CMPE 492

Enhanced Search Engine for Research Articles

Ceydanur Şen

Nurullah Uçan

Advisor:

H. Birkan Yılmaz

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Broad Impact

The Proxima project aims to transform how researchers discover, track, and engage with academic literature by addressing one of the most pressing challenges of contemporary scholarship: information overload. With millions of papers published each year, identifying research that is both relevant and conceptually novel has become increasingly difficult. Proxima addresses this challenge through an AI-driven semantic search and recommendation system designed to surface genuinely related work rather than relying on superficial keyword matching.

Unlike conventional keyword-based search engines, Proxima employs advanced natural language processing (NLP) and embedding-based similarity models to jointly interpret abstracts and keywords. This semantic approach enables the retrieval of conceptually aligned publications even when disciplines use divergent terminologies, thereby supporting deeper and more effective cross-disciplinary literature exploration. Search results can be ranked by semantic similarity or citation signals, and researchers can combine abstracts, keywords, and temporal filters to navigate large-scale scholarly corpora efficiently.

Proxima also supports a structured and proactive research workflow. Researchers can subscribe to specific queries and automatically receive newly published high-similarity papers, as well as relevant citing and cited works. This functionality helps scholars stay informed about emerging developments while maintaining focus on their core research themes.

Beyond individual productivity gains, Proxima supports the broader research ecosystem by promoting open-access discovery through its integration with OpenAlex. By prioritizing freely available publications and improving the relevance of literature recommendations across disciplines, it advances open science and enhances research visibility. In doing so, Proxima fosters interdisciplinary collaboration and accelerates the collective advancement of knowledge.

1.2. Ethical Considerations

Proxima is developed with an ethical approach that prioritizes responsible data use, algorithmic fairness, user privacy, and transparency. The platform relies on data from open-access sources such as OpenAlex and uses only publicly available metadata and abstracts in accordance with applicable licensing and intellectual property regulations. Copyrighted full-text content is neither stored nor processed, and any additional data sources are evaluated for license compliance before integration.

Proxima is designed with awareness of potential algorithmic bias. Its search and recommendation mechanisms aim to avoid overrepresentation of dominant disciplines or languages and to support balanced visibility across research areas. When evaluation mode is enabled, outputs from different retrieval approaches are compared to observe relevance differences and identify possible biases. These observations inform iterative improvements while preserving a streamlined and stable production experience. Transparency in how results are generated and differentiated further supports user trust.

User privacy is a core consideration in the system design. For subscription-based features, only minimal information—such as an email address and query terms—is collected and used solely for providing the requested functionality. Personal data is not shared with third parties, and evaluation-related feedback is handled without being linked to identifiable user profiles.

Through this approach, Proxima supports open-access discovery and equitable knowledge dissemination while maintaining ethical responsibility, respect for intellectual property, and a clear commitment to privacy and transparency.

2. PROJECT DEFINITION AND PLANNING

2.1. Project Definition

The Proxima project is an AI-driven semantic search and recommendation platform for academic literature. In production, it runs an embedding-based pipeline that interprets research abstracts and keywords to retrieve conceptually related papers beyond simple keyword matches. For internal assessment, an eval mode runs three pipelines side by side—embedding, raw OpenAlex keyword, and Gemini+OpenAlex keyword expansion—allowing users to rank results and provide feedback on relevance.

The system integrates OpenAlex to prioritize open-access discovery, supports year-range filtering, and offers sorting by similarity, publication year, or citation counts. Users can mark papers as "Good Match," subscribe to specific queries, and receive weekly automated emails with new high-similarity papers plus citing and cited works.

Built with a Django REST backend and a React/Vite frontend, Proxima combines natural language processing, multi-pipeline evaluation, recommendation systems, and subscription workflows to enhance research productivity, equitable access, and continuous improvement of literature discovery.

2.2. Project Planning

The project follows a modular and iterative development plan designed to enable parallel progress across the design, implementation, and evaluation stages. The system is organized into multiple phases encompassing user interface design, backend integration, data processing, and final evaluation. An interactive client-side interface framework is employed for user-facing components, while a server-side architecture manages query processing, data retrieval, and storage. Access to open-access research metadata is provided through an external academic data source.

- **Phase 1: Requirement Analysis and Design**

  Identification of system objectives and user requirements. Preparation of interface mockups and architectural diagrams to define core system components and data flow.

- **Phase 2: Interface Development**

  Implementation of the primary search and results interfaces. Emphasis is placed on responsive design and usability principles to ensure smooth and intuitive user interaction.

- **Phase 3: Backend and Database Integration**

  Development of the backend infrastructure responsible for query handling, data management, and integration with the external research database.

- **Phase 4: Search and Recommendation Logic**

  Integration of embedding-based similarity computation to rank relevant research papers. Incorporation of user-driven features such as "Good Match" tagging to support personalized discovery.

- **Phase 5: Subscription and Notification System**

  Implementation of the query subscription mechanism and automated email notifications for newly published or related research articles.

- **Phase 6: Evaluation and Validation**

  Assessment of system performance and usability, including comparison with existing academic literature search systems. The evaluation focuses on accuracy, relevance, and user experience metrics.

- **Phase 7: Testing and Documentation**

  Comprehensive testing of all system components, final debugging, and preparation of project documentation and results.

This structured development plan ensures systematic progress, maintainability, and the production of verifiable results throughout both the development and evaluation phases.

2.2.1. Project Time and Resource Estimation

The development process is planned to span approximately 13 weeks, covering all stages from requirement analysis to system evaluation. Each phase is allocated a proportional time frame based on its complexity and interdependencies, ensuring balanced progress across design, implementation, and testing. Frontend and backend components are developed in parallel to maintain integration consistency and enable continuous iteration throughout the project lifecycle.

In terms of resources, the project primarily utilizes the following components:

- **Software:** An integrated development environment (IDE) and project management tools are used for coding, testing, and version control. These tools ensure code integrity, modularity, and collaboration between the interface and system components.
- **Libraries and Frameworks:** The user interface is developed using a component-based frontend framework, while the backend employs a server-side structure responsible for managing data operations, API requests, and database interactions. This modular architecture supports scalability and maintainability throughout the development process.
- **Data Sources:** The system retrieves bibliographic metadata such as titles, authors, abstracts, and DOIs from an open-access research database. This ensures broad academic coverage and compliance with open science principles.
- **Hardware:** The project is designed to operate on a standard development environment with sufficient processing power and memory to support software implementation and testing. The development setup can be configured on any suitable platform, whether local or remote, depending on system requirements. This flexibility ensures scalability and sustainability across different environments.

This allocation of time and resources provides a balanced foundation for achieving both functional and non-functional requirements within the projected timeframe.

2.2.2. Success Criteria

The success of the Proxima project will be evaluated based on both functional and qualitative performance measures:

- The system successfully retrieves and ranks relevant academic papers based on semantic similarity rather than keyword overlap.
- The subscription mechanism operates reliably, enabling users to receive regular and accurate recommendation emails.
- The user interface provides a seamless, accessible, and responsive experience across devices.
- Comparative evaluation against two reference systems demonstrates measurable improvements in relevance and usability.

Successful completion will be defined by achieving these objectives while maintaining system stability, efficiency, and user satisfaction.

2.2.3. Risk Analysis

The main risks identified throughout the project are as follows:

- **Data Dependency:** The system relies on the OpenAlex API, and any downtime or schema changes may temporarily affect data retrieval.
- **Integration Complexity:** Synchronizing frontend and backend components may lead to inconsistencies if interface updates are not properly versioned.
- **Model Accuracy:** Semantic similarity computations depend on pretrained models, which may not always capture domain-specific nuances.
- **Email Delivery Reliability:** External factors such as spam filters or server delays can affect the success rate of email recommendations.
- **Time Constraints:** Concurrent workload in different development stages could extend the project timeline if dependencies are not well managed.

To mitigate these risks, backup mechanisms such as caching, version control, and

modular testing are incorporated into the development workflow.

2.2.4. Team Work

(i) **Team Composition:**

The **Proxima** project is being developed collaboratively by a two-member team:

- **Member A (Ceydanur Şen):** Responsible for backend development, DevOps configuration, and system architecture documentation.
- **Member B (Nurullah Uçan):** Responsible for user interface design, frontend implementation, DevOps and interaction design documentation.
- **Member A and B :** Responsible for research, documentation, and the implementation of advanced natural language processing (NLP) and embedding-based similarity models to analyze abstracts and keywords.

(ii) **Division of Responsibilities**

Area	**Responsible**	**Description**
Frontend Development	Member B	Design and implementation of the client-side application; ensuring responsive layouts and usability compliance.
Backend Development	Member A	Implementing the backend API, database integration, and recommendation logic.
DevOps and Deployment	Both	Configuration of the environments, CI/CD, security, and scalability.
UI/UX Design	Member B	Creating and refining mockups for unified search, results view, and subscription dashboard.
Research and Documentation	Both	Investigating embedding models, cosine similarity, and OpenAlex.
System Modeling	Both	Both members prepared the Use Case Diagrams and Sequence Diagrams

(iii) **Collaboration Process**

The team follows an iterative workflow:

(a) **Planning:** During this phase, the team identified the project objectives, clarified terminology, and defined the functional and non-functional requirements to guide subsequent development stages.

(b) **Design:** The design phase was organized into distinct sub-stages, and each deliverable was evaluated upon completion to verify its accuracy and alignment with the defined functional requirements.

(c) **Implementation:** The frontend and backend are being developed in parallel to ensure consistent integration and system coherence.

(d) **Testing and Integration:** Each member tests the units they implements, and at the end, both tests the overall functionality of the integrated features together.

(e) **Documentation:** Both contributed to diagrams and the reports.

(iv) **Communication and Tools**

- **GitHub:** Version control and issue tracking.

- **Overleaf / Google Docs / Crixet:** Report and requirements editing.
- **Zoom/ Google Meet:** Weekly coordination meetings.

(v) **Summary** The two-member structure of the project provides a clear division of responsibilities. Member B is responsible for UI/UX design and frontend development, whereas Member A focuses on backend implementation and both are responsible for DevOps. Both members actively collaborate on research and system modeling to ensure consistent progress toward a cohesive and well-integrated system.

3. RELATED WORK

Several AI-driven systems have been developed to enhance academic literature discovery beyond traditional keyword-based search engines. These systems generally combine semantic similarity modeling, citation network analysis, and visualization techniques to support more meaningful research exploration.

Connected Papers[1] constructs visual maps of related literature using three citation-based metrics: Bibliographic Coupling, Co-citation, and Temporal Dimension. This approach highlights conceptual and temporal relationships between papers, even in the absence of direct citations.

VitaLITy (Huang et al., 2021)[2] introduces a visual text analytics framework that uses transformer-based embeddings, such as Sentence-BERT and SciBERT, to measure semantic similarity between papers. It applies dimensionality reduction methods like UMAP and t-SNE to visualize research themes in a two-dimensional semantic space.

Semantic Scholar[3] , developed by the Allen Institute for AI, represents one of the most advanced AI-powered scholarly search platforms. It employs SPECTER and SPECTER2 models to generate high-dimensional semantic embeddings from abstracts and titles, incorporating both textual and citation-based context for improved relevance and recommendation quality.

Etymo (Cai & Yates, 2018)[4] focuses on constructing an adaptive similarity network combining semantic embeddings and citation graphs. The system continuously updates its relationships based on user interactions such as clicks and reads, forming a dynamic knowledge network.

[1]https://www.connectedpapers.com/
[2]https://arxiv.org/abs/2108.03366
[3]https://www.semanticscholar.org/
[4]https://arxiv.org/abs/1801.08573

Argo Scholar (Färber & Lamprecht, 2021)[5] builds an interactive scholarly knowledge graph using BERT-based embeddings and citation relationships, stored within a Neo4j database. Its D3.js-powered interface enables real-time visualization of research connections, authorship patterns, and topic evolution.

Google Scholar[6] , while the most widely used academic search engine, relies primarily on term weighting models such as TF-IDF and BM25, along with citation count and publication reputation. Although effective in ranking by impact, it lacks deep semantic understanding of research content.

PubMed Related Articles (Lin & Wilbur, 2007)[7] represents one of the earliest large-scale implementations of content-based similarity, using a probabilistic vector space model to identify semantically related biomedical articles based on TF-IDF and cosine similarity.

BASE (Bielefeld Academic Search Engine)[8] aggregates metadata from institutional repositories via OAI-PMH and indexes it with Apache Solr using TF-IDF/BM25 fielded search and recency boosts. Its focus on comprehensive coverage and metadata quality makes it a reliable classical baseline for academic search.

SciBERT[9] is a domain-specific variant of BERT pretrained on 1.14 million scientific papers. It improves token representation for scientific terminology and serves as a foundation for many downstream scholarly search and classification tasks.

Compared to these systems, **Proxima** integrates the strengths of semantic embedding models and open-access data (via OpenAlex) while focusing on user-centered personalization. It provides a unified framework for semantic search, dynamic recommendation, and subscription-based updates, bridging the gap between automated retrieval and individualized research curation.

[5]https://arxiv.org/abs/2110.14060
[6]https://scholar.google.com/
[7]https://pubmed.ncbi.nlm.nih.gov/
[8]https://www.base-search.net/
[9]https://github.com/allenai/scibert

3.1. Model Comparison and Evaluation of Text Embeddings

In this section, we compare the performance of four pre-trained models — **E5-large-v2 (E5)**, **BGE-large-en-v1.5 (BGE)**, **GTE-large-en-v1.5 (GTE)**, and **Nomic-embed-text-v1.5 (Nomic)** — in generating **text embeddings** for a set of academic abstracts. These embeddings are evaluated on their ability to capture semantic similarity between abstracts from different topics. We assess these embeddings by measuring their **cosine similarity**, performing **clustering** with **KMeans**, and evaluating the **AUC**, **F1 score**, and **Silhouette score** for clustering performance.

The models are evaluated in the following areas:

- **Cosine similarity** between pairs of abstracts.
- **Threshold-based evaluation** using **AUC**, **F1 score**, and **Youden's J** statistic.
- **Clustering** with **KMeans** and **Silhouette score**.
- **Top-K nearest neighbors** for each abstract.

3.1.1. Results

The embeddings were generated using each model and evaluated on the following metrics:

- **AUC** (Area Under the Curve)
- **Best F1 Score**
- **Silhouette score** for clustering quality
- **Adjusted Rand Index (ARI)** for clustering accuracy
- **Top-K nearest neighbors** for each abstract

3.1.2. E5-large-v2 (E5)

- **AUC**: 0.963
- **Best F1**: 0.667 (Threshold: 0.865)
- **Silhouette Score**: 0.142

- **ARI**: 0.645
- **Best-Youden-J**: 0.840

3.1.3. BGE-large-en-v1.5 (BGE)

- **AUC**: 0.956
- **Best F1**: 0.727 (Threshold: 0.794)
- **Silhouette Score**: 0.217
- **ARI**: 0.848
- **Best-Youden-J**: 0.654

3.1.4. GTE-large-en-v1.5 (GTE)

- **AUC**: 0.986
- **Best F1**: 0.857 (Threshold: 0.588)
- **Silhouette Score**: 0.230
- **ARI**: 0.848
- **Best-Youden-J**: 0.508

3.1.5. Nomic-embed-text-v1.5 (Nomic)

- **AUC**: 0.975
- **Best F1**: 0.667 (Threshold: 0.762)
- **Silhouette Score**: 0.235
- **ARI**: 0.558
- **Best-Youden-J**: 0.632

3.1.6. Visualizations

The following visualizations summarize the performance of the models:

- **Histogram of Cosine Similarity**: A histogram of the cosine similarity scores for positive vs. negative pairs, illustrating how well the models distinguish between

similar and dissimilar abstracts.

