

AI-Powered Personal Document Assistance System

✓ Overview

The **AI-Powered Personal Document Assistance System** allows users to securely upload private documents (PDFs and images) and later retrieve relevant content using **natural language queries**, with summarization and audio narration.

Built entirely on **Google Cloud Platform (GCP)**, the system leverages **Document AI**, **Vertex AI**, **Gemini-2.5-flash Model**, **Cloud Run**, **Firestore**, **Text-to-Speech API** and **BigQuery** to perform intelligent text extraction, embedding, and retrieval through **vector similarity search**, re-ranking of chunks, Summarization and audio narration.

This architecture ensures **data privacy, scalability, and multilingual adaptability** by leveraging Google's advanced AI models and server-less infrastructure.

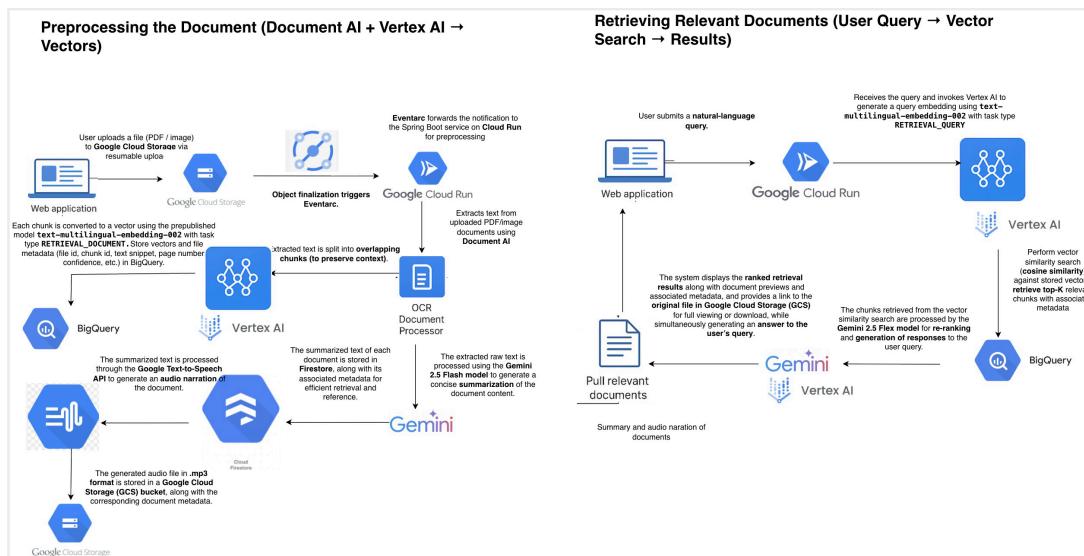
Git Hub:

<https://github.com/lalamanil/AIPrivateDocumentAssistance.git>

Web Application link:

<https://personalclerk-ivtq4ydqua-uc.a.run.app>

✓ System Architecture



Core Flow:

Upload & Storage

- Users upload documents through a web interface.
- Files are transferred to **Google Cloud Storage (GCS)** via **Resumable Upload** for reliability and large-file support.
- Upon successful upload, GCS triggers an **Eventarc event** (OBJECT_FINALIZE) to initiate downstream processing.

Document Preprocessing (Cloud Run + Document AI)

- **Eventarc** forwards the event to a **Spring Boot microservice** deployed on **Cloud Run**.
- The microservice retrieves the uploaded document from GCS and invokes **Document AI (OCR Processor)** to extract the full text content.

Text Chunking & Vector Embedding

- Extracted text is segmented into **overlapping chunks** (typically 300–500 tokens with 20–30% overlap) to preserve contextual continuity.
- Each chunk is embedded into a **vector representation** using **Vertex AI's pre-trained model**:
text-multilingual-embedding-002 with task type RETRIEVAL_DOCUMENT.
- These embeddings, along with metadata (file ID, filename, chunk index, text snippet, etc.), are stored in **BigQuery**.

Summarization and Audio Generation

- The extracted **raw text** is processed using the **Gemini 2.5 Flash model** to generate a concise **summary** of the document's content.
- Each summarized text is stored in **Firebase**, indexed by its corresponding **document ID** and associated metadata.
- The summarized text is then passed to the **Google Text-to-Speech API**, which produces an **audio narration** in .mp3 format.
- The resulting **audio file** is stored in a **GCS bucket**, accompanied by the relevant **document metadata** for reference and accessibility.

Query & Retrieval

When a user submits a natural-language query:

- The query text is embedded using the same **Vertex AI model** (task_type =

RETRIEVAL_QUERY).

- **BigQuery** executes a **vector similarity search** (cosine distance) to identify the most relevant text chunks.

Retrieved results are ranked, aggregated, and displayed on the web interface with **contextual previews** and links to the original files.

