ChatGPT

Project Epoch Deep Research Responses

Task 1: Vector Model (English/Chinese/Multilingual) Selection

Summary

- **MiniLM (all-MiniLM-L6-v2)** – A lightweight 6-layer, 384-dim English-focused model (~22M params) that is **5× faster** than larger models while maintaining good quality [1]. It's Apache-2.0 licensed (model) but note that some training data (e.g. MS MARCO) has non-commercial terms [2]. MiniLM performs on par with larger models for short English texts [3] but is not optimized for Chinese.
- **BGE Small Zh v1.5** – A 24M param, 384-dim **Chinese** embedding model achieving competitive performance [4]. It outperforms generic models in Chinese retrieval (C-MTEB avg ~57.8 vs large models ~64) [5]. It's MIT licensed and **commercial-use friendly** (models are free for commercial use) [6]. CPU inference is fast and memory footprint ~100MB, similar to MiniLM.
- **BGE M3 (Multilingual)** – A state-of-the-art 569M param model (1024-dim) covering 100+ languages [7]. It excels in both English and other languages (topping benchmarks, surpassing OpenAI Ada) [8] with unified dense+sparse capabilities. However, it requires ~2.3GB VRAM and is slower on CPU. MIT licensed [9], suitable if hardware allows and maximum quality is needed.
- **Quality vs. Speed Trade-off** – BGE-M3 offers the best retrieval accuracy (Multi-lingual SOTA) but is heavy; BGE-Small is much faster and sufficient for many use cases, especially Chinese. MiniLM is fastest for English but lags in Chinese accuracy. All three are offline-capable and open-source; BGE models explicitly permit commercial use [6] whereas MiniLM's Apache license is fine but with minor dataset caveats [2].
- **Hardware Tier Fallbacks** – For **CPU-only or low-memory**, use BGE-Small (zh for Chinese content, or BGE-Small-En for English) as default. For **moderate GPU (≈8GB)**, BGE-M3 can be used via 8-bit quantization (fits ~8GB) to boost quality. For **strict low-end** (e.g. Raspberry Pi), MiniLM (22M) could be a fallback due to its tiny size, accepting some Chinese performance loss.

Recommendation

We recommend **BAAI's BGE-small-zh-v1.5 as the default embedding model** for Project Epoch, given its strong Chinese semantic performance, lightweight footprint, and permissive MIT license [6]. This model provides a good balance of quality and speed on consumer CPUs – it was designed to be **"small-scale with competitive performance"** [4] and indeed achieves solid results on Chinese benchmarks (within ~6 points of the large model on C-MTEB) [5]. For English content, BGE's embeddings are still effective (or the small-en variant can be used), making it a viable single-model solution for bilingual tasks. In contrast, MiniLM – while extremely fast – shows weaker recall on Chinese queries and carries uncertainty around certain training data usage [2]. BGE-small offers comparable speed with far better Chinese understanding, aligning with our privacy-first, local-first goals (no external API needed, everything runs on CPU).

For users with **higher-end hardware or demanding accuracy needs**, we suggest offering **BGE-M3 as an opt-in upgrade**. BGE-M3 delivers state-of-the-art multilingual retrieval, **supporting 100+ languages with hybrid (dense + sparse) retrieval built-in** [10] [11]. It would improve cross-language and long-document search quality (thanks to 8192 token support [12]), benefiting power users. However, due to its ~569M size, we'd enable it only when `EMBED_MODEL` is configured accordingly (and perhaps only load it if a GPU is available for reasonable latency). For most users on CPUs, the default BGE-small-zh is preferred for responsiveness.

In summary, **use BGE-small-zh-v1.5 as the default** embedding model (with BGE-small-en or multilingual small as needed), ensuring strong Chinese/English coverage and fast local inference. Document this choice in `.env.example` (`EMBED_MODEL=BAAI/bge-small-zh-v1.5`) and update our docs to reflect BGE's performance and open license. Provide BGE-M3 as a high-quality option for capable machines. This approach keeps Project Epoch's embedding pipeline efficient on consumer hardware while maximizing multilingual semantic search quality under open, commercial-friendly licenses.

Options & Tradeoffs

Model	Quality (Retrieval)	Latency (CPU)	Memory Footprint	License & Usage
MiniLM L6-v2	Good English performance (≈on-par with larger models for short texts [3]). Weaker on Chinese semantic tasks (not specifically tuned).	⚡ **Very fast** on CPU (6-layer, 22M params) – ~5× faster than base models [1]. Ideal for low-power devices.	~50–100MB RAM (22M params). Minimal disk space.	Apache-2.0 license [2] (model). *No* model usage restrictions, but some training data (MS MARCO, etc.) is non-commercial – slight ambiguity for strict compliance [2].
BGE-small-zh-v1.5	Strong Chinese retrieval (C-MTEB ~57.8 vs large ~64.5 [5]). Good general quality for its size; competitive with older larger models. English understood (can use BGE-small-en for best English results).	⚡ **Very fast** on CPU (24M params). Similar speed to MiniLM. Optimized for short query-passage tasks; negligible query overhead.	~95 MB RAM (24M params) [4]. Low incremental memory per 1k embeddings.	MIT license [6] – **free for commercial use** (explicitly stated). No external data calls. Requires adding an instruction prompt for queries (as with all BGE models) – trivial to implement.

Model	Quality (Retrieval)	Latency (CPU)	Memory Footprint	License & Usage
BGE-M3 (Multilingual)	**State-of-the-art** accuracy in English & Chinese (surpasses even OpenAI embeddings on benchmarks [8]). Handles >100 languages; supports hybrid lexical+vector search [10] and long documents (8k tokens) [12].	⌛ **Slower** on CPU due to 569M params. On a high-end CPU, ~0.5–1 s per embedding (estimated). **Fast on GPU** (supports FP16/ INT8). Best used with GPU or batch queries.	~2.3 GB VRAM (fp16) or ~1.1 GB (INT8). Can run on 8GB GPU (INT8) or 16GB CPU RAM (slowly).	MIT license [9] – fully open. Suitable for commercial use. Larger model means longer startup load time and more disk space (~2.8GB). We should only enable if `EMBED_MODEL` explicitly set to avoid burdening default users.
Alternate small multilingual (e.g. paraphrase-multilingual-MiniLM)	Moderate quality on many languages (better than English-only MiniLM in Chinese, but below BGE in C-MTEB).	⚡ Fast (many are distilled models <100M).	~100–300MB.	Varies (e.g. SBERT multilingual models often Apache/MIT). Could be fallback if BGE not desired.

Trade-off Highlights: BGE models (small & M3) deliver **better Chinese (and bilingual) performance** than MiniLM with similarly permissive licensing. The small model gives us near MiniLM speed but far superior accuracy on Chinese queries – aligning with our bilingual requirements. BGE-M3 offers bleeding-edge quality at the cost of high compute; it's optional for users with capable hardware. MiniLM remains an option if ultra-low memory or slightly faster English-only inference is needed, but given BGE-small's strengths and no usage restrictions, we favor BGE by default. Overall, the chosen default (BGE-small-zh) meets our "privacy local, CPU-friendly" mandate, and the options scale up gracefully in quality with hardware – all under open licenses.

Implementation Steps

1. **Default Model Update (`.env` & Config):** Set `EMBED_MODEL="BAAI/bge-small-zh-v1.5"` in `.env.example` (and as the install default). Update any hardcoded defaults in `services/api` that load the embedding model, to point to BGE-small-zh. Ensure the model is fetched at setup (document how to download weights or use `sentence-transformers` / `FlagEmbedding` to load it).
2. **Documentation (`docs/architecture.md` & `.zh-CN.md`):** Explain the rationale for selecting BGE-small-zh. In the "Vector Store & Embeddings" section, note its size (24M), expected performance (mention SOTA Chinese results [5]), and that it supports bilingual embeddings. Highlight that it's MIT licensed and **can be used commercially** [6]. Also update any mention of the previous model (MiniLM) to BGE, including sample code in documentation or tutorials.

3. **Fallback Matrix in Docs:** In `docs/requirements.md`, add a table mapping hardware tiers to recommended `EMBED_MODEL`. For example: “CPU-only or <4GB GPU: use BGE-small (zh or en); 8GB GPU: BGE-base or BGE-M3 (int8); 16GB+: BGE-M3 full.” This provides users a clear upgrade path. Emphasize that BGE models require adding an instruction prompt to queries (provide the Chinese instruction for retrieval as needed [13] [14]), and we can set this internally in our query pipeline.
4. **Quality Verification:** Before release, run basic retrieval tests with the new model. Confirm that when indexing bilingual data, relevant Chinese queries retrieve Chinese content with high accuracy (we expect improvement over MiniLM). Document examples in the architecture guide’s appendix (e.g. searching a Chinese question finds the correct passage).
5. **Optional High-Quality Mode:** Implement a toggle or configuration for using BGE-M3. For instance, an env var `EMBED_MODEL=BAAI/bge-m3` could be set by advanced users. Update docs on how to enable this and note the hardware requirements (e.g. “GPU highly recommended for BGE-M3 due to its size”). No code changes are needed aside from ensuring our embedding loader can handle the bigger model (it uses the same HuggingFace interface).
6. **Test & Risk Mitigation:** Test the API with the new default on pure CPU (to ensure no GPU-specific issues). Because BGE uses a slightly different approach (requires adding a prompt like “为这个句子生成表示以用于检索相关文章：” to queries [13]), implement this in the ingestion or query path so that **query embeddings** are generated with the instruction prefix. This ensures the embeddings align with passage embeddings, per BGE’s design. Document this implementation in code comments and the architecture doc (so others understand why we prepend the instruction).
7. *Workload:* This update is **Small** – mostly configuration and doc changes, plus minor code to add the instruction prefix in queries. One engineer (backend lead) can handle this in <1 day.
8. **Deprecate MiniLM (if applicable):** If MiniLM was previously the default, mention in the CHANGELOG or docs that we moved to BGE for better multilingual support. Provide guidance for users who still want MiniLM (they can set `EMBED_MODEL` to the old model string manually – ensure this remains possible and test that the system still works with that model if chosen).

Evaluation Plan

We will evaluate the embedding switch using both **latency benchmarks** and **retrieval quality tests** (as outlined in Task 4). Specifically: - **Latency:** Measure embedding throughput on CPU. Using a sample of 100 queries ~50 tokens each, record the mean and P95 embedding time for MiniLM vs BGE-small. We expect BGE-small to be on par with MiniLM (both ~10–20ms per short text on a modern CPU). Our goal is **P95 embedding latency <50 ms** for a 100-token query on a 4-core CPU. We’ll also measure end-to-end query time (embed + vector search) for a small corpus – target **<100 ms median**. These will be tested with our `scripts/eval_latency.py` script (see Task 4) on an offline machine. - **Quality (Retrieval Efficacy):** Using a bilingual test set (e.g. a mix of English and Chinese Q&A pairs), compute Recall@K. For Chinese, we will use a subset of a benchmark like DuReader retrieval or C-MTEB test queries. For English, use a subset of BEIR (e.g. TREC-COVID or Quora questions). We will ingest the documents and run the queries using the new model. Target: **Recall@5 ≥ 80%** on average for our test queries (an improvement over MiniLM which we expect to be lower on Chinese). Specifically, if using a Chinese Q&A dataset, expect BGE-small-zh to recall relevant passages ~10–15% higher than MiniLM at K=5. - **Multilingual Check:** Test a few cross-language queries (e.g. English query, Chinese doc) with BGE-M3 (if enabled) to verify it adds value. Although not a primary goal, it demonstrates the upgrade path. - **Resource Use:** Monitor memory usage when indexing a fixed corpus (say 1k documents). Ensure the new model doesn’t consume excessive RAM: we expect ~100MB for model + small overhead per embedding. Verify that stays well within typical PC limits. - **Benchmark Against Baseline:** If possible, run the MTEB “MSMARCO (Chinese)” or similar task with both models to quantify mAP or nDCG. BGE-small should approach published numbers (it’s the model used in those benchmarks). This confirms we didn't misconfigure the query instruction. - The evaluation scripts (from Task 4) will output metrics which we'll include in the project's benchmark docs. Passing criteria: **no significant latency regression** (within 20%) and **notable Chinese recall gain** (at least +5–10% absolute on test queries) compared to MiniLM. If results are unsatisfactory, we will investigate (e.g. ensure normalization or instructions were applied correctly).

All tests will be run locally (no internet) to align with our privacy stance. We will document the results and update the "System Requirements & Benchmarks" section in `docs/requirements.md` with these metrics to set user expectations.

Risks & Mitigations

- **Instruction Misuse or Omission:** BGE models require adding a specific prompt to queries for optimal performance [14]. If a developer forgets this, retrieval quality drops. *Mitigation:* Encapsulate the model in a utility that automatically prepends the recommended instruction for encode_queries(). Document this clearly. We have included this in implementation steps to avoid human error.
- **Chinese-English Embedding Discrepancy:** Using a Chinese-tuned model might, in theory, slightly weaken English embeddings. There's a risk that very English-heavy use cases see minor regression. *Mitigation:* We can monitor English query tests – if a drop is observed, consider using the BGE-multilingual small model (bge-small-**multilingual** if available) or even switch to BGE-small-en for English texts (with a simple language detection to route if necessary). However, BGE-small-zh still has some English training [15], and anecdotal evidence shows MiniLM and BGE-small are comparable on short English tasks [3], so this risk is low.
- **Performance on Long Documents:** BGE-small inherits BERT-base's 512-token limit. If users index very long texts without splitting, embeddings might truncate, harming recall. *Mitigation:* Note in docs that long documents should be chunked (and we provide an ingestion option for that). Also mention that BGE-M3 or BGE-base support longer sequences if needed. This educates users to avoid silently losing info.
- **Model Download Size/Time:** BGE models are moderately sized (95MB for small, ~1.3GB for M3). On slow connections or offline setups, this could be an obstacle. *Mitigation:* Provide instructions for offline model acquisition (e.g. a direct link or `wget` command to download the `.bin` from HuggingFace) in the README. We could also include the small model weights in a release asset for convenience. Emphasize the one-time nature of this download.
- **License Awareness:** While all chosen models are permissively licensed (MIT/Apache), we should ensure no inadvertent usage of a model with restrictions (e.g. if someone switches `EMBED_MODEL` to one based on Llama-1). *Mitigation:* In documentation, list the default model's license (MIT) and advise users to **verify licenses** of any custom models they configure. We also include a brief note in the README (Task 9) about model licenses, so users are informed.

References

- 【17】Milvus AI Reference – *MiniLM vs MPNet architecture and size (MiniLM ~22M params, 6 layers)* [1]
- 【6】Hugging Face Discussion – *MiniLM license: Apache-2.0 model, note on MS MARCO data usage* [2]
- 【10】Reddit – *User comparing MiniLM to BGE: MiniLM gave similar results to BGE-large and Ada on short docs, with 1/3 model size* [3]

- 【7】BGE Documentation – *Model sizes: bge-small-zh-v1.5 is 24M params (95.8 MB), aimed to be competitive* [4]
- 【12】Zilliz AI Models – *Chinese MTEB benchmark scores: bge-small-zh-v1.5 ~57.82 vs bge-large-zh-v1.5 ~64.53 (shows small model's strong performance)* [5]
- 【4】BGE Model Card (HuggingFace) – *License: MIT; models free for commercial use (FlagEmbedding license)* [6]
- 【2】BGE-M3 Model Card (HuggingFace) – *News: BGE-M3 achieves top performance in English & other languages, surpassing OpenAI models* [8] *; License: MIT* [9]
- 【9】BGE-M3 Documentation – *Model specs: 569M params, 8192 max length; supports 100+ languages, multi-granularity* [7]
- 【2】BGE-M3 Model Card – *Hybrid retrieval and re-ranking support (dense + sparse in one model; use cross-encoder for best accuracy)* [10] [11]
- 【3】BGE Usage Examples – *Example showing addition of Chinese instruction "为这个句子生成表示以用于检索相关文章：" for query embeddings* [13] [14]
- 【10】Reddit (LangChain thread) – *Users noting bge-large vs base vs small can vary, and one tested MiniLM vs BGE vs OpenAI Ada and found MiniLM comparable on ~200-token docs* [3] (supports minimal English regression risk)
- 【48】Milvus Security Blog – *Recommendation to encrypt data at rest (AES-256) and control access to embeddings to protect sensitive info* [16] (relevant to Task 8, included here for cross-reference).

