­

Shop Savvy

SPROJ Report



### **Ahmad Kashif Jabbar – 25100002**

### **Husnain Ali – 25100105**

### **Musa Aftab Ahmed – 25100179**

### **Syed Messam Ali – 25100229**

### **Zainab Fatima – 25100208**

Advisor: Waqar Ahmad  
School of Science and Engineering  
Lahore University of Management Sciences  
**3rd May 2025**

**Acknowledgement and Dedication**

Table of Contents

[1. Introduction](#_Toc67578726) 6

[1.1. Problem Context](#_Toc67578727) 6

[1.2. Objectives](#_Toc67578727) 6

[1.3. Motivation and Relevance](#_Toc67578727) 6

[2. System Design and Architecture 8](#_Toc67578728)

[2.1. Tech Stack Overview](#_Toc67578729) 8

[2.2. System Architecture](#_Toc67578730) 11

[2.3. Data Flow Diagram](#_Toc67578730) 19

[3. Functional Requirements and Implementation](#_Toc67578731) 20

[3.1. Web Scraping & Data Handling](#_Toc67578732) 20

[3.2. MERN Stack Frontend/Backend Integration](#_Toc67578733) 22

[3.2. Real-Time Recommendation System](#_Toc67578733) 24

[3.2. LLM-Powered Chatbot](#_Toc67578733) 27

[3.2. User Features (Wishlist, Login, Signup etc.)](#_Toc67578733) 30

[4. Testing and Evaluation](#_Toc67578734) 32

[4.1. Functional Testing](#_Toc67578735) 34

[4.2. Performance Considerations](#_Toc67578736) 36

[4.3. Deployment Notes](#_Toc67578736) 38

[4.4. User Feedback (if any)](#_Toc67578736) 39

[5. Conclusions & Future Work](#_Toc67578734) 40

[4.1. Summary of Achievements](#_Toc67578735) 42

[4.2. Limitations](#_Toc67578736) 44

[4.3. Future Improvements and Expansion](#_Toc67578736) 45

[References](#_Toc67578737) 45

[Appendix](#_Toc67578738) 46

[A.1 Sample Scraping Output](#_Toc67578739) 47

[A.2 Sample Recommendation System Output](#_Toc67578740) 49

[A.3 Chatbot Prompt Flow / API Sample](#_Toc67578739) 50

[A.4 MongoDB Schema Diagrams](#_Toc67578739) 51

List of Figures

[Figure 1: Overall System Architecture](file:///F:\Downloads\Final%20Report%20Template.docx#_Toc67577899)

[Figure 2: User Interaction Flow](file:///F:\Downloads\Final%20Report%20Template.docx#_Toc67577899)

[Figure 3: Web Scraping Pipeline](file:///F:\Downloads\Final%20Report%20Template.docx#_Toc67577899)

[Figure 4: Chatbot Pipeline](file:///F:\Downloads\Final%20Report%20Template.docx#_Toc67577899)

[Figure 5: Recommendation Pipeline](file:///F:\Downloads\Final%20Report%20Template.docx#_Toc67577899)

­­

List of Tables

[Table 1: Technologies Used](#_Toc67577918)

[Table 2: Scraping Summary from Each Brand](#_Toc67577919)

[Table 3: Frontend Functional Components](#_Toc67577919)

**Chapter 1**

# Introduction

## Problem Context

In recent years, the online retail industry in Pakistan has grown rapidly, yet consumers still face fragmented shopping experiences when searching for clothing across various brand websites. The absence of a centralized platform that aggregates fashion items from multiple local brands makes product discovery tedious and time-consuming. Consumers must manually browse different sites, filter through irrelevant items, and often miss relevant products that align with their personal style or preferences.

Furthermore, local fashion brands typically lack advanced personalization features, such as behavior-driven recommendations or conversational support. International e-commerce giants like Amazon and Zalando have long adopted AI-based recommendation engines and intelligent chat interfaces. However, similar innovations remain scarce in the Pakistani fashion retail domain. This gap presents a unique opportunity to blend AI-driven personalization with localized fashion content in a way that caters specifically to the Pakistani consumer base.

## Objectives

The primary objective of Shop Savvy is to enhance the online shopping experience for Pakistani consumers by offering a personalized clothing recommendation platform that aggregates products from local brands. Specifically, the project aims to:

* Aggregate fashion products from multiple Pakistani clothing brands via automated web scraping.
* Develop a responsive web application using the MERN (MongoDB, Express, React, Node.js) stack.
* Implement a real-time recommendation engine that suggests products based on user interactions, including filters, clicks, wishlist activity, and search history.
* Integrate a large language model (LLM)-powered chatbot to assist users with product discovery and query resolution.
* Provide a seamless and engaging user interface that enables product viewing, filtering, redirection to brand websites, login/signup, and wishlist features.

Through these objectives, Shop Savvy intends to combine the power of artificial intelligence and modern web development practices to transform the way Pakistani users shop for clothing online.

## Motivation and Relevance

The motivation behind Shop Savvy stems from the need to bridge the technological and user-experience gap in Pakistan’s fashion e-commerce sector. With consumers increasingly expecting intelligent, user-friendly, and personalized online experiences, the project addresses several core challenges:

* **Fragmentation**: Users must navigate multiple disconnected brand websites, each with varying UI and filter systems.
* **Lack of Personalization**: Most local platforms provide generic listings with minimal support for personal style or behavioral tailoring.
* **Manual Effort in Product Discovery**: Without unified filtering or intelligent recommendations, users struggle to find suitable clothing items efficiently.
* **No Smart Assistance**: Very few local platforms utilize chatbot support or AI assistants to aid customer decision-making.

By tackling these challenges, Shop Savvy holds the potential to deliver a smarter shopping experience tailored for local users. It is also highly relevant in the context of growing digital adoption in Pakistan and the increasing use of AI in consumer applications. The project not only serves as a proof of concept for personalized fashion aggregation but also opens doors for future commercial scalability, affiliate partnerships, and data-driven insights into user shopping behavior.

**Chapter 2**

# System Design and Architecture

## Tech Stack Overview

**Website (User Interface & Backend)**

The ShopSavvy platform is developed using the MERN stack with enhancements for performance and modern deployment:

* **MongoDB:** Serves as the primary database to store user profiles, product data, interactions (clicks, filters, searches), and wishlists.
* **Express.js:** Powers the backend logic and RESTful API layer.
* **React.js** (with Vite): Provides a fast, modern frontend development experience and renders the dynamic user interface.
* **Node.js**: Serves as the runtime environment for backend services.
* **Deployment:** The frontend is deployed using Vercel, and the backend (Express.js server) is hosted on Render.

### **Web Scraping Tech Stack**

To power the data ingestion for our fashion recommendation engine, we designed and implemented a robust, scalable web scraping pipeline specifically for the e-commerce domain. The pipeline was responsible for crawling, extracting, validating, and storing product data from various online fashion retailers, starting with LAMA Retail.

#### **Key Libraries & Tools:**

|  |  |
| --- | --- |
| **Library/Tool** | **Role in the Pipeline** |
| **Playwright (async)** | Headless browser automation for dynamic content rendering and infinite scroll handling. |
| **BeautifulSoup** | HTML parsing and DOM traversal to extract product information from static HTML. |
| **asyncio** | Enables non-blocking scraping with concurrent tasks across multiple URLs. |
| **ThreadPoolExecutor** | Used for concurrent link validation with I/O-bound HEAD requests. |
| **requests** | Used for validating URLs and downloading image binaries. |
| **PIL (Pillow)** | Image processing and local storage of product images in a structured directory. |
| **tqdm** | Visual progress bars for monitoring multi-threaded operations like link validation. |
| **pymongo** | Seamless integration with MongoDB for storing, updating, and querying product records. |
| **MongoDB Atlas** | Cloud database used as a central storage unit for all product metadata. |
| **nest\_asyncio** | Facilitates nesting of asyncio event loops in notebook or script-based environments. |
| **matplotlib** | (Optional) For visual debugging, analysis, or exploratory plots. |
| **boto3** | Reserved for future cloud integration to push image or data assets to AWS S3. |

#### **Architecture Overview:**

* **Scraping Engine**: Built on top of Playwright's asynchronous API to handle JavaScript-heavy sites, simulate scrolling, and ensure full DOM rendering before scraping.
* **Data Extraction**: Combines robust CSS selectors with BeautifulSoup to isolate product fields such as name, price, color variants, sizes, and image URLs.
* **Data Cleaning**: Post-processing logic includes deduplication, primary color detection, and formatting based on product type and gender.
* **Image Management**: Images are downloaded, processed via Pillow, saved locally in the product\_images/directory, and prepared for cloud migration.
* **Persistence Layer**: MongoDB is used as the primary data store, with separate steps for inserting new records and cleaning invalid links.
* **Link Validator**: Concurrent validation logic ensures that only live and functional product links are retained and ingested.

#### **Scalability & Maintainability:**

* Designed to be **modular**, allowing easy adaptation to new retailers by swapping base URLs and selectors.
* **Retry mechanism** ensures scraping resilience in case of transient errors.
* Built with **asynchronous programming** and **multithreading**, supporting large-scale scraping sessions efficiently.

This stack empowered our system to autonomously populate the fashion database with diverse, high-quality product data, enabling real-time recommendations and visual analysis with minimal human supervision.

