users get answers quickly

💡 GOAL

Ensure that **users get answers quickly** from their department-specific memory, even on **low-spec machines**.

📦 Folder Structure (for the Memory Service)

```
memory_engine/
├── embeddings/
│   ├── faiss_index.bin          # Vector index (saved with
Faiss)
│   └── id_to_text.json          # Mapping of IDs to original
memory text
├── cache/
│   └── memory_cache.pkl         # Stores recent Q&A for fast
reuse
├── data/
│   └── department_tags.json     # Memory tagged by department
├── memory_engine.py             # Main memory service
├── embedder.py                  # Generates embeddings from
text
├── cache_utils.py               # Cache logic
└── utils.py                     # Helper functions
```

🧠 How It Works (Explained)

1. User sends a query (e.g. "How do I apply for leave?")

- The system checks if it's **already in cache** → fast return.
- If not, it does:

2. Filter memory (e.g. only HR department)

- Loads relevant memory texts from `department_tags.json`.

3. Convert user query to an embedding

- Uses `embedder.py` to generate a vector using your on-device model (like `gemma.cpp` or sentence-transformers).

4. Search vector DB using Faiss

- Gets top 3–5 similar memory items fast.

5. Send that context to the local LLM

- Composes a system prompt + relevant memories + query.

6. Save result to cache

- Stores input & response pair in `memory_cache.pkl`.

🧠 Step-by-Step Code: Basic System

✅ `embedder.py` — Embed text using sentence-transformers

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("all-MiniLM-L6-v2")  # Light & fast

def embed_text(text):
    return model.encode([text])[0]
```

✅ `cache_utils.py` — Memory cache using `pickle`

```
import pickle
from collections import OrderedDict

CACHE_PATH = "cache/memory_cache.pkl"
MAX_CACHE = 100

def load_cache():
    try:
        with open(CACHE_PATH, "rb") as f:
            return pickle.load(f)
    except:
        return OrderedDict()

def save_cache(cache):
    with open(CACHE_PATH, "wb") as f:
        pickle.dump(cache, f)

def get_cached_response(query):
    cache = load_cache()
    return cache.get(query, None)
```

```
def add_to_cache(query, response):
    cache = load_cache()
    if query not in cache:
        if len(cache) >= MAX_CACHE:
            cache.popitem(last=False)
        cache[query] = response
    save_cache(cache)
```

✅ `memory_engine.py` — Full Memory Query Flow

```
import faiss
import json
import numpy as np
from embedder import embed_text
from cache_utils import get_cached_response, add_to_cache

# Load index
faiss_index = faiss.read_index("embeddings/faiss_index.bin")
with open("embeddings/id_to_text.json") as f:
    id_to_text = json.load(f)

def query_memory(query, department="HR"):
    # 1. Check cache
    cached = get_cached_response(query)
    if cached:
        return cached

    # 2. Load department-filtered memory
    with open("data/department_tags.json") as f:
        department_data = json.load(f)
    relevant_ids = department_data.get(department, [])

    # 3. Generate embedding
    q_embedding = embed_text(query).astype("float32")

    # 4. Faiss search (limit to relevant memory)
    index_ids = [int(i) for i in relevant_ids]
    vectors = [faiss_index.reconstruct(i) for i in index_ids]
    local_index = faiss.IndexFlatL2(len(q_embedding))
    local_index.add(np.array(vectors))

    _, result_indices =
local_index.search(np.array([q_embedding]), k=3)
```

```
    results = [id_to_text[str(index_ids[i])] for i in
result_indices[0]]

    # 5. Compose context + ask local LLM
    context = "\n\n".join(results)
    full_prompt = f"Based on the following information:
\n\n{context}\n\nAnswer this:\n{query}"
    response = local_llm(full_prompt)  # <- connect to
llama.cpp or gemma.cpp

    # 6. Cache result
    add_to_cache(query, response)

    return response
```

✅ Example Output

```
response = query_memory("How do I apply for leave?",
department="HR")
print(response)
```

⚡ Optimization Notes

- `faiss_index.reconstruct()` loads only the required vectors — helps on low RAM.
- LRU-style cache avoids infinite growth.
- Embedding model is **lightweight**, <100MB, and can run offline.
- Easily switch `SentenceTransformer` to `gemma.cpp` or other quantized models for full offline inference.

✅ Next Steps (Optional Additions)

Feature	Why Add It
SQLite or DuckDB as a backing store	If JSON gets too big
Token-based deduplication	Avoid similar queries clogging the cache
Preloading hot queries	Instant response for most-used FAQs
Batched embedding refresh	On sync/upload, all content pre-embedded

Would you like me to package this into a simple prototype repo or help you wire it into your current Trosyn AI backend (Node, Python, etc.)?