

PrimePicks: Your Perfect Shot at Personalized Picks!

Kinjalk Srivastava Meet Oswal Aryan Rai

(Code: <https://github.com/MeetOswal/Big-Data-Project>)

Introduction:

In today's age of information overload, staying updated with relevant news can be challenging. PrimePicks aims to revolutionize how users interact with news by providing a highly personalized and intuitive recommendation platform. Built on a robust tech stack that leverages advanced machine learning, natural language processing, and scalable backend architecture, PrimePicks ensures that every user gets curated, meaningful, and relevant content tailored to their preferences.

PrimePicks is a personalized news recommendation platform designed to address the challenges of information overload by delivering tailored, meaningful content to users. The platform aggregates news articles from diverse APIs, utilizes the BART large language model for concise and accurate summarization, and stores both the summaries and their associated metadata, including keywords, in a MongoDB cluster. Article embeddings, generated for efficient similarity searches, are stored in a Qdrant vector database. A separate user database tracks login details, explicit and hidden preferences, reading history, and dynamically computed user embeddings, ensuring a continually refined recommendation process. For search functionality, PrimePicks employs a Retrieval-Augmented Generation (RAG) model that synthesizes summaries of related articles and provides URLs for the top results, delivering comprehensive insights for user queries. The platform's backend is built on Flask, with a React-powered frontend ensuring a seamless user experience. MongoDB and Qdrant form the dual-database architecture, while Kafka facilitates real-time data streaming and Celery handles asynchronous tasks such as embedding computations and periodic updates. Together, these technologies enable PrimePicks to provide highly personalized news recommendations and an innovative search experience, marking it as a cutting-edge solution in personalized content delivery.

Overview of the Architecture:

1. News API

This project utilizes the News API as the core data source for fetching news articles and associated metadata. This API offers comprehensive and up-to-date information on global events, making it an ideal choice for building a personalized news recommendation platform. This API allows the system to query news articles based on concepts, keywords, location, and other parameters, ensuring relevance and diversity in the retrieved content.

In the project, the API is used to implement multiple features, including fetching events and extracting associated concepts for news curation. The *get_events* function queries the API for recent events using parameters such as source location (*United States*) and sorting criteria (e.g., by date or popularity). The *set_concepts* function extracts high-scoring concepts from these events, forming the basis for keyword-driven article retrieval. Subsequently, the *fetch_articles* function leverages these concepts to query news articles that match specific criteria, such as language, source type, and geographic region. The API's capability to filter duplicates and prioritize high-quality sources further enhances the relevance and reliability of the data.

This API is integral to the data pipeline, enabling the platform to source real-time and diverse content, which is then processed, summarized, and stored for personalized recommendations.

2. Flask

Flask is a lightweight and flexible web framework for Python, making it an excellent choice for building the backend of our news recommendation platform. Its minimalistic design allows for the straightforward definition of routes, logic, and middleware, as seen in endpoints like */recommendations/*, */login*, and */register*, enabling rapid development and iteration. Flask provides built-in session management to track user states, such as managing logged-in users (*user_id*) and maintaining session continuity. It integrates seamlessly with external libraries, such as Kafka for producing messages in an event-driven architecture, Celery for handling background tasks like updating user history asynchronously, and MongoDB (via *pymongo*) for managing persistent data. Designed to support RESTful APIs, Flask facilitates interaction between the backend and the React-based frontend, leveraging routes for core functionality. Scalability is enhanced through extensions like Flask-CORS, which enables secure cross-origin communication, and session configurations (*SESSION_COOKIE_SAMESITE*, etc.) that ensure secure and scalable user sessions. Flask's debugging tools, such as its built-in reloader, streamline error-catching during development, making rapid iteration more efficient. While Flask itself is synchronous, it supports asynchronous workflows by integrating with tools like Celery to offload time-intensive tasks, ensuring user requests are processed without delays. This combination of flexibility, integration capabilities, and ease of development makes Flask an ideal choice for our project.