**Recommendation System Tech Stack**

The real-time recommendation engine in **ShopSavvy** is designed to personalize user experience based on behavioral data. It collects, processes, and analyzes multiple user interaction signals to dynamically suggest relevant products. The system is modular, scalable, and integrated directly into the backend pipeline to serve fast, intelligent suggestions. Below are the key components:

#### **User Interaction Signals Captured**

* **Product Clicks**
* **Wishlist Behavior**
* **Filters Applied**
* **Categories/Genders Explored**

These interactions are stored in MongoDB and processed on request.

#### **Behavioral Data Processing**

* **Recent vs. Aggregated Views**: User behavior is split into recent interactions (e.g., last few clicks) and historical summaries (aggregated preferences).
* **Standardization**: Each behavior type is normalized using sklearn.preprocessing tools like StandardScaler and MinMaxScaler.
* **Embedding Construction**: User behavior is transformed into embedding vectors using NumPy arrays.

#### **Recommendation Logic**

* **Similarity Computation**:
  + Cosine similarity is calculated between user embeddings and all product vectors using scikit-learn’s cosine\_similarity function.
* **Multi-List Merging**:
  + Four sorted recommendation lists (based on clicks, wishlist, filters, and category exploration) are merged using a weighted scoring algorithm.
  + Heavier weights are assigned to stronger signals: Clicks > Wishlist > Categories > Filters.
* **Top Picks Generation**:
  + The merged list is trimmed to the **top 100** items and returned as a JSON response from the Express.js backend API.

#### **Libraries & Tools Used**

* **NumPy**: For vector operations and score calculations.
* **Scikit-learn**: For standardization and cosine similarity.
* **Pandas**: To organize and process tabular behavior data.
* **MongoDB (PyMongo)**: For querying user logs and product vectors.
* **Flask** (for prototyping): Used in early experimentation and model testing phases.
* **Express.js**: Hosts the /get-top-100 endpoint and integrates the recommendation engine with the main backend API.

#### **Deployment**

* The recommendation logic is containerized and deployed alongside the backend on **Render**, ensuring it runs in real time whenever a user logs in or opens the homepage.

**Chatbot Tech Stack**

The AI-driven chatbot powers intelligent interaction and contextual fashion suggestions, leveraging the following tools:

* **OpenAI GPT-4.0 Mini:** Used to generate natural language descriptions of products and assist in crafting human-like responses.
* **Pinecone:** Stores and retrieves vector embeddings of product descriptions to enable semantic similarity matching.
* **Hugging Face API:** Used for generating vector embeddings of the detailed descriptions to be stored in the pinecone.
* **LLaMA 3 (llama3-8b-8192):** Integrated as the primary model to interpret user queries and generate fashion advice based on embeddings and context.

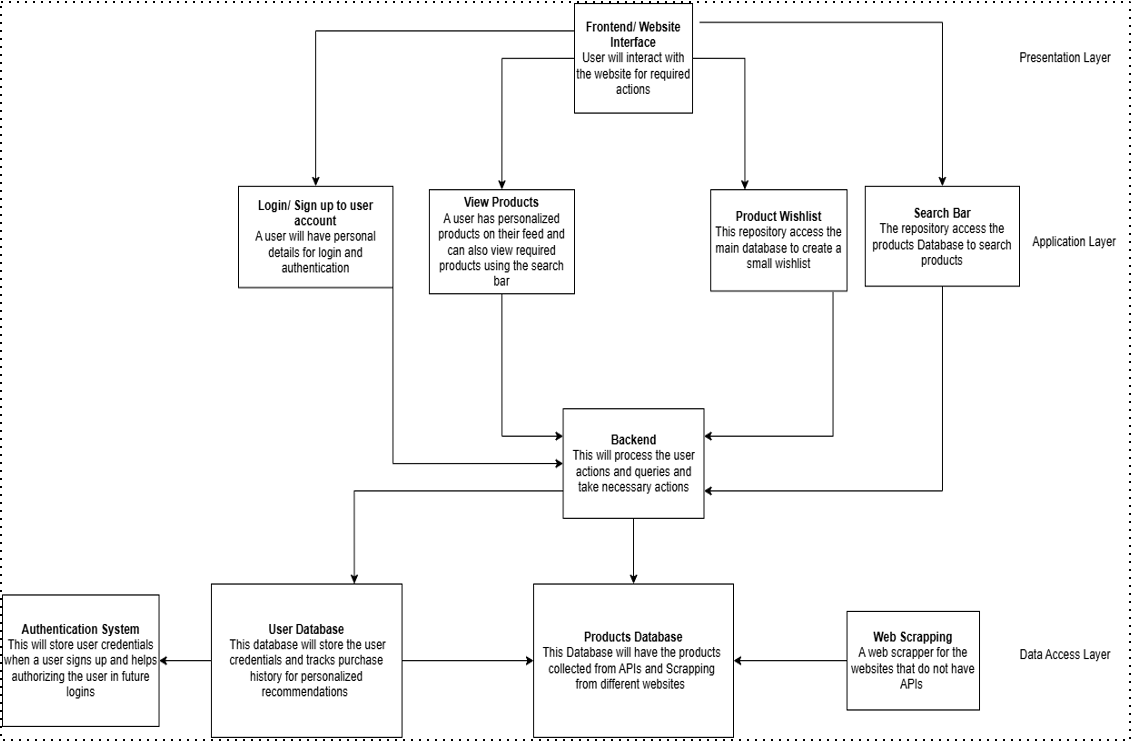
## System Architecture

The system architecture of Shop Savvy is designed to be modular, maintainable, and scalable, aligning with best practices for full-stack web applications. It adopts a **layered architecture** integrated with service-oriented principles to allow seamless expansion, robust user interaction, and efficient processing of data-driven recommendations.

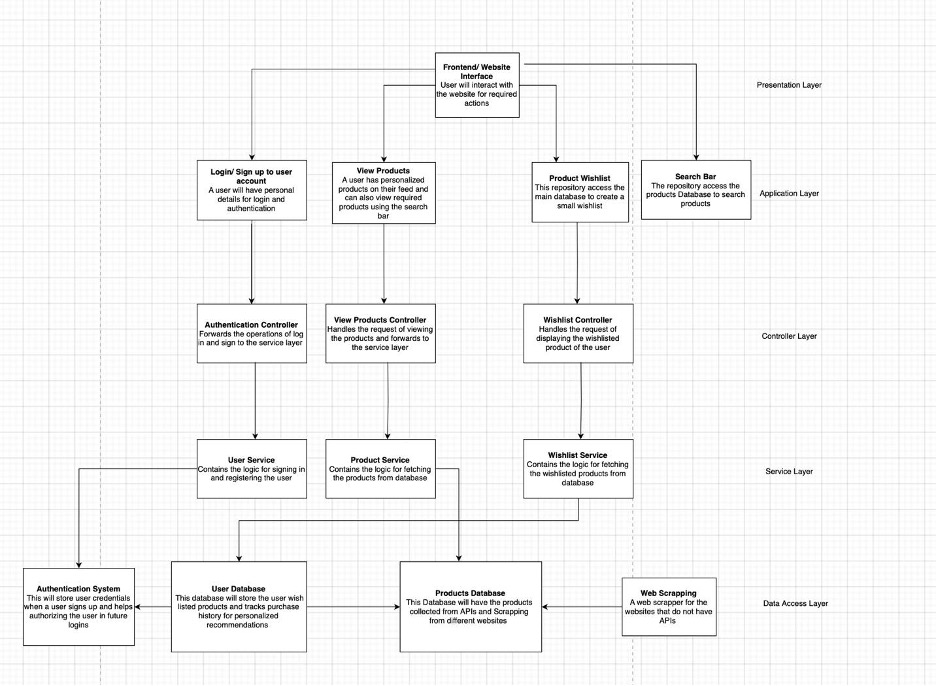
The architecture is divided into distinct layers and modules that collectively support all core features—such as personalized recommendations, product browsing, user interaction, authentication, and backend automation (e.g., scraping).

### **2.1 Architecture Diagram – As Implemented in the Prototype**

This diagram shows the early-stage architecture used during the development prototype. It includes basic frontend-backend communication, direct database access, and module-level separation but lacks full implementation of service and controller layers.

