ArteriaAI: Document Understanding Product

Author
Ekaba Bisong

Contents

Chapter 1

Introduction

Chapter 2

System Design

2.1 Storage and Retrieval Strategy for Vector Embeddings and Text Chunks

In designing a robust system for managing vector embeddings and associated text chunks, it is imperative to strike an optimal balance between latency, cost-effectiveness, and scalability. This section presents a comprehensive analysis of various storage options, culminating in a justified recommendation for the most suitable architecture.

Options for Storage

1. In-Memory Storage:
    - Pros: Extremely fast access, low latency.
    - Cons: Limited by memory capacity, not suitable for large-scale data, data is lost on restart.
2. File Storage (e.g., Cloud Storage):
    - Pros: Cost-effective, easy to implement, scalable.
    - Cons: Higher latency for read/write operations, limited querying capabilities.
3. Database Storage (e.g., Firestore, Bigtable, or a Vector Database like Pinecone):
    - Pros: Scalable, provides efficient querying and indexing, good for structured data, supports complex queries.
    - Cons: Higher cost, moderate latency compared to in-memory storage.

2.2 Recommended Approach: Hybrid Storage with Firestore and Pinecone

By using Firestore for metadata and Pinecone for embeddings, the system can quickly retrieve metadata and perform low-latency searches for embeddings. This hybrid approach

leverages the cost-effectiveness of Firestore for structured data and the specialized capabilities of Pinecone for vector data. Both Firestore and Pinecone are designed to scale automatically, ensuring the system can handle increasing amounts of data without significant performance degradation. In essence, we'll use Firestore for Metadata and Text Chunks. This will store the text chunks and associated metadata and allow for efficient querying and retrieval of text and metadata. In using Pinecone (or any other vector database) for embeddings we can store the vector embeddings efficiently. And vector databases supports low-latency similarity searches and vector operations.

Firestore for Metadata and Text Chunks

- **Real-time Capabilities:** Firestore's real-time database functionality enables swift access to metadata and text chunks, crucial for maintaining system responsiveness.
- **Low-latency Operations:** The database provides exceptionally low-latency read and write operations, particularly beneficial for structured data retrieval.
- **Cost-effective Scaling:** Firestore's pay-as-you-go pricing model ensures cost-effectiveness, especially for variable workloads, aligning well with dynamic system requirements.
- **Automatic Scalability:** The database is engineered to scale automatically with increasing data volume and read/write operations, eliminating the need for manual scaling interventions.

Pinecone (or any other Vector Database) for Embeddings

- **Vector Optimization:** Pinecone's architecture is specifically optimized for vector operations, ensuring high efficiency in storing and querying embeddings.
- **Low-latency Similarity Searches:** The database excels in performing low-latency similarity searches, a critical feature for applications relying on rapid embedding retrieval.
- **Performance-Cost Balance:** While potentially incurring higher costs than general-purpose databases, Pinecone's performance benefits for vector data often justify the investment.
- **Seamless Scalability:** Pinecone is designed to handle large-scale embedding data, offering seamless scalability as the dataset expands.

Why Not In-Memory Storage?

1. **Limited by Memory Capacity:** In-memory storage is not suitable for large-scale data as it is limited by the available memory.
2. **Data Volatility:** Data stored in memory is lost if the service is restarted or crashes, which is not ideal for persistent storage requirements.
3. **Scalability:** Managing large volumes of data in memory becomes impractical and expensive as the dataset grows.

2.3 Secure Secret Management with Google Secret Manager

In the architecture of our system, the secure management of sensitive information is paramount. To address this critical requirement, we have integrated Google Secret Manager, a robust and scalable solution for storing and managing confidential data.

Rationale for Adoption

The implementation of Google Secret Manager in our system design is predicated on several key factors:

1. **Enhanced Security Posture:** Google Secret Manager employs state-of-the-art encryption protocols for data at rest and in transit, leveraging Google's advanced security infrastructure. This significantly mitigates the risk of unauthorized access to sensitive information.
2. **Granular Access Control:** The solution integrates seamlessly with Google Cloud Identity and Access Management (IAM), enabling fine-grained control over secret access. This allows for the implementation of the principle of least privilege, ensuring that entities within the system have access only to the secrets they require for their specific functions.
3. **Version Control and Auditing:** The versioning capability of Secret Manager facilitates the management of secret lifecycles, allowing for easy rollbacks and historical tracking. Coupled with comprehensive auditing features, this provides a clear trail of secret access and modifications, enhancing our system's compliance with security best practices.
4. **Seamless Integration:** Given our system's reliance on Google Cloud Platform services such as Firestore, Secret Manager offers native integration, streamlining the process of secret retrieval and management across our application ecosystem.
5. **Centralized Management:** By providing a unified platform for secret storage, Secret Manager reduces the operational complexity associated with managing secrets across disparate system components. This centralization minimizes the attack surface and simplifies secret rotation and revocation procedures.

Implementation Strategy

In our system, Google Secret Manager is utilized to securely store and manage a variety of sensitive data, including:

- API authentication tokens for external services (e.g., Pinecone API keys)
- Database connection strings and credentials (e.g., Firestore access parameters)
- Encryption keys used for data protection within the application
- Environment-specific configuration data containing sensitive information

The application architecture is designed to retrieve these secrets at runtime, adhering to the principle of dynamic secret management. This approach ensures that sensitive data

is never hard-coded or stored in configuration files, significantly reducing the risk of inadvertent exposure through code repositories or configuration management systems.