Importing Libraries

```
!pip install langchain
!pip install langchain_community
!pip install tiktoken
!pip install faiss-cpu
!pip install chromadb
!pip install edgartools
!pip install langchain langchain-openai chromadb langchain-text-splitters python-dotenv
```

Show hidden output

```
import os
import pandas as pd
from edgar import set_identity, Company
from datetime import datetime
import pickle
import numpy as np
import faiss
import re
import json
```

Data Collection

```
set_identity("K sha kt@codes.finance")

CIK_MAP = {
    "GOOGL": "GOOGL",
    "MSFT": "MSFT",
    "NVDA": "NVDA"
}

YEARS = [2022, 2023, 2024]
SAVE_DIR = "sec_filings"
os.makedirs(SAVE_DIR, exist_ok=True)


def download_10k_for_company(ticker):
    print(f"\nProcessing: {ticker}")
    company = Company(ticker)
    filings = company.get_filings(form="10-K")

    df = filings.to_pandas()
    df['filing_date'] = pd.to_datetime(df['filing_date'])

    for year in YEARS:
        match = df[df['filing_date'].dt.year == year]
        if match.empty:
            print(f"No 10-K filing found for {ticker} in {year}")
            continue

        filing_date = match.iloc[0]['filing_date']
        filing = filings.filter(date=filing_date.strftime('%Y-%m-%d'))
        filing_obj = filing.latest().obj()

        print(f"Downloading {ticker} 10-K for {year} (filed on {filing_date.date()})...")

        try:
            content = filing_obj.items
            filename = f"{ticker}_{year}_10K.txt"
            filepath = os.path.join(SAVE_DIR, filename)

            with open(filepath, 'a', encoding='utf-8') as f:
                for i in content:
                    x = filing_obj[i]
                    f.write(x)

            print(f"Saved to {filepath}")
        except Exception as e:
            print(f"Error saving {ticker} {year}: {e}")

if __name__ == "__main__":
    for ticker in CIK_MAP.values():
        download_10k_for_company(ticker)
```

```
Processing: GOOGL
Downloading GOOGL 10-K for 2022 (filed on 2022-02-02)...
Saved to sec_filings/GOOGL_2022_10K.txt
Downloading GOOGL 10-K for 2023 (filed on 2023-02-03)...
Saved to sec_filings/GOOGL_2023_10K.txt
Downloading GOOGL 10-K for 2024 (filed on 2024-01-31)...
Saved to sec_filings/GOOGL_2024_10K.txt

Processing: MSFT
Downloading MSFT 10-K for 2022 (filed on 2022-07-28)...
Saved to sec_filings/MSFT_2022_10K.txt
Downloading MSFT 10-K for 2023 (filed on 2023-07-27)...
Saved to sec_filings/MSFT_2023_10K.txt
Downloading MSFT 10-K for 2024 (filed on 2024-07-30)...
Saved to sec_filings/MSFT_2024_10K.txt

Processing: NVDA
Downloading NVDA 10-K for 2022 (filed on 2022-03-18)...
Saved to sec_filings/NVDA_2022_10K.txt
Downloading NVDA 10-K for 2023 (filed on 2023-02-24)...
Saved to sec_filings/NVDA_2023_10K.txt
Downloading NVDA 10-K for 2024 (filed on 2024-02-21)...
Saved to sec_filings/NVDA_2024_10K.txt
```

Splitting/Chunking and Embedding

```
!pip install -q transformers sentence-transformers qdrant-client langchain
```