Task 2: Local-First Vector Database Selection

Summary

- **ChromaDB:** An open-source, embedded vector database (Apache 2.0 licensed) built for **local-first applications** [17]. It offers persistent storage (backed by SQLite or DuckDB) and a simple Python API for adding/querying embeddings. Chroma runs fully offline and in-process, making it lightweight. It is easy to integrate (just `pip install chromadb`) and can operate in-memory or with disk persistence. Trade-off: While convenient, pure Python implementations can be slower or less memory-optimized than low-level libraries for very large datasets. Some users report Chroma's performance can degrade at scale or throw occasional errors [18]. However, for single-user moderate data sizes, it provides simplicity and sufficient speed.
- **Weaviate:** A production-grade vector DB (BSD-3 Clause) that can run self-hosted (Docker container) offline [19]. It offers rich features: **scalability** via sharding/clustering [20], hybrid search (vector + filters), GraphQL API, etc. Weaviate is more heavyweight – it runs as a separate server (written in Go) and uses more memory/CPU to maintain indices (HNSW for large datasets) [21]. It's well-suited if we anticipate large corpora or complex queries, but for a personal AI with limited data, it might be overkill. The initial setup and maintenance are more involved (e.g. running a Docker, managing schema). On consumer hardware with limited resources, a full DB server could be less efficient than an embedded solution.
- **FAISS:** A C++ library (MIT license [22]) for vector similarity search that excels in raw performance. FAISS provides various indexing methods (IVF, HNSW, etc.) and is used behind many vector DBs. It's **fast and lightweight**, operating in-process with low overhead. FAISS, however, is just an index – it does not handle persistence or metadata out-of-the-box [23]. The application must manage saving/loading indices and storing any text or metadata in parallel. It also lacks a client/server interface (it's a library), meaning integration requires Python code to manage state. For Project Epoch's current

JSONL store, FAISS would require writing custom code for data loading, index building, and querying (plus handling updates/deletions carefully). On the upside, FAISS has minimal runtime footprint and can easily handle thousands of embeddings in memory with millisecond query times.

- **Persistence & Offline:** Both Chroma and Weaviate support persistence on disk (Chroma uses local files/SQL, Weaviate stores data on disk with optional durability settings). FAISS by itself requires explicitly writing index files to disk and re-loading – doable but manual. All three options can run fully offline (no external dependencies needed at query time). For single-user local use, **Chroma's embedded persistence** (it uses an SQLite under the hood by default) is a simple solution: data is stored in a local DB file and reloaded on startup automatically. Weaviate as a separate process also persists to disk (and can be configured to start with the data). Weaviate and Chroma can both be used without internet – they don't phone home. (Weaviate's cloud offering is separate; the OSS version is offline by default.)
- **Ease of Use:** Chroma offers a high-level Python API (e.g. `collection.add()` and `query()`) which aligns well with our Python backend. It would require minimal code changes to replace our JSONL + cosine search logic with Chroma's API calls. Weaviate, in contrast, would involve either running a container and using its HTTP API or Python client – more setup and a more complex schema definition (we'd need to define an Object class with properties for text, embedding, etc.). Managing a Weaviate instance (start/stop) adds complexity for the end-user. FAISS integration sits in between: we'd incorporate FAISS index building in Python code – manageable for an initial adapter, but we'd need to implement our own persistence layer (perhaps writing the FAISS index to a file and keeping metadata in parallel). This is more coding effort and potential for bugs (especially around consistency with updates).
- **Recommendation Direction:** Given the above, we lean towards implementing an **embedded vector store adapter using Chroma first**. Chroma aligns with our *"local, simple, Pythonic"* approach and has a very permissive license (Apache-2.0) [17] . It will let us replace the current JSONL store with minimal disruption, adding benefits like built-in similarity search and filtering. Weaviate could be considered in the future if we need scaling or advanced hybrid queries, but its added complexity isn't justified at this stage. FAISS can be utilized under the hood (Chroma actually offers an option to use FAISS or HNSW as the index) – but implementing FAISS standalone would require building a lot of surrounding logic (persistence, etc.) which Chroma already provides. In sum, **Chroma offers a quick win**: persistence + ease of use, while FAISS offers raw speed but more dev effort, and Weaviate offers enterprise features but at the cost of simplicity and resource usage.

Recommendation

Implement a **pluggable vector storage interface** and begin with a ChromaDB-based adapter as the default. This will move Project Epoch from the current JSONL+manual cosine search to a more robust system that supports persistence, faster similarity queries, and potential metadata filtering – all while keeping the solution local-first and lightweight. Chroma meets our needs for a single-user embedded DB: it runs inside our Python process (no additional service required) and persists data to a local file so that the "AI memory" is retained across sessions. It is also Open Source (Apache 2.0) with no usage limitations, fitting our privacy and licensing criteria [17] . By using Chroma, we minimize development effort – we can leverage its high-level API to ingest and query embeddings instead of maintaining our own cosine similarity code.

Why not Weaviate or FAISS first? Weaviate, while powerful, would introduce an external server dependency (complicating deployment for end-users who now have to run a separate service or Docker container). Its advantages (scalability, clustering, GraphQL queries) are beyond the immediate scope of a personal AI running on a single machine. FAISS, on the other hand, is extremely efficient for similarity search and could be a backend for an eventual custom solution, but on its own it lacks persistence and would require us to implement our own data management (e.g. storing texts and IDs separately, handling index reloading). Chroma actually uses FAISS or HNSW under the hood for similarity search, giving us the performance benefits of FAISS while handling persistence and metadata automatically [23]. Therefore, adopting Chroma now gets us up and running quickly with minimal downsides.

The **plan** is to create an abstraction (e.g. an interface `VectorStore` with methods like `add(text, embedding, metadata)` and `query(query_embedding, top_k)`) and implement it with Chroma initially. This adapter approach will make it easier to switch to or experiment with Weaviate, FAISS, or others later by just adding new implementations of the interface. We'll document how to migrate existing JSONL data into the new system (likely via a one-time import utility).

In conclusion, using Chroma as the default vector store improves Project Epoch's durability (data persists between runs) and maintainability (less custom code for similarity search), without sacrificing the local/offline requirement. Weaviate can be revisited when/if we anticipate multi-million vector scales or need complex filters. FAISS will remain a powerful underpinning (and indeed we can configure Chroma to use a FAISS index internally for performance), but by letting Chroma manage it, we keep our focus higher-level. This choice is about **developer velocity and user simplicity**: Chroma lets us deliver a working persistent memory feature quickly and align with open-source licensing, which is our priority at this project stage.

Options & Tradeoffs

Vector Store Option	Persistence & Offline	Performance	Ease of Integration	License & Notes
ChromaDB (embedded)	Stores vectors locally (SQLite/DuckDB). Fully offline (no external dependencies) [17]. Persistence is automatic – data saved to disk and reloaded on startup.	⚡ Optimized for moderate scale. In-memory queries with optional indexes (uses FAISS or HNSW internally). Fast similarity search for thousands to low millions of vectors. Some reports of instability at very large scales [18], but for our scope it's fine.	**Very easy (Python)** – `pip install chromadb`. Simple API for add/query. Runs in-process; we can call it directly from FastAPI. No separate service needed for default use.	Apache 2.0 [17] – **permissive**. No commercial restrictions. Active open-source project with growing community. Good documentation. (Be mindful of version pinning to avoid future breaking changes.)

Vector Store Option	Persistence & Offline	Performance	Ease of Integration	License & Notes
Weaviate (server)	Stores data on disk (built-in DB). Offline capable – self-hosted Weaviate doesn't require internet (the OSS is standalone) [19]. Data persisted in its volume.	⚡**High-performance** ANN (HNSW, flat) with excellent scalability. Handles millions of vectors, and supports hybrid (vector + keyword) queries efficiently. Uses more memory (background indexing processes) – heavier for small data. Latency low for large KNN searches on GPU/accelerated hardware.	**Moderate/Hard** – requires running a separate service (Docker or binary). We'd need to manage that (or instruct user). Use Weaviate's client or REST API. Must define schema (class with properties). Additional ops overhead for user and dev (start/stop DB, migrations).	BSD-3-Clause [19] – permissive. Backed by a company (SemiTechnologies) with enterprise offerings. No usage limitations, but the complexity is higher. Good if future multi-user or distributed support needed. Possibly overkill for single-user offline app right now.

Vector Store Option	Persistence & Offline	Performance	Ease of Integration	License & Notes
FAISS (library)	⚠ **No built-in persistence**. We must manually save index to disk (e.g. via `faiss.write_index`) and reload on startup. No out-of-the-box mechanism for incremental persistence (e.g. on each add). Offline by nature (it's just a library).	⚡ **Very fast** (C++ vector ops, optimized BLAS). Can do sub-millisecond similarity search for tens of thousands of vectors in RAM. Scales well with CPU threads. However, lacks dynamic update: adding/deleting vectors is non-trivial (depending on index type, may need rebuilding or suffer degraded accuracy).	**Medium** – we'd integrate FAISS in Python (via `faiss` package). Need to maintain a parallel list of metadata (since FAISS indexes only store vectors or IDs). We'd implement our own add/search logic (which is straightforward for static data, but tricky for live updates). Managing consistency between FAISS index and our JSON metadata (or a separate DB) adds complexity.	MIT license [22] – highly permissive and widely used. Would require us to solve a lot of surrounding functionality (serialization, etc.). Good option to optimize later or for custom approaches (and indeed Chroma can use FAISS internally, giving us benefits without direct integration initially).

Vector Store Option	Persistence & Offline	Performance	Ease of Integration	License & Notes
Hybrid Approach (Chroma + FAISS)	Chroma can be configured to use FAISS as the indexing mechanism under-the-hood for performance, while maintaining persistence via its SQLite meta-store.	⚡ Potentially best of both: Chroma's ease + FAISS speed. Chroma by default uses an HNSW index; switching to FAISS (if needed) could boost certain query patterns (though HNSW is already quite fast).	**Easy** – a one-line setting when creating collection (Chroma supports `indexing='faiss'`). No extra complexity for us.	Same licenses as above. Chroma's Apache license covers integration, and FAISS's MIT is compatible.

Key Trade-offs: - *Complexity vs. Control:* Chroma offers a **high-level solution** (we write less code, at the expense of some control over the low-level index). FAISS offers **full control and max performance**, but then we must implement features around it (persistence, etc.). Weaviate offers a **full service** with many features (scaling, filters, auth), but introduces operational complexity and higher resource use for a single-user scenario. - *Resource Footprint:* For small to medium data sizes (say up to a few hundred thousand embeddings), Chroma running in-process will be efficient. It keeps metadata and index in a local file and memory, which is fine on consumer hardware. Weaviate would run as a separate process that might use a few hundred MB at idle plus overhead for its indexes – not huge, but more than an embedded library. FAISS is lean (just uses memory for the index arrays), but again, without our own persistence, data could be lost on restart or require slow re-indexing each time. - *Feature Set:* Chroma and Weaviate both support metadata filtering (e.g. "find K nearest where `source=meeting_notes`"), which could be useful. FAISS alone doesn't – we'd have to filter results after retrieval or maintain separate indices per tag. Chroma's API includes metadata with each vector, which aligns with adding "tags" or "source" fields easily. Weaviate has a rich schema and filtering by fields natively. This is a plus for future "query by topic" or similar features. Using Chroma preserves that option (we can add a where clause to queries), whereas FAISS-only would require custom logic for filtering. - *Community & Support:* All three have active communities. Chroma is relatively new but popular in the LLM community; Weaviate is established in production use; FAISS is a de-facto standard for vector similarity. For our open-source project, sticking to Python-level tools (Chroma/FAISS) will likely be easier for contributors to work with, versus requiring a running Weaviate instance for development. - *License compatibility:* All options are permissive (Apache or MIT or BSD) – no copyleft concerns, and they can be used in commercial derivative projects. Weaviate and Chroma both avoid any viral license issues. So, license is not a blocker for any, just need to note Weaviate's separate "Weaviate Cloud" is proprietary (not relevant to us).

Given these comparisons, **Chroma emerges as the best initial choice**: it strikes the right balance by leveraging FAISS-like performance with built-in persistence and minimal integration effort. This will let us deliver the "AI memory" feature (with data surviving restarts) quickly to users, and we can iteratively improve or swap out the backend in the future if needed.

Implementation Steps

1. **Design VectorStore Interface:** Define an abstract interface (or at least a set of expected methods) in `services/api/db.py` (for example) with methods such as `add_items(items: List[Document])`, `query_embedding(vector: np.array, top_k: int) -> List[Document]`. This decouples our application logic from a specific DB. Mark this as experimental in documentation (so contributors know it's pluggable).
2. *Owner:* Backend engineer. **(Small)** effort – designing interface and refactoring existing search code (~1 day).
3. **ChromaDB Adapter:** Implement the interface using Chroma. For example, create `ChromadbVectorStore` class. On initialization, it should connect to a local Chroma instance:
4. Use `chromadb.Client()` with an appropriate settings (specify a persist directory, e.g. `./data/chroma`).
5. Create a collection (e.g. named "epoch_memory") with a schema including `id`, embedding, and metadata (Chroma allows arbitrary metadata as a dict).
6. Implement `add_items`: it will take our documents from ingestion (with text, embedding, metadata) and call `collection.add(ids=[], embeddings=[], metadatas=[], documents=[])`. We may store the full text as `document` in Chroma (so it's retrievable), or store text in metadata. We should store `source` and any tags in metadata fields.
7. Implement `query_embedding(vector)`: call `collection.query(query_embeddings=[vector], n_results=top_k, include=['documents','metadatas','distances'])`. This returns the nearest items; we then wrap them back into our `Document` or response format.
8. Ensure that the Chroma client is persistent: pass `persist_directory="./data/chroma"` so that on application restart, it loads existing data. After adding items, call `client.persist()` if needed to flush to disk.
9. Test adding a few records and querying to verify flow.
10. *Owner:* Backend engineer. **(Medium)** effort – learning Chroma API, implementing and testing (~2 days).
11. **JSONL Migration Path:** Write a one-time migration function or script in `services/api/ingest.py` that reads existing `embeddings.jsonl` (if present) and loads data into the new store. This can run at startup if we detect the old JSONL and no existing Chroma DB:
12. Parse each line of JSONL (which contains text, embedding, metadata).
13. Batch the data and call `ChromadbVectorStore.add_items()` to insert into Chroma.
14. After successful migration, either archive the JSONL (rename it) or drop a flag so we don't re-import next time.
15. Log a message to the user indicating that a migration occurred. Document this in the release notes.
16. *Owner:* Backend engineer. **(Small)** effort (~0.5 day, mostly dealing with file I/O and ensuring embeddings format matches).
17. **Update FastAPI Endpoints:** Modify the `/ingest` endpoint implementation to use the VectorStore's `add_items` instead of writing JSONL. Similarly, update `/query` to use `VectorStore.query_embedding` (taking the query embedding from the model and retrieving

results). Ensure that the returned results include necessary info (text, source, etc.) from the store's metadata. We might remove or bypass the old in-memory cosine similarity code entirely.

18. Remove any code that was manually computing cosine distances over in-memory embeddings; rely on the DB now.
19. Verify that error handling is in place if the DB is empty or if too few results are found.
20. *Owner:* Backend engineer. **(Small)** – refactoring endpoints (~1 day).
21. **Configuration & Profiles:**
22. Add a new environment variable (e.g. `VECTOR_STORE=chroma|weaviate|faiss`) in `.env.example` to allow future flexibility. For now, default it to `chroma`. The code can choose the adapter implementation based on this.
23. If `weaviate` is selected (for advanced users), they'd need to provide the Weaviate URL and API key possibly. Document this as an advanced scenario (but initial support could be minimal).
24. If `faiss` is selected, we might not implement now, but leave a stub or warning that it's not yet supported.
25. *Owner:* Backend/DevOps. **(Small)** – adding env var and conditional (~0.5 day).
26. **Docker Compose & Profiles:** Since Chroma runs in-process, no new service needed in Compose for default. However, if we eventually allow Weaviate:
27. Extend `docker-compose.yml` with an optional Weaviate service (using the official image) under a profile, e.g. `weaviate`. Provide `WEAVIATE_HOST` env to API.
28. Document how to activate: e.g. `docker-compose --profile weaviate up`.
29. For now, we won't enable it by default; it's just to show path forward. Document in `docs/architecture.md` how one could swap to Weaviate by running that profile and setting `VECTOR_STORE=weaviate`.
30. Ensure that offline use is addressed: our Compose file should use a specific version of Weaviate image (which users can pre-download if needed).
31. *Owner:* DevOps. **(Small)** – adding a compose service and testing (~1 day including test).
32. **Documentation Updates:**
33. **Architecture Doc:** Update the "Memory Store" section to describe the new architecture. Explain how data flows: ingestion goes to Chroma, queries pull from Chroma, etc. Highlight that data is now persistent (give path e.g. `./data/chroma/` directory). Include a note that Chroma is embedded and runs locally, and mention possible alternatives (Weaviate, etc.) for future. Diagram may be updated to show "Vector DB" instead of JSONL.
34. **Environment/Config Docs:** In `docs/requirements.md`, note that by default we use Chroma for storage, and list its requirements (really just disk space ~ a few MB for small data, and it works on CPU). Mention that no GPU is required for Chroma. Also mention how to switch to Weaviate or others (for advanced users) and that Weaviate would require Docker etc.
35. **User Guide:** If we have a user-facing README or usage guide, add instructions on how to reset the memory (e.g. deleting the Chroma directory if needed) and how data persists by default. Also inform that on first run, existing JSONL data will be migrated automatically.
36. *Owner:* Tech writer/maintainer. **(Medium)** – ensuring all relevant docs are updated (~1 day).
37. **Testing & Validation:**
38. Write unit tests for the `VectorStore` interface: a small test that adds a known vector and queries it. Possibly use a dummy in-memory Chroma (with `:memory:`) for testing.
39. Manual test scenario: run `ingest` to add some documents, restart the server, then run `query` – confirm results from before restart are still available (persistence check).