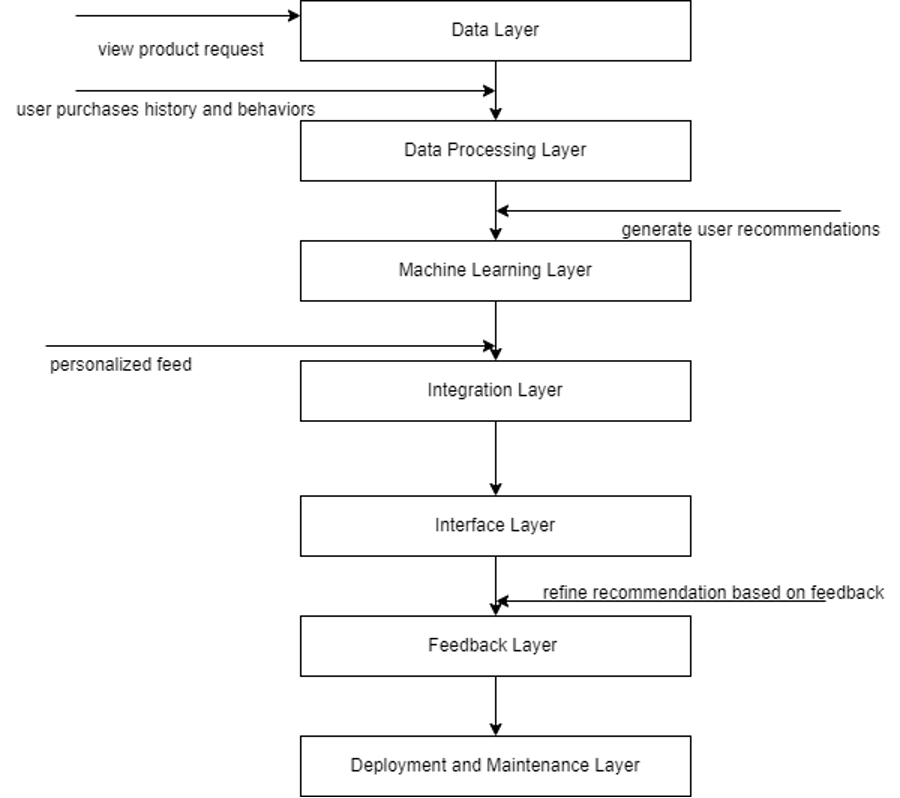


### **2.2 Architecture Diagram – As it Should Be**



The updated architecture improves upon the prototype with a well-structured **Layered Architecture**, including clearly defined responsibilities for each layer and component. This modular approach enhances maintainability, reusability, and scalability.

## Layered Architecture Overview



### **1. Presentation Layer (Frontend - React)**

The frontend is built using **React (v18.x or later)** and is hosted on **Vercel**, optimized for fast loading and dynamic rendering.

* **Main Responsibilities:**
  + Display product listings, filters, and search bar.
  + Provide interactive UI for login/signup, chat, and wishlist.
  + Communicate with backend APIs via RESTful endpoints.
* **Key Components:**
  + LoginForm – Manages user authentication interface.
  + ProductList – Dynamically renders product cards.
  + SearchBar – Accepts and transmits search queries.
  + ChatInterface – Allows real-time interaction with the GPT-powered chatbot.

### **2. Application Layer**

This layer contains **controllers** and **services**, segregating request handling from business logic.

#### **Controllers**

* Handle incoming HTTP requests and route them to the appropriate service functions.
* Examples:
  + AccountController.js – Routes for login, signup, logout.
  + ProductController.js – Routes for product retrieval and filtering.
  + WishlistController.js – Handles add/remove operations.

#### **Services**

* Encapsulate core business logic, data validation, and API logic.
* Examples:
  + AccountService.js – Validates user credentials and interacts with the DAO layer.
  + RecommendationService.js – Computes and returns user-specific product recommendations.
  + ScraperService.js – Schedules and triggers scraping routines.

### **3. Data Access Layer**

This layer interacts directly with **MongoDB (v6.x+)**, using **models** and **Data Access Objects (DAOs)** to abstract database operations.

* **Models:**
  + Define data schemas for collections like User, Product, Wishlist, and SearchQuery.
* **DAOs:**
  + Encapsulate all database queries.
  + Examples:
    - ProductDAO.js – Handles CRUD operations and filter logic for products.
    - UserDAO.js – Manages user data fetch/update.
    - WishlistDAO.js – Manages wishlist-product associations.

## Supporting Modules and Systems

### **Scraping Module (Node.js + Puppeteer)**

* Scrapes data from Pakistani clothing brand websites (currently **Outfitters** and **LAMA**).
* Standardizes the data format and pushes it to MongoDB.
* Can be triggered manually (admin) or scheduled periodically.
* Each scraping instance is stored with metadata (e.g., source\_url, last\_scraped).

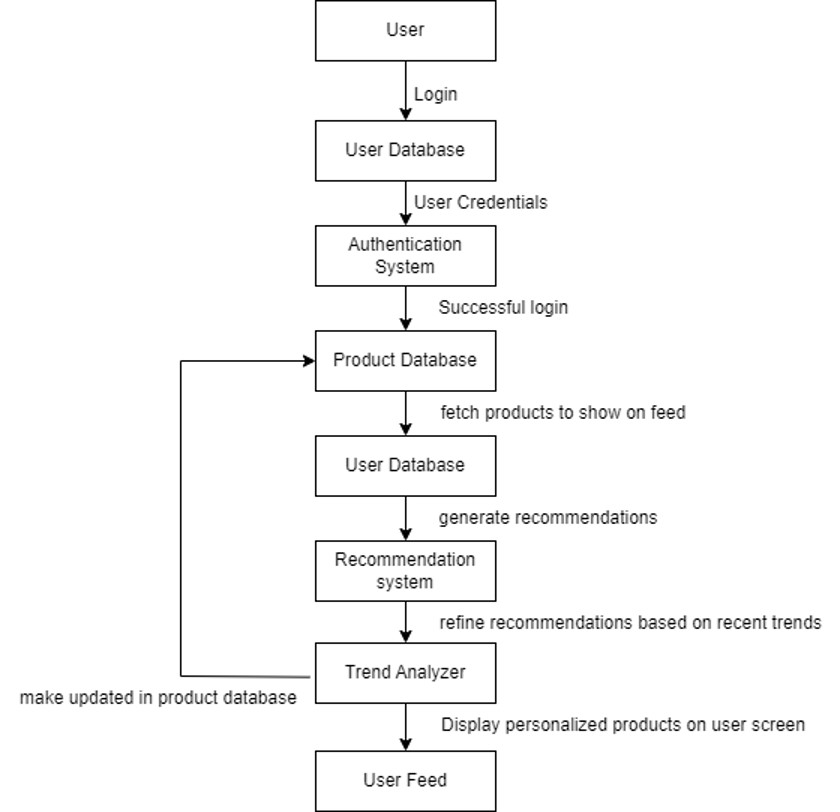
### **Chatbot Module (GPT-3.5 via OpenAI API)**

* Interacts with users for:
  + Product discovery and filtering guidance.
  + Answering general queries about fashion and shopping.
  + Enhancing user experience with natural language support.
* Integrated into the frontend via a dedicated chat interface and API.

### **Recommendation Engine**

* Analyzes user behavior such as:
  + Selected filters
  + Clicked product items
  + Wishlist contents
  + Search queries
* Uses AI models (possibly fine-tuned from **Hugging Face** or embedded using **Pinecone**) to:
  + Store and compare vector embeddings of products and user preferences.
  + Return personalized product suggestions via /api/recommendations.

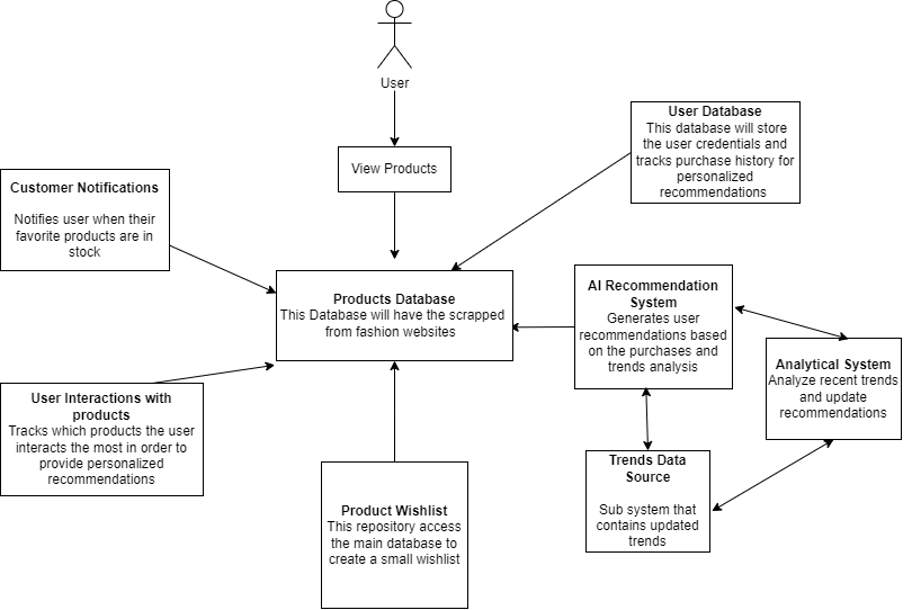
## Pipe-and-Filter View



This view reflects how the system can be seen as a pipeline of data processing steps:

1. **Input Filters:** User inputs (search, clicks, filters)
2. **Intermediate Processing:** Services (e.g., filtering, personalization, scoring)
3. **Output:** Displaying products, chatbot replies, recommendation list

## Repository Architecture

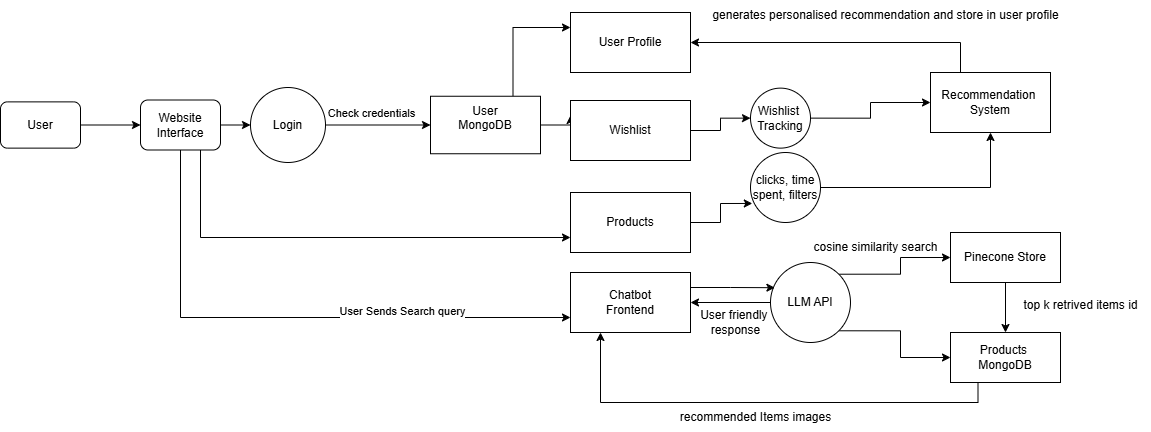


The codebase follows a **Repository Pattern**, with separate directories for controllers, services, models, DAOs, and utilities. This separation enforces modularity and aligns with MVC principles.

