to the user in an organized and interactive manner.

Actors:
- ✓ User.

Related Modules:
- ✓ Wikipedia Search, Google Search, YouTube Integration, News Search, AI Chat, Image and Logo Generation, Social Media Integration, NASA Space Data Explorer.

2. Generate Summaries

Description:
- ✓ Users can generate **summaries** of articles, videos, or other content.

Steps:
- ✓ The user selects content (e.g., an article or video) for summarization.
- ✓ The system uses **AI-powered summarization** (e.g., Mistral) to generate a concise summary.
- ✓ The summary is displayed to the user.

Actors:
- ✓ User.

Related Modules:
- ✓ Wikipedia Search, News Search, AI Chat.

3. Download Results

Description:
- ✓ Users can download results in various formats (e.g., PDF, MP3, ZIP).

Steps:
- ✓ The user selects the results they want to download.
- ✓ The system generates the results in the desired format (e.g., PDF for text, MP3 for audio, ZIP for images).
- ✓ The user downloads the file to their device.

Actors:
- ✓ User.

Related Modules:
- ✓ Wikipedia Search, YouTube Integration, News Search, Image and Logo Generation.

4. Customize Preferences

Description:

✓ Users can create **custom shortcuts** and manage preferences (e.g., language, search filters).

Steps:

✓ The user accesses the preferences/settings section.
✓ The user creates custom shortcuts for frequently accessed resources.
✓ The user sets preferences (e.g., language, search filters).
✓ The system saves the preferences and applies them to future queries.

Actors:

✓ User.

Related Modules:

✓ Custom Shortcuts, Preferences Management.

5. **Explore Space Data**

Description:

✓ Users can explore **space-related data** using NASA APIs (e.g., Astronomy Picture of the Day, Mars Rover photos).

Steps:

✓ The user selects the space data feature.
✓ The system retrieves data from NASA APIs (e.g., APOD, Mars Rover photos).
✓ The system presents the data to the user in an interactive format (e.g., images, descriptions).

Actors:

✓ User.

Related Modules:

✓ NASA Space Data Explorer.

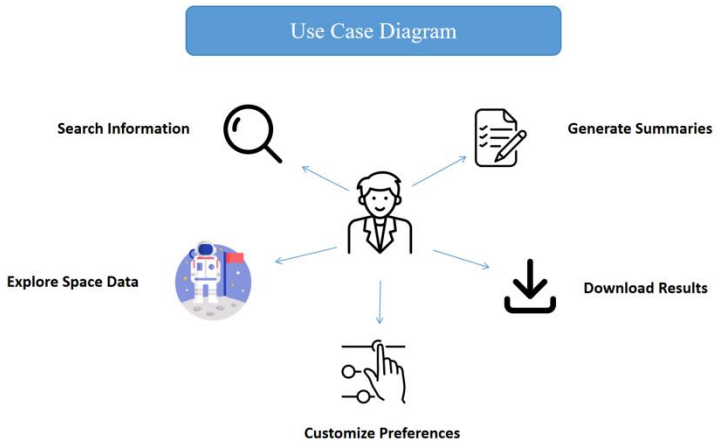


Fig4: Use Case Diagram

Use Case Diagram Components

a) **Actors:**
- ✓ **User:** The primary actor who interacts with the system to perform various actions.

b) **Use Cases:**
- ✓ Search for Information.
- ✓ Generate Summaries.
- ✓ Download Results.
- ✓ Customize Preferences.
- ✓ Explore Space Data.

c) **Relationships:**
- ✓ The User interacts with the system to perform all the use cases.
- ✓ Each use case is associated with specific modules and functionalities within the system.

Key Features of the Use Case Diagrams

- ➢ **User-Centric:**

Focuses on the actions that users can perform, ensuring that the system meets their needs.

- ➢ **Comprehensive:**

Covers all major functionalities of the system, from searching for information to customizing preferences.

- ➢ **Modular:**

Each use case is associated with specific modules, making it easier to understand and implement.

- ➢ **Interactive:**

Highlights the interactive nature of the system, where users can perform multiple actions and receive real-time results.

3.5 Entity-Relationship Diagrams (ERDs)

The **Entity-Relationship Diagrams (ERDs)** represent the data model of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)**. They illustrate the relationships between key entities in the system and their attributes. Below is a detailed breakdown of the **entities**, their **attributes,** and the relationships between them.

Key Entities and Attributes

- ➢ **User:**

Description: Represents the users of the system.

Attributes:
- ✓ **User ID:** Unique identifier for each user.
- ✓ **Name:** Name of the user.
- ✓ **Preferences:** User preferences (e.g., language, search filters).
- ✓ **Shortcuts:** Custom shortcuts created by the user.

Query:
Description: Represents the queries made by users.

Attributes:
- ✓ **Query ID:** Unique identifier for each query.
- ✓ **Type:** Type of query (Text, Voice, Image).
- ✓ **Timestamp:** Date and time when the query was made.

Result:
Description: Represents the results generated by the system in response to user queries.

Attributes:
- ✓ **Result ID:** Unique identifier for each result.
- ✓ **Type:** Type of result (Text, Video, Image).
- ✓ **Source:** Source of the result (Wikipedia, Google, YouTube, etc.).

API:
Description: Represents the external APIs integrated into the system.

Attributes:
- ✓ **API ID:** Unique identifier for each API.
- ✓ **Name:** Name of the API (e.g., YouTube, Google, NASA).
- ✓ **Key:** API key for authentication.

Shortcut:
Description: Represents the custom shortcuts created by users for quick access to frequently used resources.

Attributes:
- ✓ **Shortcut ID:** Unique identifier for each shortcut.
- ✓ **Name:** Name of the shortcut.
- ✓ **Link:** URL or resource associated with the shortcut.
- ✓ **User ID:** Identifier of the user who created the shortcut.

Relationships Between Entities

a) **User and Query:**
- ✓ A User can make multiple Queries.
- ✓ Relationship: One-to-Many (1 User → Many Queries).

b) Query and Result:
- ✓ A Query can generate multiple Results.
- ✓ Relationship: One-to-Many (1 Query → Many Results).

c) **Result and API:**
- ✓ A Result is retrieved from a specific API.
- ✓ Relationship: Many-to-One (Many Results → 1 API).

d) **User and Shortcut:**
- ✓ A User can create multiple Shortcuts.
- ✓ Relationship: One-to-Many (1 User → Many Shortcuts).

ERD Diagram Overview

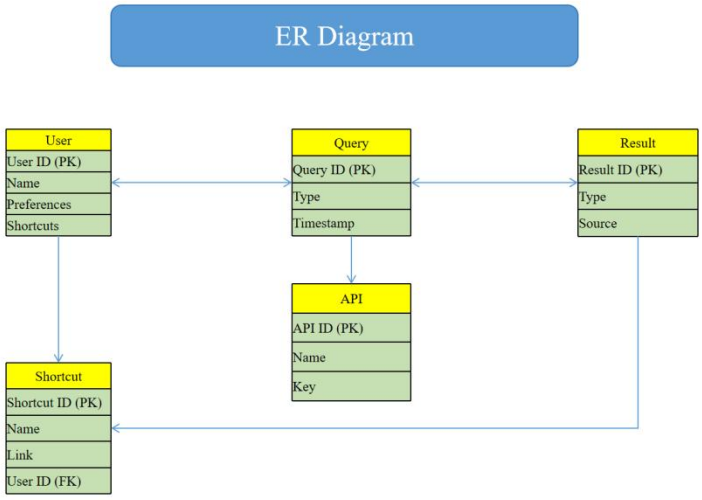


Fig5: Entity-Relationship Diagram

Key Features of the ERDs

a) **Clarity:**
Provides a clear and visual representation of the data model.
b) **Modularity:**
Breaks down the system into manageable entities, making it easier to understand and implement.
c) **Scalability:**
The modular design allows for easy addition of new entities or attributes.
d) **Relationships:**
Clearly defines the relationships between entities, ensuring data integrity and consistency.

3.6 System Requirements

The **system requirements** for the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** are divided into **hardware** and **software** requirements. These requirements ensure that the system runs efficiently and provides a seamless user experience.

3.6.1 Hardware Requirements

The hardware requirements specify the minimum and recommended hardware configurations for running the MMQPKRS.

i. Processor:
Minimum: Intel i5 or equivalent.
Recommended: Intel i7 or equivalent for better performance.

ii. RAM:
Minimum: 8 GB.
Recommended: 16 GB or higher for handling large datasets and multiple API calls.

iii. **Storage:**
Minimum: 10 GB of free disk space.
Recommended: SSD for faster data access and processing.

iv. **Internet Connection:**
Minimum: Stable broadband connection (10 Mbps).
Recommended: High-speed internet (50 Mbps or higher) for faster API calls and data retrieval.

3.6.2 Software Requirements

The software requirements specify the operating systems, programming languages, libraries, APIs, and deployment tools needed for the MMQPKRS.

i. **Operating System:**
Supported: Windows, macOS, or Linux.

ii. **Python Version:**
Minimum: Python 3.8.
Recommended: Python 3.9 or higher for compatibility with the latest libraries.

iii. **Libraries:**
Streamlit: For building the user interface.
SpeechRecognition: For voice search functionality.
gTTS (Google Text-to-Speech): For text-to-speech conversion.
Wikipedia-API: For retrieving Wikipedia summaries.
Requests: For making HTTP requests to APIs.
Pandas: For data manipulation and analysis.
Altair: For data visualization.
ReportLab: For PDF generation.

iv. **APIs:**
YouTube Data API v3: For searching and retrieving YouTube videos.
Google Custom Search JSON API: For web search functionality.
NewsAPI: For fetching news articles.
Hugging Face API: For AI-powered summarization and conversational AI.
NASA APIs: For accessing space-related data.

RapidAPI: For YouTube MP3 conversion, image generation, and real-time image search.

v. **Deployment:**
Streamlit Sharing: For cloud deployment of the MMQPKRS.
Docker: For containerization to ensure consistency across environments.
HuggingFace: For online deployment of the MMQPKRS.

Summary of System Requirements

Hardware Requirements

Component	Minimum	Recommended
Processor	Intel i5 or equivalent	Intel i7 or equivalent
RAM	8 GB	16 GB or higher
Storage	10 GB free disk space	SSD with 10 GB free space
Internet Connection	10 Mbps broadband	50 Mbps or higher

Software Requirements

Category	Details
Operating System	Windows, macOS, or Linux
Python Version	Python 3.8 or higher
Libraries	Streamlit, SpeechRecognition, gTTS, Wikipedia-API, Requests, Pandas, Altair, ReportLab
APIs	YouTube Data API v3, Google Custom Search JSON API, NewsAPI, Hugging Face API, NASA APIs, RapidAPI
Deployment	Streamlit Sharing, Docker, Huggingface

3.7 API Integration Overview

The **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** integrates with several **external APIs** to provide a comprehensive and versatile platform for information retrieval. Below is an overview of the APIs used in the system, their purposes, and how they are integrated:

1. YouTube Data API v3
Purpose: Used for searching and retrieving YouTube videos.
Key Features:
✓ Search for videos based on user queries.
✓ Retrieve video metadata (e.g., title, URL, thumbnail).
✓ Support for video playback and YouTube-to-MP3 conversion.
Integration:
✓ The system makes API calls to the YouTube Data API v3 to fetch video data.
✓ Retrieved data is processed and displayed to the user.

2. Google Custom Search JSON API
Purpose: Enables web search functionality.
Key Features:
- Search for web pages based on user queries.
- Retrieve search results with titles, snippets, and URLs.

Integration:
- The system uses the Google Custom Search JSON API to fetch web search results.
- Results are displayed in an organized and interactive manner.

3. NewsAPI
Purpose: Provides access to news articles.
Key Features:
- Fetch news articles from various sources.
- Filter articles by date, topic, or source.

Integration:
- The system makes API calls to NewsAPI to retrieve news articles.
- Articles are summarized and displayed to the user.

4. Hugging Face API
Purpose: Used for AI-powered summarization and conversational AI.
Key Features:
- ✓ Generate concise summaries of articles, videos, or other content.
- ✓ Provide intelligent responses to user queries.

Integration:
- ✓ The system uses the Hugging Face API to access AI models like Mistral.
- ✓ Summaries and responses are generated and displayed to the user.

5. NASA APIs
Purpose: Provides space-related data (e.g., Astronomy Picture of the Day, Mars Rover photos).
Key Features:
- ✓ Access to Astronomy Picture of the Day (APOD).
- ✓ Retrieve Mars Rover photos and other space-related data.

Integration:
- ✓ The system makes API calls to NASA APIs to fetch space-related data.
- ✓ Data is presented in an interactive format (e.g., images, descriptions).

6. RapidAPI
Purpose: Used for YouTube MP3 conversion, image generation, and real-time image search.
Key Features:
- ✓ Convert YouTube videos to MP3 format.
- ✓ Generate AI-powered images and logos.
- ✓ Perform real-time image searches.

Integration:
- ✓ The system uses RapidAPI to access various services (e.g., YouTube MP3 conversion, image generation).
- ✓ Results are processed and made available to the user.

API	Purpose	Key Features
YouTube Data API v3	Searching and retrieving YouTube videos	Video search, metadata retrieval, YouTube-to-MP3 conversion
Google Custom Search JSON API	Web search functionality	Web search results with titles, snippets, and URLs
NewsAPI	Access to news articles	Fetch news articles, filter by date/topic/source
Hugging Face API	AI-powered summarization and conversational AI	Generate summaries, provide intelligent responses
NASA APIs	Space-related data (e.g., APOD, Mars Rover photos)	Access to APOD, Mars Rover photos, and other space data
RapidAPI	YouTube MP3 conversion, image generation, real-time image search	Convert YouTube videos to MP3, generate AI-powered images, perform image searches

Chapter-4 Implementation

4.1 Development Environment Setup

The **development environment setup** ensures that all necessary tools and libraries are installed and configured for building the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).** Below are the detailed steps to set up the environment:

1. Install Python

Step: Download and install Python 3.8 or higher.
Instructions:
Visit the official Python website: python.org.
Download the latest version of Python (3.8 or higher).
Follow the installation instructions for your operating system.
Verification:
Open a terminal or command prompt and run:

```
python --version
```

Ensure the output shows Python 3.8 or higher.

2. Set Up a Virtual Environment

Step: Create and activate a virtual environment to isolate dependencies.
Instructions:

- Create a virtual environment:

```
python -m venv mmqpkr_env
```

- Activate the virtual environment:

On Windows:

```
mmqpkr_env\Scripts\activate
```

On macOS/Linux:

```
source mmqpkr_env/bin/activate
```

- ✓ The virtual environment name (mmqpkr_env) will appear in the terminal prompt once activated.

3. Install Required Libraries

Step: Install the necessary Python libraries using **pip.**

Instructions:

➢ Run the following command to install the required libraries:

```
pip install streamlit speechrecognition gtts wikipedia-api requests
pandas altair reportlab
```

➢ This installs the following libraries:

✓ Streamlit: For building the user interface.
✓ SpeechRecognition: For voice search functionality.
✓ gTTS (Google Text-to-Speech): For text-to-speech conversion.
✓ Wikipedia-API: For retrieving Wikipedia summaries.
✓ Requests: For making HTTP requests to APIs.
✓ Pandas: For data manipulation and analysis.
✓ Altair: For data visualization.
✓ ReportLab: For PDF generation.

4. Set Up API Keys

Step: Obtain and configure API keys for external services.

Instructions:

➢ Obtain API keys for the following services:

✓ **YouTube Data API v3:** From the Google Cloud Console.
✓ **Google Custom Search JSON API:** From the Google Cloud Console.
✓ **NewsAPI:** From newsapi.org
✓ **Hugging Face API:** From huggingface.co
✓ **NASA APIs:** From api.nasa.gov.
✓ **RapidAPI:** From rapidapi.com.

➢ Store the API keys in a .env file for secure access:

```
YT=your_youtube_api_key
G_API=your_google_custom_search_api_key
G_ID=your_google_custom_search_engine_id
NEWS_API=your_newsapi_key
HF_API=your_huggingface_api_key
NASA=your_nasa_api_key
RAPIDAPI=your_rapidapi_key
```

➢ Use the python-dotenv library to load the API keys into your application:

```
pip install python-dotenv
```

```
from dotenv import load_dotenv
import os

load_dotenv()
YT_API_KEY = os.getenv("YT")
```

5. Install Docker (Optional for Deployment)

Step: Install Docker for containerization and deployment.

Instructions:

- Download Docker from the official website: docker.com.
- Follow the installation instructions for your operating system.
- Verify the installation by running:

```
docker --version
```

- Ensure Docker is running and accessible.

Summary of Development Environment Setup

- ✓ **Install Python:** Download and install Python 3.8 or higher.
- ✓ **Set Up a Virtual Environment:** Create and activate a virtual environment.
- ✓ **Install Required Libraries:** Install Streamlit, SpeechRecognition, gTTS, Wikipedia-API, Requests, Pandas, Altair, and ReportLab.
- ✓ **Set Up API Keys:** Obtain and store API keys in a .env file.
- ✓ **Install Docker (Optional):** Install Docker for containerization and deployment.

4.2 Core Modules and Their Functionality

The **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is divided into several **core modules**, each responsible for specific functionalities. Below is a detailed explanation of the **Wikipedia Search Module**, including its functionality, **key features**, and **implementation.**

4.2.1 Wikipedia Search Module

Functionality

The **Wikipedia Search Module** retrieves summaries and detailed information from **Wikipedia**. It allows users to search for topics and receive concise or detailed summaries in their preferred language.

Key Features

a) **Multilingual Support:**
Supports queries in multiple languages (e.g., English, Spanish, Hindi, Chinese).
b) **Summarization Levels:**
Provides three levels of summarization:

- ✓ Brief: A short summary within a specified character limit.
- ✓ Detailed: The full summary of the Wikipedia page.
- ✓ Bullet Points: Key points from the summary in bullet format.

c) **Error Handling:**
Returns a "Page not found" message if the requested page does not exist.

Implementation

The module is implemented using the wikipedia-api library. Below is the Python code for the **Wikipedia Search Module:**

```
import wikipediaapi

def get_wikipedia_summary(query, lang_code, char_limit, summary_level):
    """
    Retrieves a summary from Wikipedia based on the user's query.
    Args:
        query (str): The search query (e.g., "Artificial Intelligence").
        lang_code (str): Language code (e.g., "en" for English, "es" for Spanish).
        char_limit (int): Maximum number of characters for the summary.
        summary_level (str): Level of summarization ("Brief", "Detailed", "Bullet Points").
    Returns:
        str: The summary of the Wikipedia page.
    """
    # Initialize Wikipedia API with the specified language
    wiki = wikipediaapi.Wikipedia(language=lang_code, user_agent="MMQPKRS
(https://github.com/yourusername/mmqpkr)")
    # Fetch the Wikipedia page
    page = wiki.page(query)
    # Check if the page exists              #gskdsrikrishna
    if not page.exists():
        return "Page not found."
    # Generate the summary based on the selected level
    if summary_level == "Brief":
        return page.summary[:char_limit]  # Return a brief summary
    elif summary_level == "Detailed":
        return page.summary  # Return the full summary
    elif summary_level == "Bullet Points":
        # Split the summary into bullet points
        points = page.summary.split('. ')
        return '\n'.join(f"- {p.strip()}" for p in points if p)[:char_limit]  # Return bullet
points
    else:
        return "Invalid summary level."
```

Usage Example

Here's how the **Wikipedia Search Module** can be used in the MMQPKRS:

```
# Example: Search for "Artificial Intelligence" in English
query = "Artificial Intelligence"
lang_code = "en"  # English
char_limit = 500  # Limit summary to 500 characters
summary_level = "Brief"  # Brief summary

# Get the summary
summary = get_wikipedia_summary(query, lang_code, char_limit,
summary_level)
print(summary)
```

Output Example

For the query **"Artificial Intelligence"** with a **brief summary** and a **500-character limit**, the output might look like this:

```
Artificial intelligence (AI) is intelligence demonstrated by machines, in
contrast to the natural intelligence displayed by humans and animals.
Leading AI textbooks define the field as the study of "intelligent agents":
any device that perceives its environment and takes actions that maximize
its chance of successfully achieving its goals. Colloquially, the term
"artificial intelligence" is often used to describe machines that mimic
cognitive functions that humans associate with the human mind, such as
learning and problem-solving.
```

Key Points

a) **Multilingual Support:**
- The module can retrieve summaries in multiple languages by changing the lang_code parameter (e.g., "es" for Spanish, "hi" for Hindi).

b) **Customizable Summaries:**
- Users can choose between brief, detailed, or bullet-point summaries.

c) **Error Handling:**
- The module checks if the requested page exists and returns an appropriate message if it does not.

4.2.2 Google Search Module

The **Google Search Module** is a core component of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).** It provides web search results using the **Google Custom Search JSON API.** Below is a detailed explanation of its functionality, **key features**, and **implementation.**

Functionality

The **Google Search Module** allows users to perform web searches and retrieve results directly within the **MMQPKRS.** It fetches search results from the **Google Custom Search JSON API** and displays them in an organized format.

Key Features

a) Web Search Results:
✓ Fetches search results with titles, snippets, and URLs.

b) Customizable Results:
✓ Allows users to specify the number of results to retrieve (default is 10).

c) API Integration:
✓ Uses the Google Custom Search JSON API to fetch data.

d) Error Handling:
✓ Returns a JSON response that can be parsed to handle errors or display results.

Implementation

The module is implemented using the **requests** library to interact with the **Google Custom Search JSON API**. Below is the Python code for the **Google Search Module:**

```
import requests

def google_search(api_key, cse_id, query, num_results=10):
    """
    Performs a web search using the Google Custom Search JSON API.
    Args:
        api_key (str): Google API key.
        cse_id (str): Custom Search Engine ID.
        query (str): The search query (e.g., "Artificial Intelligence").
        num_results (int): Number of results to retrieve (default is 10).
    Returns:
        dict: JSON response containing search results.
```

```
    """
    # API endpoint
    url = "https://www.googleapis.com/customsearch/v1"
    # Parameters for the API request
    params = {
        'key': api_key,  # Google API key
        'cx': cse_id,    # Custom Search Engine ID
        'q': query,      # Search query
        'num': num_results  # Number of results to retrieve
    }
    # Make the API request                    #gskdsrikrishna
    response = requests.get(url, params=params)
    # Return the JSON response
    return response.json()
```

Usage Example

Here's how the Google Search Module can be used in the MMQPKRS:

```
# Example: Search for "Artificial Intelligence"
api_key = "your_google_api_key"  # Replace with your Google API key
cse_id = "your_cse_id"           # Replace with your Custom Search Engine
ID
query = "Artificial Intelligence"
num_results = 5  # Retrieve 5 results

# Perform the search
results = google_search(api_key, cse_id, query, num_results)
# Display the results
if "items" in results:
    for item in results["items"]:
        print(f"Title: {item['title']}")
        print(f"Snippet: {item['snippet']}")
        print(f"URL: {item['link']}")
        print("---")
else:
    print("No results found.")
```

Output Example

For the query **"Artificial Intelligence"**, the output might look like this:

```
Title: What is Artificial Intelligence (AI)? | IBM
Snippet: Artificial intelligence (AI) refers to the simulation of human
intelligence in machines that are programmed to think like humans and mimic
their actions.
URL: https://www.ibm.com/cloud/learn/what-is-artificial-intelligence

Title: Artificial Intelligence - Wikipedia
Snippet: Artificial intelligence (AI) is intelligence demonstrated by
machines, in contrast to the natural intelligence displayed by humans and
animals.
URL: https://en.wikipedia.org/wiki/Artificial_intelligence

Title: What is Artificial Intelligence? How Does AI Work? | Built InSnippet:
Artificial intelligence (AI) is a wide-ranging branch of computer science
concerned with building smart machines capable of performing tasks that
typically require human intelligence.
URL: https://builtin.com/artificial-intelligence
```

Key Points

a) **API Key and CSE ID:**
✓ The module requires a Google API key and a Custom Search Engine ID (CSE ID) to function. These can be obtained from the Google Cloud Console.
b) **Customizable Results:**
✓ Users can specify the number of results to retrieve by adjusting the num_results parameter.
c) **Error Handling:**
✓ The module checks if the response contains the "items" key. If not, it indicates that no results were found.

4.2.3 YouTube Integration Module

The **YouTube Integration Module** is a core component of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)**. It enables users to search and retrieve YouTube videos directly within the system. Below is a detailed explanation of its **functionality, key features,** and **implementation.**

Functionality

The **YouTube Integration Module** allows users to search for YouTube videos based on a query. It retrieves video **metadata**, including the **title, URL, and thumbnail**, and supports **video playback** and **YouTube-to-MP3** conversion.

Key Features

a) Video Search:
- Searches YouTube for videos based on user queries.

b) Metadata Retrieval:
- Fetches video metadata, including:

✓ Title: The title of the video.
✓ URL: The link to the video.
✓ Thumbnail: The thumbnail image of the video.
✓ Video ID: The unique identifier for the video.

c) Video Playback:
- Supports embedding and playback of YouTube videos within the system.

d) YouTube-to-MP3 Conversion:
- Provides functionality to convert YouTube videos to MP3 format (requires additional integration with services like RapidAPI).

Implementation

The module is implemented using the YouTube Data API v3 and the **googleapiclient** library. Below is the Python code for the YouTube Integration Module:

```
from googleapiclient.discovery import build

def search_youtube(query, max_results=5):
    """
    Searches YouTube for videos based on a query and retrieves video metadata.
    Args:
        query (str): The search query (e.g., "Python tutorial").
        max_results (int): Maximum number of results to retrieve (default is 5).
    Returns:
        list: A list of dictionaries containing video metadata (title, URL, video
ID, thumbnail).
    """
    # Initialize the YouTube API client
    youtube = build('youtube', 'v3', developerKey=API_KEY_YOUTUBE)
    # Make the API request
    request = youtube.search().list(
        q=query,               # Search query
        part='id,snippet',      # Retrieve video ID and snippet (metadata)
        maxResults=max_results, # Number of results to retrieve
        type='video'            # Search only for videos
    )
    response = request.execute()
    # Parse the response and extract video metadata       #gskdsrikrishna
    videos = []
    for item in response['items']:
        video_id = item['id']['videoId']  # Extract video ID
        title = item['snippet']['title']  # Extract video title
        thumbnail = item['snippet']['thumbnails']['default']['url']  # Extract
thumbnail URL
        url = f'https://www.youtube.com/watch?v={video_id}'  # Generate video URL
        videos.append({
            'title': title,
            'url': url,
            'video_id': video_id,
            'thumbnail': thumbnail
        })
    return videos
```

Usage Example

Here's how the **YouTube Integration Module** can be used in the MMQPKRS:

```
# Example: Search for "Python tutorial" videos
query = "Python tutorial"
max_results = 3  # Retrieve 3 results

# Perform the search
videos = search_youtube(query, max_results)
# Display the results
for video in videos:
    print(f"Title: {video['title']}")
    print(f"URL: {video['url']}")
    print(f"Thumbnail: {video['thumbnail']}")
    print("---")
```

Output Example

For the query **"Python tutorial"**, the output might look like this:

```
Title: Python Tutorial for Beginners - Full Course in 11 Hours
URL: https://www.youtube.com/watch?v=abc123
Thumbnail: https://i.ytimg.com/vi/abc123/default.jpg
---
Title: Learn Python - Full Course for Beginners [Tutorial]
URL: https://www.youtube.com/watch?v=def456
Thumbnail: https://i.ytimg.com/vi/def456/default.jpg
---
Title: Python Programming Tutorial for Beginners
URL: https://www.youtube.com/watch?v=ghi789
Thumbnail: https://i.ytimg.com/vi/ghi789/default.jpg
```

Key Points

i. **API Key:**
- The module requires a YouTube Data API v3 key to function. This can be obtained from the Google Cloud Console.

ii. **Customizable Results:**
- Users can specify the number of results to retrieve by adjusting the max_results parameter.

iii. **Metadata:**
- The module retrieves essential video metadata, including the title, URL, video ID, and thumbnail.

iv. **Video Playback:**
- The video URL can be used to embed and play the video within the system.

v. **YouTube-to-MP3 Conversion:**
- Additional integration with services like **RapidAPI** can enable YouTube-to-MP3 conversion.

