ChatGPT

Practical Implementation Plan: Embedding Model Switch, Qdrant, FastAPI UI, and Testing

Python 3.13+ Environment

We will use **Python 3.13 or above** as the runtime, to leverage its latest optimizations and features. Python 3.13.0 (released October 2024) is the newest major release with **many new features and performance improvements** compared to 3.12 [1]. Notably, it introduced an *experimental no-GIL mode* and a preliminary JIT compiler for better concurrency and speed [2]. Using Python ≥3.13 ensures our application is future-proof and can benefit from these enhancements. All required libraries (FastAPI, Qdrant client, OpenAI SDK, HuggingFace Transformers/SentenceTransformers) are compatible with Python 3.13. Before development, we should confirm our environment is running 3.13+ (e.g. `python --version`) and upgrade if needed. This gives us a solid, up-to-date foundation for building the embedding-based system.

Switchable Embedding Model (OpenAI vs. HuggingFace)

A core requirement is the ability to **switch between OpenAI's embedding API and a local HuggingFace model** for generating text embeddings. We will implement a configuration setting (e.g. an environment variable or config file entry) that specifies which embedding backend to use.

- **OpenAI Embeddings (text-embedding-3-large)**: When configured to use OpenAI, we will call OpenAI's embedding API with the `text-embedding-3-large` model. This is OpenAI's latest and most capable text embedding model (v3) as of 2024, providing 3,072-dimensional embeddings and significantly improved semantic performance [3]. (For example, it improved multilingual embedding accuracy from 31.4% to 54.9% on MIRACL benchmark versus the older Ada-002 model [4].) Using the OpenAI Python SDK, we can obtain embeddings with a single API call, e.g. `openai.Embedding.create(model="text-embedding-3-large", input=text)`. The response includes the embedding vector (we'll extract the list of floats). We must ensure an OpenAI API key is available and the model name is valid. At startup, the application should **validate OpenAI access** – for instance, check that the API key is set and perhaps attempt a small test call or use OpenAI's list models API to verify `text-embedding-3-large` is accessible. If the user selects this model in config, our code routes all embedding requests to the OpenAI API.

- **Local HuggingFace Embeddings (SentenceTransformers)**: For an offline or cost-free option, we will support local embedding generation via Hugging Face models. We can use the **SentenceTransformers** library (built on HuggingFace Transformers) to load a pre-trained embedding model locally. A good choice for prototype is the `"sentence-transformers/all-MiniLM-L6-v2"` model, which is a small, fast, but high-quality encoder [5]. This model produces 384-dimensional sentence embeddings and is ~5× faster than its larger counterparts while sacrificing little accuracy [6]. To use it, we instantiate a `SentenceTransformer` in code (e.g. `model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')`) and call `model.encode(text)` to get an embedding vector [7]. If the config is set to a HuggingFace model, our code will load that model at startup (downloading weights if not cached) so that subsequent calls to embed text are fast. We should **validate the local model at startup** by attempting to load it and checking its embedding dimension. For example, we can verify that `model.get_sentence_embedding_dimension()` returns an expected size (384 for MiniLM, or 768/1536 for other models). If the model name is unsupported or the download fails, we should log an error and prevent the app from running, to avoid runtime failures. This approach ensures **switchability** – if the user chooses `text-embedding-3-large` in the config, we use OpenAI's API; if they choose a HuggingFace model name, we use the local model for embeddings. The embedding results (vectors) from either source will be used consistently in downstream steps.

- **Unified Interface & Switching Logic**: We will create an abstraction, e.g. an `EmbeddingService` class or function, that checks the configured `embedding_model` setting and routes to the appropriate backend. Pseudocode:

```
if config.EMBEDDING_MODEL == "text-embedding-3-large":
    # Use OpenAI API
    vec = openai.Embedding.create(model="text-embedding-3-large",
input=text)['data'][0]['embedding']
else:
    # Use local HuggingFace model
    vec = local_model.encode(text).tolist()
```

For prototype, we limit choices to exactly these two options (OpenAI v3-large or one local SentenceTransformer) as specified. This **conditional switch** makes it easy to test with the local model (no API cost) and then use the OpenAI model for better quality in a production setting. It's important to also handle any differences in vector dimensionality. Since `text-embedding-3-large` produces 3072-length vectors and our example local model produces 384-length vectors, we will need to configure the vector database (Qdrant) accordingly (more on this below). Essentially, when the app initializes, after loading the embedding model (or confirming API), it should set a variable for `vector_dim` (e.g. 3072 or 384) based on the chosen model, and use that when setting up Qdrant. By making the embedding backend easily switchable and validating it on startup, we ensure flexibility without runtime surprises [8].