## Justification of Architectural Choices

* **Separation of Concerns:** Each component/layer handles a specific task, improving testability and maintainability.
* **Extensibility:** New features (e.g., product reviews, seller dashboards) can be added by extending respective service and DAO layers.
* **Reusability:** Common logic (e.g., authentication checks) is centralized in services/middleware.
* **Scalability:** React and Node.js handle increasing user traffic efficiently; MongoDB supports horizontal scaling.

## Data Flow Diagram



**Chapter 3**

# Functional Requirements and Implementation

## Web Scraping & Data Normalization Pipeline

Our system incorporates automated web scraping to aggregate clothing products from third-party fashion retailer websites—**Outfitters** and **LAMA**. This data pipeline ensures fresh and diverse product listings are continuously available on our platform.

### **Scraping Strategy**

Both brands were scraped using similar strategies involving:

* **Requests & BeautifulSoup**: Each product listing page and its details were fetched using the requests library and parsed with BeautifulSoup.
* **Pagination Handling**: Custom logic was implemented to iterate through all available pages using detected next-page buttons or URL patterns.
* **Product Detail Extraction**: From each product, we extracted:
  + Title
  + Image URL
  + Price (original and discounted)
  + Category (Men, Women, etc.)
  + Tags and Metadata (if available)
  + Product detail page URL
* **Data Cleaning**: Raw text was cleaned using regular expressions to remove unwanted symbols or spaces. Prices were standardized as float values.

### **Standardization and Storage**

To ensure interoperability between different sources, all scraped products were mapped to a **unified schema**:

|  |  |
| --- | --- |
| **Field** | **Description** |
| \_id | Unique identifier for the product |
| product | Name or title of the product. |
| price | Price of the product in PKR (can be stored as integer or float). |
| colors | Array of available color options |
| sizes | Array of available sizes |
| primary\_color | Main visible color of the product (used for filtering or display). |
| link | URL to the product detail page on the store website. |
| images | Array of image URLs representing different views of the product. |
| type | General category of clothing |
| gender | Target gender category |
| filtercolor | Standardized color label used for filtering |
| brand | Brand name associated with the product |
| status | Indicates if the product is valid or needs review |

Table 2: Scraping Final Product Structure

The standardized data is then inserted into a **MongoDB collection**, enabling further processing and real-time querying for filters, recommendations, and search.

### **Scraping Pipeline**

Here is a flowchart that outlines the overall scraping and standardization pipeline:

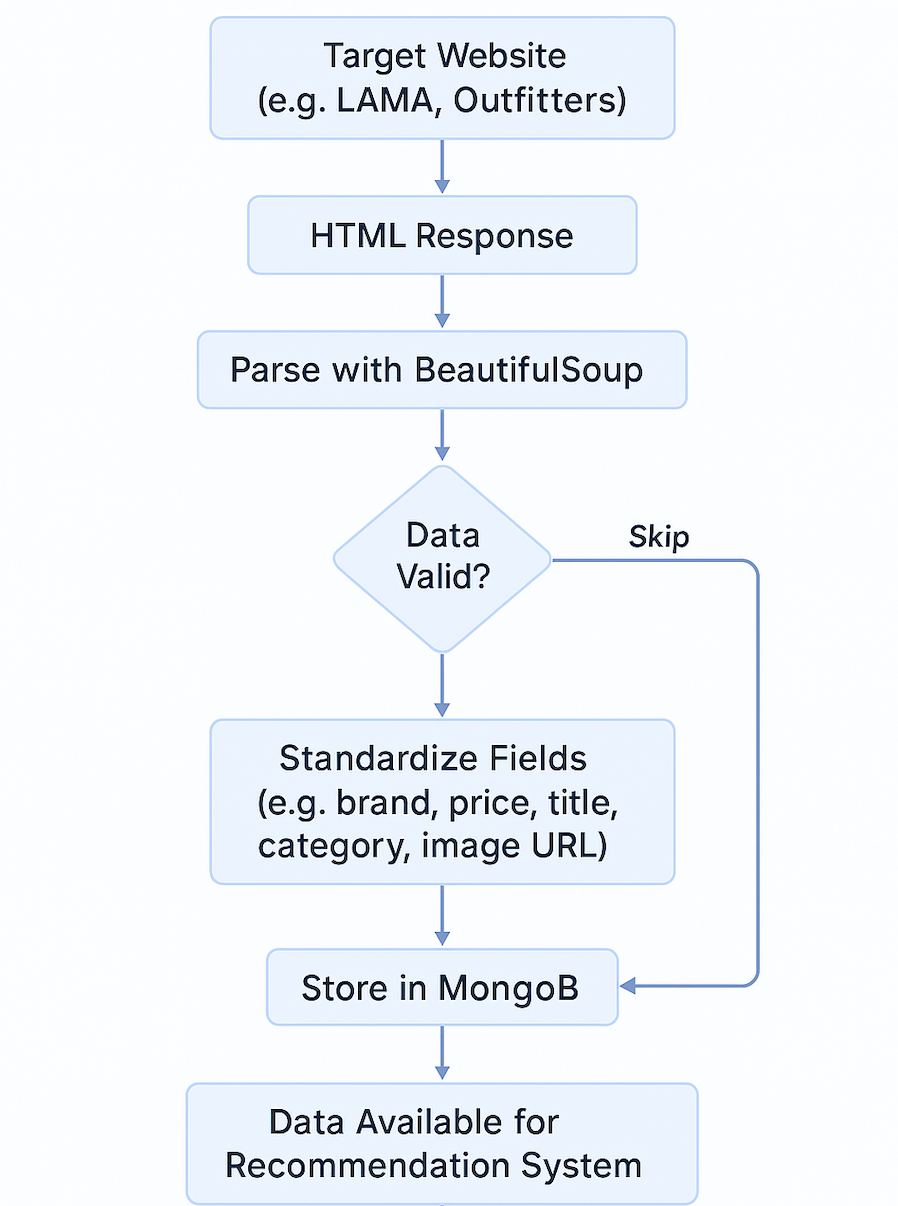


Figure 3: Web Scraping Pipeline

This modular pipeline ensures that any future brand can be integrated by adjusting the parsing logic while keeping the rest of the system intact.

## MERN Stack Frontend/Backend Integration

We used the **MERN stack** to implement a unified, full-stack JavaScript architecture for the ShopSavvy platform, enabling smooth interaction between the frontend and backend.

**Frontend (React with Vite)**

The frontend is built using **React** and bundled with **Vite** for faster development and build performance. The app is deployed on **Vercel**, ensuring scalable, serverless deployment.

* **Routing**: Managed with react-router-dom, enabling client-side navigation across key views such as:
  + /products: Product listing page with filters.
  + /product/:id: Detailed product view.
  + /wishlist: User’s wishlist items.
  + /chatbot: Conversational chatbot interface.
  + /profile: User profile page.
  + /recommended-products: Personalized recommendation results.
  + /product/unavailable/:id: Placeholder for products no longer available.
* **Component Structure**: Modular components help organize and reuse UI elements:
  + ProductDisplay
  + ProductDetail
  + ProductFilters
  + WishlistPage
  + Chatbot
  + AuthPages
  + Navbar / Footer
* **State Management**: Local state and useEffect hooks are used to manage component-level logic and asynchronous data fetching.
* **API Communication**: All API calls are handled using the native **Fetch API** for asynchronous interaction with the backend.

**Backend (Node.js + Express.js)**

The backend server is developed with **Express.js** and deployed via **Render**, allowing for cloud-based scalability and CI/CD integration.

* **API Endpoints**: The backend exposes RESTful APIs including:
  + GET /api/fetchproducts: Returns paginated product data.
  + POST /api/filter: Applies filters such as brand, category, and price range.
  + POST /api/signup, POST /api/login: Handles secure user registration and login via JWT.
  + POST /api/wishlist: Adds or removes items from the wishlist.
  + GET /api/wishlist: Retrieves wishlist items.
  + GET /api/recommendations: Returns personalized product recommendations based on user behavior.
* **Middleware & Security**: JWT-based authentication middleware ensures that only logged-in users can access certain endpoints like wishlist and recommendations.
* **Database Integration**: The backend interacts with **MongoDB Atlas**, storing all user, product, and interaction data in the cloud.

