Prism: RAG Visualization Engine
Technical Documentation & Architecture Overview

Hasibullah (hasibullah1811)

January 21, 2026

Contents

1 Executive Summary

Prism is a full-stack developer tool designed to visualize the internal mechanics of **Retrieval-Augmented Generation (RAG)** pipelines. While most RAG tools act as "black boxes," Prism exposes the hidden data transformations—tokenization, chunking, vector embedding, and similarity search—allowing engineers to optimize their retrieval strategies before deployment.

2 System Architecture

The application follows a decoupled **Client-Server architecture**, separating the interactive UI from the heavy computational logic.

- **Frontend (Client):** Built with **React (Vite)** and **Tailwind CSS**. It handles state management, real-time rendering of chunks, and data visualization (using Recharts).
- **Backend (Server):** Built with **Python (FastAPI)**. It acts as the computational engine, handling natural language processing (NLP), mathematical calculations, and tokenization.

3 Core Features & Technical Logic

3.1 Recursive Text Chunking

The Problem: LLMs have context windows. Sending a 100-page PDF at once crashes the model or costs a fortune. We must break text into smaller "chunks."

The Logic: Prism uses LangChain's `RecursiveCharacterTextSplitter`. Unlike simple splitters that chop text at fixed indices (e.g., every 500 chars), this algorithm recursively tries to split text by a hierarchy of separators (`\n\n`, `\n`, `" "`, `""`) to keep paragraphs and sentences intact.

3.2 Overlap Calculation (Sliding Window)

The Problem: If a sentence is cut in half between Chunk A and Chunk B, the semantic meaning is lost ("Context Fragmentation").

The Solution: We create a "Sliding Window" overlap.

The Algorithm:

1. Take the *last* N characters of Chunk i.
2. Take the *first* N characters of Chunk $i+1$.
3. Identify the exact intersection string.
4. The Frontend renders this intersection in **Yellow** to visually confirm that context is preserved across boundaries.

3.3 Search Simulation (TF-IDF & Cosine Similarity)

The Feature: Users can type a query (e.g., *"How do vectors work?"*) and see which chunk matches. **The Math:** Instead of calling an expensive external API (like OpenAI Embeddings), Prism simulates retrieval using **TF-IDF** and **Vector Space Models**.

3.3.1 1. The Vectorizer (TF-IDF)

We convert raw text into a numerical vector. The weight w of a term t in a document d is calculated as:

$$w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t \tag{1}$$

Where:

- **TF (Term Frequency):** How often the word appears in the chunk.
- **IDF (Inverse Document Frequency):** Measures how "unique" a word is across all chunks.

$$\text{idf}_t = \log\left(\frac{N}{df_t}\right) \tag{2}$$

(Where N is total chunks, and df_t is number of chunks containing term t).

3.3.2 2. The Similarity Metric (Cosine Similarity)

Once we have vectors for the Query (A) and the Chunk (B), we calculate how similar they are by measuring the cosine of the angle between them:

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3}$$

- **Result = 1.0:** The vectors are identical (Perfect Match).
- **Result = 0.0:** The vectors are orthogonal (No shared meaning).

3.4 Semantic Map (Dimensionality Reduction)

The Feature: A 2D Scatter Plot showing the "distance" between chunks. **The Logic:** TF-IDF vectors are **High-Dimensional** (often 10,000+ dimensions). We use **PCA (Principal Component Analysis)** to squash them into (x, y) coordinates.

1. **Covariance Matrix:** We calculate how every variable relates to every other variable.
2. **Eigen Decomposition:** We compute the principal directions (Eigenvectors) in which the data varies the most.
3. **Projection:** We project the data onto the top 2 Principal Components.

$$T = X \times W_L \tag{4}$$

(Where X is the original data and W_L is the matrix of top 2 eigenvectors).

3.5 Token "X-Ray" (Byte Pair Encoding)

The Feature: The "Matrix View" ribbon that breaks words into colored blocks. **The Logic:** LLMs do not see words; they see "Tokens." Prism uses `tiktoken` (OpenAI's tokenizer) which uses **BPE (Byte Pair Encoding)**.

- Common words (e.g., "apple") are 1 token.
- Complex words (e.g., "Unstoppable") are split into sub-words: `Un`, `stopp`, `able`.

4 Frontend Visualization Strategy

- **Recharts:** Used for the Semantic Map (ScatterChart) and Confidence Monitor (BarChart). The data is normalized in the backend before being sent to the frontend to ensure high rendering performance.
- **State Management:** React `useState` handles the complex dependency graph (e.g., changing `chunk_size` triggers a re-fetch, which recalculates `chunks`, which triggers a `PCA` re-calculation, which updates the `ScatterChart`).

5 Conclusion

Prism is not just a visualizer; it is a **mathematical proof of concept** for RAG. By exposing the vectors, tokens, and similarity scores, it transforms abstract AI concepts into concrete, debuggable engineering data.