GitHub Repository: https://github.com/abhinavbatra06/Gmail-Assistant-

Gmail Assistant RAG System - Step-by-Step Tutorial

Table of Contents

Prerequisites

Before starting, ensure you have:

- **Python 3.10 or higher** installed on your system
- **Git** installed (for cloning the repository)
- A **Gmail account** with emails you want to query
- An **OpenAI API key** (for embeddings and LLM responses)
- A **Google Cloud Project** with Gmail API enabled (for OAuth)

System Requirements

- **Operating System**: macOS, Linux, or Windows
- **RAM**: Minimum 4GB (8GB+ recommended for large email datasets)
- **Disk Space**: At least 2GB free space for data storage
- **Internet Connection**: Required for API calls (OpenAI, Gmail)

Installation Instructions

Step 1: Clone the Repository

```
# Navigate to your desired directory
cd ~/Desktop  # or your preferred location

# Clone the repository
git clone <your-repository-url>
cd Gmail-Assistant-
```

Step 2: Create Virtual Environment

For macOS/Linux:

```
# Create virtual environment
python3 -m venv venv

# Activate virtual environment
source venv/bin/activate
```

For Windows:

```
# Create virtual environment
python -m venv venv

# Activate virtual environment
venv\Scripts\activate
```

You should see `(venv)` in your terminal prompt when activated.

Step 3: Install Dependencies

```
# Make sure virtual environment is activated
pip install --upgrade pip

# Install all required packages
pip install -r requirements.txt
```

Note: Installation may take 5-10 minutes depending on your internet connection, as it downloads large packages like `docling`, `chromadb`, and `sentence-transformers`.

Step 4: Verify Installation

```
# Check Python version (should be 3.10+)
python --version

# Verify key packages are installed
python -c "import streamlit; import chromadb; import openai; print('All
packages installed successfully!')"
```

Environment Setup

Step 1: Set Up OpenAI API Key

1. **Get your OpenAI API key**:
   - Go to https://platform.openai.com/api-keys
   - Sign in or create an account
   - Click "Create new secret key"
   - Copy the key (you won't be able to see it again)

2. **Create a `.env` file** in the project root:

```
# Create .env file
touch .env
```

3. **Add your API key to `.env`**:

```
# Open .env in your text editor and add:
OPENAI_API_KEY=your_api_key_here
```

   Important: Replace `your_api_key_here` with your actual API key. Never commit this file to version control!

Step 2: Verify Environment Variables

```
# Test that the API key is loaded (with venv activated)
python -c "import os; from dotenv import load_dotenv; load_dotenv();
print('API Key loaded:', 'OPENAI_API_KEY' in os.environ)"
```

Gmail OAuth Configuration

This is a **critical step** - without proper Gmail OAuth setup, the system cannot fetch emails.

Step 1: Create Google Cloud Project

1. **Go to Google Cloud Console**:
   - Visit https://console.cloud.google.com/
   - Sign in with your Google account

2. **Create a new project**:
   - Click the project dropdown at the top
   - Click "New Project"
   - Enter project name (e.g., "Gmail Assistant")
   - Click "Create"

3. **Enable Gmail API**:

- In the project dashboard, go to "APIs & Services" > "Library"
- Search for "Gmail API"
- Click on "Gmail API"
- Click "Enable"

Step 2: Create OAuth 2.0 Credentials

1. **Configure OAuth Consent Screen**:
   - Go to "APIs & Services" > "OAuth consent screen"
   - Select "External" user type (unless you have a Google Workspace)
   - Click "Create"
   - Fill in required fields:
     - App name: "Gmail Assistant" (or your choice)
     - User support email: Your email
     - Developer contact: Your email
   - Click "Save and Continue"
   - On "Scopes" page, click "Save and Continue" (no scopes needed here)
   - On "Test users" page, add your Gmail address as a test user
   - Click "Save and Continue"
2. **Create OAuth Client ID**:
   - Go to "APIs & Services" > "Credentials"
   - Click "Create Credentials" > "OAuth client ID"
   - Application type: Select "Desktop app"
   - Name: "Gmail Assistant Client" (or your choice)
   - Click "Create"
   - **Download the JSON file** - this is your `gmail_creds.json`

Step 3: Set Up Credentials Folder

1. **Create the `creds` directory** in the project root:

```
# From project root directory
mkdir creds
```