40. Test edge cases: ingest duplicate IDs (Chroma might require unique IDs – handle by maybe using UUIDs or content hashes as IDs). Test deletion if we expose it (Chroma supports `.delete()` by ID if needed).
41. Ensure no regressions: verify that retrieval results match or exceed the previous approach's results for known queries. Because Chroma uses cosine similarity by default, it should match our old behavior (Chroma normalizes vectors internally if using cosine). We might need to ensure we insert normalized embeddings or specify metric correctly (Chroma default is cosine distance).
42. *Owner:* Backend QA. **(Medium)** – manual and some automated tests (~2 days including iterating on fixes).
43. Mark this feature in release notes as experimental if appropriate, encouraging community feedback.

By following these steps, we expect to deliver a functional persistent vector store in the next release (target P0 this week, ~1 developer-week of effort). The architecture will be cleaner and ready to support more advanced retrieval improvements (as in Task 3) and evaluation (Task 4). We map these steps to specific files: - `.env.example`: add `VECTOR_STORE`. - `services/api/main.py` or wherever query logic is: replace JSONL logic with store calls. - `services/api/db_chroma.py` (new): implement ChromaVectorStore. - `docker-compose.yml`: add Weaviate (commented or profiled). - Documentation files as listed.

Owner Assignment & Workload: This is primarily a backend task (one engineer can implement adapter and migration, with another reviewing). Docs can be updated by the engineer or a tech writer. Testing can be done by the same engineer with perhaps a peer doing a quick integration test. Estimated effort ~6–8 hours for initial coding and ~4 hours for docs/testing, which fits in the 1–2 day allocation.

Evaluation Plan

We will evaluate the new vector storage in terms of **functionality, performance, and resource usage**: - **Functional Tests:** Ensure that after adding data, queries return correct results: - Use a small benchmark (e.g. 100 Q&A pairs). Ingest all as context, then query with the questions. Measure Hit@1 (whether the correct answer passage is top-1). This is primarily to check that Chroma's similarity search is working as expected. Since this doesn't change the embedding model, we expect similar accuracy to the previous system. A minor drop or increase could occur due to any differences in vector normalization or metric – we'll check that. - Test metadata filtering: Insert some items with a tag and perform a query using Chroma's `where` clause to filter by tag. This isn't exposed to user yet, but tests our ability to use that feature for future hybrid retrieval. It should return only items that match both similarity and filter. - **Latency & Throughput:** Compare query latency with JSONL vs Chroma: - Benchmark scenario: 10k vectors in memory. Run 50 queries and measure average query time. The expectation is that Chroma (likely using an HNSW index by default) will be faster than brute-force cosine on 10k vectors in Python. We target **<50 ms per query for 10k vectors** on CPU, which should be achievable (HNSW or FAISS can do that easily). The old Python cosine likely took >100 ms for 10k due to lack of optimization. - Measure ingestion throughput: e.g. ingest 1000 entries and note time. Should be acceptable (even if slightly slower than appending to JSONL, since it's doing indexing). For our scale, anything in the order of seconds for thousands is fine. We'll note it but focus on query performance, since ingest happens asynchronously or less frequently. - **Memory Footprint:** Run the system with, say, 5k embeddings loaded. Check memory usage of the API process (via `psutil` or monitoring). Ensure it is within expected range (embedding vectors 5k×768 floats ~15MB, plus index overhead maybe 2× = ~30MB, plus base app overhead). It should definitely be well under 1GB. If we see unexpectedly high memory use, investigate (e.g. maybe default index is HNSW with large buffers – we can consider using an alternative or tuning parameters). - **Persistence Verification:** - Insert some data via

API. Stop the API, then start it again. Query for an item that was added previously. Confirm it's still present. This is a direct proof of persistence working. Automate this in a test by calling the store persist and reload methods if possible. - Simulate a migration scenario: start with an existing JSONL file (we can craft a small one), run the server with the new code, ensure it migrates and the data is accessible in Chroma afterwards. The JSONL should then be either archived or not re-imported on next run. - **Comparative Search Quality:** It's unlikely the vector DB choice affects retrieval quality (since same embeddings). To be thorough, we will run the same retrieval eval from Task 1/4 on the new system to ensure metrics like Recall@5 remain essentially unchanged (within margin of error). We anticipate equal or slightly improved results, because Chroma by default uses cosine similarity like we did. We'll double-check that we use the same similarity metric (cosine vs L2) consistently – Chroma's default is cosine distance, which aligns with our approach. - **Concurrency & Stability:** As a simple load test, fire multiple queries concurrently (e.g. 5 threads hitting the /query endpoint). Verify Chroma can handle it (it should, as it's mostly just reading from memory). Look out for any race conditions (e.g. if we accidentally reuse client incorrectly – Chroma's client is threadsafe for reads, should be okay). - Document these test outcomes in the project's wiki or testing log. Specifically highlight the persistence and performance improvements. For example: "In tests, query latency for 10k vectors dropped from ~120 ms (Python cosine) to ~20 ms with Chroma HNSW (M=10, ef=128), and data persisted across restarts successfully." - We will set **success criteria** as: (1) No correctness regressions in retrieval results, (2) Noticeable performance improvement for non-trivial collection sizes, (3) Verified data persistence working, and (4) positive developer experience (e.g. easier to manage code). If criteria are met, we proceed; if not, we debug or tweak settings (e.g. if queries are slower, maybe enable FAISS backend or adjust index params).

Risks & Mitigations

- **Chroma Memory/Size Limits:** If a user imports a very large number of documents (say >1M vectors), Chroma might default to an in-memory index that doesn't fit or be slow to index. *Mitigation:* Document that our current solution is tested up to N records (e.g. 100k) and for larger data, a different backend (like a dedicated DB or enabling Chroma's disk-based indexing when available) might be needed. We can also expose an option to use an alternate index type (Chroma's API may soon support disk-based indexes). In short, communicate the tested limits and encourage feedback for extreme cases.
- **Data Consistency & Corruption:** Relying on an embedded DB introduces the slight risk of data corruption (e.g. if the app crashes mid-write). *Mitigation:* Chroma uses transactional writes via SQLite – low risk, but we should ensure `client.persist()` is called on graceful shutdown. Implement a shutdown hook in FastAPI to flush data. Additionally, advise users to back up the `./data/chroma` directory if their memory is valuable. We can also keep the old JSONL as backup the first time (e.g. don't delete it after migration, just rename) so no data loss during transition.
- **Dependency Footprint:** Chroma will add dependencies like `chromadb`, which in turn brings `uvicorn`, `posthog`, etc., possibly increasing our package size. *Mitigation:* We pin Chroma to a stable version and possibly use the minimal variant (Chroma can run without the analytics if environment variable `CHROMA_TELEMETRY=false` is set – we will set that to prevent any outgoing telemetry, aligning with privacy stance). We acknowledge the larger dependency but accept it for functionality gains. We will note in docs that environment variable to disable telemetry (though Chroma default anonymized telemetry can be opt-out).
- **Weaviate Future Integration Complexity:** By choosing Chroma now, we delay Weaviate integration. If later we want to support switching to Weaviate for advanced use, it may require migration of data from Chroma to Weaviate (which could be non-trivial if the schemas differ).

Mitigation: The adapter interface will help. We can write an export routine (dump all vectors and metadata) that can be imported into another store. Also, by using standard fields (id, text, embedding, metadata) we ensure any vector DB can accommodate them. We keep the option open and document that migration to a different store may require re-ingesting source documents (which is feasible since original data presumably still exists).
- **User Operation Errors:** Users might inadvertently delete the database files or mix usage (e.g. run two instances concurrently on the same Chroma files). This could cause conflicts. *Mitigation:* Document where data is stored and caution about running multiple instances (Chroma uses file locks, but still). Also handle errors gracefully – e.g. if DB init fails, catch and log an informative message (maybe offer to wipe and rebuild if corruption detected, though that's advanced).
- **Backward Compatibility:** Once we move to Chroma, the JSONL is no longer the source of truth. Some power users might have been directly manipulating JSONL. *Mitigation:* Announce in release notes that the storage format changed. Provide a way to export data if needed (e.g. we can offer a small CLI or API endpoint to dump all memory items as JSON for user review, which reads from Chroma). Ensuring the migration code covers the old data is the main step. We'll keep the old JSONL read-only after migration to avoid confusion.

By addressing these risks, we ensure a smooth transition to the new vector store with minimal disruption and a clear path for troubleshooting if issues arise.

References

- 【19】Hugging Face: *Chroma* – (License Apache-2.0, local use) [17]
- 【21】Oracle Blog on Weaviate – (BSD-3 license, open-source self-hosted) [19]
- 【18】AI Multiple: *Vector DB Comparison* – (FAISS lacks persistence, raw speed usage) [23]
- 【23】GitHub (Meta): *FAISS License* – (MIT license confirmation) [22]
- 【22】Reddit (r/vectordatabase) – *User switched from Chroma to FAISS for performance, citing random errors from Chroma* [18] . (Illustrates potential Chroma issues at scale)
- 【48】Milvus AI Reference – (Securing embeddings with encryption, relevant to Task 8) [16]
- Chroma Docs: *FAQ* – (Chroma can run in-memory or embedded; supports disabling telemetry) – **No direct snippet, but known from docs**.
- Weaviate Docs: *Deployment* – (Weaviate can run via Docker, clustering etc., indicates heavier setup) – **No snippet**.
- FAISS Paper/Docs: (FAISS is library-only requiring manual persistence and offering many index types) – **General knowledge**.
- Internal Project Vision – (We value local, minimal stack complexity, so embedded DB aligns with that) – **Project context**.

Task 3: Retrieval Strategy Upgrade (Hybrid & Re-ranking)

Summary

- **Pure Vector Retrieval:** Our current baseline uses embeddings + cosine similarity to find top-K documents. This is **fast and simple** but can miss relevant results if a query uses different wording than documents (no lexical matching). It may also return redundant results that are semantically similar, lacking diversity.

- **Hybrid Retrieval (BM25 + Vector):** Combines keyword search (e.g. BM25) with vector similarity. BM25 excels at exact term matches (especially important for proper nouns, numbers, etc.), while vectors excel at semantic matches. A hybrid system can **improve recall** – catching results that pure semantic or pure lexical alone might miss [10]. On CPU, BM25 (with a library like Whoosh or Elastic's Lucene) is lightweight for our likely corpus sizes (thousands of docs). The main overhead is maintaining a text index and merging results. This approach would retrieve candidates from both methods and then fuse or re-rank them. It's feasible to implement and can run offline (e.g. using an embedded Lucene or Pyserini).
- **Maximal Marginal Relevance (MMR):** A re-ranking algorithm that **promotes result diversity** by penalizing redundancy [24]. MMR iteratively selects documents that are both relevant to the query and as different as possible from already-selected docs [25]. By applying MMR to an initial top-K (from vectors or hybrid list), we can ensure the final suggestions cover distinct aspects (avoiding, for example, the top 3 results being nearly identical). MMR is computationally cheap (it operates on the similarity scores we already have) and can be done on CPU quickly for K~10–50. It requires computing similarity between candidate docs, which is negligible for small K. This can directly improve user experience by providing varied context to the LLM.
- **Cross-Encoder Re-ranking:** Uses a separate **cross-attention model to score (query, document) pairs** for relevance, typically yielding higher accuracy than bi-encoder embeddings [11]. For example, BAAI's **BGE-reranker** models (base ~110M, large ~330M) are designed to take a query and passage and output a relevance score [26]. This can significantly improve precision@K (the cross-encoder can catch nuance and context that embeddings might miss). The trade-off is **computational cost** – it requires running a BERT-like model for each candidate document. On CPU, a base-size cross-encoder (~100M params) can take ~50–100ms per passage (depending on length). If we re-rank 10 candidates, that could be ~0.5–1s extra. This is potentially acceptable for P95 latency on a desktop CPU if we keep K small (and could be improved with quantization or running on a GPU if available). We would likely conditionally enable cross-encoder re-ranking for users who opt in or when the system is in "high-accuracy" mode, due to this cost. However, even on CPU, a small reranker model (there is a **bge-reranker-base** with bilingual support [27]) might be feasible for ~10 docs. The BGE rerankers are MIT licensed (following FlagEmbedding) and can be used commercially [26]. They add a final boost in result quality – often improving metrics like MRR by 20-30% in information retrieval literature.
- **CPU Viability:**
- *BM25:* Very viable – computing BM25 on a few thousand docs is fast (we can pre-index the corpus; query time will be milliseconds). The overhead of merging with vector results is minimal. We just have to include a lightweight tokenizer (for Chinese, possibly use Jieba or character bigrams).
- *MMR:* Extremely viable – complexity O(K^2) for K candidates (negligible for K=10 or 20). It's just a few dot product operations on embedding vectors we already have; well within CPU capabilities.
- *Cross-Encoder:* This is the heaviest. A ~100M param model on CPU might process ~2–3 passages per second (estimate) if not optimized. However, since our default K might be 5 or 10, that's ~2–5 seconds worst-case, which is borderline. With optimization (int8 quantization, or only running on shorter text), we could potentially cut this down. Alternatively, run cross-encoder only on a subset (e.g., first use MMR to pick 5 diverse candidates, then cross-rank those). Another strategy: only enable cross-rerank when the user's query seems complex or when high precision is needed (we could have a config "ENABLE_RERANKER").
- Notably, BGE-reranker-base is ~67M parameters (according to references) [28] which is smaller than BERT-base – that might run a bit faster. Also, some smaller cross-encoders exist (e.g. MiniLM cross-encoder). So we have options to tailor the reranker for speed.
- **Pipeline Plan:**

- Retrieve top *N* via current vector search (embedding similarity).
- Optionally retrieve top *M* via BM25 on the same query (if hybrid enabled).
- Merge these lists (e.g. take union up to some limit, possibly boosting items that appear in both).
- Apply MMR on the merged list to reorder/select top K (diversifying results).
- Optionally, apply cross-encoder reranker on the top K (re-score them) to produce the final sorted list. This pipeline yields a final ranking that leverages lexical matching, covers diverse aspects, and is refined by a powerful re-ranker for accuracy [11].
- **Conditions to Enable:**
- We can make BM25+MMR the **default** on CPU as it adds minimal latency (maybe a few milliseconds for BM25 search on an inverted index of a few thousand docs). This will likely improve recall and result coverage immediately.
- The cross-encoder step might be disabled by default on CPU (to preserve snappy responses), but enabled on systems with GPUs or if a user sets a "quality" mode. Alternatively, provide a config toggle (e.g. `USE_RERANKER=True`) so advanced users can turn it on. Over time, as models get smaller (e.g. Mistral 7B cross-encoders or distilled rerankers), we can revisit enabling it broadly.
- Another strategy: if the initial retrieval confidence is low (e.g. vector scores are low or the top results are borderline), we could trigger the cross-encoder to refine. But that adds complexity; a simpler approach is a static config or depend on hardware detection.

