ISSN 2278-3091

Volume 14, No.6, November – December 2025

International Journal of Advanced Trends in Computer Science and Engineering

Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse021462025.pdf

https://doi.org/10.30534/ijatcse/2025/021462025

Integration of Search Embeddings into Editorial Pipelines: Architecture, Storage, and Querying in PostgreSQL

Andreev Georgii Andreevich
Bachelor's degree, Moscow polytechnic university, Moscow, Russia
g_andreev@outlook.com

Received Date: October 22, 2025 Accepted Date: November 23, 2025 Published Date: December 06, 2025

ABSTRACT

The article explores modern text vectorization methods and their application in editorial information systems. It analyzes the architecture of a pipeline that enables the integration of search embeddings into editorial infrastructure using PostgreSQL and the pgvector extension. The approach is shown to support unified embedding generation, storage, indexing, and semantic search within a single environment. It describes techniques for handling long texts, including segmentation and aggregation of embedding representations, as well as the technical aspects of deploying such systems in production. The advantages of combining vector-based and traditional SQL search are discussed in the context of personalization, thematic navigation, and intelligent content management.

Key words : Vector search, PostgreSQL, pgvector, editorial systems, semantic analysis, embedding, text processing.

1. INTRODUCTION

Modern editorial systems are rapidly evolving toward the automation of information processing and search within large volumes of textual data. One of the most promising directions in this transformation is the integration of vector search – representations of texts in high-dimensional vector spaces generated by machine learning models. Vector embeddings significantly enhance search accuracy and relevance based on content to enable retrieval of semantically similar text segments even in the absence of any exact keyword matches. As editorial workflows increasingly rely on large-scale text data, vector-based models are increasingly becoming an important instrument for intelligent search, topic classification, and personalized content delivery.

The technical implementation of vector search in editorial systems requires a well-designed architecture for data storage and processing. Among the available solutions, PostgreSQL – a powerful, extensible object-relational database management system – holds a particular advantage. Through support for third-party extensions such as pgvector, PostgreSQL simplifies adding vector functionality to its databases. This approach not only provides scalability and flexibility for handling complex data types but also dramatically lowers the entry barrier for adopting vector search technologies, allowing teams to leverage advanced semantic capabilities without introducing additional infrastructure. As a result, PostgreSQL becomes a natural foundation for building high-performance vector indexes, semantic search, and real-time text analysis upon. Such functionality is especially relevant to media websites, news companies, and academic publishers with high-performance and precise information retrieval needs.
The objective of this study is to analyze an architectural approach to integrating search vectors into editorial pipelines using PostgreSQL. The study explores the principles of building vector storage, indexing mechanisms, and query optimization strategies, while also reviewing practical aspects of deploying such solutions in production environments.

2. SURVEY OF STATE-OF-THE-ART APPROACHES TO TEXT VECTORIZATION

The advancement of text vectorization technologies has marked a significant milestone in the field of natural language processing (NLP), enabling a shift from simple statistical methods to deep semantic models [1].
In the early stages, vectorization was carried out using techniques such as Bag-of-Words (BoW) and TF-IDF, where each document was represented as a sparse vector reflecting word frequencies. However, these methods ignored semantic relationships between words and their sequential order. This limitation was broken with the emergence of Word2Vec, which enabled word vector training based on context. Through the utilization of architectures such as Skip-gram and CBOW, these models were the first to produce dense and semantically meaningful vectors, which opened doors to more accurate retrieval and analytical systems.

These include other techniques like GloVe (Global Vectors for Word Representation) by Stanford NLP combined global word co-occurrence statistics with vector learning to provide more interpretable and resilient embeddings. Facebook AI Research's FastText pushed this even further by employing words as a mix of their character n-gram vectors. This provided effective rare word and neologism processing, which is especially crucial for efficient handling in the dynamically evolving text streams of editorial systems. A substantial breakthrough in vectorization came with contextual language models that take into account not only word meanings but also their context within a sentence. ELMo (Embeddings from Language Models) introduced position- and context-dependent representations, while BERT (Bidirectional Encoder Representations from Transformers) represented a paradigm shift by enabling bidirectional encoding of textual input through transformer-based architecture. These models enabled the generation of universal embeddings for words, sentences, and full documents.