2. **Move the downloaded JSON file**:
   - Rename the downloaded file to `gmail_creds.json`
   - Move it to the `creds/` folder:

```
# Example (adjust path to your download location)
mv ~/Downloads/client_secret_*.json creds/gmail_creds.json
```

3. **Verify the file structure**:

```
# Check that the file exists
ls -la creds/gmail_creds.json

# The file should contain JSON with keys like:
# "installed": { "client_id": "...", "client_secret": "...", etc. }
```

Important: The `creds/` folder and `gmail_creds.json` file are **required** before running the Gmail ingestion. The system will automatically generate `token.json` in this folder after the first OAuth flow.

Configuration Setup

Step 1: Configure `config.yaml`

Open `config.yaml` in a text editor and customize the settings:

1. **Gmail Settings**:

```
gmail:
  senders:
    - "sender1@example.com"
    - "sender2@example.com"
    # Or use "all" to fetch from all senders:
    # - "all"

  start_date: "2025-09-03"  # YYYY-MM-DD format
  end_date: "2025-12-03"
  label: "INBOX"  # Options: INBOX, UNREAD, STARRED, etc.
  target_total: 1000  # Maximum emails to fetch
  per_sender_cap: 100  # Max emails per sender
  include_attachments: true
```

2. **Verify Paths** (usually no changes needed):

```
paths:
  base_dir: data
  db_path: db/emails.db
```

3. **Check OpenAI Settings**:

```
embedding:
  model: "text-embedding-3-small"
  openai_api_key_env: "OPENAI_API_KEY"  # Should match .env variable
name
```

Step 2: Verify Configuration

```
# Test that config loads correctly
python -c "import yaml; config = yaml.safe_load(open('config.yaml')); print('Config loaded successfully!')"
```

Running the Pipeline

The system processes emails in stages. Run these steps **in order**:

Step 1: Ingest Emails from Gmail

```
# Make sure virtual environment is activated
source venv/bin/activate  # macOS/Linux
# OR
venv\Scripts\activate  # Windows

# Run Gmail ingestion
python -m src.gmail_ingest
```

What happens:

- First run: Browser opens for OAuth authentication
  - Sign in with your Gmail account
  - Click "Allow" to grant permissions
  - `token.json` is automatically created in `creds/`
- Emails are downloaded to `data/raw_emails/`
- Metadata is saved to `data/metadata/`
- Attachments are saved to `data/attachments/`
- Database is updated in `db/emails.db`

Expected output:

```
Starting GmailIngestor...
✅ Processed: <message_id> | <subject>
...
Done. Total emails saved: <count>
```

Step 2: Process Documents with Docling

```
# Process emails and attachments
python -m src.docling_processor
```

What happens:

- Email bodies are converted to structured JSON
- Attachments (PDFs, Word docs, images, etc.) are processed
- Calendar files (.ics) are parsed
- Processed documents saved to `data/docling/`

Expected output:

```
✅ Processed: <message_id> | <subject>
📎 Processed attachment: <filename> (<chars> chars)
```

Step 3: Chunk Emails

```
# Chunk all processed emails
python scripts/chunk_all.py
```

What happens:

- Emails are split into smaller chunks for retrieval
- Calendar events are extracted to `db/events.db`
- Chunks saved to `data/chunks/`

Expected output:

```
CHUNKING ALL EMAILS
============================================================
Total emails: <count>
Already chunked: <count>
Need chunking: <count>
...
Total chunks: <count>
```

Step 4: Generate Embeddings and Index

```
# Generate embeddings and add to vector database
python scripts/embed_and_index.py
```

What happens:

- Embeddings generated using OpenAI API
- Vectors stored in ChromaDB at `data/vector_index/`
- Database tracking updated

Expected output:

```
EMBED AND INDEX PIPELINE
============================================================
Generating embeddings for <count> emails...
✅ Added <count> chunks to vector DB
...
SUCCESS: Embeddings and ChromaDB are in sync!
```

Step 5: Verify Pipeline Completion

```
# Check database statistics
python -c "
from src.db_helper import DBHelper
db = DBHelper('db/emails.db')
stats = db.get_chunking_stats()
print(f'Emails: {stats[\"total_emails\"]}')
print(f'Chunked: {stats[\"chunked_emails\"]}')
print(f'Chunks: {stats[\"total_chunks\"]}')
db.close()
"
```