Detailed Component Architecture

| Component | Technology | Description |
|-----------------------------|------------------------------------|---|
| Frontend UI | HTML5, CSS, JavaScript (Thymeleaf) | Web interface for document upload, progress display, and querying. |
| Storage Layer | Google Cloud Storage (GCS) | Stores uploaded files; triggers Eventarc when files are finalized. |
| Event Routing | Eventarc | Detects finalized uploads in GCS and invokes Cloud Run endpoint. |
| Compute Layer | Cloud Run (Spring Boot) | Core microservice that handles event ingestion, text extraction, chunking, embedding, and querying. |
| Text Extraction | Document AI (OCR Processor) | Extracts text content from PDFs, images, and scanned documents. |
| Embedding Generation | Vertex AI Embedding API | Converts text chunks and queries into 768-dimensional vector embeddings. |

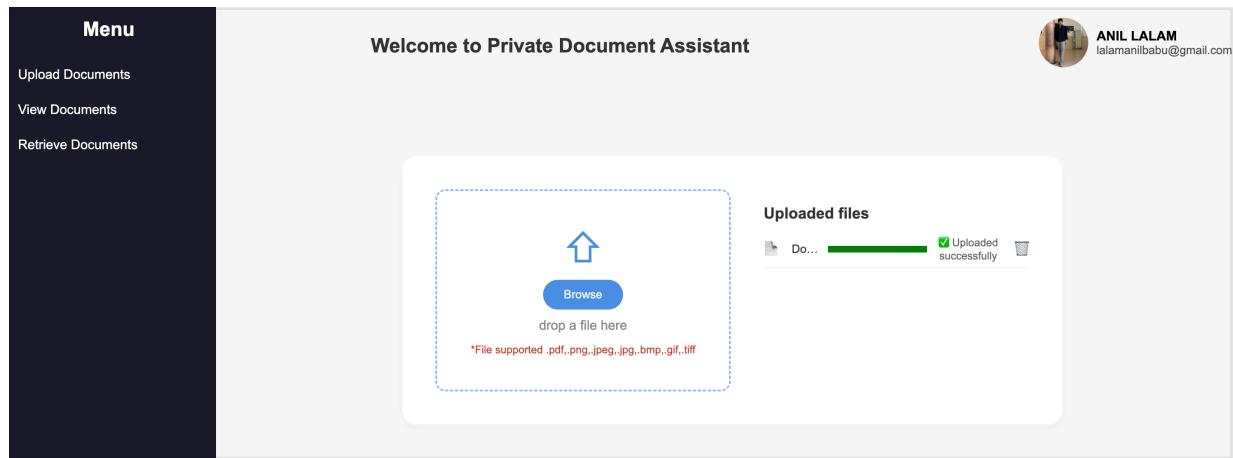
| | | |
|------------------------|---|--|
| Vector Database | BigQuery (Vector column support) | Stores embeddings and performs cosine similarity search for retrieval. |
|------------------------|---|--|

