INTEL UNNATI

PROBLEM STATEMENT 7: Visual Search Engine Using VLMS

TITLE: KAATCHI AI - A Fashion Visual Search Engine Using VLMs

Team Name: Intel DeepLearners

Submitted by:

Angeline A - 2348409

Prathish R K - 2848444

Selvakumar S - 2348454

Under the guidance of

Dr. Balakrishnan C

April - 2025

ABSTRACT

With the explosive growth in e-commerce, the need for more efficient and intelligent product search processes has grown ever more urgent. Conventional search processes that depend greatly on keyword-based searching and manual filtering are usually inadequate in capturing fashion products' fine-grained visual characteristics. This deficiency results in a less than satisfactory user experience because customers have to search endlessly for products closely resembling their choice. To solve this challenge, we propose KAATCHI, a cutting-edge fashion visual search engine which uses Vision-Language Models (VLMs) to retrieve fashion products on both text and image-based queries. By representing images and text descriptions in a shared representation space by using an advanced model like CLIP, KAATCHI makes the search more natural and accurate and provides an easy way to discover relevant products.

In order to optimize retrieval performance and scalability, KAATCHI combines FAISS (Facebook AI Similarity Search) with fast nearest neighbor search, so it supports real-time performance regardless of large data. The platform has an interactive and user-centric React-Node.js interface that incorporates adaptive themes, category filters, and product match percentage bars in order to create a rich search experience. Through the support of multi-modal search, such as text, image, and compositional queries, KAATCHI creates a new standard for fashion product discovery. This paper presents a detailed description of the system's design, architecture, implementation, and evaluation, highlighting how KAATCHI can transform fashion e-commerce through precise, scalable, and smart product retrieval.

TABLE OF CONTENTS

Abstract i

List of Figures iv

1. Introduction 1

    1.1. Problem Description 1

    1.2. Existing System 1

    1.3. Project Scope 2

2. System Analysis 3

    2.1. Functional/Non-Functional Specification 3

        2.1.1. Functional Specification 3

        2.1.2. Non-Functional Specification 4

    2.2. Block Diagram 6

    2.3. System Requirements 8

3. System Design 10

    3.1. System Architecture 10

    3.2. Module Design 11

    3.3. Database Design 14

        3.3.1. Data Flow Diagram 14

        3.3.2. ER Diagram 14

    3.4. System Configuration 16

    3.5. Interface Design 17

        3.5.1. User Interface Screen Design 17

        3.5.2. Application Flow/Class Diagram 20

    3.6. Reports Design 21

4. Implementation 22

    4.1. Coding Standard 22

    4.2. Screen Shots 23

5. Testing 30

5.1. Test Cases 30

5.2. Test Reports 31

6. Conclusion 35

6.1. Design and Implementation Issues 35

6.2. Advantages and Limitations 36

6.3. Future Enhancements 37

7. Contribution 39

8. Demo Video 39

References 40

LIST OF FIGURES

Figure Number	Title	Page Number
1	Block Diagram	7
2	System Architecture	10
3	Module Design	13
4	Data Flow Diagram	14
5	ER Diagram	15
6	Home Page -1	17
7	Home Page -2	18
8	Search Page	18
9	Chat Page	19
10	About Page	19
11	Application Flow Diagram	20
12	Search – UI/UX – 1	23
13	Search – UI/UX – 2	24
14	VLM Calling	24
15	CLIP Model Loading	25
16	Image Embeddings	25
17	Text Embeddings	26
18	FAISS Initialization	26
19	Text Query	27
20	Image Query	28
21	Multimodal Query – 1	28
22	Multimodal Query – 2	29
23	Multimodal Query – 3	29
24	Text Query – Case 1: Red Sneakers	31
25	Image Query – Test Case 1 – Red T Shirt	32
26	Multimodal Query – Test Case 1 – Black Dress	32

27	Fashion Product Validation	33
28	Non-Fashion Text Query	33
29	Non-Fashion Image Query	34
30	Non-Fashion Multimodal Query	34

1. INTRODUCTION

1.1. Problem Description

Fashion e-commerce has seen rapid growth, with customers increasingly turning towards online platforms to browse and shop for clothes and accessories. As such, conventional search mechanisms employed by these platforms, which are mainly keyword-matching and pre-defined filters, do not often provide accurate and contextually meaningful results. Users often find it challenging to discover products that suit their tastes, especially when they are looking for particular styles, patterns, or visual characteristics that cannot be easily put into words. Such a limitation does not only impact user experience but also affects customer interaction and conversion rates.

To fill this void, an intelligent and intuitive search functionality is required that not only grasps textual descriptions but also visual signals. KAATCHI AI seeks to overcome this challenge by utilizing Vision-Language Models (VLMs) [1] to enable multi-modal search functionality. In contrast to conventional keyword-based search engines, KAATCHI AI allows users to search for fashion items based on images, text descriptions, or both, to provide more accurate retrieval. By integrating text and images into a common representation space, the system is able to effectively capture semantic similarities among fashion products, enabling more accurate and relevant search results. This method not only improves the productivity of product discovery but also offers a smooth and interactive shopping experience for users.

1.2. Existing Systems

Traditional search engines for fashion use keyword-based search, category filtering, and simple image-based retrieval to enable users to find products. Though these techniques are somewhat accurate, they tend to lack semantic intelligence and flexibility.

- **Keyword-Based Search:** Text-based queries equate keywords with product metadata but don't capture visual features such as patterns or styles. This means that irrelevant search results are provided if the wording of the user doesn't perfectly match the terms in the database.

- **Category Filtering:** Users filter manually using color, material, and type filters, which are time-consuming and labor-intensive. It is also less user-intent adaptive, making it hard to locate products based on style.
- **Image-Based Retrieval:** Certain platforms enable users to upload images for search, but such systems are generally based on shallow feature matching instead of deep semantic comprehension. They tend not to differentiate between subtle design aspects or map visual similarities to textual descriptions.

Although these conventional approaches have some degree of accuracy, they are inflexible, inefficient, and do not comprehend sophisticated fashion tastes. There is a need for a more intelligent, multi-modal solution to improve product discovery.

1.3. Project Scope

KAATCHI AI aims to revolutionize fashion product discovery through multi-modal retrieval by using Vision-Language Models (VLMs) [1]. In contrast to legacy search systems based on simple keyword searching or generic image similarity comparison, KAATCHI AI enables an intelligent and context-aware search that gets both visual and text meanings.

The system accommodates text-based, image-based, and compositional search queries, enabling users to query products based on descriptive phrases, reference images, or both. The system maintains high retrieval accuracy by projecting both modalities into a common embedding space. KAATCHI AI also has non-fashion product detection, such that if a user queries for a product outside the fashion domain, the system will alert them instead of producing irrelevant results.

For effectively dealing with high volumes of datasets, KAATCHI AI incorporates FAISS [2] based indexing that allows efficient, scalable, and accurate retrieval. Intelligent search, stringent filtering, and responsive UI interact seamlessly is making KAATCHI AI a capable answer for contemporary fashion searching requirements yet ensures domain specificity for avoiding out-of-context results.

2. SYSTEM ANALYSIS

2.1. Functional/Non-Function Specifications

2.1.1. Functional Specifications

KAATCHI AI is made to offer an intelligent and efficient fashion product search experience by utilizing multi-modal retrieval strength. It incorporates Vision-Language Models (VLMs) [1] like CLIP [3] for processing textual and visual queries, ensuring effective retrieval of appropriate fashion items. The following are the functional requirements of KAATCHI AI:

- **Multi-Modal Query Processing:**
  - Fashion products can be searched by users based on textual descriptions, reference images, or both.
  - The system can process natural language queries (e.g., "a red flower dress for summer") and returns semantically similar products.
  - For visual-based queries, the system processes the visual attributes and discovers similar-looking products in the database.
  - For the multimodal query, the system also validates if both the text and image query describe the same product, if not warns the user.
- **Embedding Generation with VLMs:**
  - KAATCHI AI uses CLIP [4] to compute high-dimensional embeddings for images and text such that similar products and descriptions are embedded together in the embedding space.
  - This method boosts search precision, allowing for semantic comprehension instead of mere keyword matching.
- **Real-Time Similarity Search using FAISS:**
  - The system utilizes FAISS (Facebook AI Similarity Search) [2] to carry out efficient large-scale indexing and retrieval.
  - FAISS [2] allows fast approximate nearest neighbor (ANN) search, making it possible for the system to retrieve visually and semantically close products in real-time even when handling a large dataset.

- **Product Retrieval and Matching Percentage:**
  - Retrieved fashion products are displayed along with a matching confidence score, indicating how closely they match the user's query.
  - The system ranks results based on similarity scores, ensuring that the most relevant items appear first.
- **Intuitive User Interface with Advanced Features:**
  - A React-Node.js-based UI provides a seamless and engaging user experience.
  - The interface includes features like:
    1. **Product Filtering:** Users can refine search results based on categories, colors, seasons, and other attributes.
    2. **Adaptive Themes:** The UI dynamically adapts to user preferences for a customized experience.
    3. **Product Details Page:** Clicking on a retrieved product gives a detailed description with extra metadata.
    4. **Chat-Based Interaction:** Users can interact with a chatbot to query the dataset, search functionality, or fashion trends.
- **Domain-Specific Filtering:**
  - KAATCHI AI ensures that only fashion-related products are retrieved.
  - If a non-fashion product is requested (e.g., "a sports car" or "a mountain scene"), the system informs the user that the product lies outside its field instead of providing inappropriate results.

By combining cutting-edge AI models, effective retrieval methods, and an interactive UI, KAATCHI AI revolutionizes fashion product search, providing a scalable, smart, and user-friendly solution with respect to the requirements of contemporary e-commerce sites.

2.1.2. Non-Functional Specifications

KAATCHI AI is optimized to address high-performance, scalability, and usability needs critical to a contemporary visual search engine. The non-functional requirements guarantee that the system remains efficient, responsive, and resilient to future developments. Following are the primary non-functional elements of KAATCHI AI:

- **Scalability:**
  - The system is developed to manage big-scale fashion datasets effectively, accommodating millions of product images without serious performance impact.
  - FAISS (Facebook AI Similarity Search) facilitates optimized retrieval and indexing, which allows for smooth growth of the dataset without affecting accuracy and performance.
  - Modular architecture allows the system to be easily extended to new collections, brands, or product categories without needing a complete revamp.
- **Low Latency and High Performance:**
  - KAATCHI AI is designed for real-time search, so users get relevant results in milliseconds of entering a query.
  - FAISS indexing and approximate nearest neighbor (ANN) search algorithms reduce the time to search, keeping the system responsive despite heavy query loads.
  - Parallel computation and batch processing also improve performance, minimizing delays when retrieving pertinent fashion products.
- **Extensibility and Adaptability:**
  - Though KAATCHI AI is intended for fashion product search, its architecture provides for adaptation to other domains, i.e., art, furniture, or general e-commerce use.
  - The framework can be updated with fresh datasets or enriched with more Vision-Language Models (VLMs) [1] to enhance retrieval quality for diverse industries.
  - New filtering, ranking, or recommendation rules can be added without impacting current functionality.
- **User Interface and Experience Optimization:**
  - The UI is built using React-Node.js with ease of navigation, providing a seamless experience for occasional users as well as fashionistas.
  - Functionality such as adaptive themes, filtering by categories, product match percentage, and interactive chatbot support increases usability and engagement.
  - The error-handling system provides such that non-fashion questions (e.g., "mountains," "cars") trigger an explicit message rather than showing irrelevant results.

- The UI is designed for both desktop and mobile use, with a responsive design that adjusts to varying screen sizes.

- **Reliability and Robustness:**
    - KAATCHI AI is optimized to process simultaneous user requests effectively, making it appropriate for high-traffic large-scale e-commerce websites.
    - The backend services provide fault tolerance and effective load balancing to avoid crashes or slowing down due to heavy usage.
    - A strong error-handling mechanism provides system stability, offering users useful feedback in the event of failure or unexpected input.

By keeping scalability, efficiency, flexibility, and user experience at the center, KAATCHI AI delivers a high-performing and future-proof visual search engine that maximizes product discovery for the fashion sector.

2.2. Block Diagram

Fig. 1 represents the KAATCHI AI architecture, a fashion visual search engine employing Vision-Language Models (VLMs). The user interacts with the frontend, including a user interface with both the search interface and the chat interface. These elements forward requests to the backend API layer, which executes queries on the search engine. The search engine makes use of CLIP models for generating embeddings and FAISS indexing to achieve effective retrieval. The embeddings point towards a fashion dataset, which comprises images as well as metadata. This organized sequence facilitates multi-modal search, through which users are able to obtain fashion products either from images or text or any combination of them while maintaining swift and precise retrieval.

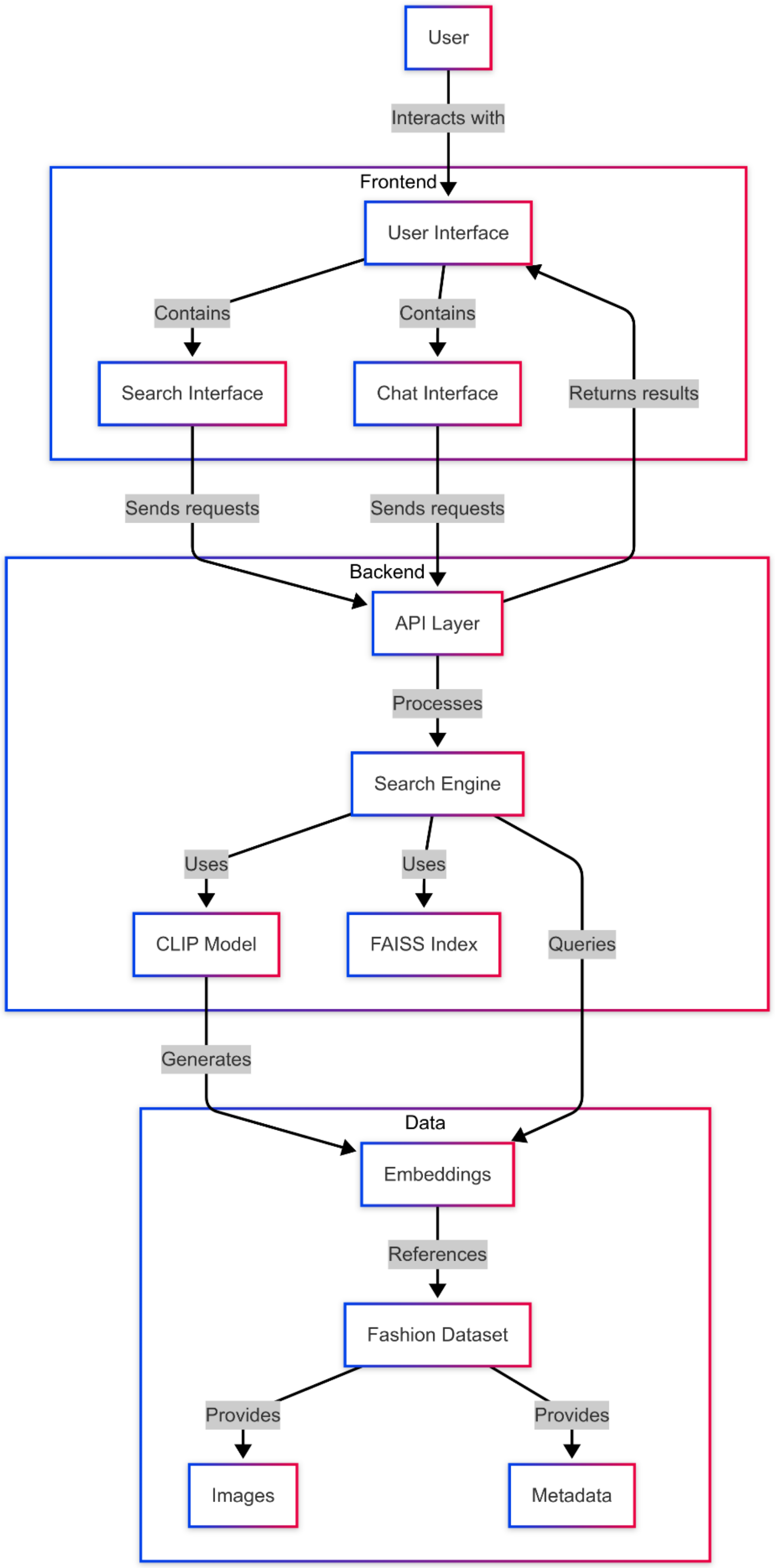


Fig.1: Block Diagram

2.3. System Requirements

KAATCHI AI calls for a set of high-performance hardware and highly optimized software pieces that facilitate the use of visual search operations. Large-scale fashion data can be handled by the system, along with generating embeddings based on Vision-Language Models (VLMs) and enabling real-time retrieval by FAISS indexing. These are the key requirements of the system:

- **Hardware Requirements:**

  For utilizing deep learning-based embeddings, optimized indexing, and real-time search, KAATCHI AI needs the following,

  - **GPU Acceleration:** A powerful NVIDIA GPU (e.g., RTX 3090, A100, or similar) for optimized processing of CLIP, BLIP, and ALIGN embeddings.
  - **RAM:** At least 16GB RAM, but 32GB+ is preferred for processing large-scale image datasets and conducting real-time similarity searches.
  - **Storage:** At least 500GB SSD (ideally NVMe) for efficient data access and indexing, particularly for storing extracted embeddings and FAISS indices.
  - **CPU:** A current multi-core processor (Intel i7/i9) to handle backend processing, data computation, and simultaneous user requests effectively.