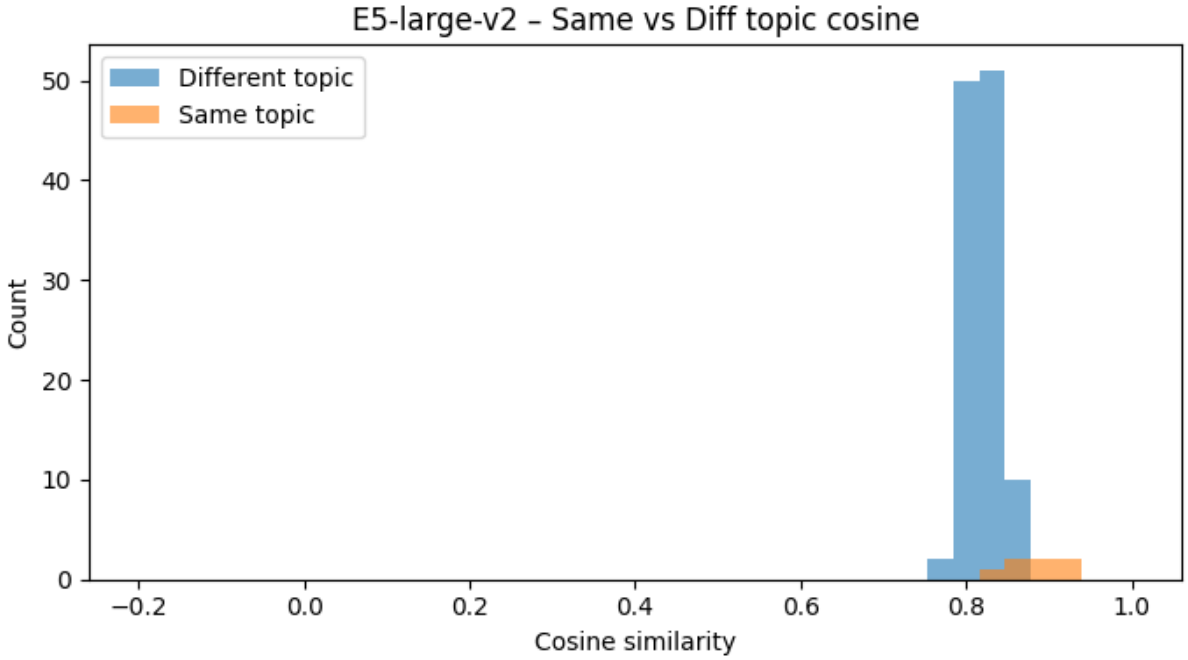

Figure 3.1. Cosine Similarity Histogram for Positive vs. Negative Pairs - **E5**

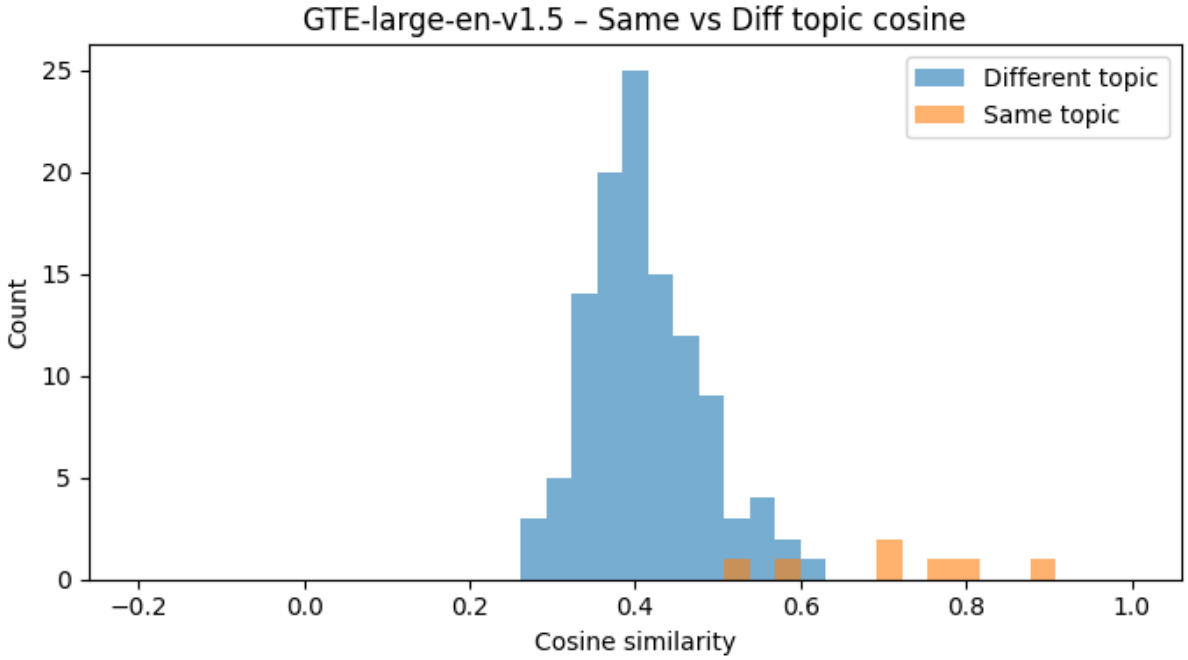

Figure 3.2. Cosine Similarity Histogram for Positive vs. Negative Pairs - **GTE**

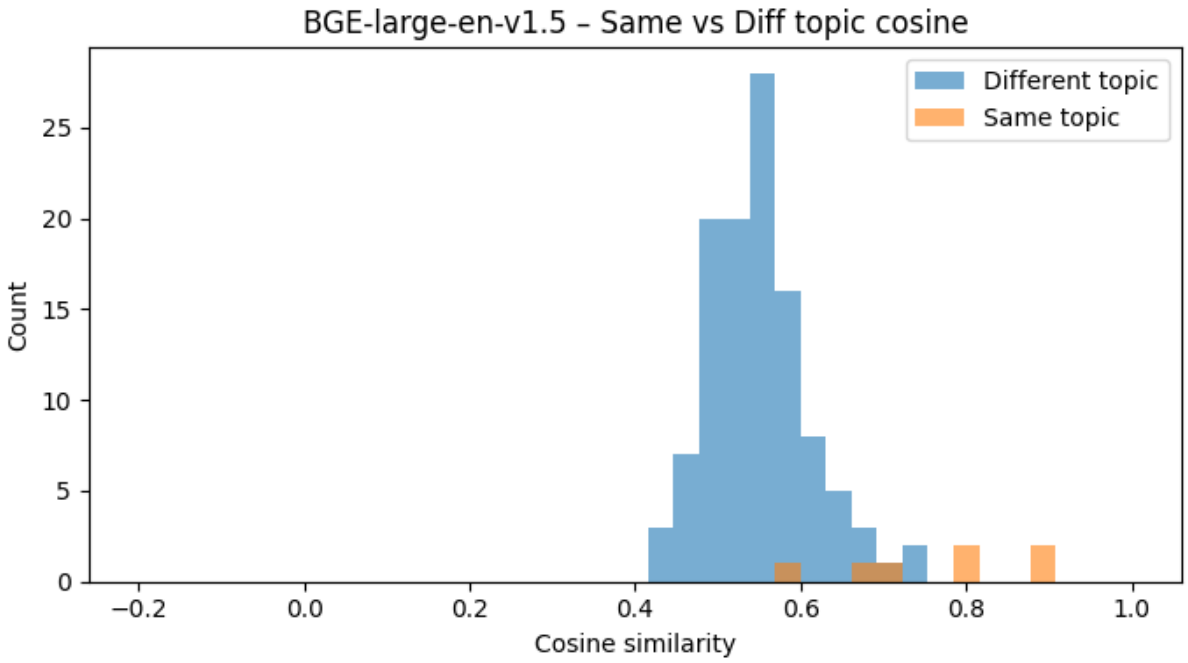

Figure 3.3. Cosine Similarity Histogram for Positive vs. Negative Pairs - **BGE**

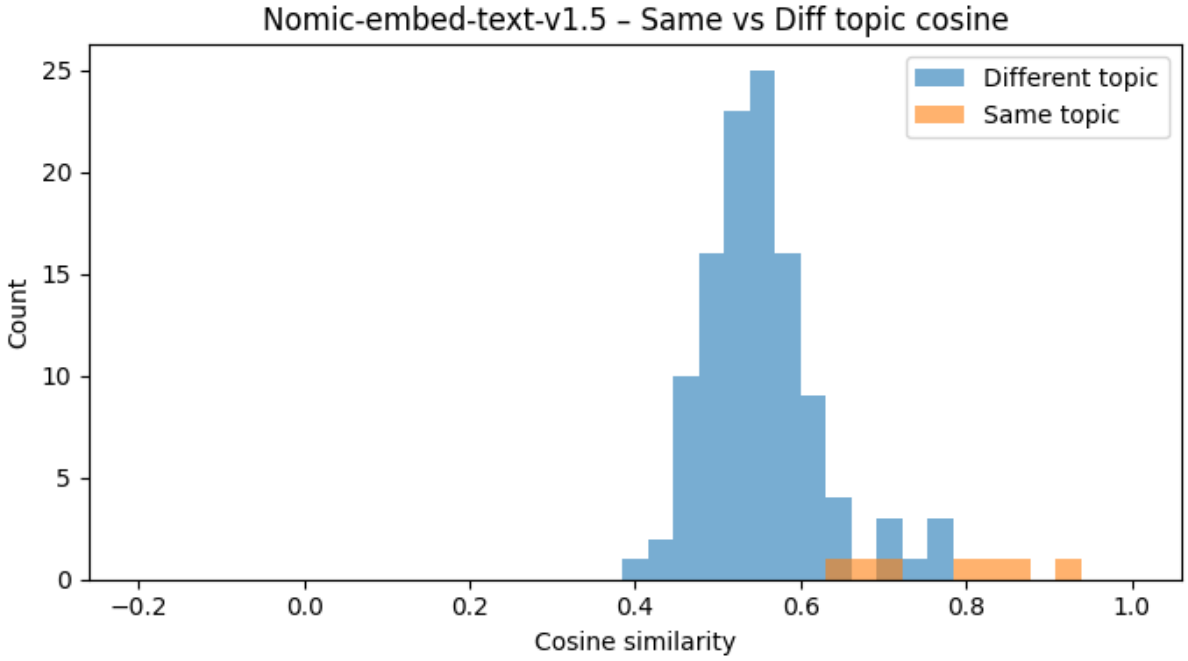

Figure 3.4. Cosine Similarity Histogram for Positive vs. Negative Pairs - **Nomic**

- **Cosine Similarity Heatmap**: A heatmap of the pairwise cosine similarity matrix for each model, sorted by labels to highlight clusters of similar abstracts.

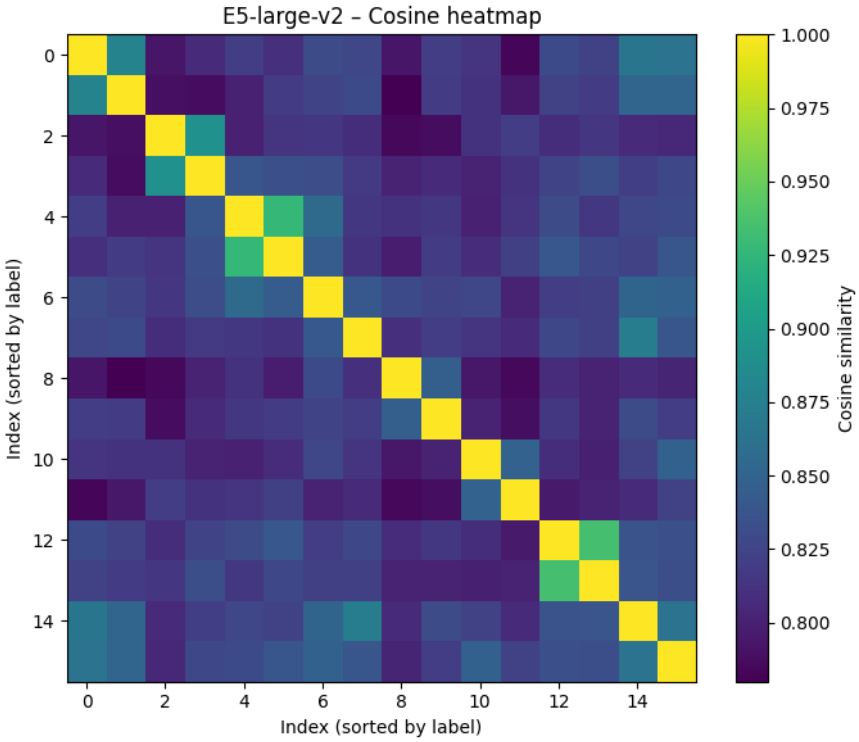

Figure 3.5. Cosine Similarity Heatmap - **E5**

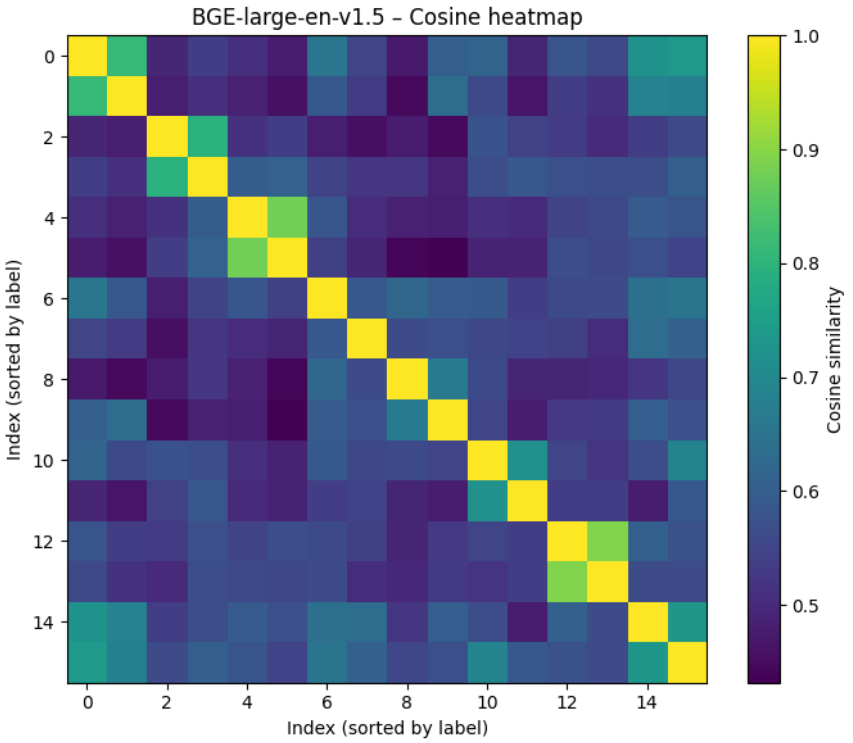

Figure 3.6. Cosine Similarity Heatmap - **BGE**

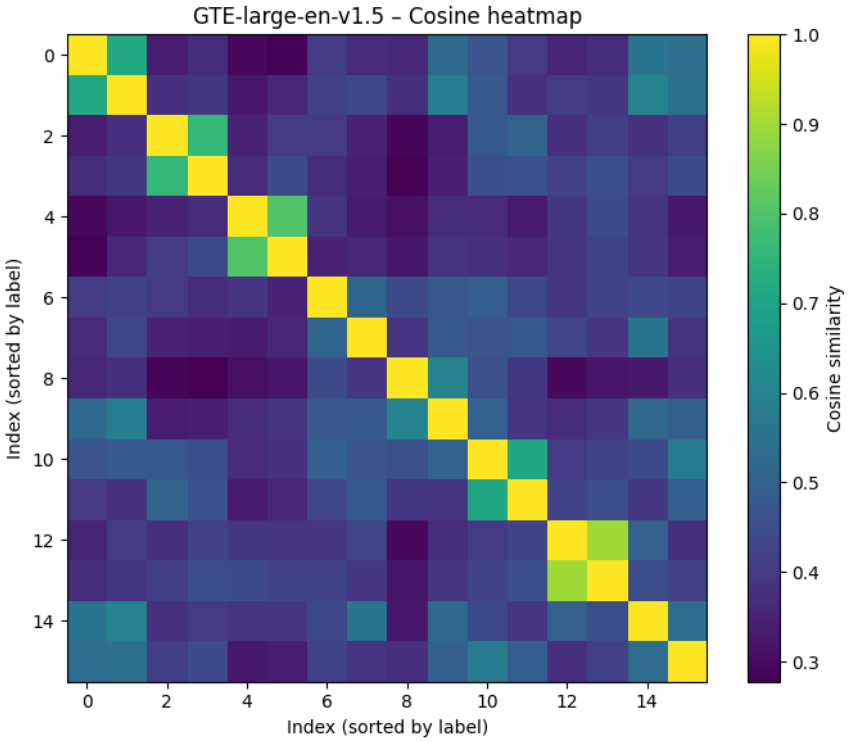

Figure 3.7. Cosine Similarity Heatmap - **GTE**

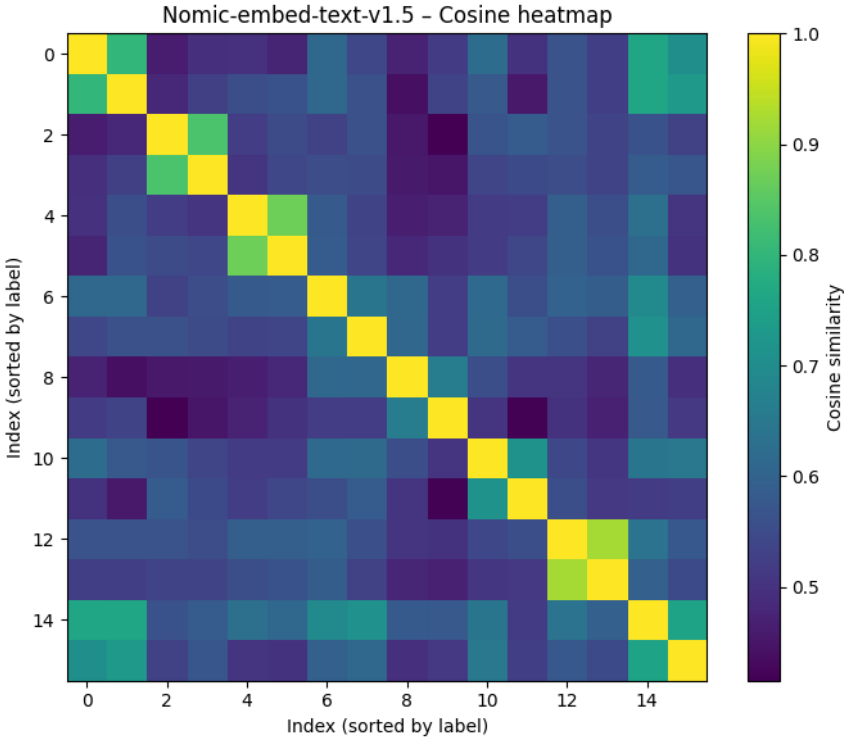

Figure 3.8. Cosine Similarity Heatmap - **Nomic**

3.2. Interpretation

Cosine Similarity - The **GTE** model outperforms all other models in distinguishing between similar and dissimilar abstracts, as indicated by its higher **AUC** and **F1 score**. - **Positive pairs** (abstracts from the same topic) show higher cosine similarity than **negative pairs** (abstracts from different topics), which is expected and reflects the model's ability to capture meaningful relationships between topics.