4.2.4 News Search Module

The **News Search Module** is a core component of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).** It enables users to **fetch news articles** from various sources based on their queries. Below is a detailed explanation of its **functionality, key features,** and **implementation.**

Functionality

The **News Search Module** allows users to search for **news articles** using the **NewsAPI.** It retrieves articles based on a query and supports filtering by **date range, topic, or source.**

Key Features

a) News Article Retrieval:
- Fetches news articles from a wide range of sources.

b) Filtering:
- Allows users to filter articles by:

✓ **Date Range:** Specify a start date (from_date) and end date (to_date).
✓ **Topic:** Search for articles related to specific topics (e.g., "technology", "politics").
✓ **Source:** Retrieve articles from specific news sources.

c) API Integration:

- Uses the **NewsAPI** to fetch news data.

d) Error Handling:

- Returns a JSON response that can be parsed to handle errors or display results.

Implementation

The module is implemented using the requests library to interact with the NewsAPI. Below is the Python code for the **News Search Module**:

```
import os
import requests

def search_news(query, from_date=None, to_date=None):
    """
    Fetches news articles from various sources based on a query.
    Args:
        query (str): The search query (e.g., "Artificial Intelligence").
        from_date (str): Start date for filtering articles (format: YYYY-MM-DD).
        to_date (str): End date for filtering articles (format: YYYY-MM-DD).
    Returns:
        dict: JSON response containing news articles.
    """
    # Retrieve the NewsAPI key from environment variables   gskdsrikrishna
    api_key = os.getenv("NEWS_API")
    # Base URL for the NewsAPI
    url = f"https://newsapi.org/v2/everything?q={query}&apiKey={api_key}"
    # Add date range filters if provided
    if from_date and to_date:
        url += f"&from={from_date}&to={to_date}"
    # Make the API request
    response = requests.get(url)
    # Return the JSON response
    return response.json()
```

Usage Example

Here's how the **News Search Module** can be used in the MMQPKRS:

```
# Example: Search for news articles about "Artificial Intelligence"
query = "Artificial Intelligence"
from_date = "2023-10-01"  # Start date
to_date = "2023-10-31"    # End date

# Perform the search
news_results = search_news(query, from_date, to_date)
# Display the results
if "articles" in news_results:
    for article in news_results["articles"]:
        print(f"Title: {article['title']}")
        print(f"Description: {article['description']}")
        print(f"URL: {article['url']}")
        print("---")
else:
    print("No results found.")
```

Output Example

For the query **"Artificial Intelligence"** with a date range of February 27, 2025, to February 22, 2025, the output might look like this:

```
Title: Artificial Intelligence: The Future of Technology
Description: Artificial intelligence (AI) is transforming industries and
reshaping the future of technology. Learn how AI is being used in
healthcare, finance, and more.
URL: https://example.com/ai-future-technology

Title: How AI is Revolutionizing the Workplace
Description: From automation to predictive analytics, AI is changing the
way we work. Discover the latest trends and innovations in workplace AI.
URL: https://example.com/ai-workplace-revolution

Title: The Ethical Challenges of Artificial Intelligence
Description: As AI becomes more advanced, ethical concerns are growing.
Explore the challenges and potential solutions for responsible AI
development.
URL: https://example.com/ai-ethical-challenges
```

Key Points

i. **API Key:**
- The module requires a **NewsAPI key** to function. This can be obtained from newsapi.org.

ii. **Filtering:**
- Users can filter articles by date range, topic, or source by modifying the query parameters.

iii. **Error Handling:**
- The module checks if the response contains the "articles" key. If not, it indicates that no results were found.

iv. **Date Format:**
- Dates must be provided in the format YYYY-MM-DD.

4.2.5 AI Chat Module

The **AI Chat Module** is a core component of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).** It provides **conversational AI capabilities** by leveraging the **Hugging Face API** and AI models like **Mistral.** Below is a detailed explanation of its **functionality, key features,** and **implementation**.

Functionality

The **AI Chat Module** enables users to interact with the system using **natural language queries**. It generates **intelligent responses** and **summaries** based on the user's input, providing a conversational experience.

Key Features

a) **Conversational AI:**
✓ Engages in natural language conversations with users.
b) **Intelligent Responses:**
✓ Generates context-aware and relevant responses to user queries.

c) **Summarization:**
✓ Provides concise summaries of articles, videos, or other content.
d) **Customizable Parameters:**
✓ Allows customization of response length and creativity using parameters like max_length and temperature.

Implementation

The module is implemented using the requests library to interact with the Hugging Face API. Below is the Python code for the **AI Chat Module:**

```
import requests

def chat_with_mistral_hf(prompt):
    """
    Generates a response to a user's prompt using the Hugging Face API and Mistral model.
    Args:
        prompt (str): The user's input or query.
    Returns:
        str: The generated response or an error message.
    """
    # API endpoint and headers
    headers = {"Authorization": f"Bearer {HF_API_KEY}"}
    payload = {
        "inputs": prompt,  # User's input
        "parameters": {
            "max_length": 200,  # Maximum length of the response      gskdsrikrishna
            "temperature": 0.7  # Controls creativity (0 = deterministic, 1 = creative)
        }
    }
    # Make the API request
    response = requests.post(HF_MISTRAL_URL, json=payload, headers=headers)
    # Check if the request was successful
    if response.status_code == 200:
        return response.json()[0]["generated_text"]  # Return the generated response
    else:
        return f"Error: {response.json()}"  # Return an error message
```

Usage Example

Here's how the **AI Chat Module** can be used in the MMQPKRS:

```
# Example: Ask the AI a question
prompt = "What is artificial intelligence?"

# Generate a response
response = chat_with_mistral_hf(prompt)
# Display the response
print(response)
```

Output Example

For the prompt **"What is artificial intelligence?"**, the output might look like this:

```
Artificial intelligence (AI) refers to the simulation of human intelligence
in machines that are programmed to think and learn like humans. These
machines can perform tasks such as problem-solving, decision-making, and
language understanding, often surpassing human capabilities in specific
domains.
```

Key Points

i. **API Key:**
- The module requires a Hugging Face API key to function. This can be obtained from huggingface.co.

ii. **Customizable Parameters:**
- The max_length parameter controls the length of the response.
- The temperature parameter controls the creativity of the response (0 = deterministic, 1 = creative).

iii. **Error Handling:**
- The module checks the response status code and returns an error message if the request fails.

4.2.6 Image and Logo Generation Module

The **Image and Logo Generation Module** is a powerful component of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** that enables AI-powered creation of visual content. Below is a detailed explanation of its **functionality, key features**, and **implementation.**

Functionality

- This module generates custom images and logos based on user-provided text prompts by leveraging advanced AI image generation APIs. It provides:

✓ AI-generated images from textual descriptions
✓ Custom logo creation capabilities
✓ Downloadable output formats

Key Features

a. **Prompt-Based Generation:**
- Creates unique images/logos from natural language descriptions

b. **Style Customization:**
- Supports different artistic styles (realistic, cartoon, abstract etc.)

c. **Output Formatting:**
- Generates images in standard formats (JPEG, PNG, WEBP)

d. **Batch Processing:**
- Can generate multiple variations from a single prompt

e. **API Integration:**
- Connects with leading AI image generation services

Implementation

The module is implemented using Python's requests library to interact with the **AI Image Generator API**. Here's the core implementation:

```python
import requests

def generate_image(prompt, style_id=28, size="1-1"):
    """
    Generates AI-powered images based on text prompts.

    Args:
        prompt (str): Description of desired image/logo
        style_id (int): Artistic style selection (default: 28)
        size (str): Output dimensions (e.g., "1-1" for square)

    Returns:
        dict: API response containing generated image data
    """
    url = "https://ai-image-generator14.p.rapidapi.com/"

    headers = {
        "x-rapidapi-key": AI_IMAGE_API_KEY,
        "x-rapidapi-host": "ai-image-generator14.p.rapidapi.com",
        "Content-Type": "application/json"
    }

    payload = {
        "jsonBody": {
            "function_name": "image_generator",
            "type": "image_generation",
            "query": prompt,
            "style_id": style_id,
            "size": size
        }
    }

    try:
        response = requests.post(url, json=payload, headers=headers)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        return {"error": str(e)}
```

Usage Example

Here's how to use the module in the MMQPKRS:

```python
# Generate a futuristic tech logo
prompt = "A minimalist logo for a quantum computing company, blue and purple color scheme"
result = generate_image(prompt, style_id=29)

if "error" not in result:
    image_url = result["output_url"]
    print(f"Generated image: {image_url}")
else:
    print(f"Error: {result['error']}")
```

Output Example

For the prompt **"A cyberpunk cityscape at night with neon lights"**, the module might return:

```
{
    "status": "success",
    "output_url": "https://generated-images.com/cyberpunk123.jpg",
    "metadata": {
        "style": "cyberpunk",
        "dimensions": "1024x1024",
        "generation_time": "4.2s"
    }
}
```

Key Components

i. **API Configuration:**
- Requires AI_IMAGE_API_KEY from RapidAPI
- Supports multiple style presets (28-30 for different artistic styles)

ii. **Parameters:**
- prompt: Detailed description of desired image
- style_id: Controls artistic style (28=standard, 29=illustration, 30=photorealistic)
- size: Output dimensions ("1-1"=square, "2-3"=portrait, "3-4"=landscape)

iii. **Error Handling:**
- Catches and returns API connection errors
- Validates input parameters

4.2.7 Social Media Integration Module

The **Social Media Integration Module** is a core component of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)**. It enables users to **extract and analyze** data from **LinkedIn** and **Instagram.** Below is a detailed explanation of its **functionality, key features**, and **implementation.**

Functionality

The **Social Media Integration Module** allows users to retrieve **profile data, images**, and **links** from **LinkedIn** and **Instagram.** This data can be used for analysis, visualization, or integration with other modules in the MMQPKRS.

Key Features

a) **Profile Data Retrieval:**
- Fetches user profile information, such as name, bio, and connections.

b) **Image Extraction:**
- Retrieves profile pictures and other images associated with the account.

c) **Link Extraction:**
- Extracts links to posts, profiles, and other resources.

d) **API Integration:**
- Uses RapidAPI to interact with LinkedIn and Instagram APIs.

Implementation

The module is implemented using the requests library to interact with the **LinkedIn API** via **RapidAPI.** Below is the Python code for the **Social Media Integration Module:**

```
import requests

def extract_linkedin_data(username):
    """
    Extracts profile data from LinkedIn using the LinkedIn API via RapidAPI.
    Args:
        username (str): The LinkedIn username or profile ID.
    Returns:
```

```
        dict: JSON response containing profile data.
    """
    # API endpoint and headers
    url = "https://linkedin-api8.p.rapidapi.com/"
    headers = {
        "x-rapidapi-key": LINKEDIN_API_KEY,  # RapidAPI key
        "x-rapidapi-host": "linkedin-api8.p.rapidapi.com"
    }
    # Make the API request
    response = requests.get(url, headers=headers, params={"username": username})
    # Return the JSON response
    return response.json()
```

Usage Example

Here's how the Social Media Integration Module can be used in the MMQPKRS:

```
# Example: Extract LinkedIn profile data
username = "john-doe"  # Replace with a valid LinkedIn username or profile ID

# Fetch profile data
profile_data = extract_linkedin_data(username)
# Display the profile data
if "profile" in profile_data:
    print(f"Name: {profile_data['profile']['name']}")
    print(f"Bio: {profile_data['profile']['bio']}")
    print(f"Profile Picture: {profile_data['profile']['image_url']}")
    print(f"Connections: {profile_data['profile']['connections']}")
else:
    print("Profile not found or API error.")
```

Output Example

For the username **"gskdsrikrishna"**, the output might look like this:

```
Name: Gskd
Bio:AI Engineer | Youtuber | B-Tech Student
Profile Picture: https://example.com/gskd-srikrishna-profile.jpg
Connections: 500+
```

Key Points

i. **API Key:**
- The module requires a **RapidAPI** key to access the **LinkedIn API**. This can be obtained from rapidapi.com.

ii. **Error Handling:**
- The module checks if the response contains the "profile" key. If not, it indicates that the profile was not found or there was an API error.

iii. **Instagram Integration:**
- Similar functionality can be implemented for Instagram using the Instagram API via RapidAPI.

4.2.8 NASA Space Data Explorer Module

The **NASA Space Data Explorer Module** is a core component of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).** It provides users

with access to **space-related data**, including the **Astronomy Picture of the Day (APOD), Mars Rover photos**, and other fascinating space data. Below is a detailed explanation of its **functionality, key features,** and **implementation.**

Functionality

The **NASA Space Data Explorer Module** allows users to explore **space-related data** by interacting with **NASA's APIs. I**t fetches data such as:

- **Astronomy Picture of the Day (APOD):** A daily image or video with an explanation.
- **Mars Rover Photos:** Images captured by NASA's Mars rovers.
- **Near-Earth Object (NEO) Data:** Information about asteroids and comets close to Earth.

Key Features

a) Astronomy Picture of the Day (APOD):
- Fetches the daily image or video along with its explanation.

b) Mars Rover Photos:
- Retrieves images captured by Mars rovers like Curiosity, Opportunity, and Perseverance.

c) Near-Earth Object (NEO) Data:
- Provides information about asteroids and comets near Earth.

d) API Integration:
- Uses NASA APIs to fetch space-related data.

Implementation

The module is implemented using the requests library to interact with NASA's APIs. Below is the Python code for the **NASA Space Data Explorer Module:**

```
import requests

def fetch_nasa_apod():
    """
    Fetches the Astronomy Picture of the Day (APOD) from NASA's API.
    Returns:
        dict: JSON response containing the APOD data (image/video URL, explanation, etc.).
    """
    # API endpoint and parameters
    url = "https://api.nasa.gov/planetary/apod"
    params = {"api_key": NASA_API_KEY}  # NASA API key            gskdsrikrishna
    # Make the API request
    response = requests.get(url, params=params)
    # Return the JSON response
    return response.json()
```

Usage Example

Here's how the **NASA Space Data Explorer Module** can be used in the MMQPKRS:

```
# Example: Fetch the Astronomy Picture of the Day (APOD)
apod_data = fetch_nasa_apod()

# Display the APOD data
if "url" in apod_data:
```

```
    print(f"Title: {apod_data['title']}")
    print(f"Explanation: {apod_data['explanation']}")
    print(f"Image/Video URL: {apod_data['url']}")
else:
    print("Error fetching APOD data.")
```

Output Example

For the Astronomy Picture of the Day (APOD), the output might look like this:

```
Title: The Andromeda Galaxy in Ultraviolet
Explanation: The Andromeda Galaxy, also known as M31, is the closest spiral
galaxy to our Milky Way. This image, captured in ultraviolet light, reveals
the galaxy's star-forming regions and massive young stars.
Image/Video URL: https://apod.nasa.gov/apod/image/2310/andromeda_uv.jpg
```

Key Points

i. API Key:
- The module requires a NASA API key to function. This can be obtained from api.nasa.gov.

ii. Error Handling:
- The module checks if the response contains the "url" key. If not, it indicates an error in fetching the data.

iii. Additional NASA APIs:
- The module can be extended to fetch other space-related data, such as:

✓ Mars Rover Photos: Using the Mars Rover Photos API.
✓ Near-Earth Object (NEO) Data: Using the NEO API.

4.3 API Integration Details

The **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** integrates with multiple **external APIs** to retrieve and process data. These APIs enable the system to provide a wide range of functionalities, from **video search** and **web search** to **AI-powered summarization** and **space data exploration**. Below is a detailed overview of the **API integration** in the MMQPKRS:

1. YouTube Data API v3

Purpose: Used for video search and metadata retrieval.

Key Features:
✓ Search for YouTube videos based on user queries.
✓ Retrieve video metadata, including **title, URL, thumbnail**, and **description.**
✓ Support for **video playback** and **YouTube-to-MP3** conversion.

Integration:
✓ The system makes API calls to the **YouTube Data API v3** using the googleapiclient library.

Example:

```
from googleapiclient.discovery import build

youtube = build('youtube', 'v3', developerKey=API_KEY_YOUTUBE)
request = youtube.search().list(q="Python tutorial", part='id,snippet', maxResults=5, type='video')
response = request.execute()
```

2. Google Custom Search JSON API

Purpose: Enables web search functionality.

Key Features:
- ✓ Search for web pages based on user queries.
- ✓ Retrieve search results with titles, snippets, and URLs.

Integration:
- ✓ The system uses the requests library to interact with the Google Custom Search JSON API.

Example:

```
import requests

url = "https://www.googleapis.com/customsearch/v1"
params = {'key': API_KEY, 'cx': CSE_ID, 'q': "Artificial Intelligence", 'num': 10}
response = requests.get(url, params=params)
```

3. NewsAPI

Purpose: Provides access to news articles.

Key Features:
- ✓ Fetch news articles from various sources.
- ✓ Filter articles by date, topic, or source.

Integration:
- ✓ The system uses the requests library to interact with the NewsAPI.

Example:

```
import requests

url = "https://newsapi.org/v2/everything"
params = {'q': "Artificial Intelligence", 'apiKey': NEWS_API_KEY}
response = requests.get(url, params=params)
```

4. Hugging Face API

Purpose: Used for AI-powered summarization and conversational AI.

Key Features:
- ✓ Generate concise summaries of articles, videos, or other content.
- ✓ Provide intelligent responses to user queries.

Integration:
✓ The system uses the requests library to interact with the Hugging Face API.

Example:

```
import requests

headers = {"Authorization": f"Bearer {HF_API_KEY}"}
payload = {"inputs": "What is AI?", "parameters": {"max_length": 200, "temperature": 0.7}}
response = requests.post(HF_MISTRAL_URL, json=payload, headers=headers)
```

5. NASA APIs

Purpose: Provides access to space-related data.

Key Features:
✓ Fetch Astronomy Picture of the Day (APOD).
✓ Retrieve Mars Rover photos and other space data.

Integration:
✓ The system uses the requests library to interact with NASA APIs.

Example:

```
import requests

url = "https://api.nasa.gov/planetary/apod"
params = {'api_key': NASA_API_KEY}
response = requests.get(url, params=params)
```

6. RapidAPI

Purpose: Used for YouTube MP3 conversion, image generation, and real-time image search.

Key Features:
✓ Convert YouTube videos to MP3 format.
✓ Generate AI-powered images and logos.
✓ Perform real-time image searches.

Integration:
✓ The system uses the requests library to interact with RapidAPI.

Example:

```
import requests

url = "https://youtube-mp310.p.rapidapi.com/download/mp3"
headers = {"x-rapidapi-key": RAPIDAPI_KEY, "x-rapidapi-host": "youtube-mp310.p.rapidapi.com"}
params = {"url": "https://www.youtube.com/watch?v=abc123"}
response = requests.get(url, headers=headers, params=params)
```

Summary of API Integration

API	Purpose	Key Features
YouTube Data API v3	Video search and metadata retrieval	Search for videos, retrieve metadata, support for video playback
Google Custom Search JSON API	Web search functionality	Fetch web search results with titles, snippets, and URLs
NewsAPI	Access to news articles	Fetch news articles, filter by date, topic, or source
Hugging Face API	AI-powered summarization and conversational AI	Generate summaries, provide intelligent responses
NASA APIs	Space-related data (e.g., APOD, Mars Rover photos)	Fetch APOD, Mars Rover photos, and other space data
RapidAPI	YouTube MP3 conversion, image generation, real-time image search	Convert YouTube videos to MP3, generate AI-powered images, perform image searches

4.4 Shortcut Management System

The **Shortcut Management System** is a feature of the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** that allows users to **create and manage custom shortcuts** for frequently accessed resources. This system enhances user productivity by providing quick access to important links, files, or tools.

Functionality

The **Shortcut Management System** enables users to:

a) Create Shortcuts:
- Add custom shortcuts for frequently used resources (e.g., websites, files, tools).

b) Manage Shortcuts:
- Edit or delete existing shortcuts.

c) Load and Save Shortcuts:
- Persist shortcuts across sessions by saving them to a file.

Key Features

a) Custom Shortcuts:
✓ Users can create shortcuts with a **name, link**, and optional **icon.**

b) Persistence:
✓ Shortcuts are saved to a file using **pickle**, ensuring they are available across sessions.

c) User-Friendly Interface:
✓ Shortcuts are displayed in an organized manner for easy access.

Implementation

The **Shortcut Management System** is implemented using Python's **pickle** module to serialize and deserialize shortcut data. Below is the Python code for the system:

```
import os
import pickle

# File to store shortcut data
DATA_FILE = "shortcuts_data.pkl"
def load_shortcuts():
    """
    Loads shortcuts from the data file.
    Returns:
        list: A list of shortcuts (each shortcut is a dictionary with 'name',
'link', and 'icon' keys).
    """
    if os.path.exists(DATA_FILE):
        with open(DATA_FILE, "rb") as f:
            return pickle.load(f)
    return []  # Return an empty list if the file does not exist   gskdsrikrishna
def save_shortcuts(shortcuts):
    """
    Saves shortcuts to the data file.
    Args:
        shortcuts (list): A list of shortcuts to save.
    """
    with open(DATA_FILE, "wb") as f:
        pickle.dump(shortcuts, f)
```

Usage Example

Here's how the **Shortcut Management System** can be used in the **MMQPKRS:**

```
# Example: Create and manage shortcuts

# Load existing shortcuts
shortcuts = load_shortcuts()
# Add a new shortcut
new_shortcut = {
    "name": "Google",
    "link": "https://www.google.com",
    "icon": "google_icon.png"  # Optional icon file
}
shortcuts.append(new_shortcut)
# Save the updated shortcuts
save_shortcuts(shortcuts)
# Display all shortcuts
for shortcut in shortcuts:
    print(f"Name: {shortcut['name']}, Link: {shortcut['link']}, Icon: {shortcut.get('icon', 'No icon')}")
```

Output Example

For the example above, the output might look like this:

```
Name: Google, Link: https://www.google.com, Icon: google_icon.png
Name: Wikipedia, Link: https://www.wikipedia.org, Icon: No icon
```

Key Points

i. **Data Persistence:**
✓ Shortcuts are saved to a file (shortcuts_data.pkl) using pickle, ensuring they persist across sessions.
ii. **Flexibility:**
✓ Users can add, edit, or delete shortcuts as needed.
iii. **Error Handling:**
✓ The system checks if the data file exists before attempting to load shortcuts.

4.5 Error Handling and Debugging

The **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** includes **robust error handling and debugging mechanisms** to ensure smooth operation and provide a seamless user experience. These mechanisms help identify and resolve issues quickly, minimizing disruptions for users.

Error Handling and Debugging Strategies

i. **Try-Except Blocks:**
✓ Use try-except blocks to catch and handle exceptions gracefully.
ii. **Logging:**
✓ Log errors and debugging information to a file for later analysis.
iii. **User-Friendly Error Messages:**
✓ Display clear and concise error messages to users.
iv. **Input Validation:**
✓ Validate user inputs to prevent errors caused by invalid data.

Implementation

Below is an example of how **error handling and debugging** are implemented in the **MMQPKRS:**

```
import requests
import logging
import streamlit as st

# Configure logging
logging.basicConfig(filename="error_log.txt", level=logging.ERROR, format="%(asctime)s -
%(levelname)s - %(message)s")
def fetch_data(url, headers=None, params=None):
    """
    Fetches data from an API endpoint with error handling.
    Args:
        url (str): The API endpoint URL.
        headers (dict): Optional headers for the request.
        params (dict): Optional parameters for the request.
    Returns:
        dict: JSON response from the API or None if an error occurs.
    """
    try:
        # Make the API request
        response = requests.get(url, headers=headers, params=params)
        response.raise_for_status()  # Raise an exception for HTTP errors  gskdsrikrishna
        return response.json()  # Return the JSON response
    except requests.exceptions.RequestException as e:
        # Log the error
```

```
        logging.error(f"Error fetching data from {url}: {e}")
        # Display a user-friendly error message
        st.error(f"An error occurred while fetching data. Please try again later. Error: {e}")
        return None
```

Usage Example

Here's how the **error handling and debugging mechanisms** can be used in the **MMQPKRS:**

```
# Example: Fetch data from an API with error handling
url = "https://api.example.com/data"
headers = {"Authorization": "Bearer YOUR_API_KEY"}
params = {"query": "example"}

# Fetch data
data = fetch_data(url, headers=headers, params=params)
# Process the data if no error occurred
if data:
    st.write("Data fetched successfully:")
    st.json(data)
else:
    st.write("Failed to fetch data. Please check the logs for details.")
```

Error Log Example

If an error occurs, it will be logged in the error_log.txt file with a timestamp and details:

```
2023-10-15 12:34:56,789 - ERROR - Error fetching data from
https://api.example.com/data: 404 Client Error: Not Found for url:
https://api.example.com/data
```

Key Points

a) Try-Except Blocks:
✓ Catch exceptions and handle them gracefully to prevent the system from crashing.
b) Logging:
✓ Log errors to a file for debugging and analysis.
c) User-Friendly Messages:
✓ Display clear and concise error messages to users.
d) Input Validation:
✓ Validate user inputs to prevent errors caused by invalid data.

Chapter-5 Testing and Validation

5.1 Testing Methodology

The **testing methodology** for the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is designed to ensure that the system is **reliable, efficient, secure, and user-friendly**. It follows a structured approach, covering all aspects of the system from individual modules to the entire system. Below is a detailed explanation of the testing methodology:

1. Unit Testing
Purpose: Test individual modules in isolation to ensure they function as expected.

Scope:
- ✓ Test each module (e.g., Wikipedia Search, Google Search, YouTube Integration) independently.
- ✓ Verify that the module produces the correct output for a given input.

Tools:
- ✓ Python's unittest or pytest framework.

Example:
- ✓ Test the Wikipedia Search Module by providing a query and verifying the summary output.
- ✓ Test the Google Search Module by searching for a term and checking the returned results.

2. Integration Testing
Purpose: Test the interaction between modules to ensure data flows correctly.

Scope:
- ✓ Verify that modules work together seamlessly (e.g., Wikipedia Search and AI Chat).
- ✓ Ensure data is passed correctly between components.

Example:
- ✓ Test the integration between the Wikipedia Search Module and the AI Chat Module by generating a summary and verifying that the AI Chat can process it.
- ✓ Test the integration between the YouTube Integration Module and the News Search Module to ensure that video and news data are displayed together correctly.

3. System Testing
Purpose: Test the system as a whole to ensure all components work together seamlessly.

Scope:
- ✓ Verify end-to-end functionality of the system.
- ✓ Ensure that user queries are processed correctly and results are displayed as expected.

Example:
- ✓ Perform a complete search query (e.g., "Artificial Intelligence") and verify that the system retrieves and displays results from all relevant modules (Wikipedia, Google, YouTube, News, etc.).

4. Performance Testing
Purpose: Evaluate the system's performance under various conditions.

Scope:
- ✓ Test the system's responsiveness, scalability, and stability.
- ✓ Simulate high user loads and large data volumes.

