perplexity

RAG System Implementation: Vector Store, Retrieval, Generation, and Evaluation Components

Based on your progress with document processing, chunking, and embedding components, I'll provide you with the complete implementations for the remaining four core components of your RAG system.

Step 4: Vector Store Implementation

4.1 Create Vector Store Base Class

Create a base class for vector stores in `src/vector_store/base.py`:

```
from abc import ABC, abstractmethod
from typing import List, Dict, Any, Optional, Tuple
import numpy as np

from src.embedding.base import EmbeddingResult
from src.chunking.base import Chunk


class VectorStoreResult:
    """Class representing a vector search result."""

    def __init__(
        self,
        chunk_id: str,
        score: float,
        chunk: Optional[Chunk] = None,
        metadata: Optional[Dict[str, Any]] = None
    ):
        """
        Initialize a vector store result.

        Args:
            chunk_id: ID of the retrieved chunk
            score: Similarity/relevance score
            chunk: The actual chunk object (if available)
            metadata: Additional metadata about the result
        """
        self.chunk_id = chunk_id
        self.score = score
        self.chunk = chunk
        self.metadata = metadata or {}

    def __repr__(self) -> str:
        return f"VectorStoreResult(chunk_id={self.chunk_id}, score={self.score:.4f})"
```

```
class BaseVectorStore(ABC):
    """Base class for vector store implementations."""

    @abstractmethod
    def add_embeddings(self, embeddings: List[EmbeddingResult]) -> None:
        """
        Add embeddings to the vector store.

        Args:
            embeddings: List of embedding results to add
        """
        pass

    @abstractmethod
    def search(
        self,
        query_embedding: np.ndarray,
        top_k: int = 5,
        filters: Optional[Dict[str, Any]] = None
    ) -> List[VectorStoreResult]:
        """
        Search for similar vectors.

        Args:
            query_embedding: Query vector to search for
            top_k: Number of results to return
            filters: Optional metadata filters

        Returns:
            List of search results
        """
        pass

    @abstractmethod
    def delete_by_document_id(self, doc_id: str) -> bool:
        """
        Delete all chunks belonging to a document.

        Args:
            doc_id: Document ID to delete

        Returns:
            True if deletion was successful
        """
        pass

    @abstractmethod
    def get_collection_info(self) -> Dict[str, Any]:
        """
        Get information about the vector store collection.

        Returns:
            Dictionary containing collection metadata
```

```
        """
        pass
```

4.2 Implement Chroma Vector Store

Create a Chroma implementation in `src/vector_store/chroma_store.py`:

```
import json
import logging
from typing import List, Dict, Any, Optional
import numpy as np
from datetime import datetime
import chromadb
from chromadb.config import Settings

from src.vector_store.base import BaseVectorStore, VectorStoreResult
from src.embedding.base import EmbeddingResult
from src.chunking.base import Chunk

logger = logging.getLogger(__name__)


class ChromaVectorStore(BaseVectorStore):
    """Chroma vector store implementation."""

    def __init__(
        self,
        collection_name: str = "rag_documents",
        persist_directory: str = "./data/chroma_db",
        distance_function: str = "cosine"
    ):
        """
        Initialize Chroma vector store.

        Args:
            collection_name: Name of the collection
            persist_directory: Directory to persist the database
            distance_function: Distance function to use ('cosine', 'euclidean', 'manhatta
        """
        self.collection_name = collection_name
        self.persist_directory = persist_directory
        self.distance_function = distance_function

        try:
            # Initialize Chroma client
            self.client = chromadb.PersistentClient(
                path=persist_directory,
                settings=Settings(anonymized_telemetry=False)
            )

            # Get or create collection
            self.collection = self.client.get_or_create_collection(
                name=collection_name,
                metadata={"hnsw:space": distance_function}
            )
```

```
            logger.info(f"Initialized ChromaVectorStore with collection '{collection_name

        except Exception as e:
            logger.error(f"Failed to initialize ChromaVectorStore: {str(e)}")
            raise

    def add_embeddings(self, embeddings: List[EmbeddingResult]) -> None:
        """Add embeddings to Chroma collection."""
        try:
            if not embeddings:
                logger.warning("No embeddings provided to add")
                return

            # Prepare data for Chroma
            ids = []
            vectors = []
            metadatas = []
            documents = []

            for emb in embeddings:
                ids.append(emb.chunk_id)
                vectors.append(emb.embedding.tolist())

                # Prepare metadata (Chroma requires string values)
                metadata = {}
                for key, value in emb.metadata.items():
                    if isinstance(value, (str, int, float, bool)):
                        metadata[key] = value
                    else:
                        metadata[key] = json.dumps(value)

                metadata['added_at'] = datetime.now().isoformat()
                metadatas.append(metadata)

                # Use chunk text if available, otherwise use chunk_id
                documents.append(
                    getattr(emb, 'chunk_text', emb.chunk_id)
                )

            # Add to collection
            self.collection.add(
                ids=ids,
                embeddings=vectors,
                metadatas=metadatas,
                documents=documents
            )

            logger.info(f"Added {len(embeddings)} embeddings to Chroma collection")

        except Exception as e:
            logger.error(f"Error adding embeddings to Chroma: {str(e)}")
            raise

    def search(
        self,
```

```
        query_embedding: np.ndarray,
        top_k: int = 5,
        filters: Optional[Dict[str, Any]] = None
    ) -> List[VectorStoreResult]:
        """Search Chroma collection for similar vectors."""
        try:
            # Prepare query
            query_vector = query_embedding.tolist()

            # Prepare where clause for filtering
            where_clause = None
            if filters:
                where_clause = {}
                for key, value in filters.items():
                    if isinstance(value, list):
                        where_clause[key] = {"$in": value}
                    else:
                        where_clause[key] = {"$eq": value}

            # Query collection
            results = self.collection.query(
                query_embeddings=[query_vector],
                n_results=top_k,
                where=where_clause,
                include=["metadatas", "documents", "distances"]
            )

            # Process results
            search_results = []
            if results['ids'] and results['ids'][^0]:
                for i, chunk_id in enumerate(results['ids'][^0]):
                    # Convert distance to similarity score (for cosine distance)
                    distance = results['distances'][^0][i]
                    if self.distance_function == "cosine":
                        score = 1.0 - distance
                    else:
                        score = 1.0 / (1.0 + distance)  # Convert distance to similarity

                    # Get metadata
                    metadata = results['metadatas'][^0][i] if results['metadatas'][^0] el
                    document = results['documents'][^0][i] if results['documents'][^0] el

                    # Create chunk object from stored data
                    chunk = None
                    if document and metadata:
                        chunk = Chunk(
                            text=document,
                            doc_id=metadata.get('doc_id', ''),
                            chunk_id=chunk_id,
                            metadata=metadata
                        )

                    search_results.append(VectorStoreResult(
                        chunk_id=chunk_id,
                        score=score,
                        chunk=chunk,
```

```
                        metadata=metadata
                    ))

            logger.debug(f"Found {len(search_results)} results for query")
            return search_results

        except Exception as e:
            logger.error(f"Error searching Chroma collection: {str(e)}")
            raise

    def delete_by_document_id(self, doc_id: str) -> bool:
        """Delete all chunks belonging to a document."""
        try:
            # Query for chunks with the specified doc_id
            results = self.collection.get(
                where={"doc_id": {"$eq": doc_id}},
                include=["metadatas"]
            )

            if results['ids']:
                # Delete the chunks
                self.collection.delete(ids=results['ids'])
                logger.info(f"Deleted {len(results['ids'])} chunks for document {doc_id}'
                return True
            else:
                logger.info(f"No chunks found for document {doc_id}")
                return False

        except Exception as e:
            logger.error(f"Error deleting document {doc_id} from Chroma: {str(e)}")
            return False

    def get_collection_info(self) -> Dict[str, Any]:
        """Get information about the Chroma collection."""
        try:
            count = self.collection.count()
            return {
                "name": self.collection_name,
                "count": count,
                "distance_function": self.distance_function,
                "persist_directory": self.persist_directory
            }
        except Exception as e:
            logger.error(f"Error getting collection info: {str(e)}")
            return {"error": str(e)}
```