- **Software Requirements:**

  KAATCHI AI is built on a mix of deep learning, web development, and database technologies to facilitate seamless integration between model inference and user interaction. The primary software requirements are:

  - **Operating System:** Linux-based distributions (Ubuntu 20.04+ recommended) or Windows (WSL2 for Linux support).
  - **Programming Languages:**
    1. Python – For training and inference of Vision-Language Model (CLIP) [3].
    2. JavaScript (Node.js, React) – For frontend and backend development.
  - **Machine Learning Frameworks:**
    1. PyTorch – For loading and fine-tuning CLIP [4] model.
    2. FAISS – For efficient large-scale similarity search and indexing [2].
  - **Web Development Frameworks & Libraries:**

1. Node.js & Express.js – Backend API development for processing user queries.
2. React.js – Frontend framework for the interactive UI.

- **Database Management:** FAISS Index Storage – For quick retrieval of precomputed embeddings.

- **Libraries & Dependencies**

Several essential libraries power KAATCHI AI's multi-modal retrieval, image processing, and data handling:

- **Vision-Language Model:** CLIP [4] – For embedding both images and text into a shared feature space.
- **Indexing & Retrieval:** FAISS – For performing approximate nearest neighbor (ANN) search at scale.
- **Data Processing & Analysis:**

1. Pandas – For managing and processing structured metadata.
2. NumPy – For efficient numerical computations related to embeddings and indexing.

- **Image Processing:** Pillow & OpenCV – For image preprocessing prior to embedding extraction.

Through utilization of high-end hardware, cutting-edge deep learning platforms, and search optimization libraries, KAATCHI AI provides an optimal, scalable, and precise visual search experience suited to the fashion domain.

3. SYSTEM DESIGN

3.1. System Architecture

Fig. 2 shows the system architecture of KAATCHI AI, a fashion vision search engine with Vision-Language Model (VLM) as its backend. The frontend interface handles the interaction of the user with text search, image search, and multimodal search. All these queries are handled by the API layer that sends them either to the search service or to the chat service.

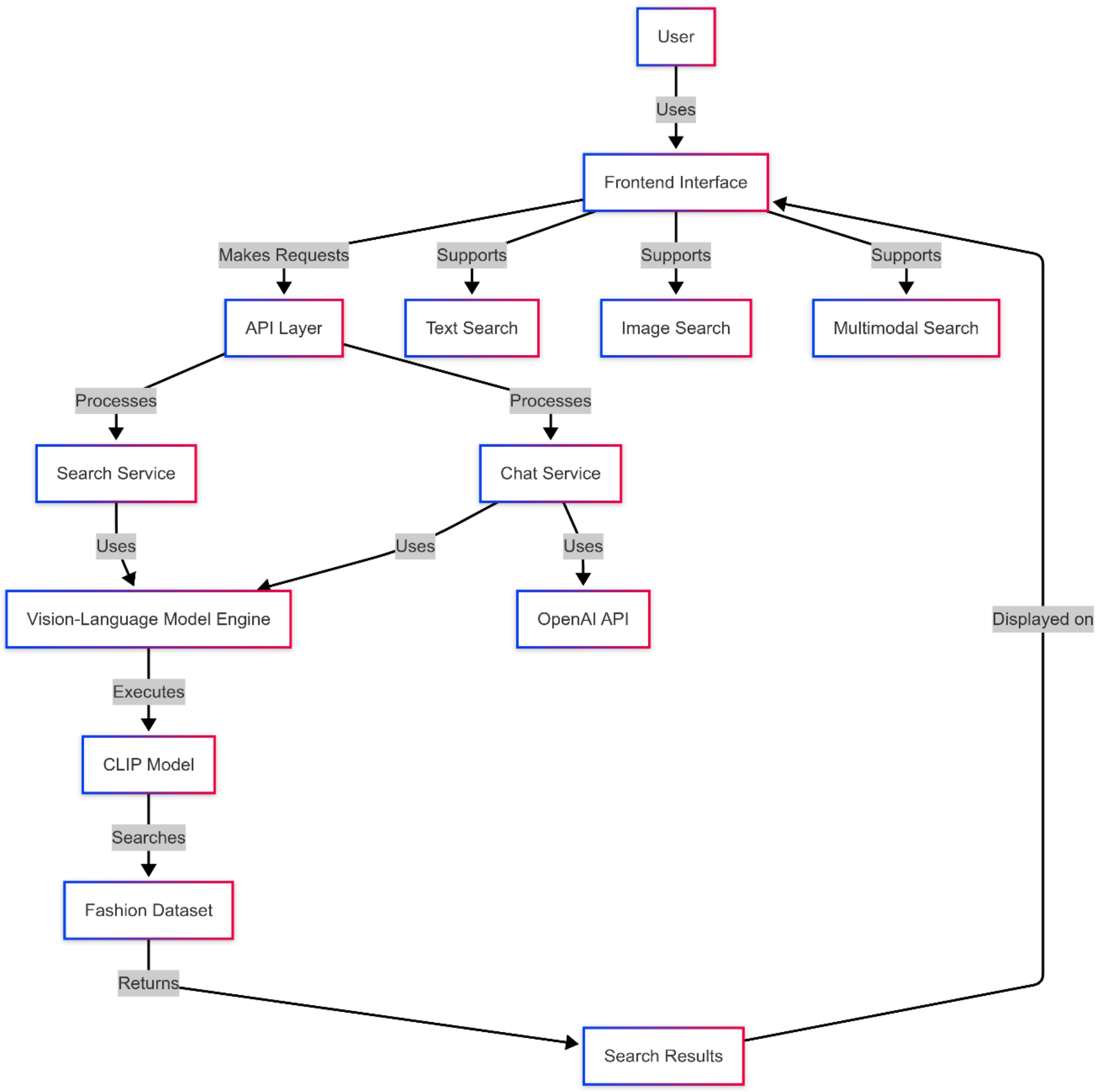


Fig.2. System Architecture

The search service consumes the Vision-Language Model Engine, which runs the CLIP model to generate embeddings that would search the fashion dataset for relevant results. On the other hand, the chat service relies on an API from OpenAI to process the inquiries from the user. When the search or chat is done, the results retrieved would be displayed to the user. This architecture is designed to offer a smooth and effective multi-modal search experience to fashion product discovery.

3.2. Module Design

KAATCHI AI is organized into several functional modules, with each handling a vital component of the visual search pipeline. In cooperation, they facilitate efficient multi-modal retrieval by processing queries, producing embeddings, indexing data, and providing an interactive search experience. The main modules of the system are as follows:

Data Preprocessing:

The dataset involves fashion related images, their metadata and a file describing all the images. The size of the dataset used for this project is 15GB and there are 44.4K fashion images and their related product descriptions in the dataset. The preprocessing module for data guarantees metadata and images are best organized for embedding generation and retrieval. This module entails:

- **Metadata Cleaning:** Deletion of inconsistencies, missing value handling, and attribute label standardization (e.g., categories, colors, seasons).
- **Image Processing:** Resizing, normalization, and transforming images to the desired format for Vision-Language Model (VLM) [1] processing.

Embedding Generation:

KAATCHI AI uses Vision-Language Model (CLIP) to create embeddings that project both text and images into a common representation space. The embedding generation module consists of:

- **Image Embeddings:** Obtaining high-dimensional representations from product images using pre-trained VLMs.
- **Text Embeddings:** Representing product names, descriptions, and user queries as vectors that correspond to image embeddings.

- **Multi-Modal Representation:** Mapping visually similar items and semantically close text descriptions proximal to one another in the embedding space.

Indexing & Retrieval:

In order to enable real-time search, KAATCHI AI deploys an optimized FAISS-based retrieval module for similarity matching with high efficiency. The module consists of:

- **Indexing Image Embeddings:** Precomputing image embeddings to store in FAISS for efficient nearest neighbor search.
- **Indexing Text Embeddings:** Facilitating direct text-based queries through the indexing of text embeddings and image embeddings.
- **Efficient Query Processing:** Upon a user's input query (text or image), KAATCHI AI finds the nearest match from the most relevant products through approximate nearest neighbor (ANN) search to provide fast retrieval.
- **Product Matching Percentage:** Determining the similarity score between the query and results retrieved to represent the relevance of each product.

UI/UX Module:

The frontend module offers an interactive and intuitive user interface for searching and browsing returned products. The UI/UX is implemented with React and Node.js, providing:

- **Multi-Modal Search Interface:** Users are able to search by text, image, or both, and results are returned in a visually pleasing manner.
- **Product Display & Filtering:** Returned products are displayed with relevance scores, and users are able to narrow searches through filters on categories, colors, and other features.
- **Adaptive Themes & Accessibility:** Facility to change interface to dark/light theme and accessibility-friendly options.
- **Chat-Based Query System:** Integrated chatbot that resolves dataset and product search questions, improving user interaction.

Combining these modules seamlessly, KAATCHI AI provides a scalable high-performance and user-friendly visual search engine optimized for the fashion world.

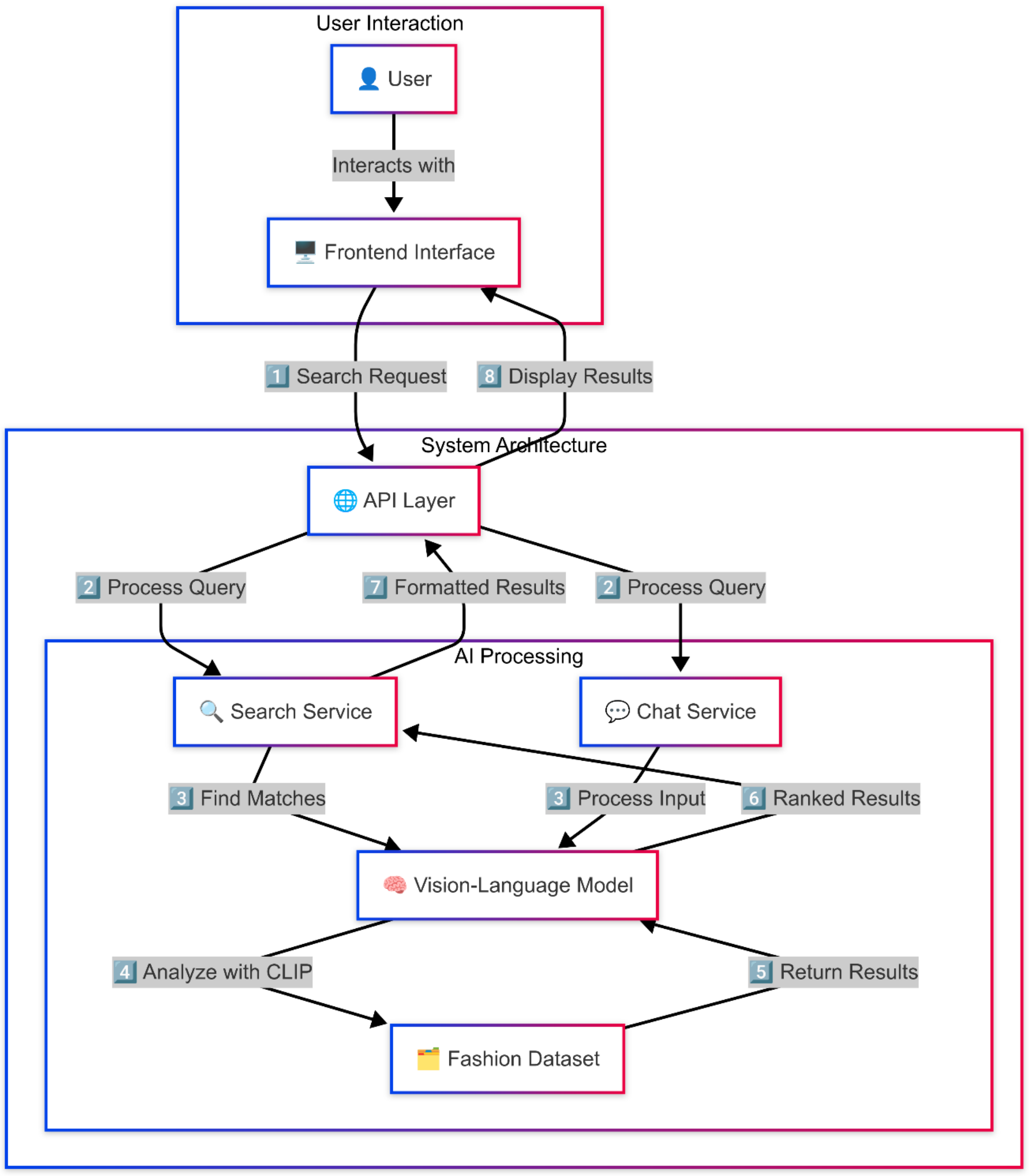


Fig. 2: Module Design

The same is being is explained in the Fig.2. Fig.2, System Architecture Diagram depicts the KAATCHI AI fashion visual search engine workflow with a focus on user interaction, system architecture, and AI processing. The workflow begins when a user interacts with the frontend interface, entering a search request (text or image). The API layer handles this query and passes it to the Search Service (for retrieval) or Chat Service (for text-based inquiries). The Vision-

Language Model (VLM), which is driven by CLIP, processes the query and looks for appropriate matches within the Fashion Dataset. The results retrieved are ranked and structured before being sent back to the user interface for display. This systematic approach facilitates efficient query processing, smart retrieval, and a smooth user experience.

3.3. Database Design

3.3.1. Data Flow Diagram

Fig.4. represents data flow in KAATCHI AI, a fashion visual search engine. The process starts with the input of the user, where the users give a text query or upload an image. The input is processed at the text level and image level, creating vector embeddings using text encoding and image encoding. The embeddings are utilized for index lookup in the vector index, which fetches related IDs from the fashion database. Similarity search is conducted to retrieve product information that is relevant. The results obtained are ranked and formatted before displaying it in the user interface. This formatted process ensures that the visual search is efficient and precise.