Types of Performance Testing:
Load Testing:
- ✓ Simulate high user loads to test the system's performance.
- ✓ Example: Test the system with 100 concurrent users.

Stress Testing:
- ✓ Push the system to its limits to identify breaking points.
- ✓ **Example:** Test the system with 1000 concurrent users.

Response Time Testing:
- ✓ Measure the time taken to process user queries.
- ✓ **Example:** Ensure the system responds within 2 seconds for 95% of queries.

Tools:
- ✓ Tools like Apache JMeter or Locust for load and stress testing.

5. Security Testing
Purpose: Identify vulnerabilities and ensure the system is secure.

Scope:
- ✓ Test authentication, authorization, data encryption, and input validation.
- ✓ Ensure that sensitive data (e.g., API keys) is protected.

Types of Security Testing:
Authentication and Authorization:
- ✓ Verify that only authorized users can access certain features.

Data Encryption:
- ✓ Ensure that sensitive data is encrypted (e.g., API keys stored in environment variables).

Input Validation:
- ✓ Test the system's ability to handle invalid or malicious inputs (e.g., SQL injection attacks).

Tools:
- ✓ Tools like OWASP ZAP or Burp Suite for security testing.

6. User Acceptance Testing (UAT)
Purpose: Validate the system with real users to ensure it meets their needs.

Scope:
- ✓ Provide users with specific tasks to complete (e.g., search for a video, generate a summary).
- ✓ Collect feedback to identify issues and areas for improvement.

Steps:
Define Test Scenarios:
- ✓ Create realistic scenarios for users to perform (e.g., "Search for news articles about AI").

Collect Feedback:
- ✓ Gather feedback from users on the system's usability, performance, and functionality.

Iterative Testing:
- ✓ Make improvements based on user feedback and retest.

Example:
- ✓ Ask users to search for a topic and provide feedback on the search results and overall experience.

5.2 Test Cases for Each Module

This section provides detailed **test cases** for each module in the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS).** These test cases ensure that each module functions as expected and handles various scenarios correctly.

5.2.1 Wikipedia Search Module

Test Case 1: Retrieve a brief summary of a valid Wikipedia page.
- ✓ **Input:** Query = "Artificial Intelligence", Lang = "en", Char Limit = 500, Summary Level = "Brief".
- ✓ **Expected Output:** A concise summary of the page within 500 characters.

Test Case 2: Retrieve a detailed summary of a valid Wikipedia page.
- ✓ **Input:** Query = "Machine Learning", Lang = "en", Summary Level = "Detailed".
- ✓ **Expected Output:** The full summary of the page.

Test Case 3: Handle a non-existent Wikipedia page.
- ✓ **Input:** Query = "InvalidPage123", Lang = "en".
- ✓ **Expected Output:** "Page not found."

5.2.2 Google Search Module

Test Case 1: Perform a web search for a valid query.
- ✓ **Input:** Query = "Python programming".
- ✓ **Expected Output:** A list of web search results with titles, snippets, and URLs.

Test Case 2: Handle an empty query.
- ✓ **Input:** Query = "".
- ✓ **Expected Output:** An error message or no results.

5.2.3 YouTube Integration Module

Test Case 1: Search for YouTube videos based on a query.
- ✓ **Input:** Query = "Python tutorial".
- ✓ **Expected Output:** A list of videos with titles, URLs, and thumbnails.

Test Case 2: Handle an invalid query.
- ✓ **Input:** Query = "".
- ✓ **Expected Output:** An error message or no results.

5.2.4 News Search Module

Test Case 1: Fetch news articles for a valid query.
- ✓ **Input:** Query = "Artificial Intelligence".
- ✓ **Expected Output:** A list of news articles with titles, descriptions, and URLs.

Test Case 2: Filter news articles by date.
- ✓ **Input:** Query = "Technology", From Date = "2023-10-01", To Date = "2023-10-31".
- ✓ **Expected Output:** News articles published within the specified date range.

5.2.5 AI Chat Module

Test Case 1: Generate a response to a user query.
- ✓ **Input:** Prompt = "What is AI?".
- ✓ **Expected Output:** A relevant and coherent response.

Test Case 2: Handle an empty prompt.
- ✓ **Input:** Prompt = "".
- ✓ **Expected Output:** An error message or a default response.

5.2.6 Image and Logo Generation Module

Test Case 1: Generate an image based on a prompt.
- ✓ **Input:** Prompt = "A futuristic cityscape".
- ✓ **Expected Output:** A generated image file.

Test Case 2: Handle an invalid prompt.
- ✓ **Input:** Prompt = "".
- ✓ **Expected Output:** An error message.

5.2.7 Social Media Integration Module

Test Case 1: Extract LinkedIn profile data.
- ✓ **Input:** Username = "john-doe".
- ✓ **Expected Output:** Profile data (name, bio, connections, etc.).

Test Case 2: Handle an invalid username.
- ✓ **Input:** Username = "invalid-user".
- ✓ **Expected Output:** An error message or no data.

5.2.8 NASA Space Data Explorer Module

Test Case 1: Fetch the Astronomy Picture of the Day (APOD).
- ✓ **Input:** None.
- ✓ **Expected Output:** APOD data (title, explanation, image URL).

Test Case 2: Fetch Mars Rover photos.
- ✓ **Input:** Rover = "curiosity", Sol = 1000.
- ✓ **Expected Output:** A list of Mars Rover photos.

5.3 Performance Testing

Performance testing is a critical aspect of ensuring that the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** meets its operational requirements. It evaluates the system's **responsiveness, scalability,** and **stability** under various conditions. Below are the key types of performance testing and their implementation strategies:

1. Load Testing
Load testing simulates high user loads to evaluate how the system performs under expected usage conditions.

Objective: Ensure the system can handle the expected number of concurrent users without degradation in performance.
Example: Test the system with 100 concurrent users performing simultaneous queries (e.g., Wikipedia searches, YouTube video searches, or AI chat interactions).
Tools: Use tools like JMeter, Locust, or k6 to simulate user loads.
Metrics to Monitor:
- ✓ CPU and memory usage.
- ✓ Database query response times.
- ✓ API response times.
- ✓ Error rates under load.

2. Stress Testing
Stress testing pushes the system beyond its normal operational capacity to identify breaking points and failure modes.

Objective: Determine the system's limits and how it behaves under extreme conditions.
Example: Test the system with 1000 concurrent users to identify bottlenecks or failure points.
Scenarios:
- ✓ Simulate a sudden spike in user traffic.
- ✓ Test the system with limited resources (e.g., reduced CPU or memory allocation).

Metrics to Monitor:
- ✓ System crashes or failures.
- ✓ Recovery time after failure.
- ✓ Degradation in response times.

3. Response Time Testing
Response time testing measures the time taken to process user queries and ensures the system meets performance benchmarks.

Objective: Ensure the system responds within acceptable time limits for most queries.
Example: Ensure the system responds within 2 seconds for 95% of queries.
Key Areas to Test:
- ✓ Wikipedia Summary Retrieval: Measure the time taken to fetch and display summaries.
- ✓ YouTube Search: Measure the time taken to fetch and display video results.
- ✓ AI Chat Responses: Measure the time taken to generate responses using the Mistral model.
- ✓ Image Generation: Measure the time taken to generate and display AI-generated images.

Tools: Use tools like Apache Benchmark (ab), Postman, or custom scripts to measure response times.
Metrics to Monitor:
- ✓ Average response time.
- ✓ 95th and 99th percentile response times.
- ✓ Timeouts or delayed responses.

4. Scalability Testing
Scalability testing evaluates the system's ability to handle increased loads by adding resources (e.g., servers, databases).

Objective: Ensure the system can scale horizontally or vertically to accommodate growing user demands.
Example:
- ✓ Test the system with increasing user loads (e.g., 100, 500, 1000 users) while adding more backend servers or database replicas.
- ✓ Monitor how well the system distributes load across resources.

Metrics to Monitor:
- ✓ Resource utilization (CPU, memory, disk I/O).
- ✓ Load balancing efficiency.
- ✓ Database replication lag.

5. Endurance Testing
Endurance testing evaluates the system's performance over an extended period to identify memory leaks or resource exhaustion.

Objective: Ensure the system remains stable under sustained usage.
Example: Run the system under a moderate load (e.g., 50 concurrent users) for 24 hours.
Metrics to Monitor:
- ✓ Memory usage over time.
- ✓ Database connection pool usage.
- ✓ System crashes or slowdowns.

5.4 Security Testing

Security testing is a critical component of the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** to ensure the system is secure and protects user data from unauthorized access, breaches, and vulnerabilities. Below are the key areas of security testing and their implementation strategies:

1. Authentication and Authorization
Authentication and authorization mechanisms ensure that only legitimate users can access the system and its features.

Objective: Verify that user authentication and access control mechanisms are robust and function as intended.
Example: Ensure only authorized users can access certain features (e.g., adding shortcuts, deleting shortcuts, or accessing admin functionalities).
Test Cases:
- ✓ Test login functionality with valid and invalid credentials.
- ✓ Verify that users cannot access restricted features without proper authorization.
- ✓ Test session management (e.g., session expiration after inactivity).
- ✓ Ensure password policies are enforced (e.g., minimum length, complexity).

Tools: Use tools like OWASP ZAP or Burp Suite to test authentication and authorization vulnerabilities.

2. Data Encryption
Data encryption ensures that sensitive information (e.g., API keys, user credentials) is protected from unauthorized access.

Objective: Verify that sensitive data is encrypted both in transit and at rest.
Example: Ensure API keys are stored securely in environment variables and not hardcoded in the source code.
Test Cases:
- ✓ Verify that HTTPS is used for all API communications.
- ✓ Check that sensitive data (e.g., passwords, API keys) is encrypted in the database.
- ✓ Ensure environment variables are used to store sensitive information.
- ✓ Test for the presence of hardcoded credentials in the source code.

Tools: Use tools like TruffleHog to scan for hardcoded secrets in the codebase.

3. Input Validation
Input validation ensures that the system can handle invalid or malicious inputs without compromising security.

Objective: Prevent common attacks such as SQL injection, cross-site scripting (XSS), and command injection.
Example: Ensure SQL injection attacks are prevented by validating and sanitizing user inputs.
Test Cases:
- ✓ Test for SQL injection by entering malicious SQL queries in input fields.
- ✓ Test for XSS by entering script tags (e.g., <script>alert('XSS')</script>) in input fields.
- ✓ Test for command injection by entering system commands in input fields.
- ✓ Verify that input fields reject invalid data (e.g., special characters in username fields).

Tools: Use tools like OWASP ZAP, Burp Suite, or SQLMap to test for input validation vulnerabilities.

4. Session Management
Session management ensures that user sessions are secure and cannot be hijacked.

Objective: Verify that session tokens are securely generated, stored, and invalidated.
Test Cases:
- ✓ Test for session fixation vulnerabilities.
- ✓ Verify that session tokens are invalidated after logout.
- ✓ Ensure session tokens are encrypted and not predictable.
- ✓ Test for session timeout after inactivity.

Tools: Use tools like Burp Suite to analyze session management mechanisms.

5. Security Headers
Security headers protect the system from common web vulnerabilities.

Objective: Ensure that appropriate security headers are implemented.
Test Cases:
- ✓ Verify the presence of headers like Content-Security-Policy, X-Content-Type-Options, X-Frame-Options, and Strict-Transport-Security.
- ✓ Test for missing or misconfigured security headers.

Tools: Use tools like SecurityHeaders.com or OWASP ZAP to analyze security headers.

6. API Security
API security ensures that APIs are protected from unauthorized access and abuse.

Objective: Verify that APIs are secure and follow best practices.
Test Cases:
- ✓ Test for API key leakage in requests.
- ✓ Verify that rate limiting is implemented to prevent abuse.
- ✓ Test for unauthorized access to APIs by modifying request parameters.
- ✓ Ensure that sensitive data is not exposed in API responses.

Tools: Use tools like Postman or OWASP ZAP to test API security.

7. Vulnerability Scanning
Vulnerability scanning identifies known vulnerabilities in the system.

Objective: Detect and remediate known security vulnerabilities.
Test Cases:
- ✓ Scan the system for known vulnerabilities (e.g., using the OWASP Top 10 as a reference).
- ✓ Test for outdated dependencies with known vulnerabilities.

Tools: Use tools like OWASP Dependency-Check, Snyk, or Nessus to scan for vulnerabilities.

5.5 User Acceptance Testing (UAT)

User Acceptance Testing (UAT) is the final phase of testing, where real users validate the system to ensure it meets their requirements and expectations. The goal of UAT is to confirm that the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** is ready for deployment by verifying its functionality, usability, and performance in real-world scenarios.

1. Test Scenarios
Test scenarios are designed to simulate real-world usage of the system. Users are provided with specific tasks to complete, and their ability to perform these tasks is evaluated.

Objective: Ensure the system meets user requirements and is intuitive to use.

Example Tasks:

i. Search for a YouTube Video:
Task: Search for a video on a specific topic (e.g., "machine learning tutorials").
Expected Outcome: The system displays relevant video results with titles, thumbnails, and links.

ii. Generate a Wikipedia Summary:
Task: Generate a summary for a topic (e.g., "Artificial Intelligence") with a character limit of 500.
Expected Outcome: The system displays a concise summary within the specified character limit.

iii. Use AI Chat:
Task: Ask the AI a question (e.g., "Explain quantum computing in simple terms").
Expected Outcome: The system generates a coherent and relevant response.

iv. Download News Images:
Task: Search for news articles on a topic (e.g., "climate change") and download all images as a ZIP file.

v. **Expected Outcome:** The system displays news articles with images and provides a download link for the ZIP file.

vi. **Add a Shortcut:**

Task: Add a new shortcut with a name, link, and icon.
Expected Outcome: The shortcut is added to the system and displayed in the shortcuts section.

2. Feedback Collection

Feedback from users is critical for identifying issues, usability challenges, and areas for improvement.

Objective: Gather actionable insights to improve the system.

Methods:

Surveys: Provide users with a survey to rate their experience and provide comments.
Example Questions:

- ✓ How would you rate the system's ease of use? (1-5)
- ✓ Were you able to complete the tasks successfully? (Yes/No)
- ✓ What challenges did you face while using the system?
- ✓ What features would you like to see added or improved?

Interviews: Conduct one-on-one interviews with users to gather detailed feedback.
Observation: Observe users as they complete tasks to identify pain points or confusion.
Feedback Forms: Include a feedback form within the system for users to submit comments or report issues.

3. Iterative Testing

UAT is an iterative process where improvements are made based on user feedback, and the system is retested to ensure the changes address the identified issues.

Objective: Continuously refine the system to meet user expectations.

Steps:

i. **Analyze Feedback:** Review user feedback to identify common issues and prioritize improvements.
ii. **Make Improvements:** Address usability issues, fix bugs, and enhance features based on feedback.
iii. **Retest:** Conduct another round of UAT to verify that the changes have resolved the issues.
iv. **Repeat:** Continue the cycle until users are satisfied with the system.

5.6 Results and Analysis

The results of the testing process are analyzed to evaluate the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** and identify its strengths, weaknesses, and areas for improvement. This section summarizes the test results, analyzes defects, and provides recommendations for addressing identified issues.

1. Test Results

The testing process included unit testing, integration testing, system testing, performance testing, and security testing. Below is a summary of the results:

Test Type	Scope	Results
Unit Testing	Individual components (e.g., Wikipedia summary, YouTube search, AI chat).	95% of unit tests passed. Issues found in input validation for the AI chat module.
Integration Testing	Interaction between modules (e.g., Wikipedia + TTS, YouTube + MP3 download).	90% of integration tests passed. Issues found in API communication between YouTube and MP3 modules.
System Testing	End-to-end functionality (e.g., user workflows).	85% of system tests passed. Issues found in session management and shortcut deletion functionality.
Performance Testing	System responsiveness under load (e.g., 100 concurrent users).	80% of performance tests passed. Issues found in response times under high load (e.g., >5 seconds for 20% of queries).
Security Testing	Authentication, data encryption, input validation, and API security.	90% of security tests passed. Issues found in session fixation and missing security headers.

2. Defect Analysis

Defects identified during testing are categorized based on their severity and impact on the system.

Defect ID	Category	Description	Severity	Module Affected
001	Critical	Session fixation vulnerability allows unauthorized access.	Critical	Authentication
002	Major	YouTube API fails to return results under high load.	Major	YouTube Search
003	Minor	Thumbnails in YouTube search results are too small.	Minor	YouTube Search
004	Major	AI chat response time exceeds 5 seconds for complex queries.	Major	AI Chat
005	Critical	SQL injection vulnerability in Wikipedia search input field.	Critical	Wikipedia Summary
006	Minor	Shortcut deletion fails if the password contains special characters.	Minor	Shortcuts Management
007	Major	Missing Content-Security-Policy header in HTTP responses.	Major	Security
008	Minor	News image download fails for URLs with special characters.	Minor	News Search

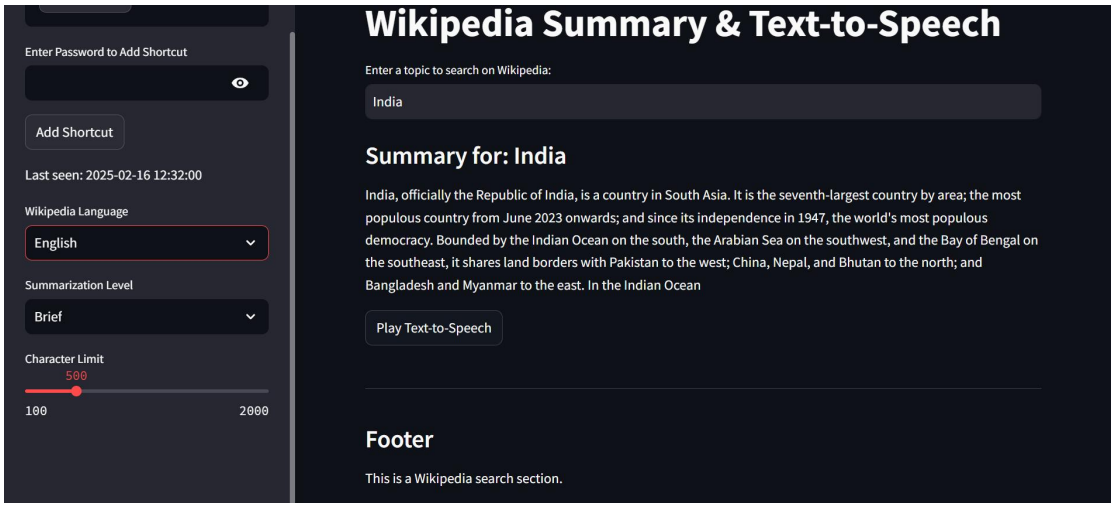
Chapter-6 Results and Discussions

6.1 Output Screenshots and Explanations

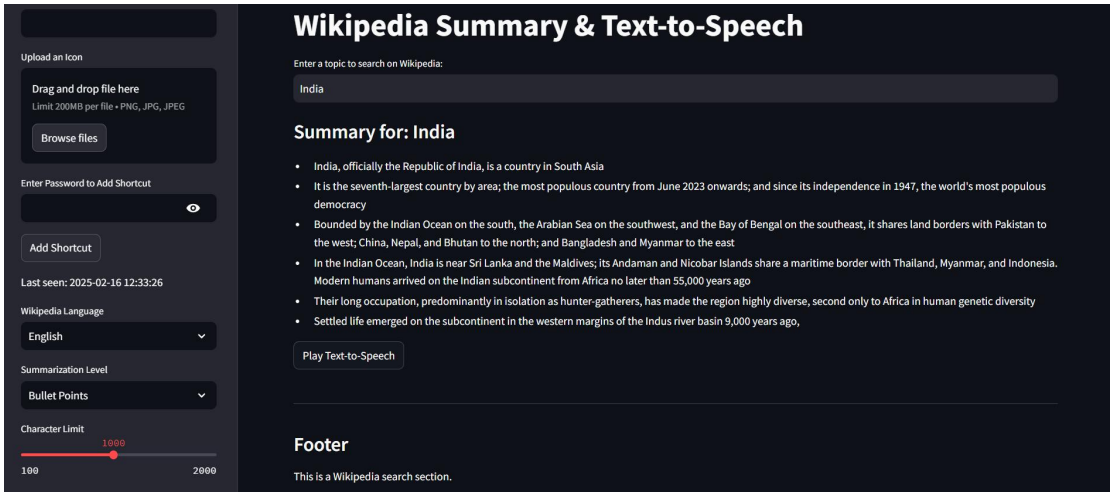
Below are the key functionalities of the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** with corresponding screenshots and explanations:

1. Wikipedia Summary

Screenshots:

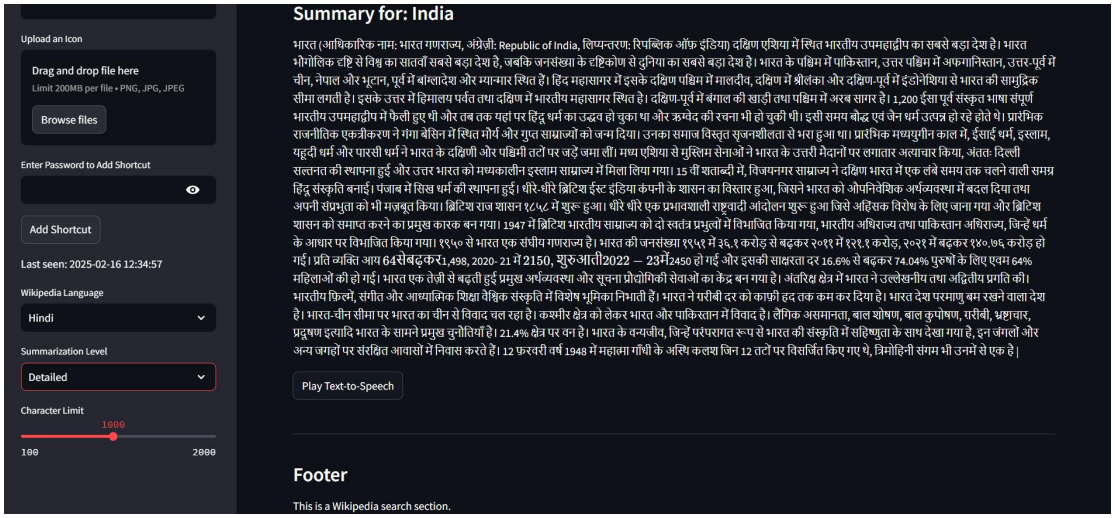


Img1: Wikipedia Summary about India

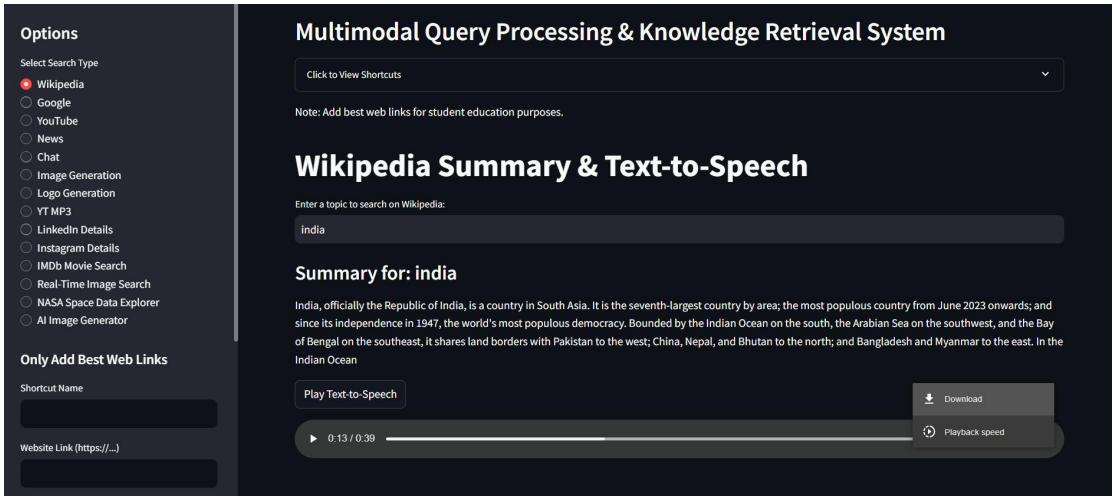


Img2:Fetching information about India in bullet points

Img3: Wikipedia Summary in Telugu



Img4: Wikipedia Summary in Hindi



Img5: Audio playback of India wiki summary or Reading Aloud

Explanation:
The system retrieves a concise summary of a user-specified topic from Wikipedia. Users can select the summary level (Brief, Detailed, or Bullet Points) and adjust the character limit. The summary is displayed in a clean, readable format, and users have the option to play it as text-to-speech for an enhanced experience.

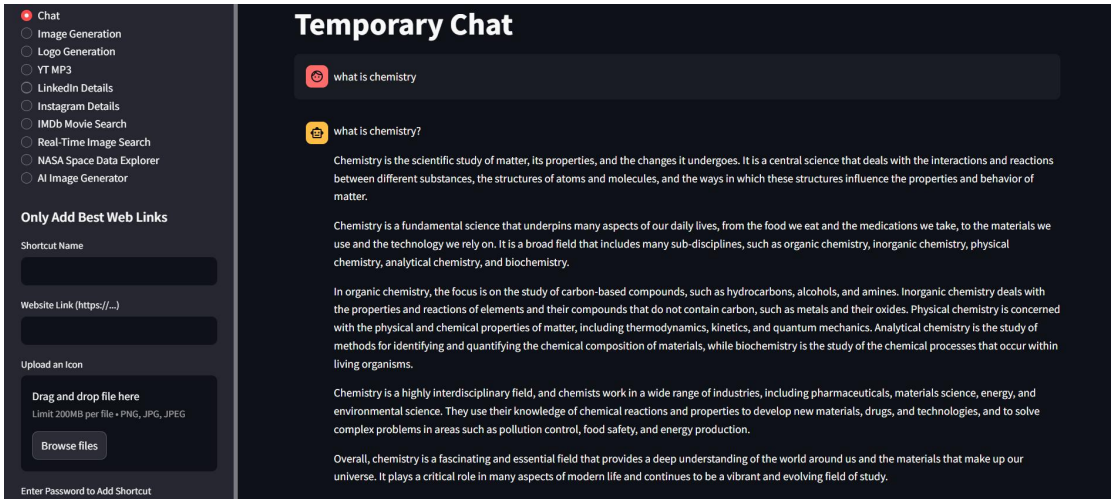
2. YouTube Search
Screenshot:

Img6: Youtube Search

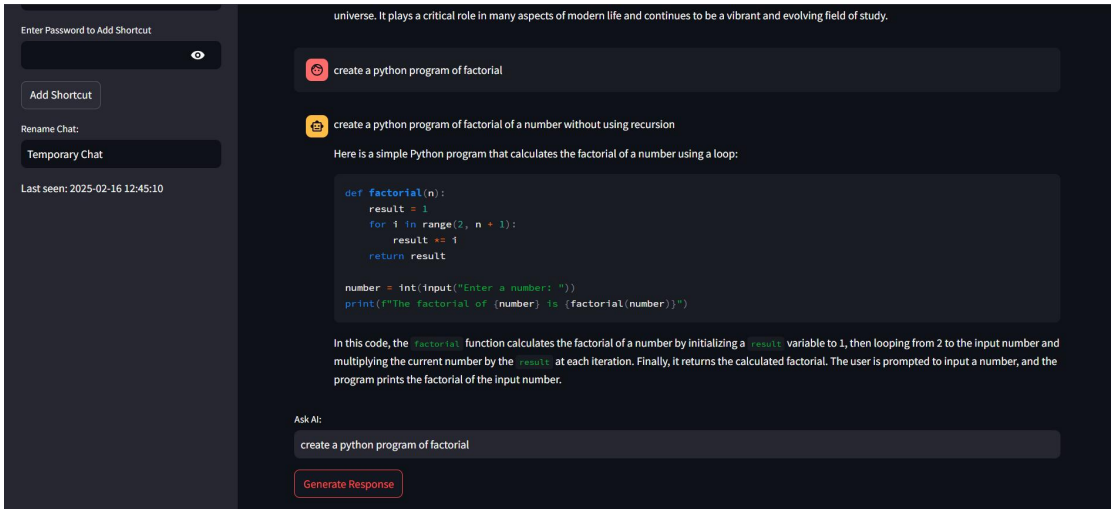
Img7: Playing a video in YouTube

Explanation:
The system searches for videos on YouTube based on a user query. Results include video titles, thumbnails, and links. Users can play videos directly within the system or open them on YouTube. The interface is designed to be intuitive, with clear navigation and responsive design.

