FAQ building kit instruction manual using Pinecone + Rag + OpenAI

What is Pinecone?

Pinecone is a cloud-based vector database that allows users to store and search documents and FAQs as numerical vectors. In this system, OpenAI's embedding model is used to vectorize user questions, and the vectors are searched in Pinecone to generate accurate answers based on the most similar information.

A vector database stores features (embedding vectors) extracted from text, images, audio, etc., and enables efficient similarity searches based on semantic distance. While traditional RDBs are strong in keyword matching searches, a vector database is unique in that it can quickly calculate semantic distance.

What is

Rag? RAG (Retrieval-Augmented Generation) is a system that pre-vectorizes the data that will be used for searches, searches for similar information to the user's question, and then generates an answer, making it ideal for FAQ responses like this system.

1. Main specifications of Rag-related Pinecone and OpenAI

Here we will only list the specifications that you should be aware of and omit the details.

1-1. Index

It is the basic unit of vector search, and all vectors stored in the index have the same number of dimensions.

1-2. Logical division

within the Namespace Index

Data can be separated by customer or purpose.

The same ID will be treated as separate data if the Namespace is different.

Metric

Define the calculation method for

similarity cosine:

Emphasizes direction

dotproduct: Uses the dot product euclidean: Euclidean distance

1-4. Dimensions

The number of features in the vector

This is fixed when creating an

index. All registered vectors must match this number of dimensions.

1-5. Operation Functions and Components of Endpoint URL

ÿ Pinecone endpoint configuration

https://<Index>-<Project ID>.<Service Type>.<Region Script>.pinecone.io

Example: https://urata-soft-abc12de.svc.us-east-1.pinecone.io

ÿ Operation functions and URL structure

When searching for similarity in vector metadata, specify as follows:

Example: https://urata-soft-abc12de.svc.us-east-1.pinecone.io/query

List of operation functions

Operation	URL
functions Vector	/vectors/upsert
registration/	/query
update Similarity	/vectors/delete
search Delete vector ID Get vector	and metadata /vectors/fetch?id=<ID>
Vector metadata update /vectors/up	date

1-6. Metadata

Each vector can have attribute information attached (e.g. title, tags, date)

Can be used as a filter condition when searching

1-7. OpenAI's main embedding models

An embedding model is an AI model that converts the meaning of text into a numerical vector.

Model name		Token limit
text-embedding-3-small	Features The latest, lightweight, and highly accurate flagship model (After the end of 2023)	8191 tokens
text-embedding-3-large	Higher precision, higher cost, and mainly high load For analysis	8191 tokens
text-embedding-ada-002	Old flagship model. Fast and cheap. Accuracy is or inferior	8191 tokens

2. Environment construction

First, unzip the package and it will have the following structure, so please place it as appropriate.

ÿÿÿ Flask

ÿ ÿÿÿ .env ÿ ÿÿÿ API key, environment variable setting file

app.py ÿ ÿÿÿ config.py Flask itself, Pinecone search and OpenAI response processing

ÿ ÿÿÿ templates Reading API keys and environment variables

ÿ ÿÿÿ index.html

ÿÿÿ PDF

ÿ ÿÿÿ sample_specification.pdf Original FAQ document

ÿÿÿ README.md

ÿÿÿ config.env.template API key, environment variable settings (for POC verification)

ÿÿÿ docs

ÿ ÿÿÿ operating_instructions.pdf User operation manual

ÿÿÿ query_embeddings.py Searches Pinecone using query strings

ÿÿÿ requirements.txt Install Python libraries in bulk with configuration files

ÿÿÿ upload_embeddings.py: Process to vectorize documents and register them in Pinecone

2-1. Install Python

This system uses Python, and the reasons for this are as follows:

- **Pinecone provides an official Python SDK for vector DB operations.**

  All operations such as upsert, query, and fetch can be written succinctly.

- •Pipeline integration of the entire RAG configuration

RAG templates are written in Python and have a rich framework

By using LangChain etc., you can seamlessly build chunking ÿ embedding ÿ vector DB registration ÿ query response.