3. React

React is employed as the frontend framework for this project due to its robust component-based architecture, efficient state management, and dynamic rendering capabilities. Its modular structure enables the development of reusable and maintainable components, such as *LoginPage*, *CreateUser*, *AddKeywords*, and *Search*, streamlining both the implementation and scalability of the platform. React's state and lifecycle management through hooks like *useState* and *useEffect* facilitates seamless data handling and user interaction, such as capturing user feedback, fetching recommendations, and real-time updates. The virtual DOM optimizes rendering performance, ensuring a responsive interface even during frequent user actions like

navigating recommendations or submitting feedback. Integration with *axios* for API communication supports backend interactions, including user authentication, feedback submission, and dynamic data retrieval. Furthermore, React Router provides a structured and intuitive navigation experience across multiple views, enhancing the user experience. With its support for custom styling through CSS and its extensive ecosystem of tools and libraries, React delivers an interactive, scalable, and efficient frontend solution aligned with the goals of this research platform.

4. MongoDB

MongoDB plays a crucial role in managing articles and user-related data for the platform. The *articles* collection is designed to store all fetched news articles along with their metadata, including summarized content generated by the *Model* class, titles, URLs, publication dates, keywords, and source locations. Additionally, article embeddings computed for similarity-based searches and recommendations are stored in MongoDB to facilitate seamless integration with the Qdrant vector database. This collection enables advanced querying based on metadata fields like keywords and dates, supporting multiple recommendation strategies such as keyword-based and vector-based approaches. In parallel, the *user* collection manages user data, including authentication details (email and password), user-selected and hidden preferences, interaction history, and dynamically updated embedding vectors. These embeddings represent users' evolving interests, supporting highly personalized recommendations and ensuring alignment with individual behavior through adaptive profiling.

MongoDB also houses the *keywords* collection, which plays a critical role in tracking keyword metrics, maintaining real-time updates, and enabling keyword-based recommendations. The *last_24_hours* field logs hourly updates on keyword scores, helping the platform identify trending topics and provide context-aware recommendations. MongoDB's robust querying capabilities support the platform's recommendation pipelines by fetching articles based on user-selected or inferred keywords, incorporating date-based filtering, and leveraging aggregation pipelines for operations like calculating recency scores and ranking articles. These capabilities ensure efficient retrieval and enable the platform to deliver personalized and dynamic content tailored to user interests.

MongoDB is integral to the data injection pipeline and its integration with backend services. Articles fetched from the NewsAPI are stored in the *articles* collection after preprocessing and summarization, while keywords extracted from these articles or user interactions are dynamically updated in the *keywords* collection. MongoDB also serves as an intermediate store for embeddings, supporting vector-based similarity searches in Qdrant. Seamlessly integrating with Flask via the *MongoConnection* class, MongoDB simplifies operations across backend endpoints such as */recommendations/* for fetching article recommendations, */register* for managing user profiles and preferences, */update-user-activity* for logging interactions and feedback, and */search-keyword* for retrieving relevant keywords. This integration ensures a

cohesive and efficient data management process, enabling the platform's robust functionality.

5. Qdrant

Qdrant serves as the vector database for the project, specializing in managing high-dimensional embeddings to support advanced similarity searches and personalized recommendation pipelines. The embeddings, generated using the *Model* class, capture the semantic meaning of articles and are stored in Qdrant for efficient retrieval. By leveraging Qdrant's cosine similarity metric, the platform performs content-based and user-based recommendations, matching user preferences or queries with semantically similar articles. This enables the system to deliver accurate and contextually relevant suggestions tailored to user interests. Qdrant seamlessly integrates with MongoDB and backend services through the *QdrantConnect* class, which abstracts operations like collection creation, data upserts, and vector queries. This integration supports dynamic data flow, ensuring that embeddings and metadata remain synchronized between the databases. Additionally, Qdrant enhances the recommendation pipelines by enabling hybrid approaches that combine vector similarity with metadata filtering, such as prioritizing articles by recency or relevance. Its scalability and efficiency make it an ideal choice for managing the growing repository of embeddings, ensuring real-time performance even as the dataset expands. With its ability to dynamically handle large-scale vector operations and facilitate seamless integration with other components, Qdrant is a cornerstone of the platform's architecture, enabling advanced, scalable, and personalized content delivery.