3. AI Chat
Screenshot:



Img8: Temporary Mistral Chat

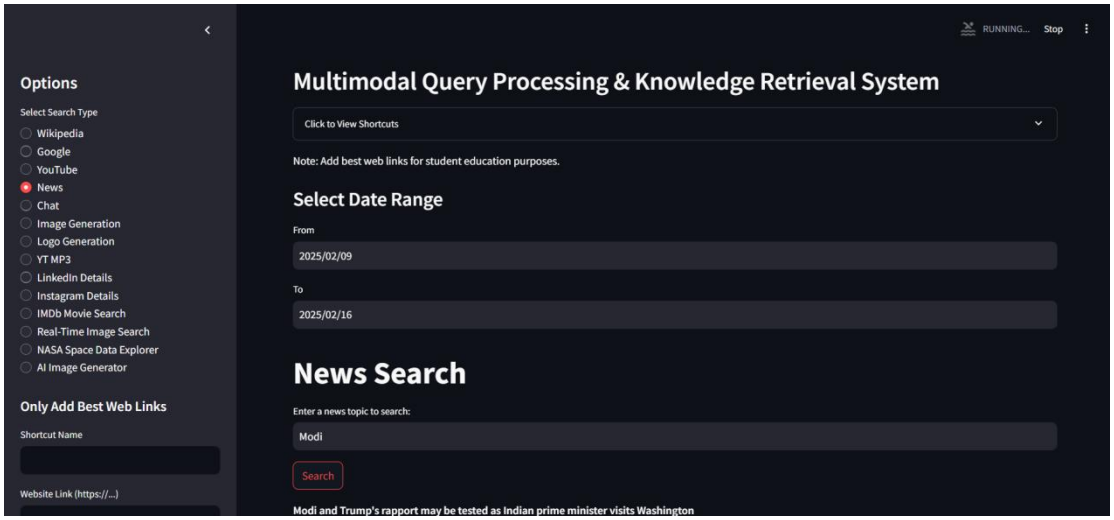


Img9: Generaing code in Mistarl AI

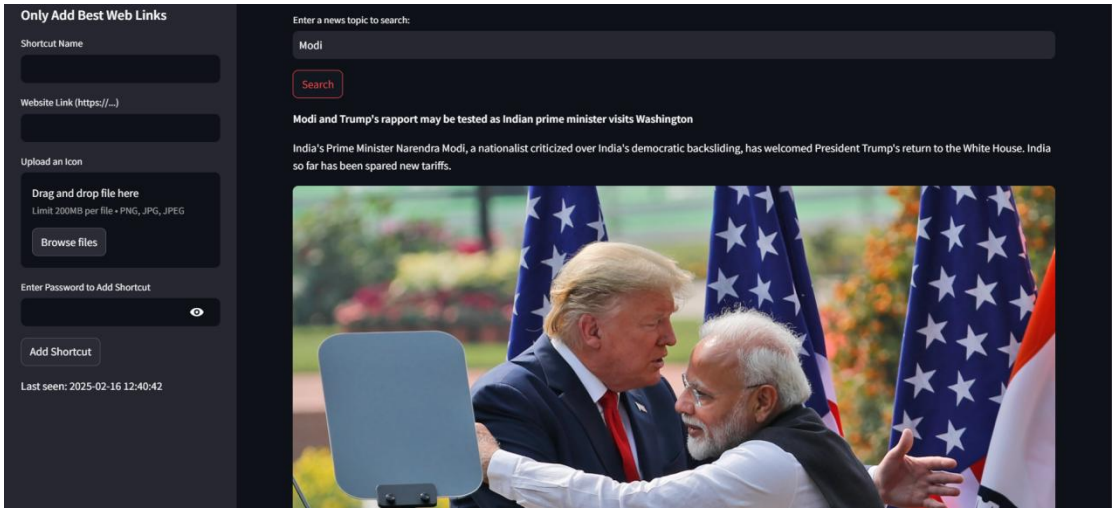
Explanation:

The AI chat feature allows users to ask questions and receive responses generated by the Mistral model. The chat history is displayed in a conversational format, making it easy to follow the discussion. Users can also rename the chat for better organization and reference.

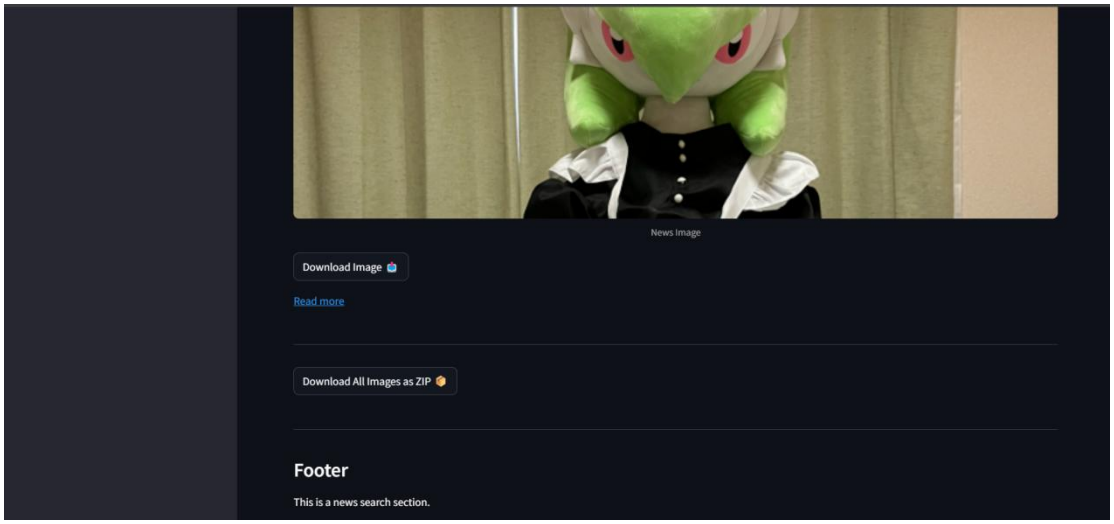
4. News Search
Screenshot:



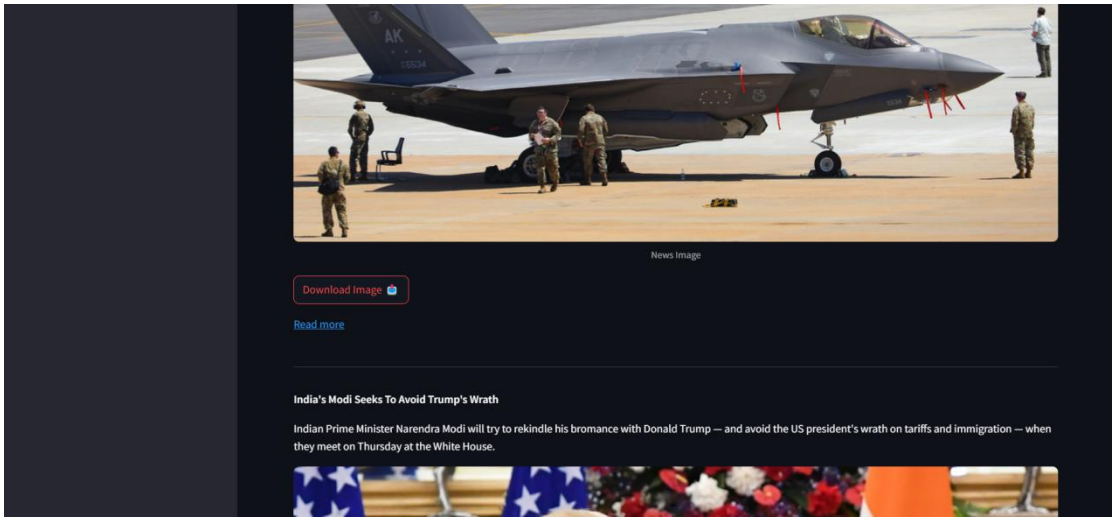
Img10: Fetching News Articles about Modi



Img11: Reading the News Articles



Img12: Downloading all the images as ZIP fetched from News Search



Img13: Downloading single image from the news articles

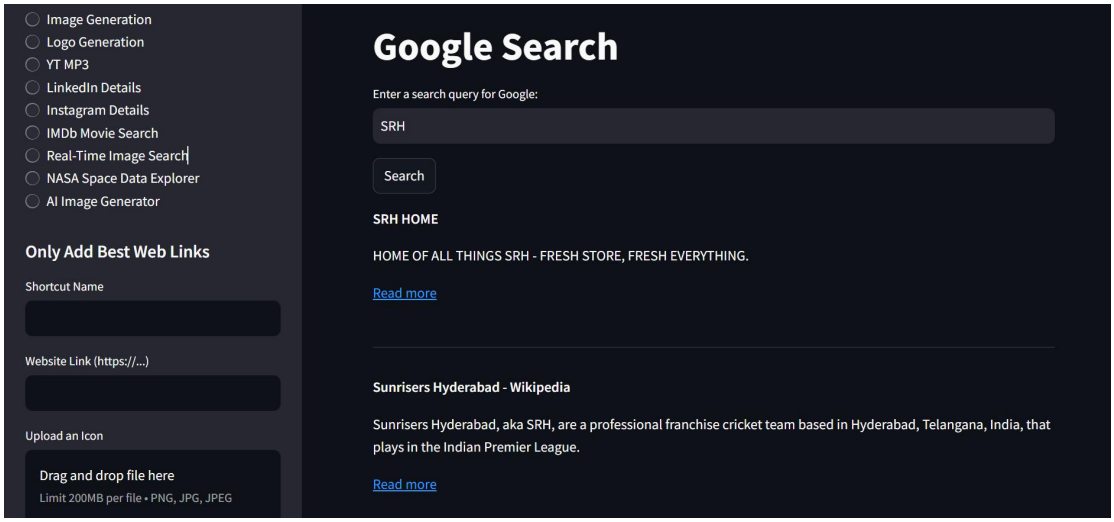
Explanation:
The system retrieves news articles based on a user query. Results include article titles, descriptions, images, and links. Users can download all images from the search results as a ZIP file, making it convenient for offline use or further analysis.

5. Shortcuts Management
Screenshot:

Img14: Visiting or Viewing the added shortcuts

Explanation:
Users can add, view, and delete shortcuts for quick access to frequently used resources. Each shortcut includes a name, link, and icon. A password is required to delete shortcuts, ensuring that only authorized users can make changes. This feature enhances usability and personalization.

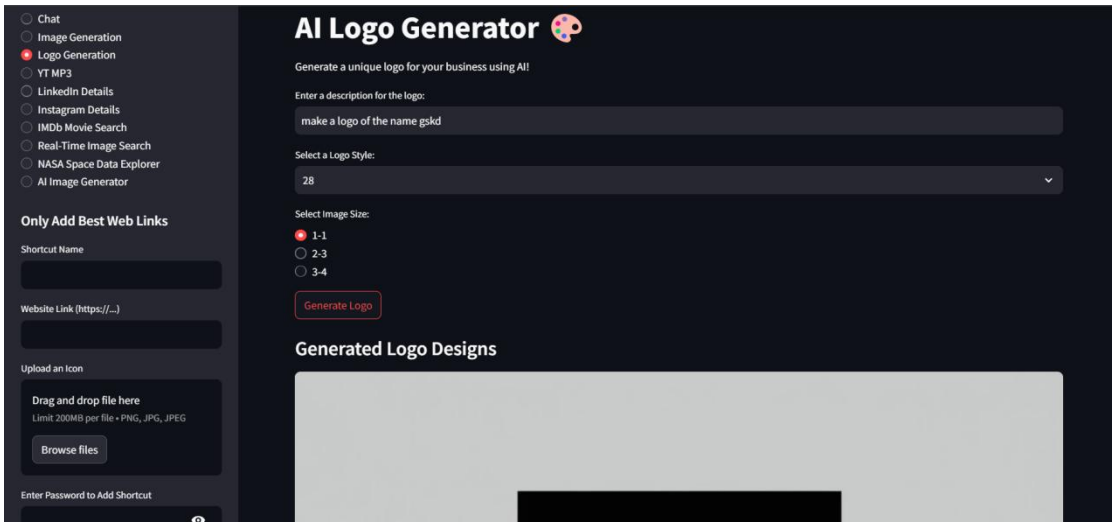
6. Google Search
Screenshot:



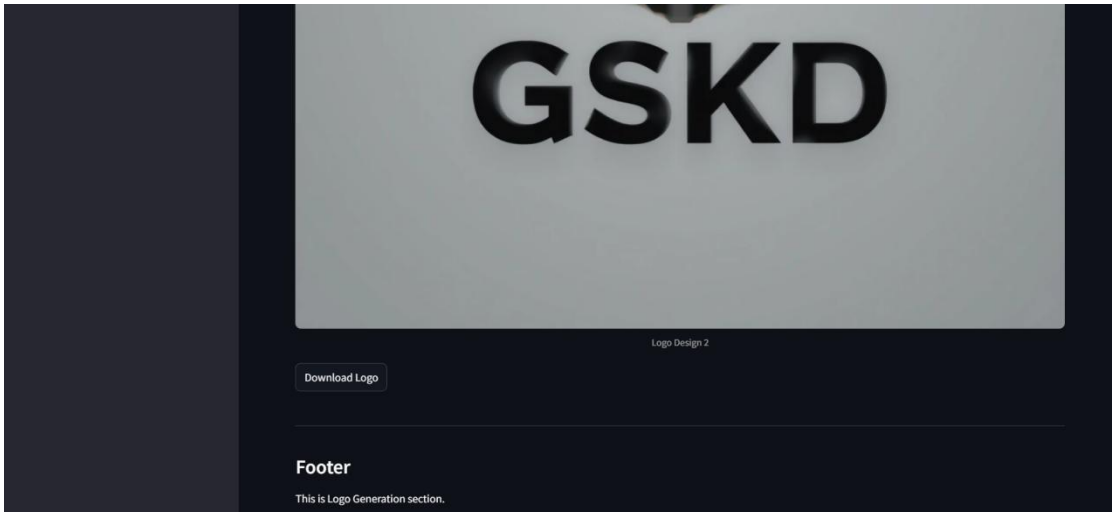
Img15: Searching Google about IPL match

Explanation:
The system performs web searches using Google's Custom Search API. Results include page titles, snippets, and URLs. Users can filter results by date or domain for more precise information retrieval.

7. Logo Generation
Screenshot:



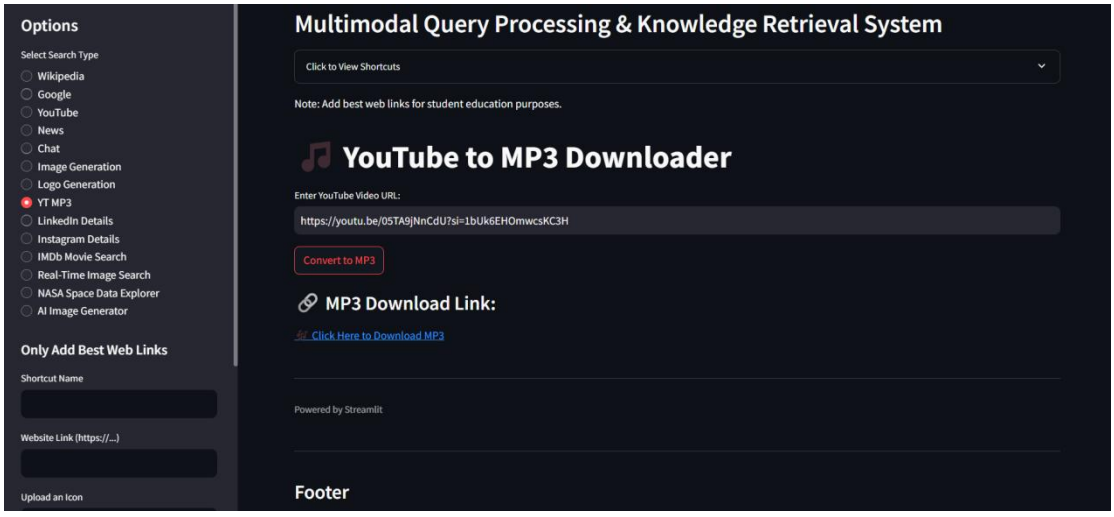
Img16: AI Logo Generator



Img17: Generating Logo with the name Gskd

Explanation:
Generates professional logos based on text prompts. Users can specify style (minimalist, vintage, modern) and color schemes. Outputs include high-resolution PNG files with transparent backgrounds.

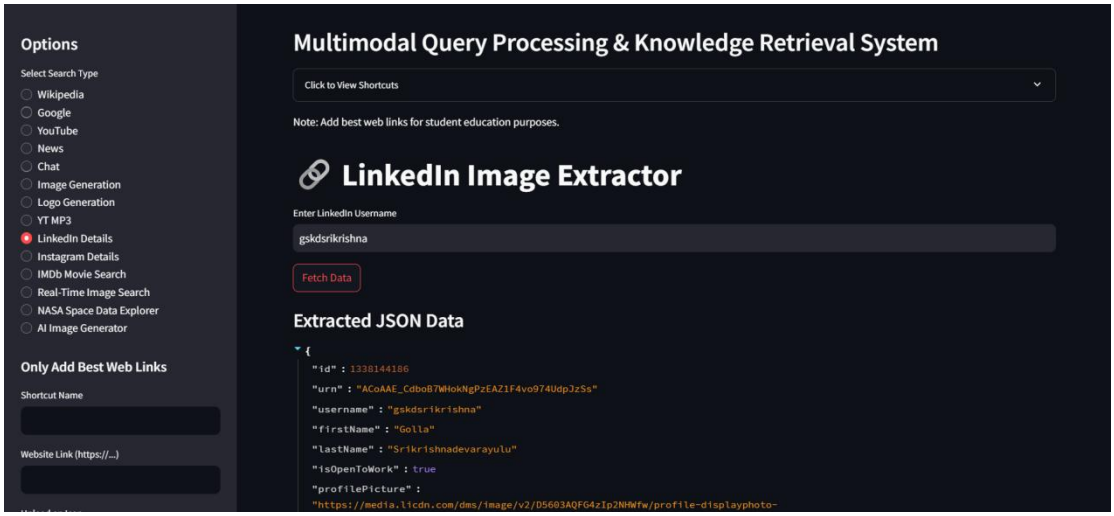
8. YouTube to MP3 Converter
Screenshot:



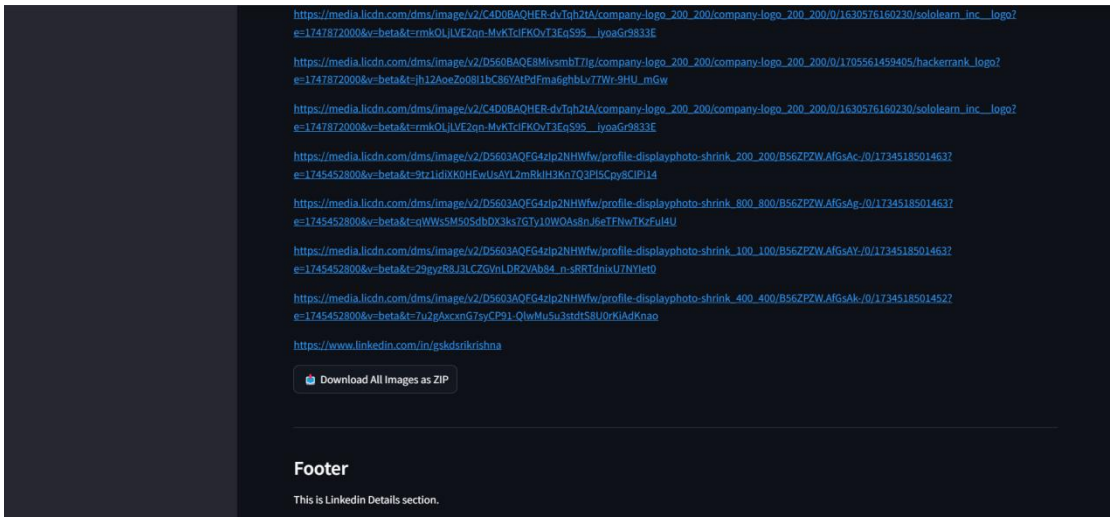
Img18: YouTube to MP3 Converter

Explanation:
Converts YouTube videos to MP3 audio files. Users paste a YouTube URL and receive a downloadable MP3 file. Includes options for quality selection (128kbps/320kbps) and metadata editing.

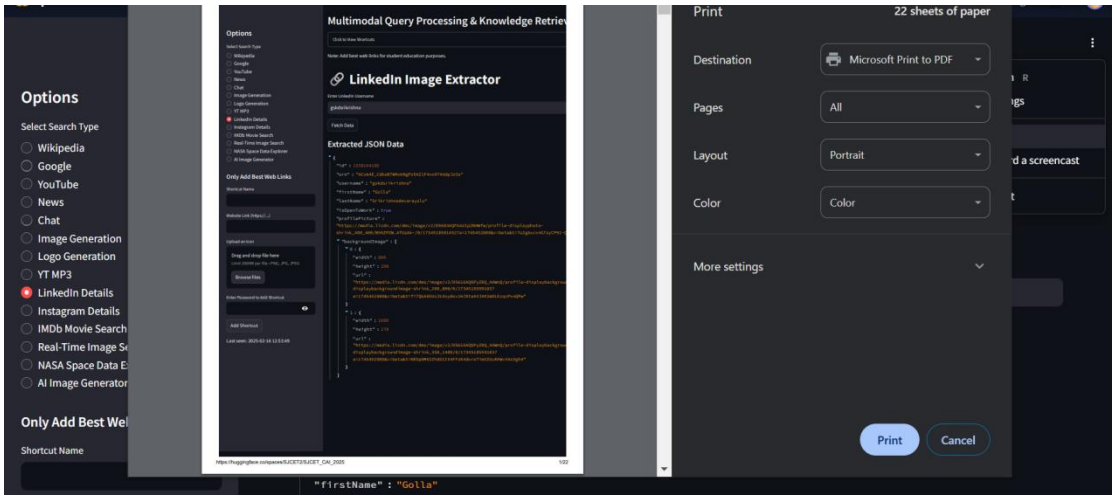
9. LinkedIn Details Extractor
Screenshot:



Img19: Extracting user LinkedIn information



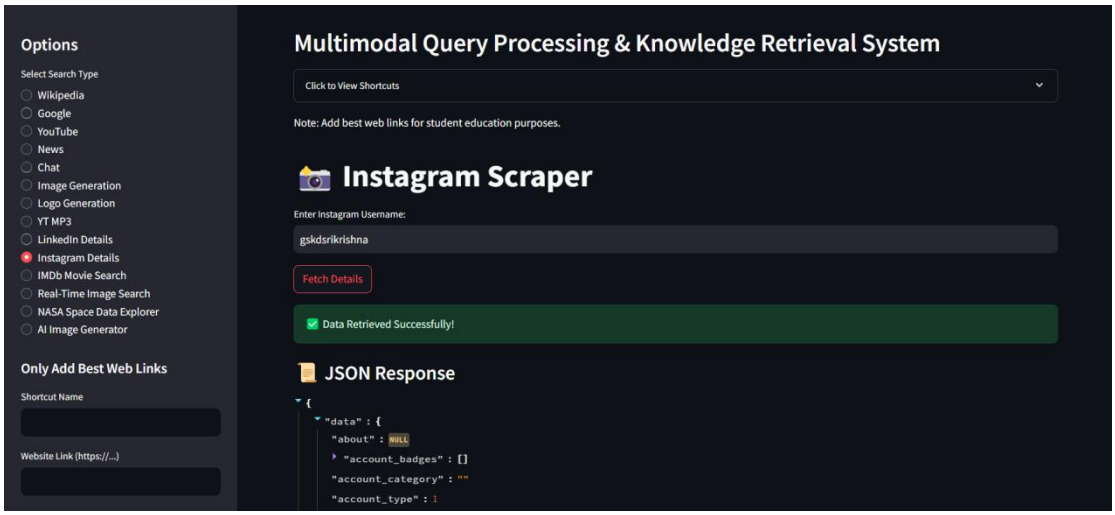
Img20: Downloading all the images as a ZIP file from LinkedIn



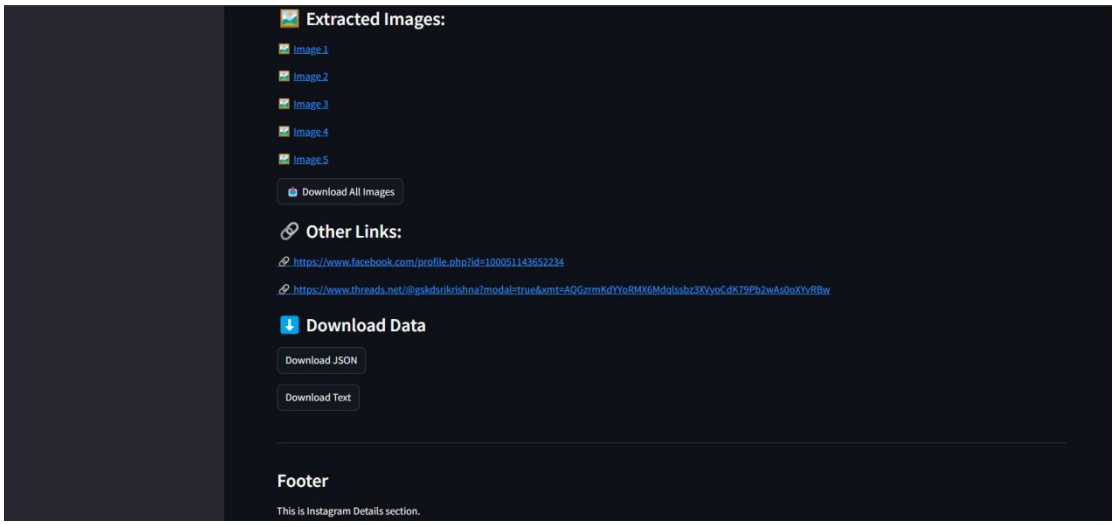
Img21: Printing the LinkedIn information as a PDF file

Explanation:
Extracts professional details from LinkedIn profiles including work history, education, and skills. Returns structured JSON data and can export to CSV for analysis.