- •Generation and embedding processes can be performed consistently using the OpenAI library, and JSON manipulation of API responses is simple.

ÿ Install from the Microsoft Store (recommended)

```
start mswindowsstore://pdp/?productid=9PJPW5LDXLZ5 ÿRunning
```

this command will open "Python 3.10" in the Microsoft Store. Simply press the "Install" button to complete the installation.

ÿ Install via Chocolatey (for developers)

```
SetExecutionPolicy Bypass Scope Process Force; `iex
((NewObject System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1')) choco install python
version=3.10.9 y
```

ÿÿLinuxÿUbuntu/Debian

Install Python 3.10 / 3.11 / 3.12 (optional) sudo apt update

```
sudo apt install y
softwarepropertiescommon sudo addaptrepository
ppa:deadsnakes/ppa sudo apt update sudo apt
install y python3.12
python3.12venv python3.12dev
```

ÿ [Linux version] CentOS / RHEL 7, 8, 9 series

Python 3.6 to 3.9 (from standard repository or EPEL)

```
sudo yum update y
sudo yum install y epelrelease
sudo yum install y python3
```

ÿ Installation confirmation command

```
python version
```

Example: If it shows Python 3.10.9 then it's OK.

2-2. Procedure for obtaining an OpenAI API key

Go to https://platform.openai.com/signup and create an account or log in. After logging in, click the profile icon in the top right corner and select "View API keys."

Click the "Create new secret key" button to generate a key, then copy and save it.

2-3. How to obtain a Pinecone API key

Go to https://www.pinecone.io/start/ and create an account or log in. After logging in, go to the dashboard and go to the "API Keys" section.

Press the "Create API Key" button, enter a name, generate a key, copy it and keep it.

2-4.Set the following in the config.env.template or Flask/.env file.

```
OPENAI_API_KEY=<OpenAI API key obtained in 2-2>
```

```
PINECONE_API_KEY=<Pinecone API key obtained in 2-3>
PINECONE_URL=<Pinecone endpoint URL *See 1-5>
PINECONE_INDEX_NAME=<pinecone index name>
```

2-5. Create requirements.txt

This is a configuration file that installs all the Python libraries required to build this system. It is convenient because you can automatically install these libraries with the following command.

```
> pip install -r requirements.txt
```

The initial settings are as shown below, but please change them as necessary when upgrading the OS etc.

```
openai>=1.2.0
pinecone>=3.0.0
tiktoken>=0.5.1
PyMuPDF>=1.23.0
python-dotenv>=1.0.0
pdfplumber==0.10.2
python-docx==1.1.0
Flask>=2.0.0
```

2-7. Security Precautions

In this system, as a POC to first verify its operation, the API key etc. is read from config.txt, but this method has security concerns.

Since **config.txt is placed in the same directory as HTML and JS, it is easy to include it in the web public area, and you can access it from the network or source of the developer tools** by pressing **F12** in the browser.

The API key will be visible.

Therefore, in the next chapter, 4-2, we will explain how to set the API key in the .env file when publishing on the web.

3. Details of each script

The purpose of each included script is as follows. Note that required parameters are obtained from external files or passed as parameters whenever possible. Some scripts, such as the script in Chapter 5, have fixed settings in the script for clear purposes and require consistency. Please edit the script as appropriate when customizing.

File name	explanation
upload_embeddings.py	Reads knowledge files such as PDFs and text, vectorizes them using OpenAI's embedded model, and registers (upserts) them in Pinecone. This is executed when building an index for the first time or updating a document.
query_embeddings.py	It is used for other purposes besides web systems, and in this case it is provided as a POC to verify its operation. are The user's input text is vectorized using the OpenAI embedding model. Search and retrieve similar documents from Pinecone and perform search processing for the FAQ bot
config.py	Reads OpenAI API key, Pinecone API key, endpoint URL, etc. from the .env file where API key etc. are set, and references common settings across the entire app. Illuminate

app.py	The main Flask app. Server-side processing that defines the /query endpoint, calls query_embeddings.py and the OpenAI API, and returns a response

4. Verify actual behavior from use cases

Let's say your company manufactures and sells mobile batteries. You want to have an AI help bot on your web page that can automatically answer frequently asked questions (FAQs). To do this, you first train the bot using the included sample_specification.pdf and then ask actual questions from users.