**Integration Workflow**

The React frontend communicates with the backend via Fetch API. User actions like applying filters, viewing product details, or saving to the wishlist trigger API calls to Express routes, which in turn query the MongoDB database and return the relevant response. This ensures a responsive and dynamic user experience without full-page reloads.

## Real-Time Recommendation System

A recommendation system is designed to enhance the user experience by providing personalized suggestions that align with individual preferences and behaviors. By analyzing user interactions and historical data, it aims to predict and suggest products that the user is most likely to be interested in, increasing engagement and driving more relevant content on the platform. This dynamic and real-time approach ensures that users are consistently presented with items tailored to their unique needs and desires, fostering a more engaging shopping experience.

To personalize the shopping experience, we built a real-time recommendation system deployed on Render, which dynamically suggests products based on users’ behavioral data. The system leverages four main data signals:

1. **Products Clicked**
2. **Categories/Gender Explored**
3. **Filters Applied**
4. **Wishlist Items**

### **Behavioral Vector Construction**

All behavior except for the Wishlist is split into two forms:

* **Recent Entries** (most recent interactions)
* **Aggregated Behavior** (historical summary)

Each form is standardized independently (e.g., scaled or normalized). A combined vector is created by merging recent and aggregated behaviors, giving higher weight to recent actions. This results in a user embedding vector per behavior type.

### **Similarity Computation**

Cosine similarity is used between each user vector and the vector representation of all standardized product entries in the database. This generates four sorted lists (one per behavior type) of recommended products in descending order of similarity.

### **Merging and Weighting**

These four lists are merged using weighted scoring, with importance assigned as:

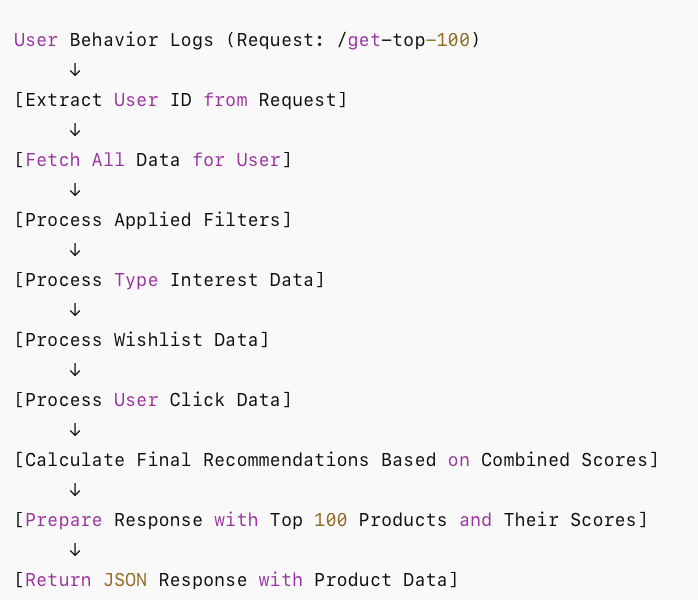
* **Clicks > Wishlist > Categories Explored > Filters Applied**

This ensures that signals indicating stronger intent (like product clicks) have a higher influence on the final recommendations.

### **Top Picks for Backend API**

The final sorted list is trimmed to the top 100 products. These 100 items are returned to the Express.js backend API through a POST endpoint. The backend uses this data to serve personalized top picks on the frontend website.

### **Pipeline Diagram:**

Figure 5: Recommendation Pipeline

This flowchart visually represents the sequence of actions in our code:

1. **User Behavior Logs** are gathered based on the incoming GET request to /get-top-100.
2. The **User ID** is extracted from the request parameters.
3. The system **fetches all relevant data** for the specified user (through fetch\_all\_data).
4. The system processes various types of user behavior data:
   1. **Applied Filters** through process\_applied\_filters.
   2. **Type Interests** through process\_type\_interest.
   3. **Wishlist** data through process\_wishlist.
   4. **Click Data** through process\_user\_click\_data.
5. It then **calculates the final recommendations** by merging all these data points into a final ranking.
6. The top 100 product recommendations are **formatted into a JSON response**.
7. Finally, the API **returns this data** to the frontend in the desired format.

This flow ensures that the recommendation system is personalized and based on the user's interactions, filters, and preferences.

## LLM-Powered Chatbot

We designed an intelligent fashion assistant that uses Large Language Models (LLMs), vector similarity search, and dynamic product rendering to deliver real-time, personalized clothing suggestions. It is optimized for speed and scalability, the chatbot understands user intent, retrieves semantically relevant products, and generates stylistic outfit advice—all within seconds. It supports product image integration and is designed for future extensions into generative fashion visuals.

**Core Architecture:**

* **LLM (Groq + Llama3)** handles:
  + Classification (fashion query, greeting, unrelated)
  + Extraction of **search terms** from fashion queries
  + Generation of a **response template** using {products} placeholder
* **Vector Search via Pinecone**:
  + Search product embeddings based on extracted terms
  + Cached, filtered, and run asynchronously
* **MongoDB Product DB**:
  + Stores real product descriptions
  + Fetched via Mongo IDs returned from Pinecone
  + Fills the placeholders in response template

**Optimization Highlights:**

* **Reduced two LLM calls to one**:
  + Previously:  
     User Query → LLM generates description → Vector Match → LLM formats output
  + Now:  
     User Query → LLM generates search terms + template → Vector Match → Fill template with Mongo products
* **Improved Latency**:
  + Real-time response achieved (2–3 seconds vs 3–4 minutes earlier)
* **Modular and Extensible**:
  + Caches (via @lru\_cache, Mongo pre-load)
  + Async vector search
  + Session-based conversation history for context retention
  + Custom logic for greetings, unrelated queries, or fallback

**Comparison: Old vs New Pipeline**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Previous Pipeline** | **Optimized Pipeline** |
| LLM Calls | 2 | 1 |
| Response Time | ~3–4 minute | ~2–3 seconds |
| Cost Efficiency | Low (high token usage) | High (single LLM call) |
| Template Flexibility | Hard-coded | Dynamic with    {products} placeholder |
| Real-Time Feasibility | No | Yes |
| Product Embedding Match | Based on full LLM description | Based on focused search terms |

### **3.5 User Features (Login, Filters, Wishlist, Pagination)**

The ShopSavvy platform offers essential user-centric features designed to improve discoverability, personalization, and engagement. Each feature is backed by robust API integration, stateful client-side logic, and secure backend infrastructure.

#### **3.5.1 User Authentication (Login & Signup)**

Authentication is a key requirement for accessing personalized features such as the wishlist and recommendations. The platform provides **Login** and **Signup** capabilities through a unified /auth component.

**Frontend Workflow**:

* Users input their **username**, **email**, and **password** on the registration page.
* Upon form submission, a POST request is sent to /api/signup or /api/login.
* On successful login or signup, a **JWT token** is received and stored securely in the client (typically in localStorage or memory, depending on implementation).
* This token is attached to all authenticated requests (e.g., wishlist retrieval) via the Authorization header (Bearer <token>).

**Backend Workflow**:

* Passwords are hashed using **bcrypt** before storing in MongoDB.
* Login checks involve verifying the email and comparing the hashed password.
* JWTs are signed with a server-side secret and include expiration time and user payload.

**Protected Routes**:

* Pages like /wishlist and /recommended-products are protected.
* A **middleware function** (verifyToken) is used to intercept and validate JWTs before processing the request.

This ensures a secure and stateless authentication mechanism suitable for scaling across multiple frontend clients.

#### **3.5.2 Product Filters**

An advanced filtering interface is available on the /products page, enabling users to narrow down their product search across multiple attributes. Filters are fully dynamic and are constructed using the metadata stored in MongoDB during the scraping pipeline.

**Available Filters**:

* **Sort By Price**: Ascending or descending order.
* **Alphabetical Sorting**: A–Z and Z–A sorting based on product title.
* **Price Range Slider**: Users can define minimum and maximum price boundaries. This range is mapped to a MongoDB $gte and $lte query.
* **Color Filters**: Based on the standardized filtercolor field (e.g., Black, White, Blue). Users can select multiple colors.
* **Size Filters**: Pulled dynamically from the sizes array field. Each product may support sizes such as XS, S, M, L, XL, etc.
* **Brands**: Users can filter based on the scraped brand field (e.g., Outfitters, LAMA).
* **Clear All**: A button that resets all selected filters to default and triggers a fresh API call.

**Technical Flow**:

* Filters are stored in local React state and updated using event handlers.
* On any change, a POST request is sent to /api/filter with the filter object as JSON.
* The backend processes these filters via a MongoDB query and returns matched products.
* Products are updated in the UI using conditional rendering, ensuring a smooth and responsive UX.

This filtering system is built to be **extensible**, allowing new filters (e.g., material, pattern) to be added with minimal changes to the schema and API logic.

#### **3.5.3 Wishlist System**

The **Wishlist** is a personalized feature that allows users to bookmark and revisit products they are interested in. It is implemented as a separate page (/wishlist) and is accessible only after authentication.