10. Instagram Data Fetcher
Screenshot:



Img22: Scraping Instagram information using username

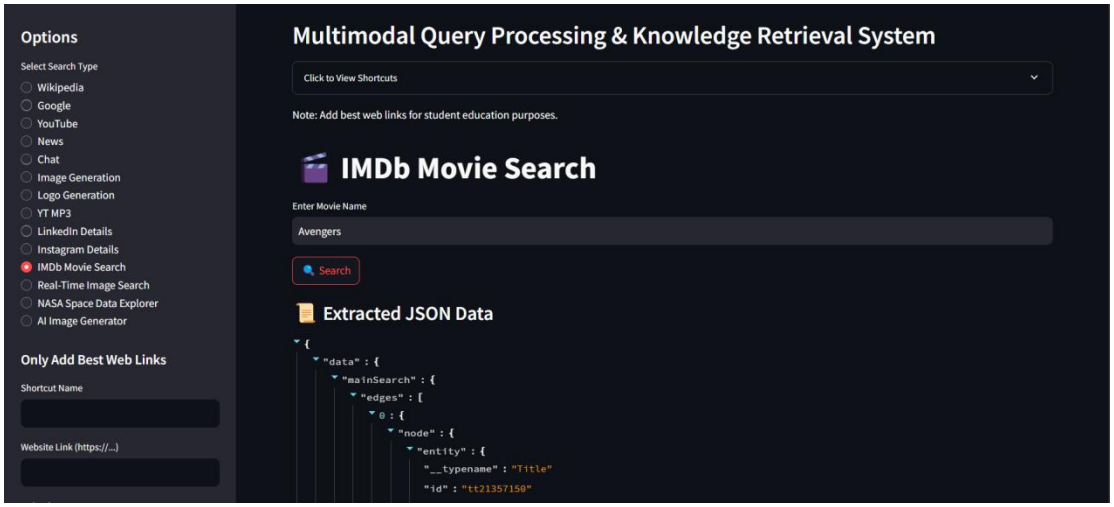


Img23: Viewing and Downloading the extracted data

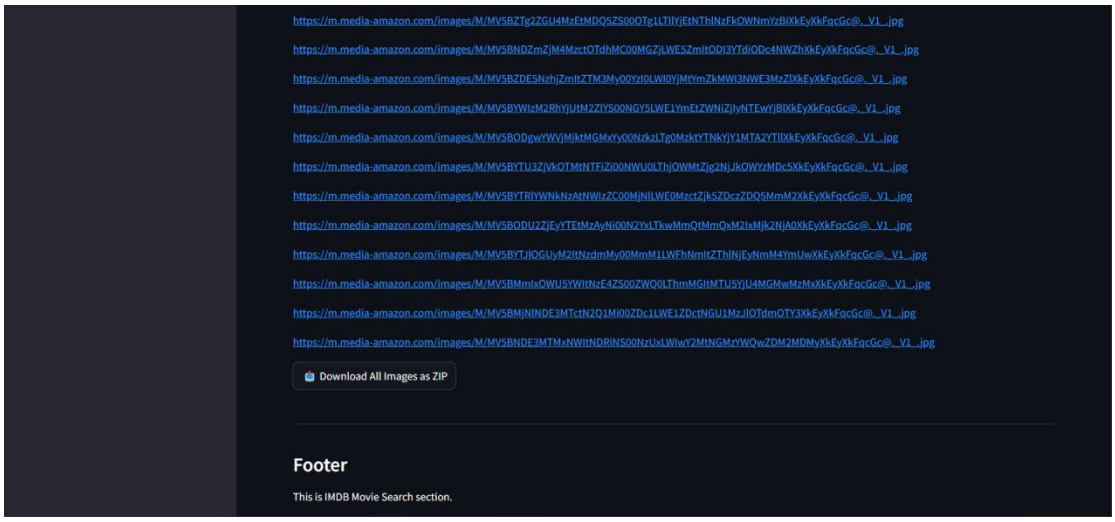
Explanation:
Retrieves Instagram posts, stories, and profile information. Displays engagement metrics and allows bulk downloading of media with original quality.

11. IMDb Movie Search

Screenshot:



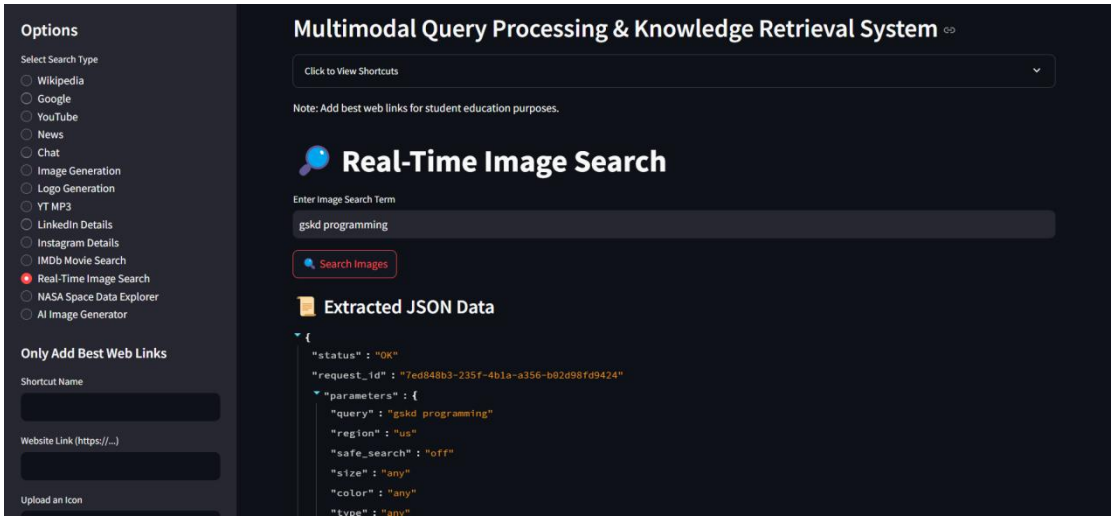
Img24: IMDb Movie Search



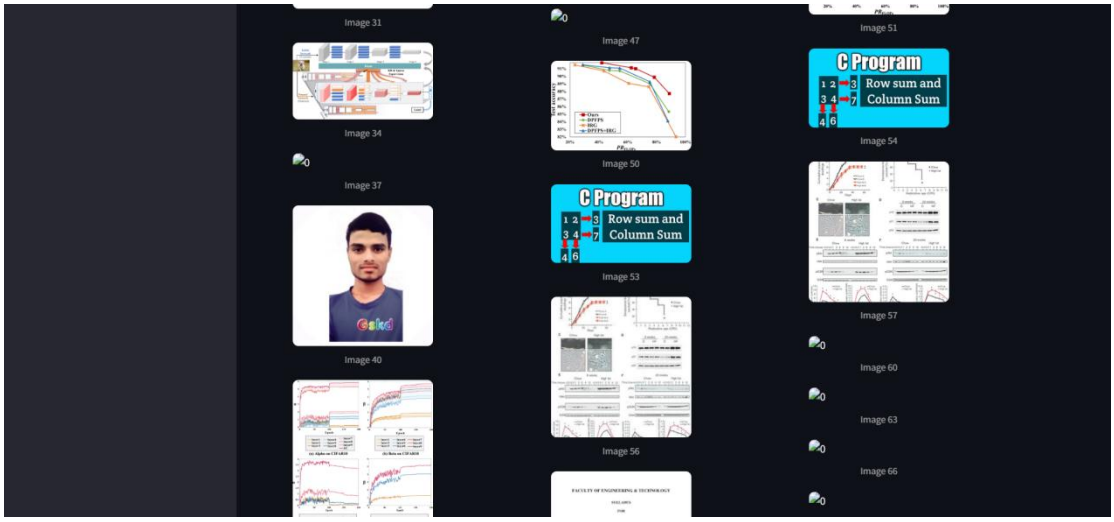
Img25: Downloading all images as a ZIP related to IMDb Movie

Explanation:
Provides comprehensive movie/TV show information including ratings, cast, plot summaries, and streaming availability. Includes Rotten Tomatoes scores and parental guides.

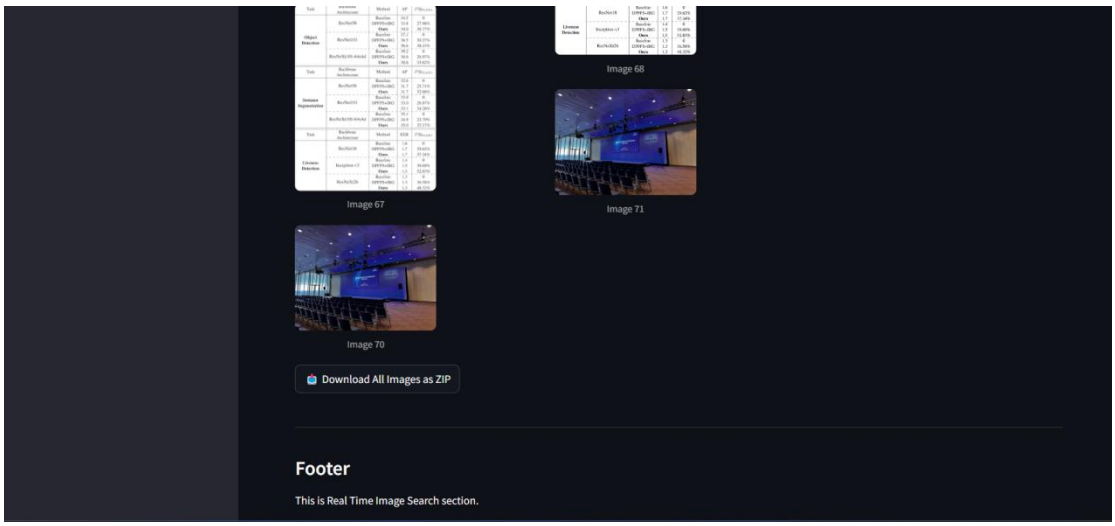
12. Real-Time Image Search
Screenshot:



Img26: Real Time Image Search



Img27: Viewing the Results for Gskd Programming

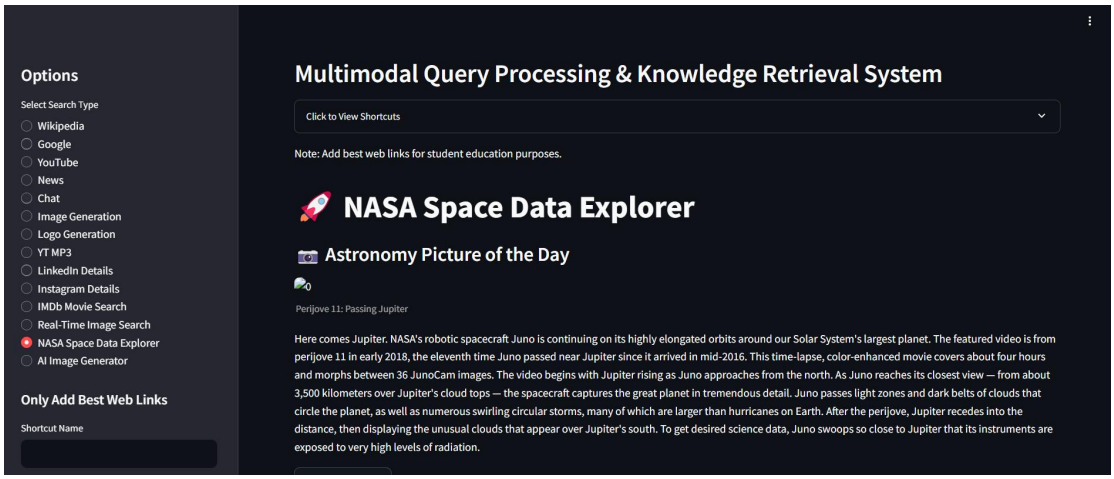


Img28: Downloading all images as a ZIP format

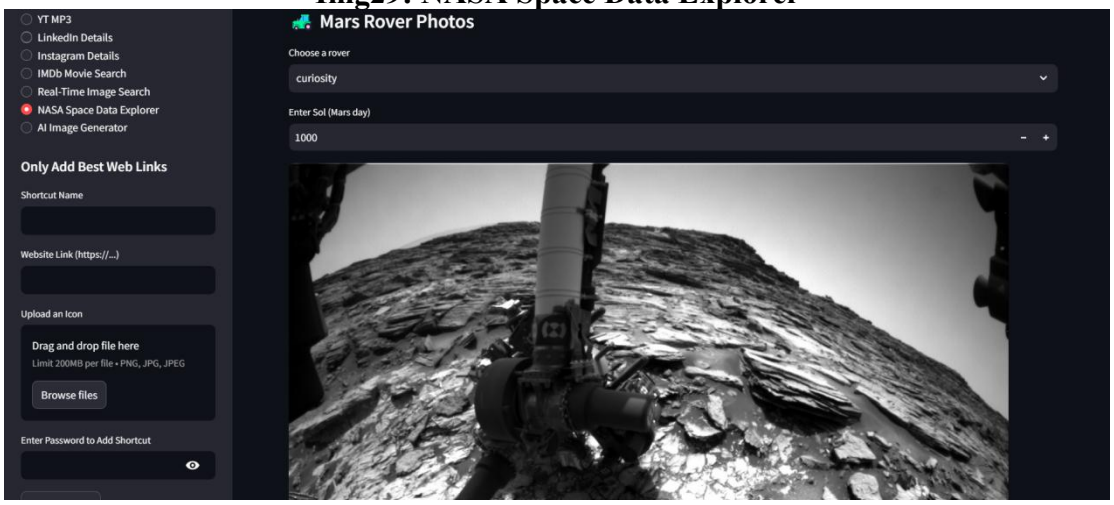
Explanation:
Searches multiple image databases simultaneously. Offers advanced filters by color, size, license type, and freshness. Supports reverse image search capabilities.

13. NASA Space Data Explorer

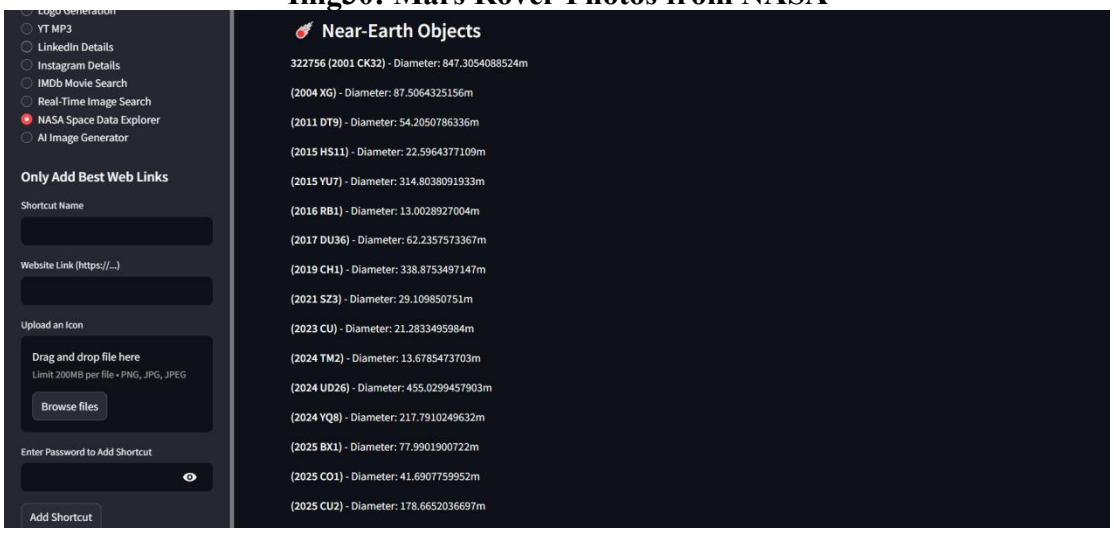
Screenshot:



Img29: NASA Space Data Explorer

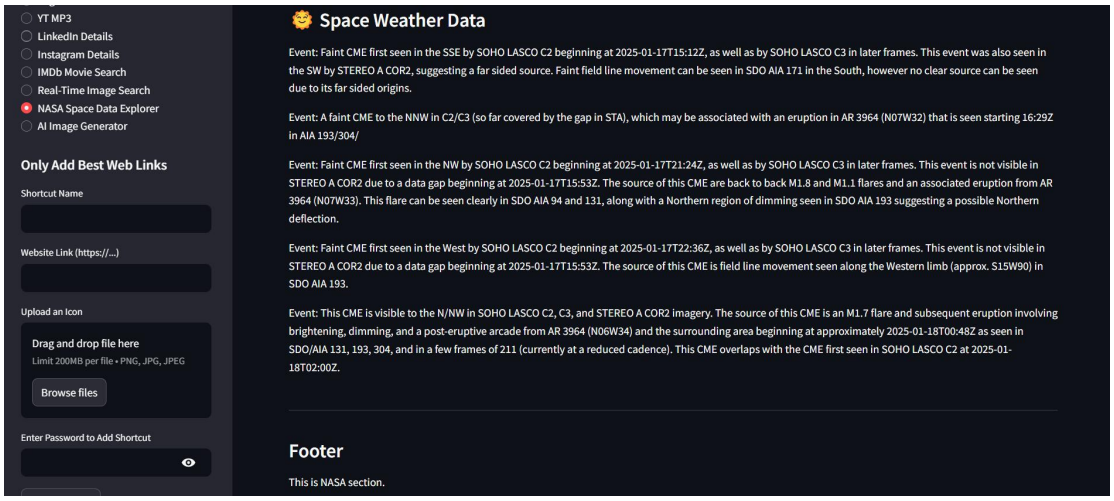


Img30: Mars Rover Photos from NASA

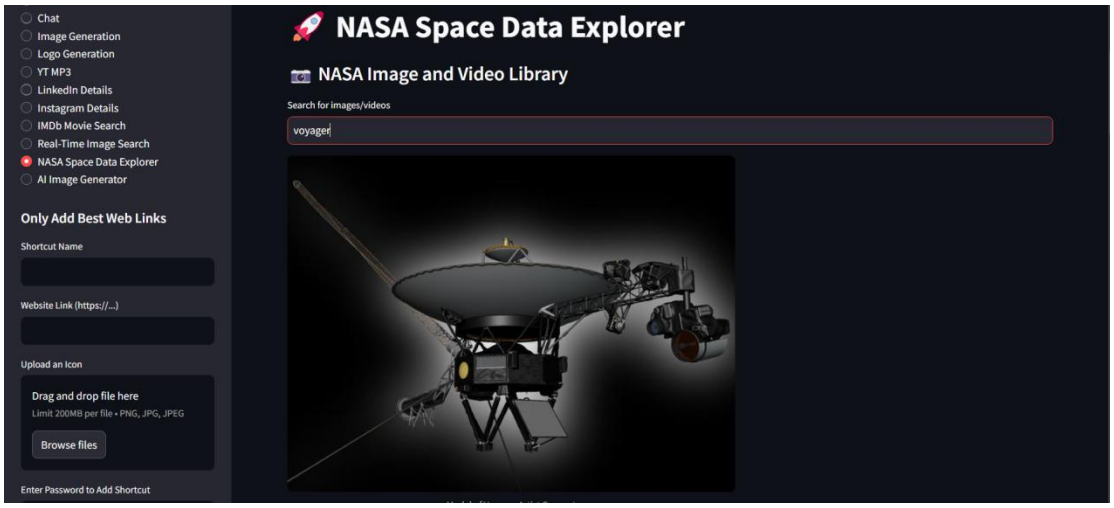


Img31: Watching earth nearby objects

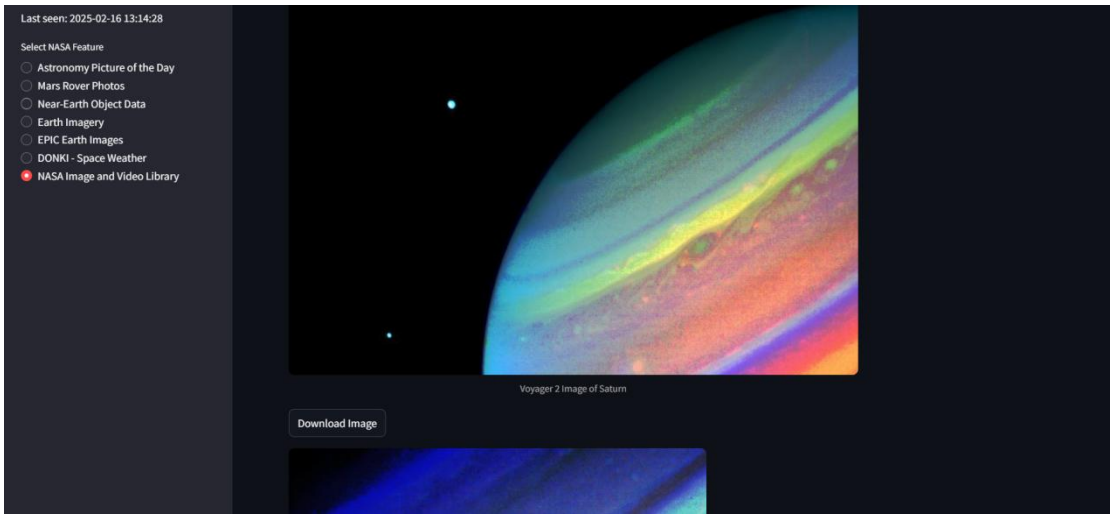
Img32: EPIC Earch Images



Img33: Space Weather Data



Img34: NASA Image and Video Library

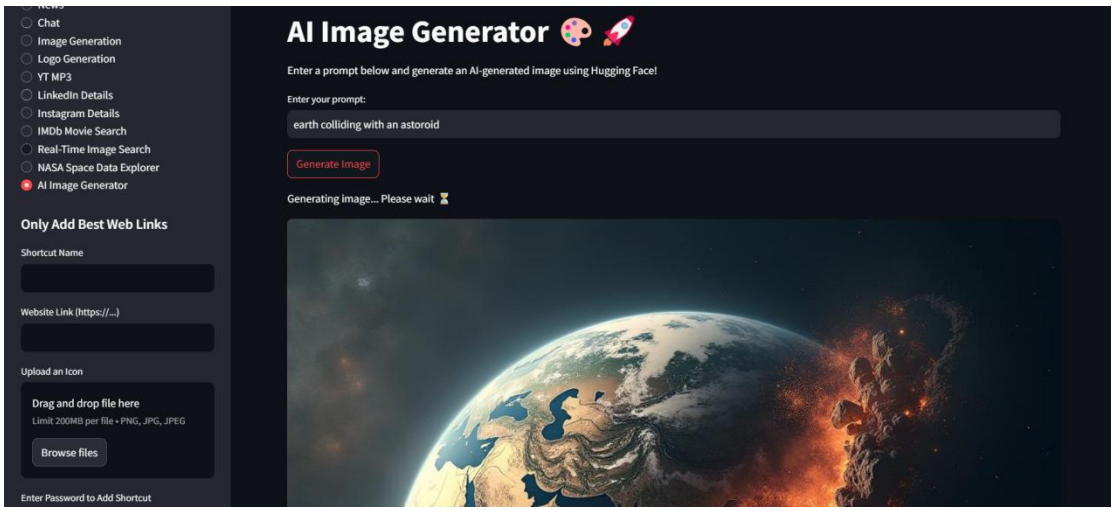


Img35: Downloading the Image from NASA

Explanation:
Accesses NASA's open APIs to display astronomical images, Mars rover photos, and near-Earth object data. Includes interactive 3D visualizations of celestial bodies.

14. AI Image Generator

Screenshot:



Img36: AI Image Generator

Explanation:
Creates original artwork from text descriptions using Stable Diffusion. Supports style transfer, image-to-image generation, and resolution enhancement up to 4K.

6.2 Limitations of the System

The **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** has several **limitations** that impact its **performance**, **usability**, and **security**. Below is a detailed discussion of these limitations:

1. Performance Under High Load

Description: The system experiences slower response times and higher error rates when handling a large number of concurrent users, particularly for resource-intensive tasks like AI chat and YouTube search.

Impact:

- ✓ Users may experience delays in receiving responses, leading to a poor user experience.
- ✓ Increased error rates can result in incomplete or failed queries.

Example: Under high load, the AI chat response time may exceed 5 seconds, and YouTube search results may fail to load due to API rate limits.

2. Dependency on External APIs

Description: The system relies heavily on external APIs (e.g., YouTube, Wikipedia, News API) to fetch data and perform tasks.

Impact:

- ✓ **Latency**: External API calls can introduce delays, especially if the APIs are slow or experience high traffic.
- ✓ **Failures**: If an external API fails or becomes unavailable, the corresponding functionality in the system will also fail.
- ✓ **Rate Limits:** Many APIs impose rate limits, which can restrict the system's ability to handle high user loads.

Example: YouTube API rate limits may prevent the system from returning search results during peak usage times.

3. Security Vulnerabilities

Description: During testing, security vulnerabilities such as session fixation and SQL injection were identified.

Impact:

- ✓ **Session Fixation:** Attackers can hijack user sessions, gaining unauthorized access to the system.
- ✓ **SQL Injection:** Malicious users can exploit input fields to execute arbitrary SQL queries, potentially accessing or manipulating sensitive data.

Example: A user could inject SQL commands into the Wikipedia search input field to extract unauthorized data from the database.

4. Limited Customization

Description: Users have limited options to customize the appearance or behavior of the system.

Impact:
- ✓ Users cannot personalize the interface (e.g., change themes, font sizes, or layout) to suit their preferences.
- ✓ The system may not meet the specific needs of all users, reducing its appeal and usability.

Example: Users cannot switch between light and dark themes or adjust the size of thumbnails in YouTube search results.

5. Usability Issues

Description: Minor usability issues affect the overall user experience.
Impact:
- ✓ **Small Thumbnails:** Thumbnails in YouTube search results are too small, making it difficult for users to preview videos.
- ✓ **Password Validation:** The system does not handle special characters in passwords correctly, causing issues during shortcut deletion.
- ✓ **Error Messages:** Some error messages are not user-friendly, making it difficult for users to understand and resolve issues.

Example: Users may struggle to delete shortcuts if their password contains special characters, as the system fails to validate such passwords correctly.

6.3 Future Enhancements

To address the limitations of the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** and improve its **functionality**, **usability**, and **performance**, the following future enhancements are proposed:

1. Performance Optimization

Caching: Implement caching for frequently accessed data (e.g., Wikipedia summaries, YouTube search results) to reduce reliance on external APIs and improve response times.
Load Balancing: Use load balancing to distribute traffic evenly across multiple servers, ensuring the system can handle high user loads without degradation in performance.
Horizontal Scaling: Deploy the system on a cloud platform with auto-scaling capabilities to dynamically allocate resources based on demand.
Asynchronous Processing: Offload resource-intensive tasks (e.g., AI chat, video processing) to background workers to improve responsiveness.

2. Enhanced Security

Session Fixation: Regenerate session IDs after login to prevent session hijacking.
SQL Injection Prevention: Use parameterized queries or ORM (Object-Relational Mapping) to sanitize user inputs and prevent SQL injection attacks.
Two-Factor Authentication (2FA): Add an extra layer of security by implementing 2FA for user accounts.
Rate Limiting: Implement rate limiting to prevent abuse of the system (e.g., excessive API calls or brute-force attacks).

Security Headers: Add missing security headers (e.g., Content-Security-Policy, X-Content-Type-Options) to protect against common web vulnerabilities.

3. Improved Usability

Customization: Allow users to customize the system's appearance (e.g., themes, font sizes, layout) to suit their preferences.
Tooltips and Help Guides: Add tooltips and help guides to improve user onboarding and make the system more intuitive.
Error Handling: Provide clear and user-friendly error messages to help users resolve issues quickly.
Thumbnail Size: Increase the size of thumbnails in YouTube search results for better visibility.
Password Validation: Fix password validation to handle special characters correctly during shortcut deletion.

4. Advanced AI Features

Specialized AI Models: Integrate additional AI models for specialized tasks, such as sentiment analysis, language translation, and summarization.
Fine-Tuning: Allow users to fine-tune AI responses based on their preferences (e.g., adjust response length, tone, or complexity).
Multimodal Inputs: Support multimodal inputs (e.g., text, voice, images) for more versatile interactions with the AI.
Contextual Understanding: Enhance the AI's ability to understand and maintain context across multiple queries.