Using the Streamlit Interface

Step 1: Launch the Application

```
# Make sure virtual environment is activated
source venv/bin/activate  # macOS/Linux
# OR
venv\Scripts\activate  # Windows

# Launch Streamlit
streamlit run app.py
```

What happens:

- Streamlit server starts
- Browser automatically opens to `http://localhost:8501`
- If browser doesn't open, manually navigate to the URL shown in terminal

Step 2: Using the Chat Interface

1. **Enter a query** in the chat input at the bottom
2. **Wait for response** - the system will:
    - Route your query (calendar vs. general)

    - Retrieve relevant email chunks
    - Generate an answer using GPT-4o-mini
3. **View details** - click "Details" expander to see:
    - Detected intent
    - Number of chunks retrieved
    - Response latency
    - Source emails

Step 3: Example Queries

Try these queries to test the system:

- **Calendar queries**: "What events are happening this week?"
- **Deadline queries**: "What assignments are due soon?"
- **Sender queries**: "Show me emails from advisor@nyu.edu"
- **Information queries**: "What did the professor say about the final exam?"
- **General queries**: "Tell me about research opportunities"

Running Experiments

This section describes how to run the evaluation framework to assess system performance across different query types and scenarios.

Prerequisites

Before running experiments, ensure:

- The full pipeline has been completed (ingest → process → chunk → embed)
- The system is ready to answer queries
- You have sufficient API quota for OpenAI (experiments will make many API calls)

Step 1: Run Test Queries

The evaluation framework includes a batch query runner that executes test queries organized by category:

```
# Make sure virtual environment is activated
source venv/bin/activate  # macOS/Linux
# OR
venv\Scripts\activate  # Windows

# Run all test queries
python Eval/run_test_queries.py
```

What happens:

- The script runs queries from 5 categories:
    - **Vague/Ambiguous**: Queries with unclear intent or missing context
    - **Typos/Casual**: Queries with spelling errors or informal language

    - **Multi-intent/Complex**: Queries requiring multiple pieces of information
    - **Implicit/Contextual**: Queries requiring inference or background knowledge
    - **Negation/Edge cases**: Queries with negation or unusual patterns
- Each query is executed through the RAG pipeline
- Results are logged to the Memory module's database (`db/memory.db`)
- Progress and results are displayed in the terminal

Expected output:

```
============================================================
BATCH TEST QUERY RUNNER
============================================================

============================================================
CATEGORY: Vague/Ambiguous
============================================================

[1/70] Query: Tell me about the party
------------------------------------------------------------------------------
------

Answer: [System response...]

Time: 2.34s | Chunks: 5 | Intent: general
------------------------------------------------------------------------------
------
...
```

Note: The total number of queries may vary. The script will show progress for each query and category.

Step 2: Export Results to CSV

After running all test queries, export the results for analysis:

```
# Export evaluation logs to CSV
python Eval/export_eval_logs_csv.py
```

What happens:

- Queries and responses are extracted from `db/memory.db`
- Results are formatted and exported to `Eval/eval_logs.csv`
- The CSV includes:
    - Query text and detected intent
    - Generated answer
    - Routing module used (Predict vs. retriever)
    - Retrieval method
    - Number of chunks retrieved

- Response latency
- Top chunk IDs and similarity scores
- Source email metadata

Expected output:

```
✅ Exported 70 queries to Eval/eval_logs.csv
📄 File size: 245.3 KB
```

Step 3: Analyze Results

The exported CSV can be analyzed using spreadsheet software or Python:

1. **Open** `Eval/eval_logs.csv` in Excel, Google Sheets, or a data analysis tool
2. **Manual Accuracy Scoring**:
   - Review each query-answer pair
   - Score accuracy as 0 (incorrect) or 1 (correct)
   - Add accuracy column to the spreadsheet
3. **Create Analysis Spreadsheet**:
   - Use `eval logs (Concluded).xlsx` as a template
   - Create pivot tables by:
     - Query category
     - Intent type
     - Routing module
     - Accuracy scores
4. **Calculate Metrics**:
   - Overall accuracy rate
   - Accuracy by category
   - Average latency by query type
   - Chunk retrieval statistics

Step 4: Interpret Results

Key metrics to analyze:

- **Accuracy by Category**: Which query types perform best/worst?
- **Intent Classification**: How accurately are queries routed?
- **Retrieval Quality**: Are relevant chunks being retrieved?
- **Response Latency**: How long do queries take to process?
- **Module Performance**: When does Predict module outperform RAG retrieval?