Threshold Evaluation - The **best F1 score** and **AUC** for **GTE** indicate that it is the most accurate model in distinguishing between similar and dissimilar pairs. - The threshold values for **F1 score** and **Youden's J statistic** indicate the optimal cut-off point for classifying pairs as similar or dissimilar, and **GTE** consistently produces the best results across these metrics.

Clustering - **GTE** produces the best **Silhouette score**, indicating that its embeddings allow for better separation of abstracts into meaningful clusters when using **KMeans**. - **E5** and **Nomic** show decent clustering results, but **BGE** has slightly weaker clustering performance, as evidenced by the lower **Silhouette score** and **ARI**.

3.3. Explanation

- **Why GTE Performs Well**: - **GTE** performs well because it uses a higher-dimensional embedding (8192-dimensional vectors), which allows it to capture more complex semantic relationships compared to models like **E5** and **BGE** that use 512-dimensional embeddings.

- **Cosine Similarity**: Cosine similarity is effective for comparing text embeddings because it measures the **angular distance** between vectors, making it well-suited for evaluating text embeddings where the magnitude may vary but the direction (or semantic meaning) remains consistent.

- **Clustering**: The higher **Silhouette score** of **GTE** indicates that its embeddings form distinct clusters, which is crucial for tasks like **topic modeling** or **semantic search**,

where we want to group similar topics together and separate dissimilar ones.

3.4. Evaluation

Performance of Different Models - **GTE** consistently outperforms the other models, achieving the highest **AUC**, **Silhouette score**, and **F1 score**. Its higher-dimensional embeddings provide better semantic separation of topics, which results in superior performance in clustering and similarity-based tasks. - **E5** and **BGE** are efficient but perform slightly worse than **GTE** in all metrics, likely due to their lower-dimensional embeddings. - **Nomic**, while slightly less performant than **GTE**, still provides strong results and is a good alternative for those requiring smaller embeddings.

Limitations - **GTE**'s larger embeddings are computationally more expensive than **E5** and **BGE**, which could be a consideration for scalability in large datasets or real-time applications. - The **AUC** and **F1 score** can be influenced by the specific dataset and task, and different domain-specific tasks may require further fine-tuning of the models.

3.5. Takeaways

- **GTE** is the top-performing model for **semantic search**, **clustering**, and **text similarity** due to its higher-dimensional embeddings and ability to capture more complex semantic relationships. - **E5**, **BGE**, and **Nomic** are still strong contenders, especially in situations where **computational efficiency** is a concern. - The embeddings produced by these models can be effectively used for **semantic search**, **topic modeling**, **classification**, and **recommendation systems** in academic literature and beyond.

4. METHODOLOGY

This chapter describes the technical methods and system architecture used in developing the Proxima project. The system combines natural language processing, semantic similarity computation, and web-based interaction to enhance academic literature discovery and personalization. The methodology consists of five main components: data retrieval, semantic processing, search and ranking, user interaction, and automated recommendation delivery.

4.1. System Overview

Proxima follows a modular client–server architecture, separating the user interface from the backend logic. The frontend provides an interactive environment for entering abstracts and keywords, while the backend handles query processing, semantic analysis, and communication with the external database. This architecture ensures maintainability, scalability, and independent testing of each component.

4.2. Data Retrieval

The system retrieves academic paper metadata such as titles, authors, abstracts, and publication years from the **OpenAlex API**. Each paper obtained from OpenAlex is represented as a semantic embedding vector generated by a pretrained language model. When the user submits one or more abstracts and keywords, these inputs are also transformed into vectors using the same embedding model to ensure semantic consistency between the user query and the dataset.

4.3. Semantic Processing

Proxima employs an **embedding-based semantic similarity model** to evaluate conceptual relatedness between the user's input and the retrieved papers. Both user-generated vectors and OpenAlex paper vectors reside in a shared semantic space. Cosine similarity is computed between these vectors to quantify conceptual closeness. This ap-

proach allows the system to match papers that are semantically related even when their vocabulary or phrasing differs significantly.

4.4. Search and Ranking

After calculating similarity scores, papers are ranked from highest to lowest based on their semantic proximity to the user's input. This ranking mechanism prioritizes conceptually aligned studies rather than simple lexical matches, improving relevance across interdisciplinary research areas. The top results are then displayed to the user through the web interface.

4.5. User Interaction and Subscription

The frontend allows users to view and manage search results interactively. Additionally, they may subscribe to a search query via a double opt-in email system by providing their name and email address. Sunbscribed users can manually mark the papers recommended via mail as **"Good Match"** for future reference. All subscription data is securely stored and used to deliver updated recommendations automatically over time.

4.6. Recommendation Delivery

A scheduled backend service periodically re-evaluates the user's saved queries using newly added papers from OpenAlex. The recommendation engine recomputes semantic similarity and compiles a ranked list of relevant new papers. An automated email generation module formats these papers—containing titles, abstracts, and links to OpenAlex—into a structured, user-friendly layout for weekly delivery. In addition to prioritizing newly published or newly indexed papers, the system also recommends older papers in a pre-scheduled manner to support deeper discovery beyond recent updates.

4.7. Error Handling and Validation

To maintain robustness, the system includes multiple validation and recovery mechanisms. Client- and server-side validation prevent invalid or duplicate submissions, and

modular testing isolates potential faults across components. Error messages are displayed clearly to guide user actions in case of invalid inputs or failed operations. These safeguards collectively improve reliability, reduce system errors, and ensure a consistent and smooth user experience.

5. REQUIREMENTS SPECIFICATION

5.1. Glossary

- **Guest** - A non-registered user who can perform searches (abstract or keyword) but cannot save queries or receive recommendation emails.
- **Subscriber** - A registered user who has confirmed their email and receives weekly personalized recommendations based on subscribed queries.
- **Abstract Search** - Function allowing users to find research papers by pasting one or more abstract texts.
- **Keyword Search** - Function allowing users to input multiple thematic tags in the same query form.
- **Unified Query** - A hybrid input combining abstracts and keywords, forming a single semantic search vector.
- **Subscribed Query** - A saved combination of abstracts and keywords generating weekly recommendations (replacing per-paper subscriptions).
- **Article Card** - Component displaying title, authors, venue, publication year, abstract snippet, and relevance score.
- **Cosine Similarity** - Metric measuring similarity between user query embeddings and paper embeddings.
- **Embedding Model** - Transformer-based model encoding text into vector space for semantic matching.
- **OpenAlex API** - External open-access academic database providing metadata on papers, authors, and journals.
- **Recommendation Engine** - Module ranking and suggesting new research papers to subscribers based on query similarity.
- **Unsubscribe Modal** - Confirmation dialog preventing accidental removal of a subscribed query.
- **Weekly Email** - Personalized HTML message summarizing top new papers related to each subscribed query.
- **Query Editor** - Interface for updating abstracts and keywords within an existing subscribed query.

5.2. Functional Requirements

5.2.1. User Requirements

5.2.1.0.1. Guest Features

- Guests shall perform searches using a single unified input that accepts both abstracts and keywords.
- Each query may contain multiple abstracts and multiple keywords simultaneously.
- The system shall generate a single embedding representation combining both input types.
- Search results shall be ranked by cosine similarity (0–1), converted into a percentage score.
- Each result card shall include title, authors, venue, year, citation count, and relevance score.
- Clicking the title shall open the OpenAlex page in a new tab.
- Guests shall not be able to subscribe to queries until email confirmation.

5.2.1.0.2. Subscriber Features

- Users shall subscribe by providing a query name and valid email address.
- Subscription shall follow a double opt-in verification process via email.
- Subscriptions shall store abstracts and keywords as part of a unified query.
- Weekly recommendations shall be based on semantic similarity to the subscribed query, not individual papers.
- Users shall manage their saved queries via the *My Subscribed Searches* page.
- Unsubscribe actions shall require explicit confirmation in a modal window.
- Successful unsubscription shall trigger a green "Subscription removed" alert.
- Subscribers shall receive weekly emails containing top relevant new papers.
- Recommendations shall be derived via cosine similarity between stored queries and new indexed papers.
- Each recommendation shall include title, year, authors, venue, similarity percentage,

and DOI link.

- The email should be sent to subscibed user including: *Weekly update for Saved search* and *Relevant From Archive*
- A button in mail labeled "Manage my subscriptions" shall link to the user's subscription dashboard.
- Users shall view all subscribed queries in a consolidated management dashboard.
- Each entry shall list abstracts, keywords, and Good Matches.
- Users shall edit existing queries through the *Edit Query Modal*, enabling updates to abstracts and keywords.
- Unsubscribing shall remove the entire query, not individual papers.

5.2.2. System Requirements

5.2.2.0.1. Main Page

- The main interface shall feature a unified form containing both Abstracts and Keywords fields.
- Abstracts shall support adding and removing multiple entries dynamically.
- Keywords shall be represented as removable tag elements.
- The layout shall be responsive and mobile-friendly.

5.2.2.0.2. Search Results

- Each paper result shall display metadata and similarity percentage.
- Each query shall include Subscibe to this query button for users to subscibe the search providing a unique query name and mail address.
- Results shall include badges for Open Access status.
- Data shall be fetched from the OpenAlex API, processed through the embedding model, and ranked.

5.2.2.0.3. Email Notifications

- Each email shall include the user's saved abstracts, keywords, and top matches.
- HTML templates shall be responsive (maximum width: 700 px).
- The footer shall include links to "View my subsciptions".

5.2.2.0.4. Subscription Management

- The *Manage your subscriptions* page shall display all active queries with the date of subscription.
- Users shall edit abstracts and keywords in place using a modal dialog.
- The *Unsubscribe Modal* shall confirm user intent before deletion.
- A success message ("Subscription removed") shall appear immediately after confirmation.

5.3. Non-Functional Requirements

5.3.0.0.1. Performance

- Search responses shall be delivered under standard conditions.
- The recommendation system shall update embeddings weekly.

5.3.0.0.2. Usability and Accessibility

- The UI shall follow HCI best practices for clarity, feedback, and consistency.
- Colors, contrasts, and font sizes shall comply with WCAG 2.1 AA standards.
- The design system shall use Tailwind CSS components.

5.3.0.0.3. Security and Privacy

- All user data shall be transmitted over HTTPS.

- Subscriber emails shall be encrypted with AES-256.
- The system shall comply with GDPR for data consent and deletion.

5.3.0.0.4. Scalability and Maintainability

- The backend shall scale to support at least 10,000 subscribers.
- API endpoints shall be modular and independently upgradable.
- Documentation and API reference shall be maintained in version control.

5.4. Use Case Diagram

In this chapter, the functional requirements of the system are presented. Figure 5.1 shows the overall use case diagram of the *Proxima – Research Article Search & Subscription* system. The diagram organises the behaviour of the system into four main function groups: *Search*, *Subscriptions*, *Weekly Recommendations*, and *GoodMatch*.

The *Search* group covers the unified search over abstracts and keywords and viewing individual papers via OpenAlex links. The *Subscriptions* group models the double opt-in flow, management of saved searches (token-based manage page), soft unsubscribe and reactivation of queries. The *Weekly Recommendations* group represents the background process that periodically composes, ranks and sends weekly e-mails based on the stored subscriptions. Finally, the *GoodMatch* group captures the feedback loop where users can mark recommended papers as a "good match" from e-mail links and later view or remove these saved items from the manage page.

Actors of the system are the *Guest* (who can perform searches without subscribing), the *Subscriber* (who can subscribe to queries, manage saved searches and good matches), and the background actor *System Scheduler*, which triggers the weekly recommendation job.

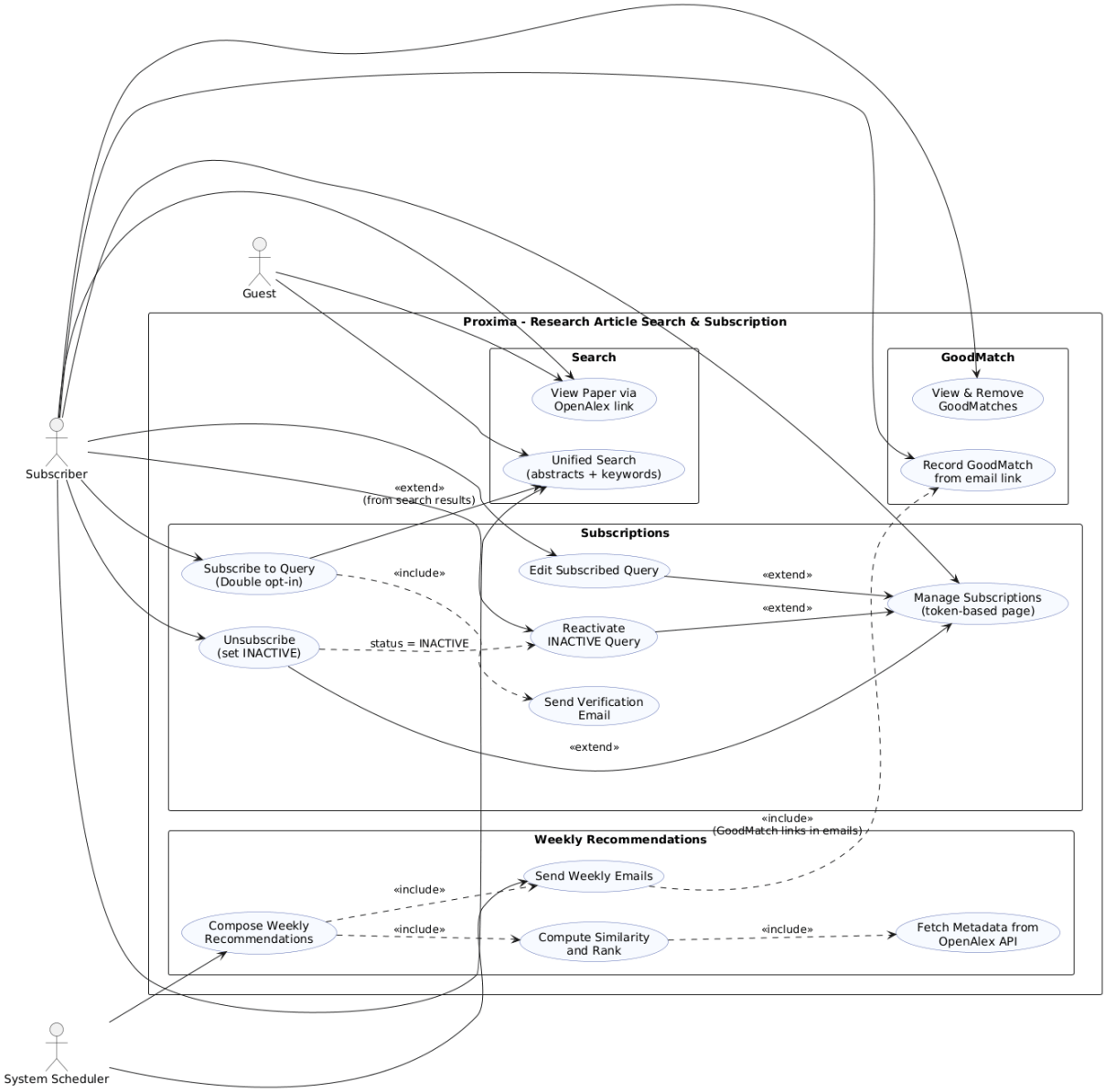

Figure 5.1. Use case diagram of the Proxima system.

6. DESIGN

6.1. Information Structure

6.1.1. Data Model and ER Diagram

The data model of the system is designed for a subscription–based recommendation engine and consists of four main entities: **Subscriber**, **Subscription**, **SentWork**, and **GoodMatch**.

- **Subscriber** represents a user identified by an email address. For each subscriber, the system stores a unique `email`, an optional `name`, and two tokens: a `verification_token` for email verification and a `manage_token` that provides token–based access to the subscription management page. The fields `is_verified`, `created_at`, and `verified_at` track the verification status and lifecycle of the subscriber.
- **Subscription** represents a user's saved search query. Each subscription is linked to a **Subscriber** and stores the query under a human–readable `query_name` together with its parameters in `abstracts` and `keywords` (both as JSON lists). The double opt–in flow is modeled using `is_verified` and `verification_token`, while `is_active` implements soft unsubscribe/reactivation without losing history. The fields `frequency` (currently "weekly") and `last_sent_at` are used by the weekly recommendation job to decide when to send the next email.
- **SentWork** records which papers have been delivered to which subscription via email. Each record references a **Subscriber** and a **Subscription** and stores the OpenAlex identifier in `work_id` together with the timestamp `sent_at`. This table is used to avoid sending the same paper multiple times for the same subscription and to populate "archive" recommendations from previously sent items.
- **GoodMatch** stores the papers that a user has explicitly marked as a "good match" by clicking a link in the weekly email. Each GoodMatch is linked to a **Subscriber** and, optionally, to the **Subscription** under which the paper was recommended. It contains the paper identifier `work_id`, the `title` and `openalex_url`, as well as an optional similarity `score` and timestamps `first_clicked_at` and `last_clicked_at`

to capture user engagement over time.