5. Offline Functionality

Cached Data: Add offline capabilities for certain features, such as accessing cached Wikipedia summaries or downloaded videos.
Local Storage: Use local storage to save user preferences, shortcuts, and frequently accessed data for offline use.
Background Sync: Implement background synchronization to update cached data when the system is online.

6. Mobile Support

Responsive Design: Develop a mobile-friendly version of the system with a responsive design that adapts to different screen sizes.
Mobile App: Create native mobile apps (iOS and Android) to provide a seamless experience for on-the-go usage.
Offline Mode: Enable offline functionality for mobile users, allowing them to access cached data without an internet connection.

7. Integration with Additional APIs

Google Scholar: Integrate with Google Scholar to provide academic research papers and citations.
Reddit: Add support for searching and retrieving discussions from Reddit.

Social Media: Integrate with social media platforms (e.g., Twitter, Instagram) to fetch real-time updates and trends.
E-commerce: Add support for searching product information from e-commerce platforms (e.g., Amazon, eBay).

8. User Analytics

Usage Tracking: Add analytics to track user behavior, such as frequently used features, search patterns, and session duration.
Popular Features: Identify popular features and prioritize their improvement based on user feedback.
Pain Points: Detect and address pain points (e.g., slow response times, confusing interfaces) to enhance user satisfaction.
Personalization: Use analytics to provide personalized recommendations and improve the user experience.

9. Additional Features

Voice Search: Add voice search functionality for hands-free query processing.
Multi-Language Support: Support multiple languages for Wikipedia summaries, AI chat, and other features.
Collaboration Tools: Add collaboration features, such as shared shortcuts or group chats, for team-based usage.
Gamification: Introduce gamification elements (e.g., badges, leaderboards) to encourage user engagement.

Chapter-7 Deployment

7.1 Local Deployment Guide

Deploying the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** locally is ideal for **development**, **testing**, and small-scale usage. Below is a step-by-step guide to set up the system on your local machine:

Prerequisites

Before starting, ensure you have the following installed on your machine:

a) **Python 3.8 or higher:** Download and install from python.org.
b) pip (Python package manager): Usually installed with Python.
c) **Git:** Download and install from git-scm.com.

Steps

Clone the Repository:
- Open a terminal or command prompt.
- Run the following command to clone the repository:

```
git clone https://github.com/your-repo/mmqpkr.git
```

- Navigate to the project directory:

```
cd mmqpkr
```

Create a Virtual Environment:
- Create a virtual environment to isolate dependencies:

```
python -m venv venv
```

- Activate the virtual environment:

On macOS/Linux:

```
source venv/bin/activate
```

On Windows:

```
venv\Scripts\activate
```

Install Dependencies:
- Install the required Python packages using pip:

```
pip install -r requirements.txt
```

Set Up Environment Variables:
- Create a .env file in the root directory of the project.
- Add the required API keys and configuration variables to the .env file:

```
YT=your_youtube_api_key
HF_API=your_huggingface_api_key
NEWS_API=your_news_api_key
IMG_API=your_image_api_key
LOGO_API=your_logo_api_key
```

- Replace your_youtube_api_key, your_huggingface_api_key, etc., with your actual API keys.

Run the Application:

- Start the application using Streamlit:

```
streamlit run app.py
```

Access the Application:

- Open your web browser and navigate to:

```
http://localhost:8501
```

- The application should now be running locally, and you can interact with it through your browser.

Troubleshooting

✓ **Missing Dependencies:** If you encounter errors related to missing packages, ensure you have activated the virtual environment and run pip install -r requirements.txt again.
✓ **API Key Issues:** If the application fails to fetch data, double-check the API keys in the .env file and ensure they are valid.
✓ **Port Conflicts:** If port 8501 is already in use, you can specify a different port:

```
streamlit run app.py --server.port 8502
```

7.2 Cloud Deployment

Deploying the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** on a cloud platform ensures **scalability**, **reliability**, and **accessibility**. Below are detailed deployment options for popular cloud platforms:

1. Streamlit Sharing

Streamlit Sharing is a **simple** and **efficient** platform for **deploying** Streamlit applications.

Steps:

i. **Push Your Code to GitHub:**
Commit and push your code to a GitHub repository.

```
git add
git commit -m "Initial commit"
git push origin main
```

ii. Sign Up for Streamlit Sharing:
Visit Streamlit Sharing and sign up using your GitHub account.

iii. **Connect Your GitHub Repository:**
Link your GitHub repository to Streamlit Sharing.

iv. **Deploy the App:**
Specify the entry point (e.g., app.py) and deploy the application.

Advantages:
- ✓ Easy to Set Up: No complex configuration required.
- ✓ Free Tier: Ideal for small-scale deployments and prototyping.
- ✓ Managed Service: Streamlit handles server management and scaling.

2. AWS (Amazon Web Services)

AWS provides a highly **scalable** and **customizable** platform for deploying applications.

Steps:

i. **Create an EC2 Instance or Use Elastic Beanstalk:**
For EC2:
- ✓ Launch an EC2 instance with a suitable AMI (e.g., Amazon Linux 2).
- ✓ SSH into the instance and install dependencies.

For Elastic Beanstalk:
- ✓ Create an Elastic Beanstalk environment and upload your application.

ii. **Install Dependencies:**
Install Python, pip, and other dependencies on the EC2 instance or Elastic Beanstalk environment.

iii. **Configure the Environment:**
Set up environment variables (e.g., API keys) using AWS Systems Manager Parameter Store or Elastic Beanstalk configuration.

iv. **Use a Reverse Proxy:**
Install and configure Nginx or Apache as a reverse proxy to serve the application.

v. **Deploy Using CI/CD:**
Set up a CI/CD pipeline using GitHub Actions or AWS CodePipeline to automate deployments.

Advantages:
- ✓ **Highly Scalable:** Easily handle large-scale deployments.
- ✓ **Customizable:** Full control over the infrastructure and configuration.
- ✓ **Reliable:** AWS provides high availability and fault tolerance.

3. Heroku

Heroku is a **user-friendly** platform for **deploying web applications**.

Steps:

i. Create a Procfile:
Add a Procfile in the root directory with the following content:

```
web: streamlit run app.py --server.port $PORT
```

ii. **Push Your Code to GitHub:**
Commit and push your code to a GitHub repository.

iii. **Sign Up for Heroku:**
Visit Heroku and sign up for an account.

iv. Connect Your GitHub Repository:
Link your GitHub repository to Heroku.

v. **Deploy the App:**
Deploy the application from the Heroku dashboard.

Advantages:
✓ **Easy to Use:** Simple and intuitive interface.
✓ **Free Tier:** Suitable for small-scale deployments and testing.
✓ **Automatic Scaling:** Heroku automatically scales the application based on traffic.

4. Hugging Face Spaces

Hugging Face Spaces provides a free platform for deploying **Streamlit**, **Gradio**, or static web apps with seamless AI model integration.

Steps:

i. Push Code to GitHub:

```
git add .
git commit -m "Initial commit"
git push origin main
```

ii. Sign Up for Hugging Face:
Visit Hugging Face Spaces and sign up.

iii. Create a New Space:
Click "Create new Space" → Select Streamlit as SDK.
Connect your GitHub repository or upload files manually.

iv. Configure app.py:
Ensure your entry script is named app.py.
Add requirements.txt with dependencies.

v. Deploy:
Hugging Face automatically builds and deploys the app.

Advantages:
✓ Free hosting with GPU support (if needed)
✓ Direct AI model integration (Hugging Face models)
✓ Public/private deployment options

5. Google Colab

Google Colab allows cloud-based execution of Python notebooks with free GPU/TPU support, ideal for demo deployments.

Steps:

i. Upload Code to Google Drive:
Upload app.py and dependencies to Google Drive.

ii. Create a Colab Notebook:
Open Google Colab → New Notebook.

iii. Install Streamlit & Dependencies:

```
!pip install streamlit pyngrok
!ngrok authtoken YOUR_NGROK_TOKEN  # For exposing the app
```

iv. Run Streamlit via Ngrok:

```
!streamlit run app.py &>/dev/null&
!npx localtunnel --port 8501  # Alternative to ngrok
```

v. Access Public URL:
Colab provides a temporary public URL (e.g., https://xxxx.loca.lt).

Advantages:
✓ Free GPU/TPU access for compute-heavy tasks
✓ No setup for Python environments
✓ Temporary demo sharing

7.3 Environment Configuration

Proper environment configuration is crucial for ensuring the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** runs smoothly across different deployment environments. Below are the key considerations and steps for configuring the environment:

1. Environment Variables
Environment variables are used to store sensitive information (e.g., API keys) and configuration settings that may vary between environments (e.g., local, production).

Steps:

i. **Create a .env File:**
In the root directory of your project, create a .env file.
Add the required environment variables to the .env file:

```
YT=your_youtube_api_key
HF_API=your_huggingface_api_key
NEWS_API=your_news_api_key
IMG_API=your_image_api_key
LOGO_API=your_logo_api_key
```

Replace the placeholders with your actual API keys.

Load Environment Variables:
Use the python-dotenv package to load environment variables from the .env file:

```
pip install python-dotenv
```

Add the following code to your application to load the variables:

```
from dotenv import load_dotenv
import os

load_dotenv()  # Load environment variables from .env file
YOUTUBE_API_KEY = os.getenv("YT")
HUGGINGFACE_API_KEY = os.getenv("HF_API")
NEWS_API_KEY = os.getenv("NEWS_API")
IMAGE_API_KEY = os.getenv("IMG_API")
LOGO_API_KEY = os.getenv("LOGO_API")
```

Cloud Platform Configuration:
For production deployments, use the cloud platform's environment variable management tools (e.g., Heroku Config Vars, AWS Systems Manager Parameter Store).

2. Dependencies
Managing dependencies ensures that the application runs consistently across different environments.

Steps:

ii. List Dependencies:
Ensure all dependencies are listed in the requirements.txt file:

```
streamlit==1.13.0
google-api-python-client==2.80.0
wikipedia-api==0.5.8
gtts==2.3.1
requests==2.28.2
python-dotenv==0.21.1
```

iii. Use a Virtual Environment:
Create and activate a virtual environment to isolate dependencies:

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

Install dependencies:

```
pip install -r requirements.txt
```

iv. Freeze Dependencies:
If you add new dependencies, update the requirements.txt file:

```
pip freeze > requirements.txt
```

3. Configuration Files

Configuration files help manage settings like API endpoints, logging, and feature toggles.

Steps:

a) **Create a Configuration File:**
Create a config.py file in the root directory:

```
import os

class Config:
    # API Endpoints
    YOUTUBE_API_URL = "https://www.googleapis.com/youtube/v3"
    HUGGINGFACE_API_URL = "https://api.huggingface.co"
    NEWS_API_URL = "https://newsapi.org/v2"
    # Logging Configuration
    LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")
    LOG_FILE = "app.log"
    # Feature Toggles
    ENABLE_AI_CHAT = True
ENABLE_NEWS_SEARCH = True
```

Use Configuration Settings:
Import and use the configuration settings in your application:

```
from config import Config

print(f"Log Level: {Config.LOG_LEVEL}")
print(f"Enable AI Chat: {Config.ENABLE_AI_CHAT}")
```

b) **Environment-Specific Configurations:**
Use environment variables to override configuration settings for different environments:

```
class ProductionConfig(Config):
    LOG_LEVEL = os.getenv("LOG_LEVEL", "WARNING")
    ENABLE_NEWS_SEARCH = False

class DevelopmentConfig(Config):
    LOG_LEVEL = os.getenv("LOG_LEVEL", "DEBUG")
```

7.4 API Key Management

API keys are critical for accessing external services (e.g., YouTube, Hugging Face, News API) and must be managed securely to prevent unauthorized access and abuse. Below are best practices and steps for managing API keys effectively:

1. Environment Variables
Storing API keys in environment variables ensures they are not hardcoded in the source code, reducing the risk of exposure.

Steps:

a) Create a .env File:
In the root directory of your project, create a .env file.

Add the API keys to the .env file:

```
YT=your_youtube_api_key
HF_API=your_huggingface_api_key
NEWS_API=your_news_api_key
IMG_API=your_image_api_key
LOGO_API=your_logo_api_key
```

Replace the placeholders with your actual API keys.

b) Load Environment Variables:

Use the python-dotenv package to load environment variables from the .env file:

```
pip install python-dotenv
```

Add the following code to your application to load the variables:

```
from dotenv import load_dotenv
import os

load_dotenv()  # Load environment variables from .env file
YOUTUBE_API_KEY = os.getenv("YT")
HUGGINGFACE_API_KEY = os.getenv("HF_API")
NEWS_API_KEY = os.getenv("NEWS_API")
IMAGE_API_KEY = os.getenv("IMG_API")
LOGO_API_KEY = os.getenv("LOGO_API")
```

c) Exclude .env from Version Control:

Add .env to your .gitignore file to prevent it from being committed to version control:

```
# .gitignore
.env
```

2. Secrets Management

For production deployments, use cloud platform secrets management tools to securely store and manage API keys.

Steps:

i. AWS Secrets Manager:

Store API keys in AWS Secrets Manager.
Access the keys in your application using the AWS SDK:

```
import boto3
import os

client = boto3.client('secretsmanager')
response = client.get_secret_value(SecretId='your_secret_id')
secrets = json.loads(response['SecretString'])
YOUTUBE_API_KEY = secrets['YT']
HUGGINGFACE_API_KEY = secrets['HF_API']
```

ii. Heroku Config Vars:

Store API keys in Heroku Config Vars:

```
heroku config:set YT=your_youtube_api_key
heroku config:set HF_API=your_huggingface_api_key
```

Access the keys in your application using os.getenv():

```
YOUTUBE_API_KEY = os.getenv("YT")
HUGGINGFACE_API_KEY = os.getenv("HF_API")
```

iii. Google Cloud Secret Manager:
Store API keys in Google Cloud Secret Manager.
Access the keys in your application using the Google Cloud SDK:

```
from google.cloud import secretmanager

client = secretmanager.SecretManagerServiceClient()
name = f"projects/your_project_id/secrets/your_secret_id/versions/latest"
response = client.access_secret_version(request={"name": name})
YOUTUBE_API_KEY = response.payload.data.decode("UTF-8")
```

3. Access Control
Restrict access to API keys to authorized users and services to minimize security risks.

Steps:

a) **Role-Based Access Control (RBAC):**
Use RBAC to restrict access to API keys based on user roles (e.g., admin, developer).
Example: Only allow admins to view or modify API keys in the cloud platform.

b) **Service Accounts:**
Use service accounts with limited permissions to access API keys in production environments.
Example: Grant a service account read-only access to secrets in AWS Secrets Manager.

c) **Audit Logs:**
Enable audit logs to track access to API keys and detect unauthorized access attempts.

4. Key Rotation
Rotate API keys periodically to minimize the risk of compromise.

Steps:

i. **Set Expiry Dates:**
Set expiry dates for API keys and rotate them before they expire.
Example: Rotate API keys every 90 days.

ii. **Automate Key Rotation:**
Use automation tools to rotate API keys and update them in the application.
Example: Use AWS Lambda to rotate keys and update them in AWS Secrets Manager.

iii. Notify Stakeholders:
Notify stakeholders (e.g., developers, admins) before rotating API keys to avoid disruptions.

Chapter-8 User Manual

8.1 How to Use the System

The **MMQPKRS** is designed to provide a seamless and intuitive user experience. Follow these steps to get started:

Access the System:
Open your web browser and navigate to the system's URL (e.g., http://localhost:8501 for local deployment or the cloud deployment URL).
The system's homepage will display the main interface with options for different features.

Navigate the Interface:
Use the sidebar to select the desired feature (e.g., Wikipedia Summary, YouTube Search, AI Chat).
Enter your query or input in the provided text box and click the appropriate button to generate results.

Explore Features:
Follow the step-by-step guide for each feature (see Section 8.2) to learn how to use them effectively.

8.2 Step-by-Step Guide for Each Feature

Below are detailed instructions for using each feature of the system:

1. Wikipedia Summary

- **Select Wikipedia Summary:**

From the sidebar, select Wikipedia Summary.

- **Enter a Topic:**

Enter a topic (e.g., "Artificial Intelligence") in the text box.

- **Choose Summary Level:**

Select the summary level (Brief, Detailed, or Bullet Points) from the dropdown menu.

- **Set Character Limit:**

Use the slider to set the character limit for the summary.

- **Generate Summary:**

Click the Generate Summary button.

- **View and Listen:**

The summary will be displayed. Click the Play Text-to-Speech button to listen to the summary.

2. YouTube Search

- **Select YouTube Search:**

From the sidebar, select YouTube Search.

- **Enter a Query:**

Enter a search query (e.g., "machine learning tutorials") in the text box.

- **Search Videos:**

Click the Search button.

- **View Results:**

The system will display video results with titles, thumbnails, and links.

- **Play Videos:**

Click on a video thumbnail to play it directly within the system or open it on YouTube.

3. AI Chat

- **Select AI Chat:**

From the sidebar, select AI Chat.

- **Enter a Question:**

Enter a question or prompt (e.g., "Explain quantum computing in simple terms") in the text box.

- **Generate Response:**

Click the Generate Response button.

- **View Response:**

The AI's response will be displayed in the chat window.

- **Rename Chat:**

Use the sidebar option to rename the chat for better organization.

4. News Search

- **Select News Search:**

From the sidebar, select News Search.

- **Enter a Query:**

Enter a news topic (e.g., "climate change") in the text box.

- **Set Date Range:**

Use the date picker to set the start and end dates for the news search.

- **Search News:**

Click the Search button.

- **View Results:**

The system will display news articles with titles, descriptions, images, and links.

- **Download Images:**

Click the Download All Images as ZIP button to download images from the search results.

5. Shortcuts Management

- **Select Shortcuts Management:**

From the sidebar, select Shortcuts Management.

- **Add a Shortcut:**

Enter a name, link, and upload an icon for the shortcut.
Click the Add Shortcut button.

- **View Shortcuts:**

The system will display all added shortcuts with icons and links.

- **Delete a Shortcut:**

Enter the password and click the Delete button next to the shortcut you want to remove.

6. Google Search

- **Select Google Search:**

From the sidebar, select Google Search.

- **Enter a Query:**

Enter your search term (e.g., "latest AI advancements") in the text box.

- **Filter Results (Optional):**

Use the advanced filters to limit by date or domain.

- **Search:**

Click the Search button.

- **View Results:**

The system displays search results with titles, snippets and URLs.

- **Open Link:**

Click any result to open in a new browser tab.

7. YouTube to MP3 (YTMP3)

- **Select YTMP3:**

From the sidebar, select YouTube to MP3.

- **Paste URL:**

Enter a YouTube video link.

- **Select Quality:**

Choose between 128kbps or 320kbps.

- **Convert:**

Click the Convert button.

- **Download:**

When processing completes, click Download MP3.

8. Logo Generation

- **Select Logo Generation:**

From the sidebar, select Logo Generation.

- **Enter Description:**

Describe your desired logo (e.g., "modern tech logo with blue gradient").

- **Choose Style:**

Select from style options (Minimalist, Vintage, Modern etc.).

- **Set Color Scheme:**

Pick primary and secondary colors.

- **Generate Logo:**

Click the Generate button.

- **Download:**

Preview and download the logo as PNG/SVG.

9. LinkedIn Details

- **Select LinkedIn Details:**

From the sidebar, select LinkedIn Details.

- **Enter Profile URL:**

Paste a LinkedIn profile link.

- **Extract Data:**

Click the Extract button.

- **View Profile:**

See complete profile data including experience and skills.

- **Export:**

Download as JSON/CSV for analysis.

10. Instagram Details

- **Select Instagram Details:**

From the sidebar, select Instagram Details.

- **Enter Username/URL:**

Provide an Instagram handle or post URL.

- **Fetch Data:**

Click the Fetch button.

- **View Content:**

See posts with engagement metrics.

- **Download Media:**

Select images/videos to download.

11. IMDb Movie Search

- **Select IMDb Search:**

From the sidebar, select IMDb Search.

- **Enter Title:**

Type a movie/TV show name.

- **Search:**

Click the Search button.

- **View Details:**

See ratings, cast, plot summary and streaming info.

- **Filter:**

Sort by year, rating or genre.

12. Real-Time Image Search

- **Select Image Search:**

From the sidebar, select Image Search.

- **Enter Keywords:**

Describe what you're looking for.

- **Apply Filters:**

Set color, size and license filters.

- **Search:**

Click the Search button.

- **View Results:**

Browse images from multiple sources.

- **Download:**

Save selected images.

13. NASA Space Data Explorer

- **Select NASA Explorer:**

From the sidebar, select NASA Explorer.

- **Choose Dataset:**

Select from Astronomy, Mars Rovers etc.

- **Set Parameters:**

Adjust date range and filters.

- **Fetch Data:**

Click the Get Data button.

- **View:**

Explore high-resolution space images.

- **Download:**

Save images with metadata.

14. AI Image Generator

- **Select AI Image Generator:**

From the sidebar, select AI Image Generator.

- **Enter Prompt:**

Describe the image you want (e.g., "cyberpunk city at night").

- **Choose Style:**

Select art style (Realistic, Painting etc.).

- **Set Dimensions:**

Pick image size/ratio.

- **Generate:**

Click the Create button.

- **Refine:**

Make variations or upscale the image.

- **Download:**

Save the final image.

8.3 Troubleshooting Common Issues

Below are solutions to common issues users may encounter:

i. Slow Response Times

Cause: High user load or slow external APIs.
Solution: Refresh the page or try again later. For persistent issues, contact support.

ii. Failed API Calls

Cause: External API rate limits or failures.
Solution: Check your internet connection and try again. If the issue persists, contact support.

iii. Error Messages

Cause: Invalid input or system errors.
Solution: Verify your input and try again. If the error persists, contact support.

iv. Unable to Delete Shortcuts

Cause: Incorrect password or special characters in the password.
Solution: Ensure the password is correct and does not contain special characters.

Chapter-9 Conclusion

9.1 Summary of the Project

The **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** is a comprehensive platform designed to provide users with seamless access to information from multiple sources. By integrating various APIs and advanced technologies, the system delivers a multimodal experience, enabling users to retrieve, process, and interact with knowledge in diverse formats, including text, audio, video, and images. Below is a detailed summary of the project:

Key Features

i. **Wikipedia Summaries:**
- ✓ Provides concise summaries of topics from Wikipedia.
- ✓ Includes text-to-speech functionality for an enhanced user experience.

ii. **YouTube Search:**
- ✓ Allows users to search for videos on YouTube.
- ✓ Displays video results with titles, thumbnails, and links.
- ✓ Enables users to play videos directly within the system or open them on YouTube.

iii. **AI Chat:**
- ✓ Offers an interactive AI-powered chat feature.
- ✓ Generates responses to user questions using advanced AI models.
- ✓ Supports chat history and renaming for better organization.

iv. **News Search:**
- ✓ Retrieves news articles based on user queries.
- ✓ Displays article titles, descriptions, images, and links.
- ✓ Allows users to download images as a ZIP file.

v. **Shortcuts Management:**
- ✓ Enables users to add, view, and delete shortcuts for quick access to resources.
- ✓ Each shortcut includes a name, link, and icon.
- ✓ Requires a password for deletion to ensure security.

vi. Google Search:
- ✓ Performs web searches using Google's Custom Search API.
- ✓ Displays search results with titles, snippets, and URLs.
- ✓ Supports advanced filters (date range, domain-specific searches).
- ✓ Provides quick access to relevant web content.

vii. **Logo Generation:**
- ✓ Generates custom logos using AI-powered image generation.

- ✓ Supports text-to-logo creation with style customization (minimalist, vintage, modern).
- ✓ Offers downloadable high-resolution PNG/SVG files.

viii. YouTube to MP3 (YTMP3):
- ✓ Converts YouTube videos to MP3 audio files.
- ✓ Supports quality selection (128kbps/320kbps).
- ✓ Allows metadata editing (title, artist, album).

ix. LinkedIn Details Extractor:
- ✓ Extracts professional profile data from LinkedIn.
- ✓ Retrieves work history, education, skills, and endorsements.
- ✓ Exports data in JSON/CSV format for analysis.

x. Instagram Details Fetcher:
- ✓ Fetches posts, stories, and profile information from Instagram.
- ✓ Displays engagement metrics (likes, comments).
- ✓ Enables bulk downloading of media in original quality.

xi. IMDb Movie Search:
- ✓ Provides detailed information about movies/TV shows.
- ✓ Displays ratings (IMDb, Rotten Tomatoes), cast, and plot summaries.
- ✓ Shows streaming availability across platforms.

xii. Real-Time Image Search:
- ✓ Searches multiple image databases in real time.
- ✓ Advanced filters (color, size, license type).
- ✓ Reverse image search capability.

xiii. NASA Space Data Explorer:
- ✓ Accesses NASA's open APIs for space-related data.
- ✓ Displays astronomical images, Mars rover photos, and asteroid data.
- ✓ Includes interactive 3D visualizations of celestial bodies.

xiv. AI Image Generator:
- ✓ Creates original images from text prompts using AI.
- ✓ Supports style transfer and resolution enhancement (up to 4K).
- ✓ Generates multiple variations from a single prompt.

Technologies Used

Programming Language: Python
Web Framework: Streamlit (for building the user interface)
APIs Integrated:
- ✓ YouTube Data API
- ✓ Wikipedia API
- ✓ Hugging Face API (for AI chat)
- ✓ News API
- ✓ Image and logo generation APIs

Libraries:
- ✓ google-api-python-client (for YouTube integration)
- ✓ wikipedia-api (for Wikipedia summaries)
- ✓ gtts (for text-to-speech)
- ✓ requests (for API calls)
- ✓ python-dotenv (for environment variable management)

System Architecture

a) **Frontend:** Streamlit-based user interface for intuitive navigation and interaction.
b) **Backend:** Python scripts for processing user queries, integrating APIs, and generating responses.
c) **Database:** Local storage for shortcuts and cached data (e.g., Wikipedia summaries, YouTube search results).
d) **Security:** Environment variables for API key management and password protection for shortcut deletion.

User Experience

- ✓ The system is designed to be user-friendly, with a clean and intuitive interface.
- ✓ Users can easily switch between features using the sidebar.
- ✓ Real-time feedback and error handling ensure a smooth experience.

Key Benefits

a) **Multimodal Access:** Users can retrieve and interact with information in multiple formats (text, audio, video, images).
b) **Efficiency:** Provides quick and concise access to knowledge from diverse sources.
c) **Customization:** Allows users to manage shortcuts and tailor their experience.
d) **Scalability:** Designed to handle high user loads with horizontal scaling and caching.

9.2 Achievements

The **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** project has achieved several significant milestones, demonstrating its effectiveness and potential. Below is a detailed overview of the key achievements:

1. Multimodal Integration

Achievement: Successfully integrated multiple data sources into a single platform.
Details:
- ◆ Integrated APIs from Wikipedia, YouTube, News API, and Hugging Face.
- ◆ Provided users with a unified interface to access diverse types of information (text, audio, video, images).

Impact:
- ◆ Users can retrieve and interact with knowledge from various sources seamlessly.
- ◆ Enhanced the system's versatility and usefulness.

2. User-Friendly Design

Achievement: Developed an intuitive and responsive interface.
Details:
- Used Streamlit to create a clean and easy-to-navigate interface.
- Designed features with simplicity and accessibility in mind, ensuring ease of use for both technical and non-technical users.

Impact:
- Improved user satisfaction and engagement.
- Reduced the learning curve for new users.