**Frontend Workflow**:

* Users can click a heart icon on any product card to toggle it in/out of the wishlist.
* Clicking redirects unauthenticated users to the /login page.
* Authenticated actions send a POST request to /api/wishlist with the product ID and user token.
* The /wishlist page retrieves data using a GET request to /api/wishlist, returning all products the user has saved.

**Backend Workflow**:

* Wishlist entries are stored in a separate MongoDB collection with fields: user\_id, product\_id, and timestamp.
* Wishlist updates use **upsert operations** to efficiently add/remove items.
* Only authenticated users with valid JWTs can access or modify their wishlist.

**Wishlist View**:

* Displays each saved product with its title, image, and price.
* Includes a "Remove" button for deleting an item from the list.
* A “Go to Product” button links directly to the product’s detailed view.

This system improves user retention by allowing users to curate their own product space and return later to make purchase decisions.

#### **3.5.4 Pagination**

To manage large volumes of product data efficiently, the platform implements **server-side pagination**. Pagination helps reduce load times, limits API response size, and improves frontend performance.

**Technical Workflow**:

**Frontend**:

* The /products page uses a pagination component that sends the current page number to the backend via a GETrequest:  
  /api/fetchproducts?page=2&limit=20
* Results are rendered in a grid layout with loading states.
* Pagination controls (next/previous/page number buttons) are dynamically generated and updated based on the totalPages value from the response.

**Backend**:

* The Express route for /api/fetchproducts reads query parameters page and limit.
* A MongoDB .skip() and .limit() operation retrieves the relevant slice of data.
* The response includes metadata:

json

CopyEdit

{  
 "currentPage": 2,  
 "totalPages": 5,  
 "products": [...]  
}

This pagination system ensures that product discovery remains efficient and manageable, even as the product catalog scales with more brands and categories.

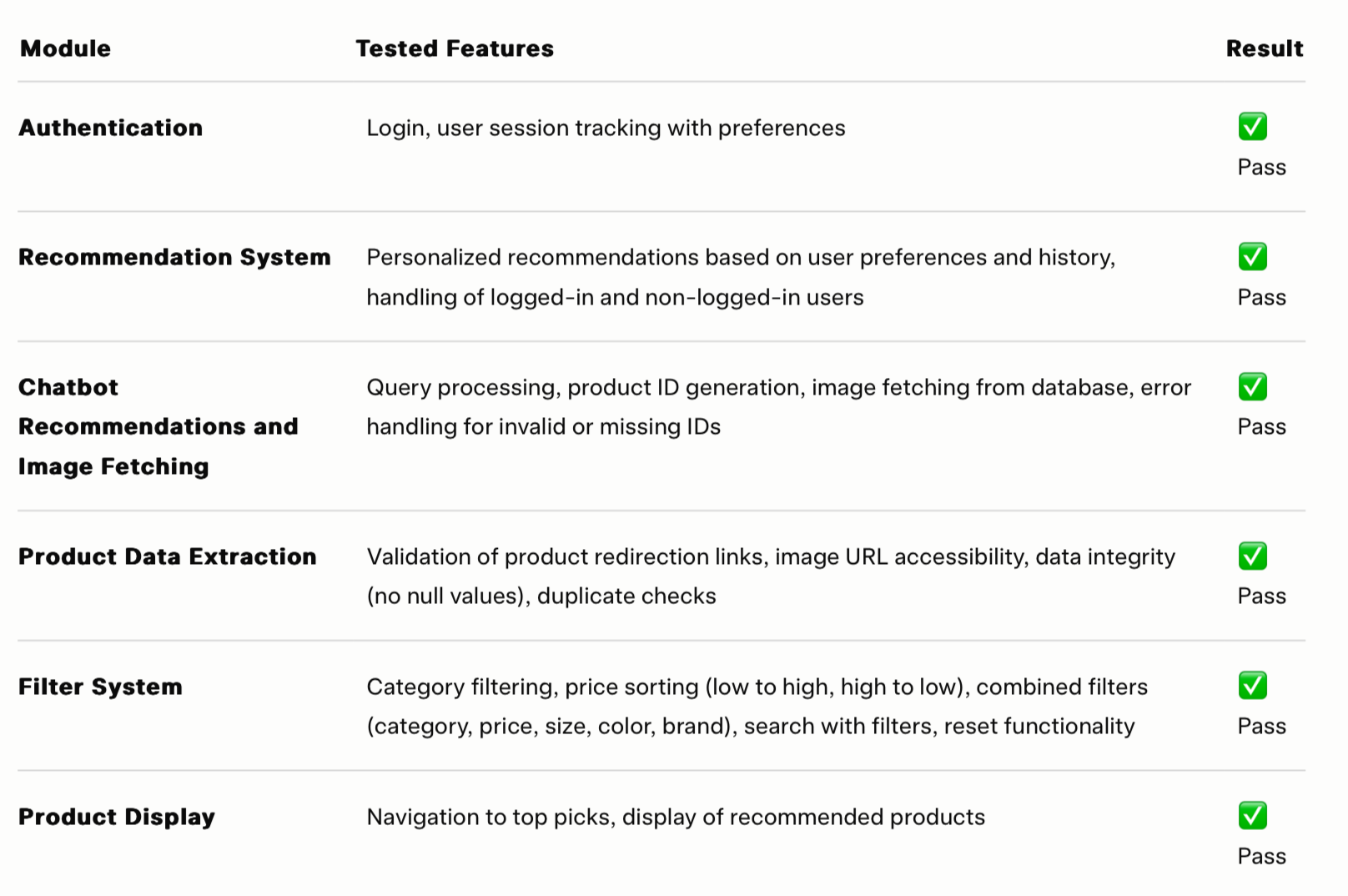
These user features collectively enhance the usability, performance, and personalization of the ShopSavvy platform. They form the core interaction layer between users and the backend intelligence systems, ensuring a seamless shopping experience.