To represent longer textual units, various strategies have been developed to aggregate word or sentence vectors into a unified document-level embedding. A few of the well-known models in this category include Doc2Vec, InferSent, and universal sentence encoders such as Google's Universal Sentence Encoder. Others such as multilingual models XLM-RoBERTa and LaBSE enable effective vectorization of language content and are especially convenient for multilingual editorial processes where cross-lingual information search and semantic normalization are paramount.

The development of neural network-based approaches, particularly transformer models, not only expanded the boundaries of semantic representation but also yielded strong applied technology made available through APIs. One of the most widely used solutions today is the OpenAI Embeddings API, which provides access to the text-embedding-3 family of models, including text-embedding-3-small and text-embedding-3-large:

• text-embedding-3-small is a lightweight resource model that generates 1536-dimensional vectors. It is optimized for high-throughput query processing and can be utilized for building large-scale search indexes, content recommender systems, and user interest clustering.

• text-embedding-3-large produces vectors with up to 3072 dimensions and delivers higher benchmark scores; both models support dimensionality reduction via the dimensions parameter (an accuracy–cost trade-off).

Thus, leveraging the latest generation of embedding models enables editorial systems to achieve a qualitatively new level of text processing – from intelligent search and thematic navigation to contextual personalization and automated content annotation.

To justify the choice of vectorization architecture in editorial pipelines, it is necessary to consider the differences between two principal approaches: using cloud-based API services versus deploying self-hosted infrastructure with local models. Each option carries its own implications in terms of cost, performance, operational complexity, and legal or compliance-related risks (table 1).

Table 1: Cloud APIs vs. self-hosted embedding infrastructure [2, 3]

Criterion	**Cloud APIs (e.g., OpenAI)**	**Self-hosted infrastructure**
Total cost of ownership & deployment	Usage-based pricing; no buying GPUs or managing clusters.Example: text-embedding-3-small at $0.00002 per 1,000 tokens (as of 2025).	Requires GPUs/servers, containerization, orchestration, monitoring, and maintenance; high overhead at scale.
Criterion	**Cloud APIs (e.g., OpenAI)**	**Self-hosted infrastructure**
Performance & scalability	Provider-managed autoscaling; typical end-to-end latency in the low hundreds of milliseconds depending on region, load, and batching.	Latency rivaled only by correctly sized clusters, model servers, job queues, and load balancers; engineering effort.
Quality & freshness	Centralized model updates and re-training; strong relevance on fast-moving topics; dimension-shortening available for some providers.	Weights and fine-tuning must be updated to preserve relevance; ongoing evaluation and monitoring to prevent drift.
Data governance & compliance	Governed by the provider's data-processing terms; compliance posture depends on DPA, regional routing, and logging configuration.	Full control over access, audit logging, encryption, and retention policies; responsibility to implement and operate controls.

Thus, the current landscape of text vectorization solutions encompasses a wide range of models and architectures – from lightweight algorithms to deeply contextualized transformer-based systems. The choice among them depends on task-specific requirements, data scale, and the desired quality of semantic processing. In practice, editorial systems must consider not only the type of embedding model, but also the integration methodology within the existing infrastructure: whether to utilize cloud-based embedding services or deploy local solutions. This decision impacts technical considerations such as performance and cost, as well as how much data control, regulatory compliance, and flexibility to design the pipeline upgradable in the future.

3. ARCHITECTURE OF THE EDITORIAL PIPELINE WITH VECTOR SEARCH

Vector search is becoming the central feature of the editorial systems of the future, enabling semantically significant distribution and navigation of content. Forming an effective vector-based search in the context of editing must have a well thought-out architecture, where each component plays a specific role (figure 1).