Data Flow

Document Ingestion

1. User → Uploads file → GCS via web app (Resumable Upload)

File sent to **GCS** via **Resumable Upload**



Java code handles session URL creation for resumable upload

```

@Override
public Map<String, String> initiateResumableUploadSessionUri(String objectName, String contentType) {
    // TODO Auto-generated method stub

    if (!NotNullEmptyUtility.notNullEmptyCheck(objectName)) {
        LOGGER.info("objectName cannot be null or empty");
        throw new PrivateDocumentException("objectName cannot be null or empty", 400);
    }

    if (!NotNullEmptyUtility.notNullEmptyCheck(contentType)) {
        LOGGER.info("Object contentType cannot be null or empty");
        throw new PrivateDocumentException("object ContentType cannot be null or empty", 400);
    }

    if (!ApplicationConstants.SUPPORTED_CONTENT_TYPE.contains(contentType)) {
        LOGGER.info(contentType + " is not supported. Supported contentTpes are:" +
                    + ApplicationConstants.SUPPORTED_CONTENT_TYPE);
        throw new PrivateDocumentException(contentType + " is not supported. Supported contentTpes are:" +
                    + ApplicationConstants.SUPPORTED_CONTENT_TYPE, 400);
    }

    String accessToken = AccessTokenUtility.getAccessToken();
    if (null == accessToken) {
        LOGGER.info("accessToken is null or empty.");
        throw new PrivateDocumentException("accessToken is null or empty", 500);
    }
    Map<String, String> metadata = new HashMap<String, String>();
    metadata.put("name", objectName);
    metadata.put("contentType", contentType);
    try {
        String metadataJson = ObjectMapperSingleton.INSTANCE.get.ObjectMapper().writeValueAsString(metadata);
        LOGGER.info("metadataJson:" + metadataJson);
        // Http Client
        HttpClient httpClient = HttpClient.newBuilder().version(Version.HTTP_2).build();

        LOGGER.info(ApplicationConstants.INITIATE_RESUMABLE_UPLOAD_URL);
        // Build HttpRequest
        HttpRequest httpRequest = HttpRequest.newBuilder()
            .uri(URI.create(ApplicationConstants.INITIATE_RESUMABLE_UPLOAD_URL)).timeout(Duration.ofSeconds(20))
            .header("Authorization", "Bearer " + accessToken)
            .header("Content-Type", "application/json; charset=UTF-8")
            .POST(BodyPublishers.ofString(metadataJson)).build();

        // Send Http Request
        HttpResponse<Void> httpResponse = httpClient.send(httpRequest, BodyHandlers.discard());
        if (200 != httpResponse.statusCode() && 201 != httpResponse.statusCode()) {
            String msg = String.format("Failed to initiate resumable upload. HTTP %d", httpResponse.statusCode());
            LOGGER.info(msg);
            throw new PrivateDocumentException(msg, 500);
        }
        String uploadSessionUrl = httpResponse.headers().firstValue("Location")
            .orElseThrow(() -> new PrivateDocumentException("No Location header returned from GCS", 500));
        LOGGER.info("uploaded Session url:" + uploadSessionUrl);

        return Map.of("uploadUrl", uploadSessionUrl);
    } catch (JsonProcessingException e) {
        // TODO: handle exception
        String msg = "Exception while converting metadata object to json:" + e.getMessage();
        LOGGER.info(msg);
        throw new PrivateDocumentException(msg, 500);
    } catch (InterruptedException e) {
        // TODO: handle exception
        e.printStackTrace();
        Thread.currentThread().interrupt();
        throw new PrivateDocumentException("Interrupted while Initiating Resumable upload to get session uri", 500);
    } catch (IOException e) {
        // TODO: handle exception
        e.printStackTrace();
        throw new PrivateDocumentException("I/O exception while Initiating Resumable upload to session uri", 500);
    }
}

```

Java script Code Upload file to GCS in Chunks:

```
⊕async function uploadFileWithResumableGCS(email, file, li) {
    const progress = li.querySelector("progress");
    const status = li.querySelector(".upload-status");
    ⊕ try {
        const encodedObjectName = encodeURIComponent(email + "/" + file.name);
        const encodedType = encodeURIComponent(file.type);
        const resumableUrl = `/initiateResumableUpload?objectName=${encodedObjectName}&contentType=${encodedType}`;

        console.log(resumableUrl);
        const response = await fetch(resumableUrl);
        console.log('status:' + response.status)
        if (!response.ok) {
            status.textContent = "✖ Not uploaded (session URI failed)";
            return;
        }
        const data = await response.json();
        console.log('response json:' + data);
        const uploadUrl = data.uploadUrl;
        console.log('uploadUrl:' + uploadUrl);
        // upload the chunks

        if (!uploadUrl) {
            status.textContent = "✖ Not uploaded (session URI failed)";
            return;
        }

        status.textContent = "Uploading...";

        const chunkSize = 5 * 1024 * 1024; //5MB
        let offset = 0;
        while (offset < file.size) {
            const chunk = file.slice(offset, offset + chunkSize);
            const chunkStart = offset;

            const chunkEnd = Math.min(offset + chunkSize - 1, file.size - 1);
            const totalSize = file.size;
```

```

let success = false;
let attempts = 0;
const maxRetries = 3;

while (!success && attempts < maxRetries) {

    try {
        const res = await fetch("/uploadChunks", {
            method: "POST",
            headers: {
                "Content-Type": 'application/octet-stream',
                "SessionUrl": uploadUrl,
                "Content-Length": chunk.size.toString(),
                "Content-Range": `bytes ${chunkStart}-${chunkEnd}/${totalSize}`,
            },
            body: chunk
        });
        if (res.status === 200) {
            const finalMetadata = await res.text();
            console.log('upload Completed! final metadata received:', finalMetadata);
            progress.value = 100;
            offset = file.size;
            success = true;
        } else {
            if (res.status === 308) {
                offset = offset + chunk.size;
                progress.value = Math.round((offset / totalSize) * 100);
                success = true;
            } else {
                const errorBody = await res.text();
                console.error('GCS Error Details:', errorBody);
                throw new Error(`Chunk upload failed: HTTP ${res.status}`);
            }
        }
    }
}

```

```

} catch (err) {
    alert(err);
    attempts++;
    console.warn(`Retry ${attempts}/${maxRetries} for chunk at offset ${offset}`);
    if (attempts < maxRetries) {
        await new Promise(r => setTimeout(r, 2000)); //wait before retry
    } else {
        status.textContent = "❌ Upload failed (chunk retry limit)";
        return;
    }
}
}

progress.value = 100;
status.textContent = "✅ Uploaded successfully";
} catch (error) {
    console.error("Upload error:", error);
    status.textContent = "❌ Upload failed (network/unknown)";
}