Let's build something that answers the question, "What is the capacity of the battery?"

4-1. Load sample_specification.pdf into pinecone

The system vectorizes the user's question and matches it with registered text documents such as text and PDFs to extract the most relevant information. Vectors are generated from sample_specification.pdf and uploaded to pinecone.

* Please use English names for the file to be read. In the pinecone specifications, Vector IDs are specified as ASCII characters, and this system obtains Vector IDs from file names, so Japanese file names are not supported. This will result in an error.

Run upload_embeddings.py, specifying the folder as the first parameter and the namespace as the second parameter.

```
Example) > python upload_embeddings.py "<Enter the path to the folder you want to load here (e.g. ./PDF)>"
                                        "<Set your Pinecone namespace here>"
```

```
Example) [Start processing] PDF/sample_specification.pdf [Success] Upload: sample_specification.pdf-chunk-1 Example)
[Success] Upload: sample_specification.pdf-chunk-1
```

4-2. Let's actually ask a question

When you pass the question "What is the battery capacity?" as a parameter to query_embeddings.py, it extracts the information most relevant to the content of specification.txt registered in 4-1 from Pinecone, Pass that information to OpenAI and have it respond in natural language. You can run query_embeddings.py in one line, with the question as the first parameter and the namespace as the second parameter:

```
Example) > python -c "from query_embeddings import ask_direct_answer;
     print(ask_direct_answer('What's the battery capacity?', '"<Pinecone namespace here>"'))"
```

The response is successful if it returns the following: Example: Battery capacity is 10,000mAh.

4-3. Ask a question from a web page

Next, we'll use a practical web page to input and submit a question and receive a natural language response from the AI.

First, we'll set the API key and other information set in config.env.template in 2-4 in the .env file as a security measure. We also used the Python framework Flask to build the REST API that receives and processes questions.

Flask is a Python framework that allows you to easily define URL routing and API endpoints, has lightweight dependencies, and allows you to define an API with just a few lines of script.

ÿ Operation procedure

1. Open the terminal and run the following command in the directory where Flask/app.py is located: > python

```
app.py "<Set the Pinecone namespace here>"
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit *
 Restarting with stat *
 Debugger is active!
 * Debugger PIN: 963-682-455 127.0.0.1
- - [16/Oct/2025 05:45:16] "GET / HTTP/1.1" 200 -
```

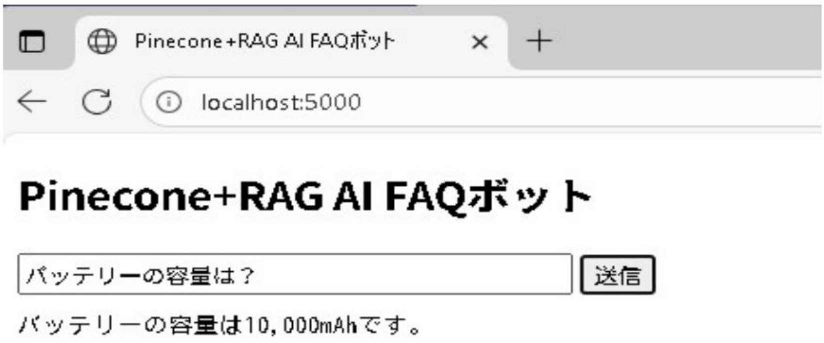
* If you get a WARNING like the one above, this is a warning that the Flask development server (flask run or app.run()) is not for production use and has limitations in security, stability, simultaneous connection handling, etc. However, a WSGI server (Web Server Gateway Interface server) is recommended for production environments, so this is not an issue here.

2. Access the following URL in your browser:

http://localhost:5000 3. Enter "What is the battery capacity?" in the question form on the page that appears and click submit. 4. OpenAI will respond to your search in natural language.

ÿ Execution results

If you reply as follows, it is successful.