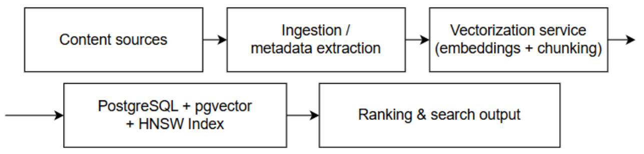
Content sources
Ingestion / metadata extraction
Vectorization service (embeddings + chunking)
PostgreSQL + pgvector + HNSW Index
Ranking & search output

Figure 1: Components of the editorial pipeline architecture

Sources of content are internal CMS, user-generated materials, social media channels, partner integrations, or third-party APIs. They supply the system with headlines, summaries, full texts, and metadata (author, date, category, etc.). Vectorization module subsequently transform incoming text data to fixed-size numerical vectors. As a microservice that consumes batches of text and outputs embedding vectors, the module utilizes cloud services like OpenAI Embeddings. If the input string exceeds the token limit of the model, it gets chunked up automatically into smaller pieces and the resultant embedding is based on averaging the vectors for each piece.
The resulting vectors are stored in a PostgreSQL database with the pgvector extension, and a record stores the source text, metadata, and the embedding vector (typically 1536 dimensions). To enable efficient search, an HNSW index is used with the operator class vector_cosine_ops to enable efficient and accurate nearest-neighbor search on cosine similarity.
The final component of the system is the recommendation and retrieval engine, which is fed with either a search user query or an existing article from the database and translates it into a vector, searches the closest entries in the database, and builds the final output. The ranking process also considers parameters such as publication date, source, topic, and user-specific personal preference.
To ensure stability and scalability of the pipeline, additional technical components must be incorporated, including text normalization, task orchestration, and system monitoring (table 2).

Table 2: Technical components of the editorial pipeline [4, 5]

Component	Purpose	Technologies & tools
Text normalization	HTML cleaning, stemming/lemmatization, tokenization, formatting.	spaCy, NLTK, custom scripts.
Task orchestration	Managing queues and distributing load between services.	Kafka, RabbitMQ, Celery.
Result caching	Storing frequently used embeddings and search results to reduce latency.	Redis, Memcached.
Monitoring and observability	Tracking metrics (latency, errors), logging, alerting.	Prometheus, Grafana, ELK.
Search interface / API	Processing user queries, vector conversion, result filtering.	REST API, gRPC, FastAPI.
Hybrid search (vector + full-text)	Improving search quality by combining multiple indexing methods.	PostgreSQL, pgvector, tsvector.

Thus, a whole editorial pipeline architecture for vector search is a modular system that incorporates text preprocessing, semantic representation, efficient storage, and an intelligent retrieval mechanism. The inclusion of infrastructure modules – such as task orchestration, monitoring, caching, and hybrid search – ensures fault tolerance, scalability, and low-latency performance under real-world editorial workloads. This architecture can be tailored to specific needs of media websites, recommender engines, or corporate knowledge management systems, providing a modular and extensible foundation for advanced content management.

4. IMPLEMENTING VECTOR STORAGE AND SEARCH IN POSTGRESQL

Building an effective semantic search system is not just a matter of the accurate creation of vector representations but also of the effective means of their storage and retrieval. In editorial workflows, where the volumes of data can reach millions of texts, the optimal solution must trade off scalability, low latency, and flexibility of querying. One of the most popular and easiest to use is PostgreSQL with the pgvector extension, which natively supports storage, indexing, and comparison of embeddings directly within the relational database engine.
According to the Stack Overflow Developer Survey 2024, in which over 65,000 developers took part, 48.7 % indicated they utilized PostgreSQL, which confirms its position as one of the most popular and trustworthy database systems within the developer community (figure 2).

