RagInABox

RagInABox aims to be a configurable, end-to-end RAG starter kit. It should be able to ingest local folders, scrape websites (including linked files like PDFs), chunk the content, and store it in a vector database (Azure AI Search by default). A Gradio frontend provides the chat interface, with Azure OpenAI GPT5.2 as the default LLM and text-embedding-3-small for embeddings. All keys and URLs are supplied via a `.env` file.

High-level Architecture:

A. Core (framework-agnostic)

- Interfaces (Protocols/ABCs) for each stage
- Plain data models (Document, Chunk, etc.)
- Orchestrators that run the ingestion/indexing pipeline and the chat pipeline

B. Adapters (implementations)

- Local folder source, website crawler source
- PDF/HTML/text extractors
- Chunkers
- Embedding providers (Azure OpenAI by default)
- Vector stores (Azure AI Search by default)
- LLM providers (Azure OpenAI GPT-5.2 by default)

C. App (composition + UI)

- Config loader from .env
- Factory to assemble the chosen adapters
- Gradio frontend calling the chat orchestrator
- CLI commands to run ingestion and serve UI

Main Components ad Interfaces

Data models (simple, shared everywhere)

- Document
    - id: str

    - uri: str (file path or URL)
    - content: str
    - mime_type: str
    - metadata: dict
- Chunk
    - id: str
    - document_id: str
    - text: str
    - metadata: dict
- SearchResult
    - chunk: Chunk
    - score: float

“Ports” (interfaces you code against)

These are the key to composability:

- ContentSource
    - iter_items() -> Iterable[SourceItem]
    - SourceItem could be {uri, bytes|text, mime_type, metadata}
- Extractor
    - extract(item: SourceItem) -> Document
- Chunker
    - chunk(doc: Document) -> list[Chunk]
- Embedder
    - embed_texts(texts: list[str]) -> list[list[float]]
- VectorStore
    - upsert(chunks: list[Chunk], vectors: list[list[float]]) -> None
    - query(query_vector: list[float], k: int, filters: dict|None) -> list[SearchResult]
- LLM
    - chat(messages: list[dict], **kwargs) -> str (keep it minimal)

- Reranker (optional later)
    - rerank(query: str, results: list[SearchResult]) -> list[SearchResult]

Orchestrators (the “brains”)

- IngestionPipeline
    - takes source(s) → extractor(s) → chunker → embedder → vectorstore
- RAGChatEngine
    - takes user message → embed query → vectorstore retrieve → build prompt → LLM answer
    - optionally returns citations (chunk metadata)

Composition

- Settings
    - loads .env and provides typed config (endpoints, keys, model names, index name, etc.)
- AppFactory
    - build_ingestion_pipeline(settings) -> IngestionPipeline
    - build_chat_engine(settings) -> RAGChatEngine

TOC:

    - smoketest: **smoke_step4_azure.py**
- Step 5: Build Ingestion pipeline (local-only, no web yet)
    - Chunker (simple version): **adaptors/chunking/simple_chunker.py**
    - Extractors(txt/md/html/pdf):
        - txt: **adaptors/extractors/text_extractor.py**
        - md: **adaptors/extractors/md_extractor.py**
        - html: **adaptors/extractors/html_extractor.py**
        - pdf: **adaptors/extractors/pdf_extractor.py**
    - Extractor registry: **adaptors/extractors/registry.py**
    - Local Folder Source: **adaptors/sources/local_folder.py**
    - Ingestion pipeline(batch embeddings+upsert): **pipelines/ingestion.py**
    - cli command to ingest: **cli.py**

Step 1: Create the project skeleton + Settings

Step 1a: Create the project skeleton (uv)

Run these commands:

```
# 1) Create a folder and initialize a uv-managed project
mkdir RagInABox
cd RagInABox
uv init --name rag-in-a-box

# 2) Create the package module folder
mkdir -p
rag_in_a_box/{config,core,adapters/{llm,embeddings,vectorstores,sources,ex
tractors,chunking},pipelines,app}

# 3) Add empty __init__.py files so Python treats folders as packages
touch rag_in_a_box/__init__.py
touch rag_in_a_box/config/__init__.py
touch rag_in_a_box/core/__init__.py
touch rag_in_a_box/pipelines/__init__.py
touch rag_in_a_box/app/__init__.py
touch rag_in_a_box/adapters/__init__.py
touch
rag_in_a_box/adapters/{llm,embeddings,vectorstores,sources,extractors,chun
king}/__init__.py

# 4) Add dependencies (we'll keep it minimal but practical)
uv add pydantic pydantic-settings gradio httpx beautifulsoup4 pypdf azure-
search-documents openai tenacity rich

# 5) (Optional but recommended) dev tools
uv add --dev ruff pyright pytest
```