4.3 Implement Qdrant Vector Store

Create a Qdrant implementation in `src/vector_store/qdrant_store.py`:

```
import json
import logging
from typing import List, Dict, Any, Optional
import numpy as np
from datetime import datetime
```

```
import uuid
from qdrant_client import QdrantClient
from qdrant_client.models import (
    Distance, VectorParams, PointStruct,
    Filter, FieldCondition, MatchValue
)

from src.vector_store.base import BaseVectorStore, VectorStoreResult
from src.embedding.base import EmbeddingResult
from src.chunking.base import Chunk

logger = logging.getLogger(__name__)


class QdrantVectorStore(BaseVectorStore):
    """Qdrant vector store implementation."""

    def __init__(
        self,
        collection_name: str = "rag_documents",
        url: str = "http://localhost:6333",
        api_key: Optional[str] = None,
        vector_size: int = 384,
        distance: str = "Cosine"
    ):
        """
        Initialize Qdrant vector store.

        Args:
            collection_name: Name of the collection
            url: Qdrant server URL
            api_key: Optional API key for authentication
            vector_size: Dimension of vectors
            distance: Distance metric ('Cosine', 'Euclidean', 'Manhattan')
        """
        self.collection_name = collection_name
        self.url = url
        self.vector_size = vector_size

        try:
            # Initialize Qdrant client
            self.client = QdrantClient(
                url=url,
                api_key=api_key
            )

            # Map distance string to Qdrant enum
            distance_map = {
                "Cosine": Distance.COSINE,
                "Euclidean": Distance.EUCLID,
                "Manhattan": Distance.MANHATTAN
            }
            self.distance = distance_map.get(distance, Distance.COSINE)

            # Create collection if it doesn't exist
            self._ensure_collection_exists()
```

```
            logger.info(f"Initialized QdrantVectorStore with collection '{collection_name

        except Exception as e:
            logger.error(f"Failed to initialize QdrantVectorStore: {str(e)}")
            raise

    def _ensure_collection_exists(self):
        """Create collection if it doesn't exist."""
        try:
            # Check if collection exists
            collections = self.client.get_collections().collections
            collection_names = [col.name for col in collections]

            if self.collection_name not in collection_names:
                # Create collection
                self.client.create_collection(
                    collection_name=self.collection_name,
                    vectors_config=VectorParams(
                        size=self.vector_size,
                        distance=self.distance
                    )
                )
                logger.info(f"Created Qdrant collection '{self.collection_name}'")
            else:
                logger.info(f"Using existing Qdrant collection '{self.collection_name}'")

        except Exception as e:
            logger.error(f"Error ensuring collection exists: {str(e)}")
            raise

    def add_embeddings(self, embeddings: List[EmbeddingResult]) -> None:
        """Add embeddings to Qdrant collection."""
        try:
            if not embeddings:
                logger.warning("No embeddings provided to add")
                return

            # Prepare points for Qdrant
            points = []

            for emb in embeddings:
                # Create payload (metadata)
                payload = dict(emb.metadata)
                payload['chunk_id'] = emb.chunk_id
                payload['added_at'] = datetime.now().isoformat()

                # Ensure vector dimension matches
                vector = emb.embedding
                if len(vector) != self.vector_size:
                    logger.warning(
                        f"Vector size mismatch: expected {self.vector_size}, "
                        f"got {len(vector)} for chunk {emb.chunk_id}"
                    )
                    continue
```

```
                # Create point
                point = PointStruct(
                    id=str(uuid.uuid4()),  # Generate UUID for Qdrant
                    vector=vector.tolist(),
                    payload=payload
                )
                points.append(point)

            if points:
                # Upload points
                self.client.upsert(
                    collection_name=self.collection_name,
                    points=points
                )
                logger.info(f"Added {len(points)} embeddings to Qdrant collection")

        except Exception as e:
            logger.error(f"Error adding embeddings to Qdrant: {str(e)}")
            raise

    def search(
        self,
        query_embedding: np.ndarray,
        top_k: int = 5,
        filters: Optional[Dict[str, Any]] = None
    ) -> List[VectorStoreResult]:
        """Search Qdrant collection for similar vectors."""
        try:
            # Prepare filter
            filter_obj = None
            if filters:
                conditions = []
                for key, value in filters.items():
                    if isinstance(value, list):
                        # Handle list of values (OR condition)
                        for v in value:
                            conditions.append(
                                FieldCondition(key=key, match=MatchValue(value=v))
                            )
                    else:
                        conditions.append(
                            FieldCondition(key=key, match=MatchValue(value=value))
                        )

                if conditions:
                    filter_obj = Filter(must=conditions)

            # Search
            search_results = self.client.search(
                collection_name=self.collection_name,
                query_vector=query_embedding.tolist(),
                limit=top_k,
                query_filter=filter_obj,
                with_payload=True
            )
```

```
            # Process results
            results = []
            for result in search_results:
                chunk_id = result.payload.get('chunk_id', str(result.id))
                score = result.score

                # Create chunk object from payload
                chunk = None
                if 'doc_id' in result.payload:
                    chunk = Chunk(
                        text=result.payload.get('text', ''),
                        doc_id=result.payload['doc_id'],
                        chunk_id=chunk_id,
                        metadata=result.payload
                    )

                results.append(VectorStoreResult(
                    chunk_id=chunk_id,
                    score=score,
                    chunk=chunk,
                    metadata=result.payload
                ))

            logger.debug(f"Found {len(results)} results for query")
            return results

        except Exception as e:
            logger.error(f"Error searching Qdrant collection: {str(e)}")
            raise

    def delete_by_document_id(self, doc_id: str) -> bool:
        """Delete all chunks belonging to a document."""
        try:
            # Create filter for document ID
            filter_obj = Filter(
                must=[FieldCondition(key="doc_id", match=MatchValue(value=doc_id))]
            )

            # Delete points
            result = self.client.delete(
                collection_name=self.collection_name,
                points_selector=filter_obj
            )

            logger.info(f"Deleted chunks for document {doc_id}")
            return True

        except Exception as e:
            logger.error(f"Error deleting document {doc_id} from Qdrant: {str(e)}")
            return False

    def get_collection_info(self) -> Dict[str, Any]:
        """Get information about the Qdrant collection."""
        try:
            info = self.client.get_collection(self.collection_name)
            return {
```

```
                "name": self.collection_name,
                "count": info.points_count,
                "vector_size": self.vector_size,
                "distance": self.distance.name,
                "url": self.url
            }
        except Exception as e:
            logger.error(f"Error getting collection info: {str(e)}")
            return {"error": str(e)}
```

4.4 Vector Store Factory

Create a factory in `src/vector_store/factory.py`:

```
from typing import Dict, Type, Optional
import logging

from src.vector_store.base import BaseVectorStore
from src.vector_store.chroma_store import ChromaVectorStore
from src.vector_store.qdrant_store import QdrantVectorStore

logger = logging.getLogger(__name__)


class VectorStoreFactory:
    """Factory for creating vector store instances."""

    def __init__(self, config=None):
        """
        Initialize vector store factory.

        Args:
            config: Optional configuration dictionary
        """
        self.config = config or {}
        self._stores: Dict[str, Type[BaseVectorStore]] = {}

        # Register default stores
        self.register_store("chroma", ChromaVectorStore)
        self.register_store("qdrant", QdrantVectorStore)

    def register_store(self, name: str, store_class: Type[BaseVectorStore]) -> None:
        """Register a vector store class."""
        self._stores[name.lower()] = store_class
        logger.debug(f"Registered vector store {store_class.__name__} as '{name}'")

    def get_store(self, store_type: str) -> Optional[BaseVectorStore]:
        """
        Get a vector store instance.

        Args:
            store_type: Type of vector store to create

        Returns:
            Configured vector store instance or None
```

```
        """
        store_type = store_type.lower()
        store_class = self._stores.get(store_type)

        if not store_class:
            logger.error(f"No vector store found for type '{store_type}'")
            return None

        # Get configuration for this store type
        store_config = {}
        if self.config and "options" in self.config:
            store_config = self.config.get("options", {}).get(store_type, {})

        try:
            return store_class(**store_config)
        except Exception as e:
            logger.error(f"Error creating vector store '{store_type}': {str(e)}")
            return None

    def get_default_store(self) -> BaseVectorStore:
        """Get the default vector store as specified in config."""
        default_type = "chroma"
        if self.config:
            default_type = self.config.get("default", default_type)

        store = self.get_store(default_type)
        if not store:
            logger.warning(f"Default store '{default_type}' failed. Falling back to 'chro
            store = self.get_store("chroma")

            if not store:
                raise RuntimeError("Could not create any vector store")

        return store
```

Step 5: Retrieval Implementation

5.1 Create Retrieval Base Class

Create a base class in `src/retrieval/base.py`:

```
from abc import ABC, abstractmethod
from typing import List, Dict, Any, Optional
import numpy as np

from src.vector_store.base import VectorStoreResult


class RetrievalResult:
    """Class representing a retrieval result."""

    def __init__(
        self,
        query: str,
```

```
        results: List[VectorStoreResult],
        metadata: Optional[Dict[str, Any]] = None
    ):
        """
        Initialize a retrieval result.

        Args:
            query: The original query
            results: List of retrieved results
            metadata: Additional metadata about the retrieval
        """
        self.query = query
        self.results = results
        self.metadata = metadata or {}

    def get_context(self) -> str:
        """Get concatenated context from all results."""
        contexts = []
        for result in self.results:
            if result.chunk and result.chunk.text:
                contexts.append(result.chunk.text)
        return "\n\n".join(contexts)

    def __len__(self) -> int:
        return len(self.results)

    def __repr__(self) -> str:
        return f"RetrievalResult(query='{self.query[:50]}...', results={len(self.results)}


class BaseRetriever(ABC):
    """Base class for retrieval strategies."""

    @abstractmethod
    def retrieve(
        self,
        query: str,
        top_k: int = 5,
        filters: Optional[Dict[str, Any]] = None
    ) -> RetrievalResult:
        """
        Retrieve relevant documents for a query.

        Args:
            query: The search query
            top_k: Number of results to return
            filters: Optional metadata filters

        Returns:
            Retrieval result object
        """
        pass
```