Relationally, a **Subscriber** can have many **Subscriptions**, and each **Subscription** can have many associated **SentWork** and **GoodMatch** records (1–to–many relationships). The pair `(email, query_name)` is enforced as unique for subscriptions, `(subscription, work_id)` is unique for sent works, and `(subscriber, subscription, work_id)` is unique for good matches. This design ensures that duplicate subscriptions are prevented, the same paper is not emailed repeatedly for a given subscription, and each "good match" is stored once per user–subscription–paper combination.

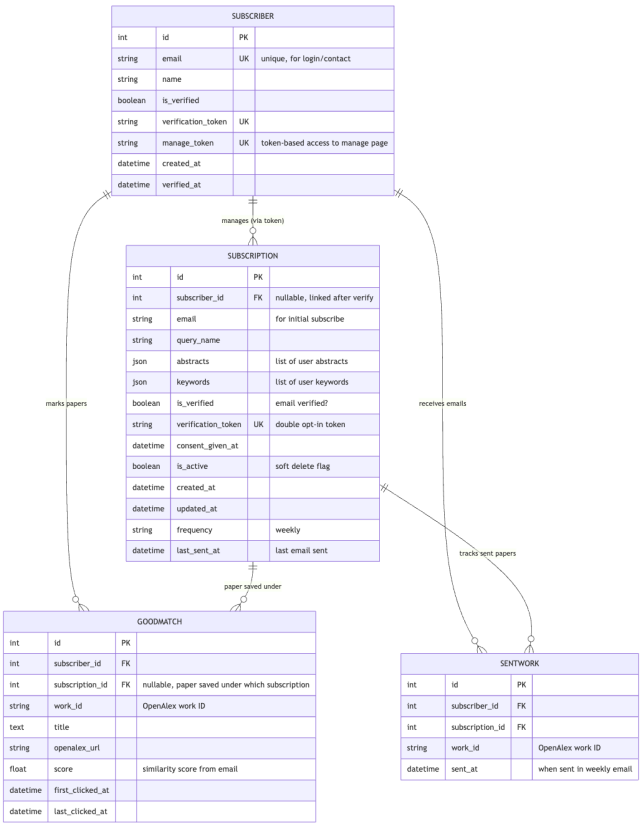

Figure 6.1. Entity–relationship diagram of the Proxima system, showing the relations between Subscriber, Subscription, SentWork and GoodMatch entities.

6.2. Information Flow

This section describes the interaction flows between the main components of the Proxima platform. Each subsection corresponds directly to a sequence diagram and explains how users, frontend components, backend services, and external APIs interact to complete a specific task.

6.2.1. Make Search

The “Make Search” flow begins when a user submits abstracts and/or keywords through the Proxima interface. The backend validates the request, constructs a query plan, and retrieves candidate works from the OpenAlex API. Semantic embeddings are computed and results are reranked based on similarity scores before being returned to the user. When evaluation mode is enabled, multiple result sets may be generated for comparison. Input validation errors or upstream service failures are propagated back to the user interface.

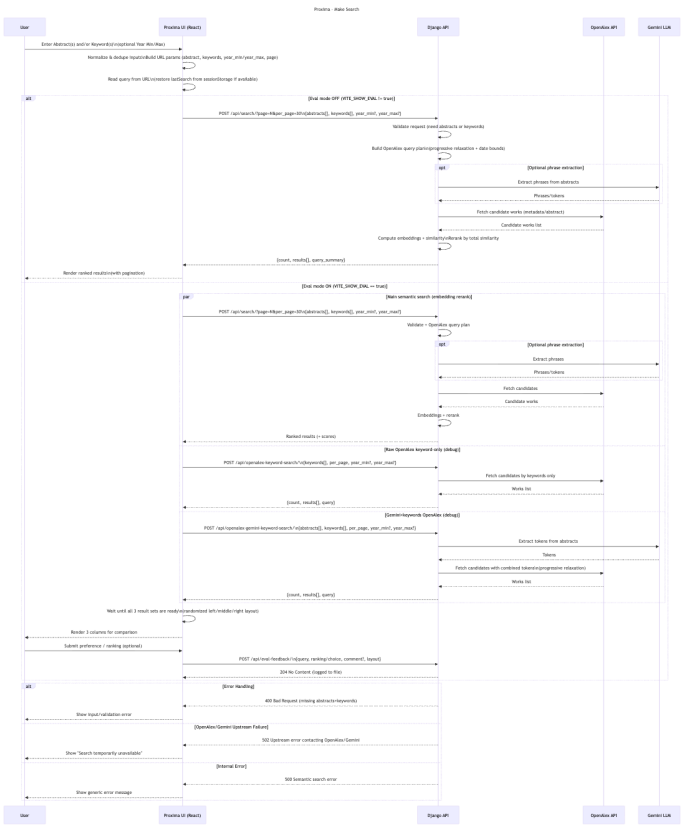

Figure 6.2. Proxima – Make Search Sequence Diagram

6.2.2. Subscribe

This flow describes how a user subscribes to a search query. After the user submits an email address and query information, the backend creates or updates the corresponding subscriber and subscription records. A verification email is sent to the user, and the subscription remains inactive until the verification step is completed.

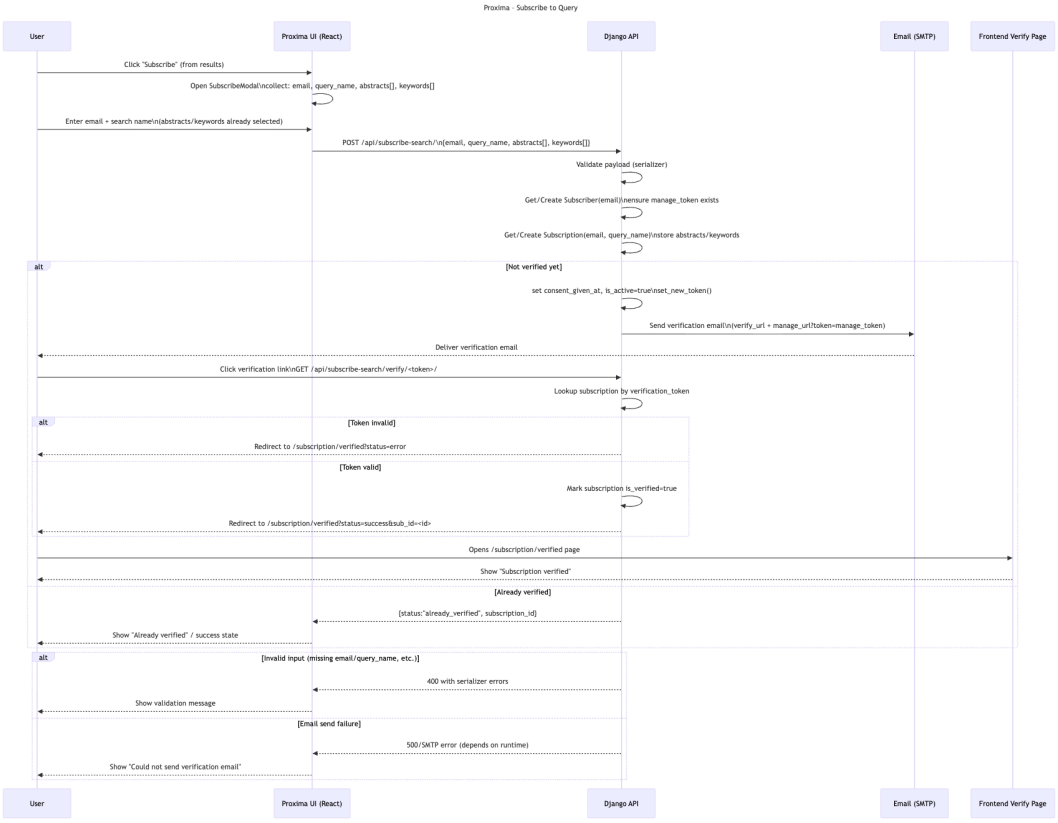

Figure 6.3. Proxima – Subscribe to Query Sequence Diagram

6.2.3. Double Opt-In

The double opt-in flow ensures that subscriptions are activated only after explicit user confirmation. A verification token is generated and sent to the user via email. When the user clicks the confirmation link, the backend validates the token and updates the subscription state accordingly. Invalid or expired tokens result in an error state.

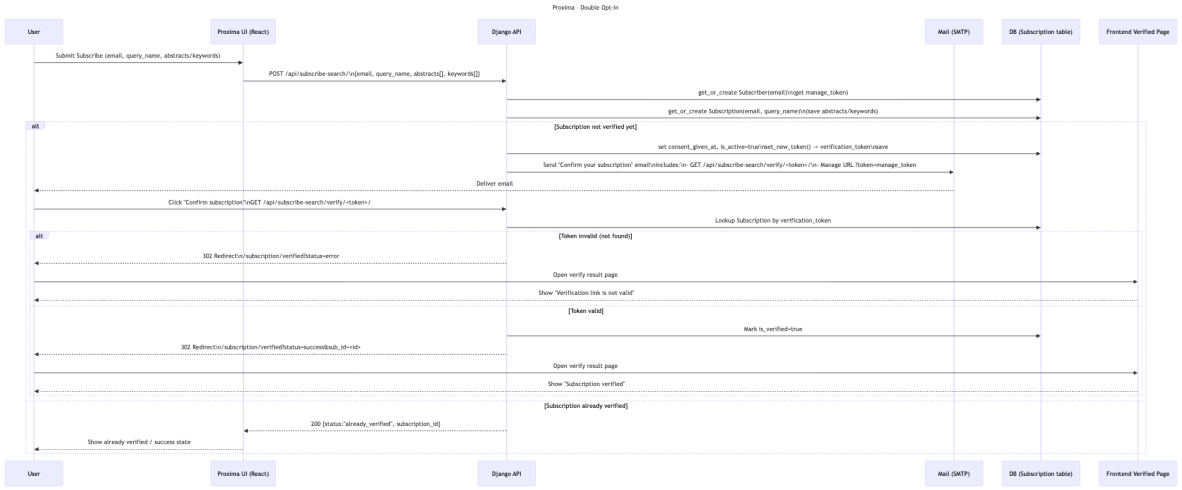

Figure 6.4. Proxima – Double Opt-In Sequence Diagram

6.2.4. Weekly Recommendation Email

This flow illustrates the automated weekly recommendation process. A scheduled backend job selects eligible subscriptions, retrieves recent works from OpenAlex, computes semantic similarity scores, and filters results based on relevance. The final recommendations are rendered using an email template and delivered via the mail service. Previously sent items are excluded to avoid duplication.

Figure 6.5. Proxima – Weekly Recommendation Email Flow

6.2.5. Email Links

This flow describes how links embedded in emails are handled. Subscription management links open directly in the Proxima web application using token-based access, while paper links redirect users to the corresponding OpenAlex pages. Certain links may pass through the backend before redirection in order to record user interactions.

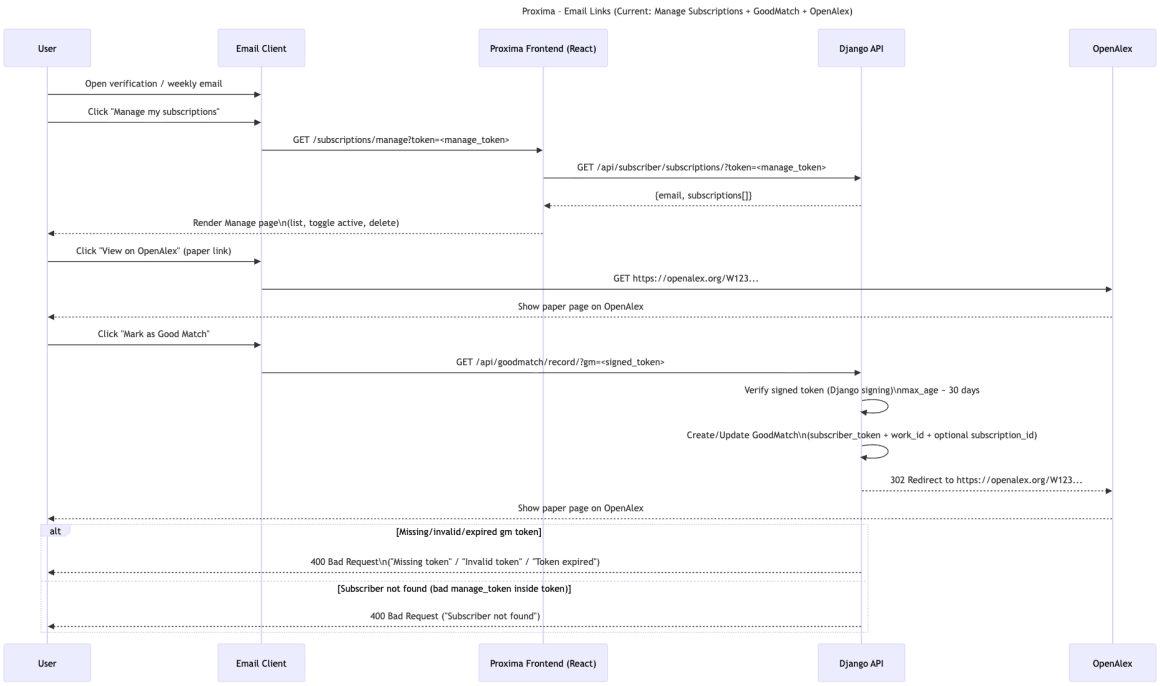

Figure 6.6. Proxima – Email Links Flow

6.2.6. Subscribe via Confirmation Email

This flow represents the completion of the subscription process through the confirmation email. When a user clicks the verification link, the backend validates the token, activates the subscription if valid, and redirects the user to a confirmation page indicating the result of the operation.

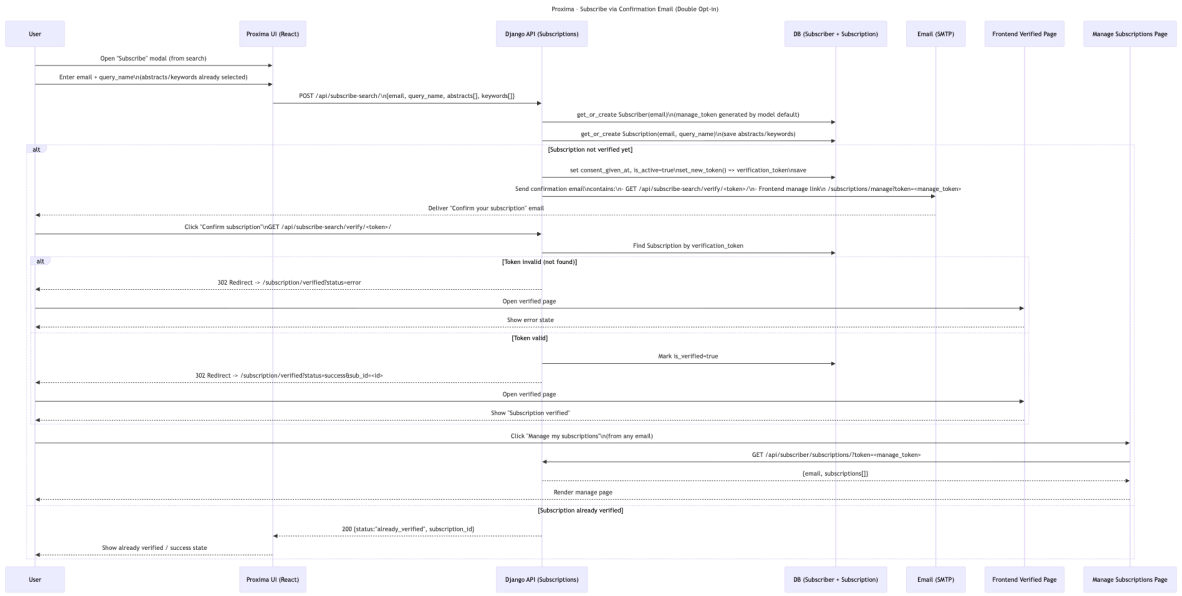

Figure 6.7. Proxima – Subscribe via Confirmation Email

6.2.7. Unsubscribe (Soft State) and Reactivation

This flow shows how users can deactivate and later reactivate subscriptions. Instead of permanently deleting subscription data, the system marks subscriptions as inactive. This soft-state approach preserves historical information and allows users to reactivate subscriptions without re-entering query details.