What you should have afterward

- `pyproject.toml` created by uv
- `rag_in_a_box/` package folder with subpackages
- dependencies tracked in `pyproject.toml`

Step 1b: Add `.env.example` and `.gitignore`

Create **`.env.example`** at the project root:

```
# ------------------
# Azure OpenAI
# ------------------
AZURE_OPENAI_ENDPOINT=https://YOUR-RESOURCE-NAME.openai.azure.com/
AZURE_OPENAI_API_KEY=YOUR_KEY
AZURE_OPENAI_API_VERSION=2024-xx-xx
AZURE_OPENAI_CHAT_DEPLOYMENT=gpt-5-2-deployment
AZURE_OPENAI_EMBEDDING_DEPLOYMENT=text-embedding-3-small-deployment

# ------------------
# Azure AI Search
# ------------------
AZURE_SEARCH_ENDPOINT=https://YOUR-SEARCH-NAME.search.windows.net
AZURE_SEARCH_API_KEY=YOUR_KEY
AZURE_SEARCH_INDEX=raginabox

# ------------------
# Ingestion defaults
# ------------------
INGEST_LOCAL_PATH=./data
INGEST_START_URLS=["https://fluvius.be/nl", "https://site2.com/docs"]
INGEST_ALLOWED_DOMAINS=["fluvius.be", "site2.com"]

# ------------------
# Chunking + Retrieval
# ------------------
CHUNK_SIZE=800
CHUNK_OVERLAP=120
TOP_K=5
```

Create **`.gitignore`**:

```
.env
__pycache__/
*.pyc
.venv/
dist/
build/
.pytest_cache/
.mypy_cache/
.ruff_cache/
```

Then copy to a real `.env`:

```
cp .env.example .env
```

Step 2: implement typed settings + a config smoke test

Create `rag_in_a_box/config/settings.py`:

```
from __future__ import annotations

from pydantic import Field, field_validator
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    """
    Typed settings loaded from .env (and environment variables).
    Keep this as the single source of truth for configuration keys.
    """

    model_config = SettingsConfigDict(env_file=".env",
env_file_encoding="utf-8", extra="ignore")

    # Azure OpenAI
    azure_openai_endpoint: str = Field(..., alias="AZURE_OPENAI_ENDPOINT")
    azure_openai_api_key: str = Field(..., alias="AZURE_OPENAI_API_KEY")
    azure_openai_api_version: str = Field(...,
alias="AZURE_OPENAI_API_VERSION")
    azure_openai_chat_deployment: str = Field(...,
alias="AZURE_OPENAI_CHAT_DEPLOYMENT")
    azure_openai_embedding_deployment: str = Field(...,
alias="AZURE_OPENAI_EMBEDDING_DEPLOYMENT")

    # Azure AI Search
    azure_search_endpoint: str = Field(..., alias="AZURE_SEARCH_ENDPOINT")
    azure_search_api_key: str = Field(..., alias="AZURE_SEARCH_API_KEY")
    azure_search_index: str = Field("raginabox",
alias="AZURE_SEARCH_INDEX")

    # Ingestion defaults
    ingest_local_path: str = Field("./data", alias="INGEST_LOCAL_PATH")
    ingest_start_urls: list[str] = Field(default_factory=list,
alias="INGEST_START_URLS")
    ingest_allowed_domains: list[str] = Field(default_factory=list,
alias="INGEST_ALLOWED_DOMAINS")

    # Chunking + retrieval
    chunk_size: int = Field(800, alias="CHUNK_SIZE")
```

```
    chunk_overlap: int = Field(120, alias="CHUNK_OVERLAP")
    top_k: int = Field(5, alias="TOP_K")
```