5.2 Implement Semantic Retriever

Create a semantic retrieval strategy in `src/retrieval/semantic_retriever.py`:

```
import logging
from typing import Dict, Any, Optional
from datetime import datetime

from src.retrieval.base import BaseRetriever, RetrievalResult
from src.vector_store.base import BaseVectorStore
from src.embedding.base import BaseEmbedder

logger = logging.getLogger(__name__)


class SemanticRetriever(BaseRetriever):
    """Semantic retrieval using vector similarity search."""

    def __init__(
        self,
        vector_store: BaseVectorStore,
        embedder: BaseEmbedder,
        similarity_threshold: float = 0.0
    ):
        """
        Initialize semantic retriever.

        Args:
            vector_store: Vector store instance
            embedder: Embedder instance for query encoding
            similarity_threshold: Minimum similarity score threshold
        """
        self.vector_store = vector_store
        self.embedder = embedder
        self.similarity_threshold = similarity_threshold

    def retrieve(
        self,
        query: str,
        top_k: int = 5,
        filters: Optional[Dict[str, Any]] = None
    ) -> RetrievalResult:
        """Retrieve documents using semantic similarity."""
        start_time = datetime.now()

        try:
            # Generate query embedding
            query_embedding = self.embedder.embed_query(query)

            # Search vector store
            search_results = self.vector_store.search(
                query_embedding=query_embedding,
                top_k=top_k,
                filters=filters
            )
```

```
            # Filter by similarity threshold
            filtered_results = [
                result for result in search_results
                if result.score >= self.similarity_threshold
            ]

            # Calculate retrieval time
            retrieval_time = (datetime.now() - start_time).total_seconds()

            # Create metadata
            metadata = {
                "strategy": "semantic",
                "retrieval_time": retrieval_time,
                "total_candidates": len(search_results),
                "filtered_results": len(filtered_results),
                "similarity_threshold": self.similarity_threshold,
                "timestamp": datetime.now().isoformat()
            }

            logger.debug(
                f"Semantic retrieval: {len(filtered_results)} results "
                f"in {retrieval_time:.3f}s for query: {query[:100]}"
            )

            return RetrievalResult(
                query=query,
                results=filtered_results,
                metadata=metadata
            )

        except Exception as e:
            logger.error(f"Error in semantic retrieval: {str(e)}")
            raise
```

5.3 Implement Hybrid Retriever

Create a hybrid retrieval strategy in `src/retrieval/hybrid_retriever.py`:

```
import logging
from typing import Dict, Any, Optional, List
from datetime import datetime
import re
from collections import Counter

from src.retrieval.base import BaseRetriever, RetrievalResult
from src.vector_store.base import BaseVectorStore, VectorStoreResult
from src.embedding.base import BaseEmbedder

logger = logging.getLogger(__name__)


class HybridRetriever(BaseRetriever):
    """Hybrid retrieval combining semantic and keyword-based search."""

    def __init__(
```

```
        self,
        vector_store: BaseVectorStore,
        embedder: BaseEmbedder,
        semantic_weight: float = 0.7,
        keyword_weight: float = 0.3,
        similarity_threshold: float = 0.0
    ):
        """
        Initialize hybrid retriever.

        Args:
            vector_store: Vector store instance
            embedder: Embedder instance for query encoding
            semantic_weight: Weight for semantic similarity scores
            keyword_weight: Weight for keyword matching scores
            similarity_threshold: Minimum similarity score threshold
        """
        self.vector_store = vector_store
        self.embedder = embedder
        self.semantic_weight = semantic_weight
        self.keyword_weight = keyword_weight
        self.similarity_threshold = similarity_threshold

        # Normalize weights
        total_weight = semantic_weight + keyword_weight
        if total_weight > 0:
            self.semantic_weight = semantic_weight / total_weight
            self.keyword_weight = keyword_weight / total_weight

    def _extract_keywords(self, text: str) -> List[str]:
        """Extract keywords from text."""
        # Simple keyword extraction (remove stopwords, punctuation)
        stopwords = {
            'a', 'an', 'and', 'are', 'as', 'at', 'be', 'by', 'for', 'from',
            'has', 'he', 'in', 'is', 'it', 'its', 'of', 'on', 'that', 'the',
            'to', 'was', 'will', 'with', 'what', 'when', 'where', 'who', 'how'
        }

        # Clean and tokenize
        words = re.findall(r'\b[a-zA-Z]+\b', text.lower())
        keywords = [word for word in words if word not in stopwords and len(word) > 2]

        return keywords

    def _calculate_keyword_score(self, query_keywords: List[str], chunk_text: str) -> flo
        """Calculate keyword matching score using TF-IDF-like approach."""
        if not query_keywords or not chunk_text:
            return 0.0

        chunk_keywords = self._extract_keywords(chunk_text)
        if not chunk_keywords:
            return 0.0

        # Calculate keyword frequencies
        chunk_freq = Counter(chunk_keywords)
        query_freq = Counter(query_keywords)
```

```
        # Calculate score based on keyword overlap
        score = 0.0
        for keyword in query_keywords:
            if keyword in chunk_freq:
                # TF-IDF-like score: term frequency * inverse document frequency
                tf = chunk_freq[keyword] / len(chunk_keywords)
                # Simplified IDF (could be improved with corpus statistics)
                idf = 1.0 + (1.0 / (1.0 + chunk_freq[keyword]))
                score += tf * idf * query_freq[keyword]

        # Normalize by query length
        return score / len(query_keywords)

    def retrieve(
        self,
        query: str,
        top_k: int = 5,
        filters: Optional[Dict[str, Any]] = None
    ) -> RetrievalResult:
        """Retrieve documents using hybrid approach."""
        start_time = datetime.now()

        try:
            # Step 1: Semantic retrieval (get more candidates for reranking)
            initial_k = min(top_k * 3, 50)  # Get more candidates
            query_embedding = self.embedder.embed_query(query)

            semantic_results = self.vector_store.search(
                query_embedding=query_embedding,
                top_k=initial_k,
                filters=filters
            )

            # Step 2: Extract query keywords
            query_keywords = self._extract_keywords(query)

            # Step 3: Calculate hybrid scores
            hybrid_results = []
            for result in semantic_results:
                if not result.chunk or not result.chunk.text:
                    continue

                # Get semantic score
                semantic_score = result.score

                # Calculate keyword score
                keyword_score = self._calculate_keyword_score(
                    query_keywords,
                    result.chunk.text
                )

                # Combine scores
                hybrid_score = (
                    self.semantic_weight * semantic_score +
                    self.keyword_weight * keyword_score
```

```
            )

            # Create new result with hybrid score
            hybrid_result = VectorStoreResult(
                chunk_id=result.chunk_id,
                score=hybrid_score,
                chunk=result.chunk,
                metadata={
                    **result.metadata,
                    'semantic_score': semantic_score,
                    'keyword_score': keyword_score,
                    'hybrid_score': hybrid_score
                }
            )
            hybrid_results.append(hybrid_result)

        # Step 4: Sort by hybrid score and take top_k
        hybrid_results.sort(key=lambda x: x.score, reverse=True)
        final_results = hybrid_results[:top_k]

        # Step 5: Filter by similarity threshold
        filtered_results = [
            result for result in final_results
            if result.score >= self.similarity_threshold
        ]

        # Calculate retrieval time
        retrieval_time = (datetime.now() - start_time).total_seconds()

        # Create metadata
        metadata = {
            "strategy": "hybrid",
            "retrieval_time": retrieval_time,
            "semantic_weight": self.semantic_weight,
            "keyword_weight": self.keyword_weight,
            "query_keywords": query_keywords,
            "initial_candidates": len(semantic_results),
            "final_results": len(filtered_results),
            "similarity_threshold": self.similarity_threshold,
            "timestamp": datetime.now().isoformat()
        }

        logger.debug(
            f"Hybrid retrieval: {len(filtered_results)} results "
            f"in {retrieval_time:.3f}s for query: {query[:100]}"
        )

        return RetrievalResult(
            query=query,
            results=filtered_results,
            metadata=metadata
        )

    except Exception as e:
        logger.error(f"Error in hybrid retrieval: {str(e)}")
        raise
```