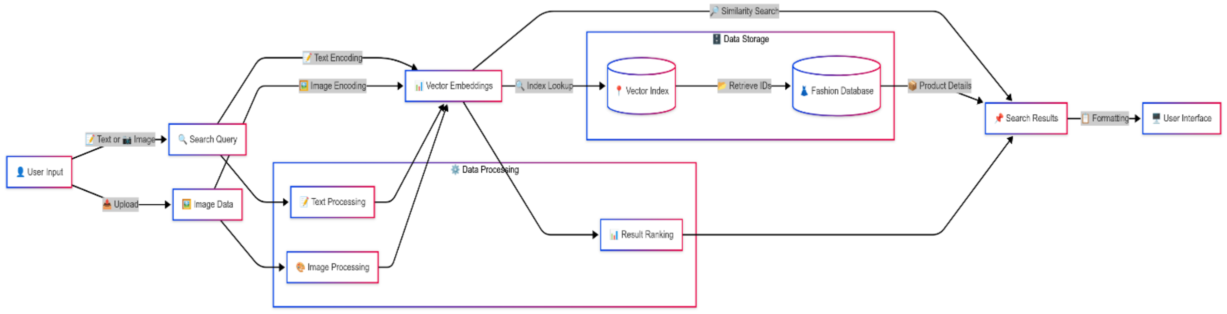


Fig.4. Data Flow Diagram

3.3.2. ER Diagram

Fig. 5, Entity-Relationship (ER) diagram illustrates the database schema for KAATCHI AI, a fashion visual search engine. It captures the relationships among various entities involved in the retrieval and search process. The SEARCH_QUERY entity logs user queries, referencing SEARCH_RESULT, which stores retrieved products ranked according to similarity. PRODUCT is the core entity that references different attributes: PRODUCT_IMAGE (hosting more than one image for a product), COLOR, GENDER, SEASON, and USAGE (denoting intended use). The EMBEDDING entity refers to embedding products within vector representations, allowing for effective searching. Products are of SUB_CATEGORY type, which is further classified under MASTER_CATEGORY. ARTICLE_TYPE further classifies fashion goods under subcategories. This organized schema guarantees efficient storage, retrieval, and categorization of fashion items in the visual search engine.

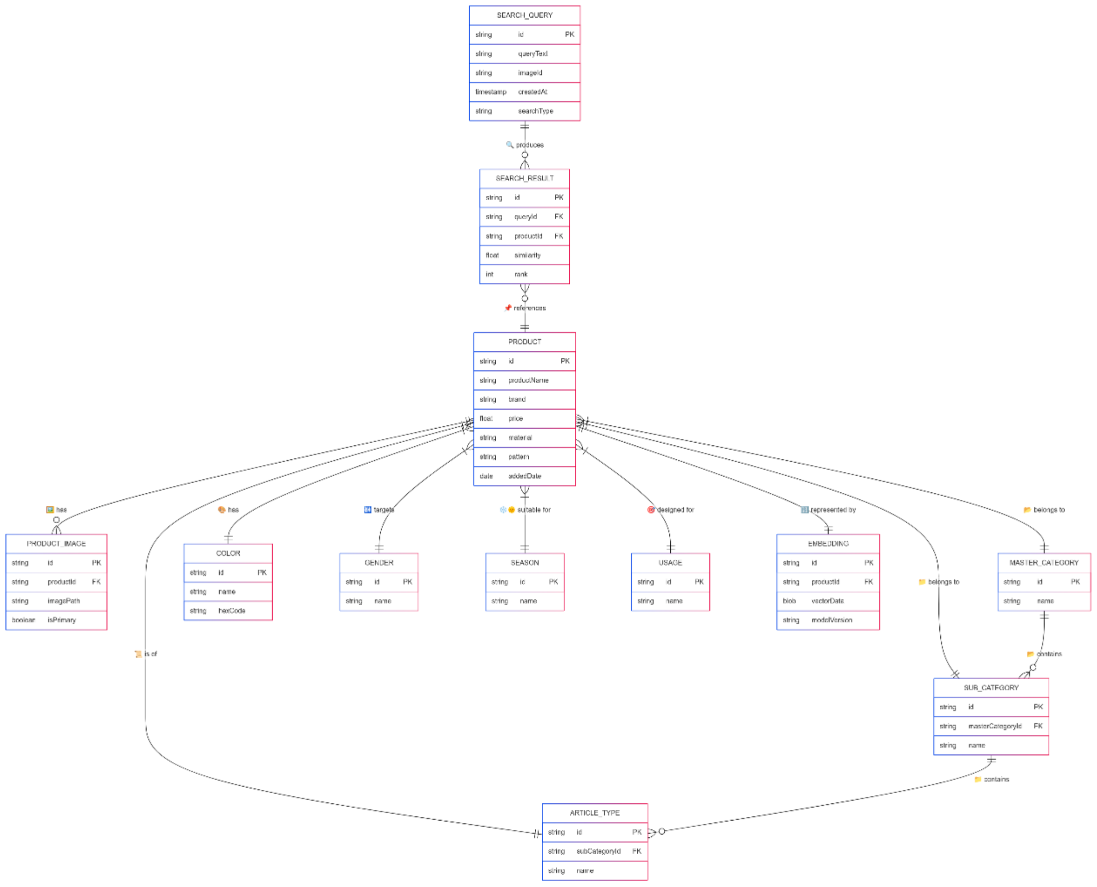


Fig. 5: ER Diagram

3.4. System Configuration

KAATCHI AI system configuration is tailored to optimize performance, scalability, and responsiveness of all parts of the system, including server infrastructure, frontend application, and indexing configuration. The configuration follows this structure:

Server Configuration:

KAATCHI AI backend is implemented with Node.js for asynchronous and efficient processing of user requests. The backend processes the query processing, retrieval of embedding, and interaction with the FAISS [2] indexing system. Server configurations are as follows:

- **Hosting Environment:** The backend server can be hosted on cloud platforms (AWS, Google Cloud, Azure) or on-premise servers for scalability considering the size of the dataset and the packages as platforms like github can only handle 5 GB data.
- **Processing Power:** An NVIDIA A100 or RTX 3090-class high-performance GPU is needed for query processing and embedding generation.
- **Memory & Storage:** At least 16GB of RAM and SSD storage to manage metadata, embeddings, and quick retrieval operations.
- **API Services:** OpenAI API is used for interaction between the frontend, retrieval system, and embedding models.

Frontend Configuration:

The KAATCHI AI user interface is built with React.js for a responsive, interactive, and fast search experience. The frontend setup is:

- **Framework:** Built upon React.js for dynamic rendering and state management.
- **Search Components:** Supports text, image, and hybrid queries with real-time search results.
- **Adaptive UI:** Supports dark mode, accessibility settings, and multi-device support for better user experience.

Indexing & Retrieval Configuration:

The retrieval and indexing system are driven by FAISS (Facebook AI Similarity Search) [2], providing scalable and efficient nearest-neighbor search. The indexing configuration is set as follows:

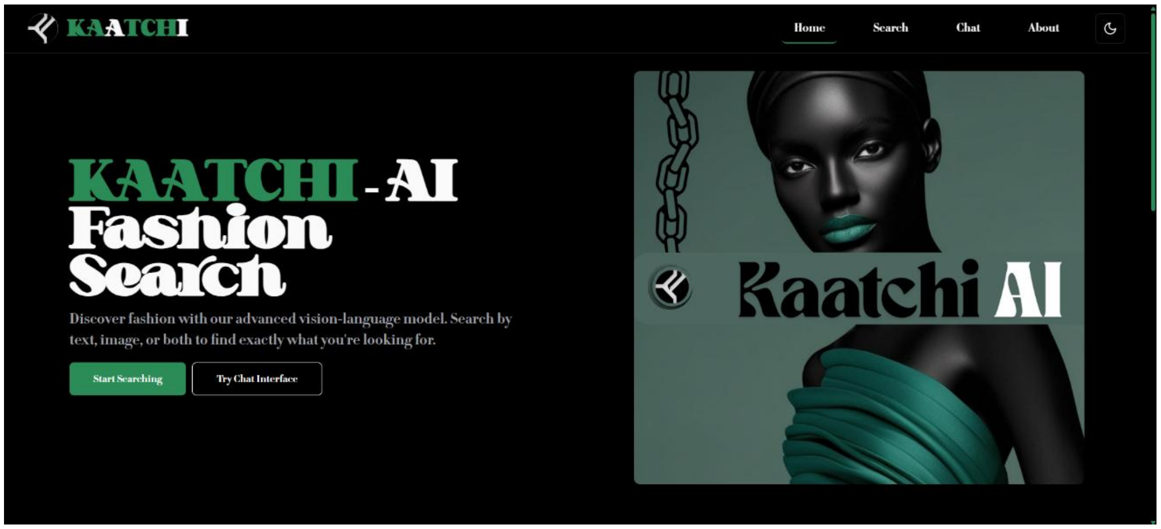
- **Embedding Storage:** Image and text embeddings are stored in an optimized search structure vector database.

- **Batch Processing:** Precomputed embeddings are bulk-indexed to reduce real-time computational complexity.
- **Latency Optimization:** Latency of response is minimized for query response with approximate nearest neighbor (ANN) search algorithms so that retrieval can be done within milliseconds.

Integrating all these settings, KAATCHI AI provides smooth, scalable, and high-speed visual search engine for accurate and responsive fashion product discovery.

3.5. Interface Design

3.5.1. User Interface Screen Design



Discover fashion with our advanced vision-language model. Search by text, image, or both to find exactly what you're looking for.

Fig.6: Home Page – 1

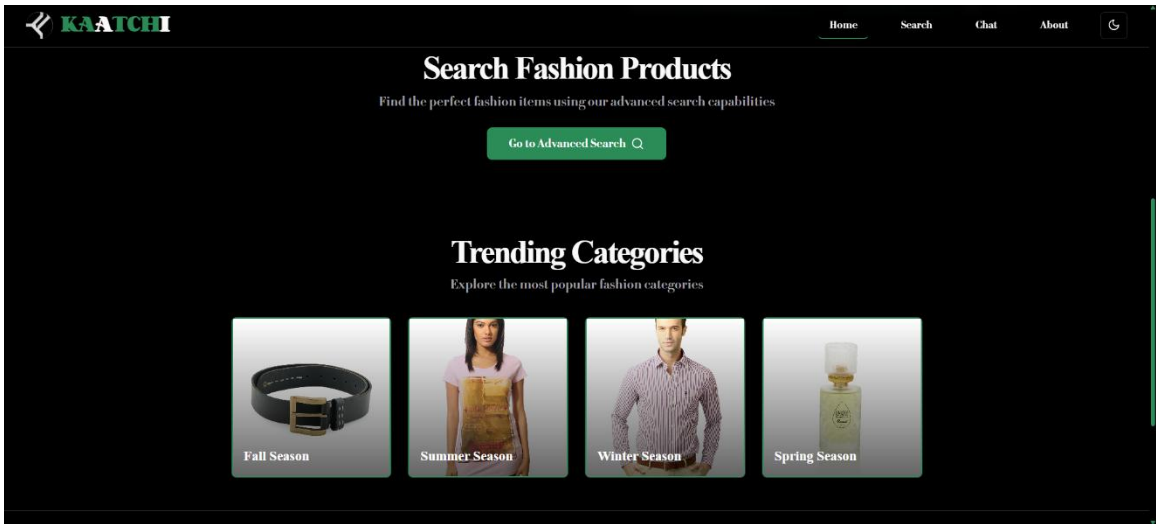


Fig.7: Home Page – 2

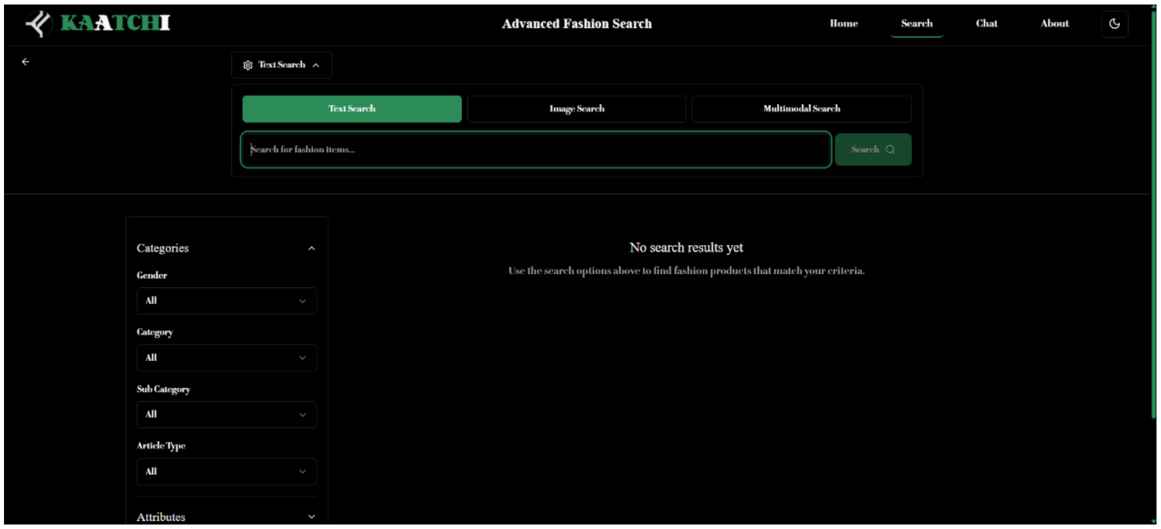
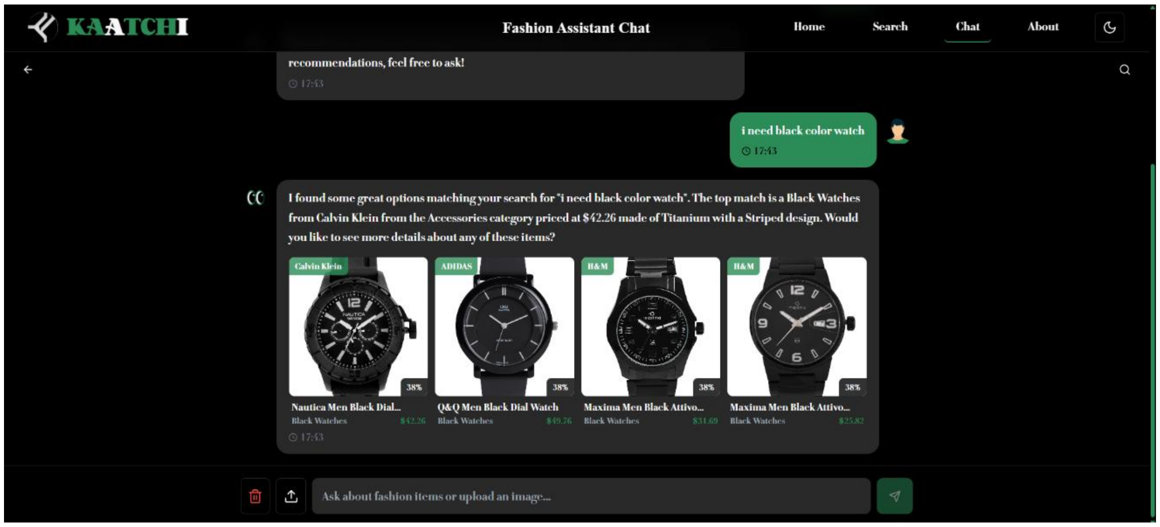


Fig.8: Search Page



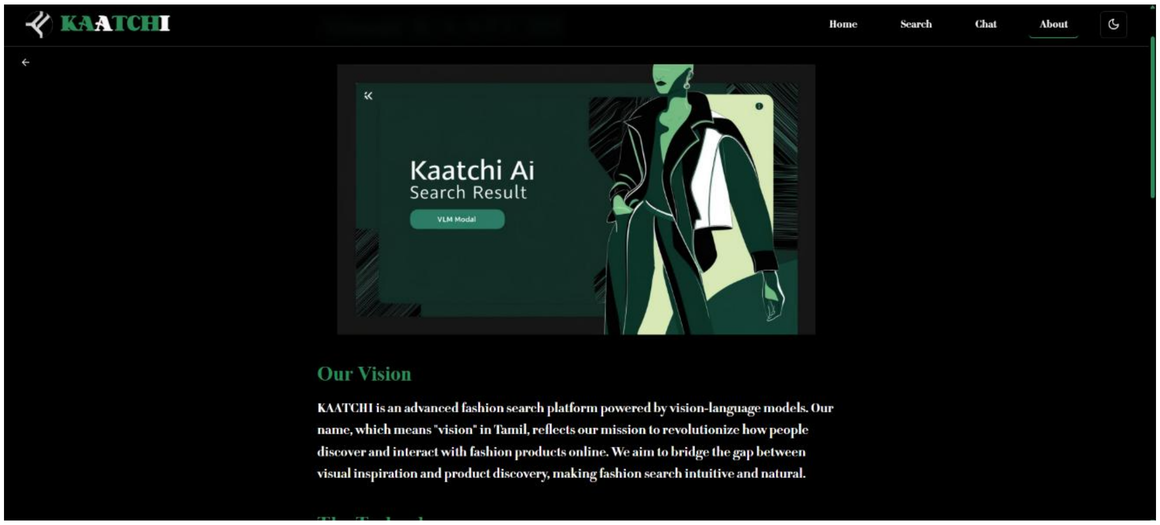
recommendations, feel free to ask!
17:43

i need black color watch
17:43

I found some great options matching your search for "i need black color watch". The top match is a Black Watches from Calvin Klein from the Accessories category priced at $42.26 made of Titanium with a Striped design. Would you like to see more details about any of these items?