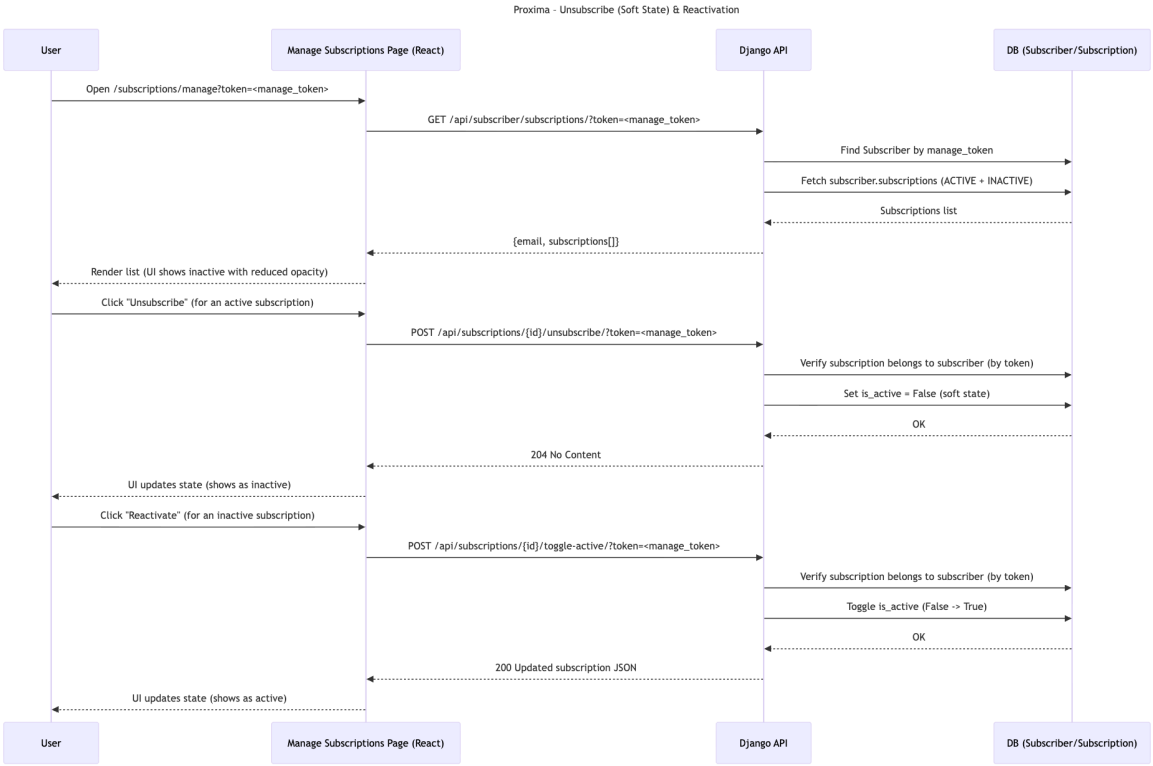

Figure 6.8. Proxima – Unsubscribe and Reactivation Flow

6.2.8. Good Match

The “Good Match” flow allows users to mark recommended papers as relevant. When the user selects this option from an email link, the backend records the interaction and then redirects the user to the corresponding OpenAlex page. Marked items can later be viewed in the subscription management interface.

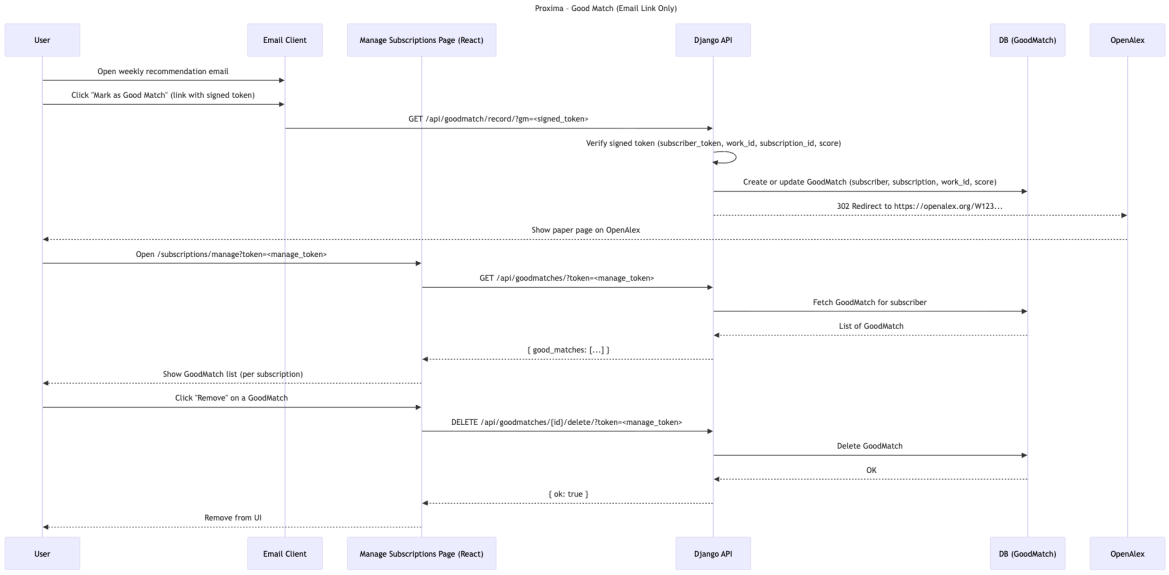

Figure 6.9. Proxima – Good Match Flow

6.2.9. Good Match Removal

This flow describes how users remove previously marked papers from their Good Match list. The request is initiated from the subscription management interface and processed by the backend, which deletes the corresponding record and updates the user interface accordingly.

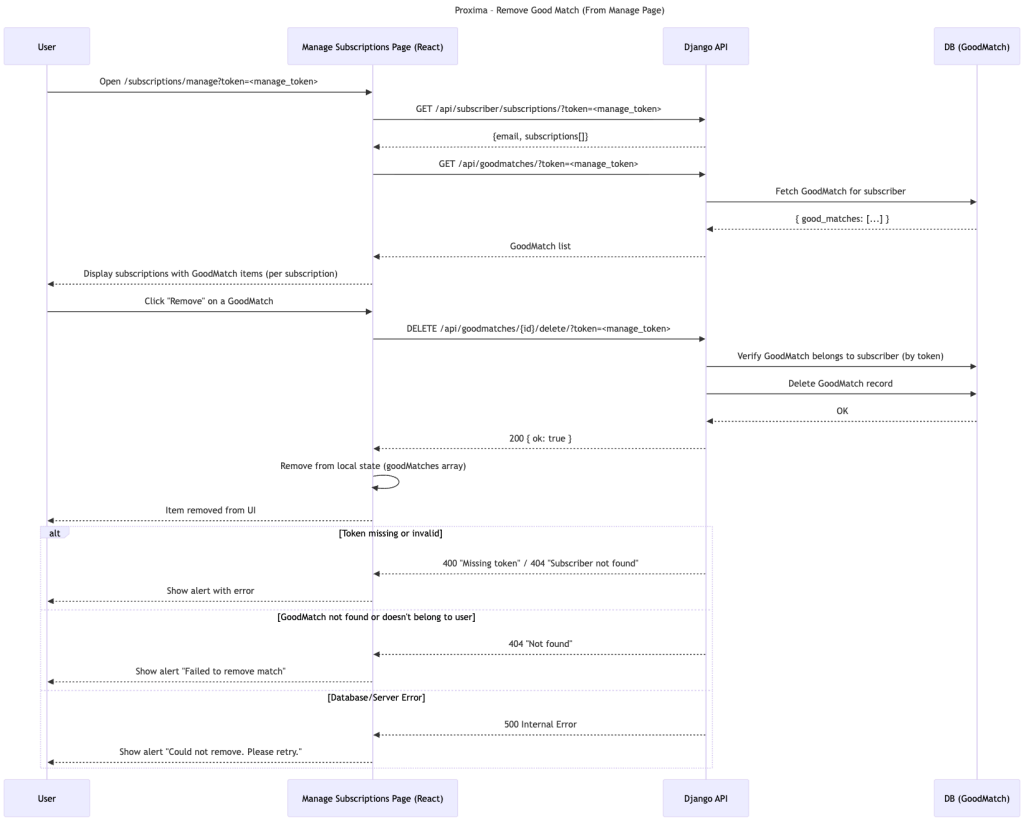

Figure 6.10. Proxima – Good Match Removal Flow

6.3. User Interface Design

This section presents the updated user interface design of the **Proxima** platform. The interfaces are designed to support fast, intuitive, and interactive exploration of academic literature by researchers.

6.3.1. Homepage – Abstract and Keyword-Based Search

The homepage is the primary entry point of the system where users initiate their research process. Users can enter one or more keywords and paste one or multiple research abstracts into the input area. An optional year filter allows the search results to be constrained within a specific publication time range.

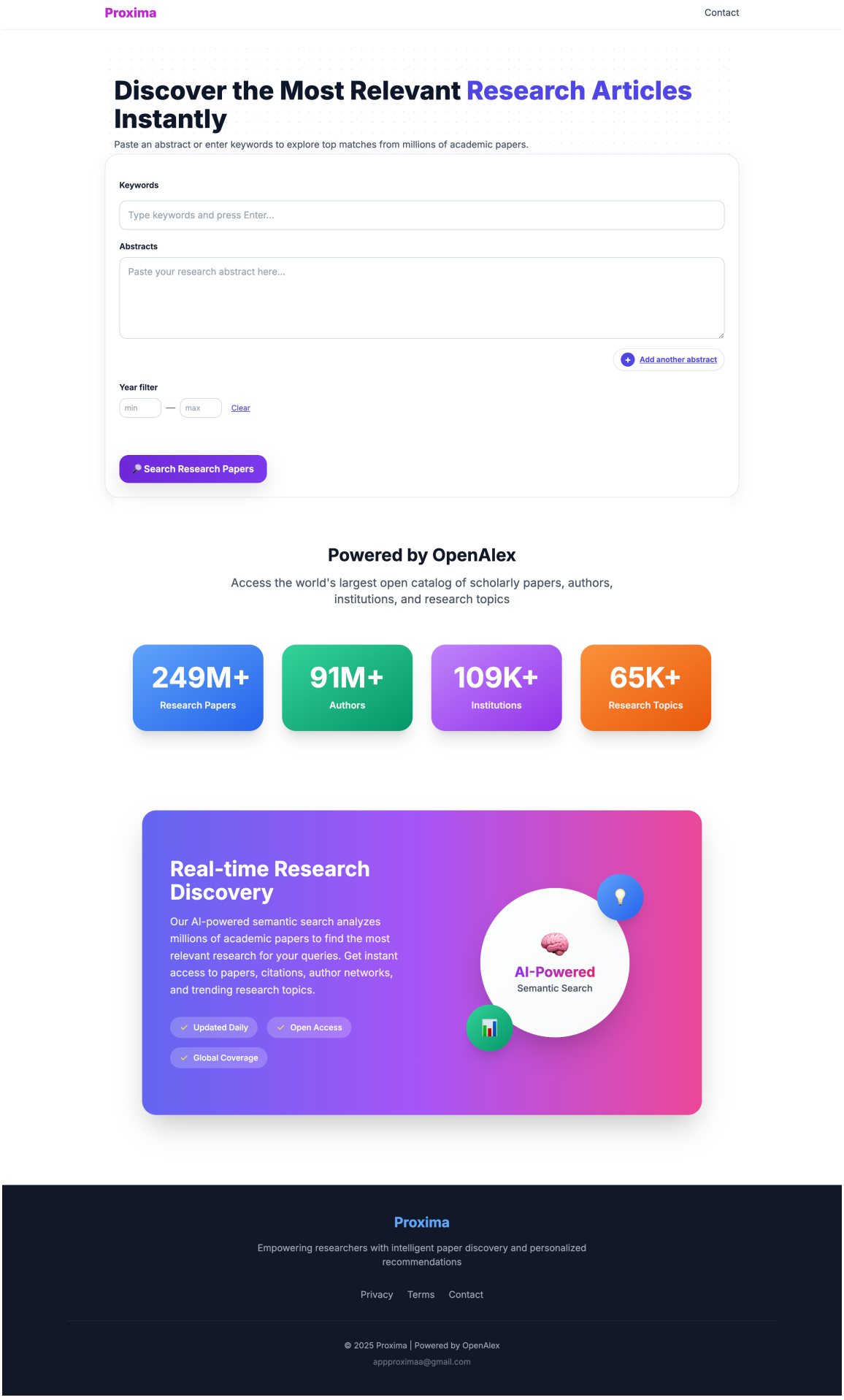

Figure 6.11. Proxima Homepage – Abstract and Keyword-Based Search

6.3.2. Search Input Interface

This interface combines keyword tags, multi-abstract input fields, and the year filter within a unified form structure. Users can add or remove abstracts dynamically and initiate the semantic search through a single action.

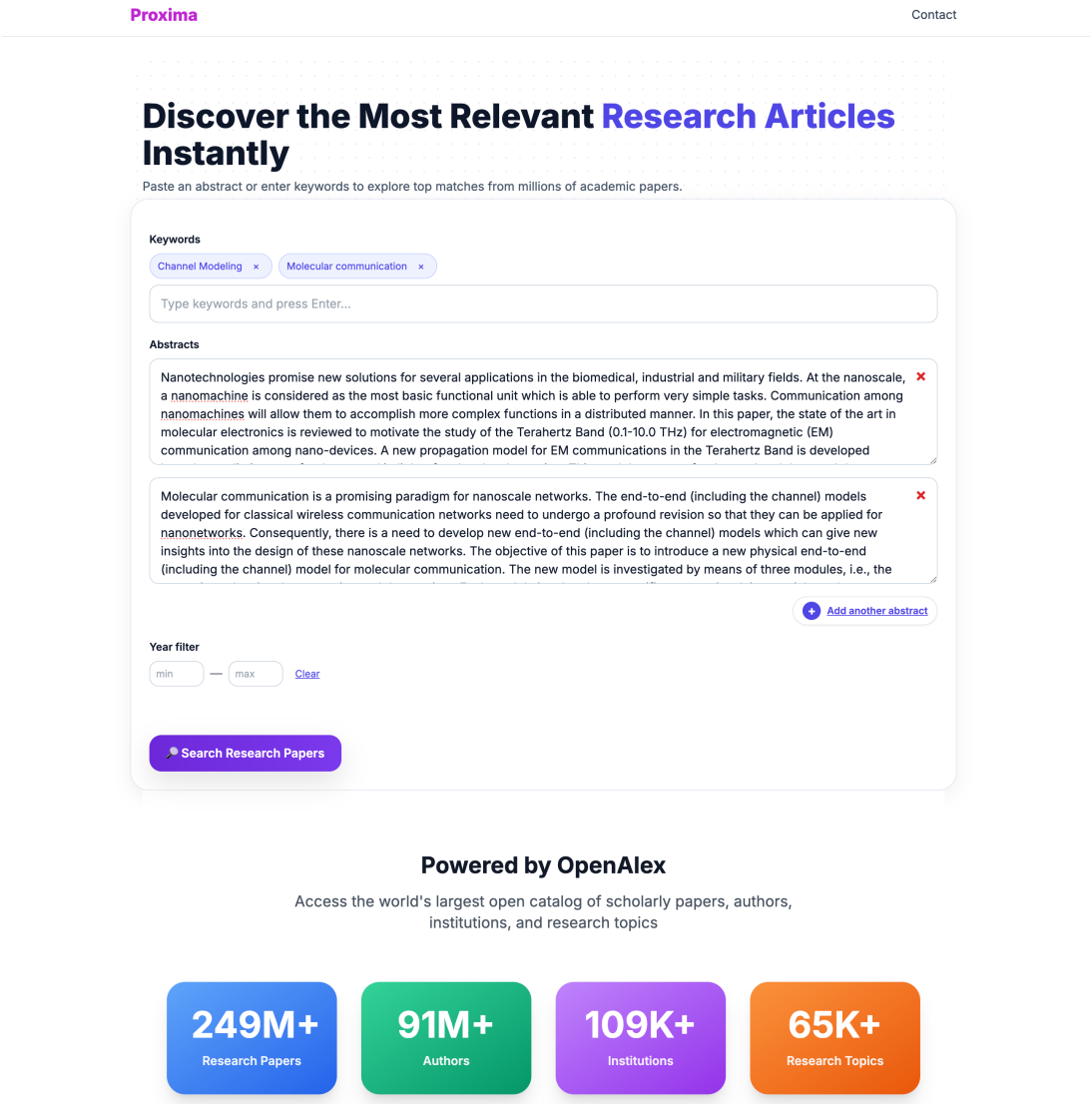

Figure 6.12. Abstract and Keyword Input Interface

6.3.3. Interactive Search Loading Feedback

During the semantic search process, Proxima displays an interactive mini-game as a loading feedback mechanism. This component is shown while embeddings are generated and external APIs are queried, providing visual progress indicators and maintaining user engagement.