5.4 Implement Reranking Retriever

Create a reranking strategy in `src/retrieval/reranking_retriever.py`:

```
import logging
from typing import Dict, Any, Optional, List
from datetime import datetime
import torch
from sentence_transformers import CrossEncoder

from src.retrieval.base import BaseRetriever, RetrievalResult
from src.vector_store.base import BaseVectorStore, VectorStoreResult
from src.embedding.base import BaseEmbedder

logger = logging.getLogger(__name__)


class RerankingRetriever(BaseRetriever):
    """Retrieval with cross-encoder reranking for improved relevance."""

    def __init__(
        self,
        vector_store: BaseVectorStore,
        embedder: BaseEmbedder,
        rerank_model: str = "cross-encoder/ms-marco-MiniLM-L-6-v2",
        initial_k: int = 20,
        similarity_threshold: float = 0.0,
        device: Optional[str] = None
    ):
        """
        Initialize reranking retriever.

        Args:
            vector_store: Vector store instance
            embedder: Embedder instance for query encoding
            rerank_model: Cross-encoder model for reranking
            initial_k: Number of candidates to retrieve before reranking
            similarity_threshold: Minimum similarity score threshold
            device: Device to run reranking model on
        """
        self.vector_store = vector_store
        self.embedder = embedder
        self.initial_k = initial_k
        self.similarity_threshold = similarity_threshold

        try:
            # Initialize cross-encoder for reranking
            self.cross_encoder = CrossEncoder(rerank_model, device=device)
            logger.info(f"Initialized cross-encoder: {rerank_model}")
        except Exception as e:
            logger.error(f"Failed to initialize cross-encoder: {str(e)}")
            self.cross_encoder = None

    def retrieve(
        self,
        query: str,
```

```
        top_k: int = 5,
        filters: Optional[Dict[str, Any]] = None
    ) -> RetrievalResult:
        """Retrieve documents with reranking."""
        start_time = datetime.now()

        try:
            # Step 1: Initial semantic retrieval
            query_embedding = self.embedder.embed_query(query)

            initial_results = self.vector_store.search(
                query_embedding=query_embedding,
                top_k=self.initial_k,
                filters=filters
            )

            if not initial_results:
                logger.warning("No initial results found")
                return RetrievalResult(
                    query=query,
                    results=[],
                    metadata={"strategy": "reranking", "error": "no_initial_results"}
                )

            # Step 2: Reranking with cross-encoder (if available)
            if self.cross_encoder:
                reranked_results = self._rerank_with_cross_encoder(
                    query, initial_results
                )
            else:
                logger.warning("Cross-encoder not available, using semantic scores")
                reranked_results = initial_results

            # Step 3: Take top_k results
            final_results = reranked_results[:top_k]

            # Step 4: Filter by similarity threshold
            filtered_results = [
                result for result in final_results
                if result.score >= self.similarity_threshold
            ]

            # Calculate retrieval time
            retrieval_time = (datetime.now() - start_time).total_seconds()

            # Create metadata
            metadata = {
                "strategy": "reranking",
                "retrieval_time": retrieval_time,
                "rerank_model": getattr(self.cross_encoder, 'model_name', 'none'),
                "initial_candidates": len(initial_results),
                "final_results": len(filtered_results),
                "similarity_threshold": self.similarity_threshold,
                "timestamp": datetime.now().isoformat()
            }
```

```
            logger.debug(
                f"Reranking retrieval: {len(filtered_results)} results "
                f"in {retrieval_time:.3f}s for query: {query[:100]}"
            )

            return RetrievalResult(
                query=query,
                results=filtered_results,
                metadata=metadata
            )

        except Exception as e:
            logger.error(f"Error in reranking retrieval: {str(e)}")
            raise

    def _rerank_with_cross_encoder(
        self,
        query: str,
        results: List[VectorStoreResult]
    ) -> List[VectorStoreResult]:
        """Rerank results using cross-encoder."""
        if not results or not self.cross_encoder:
            return results

        try:
            # Prepare query-document pairs for reranking
            pairs = []
            valid_results = []

            for result in results:
                if result.chunk and result.chunk.text:
                    pairs.append([query, result.chunk.text])
                    valid_results.append(result)

            if not pairs:
                logger.warning("No valid pairs for reranking")
                return results

            # Get reranking scores
            rerank_scores = self.cross_encoder.predict(pairs)

            # Create new results with reranking scores
            reranked_results = []
            for i, (result, score) in enumerate(zip(valid_results, rerank_scores)):
                # Convert score to float if it's a tensor
                if torch.is_tensor(score):
                    score = score.item()

                new_result = VectorStoreResult(
                    chunk_id=result.chunk_id,
                    score=float(score),
                    chunk=result.chunk,
                    metadata={
                        **result.metadata,
                        'original_score': result.score,
                        'rerank_score': float(score)
```

```
                    }
                )
                reranked_results.append(new_result)

            # Sort by reranking score
            reranked_results.sort(key=lambda x: x.score, reverse=True)

            logger.debug(f"Reranked {len(reranked_results)} results")
            return reranked_results

        except Exception as e:
            logger.error(f"Error during reranking: {str(e)}")
            return results  # Fall back to original results
```

5.5 Retrieval Factory

Create a factory in `src/retrieval/factory.py`:

```
from typing import Dict, Type, Optional
import logging

from src.retrieval.base import BaseRetriever
from src.retrieval.semantic_retriever import SemanticRetriever
from src.retrieval.hybrid_retriever import HybridRetriever
from src.retrieval.reranking_retriever import RerankingRetriever
from src.vector_store.base import BaseVectorStore
from src.embedding.base import BaseEmbedder

logger = logging.getLogger(__name__)


class RetrieverFactory:
    """Factory for creating retriever instances."""

    def __init__(self, config=None):
        """
        Initialize retriever factory.

        Args:
            config: Optional configuration dictionary
        """
        self.config = config or {}
        self._retrievers: Dict[str, Type[BaseRetriever]] = {}

        # Register default retrievers
        self.register_retriever("semantic", SemanticRetriever)
        self.register_retriever("hybrid", HybridRetriever)
        self.register_retriever("reranking", RerankingRetriever)

    def register_retriever(self, name: str, retriever_class: Type[BaseRetriever]) -> None
        """Register a retriever class."""
        self._retrievers[name.lower()] = retriever_class
        logger.debug(f"Registered retriever {retriever_class.__name__} as '{name}'")

    def get_retriever(
```

```
        self,
        strategy: str,
        vector_store: BaseVectorStore,
        embedder: BaseEmbedder
    ) -> Optional[BaseRetriever]:
        """
        Get a retriever instance.

        Args:
            strategy: Retrieval strategy name
            vector_store: Vector store instance
            embedder: Embedder instance

        Returns:
            Configured retriever instance or None
        """
        strategy = strategy.lower()
        retriever_class = self._retrievers.get(strategy)

        if not retriever_class:
            logger.error(f"No retriever found for strategy '{strategy}'")
            return None

        # Get configuration for this strategy
        strategy_config = {}
        if self.config and "strategies" in self.config:
            strategy_config = self.config.get("strategies", {}).get(strategy, {})

        try:
            return retriever_class(
                vector_store=vector_store,
                embedder=embedder,
                **strategy_config
            )
        except Exception as e:
            logger.error(f"Error creating retriever '{strategy}': {str(e)}")
            return None

    def get_default_retriever(
        self,
        vector_store: BaseVectorStore,
        embedder: BaseEmbedder
    ) -> BaseRetriever:
        """Get the default retriever as specified in config."""
        default_strategy = "semantic"
        if self.config:
            default_strategy = self.config.get("default_strategy", default_strategy)

        retriever = self.get_retriever(default_strategy, vector_store, embedder)
        if not retriever:
            logger.warning(
                f"Default retriever '{default_strategy}' failed. "
                f"Falling back to 'semantic'."
            )
            retriever = self.get_retriever("semantic", vector_store, embedder)
```

```
            if not retriever:
                raise RuntimeError("Could not create any retriever")

        return retriever
```

Step 6: Generation Implementation

6.1 Create Generation Base Class

Create a base class in `src/generation/base.py`:

```
from abc import ABC, abstractmethod
from typing import Dict, Any, Optional, List
from dataclasses import dataclass


@dataclass
class GenerationResult:
    """Class representing a generation result."""

    query: str
    response: str
    context: str
    metadata: Optional[Dict[str, Any]] = None

    def __post_init__(self):
        if self.metadata is None:
            self.metadata = {}

    def __repr__(self) -> str:
        return f"GenerationResult(query='{self.query[:50]}...', response_len={len(self.re


class BaseGenerator(ABC):
    """Base class for text generation."""

    @abstractmethod
    def generate(
        self,
        query: str,
        context: str,
        **kwargs
    ) -> GenerationResult:
        """
        Generate a response based on query and context.

        Args:
            query: The user query
            context: Retrieved context for augmentation
            **kwargs: Additional generation parameters

        Returns:
            Generation result object
        """
```

```
        pass

    @abstractmethod
    def get_model_info(self) -> Dict[str, Any]:
        """
        Get information about the generation model.

        Returns:
            Dictionary containing model metadata
        """
        pass
```