```

Documents are stored in GCS Bucket

The screenshot shows the 'Bucket details' page for the 'documentassistance' bucket. The bucket is located in 'us-central1 (Iowa)', has a 'Standard' storage class, and 'Not public' public access. It uses 'Soft Delete' protection. The 'Objects' tab is selected, displaying a list of PDF files. The table includes columns for Name, Size, Type, Created, and Storage class. The files were uploaded on October 20, 2025, between 5:09:44 PM and 5:12:18 PM.

| Name | Size | Type | Created | Storage class |
|----------------|----------|-----------------|--------------------------|---------------|
| [REDACTED] | 416.8 KB | application/pdf | Oct 20, 2025, 5:09:44 PM | Standard |
| [REDACTED] | 656.5 KB | application/pdf | Oct 20, 2025, 5:09:44 PM | Standard |
| [REDACTED] (1) | 6 MB | application/pdf | Oct 20, 2025, 5:11:23 PM | Standard |
| [REDACTED] (1) | 868.6 KB | application/pdf | Oct 20, 2025, 5:10:31 PM | Standard |
| [REDACTED] (1) | 25 MB | application/pdf | Oct 20, 2025, 5:11:26 PM | Standard |
| [REDACTED] | 293 KB | application/pdf | Oct 20, 2025, 5:10:31 PM | Standard |
| [REDACTED].pdf | 1.6 MB | application/pdf | Oct 20, 2025, 5:10:31 PM | Standard |
| [REDACTED] | 371.3 KB | application/pdf | Oct 20, 2025, 5:12:18 PM | Standard |
| [REDACTED] | 708.2 KB | application/pdf | Oct 20, 2025, 5:12:18 PM | Standard |
| [REDACTED] (1) | 7.8 MB | application/pdf | Oct 20, 2025, 5:11:23 PM | Standard |

2. GCS → triggers Eventarc → Cloud Run service

Detects finalized uploads and routes the event to **Cloud Run**.

The screenshot shows the 'Service details' page for the 'personalclerk' service. The 'Triggers' tab is selected. A new trigger is being created with the following configuration:

- Name:** trigger-embedded
- Event provider:** Cloud Storage
- Event type:** google.cloud.storage.object.v1.finalized
- Receive events from:** documentassistance (us-central1)
- Destination:** personalclerk (us-central1)
- Service URL path:** /event
- Service account:** 852426054869-compute@developer.gserviceaccount.com

3. Cloud Run → Calls Document AI → Extracts text

Code to extract raw text from documents using Document OCR (Document AI)

```
public static String processDocument(byte[] filedata, String mimeType) {
    String rawText = null;
    if (null != documentProcessorServiceClient) {
        if (null != filedata) {
            String name = String.format("projects/%s/locations/%s-processors/%s", ApplicationConstants.PROJECT_ID,
                ApplicationConstants.DOCUMENT_AI_PROCESSORS_LOCATION, ApplicationConstants.PROCESSOR_ID);
            // converting the image data to a Buffer and base64 encode it.
            ByteString content = ByteString.copyFrom(filedata);
            RawDocument document = RawDocument.newBuilder().setContent(content).setMimeType(mimeType).build();
            // Configure the process request
            ProcessRequest request = ProcessRequest.newBuilder().setName(name).setRawDocument(document).build();
            ProcessResponse processResponse = documentProcessorServiceClient.processDocument(request);
            Document documentResponse = processResponse.getDocument();
            LOGGER.info("Document processing is completed..");
            rawText = documentResponse.getText();
        } else {
            LOGGER.info("filedata bytes is null. Please provide valid byte array.");
        }
    } else {
        LOGGER.info("documentProcessorServiceClient is null. Please check application logs");
    }
    return rawText;
}
```

4. Text → Chunked + Overlapped

Extracted text is divided into overlapping chunks to preserve semantic continuity.

```
package com.document.personal.assistance.utility;

import java.util.ArrayList;

public class TextChunkerUtility {

    public static List<String> chunkText(String text, int maxChars, int overlap) {
        List<String> chunkList = new ArrayList<String>();
        int start = 0;

        int minChunckSize = maxChars / 2;

        while (start < text.length()) {
            if ((text.length() - start) <= maxChars) {
                chunkList.add(text.substring(start));
                break;
            }
            int end = Math.min(start + maxChars, text.length());
            int lastboundary = Math.max(Math.max(text.lastIndexOf("."), end - 1), text.lastIndexOf("!", end - 1));
            if (lastboundary > start + minChunckSize) {
                end = lastboundary + 1;
            }
            String chunk = text.substring(start, end).trim();
            if (!chunk.isEmpty()) {
                chunkList.add(chunk);
            }
            start = Math.max(end - overlap, start + 1);
        }
        return chunkList;
    }
}
```