Calvin Klein 38% Nautica Men Black Dial... Black Watches $42.26
ADIDAS 38% Q&Q Men Black Dial Watch Black Watches $49.76
H&M 38% Maxima Men Black Attivo... Black Watches $31.69
H&M 38% Maxima Men Black Attivo... Black Watches $25.82
17:43

Fig.9: Chat Page



Our Vision

KAATCHI is an advanced fashion search platform powered by vision-language models. Our name, which means "vision" in Tamil, reflects our mission to revolutionize how people discover and interact with fashion products online. We aim to bridge the gap between visual inspiration and product discovery, making fashion search intuitive and natural.

Fig.10: About Page

3.5.2. Application Flow/Class Diagram

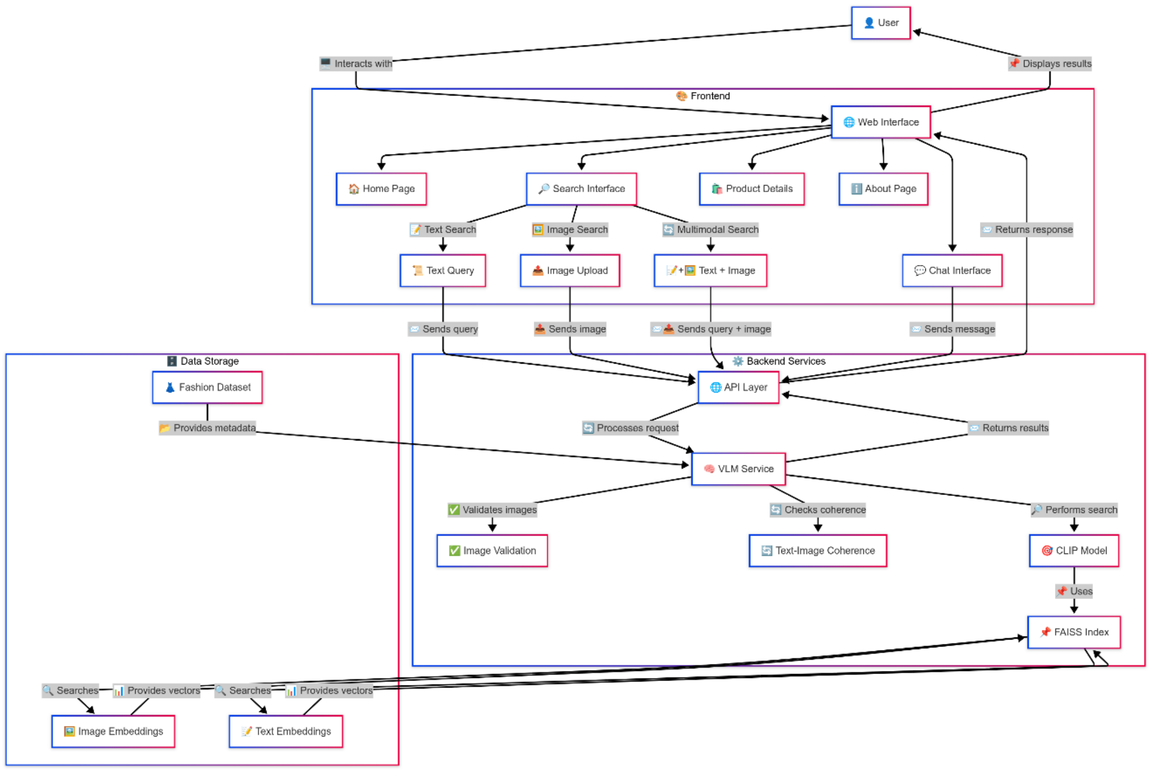


Fig.11: Application Flow Diagram

Fig. 11 depicts the design of KAATCHI AI, a fashion visual search engine, divided into three primary sections: Frontend, Backend Services, and Data Storage. Users operate on the web interface, such as a home page, search interface (text, image, and multimodal search), product details, and a chat interface for querying the dataset. The API Layer receives user queries and sends them to the VLM Service, which processes image validation, text-image consistency, and search using CLIP and FAISS Index for quick search. Product metadata is stored in the Fashion Dataset and is used to deliver image and text embeddings, upon which similarity searches are executed. Overall, the system provides an uninterrupted user experience through the utilization of vision-language models, effective indexing, and retrieval functions to perform fast and accurate fashion product searches.

3.6. Report Design

The reporting feature of KAATCHI AI is meant to analyze the effectiveness, efficiency, and precision of the visual search system. It provides detailed reports in terms of retrieval performance and system efficiency to ensure ongoing monitoring and optimization.

Retrieval Performance Reports:

These reports measure the quality of search results based on industry-standard evaluation criteria:

- **Precision@K:** It computes the percentage of retrieved images that are relevant among the top K results.
- **Recall@K:** Expects the number of correct relevant images retrieved in the top K results.
- **Mean Average Precision (mAP):** Presents a weighted average measure of ranking performance over multiple queries.
- **Query Success Rate:** Examines the proportion of successful queries returning significant results, used to pinpoint areas of improvement.

System Performance Reports:

We highlight the performance and scalability of KAATCHI AI through monitoring:

- **Indexing Speed:** Tracks how long it takes to create and store embeddings within FAISS.
- **Query Latency:** Measures the system's response time for text, image, and multi-modal queries.
- **Search Throughput:** Monitors how many queries are processed per second to enable real-time searching.
- **System Load Analysis:** Keeps tabs on resource consumption (CPU, GPU, memory) to help optimize deployment.

Through the production of these reports, KAATCHI AI facilitates ongoing performance assessment, allowing for improvement in retrieval accuracy, search effectiveness, and user experience.

4. IMPLEMENTATION

4.1. Coding Standard

To provide readability, maintainability, and efficiency, KAATCHI AI follows established coding standards in various parts of the system. The coding architecture is based on best practices for Python (backend and ML processing) and JavaScript (frontend and API development), ensuring modularity and scalability.

Python (Backend & Machine Learning Processing) – PEP 8 Compliance:

The KAATCHI AI backend that implements generation, indexing, and retrieval adheres to PEP 8, the Python official style guide. This provides:

- Uniform formatting (correct indentation, spacing, and linebreaks).
- Readability and organization of code with descriptive variable and function names.
- Error handling and exception management to avoid surprises.
- Docstrings and comments for code readability and debugging simplicity.

JavaScript (React-Node.js UI) – ESLint Compliance:

The frontend, implemented using React (JavaScript) and Node.js, has ESLint coding standards to ensure high-quality, error-free code. These include:

- Uniform syntax and indentation for improved readability.
- Strict type checking to minimize runtime errors and ensure stability.
- Component-based architecture in React for optimal UI development.
- Asynchronous handling in Node.js for non-blocking API calls.

Modular Code Structure:

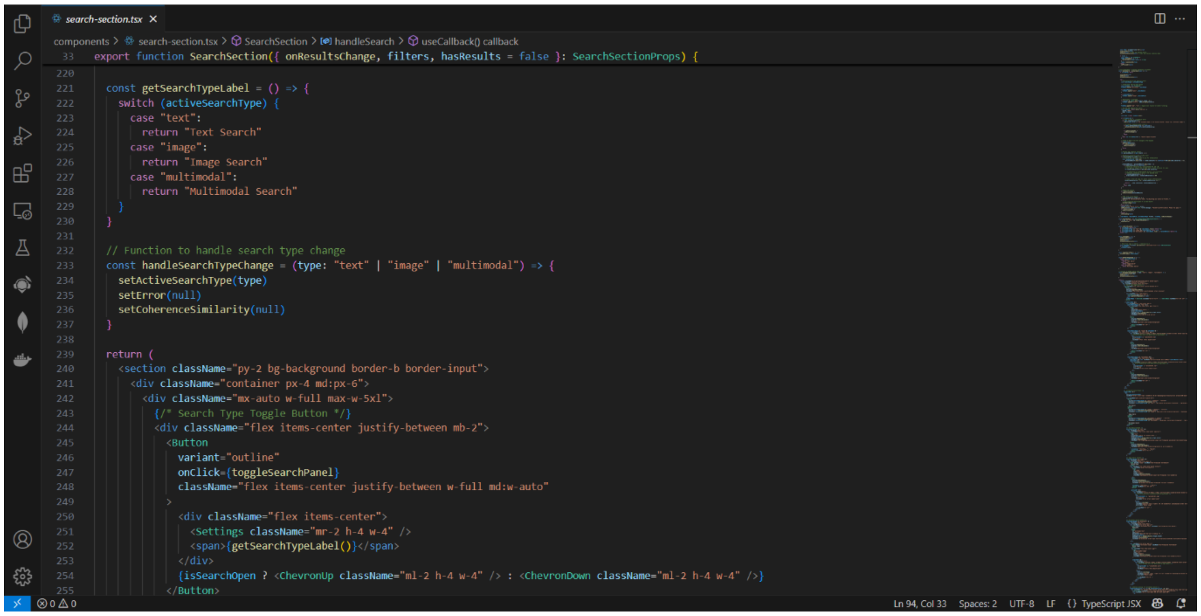
KAATCHI AI is based on a modular architecture, where various functionalities are realized in independent modules, thus making the system scalable, maintainable, and reusable. The most important modules are:

- **Data Preprocessing Module:** Responsible for metadata cleaning and transformation.

- **Embedding Generation Module:** Transforms text and image data into embeddings by utilizing CLIP [3].
- **Indexing and Retrieval Module:** Leverages FAISS for similarity search efficiency.
- **User Interface Module:** Realizes the React-Node.js based frontend, providing smooth interaction.

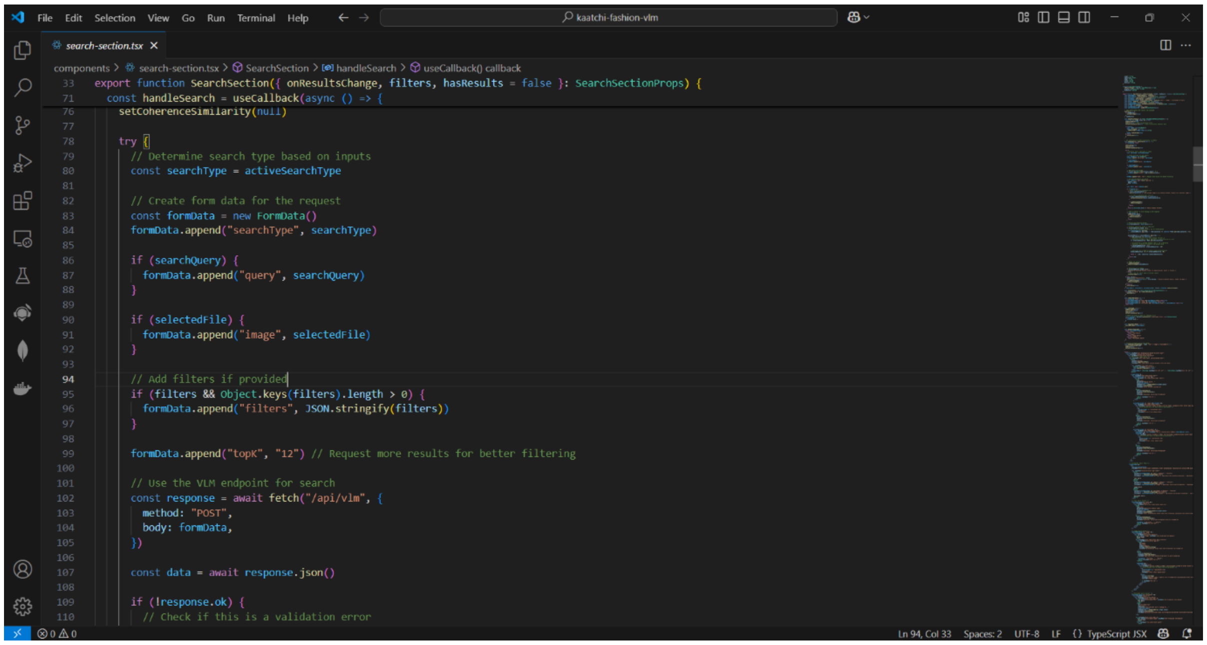
By adhering to these coding guidelines, KAATCHI AI has high-quality code, provides smooth integration among components, and maximizes system performance while enabling future development and scalability.