6.2 Implement OpenAI Generator

Create an OpenAI generator in `src/generation/openai_generator.py`:

```
import logging
import os
from typing import Dict, Any, Optional
from datetime import datetime
import openai
from openai import OpenAI

from src.generation.base import BaseGenerator, GenerationResult

logger = logging.getLogger(__name__)


class OpenAIGenerator(BaseGenerator):
    """OpenAI API-based text generator."""

    def __init__(
        self,
        model: str = "gpt-3.5-turbo",
        api_key: Optional[str] = None,
        temperature: float = 0.2,
        max_tokens: int = 1024,
        system_prompt: Optional[str] = None
    ):
        """
        Initialize OpenAI generator.

        Args:
            model: OpenAI model name
            api_key: OpenAI API key (defaults to OPENAI_API_KEY env var)
            temperature: Sampling temperature
            max_tokens: Maximum tokens to generate
            system_prompt: Optional system prompt
        """
        self.model = model
        self.temperature = temperature
        self.max_tokens = max_tokens

        # Set up API key
        api_key = api_key or os.environ.get("OPENAI_API_KEY")
```

```
        if not api_key:
            raise ValueError(
                "OpenAI API key not provided and OPENAI_API_KEY not set"
            )

        # Initialize client
        self.client = OpenAI(api_key=api_key)

        # Default system prompt
        self.system_prompt = system_prompt or """
You are a helpful AI assistant. Answer the user's question based on the provided context.
If the context doesn't contain enough information to fully answer the question, say so cl
Be accurate, concise, and helpful.
"""

        logger.info(f"Initialized OpenAI generator with model '{model}'")

    def generate(
        self,
        query: str,
        context: str,
        temperature: Optional[float] = None,
        max_tokens: Optional[int] = None,
        **kwargs
    ) -> GenerationResult:
        """Generate response using OpenAI API."""
        start_time = datetime.now()

        # Use instance defaults if not provided
        temperature = temperature or self.temperature
        max_tokens = max_tokens or self.max_tokens

        try:
            # Construct messages
            messages = [
                {"role": "system", "content": self.system_prompt},
                {"role": "user", "content": f"Context:\n{context}\n\nQuestion: {query}"}
            ]

            # Make API call
            response = self.client.chat.completions.create(
                model=self.model,
                messages=messages,
                temperature=temperature,
                max_tokens=max_tokens,
                **kwargs
            )

            # Extract response
            generated_text = response.choices[^0].message.content

            # Calculate generation time
            generation_time = (datetime.now() - start_time).total_seconds()

            # Create metadata
            metadata = {
```

```
                "model": self.model,
                "temperature": temperature,
                "max_tokens": max_tokens,
                "generation_time": generation_time,
                "usage": {
                    "prompt_tokens": response.usage.prompt_tokens,
                    "completion_tokens": response.usage.completion_tokens,
                    "total_tokens": response.usage.total_tokens
                },
                "finish_reason": response.choices[^0].finish_reason,
                "timestamp": datetime.now().isoformat()
            }

            logger.debug(
                f"Generated response ({response.usage.completion_tokens} tokens) "
                f"in {generation_time:.3f}s"
            )

            return GenerationResult(
                query=query,
                response=generated_text,
                context=context,
                metadata=metadata
            )

        except Exception as e:
            logger.error(f"Error generating response with OpenAI: {str(e)}")
            raise

    def get_model_info(self) -> Dict[str, Any]:
        """Get OpenAI model information."""
        return {
            "provider": "OpenAI",
            "model": self.model,
            "temperature": self.temperature,
            "max_tokens": self.max_tokens,
            "type": "api"
        }
```

6.3 Implement Local LLM Generator

Create a local LLM generator in `src/generation/local_generator.py`:

```
import logging
from typing import Dict, Any, Optional, Union
from datetime import datetime
import requests
import json

from src.generation.base import BaseGenerator, GenerationResult

logger = logging.getLogger(__name__)


class LocalLLMGenerator(BaseGenerator):
```

```
    """Local LLM generator using Ollama or similar local APIs."""

    def __init__(
        self,
        model: str = "llama2",
        base_url: str = "http://localhost:11434",
        temperature: float = 0.2,
        max_tokens: int = 1024,
        system_prompt: Optional[str] = None,
        timeout: int = 120
    ):
        """
        Initialize local LLM generator.

        Args:
            model: Local model name
            base_url: Base URL for the local LLM API
            temperature: Sampling temperature
            max_tokens: Maximum tokens to generate
            system_prompt: Optional system prompt
            timeout: Request timeout in seconds
        """
        self.model = model
        self.base_url = base_url.rstrip('/')
        self.temperature = temperature
        self.max_tokens = max_tokens
        self.timeout = timeout

        # Default system prompt
        self.system_prompt = system_prompt or """
You are a helpful AI assistant. Answer the user's question based on the provided context.
If the context doesn't contain enough information to fully answer the question, say so cl
Be accurate, concise, and helpful.
"""

        # Test connection
        self._test_connection()

        logger.info(f"Initialized local LLM generator with model '{model}'")

    def _test_connection(self) -> None:
        """Test connection to local LLM API."""
        try:
            response = requests.get(
                f"{self.base_url}/api/tags",
                timeout=10
            )
            response.raise_for_status()
            logger.info("Successfully connected to local LLM API")
        except requests.exceptions.RequestException as e:
            logger.warning(f"Could not connect to local LLM API: {str(e)}")

    def generate(
        self,
        query: str,
        context: str,
```

```
        temperature: Optional[float] = None,
        max_tokens: Optional[int] = None,
        **kwargs
    ) -> GenerationResult:
        """Generate response using local LLM."""
        start_time = datetime.now()

        # Use instance defaults if not provided
        temperature = temperature or self.temperature
        max_tokens = max_tokens or self.max_tokens

        try:
            # Construct prompt
            prompt = f"{self.system_prompt}\n\nContext:\n{context}\n\nQuestion: {query}\n

            # Prepare request payload (Ollama format)
            payload = {
                "model": self.model,
                "prompt": prompt,
                "options": {
                    "temperature": temperature,
                    "num_predict": max_tokens,
                    **kwargs
                },
                "stream": False
            }

            # Make API call
            response = requests.post(
                f"{self.base_url}/api/generate",
                json=payload,
                timeout=self.timeout
            )
            response.raise_for_status()

            # Parse response
            result = response.json()
            generated_text = result.get("response", "")

            # Calculate generation time
            generation_time = (datetime.now() - start_time).total_seconds()

            # Create metadata
            metadata = {
                "model": self.model,
                "provider": "local",
                "temperature": temperature,
                "max_tokens": max_tokens,
                "generation_time": generation_time,
                "eval_count": result.get("eval_count", 0),
                "eval_duration": result.get("eval_duration", 0),
                "timestamp": datetime.now().isoformat()
            }

            logger.debug(
                f"Generated response ({result.get('eval_count', 0)} tokens) "
```

```
                f"in {generation_time:.3f}s"
            )

            return GenerationResult(
                query=query,
                response=generated_text.strip(),
                context=context,
                metadata=metadata
            )

        except requests.exceptions.RequestException as e:
            logger.error(f"Error with local LLM API request: {str(e)}")
            raise
        except Exception as e:
            logger.error(f"Error generating response with local LLM: {str(e)}")
            raise

    def get_model_info(self) -> Dict[str, Any]:
        """Get local LLM model information."""
        return {
            "provider": "local",
            "model": self.model,
            "base_url": self.base_url,
            "temperature": self.temperature,
            "max_tokens": self.max_tokens,
            "type": "local_api"
        }
```

6.4 Generation Factory

Create a factory in `src/generation/factory.py`:

```
from typing import Dict, Type, Optional
import logging

from src.generation.base import BaseGenerator
from src.generation.openai_generator import OpenAIGenerator
from src.generation.local_generator import LocalLLMGenerator

logger = logging.getLogger(__name__)


class GeneratorFactory:
    """Factory for creating generator instances."""

    def __init__(self, config=None):
        """
        Initialize generator factory.

        Args:
            config: Optional configuration dictionary
        """
        self.config = config or {}
        self._generators: Dict[str, Type[BaseGenerator]] = {}
```

```
        # Register default generators
        self.register_generator("openai", OpenAIGenerator)
        self.register_generator("local", LocalLLMGenerator)

    def register_generator(self, name: str, generator_class: Type[BaseGenerator]) -> None
        """Register a generator class."""
        self._generators[name.lower()] = generator_class
        logger.debug(f"Registered generator {generator_class.__name__} as '{name}'")

    def get_generator(self, model_type: str) -> Optional[BaseGenerator]:
        """
        Get a generator instance.

        Args:
            model_type: Type of generator to create

        Returns:
            Configured generator instance or None
        """
        model_type = model_type.lower()
        generator_class = self._generators.get(model_type)

        if not generator_class:
            logger.error(f"No generator found for type '{model_type}'")
            return None

        # Get configuration for this generator type
        generator_config = {}
        if self.config and "models" in self.config:
            # Find matching model configuration
            for model_config in self.config["models"]:
                if model_config.get("type") == model_type:
                    generator_config = {
                        k: v for k, v in model_config.items()
                        if k not in ["name", "type"]
                    }
                    break

        try:
            return generator_class(**generator_config)
        except Exception as e:
            logger.error(f"Error creating generator '{model_type}': {str(e)}")
            return None

    def get_default_generator(self) -> BaseGenerator:
        """Get the default generator as specified in config."""
        default_model = "gpt-3.5-turbo"
        if self.config:
            default_model = self.config.get("default_model", default_model)

        # Find the generator type for this model
        generator_type = "openai"  # default
        if self.config and "models" in self.config:
            for model_config in self.config["models"]:
                if model_config.get("name") == default_model:
                    generator_type = model_config.get("type", generator_type)
```

```
                    break

        generator = self.get_generator(generator_type)
        if not generator:
            logger.warning(
                f"Default generator '{generator_type}' failed. "
                f"Trying 'openai'."
            )
            generator = self.get_generator("openai")

            if not generator:
                raise RuntimeError("Could not create any generator")

        return generator
```