6. Celery: Asynchronous Task Execution

Celery is employed in the project as a task queue to handle long-running or resource-intensive operations asynchronously. Tasks such as updating user activity, recalculating keyword scores, and injecting data into MongoDB and Qdrant are delegated to Celery workers, ensuring that the Flask application remains responsive. For instance:

- **User Activity Updates:** Tasks like *update_user_history* and *update_user_keyword_score* process feedback, adjust user embeddings, and maintain interaction logs without interrupting the user's workflow.
- **Data Pipeline Management:** Operations like fetching articles, summarizing content, and generating embeddings are offloaded to Celery, allowing for batch processing and periodic updates.
- **Scheduled Tasks:** Celery's periodic task scheduling, configured through the *celery.conf.beat_schedule*, automates recurring operations such as refreshing keyword data or injecting new content into the database. This asynchronous execution model ensures that background tasks are handled efficiently, enabling scalability and a smooth user experience.

7. Kafka

Kafka is utilized as the message broker for Celery and serves as a backbone for real-time communication between microservices. Its high-throughput, distributed design ensures reliable and scalable message passing. Key functionalities include:

- **Task Distribution:** Kafka acts as the intermediary for queuing tasks from the Flask application to Celery workers, ensuring task reliability and fault tolerance.
- **Event-Driven Architecture:** Kafka enables real-time event streaming, such as user interactions triggering asynchronous updates to keywords or embeddings.
- **Scalability and Load Handling:** Kafka's distributed nature ensures that the platform can scale horizontally by adding more partitions or brokers, making it well-suited for handling the growing demands of the system. By integrating Kafka, the platform achieves robust communication between components, ensuring that tasks are executed consistently and promptly, even under heavy load.

Combined Utility of Celery and Kafka

- **Scalability:** The integration ensures that tasks can be distributed across multiple workers and handled concurrently, accommodating increasing workloads.
- **Reliability:** Kafka's message persistence guarantees that tasks are not lost during system failures, while Celery ensures retries for failed tasks.
- **Real-Time Processing:** Kafka streams events to Celery workers in near real-time, allowing the system to process user interactions, update data, and refresh recommendations dynamically.
- **Seamless Integration:** Celery and Kafka are seamlessly integrated with the Flask application, enabling efficient task management for endpoints like */update-user-activity* and */fetch-data*.