4.2. Screenshots


search-section.tsx
components > search-section.tsx > SearchSection > handleSearch > useCallback() callback
Ln 94, Col 33 Spaces: 2 UTF-8 LF {} TypeScript JSX

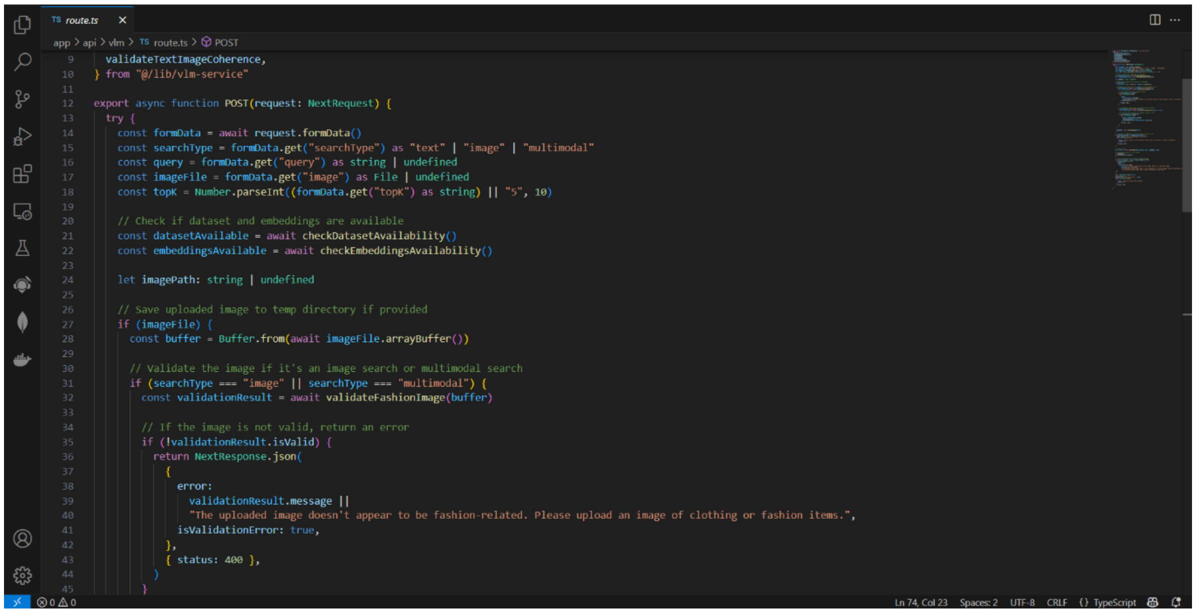
```tsx
export function SearchSection({ onResultsChange, filters, hasResults = false }: SearchSectionProps) {

  const getSearchTypeLabel = () => {
    switch (activeSearchType) {
      case "text":
        return "Text Search"
      case "image":
        return "Image Search"
      case "multimodal":
        return "Multimodal Search"
    }
  }

  // Function to handle search type change
  const handleSearchTypeChange = (type: "text" | "image" | "multimodal") => {
    setActiveSearchType(type)
    setError(null)
    setCoherenceSimilarity(null)
  }

  return (
    <section className="py-2 bg-background border-b border-input">
      <div className="container px-4 md:px-6">
        <div className="mx-auto w-full max-w-5xl">
          {/* Search Type Toggle Button */}
          <div className="flex items-center justify-between mb-2">
            <Button
              variant="outline"
              onClick={toggleSearchPanel}
              className="flex items-center justify-between w-full md:w-auto"
            >
              <div className="flex items-center">
                <Settings className="mr-2 h-4 w-4" />
                <span>{getSearchTypeLabel()}</span>
              </div>
              {isSearchOpen ? <ChevronUp className="ml-2 h-4 w-4" /> : <ChevronDown className="ml-2 h-4 w-4" />}
            </Button>
```

Fig.12: Search – UI/UX – 1


File Edit Selection View Go Run Terminal Help
kaatchi-fashion-vlm
search-section.tsx
components > search-section.tsx > SearchSection > handleSearch > useCallback() callback
Ln 94, Col 33 Spaces: 2 UTF-8 LF {} TypeScript JSX

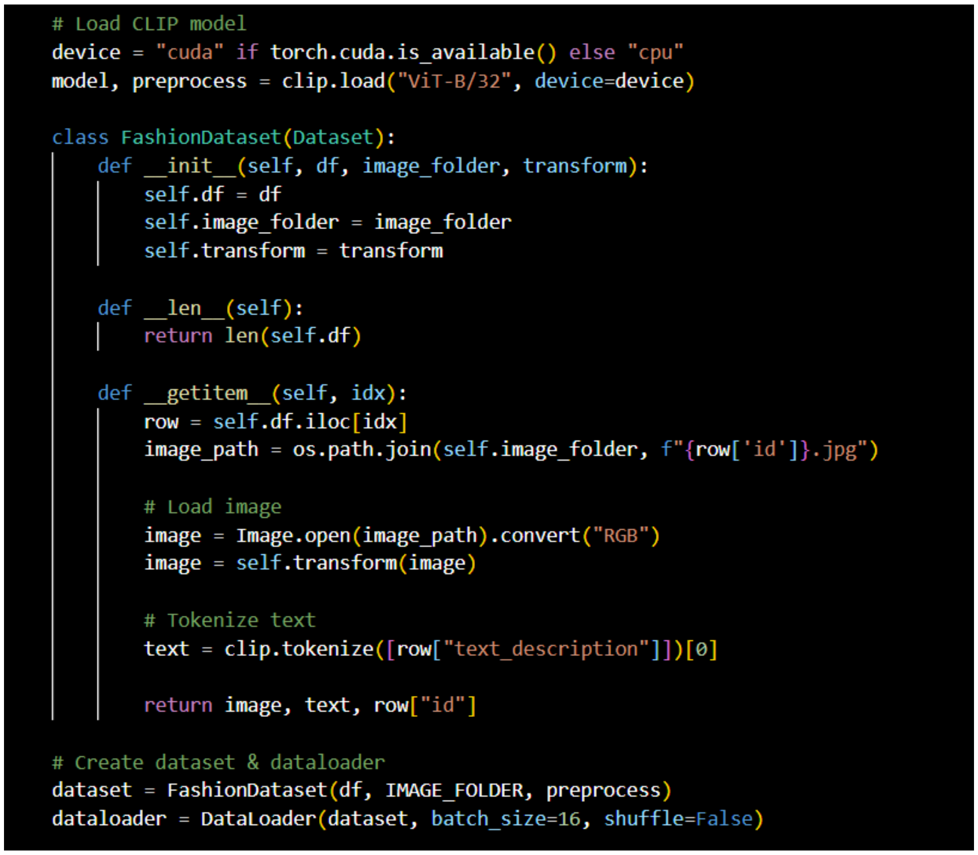
```
export function SearchSection({ onResultsChange, filters, hasResults = false }: SearchSectionProps) {
  const handleSearch = useCallback(async () => {
    setCoherenceSimilarity(null)

    try {
      // Determine search type based on inputs
      const searchType = activeSearchType

      // Create form data for the request
      const formData = new FormData()
      formData.append("searchType", searchType)

      if (searchQuery) {
        formData.append("query", searchQuery)
      }

      if (selectedFile) {
        formData.append("image", selectedFile)
      }

      // Add filters if provided
      if (filters && Object.keys(filters).length > 0) {
        formData.append("filters", JSON.stringify(filters))
      }

      formData.append("topK", "12") // Request more results for better filtering

      // Use the VLM endpoint for search
      const response = await fetch("/api/vlm", {
        method: "POST",
        body: formData,
      })

      const data = await response.json()

      if (!response.ok) {
        // Check if this is a validation error
```

Fig.13: Search – UI/UX – 2


route.ts
app > api > vlm > route.ts > POST
Ln 74, Col 23 Spaces: 2 UTF-8 CRLF {} TypeScript

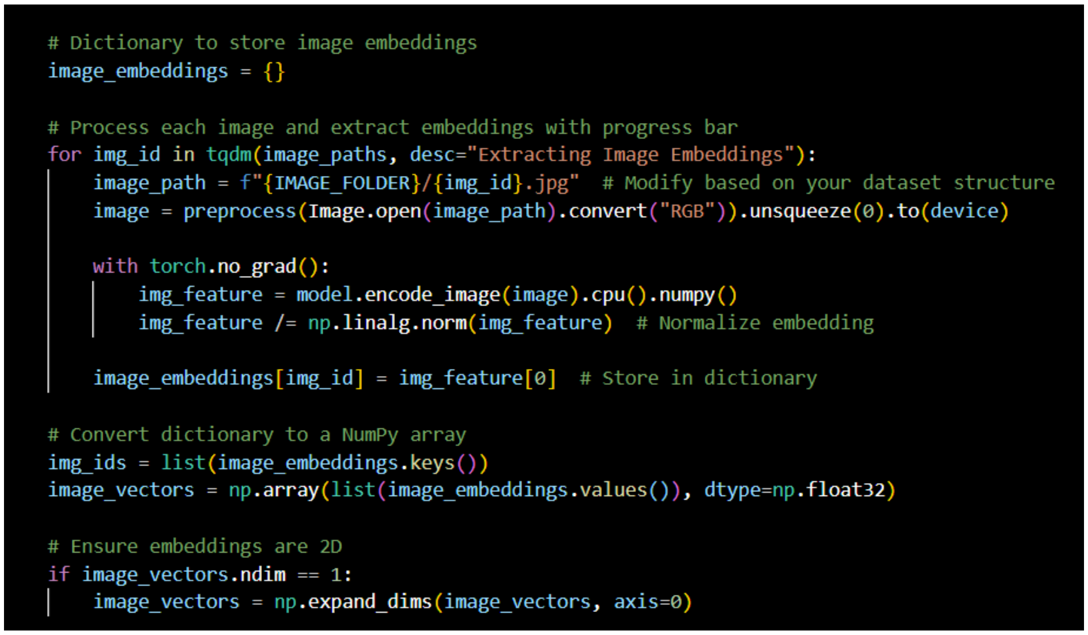
```
  validateTextImageCoherence,
} from "@/lib/vlm-service"

export async function POST(request: NextRequest) {
  try {
    const formData = await request.formData()
    const searchType = formData.get("searchType") as "text" | "image" | "multimodal"
    const query = formData.get("query") as string | undefined
    const imageFile = formData.get("image") as File | undefined
    const topK = Number.parseInt((formData.get("topK") as string) || "5", 10)

    // Check if dataset and embeddings are available
    const datasetAvailable = await checkDatasetAvailability()
    const embeddingsAvailable = await checkEmbeddingsAvailability()

    let imagePath: string | undefined

    // Save uploaded image to temp directory if provided
    if (imageFile) {
      const buffer = Buffer.from(await imageFile.arrayBuffer())

      // Validate the image if it's an image search or multimodal search
      if (searchType === "image" || searchType === "multimodal") {
        const validationResult = await validateFashionImage(buffer)

        // If the image is not valid, return an error
        if (!validationResult.isValid) {
          return NextResponse.json(
            {
              error:
                validationResult.message ||
                "The uploaded image doesn't appear to be fashion-related. Please upload an image of clothing or fashion items.",
              isValidationError: true,
            },
            { status: 400 },
          )
        }
```

Fig.14: VLM Calling



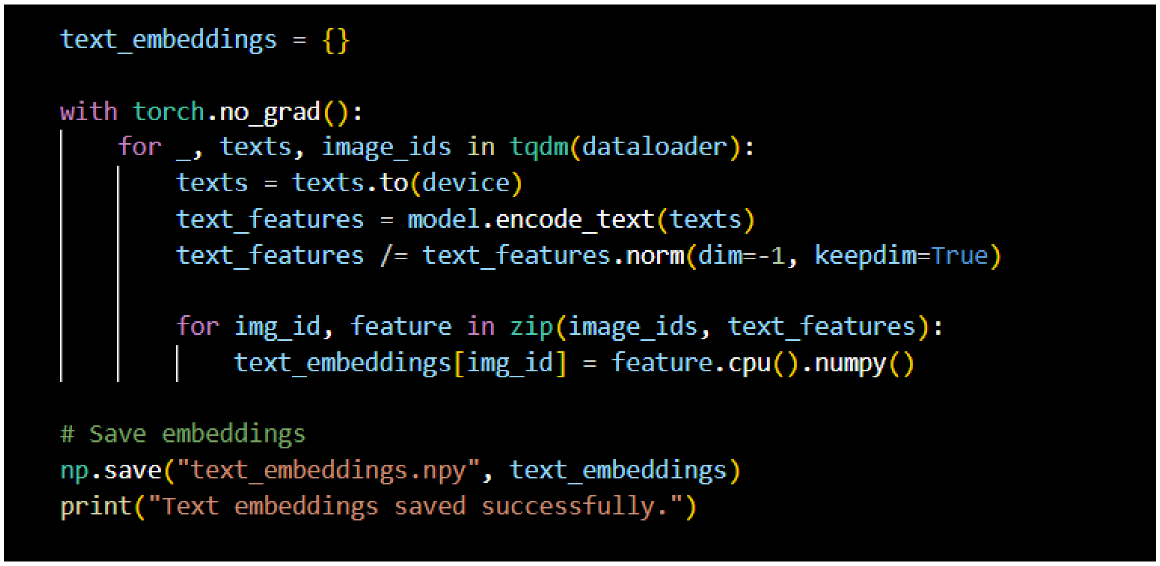
```
# Load CLIP model
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)

class FashionDataset(Dataset):
    def __init__(self, df, image_folder, transform):
        self.df = df
        self.image_folder = image_folder
        self.transform = transform

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        image_path = os.path.join(self.image_folder, f"{row['id']}.jpg")

        # Load image
        image = Image.open(image_path).convert("RGB")
        image = self.transform(image)

        # Tokenize text
        text = clip.tokenize([row["text_description"]])[0]

        return image, text, row["id"]

# Create dataset & dataloader
dataset = FashionDataset(df, IMAGE_FOLDER, preprocess)
dataloader = DataLoader(dataset, batch_size=16, shuffle=False)
```

Fig.15: CLIP Model Loading



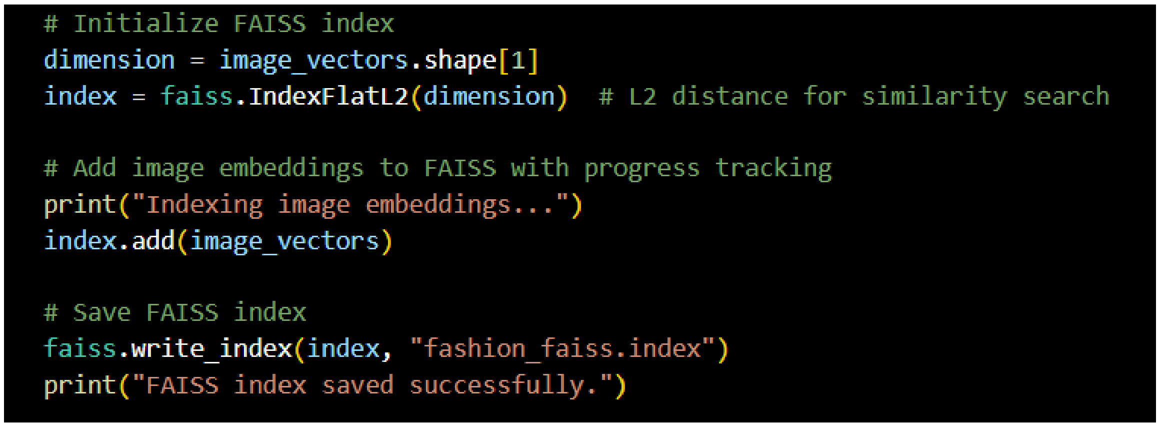
```
# Dictionary to store image embeddings
image_embeddings = {}

# Process each image and extract embeddings with progress bar
for img_id in tqdm(image_paths, desc="Extracting Image Embeddings"):
    image_path = f"{IMAGE_FOLDER}/{img_id}.jpg"  # Modify based on your dataset structure
    image = preprocess(Image.open(image_path).convert("RGB")).unsqueeze(0).to(device)

    with torch.no_grad():
        img_feature = model.encode_image(image).cpu().numpy()
        img_feature /= np.linalg.norm(img_feature)  # Normalize embedding

    image_embeddings[img_id] = img_feature[0]  # Store in dictionary

# Convert dictionary to a NumPy array
img_ids = list(image_embeddings.keys())
image_vectors = np.array(list(image_embeddings.values()), dtype=np.float32)

# Ensure embeddings are 2D
if image_vectors.ndim == 1:
    image_vectors = np.expand_dims(image_vectors, axis=0)
```