Create a tiny smoke test `rag_in_a_box/cli.py`:

```
from __future__ import annotations

from rag_in_a_box.config.settings import Settings


def main() -> None:
    s = Settings()
    print("✅ Settings loaded")
    print(f"Azure OpenAI endpoint: {s.azure_openai_endpoint}")
    print(f"Chat deployment:       {s.azure_openai_chat_deployment}")
    print(f"Embedding deployment:  {s.azure_openai_embedding_deployment}")
    print(f"Search endpoint:       {s.azure_search_endpoint}")
    print(f"Search index:          {s.azure_search_index}")
    print(f"Ingest local path:     {s.ingest_local_path}")
    print(f"Start URLs:            {s.start_urls()}")
    print(f"Allowed domains:       {s.allowed_domains()}")
    print(f"Chunk size/overlap:    {s.chunk_size}/{s.chunk_overlap}")
    print(f"Top K:                 {s.top_k}")


if __name__ == "__main__":
    main()
```

Run it:

```
uv run python -m rag_in_a_box.cli
```

If `.env` is filled with real values, you should see the printed config. If something is missing, Pydantic will tell you exactly which env var is absent.

Why we're doing this first

Once `Settings` is stable, every adapter can depend on it and you get:

- one place to change env keys
- validation errors early
- easy future extension (Azure Storage settings later, etc.)

Step 3: Define data models + interfaces

We'll define the **core models + interfaces** (`Document`, `Chunk`, `ContentSource`, `Extractor`, `Chunker`, `Embedder`, `VectorStore`, `LLM`) and write a "fake" in-memory vector store so you can test the pipeline without Azure yet.

When you've run the smoke test successfully, tell me and we'll move to Step 3.

We'll do this in small, safe pieces:

1. core/models.py (shared dataclasses)
2. core/interfaces.py (Protocols you code against)
3. A minimal in-memory vector store + dummy embedder + dummy LLM
4. Minimal RAG chat engine (core pipeline piece)
5. A tiny smoke demo that proves "retrieve + generate" works end-to-end

3.1 Create rag_in_a_box/core/models.py

```
from __future__ import annotations

from dataclasses import dataclass, field
from typing import Any, Dict, Optional


Metadata = Dict[str, Any]


@dataclass(frozen=True)
class SourceItem:
    """
    Raw item produced by a ContentSource.
    - uri: where it came from (file path, URL, blob URL, etc.)
    - data: either bytes (e.g., pdf) or text
    - mime_type: used to select an extractor
    """
    uri: str
    data: bytes | str
    mime_type: str = "text/plain"
    metadata: Metadata = field(default_factory=dict)


@dataclass(frozen=True)
class Document:
    """
    Extracted, readable text with metadata.
    """
    id: str
    uri: str
    content: str
    mime_type: str = "text/plain"
    metadata: Metadata = field(default_factory=dict)
```

```python
@dataclass(frozen=True)
class Chunk:
    """
    A chunk of text derived from a Document.
    """
    id: str
    document_id: str
    uri: str
    text: str
    metadata: Metadata = field(default_factory=dict)


@dataclass(frozen=True)
class SearchResult:
    chunk: Chunk
    score: float


@dataclass(frozen=True)
class ChatMessage:
    role: str  # "system" | "user" | "assistant"
    content: str


@dataclass(frozen=True)
class ChatResponse:
    answer: str
    sources: list[SearchResult] = field(default_factory=list)
```

3.2 Create rag_in_a_box/core/interfaces.py

```python
from __future__ import annotations

from typing import Iterable, Optional, Protocol, runtime_checkable

from rag_in_a_box.core.models import Chunk, ChatMessage, Document,
SearchResult, SourceItem


@runtime_checkable
class ContentSource(Protocol):
    def iter_items(self) -> Iterable[SourceItem]:
        ...


@runtime_checkable
class Extractor(Protocol):
    def can_handle(self, mime_type: str) -> bool:
        ...
```

```
    def extract(self, item: SourceItem) -> Document:
        ...


@runtime_checkable
class Chunker(Protocol):
    def chunk(self, doc: Document) -> list[Chunk]:
        ...


@runtime_checkable
class Embedder(Protocol):
    def embed_texts(self, texts: list[str]) -> list[list[float]]:
        ...


@runtime_checkable
class VectorStore(Protocol):
    def upsert(self, chunks: list[Chunk], vectors: list[list[float]]) ->
None:
        ...

    def query(
        self,
        query_vector: list[float],
        k: int,
        filters: Optional[dict] = None,
    ) -> list[SearchResult]:
        ...


@runtime_checkable
class LLM(Protocol):
    def chat(self, messages: list[ChatMessage], **kwargs) -> str:
        ...
```