5. Each chunk → Embedded via Vertex AI → Embeddings returned

Each chunk is embedded into a numerical vector using the Vertex AI model (text-multilingual-embedding-002)

```
79  public static List<float[]> getEmbeddings(List<String> chunkList, String taskType) {
80     List<float[]> embeddings = new ArrayList<float[]>();
81     if (null != chunkList && !chunkList.isEmpty()) {
82         if (null != predictionServiceClient) {
83             // Important: For embeddings, you don't deploy your own endpoint.
84             // You call the published model directly.
85             String endPointName = String.format("projects/%s/locations/%s/publishers/google/models/%s",
86                 ApplicationConstants.PROJECT_ID, ApplicationConstants.LOCATION_ID,
87                 ApplicationConstants.TEXT_EMBEDDING_MODEL);
88             System.out.println("endPointName is:" + endPointName);
89
90             int exactChuckSize = chunkList.size() / ApplicationConstants.NUMBER_OF_CHUNKS_PER_REQUEST;
91             int leftOverChunk = chunkList.size() % ApplicationConstants.NUMBER_OF_CHUNKS_PER_REQUEST;
92             int track = 0;
93             if (exactChuckSize > 0) {
94                 for (int i = 0; i < exactChuckSize; i++) {
95                     // predictRequest
96                     PredictRequest.Builder request = PredictRequest.newBuilder().setEndpoint(endPointName);
97                     List<String> subsetChunkList = chunkList.sublist(track,
98                         track + ApplicationConstants.NUMBER_OF_CHUNKS_PER_REQUEST);
99                     predictionRequestCall(subsetChunkList, request, embeddings, track, taskType);
100                     track = track + ApplicationConstants.NUMBER_OF_CHUNKS_PER_REQUEST;
101                     System.out.println("**** at track:" + track);
102                 }
103
104                 if (leftOverOverChunk > 0) {
105                     // predictRequest
106                     PredictRequest.Builder request = PredictRequest.newBuilder().setEndpoint(endPointName);
107                     List<String> subsetChunkList = chunkList.sublist(track, chunkList.size());
108                     predictionRequestCall(subsetChunkList, request, embeddings, track, taskType);
109                     System.out.println("**** at leftOver track:" + (chunkList.size() - track));
110                 }
111             } else {
112                 System.out.println("predictionServiceClient is null. Please check your application logs");
113                 throw new PrivateDocumentException(
114                     "predictionServiceClient is null. Please check your application logs", 500);
115             }
116         } else {
117             System.out.println("chunlklist is null or empty.");
118             throw new PrivateDocumentException("chunlklist is null or empty for userprompt.", 500);
119         }
120     }
121
122     return embeddings;
123 }
124 }
```

6. Cloud Run → Stores embeddings + metadata → BigQuery

Create the dataset and table in Google Big Query:

The screenshot shows the Google BigQuery web interface. The top navigation bar includes options like 'Search BigQuery resources', 'Schema', 'Details', 'Preview', 'Table Explorer', 'Preview' (which is selected), 'Insights', 'Lineage', 'Data Profile', and 'Data Quality'. Below the navigation is a search bar and a 'Filter' input field. The main area displays the schema for a table named 'chunks'. The schema table has columns: Field name, Type, Mode, Description, Key, Collation, Default Value, Policy Tags, and Data Policy. The 'embedding' column is highlighted with a red border. The table data includes:

| Field name | Type | Mode | Description | Key | Collation | Default Value | Policy Tags | Data Policy |
|--------------|-----------|----------|-----------------|-----|-----------|---------------------|-------------|-------------|
| row_id | STRING | NULLABLE | - | - | - | - | - | - |
| doc_id | STRING | NULLABLE | - | - | - | - | - | - |
| chunk_id | STRING | NULLABLE | - | - | - | - | - | - |
| text | STRING | NULLABLE | - | - | - | - | - | - |
| embedding | FLOAT | REPEATED | - | - | - | - | - | - |
| created_at | TIMESTAMP | NULLABLE | - | - | - | CURRENT_TIMESTAMP() | - | - |
| userid | STRING | NULLABLE | - | - | - | - | - | - |
| content_type | STRING | NULLABLE | Type Of Content | - | - | - | - | - |

At the bottom of the schema view are two buttons: 'Edit schema' and 'View row access policies'.