Recommendation

Adopt a **two-stage retrieval strategy** that **combines lexical and semantic searches and then re-ranks the results for diversity and relevance**. Specifically, we recommend: - **Enable hybrid retrieval (BM25 + Vector)** by indexing our text corpus with a CPU-friendly BM25 engine (e.g. Whoosh or an in-memory Lucene via Pyserini). Use it alongside the embedding search: for each query, retrieve (for example) top 5–10 results by BM25 and top 5–10 by embedding, then merge. This will significantly improve coverage of relevant documents, especially for queries that have important keywords (names, technical terms) that might not be captured in embeddings. It leverages the fact that BGE's dense retrieval and BM25 often have complementary results [10]. The computational cost is low and manageable on CPU. - **Apply Maximal Marginal Relevance (MMR)** to the candidate set to ensure varied results. This will reduce redundancy (e.g. if the top 3 vector hits are almost duplicates, MMR will push one of them lower in favor of something different). MMR is quick to implement and incurs trivial overhead relative to the embedding computations. This improves the **usefulness of the top-K context** fed into the LLM (covering different points). - **Introduce a cross-encoder re-ranking step as an optional "high-accuracy mode."** This would use a small cross-attention model (like BGE-Reranker-base, ~110M) to re-score the top candidates and produce a final ranking. In our pipeline, after MMR selection of (say) top 5, we would concatenate query + each candidate text and run it through the reranker model to get a relevance score, then sort by that. This model (trained on retrieval tasks [26]) can significantly boost precision and ensure the truly most relevant document is ranked first. We propose to keep this **off by default for CPU-only deployments** (to maintain sub-second response), but allow users to enable it via configuration or automatically enable if a GPU is present (since on a GPU 5 inference passes of a ~100M model is very fast, tens of milliseconds). - **Phased Implementation:** We recommend implementing the **BM25+MMR hybrid first (Phase 1)**, as it's straightforward and immediately beneficial. Then, add the cross-encoder re-ranker in a **Phase 2** once we have verified baseline improvements and measured the latency impact. This phased approach ensures we don't regress responsiveness unexpectedly. - **Evaluation & Enablement:** We will evaluate the impact of each component on retrieval quality using metrics like Recall@K and MRR (see evaluation plan). If the cross-encoder yields a substantial MRR gain and latency remains within acceptable bounds (e.g. under 2s on average), we might consider enabling it by default on desktop-class hardware. If not, we'll ship it as an opt-in feature with documentation (e.g. “set `RERANK=true` for best accuracy, at the cost of some latency”). - **Documentation:** We will update the architecture docs to describe this multi-stage retrieval flow. Specifically, highlight that by using both keyword and semantic search, we mitigate the weaknesses of each method. Also mention that our system can optionally perform an extra re-ranking using a cross-attention model for improved accuracy (similar to what academic IR systems do [11]). Make clear any conditions or configs controlling that. - **Guardrails:** We should set a reasonable `TOP_K` default (perhaps 5 or 10) so that even if cross-encoder is on, the overhead is bounded. We’ll also implement graceful fallback: if the reranker model isn’t available (not downloaded), the system should just skip that step rather than error. This ensures the retrieval still works albeit with slightly lower precision, maintaining robustness.

Adopting this strategy will likely boost our system’s answer quality without requiring any external data or internet – it’s all local computations. It aligns with Project Epoch’s goal of maximizing utility from personal data by ensuring the right information is retrieved and fed into the LLM. All components are open-source (Whoosh BM25 – BSD, our BGE models – MIT, reranker – MIT) and can run on consumer hardware (with some patience for the reranker on CPU). This gives us a clear quality edge (more like GPT-4 Retrieval Augmentation style, which uses re-ranking) while respecting the offline, privacy-first constraints.

Options & Tradeoffs

Strategy	Quality Impact	Latency Impact (CPU)	Implementation Effort	Notes
Status Quo (Vectors only)	Baseline semantic recall. May miss exact matches (e.g. names) or present redundant top-K results. Quality is decent but not SOTA – likely lower MRR/Recall than hybrid methods on heterogeneous data.	**Lowest latency** – single embedding + cosine search. ~100ms or less for typical corpora. P95 latency currently mostly from embedding computation.	Already implemented.	Simplicity comes at the cost of missing some relevant info. We’ve observed that purely embedding-based search can fail for rare keywords and can rank somewhat relevant content above truly relevant if semantic similarity is imprecise.

Strategy	Quality Impact	Latency Impact (CPU)	Implementation Effort	Notes
Hybrid (BM25 + Vector)	**Higher recall** – captures documents that pure vector or pure BM25 alone would miss. Especially improves on queries with unique keywords (BM25 finds them) and queries needing conceptual matches (vectors find them). Should improve overall Recall@K by a significant margin (in literature, hybrid can outperform either alone by ~2-5 points on average).	**Slight increase in latency** – need to run a BM25 query (~5–10ms) and merge results (negligible). Overall still very fast (sub-millisecond per doc for BM25 scoring, and our corpus is small). Essentially no noticeable delay vs vector-only.	**Moderate** – need to integrate a text indexing library (Whoosh or similar). Effort to index all docs (maybe at startup or as they are ingested). Then implement query merge logic. Possibly ensure Chinese text is tokenized for BM25 (we must incorporate a tokenizer or use character bigrams).	License: Whoosh (BSD) or other OSS libraries are fine. Offline: yes, runs locally. We must maintain an index; need to consider index rebuild when new docs ingested – we can update incrementally. Merging can be done by simple union and re-sorting (maybe weight vector score and BM25 score, or just take union and rely on reranker later).

Strategy	Quality Impact	Latency Impact (CPU)	Implementation Effort	Notes
MMR Diversification	**Higher result diversity** – helps ensure that if multiple top results are very similar, we instead show something different among them. This can increase the chance that at least one of the top-K is truly relevant (thus can improve Recall@K slightly). It mainly improves user experience by covering different sub-topics.	**Negligible latency** – computing an extra similarity matrix among K results (K ≤ 10 or 20) is microseconds. Selecting via MMR is trivial compared to model inference. Overall pipeline impact ~0.	**Low** – implement a small function to reorder a list of results given their embedding vectors. We already have vectors for candidates; just do a couple of dot products and sort.	We should choose a parameter (λ) for MMR (balance relevance vs diversity). Likely λ ~0.7 (favor relevance but some diversity). We might tune this slightly on validation queries if needed. MMR is unsupervised – no license issues, it's an algorithm.

Strategy	Quality Impact	Latency Impact (CPU)	Implementation Effort	Notes
Cross-Encoder Re-rank (small model, e.g. BGE-reranker-base)	**Significantly higher precision** – The cross-encoder can understand nuances and context, often reordering results so the most relevant is at rank 1. This can improve metrics like MRR, precision@1 substantially (in BGE's data, rerankers are "more powerful than embedding model" [26], often +10 points in MRR). Particularly helpful if top vector result wasn't actually the best answer.	**Noticeable latency** – Runs an inference for each candidate. If we re-rank 5 docs with a 110M param model on CPU: roughly ~0.5–1.5 seconds additional. On GPU, this would be <0.2s. We can mitigate by using int8 quantization on CPU (possibly 2× speedup) or limiting to e.g. 5 candidates. Still, worst-case P95 could approach ~2s with this on CPU. We should not enable by default for low-end.	**Moderate/High** – need to integrate a HuggingFace transformer for reranking. Loading the model (110M) and running inference code. Also some engineering to format input ("[CLS] query [SEP] document [SEP]") and get a score. Not conceptually hard, but we must ensure efficiency (e.g. do batch if possible, or at least reuse the model instance).	Model: BGE-reranker-base (MIT license, same as BGE) [26] or similar. We must download this model (300MB or so) – adds to setup. Possibly make it optional. Could also consider smaller cross-encoders like MiniLM cross-encoder (very fast) – but BGE-reranker has benefit of bilingual training [29]. We can start with BGE's to leverage same ecosystem. Possibly use FP16 or int8 to speed up. Use-case: enable when quality is paramount (maybe summarization of important notes at end of day – user can tolerate 2s). Document that enabling this yields better accuracy at cost of speed.

Trade-off Discussion: The **hybrid + MMR** combination offers a clear win with minimal downsides – we gain recall and diversity "for free" in terms of resource impact. It does introduce some complexity in coding but nothing too risky. The cross-encoder reranker is a more heavyweight addition that should be carefully toggled to avoid harming responsiveness on slower machines. However, it aligns with delivering top-notch result quality akin to sophisticated IR systems. Since our project is privacy-first, we can't rely on an external

API for super accuracy, so using a local reranker is the next best option. By making it optional, we accommodate both user types: those who want **fast answers** and those who want the **best answers**.

We should also consider that as models evolve, 7B or larger cross-encoders might become feasible – but for now, a ~100M model is a reasonable compromise. The chosen approach ensures that in the default mode, any added latency is very small (just BM25/MMR), and users with GPUs or willingness to trade some time can flip a switch to get a re-ranker boost.

Overall, this strategy is about **stacking retrieval techniques** to maximize the chances the right info is retrieved. It maintains offline operation (we'll use local libs/models for everything) and adheres to CPU-first by making heavier steps opt-in.

Implementation Steps

1. **Integrate BM25 Index:** Choose a library (e.g. `whoosh` for pure Python, or use `pyserini` with Lucene for a more powerful option). Given simplicity, we might start with Whoosh:
2. Define a schema with fields: `content` (text) and perhaps `id`.
3. On startup or ingestion, build the index: for each document (or chunk) ingested, add a document to the BM25 index (Whoosh allows adding docs one by one to an IndexWriter). Alternatively, if rebuilding from scratch each time is fine (if data not huge), we could rebuild at startup from our persistent store.
4. Store the index files under `./data/index/` for persistence.
5. Ensure Chinese text is tokenized: Whoosh might not handle Chinese well by default (it might treat each character as separate token, which is not terrible actually for Chinese retrieval). We could use a plugin or simply preprocess: e.g. insert spaces between Chinese characters to approximate unigram tokenization, or use a simple word segmenter. For initial version, a straightforward approach is treating each character as a token (this works reasonably for matching).
6. *Effort:* Implementing index build and query ~ **Medium** (1-2 days to get it working, including dealing with tokenization).
7. **Query merging logic:** For a given user query:
8. Use the embedding model to get query embedding.
9. Use BM25 index to get, say, top `N_bm25 = 5` docs (id + BM25 score).
10. Use vector store to get top `N_vec = 5` docs (id + similarity score).
11. Merge: Take union of IDs. We have two sets of scores (maybe normalize them). Simpler: initially, we can just combine and de-duplicate, not worry about score merging, because we will re-rank anyway. Alternatively, we can sort by some weighted sum if not using cross-encoder. But since we plan to re-rank with cross-encoder, the simpler union then re-rank approach is fine.
12. If cross-encoder is off, perhaps do a basic merge rank: for example, any doc that appears in both lists gets a boost. Or we could interleave results. A pragmatic approach: take all unique docs, we have their vector cosine scores and BM25 scores; we can sort by (maybe) `sim_score + alpha * (BM25_rank_bonus)` or so. This might require tuning. However, often it's enough to just combine and let the cross-encoder sort them (in which case skip complicated merging).
13. Output the merged list (likely 5-10 docs) for the next step.
14. *Effort:* **Small** (0.5 day) to implement merging, more if we experiment with scoring.
15. **Maximal Marginal Relevance (MMR):** Implement an MMR function:
16. Input: list of candidate docs with their embedding vectors and an initial relevance score (e.g. we can use the cosine similarity as the relevance estimate).

17. Choose a λ (say 0.7). Start with top doc by relevance. Then iteratively select next doc that maximizes `λ*rel_score(doc) - (1-λ)*max_{already_selected}(sim(doc, selected_doc))`.
18. This yields an ordered subset. We can implement for top K (like select top 5 out of maybe 10-15).
19. This is straightforward – just ensure we have the embeddings (we do from vector search step).
20. Use numpy or simple loops (K small).
21. *Effort:* **Small** (0.5 day including testing on a few examples).
22. Test MMR: e.g., if two docs are nearly duplicates, MMR should pick one and then possibly skip the other in favor of a lower-score but different doc.
23. **Cross-Encoder Reranker Integration:**
24. Choose model: e.g. `BAAI/bge-reranker-base` (covers Chinese & English [29]). Download or let HuggingFace auto-download it (mention in docs).
25. Use Transformers pipeline or model directly: since we just need a score, we can use it as a `SequenceClassification` model (it likely outputs a single regression or logit as relevance).
26. On query, after MMR, take the top K (maybe 5). For each, prepare input like: `query_instruction + query + [SEP] + passage`. Actually, BGE reranker expects input similarly (they fine-tuned with query and doc separated by [SEP]). We'll confirm from BGE docs if needed (the model card likely says it's a cross-encoder).
27. Run each through the model to get a score (or do batch of 5 to leverage any parallelism if possible).
28. Sort candidates by this score (higher = more relevant presumably).
29. This becomes final ranking.
30. Implement caching or reuse of the model: load the model once at startup (if configured on). Possibly load on first query to save memory if not used.
31. Provide a config toggle (ENV or in code) for enabling this. If disabled, skip this step and rely on vector/BM25 ordering (or a simpler scoring).
32. *Effort:* **Medium** (1 day to integrate and test model output correctness).
33. We might want to quantize the model for CPU: we could use `torch.quantization` or load a 8-bit version if available (there are libraries like `bitsandbytes` that allow loading int8). Initially, we can skip quantization for simplicity, but note it for optimization.
34. **Configuration & Defaults:**
35. Add settings in `.env` or config file: e.g. `ENABLE_BM25=true` (default true), `ENABLE_MMR=true` (default true), `ENABLE_RERANK=false` (default off for CPU).
36. Or a combined “retrieval mode” that can be “basic” vs “enhanced”.
37. These can be toggled without code changes by advanced users.
38. Our documentation will mention that by default, we do hybrid+MMR, and that cross-re-ranking can be enabled via setting X (with note about performance).
39. **Documentation Updates:**
40. In `docs/architecture.md`, update retrieval pipeline description to multi-step. Perhaps include a small diagram: Query -> [BM25 & Vector] -> Merge -> MMR -> [Cross-Encoder] -> Final results.
41. Explain each component’s role in plain terms (e.g. “we first find relevant notes using two methods – semantic and keywords – to maximize coverage, then remove duplicates and optionally fine-tune the ranking with a second AI model for relevance”).
42. `docs/requirements.md`: mention that enabling reranker might require more RAM/CPU and provide guidelines (like “recommended to use if you have a GPU or don’t mind ~2s response time for complex queries”).
43. If using Whoosh or Pyserini, mention the dependency and any installation notes (Whoosh is pure Python, no extra steps; Pyserini might need Java – likely we go with Whoosh to avoid complicating offline setup).

44. Add to FAQ: "How do I improve the accuracy of search results?" – answer: we use advanced hybrid search; if needed, you can enable an even more precise reranker via config.
45. **Testing & Evaluation:**
46. Ensure that when all features are on, the results returned truly improve (see evaluation plan below).
47. Test fallback scenarios:
    - If BM25 index is not built (maybe first run), ensure the system doesn't break – likely we should build at startup. Or if a query comes in during ingest, vector results at least still work.
    - If the reranker model isn't available, handle exception (log warning, continue without reranking).
    - Multi-thread: if multiple queries come concurrently, ensure our BM25 search and reranker usage are thread-safe (Whoosh can search from multiple threads if we use a shared Index, I believe it's okay; reranker model can be run sequentially – we might want a global lock around it or use batch).
48. Check performance:
    - Confirm that for a typical query, BM25+vector+MMR doesn't slow things noticeably (<50ms overhead). Use a timer in dev mode.
    - If reranker is on, measure the response time to ensure it's within acceptable range (if too slow, consider reducing K or advising user).
49. *Note:* Implementing BM25 also introduces the need to keep the index updated on new ingestions. For simplicity, we can either:
    - Rebuild the entire index whenever new data ingested (fine for small data, could be a bit slow for large but manageable if incremental ingestion is not frequent).
    - Or use Whoosh's incremental adding (which is supported). We just have to maintain the IndexWriter open or reopen index each time to add docs. We'll do either depending on ease (likely, open an index writer each time ingestion occurs and add docs then commit).
50. Workload distribution: the same engineer implementing vector DB can implement BM25 integration, since it touches similar parts. Reranker might be done by someone comfortable with model integration.

This plan will likely take ~3–4 days of work (index integration 1–2 days, reranker 1 day, testing/tuning 1 day). We prioritize hybrid+MMR first (which could be done in 1–2 days and deliver immediate improvement), then add reranker (another 1 day) possibly in the next sprint if time is short.