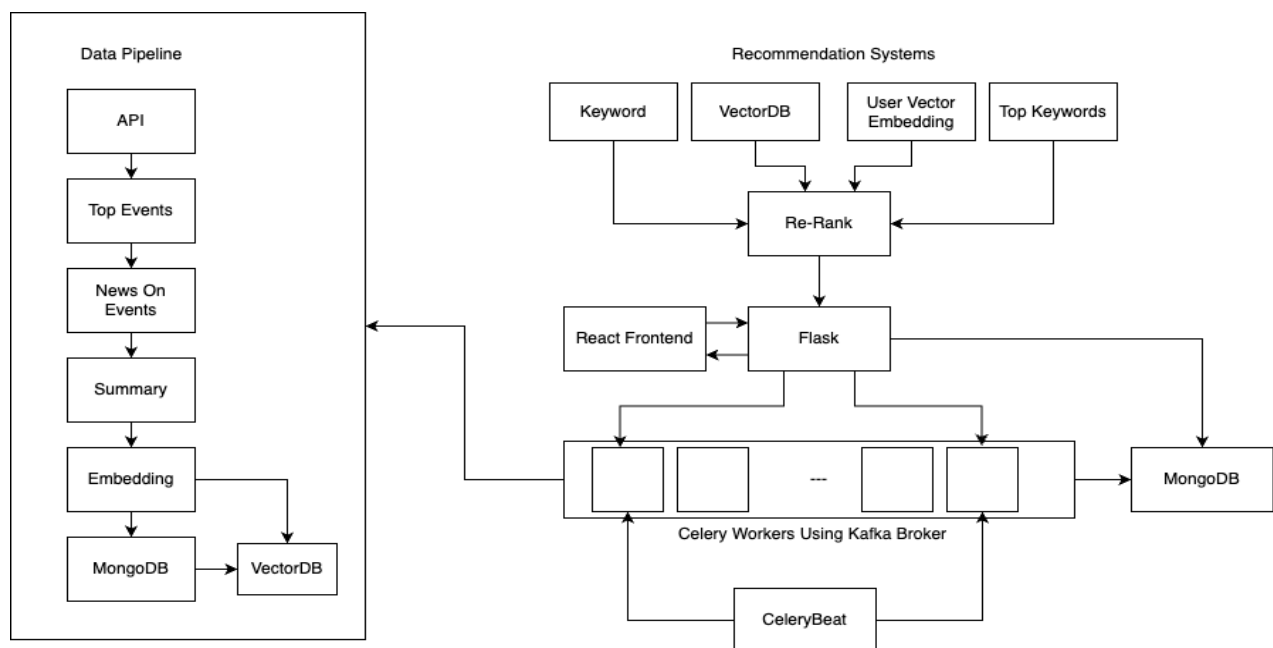


Figure 1: Overview of Application Architecture

Application Workflow

The application workflow is designed to provide a seamless and responsive user experience by integrating data ingestion, processing, and recommendation generation through a robust and scalable architecture. The workflow comprises distinct yet interconnected components that enable real-time data updates, efficient user interaction handling, and personalized content delivery. Below is an overview of the application's workflow:

1. Data Ingestion and Preprocessing:

The platform fetches news articles from the NewsAPI, filtering them based on parameters such as concepts, keywords, and locations to ensure relevance. The raw articles are then processed using the BART language model to generate concise summaries that capture the essence of the content. Each summarized article is further transformed into high-dimensional embeddings, which encode semantic relationships for similarity-based operations. These processed datasets, including summaries, metadata (titles, URLs, publication dates), and embeddings, are stored in MongoDB for structured querying and management. Simultaneously, the embeddings are inserted into Qdrant for efficient vector-based similarity searches. The dataset undergoes a structured transformation pipeline, ensuring both storage systems are synchronized and optimized for their respective roles in supporting personalized recommendations and real-time search capabilities.

2. User Interaction and Preference Management:

Users engage with the platform through actions like logging in, registering preferences, and exploring recommended content. Explicit preferences, chosen by users as keywords, and hidden preferences, inferred from their interactions, are stored and continuously updated in MongoDB. Additionally, the platform tracks interaction history, feedback scores, and session details, using this data to dynamically refine each user's embedding vector. This evolving representation of user interests ensures that the recommendations remain personalized, relevant, and aligned with their changing preferences over time.

3. Recommendation Generation:

The recommendation engine employs a multi-faceted strategy to deliver personalized and context-aware suggestions. It integrates keyword-based filtering, vector-based similarity searches, and popularity-driven insights to create a comprehensive recommendation pipeline. For vector-based recommendations, Qdrant is queried to retrieve articles with embeddings that are semantically similar to the user's embedding vector or search queries, leveraging high-dimensional data to understand nuanced content relationships. Simultaneously, MongoDB is used to apply metadata-based filtering, such as narrowing results by publication date, keywords, or source relevance. These results are further refined through a hybrid ranking approach that evaluates user feedback, article recency, and semantic relevance. By combining these layers of analysis, the engine ensures that the recommendations not only reflect the user's explicit and inferred preferences but are also timely and of high quality, delivering a personalized and engaging experience.