3.3 Add tiny in-memory adaptors (for testing)

3.3.1 Dummy Embedder:

Create rag_in_a_box/adapters/embeddings/dummy_embedder.py:

```
from __future__ import annotations

import hashlib
import math


class DummyHashEmbedder:
    """
```

```
    Deterministic embedder for smoke testing WITHOUT external services.
    Produces a fixed-length vector by hashing the text.
    """

    def __init__(self, dim: int = 64):
        self.dim = dim

    def embed_texts(self, texts: list[str]) -> list[list[float]]:
        return [self._embed_one(t) for t in texts]

    def _embed_one(self, text: str) -> list[float]:
        v = [0.0] * self.dim
        if not text:
            return v

        # Hash into buckets
        for token in text.lower().split():
            h = hashlib.sha256(token.encode("utf-8")).digest()
            idx = int.from_bytes(h[:2], "big") % self.dim
            v[idx] += 1.0

        # L2 normalize (helps cosine similarity)
        norm = math.sqrt(sum(x * x for x in v)) or 1.0
        return [x / norm for x in v]
```

3.3.2 In-memory vectorstore:

Create rag_in_a_box/adapters/vectorstores/in_memory.py:

```
from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Optional

from rag_in_a_box.core.models import Chunk, SearchResult


def _dot(a: list[float], b: list[float]) -> float:
    return sum(x * y for x, y in zip(a, b))


def _norm(a: list[float]) -> float:
    return math.sqrt(sum(x * x for x in a)) or 1.0


def _cosine(a: list[float], b: list[float]) -> float:
    return _dot(a, b) / (_norm(a) * _norm(b))


@dataclass
```

```
class _Row:
    chunk: Chunk
    vector: list[float]


class InMemoryVectorStore:
    """
    Minimal VectorStore for local testing.
    """

    def __init__(self):
        self._rows: list[_Row] = []

    def upsert(self, chunks: list[Chunk], vectors: list[list[float]]) ->
None:
        if len(chunks) != len(vectors):
            raise ValueError("chunks and vectors must have the same
length")

        # simple append; later we'll do true upsert by id
        for c, v in zip(chunks, vectors):
            self._rows.append(_Row(chunk=c, vector=v))

    def query(
        self,
        query_vector: list[float],
        k: int,
        filters: Optional[dict] = None,
    ) -> list[SearchResult]:
        # filters are ignored in this toy implementation
        scored = [
            SearchResult(chunk=row.chunk, score=_cosine(query_vector,
row.vector))
            for row in self._rows
        ]
        scored.sort(key=lambda r: r.score, reverse=True)
        return scored[:k]
```

3.3.3 Dummy LLM:

Create rag_in_a_box/adapters/llm/dummy_llm.py:

```
from __future__ import annotations

from rag_in_a_box.core.models import ChatMessage


class DummyLLM:
    """
    Fake LLM for smoke testing.
    It just echoes the question and shows that it received context.
```

```
    """

    def chat(self, messages: list[ChatMessage], **kwargs) -> str:
        user_msg = next((m.content for m in reversed(messages) if m.role
== "user"), "")
        return (
            "DUMMY ANSWER\n"
            "------------\n"
            "I received your prompt. Here is the final user message:\n\n"
            f"{user_msg}\n"
        )
```

3.4 Minimal RAG chat engine(core pipeline piece)

Create rag_in_a_box/adapters/pipelines/chat.py:

```
from __future__ import annotations

from rag_in_a_box.core.interfaces import Embedder, LLM, VectorStore
from rag_in_a_box.core.models import ChatMessage, ChatResponse


class RAGChatEngine:
    def __init__(self, *, embedder: Embedder, vectorstore: VectorStore,
llm: LLM, top_k: int = 5):
        self.embedder = embedder
        self.vectorstore = vectorstore
        self.llm = llm
        self.top_k = top_k

    def answer(self, question: str) -> ChatResponse:
        # 1) embed the query
        qvec = self.embedder.embed_texts([question])[0]

        # 2) retrieve
        results = self.vectorstore.query(qvec, k=self.top_k)

        # 3) build context
        context_blocks = []
        for i, r in enumerate(results, start=1):
            uri = r.chunk.uri
            context_blocks.append(f"[{i}] {uri}\n{r.chunk.text}")

        context = "\n\n".join(context_blocks) if context_blocks else "(no
context found)"

        # 4) prompt + generate
        messages = [
            ChatMessage(role="system", content="You are a helpful RAG
assistant. Use provided context when relevant."),
            ChatMessage(
```

```
                role="user",
                content=(
                    "Answer the question using the context.\n\n"
                    f"CONTEXT:\n{context}\n\n"
                    f"QUESTION:\n{question}"
                ),
            ),
        ]

        answer = self.llm.chat(messages)
        return ChatResponse(answer=answer, sources=results)
```