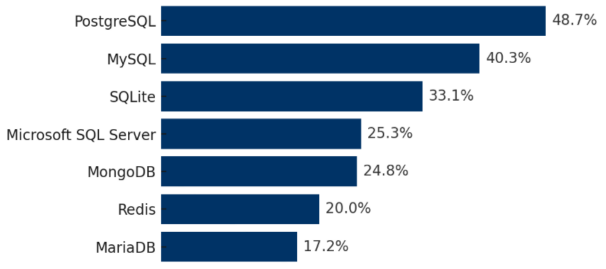

Figure 2: Most popular databases among developers in 2024 [6]

According to independent production benchmarks, PostgreSQL with the pgvector extension demonstrates stable performance when working with large datasets. For example, in tests conducted on Amazon Aurora PostgreSQL Compatible Edition (pgvector 0.8.0), the average query time dropped from 123 ms to 13 ms, resulting in more than a 9× speedup while maintaining output quality [7]. These results confirm that PostgreSQL with pgvector can be effectively used not only as a traditional relational database system, but also as an industrial and media-rich application's scalable solution for semantic search.

To integrate vector search into an editorial workflow, the content from CMS, user-generated materials, social media channels, partner integrations or API must be preprocessed, vectorized, and saved into PostgreSQL. Here is a simple Python example of how embedding vectors can be stored in a PostgreSQL table using the psycopg library and an OpenAI model.

PostgreSQL core is designed around a modular architecture of query planning and optimization layers, transaction management, execution, and extensibility through plugins and extensions [8]. With support for custom data types and operator classes, PostgreSQL can be easily adapted to non-standard use cases such as working with high-dimensional vectors. The pgvector extension – natively integrated via the C API – introduces a vector data type, similarity functions for Euclidean, cosine, and L1 distances, and support for HNSW indexing, enabling fast approximate nearest neighbor (ANN) search.

Once a vector representation of a text (e.g., from OpenAI's API or a local model) is generated, it can be stored in a PostgreSQL table with an explicitly defined dimensionality:

```
CREATE TABLE documents (
   id SERIAL PRIMARY KEY,
   title TEXT,
   embedding VECTOR(1536)
);
SELECT id, title
FROM documents
ORDER BY embedding <-> '[...]'
LIMIT 5;
```

One of the key advantages of using PostgreSQL is the ability to combine vector operations with traditional SQL filters – such as date, author, or category – without the need to offload data to external services. This is particularly important for editorial systems, where query control and completeness are critical.

Production-grade PostgreSQL configurations with pgvector can handle millions of records with response times under 50 ms, provided the HNSW index is properly configured. The index can be created as follows:

```
CREATE INDEX ON t_articles_embeddings
USING hnsw (embedding vector_cosine_ops);
```

Thus, PostgreSQL functions not only as a reliable relational database but also as a platform for building scalable semantic search systems, thanks to its extensible architecture and support for hybrid SQL + vector queries.

5. HANDLING LONG TEXTS AND AGGREGATING EMBEDDING VECTORS

Modern embedding models – including OpenAI Embeddings, BERT-based transformers, and other contextual encoders – are subject to context window limitations, i.e., the maximum number of tokens that can be processed in a single model call. For example, the text-embedding-3-small model from OpenAI has a limit of approximately 8192 tokens, which is often insufficient for full-length editorial materials such as analytical articles, interviews, or multi-part longreads.

To ensure correct vectorization of such content, a technique known as chunking is applied. This involves splitting the source text into logically continuous fragments, each of which fits within the model's allowable input size. It is important to note that if a document is too large to fit into a single model call, it can – without quality loss – be split into equal-sized parts (while preserving word and sentence boundaries), and then the arithmetic mean of the resulting vectors can be used as the overall embedding for the entire document.

This approach helps preserve the document's global semantic meaning while mitigating risks associated with context breaks or fragment-level inconsistencies. In practice, the averaged vector shows robust performance in tasks such as search, classification, and clustering, particularly when the total length of the input text does not exceed 3–5× the model's context window. Several aggregation methods can be used after chunking (table 3).