Embeddings and metadata are stored in **BigQuery** for efficient vector search.

```

List<Map<String, Object>> rowList = new ArrayList<Map<String, Object>>();
for (int i = 0; i < vectorList.size(); i++) {
    Map<String, Object> row = new HashMap<String, Object>();
    row.put("row_id", objectName + "_chunck" + chunkCount);
    row.put("doc_id", objectName);
    row.put("chunk_id", "chunck" + chunkCount);
    row.put("text", chunkList.get(i));
    row.put("userid", userid);
    row.put("content_type", contentType);
    float[] floatvectors = vectorList.get(i);
    List<Double> emplist = new ArrayList<Double>();
    for (float f : floatvectors) {
        emplist.add((double) f);
    }
    row.put("embedding", emplist);
    rowList.add(row);
    chunkCount++;
}
LOGGER.info("Total vector rows trying to insert into Bigquery is:" + rowList.size());
if (!rowList.isEmpty()) {
    BigQueryUtility.insertRowsBatch(rowList, ApplicationConstants.TABLE_NAME);
}

```

```

public static void insertRowsBatch(List<Map<String, Object>> rowData, String tableName) {
    if (null != bigQuery) {
        TableId tableId = TableId.of(ApplicationConstants.DATA_SET, tableName);
        InsertAllRequest.Builder builder = InsertAllRequest.newBuilder(tableId);
        for (Map<String, Object> row : rowData) {
            builder.addRow(row);
        }
        try {
            InsertAllResponse response = bigQuery.insertAll(builder.build());
            if (response.hasErrors()) {
                response.getInsertErrors().forEach((index, error) -> {
                    System.err.println("Row at index " + index + " failed with errors:" + error);
                });
            } else {
                LOGGER.info("Inserted " + rowData.size() + " rows to " + tableId);
            }
        } catch (BigQueryException e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    } else {
        LOGGER.info("bigQuery object is null. Please check application logs");
    }
}

```

Query & Retrieval

1. User → Enters natural-language query → Web UI

The user submits a natural-language question via the web UI.
A “View Documents” option displays previously uploaded files.

The screenshot shows a web-based document management system. On the left, a sidebar menu includes options like 'Upload Documents', 'View Documents', and 'Retrieve Documents'. The main area is titled 'Welcome to Private Document Assistant' and features a 'Your Documents' section listing several PDF files. To the right, a terminal window displays instructions for deploying a Spring Boot application to Google Cloud. It lists prerequisites (Google Cloud account, SDK installed locally, a simple Spring Boot application), provides a command to check gcloud version (gcloud --version), and shows the output of that command. Below the terminal, it says 'Then Check the currently active project' followed by the command 'gcloud config get-value project'.

2. Web UI → Sends query → Cloud Run service

Java script code to send user query to Cloud Run Service to get relevant Documents based on Context.

```

async function sendQuery() {
  const query = queryInput.value.trim();
  if (!query) {
    return;
  }
  appendMessage("user", query);
  queryInput.value = "";
  appendMessage("assistant", "Searching your documents...");
  const body = {
    userId: emailId,
    userPromptForDocSearch: query,
    relevanceCutoff: 0.37 // we can adjust this or make it configurable
  };
  try {
    const response = await fetch("/userdocsbyprompt", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(body)
    });
    const docs = await response.json();
    if (!response.ok) {
      throw new Error(`HTTP ${response.status}`);
    }
    if (!Array.isArray(docs.documents) || docs.documents.length === 0) {
      chatMessages.lastChild.textContent = "No relevant documents found.";
      docList.innerHTML = "";
      preview.style.display = "none";
      preview.innerHTML = ""; //clear old preview
      return;
    }
    chatMessages.lastChild.textContent = `Found ${docs.documents.length} matching documents.`;
    documents = docs.documents;
    renderDocuments(currentPage);
  } catch (err) {
    console.error("Error retrieving docs:", err);
    chatMessages.lastChild.textContent = "✖ Error retrieving documents.";
  }
}

```

3. Cloud Run → Embeds query → Vertex AI (task type: RETRIEVAL_QUERY)

The Cloud Run service converts the user query into a vector using Vertex AI with task_type = RETRIEVAL_QUERY

```

@Override
public List<VectorSearchResultsModel> searchDocumentsForUserPrompt(String userid, String promptString,
    float relevanceCutoff) {
    // TODO Auto-generated method stub
    if (!NotNullEmptyUtility.notNullEmptyCheck(userid)) {
        throw new PrivateDocumentException("userid cannot be null or empty. Please provide valid userid", 400);
    }
    if (!NotNullEmptyUtility.notNullEmptyCheck(promptString)) {
        throw new PrivateDocumentException("promptString cannot be null or empty. Please provide valid userid",
            400);
    }
    List<float[]> vectorList = TextEmbeddingWithPredictionServiceUtility.getEmbeddings(NSArray.asList(promptString),
        ApplicationConstants.TASK_TYPE_RETRIEVAL_QUERY);
    if (null != vectorList && !vectorList.isEmpty()) {
        float[] vector = vectorList.get(0);
        List<Double> doubleVector = new ArrayList<Double>(vector.length);
        for (float f : vector) {
            doubleVector.add((double) f);
        }
        LOGGER.info("vector for user prompt is: " + doubleVector);
        List<VectorSearchResultsModel> list = BigQueryUtility.vectorSearch(userid,
            ApplicationConstants.TOP_CHUNK_COUNT, doubleVector);
        return GeneralUtility.filterVectorSearchBasedOnRelevanceCutoff(list, relevanceCutoff);
    } else {
        LOGGER.info("vectorList for user prompt is null or empty.");
        throw new PrivateDocumentException("vectorList for user prompt is null or empty.", 500);
    }
}