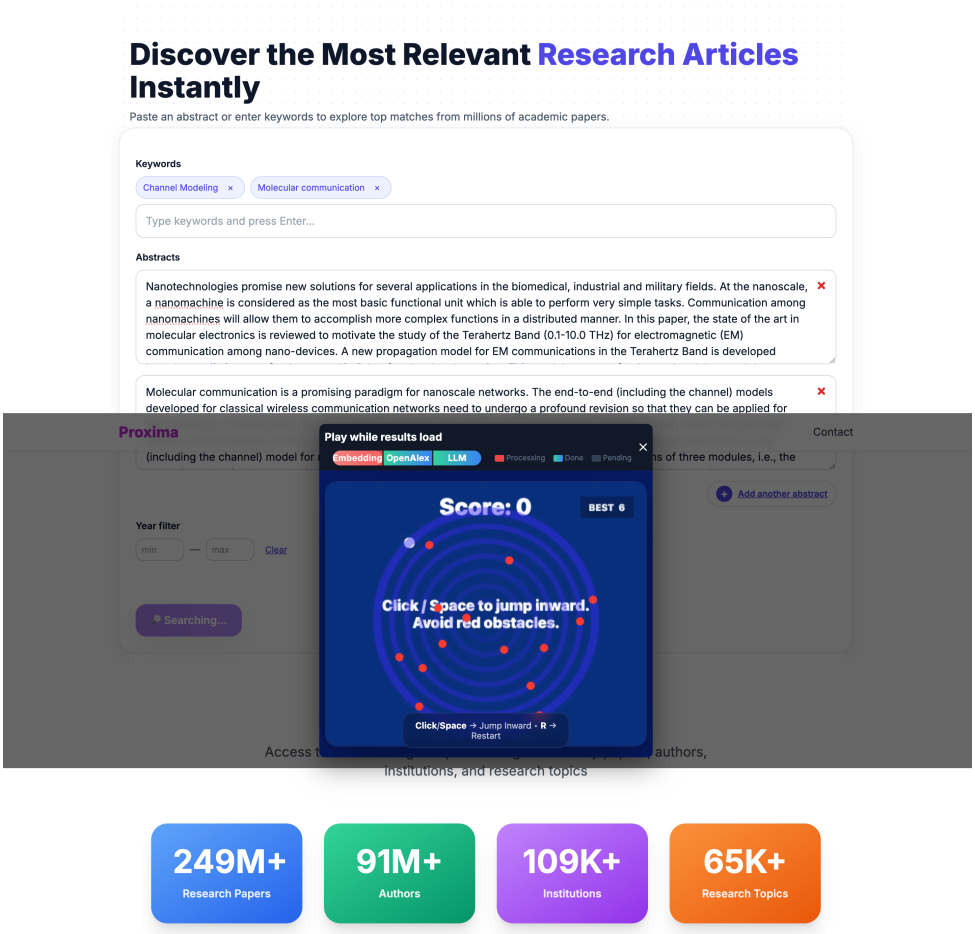

Figure 6.13. Interactive Loading Feedback During Semantic Search

6.3.4. Search Results View

The search results are presented as a ranked list of academic papers ordered by semantic similarity. Each result card displays the paper title, author information, publication year, a short abstract preview, associated keywords, and a similarity score. Users may sort the results based on similarity.

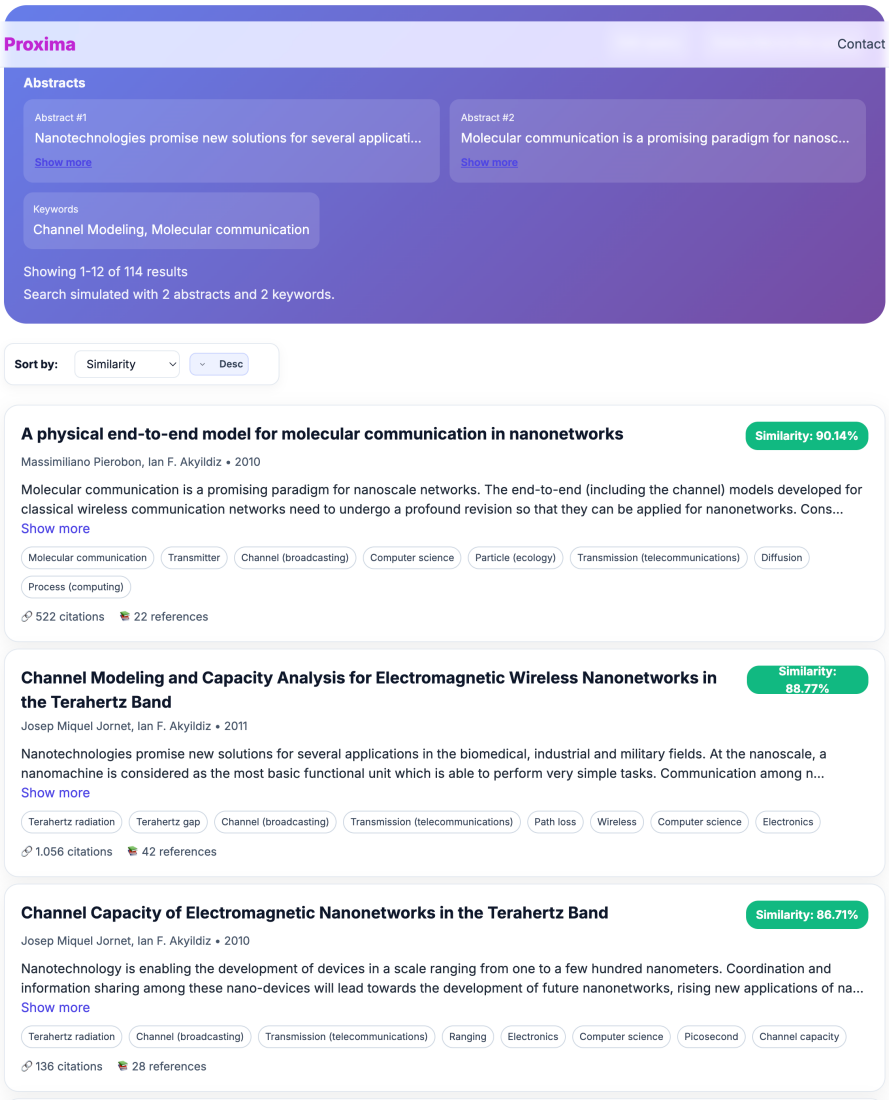

Figure 6.14. Search Results View

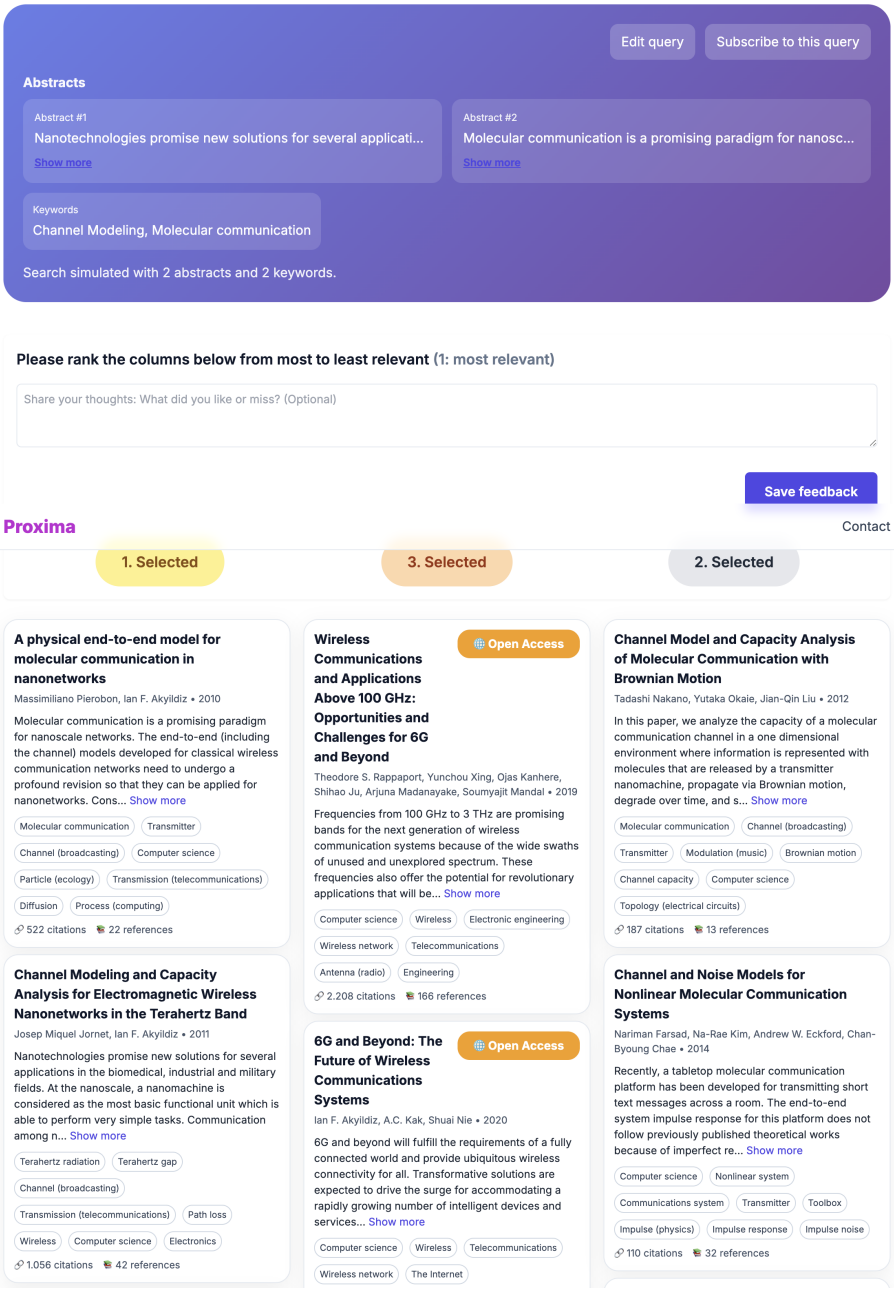

Figure 6.15. Search Results View2

6.3.5. Edit Query Modal

The *Edit Query* modal allows users to revise their existing search queries. Within this modal, abstracts and keywords can be updated, and the semantic search can be re-executed based on the modified inputs.

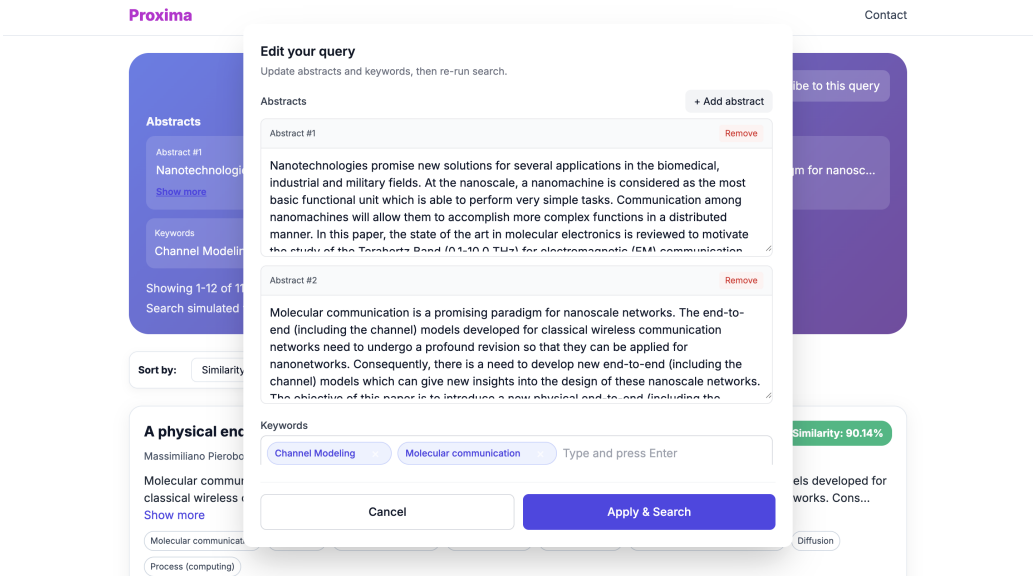

Figure 6.16. Edit Query Modal

6.3.6. Subscribe to Query Modal

Users can subscribe to a specific search query in order to receive weekly updates. This modal collects a query name and an email address and requires explicit user consent before proceeding. The subscription process is finalized through a double opt-in verification mechanism.

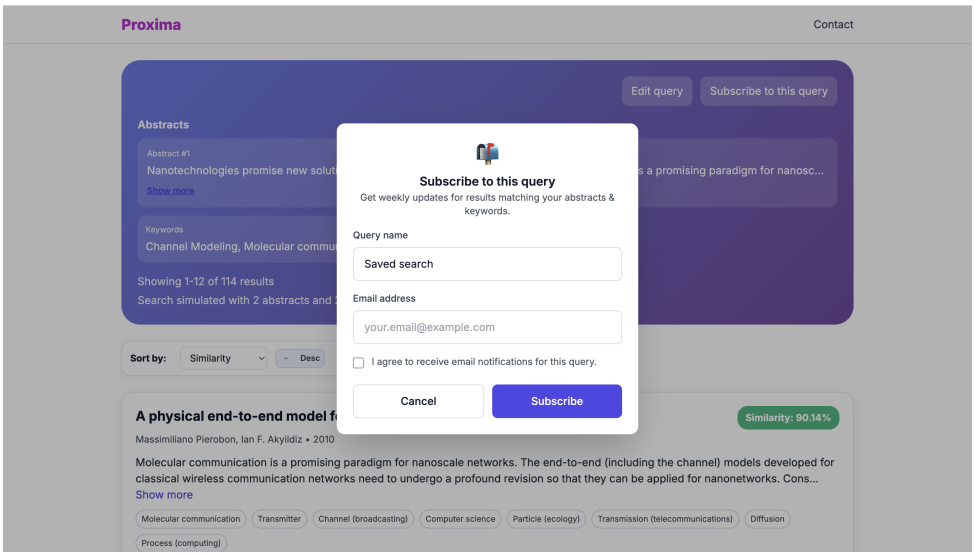

Figure 6.17. Subscribe to Query Modal

6.3.7. Subscription Confirmation Email

After submitting a subscription request, the system sends a confirmation email to the user. The email includes a verification link to activate the subscription as well as a link to manage existing subscriptions.

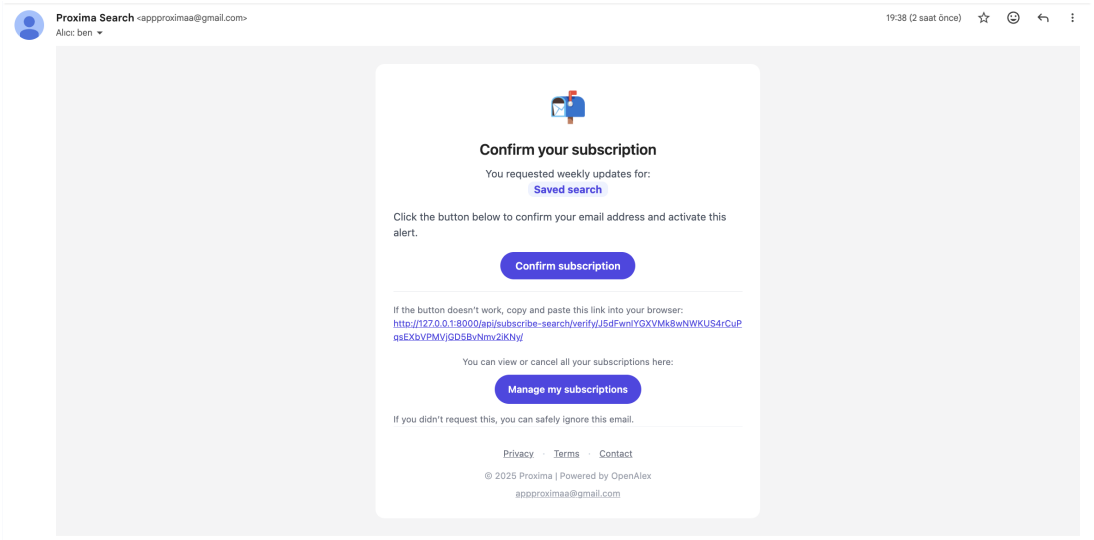

Figure 6.18. Subscription Confirmation Email

6.3.8. Subscription Verification Result

When the user clicks the verification link, the subscription is successfully activated. The system displays a confirmation screen indicating that email updates will start for the subscribed query.

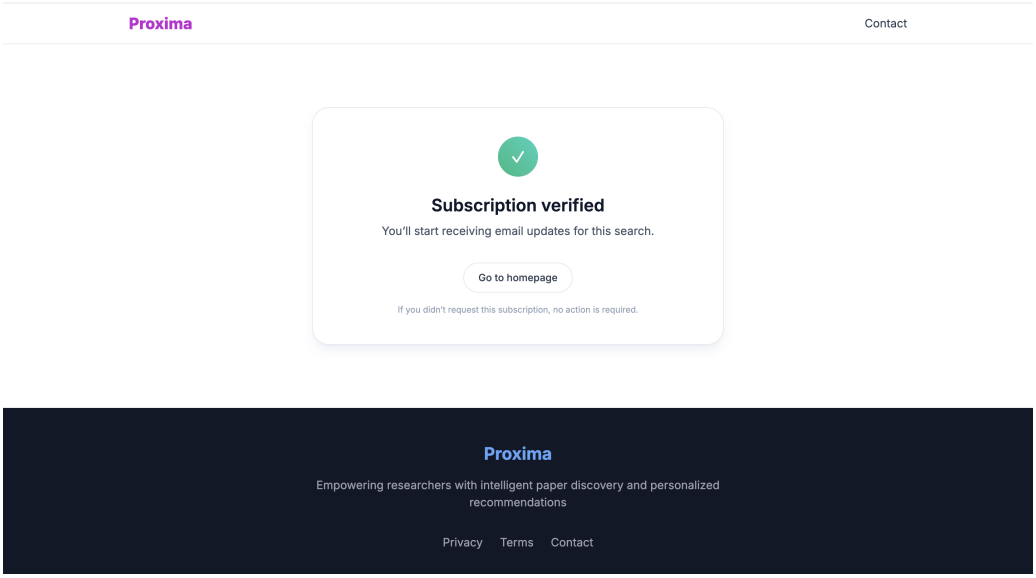

Figure 6.19. Subscription Verified Screen

6.3.9. Manage Subscriptions Page

The Manage Subscriptions page enables users to view and control their active subscriptions. For each subscription, the associated abstracts, keywords, and verification status are displayed. Users may initiate the unsubscribe action directly from this interface.