Pinecone+RAG AI FAQボット

バッテリーの容量は？

バッテリーの容量は10,000mAhです。

5. Parameter adjustment for better answers

5-1. Adjusting the required parameters on Pinecone

Parameters	Purpose	Recommended value

index	Vector search target physical data Data store unit	
namespace (* Script execution parameters (Specify on the meter)	Logically group data in the same index Label to separate loops	
top_k	Number of similar documents obtained	3 to 5

5-2. Main parameter adjustments on the OpenAI side (all models are required)

Parameters	Purpose	Recommended value
model (Embedding Model) Convert	text to a numeric vector text-embedding-3-small	
model (Chat Model) Generates natural	language responses gpt-4o	
temperature	Randomness	0–0.5
max_tokens	Maximum number of output	200 to 500
System Prompt	tokens Role and constraint instructions	Set according to the purpose
response_format	for the model Output format control	JSON is recommended

6. Script Explanation

The source script for the script is below, which can be downloaded from Github below.

https://github.com/NextGenAI-corder/Pinecone_Rag_OpenAI/tree/main

The details are commented, but the important and customizable parts are highlighted in red.

6-1. upload_embeddings.py

```
import os
import requests
import pdfplumber
import docx
import argparse
from dotenv import load_dotenv

# --------------------------
# Load environment variables (securely obtain API key and connection URL from external file)
# --------------------------
load_dotenv("config.env.template")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
PINECONE_API_KEY = os.getenv("PINECONE_API_KEY")
PINECONE_URL = os.getenv("PINECONE_URL") # Do not include /query at the end

# --------------------------
# Text extraction process according to file type
# PDF ÿ pdfplumber
# Word ÿ python-docx
# Others ÿ Read as is in UTF-8 (e.g. .txt, .md, .py)
# --------------------------
def extract_text(file_path):
    ext = os.path.splitext(file_path)[1].lower()
    try: if ext == ".pdf":
            text = ""
            with pdfplumber.open(file_path) as pdf:
                for page in pdf.pages:
                    text += page.extract_text() + "\n"
            return text
        elif ext == ".docx":
            doc = docx.Document(file_path)
            return "\n".join([para.text for para in doc.paragraphs])
        else:
            with open(file_path, "r", encoding="utf-8", errors="ignore") as f:
                return f.read()
    except Exception as e:
        print(f"[Warning] Text extraction failure: {file_path} ÿ {e}")
        return ""


# ---------------------------
# Chunking process (splitting long sentences into specified
byte units) # chunk_size: number of characters in one chunk
(e.g. 1000 characters) # overlap: overlap between chunks (overlapped to maintain
context) # ÿ Important preprocessing for improving RAG accuracy
# ---------------------------
def chunk_text(text, chunk_size=1000, overlap=200):
    chunks = []
    start = 0
    while start < len(text):
        end = start + chunk_size
        chunks.append(text[start:end])
        start += chunk_size - overlap
    return [c.strip() for c in chunks if c.strip()]


# ---------------------------
# Embedding generation using OpenAI
API # Model: text-embedding-3-small (high-precision version for 2024
and beyond) # Vectorize each chunk and use it for semantic search with Pinecone
# ---------------------------
def get_embedding(text):
    headers = {
        "Authorization": f"Bearer {OPENAI_API_KEY}",
        "Content-Type": "application/json",

    } data = {"input": text, "model": "text-embedding-3-small"}
    response =
        requests.post( "https://api.openai.com/v1/embeddings", headers=headers, json=data

    ) response.raise_for_status()
    return response.json()["data"][0]["embedding"]

# --------------------------
# Upload process to Pinecone
# ID: Uniquely generated by file name + chunk number
# namespace: User-specified logical group (can be switched depending on the
purpose) # metadata: Stores the original text and file information returned when searching
# --------------------------
def upload_to_pinecone(vector_id, embedding, metadata, namespace):
    headers = {"Api-Key": PINECONE_API_KEY, "Content-Type": "application/json"} url
    = f"{PINECONE_URL}/vectors/upsert" data
    =
      { "vectors": [{"id": vector_id, "values": embedding, "metadata": metadata}],
        "namespace": namespace,

    } response = requests.post(url, headers=headers, json=data) if
    response.status_code != 200:
        print(f"[Error] Upload failed: {vector_id} ÿ {response.text}")
    else:
        print(f"[Success] Upload: {vector_id}")

# --------------------------
# Split the entire text of a single file into chunks ÿ Vectorize ÿ Register
on Pinecone # Upload and log each chunk
# --------------------------
def process_file(file_path, namespace):
    text = extract_text(file_path) if
    not text.strip():
        print(f"[Skip] Empty or not extractable: {file_path}")
        return
    chunks = chunk_text(text)
    for idx, chunk in enumerate(chunks):
        vector_id = f"{os.path.basename(file_path)}-chunk-{idx+1}" try:

        embedding = get_embedding(chunk)
            metadata = {"source": file_path, "text": chunk}
            upload_to_pinecone(vector_id, embedding, metadata, namespace)
        except Exception as e:
            print(f"[Error] Failed while processing {vector_id}: {e}")

# --------------------------
# Scan all files in the directory and process them
sequentially # Recursively including subdirectories
# --------------------------
def process_directory(directory_path, namespace):
    for root, _, files in os.walk(directory_path):
        for file in files:
            file_path = os.path.join(root, file)
            print(f"[Start processing]
            {file_path}") process_file(file_path, namespace)

# --------------------------
# Read and execute command line
arguments # directory: Document folder to process (e.g.
Flask/PDF) # namespace: Logical group name on Pinecone to save vectors to
# --------------------------
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("directory", help="Target directory (e.g., Flask/PDF)")

    parser.add_argument( "namespace", help="Pinecone namespace (e.g., our-project-specs)"

    ) args = parser.parse_args()

    # Check if folder exists if
    not os.path.exists(args.directory) or not os.path.isdir(args.directory):
        print(f"[Error] Directory does not exist: {args.directory}") exit(1)


    # Check if contents
    are empty if not any(os.scandir(args.directory)):
        print(f"[Error] Directory is empty: {args.directory}") exit(1)


    # Start batch
    processing process_directory(args.directory, args.namespace)
```