3.5 Smoke test

Create rag_in_a_box/adapters/smoke_step3.py:

```
from __future__ import annotations

import uuid

from rag_in_a_box.adapters.embeddings.dummy_embedder import
DummyHashEmbedder
from rag_in_a_box.adapters.llm.dummy_llm import DummyLLM
from rag_in_a_box.adapters.vectorstores.in_memory import
InMemoryVectorStore
from rag_in_a_box.core.models import Chunk
from rag_in_a_box.pipelines.chat import RAGChatEngine


def _cid() -> str:
    return str(uuid.uuid4())


def main() -> None:
    embedder = DummyHashEmbedder(dim=64)
    store = InMemoryVectorStore()
    llm = DummyLLM()

    # Pretend these are produced by extractor+chunker (we'll build those
later)
    chunks = [
        Chunk(id=_cid(), document_id="doc1", uri="local://doc1", text="RAG
combines retrieval with generation."),
        Chunk(id=_cid(), document_id="doc1", uri="local://doc1",
text="Azure AI Search can be used as a vector database."),
        Chunk(id=_cid(), document_id="doc2", uri="https://example.com",
text="Gradio can host a chat UI for LLM apps."),
    ]
    vectors = embedder.embed_texts([c.text for c in chunks])
    store.upsert(chunks, vectors)
```

```
    engine = RAGChatEngine(embedder=embedder, vectorstore=store, llm=llm,
top_k=2)

    q = "What can be used as the vector database?"
    resp = engine.answer(q)

    print("QUESTION:", q)
    print("\nANSWER:\n", resp.answer)
    print("\nSOURCES:")
    for r in resp.sources:
        print(f"- score={r.score:.3f} uri={r.chunk.uri} text=
{r.chunk.text}")


if __name__ == "__main__":
    main()
```

Step 4: Replace dummy retrieval with real Azure(Embeddings + Vectorstore AI-Search)

We will implement these real components:

- Embedder(Azure OpenAI, default *text-embedding-3-small*)
- LLM: Azure OpenAI
- Vectorstore: Azure AI Search
- Retriever: embed query -> vector search -> return ranked sources

Plus Smoke test

Add dependencies + check .env keys:

```
uv add openai azure-search-documents
```

Step 4.1: Azure OpenAI Embedder

create: rag_in_a_box/adapters/embeddings/azure_openai.py

```
from __future__ import annotations

from dataclasses import dataclass
from openai import OpenAI


@dataclass
class AzureOpenAIEmbedder:
```

```
    endpoint: str
    api_key: str
    deployment: str

    def __post_init__(self) -> None:
        # Azure OpenAI: base_url ends with /openai/v1/
:contentReference[oaicite:2]{index=2}
        self._client = OpenAI(
            api_key=self.api_key,
            base_url=f"{self.endpoint.rstrip('/')}/openai/v1/",
        )

    def embed_texts(self, texts: list[str]) -> list[list[float]]:
        # Azure OpenAI: model=deployment_name :contentReference[oaicite:3]
{index=3}
        resp = self._client.embeddings.create(model=self.deployment,
input=texts)
        return [d.embedding for d in resp.data]

    def embedding_dim(self) -> int:
        # Avoid guessing dimensions: ask the API once
        return len(self.embed_texts(["dim probe"])[0])
```

