# Dynamic Chatbot Knowledge Base — Design & Implementation

## Goal

Implement a system that automatically discovers new information from specified sources, converts that information into embeddings, and updates a vector database (e.g., FAISS, Milvus, Pinecone) so the chatbot can incorporate fresh content into responses over time.

## High-level Architecture

1) Source Fetchers — pull content from RSS, webpages, PDFs, Google Drive, S3, databases.
2) Text Extractors & Normalizers — convert HTML/PDF/docx to clean text, deduplicate and normalize.
3) Embedding Generator — convert cleaned text chunks to vector embeddings (provider-agnostic).
4) Vector Store Updater — add new vectors and metadata to a vector DB (FAISS, Milvus, Pinecone).
5) Periodic Scheduler & Watchers — cron/APScheduler watchers that trigger updates.
6) Change Detection & Incremental Update — detect changed/new documents to avoid reprocessing unchanged content.
7) Atomic Swap Strategy — build a new index then swap it atomically to avoid downtime.
8) Monitoring & Alerts — log successes/failures, metrics, health checks.

## Design Details

Sources:
- RSS/Atom feeds: quick feed of articles.
- Webpages: HTML scraping with rate limiting and politeness (robots.txt).
- PDFs/DOCX: PDF parsing (PyPDF2 / pdfminer.six) and docx extraction.

Chunking:
- Split text into overlapping chunks (e.g., 500 tokens / ~ 400-800 characters with 50-100 char overlap).
- Keep metadata (source URL, title, published date, chunk_id).

Deduplication & Change Detection:
- Compute content hashes (SHA256) per source; store last-hash in a small metadata DB (SQLite).
- If hash unchanged, skip re-embedding.

Embeddings:
- Keep an abstraction layer `embed(text)` so provider can be swapped.
- Batch embedding requests for throughput and cost efficiency.

Vector DB Update Patterns:
- For FAISS (local): create a new index file for the update batch, merge, or rebuild periodically, then swap files.
- For remote vector DBs (Pinecone/Milvus): use upsert APIs for incremental updates and metadata filtering.

Atomic Swap:
- Write new index to a temp location, validate it, then rename/move to production path (or use aliasing in managed DB).

## Operational Flow (step-by-step)

1. Scheduler triggers: e.g., every hour or using file-system event.
2. For each configured source:
a) Fetch content.
b) Extract and normalize text.
c) Compute hash and compare to stored hash in metadata DB.

d) If changed or new: chunk, create embeddings, push to vector store.
3. Run a quick validation (sample queries) to ensure the updated index works.
4. Swap indexes atomically.
5. Log results and send alerts if failures occurred.

## Sample Implementation (Python) — Key files

Files included in the sample:
- fetchers.py — functions to fetch RSS, web pages, and PDFs.
- extractor.py — text extraction & normalization.
- embeddings_provider.py — embedding abstraction.
- vector_store.py — FAISS / Pinecone wrapper.
- updater.py — the scheduler and update logic.
- metadata_db.py — small SQLite metadata table for hashes and processed timestamps.
- README.md — configuration and deployment notes.

Below are concise, ready-to-use code snippets that you can drop into the above files.

## Code: metadata_db.py

```
python
import sqlite3
from contextlib import closing

DB = 'metadata.db'

def init_db():
    with closing(sqlite3.connect(DB)) as conn:
        c = conn.cursor()
        c.execute('''CREATE TABLE IF NOT EXISTS source_meta(
                     source_id TEXT PRIMARY KEY,
                     content_hash TEXT,
                     last_processed TIMESTAMP)''')
        conn.commit()

def get_hash(source_id):
    with closing(sqlite3.connect(DB)) as conn:
        c=conn.cursor()
        c.execute('SELECT content_hash FROM source_meta WHERE source_id=?',(source_id,))
        r=c.fetchone()
        return r[0] if r else None

def set_hash(source_id, content_hash, ts):
    with closing(sqlite3.connect(DB)) as conn:
        c=conn.cursor()
        c.execute('REPLACE INTO source_meta(source_id,content_hash,last_processed) VALUES(?,?,?)',
                  (source_id,content_hash,ts))
        conn.commit()
```