6-2. querry_embeddings.py

```
from dotenv import dotenv_values import
openai
from pinecone import Pinecone

# --------------------------
# Loading environment variables (obtained directly from
config.env.template) # * A structure that loads keys and settings from a file without relying on
system environment variables # ÿ High security and portability, suitable for switching between test and production environments
# --------------------------
config = dotenv_values("config.env.template")

# Set the OpenAI API key directly (not via an environment
variable) openai.api_key = config.get("OPENAI_API_KEY")

# Initialize Pinecone (obtain API key directly)
```

```
pc = Pinecone(api_key=config.get("PINECONE_API_KEY"))

# Pinecone index names are also obtained from
external settings # ÿ Flexible structure that can handle multiple projects and datasets
index = pc.Index(config.get("PINECONE_INDEX_NAME"))

# ---------------------------
# Similar document search (Pinecone + OpenAI
embedding) # Input: Question (natural language), namespace
(dataset identifier) # Output: List of searched meta information (text)
# ---------------------------
def get_similar_chunks(question, namespace):
    # Vectorize the question with OpenAI API (using an embedding
    model) embedding = (
        openai.embeddings.create(
            input=question,
            model="text-embedding-3-small", # The latest lightweight and accurate embedding model


        ) .data[0] .embedding
    )

    # Perform a similarity search from the
    Pinecone vector DB results
        =
        index.query( vector=embedding, top_k=5,
        # Get the top 5 similar documents include_metadata=True, # Return
        metadata including the original text namespace=namespace, # Identifier for logical separation by project or purpo
    )

    # Extract and return only the text body from the meta information
    return [match["metadata"]["text"] for match in results["matches"]]

# ---------------------------
# Response generation (OpenAI
Chat model) # Input: User question, namespace
(search target) # Output: Natural language response text (str) by gpt-4o
# ---------------------------
def ask_direct_answer(question, namespace): # Get
    similar documents and use them as the context for the answer
    chunks = get_similar_chunks(question, namespace) context
    = "\n".join(chunks)

    # Prompt design: Generate answers based on a FAQ
    document
        prompt = ( f"Answer the question based on the following information:\n\n{context}\n\nQ: {question}\nA:"
    )

    # Generate a response using gpt-4o (latest in
    2024) response = openai.chat.completions.create(
        model="gpt-4o", messages=[{"role": "user", "content": prompt}]
    )

    # Extract and return only the generated text from the response
    return response.choices[0].message.content.strip()
```