Step 7: Evaluation Implementation

7.1 Create Evaluation Base Class

Create a base class in `src/evaluation/base.py`:

```
from abc import ABC, abstractmethod
from typing import Dict, Any, List, Optional
from dataclasses import dataclass
import json


@dataclass
class EvaluationResult:
    """Class representing an evaluation result."""

    metric_name: str
    score: float
    details: Optional[Dict[str, Any]] = None

    def __post_init__(self):
        if self.details is None:
            self.details = {}

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary for serialization."""
        return {
            "metric_name": self.metric_name,
            "score": self.score,
            "details": self.details
        }


@dataclass
class RAGEvaluationInput:
    """Input for RAG evaluation."""

    query: str
    expected_answer: Optional[str] = None
```

```
    retrieved_context: Optional[str] = None
    generated_answer: Optional[str] = None
    metadata: Optional[Dict[str, Any]] = None

    def __post_init__(self):
        if self.metadata is None:
            self.metadata = {}


class BaseEvaluationMetric(ABC):
    """Base class for evaluation metrics."""

    @abstractmethod
    def evaluate(self, input_data: RAGEvaluationInput) -> EvaluationResult:
        """
        Evaluate the given input.

        Args:
            input_data: Input data for evaluation

        Returns:
            Evaluation result
        """
        pass

    @property
    @abstractmethod
    def name(self) -> str:
        """Return the name of this metric."""
        pass

    @property
    @abstractmethod
    def requires_expected_answer(self) -> bool:
        """Return True if this metric requires an expected answer."""
        pass
```

7.2 Implement Basic Evaluation Metrics

Create basic metrics in `src/evaluation/basic_metrics.py`:

```
import logging
import re
from typing import Dict, Any, List
from datetime import datetime
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

from src.evaluation.base import BaseEvaluationMetric, EvaluationResult, RAGEvaluationInpu

logger = logging.getLogger(__name__)


class ContextRelevanceMetric(BaseEvaluationMetric):
```

```
    """Evaluate how relevant the retrieved context is to the query."""

    def __init__(self, threshold: float = 0.1):
        """
        Initialize context relevance metric.

        Args:
            threshold: Minimum relevance threshold
        """
        self.threshold = threshold

    @property
    def name(self) -> str:
        return "context_relevance"

    @property
    def requires_expected_answer(self) -> bool:
        return False

    def evaluate(self, input_data: RAGEvaluationInput) -> EvaluationResult:
        """Evaluate context relevance using TF-IDF similarity."""
        try:
            if not input_data.retrieved_context or not input_data.query:
                return EvaluationResult(
                    metric_name=self.name,
                    score=0.0,
                    details={"error": "Missing context or query"}
                )

            # Calculate TF-IDF similarity between query and context
            texts = [input_data.query, input_data.retrieved_context]

            vectorizer = TfidfVectorizer(
                stop_words='english',
                max_features=1000,
                ngram_range=(1, 2)
            )

            tfidf_matrix = vectorizer.fit_transform(texts)
            similarity = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:2])[^0][^0]

            # Normalize score to 0-1 range
            score = max(0.0, min(1.0, similarity))

            return EvaluationResult(
                metric_name=self.name,
                score=score,
                details={
                    "similarity": similarity,
                    "threshold": self.threshold,
                    "meets_threshold": score >= self.threshold
                }
            )

        except Exception as e:
            logger.error(f"Error evaluating context relevance: {str(e)}")
```

```
            return EvaluationResult(
                metric_name=self.name,
                score=0.0,
                details={"error": str(e)}
            )


class AnswerRelevanceMetric(BaseEvaluationMetric):
    """Evaluate how relevant the generated answer is to the query."""

    def __init__(self, threshold: float = 0.2):
        """
        Initialize answer relevance metric.

        Args:
            threshold: Minimum relevance threshold
        """
        self.threshold = threshold

    @property
    def name(self) -> str:
        return "answer_relevance"

    @property
    def requires_expected_answer(self) -> bool:
        return False

    def evaluate(self, input_data: RAGEvaluationInput) -> EvaluationResult:
        """Evaluate answer relevance using TF-IDF similarity."""
        try:
            if not input_data.generated_answer or not input_data.query:
                return EvaluationResult(
                    metric_name=self.name,
                    score=0.0,
                    details={"error": "Missing answer or query"}
                )

            # Calculate TF-IDF similarity between query and answer
            texts = [input_data.query, input_data.generated_answer]

            vectorizer = TfidfVectorizer(
                stop_words='english',
                max_features=1000,
                ngram_range=(1, 2)
            )

            try:
                tfidf_matrix = vectorizer.fit_transform(texts)
                similarity = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:2])[^0]|
            except ValueError:  # Handle case where vocabulary is empty
                similarity = 0.0

            # Normalize score to 0-1 range
            score = max(0.0, min(1.0, similarity))

            return EvaluationResult(
```

```
                metric_name=self.name,
                score=score,
                details={
                    "similarity": similarity,
                    "threshold": self.threshold,
                    "meets_threshold": score >= self.threshold
                }
            )

        except Exception as e:
            logger.error(f"Error evaluating answer relevance: {str(e)}")
            return EvaluationResult(
                metric_name=self.name,
                score=0.0,
                details={"error": str(e)}
            )


class FaithfulnessMetric(BaseEvaluationMetric):
    """Evaluate how faithful the generated answer is to the retrieved context."""

    def __init__(self, threshold: float = 0.3):
        """
        Initialize faithfulness metric.

        Args:
            threshold: Minimum faithfulness threshold
        """
        self.threshold = threshold

    @property
    def name(self) -> str:
        return "faithfulness"

    @property
    def requires_expected_answer(self) -> bool:
        return False

    def _extract_claims(self, text: str) -> List[str]:
        """Extract factual claims from text (simplified approach)."""
        # Split by sentences and filter out questions and very short sentences
        sentences = re.split(r'[.!?]+', text)
        claims = []

        for sentence in sentences:
            sentence = sentence.strip()
            if (len(sentence) > 10 and
                not sentence.endswith('?') and
                not sentence.startswith(('However', 'But', 'Although'))):
                claims.append(sentence)

        return claims

    def evaluate(self, input_data: RAGEvaluationInput) -> EvaluationResult:
        """Evaluate faithfulness by checking if answer claims are supported by context."'
        try:
```

```
            if not input_data.generated_answer or not input_data.retrieved_context:
                return EvaluationResult(
                    metric_name=self.name,
                    score=0.0,
                    details={"error": "Missing answer or context"}
                )

            # Extract claims from the generated answer
            answer_claims = self._extract_claims(input_data.generated_answer)

            if not answer_claims:
                return EvaluationResult(
                    metric_name=self.name,
                    score=1.0,  # No claims to verify
                    details={"claims": [], "supported_claims": []}
                )

            # Check how many claims are supported by the context
            supported_claims = []
            context_lower = input_data.retrieved_context.lower()

            for claim in answer_claims:
                claim_lower = claim.lower()

                # Simple keyword overlap check (can be improved with semantic similarity)
                claim_words = set(re.findall(r'\b\w+\b', claim_lower))
                context_words = set(re.findall(r'\b\w+\b', context_lower))

                # Calculate word overlap
                overlap = len(claim_words.intersection(context_words))
                overlap_ratio = overlap / len(claim_words) if claim_words else 0

                if overlap_ratio > 0.3:  # At least 30% word overlap
                    supported_claims.append(claim)

            # Calculate faithfulness score
            score = len(supported_claims) / len(answer_claims) if answer_claims else 1.0

            return EvaluationResult(
                metric_name=self.name,
                score=score,
                details={
                    "total_claims": len(answer_claims),
                    "supported_claims": len(supported_claims),
                    "claims": answer_claims,
                    "supported_claims": supported_claims,
                    "threshold": self.threshold,
                    "meets_threshold": score >= self.threshold
                }
            )

        except Exception as e:
            logger.error(f"Error evaluating faithfulness: {str(e)}")
            return EvaluationResult(
                metric_name=self.name,
                score=0.0,
```

```
                details={"error": str(e)}
            )


class AnswerAccuracyMetric(BaseEvaluationMetric):
    """Evaluate accuracy of generated answer against expected answer."""

    def __init__(self, threshold: float = 0.5):
        """
        Initialize answer accuracy metric.

        Args:
            threshold: Minimum accuracy threshold
        """
        self.threshold = threshold

    @property
    def name(self) -> str:
        return "answer_accuracy"

    @property
    def requires_expected_answer(self) -> bool:
        return True

    def evaluate(self, input_data: RAGEvaluationInput) -> EvaluationResult:
        """Evaluate answer accuracy using semantic similarity."""
        try:
            if not input_data.generated_answer or not input_data.expected_answer:
                return EvaluationResult(
                    metric_name=self.name,
                    score=0.0,
                    details={"error": "Missing generated or expected answer"}
                )

            # Calculate TF-IDF similarity between generated and expected answers
            texts = [input_data.generated_answer, input_data.expected_answer]

            vectorizer = TfidfVectorizer(
                stop_words='english',
                max_features=1000,
                ngram_range=(1, 2)
            )

            try:
                tfidf_matrix = vectorizer.fit_transform(texts)
                similarity = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:2])[^0][
            except ValueError:
                similarity = 0.0

            # Normalize score to 0-1 range
            score = max(0.0, min(1.0, similarity))

            return EvaluationResult(
                metric_name=self.name,
                score=score,
                details={
```

```
                    "similarity": similarity,
                    "threshold": self.threshold,
                    "meets_threshold": score >= self.threshold,
                    "generated_length": len(input_data.generated_answer),
                    "expected_length": len(input_data.expected_answer)
                }
            )

        except Exception as e:
            logger.error(f"Error evaluating answer accuracy: {str(e)}")
            return EvaluationResult(
                metric_name=self.name,
                score=0.0,
                details={"error": str(e)}
            )
```