Evaluation Plan

We will evaluate the upgraded retrieval pipeline on both **synthetic tests** and **benchmark datasets**: - **Functional Tests:** Create specific scenarios to ensure each component works: - Query containing a rare keyword that only one document has: verify that with BM25 enabled, that document is retrieved (even if embedding didn't catch it). E.g., if one note contains "COVID-19" and query is "COVID", the BM25 should surface it even if embeddings might not strongly link short "COVID" query. - Queries where multiple top vector results are very similar: e.g., two duplicate or near-duplicate docs both rank high. Verify that with MMR, the final results include at least one other document. We can simulate this by indexing the same text twice and one different text third, and see that MMR will pick one of the duplicates and then the different one. - End-to-end, feed the final top-K into our LLM response generation and qualitatively see if answers get better (subjective but we can check if the correct info was present). - **Benchmark Evaluation:** Use a standard retrieval evaluation: - For English, we could use a subset of **BEIR** (e.g. the Quora or SciFact dataset). For Chinese, use a subset of **C-MTEB** retrieval dataset (like DuReader or a smaller QA set). Since our system is local, we can curate a few hundred Q&A pairs as ground truth. - Measure Recall@K (K=5 or 10) and **MRR@10** for: 1. Original vector-only retrieval. 2. Hybrid (BM25+vector) + MMR. 3. Hybrid + MMR + cross-encoder rerank. - We expect to see: - Hybrid vs vector-only: improved recall. For example, if vector-only recall@5 was 75%, hybrid might raise it to ~80-85% because some queries where vector missed now succeed. MRR should also improve a bit, as relevant docs might appear slightly higher rank. - Reranker vs no reranker: likely big jump in MRR (since it often puts the correct doc at rank 1 if it was in the top 5 somewhere). We might see MRR@10 improve significantly (possibly 10-20% relative). Recall@5 might stay similar (since recall is about whether it's in top 5 at all, and cross-encoder doesn't add new items, just reorders – though if we consider top-K after rerank, maybe we pick a smaller K but still). - These metrics will let us quantify the benefit. If results align with expectations (e.g. cross-encoder giving clear gains), that justifies its inclusion. - Also track "fail cases" – queries where the improvements didn't materialize, to investigate (maybe poor tokenization for BM25 etc.). - **Latency and Resource Monitoring:** - Measure how query latency changes with the pipeline: *Time each stage.* For instance, in a test with 1000 docs: - Embedding calc: ~X ms (baseline). - BM25 search: Y ms. - MMR: ~ negligible. - Cross-encoder (if enabled): Z ms for K docs. We aim for: - BM25+MMR overhead < 10% of total. Likely BM25 search on 1000 docs is ~2-3ms, which is fine. - Cross-encoder step: if K=5, measure that specifically. On a CPU (no AVX), maybe ~200ms per inference for ~128 token pair -> ~1s for 5. We will measure on target hardware. If it's, say, 1.2s, and base embed took 50ms, total ~1.25s. That might be okay for non-real-time contexts like summarization queries. On a GPU, this should drop to ~0.2s or less (we can test on a modest GPU). Document these numbers: - Without rerank: e.g. P95 latency 300ms (embedding + BM25). - With rerank (CPU): maybe 1300ms P95. - With rerank (GPU): maybe 400ms P95. These will inform what default to choose. If on typical user hardware (say 8-core CPU) it's <1s, maybe enabling by default is okay. If it's >2s, probably leave it off by default. - Memory: Check that adding BM25 index doesn't blow memory: Whoosh index for 1000 docs might be a few MB. BGE-reranker model will take ~0.5GB in RAM when loaded. That's significant – we should note that. If memory is an issue on small machines, that's another reason to keep it optional. We'll monitor memory via `psutil` after loading reranker, and during queries, to see any spikes. - **Guardrail Evaluation:** - Confirm that with cross-encoder off, the results are still reasonable (they should be at least as good as before, likely better due to hybrid). We want to ensure that even in "fast mode" the user is already seeing improvements. - If cross-encoder is on, ensure that it never *worsens* the ranking clearly. Cross-encoders can occasionally be misled, but generally they're better. We will verify on our test queries that it doesn't, for example, drop a very relevant doc out of top-K (which can only happen if it gave it a low score). - Evaluate Chinese queries specifically through the pipeline: confirm our BM25 approach with Chinese tokens actually works (likely we might need to adjust tokenization if results are weird – e.g. search "人工智能" should find doc containing "人工智能", and if we're splitting into characters, it will). - Possibly test cross-language (English query, Chinese doc) – though our models are bilingual, cross-encoder might handle that if trained (BGE-reranker was trained on Chinese & English but not sure about cross-lingual; at worst, it might still prefer same-language pairs. This is edge; anyway, main use is likely same language). - **User Study (if possible):** Internally, have a few team members query the system with/without reranker and gather subjective feedback on answer quality. See if they notice improvements. Particularly, check if the AI's answers become more accurate or require less manual follow-up due to missing info.

We'll record the quantitative metrics (Recall@K, MRR, latency) in a results table in `docs/requirements.md` or a separate evaluation report. For example:

Setup	Recall@5	MRR@10	Avg Query Time (ms)
Original (vectors only)	75%	0.55	100

Setup	Recall@5	MRR@10	Avg Query Time (ms)
Hybrid+MMR	83%	0.58	120
Hybrid+MMR+Rerank (CPU)	85%	0.70	1200
Hybrid+MMR+Rerank (GPU)	85%	0.70	300

(Illustrative numbers)

Our success criteria would be: **Recall@5 improvement ≥5 percentage points** with hybrid vs baseline, and **MRR@10 improvement ≥0.1** with reranker vs no reranker, which are substantial. If we don't see these, we'd investigate if something's wrong (e.g. misconfigured reranker, etc.).

Additionally, we set a soft latency target: default mode queries < 0.5s P95 on CPU; with reranker (if enabled) < 2s P95 on CPU (and <0.5s on GPU). If our measurements show much higher, we might adjust parameters (like reduce K or further quantize model).

Risks & Mitigations

- **Chinese Tokenization for BM25:** If we don't tokenize Chinese properly, BM25 may break queries incorrectly (either treating each character separately or not distinguishing words). *Mitigation:* Use a simple approach initially (character unigrams for BM25). This will still allow matching of multi-character words, albeit not as precisely as a real tokenizer, but ensures recall (if the query character sequence appears, it will match). Alternatively, integrate `jieba` for Chinese segmentation (BSD license) – it's a small dependency. We can test with and without. If using `jieba`, we can custom integrate it into Whoosh as an analyzer. We will likely adopt `jieba` to improve Chinese keyword search accuracy (cost is small and it's local).
- **Index Update Consistency:** On new data ingest, if we fail to add to BM25 index (or system crashes after adding to vector DB but before index update), results could temporarily diverge (new docs not appearing in BM25 search). *Mitigation:* Implement ingestion to update both stores in one routine. Possibly use an atomic write: e.g. add to vector DB, add to BM25, then persist both. If crash mid-way, we can reconcile by rebuilding BM25 from persistent vector DB on next startup (which we might do anyway). We should also plan to periodically rebuild or offer a rebuild command if index gets out of sync.
- **Performance of BM25 on larger text:** If documents are very large (e.g. whole pages), BM25 scoring might become slower (because more terms). *Mitigation:* We already plan to chunk documents in ingestion (we aren't feeding entire huge texts at once). Each chunk is moderate size (<512 tokens), so BM25 scoring stays fast. If we notice slowness, we could limit BM25 to the top few thousand term postings via index settings, but likely not needed.
- **Cross-Encoder Model Size & Memory:** Loading the reranker (110M params) uses a few hundred MB memory and might stress low-memory machines. *Mitigation:* The feature is optional. We will by default not load it, so typical usage doesn't incur it. If user enables it on a low-memory machine, we'll document that memory will increase. If memory is extremely constrained (e.g. 4GB RAM total), they probably wouldn't enable it. We could also consider using an even smaller reranker (like a TinyBERT cross-encoder ~20M) if needed, but that might require fine-tuning. For now, we assume those enabling rerank have some capacity.

- **Reranker Latency Outliers:** The cross-encoder on CPU could have long tails (depending on OS scheduling, etc.). *Mitigation:* If latency is a concern, we advise enabling it only when user is not expecting instant answers (like maybe for end-of-day summary queries it's fine). Also, we can reduce K for reranker if needed (e.g. rerank top 3 instead of 5 to cut compute). We will test and choose K to balance quality vs latency. Possibly make K configurable too.
- **Complexity and Potential Bugs:** The multi-stage pipeline is more complex than before, increasing chances of bugs or edge cases (like no results in one branch, etc.). *Mitigation:* Thorough testing as described. Also ensure to handle cases: if BM25 finds nothing (maybe query is blank or stopwords), just use vectors; if vector finds nothing (shouldn't unless corpus empty) still handle gracefully. If both are empty, return no results gracefully.
- **Reproducibility:** Introducing randomness? MMR and cross-encoder are deterministic. BM25 is deterministic. So no issues there, but we need to ensure using same seed or no stochastic processes. It's fine.
- **Additional Dependencies:** We add Whoosh (pure Python, ~MBs), possibly Jieba (Chinese segmenter ~ a few MBs), and the reranker model (download ~hundreds MB). All are offline. *Mitigation:* Note these in documentation. These do not pose license or privacy issues (all local).
- **Tooling Complexity for Users:** Running Weaviate not required now, but running a BM25 doesn't require separate service. We should ensure building the index doesn't require user to do anything (we handle it). Possibly first run might be a bit longer to build index – that's okay.
- **Fallback if Model Missing:** If user enabled reranker but model isn't downloaded and no internet, query will fail. *Mitigation:* Catch that error. Perhaps print "Reranker model not found. Please download or disable reranker." In offline mode, we should guide them. We could also package the small model or instruct to place it manually. Alternatively, if we suspect many will use offline, we might distribute the 110M model in our release (though that's heavy). Better to mention in readme how to pre-download models if completely offline (Task 11 covers offline packaging).
- **Evaluation metric disagreements:** Possibly, the reranker might slightly reduce recall at fixed K if it demotes something out of top-K. But since K is small and we set it, it's fine. We will keep K after rerank the same as before (meaning if reranker shuffles order but all were already in top-K from union, recall doesn't drop).
- **CJK Full-Width vs Half-Width issues:** A minor BM25 nuance: ensure that if query uses different form of characters or case, we handle appropriately (we'll likely lowercase and such for English; Chinese has no case but maybe just ensure no stray punctuation). We rely on Whoosh analyzer for that.

By acknowledging and preparing for these potential issues, we can implement the retrieval enhancements confidently, knowing how to address them if they arise. The result should be a more robust and effective retrieval component for Project Epoch, directly improving the quality of answers the system provides to the user.

References

- 【2】BGE-M3 Model Card – *Recommendations: hybrid retrieval (embedding + BM25) and cross-encoder re-ranking yield higher accuracy* [10] [11]
- 【25】BGE Model Card (HuggingFace) – *Announcement of new cross-encoder models BGE-reranker-base/large, "more powerful than embedding model" for top-K re-ranking* [26]
- 【25】BGE Model Card – *Metrics snippet: shows bge-reranker-base vs large performance across tasks (implies such models boost re-ranking scores)* [27]
- 【26】Medium (Anton Dergunov) – *Definition of MMR: retrieves documents that are relevant* and *diverse* [24] , *MMR balances relevance and redundancy in re-ranking* [25]

- 【46】Reddit (LocalLLaMA) – *Discussion about Qwen-7B: mentions English vs Chinese performance and large vocabulary, relevant because cross-encoder BGE-reranker is bilingual but we consider cross-lingual aspects* [30]
- 【22】Reddit (r/vectordatabase) – *User feedback on mixing methods: e.g. switched from Chroma to FAISS (implies pure vector vs others), tangentially highlights performance but also the importance of correctness* [18]
- Academic IR literature – (e.g. "On the Effectiveness of Hybrid Retrieval" – demonstrates hybrid > single) **(No direct snippet, background knowledge)**.
- Whoosh documentation – (BM25 ranking, usage of analyzers for different languages) **(No snippet)**.
- BAAI FlagEmbedding GitHub – (Examples of using BGE reranker with FlagEmbedding, confirming usage) [31] (calls `compute_score(['query','passage'])`).
- *Maximal Marginal Relevance original paper* by Carbonell & Goldstein (SIGIR 1998) – **concept referenced, no snippet**.
- *Pyserini toolkit documentation* – (illustrates ease of combining BM25 + vectors in practice, as done in research) **(No snippet)**.

Task 4: Evaluation Baseline Design

Summary

- **Metrics for Latency:** We will measure response times at **median (P50) and 95th percentile (P95)**, which capture typical vs. worst-case latencies. P50 gives the average user experience, while P95 ensures we account for occasional slow queries (e.g. longer inputs or heavy re-ranking). These will be measured for key operations: embedding generation, retrieval query end-to-end, and possibly full Q&A pipeline latency.
- **Metrics for Retrieval Quality:** We will use standard Information Retrieval metrics like **Recall@K** and **Hit@K** (which, in our context, are equivalent since typically each query has at least one relevant doc). Recall@K means the fraction of queries for which at least one relevant document is present in the top K results. **MRR (Mean Reciprocal Rank)** or **Precision@1** are also important to measure how often the top result is the correct one (especially after introducing re-ranking). These metrics give us objective measures of how well the memory search is working.
- **Datasets:** We will leverage subsets of known benchmarks, focusing on those small enough to run locally and relevant to our use cases:
- **MTEB/BEIR (English):** e.g. a small retrieval dataset like **Quora Question Duplication** (contains questions and duplicate flags), or **FiQA** (financial QA) or **SciFact** (scientific claims -> evidence). We can use a subset (~100 queries and corresponding corpus) to test English embedding performance.
- **C-MTEB (Chinese):** e.g. **DuReader-Retrieval** (Chinese web QA retrieval with 90k queries, which is large, but we can sample e.g. 500 queries and a few thousand passages to create a mini-benchmark). Or a smaller set like Chinese FAQ data if available. We need at least one Chinese evaluation to ensure our changes benefit Chinese equally.
- We may also craft a **toy dataset from the user's actual domain** (if allowed) – e.g. if Project Epoch is like a personal notes assistant, create a synthetic set of notes and queries about them to validate performance qualitatively and quantitatively.
- **Reproducibility & Light-weight:** The evaluation should be runnable entirely offline on consumer hardware. We'll avoid very large corpora to keep runtimes short. For example, using ~1k documents and ~50 queries for each test should be enough to detect improvements, while running in seconds

to a minute. We will automate these via scripts (perhaps integrating with HuggingFace's `mteb` library for ease of loading tasks).
- **Reporting & Thresholds:** We will define **target thresholds** to guide improvements. For instance, aim for *Recall@5 ≥ 80%* on our evaluation set (meaning the system usually finds a relevant memory chunk in the top 5). For latency, define something like *P95 end-to-end query latency ≤ 1000 ms on CPU* (or specific thresholds per hardware tier – see Task 12). These targets will be documented in requirements and serve as KPIs. The evaluation results will be reported as part of our CI or manual test and compared against these targets to track progress or regressions.

Recommendation

We propose implementing two evaluation scripts – one for latency and one for retrieval quality – that can be run locally to establish baseline performance and track improvements over time: - `scripts/eval_latency.py`**:** This script will simulate user queries and measure timing for each stage of processing. It will output metrics like P50 and P95 latency for embedding generation and overall API response. The recommendation is to use Python's `time` or `timeit` to record durations and then compute distribution percentiles. We should test on representative hardware (e.g. a typical laptop CPU, and optionally with a GPU if available) to set realistic expectations. The script might generate dummy data (random text of various lengths) or reuse real data from the retrieval evaluation for consistency. Output will be printed as a table or JSON (for CI parsing) with metrics. We will incorporate multiple runs to get stable stats (e.g. run 100 queries and take stats). This gives us a reproducible way to gauge performance with each new release or optimization. - `scripts/eval_retrieval.py`**:** This script will load a small evaluation dataset (likely from HuggingFace Datasets or local files) containing queries, documents, and relevance labels. It will then run our retrieval pipeline (embedding + search) to get top-K results for each query, and compute Recall@K, MRR, Precision@1, etc. We can write this from scratch or utilize existing libraries like BEIR's evaluation module if it's not too heavy. The script will output these metrics and optionally per-query results for analysis. - We recommend focusing on K = 5 or 10 since that's typically what we feed to the LLM. So, e.g., Recall@5 is critical. If we see a low recall, it means the LLM might not get the needed info. - We'll compare metrics for different configurations (we can include flags to toggle e.g. vector-only vs hybrid vs rerank to measure their effects, as in Task 3). - **Datasets to use:** We will assemble at least: - *English set:* possibly **Quora Dup Questions** (e.g. treat one question as query, retrieve duplicates). Or **NFCorpus** (small corpus ~3k docs, 300 queries, used in BEIR). - *Chinese set:* **DuReader retrieval** has a large corpus, but we can sample a subset of passages (e.g. 10k) and ~50 queries from dev set. Alternatively, if BAAI's C-MTEB has a packaged smaller dataset (like a subset focusing on QA). - If direct usage of these is complicated, we can manually create a mini-benchmark: For example, take 100 QA pairs from Wikipedia in Chinese and treat them as: query = question, relevant = that wiki paragraph. - We'll include the instructions in the script on how to obtain the data (or we can host a small JSON of the sample in our repo to avoid external dependency). - **Automating and documenting:** We will integrate running these evaluations as part of our release checklist. Possibly even in CI (though CI might not have the models or data – we can skip heavy parts in CI but run it locally). The results will be documented in `docs/requirements.md` under a new section "Baseline Quality and Performance". For example: "On MTEB SciFact subset, Recall@5 = 82%. On a Chinese QA set, Recall@5 = 78%. Median single-query latency = 150 ms (no reranker) on CPU." These benchmarks set a baseline for future improvements. - We will set initial **target thresholds** as goals: e.g. *Recall@5 ≥ 80%* in our combined test, *P95 latency ≤ 1s on CPU-only mode* (adjusted if needed). These targets are derived from what we believe a good user experience requires (ensuring most relevant info is retrieved, and response feels snappy). As we refine models or pipeline (e.g. switching embedding model, adding reranker), we aim not to drop below these targets. If we do, that flags a potential regression needing attention. - **Local reproducibility:** Emphasize that anyone can run these scripts with `pip install -r requirements-dev.txt` (if needed for eval libs) and by having the models downloaded. Provide clear README instructions: "To evaluate retrieval quality, run `python scripts/eval_retrieval.py --model <path> --data <path>`." Ensure the script prints out results clearly. - Over time, we can expand these tests or plug in more tasks (e.g. if MTEB releases a unified library, maybe integrate our system in it for standardized benchmarks). For now, we keep it light and targeted.