Step 4.2: Azure OpenAI LLM (chat)

create: rag_in_a_box/adapters/llm/azure_openai.py

```
from __future__ import annotations

from dataclasses import dataclass
from openai import OpenAI

from rag_in_a_box.core.models import ChatMessage


@dataclass
class AzureOpenAIChatLLM:
    endpoint: str
    api_key: str
    deployment: str

    def __post_init__(self) -> None:
        self._client = OpenAI(
            api_key=self.api_key,
            base_url=f"{self.endpoint.rstrip('/')}/openai/v1/",
        )

    def chat(self, messages: list[ChatMessage], **kwargs) -> str:
        # Azure OpenAI: model=deployment_name :contentReference[oaicite:5]
{index=5}
```

```
        resp = self._client.chat.completions.create(
            model=self.deployment,
            messages=[{"role": m.role, "content": m.content} for m in
messages],
            temperature=kwargs.get("temperature", 0.2),
        )
        return resp.choices[0].message.content or ""
```

Step 4.3: Azure AI Search Vectorstore

create: rag_in_a_box/adapters/vectorstores/azure_ai_search.py

```
from __future__ import annotations

from dataclasses import dataclass
from typing import Optional

from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import (
    SearchIndex,
    SearchField,
    SearchFieldDataType,
    SearchableField,
    SimpleField,
    VectorSearch,
    VectorSearchProfile,
    HnswAlgorithmConfiguration,
)
from azure.search.documents.models import VectorizedQuery

from rag_in_a_box.core.models import Chunk, SearchResult


VECTOR_PROFILE = "raginabox-vector-profile"
HNSW_CONFIG = "raginabox-hnsw"


@dataclass
class AzureAISearchVectorStore:
    endpoint: str
    api_key: str
    index_name: str
    vector_dim: int

    def __post_init__(self) -> None:
        cred = AzureKeyCredential(self.api_key)
        self._index_client = SearchIndexClient(self.endpoint, cred)
        self._search_client = SearchClient(self.endpoint, self.index_name,
```

```
cred)

    def ensure_index(self) -> None:
        # Vector index creation pattern matches Azure vector search
quickstart :contentReference[oaicite:6]{index=6}
        fields = [
            SimpleField(name="id", type=SearchFieldDataType.String,
key=True),
            SimpleField(name="document_id",
type=SearchFieldDataType.String, filterable=True),
            SimpleField(name="uri", type=SearchFieldDataType.String,
filterable=True),
            SimpleField(name="chunk_index",
type=SearchFieldDataType.Int32, filterable=True),
            SearchableField(name="content",
type=SearchFieldDataType.String),
            SearchField(
                name="content_vector",

type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
                searchable=True,
                vector_search_dimensions=self.vector_dim,
                vector_search_profile_name=VECTOR_PROFILE,
            ),
        ]

        vector_search = VectorSearch(
            algorithms=[HnswAlgorithmConfiguration(name=HNSW_CONFIG)],
            profiles=[VectorSearchProfile(name=VECTOR_PROFILE,
algorithm_configuration_name=HNSW_CONFIG)],
        )

        index = SearchIndex(name=self.index_name, fields=fields,
vector_search=vector_search)
        self._index_client.create_or_update_index(index)

    def upsert(self, chunks: list[Chunk], vectors: list[list[float]]) ->
None:
        if len(chunks) != len(vectors):
            raise ValueError("chunks and vectors must have the same
length")

        docs = []
        for i, (c, v) in enumerate(zip(chunks, vectors)):
            docs.append(
                {
                    "id": c.id,
                    "document_id": c.document_id,
                    "uri": c.uri,
                    "chunk_index": c.metadata.get("chunk_index", i),
                    "content": c.text,
                    "content_vector": v,
                }
            )
```

```
        self._search_client.upload_documents(documents=docs)

    def query(self, query_vector: list[float], k: int, filters:
Optional[dict] = None) -> list[SearchResult]:
        # Vector query pattern matches quickstart: VectorizedQuery +
vector_queries=[...] :contentReference[oaicite:7]{index=7}
        vq = VectorizedQuery(vector=query_vector, k_nearest_neighbors=k,
fields="content_vector", kind="vector")
        results = self._search_client.search(
            vector_queries=[vq],
            select=["id", "document_id", "uri", "chunk_index", "content"],
            top=k,
        )

        out: list[SearchResult] = []
        for r in results:
            chunk = Chunk(
                id=r["id"],
                document_id=r["document_id"],
                uri=r["uri"],
                text=r["content"],
                metadata={"chunk_index": r.get("chunk_index")},
            )
            out.append(SearchResult(chunk=chunk,
score=r.get("@search.score", 0.0)))
        return out
```

Step 4.4: Wire into RAGChatEngine (nothing todo!)