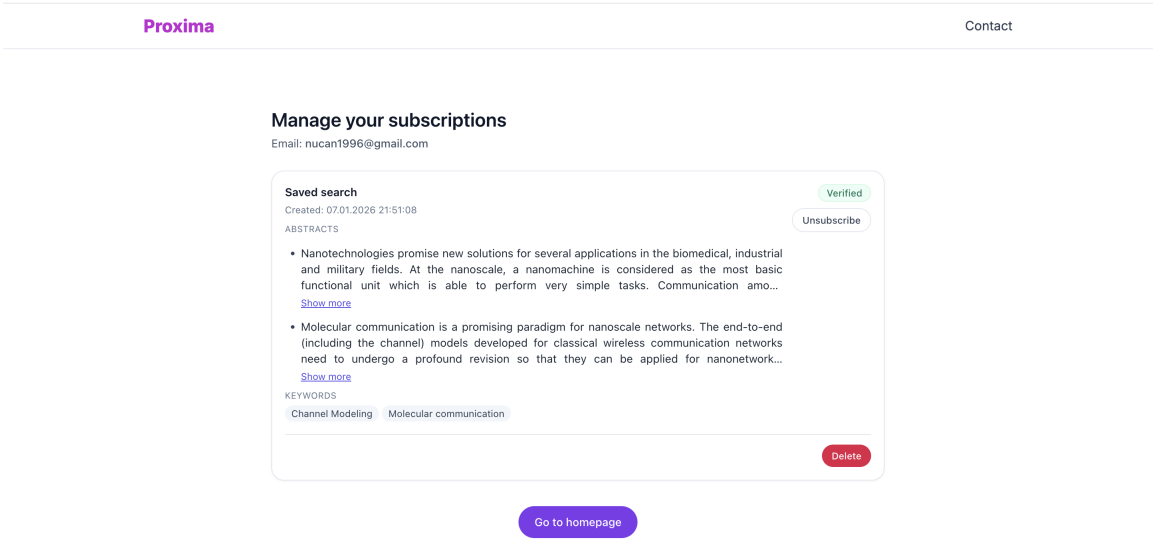

Figure 6.20. Manage Subscriptions Page

6.3.10. Unsubscribe Confirmation Modal

To prevent accidental removals, unsubscribing from a query requires explicit confirmation. This modal prompts the user to confirm the unsubscribe action before the subscription status is updated.

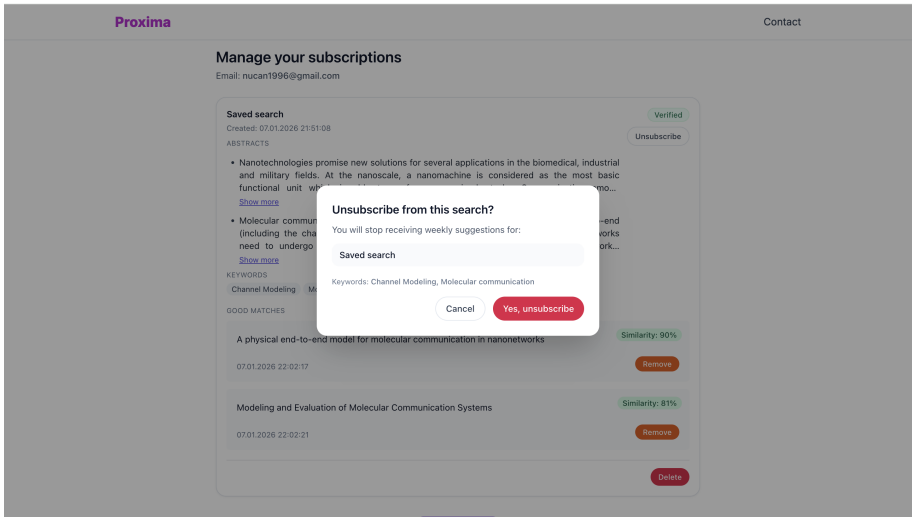

Figure 6.21. Unsubscribe Confirmation Modal

6.3.11. Weekly Recommendation View

For subscribed queries, the system provides weekly recommendations consisting of newly published papers and relevant archival works. Results are grouped into two sections: *New this week* and *Relevant from archive*, each ranked by semantic similarity.

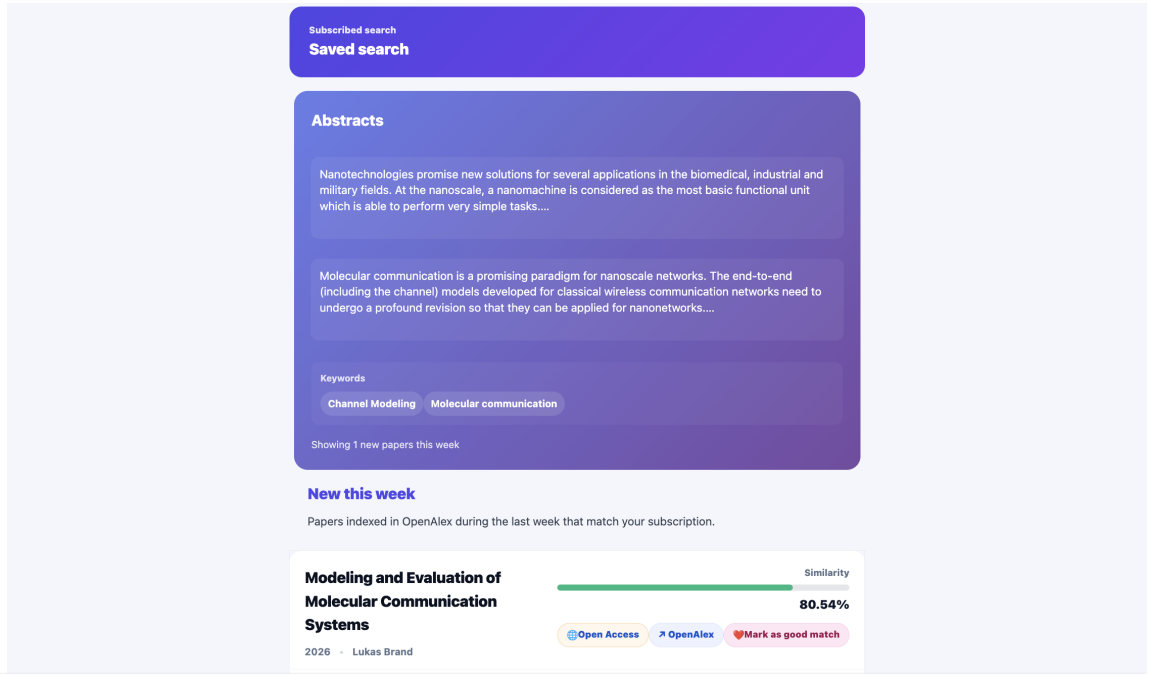

Figure 6.22. Weekly Recommendation View

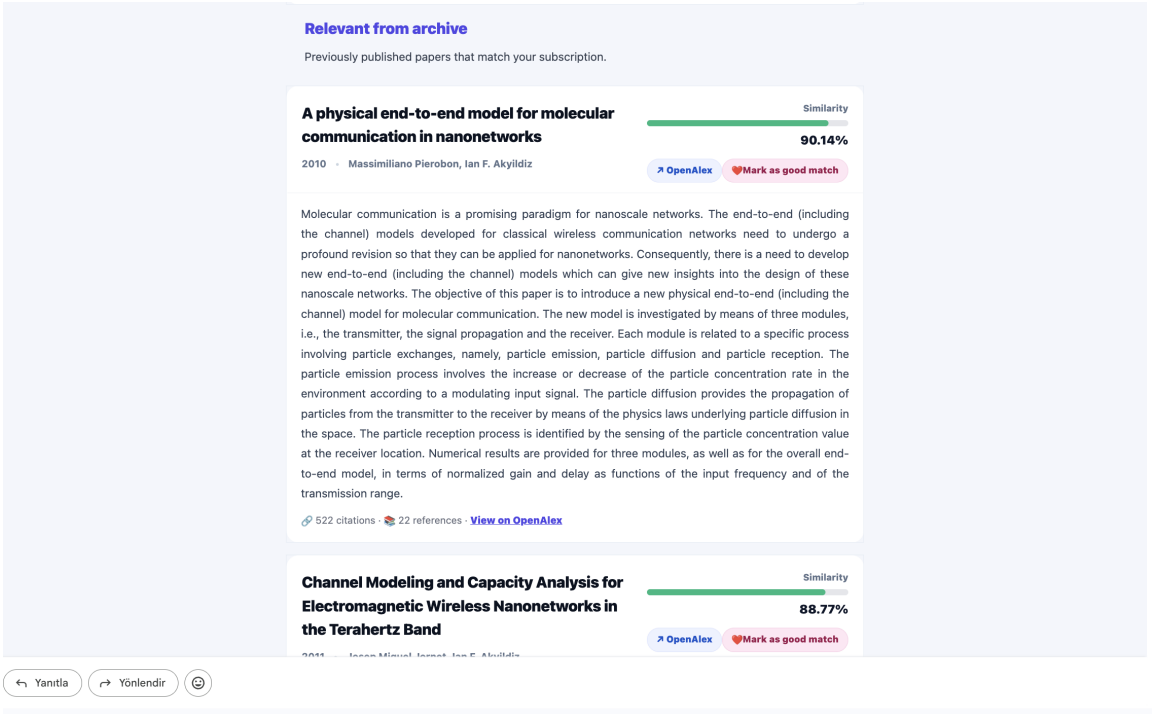

Figure 6.23. Weekly Recommendation View2

7. IMPLEMENTATION AND TESTING

7.1. Implementation

7.1.1. System Overview

Proxima is implemented as a web-based client–server system that combines a React single-page frontend, a Django REST backend, and external services such as OpenAlex and a pretrained sentence embedding model. Users interact with the frontend to run semantic searches, compare different retrieval pipelines, and manage email-based subscriptions through secure, tokenized links.

The backend exposes JSON APIs for search, subscription lifecycle, GoodMatch tracking, and evaluation feedback. It fetches candidate papers from OpenAlex, encodes queries and works with the Alibaba–NLP/gte-large-en-v1.5 embedding model, and reranks results by semantic similarity before returning them to the client. A relational database stores subscribers, subscriptions, sent works, and good matches, while a scheduled management command periodically re-executes saved searches and sends personalized weekly recommendation emails. Environment-based configuration allows the same codebase to run consistently across development and production deployments.

7.1.2. Backend Implementation

The backend of Proxima is implemented in Python using Django and Django REST Framework. It exposes a set of JSON APIs that handle semantic search over the OpenAlex corpus, email-based subscriptions, weekly recommendation delivery, and user feedback logging. The backend is deployed with PostgreSQL and environment-based configuration.

7.1.2.0.1. Technology Stack and Project Structure

The backend is organized as a standard Django project under the `backend` folder. The `core` app contains global configuration (database, CORS, email, allowed origins) and

the root URL router. The `api` app implements the domain models, REST endpoints, templates for transactional and weekly emails, and a custom management command for scheduled recommendations. A separate `app` package holds reusable service code such as OpenAlex API integration, text preprocessing, and embedding computation.

Django REST Framework is used to implement API views and serializers, while email delivery is handled through Django's email backend configured via environment variables. Database access is handled using Django's ORM with PostgreSQL.

7.1.2.0.2. Semantic Search Pipeline

Semantic search is built around OpenAlex as the primary data source. The module `app/openalex_client.py` handles HTTP access to the OpenAlex REST API and extracts key text fields (title, abstract, topics, concepts) via the helper `work_to_text_fields`. The `app/embeddings.py` module loads a pretrained sentence embedding model, `gte-large-en-v1.5`, using the Sentence-Transformers library to produce 1024-dimensional vectors.

Each OpenAlex work is represented by a combined embedding from three sources: main text (title + abstract), topic descriptions, and concept labels. These are aggregated into a normalized vector using fixed weights (0.7 / 0.2 / 0.1). At query time, the `SearchView` retrieves candidate papers using keyword and metadata filters, encodes the query and papers, computes cosine similarities, and returns top-ranked results with metadata and similarity scores. Additional endpoints implement baseline keyword-only and Gemini-augmented search for evaluation.

7.1.2.0.3. Subscription and Weekly Recommendation Engine

The subscription subsystem supports long-term topic tracking. `Subscriber` and `Subscription` models store user emails, query configuration (name, abstracts, keywords), and verification state. A double opt-in flow is implemented by generating tokens, sending HTML confirmation emails, and activating subscriptions upon token validation in `SubscriptionVerifyView`.

Weekly recommendation delivery is handled by a Django management command, `send_weekly_subscriptions`, which selects active and verified subscriptions, re-runs the

semantic search pipeline, filters out previously sent papers using the `SentWork` table, and sends a personalized email digest. The HTML emails are rendered using `weekly_update_email.html`, and delivery is handled by Django's email backend. All sent papers are logged to prevent duplicates.

7.1.2.0.4. GoodMatch Tracking and Management API

User feedback is collected through the `GoodMatch` model. Each weekly email includes signed links to mark individual papers as a "good match." These links encode subscription and paper details. The `record_goodmatch` view validates the token, stores the interaction, and redirects the user to the OpenAlex page.

Management endpoints allow token-based access to view and delete saved good matches without requiring login. The backend returns good matches ordered by recency and handles deletions securely.

7.1.2.0.5. Evaluation Feedback Logging

To evaluate different search methods, the backend includes an endpoint for logging user preferences from the UI. The evaluation interface presents three result lists (embedding-based, keyword-only, and Gemini-augmented). When the user selects or ranks a list, anonymized feedback is appended to a log file for offline analysis. This helps improve ranking strategies in future iterations.

7.1.2.0.6. Configuration and Environment Management

Runtime configuration is managed via environment variables defined in `core/settings.py`. Variables include API credentials, email server settings, and frontend URLs used in confirmation and management links. The environment-based setup ensures consistent behavior across development, testing, and production deployments without modifying the application code.

7.1.3. Frontend Implementation

The frontend of Proxima is implemented as a single-page application using React and Vite. It provides the main user interface for semantic search, evaluation experiments, subscription verification, and token-based subscription management. All communication with the backend occurs over JSON APIs.

7.1.3.0.1. Technology Stack and Project Structure

The frontend code is located under the `frontend` directory. The entry point `src/main.jsx` initializes the React application and configures client-side routing for the home page, the search page, the subscription verification status page, and the subscription management page.

Page-level components are organized under `src/pages`, while reusable UI elements such as paper cards and error views are placed in `src/components`. Styling is handled with standard CSS modules in `src/index.css`. A dedicated API layer in `src/services/api.js` wraps the browser `fetch` API and exposes typed helper functions for search, subscription, GoodMatch, and evaluation endpoints.

7.1.3.0.2. Search and Evaluation Interface

The search page allows users to paste one or more seed abstracts, optionally add keyword hints, and specify publication year bounds. When a search is submitted, the frontend sends a JSON request to the main semantic search endpoint and renders a list of OpenAlex papers with titles, authors, venues, years, open-access indicators, and similarity scores. Each paper is displayed using a reusable `PaperCard` component that truncates long abstracts and highlights similarity percentages.

In evaluation mode, the search page orchestrates three parallel pipelines: the main embedding-based search, a baseline OpenAlex keyword-only search, and a Gemini-augmented OpenAlex search that first expands the query into key phrases. The UI randomizes column order, shows top-k results for each pipeline side-by-side, and lets the user select or rank the preferred pipeline. The resulting feedback is sent to the backend evaluation

endpoint for offline analysis.

7.1.3.0.3. Subscription Verification and Management UI

After subscribing from the search results, users are redirected back from the email verification link to a dedicated verification status page. This page reads query parameters (token and status), optionally fetches the verified subscription details, and provides shortcuts to start a search with the stored configuration.

Ongoing management of subscriptions is implemented through a token-based interface. The management page is accessed with a `token` parameter in the URL, corresponding to the subscriber's `manage_token`. Using this token, the frontend calls the backend to retrieve all subscriptions and associated GoodMatch entries for that subscriber. The UI presents actions to unsubscribe, toggle active status, permanently delete a subscription, and remove individual good matches, without requiring a traditional username–password login.

7.1.3.0.4. API Integration and Error Handling

The `api.js` module centralizes all HTTP interactions with the backend. It defines functions for semantic search, OpenAlex keyword and Gemini-augmented search, creating and verifying subscriptions, listing and mutating subscriptions by token, listing and deleting GoodMatch records, and submitting evaluation feedback. Each function builds the appropriate URL, HTTP method, headers, and JSON body, and normalizes error responses.