Fig.16: Image Embeddings



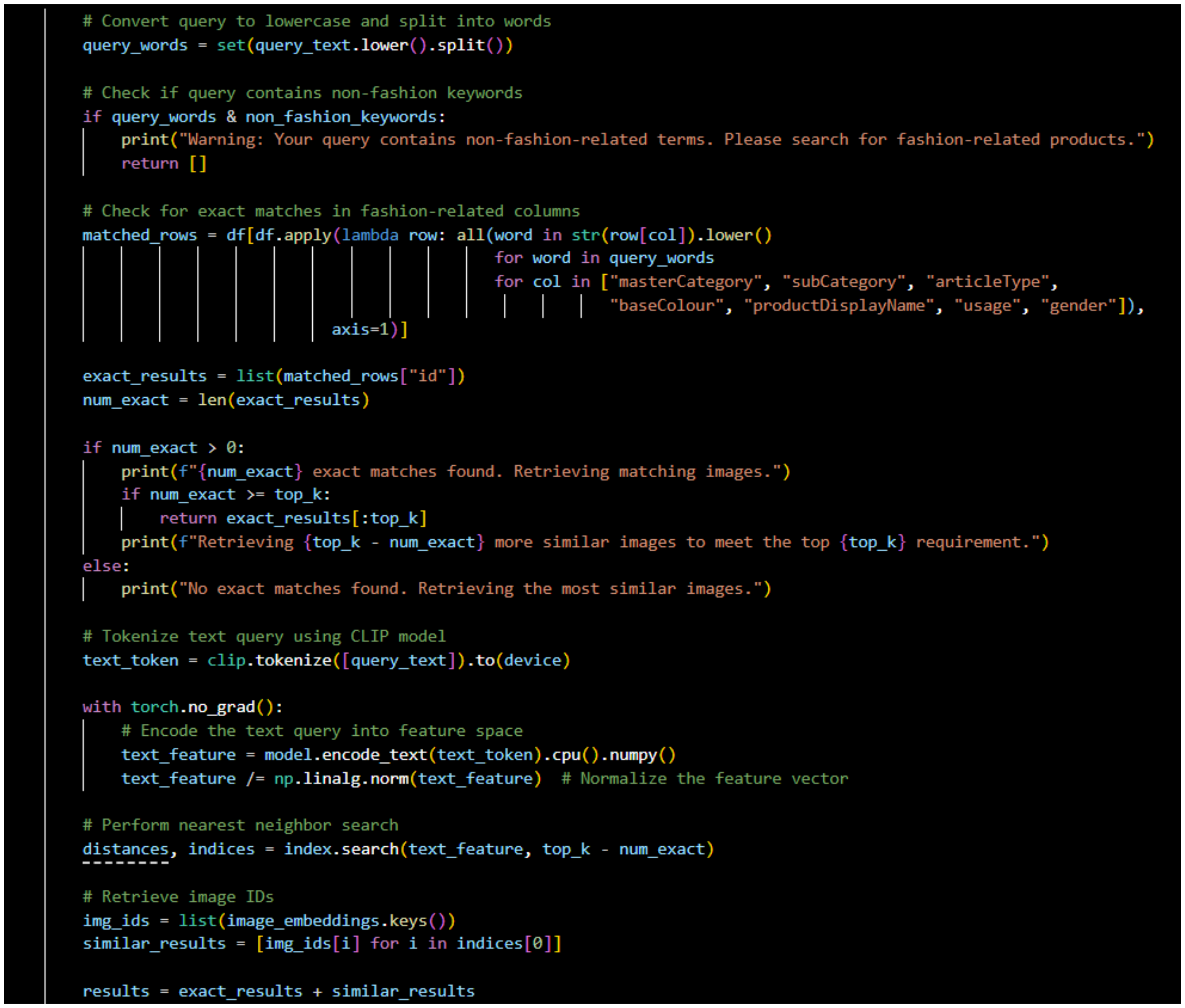
```
text_embeddings = {}

with torch.no_grad():
    for _, texts, image_ids in tqdm(dataloader):
        texts = texts.to(device)
        text_features = model.encode_text(texts)
        text_features /= text_features.norm(dim=-1, keepdim=True)

        for img_id, feature in zip(image_ids, text_features):
            text_embeddings[img_id] = feature.cpu().numpy()

# Save embeddings
np.save("text_embeddings.npy", text_embeddings)
print("Text embeddings saved successfully.")
```

Fig.17: Text Embeddings



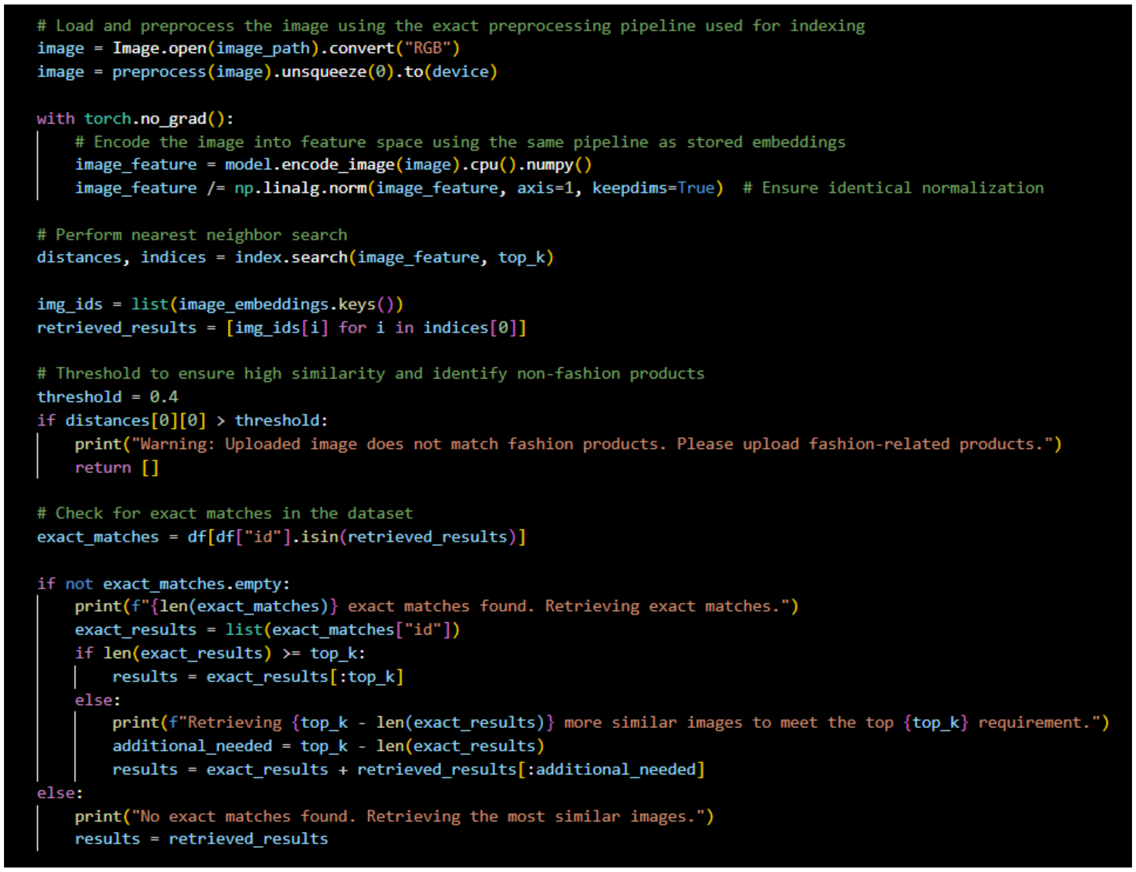
```
# Initialize FAISS index
dimension = image_vectors.shape[1]
index = faiss.IndexFlatL2(dimension)  # L2 distance for similarity search

# Add image embeddings to FAISS with progress tracking
print("Indexing image embeddings...")
index.add(image_vectors)

# Save FAISS index
faiss.write_index(index, "fashion_faiss.index")
print("FAISS index saved successfully.")
```

Fig.18: FAISS Initialization



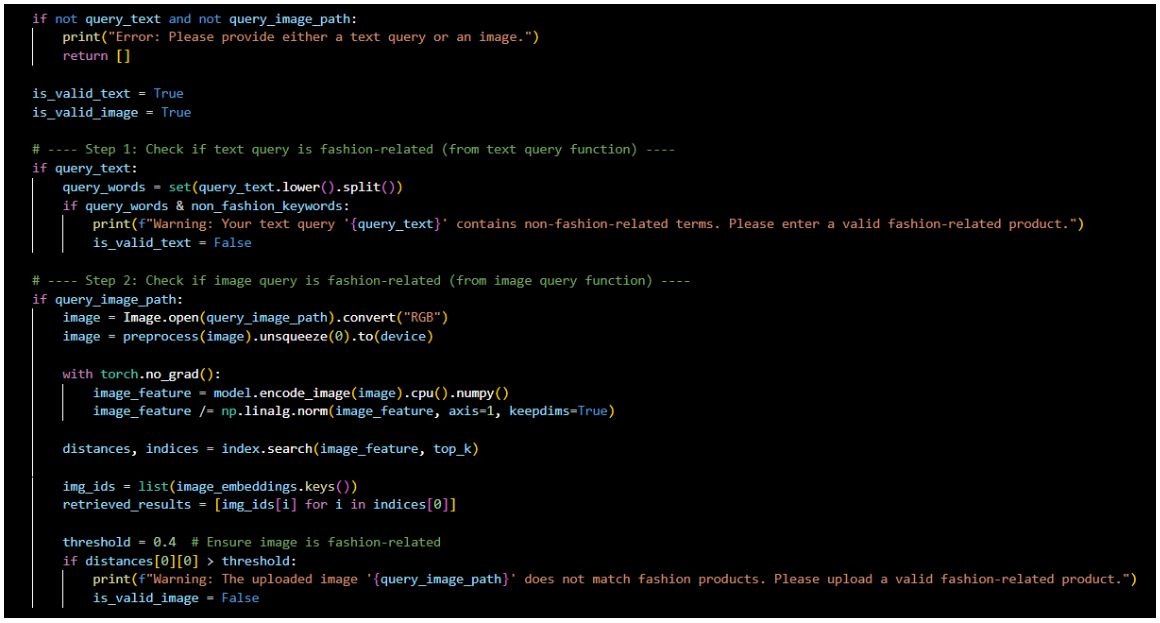
```
    # Convert query to lowercase and split into words
    query_words = set(query_text.lower().split())

    # Check if query contains non-fashion keywords
    if query_words & non_fashion_keywords:
        print("Warning: Your query contains non-fashion-related terms. Please search for fashion-related products.")
        return []

    # Check for exact matches in fashion-related columns
    matched_rows = df[df.apply(lambda row: all(word in str(row[col]).lower()
                                              for word in query_words
                                              for col in ["masterCategory", "subCategory", "articleType",
                                                          "baseColour", "productDisplayName", "usage", "gender"]),
                              axis=1)]

    exact_results = list(matched_rows["id"])
    num_exact = len(exact_results)

    if num_exact > 0:
        print(f"{num_exact} exact matches found. Retrieving matching images.")
        if num_exact >= top_k:
            return exact_results[:top_k]
        print(f"Retrieving {top_k - num_exact} more similar images to meet the top {top_k} requirement.")
    else:
        print("No exact matches found. Retrieving the most similar images.")

    # Tokenize text query using CLIP model
    text_token = clip.tokenize([query_text]).to(device)

    with torch.no_grad():
        # Encode the text query into feature space
        text_feature = model.encode_text(text_token).cpu().numpy()
        text_feature /= np.linalg.norm(text_feature)  # Normalize the feature vector

    # Perform nearest neighbor search
    distances, indices = index.search(text_feature, top_k - num_exact)

    # Retrieve image IDs
    img_ids = list(image_embeddings.keys())
    similar_results = [img_ids[i] for i in indices[0]]

    results = exact_results + similar_results
```

Fig.19: Text Query



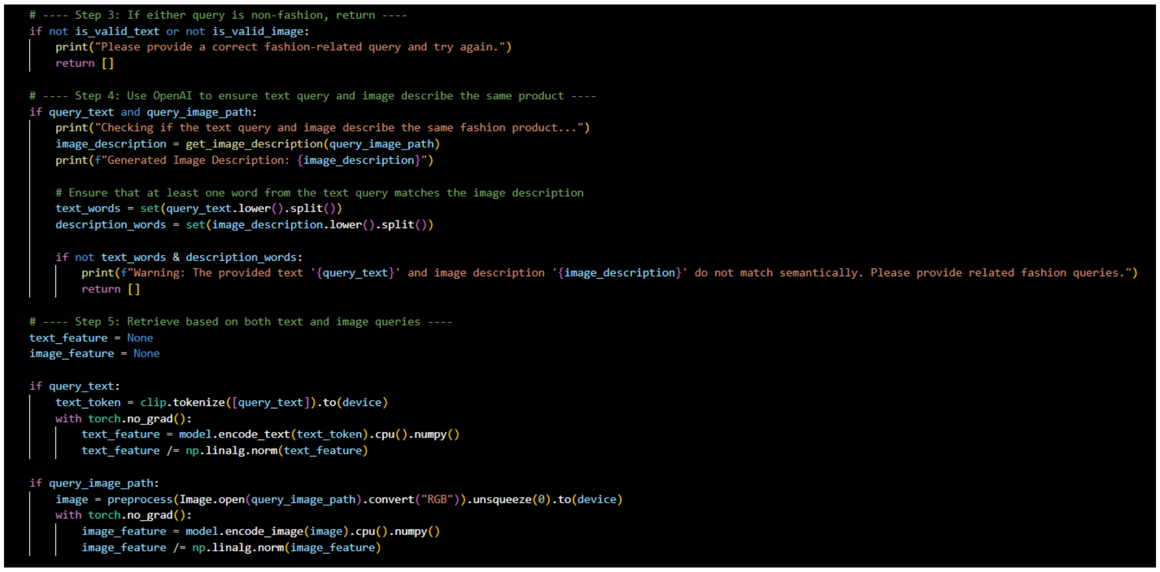
```python
# Load and preprocess the image using the exact preprocessing pipeline used for indexing
image = Image.open(image_path).convert("RGB")
image = preprocess(image).unsqueeze(0).to(device)

with torch.no_grad():
    # Encode the image into feature space using the same pipeline as stored embeddings
    image_feature = model.encode_image(image).cpu().numpy()
    image_feature /= np.linalg.norm(image_feature, axis=1, keepdims=True)  # Ensure identical normalization

# Perform nearest neighbor search
distances, indices = index.search(image_feature, top_k)

img_ids = list(image_embeddings.keys())
retrieved_results = [img_ids[i] for i in indices[0]]

# Threshold to ensure high similarity and identify non-fashion products
threshold = 0.4
if distances[0][0] > threshold:
    print("Warning: Uploaded image does not match fashion products. Please upload fashion-related products.")
    return []

# Check for exact matches in the dataset
exact_matches = df[df["id"].isin(retrieved_results)]

if not exact_matches.empty:
    print(f"{len(exact_matches)} exact matches found. Retrieving exact matches.")
    exact_results = list(exact_matches["id"])
    if len(exact_results) >= top_k:
        results = exact_results[:top_k]
    else:
        print(f"Retrieving {top_k - len(exact_results)} more similar images to meet the top {top_k} requirement.")
        additional_needed = top_k - len(exact_results)
        results = exact_results + retrieved_results[:additional_needed]
else:
    print("No exact matches found. Retrieving the most similar images.")
    results = retrieved_results
```