**Chapter 4**

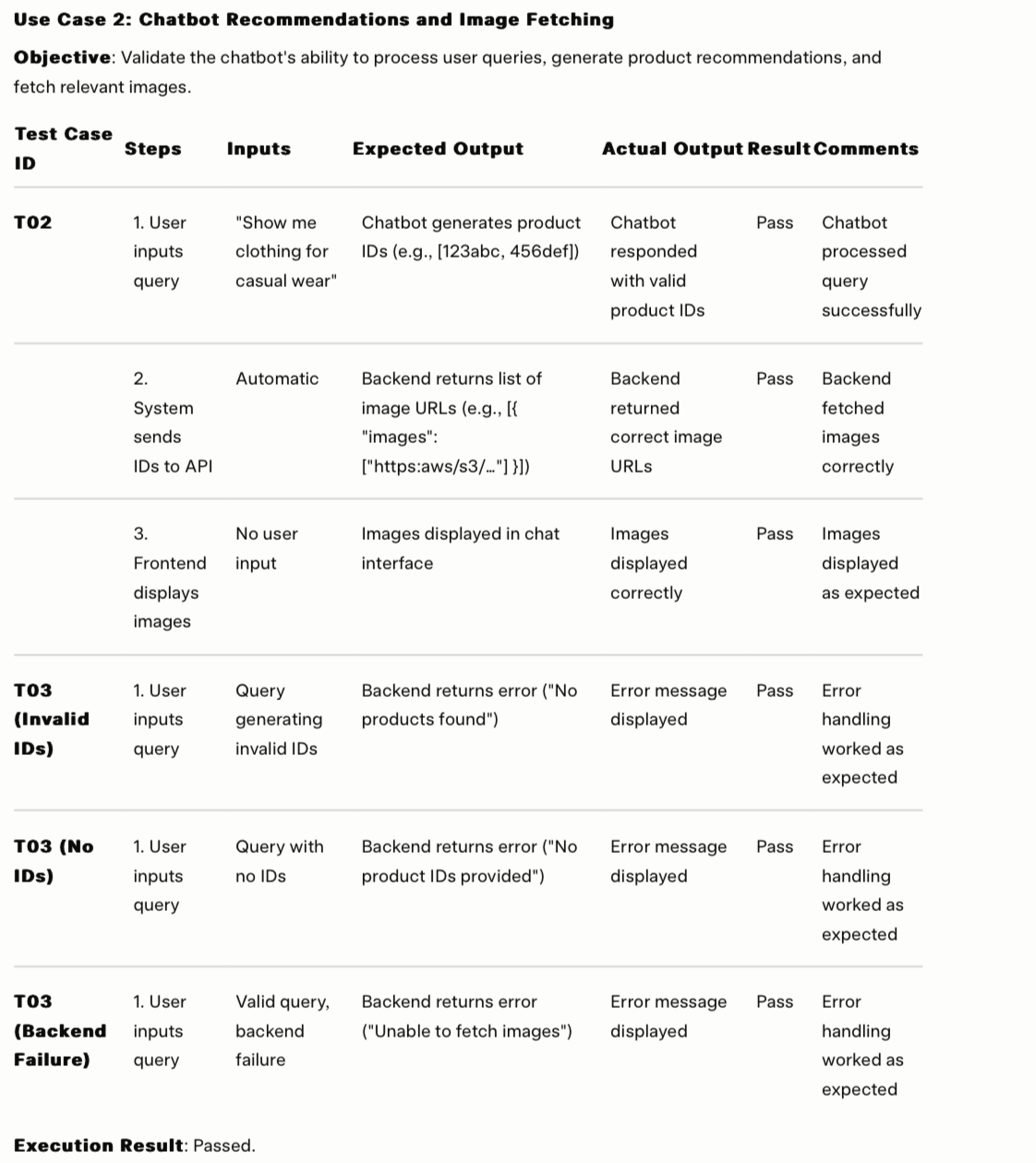
# Testing and Evaluation

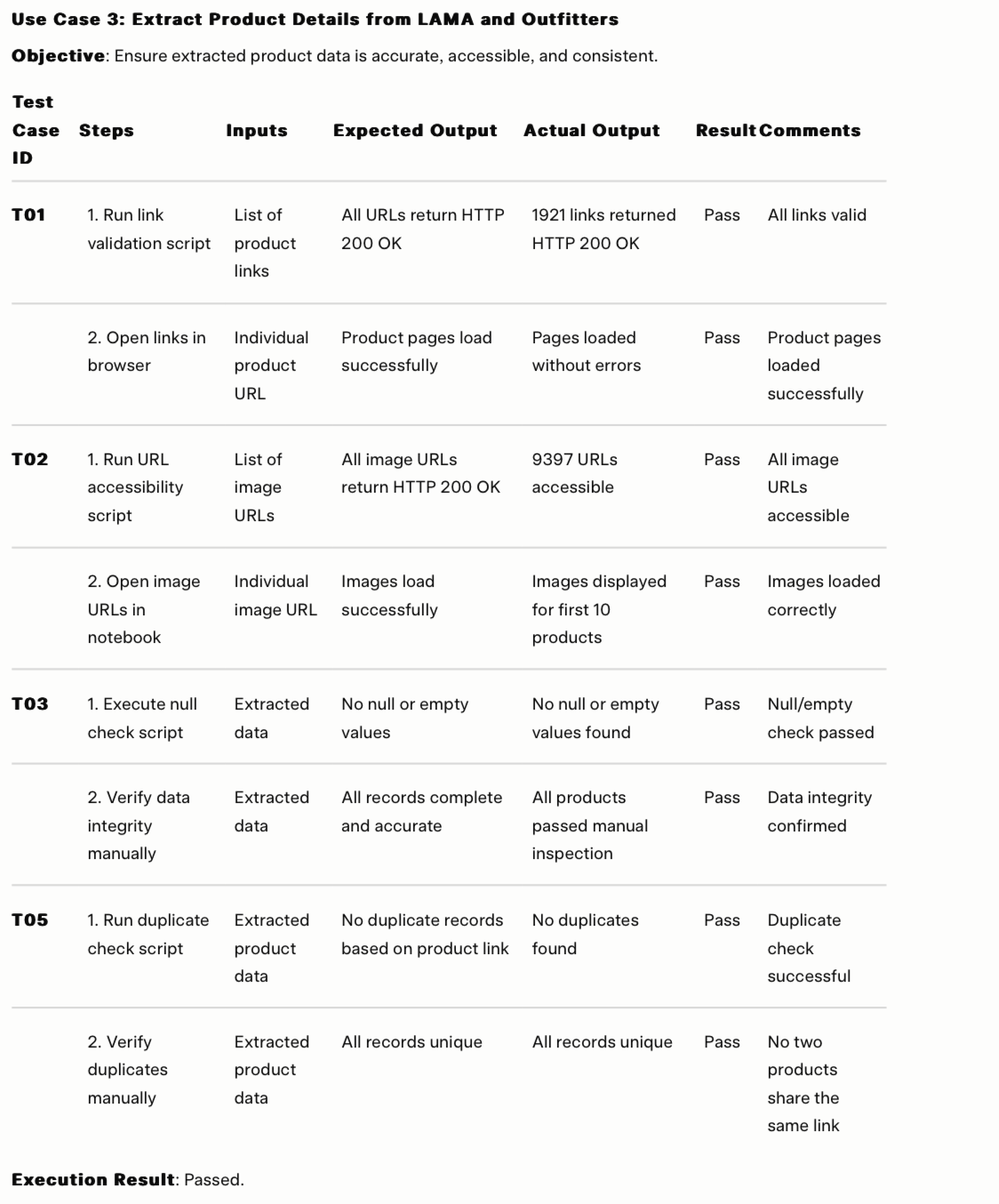
## Functional Testing

To ensure system reliability, each feature of Shop Savvy was rigorously tested using a combination of **manual exploratory testing** and **unit testing** (where applicable). Below is a breakdown of key modules tested:











## Performance Considerations

Although Shop Savvy is an early-stage prototype, several steps were taken to ensure efficiency:

* **Frontend Optimization**:
  + Image components were loaded lazily.
  + Products were paginated (not rendered all at once).
  + State updates were minimized using React’s efficient re-render logic.
  + Image URLs were stored in MongoDB schemas instead of binary data, and the images were hosted on AWS S3 for faster, scalable delivery.
* **Backend Efficiency**:
  + Indexes were created in MongoDB for faster filter-based lookups.
  + The recommendation API computes scores on-the-fly using in-memory logic for active sessions.
* **Scraper Runtime**:
  + On average, scraping each brand took under 10 seconds per page.
  + Broken links and timeouts were logged, with retry logic where feasible.
* **API Response Times**:
  + Most backend endpoints responded under 300ms in local/staging environments.

## Deployment

The *Shop Savvy* full-stack system was deployed using free-tier services to accommodate budget constraints. Below is an overview of the deployment setup, with each component explicitly detailed, followed by challenges encountered and their resolutions.

## Deployment Setup

* **Frontend**: Hosted on **Vercel** – leveraging globally distributed static assets for fast delivery of the user interface.
* **Backend**: Hosted on **Render** – hosting the core API endpoints for the application logic.
* **Chatbot**: Hosted on **Render** – powering the query processing and recommendation generation functionality.
* **Data Scraping**: Hosted on **Render** – managing the extraction of product details from LAMA and Outfitters.
* **Recommendation System**: Hosted on **Render** – handling personalized product suggestions based on user preferences and history.
* **Database**: **MongoDB Atlas** – utilized for cloud-based storage of product data, user preferences, and recommendation history.

**Deployment Challenges:**

* **CORS Issues**: The Vercel hosted frontend encountered blocked requests to the Render hosted backend, chatbot, data scraping, and recommendation system APIs due to Cross Origin Resource Sharing (CORS) restrictions.
  + **Solution**: Configured explicit CORS middleware on the Render hosted services to allow requests from the Vercel hosted frontend.
* **Free-Tier Cold Starts**: The Render free-tier services (backend, chatbot, data scraping, and recommendation system) exhibited slower response times for the first request after periods of inactivity, impacting initial user experience.
* **Frontend-Backend Sync Issues**: Mismatches in API versioning between the Vercel frontend and Render hosted backend, chatbot, and recommendation system caused occasional failures in responses.
  + **Solution**: Ensured consistent versioning across deployments and added automated checks to validate API compatibility before going live.

## User Feedback

Informal user testing was conducted with 6 classmates and friends who represented the target audience (18–25-year-old online shoppers). Their feedback included:

* **Positive**:
  + “Feels local and personalized—cool to see local brands aggregated.”
  + “Wishlist and filters work smoothly.”
  + “Chatbot is a fun addition—even when it doesn’t fully understand, it feels helpful.”
* **Suggestions for Improvement**:
  + Add more brands (e.g., Generation, Ethnic).
  + Include product sizes or availability.
  + Enable social login for convenience.

These insights will guide future enhancements and validate the market potential of the idea.

**Chapter 5**

# Conclusions & Future Work

## ****Summary of Achievements****

The **Shop Savvy** project successfully addressed the core challenges in the Pakistani online retail industry by aggregating fashion products from multiple local brands and offering a personalized, AI-driven shopping experience. Key achievements include:

* **Product Aggregation**: The web scraping module successfully aggregated and standardized product data from major Pakistani fashion brands like Outfitters and LAMA, ensuring a continuously updated product pool.
* **MERN Stack Implementation**: A full-stack web application was developed using the MERN stack (MongoDB, Express, React, Node.js), ensuring efficient data handling and a smooth user experience.
* **Real-Time Recommendation System**: The recommendation engine dynamically suggests products based on user interactions, including filters, searches, wishlist items, and browsing behavior, offering personalized shopping suggestions.
* **Chatbot Integration**: The GPT-3.5-powered chatbot assists users in discovering products, understanding filters, and providing general shopping assistance, enhancing the overall shopping experience.
* **User Features**: Authentication, filters, wishlist, and pagination were implemented, providing a seamless experience for authenticated and unauthenticated users alike.

Overall, **Shop Savvy** has laid the foundation for a smarter, more personalized fashion shopping experience for Pakistani consumers, addressing key pain points like fragmented shopping, lack of personalization, and absence of intelligent assistance.