Using Qdrant for Vector Storage and Search

For storing embeddings and performing similarity search, we will use **Qdrant**, an open-source vector database. Qdrant is a great choice because it's self-hostable (no external dependencies), supports high-performance vector similarity search, and even offers an in-memory mode for quick prototyping [9]. We will run Qdrant either as a Docker container or use the Python in-memory mode during development.

Setup and Initialization: We'll utilize the official `qdrant-client` Python SDK to interface with Qdrant. At application startup, after determining the embedding vector dimension (`D`), we will **create a Qdrant collection** (if not already existing) to store our vectors. For example, using the client:

```
from qdrant_client import QdrantClient, models
client = QdrantClient(":memory:")  # for dev, or QdrantClient(host="localhost",
port=6333) for server
client.recreate_collection(
    collection_name="embeddings",
    vectors_config=models.VectorParams(size=D, distance=models.Distance.COSINE)
)
```

This defines a collection named `"embeddings"` with vector size `D` and cosine similarity metric [10]. (Cosine distance is a common choice for text embeddings [11].) We use `recreate_collection` to drop any existing and start fresh each run (for deterministic prototyping), or `create_collection` if we want to preserve data. If the user switches embedding models (thus dimension changes), recreating ensures the collection schema matches the current model's vector size. In a more advanced setup, one could maintain separate collections per embedding model or re-index data when switching.

Data Ingestion (Upserting Vectors): With the collection ready, we will need to insert vectors for our data. Depending on the use-case, this could be done by embedding a corpus of documents or receiving user text to store. Assuming we have some text data to index, we convert each piece of text to its embedding (via the chosen model) and then **upsert** it into Qdrant. Using the client, we can batch inserts for efficiency. For example:

```
points = []
for idx, emb in enumerate(embeddings):
    points.append(models.PointStruct(
        id=idx,
        vector=emb,
        payload={"text": source_texts[idx]}  # store original text or metadata
    ))
# When batch is ready:
client.upload_points(collection_name="embeddings", points=points, wait=True)
```

The `PointStruct` holds an `id`, the embedding vector, and an optional payload (we include the original text or any metadata as payload) [12] [13]. Qdrant recommends uploading in batches (e.g. 100 or 1000 at a time) for large datasets [14]. The `upload_points` (or `upsert` method) will add these vectors to the collection. In our prototype, we might embed a small set of example texts for demonstration. After upserting, we can confirm the collection size via `client.count()` if needed [15]. By the end of this step, our Qdrant collection holds all vectors ready for similarity search.

Similarity Search Query: Qdrant allows efficient nearest-neighbor search to find vectors most similar to a query vector. For a given user query, we will get its embedding (again via the selected model) and then query Qdrant. We can use `client.search` to retrieve the top *k* most similar points. For instance:

```
query_vec = get_embedding(user_query)  # our embedding function
results = client.search(collection_name="embeddings", query_vector=query_vec,
limit=5)
for hit in results:
    print(hit.payload, "score:", hit.score)
```

This will return (up to) 5 records with highest cosine similarity to the query [16]. Each result `hit` contains the stored `payload` (so we can see the original text or metadata) and a similarity `score`. We will use these results to display back to the user (e.g. as search results or context for answering a question). Qdrant's search is very fast, even for large collections, and supports filtering by metadata as well if needed (for example, restricting by document type, date, etc., though our initial prototype may not need filters). The vector search happens in sub-millisecond time for thousands of vectors, enabling real-time responses.

One advantage of Qdrant is we can run it in-memory for tests and development [9], avoiding external database setup. When deploying, we can run Qdrant as a separate service (with persistence). It provides REST and gRPC APIs under the hood, but the Python client abstracts those details. The design is also scalable to multi-tenant use (we could have multiple collections or use metadata filters to isolate data per user or per bot, as noted in Qdrant's docs and our earlier research) [17]. Overall, using Qdrant will give us a robust vector store to manage our embeddings and perform similarity lookups quickly, which is essential for any embedding-based search or Q&A system.

Simple Web UI with FastAPI and Jinja2

For the frontend, we aim to build a **simple HTML interface** using FastAPI with server-side templates (Jinja2), rather than a complex React app. This keeps our prototype lightweight and quick to develop. We will use **FastAPI** to define web endpoints and **Jinja2 templates** for rendering HTML pages.

FastAPI Setup: We create a FastAPI app and mount Jinja2 templates. For example:

```
from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates
from fastapi.responses import HTMLResponse

app = FastAPI()
templates = Jinja2Templates(directory="templates")

@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    return templates.TemplateResponse("index.html", {"request": request})
```

FastAPI's `Jinja2Templates` makes it straightforward to render a template with a context [18]. In this snippet, a GET request to `/` returns the `index.html` template, passing along the `request` (required for Jinja2 templates to use `url_for`) and any context data (none in this basic example). We will create an `index.html` template that contains a simple **search form** – perhaps a text input for the user query and a submit button. The form will POST the query to a search route.