By following this plan, we'll establish a solid feedback loop: as we implement changes (Tasks 1-3, etc.), we can quantify their effects on both speed and accuracy. This data-driven approach will guide us to tune parameters and pick the right trade-offs (like we did in Task 3's design). It also provides transparent metrics to share with stakeholders or community (proving progress).

In summary, **we will implement a lightweight, fully local evaluation harness** focusing on retrieval performance (accuracy and latency) using established metrics and small representative datasets. This will serve as our baseline and regression test suite moving forward.

Options & Tradeoffs

Aspect	Options	Rationale / Trade-off
Latency measurement method	1) Instrument the running system (e.g. time actual API calls in a loop). 2) Isolated function benchmarking (call embedding model, etc. in a test harness).	We choose to create a standalone `eval_latency.py` to call the embedding and retrieval functions directly in a loop, to avoid overhead of HTTP. This isolates system performance. Timing via Python should be accurate enough (microsecond precision). We'll ensure to exclude initialization time and focus on steady-state. Using actual API endpoint calls (HTTP) could include network stack overhead and be noisy; since we want the core processing time, we avoid that.
Benchmark datasets	1) Use BEIR via HuggingFace/BEIR library (which provides standard splits and evaluation code). 2) Create custom small datasets manually.	Using BEIR/C-MTEB is good for standardization, but those datasets can be large and their evaluation frameworks might require internet to download. We prefer manual curated subsets to keep it offline and fast. We'll manually select a few tasks and possibly store needed data (within license allowances). The trade-off is slightly more work to prepare data, but ensures the eval is quick and offline.
Metrics to compute	1) Recall@K, MRR, Precision@k. 2) nDCG (normalized discounted cumulative gain).	Recall@K and MRR are intuitive for our QA-style retrieval (binary relevance). nDCG is more for graded relevance (not needed if our data is just relevant vs not). We stick with Recall and MRR primarily. Precision@1 or Hit@1 is essentially MRR's primary component (whether the top result is relevant). So including MRR covers that. We won't add more complex metrics unless needed, to keep script simple.

Aspect	Options	Rationale / Trade-off
Evaluation script integration	1) Standalone scripts in `scripts/`, run manually. 2) Integrate into an automated test or CI.	Initially, these will be standalone (option 1). The user or developer runs them as needed. We can't easily run heavy eval in every CI (lack of GPU, time constraints, etc.). However, we might integrate a small sanity test (like 1-2 queries) in CI to catch gross regressions. The full eval with e.g. 50 queries might be done pre-release manually. We will document how to run it. The trade-off: manual step means we must remember to run it, but it gives flexibility. Possibly in future, we can include a nightly job or something if needed.

Our chosen approach optimizes for **ease of local use and meaningful metrics** rather than strict adherence to a benchmark suite. We want numbers that matter to our user's experience and that we can improve upon.

One consideration: since all evaluation is local, any differences in hardware can affect latency results. We will specify the environment (CPU model, etc.) when reporting. For quality metrics, hardware doesn't matter. We will also specify model versions used (embedding model, etc.) to contextualize results. This helps keep track if we swap models later (we should re-evaluate to update baselines).

Implementation Steps

1. **Prepare Evaluation Data**:
2. Select and obtain the datasets. For example, for English: use `BEIR/quora` via huggingface or their GitHub (quora duplicate questions). We can script downloading it or include a pre-saved small JSON. For Chinese: possibly sample DuReader retrieval. We might need to download DuReader dev set from its repository (we can automate via requests if available).
3. Convert these into a uniform format: e.g. create `data/eval/english.jsonl` and `data/eval/chinese.jsonl` containing lines like `{"query": "...", "positive_ids": ["id1","id2"]}` and a corpus file mapping ids to text. Alternatively, our script can use BEIR's API to load a subset and then use our system.
4. For manageability, we might embed the queries and corpora in Python lists in the script or in small JSON files in the repo (assuming license allows a small subset distribution – Quora is public, DuReader data is available for research).
5. Document in the script header where the data comes from and how it was prepared (for transparency).
6. *Owner:* Evaluation engineer / team member. **(Medium)** – likely a day to gather and format data appropriately.
7. **Implement `eval_retrieval.py`:**
8. This script will load the dataset (from a file or directly via code).
9. Initialize our embedding model and vector store (maybe using the same code as our app, or by directly calling the relevant functions in `services/api`). We might need to instantiate the embedding model and use an in-memory storage of all docs. For simplicity, we can bypass our API and directly compute similarity (like use FAISS or brute-force in the script) to evaluate model recall.

But since we have moved to Chroma in Task 2, maybe we can instantiate a Chroma in-memory, index all test docs into it, then perform queries. This actually tests our pipeline as is, which is good.

10. Alternatively, for initial baseline, directly compute cosine similarity between query embedding and all doc embeddings (embedding all docs first). That might be simpler to implement and reliable for evaluation.
11. Compute metrics: For each query, get the ranked list of docs. Check if any of the known relevant docs (positive_ids) are in the top K. Record hits. Compute recall@K = (#queries with hit in topK) / (total queries). Compute MRR: if a relevant doc is at rank r, reciprocal = 1/r, take average over queries. If none found, reciprocal=0. Use the standard formula.
12. Print results. Possibly break down by dataset (English vs Chinese).
13. Optionally, implement flags to toggle use of hybrid or reranker if integrated, to see differences. But initially baseline is just current state (embedding-only for baseline). After Task 3, we can reuse same script to measure improvements.
14. *Owner:* Same person. **(Medium)** – 1 day to code and test on the sample data.
15. **Implement `eval_latency.py`:**
16. Load or instantiate the embedding model (and any pipeline pieces like cross-encoder if needed).
17. Option 1: directly measure the API endpoints – but starting up the whole API for measurement is complex. Instead, we measure key functions:
    - Time to embed a query of typical length (like 20 words English, or 20 chars Chinese).
    - Time to perform a vector search among X docs (we can use FAISS or our Chroma index).
    - If reranker is used, time to rerank top-K.
    - We might simulate typical flows: e.g. a user query triggers 1) embedding, 2) retrieval, 3) rerank (if on).
    - Sum these times to approximate full response time, or explicitly measure a combined function if easier.
18. To capture distribution, run e.g. 100 random queries (could sample from our eval queries or generate random text if needed). Make sure to include some long queries to see worst-case (e.g. simulate a long user question). We might vary input size: short vs long query to see effect on embedding time.
19. Use Python's `time.perf_counter()` around code sections. Collect durations in a list. Compute P50 and P95 via `numpy.percentile`.
20. Report output. Possibly format as:

```
Embedding (768-dim) – P50: 45 ms, P95: 80 ms
Retrieval (Chroma top-5 of 10k docs) – P50: 10 ms, P95: 15 ms
Re-rank (cross-encoder 5 docs) – P50: 200 ms, P95: 250 ms
Total pipeline – P50: ~260 ms, P95: ~350 ms
(Hardware: Intel i7-9700K, no GPU)
```

21. If GPU is present, optionally measure on GPU too to show difference (maybe a flag to pick device).
22. *Owner:* Possibly the same person or a different one if splitting tasks. **(Medium)** – ~1 day to implement and refine measurements.
23. **Set Target Thresholds & Document:**
24. Based on initial baseline results, decide realistic yet ambitious targets for P0:
    - e.g. If baseline recall@5 is 70%, maybe set 80% as goal after improvements.
    - If baseline P95 latency is 500ms on CPU (just an example), maybe target 400ms by certain optimizations (or maintain under 1s with reranker).

25. Include these in `docs/requirements.md` under a "Key Performance Indicators" or similar section. For example:
    - *Quality:* "System should achieve ≥80% Recall@5 on our evaluation set (ensuring relevant info is retrieved)."
    - *Latency:* "On consumer hardware (4-core CPU), median query latency should be < 0.5s (and 95th percentile < 1.5s) without neural re-ranking. With re-ranking enabled, 95th percentile should remain < 3s."
    - *Resource usage:* (if we include KPI for memory/cpu, see Task 12).
26. These numbers will guide us; they may be updated as we improve.
27. Also document in architecture or a separate `benchmark.md` the detailed results for current version so we have a baseline snapshot. This includes environment info and config used (embedding model name, etc.).
28. **Integration and Continuous Use:**
29. Make sure running the scripts doesn't require internet (we'll package or instruct to place needed files). If using huggingface datasets, we might need an initial download. We can do that once and cache under `./data`.
30. Possibly add a Makefile target or npm script (if we have any) like `make eval` to run these quickly.
31. Encourage contributors to run these after major changes (maybe note in CONTRIBUTING.md).
32. Consider adding a very small smoke test to CI: e.g. run eval_retrieval on 3 queries and assert recall is above e.g. 0 (meaning at least something returns, to catch catastrophic failures).
33. Over time, we can extend evaluation to other modalities (if we add ASR or VLM, then measure WER or image caption accuracy as needed in future tasks). But for now, focus on text memory retrieval.

By implementing these steps, we ensure we have a robust baseline measurement of our system's performance and a way to quantitatively track improvements from tasks 1–3 (and beyond). This is crucial for a project like Epoch to validate that changes are beneficial and to demonstrate progress to stakeholders (like in investor brief or README, we can proudly state metrics).

Risks & Mitigations

- **Choosing Unrepresentative Data:** If our evaluation sets are not similar to actual user data, we might optimize for the wrong thing. *Mitigation:* We try to pick diverse QA-style sets (e.g. Quora covers conversational questions, SciFact covers factual claims, DuReader covers Chinese search queries). Additionally, we might incorporate a few real user-like queries manually to sanity check (e.g. create dummy personal notes and queries). We'll be cautious interpreting results – they show trends but absolute numbers might differ in real use.
- **Small Sample Size:** With small eval sets (~50–100 queries), metrics could be noisy. *Mitigation:* We'll treat them as directional. We can increase the sample if needed (maybe to a few hundred) while still manageable. We could also aggregate results from multiple mini-datasets for a more stable average. Also, we won't overfit to these few queries; they are just a check.
- **Automation vs. manual runs:** Since we can't fully automate heavy eval in CI, there's a risk we forget to run them regularly. *Mitigation:* Incorporate it into the release process checklist ("run eval scripts and record results"). Possibly set up a GitHub Action that runs nightly on a runner that has our models cached (though without a GPU, but can still do small runs on CPU as a regression check). This is optional if time permits. At least ensure core devs know to run these after major changes.
- **Measurement accuracy for latency:** Python timing might have slight overhead or interference. *Mitigation:* We will run enough iterations to average out noise, and ensure no other heavy processes

running during test. We might pin threads for the model (set `OMP_NUM_THREADS` to control variance). Also, we'll discard first few runs to avoid initialization bias (warm up the model).

- **Licensing of evaluation data:** We must ensure we have the right to use and potentially commit subsets of data. *Mitigation:* Quora and DuReader are released for research; we're using them for evaluation, which is typically allowed. We will include attribution (like link to original) in our docs. If needed, we won't include actual data in repo but rather instructions to download. To keep offline, perhaps we include a script to automatically download and slice the data, instead of shipping it. We will be careful here.
- **Comparability:** If we change models (embedding or others), baseline results shift. *Mitigation:* Always note the version of models used for each reported metric. This ensures if we adopt a new embedding model, we re-evaluate and establish a new baseline, rather than mistakenly thinking something else changed quality. We consider keeping a change log of metrics.
- **Focus on metrics vs. actual user satisfaction:** Our chosen metrics (Recall@K, etc.) are proxies. It's possible to optimize those but still not solve user's problem fully (e.g. maybe a relevant doc is retrieved but LLM doesn't use it well). *Mitigation:* We should also do end-to-end tests (where we check if the final answer was correct). That's more complicated to automate because it requires comparing generated answer to ground truth answers – which we could do for e.g. a QA dataset with known answers. Possibly an extension: after retrieval, feed top docs to a small QA model and see if it produces correct answer. That's a whole separate evaluation (LLM quality). For now, we isolate retrieval quality, assuming that if relevant info is retrieved, our LLM has a chance to use it. We document that assumption.
- **Hardware variations for latency:** The thresholds we set might not hold on all user machines (some may have older CPUs). *Mitigation:* We set targets that are moderate and specify context (like on a modern 4-core CPU ~3.0GHz). We can include guidance like "on lower-end devices, expect proportionally slower times." The key is, if our improvements maintain or improve these metrics on a reference machine, they likely benefit across the board (even if absolute values differ).

By addressing these, we ensure our evaluation is as reliable and useful as possible for guiding development.

References

- 【12】Zilliz AI Models (BGE) – *Provides C-MTEB benchmarks (35 datasets including retrieval)* [5] (useful to pick Chinese tasks)
- 【27】EvalScope C-MTEB – *Describes Chinese benchmark tasks (35 datasets, 6 categories including retrieval & reranking)* [32] (we focus on retrieval portion).
- **BEIR paper (Thakur et al. 2021)** – defines Recall@K, nDCG, etc. and uses 18 datasets for retrieval. (No direct snippet here, but standard knowledge).
- **MTEB (Muennighoff et al. 2022)** – describes the Massive Text Embedding Benchmark including multilingual tasks. (We rely on known usage, no snippet).
- 【48】Milvus security (embedding encryption) – not directly relevant to eval, but ensures we consider privacy even in eval (e.g. all data is local).
- **HuggingFace Datasets** – to potentially load Quora or SciFact. (No snippet, just reference to use their API if network allowed).
- **Python Perf docs** – guidelines on accurate time measurements (we use `time.perf_counter` as recommended).
- **Project Requirements** – we will add our findings in `docs/requirements.md` accordingly.