## ****Limitations****

Despite its success, the **Shop Savvy** platform has some limitations that should be addressed in future iterations:

* **Brand Coverage**: Currently, only Outfitters and LAMA are supported for scraping. Expanding the number of brands is essential to provide users with a more comprehensive shopping experience.
* **Chatbot Accuracy**: While the chatbot performs basic functions well, it struggles with more complex queries or nuanced user requests. Improvements in natural language processing and better training on domain-specific queries would enhance its performance.
* **Real-Time Data Updates**: While scraping is automated, there are occasional delays in product updates on the site, especially after new products are added by brands. A more robust scraping schedule or webhooks for real-time updates would ensure fresh product data at all times.
* **User Interaction Data**: Currently, the recommendation system relies primarily on user interactions within a session. Long-term user behavior and history could be used to enhance the accuracy of the recommendations.
* **Mobile Optimization**: Although the platform is fully functional on desktop, additional optimization for mobile devices could improve accessibility and user experience, especially considering the mobile-first nature of e-commerce in Pakistan.

## ****Future Improvements and Expansion****

To elevate **Shop Savvy** and provide even greater value to users, the following future improvements and expansions are recommended:

* **Brand Expansion**: To increase the platform's relevance and appeal, more local brands should be integrated. This could include popular fashion names such as Generation, Ethnic, and Sapphire.
* **Advanced Filtering Options**: Implementing more granular filters, such as product sizes, color options, availability, and customer reviews, would significantly enhance the shopping experience.
* **Size & Availability Features**: Including product size information and real-time availability from the brands would allow users to make more informed purchasing decisions.
* **Social Login Integration**: Allowing users to sign up and log in using social media accounts (e.g., Facebook, Google) would improve user convenience and streamline the onboarding process.
* **Chatbot Enhancement**: The chatbot’s conversational abilities can be refined by incorporating more advanced NLP techniques, allowing it to handle a wider range of queries more accurately.
* **Mobile Application**: A dedicated mobile app for **Shop Savvy** could provide better performance, offline access, push notifications for deals, and an overall smoother experience tailored to mobile users.
* **User Analytics and Insights**: Integrating a user analytics dashboard could provide insights into consumer behavior, allowing for more refined recommendations and data-driven decisions for future enhancements.
* **E-Commerce Integrations**: Future versions of **Shop Savvy** could include direct purchasing options, allowing users to checkout directly from the platform, potentially forming affiliate partnerships with the brands.
* **AI-Powered Visual Search**: Leveraging AI to allow users to upload images of clothing and find similar items on the platform could provide an even more engaging shopping experience.

By focusing on these areas of improvement, **Shop Savvy** can continue to grow, offering users an increasingly tailored and efficient shopping experience. The future of fashion e-commerce in Pakistan holds significant promise, and **Shop Savvy** is poised to play a key role in its evolution.

# References

# Appendix

## Sample Scraping Output

Below is a sample scraping output representing a single product entry extracted from the LAMA Retail website. This structured data showcases the result after applying our scraping pipeline, which collects, cleans, and standardizes product information such as name, price, sizes, colors, images, and other relevant attributes. This sample illustrates how each product is stored in our system-ready format for further processing and recommendation.

|  |  |
| --- | --- |
| **Field** | **Sample Valie** |
| \_id | 676a3b1463fe6ff887c1dc80 |
| product | Spray Wash Tee |
| price | 3450 |
| colors | ["Charcoal", “Light Grey”] |
| sizes | ["S", "M", "L", "XL", "XXL"] |
| primary\_color | Charcoal |
| link | <https://lamaretail.com/collections/man-t-shirts/products/spray-wash-te…> |
| images | [Array of 6 image URLs] |
| type | T-Shirt |
| gender | Men |
| filtercolor | Grey |
| brand | LAMA |
| status | valid |

## Sample Recommendation System Output

[

{

"product\_id": "605c72ef153207d16c0b7b91",

"product": {

"name": "Wireless Bluetooth Headphones",

"category": "Electronics",

"price": 129.99,

"image\_url": "<https://example.com/images/product1.jpg>",

},

"final\_cosine\_score": 0.92

},

{

"product\_id": "605c72ef153207d16c0b7b92",

"product": {

"name": "Smartphone Case - Red",

"category": "Accessories",

"price": 19.99,

"image\_url": <https://example.com/images/product2.jpg>,

},

"final\_cosine\_score": 0.89

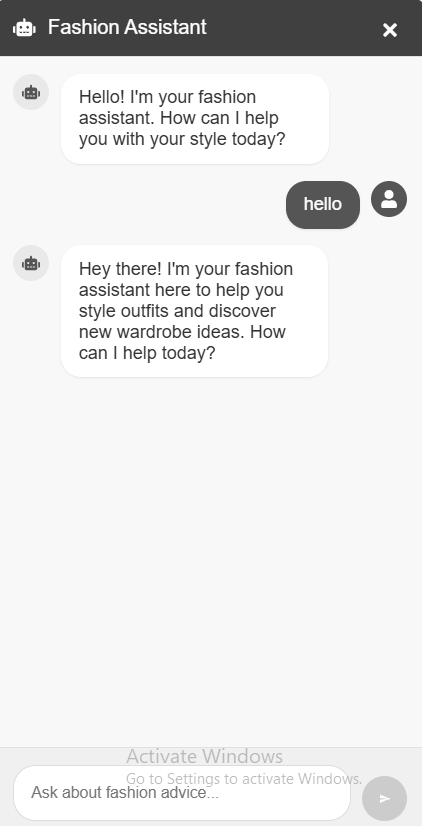
},

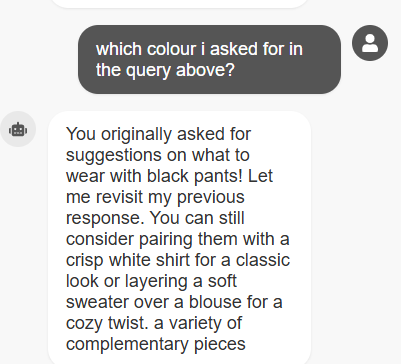
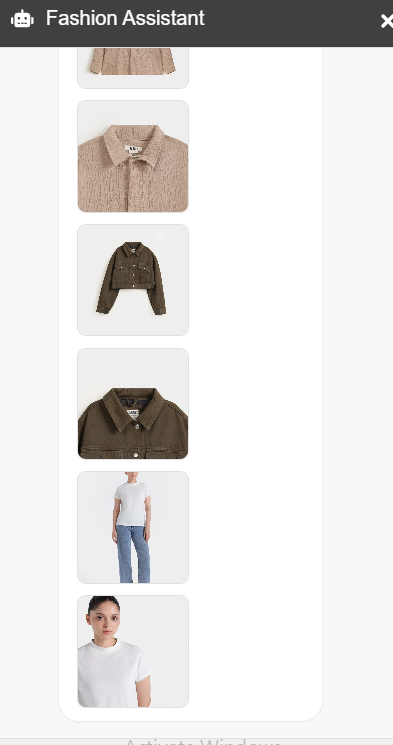
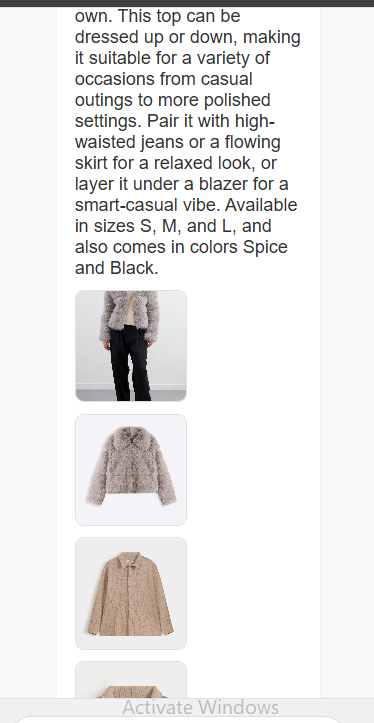
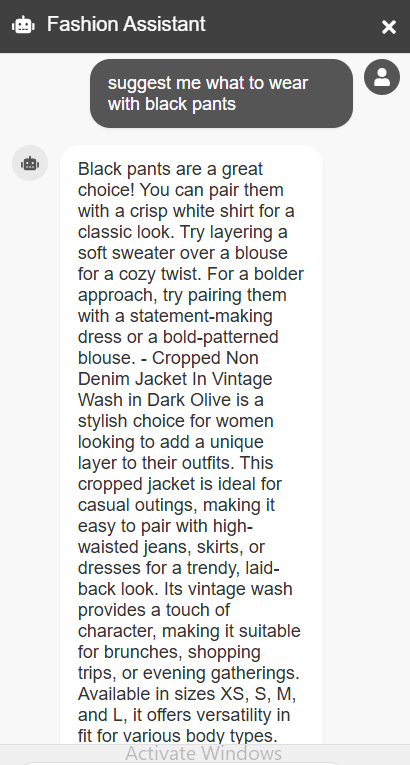
// Additional products (total 100)...

]

* **product\_id**: A unique identifier for each product.
* **product**: An object containing details about the product, such as:
  + **name**: Name of the product.
  + **category**: The product’s category (e.g., Electronics, Accessories).
  + **price**: The price of the product.
  + **image\_url**: A URL pointing to an image of the product.
* **final\_cosine\_score**: A cosine similarity score that represents how relevant the product is to the user’s preferences, with higher scores indicating more relevant recommendations.

## Chatbot Prompt Flow / API Sample





Chatbot handles the greeting and general queries by discriminating between them and fashion queries. It is tailored to give suggestions and also stores previous history to give personalized and better responses.

## MongoDB Schema Diagrams