4. Real-Time Task Management:

Background tasks, including updating user history, recalculating keyword scores, and refreshing article embeddings, are managed asynchronously to maintain the platform's responsiveness. These tasks are offloaded to Celery workers, which handle them independently of the main application flow. Kafka serves as the messaging backbone, ensuring reliable communication and seamless task distribution between the Flask application and Celery. This architecture enables the system to process time-intensive operations in parallel, ensuring scalability and uninterrupted user interactions, even during high-load scenarios.

5. Dynamic Frontend Interaction:

The React frontend interacts with the Flask backend through RESTful APIs, facilitating real-time data updates and smooth navigation across components like keyword search, user registration, and recommendation browsing. User actions, such as submitting feedback on articles or updating preferences, are seamlessly handled and processed asynchronously. This ensures that the platform remains highly responsive, delivering a fluid and uninterrupted user experience while backend operations execute in parallel.

6. Periodic Updates and Scalability:

Scheduled tasks play a crucial role in maintaining the platform's relevance and accuracy by periodically updating keyword trends and refreshing article datasets. This ensures that the recommendations and search results align with current user interests and emerging content. The architecture, built on scalable technologies like MongoDB, Qdrant, Celery, and Kafka, provides the capacity to efficiently manage increasing data volumes and user activity. These components work together to deliver a robust and responsive system, ensuring consistent performance and adaptability as the platform scales.

Comprehensive Overview of Some Key Components:

1. Recommendation Systems and Re-ranking:

a. Keyword-Based Recommendations

Keyword-based recommendations form the foundation of the system by leveraging user-selected and hidden preferences stored in MongoDB. These preferences are dynamically updated based on user interactions and are used to query articles that share similar keywords. We update each keyword score as:

Updated score = original-score + (0.001) * (0.35 * read-time + 0.20 * reaction + 0.45 * clicked-url)

The system uses `$in` queries to identify articles matching these preferences, ensuring relevance and alignment with user interests. To enhance precision, the most significant preferences are selected through sorting and slicing operations, focusing on the top-ranked keywords. This approach is particularly effective for delivering personalized content when user preferences are explicitly known.

b. Vector Database Recommendations

The vector-based recommendation system enhances personalization by utilizing semantic embeddings generated from user interactions and articles. Scores are calculated for user history, factoring in feedback scores and exponential decay to prioritize recent and relevant articles.

$$\text{Score} = e^{-0.15(\Delta t)} \times (0.35 * \text{read-time} + 0.20 * \text{reaction} + 0.45 * \text{clicked-url})$$

Where,

Δt = Today's date - Date of Article Release

read-time = user read-time in seconds

Reaction = like(1) or neutral(0) and clicked-url = 1 or 0

These articles' embeddings are compared with stored vectors in Qdrant to identify similar content. The cosine similarity metric is employed to retrieve semantically related articles while filtering results based on recency using metadata constraints. This approach ensures that recommendations capture nuanced user interests and deliver contextually relevant suggestions.

c. User Vector Recommendations

User vector recommendations focus on leveraging the dynamically updated *userVector*, which represents the user's evolving preferences. This embedding is derived from the user's interaction history, capturing the semantic essence of their interests. We update user's vector using formula:

$$\text{Updated-vector} = (0.9) * \text{Original Vector} + (0.1) * \text{article vector} * (0.35 * \text{read-time} + 0.20 * \text{reaction} + 0.45 * \text{clicked-url})$$

Qdrant is queried with this embedding to retrieve articles with similar vectors, generating recommendations that are closely aligned with the user's overall behavior and historical patterns. This strategy ensures a holistic and adaptive recommendation system that evolves as the user interacts more with the platform.

d. Popularity-Driven Recommendations

The popularity-driven recommendation system identifies trending content based on keyword popularity metrics stored in MongoDB. The *keywords* collection tracks read time over time, with updates logged hourly to capture dynamic trends. Popular keywords are queried to fetch associated articles, prioritizing those with high relevance and recency. This method provides a balanced perspective by surfacing widely popular and contextually relevant content, making it especially effective for users without well-defined preferences or for delivering broadly appealing recommendations.