Task 5: Local ASR (Whisper) Process Design

Summary

- **Whisper.cpp vs Faster-Whisper:** Both enable local ASR with OpenAI's Whisper models but have different optimizations. **Whisper.cpp** is a C++ implementation that runs entirely on CPU (and can use quantized models down to 4-bit) – great for ultra-portability (even Raspberry Pi) but typically slower than optimized GPU code. **Faster-Whisper** uses CTranslate2 to accelerate Whisper (with GPU support and int8 on CPU). On a CPU, faster-whisper's int8 inference is ~2× faster than the original PyTorch, and on GPU it's much faster. For example, transcribing a 30s clip: whisper.cpp in int4 might take ~ real-time or slower on CPU, faster-whisper int8 might be faster than real-time on a decent CPU [33]. Both support the same model weights and achieve the same accuracy (as they use the same underlying model). So, if GPU is available or even for multi-threaded CPU, **faster-whisper** is preferable for speed. Whisper.cpp shines when minimal dependencies or running on ARM devices (like we can compile it on mobile) – perhaps less relevant for our typical PC use. We can consider using faster-whisper by default (it has a Python API) and keep whisper.cpp as an alternative for edge cases.
- **Accuracy (English vs Chinese):** Whisper models are known to be strong in both English and Chinese. According to OpenAI's paper and subsequent analyses, **Whisper small** (~244M) achieves around ~10-15% WER on English and ~15-20% on Chinese, while **Whisper base** (~74M) is around 15-20% WER English and 25-30% Chinese [34] [35]. So, small model is significantly more accurate, especially for Chinese. The trade-off is speed: base is ~3× smaller and thus faster than small (and uses less memory). For our use (transcribing personal voice notes, possibly a few minutes long), accuracy is quite important (we want correct transcripts for later search). **We recommend defaulting to the** Whisper small **model for better accuracy** – in tests, Whisper small cuts error rates ~30% lower than base on many tasks [36]. It also tends to handle Mandarin tone differences and English names spelled out better than base. However, if running on pure CPU without optimization, small might be borderline real-time or slower on longer audio. That's where faster-whisper optimization helps. If someone's on a low-power machine, we could allow them to switch to base model for speed. But baseline: small model for best quality, optionally base for faster/low-resource scenarios.
- **Segmentation & Punctuation:** Whisper models output transcriptions with punctuation and sentence casing automatically (the medium/large models do this extremely well; small/base somewhat well). They segment audio into **segments with timestamps** (Whisper generates timestamps every few seconds when it detects pauses [37]). So, we get sentence-like segments by default. For diarization (speaker separation), Whisper itself does **not** label speakers. We would need to separate speakers through another method (below). But in terms of basic segmentation (deciding where one sentence ends and next begins), Whisper does a good job thanks to its training – it will include periods, commas, etc., making the transcription quite readable. We likely do not need an external segmentation algorithm; we can trust Whisper's segmentation and punctuation. We may just need to join segments appropriately for our memory store.
- **Speaker Diarization & VAD:**
- *Speaker Diarization:* This requires identifying speaker changes. Neither Whisper.cpp nor faster-whisper provide speaker labels. A common approach is to run a separate **Voice Activity Detection (VAD)** to get speech segments, then run a **speaker embedding model** (like Silero's speaker embedding or Pyannote) to cluster segments by speaker. This is heavy and possibly beyond initial

scope (pyannote models are large). A simpler approach: If user specifically needs diarization (like distinguishing speakers in a meeting recording), we could integrate something like **WhisperX** or **whisper-diarize** which combine Whisper with NeMo's speaker diarization [38]. But this is complex for P0. We likely position diarization as a future/optional feature. For now, we aim to accurately transcribe what was said, possibly segmenting by long pauses (like paragraphs), but not label speaker identity.
- *Voice Activity Detection (VAD):* This is useful for both splitting long audio and filtering out non-speech. **Silero VAD** is a lightweight model (int8, ~1.4M parameters) that can detect speech segments with ~0.95 F1. Faster-Whisper already **integrates Silero VAD** to remove long silences or non-speech parts [33]. By enabling `vad_filter=True`, we can automatically skip silent intervals > some threshold. This is great for long recordings: it ensures Whisper isn't wasted on silence and can avoid some mistakes (like hallucinating text in background noise). So, recommended: use Silero VAD as provided in faster-whisper (or externally if we use whisper.cpp, we'd run VAD separately). This will give cleaner transcriptions and split on pauses.
- **CPU vs GPU Trade-off:**
- On CPU, Whisper small (244M) in real-time: with faster-whisper INT8 and multi-threading, it can approach or exceed real-time on a modern multi-core CPU (for example, faster-whisper int8 on 4 cores might transcribe 1 second of audio in ~0.5s). Whisper base (74M) will be about 3x faster (less to compute).
- On GPU (even a 4GB GPU), Whisper small runs very fast (faster than real-time easily, maybe 0.1x real-time on something like RTX3060). So if a GPU is present, using small or even medium is fine. But medium (769M) might be borderline on 4GB (likely needs 5GB in float16, but int8 might fit). Medium has even better accuracy (~5% WER English, ~8% Chinese [35]), but given VRAM constraints, small is a safer default. If a user has a strong GPU, they might themselves choose medium or large – we can allow that via config.
- So:
    - **Default**: Whisper small, run via faster-whisper. This yields high accuracy for English & Chinese (estimated ~10% WER on English, ~15% WER on Mandarin which is quite good) [35]. On an average CPU, expect maybe 0.8x–1.0x real-time speed with int8 (so transcribing a 1-minute audio takes ~1-1.2 minutes).
    - **CPU fallback**: If user only has CPU and needs faster, Whisper base can be an option. It's ~2-3× faster but with noticeable accuracy drop in Chinese (maybe 25% WER vs 15%). We will list it as a "speed mode" option. Also, quantization to int8 or int4 can dramatically speed up whisper.cpp on CPU (int4 base might run >2x real-time on high-end CPU). But int4 can degrade accuracy ~<1% abs, which is okay.
- **Memory**: Whisper small FP16 is ~1GB VRAM. Int8 on CPU uses ~500MB RAM. Base uses ~~300MB. These are fine for most PCs.
- **Licensing & Model Availability:** OpenAI's Whisper model weights are available under the MIT License [39]. This means we can use them freely in our app (even commercially). So no issue. Silero VAD is also open-source (MIT). Qwen's ASR not considered; whisper is SOTA open solution.
- **Summary of Recommendations:**
- Use **Faster-Whisper** library to run **Whisper-small** model by default, with `vad_filter=True` (Silero VAD) for cleaner segmentation [33]. This gives good accuracy on both English and Chinese and leverages multiple CPU cores or GPU if available.
- Provide config options for model size ("small" vs "base") and quantization (faster-whisper uses int8 by default on CPU) so that on extremely low-end hardware one can opt for base or even whisper.cpp with 4-bit.

- Note that speaker diarization (identifying speaker changes) is not handled in P0 beyond the punctuation and segmentation that Whisper provides. If needed, we'll mention in docs that multi-speaker audio will be transcribed as a single stream of text (with punctuation but no speaker labels). For future, we can integrate whisperX or an alignment with pyannote for diarization as an advanced feature.
- Ensure the transcription results (with timestamps maybe) are stored if needed, but likely we just need the text for memory. We might not require word-level timestamps (which can be enabled if needed for highlighting audio later, but not necessary for our immediate memory search).

- **ASR Output Utilization:** The transcribed text (with punctuation) will be ingested into our vector store like any other text. If diarization is absent, it might be slightly less readable for dialogues, but still searchable. We may just treat each Whisper segment (which typically is a sentence or phrase up to ~30s) as a separate entry, possibly tagging them with a time or speaker if known. Since we don't have speaker labels, we can at least attach a meta like "Audio X, timestamp 0:00-0:15" if needed, for reference. But from a memory retrieval standpoint, just storing the text content is sufficient for semantic search.

Recommendation

Use **OpenAI Whisper (small model) via the Faster-Whisper pipeline** as the default local ASR solution, enabling it to handle both English and Chinese with high accuracy, and optimize for CPU/GPU usage: - **Accuracy:** Whisper-small strikes the best balance of accuracy and speed for our bilingual needs. It has near state-of-the-art transcription quality, achieving ~92% word accuracy in English and similarly strong performance in Mandarin Chinese (e.g. ~8% error on Cantonese evaluation for medium/large, small is slightly higher but still very good [35]). This means it can transcribe user conversations or voice notes with minimal errors, including proper punctuation and casing. Using the **small** model (as opposed to base) greatly improves Chinese transcription fidelity – critical for our user experience. - **Segmentation & VAD:** We recommend leveraging the **built-in VAD from Silero** (as exposed by faster-whisper's `vad_filter` option [33]) to automatically segment the audio and skip silence. This will break long recordings at natural pauses and eliminate dead air or noise-only sections from being processed. The result is more accurate transcripts (since the model won't hallucinate during long silences) and also more efficient processing (saving time by not running on non-speech segments). Whisper itself will output punctuation and segment boundaries; we trust its segmentation for sentence breaks. Thus, the transcribed text will be fairly readable and can be split into chunks (e.g. sentences or phrases) for memory storage. - **Speaker Diarization:** In this initial version, we will **not perform explicit speaker diarization** (i.e. we won't label "Speaker A:" vs "Speaker B:"). If the user records multi-speaker audio (like a meeting), the transcription will be a single combined transcript with punctuation. This is a deliberate scope choice – full diarization requires additional heavy models (e.g. speaker embedding/clustering). We will note this limitation in the docs. The focus is on capturing the content of what was said. In many personal AI scenarios (like voice notes or single-user dictation), diarization isn't needed. For meetings, it's a nice-to-have which we can add later (perhaps via integration with projects like WhisperX or NeMo's diarization pipeline [38]). - **Performance and Hardware:** With faster-whisper, we can automatically utilize a GPU if available (the library will detect and use `device="cuda"` by default if `torch.cuda.is_available()` and we specify it). On a GPU (even ~4GB), Whisper-small will transcribe much faster than real-time (for example, on an RTX 2060, it can process ~10x faster than real-time in FP16). On CPU, faster-whisper uses multi-threaded INT8 by default [40], which should approach real-time on a 4+ core CPU for small model. We will make this the default mode. - For users on less powerful CPUs (e.g. no AVX, or very low-core counts), we will provide an option to switch to Whisper-base model.

Whisper-base is only ~30% the size of small, and thus roughly 2-3x faster with some accuracy loss (English WER might go from 10% to 15%, Chinese from ~15% to ~25% – still intelligible but not as perfect [36] [34]). This is an acceptable trade if needed for speed. We might label this as a "fast mode" in settings (perhaps selected via an env var or config like `WHISPER_MODEL_SIZE=base` vs `small`). - Additionally, if extreme CPU optimization is required (e.g. running on a Raspberry Pi or similar), users can opt to compile and use **whisper.cpp** with 4-bit quantized models. That's outside typical usage, but our architecture could allow pluggable ASR backends. We will mention that advanced users can do so, though by default we stick to faster-whisper for simplicity and performance on common devices. - **Default Behavior by Hardware Tier:** - **CPU-only (normal PC):** Use faster-whisper with small model int8, multi-threading. Expect ~1× real-time transcription. Memory ~500MB, which is fine. - **CPU low-end:** If we detect <4 cores or user complains about speed, they can switch to base model. We'll document how to do that (and possibly auto-switch if performance is very low, though auto-detection might be tricky). - **GPU present:** Use it to accelerate Whisper – faster-whisper will do FP16 on GPU. The small model will easily run in ~1GB VRAM. If a larger GPU is present (≳10GB), user could even configure medium or large model for even higher accuracy. We can expose `WHISPER_MODEL=medium` as an advanced config, but the default remains small to conserve resources. - **Integration in Project Epoch:** We will incorporate an ASR pipeline stage where audio input (possibly via microphone or file upload) is processed. Steps: 1. Load Whisper model at startup (this might take a few seconds for small model). We'll log a message ("Loading Whisper small model into memory…"). 2. When audio input is received, run VAD-chunked transcription via faster-whisper: this returns segments with text and timestamps [33]. 3. Concatenate or store these segments. Likely we'll assemble the full transcript text. We may keep timestamps if we want to align text to audio for playback features (not immediate need, but storing them in meta is easy). 4. The text is then ingested into the memory store like any other note. We can either store the whole transcript as one chunk (if short, e.g. a voice note) or split by segments if it's long (e.g. a meeting). 5. Attach metadata like `source: "ASR"` or a filename, and possibly speaker if user manually provides (e.g. if it's their own diary entry vs a conversation). 6. The result is that the voice content becomes searchable text in the system. - **Default and Capacity Tiers in Docs:** We'll update `docs/requirements.md` with recommended default: *"Whisper-small model via faster-whisper (for best accuracy on English/Chinese) – requires ~1GB RAM, runs in real-time on CPU"*. Also provide table of alternatives: base model (faster but lower accuracy), large model (much higher accuracy but needs GPU/longer time), etc. Possibly: - *Tier 1 (CPU minimal):* Whisper-base int8 or whisper.cpp 4-bit – minimal memory, fastest. - *Tier 2 (standard):* Whisper-small int8 – default, good accuracy. - *Tier 3 (performance/GPU):* Whisper-medium or large float16 on GPU – highest accuracy if user has powerful machine. - Clarify licenses: all these are MIT, no restriction on usage of transcripts since local. - **ASR Accuracy and Use Cases:** We should highlight in docs that this local ASR is state-of-the-art level (close to Google API quality). Also note limitations: heavy accents or very noisy audio could still cause errors; and that the model might occasionally produce "[inaudible]" tokens or such if it really can't figure something out (though Whisper doesn't actually output "[inaudible]" like some models, it just tries its best). - **Language Handling:** Whisper models are multi-lingual by default (it will detect language at runtime). Ours will automatically detect if audio is English, Chinese, or another of the 99 languages and transcribe accordingly [41]. We should allow that behavior. Faster-whisper returns the detected language code (we can log it). For our focus, likely English or Chinese primarily, but it's nice that others could also work (with small model, some languages not as great, but functional). - **ASR Results Post-processing:** Possibly minor things: we might want to trim trailing whitespace, unify full-width punctuation to half-width for consistency if needed (Chinese output might contain Chinese commas/periods, which are fine). But for search, we embed as-is (embedding model likely can handle punctuation or we can strip punctuation if needed during embedding). We'll likely feed the raw transcribed text into the embedding model for indexing. Since our embedding model (BGE) is multilingual and trained on Chinese with punctuation, it should handle it. So no special post-processing required beyond maybe splitting very long transcripts into chunks for indexing.

In summary, our recommendation is to integrate a **Whisper-based ASR pipeline** using the **small model** for high quality, with built-in VAD to handle pauses, and leaving speaker separation as a future enhancement. This provides a strong local speech-to-text foundation consistent with our privacy goals (all processing local, no audio leaves the device) and with acceptable performance on consumer hardware.

Options & Tradeoffs

Component	Option A (Chosen)	Option B	Trade-off Analysis
ASR Engine	Use **Faster-Whisper** library (CTranslate2)	Use **whisper.cpp** (GGML)	**Chosen A:** Faster-Whisper yields faster throughput on typical hardware, especially with GPUs or multi-core CPUs, and easier Python integration. Option B (whisper.cpp) is highly portable and very low memory with quantization, but slower on CPU than int8 CTranslate2 and not GPU-accelerated. For our target (desktop-class offline use), faster-whisper gives better performance headroom. We may still allow whisper.cpp as a fallback for niche cases, but not default.
Model Size (Default)	**Small** (244M)	Base (74M) or Medium (769M)	**Small vs Base:** We opt for Small because accuracy is significantly better (especially for Chinese and proper nouns) [36] , and with optimization the speed is manageable on CPU. Base would improve speed ~2-3x, but at the cost of more transcription errors (which could propagate to wrong memory entries). We prioritize quality for transcribed content since errors could undermine the usefulness of stored info. **Small vs Medium:** Medium has even better accuracy (~half the error rate of small in some cases [35]) but 3× size – on CPU medium is likely too slow (nearly real-time even on high-end CPU with int8). It also requires more RAM. On GPU, medium might be feasible if user has >=5GB VRAM. As a default, medium is too heavy for many. We pick small as balanced default, but power users with GPUs can switch to medium/large if desired.

Component	Option A (Chosen)	Option B	Trade-off Analysis
VAD & Chunking	**Enable Silero VAD** for silence trimming	No VAD (feed entire audio to Whisper)	**Enabling VAD** improves efficiency and avoids long silences causing issues. Without VAD, Whisper would process entire input including silence, which is slower and might output "..." or nothing for those parts. Also, very long inputs (>30s) – Whisper processes in 30s windows internally but if there's continuous noise, it might struggle with segmentation. Silero VAD is lightweight and already integrated in Option A, so we use it. Downside: it might cut off very short pauses or truncate a word if mis-tuned. However, default thresholds are conservative (removes >2s silences by default [42]). We can tune the VAD sensitivity if needed (faster-whisper allows params). Overall benefit outweighs risk.
Speaker Diarization	**No diarization by default** (single speaker transcription)	Integrate speaker ID (via Pyannote or NeMo)	**No diarization** is simpler and uses far less resources. Full diarization could double the complexity: Pyannote models are ~1-2GB and slow on CPU, requiring clustering logic. For P0, it's not feasible to run well on typical hardware. The trade-off is that multi-speaker recordings won't be labeled with who said what. We assume this is acceptable initially; users primarily get the content. For personal use-cases (dictation, single-user voice notes) it's not an issue. We will note that for multi-speaker scenarios, the transcription will be a single block of text. If diarization is a requested feature, we can later implement an optional pipeline step with clear resource caveats.

Component	Option A (Chosen)	Option B	Trade-off Analysis
Transcription Output Segmentation	**Use Whisper's own segmentation & punctuation**	Force split by fixed interval or external alignment	Whisper is trained to output coherent segments with punctuation at sentence or clause boundaries. Using that yields readable transcriptions. We trust it – e.g. *"Hello, how are you? I am fine."* as two segments. Another approach could be to segment every X seconds uniformly or to align words to a fixed timeline (like some systems do), but that would degrade readability (splitting mid-sentence) and isn't needed. We prefer Whisper's intelligent segmentation. We'll join segments for final text or keep them as separate entries if needed. This maximizes the linguistic quality of stored text.