Template Design: Inside `templates/index.html`, we might include: an `<h1>` title, a `<form method="post" action="/search">` with a text `<input name="query">` and a submit button. If results are present (passed in context), we loop over them to display them (for example, showing the text snippet and score or any metadata). Using Jinja2, we can do something like:

```
{% if results %}
  <h2>Results for "{{ query }}"</h2>
  <ul>
    {% for item in results %}
      <li>{{ item.text }} (score: {{ item.score }})</li>
    {% endfor %}
  </ul>
{% endif %}
```

This will render a list of results if they exist. We keep the design minimal (no complex CSS or JS) to focus on functionality. FastAPI can also serve static files (like a CSS file) if we want to add basic styling, by using `app.mount("/static", StaticFiles(directory="static"), name="static")` and referring to them in the template [19], but we might skip heavy styling for now.

Search Endpoint: We define a POST route `/search` that handles the form submission. For example:

```
from pydantic import BaseModel

class QueryForm(BaseModel):
    query: str

@app.post("/search", response_class=HTMLResponse)
async def search(request: Request, form: QueryForm = Depends()):
    user_query = form.query
    # Get embedding and search Qdrant
    query_vec = get_embedding(user_query)
    results = client.search(collection_name="embeddings",
query_vector=query_vec, limit=5)
    # Prepare results data (e.g., list of dicts with text and score)
    hits = [{"text": hit.payload.get("text"), "score": f"{hit.score:.3f}"} for
hit in results]
    return templates.TemplateResponse("index.html", {
        "request": request, "query": user_query, "results": hits
    })
```

Here we use a Pydantic model `QueryForm` to parse the form data (FastAPI can also use `Request.form()` directly, but Pydantic makes it clean). We then embed the query, perform the Qdrant search, and build a list of result items. Finally, we render `index.html` again, but this time supply the original query and the results in the context. The template will show the results as described above. This provides a **round-trip refresh** approach — after submitting, the user sees a new page with the query and results. Since our focus is not on single-page interactivity, this simple post-redirect-get (or direct post to page) pattern is sufficient. It avoids needing any JavaScript.

By not using React, we significantly reduce complexity and setup. FastAPI with Jinja2 lets us achieve a functional UI quickly, which is ideal for a prototype or internal tool. If needed, we could enhance user experience with small JavaScript (e.g., to keep the query in the input after search or to highlight text), but those are optional improvements. The key point is that **FastAPI + Jinja2** allows us to serve HTML pages that include our dynamic data (search results) easily [18]. This meets the requirement of a “simple UI” without the overhead of a frontend framework.

Ensuring Test Coverage for Components

To keep the implementation robust and maintainable, we will write **unit tests and integration tests** for each major component. The goal is to have tests at each functional unit (embedding logic, database operations, and web endpoints) to catch issues early. We can use **pytest** as our testing framework and leverage FastAPI’s built-in **TestClient** for web endpoint tests.

1. Embedding Function Tests: We will test the embedding generation function for both modes. For the HuggingFace path, we can use a small sample input (e.g. `"Hello world"`) and ensure the output vector has the expected length (384 for MiniLM). We can also test that multiple calls are consistent (maybe embed the same text twice and check if vectors are nearly identical, accounting for any nondeterminism – though these models are deterministic). For the OpenAI path, since calling the real API in tests is not ideal (could be slow/costly), we can **mock** the OpenAI response. Using Python’s `unittest.mock`, we might simulate `openai.Embedding.create` to return a dummy vector (e.g., a list of zeros of length 3072). Then our function should return that vector. The test ensures that when config is set to "text-embedding-3-large", the code attempts to call OpenAI, and when set to local model, it uses the local model object. We also test the startup validation logic: e.g., if an invalid model name is given, our init function should raise an error or exit (this can be tested by simulating the condition and expecting an Exception).

2. Qdrant Integration Tests: Using Qdrant’s in-memory mode, we can write tests for storing and querying vectors. For example, start a QdrantClient with `":memory:"` and create a collection of a fixed dimension (say 3 for test). Insert a few known vectors (we can use small manually-crafted 3-d vectors for simplicity). Then query for a vector and verify the correct point is returned. For instance, insert points with vectors [1.0, 0.0, 0.0] and [0.0, 1.0, 0.0] and then query with [1.0, 0.0, 0.0] – expect the first point to have the highest score. We can also test that the `payload` comes back intact. These tests ensure our Qdrant calls (create, upsert, search) are working as expected. Since Qdrant in-memory is fast and ephemeral, the tests won’t require any external services. (If needed, we can also start a test Qdrant Docker container in CI, but that's likely overkill for prototype testing when in-memory mode suffices.) The GPTech blog example confirms that Qdrant’s Python client can be used in-memory for quick testing [9], which we'll utilize.