Example Analysis Workflow

```
# 1. Run experiments
python Eval/run_test_queries.py

# 2. Export results
```

```
python Eval/export_eval_logs_csv.py

# 3. Open CSV for analysis
# - Open Eval/eval_logs.csv in Excel
# - Add accuracy column (0/1)
# - Create pivot tables
# - Compare with eval logs (Concluded).xlsx template
```

Troubleshooting Experiments

Issue: Queries fail with API errors

- **Solution**: Check OpenAI API quota and rate limits. The script makes many API calls.

Issue: Results CSV is empty

- **Solution**: Ensure `run_test_queries.py` completed successfully and check `db/memory.db` exists.

Issue: Memory database not found

- **Solution**: Ensure Memory module is enabled in `config.yaml` and the database was created during query execution.

For detailed experimental results and analysis, see **EXPERIMENTS.md**.

Usage Examples

Example 1: Complete Fresh Setup

```
# 1. Setup
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

# 2. Configure
# - Add OPENAI_API_KEY to .env
# - Add gmail_creds.json to creds/
# - Update config.yaml with your email settings

# 3. Run pipeline
python -m src.gmail_ingest
python -m src.docling_processor
python scripts/chunk_all.py
python scripts/embed_and_index.py

# 4. Launch UI
streamlit run app.py
```

Example 2: Adding New Emails

```
# 1. Update config.yaml with new date range or senders
# 2. Re-run ingestion (only new emails will be fetched)
python -m src.gmail_ingest

# 3. Process new emails
python -m src.docling_processor

# 4. Chunk new emails
python scripts/chunk_all.py

# 5. Embed and index new chunks
python scripts/embed_and_index.py
```

Example 3: Command-Line Query (Without UI)

```
# Query directly from command line
python -m src.rag_query --query "What events are happening this week?"
```

Troubleshooting Guide

Issue 1: "ModuleNotFoundError" or Import Errors

Symptoms: `ModuleNotFoundError: No module named 'src'` or similar

Solutions:

```
# 1. Ensure you're in the project root directory
pwd  # Should show path ending in Gmail-Assistant-

# 2. Verify virtual environment is activated
which python  # Should show path to venv/bin/python

# 3. Reinstall dependencies
pip install -r requirements.txt
```

Issue 2: Gmail OAuth Fails

Symptoms:

- "FileNotFoundError: creds/gmail_creds.json"
- "Invalid credentials" error
- OAuth flow doesn't start

Solutions:

```
# 1. Verify creds folder exists
ls -la creds/

# 2. Verify gmail_creds.json exists and is valid JSON
cat creds/gmail_creds.json | python -m json.tool

# 3. Check file permissions
chmod 600 creds/gmail_creds.json  # macOS/Linux

# 4. Delete token.json and re-authenticate
rm creds/token.json
python -m src.gmail_ingest  # Will trigger new OAuth flow
```

Issue 3: OpenAI API Errors

Symptoms:

- "Invalid API key"
- "Rate limit exceeded"
- "Insufficient quota"

Solutions:

```
# 1. Verify API key is set
python -c "import os; from dotenv import load_dotenv; load_dotenv();
print(os.getenv('OPENAI_API_KEY')[:10] + '...')"

# 2. Check .env file exists and has correct format
cat .env  # Should show: OPENAI_API_KEY=sk-...

# 3. Verify API key is valid
# Visit https://platform.openai.com/api-keys to check key status

# 4. Check billing/quota
# Visit https://platform.openai.com/account/billing
```

Issue 4: Database Locked Errors

Symptoms: `sqlite3.OperationalError: database is locked`

Solutions:

```
# 1. Close any other processes using the database
# Check for running Python processes
ps aux | grep python
```

```
# 2. If using Streamlit, restart it
# Stop current instance (Ctrl+C) and restart

# 3. Check database file permissions
ls -la db/emails.db
chmod 644 db/emails.db  # macOS/Linux
```

Issue 5: ChromaDB Errors

Symptoms:

- "Collection not found"
- "Permission denied" errors
- Vector count mismatch

Solutions:

```
# 1. Check vector_index directory exists
ls -la data/vector_index/

# 2. Reset ChromaDB if corrupted
rm -rf data/vector_index/*
python scripts/embed_and_index.py  # Rebuild index

# 3. Check disk space
df -h  # macOS/Linux
```

Issue 6: Docling Processing Fails

Symptoms:

- "ImportError: cannot import name 'DocumentConverter'"
- Processing hangs or crashes
- SSL certificate errors

Solutions:

```
# 1. Reinstall docling
pip uninstall docling
pip install docling

# 2. For SSL errors (Windows), ensure certifi is installed
pip install --upgrade certifi

# 3. Check HuggingFace access
# Docling downloads models from HuggingFace - ensure internet connection
```

Issue 7: Streamlit Won't Start

Symptoms:

- "Port 8501 is already in use"
- Browser doesn't open
- Connection refused

Solutions:

```
# 1. Find and kill process using port 8501
lsof -ti:8501 | xargs kill -9  # macOS/Linux
# OR on Windows: netstat -ano | findstr :8501, then taskkill /PID <pid>

# 2. Use different port
streamlit run app.py --server.port 8502

# 3. Check firewall settings
# Ensure localhost connections are allowed
```

Issue 8: No Results from Queries

Symptoms:

- Queries return "No relevant emails found"
- Empty results despite having emails

Solutions:

```
# 1. Verify emails were ingested
python -c "from src.db_helper import DBHelper; db =
DBHelper('db/emails.db'); print(f'Emails: {db.conn.execute(\"SELECT
COUNT(*) FROM emails\").fetchone()[0]}')"

# 2. Verify chunks exist
ls -la data/chunks/ | wc -l  # Should show chunk files

# 3. Verify embeddings exist
python -c "from src.vector_db import EmailVectorDB; vdb = EmailVectorDB();
print(f'Vectors: {vdb.count()}')"

# 4. Re-run embedding pipeline if needed
python scripts/embed_and_index.py
```

Issue 9: Memory/Performance Issues

Symptoms:

- System slows down
- Out of memory errors
- Processing takes very long

Solutions:

```
# 1. Reduce batch sizes in config.yaml
# embedding:
#   batch_size: 50  # Reduce from 100

# 2. Process fewer emails at once
# In config.yaml:
# gmail:
#   target_total: 500  # Reduce from 1000

# 3. Clear old data
rm -rf data/docling/*.json  # Re-process if needed
```

Issue 10: Configuration Errors

Symptoms:

- "KeyError" in config
- Invalid date format errors
- Path not found errors

Solutions:

```
# 1. Validate config.yaml syntax
python -c "import yaml; yaml.safe_load(open('config.yaml'))"

# 2. Check date formats (must be YYYY-MM-DD)
# start_date: "2025-09-03"  # Correct
# start_date: "09/03/2025"  # Wrong

# 3. Verify all paths exist or will be created
# The system creates directories automatically, but check permissions
```

Additional Resources

Useful Commands

```
# Check pipeline status
python -c "
from src.db_helper import DBHelper
db = DBHelper('db/emails.db')
print('Emails:', db.conn.execute('SELECT COUNT(*) FROM emails').fetchone()
[0])
print('Chunked:', db.get_chunking_stats()['chunked_emails'])
print('Embedded:', db.get_embedding_stats()['embedded_emails'])
db.close()
```

```
"

# View database contents
sqlite3 db/emails.db "SELECT id, subject, sender FROM emails LIMIT 5;"

# Check vector database
python -c "from src.vector_db import EmailVectorDB; vdb = EmailVectorDB();
print(vdb.get_stats())"
```

Getting Help

If you encounter issues not covered here:

1. **Check logs**: Look in `logs/` directory for error messages
2. **Verify versions**: Ensure all packages are up to date
3. **Review config**: Double-check `config.yaml` syntax
4. **Test components**: Run each pipeline step individually to isolate issues

---

Next Steps

After successful setup:

1. **Customize queries**: Experiment with different query types
2. **Adjust retrieval**: Modify `top_k`, `hybrid_alpha` in `config.yaml`
3. **Add preferences**: Use Memory module to set preferred senders
4. **Evaluate performance**: Check query logs and refine system

---

Congratulations! You've successfully set up the Gmail Assistant RAG system. Happy querying!