Fig.20: Image Query



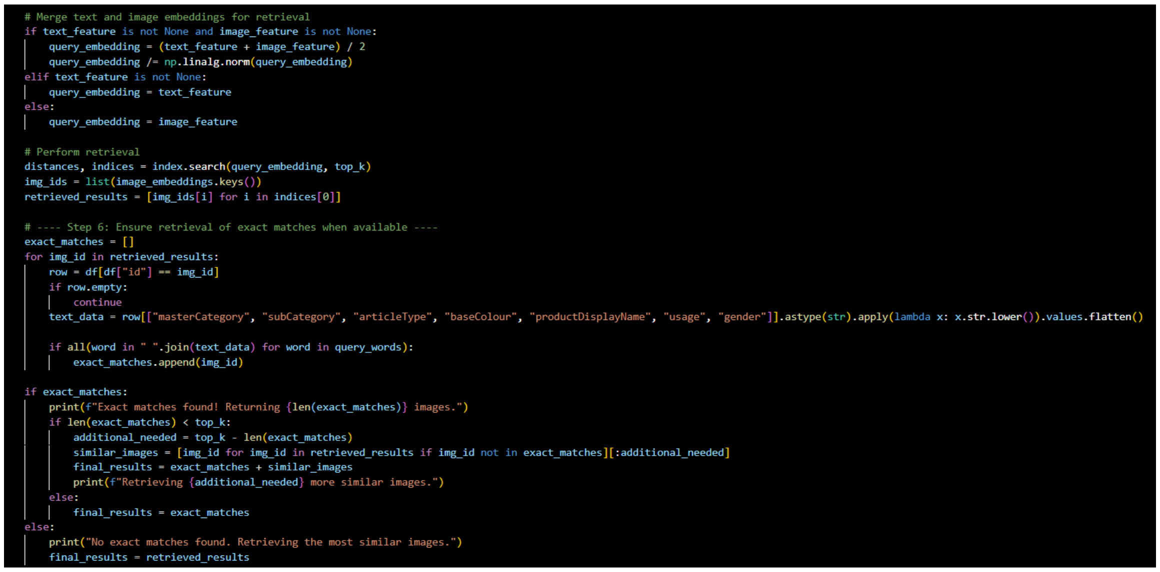
```python
if not query_text and not query_image_path:
    print("Error: Please provide either a text query or an image.")
    return []

is_valid_text = True
is_valid_image = True

# ---- Step 1: Check if text query is fashion-related (from text query function) ----
if query_text:
    query_words = set(query_text.lower().split())
    if query_words & non_fashion_keywords:
        print(f"Warning: Your text query '{query_text}' contains non-fashion-related terms. Please enter a valid fashion-related product.")
        is_valid_text = False

# ---- Step 2: Check if image query is fashion-related (from image query function) ----
if query_image_path:
    image = Image.open(query_image_path).convert("RGB")
    image = preprocess(image).unsqueeze(0).to(device)

    with torch.no_grad():
        image_feature = model.encode_image(image).cpu().numpy()
        image_feature /= np.linalg.norm(image_feature, axis=1, keepdims=True)

    distances, indices = index.search(image_feature, top_k)

    img_ids = list(image_embeddings.keys())
    retrieved_results = [img_ids[i] for i in indices[0]]

    threshold = 0.4  # Ensure image is fashion-related
    if distances[0][0] > threshold:
        print(f"Warning: The uploaded image '{query_image_path}' does not match fashion products. Please upload a valid fashion-related product.")
        is_valid_image = False
```

Fig. 21: Multimodal Query – 1



```python
# ---- Step 3: If either query is non-fashion, return ----
if not is_valid_text or not is_valid_image:
    print("Please provide a correct fashion-related query and try again.")
    return []

# ---- Step 4: Use OpenAI to ensure text query and image describe the same product ----
if query_text and query_image_path:
    print("Checking if the text query and image describe the same fashion product...")
    image_description = get_image_description(query_image_path)
    print(f"Generated Image Description: {image_description}")

    # Ensure that at least one word from the text query matches the image description
    text_words = set(query_text.lower().split())
    description_words = set(image_description.lower().split())

    if not text_words & description_words:
        print(f"Warning: The provided text '{query_text}' and image description '{image_description}' do not match semantically. Please provide related fashion queries.")
        return []

# ---- Step 5: Retrieve based on both text and image queries ----
text_feature = None
image_feature = None

if query_text:
    text_token = clip.tokenize([query_text]).to(device)
    with torch.no_grad():
        text_feature = model.encode_text(text_token).cpu().numpy()
        text_feature /= np.linalg.norm(text_feature)

if query_image_path:
    image = preprocess(Image.open(query_image_path).convert("RGB")).unsqueeze(0).to(device)
    with torch.no_grad():
        image_feature = model.encode_image(image).cpu().numpy()
        image_feature /= np.linalg.norm(image_feature)
```

Fig.22: Multimodal Query – 2



```python
# Merge text and image embeddings for retrieval
if text_feature is not None and image_feature is not None:
    query_embedding = (text_feature + image_feature) / 2
    query_embedding /= np.linalg.norm(query_embedding)
elif text_feature is not None:
    query_embedding = text_feature
else:
    query_embedding = image_feature

# Perform retrieval
distances, indices = index.search(query_embedding, top_k)
img_ids = list(image_embeddings.keys())
retrieved_results = [img_ids[i] for i in indices[0]]

# ---- Step 6: Ensure retrieval of exact matches when available ----
exact_matches = []
for img_id in retrieved_results:
    row = df[df["id"] == img_id]
    if row.empty:
        continue
    text_data = row[["masterCategory", "subCategory", "articleType", "baseColour", "productDisplayName", "usage", "gender"]].astype(str).apply(lambda x: x.str.lower()).values.flatten()

    if all(word in " ".join(text_data) for word in query_words):
        exact_matches.append(img_id)

if exact_matches:
    print(f"Exact matches found! Returning {len(exact_matches)} images.")
    if len(exact_matches) < top_k:
        additional_needed = top_k - len(exact_matches)
        similar_images = [img_id for img_id in retrieved_results if img_id not in exact_matches][:additional_needed]
        final_results = exact_matches + similar_images
        print(f"Retrieving {additional_needed} more similar images.")
    else:
        final_results = exact_matches
else:
    print("No exact matches found. Retrieving the most similar images.")
    final_results = retrieved_results
```

Fig. 23: Multimodal Query – 3

5. TESTING

5.1. Tests Cases

Testing in KAATCHI AI was conducted to evaluate the accuracy, efficiency, and robustness of the visual search engine. The system was tested across various query types, ensuring that it retrieves the most relevant results while maintaining low latency and high precision.

Test Case Categories:

1. **Text Query Tests**
   - **Objective:** Verify that KAATCHI AI correctly retrieves fashion products based on textual descriptions.
   - **Test Case 1:** Input a generic query (e.g., "red sneakers").
   - **Test Case 2:** Input a detailed query (e.g., "women's leather handbag in black color").
   - **Expected Outcome:** The system retrieves images matching the provided textual description with high accuracy.
2. **Image Query Tests**
   - **Objective:** Ensure that the system retrieves visually similar products based on a given image.
   - **Test Case 1:** Upload an image of a red t shirt and verify if similar t shirts appear.
   - **Test Case 2:** Test with a different angle or slightly modified version of the same product.
   - **Expected Outcome:** The retrieved images should resemble the query image in style, color, and category.
3. **Multi-Modal Query Tests (Text + Image)**
   - **Objective:** Evaluate the system's ability to refine search results by combining both text and image inputs.
   - **Test Case 1:** Provide an image of a black dress and add the text "black dress with floral prints".
   - **Test Case 2:** Upload an image of white sneakers with the text "White Shoes with red stripes".

- **Test Case 3:** Check the validation for the multimodal query, for example if given with fashion image as well as fashion text query, model should validate of they both are describing the same product.
- **Expected Outcome:** The search engine retrieves products that closely match the combination of both inputs.

4. **Non-Fashion Query Tests**
   - **Objective:** Verify that KAATCHI AI correctly identifies and filters out non-fashion products.
   - **Test Case 1 – Text Query:** Search for unrelated items like "Table" or "car accessories".
   - **Test Case 2 – Image Query:** Input the unrelated images like “Burger” or “Broom”.
   - **Test Case 3 – Multimodal Query:** Validate the text and image for non-fashion products for the multimodal query as well
   - **Expected Outcome:** The system should not retrieve irrelevant results but instead return a message indicating that the input is outside the fashion category.

5.2. Tests Reports

1. **Text Query Tests**
   - **Test Case 1: Red Sneakers**

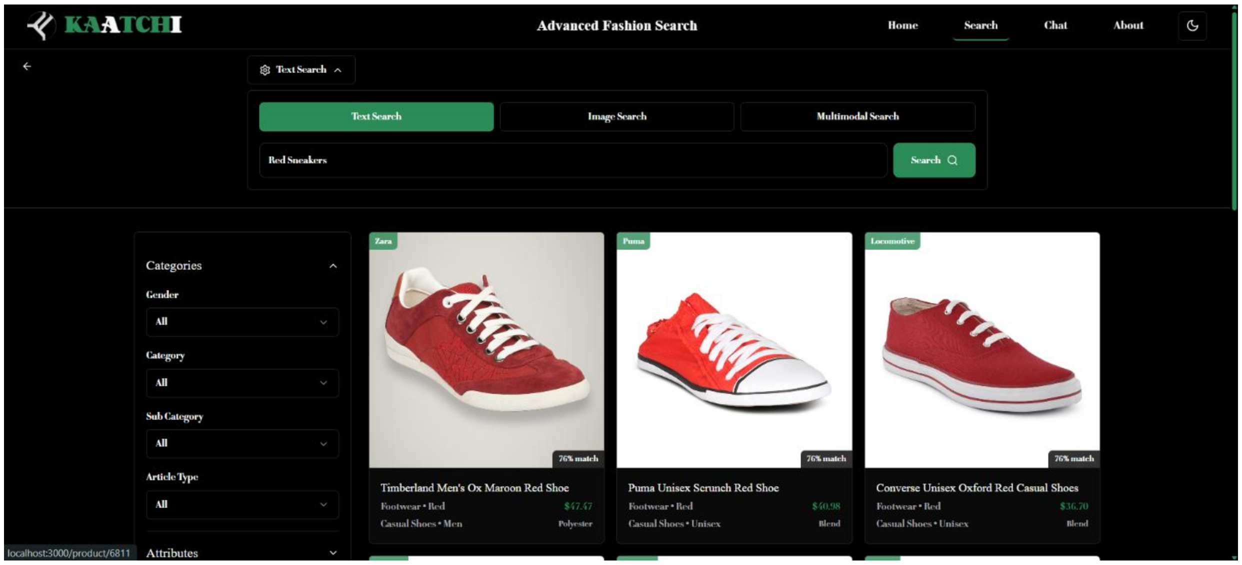
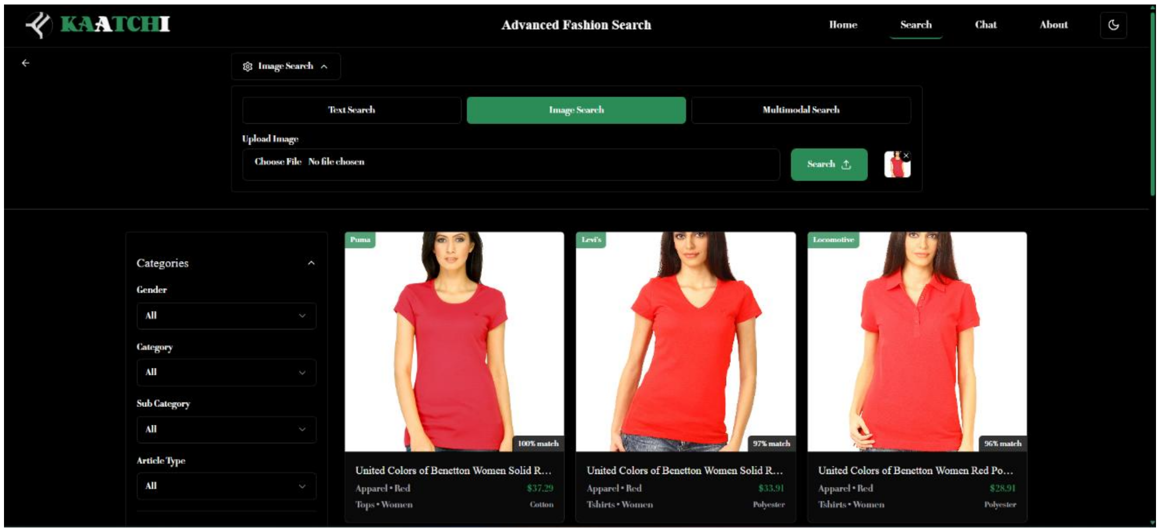


Fig.24: Text Query – Case 1: Red Sneakers

2. Image Query Tests

- **Test Case 1: Red T Shirt**



Upload Image
Choose File No file chosen

Gender: All
Category: All
Sub Category: All
Article Type: All

100% match
United Colors of Benetton Women Solid R…
Apparel • Red $37.29
Tops • Women Cotton

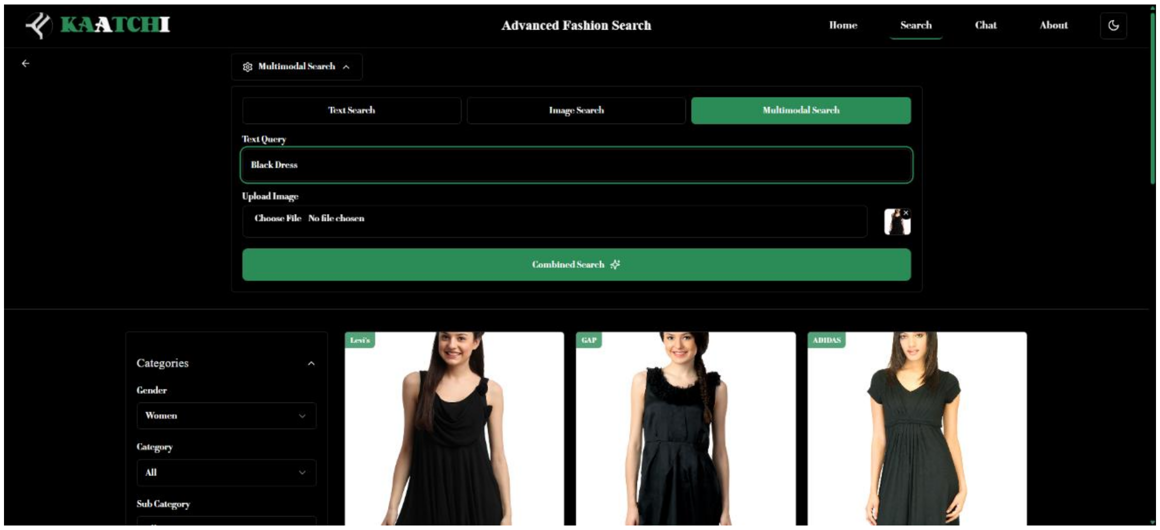
97% match
United Colors of Benetton Women Solid R…
Apparel • Red $33.91
Tshirts • Women Polyester

96% match
United Colors of Benetton Women Red Po…
Apparel • Red $28.91
Tshirts • Women Polyester

Fig. 25: Image Query – Test Case 1 – Red T Shirt

3. Multimodal Query Tests

- **Test Case 1: Black Dress**

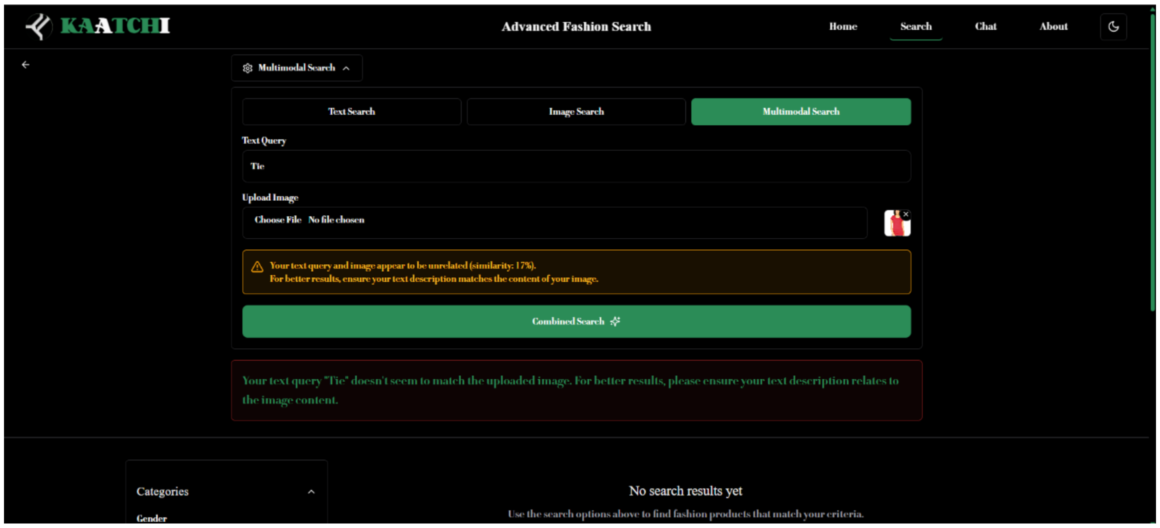


Text Query
Black Dress
Upload Image
Choose File No file chosen

Gender: Women
Category: All
Sub Category

Fig.26: Multimodal Query – Test Case 1 – Black Dress

- **Test Case 3: Fashion Product Validation**



Text Query: Tie

Upload Image: Choose File No file chosen

Your text query and image appear to be unrelated (similarity: 17%).
For better results, ensure your text description matches the content of your image.