3. FastAPI Endpoint Tests: FastAPI provides a `TestClient` (powered by Starlette) that allows calling our API routes in tests as if they were HTTP requests. We will write tests to cover both the GET and POST behavior of our web app. For example, using the TestClient we can simulate a GET request to "/" and verify we got a 200 OK and the response contains the expected HTML (e.g., the form). More importantly, we test the `/search` POST. We can call `client.post("/search", data={"query": "some query"})` and then check that the response is 200 and contains the query string in the rendered HTML (since our template would echo it) and perhaps that it contains at least one result or the "no results" message. For these tests, we might need to monkeypatch the `get_embedding` or Qdrant search if we don't want to rely on the whole pipeline. One approach is to **dependency-inject** a fake search function during tests. Alternatively, we can set up a dummy small Qdrant collection in the app context just for testing: e.g., pre-load a known vector so that searching "hello" returns a predictable result. Given it's a prototype, a simpler way is to patch `client.search` to return a fixed value. Using `unittest.mock`, we can stub out `client.search` to return a known list of points (with fake payload and score). Then when the test posts "hello", the response HTML should contain that fake payload. This verifies that our `/search` route properly takes the form input, calls the search, and renders the template.

4. Continuous Testing During Development: We will run these tests frequently (e.g., via `pytest -vv`) as we develop each feature. This way, **each unit of functionality is validated** independently. For example, after implementing the embedding switch logic, run embedding tests; after Qdrant integration, run the DB tests; after the FastAPI routes, run the client tests. This practice will catch issues like mismatched vector dimensions or template context errors early. It also provides documentation of intended behavior (for future contributors).

By having a solid test suite, we ensure that the final system is reliable despite the rapid development approach. The tests act as a safety net when refactoring or switching out components (for instance, if we upgrade the embedding model, we can rerun tests to confirm nothing breaks). In summary, **each major component (embedding service, vector store, and web interface) will be covered by tests**, fulfilling the requirement for *function-level testing at each step*. This not only increases code quality but also confidence when making changes.

Conclusion

In this refined plan, we focused on practical and efficient implementation strategies for the desired system. We will use **Python 3.13+** to stay current and efficient, and implement a **flexible embedding module** that can switch between OpenAI's state-of-the-art `text-embedding-3-large` model and a local HuggingFace model for development flexibility. We chose **Qdrant** as the vector database for its ease of use and performance, allowing us to store embeddings and perform semantic searches quickly (with simple Python client calls to create collections, upsert data, and query by similarity) [10] [16]. For the user interface, we avoid unnecessary complexity by using **FastAPI with Jinja2 templates** to render a basic web page that lets users input queries and see results – this is straightforward to build and iterate on, without involving a front-end framework. Finally, we emphasize **testing throughout development** to ensure each piece works correctly in isolation and together. By following this plan, we should be able to implement a working prototype of an embedding-based search/Q&A system that is both **practical for quick development and grounded in solid engineering practices**. The result will be a simple yet effective application, where one can enter a query, have it embedded by either OpenAI or a local model, find similar content via Qdrant, and see the results on a web page – all built in a clean and maintainable way.

Sources:

- OpenAI *text-embedding-3-large* model – 3072-dimensional embeddings with improved multilingual and English performance [3].

- HuggingFace SentenceTransformers *all-MiniLM-L6-v2* – lightweight local embedding model (~384 dims) suitable for quick, local embedding generation [5].
- Qdrant vector DB usage – creating collections, upserting embeddings in batches, and querying by vector similarity via Python client [10] [20] [16].
- FastAPI with Jinja2 templates – rendering HTML responses using `Jinja2Templates` (server-side templating for simple UI) [18].
- Python 3.13 release – latest Python version with new features and optimizations (e.g. optional no-GIL mode for better concurrency) [2] [1].

---

[1] [2] Python Release Python 3.13.0 | Python.org
https://www.python.org/downloads/release/python-3130/

[3] [4] OpenAI's Text Embeddings v3 | Pinecone
https://www.pinecone.io/learn/openai-embeddings-v3/

[5] [6] [7] [8] [9] [10] [11] [14] [16] [20] From HuggingFace dataset to Qdrant vector database in 12 minutes flat
https://www.gptechblog.com/from-huggingface-dataset-to-qdrant-vector-database-in-12-minutes-flat/

[12] [13] [15] Code Search with Vector Embeddings and Qdrant - Hugging Face Open-Source AI Cookbook
https://huggingface.co/learn/cookbook/en/code_search

[17] multi-domain-ai-chatbot-design.md
file://file-XhJxaiobAff8wgosnHo8H2

[18] [19] Templates - FastAPI
https://fastapi.tiangolo.com/advanced/templates/