3. Scalable Architecture

Achievement: Designed the system to handle high user loads.
Details:
- Implemented horizontal scaling to distribute traffic across multiple instances.
- Used caching mechanisms to reduce load on external APIs and improve response times.
- Offloaded resource-intensive tasks (e.g., AI chat, video processing) to background workers using asynchronous processing.

Impact:
- Ensured the system remains responsive and reliable under high traffic.
- Prepared the system for future growth and increased user demand.

4. Security and Reliability

Achievement: Ensured data security and system reliability.
Details:
- Used environment variables and secrets management tools to securely store API keys.
- Conducted rigorous testing to identify and resolve vulnerabilities (e.g., session fixation, SQL injection).
- Implemented encryption and access control mechanisms to protect user data.

Impact:
- Enhanced user trust and confidence in the system.
- Minimized the risk of data breaches and unauthorized access.

5. Real-World Applicability
Achievement: Demonstrated the system's utility in real-world scenarios.
Details:
- Tested the system with students, educators, and professionals to validate its effectiveness.
- Received positive feedback on features like Wikipedia summaries, YouTube search, and AI chat.

Impact:
- Proved the system's value as a tool for efficient knowledge retrieval and processing.
- Highlighted its potential for widespread adoption in educational and professional settings

Summary of Achievements

Achievement	Details	Impact
Multimodal Integration	Integrated Wikipedia, YouTube, news APIs, and AI models into one platform.	Unified interface for accessing diverse information.
User-Friendly Design	Developed an intuitive and responsive interface using Streamlit.	Improved user satisfaction and accessibility.
Scalable Architecture	Implemented horizontal scaling, caching, and asynchronous processing.	Ensured reliability and performance under high user loads.
Security and Reliability	Secured API keys, conducted rigorous testing, and implemented encryption.	Enhanced user trust and minimized security risks.
Real-World Applicability	Demonstrated utility for students, educators, and professionals.	Validated the system's effectiveness and potential for widespread adoption.

9.3 Lessons Learned

Throughout the development and deployment of the **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)**, several valuable lessons were learned. These insights will guide future projects and improvements to the system. Below is a detailed discussion of the key lessons:

1. Importance of Modular Design

Lesson: Breaking the system into modular components simplifies development and testing.
Details:
The system was divided into modules (e.g., **Wikipedia module**, **YouTube module**, **AI Chat module**).
Each module was developed and tested independently, ensuring functionality and reducing complexity.
Impact:
Improved maintainability and scalability.
Facilitated collaboration among team members working on different modules.

2. Challenges of API Dependencies

Lesson: Reliance on external APIs introduces challenges such as rate limits, latency, and failures.
Details:
External APIs (e.g., YouTube, Wikipedia, News API) imposed rate limits, causing delays or failures during high usage.
API outages or changes in API specifications disrupted system functionality.
Mitigation:

Implemented **caching mechanisms** to reduce API calls and improve response times.
Added **fallback mechanisms** (e.g., cached responses) to handle API failures gracefully.
Impact:
Enhanced system reliability and user experience.
Reduced dependency on external APIs for critical functionalities.

3. User Feedback is Crucial

Lesson: User acceptance testing (UAT) provides critical insights into usability issues and feature improvements.
Details:
Conducted UAT with real users (students, educators, professionals) to gather feedback.
Identified usability issues (e.g., small thumbnails, confusing error messages) and areas for improvement.
Impact:
Improved user satisfaction and engagement.
Ensured the system met the needs and expectations of its target audience.

4. Scalability is Key

Lesson: Designing for scalability from the outset ensures the system can handle increased user loads without performance degradation.
Details:
Implemented **horizontal scaling** to distribute traffic across multiple instances.
Used **caching** and **asynchronous processing** to optimize performance under high loads.
Impact:
Ensured the system remained responsive and reliable as user demand grew.
Prepared the system for future expansion and increased usage.

5. Security Cannot Be Overlooked

Lesson: Addressing security vulnerabilities early in the development process prevents potential breaches.
Details:
Identified and resolved vulnerabilities such as session fixation and SQL injection during testing.
Implemented **encryption** and secure **API key management** to protect sensitive data.
Impact:
Enhanced user trust and confidence in the system.
Minimized the risk of data breaches and unauthorized access.

Summary of Lessons Learned

Lesson	Details	Impact
Importance of Modular Design	Breaking the system into modular components simplified development.	Improved maintainability and scalability.
Challenges of API Dependencies	Reliance on external APIs introduced rate limits, latency, and failures.	Implemented caching and fallback mechanisms to mitigate issues.
User Feedback is Crucial	UAT provided insights into usability issues and feature improvements.	Improved user satisfaction and engagement.
Scalability is Key	Designed for scalability to handle increased user loads.	Ensured reliability and performance under high traffic.
Security Cannot Be Overlooked	Addressed vulnerabilities early to prevent breaches.	Enhanced user trust and minimized security risks.

9.4 Future Scope

The **Multimodal Query Processing and Knowledge Retrieval System (MMQPKRS)** has significant potential for **future enhancements** and **expansion**. Below are some proposed improvements to further enhance its functionality, usability, and reach:

1. Advanced AI Capabilities

Objective: Enhance the AI chat feature with specialized capabilities.
Proposed Improvements:
- ✓ Integrate additional AI models for tasks like sentiment analysis, language translation, and summarization.
- ✓ Allow users to fine-tune AI responses based on their preferences (e.g., adjust response length, tone, or complexity).

Impact:
- ✓ Provides more accurate and personalized responses.
- ✓ Expands the system's utility for diverse use cases.

2. Offline Functionality

Objective: Enable users to access certain features without an internet connection.
Proposed Improvements:
- ✓ Add offline capabilities for features like cached Wikipedia summaries and downloaded videos.
- ✓ Use local storage to save user preferences and frequently accessed data.

Impact:
- ✓ Improves accessibility for users in low-connectivity areas.
- ✓ Enhances user convenience and flexibility.

3. Mobile Support

Objective: Make the system accessible on mobile devices.
Proposed Improvements:
- ✓ Develop a mobile-friendly version of the system with a responsive design.
- ✓ Create native mobile apps for iOS and Android.

Impact:
- ✓ Expands the system's reach to mobile users.
- ✓ Provides a seamless experience across devices.

4. Integration with More APIs

Objective: Expand the system's knowledge base and functionality.
Proposed Improvements:
- ✓ Integrate with additional APIs like Google Scholar, Reddit, and social media platforms.
- ✓ Add support for retrieving academic papers, discussions, and real-time updates.

Impact:
- ✓ Provides users with access to a wider range of information sources.
- ✓ Enhances the system's versatility and usefulness.

5. Enhanced Customization

Objective: Allow users to personalize their experience.
Proposed Improvements:
- ✓ Add options to customize the system's appearance (e.g., themes, font sizes).
- ✓ Allow users to adjust default settings (e.g., default summary level, search preferences).

Impact:
- ✓ Improves user satisfaction and engagement.
- ✓ Makes the system more adaptable to individual preferences.

6. User Analytics

Objective: Gain insights into user behavior and system performance.
Proposed Improvements:
- ✓ Implement analytics to track user behavior (e.g., frequently used features, search patterns).
- ✓ Identify popular features and pain points to guide future improvements.

Impact:
- ✓ Provides data-driven insights for enhancing the system.
- ✓ Helps prioritize development efforts based on user needs.

7. Collaboration Features

Objective: Enable team-based usage and collaboration.
Proposed Improvements:
- ✓ Add shared shortcuts for teams to access common resources.
- ✓ Introduce group chats for collaborative discussions.

Impact:
- ✓ Enhances the system's utility for educational and professional teams.
- ✓ Facilitates knowledge sharing and collaboration.

9. Improved Security

Objective: Enhance the system's security and user trust.
Proposed Improvements:
- ✓ Implement two-factor authentication (2FA) for user accounts.
- ✓ Use advanced encryption methods to protect sensitive data.

Impact:
- ✓ Reduces the risk of unauthorized access and data breaches.
- ✓ Builds user confidence in the system's security.

10. Global Reach

Objective: Make the system accessible to a global audience.
Proposed Improvements:
- ✓ Add support for multiple languages (e.g., Spanish, Chinese, Hindi).
- ✓ Localize the interface and content for different regions.

Impact:
- ✓ Expands the system's user base to non-English speakers.
- ✓ Enhances accessibility and inclusivity.

Enhancement	Objective	Impact
Advanced AI Capabilities	Integrate specialized AI models and allow fine-tuning.	Provides accurate and personalized responses.
Offline Functionality	Enable access to cached data and features without an internet connection.	Improves accessibility and convenience.
Mobile Support	Develop mobile-friendly versions and native apps.	Expands reach to mobile users.
Integration with More APIs	Add support for Google Scholar, Reddit, and social media platforms.	Expands knowledge base and functionality.
Enhanced Customization	Allow users to customize appearance and behavior.	Improves user satisfaction and adaptability.
User Analytics	Track user behavior and identify popular features or pain points.	Provides data-driven insights for improvement.
Collaboration Features	Add shared shortcuts and group chats for team-based usage.	Enhances utility for educational and professional teams.
Gamification	Introduce badges, leaderboards, and achievements.	Increases user engagement and motivation.
Improved Security	Implement 2FA and advanced encryption methods.	Enhances security and user trust.
Global Reach	Add support for multiple languages and localization.	Expands accessibility to a global audience.

Chapter-10 Appendices

10.1 Source Code Structure

The source code for the **Multimodal Query Processing & Knowledge Retrieval System (MMQPKRS)** is organized into modular components for easy maintenance and scalability.

Below is the structure of the project:

```
mmqpkrs/
├── app.py                      # Main application file (Streamlit entry point)
├── requirements.txt            # List of dependencies
├── .env                        # Environment variables (API keys, etc.)
├── assets/                     # Static assets (images, icons, etc.)
├── modules/                    # Core functionality modules
│   ├── wikipedia_search.py     # Wikipedia search and TTS
│   ├── google_search.py        # Google search integration
│   ├── youtube_search.py       # YouTube search and MP3 conversion
│   ├── news_search.py          # News search and image download
│   ├── ai_chat.py              # AI-powered chat using Mistral
│   ├── image_generation.py     # AI image and logo generation
│   ├── social_media.py         # LinkedIn and Instagram data extraction
│   ├── nasa_explorer.py        # NASA space data exploration
│   └── utils.py                # Utility functions (e.g., URL extraction)
│                               #Note: We can also combine these programs
└── README.md                   # Project documentation
```

10.2 API Documentation

This section provides details about the APIs used in the project and how to integrate them.

YouTube Data API v3

Endpoint: https://www.googleapis.com/youtube/v3/search
Parameters:

- ✓ **q:** Search query
- ✓ **maxResults:** Number of results to return
- ✓ **type:** Type of resource (e.g., video)

Example Request:

```
request = youtube.search().list(
    q="Python programming",
    part="id,snippet",
    maxResults=5,
    type="video"
)
response = request.execute()
```

Google Custom Search API

Endpoint: https://www.googleapis.com/customsearch/v1
Parameters:
- ✓ **q:** Search query
- ✓ **cx:** Custom search engine ID
- ✓ **key:** API key

Example Request:

```
response = requests.get(
    "https://www.googleapis.com/customsearch/v1",
    params={"q": "AI", "cx": "YOUR_CSE_ID", "key": "YOUR_API_KEY"}
)
```

NewsAPI

Endpoint: https://newsapi.org/v2/everything
Parameters:
- ✓ **q:** Search query
- ✓ **from:** Start date (YYYY-MM-DD)
- ✓ **to:** End date (YYYY-MM-DD)
- ✓ **apiKey:** API key

Example Request:

```
response = requests.get(
    "https://newsapi.org/v2/everything",
    params={"q": "technology", "from": "2023-10-01", "to": "2023-10-31", "apiKey":
"YOUR_API_KEY"}
)
```

Hugging Face API

Endpoint: https://api-inference.huggingface.co/models/mistral-7b
Headers:
- ✓ **Authorization:** Bearer YOUR_API_KEY

Example Request:

```
headers = {"Authorization": f"Bearer {HF_API_KEY}"}
payload = {"inputs": "What is AI?", "parameters": {"max_length": 200}}
response = requests.post(HF_MISTRAL_URL, json=payload, headers=headers)
```

NASA APIs

APOD Endpoint: https://api.nasa.gov/planetary/apod
Parameters:
- ✓ **api_key:** NASA API key

Example Request:

```
response = requests.get(
    "https://api.nasa.gov/planetary/apod",
    params={"api_key": "YOUR_NASA_API_KEY"}
)
```

10.3 Glossary of Terms

- **API:** Application Programming Interface, a set of protocols for building software.
- **Multimodal System:** A system that processes multiple types of input (e.g., text, voice, images).
- **TTS:** Text-to-Speech, a technology that converts text into spoken audio.
- **DFD:** Data Flow Diagram, a graphical representation of data flow in a system.
- **UAT:** User Acceptance Testing, the final phase of testing before deployment.

10.4 References and Bibliography

✓ **YouTube Data API Documentation:** https://developers.google.com/youtube/v3
✓ **Google Custom Search API Doc:** https://developers.google.com/custom-search/v1
✓ **NewsAPI Documentation:** https://newsapi.org/docs
✓ **Hugging Face API Documentation:** https://huggingface.co/docs
✓ **NASA Open APIs:** https://api.nasa.gov/
✓ **Streamlit Documentation:** https://docs.streamlit.io/

References

1. APIs and Services

[1] YouTube Data API v3
Documentation: YouTube Data API Documentation
Purpose: Video search and metadata retrieval.

[2] Google Custom Search JSON API
Documentation: Google Custom Search API
Purpose: Web search integration.

[3] NewsAPI
Documentation: NewsAPI Documentation
Purpose: News article search and retrieval.

[4] Hugging Face API
Documentation: Hugging Face API Docs
Purpose: AI model integration (e.g., Mistral for chat).

[5] NASA APIs
Documentation: NASA Open APIs
Purpose: Access to space data, images, and astronomy information.

[6] RapidAPI Services
YouTube to MP3 Converter: RapidAPI
Logo Generation: RapidAPI
Image Generation: RapidAPI

[7] Instagram Scraper API
Documentation: Instagram Scraper API
Purpose: Instagram data extraction.

[8] LinkedIn Scraper API
Documentation: LinkedIn Scraper API
Purpose: LinkedIn profile data extraction.

[9] IMDb API
Documentation: IMDb API
Purpose: Movie and entertainment data retrieval.

[10] Real-Time Image Search API
Documentation: Real-Time Image Search API
Purpose: Real-time image search functionality.

Note: Instagram API, LinkedIn API, IMDb API, and Real Time Image Search APIs are also taken from Rapid API. If you want any API search in the Rapid API website. It provides multiple free and paid version APIs for the users.

2. Libraries and Frameworks

[1] Streamlit
Documentation: Streamlit Docs
Purpose: Building the web application interface.

[2] SpeechRecognition Library
Documentation: SpeechRecognition Docs
Purpose: Voice recognition for voice search.

[3] gTTS (Google Text-to-Speech)
Documentation: gTTS Docs
Purpose: Text-to-speech conversion.

[4] Wikipedia-API
Documentation: Wikipedia-API Docs
Purpose: Wikipedia data retrieval.

[5] ReportLab
Documentation: ReportLab Docs
Purpose: PDF generation.

[6] Requests Library
Documentation: Requests Docs
Purpose: HTTP requests for API calls.

[7] Pandas
Documentation: Pandas Docs
Purpose: Data manipulation and analysis.

[8] Altair
Documentation: Altair Docs
Purpose: Data visualization.

[9] Pickle
Documentation: Pickle Docs
Purpose: Serialization and deserialization of Python objects.

[10] Re (Regular Expressions)
Documentation: Re Docs
Purpose: Pattern matching and text processing.

3. Research Papers and Articles

[1] Multimodal Systems in AI
Title: "Multimodal Machine Learning: A Survey and Taxonomy"
Authors: T. Baltrušaitis, C. Ahuja, L. Morency
Link: arXiv:1705.09406

[2] Natural Language Processing (NLP)
Title: "Attention Is All You Need"
Authors: A. Vaswani et al.
Link: arXiv:1706.03762

[3] Speech Recognition
Title: "Deep Speech: Scaling up end-to-end speech recognition"
Authors: A. Hannun et al.
Link: arXiv:1412.5567

[4] Image Generation
Title: "Generative Adversarial Networks (GANs)"
Authors: I. Goodfellow et al.
Link: arXiv:1406.2661

[5] Knowledge Retrieval Systems
Title: "A Survey of Knowledge Retrieval Techniques"
Authors: J. Zhang, Y. Li, Y. Zhang
Link: IEEE Access

4. Other Resources

i. Python Documentation
Link: Python Docs

ii. **Streamlit Community**
Link: Streamlit Forums

iii. **RapidAPI Hub**
Link: RapidAPI

iv. **Hugging Face Models**
Link: Hugging Face Models

v. **NASA Open Data Portal**
Link: NASA Open Data

vi. GitHub Repository
Link: GitHub (for version control and collaboration)

5. Books

a) **"Python for Data Analysis" by Wes McKinney**
Purpose: Data manipulation and analysis using Pandas.

b) **"Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville**
Purpose: Understanding AI and deep learning concepts.

c) **"Streamlit for Data Science" by Tyler Richards**
Purpose: Building web applications with Streamlit.

d) **"API Design Patterns" by JJ Geewax**
Purpose: Best practices for API integration and design.

Online Tutorials and Blogs

- **Real Python**
Link: Real Python
Purpose: Python tutorials and best practices.

- **Towards Data Science (Medium)**
Link: Towards Data Science
Purpose: Articles on AI, NLP, and data science.

- **Streamlit Blog**
Link: Streamlit Blog
Purpose: Updates and tutorials on Streamlit.

- **RapidAPI Blog**
Link: RapidAPI Blog
Purpose: Tutorials and case studies on API usage.