Your text query "Tie" doesn't seem to match the uploaded image. For better results, please ensure your text description relates to the image content.

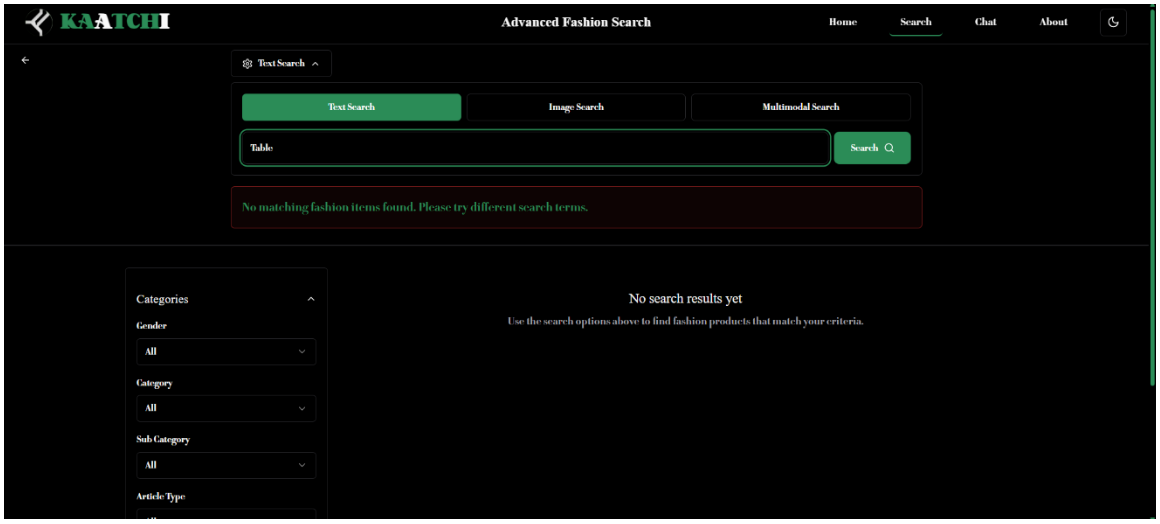
No search results yet

Use the search options above to find fashion products that match your criteria.

Fig.27: Fashion Product Validation

4. Non-Fashion Query Tests

- **Test Case 1 – Text Query**



Table

No matching fashion items found. Please try different search terms.

No search results yet

Use the search options above to find fashion products that match your criteria.

Gender: All

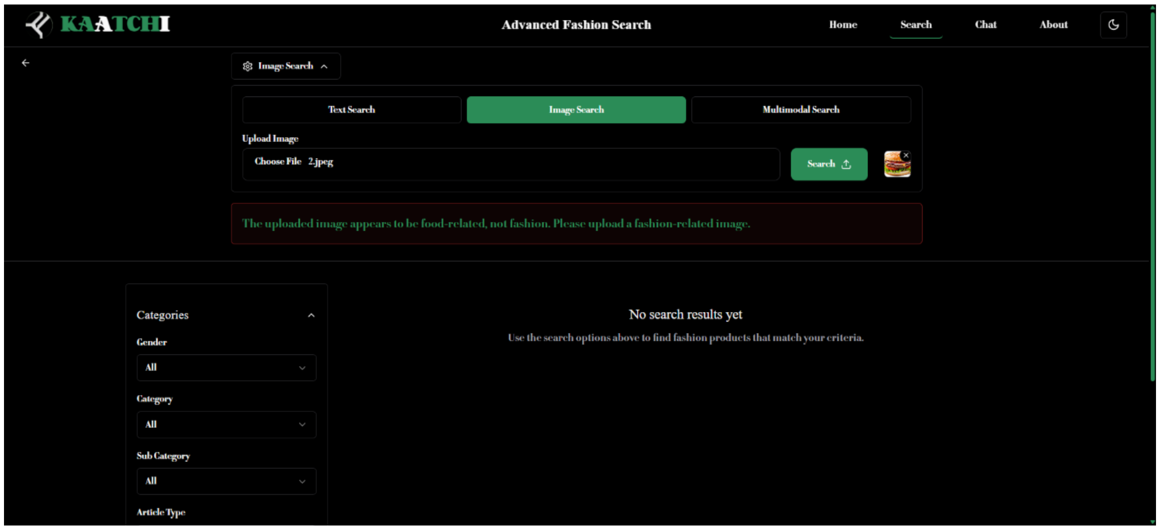
Category: All

Sub Category: All

Article Type

Fig.28: Non-Fashion Text Query

- **Test Case 2 – Image Query**



Upload Image

Choose File 2.jpeg

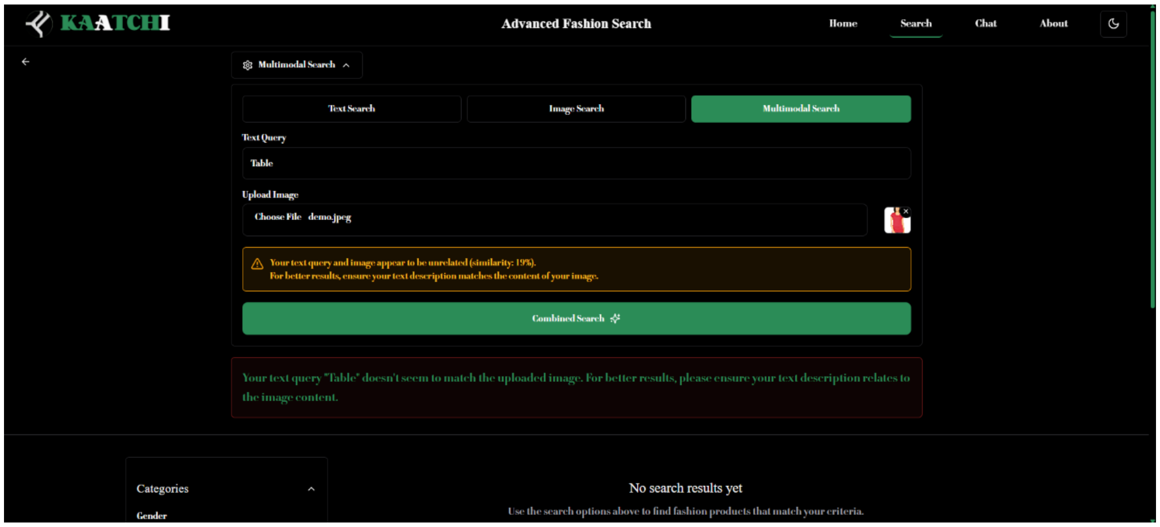
The uploaded image appears to be food-related, not fashion. Please upload a fashion-related image.

No search results yet

Use the search options above to find fashion products that match your criteria.

Fig.29: Non-Fashion Image Query

- **Test Case 3 – Multimodal Query**



Text Query

Table

Upload Image

Choose File demo.jpeg

Your text query and image appear to be unrelated (similarity: 19%).
For better results, ensure your text description matches the content of your image.

Your text query "Table" doesn't seem to match the uploaded image. For better results, please ensure your text description relates to the image content.

No search results yet

Use the search options above to find fashion products that match your criteria.

Fig. 30: Non-Fashion Multimodal Query

6. CONCLUSION

6.1. Design and Implementation Issues

The evolution of KAATCHI AI posed some challenges in the context of incorporating alignment, scalability, and efficiency in retrieval. Overcoming these challenges was paramount to achieving correct and real-time fashion product search.

Challenges:

- **Embedding Alignment:**
  - Maintaining that text and image embeddings from various Vision-Language Model (CLIP) [4] aligned correctly in the common representation space was intricate.
  - Misalignment might result in incorrect retrievals, where images did not match the textual or visual queries correctly.
- **Scalability & Indexing Efficiency:**
  - As the fashion products dataset expanded into millions, efficient retrieval and indexing became imperative in order to ensure real-time performance.
  - Under optimization, FAISS indexing could cause severe search latency and impact user experience.

Solutions Implemented:

- **Fine-Tuning CLIP:**
  - CLIP pre-trained model was fine-tuned on the fashion dataset to enhance semantic consistency between image and text embeddings.
  - Other contrastive learning methods were employed to cluster similar products well.
- **Optimized FAISS Indexing:**
  - Hierarchical Navigable Small World (HNSW) [5] indexing was utilized to accelerate retrieval speed and lower memory usage.
  - Hybrid IVF-PQ (Inverted File System with Product Quantization) [6] was employed to balance between search precision and efficiency to return results within milliseconds in response to queries.

By overcoming these challenges, KAATCHI AI successfully achieved high retrieval accuracy, low latency, and robust multi-modal search capabilities, making it a scalable and efficient fashion visual search engine.

6.2. Advantages and Limitations

Advantages:

- **Multi-Modal Search Capabilities:**
    - KAATCHI AI accommodates text-based, image-based, and multi-modal queries, enabling users to search fashion products with natural language descriptions, sample images, or both.
    - KAATCHI AI improves user experience by delivering more accurate and flexible search results than conventional keyword-based search processes.
- **Scalability with FAISS Indexing:**
    - The system processes large-scale fashion datasets effectively using FAISS [7] indexing, supporting fast and accurate similarity searches.
    - It guarantees that the search engine is very responsive even when dealing with millions of fashion product embeddings.
- **Feature-Rich and Intuitive UI:**
    - The frontend, built with React and Node.js, offers an interactive and seamless user experience.
    - Adaptive theme, product filtering, and percentage product matching display features improve usability and enable users to narrow their searches effectively.

Limitations:

- **Dependency on GPU Acceleration:**
    - For peak performance, KAATCHI AI needs GPU acceleration to make embeddings and retrieve in real time efficiently.
    - Operation on CPU-only setups can cause increased latency and slower search speeds.

- **Difficulty in Fine-Grained Fashion Discrimination:**
    - The system can find it difficult to separate minor differences in fashion products, e.g., slight variations in texture, material quality, or minor design variations.
    - While the model successfully embodies coarse stylistic resemblances, additional advances in fine-tuned embeddings and domain training can be required to increase differentiation.

In spite of these constraints, KAATCHI AI provides a robust, scalable, and accessible fashion image search engine, filling the gap between text-based descriptions and visual product discovery.

6.3. Future Enhancements

To further enhance KAATCHI AI, a number of principal upgrades are scheduled to enhance its accuracy, scalability, and user experience. These upgrades will be designed to maximize search results, tailor recommendations, and broaden the system's uses outside the fashion industry.

- **Enhanced Embedding Alignment with Fine-Tuned Models**
    - Although the existing system utilizes pre-trained VLM (CLIP) for text and image embeddings, fine-tuning on a fashion database can improve the accuracy of retrieval.
    - Domain-specific training will enable finer differentiation of subtle fashion characteristics, including texture of fabrics, intricacies of patterns, and seasonal changes.
- **Personalized Recommendations Based on User Preferences**
    - The incorporation of user behavior analytics will facilitate personalized search results and recommendations based on browsing history, previous purchases, and search interactions.
    - The use of a feedback system in which users can narrow down search results by choosing "more like this" options will improve search relevance over time.
- **Extension to Other Categories (e.g., Home Decor, Art, Accessories)**
    - Although KAATCHI AI is presently targeted at fashion products, its essential multi-modal search architecture can be generalized to home decor, artwork, accessories, and other categories of retail items.

    - This generalization would involve retraining the model on fresh domain-specific datasets but with the same underlying VLM-based retrieval mechanism.

These updates will further cement KAATCHI AI as a scalable, smart, and adaptive visual search technology, enhancing search accuracy, user experience, and usability in various industries.

7. CONTRIBUTION

1. **Angeline A -** Data Preprocessing, VLM (Image and Text Embeddings), Fine Tuning of CLIP Model and Documentation.
2. **Prathish R K -** Data Preprocessing, VLM (Image and Text Embeddings), Fine Tuning of CLIP Model and Documentation.
3. **Selvakumar S** – Complete Interface (Frontend, Backend, Server Management) and All Flow Diagrams

8. Demo Video

Link:

https://drive.google.com/drive/folders/1On8nQc3aYIA0pfA3AFseu0Sl8-Y2DBts?usp=sharing

REFERENCES

[1] Zhang, J., Huang, J., Jin, S., & Lu, S. (2024). Vision-language models for vision tasks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[2] Danopoulos, D., Kachris, C., & Soudris, D. (2019, July). Approximate similarity search with faiss framework using fpgas on the cloud. In International Conference on Embedded Computer Systems (pp. 373-386). Cham: Springer International Publishing.

[3] Conde, M. V., & Turgutlu, K. (2021). Clip-art: Contrastive pre-training for fine-grained art classification. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 3956-3960).

[4] Fang, A., Ilharco, G., Wortsman, M., Wan, Y., Shankar, V., Dave, A., & Schmidt, L. (2022, June). Data determines distributional robustness in contrastive language image pre-training (clip). In International Conference on Machine Learning (pp. 6216-6234). PMLR.

[5] Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE transactions on pattern analysis and machine intelligence, 42(4), 824-836.

[6] Zhang, P., Liu, Z., Xiao, S., Dou, Z., & Yao, J. (2022). Hybrid inverted index is a robust accelerator for dense retrieval. arXiv preprint arXiv:2210.05521.

[7] Rahman, M. D., Rabbi, S. M. E., & Rashid, M. M. (2024). Optimizing Domain-Specific Image Retrieval: A Benchmark of FAISS and Annoy with Fine-Tuned Features. arXiv preprint arXiv:2412.01555.

[8] Lin, X., Gokturk, B., Sumengen, B., & Vu, D. (2008, January). Visual search engine for product images. In Multimedia Content Access: Algorithms and Systems II (Vol. 6820, pp. 242-250). SPIE.

[9] Boriya, A., Malla, S. S., Manjunath, R., Velicheti, V., & Eirinaki, M. (2019, September). ViSeR: A visual search engine for e-retail. In 2019 First International Conference on Transdisciplinary AI (TransAI) (pp. 76-83). IEEE.

[10] Zhong, Y., Garrigues, P. J., & Bigham, J. P. (2013, October). Real time object scanning using a mobile phone and cloud-based visual search engine. In Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility (pp. 1-8).