## Code: embeddings_provider.py

```
python
# Keep this provider-agnostic. Swap in OpenAI, Cohere, HuggingFace, etc.
from typing import List

class EmbeddingsProvider:
    def __init__(self, client):
        self.client = client

    def embed_batch(self, texts: List[str]) -> List[List[float]]:
        """Return list of vectors corresponding to texts."""
```

```
        # Example with a hypothetical client
        return self.client.embed(texts)

# Example wrapper for OpenAI-like client (pseudocode):
# def openai_embed(client, texts):
#     ...
```

## Code: vector_store.py (FAISS example)

```
python
import faiss
import numpy as np
import os

INDEX_DIR = './indexes'
os.makedirs(INDEX_DIR, exist_ok=True)

def create_index(embeddings_np: np.ndarray, ids: List[str]):
    d = embeddings_np.shape[1]
    index = faiss.IndexFlatIP(d)  # or IndexHNSWFlat / IVF depending on scale
    index.add(embeddings_np)
    return index

def save_index(index, filename):
    faiss.write_index(index, filename)

def load_index(filename):
    return faiss.read_index(filename)

# Use metadata mapping (id -> metadata) stored as JSON or in a small DB.
```

## Code: updater.py (scheduler + update loop)

```
python
import time
import hashlib
from datetime import datetime
from apscheduler.schedulers.blocking import BlockingScheduler
from metadata_db import get_hash, set_hash, init_db
from fetchers import fetch_from_source  # user to implement
from extractor import extract_text  # user to implement
from embeddings_provider import EmbeddingsProvider
from vector_store import create_index, save_index

init_db()

def process_source(source):
    raw = fetch_from_source(source)
    text = extract_text(raw)
    content_hash = hashlib.sha256(text.encode('utf-8')).hexdigest()
    prev = get_hash(source['id'])
    if prev == content_hash:
        print(f"{source['id']} unchanged — skipping")
        return
    # chunking (simple split)
    chunks = [text[i:i+1000] for i in range(0,len(text),900)]
    provider = EmbeddingsProvider(client=...)  # configure
    embeddings = provider.embed_batch(chunks)
    import numpy as np
    emb_np = np.array(embeddings).astype('float32')
    index = create_index(emb_np, ids=[f"{source['id']}_chunk_{i}" for i in range(len(chunks))])
    timestamp = datetime.utcnow().isoformat()
    filename = f"./indexes/{source['id']}_{int(time.time())}.index"
    save_index(index, filename)
    set_hash(source['id'], content_hash, timestamp)
    print('Updated', filename)
```

```
def run_once():
    sources = [
        {'id':'example_rss','type':'rss','url':'https://example.com/feed'},
        # add real sources
    ]
    for s in sources:
        try:
            process_source(s)
        except Exception as e:
            print('Failed', s['id'], e)

if __name__ == '__main__':
    scheduler = BlockingScheduler()
    scheduler.add_job(run_once, 'interval', minutes=60)  # every hour
    print('Starting updater...')
    scheduler.start()
```

## Deployment Notes & Best Practices

- Use Docker for reproducible environments.
- Secrets (API keys) should be stored in environment variables or secret managers — never in code.
- Respect source `robots.txt` and rate limits.
- Monitor index size and reindex periodically (e.g., weekly) to reclaim space.
- Add tests for correctness (sample queries before swap).
- Implement ACLs and data governance for what sources are allowed.

## What I included in this PDF

- Clear architecture and operational flow.
- Code snippets (metadata DB, embeddings provider abstraction, vector store wrapper, updater scheduler).
- Best practices and deployment notes.

Feel free to copy the code snippets into files in your repo and adapt the provider and vector-store details to your stack.

## Usage in your chatbot code

When answering user queries, pass the latest vector store and metadata into the retrieval pipeline. Example pseudocode:

```
# query flow:
q_vec = embed(query)
results = vector_store.search(q_vec, top_k=10)
context = assemble_context(results)
answer = llm.generate_with_context(query, context)
```

## Contact / Next Steps

If you'd like, I can:
- Produce full-file versions (complete fetchers, extractor implementations).
- Convert the snippets into a runnable Git repo and produce a zip.
- Customize for Pinecone / Milvus / your cloud provider.

Tell me which option you prefer and I'll prepare the files.