Reranking Mechanism

After generating a list of candidate articles from the four recommendation strategies, a reranking mechanism refines the results to ensure optimal quality and relevance. Articles are deduplicated, and their scores are computed based on a combination of factors, including cosine similarity with user history embeddings, recency scores, and user feedback.

Article-score = $e^{-0.15(\Delta t)}$ / cosine similarity with user's last 10 history.

This hybrid approach prioritizes articles that align with the user's preferences, are semantically similar to their interaction history, and are contextually recent. The reranking process ensures that the final recommendations are diverse, personalized, and reflective of both explicit and implicit user interests, delivering a curated experience tailored to the individual.

2. RAG Based Search:

The platform integrates a Retrieval-Augmented Generation (RAG) system to provide users with a powerful and context-aware search experience. This approach combines the strengths of information retrieval and generative language modeling, enabling the system to synthesize concise yet comprehensive responses while linking users to relevant resources for further exploration. Below is a detailed breakdown of how the RAG-based search operates:

a. Query Embedding and Vector Search

When a user submits a search query, the platform generates an embedding for the query using the BART model's encoder. This embedding captures the semantic intent of the query, ensuring that it aligns with the conceptual representations of stored article embeddings. Qdrant is then queried using this vector to retrieve articles with high semantic similarity to the query. The vector search identifies the top-matching articles by calculating cosine similarity between the query vector and stored article embeddings, ensuring that results are contextually relevant and conceptually aligned.

b. Article Summarization and Context Compilation

Once the top-matching articles are retrieved, their summaries are combined to create a cohesive context. These summaries, which are precomputed and stored in MongoDB, are concatenated into a single block of text that provides a comprehensive overview of the query-related content. This context serves as input for the generative model, ensuring that the generated response is both informative and aligned with the retrieved articles.

c. Generative Response Creation

The concatenated context is fed into the BART model's decoder to generate a natural language response. The model synthesizes this input into a coherent summary or

explanation tailored to the user's query. By leveraging the context from multiple articles, the system ensures that the generated response is not only concise but also captures diverse perspectives and essential details.

d. Linking to Additional Resources

To complement the generated response, the system provides direct links to the original articles retrieved from Qdrant. These URLs are included as part of the output, allowing users to delve deeper into the sources for more detailed information. This dual approach—summarization and linking—ensures that users receive both a high-level overview and access to in-depth content.

3. Scalability of the Platform: An In-Depth Analysis

Scalability is a cornerstone of the platform's design, ensuring that it can handle increasing user loads, larger datasets, and growing computational demands without compromising performance. The platform's architecture is built with scalability in mind, leveraging a combination of distributed systems, asynchronous processing, and robust database solutions to adapt seamlessly to dynamic requirements. Below is a comprehensive exploration of how scalability is achieved and validated.

a. Distributed Architecture

The platform employs a distributed architecture that separates concerns across its various components:

- Frontend: The React-based frontend communicates with the backend using RESTful APIs, ensuring that client-side processing is lightweight and offloads heavy computation to the backend.
- Backend: The Flask-based backend integrates Celery and Kafka to manage asynchronous task execution and message distribution. This ensures that background operations, such as updating user embeddings or fetching data, do not interfere with real-time user interactions.
- Databases: MongoDB and Qdrant handle structured metadata and high-dimensional embeddings, respectively. Their independent roles and optimized query capabilities ensure efficient data retrieval and storage even under high loads.

b. Task Management and Real-Time Processing

The use of Celery for asynchronous task management and Kafka for real-time messaging allows the platform to scale horizontally. Tasks such as data ingestion, user activity updates, and keyword scoring are processed in parallel across multiple workers. This design ensures that the system remains responsive even during peak usage, as time-consuming operations are offloaded to the background.