```

4. Cloud Run → Performs cosine vector similarity in BigQuery

Cloud Run executes a **cosine similarity** search to identify relevant document chunks

```

public static List<VectorSearchResultsModel> vectorSearch(String userid, int topK, List<Double> queryVector) {
    List<VectorSearchResultsModel> vectorSearchResultsModels = new ArrayList<VectorSearchResultsModel>();
    if (null != bigQuery) {
        String vectorSearchQuery = getVectorQuery();
        if (null != vectorSearchQuery && !vectorSearchQuery.trim().isEmpty()) {
            // prepare query Configuration with parameters
            QueryJobConfiguration queryJobConfiguration = QueryJobConfiguration.newBuilder(vectorSearchQuery)
                .addNamedParameter("userId", QueryParameterValue.string(userid))
                .addNamedParameter("topK", QueryParameterValue.int64(topK))
                .addNamedParameter("queryVector", QueryParameterValue.array(queryVector.toArray(new Double[0]),
                    StandardSQLTypeName.FLOAT64))
                .build();
            // Run the Query
            try {
                TableResult tableResult = bigQuery.query(queryJobConfiguration);
                Iterable<FieldValueList> iterable = tableResult.iterateAll();
                for (FieldValueList row : iterable) {
                    VectorSearchResultsModel vectorSearchResultsModel = new VectorSearchResultsModel();
                    vectorSearchResultsModel.setDocumentId(row.get("doc_id").getStringValue());
                    List<Double> distanceList = new ArrayList<Double>();
                    List<FieldValues> distanceValues = row.get("distances").getRepeatedValue();
                    if (null != distanceValues) {
                        distanceValues.forEach(fv -> {
                            distanceList.add(fv.getDoubleValue());
                        });
                    }
                    vectorSearchResultsModel.setDistanceList(distanceList);
                    vectorSearchResultsModel.setNumberOfChunks(row.get("numberOfChucks").getLongValue());
                    vectorSearchResultsModel.setContentType(row.get("contenttype").getStringValue());
                    long micros = row.get("created_at").getTimestampValue();
                    Instant instant = Instant.ofEpochMilli(micros / 1000);
                    LocalDateTime localDateTime = LocalDateTime.ofInstant(instant, ZoneId.of("UTC"));
                    vectorSearchResultsModel.setCreatedAt(localDateTime.toString());
                    vectorSearchResultsModels.add(vectorSearchResultsModel);
                }
            } catch (InterruptedException e) {
                // TODO: handle exception
                LOGGER.info("if the current thread gets interrupted while waiting for the query to complete:"
                    + e.getMessage());
                e.printStackTrace();
                throw new PrivateDocumentException(e.getMessage(), e, 500);
            } catch (JobException e) {
                // TODO: handle exception
            }
        }
    }
}

```

```

    // TODO: handle exception
    LOGGER.info("if the job completes unsuccessfully:" + e.getMessage());
    e.printStackTrace();
    throw new PrivateDocumentException(e.getMessage(), e, 500);

} catch (BigQueryException e) {
    // TODO: handle exception
    LOGGER.info("upon failure:" + e.getMessage());
    e.printStackTrace();
    throw new PrivateDocumentException(e.getMessage(), e, 500);
}

} else {
    LOGGER.info("VectorSearchQuery is either null or empty. Please check application logs");
    throw new PrivateDocumentException(
        "VectorSearchQuery is either null or empty. Please check application logs", 500);
}
} else {
    LOGGER.info("bigQuery object is null. Please check application logs.");
    throw new PrivateDocumentException("bigQuery object is null. Please check application logs.", 500);
}

return vectorSearchResultsModels;
}
}

```

5. BigQuery → Returns top-K (e.g., top 10) matching chunks

BigQuery returns the top-K (e.g., top 10) most relevant results.

```

SELECT base.doc_id,
ARRAY_AGG(distance order by distance ASC) as distances ,
ARRAY_AGG(base.text order by distance ASC) as text,
count(base.doc_id) as numberOfChucks,
ARRAY_AGG( distinct base.created_at)[offset(0)] as created_at,
ARRAY_AGG(base.content_type)[offset(0)] as contenttype
from
VECTOR_SEARCH( (select * from `videoanalyzer-455321.text_embeddings_dataset.chunks`  

where created_at in (select ARRAY_AGG( created_at  

order by created_at DESC LIMIT 1 )[offset(0)]  

from `videoanalyzer-455321.text_embeddings_dataset.chunks`  

where userid=@userId group by doc_id))  

,"embedding",(SELECT @queryVector AS query_embedding) , top_k =>@topK,  

distance_type =>'COSINE' ) group by base.doc_id order by min(distance) ASC

```

6. Response Delivery :

Cloud Run sends the ranked results with text snippets and metadata back to the web UI for preview.