7.3 Implement RAG Evaluator

Create the main evaluator in `src/evaluation/rag_evaluator.py`:

```
import logging
from typing import List, Dict, Any, Optional
import json
from datetime import datetime
from pathlib import Path

from src.evaluation.base import (
    BaseEvaluationMetric, EvaluationResult, RAGEvaluationInput
)
from src.evaluation.basic_metrics import (
    ContextRelevanceMetric, AnswerRelevanceMetric,
    FaithfulnessMetric, AnswerAccuracyMetric
)

logger = logging.getLogger(__name__)


class RAGEvaluator:
    """Main class for evaluating RAG system performance."""

    def __init__(
        self,
        metrics: Optional[List[BaseEvaluationMetric]] = None,
        save_results: bool = True,
        results_dir: str = "./data/evaluations"
    ):
        """
        Initialize RAG evaluator.

        Args:
            metrics: List of evaluation metrics to use
            save_results: Whether to save evaluation results
            results_dir: Directory to save results
        """
        self.save_results = save_results
```

```
        self.results_dir = Path(results_dir)

        # Initialize default metrics if none provided
        if metrics is None:
            self.metrics = [
                ContextRelevanceMetric(),
                AnswerRelevanceMetric(),
                FaithfulnessMetric(),
                AnswerAccuracyMetric()
            ]
        else:
            self.metrics = metrics

        # Create results directory
        if self.save_results:
            self.results_dir.mkdir(parents=True, exist_ok=True)

        logger.info(f"Initialized RAG evaluator with {len(self.metrics)} metrics")

    def evaluate_single(self, input_data: RAGEvaluationInput) -> Dict[str, EvaluationResu
        """
        Evaluate a single query-answer pair.

        Args:
            input_data: Input data for evaluation

        Returns:
            Dictionary of metric name to evaluation result
        """
        results = {}

        for metric in self.metrics:
            try:
                # Skip metrics that require expected answer if not provided
                if (metric.requires_expected_answer and
                    not input_data.expected_answer):
                    logger.debug(
                        f"Skipping {metric.name} - requires expected answer"
                    )
                    continue

                result = metric.evaluate(input_data)
                results[metric.name] = result

            except Exception as e:
                logger.error(f"Error evaluating {metric.name}: {str(e)}")
                results[metric.name] = EvaluationResult(
                    metric_name=metric.name,
                    score=0.0,
                    details={"error": str(e)}
                )

        return results

    def evaluate_batch(
        self,
```

```
        input_batch: List[RAGEvaluationInput],
        experiment_name: Optional[str] = None
    ) -> Dict[str, Any]:
        """
        Evaluate a batch of query-answer pairs.

        Args:
            input_batch: List of input data for evaluation
            experiment_name: Optional name for this evaluation run

        Returns:
            Aggregated evaluation results
        """
        start_time = datetime.now()

        # Evaluate each input
        all_results = []
        for i, input_data in enumerate(input_batch):
            logger.debug(f"Evaluating sample {i+1}/{len(input_batch)}")

            sample_results = self.evaluate_single(input_data)

            # Add input metadata to results
            result_entry = {
                "sample_id": i,
                "query": input_data.query,
                "has_expected_answer": input_data.expected_answer is not None,
                "has_context": input_data.retrieved_context is not None,
                "has_generated_answer": input_data.generated_answer is not None,
                "metadata": input_data.metadata,
                "metrics": {
                    name: result.to_dict()
                    for name, result in sample_results.items()
                }
            }
            all_results.append(result_entry)

        # Aggregate results
        aggregated = self._aggregate_results(all_results)

        # Add evaluation metadata
        evaluation_time = (datetime.now() - start_time).total_seconds()
        aggregated["evaluation_metadata"] = {
            "experiment_name": experiment_name,
            "total_samples": len(input_batch),
            "evaluation_time": evaluation_time,
            "timestamp": datetime.now().isoformat(),
            "metrics_used": [metric.name for metric in self.metrics]
        }

        # Save results if enabled
        if self.save_results:
            self._save_results(aggregated, experiment_name)

        logger.info(
            f"Completed batch evaluation of {len(input_batch)} samples "
```

```
            f"in {evaluation_time:.2f}s"
        )

        return aggregated

    def _aggregate_results(self, all_results: List[Dict[str, Any]]) -> Dict[str, Any]:
        """Aggregate individual evaluation results."""
        if not all_results:
            return {"error": "No results to aggregate"}

        # Initialize aggregation structures
        metric_scores = {}
        metric_counts = {}

        # Collect all scores by metric
        for result_entry in all_results:
            for metric_name, metric_data in result_entry["metrics"].items():
                if metric_name not in metric_scores:
                    metric_scores[metric_name] = []
                    metric_counts[metric_name] = 0

                score = metric_data.get("score", 0.0)
                if score is not None and not (isinstance(score, float) and score != score
                    metric_scores[metric_name].append(score)
                    metric_counts[metric_name] += 1

        # Calculate aggregated statistics
        aggregated_metrics = {}
        for metric_name, scores in metric_scores.items():
            if scores:
                aggregated_metrics[metric_name] = {
                    "mean": sum(scores) / len(scores),
                    "min": min(scores),
                    "max": max(scores),
                    "count": len(scores),
                    "scores": scores
                }
            else:
                aggregated_metrics[metric_name] = {
                    "mean": 0.0,
                    "min": 0.0,
                    "max": 0.0,
                    "count": 0,
                    "scores": []
                }

        # Calculate overall performance score (average of all metric means)
        if aggregated_metrics:
            overall_score = sum(
                metrics["mean"] for metrics in aggregated_metrics.values()
            ) / len(aggregated_metrics)
        else:
            overall_score = 0.0

        return {
            "overall_score": overall_score,
```

```
            "metric_summaries": aggregated_metrics,
            "detailed_results": all_results
        }

    def _save_results(
        self,
        results: Dict[str, Any],
        experiment_name: Optional[str] = None
    ) -> None:
        """Save evaluation results to file."""
        try:
            # Generate filename
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            if experiment_name:
                filename = f"{experiment_name}_{timestamp}.json"
            else:
                filename = f"evaluation_{timestamp}.json"

            filepath = self.results_dir / filename

            # Save results
            with open(filepath, 'w', encoding='utf-8') as f:
                json.dump(results, f, indent=2, ensure_ascii=False)

            logger.info(f"Saved evaluation results to {filepath}")

        except Exception as e:
            logger.error(f"Error saving evaluation results: {str(e)}")

    def add_metric(self, metric: BaseEvaluationMetric) -> None:
        """Add a new evaluation metric."""
        self.metrics.append(metric)
        logger.info(f"Added evaluation metric: {metric.name}")

    def get_metric_names(self) -> List[str]:
        """Get names of all configured metrics."""
        return [metric.name for metric in self.metrics]
```