c. Database Scalability

- MongoDB: As the primary database, MongoDB's ability to shard data and handle high-throughput queries supports the platform's scalability. Collections such as *articles*, *user*, and *keywords* are optimized to handle dynamic updates and queries without performance degradation.
- Qdrant: Qdrant's vector database is designed to efficiently store and retrieve high-dimensional embeddings. Its scalability is achieved through partitioning and indexing, enabling fast vector similarity searches even as the dataset grows.

d. Scalability Validation: Local Deployment Example

To validate the platform's scalability, it was deployed on a local Wi-Fi network with limited computational resources. During testing, the system successfully handled concurrent interactions from more than 10 devices, each performing a mix of operations such as fetching recommendations, updating preferences, and conducting RAG-based searches. Despite the shared network constraints, the platform maintained low latency, demonstrating its ability to scale horizontally and manage concurrent user activity effectively. This real-world scenario underscores the robustness of the platform's design and its readiness for larger-scale deployments.

e. Scalability-Enabling Features

- Asynchronous Processing: By offloading tasks like data updates and embedding generation to Celery workers, the system reduces bottlenecks and enhances throughput.
- Message-Driven Architecture: Kafka's distributed messaging system ensures reliable and scalable communication between components.
- Elasticity: The system can easily accommodate additional resources, such as more Celery workers or database replicas, to handle increased loads.
- Load Balancing: The architecture supports deployment with load balancers to distribute traffic evenly across backend servers.

Working

1. Homepage after Login

PrimePicks

Logout

Search Keyword

Search

Title: Manchester United Tipped to Sell £40M-Rated Star Amid PSG Transfer Rumors

Summary:Paris Saint-Germain have reportedly submitted their first bid for Manchester United's Marcus Rashford. The Ligue 1 club could revisit their pursuit of Rashford next year. Manchester United are reportedly willing to sell Rashford for around £40 million. Former England striker Rob Earnshaw has voiced concern about Rash

URL:URL

Next

Like

This page displays a curated list of recommended articles, offering users the ability to engage with the content by liking the articles or clicking on their respective URLs for detailed reading. The platform tracks the time spent on each article and records user interactions, such as likes and URL clicks. This data is then sent to the backend, where it is processed to calculate a feedback score for each article. These feedback scores are dynamically integrated into the recommendation engine, refining future suggestions to better align with user preferences and behavior.

2. Keyword Search

Select Your Favorite Keywords

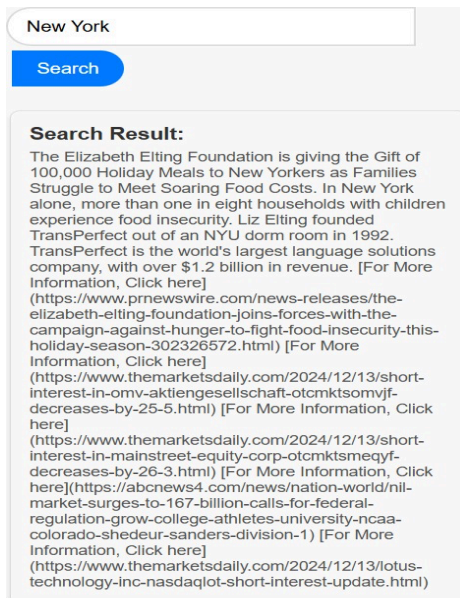
New York City

Find Keyword

<input type="checkbox"/>	New York City
<input type="checkbox"/>	New York City Subway
<input type="checkbox"/>	New York City Fire Department
<input type="checkbox"/>	New York City Police Department
<input type="checkbox"/>	New York Daily News
<input type="checkbox"/>	New York City Economic Development Corporation
<input type="checkbox"/>	New York Bight
<input type="checkbox"/>	Binghamton, New York

This page enables users to search for keywords they wish to add to their personalized list of preferences, which directly influences the recommendations generated for them in the future. The search functionality is enhanced with an advanced fuzzy search mechanism, allowing for up to two edits—such as minor spelling errors, missing characters, or typos—while still retrieving relevant results. This flexibility ensures that users can easily refine their keyword preferences without worrying about perfect accuracy in their search queries. By integrating these keywords into the user profile, the platform dynamically adapts to better align with individual interests, delivering more precise and meaningful recommendations over time.