Menu

- Upload Documents
- View Documents
- Retrieve Documents**
- System Design Document
- Contact & Support

Welcome to Private Document Assistant



ANIL LALAM
lalamanilbabu@gmail.com

Show me the documents related to AI Projects
Preview Summary

Matched Documents

| | |
|--|--|
| | AI-Powered Personal Document Assistance System.pdf |
| | HackathonSubmission.pdf |
| | AIPoweredVideoSummarization.pdf |
| | Document Privacy Enhancement.pdf |
| | PrivateDocumentSearchAssistant.pdf |
| | DocumentAssistantDemoScript.pdf |
| | Document Assistance.jpg |

Found 7 matching documents.

AI-Powered Personal Document Assistance System.pdf: This document directly describes two specific AI projects: 'AI-Powered Personal Document Assistance System' and 'AI-Powered Accident Detection System', detailing their purpose and AI technologies used.

HackathonSubmission.pdf: This document describes the 'AI-Powered Personal Document Assistance System' project, detailing its inspiration, functionality, how it was built using Google Cloud AI, and future enhancements involving Generative AI.

Ask about your documents....

Send

The diagram illustrates the system architecture for document processing and retrieval. It shows the flow from user upload to document preprocessing, vector embedding, and finally, query search and retrieval of relevant documents.

Preprocessing the Document (Document AI + Vertex AI → Vectors)

- User uploads a PDF image or image file to the interface.
- Document AI extracts text from the image.
- Each chunk is converted into overlapping vectors using Vertex AI's "text-multilingual-embedding-002" model.
- These embeddings, along with metadata, are stored in BigQuery.
- Google Cloud Run triggers the preprocessing process.
- Vertex AI performs vector embeddings on the extracted text.
- Google Cloud Run triggers the retrieval process.
- BigQuery provides the embeddings and metadata to Vertex AI.
- Vertex AI performs cosine similarity search against the stored embeddings.
- Google Cloud Run triggers the retrieval process.
- BigQuery provides the ranked results with document previews and links to the original files.

Retrieving Relevant Documents (User Query → Vector Search → Results)

- User submits a search query.
- Google Cloud Run triggers the retrieval process.
- BigQuery provides the ranked results with document previews and links to the original files.

Menu

- Upload Documents
- View Documents
- Retrieve Documents**
- System Design Document
- Contact & Support

Welcome to Private Document Assistant



ANIL LALAM
lalamanilbabu@gmail.com

Show me the documents related to AI Projects
Preview Summary

Matched Documents

| | |
|--|--|
| | AI-Powered Personal Document Assistance System.pdf |
| | HackathonSubmission.pdf |
| | AIPoweredVideoSummarization.pdf |
| | Document Privacy Enhancement.pdf |
| | PrivateDocumentSearchAssistant.pdf |
| | DocumentAssistantDemoScript.pdf |
| | Document Assistance.jpg |

Found 7 matching documents.

AI-Powered Personal Document Assistance System.pdf: This document directly describes two specific AI projects: 'AI-Powered Personal Document Assistance System' and 'AI-Powered Accident Detection System', detailing their purpose and AI technologies used.

HackathonSubmission.pdf: This document describes the 'AI-Powered Personal Document Assistance System' project, detailing its inspiration, functionality, how it was built using Google Cloud AI, and future enhancements involving Generative AI.

Ask about your documents....

Send

The AI-Powered Personal Document Assistance System enables users to securely upload private documents (PDFs and images) and retrieve specific content using natural language queries.

Built on Google Cloud Platform (GCP), the system leverages Google Cloud Storage (GCS) for document storage, Eventarc to trigger Processing, and Cloud Run as the core microservice orchestrator.

Key processes include:

- 1. Document Ingestion and Preprocessing:** Upon uploading a document to GCS, Eventarc notifies a Cloud Run service. This service uses Document AI to extract text, which is then segmented into overlapping chunks. Each chunk is converted into a vector embedding using Vertex AI's "text-multilingual-embedding-002" model. These embeddings, along with metadata, are stored in BigQuery.
- 2. Query and Retrieval:** When a user submits a natural language query, Cloud Run generates a corresponding vector embedding using Vertex AI. A vector similarity search (cosine similarity) is performed in BigQuery against the stored document embeddings to identify and retrieve the most relevant text chunks.

The system then displays ranked results with document previews and links to the original files, ensuring data privacy, scalability, and multilingual adaptability.