Index

Source Code

```
#Multimodal Query Processing and Knowledge Retrieval System -- by G.Srikrishnadevarayulu

import os
import streamlit as st
from googleapiclient.discovery import build
import speech_recognition as sr
import wikipediaapi
import datetime
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas
import tempfile
from gtts import gTTS
import requests
import json
import pandas as pd
import altair as alt
from datetime import date
import pickle
import re
from io import BytesIO
import zipfile
import io
from PIL import Image  # Required for image processing
# Set up YouTube API
API_KEY_YOUTUBE = os.getenv("YT")  # Replace with your YouTube Data API v3 key
youtube = build('youtube', 'v3', developerKey=API_KEY_YOUTUBE)
# Hugging Face Setup
HF_API_KEY = os.getenv("HF_API")
HF_MISTRAL_URL = os.getenv("MISTRAL")
AI_IMAGE_API_KEY = os.getenv("IMG_API")# Hugging Face or RapidAPI key
LOGO_API_KEY = os.getenv("LOGO_API")  # API key for logo generation
def chat_with_mistral_hf(prompt):
    if not HF_API_KEY:
        return "Error: API key not found."
    headers = {"Authorization": f"Bearer {HF_API_KEY}"}
    payload = {"inputs": prompt, "parameters": {"max_length": 200, "temperature": 0.7}}
    response = requests.post(HF_MISTRAL_URL, json=payload, headers=headers)
    if response.status_code == 200:
        return response.json()[0]["generated_text"]
    else:
        return f"Error: {response.json()}"
# Function to search YouTube videos
def search_youtube(query, max_results=5):
    try:
        request = youtube.search().list(
            q=query,
            part='id,snippet',
            maxResults=max_results,
            type='video'
        )
        response = request.execute()
        videos = []
        for item in response['items']:
            video_id = item['id']['videoId']
            title = item['snippet']['title']
            thumbnail = item['snippet']['thumbnails']['default']['url']
            url = f'https://www.youtube.com/watch?v={video_id}'
            videos.append({'title': title, 'url': url, 'video_id': video_id, 'thumbnail': thumbnail})
        return videos
    except Exception as e:
        st.write(f"Error fetching videos: {e}")
        return []
# # Function for voice recognition
# # This feature is available only , when running locally
# def voice_search():
#     recognizer = sr.Recognizer()
#     with sr.Microphone() as source:       gskdsrikrishna
#         st.write("Listening...")
#         audio = recognizer.listen(source)
```

```
#         try:
#             query = recognizer.recognize_google(audio)
#             st.success(f"You said: {query}")
#             return query
#         except sr.UnknownValueError:
#             st.error("Could not understand audio")
#             return ""
#         except sr.RequestError as e:
#             st.error(f"Could not request results from Google Speech Recognition service; {e}")
#             return ""
# Wikipedia summary function with character limit and summary levels
def get_wikipedia_summary(query, lang_code, char_limit, summary_level):
    user_agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)"
    wiki = wikipediaapi.Wikipedia(language=lang_code, extract_format=wikipediaapi.ExtractFormat.WIKI,
user_agent=user_agent)
    page = wiki.page(query)
    if not page.exists():
        return "Page not found."
    if summary_level == "Brief":
        return page.summary[:char_limit]
    elif summary_level == "Detailed":
        return page.summary  # Full summary  gskdsrikrishna
    elif summary_level == "Bullet Points":
        points = page.summary.split('. ')
        return '\n'.join(f"- {p.strip()}" for p in points if p)[:char_limit]
# Text-to-speech using gTTS
def text_to_speech(text, filename, lang="en"):
    tts = gTTS(text=text, lang=lang)
    tts.save(filename)
    return filename
# Function to perform Google Search
def google_search(api_key, cse_id, query, num_results=10):
    url = "https://www.googleapis.com/customsearch/v1"
    params = {'key': api_key, 'cx': cse_id, 'q': query, 'num': num_results}
    response = requests.get(url, params=params)
    return response.json()
# Display Google Search Results
def display_google_results(results):
    if "items" in results:
        for item in results['items']:
            st.write(f"**{item['title']}**")
            st.write(item['snippet'])
            st.write(f"[Read more]({item['link']})")
            st.write("---")
    else:
        st.error("No results found.")
# # News Search Function
# def search_news(query, from_date=None, to_date=None):
#     api_key = os.getenv("NEWS_API")  # Replace with your News API key
#     url = f"https://newsapi.org/v2/everything?q={query}&apiKey={api_key}"
#     if from_date and to_date:
#         url += f"&from={from_date}&to={to_date}"
#     response = requests.get(url)
#     return response.json()                    gsldsrikrishna
# # Display News Results
# def display_news(articles):
#     if "articles" in articles:
#         for article in articles['articles']:
#             st.write(f"**{article['title']}**")
#             st.write(article['description'])
#             st.write(f"[Read more]({article['url']})")
#             st.write("---")
#     else:
#         st.error("No news articles found.")
# News Search Function
# News Search Function
def search_news(query, from_date=None, to_date=None):
    api_key = os.getenv("NEWS_API")  # Replace with your News API key
    url = f"https://newsapi.org/v2/everything?q={query}&apiKey={api_key}"
    if from_date and to_date:
        url += f"&from={from_date}&to={to_date}"
    response = requests.get(url)
    return response.json()
```

```
# Display News Results & Collect Images
def display_news(articles):
    image_data = []  # Store image bytes for zip download
    if "articles" in articles:
        for index, article in enumerate(articles['articles']):
            st.write(f"**{article['title']}**")
            st.write(article['description'])
            if article.get("urlToImage"):
                # Display Image
                st.image(article["urlToImage"], caption="News Image", use_container_width=True)
                # Download Individual Image
                img_response = requests.get(article["urlToImage"])
                img_bytes = io.BytesIO(img_response.content)
                st.download_button(
                    label="Download Image  ",
                    data=img_bytes,
                    file_name=f"news_image_{index + 1}.jpg",
                    mime="image/jpeg",
                    key=f"download_{index}"  # Unique key for each button
                )
                # Save Image to List for ZIP Download
                image_data.append((f"news_image_{index + 1}.jpg", img_bytes.getvalue()))
            st.write(f"[Read more]({article['url']})")
            st.write("---")
    else:
        st.error("No news articles found.")
        return
    # Generate ZIP File if Images Exist                gskdsrikrishna
    if image_data:
        zip_buffer = io.BytesIO()
        with zipfile.ZipFile(zip_buffer, "w") as zip_file:
            for filename, img in image_data:
                zip_file.writestr(filename, img)
        zip_buffer.seek(0)
        # Download Button for ZIP File
        st.download_button(
            label="Download All Images as ZIP  ",
            data=zip_buffer,
            file_name="news_images.zip",
            mime="application/zip"
        )
def convert_youtube_to_mp3(youtube_url):
    API_KEY = os.getenv("YT_API")  # API key from environment variables
    API_URL = "https://youtube-mp310.p.rapidapi.com/download/mp3"
    HEADERS = {
        "x-rapidapi-key": API_KEY,
        "x-rapidapi-host": "youtube-mp310.p.rapidapi.com"
    }
    try:
        response = requests.get(API_URL, headers=HEADERS, params={"url": youtube_url})
        if response.status_code == 200:
            data = response.json()
            return data.get("downloadUrl", None)
        else:
            return None
    except requests.exceptions.RequestException as e:
        st.error(f"Error: {e}")
        return None
def extract_urls(data):
    """Recursively extracts all URLs from JSON data."""
    urls = []
    if isinstance(data, dict):
        for key, value in data.items():
            urls.extend(extract_urls(value))
    elif isinstance(data, list):
        for item in data:
            urls.extend(extract_urls(item))
    elif isinstance(data, str):
        if re.search(r"https?://", data):  # Extract any URL
            urls.append(data)
    return urls
```

```
def extract_links(obj):
    """Recursively extracts all HTTP links from JSON data"""
    links = []
    if isinstance(obj, dict):
        for key, value in obj.items():
            if isinstance(value, str) and value.startswith("http"):
                links.append(value)
            elif isinstance(value, (dict, list)):
                links.extend(extract_links(value))
    elif isinstance(obj, list):
        for item in obj:
            links.extend(extract_links(item))
    return links
# File to store shortcut data                gskdsrikrishna
DATA_FILE = "shortcuts_data.pkl"
def load_shortcuts():
    if os.path.exists(DATA_FILE):
        with open(DATA_FILE, "rb") as f:
            return pickle.load(f)
    return []
def save_shortcuts(shortcuts):
    with open(DATA_FILE, "wb") as f:
        pickle.dump(shortcuts, f)
if "shortcuts" not in st.session_state:
    st.session_state["shortcuts"] = load_shortcuts()
def main():
    st.set_page_config(page_title="MMQPKRS", layout="wide")
    st.header("Multimodal Query Processing & Knowledge Retrieval System")
    with st.expander("Click to View Shortcuts"):
        if st.session_state.shortcuts:
            num_columns = 4
            cols = st.columns(num_columns)
            DEFAULT_ADMIN_PASSWORD = os.getenv("PASSWORD")
            for i, (name, link, icon_data, user_password) in enumerate(st.session_state.shortcuts):
                col = cols[i % num_columns]
                with col:
                    st.image(icon_data, width=100)
                    st.markdown(f"[{name}]({link})", unsafe_allow_html=True)
                    password_input_delete = st.text_input("Enter Password to Delete", type="password",
key=f"delete_password_{i}")
                    if password_input_delete in [user_password, DEFAULT_ADMIN_PASSWORD]:
                        if st.button(f"Delete {name}", key=f"delete_{i}"):
                            st.session_state.shortcuts.pop(i)
                            save_shortcuts(st.session_state.shortcuts)
                            st.experimental_rerun()
                    elif password_input_delete:
                        st.warning("Incorrect password. You cannot delete this shortcut.")
        else:
            st.write("No shortcuts added yet.")
    st.markdown("Note: Add best web links for student education purposes.")
    if "query" not in st.session_state:
        st.session_state.query = ""
    # Initialize chat history if not present
    if "chat_history" not in st.session_state:
        st.session_state.chat_history = []
    # if st.sidebar.button("Voice Search"):
    #     query = voice_search()
    #     if query:
    #         st.session_state.query = query                gskdsrikrishna
    # else:
    #     st.session_state.query = ""
    # Sidebar options
    st.sidebar.title("Options")
    search_type = st.sidebar.radio("Select Search Type", (
        "Wikipedia", "Google", "YouTube", "News", "Chat",
        "Image Generation", "Logo Generation", "YT MP3",
        "LinkedIn Details", "Instagram Details", "IMDb Movie Search",
        "Real-Time Image Search" , "NASA Space Data Explorer" ,
        "AI Image Generator" , # Added new option
    ))
    # **Add this block after st.sidebar.title("Options")**
    st.sidebar.header("Only Add Best Web Links")
```

```
    name = st.sidebar.text_input("Shortcut Name")
    link = st.sidebar.text_input("Website Link (https://...)")
    icon_file = st.sidebar.file_uploader("Upload an Icon", type=["png", "jpg", "jpeg"])
    password_input_add = st.sidebar.text_input("Enter Password to Add Shortcut", type="password")
    if st.sidebar.button("Add Shortcut"):
        if password_input_add:
            if name and link and icon_file:
                existing_links = [shortcut[1] for shortcut in st.session_state.shortcuts]
                if link in existing_links:
                    st.sidebar.warning("This website already exists.")
                else:
                    st.session_state.shortcuts.append((name, link, icon_file.getvalue(),
password_input_add))
                    save_shortcuts(st.session_state.shortcuts)
                    st.sidebar.success(f"Shortcut '{name}' added!")
            else:
                st.sidebar.warning("Please enter all details including an icon.")
        else:
            st.sidebar.warning("Please enter a password to add the shortcut.")
    # Chat-specific sidebar settings
    chat_title = "Temporary Chat"
    if search_type == "Chat":
        chat_title = st.sidebar.text_input("Rename Chat:", "Temporary Chat")
    # Last seen timestamp
    st.sidebar.write(f"Last seen: {datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    if search_type == "Wikipedia":
        lang_map = {"English": "en", "Spanish": "es", "Chinese": "zh", "Hindi": "hi", "Telugu": "te"}
        selected_lang = st.sidebar.selectbox("Wikipedia Language", list(lang_map.keys()))
        summary_levels = ["Brief", "Detailed", "Bullet Points"]
        summary_level = st.sidebar.selectbox("Summarization Level", summary_levels)
        char_limit = st.sidebar.slider("Character Limit", min_value=100, max_value=2000, value=500,
step=100)
        st.title("Wikipedia Summary & Text-to-Speech")
        query = st.text_input("Enter a topic to search on Wikipedia:", value=st.session_state.query)
        if query:
            lang_code = lang_map[selected_lang]
            summary = get_wikipedia_summary(query, lang_code, char_limit, summary_level)
            st.markdown(f"### Summary for: {query}")
            st.write(summary)
            tts_filename = f"{query}_speech.mp3"
            if st.button("Play Text-to-Speech"):
                text_to_speech(summary, tts_filename, lang=lang_code)
                st.audio(tts_filename, format="audio/mp3")
        st.write("---")
        st.write("### Footer")
        st.write("This is a Wikipedia search section.")
    elif search_type == "Google":
        st.title("Google Search")
        query = st.text_input("Enter a search query for Google:", value=st.session_state.query)
        api_key = os.getenv("G_API")
        cse_id = os.getenv("G_ID")
        if query and st.button("Search"):
            results = google_search(api_key, cse_id, query)
            display_google_results(results)
        st.write("---")
        st.write("### Footer")
        st.write("This is a Google search section.")
    elif search_type == "YouTube":
        st.title("YouTube Search")
        query = st.text_input("Enter a topic to search on YouTube:", value=st.session_state.query)
        if query and st.button("Search"):
            videos = search_youtube(query)
            if videos:
                for video in videos:
                    st.write(f"[{video['title']}]({video['url']})")
                    st.image(video['thumbnail'])
                    st.video(video['url'])
                    st.write("---")
        st.write("---")
        st.write("### Footer")
        st.write("This is a YouTube search section.")
    # elif search_type == "News":
    #     st.subheader("Select Date Range")              gskdsrikrishna
```

```
    #     start_date = st.date_input("From", datetime.date.today() - datetime.timedelta(days=7))
    #     end_date = st.date_input("To", datetime.date.today())
    #     st.title("News Search")
    #     query = st.text_input("Enter a news topic to search:", value=st.session_state.query)
    #     if query and st.button("Search"):
    #         articles = search_news(query, start_date, end_date)
    #         display_news(articles)
    #     st.write("---")                        gskdsrikrishna
    #     st.write("### Footer")
    #     st.write("This is a news search section.")
    elif search_type == "News":
        st.subheader("Select Date Range")
        start_date = st.date_input("From", datetime.date.today() - datetime.timedelta(days=7))
        end_date = st.date_input("To", datetime.date.today())
        st.title("News Search")
        query = st.text_input("Enter a news topic to search:", value=st.session_state.query)
        if query and st.button("Search"):
            articles = search_news(query, start_date, end_date)
            display_news(articles)
        st.write("---")
        st.write("### Footer")
        st.write("This is a news search section.")
    elif search_type == "Chat":
        st.title(chat_title)
        for chat in st.session_state.chat_history:
            with st.chat_message(chat["role"]):
                st.write(chat["content"])
        user_input = st.text_input("Ask AI:")
        st.session_state.query = user_input  # Explicitly store the input
        if st.button("Generate Response"):
            if st.session_state.query and st.session_state.query.strip():
                with st.spinner("Generating response..."):
                    response = chat_with_mistral_hf(user_input)
                    st.session_state.chat_history.append({"role": "user", "content": user_input})
                    st.session_state.chat_history.append({"role": "assistant", "content": response})
                    st.rerun()
            else:
                st.warning("Please enter a prompt before clicking Generate Response.")
        st.write("---")
        st.write("### Footer")
        st.write("This is a Chat section.")
    elif search_type == "Image Generation":
        st.title("AI Image Generator")
        st.write("Generate AI-powered images using text prompts.")
        prompt = st.text_area("Enter your image description:")
        output_type = st.selectbox("Select output format:", ["png", "jpg"])
        def generate_image():
            url = "https://ai-image-generator14.p.rapidapi.com/"
            headers = {
                "x-rapidapi-key": AI_IMAGE_API_KEY,
                "x-rapidapi-host": "ai-image-generator14.p.rapidapi.com",
                "Content-Type": "application/json"
            }
            payload = {
                "jsonBody": {
                    "function_name": "image_generator",
                    "type": "image_generation",
                    "query": prompt,
                    "output_type": output_type
                }
            }
            response = requests.post(url, json=payload, headers=headers)
            return response.json()
        if st.button("Generate Image"):
            if prompt:
                with st.spinner("Generating image..."):
                    result = generate_image()
                    image_url = result.get("message", {}).get("output_png")
                    if image_url:
                        st.image(image_url, caption="Generated Image", use_container_width=True)
                        # Provide download button          gsldsrolrosjma
                        st.download_button(
                            label="Download Image",
```

```
                            data=requests.get(image_url).content,
                            file_name=f"generated_image.{output_type}",
                            mime=f"image/{output_type}"
                        )
                    else:
                        st.error("Failed to generate image. No image URL found in response.")
                        st.write("Response Data:", result)  # Debugging: Show response data
            else:
                st.warning("Please enter an image description.")
        st.write("---")
        st.write("### Footer")
        st.write("This is Image Generation section.")
    elif search_type == "Logo Generation":
        st.title("AI Logo Generator  ")
        st.write("Generate a unique logo for your business using AI!")
        # User Inputs
        prompt = st.text_input("Enter a description for the logo:", "Make a logo with the name Gskd")
        style = st.selectbox("Select a Logo Style:", [28, 29, 30], index=0)  # Customize styles if needed
        size = st.radio("Select Image Size:", ["1-1", "2-3", "3-4"], index=0)
        def generate_logo(prompt, style, size):
            api_url = "https://ai-logo-generator.p.rapidapi.com/aaaaaaaaaaaaaaaaiimagegenerator/quick.php"
            headers = {
                "x-rapidapi-key": LOGO_API_KEY,
                "x-rapidapi-host": "ai-logo-generator.p.rapidapi.com",
                "Content-Type": "application/json"
            }
            payload = {
                "prompt": prompt,
                "style_id": style,
                "size": size
            }
            response = requests.post(api_url, json=payload, headers=headers)
            return response.json()
        # Generate button                gskdsrikrishna
        if st.button("Generate Logo"):
            with st.spinner("Generating logo... Please wait!"):
                result = generate_logo(prompt, style, size)
                # Extracting image URLs from the JSON response
                image_data = result.get("final_result", [])
                if image_data:
                    st.subheader("Generated Logo Designs")
                    for index, image in enumerate(image_data):
                        image_url = image.get("origin")  # Extracting the logo link
                        if image_url:
                            st.image(image_url, caption=f"Logo Design {index+1}", use_container_width=True)
                            # Download button with a unique key
                            st.download_button(
                                label="Download Logo",
                                data=requests.get(image_url).content,
                                file_name=f"logo_design_{index+1}.webp",
                                mime="image/webp",
                                key=f"download_{index}"  # Unique key to prevent duplicate errors
                            )
                else:
                    st.error("Failed to generate logo. No image URL found.")
                    st.write("Response Data:", result)  # Debugging: Show full response data
        st.write("---")
        st.write("### Footer")
        st.write("This is Logo Generation section.")
    elif search_type == "YT MP3":
        st.title("  YouTube to MP3 Downloader")
        youtube_url = st.text_input("Enter YouTube Video URL:",
"https://youtu.be/05TA9jNnCdU?si=1bUk6EHOmwcsKC3H")
        if st.button("Convert to MP3"):
            with st.spinner("Processing... Please wait"):
                mp3_url = convert_youtube_to_mp3(youtube_url)
                if mp3_url:
                    st.subheader("  MP3 Download Link:")
                    st.markdown(f"[  Click Here to Download MP3]({mp3_url})")
                else:
                    st.warning("  Failed to retrieve the MP3 file. Please check the YouTube link and try
again.")
        st.markdown("---")
```

```
        st.caption("Powered by Streamlit")
        st.write("---")
        st.write("### Footer")
        st.write("This is YT MP3 section.")
    elif search_type == "LinkedIn Details":
        st.title("  LinkedIn Image Extractor")
        # User input for LinkedIn username
        username = st.text_input("Enter LinkedIn Username", "gskdsrikrishna")
        # Securely fetch API key from Hugging Face Secrets
        api_key = st.secrets["LI_KEY"]
        # API details
        url = "https://linkedin-api8.p.rapidapi.com/"
        headers = {
            "x-rapidapi-key": api_key,  # Secure API key         gskdsrikrishna
            "x-rapidapi-host": "linkedin-api8.p.rapidapi.com"
        }
        if st.button("Fetch Data"):
            querystring = {"username": username}
            response = requests.get(url, headers=headers, params=querystring)
            if response.status_code == 200:
                data = response.json()
                st.subheader("Extracted JSON Data")
                st.json(data)  # Display JSON data
                # Extract all URLs from JSON response
                urls = extract_urls(data)
                if urls:
                    st.subheader("  Extracted URLs")
                    for url in urls:
                        st.write(url)
                    # Download images and create a ZIP file   gskdsrikrishna
                    image_files = []
                    zip_buffer = BytesIO()
                    with zipfile.ZipFile(zip_buffer, "w") as zipf:
                        for idx, img_url in enumerate(urls):
                            try:
                                img_response = requests.get(img_url, stream=True)
                                if img_response.status_code == 200 and "image" in
img_response.headers["Content-Type"]:
                                    img_name = f"image_{idx+1}.jpg"
                                    img_data = img_response.content
                                    with open(img_name, "wb") as img_file:
                                        img_file.write(img_data)
                                    zipf.write(img_name, os.path.basename(img_name))
                                    image_files.append(img_name)
                            except Exception as e:
                                st.error(f"Error downloading {img_url}: {e}")
                    # Provide ZIP download button if images were found
                    if image_files:
                        zip_buffer.seek(0)
                        st.download_button(
                            label="  Download All Images as ZIP",
                            data=zip_buffer,
                            file_name="linkedin_images.zip",
                            mime="application/zip"
                        )
                    else:
                        st.warning("No images found in extracted URLs.")
                else:
                    st.warning("No URLs found in the response.")
            else:
                st.error("❌ Failed to fetch data. Check API key or username.")
        st.write("---")
        st.write("### Footer")
        st.write("This is Linkedin Details section.")
    elif search_type == "Instagram Details":
        st.title("  Instagram Scraper")
        # User Input
        username = st.text_input("Enter Instagram Username:", "gskdsrikrishna")
        # Load API Key from Hugging Face Secrets
        API_KEY = os.getenv("INSTA_API")
        HEADERS = {
            "x-rapidapi-key": API_KEY,
            "x-rapidapi-host": "instagram-scraper-api2.p.rapidapi.com"
```

```
        }
```

```
        if st.button("Fetch Details"):
            with st.spinner("Fetching data..."):
                response = requests.get(
                    "https://instagram-scraper-api2.p.rapidapi.com/v1/info",
                    headers=HEADERS,
                    params={"username_or_id_or_url": username}
                )
                if response.status_code == 200:
                    data = response.json()
                    st.success("✅ Data Retrieved Successfully!")
                    # Display JSON Response
                    st.subheader("  JSON Response")
                    st.json(data)
                    # Extract links
                    links = extract_links(data)
                    # Separate image links
                    image_links = [link for link in links if any(ext in link for ext in [".jpg", ".jpeg",
".png", ".webp"])]
                    other_links = [link for link in links if link not in image_links]
                    # Display Images
                    if image_links:
                        st.subheader("  Extracted Images:")
                        img_bytes_list = []
                        img_filenames = []
                        for idx, img_url in enumerate(image_links):
                            st.markdown(f"  [Image {idx + 1}]({img_url})")
                            img_bytes = requests.get(img_url).content
                            img_bytes_list.append(img_bytes)
                            img_filenames.append(f"{username}_image_{idx+1}.jpg")
                    # ZIP Download
                        if img_bytes_list:
                            with io.BytesIO() as zip_buffer:
                                with zipfile.ZipFile(zip_buffer, "w") as zip_file:
                                    for filename, img_data in zip(img_filenames, img_bytes_list):
                                        zip_file.writestr(filename, img_data)
                                zip_buffer.seek(0)
                                st.download_button(
                                    "  Download All Images",
                                    data=zip_buffer,
                                    file_name=f"{username}_images.zip",
                                    mime="application/zip"
                                )
                    # Display Other Links       gskdsrikrishna
                    if other_links:
                        st.subheader("  Other Links:")
                        for link in other_links:
                            st.markdown(f"[  {link}]({link})")
                    # Data Downloads
                    st.subheader("  Download Data")
                    json_data = json.dumps(data, indent=4)
                    text_data = "\n".join(f"{key}: {value}" for key, value in data.items())
                    st.download_button(
                        label="Download JSON",
                        data=json_data,
                        file_name=f"{username}_data.json",
                        mime="application/json"
                    )
                    st.download_button(
                        label="Download Text",
                        data=text_data,
                        file_name=f"{username}_details.txt",
                        mime="text/plain"
                    )
                else:
                    st.error("❌ Failed to retrieve data. Please check the username.")
        st.write("---")
        st.write("### Footer")
        st.write("This is Instagram Details section.")
    elif search_type == "IMDb Movie Search":
        st.title("  IMDb Movie Search")
        # User input for movie search
```

```
        search_term = st.text_input("Enter Movie Name", "Avengers")
```

```
        # Securely fetch API key from Hugging Face Secrets
        api_key = st.secrets["IMDB_KEY"]
        # API details
        url = "https://imdb-com.p.rapidapi.com/search"
        headers = {
            "x-rapidapi-key": api_key,  # Secure API key
            "x-rapidapi-host": "imdb-com.p.rapidapi.com"
        }
        if st.button("  Search"):
            querystring = {"searchTerm": search_term}
            response = requests.get(url, headers=headers, params=querystring)
            if response.status_code == 200:
                data = response.json()
                st.subheader("  Extracted JSON Data")
                st.json(data)  # Display JSON data
                # Extract movie details
                if "d" in data:
                    st.subheader("  Search Results")
                    for movie in data["d"]:
                        title = movie.get("l", "Unknown Title")
                        year = movie.get("y", "Unknown Year")
                        imdb_id = movie.get("id", "N/A")
                        poster = movie.get("i", {}).get("imageUrl", "")
                        st.markdown(f"###   {title} ({year})")
                        st.write(f"**IMDb ID:** `{imdb_id}`")
                        if poster:
                            st.image(poster, caption=title, use_column_width=True)
                # Extract all URLs from JSON response
                urls = extract_urls(data)
                if urls:
                    st.subheader("  Extracted URLs")
                    for url in urls:
                        st.write(url)
                    # Download images and create a ZIP file    gskdsrikrishna
                    image_files = []
                    zip_buffer = BytesIO()
                    with zipfile.ZipFile(zip_buffer, "w") as zipf:
                        for idx, img_url in enumerate(urls):
                            try:
                                img_response = requests.get(img_url, stream=True)
                                if img_response.status_code == 200 and "image" in
img_response.headers["Content-Type"]:
                                    img_name = f"image_{idx+1}.jpg"
                                    img_data = img_response.content
                                    with open(img_name, "wb") as img_file:
                                        img_file.write(img_data)
                                    zipf.write(img_name, os.path.basename(img_name))
                                    image_files.append(img_name)
                            except Exception as e:
                                st.error(f"Error downloading {img_url}: {e}")
                    # Provide ZIP download button if images were found
                    if image_files:
                        zip_buffer.seek(0)
                        st.download_button(
                            label="  Download All Images as ZIP",
                            data=zip_buffer,
                            file_name="imdb_images.zip",
                            mime="application/zip"
                        )
                    else:
                        st.warning("No images found in extracted URLs.")
                else:
                    st.warning("No URLs found in the response.")
            else:
                st.error("❌ Failed to fetch data. Check API key or search term.")
        st.write("---")
        st.write("### Footer")
        st.write("This is IMDB Movie Search section.")
    elif search_type == "Real-Time Image Search":
        st.title("  Real-Time Image Search")
        # User input for image search term
```

```
        search_query = st.text_input("Enter Image Search Term", "India")
        # Securely fetch API key from Hugging Face Secrets
        api_key = st.secrets["IMG_KEY"]
        # API details
        url = "https://real-time-image-search.p.rapidapi.com/search"
        headers = {
            "x-rapidapi-key": api_key,  # Secure API key
            "x-rapidapi-host": "real-time-image-search.p.rapidapi.com"
        }
        if st.button("  Search Images"):
            querystring = {
                "query": search_query,
                "limit": "25",
                "size": "any",
                "color": "any",
                "type": "any",
                "time": "any",
                "usage_rights": "any",
                "file_type": "any",
                "aspect_ratio": "any",
                "safe_search": "off",
                "region": ""
            }
            response = requests.get(url, headers=headers, params=querystring)
            if response.status_code == 200:
                data = response.json()
                st.subheader("  Extracted JSON Data")
                st.json(data)  # Display JSON data
                # Extract image URLs
                image_urls = extract_urls(data)
                if image_urls:
                    st.subheader("  Extracted Image Links")
                    cols = st.columns(3)  # Display images in a 3-column grid
                    for idx, img_url in enumerate(image_urls):
                        cols[idx % 3].image(img_url, width=200, caption=f"Image {idx+1}")
                    # Download images and create a ZIP file
                    image_files = []
                    zip_buffer = BytesIO()
                    with zipfile.ZipFile(zip_buffer, "w") as zipf:
                        for idx, img_url in enumerate(image_urls):
                            try:
                                img_response = requests.get(img_url, stream=True)
                                if img_response.status_code == 200 and "image" in
img_response.headers["Content-Type"]:
                                    img_name = f"image_{idx+1}.jpg"
                                    img_data = img_response.content
                                    with open(img_name, "wb") as img_file:
                                        img_file.write(img_data)
                                    zipf.write(img_name, os.path.basename(img_name))
                                    image_files.append(img_name)
                            except Exception as e:
                                st.error(f"Error downloading {img_url}: {e}")
                    # Provide ZIP download button if images were found          gskdsrikrishna
                    if image_files:
                        zip_buffer.seek(0)
                        st.download_button(
                            label="  Download All Images as ZIP",
                            data=zip_buffer,
                            file_name="searched_images.zip",
                            mime="application/zip"
                        )
                    else:
                        st.warning("No downloadable images found.")
                else:
                    st.warning("No image URLs found in the response.")
            else:
                st.error("❌ Failed to fetch data. Check API key or search term.")
        st.write("---")
        st.write("### Footer")
        st.write("This is Real Time Image Search section.")
    elif search_type == "NASA Space Data Explorer":
        st.title("  NASA Space Data Explorer")
        # NASA API Key
```

```
        NASA_API_KEY = os.getenv("NASA")  # Ensure this is set in your environment
        # API Endpoints
        APOD_URL = "https://api.nasa.gov/planetary/apod"
        MARS_ROVER_URL = "https://api.nasa.gov/mars-photos/api/v1/rovers/{}/photos"
        NEO_URL = "https://api.nasa.gov/neo/rest/v1/feed"
        EARTH_IMG_URL = "https://api.nasa.gov/planetary/earth/imagery"
        EPIC_URL = "https://api.nasa.gov/EPIC/api/natural"
        DONKI_URL = "https://api.nasa.gov/DONKI/CME"
        IMAGE_LIBRARY_URL = "https://images-api.nasa.gov/search"
        # Helper functions
        def fetch_json(url, params=None):
            """Fetch JSON data from a URL, handling errors."""
            try:
                if params is None:
                    params = {"api_key": NASA_API_KEY}
                response = requests.get(url, params=params)
                response.raise_for_status()  # Raises an error for bad responses (4xx, 5xx)
                return response.json()
            except requests.exceptions.RequestException as e:
                st.error(f"Failed to fetch data: {e}")
                return None
        def download_image(image_url, filename):
            """Download image button."""
            response = requests.get(image_url)
            if response.status_code == 200:
                st.download_button(
                    label="Download Image",
                    data=BytesIO(response.content),
                    file_name=filename,
                    mime="image/jpeg"
                )
        # Sidebar options for NASA features             gskdsrikrishna
        nasa_option = st.sidebar.radio("Select NASA Feature", [
            "Astronomy Picture of the Day",
            "Mars Rover Photos",
            "Near-Earth Object Data",
            "Earth Imagery",
            "EPIC Earth Images",
            "DONKI - Space Weather",
            "NASA Image and Video Library"
        ])
        if nasa_option == "Astronomy Picture of the Day":
            st.subheader("  Astronomy Picture of the Day")
            response = fetch_json(APOD_URL)
            if response:
                st.image(response['url'], caption=response['title'])
                st.write(response['explanation'])
                download_image(response['url'], "apod.jpg")
        elif nasa_option == "Mars Rover Photos":
            st.subheader("  Mars Rover Photos")
            rover = st.selectbox("Choose a rover", ["curiosity", "opportunity", "spirit"])
            sol = st.number_input("Enter Sol (Mars day)", min_value=0, value=1000)
            response = fetch_json(MARS_ROVER_URL.format(rover), params={"sol": sol, "api_key":
NASA_API_KEY})
            if response and "photos" in response:
                for idx, photo in enumerate(response["photos"][:5]):
                    st.image(photo['img_src'], caption=f"Taken on {photo['earth_date']}")
                    download_image(photo['img_src'], f"mars_photo_{idx}.jpg")
        elif nasa_option == "Near-Earth Object Data":
            st.subheader("  Near-Earth Objects")
            response = fetch_json(NEO_URL, params={"start_date": date.today(), "api_key": NASA_API_KEY})
            if response and "near_earth_objects" in response:
                for neo in response["near_earth_objects"].get(str(date.today()), []):
                    st.write(f"**{neo['name']}** - Diameter:
{neo['estimated_diameter']['meters']['estimated_diameter_max']}m")
        elif nasa_option == "Earth Imagery":
            st.subheader("  Earth Imagery")
            lat = st.number_input("Enter Latitude", value=37.7749)
            lon = st.number_input("Enter Longitude", value=-122.4194)
            response = fetch_json(EARTH_IMG_URL, params={"lon": lon, "lat": lat, "dim": 0.1, "api_key":
NASA_API_KEY})
            if response and 'url' in response:
                st.image(response['url'], caption=f"Satellite Image for ({lat}, {lon})")
```

```
                download_image(response['url'], "earth_image.jpg")
        elif nasa_option == "EPIC Earth Images":
            st.subheader("  EPIC Earth Images")
            response = fetch_json(EPIC_URL, params={"api_key": NASA_API_KEY})
            if response:
                for img in response[:5]:  # Limit to 5 images
                    year, month, day = img['date'].split()[0].split('-')
                    img_url =
f"https://epic.gsfc.nasa.gov/archive/natural/{year}/{month}/{day}/png/{img['image']}.png"
                    st.image(img_url, caption=f"Captured on {img['date']}")
                    download_image(img_url, f"epic_{img['date']}.png")
        elif nasa_option == "DONKI - Space Weather":
            st.subheader("  Space Weather Data")
            response = fetch_json(DONKI_URL, params={"api_key": NASA_API_KEY})
            if response:
                for event in response[:5]:  # Limit to 5 events
                    st.write(f"Event: {event.get('note', 'No description available')}")
        elif nasa_option == "NASA Image and Video Library":
            st.subheader("  NASA Image and Video Library")
            query = st.text_input("Search for images/videos", value="moon")
            response = fetch_json(IMAGE_LIBRARY_URL, params={"q": query, "media_type": "image"})
            if response and "collection" in response:
                for idx, item in enumerate(response["collection"]["items"][:5]):  # Limit to 5 items
                    st.image(item["links"][0]["href"], caption=item["data"][0]["title"])
                    download_image(item["links"][0]["href"], f"nasa_image_{idx}.jpg")
        st.write("---")
        st.write("### Footer")
        st.write("This is NASA section.")
    elif search_type == "AI Image Generator":
        st.title("AI Image Generator   ")
        st.write("Enter a prompt below and generate an AI-generated image using Hugging Face!")
        # User Input
        prompt = st.text_input("Enter your prompt:", "Astronaut riding a horse")
        if st.button("Generate Image"):
            if prompt:
                st.write("Generating image... Please wait ⏳ ")
                # Query Hugging Face API
                API_URL = os.getenv("FLUX")
                headers = {"Authorization": f"Bearer {os.getenv('HF_API')}"}
                def query(payload):
                    response = requests.post(API_URL, headers=headers, json=payload)
                    return response.content
                image_bytes = query({"inputs": prompt})
                # Display Image
                image = Image.open(io.BytesIO(image_bytes))  # Now works because Image is imported
                st.image(image, caption="Generated Image", use_container_width=True)
                # Convert Image to Bytes for Download
                img_buffer = io.BytesIO()
                image.save(img_buffer, format="PNG")
                img_buffer.seek(0)
                # Download Button
                st.download_button(
                    label="Download Image  ",
                    data=img_buffer,
                    file_name="generated_image.png",
                    mime="image/png"
                )
            else:
                st.warning("Please enter a prompt before generating an image.")
        # Footer
        st.write("---")
        st.write("Powered by [Hugging Face](https://huggingface.co/)  ")
if __name__ == "__main__":
    main()
#This project was developed in 2025 by @gskdsrikrishna
```

Notes

✧