6-3. config.py

```
import os
from dotenv import load_dotenv

load_dotenv()

OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
PINECONE_API_KEY = os.getenv("PINECONE_API_KEY")
PINECONE_ENVIRONMENT = os.getenv("PINECONE_ENVIRONMENT")
PINECONE_INDEX_NAME = os.getenv("PINECONE_INDEX_NAME")
```

6-4. app.py

```
from flask import (
        Flask,
        render_template,
        request,
        jsonify,
) # Import Flask's main functions
import openai # Library for using the OpenAI API
from pinecone import Pinecone # Pinecone's Python SDK import
config # Custom module that stores API keys etc. import sys #
Standard library for handling command line arguments

# --------------------------
# Initialize your Flask application
# --------------------------
app = Flask(__name__)

# --------------------------
# Initial settings for OpenAI and Pinecone
# Uses the API key defined in config.py #
Pinecone host URL is specified manually (supports self-hosted endpoint)
# --------------------------
openai.api_key = config.OPENAI_API_KEY pc =
Pinecone(api_key=config.PINECONE_API_KEY) index =
pc.Index(
        name=config.PINECONE_INDEX_NAME,
        host="https://urata-soft-js34rwd.svc.aped-4627-b74a.pinecone.io",
)

# ---------------------------
# Get namespace from command line argument at startup # If not
specified, an error message will be displayed and the
application will exit # * Configuration that uses a fixed namespace for the entire Flask app
# ---------------------------
if len(sys.argv) < 2:
        print("Usage: python app.py Please specify the parameter <namespace>!") exit(1)


NAMESPACE = sys.argv[1] # Set the namespace to be used throughout as a global variable


# ---------------------------
# Display index.html when the root endpoint ("/") is accessed # templates/
index.html is automatically loaded
# ---------------------------
@app.route("/")
def index_page():
        return render_template("index.html")


# ---------------------------
# API endpoint that processes POST request "/query" #
Based on the question sent by the user, perform vector search and generate response
# ---------------------------
@app.route("/query", methods=["POST"]) def
query():
        data = request.json # Get the JSON sent from the front end user_input
        = data.get("query") # Extract the user's question text

        # Perform embedding (vectorization) with OpenAI
        API embedding = (
                openai.embeddings.create(
                        model="text-embedding-3-small", # Lightweight and highly accurate embedding model
                        input=user_input,


                ) .data[0] .embedding
        )

        # Perform vector search on Pinecone (Top 5 results)
        result =
                index.query( vector=embedding,
            top_k=5,
            include_metadata=True, # Return meta information such as the original
            text namespace=NAMESPACE, # Use the namespace specified at startup (fixed)
        )

        # Get matches from search results (flexible support for Pinecone's return format)
        matches = result["matches"] if isinstance(result, dict) else result.matches

        if matches:
            # Concatenate text from multiple matches as a context
            context = "\n\n".join([m["metadata"]["text"] for m in matches])

            # Generate responses in natural language using the ChatGPT
            API (with constraints) completion =
                openai.chat.completions.create( model="gpt-4o",
                # Fast and
                    accurate model messages=[ {
                        "role": "system",
                        "content": "Please respond to user questions concisely in Japanese, within 50 characters.
                    },
                    {"role": "user", "content": f"Question: {user_input}\nInfo:\n{context}"},
                ],
            )

            # Extract and return the response
            answer_text = completion.choices[0].message.content.strip() return
            jsonify({"answer": answer_text})
        else:
            # Error message if no match found return
            jsonify({"answer": "No matching answers found"})


# ---------------------------
# Start a Flask application (debug mode enabled)
# ---------------------------
if __name__ == "__main__":
        app.run(debug=True)
```