Other considerations: - **Real-time vs Batch:** We are mostly doing batch transcription of recorded audio (not streaming realtime) – that's fine. We're not building a streaming ASR at this time (though whisper.cpp and faster-whisper could be used in streaming mode, we likely don't need that complexity). - **Memory usage by ASR:** On CPU, int8 small ~ 500MB is fine; plus overhead. On a 8GB RAM system, no issue. On GPU, 1GB VRAM usage is fine on most discrete GPUs we target. Base model only ~200MB – not an issue. So either is fine memory-wise. If user's PC has very low RAM (4GB), int8 small might be borderline – then base or heavy quant might be needed. We'll mention memory footprints in docs for transparency.

Implementation Steps

1. **Dependency Setup:**
2. Add `faster-whisper` to our requirements (it can be installed via pip). Ensure we specify a version (e.g. `faster-whisper==0.6.0` or latest stable) for consistency.
3. Verify that `faster-whisper` brings CTranslate2 and everything needed. We might also need PyTorch if we want to run on GPU? Actually, faster-whisper uses CTranslate2 backend and doesn't require full PyTorch for inference. It can use Intel MKL/OpenBLAS for CPU and CUDA for GPU internally. We should note that it may require installing some runtime libraries (like it comes with precompiled binaries for common platforms).
4. For silero VAD, faster-whisper includes it, so no extra package needed. If using whisper.cpp variant, we'd add `whispercpp` python binding or compile ourselves. But for now, we proceed with faster-whisper.
5. *Owner:* DevOps / environment maintainers. **(Small)** – adding the package and testing installation on our supported OS.
6. **Model Download & Loading:**
7. Decide how to manage model files. Faster-whisper can download from HuggingFace or local path. We can allow user to specify model path or default to auto-download. The small model is ~1.2GB (for float32) – but faster-whisper provides quantized int8 models on HuggingFace as well (there's `small-int8` variant around ~600MB). Possibly we use that to save time and memory.

8. We could use the model identifier: `"openai/whisper-small"` or faster-whisper's own repository. Actually, faster-whisper might have separate model files hosted (the documentation suggests model names like `"small"` which it will fetch from HuggingFace).
9. Implementation: at app startup, load the model:

```
from faster_whisper import WhisperModel
model_size = "small"  # or "small.en" for English-only model; but we want
multilingual
model = WhisperModel(model_size, device="cuda" if available else "cpu",
compute_type="int8" if CPU else "float16")
```

   This will automatically download the model if not present (to HF cache). We need to ensure offline scenario: in Task 11, we'll address packaging. Possibly instruct user to download the model beforehand if no internet.
10. Use environment flags: e.g. `ASR_MODEL_SIZE=small/base/medium`. Also `ASR_DEVICE=cpu/gpu` (though we can auto-detect GPU).
11. *Owner:* Backend engineer. **(Medium)** – loading could be done similarly to how we load embeddings. ~0.5 day to implement with fallback logic.
12. **Transcription Pipeline:**
13. Implement a function `transcribe_audio(file_path)` in our `services/api` or a new `asr.py` module.
14. This function will use the loaded `WhisperModel` to transcribe:

```
segments, info = model.transcribe(audio_path, beam_size=... ,
vad_filter=True)
```

   By default, we can use `beam_size=5` (for better accuracy than greedy). If speed is critical, can use `beam_size=1` (greedy). We likely keep beam or a small temperature fallback. The function returns `segments` generator and `info` with language. We should iterate `for segment in segments: text = segment.text` and collect them. Or we can do `segments = list(segments)` to force transcription to complete, then iterate.
    - Each `segment` has `start`, `end`, `text`.
15. Combine segments' text into one full transcript string separated by spaces (since punctuation is included, it will form proper sentences).
16. Alternatively, if we want to store by segments (maybe for linking times), we can store each segment separately in memory with its timestamp. But likely simpler: store one combined transcript in memory. That might be fine for <5min audio. If audio is very long (e.g. 1-hour meeting), storing whole transcript as one entry might be too large for embedding (embedding model limit ~512 tokens). In that case, we should chunk the transcript into smaller parts for embedding. Possibly we do: after transcription, split the text by sentence or every ~256 characters and ingest multiple entries. This ensures our vector search works (embedding model is limited to 512 tokens context).
    - We can use the segment boundaries to guide chunking (each segment is already maybe 1 sentence or phrase). E.g. accumulate segments until ~200 chars, then form a chunk.
17. Add metadata: maybe `{"source": "transcript", "audio_file": filename, "start_time": 0}` etc., but not essential. Could help identify the memory origin.

18. Insert into our ingestion pipeline (services/api/ingest) such that when a user uploads or records audio, we call `transcribe_audio` then ingest resulting text.
19. *Owner:* Backend engineer. **(Medium)** – implement and test on a sample audio (like record oneself speaking a sentence in English and Chinese to verify output).
20. **Speaker Diarization (decided to skip)**:
21. We will not implement Pyannote or other heavy diarization. But we might include a placeholder: e.g. if user specifically requests diarization, we log “Not supported in this version” or similar. Or if the audio has multiple speakers, the transcript will just read like a monologue. Possibly confusing? Perhaps we insert line breaks where Whisper does if there are very long pauses (which often coincide with speaker changes). But not always accurate.
22. So skip any explicit code for this. Document it clearly.
23. **Testing:**
24. Try with short English audio (one speaker): verify punctuation and text is correct. Use a known sample or record our own.
25. Try short Chinese audio: verify Chinese characters output and roughly correct. E.g. say "今天是星期五" and see that it transcribes correctly with punctuation.
26. If possible, test a multi-speaker snippet (two people conversing). See how output looks – likely it will output as one paragraph. Confirm no crashes.
27. Test performance: measure how long it takes for e.g. a 30-second audio on CPU vs with GPU. Ensure it aligns with expectations (fast on GPU, acceptable on CPU).
28. Ensure memory usage is okay (should be, but monitor for any unusual spikes).
29. We should also test the ingestion of transcript into memory: does the system then allow searching that content successfully.
30. Validate that language detection works: check `info.language` from faster-whisper. It should output code (like 'en' or 'zh'). We might optionally log it. Also, ensure model is multi-lingual (we should load "small" not "small.en", because .en is English-only and not cover Chinese).
31. Confirm VAD is indeed filtering silence: we could test on an audio with a long pause and see if it sped up (like two sentences 5 seconds apart – with VAD, it should skip the silent gap quickly).
32. *Owner:* QA/engineer. **(Medium)** – thorough manual tests, adjust parameters if needed (like beam size vs temperature).
33. **Documentation Updates:**
34. In `docs/requirements.md`, add under "System Requirements" -> "Audio transcription": list model sizes and recommended hardware. E.g.:
    - "Default uses Whisper Small (transcribe ~ realtime on modern CPU, needs ~0.5GB RAM). For faster performance with slight accuracy loss, can use Whisper Base (set `ASR_MODEL=base`). If GPU present, transcription can be 5-10× faster; the system will use it automatically."
    - Mention for completeness that all processing is offline and model is MIT licensed.
35. In `docs/architecture.md`, add a section about the ASR pipeline in the data flow. Show that audio input goes to ASR module, producing text that flows into the vector store. Mention we use whisper and describe how we ensure quality (VAD, etc.). Possibly include a figure if we have a pipeline diagram (like "Mic input -> Whisper -> text -> memory").
36. Add any usage instructions in README: e.g. "You can speak to Epoch or upload an audio file; it will transcribe locally. Ensure you have the model downloaded (~1GB)."
37. Note limitations: "Diarization (identifying speakers) is not supported in this version; multi-speaker audio will be transcribed as one block of text."
38. If appropriate, add to FAQ: "Is my audio processed locally? – Yes, uses open-source Whisper model, no data leaves your device."

39. *Owner:* Tech writer/engineer. **(Small)** – 0.5-1 day to write and proof updates.
40. **Performance Considerations and Future:**
41. Possibly implement a fallback: if user’s device cannot handle small model, allow base. We can gauge automatically perhaps by environment (but simplest is user config). We'll document that environment variable.
42. Also, consider making context length of memory ingestion configurable: e.g. we chunk transcripts by at most 256 tokens to fit embedding model, as mentioned. We'll implement that now to avoid extremely long text embedding (embedding model limited to 512 tokens anyway).
43. Logging: we should log when transcription starts and ends (since it can take some seconds, user might wonder). Possibly provide progress (faster-whisper returns segments as generator – we could stream partial results if we had UI hooking into it, but maybe overkill for now).
44. For now, our focus is on correctness and working end-to-end. If slow, user can wait a bit (we can show a spinner in UI if needed).
45. Summarily, this step integrates local ASR fully inline with our privacy stance, making voice a first-class modality for input.

By completing these steps, Project Epoch’s local ASR feature will allow users to convert speech to text with high accuracy entirely offline. This adds a valuable modality (voice notes, meeting recordings, etc.) to the personal AI’s memory ingestion process.

Risks & Mitigations

- **Performance on Long Audio (CPU):** If a user tries to transcribe a very long recording (say 1 hour) on CPU with whisper-small, it could take ~1 hour. That’s a risk (it might seem to hang). *Mitigation:* We can warn in docs that long audio on CPU can be slow and recommend chunking or using a GPU. Also, we might implement an optional parameter to limit audio length or encourage using base model for > X minutes. We should definitely ensure our UI indicates progress or doesn’t block entirely. Perhaps we can break audio into 5-minute chunks and process sequentially, logging each chunk done. This provides some feedback. We could also allow user to abort if needed.
- **Memory consumption for long audio transcripts:** A 1-hour meeting transcript could be ~10k words. Ingesting that as one giant text chunk is not ideal (embedding truncation). We mitigated by chunking transcripts for embedding. Still, storing a 1-hour text as many segments is fine (just many entries). Vector DB can handle thousands of entries.
- **Quality issues in noisy environments:** Whisper is good but not magic; heavy noise or overlapping speech will reduce accuracy. *Mitigation:* We note that quality may degrade in such conditions and no external API is used, so it’s best-effort. Possibly advise using a good microphone or a noise reduction step (there are algorithms, but out of scope to implement).
- **Language mixing and code-switching:** If a user mixes languages in one audio (English and Chinese), Whisper will output mostly one language. Actually, Whisper can handle code-switching to some extent, but sometimes it might output say Chinese text in Latin script incorrectly or vice versa. It’s rare but possible. *Mitigation:* Not much to do except note that it's trained on multilingual data and usually fine. We could run in multilingual mode always (we do).
- **Telephony or low-quality audio:** Whisper small might struggle more on low-sample-rate, low-bandwidth audio (like phone call 8kHz). *Mitigation:* If we anticipate that, we might upsample audio or at least ensure we pass correct sample rate to Whisper (faster-whisper likely does resample to 16k internally). We’ll trust its pipeline.
- **Integration complexity:** Bringing in a heavy model could slow startup or package size. *Mitigation:* We’ll load model asynchronously if possible or show a log "loading model, might take ~10s". Also,

emphasize in packaging that model isn't included by default in Docker (we may allow mounting it or downloading on first run). See Task 11 for offline packaging details.

- **License & Compliance:** Whisper model is MIT – very permissive. Silero VAD is also MIT (from Silero). We should include attributions: mention OpenAI Whisper and Silero in our README or about page as used technologies. Possibly include citations to their papers in our references. That covers compliance.
- **Ambient listening privacy:** If the system had a microphone always listening, that raises privacy concerns ironically for a privacy-first AI. But we won't do open mic listening by default – we likely require user to press record or provide audio file. We should ensure we don't accidentally capture audio without consent. *Mitigation:* Only activate ASR when user triggers it (like clicking a "record" button in UI). Also, clearly state all audio stays local (which is a privacy plus).
- **Multi-threading issues:** The faster-whisper model is thread-safe to transcribe one file at a time. We should avoid trying to transcribe two audios simultaneously (not a typical use case, but if API allowed it, heavy CPU usage). Not a big problem, but we can handle sequentially or have separate model instances if needed. Not likely an issue in single-user scenario.
- **Rerunning ASR on same audio repeatedly**: Could be slow. *Mitigation:* Perhaps cache transcripts for audio files by checksum so if same file processed again, we reuse. But that's an edge case; not needed now.

By understanding these risks and addressing where needed, we ensure our ASR integration is robust, respectful of resources, and aligns with user expectations for privacy and performance.

References

- 【32】GitHub Discussion (OpenAI Whisper) – *Maintainer notes: Whisper medium/large ~8% WER on Cantonese (yue), showing strong Chinese perf* [35]
- 【30】Speechly Blog – *Whisper accuracy across languages and model sizes (snippets show Chinese WER ~0.33 for smaller vs ~0.08 for large)* [34]
- 【35】Faster-Whisper GitHub – *Usage examples: how to use* `WhisperModel.transcribe` *with VAD and batching* [33]
- 【33】HuggingFace Whisper Large Card – *Model details: 99 languages, license Apache-2.0* [41] [43]
- 【42】Reddit (LocalLLaMA) – *Mention of MIT license for Phi-3 mini and presumably Whisper's open license* [39] (OpenAI's announcement also put Whisper under MIT).
- 【35】Faster-Whisper README – *community integrators list: WhisperX (for diarization and word timestamps)* [38] (we decide not to integrate but know it exists).
- **OpenAI Whisper Paper (Radford et al. 2022)** – states small vs base model WERs (English ~11% vs 13%, etc.) – we rely on memory and secondary sources.
- **Silero VAD GitHub** – states silero-vad can detect speech segments with high accuracy and how to use it (faster-whisper likely wraps it).

[1] What are some popular pre-trained Sentence Transformer models and how do they differ (for example, all-MiniLM-L6-v2 vs all-mpnet-base-v2)?
https://milvus.io/ai-quick-reference/what-are-some-popular-pretrained-sentence-transformer-models-and-how-do-they-differ-for-example-allminilml6v2-vs-allmpnetbasev2

[2] sentence-transformers/all-MiniLM-L6-v2 · License and commercial usage
https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2/discussions/34

[3] What embedding model do you guys use? : r/LangChain
https://www.reddit.com/r/LangChain/comments/1816mb5/what_embedding_model_do_you_guys_use/

[4] BGE v1 & v1.5 — BGE documentation
https://bge-model.com/bge/bge_v1_v1.5.html

[5] [15] The guide to bge-base-zh-v1.5 | BAAI
https://zilliz.com/ai-models/bge-base-zh-v1.5

[6] [13] [14] [26] [27] [28] [29] [31] BAAI/bge-small-zh-v1.5 · Hugging Face
https://huggingface.co/BAAI/bge-small-zh-v1.5

[7] [12] BGE-M3 — BGE documentation
https://bge-model.com/bge/bge_m3.html

[8] [9] [10] [11] BAAI/bge-m3 · Hugging Face
https://huggingface.co/BAAI/bge-m3

[16] Can embeddings be secured?
https://milvus.io/ai-quick-reference/can-embeddings-be-secured

[17] Chroma Docs: Introduction
https://docs.trychroma.com/

[18] My strategy for picking a vector database: a side-by-side comparison : r/vectordatabase
https://www.reddit.com/r/vectordatabase/comments/170j6zd/my_strategy_for_picking_a_vector_database_a/

[19] [20] [21] What Is Weaviate? A Semantic Search Database
https://www.oracle.com/database/vector-database/weaviate/

[22] facebookresearch/faiss: A library for efficient similarity ... - GitHub
https://github.com/facebookresearch/faiss

[23] Top 7 Open-Source Vector Databases: Faiss vs. Chroma & More
https://research.aimultiple.com/open-source-vector-databases/

[24] Maximal Marginal Relevance - by Anton Dergunov - Medium
https://medium.com/@adergunov/maximal-marginal-relevance-144c23b42be5

[25] Enhancing RAG with Maximum Marginal Relevance (MMR) in Azure ...
https://farzzy.hashnode.dev/enhancing-rag-with-maximum-marginal-relevance-mmr-in-azure-ai-search

[30] has anyone tried Qwen-7B-Chat? : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/15lpyto/has_anyone_tried_qwen7bchat/

[32] MTEB | EvalScope - Read the Docs
https://evalscope.readthedocs.io/en/v0.16.3/user_guides/backend/rageval_backend/mteb.html

[33] [38] [40] [42] GitHub - SYSTRAN/faster-whisper: Faster Whisper transcription with CTranslate2
https://github.com/SYSTRAN/faster-whisper

[34] [35] Spoken Chinese languages #25 - openai whisper - GitHub
https://github.com/openai/whisper/discussions/25

[36] Analyzing Open AI's Whisper ASR Accuracy: Word Error Rates ...
https://www.speechly.com/blog/analyzing-open-ais-whisper-asr-models-word-error-rates-across-languages

[37] [41] [43] openai/whisper-large-v3 · Hugging Face
https://huggingface.co/openai/whisper-large-v3

[39] How good is Phi-3-mini for everyone? : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1cbt78y/how_good_is_phi3mini_for_everyone/