On the UI side, pages handle loading and error states explicitly. For example, the search page displays inline error messages when network requests fail, and prevents duplicate submissions while a request is in progress. The management page shows user-friendly messages when an invalid or expired token is used.

7.2. Testing

7.2.1. Frontend Testing

The frontend of Proxima is tested using **Vitest** together with **React Testing Library** and **@testing-library/user-event**. The existing automated tests focus on the most critical user-facing components: the home page (search entry point) and the paper result cards.

7.2.1.0.1. Home page tests. The test suite `Home.test.jsx` mounts the home page inside a `MemoryRouter` and mocks the API layer in `services/api.js`. The following behaviours are verified:

- Rendering of the main heading, keyword input field and at least one abstract textarea.
- Client-side validation: when the user clicks "Search" with no abstracts or keywords, the search is blocked, `searchPapersPOST` is not called and a browser alert is triggered.
- Year range validation: entering a minimum year greater than the maximum year displays an inline validation message and disables the search button.
- Correct request payload: when the user enters a keyword and a valid year range, the component calls `searchPapersPOST` exactly once with a normalised payload (`keywords = ["AI"]`, empty `abstracts`, numeric `year_min`/`year_max`, and default pagination parameters).
- Error handling: if the search API rejects (e.g. network error), the page shows a user-friendly inline error message ("Search failed. Please try again.").

7.2.1.0.2. Paper card tests. The test suite `PaperCard.test.jsx` checks the rendering logic of individual search results:

- Display of the paper title as a link, together with authors, year, venue, citation count and reference count.

- Conditional similarity badge: when per-abstract similarity scores are present, the badge is shown; when the `hideSimilarity` prop is true, the badge is hidden.
- Open access indicator: when `is_open_access` and `oa_url` are provided, an "Open Access" link is rendered with the correct target URL.
- Abstract truncation: long abstracts are initially truncated, and the user can expand or collapse the full text via "Show more" / "Show less" toggles.

7.2.2. Backend Testing

The backend of Proxima is tested using Django's built-in test framework together with the Django REST Framework `APITestCase`. To avoid modifying the primary PostgreSQL database during test runs, the settings are configured so that `manage.py test` uses an isolated SQLite database (see `core/settings.py`), ensuring that all API tests run against a disposable test database.

7.2.2.0.1. Search API tests. The main semantic search endpoint `/api/search/` is covered by the `PaperSearchViewTests` test case in `api/tests.py`. The following behaviours are verified:

- Input validation: when both `abstracts` and `keywords` are empty (and no year filter is provided), the API returns `400 Bad Request` with a clear validation message instructing the client to provide at least one abstract, keyword or year filter.
- Response structure with mocked pipeline: the expensive OpenAlex + embedding reranking pipeline is mocked, and a synthetic result is injected. The test then asserts that the JSON response includes correct pagination metadata (`count`, `query_summary.page`, `query_summary.per_page`) and that each result is properly mapped to the frontend format (`id`, `title`, `year`, `venue`, `doi`, `is_open_access`, `oa_url`, authors, concepts). It also checks the presence of semantic fields such as `per_abstract_sims`, `per_abstract_contribs`, `total_score` and `score`.

7.2.2.0.2. OpenAlex keyword search tests. The debug endpoint `/api/openalex-keyword-search` is tested by the `OpenAlexKeywordSearchViewTests` suite:

- Happy-path behaviour: `fetch_openalex_candidates` is mocked to return a single OpenAlex work, and a POST request with `keywords = ["ai"]` is issued. The test asserts that the response has status `200 OK`, that the `query` and `keywords` fields are correctly echoed, that `count = 1`, and that the mapped result has the expected short OpenAlex identifier.
- Validation of required keywords: when the endpoint is called with an empty keyword list, it returns `400 Bad Request` and a human-readable error message ("At least one keyword must be provided.").

7.2.2.0.3. Subscription flow tests. The subscription lifecycle is exercised end-to-end in the `SubscriptionViewsTests` test case, also in `api/tests.py`. To keep the tests fast and deterministic, the email sending helper is mocked:

- Subscription creation: a POST request to `/api/subscribe-search/` with `email`, `query_name`, `abstracts`, `keywords` and `agree_to_emails = true` is sent. The test asserts that the API responds with `201 Created` or `200 OK`, returns a `subscription_id` and a status flag (e.g. `"pending_verification"`).
- Email verification: using the `verification_token` stored on the created subscription, the test calls `/api/subscribe-search/verify/<token>/` and expects an HTTP redirect to the frontend. After this call, the subscription is reloaded from the database and verified to have `is_verified = true`.
- Subscription detail retrieval: once verified, a GET request to `/api/subscriptions/<id>/` returns `200 OK` with the serialized subscription details (query name, abstracts, keywords, verification and activity flags, timestamps), confirming that the access control around unverified subscriptions works as intended.

7.3. Deployment

7.3.1. Deployment Diagram

In production, Proxima is deployed on a single DigitalOcean droplet using Docker Compose. The droplet runs three main services as Docker containers: a *web* service with Nginx, a *backend* service with Gunicorn and the Django REST API, and a *db* service with PostgreSQL.

Nginx serves the built React single-page application from the `/usr/share/nginx/html` directory and proxies all `/api/` requests to the backend container on port 8000. Inside the backend container, Gunicorn runs the Django application (`core.wsgi:application`) and exposes JSON APIs for semantic search, subscriptions, GoodMatch tracking, and evaluation feedback.

The backend connects to the PostgreSQL database for persisting subscribers, subscriptions, sent works, and good matches, and communicates with external services: the OpenAlex API for scholarly metadata, the Gemini API for query expansion, and an SMTP email server for sending verification and weekly recommendation emails. Environment variables supplied through Docker Compose configure database credentials, API keys, and email settings.

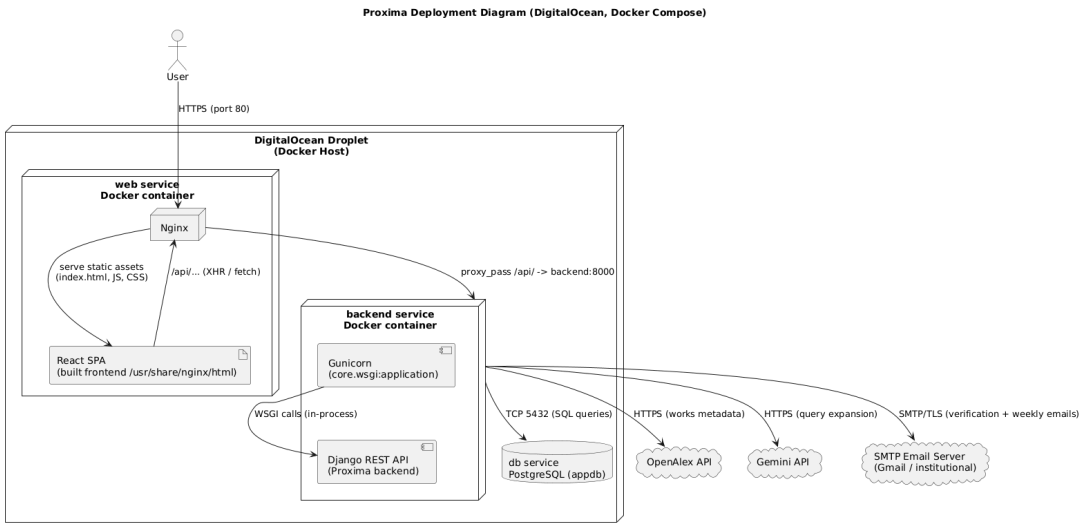

Figure 7.1. Deployment diagram of the Proxima system.

7.3.2. Building and Running the System

The project is organized into two main components: a Django backend under `backend/` and a React/Vite frontend under `frontend/`. For local development, the backend can be set up by following the step-by-step instructions in `backend/README.md`, which cover creating a virtual environment, installing Python dependencies, applying database migrations, running tests, and starting the development server.

The frontend can be built and run by following the instructions in `frontend/README.md`. This document explains the required Node.js version, how to install JavaScript dependencies, how to start the Vite development server, and how to run the automated frontend tests.

7.3.3. Docker-based Deployment

For production, Proxima is deployed on a single virtual machine using Docker Compose. The repository contains a `compose.yml` file that defines three services: `db` (PostgreSQL), `backend` (Django + Gunicorn), and `web` (Nginx serving the built React frontend and proxying `/api/` requests to the backend).

After copying the project to the target server and installing Docker and Docker Compose, the operator creates a backend environment file (`backend/.env`) with the required database credentials, OpenAlex and Gemini API keys, email configuration, and allowed host settings. Using `docker compose up --build -d` builds the images and starts all services. Nginx listens on port 80, serving the static frontend and forwarding API calls to the backend container on port 8000, while the backend connects to the PostgreSQL container over the internal Docker network.

7.3.4. User Manual

Accessing the Application. End users access Proxima through a standard web browser. The home page provides an entry point to the semantic search interface and to the subscription management page (via token-based links received by email). Screenshots

and detailed descriptions of the user interface are provided in Section 6.3, in particular Figures 6.12 and 6.20.

Running a Search. On the search page, users can paste one or more seed abstracts into the main text area and optionally add keyword hints and publication year bounds. After clicking the *Search* button, the system sends the query to the backend and displays a list of matching OpenAlex papers. Each result shows the paper title, authors, venue, year, similarity score, and open-access status, and users can expand long abstracts or follow links to OpenAlex.

Evaluation Mode. In evaluation mode, the interface presents three parallel result lists: the main embedding-based search, a baseline keyword-only OpenAlex search, and a Gemini-augmented OpenAlex search. Columns are randomized for each session. Users select the list they prefer and can optionally rank all three; this feedback is stored anonymously for offline analysis.

Creating a Subscription. Below the search results, users can turn a query into a subscription by entering their email address and a short name. Proxima sends a confirmation email with a verification link, and the subscription becomes active only after this link is clicked (double opt-in). A confirmation page reports whether the subscription was created successfully.

Weekly Recommendation Emails. For each active and verified subscription, Proxima periodically sends a weekly email digest containing newly indexed and archive papers that match the query. Each entry includes metadata, a short abstract, and quick actions: open-access links, an OpenAlex link, and a *Mark as good match* button.

Managing Subscriptions and Good Matches. At the bottom of each email, a *Manage subscriptions* link opens a management page identified by a secure token. On this page, users can view all subscriptions and good matches, unsubscribe or toggle the active status of a subscription, delete subscriptions, and remove individual good matches. No separate username or password is required; access is controlled by the token in the link.

Troubleshooting. If verification or weekly emails are not received, users should check their spam folder and verify their email address. If the *Manage subscriptions* link reports an invalid or expired token, a new link can be obtained by creating a fresh subscription with the same email or by contacting the system administrator.

7.3.5. System Manual

Purpose and Audience. This system manual is intended for administrators who install, deploy, and operate Proxima in a production environment. It assumes basic familiarity with Linux, Docker, and Docker Compose.

Server Requirements. Proxima is deployed on a single Linux virtual machine (e.g., a DigitalOcean droplet) using Docker Compose. A typical setup requires at least 4 GB RAM, 2 vCPUs, a recent Docker and Docker Compose installation, and a public DNS name pointing to the server.

Configuration. Runtime configuration is loaded from `backend/.env`. The most important groups of environment variables are:

- **Database:** `DB_HOST`, `DB_NAME`, `DB_USER`, `DB_PASSWORD`.
- **APIs:** OpenAlex and Gemini API keys.
- **Email:** SMTP host, port, username, password, `DEFAULT_FROM_EMAIL`.
- **Public URLs:** `SUBSCRIPTION_FRONTEND_MANAGE_URL`, `SUBSCRIPTION_BACKEND_BASE_URL` or `SITE_URL`.

The `compose.yml` file defines three services: `db` (PostgreSQL), `backend` (Django + Gunicorn), and `web` (Nginx serving the built React frontend and proxying `/api/` requests).

Deployment and Operations. After configuring `backend/.env`, the full stack is built and started with:

```
docker compose up --build -d
```

Nginx listens on port 80, serves the frontend, and forwards `/api/` traffic to the backend on port 8000, while the backend connects to PostgreSQL over the internal Docker network. Logs can be inspected using `docker compose logs` and services can be restarted with `docker compose restart`.

Scheduled Weekly Recommendations. Weekly recommendation emails are generated by the Django management command `send_weekly_subscriptions`. It should be executed periodically (e.g., once per week) using `cron` or a similar scheduler:

```
docker compose exec backend \
  python manage.py send_weekly_subscriptions
```

The command selects active, verified subscriptions, runs the semantic search pipeline, sends HTML digests via SMTP, and records sent works to avoid duplicates.

Backups and Maintenance. The most important state is stored in the PostgreSQL volume (`pgdata`) and the `backend/.env` file. Regular backups should include database dumps (e.g. via `pg_dump` inside the `db` container) and a secure copy of `backend/.env`. Upgrades follow the usual workflow: pull the latest code, rebuild and restart the stack with `docker compose up --build -d`, then check logs and health endpoints to verify correct operation.

8. RESULTS

This chapter presents the overall results obtained from the design, implementation, and evaluation of the Proxima semantic research article search and recommendation system. The results are analyzed from three complementary perspectives: search relevance and retrieval quality, recommendation and subscription behavior, and system performance and reliability.

8.1. Semantic Search Effectiveness

The primary objective of Proxima is to retrieve conceptually relevant academic papers beyond traditional keyword overlap. Experimental usage and evaluation mode comparisons demonstrate that the embedding-based semantic search pipeline consistently produces more coherent and topically aligned results than keyword-only retrieval. Papers returned by the semantic pipeline exhibit stronger conceptual consistency with user-provided abstracts, even when terminology differs significantly across disciplines.

User feedback collected through the evaluation interface confirms this observation. When presented with three parallel result sets (semantic embedding-based search, raw OpenAlex keyword search, and Gemini-augmented keyword search), users most frequently selected the embedding-based results as the most relevant. This validates the effectiveness of cosine similarity over dense embeddings for capturing semantic relatedness in academic texts.

8.2. Recommendation Quality and Subscription Outcomes

The weekly recommendation mechanism successfully delivers personalized research updates based on subscribed queries. By re-evaluating stored queries against newly indexed OpenAlex works, the system identifies newly published or newly indexed papers with high semantic similarity. Additionally, the inclusion of pre-scheduled archival recommendations enables users to discover older but still highly relevant works, supporting deeper literature exploration beyond recency alone.

System logs confirm that duplicate recommendations are avoided through the `SentWork` tracking mechanism, ensuring that users do not receive the same paper multiple times for a given subscription. The *Good Match* feedback feature further demonstrates engagement with recommendations, allowing users to explicitly mark relevant papers and build a curated archive over time.

8.3. System Performance and Reliability

From a system perspective, Proxima operates reliably under typical usage conditions. Semantic search requests complete within acceptable response times, even when multiple abstracts and keywords are provided. The subscription workflow—including double opt-in verification, token-based management, and automated weekly email delivery—functions consistently across both test and deployment environments.

Frontend and backend testing results indicate stable behavior across core features such as search, subscription creation, verification, and management. Error handling mechanisms effectively prevent invalid inputs and provide clear feedback to users in case of failures. Overall, the system satisfies the functional and non-functional requirements defined earlier in the project.

9. CONCLUSION

This project introduced Proxima, an AI-driven semantic search and recommendation platform designed to address the growing challenge of academic information overload. By leveraging transformer-based text embeddings and cosine similarity, Proxima enables researchers to discover conceptually relevant literature beyond surface-level keyword matching.

The system successfully integrates semantic search, personalized query subscriptions, and automated recommendation delivery into a unified workflow. Evaluation results demonstrate that embedding-based retrieval outperforms traditional keyword-based approaches in terms of relevance and coherence, particularly for interdisciplinary research topics. The subscription and weekly recommendation features further support a proactive research workflow, helping users stay informed about both newly published and archival works relevant to their interests.

From an engineering perspective, Proxima demonstrates a modular and scalable architecture built on a Django REST backend and a React-based frontend. The use of OpenAlex as an open-access data source aligns the system with open science principles, while careful handling of user data and subscriptions ensures privacy, transparency, and ethical responsibility.

Despite these strengths, the system has several limitations. Semantic embedding models introduce computational overhead, which may affect scalability for very large datasets or high request volumes. In addition, embedding quality depends on pretrained models that may not fully capture domain-specific nuances without further fine-tuning.

Future work may explore hybrid ranking strategies that combine semantic similarity with citation-based signals, adaptive model selection for different research domains, and more advanced personalization using long-term user feedback. Expanding multilingual support and improving scalability through approximate nearest neighbor indexing are also promising directions.

In conclusion, Proxima demonstrates that semantic, subscription-based literature discovery is both feasible and effective, offering a meaningful improvement over traditional academic search tools and providing a strong foundation for future research-oriented recommender systems.

REFERENCES

1. J. Priem et al., *OpenAlex: A Fully Open Index of Scholarly Works*, 2022. https://openalex.org

2. Alibaba-NLP, *GTE-large-en-v1.5*, Hugging Face, 2024. https://huggingface.co/Alibaba-NLP/gte-large-en-v1.5