Table 3: Methods for handling long texts in vectorization [9, 10]

Method	Description	Advantages	Limitations
Chunk-based segmentation	Text is split into chunks based on token count, preserving lexical boundaries.	Simple implementation, controlled input length.	Potential loss of semantic coherence between fragments.
Mean pooling	Arithmetic mean of all chunk embeddings.	Stable, efficient, low-complexity.	Ignores varying importance of different parts of the text.
Weighted pooling	Aggregation with weights (e.g., TF-IDF, header importance).	Prioritizes semantically significant segments.	Requires additional metadata and preprocessing.
Max pooling / attention	Retains only strongest activations or uses self-attention across chunk vectors.	Preserves key semantic signals.	More complex, may be unstable on short or uneven inputs.

In real-world editorial pipelines, mean pooling of chunk embeddings is the most widely adopted method. It offers a practical trade-off between implementation simplicity, efficiency, and semantic quality. With tens of thousands of articles processed daily by the embedding pipeline, this approach offers scalability with semantic coherence without requiring more complex models such as long-form transformers or memory-augmented architectures.

In summary, the chunk-agnostic segmentation and embedding vector aggregation provide a reliable and scalable long-text vectorization solution. The solution is fully compatible with PostgreSQL-based vector search infrastructures and third-party embedding APIs, allowing editorial infrastructures to perform precise semantic analysis without compromising performance.

6. CONCLUSION

The integration of search embeddings into editorial processes brings new opportunities for intelligent content management, enabling semantic search, personalization, and topic-based navigation. Novel embedding models, combined with PostgreSQL and the pgvector extension, provide a high-scalability and high-performance vector storage and retrieval infrastructure that can be seamlessly integrated into existing editorial systems. Nice architecture tone, effective handling of long strings of text, and the flexibility of SQL-based querying make this solution especially attractive to media sites wishing to improve search and recommendations quality and automate content processing.

REFERENCES

1. J. Rojas-Simón, Y. Ledeneva, R. A. García-Hernández. **A Dimensionality Reduction Approach for Text Vectorization in Detecting Human and Machine-generated Texts,** *Computación y Sistemas,* vol. 28, no. 4, pp. 1919-1929, 2024.
2. New embedding models and API updates / Open AI // URL: https://openai.com/index/new-embedding-models-and-api-updates (date of application: 25.10.2025).
3. P. John, K. Zaklová, J. Lazúr, J. Hynek, T. Hruška. **A Self-Hosted Approach to Automatic CI/CD Using Open-Source Tools on Low-Power Devices,** *2024 IEEE 17th International Scientific Conference on Informatics (Informatics), IEEE,* pp. 100-107, 2024.
4. M. Sarim. **Generating reliable software project task flows using large language models through prompt engineering and robust evaluation,** *Scientific Reports,* vol. 15, no. 1, pp. 35194, 2025.
5. G. Khekare, C. Masudi, Y. K. Chukka, D. P. Koyyada. **Text normalization and summarization using advanced natural language processing,** *2024 International Conference on Integrated Circuits and Communication Systems (ICICACS), IEEE,* pp. 1-6, 2024.
6. Databases / Stack Overflow Developer Survey 2024 // URL: https://survey.stackoverflow.co/2024/technology#1-databases (date of application: 27.10.2025).
7. Supercharging vector search performance and relevance with pgvector 0.8.0 on Amazon Aurora PostgreSQL / AWS Database Blog // URL: https://aws.amazon.com/ru/blogs/database/supercharging-vector-search-performance-and-relevance-with-pgvector-0-8-0-on-amazon-aurora-postgresql/ (date of application: 26.10.2025).
8. M. Abbasi. **Adaptive and scalable database management with machine learning integration: A PostgreSQL case study,** *Information,* vol. 15, no. 9, pp. 574, 2024.
9. A. Blazhkovskii. **Developing user interfaces with a focus on inclusivity,** *International journal of Professional Science,* no. 2-2, pp. 73-80, 2025.
10. L. Stankevičius, M. Lukoševičius. **Extracting sentence embeddings from pretrained transformer models,** *Applied Sciences,* vol. 14, no. 19, pp. 8887, 2024.