Important: Our RAGChatEngine from Step 3 already depends only on interfaces. This composability means that we do **NOT** have to do anything!!!

Step 4.5: Smoke test

create: rag_in_a_box/smoke_test_step4_azure.py

```
from __future__ import annotations

import uuid

from rag_in_a_box.config.settings import Settings
from rag_in_a_box.adapters.embeddings.azure_openai import
AzureOpenAIEmbedder
from rag_in_a_box.adapters.llm.azure_openai import AzureOpenAIChatLLM
from rag_in_a_box.adapters.vectorstores.azure_ai_search import
AzureAISearchVectorStore
from rag_in_a_box.core.models import Chunk
from rag_in_a_box.pipelines.chat import RAGChatEngine
```

```
def _id() -> str:
    return str(uuid.uuid4())


def main() -> None:
    s = Settings()

    embedder = AzureOpenAIEmbedder(
        endpoint=s.azure_openai_endpoint,
        api_key=s.azure_openai_api_key,
        deployment=s.azure_openai_embedding_deployment,
    )

    vector_dim = embedder.embedding_dim()  # don't guess dimensions
    store = AzureAISearchVectorStore(
        endpoint=s.azure_search_endpoint,
        api_key=s.azure_search_api_key,
        index_name=s.azure_search_index,
        vector_dim=vector_dim,
    )
    store.ensure_index()

    llm = AzureOpenAIChatLLM(
        endpoint=s.azure_openai_endpoint,
        api_key=s.azure_openai_api_key,
        deployment=s.azure_openai_chat_deployment,
    )

    # Index a few chunks
    chunks = [
        Chunk(id=_id(), document_id="demo", uri="local://demo",
text="Azure AI Search can be used as a vector database.", metadata=
{"chunk_index": 0}),
        Chunk(id=_id(), document_id="demo", uri="local://demo",
text="Gradio can host a chat UI for LLM apps.", metadata={"chunk_index":
1}),
    ]
    vectors = embedder.embed_texts([c.text for c in chunks])
    store.upsert(chunks, vectors)

    # Ask a question
    engine = RAGChatEngine(embedder=embedder, vectorstore=store, llm=llm,
top_k=s.top_k)
    resp = engine.answer("What can be used as the vector database?")

    print(resp.answer)
    print("\nSOURCES:")
    for r in resp.sources:
        print(f"- score={r.score:.3f} uri={r.chunk.uri} text=
{r.chunk.text}")


if __name__ == "__main__":
```

```
    main()
```

```
#Run the test
uv run python -m rag_in_a_box.smoke_step4_azure
```

Step 5: Build Ingestion pipeline (local-only, no web yet)

Goal:

`SourceItem -> Document -> Chunks -> Embeddings -> VectorStore.upsert()`

We will build:

- Chunker: rag_in_a_box/adaptors/chunking/simple_chunker.py
- Extractors: rag_in_a_box/adaptors/extractors
  - TextExtractor(txt,md)
  - PdfExtractors(pypdf)
  - htmlExtractor
- Local folder source: rag_in_a_box/adaptors/local_folder.py
- Ingestion Orchestrator: rag_in_a_box/pipelines/ingestion.py
- cli command: rag_in_a_box/cli.py

```
# Run test
uv run python -m rag_in_a_box.cli ingest-local
```

Step 6: Website Ingestion

Step 6.1: Add extra keys in .env

```
INGEST_START_URLS=["https://fluvius.be/nl", "https://site2.com/docs"]
INGEST_ALLOWED_DOMAINS=["fluvius.be", "site2.com"]
## ToDo INGEST_EXCLUDED_URLS=["https://fluvius.be/nl/exclude-this-page"]
CRAWL_EXCLUDED_PREFIXES=["https://fluvius.be/fr, https://www.fluvius.be/fr
https://fluvius.be/en, https://www.fluvius.be/en]

WEB_MAX_PAGES=200
WEB_MAX_DEPTH=3
WEB_TIMEOUT_SECONDS=20
WEB_USER_AGENT=RagInABoxBot/0.1
WEB_RESPECT_ROBOTS=false
```

Step 6.x Ingest

```
uv run python -m rag_in_a_box.cli ingest-web
```