7.4 Create Evaluation Factory

Create a factory in `src/evaluation/factory.py`:

```
from typing import Dict, Type, List, Optional
import logging

from src.evaluation.base import BaseEvaluationMetric
from src.evaluation.basic_metrics import (
    ContextRelevanceMetric, AnswerRelevanceMetric,
    FaithfulnessMetric, AnswerAccuracyMetric
)
from src.evaluation.rag_evaluator import RAGEvaluator

logger = logging.getLogger(__name__)
```

```
class EvaluationFactory:
    """Factory for creating evaluation components."""

    def __init__(self, config=None):
        """
        Initialize evaluation factory.

        Args:
            config: Optional configuration dictionary
        """
        self.config = config or {}
        self._metrics: Dict[str, Type[BaseEvaluationMetric]] = {}

        # Register default metrics
        self.register_metric("context_relevance", ContextRelevanceMetric)
        self.register_metric("answer_relevance", AnswerRelevanceMetric)
        self.register_metric("faithfulness", FaithfulnessMetric)
        self.register_metric("answer_accuracy", AnswerAccuracyMetric)

    def register_metric(self, name: str, metric_class: Type[BaseEvaluationMetric]) -> Nor
        """Register an evaluation metric class."""
        self._metrics[name.lower()] = metric_class
        logger.debug(f"Registered metric {metric_class.__name__} as '{name}'")

    def create_metric(self, metric_name: str, **kwargs) -> Optional[BaseEvaluationMetric]
        """
        Create a metric instance.

        Args:
            metric_name: Name of the metric to create
            **kwargs: Additional arguments for metric initialization

        Returns:
            Metric instance or None
        """
        metric_name = metric_name.lower()
        metric_class = self._metrics.get(metric_name)

        if not metric_class:
            logger.error(f"No metric found for name '{metric_name}'")
            return None

        try:
            return metric_class(**kwargs)
        except Exception as e:
            logger.error(f"Error creating metric '{metric_name}': {str(e)}")
            return None

    def create_evaluator(
        self,
        metric_names: Optional[List[str]] = None,
        **kwargs
    ) -> RAGEvaluator:
        """
        Create a RAG evaluator with specified metrics.
```

```
        Args:
            metric_names: List of metric names to include
            **kwargs: Additional arguments for evaluator initialization

        Returns:
            Configured RAG evaluator
        """
        # Use default metrics if none specified
        if metric_names is None:
            if self.config and "metrics" in self.config:
                metric_names = self.config["metrics"]
            else:
                metric_names = ["context_relevance", "answer_relevance", "faithfulness"]

        # Create metric instances
        metrics = []
        for metric_name in metric_names:
            metric = self.create_metric(metric_name)
            if metric:
                metrics.append(metric)
            else:
                logger.warning(f"Could not create metric '{metric_name}', skipping")

        if not metrics:
            logger.warning("No metrics available, using defaults")
            metrics = [
                ContextRelevanceMetric(),
                AnswerRelevanceMetric(),
                FaithfulnessMetric()
            ]

        # Get evaluator configuration
        evaluator_config = {}
        if self.config:
            evaluator_config.update({
                "save_results": self.config.get("logging_enabled", True),
                "results_dir": self.config.get("results_dir", "./data/evaluations")
            })
        evaluator_config.update(kwargs)

        return RAGEvaluator(metrics=metrics, **evaluator_config)

    def get_available_metrics(self) -> List[str]:
        """Get list of available metric names."""
        return list(self._metrics.keys())
```

Step 8: Integration and Usage Example

Create a main integration script in `src/rag_system.py`:

```
import logging
from typing import List, Dict, Any, Optional
from pathlib import Path

from src.document_processing.factory import DocumentProcessorFactory
```

```
from src.chunking.factory import ChunkerFactory
from src.embedding.sentence_transformer import SentenceTransformerEmbedder
from src.vector_store.factory import VectorStoreFactory
from src.retrieval.factory import RetrieverFactory
from src.generation.factory import GeneratorFactory
from src.evaluation.factory import EvaluationFactory
from src.evaluation.base import RAGEvaluationInput

logger = logging.getLogger(__name__)


class RAGSystem:
    """Complete RAG system integrating all components."""

    def __init__(self, config: Dict[str, Any]):
        """
        Initialize the RAG system with configuration.

        Args:
            config: System configuration dictionary
        """
        self.config = config

        # Initialize components
        self.doc_processor = DocumentProcessorFactory()
        self.chunker_factory = ChunkerFactory(config.get('chunking', {}))
        self.embedder = SentenceTransformerEmbedder(
            **config.get('embedding', {})
        )

        vector_store_factory = VectorStoreFactory(config.get('vector_store', {}))
        self.vector_store = vector_store_factory.get_default_store()

        retriever_factory = RetrieverFactory(config.get('retrieval', {}))
        self.retriever = retriever_factory.get_default_retriever(
            self.vector_store, self.embedder
        )

        generator_factory = GeneratorFactory(config.get('generation', {}))
        self.generator = generator_factory.get_default_generator()

        evaluation_factory = EvaluationFactory(config.get('evaluation', {}))
        self.evaluator = evaluation_factory.create_evaluator()

        logger.info("RAG system initialized successfully")

    def ingest_documents(self, file_paths: List[Path]) -> Dict[str, Any]:
        """
        Ingest documents into the RAG system.

        Args:
            file_paths: List of document file paths

        Returns:
            Ingestion summary
        """
```

```
        logger.info(f"Starting ingestion of {len(file_paths)} documents")

        processed_docs = 0
        total_chunks = 0
        errors = []

        for file_path in file_paths:
            try:
                # Process document
                document = self.doc_processor.process_document(file_path)
                if not document:
                    errors.append(f"Failed to process {file_path}")
                    continue

                # Chunk document
                chunks = self.chunker_factory.chunk_document(document)
                if not chunks:
                    errors.append(f"No chunks created for {file_path}")
                    continue

                # Generate embeddings
                embeddings = self.embedder.embed_chunks(chunks)

                # Store in vector database
                self.vector_store.add_embeddings(embeddings)

                processed_docs += 1
                total_chunks += len(chunks)

                logger.info(f"Processed {file_path}: {len(chunks)} chunks")

            except Exception as e:
                error_msg = f"Error processing {file_path}: {str(e)}"
                errors.append(error_msg)
                logger.error(error_msg)

        summary = {
            "processed_documents": processed_docs,
            "total_chunks": total_chunks,
            "errors": errors,
            "vector_store_info": self.vector_store.get_collection_info()
        }

        logger.info(f"Ingestion complete: {processed_docs} documents, {total_chunks} chur
        return summary

    def query(self, question: str, top_k: int = 5) -> Dict[str, Any]:
        """
        Query the RAG system.

        Args:
            question: User question
            top_k: Number of results to retrieve

        Returns:
            Query result with answer and metadata
```

```
        """
        try:
            # Retrieve relevant documents
            retrieval_result = self.retriever.retrieve(question, top_k=top_k)

            # Generate answer
            context = retrieval_result.get_context()
            generation_result = self.generator.generate(question, context)

            return {
                "question": question,
                "answer": generation_result.response,
                "context": context,
                "retrieval_metadata": retrieval_result.metadata,
                "generation_metadata": generation_result.metadata,
                "retrieved_chunks": len(retrieval_result.results)
            }

        except Exception as e:
            logger.error(f"Error processing query '{question}': {str(e)}")
            return {
                "question": question,
                "answer": f"Error processing query: {str(e)}",
                "error": True
            }

    def evaluate(
        self,
        test_data: List[Dict[str, str]],
        experiment_name: Optional[str] = None
    ) -> Dict[str, Any]:
        """
        Evaluate the RAG system performance.

        Args:
            test_data: List of test cases with 'query' and optionally 'expected_answer'
            experiment_name: Optional experiment name

        Returns:
            Evaluation results
        """
        logger.info(f"Starting evaluation with {len(test_data)} test cases")

        # Prepare evaluation inputs
        evaluation_inputs = []

        for test_case in test_data:
            query = test_case['query']
            expected_answer = test_case.get('expected_answer')

            # Get system response
            result = self.query(query)

            # Create evaluation input
            eval_input = RAGEvaluationInput(
                query=query,
```

```
                expected_answer=expected_answer,
                retrieved_context=result.get('context'),
                generated_answer=result.get('answer'),
                metadata={
                    'retrieval_metadata': result.get('retrieval_metadata'),
                    'generation_metadata': result.get('generation_metadata')
                }
            )
            evaluation_inputs.append(eval_input)

        # Run evaluation
        return self.evaluator.evaluate_batch(evaluation_inputs, experiment_name)


def main():
    """Example usage of the RAG system."""
    # Load configuration
    config = {
        "embedding": {
            "model_name": "sentence-transformers/all-MiniLM-L6-v2"
        },
        "vector_store": {
            "default": "chroma",
            "options": {
                "chroma": {
                    "collection_name": "rag_documents",
                    "persist_directory": "./data/chroma_db"
                }
            }
        },
        "retrieval": {
            "default_strategy": "semantic"
        },
        "generation": {
            "default_model": "gpt-3.5-turbo"
        }
    }

    # Initialize RAG system
    rag = RAGSystem(config)

    # Example: Ingest documents
    document_paths = [
        Path("./data/raw/document1.pdf"),
        Path("./data/raw/document2.pdf")
    ]

    if any(path.exists() for path in document_paths):
        ingestion_result = rag.ingest_documents(document_paths)
        print("Ingestion Result:", ingestion_result)

    # Example: Query system
    result = rag.query("What is the main topic of the documents?")
    print("Query Result:", result)

    # Example: Evaluate system
```

```
    test_data = [
        {
            "query": "What is the main topic?",
            "expected_answer": "The main topic is artificial intelligence."
        }
    ]

    eval_result = rag.evaluate(test_data, "example_evaluation")
    print("Evaluation Result:", eval_result)


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    main()
```

This completes the implementation of your RAG system with all four remaining components:

1. **Vector Store**: Chroma and Qdrant implementations with factory pattern
2. **Retrieval**: Semantic, hybrid, and reranking retrieval strategies
3. **Generation**: OpenAI API and local LLM generators
4. **Evaluation**: Context relevance, answer relevance, faithfulness, and accuracy metrics

The system is modular, configurable, and production-ready. You can now run the complete RAG pipeline from document ingestion to evaluation[1] [2] [3] [4] [5].

⁂

1. https://docs.aws.amazon.com/prescriptive-guidance/latest/choosing-an-aws-vector-database-for-rag-use-cases/introduction.html
2. https://www.datacamp.com/tutorial/chromadb-tutorial-step-by-step-guide
3. https://www.datacamp.com/blog/rag-advanced
4. https://www.confident-ai.com/blog/rag-evaluation-metrics-answer-relevancy-faithfulness-and-more
5. https://github.com/explodinggradients/ragas