3. RAG Based Search



New York

Search

Search Result:

The Elizabeth Elting Foundation is giving the Gift of 100,000 Holiday Meals to New Yorkers as Families Struggle to Meet Soaring Food Costs. In New York alone, more than one in eight households with children experience food insecurity. Liz Elting founded TransPerfect out of an NYU dorm room in 1992. TransPerfect is the world's largest language solutions company, with over \$1.2 billion in revenue. [For More Information, Click here] (<https://www.prnewswire.com/news-releases/the-elizabeth-eling-foundation-joins-forces-with-the-campaign-against-hunger-to-fight-food-insecurity-this-holiday-season-302326572.html>) [For More Information, Click here] (<https://www.themarketsdaily.com/2024/12/13/short-interest-in-omv-aktiengesellschaft-otcmktsomvjf-decreases-by-25-5.html>) [For More Information, Click here] (<https://www.themarketsdaily.com/2024/12/13/short-interest-in-mainstreet-equity-corp-otcmktsmeqyf-decreases-by-26-3.html>) [For More Information, Click here] (<https://abcnews4.com/news/nation-world/nfl-market-surges-to-167-billion-calls-for-federal-regulation-grow-college-athletes-university-ncaa-colorado-shedeur-sanders-division-1>) [For More Information, Click here] (<https://www.themarketsdaily.com/2024/12/13/lotus-technology-inc-nasdaqlot-short-interest-update.html>)

This image demonstrates how a new article is generated by synthesizing summaries extracted from MongoDB, specifically selecting those most closely related to the user's query. In addition to the synthesized summary, the platform provides summaries of the top five related articles, offering users a deeper dive into the topic for additional context and information. For instance, in the given example, the query "New York" yields results highly relevant to New York, showcasing the system's ability to deliver precise and contextually aligned content based on the user's input.

Conclusion

PrimePicks successfully demonstrates the power of integrating cutting-edge technologies to address the challenges of information overload. By combining personalized recommendation strategies, real-time updates, and dynamic search capabilities, the platform provides a seamless and engaging user experience. The innovative use of MongoDB and Qdrant enables efficient storage and retrieval of both structured metadata and high-dimensional embeddings, while the combination of Flask, Celery, and Kafka ensures scalability and responsiveness. Through features like Retrieval-Augmented Generation (RAG), multi-layered recommendation pipelines, and periodic updates, PrimePicks is a robust solution for personalized content delivery. The system's successful scalability tests further validate its readiness for broader deployment, highlighting its ability to manage increasing user demands and dataset growth efficiently. PrimePicks stands as a testament to the potential of interdisciplinary approaches in creating transformative digital solutions.

Limitations and Future Work

Limitations

1. Cold Start Problem: The system relies heavily on user interaction data to refine recommendations. New users with minimal activity may receive less personalized suggestions, impacting their initial experience.
2. Resource-Intensive Components: The use of large language models like BART for summarization and embedding generation requires substantial computational resources, which may limit real-time scalability for smaller deployments.
3. Dependency on API Quality: The platform's reliance on the News API means that its performance and diversity are inherently tied to the quality and comprehensiveness of the external data source.

Future Work

1. Enhancing Cold Start Solutions: Implement hybrid recommendation strategies that incorporate demographic and contextual data to provide initial personalization for new users.
2. Optimizing Computational Overheads: Explore lightweight alternatives to BART or integrate model distillation techniques to reduce resource requirements without compromising performance.
3. Expanding Data Sources: Integrate multiple news APIs and explore real-time scraping techniques to diversify and enhance the quality of available content.
4. Real-Time Recommendations: Transition to streaming-based recommendations to further reduce latency and enhance the responsiveness of the system in dynamic environments.
5. User-Centric Features: Add support for multilingual articles, advanced filters, and interactive dashboards to cater to a broader audience and enhance user engagement.