Show hidden output

```
import os
import logging
from pathlib import Path
from typing import List, Optional
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.embeddings import SentenceTransformerEmbeddings
from langchain_community.document_loaders import DirectoryLoader, TextLoader
from langchain_community.vectorstores import Qdrant
from langchain.schema import Document

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

class VectorDatabaseIngestion:
  def __init__(self,
               data_directory: str = "sec_filings/",
               qdrant_url: str = ":memory:",
               collection_name: str = "sec_filings_collection",
               embedding_model: str = "sentence-transformers/all-MiniLM-L6-v2",
               chunk_size: int = 500,
               chunk_overlap: int = 100):
    """
        Initialize the vector database ingestion pipeline.

        Args:
            data_directory: Path to directory containing text files
            qdrant_url: URL for Qdrant vector database (using ":memory:" for in-memory)
            collection_name: Name of the collection in Qdrant
            embedding_model: Name of the sentence transformer model
            chunk_size: Size of text chunks for splitting
            chunk_overlap: Overlap between consecutive chunks
        """
    self.data_directory = Path(data_directory)
    self.qdrant_url = qdrant_url
    self.collection_name = collection_name
    self.chunk_size = chunk_size
    self.chunk_overlap = chunk_overlap


    try:
      self.embeddings = SentenceTransformerEmbeddings(model_name=embedding_model)
      logger.info(f"Successfully loaded embeddings model: {embedding_model}")
    except Exception as e:
      logger.error(f"Failed to load embeddings model: {e}")

  def load_documents(self) -> List[Document]:
    """Loads documents from the data directory."""
    logger.info(f"Loading documents from {self.data_directory}...")
    try:
```

```
        loader = DirectoryLoader(str(self.data_directory), glob="*.txt", loader_cls=TextLoader)
        documents = loader.load()
        logger.info(f"Loaded {len(documents)} documents.")
        return documents
      except Exception as e:
        logger.error(f"Failed to load documents: {e}")
        return []

    def split_documents(self, documents: List[Document]) -> List[Document]:
      """Splits documents into smaller chunks quickly using basic character splitting."""
      logger.info(f"Splitting documents into chunks (size={self.chunk_size}, overlap={self.chunk_overlap})...")
      from langchain.text_splitter import CharacterTextSplitter
      text_splitter = CharacterTextSplitter(separator="\n\n", chunk_size=self.chunk_size, chunk_overlap=self.chunk_overlap)
      chunks = text_splitter.split_documents(documents)
      logger.info(f"Split into {len(chunks)} chunks.")
      return chunks


    def ingest_documents(self):
      """Loads, splits, and ingests documents into the Qdrant vector database."""
      documents = self.load_documents()
      if not documents:
        return

      chunks = self.split_documents(documents)
      if not chunks:
        return

      logger.info(f"Ingesting {len(chunks)} chunks into Qdrant collection '{self.collection_name}'...")
      try:
        qdrant = Qdrant.from_documents(
            chunks,
            self.embeddings,
            location=self.qdrant_url,
            collection_name=self.collection_name,
        )
        self.qdrant_db = qdrant
        logger.info("Successfully ingested documents into Qdrant.")
        return qdrant
      except Exception as e:
        logger.error(f"Failed to ingest documents into Qdrant: {e}")

    def search_similar_chunks(self, query: str, top_k: int = 5):
      """
      Perform semantic search over the vector database to retrieve top-k relevant chunks.

      Args:
          query: Natural language query
          top_k: Number of top relevant chunks to return

      Returns:
          List of top-k Document objects
      """
      if not hasattr(self, "qdrant_db"):
          logger.error("Vector store not found. Please run ingest_documents() first.")
          return []

      try:
          logger.info(f"Performing similarity search for query: '{query}' (top {top_k})")
          results = self.qdrant_db.similarity_search(query=query, k=top_k)
          logger.info(f"Retrieved {len(results)} chunks.")
          return results
      except Exception as e:
          logger.error(f"Failed to perform similarity search: {e}")
          return []

# Example Usage:
if __name__ == "__main__":
  ingester = VectorDatabaseIngestion(
      data_directory=SAVE_DIR,
      qdrant_url=":memory:",
      collection_name="sec_filings_vector_db",
      embedding_model="sentence-transformers/all-MiniLM-L6-v2"
  )
  qdrant_db = ingester.ingest_documents()

  if qdrant_db:
    logger.info("Vector database created in memory.")
```

Show hidden output

> Testing Embeddings

```
query = "What was Microsoft’s total revenue in 2023?"
docs = qdrant_db.similarity_search(query)
print("\nSearch Results:")
for doc in docs:
    print(doc.page_content[:500] + "...")
```

```
Search Results:
We have recast certain prior period amounts to conform to the way we internally manage and monitor our business.
Our Microsoft Cloud revenue, which includes Azure and other cloud services, Office 365 Commercial, the commercial portio
Our Microsoft Cloud revenue, which includes Azure and other cloud services, Office 365 Commercial, the commercial portio
•   Operating expenses increased $2.8 billion or 16% driven by investments in Azure and other cloud services.
----------------------------------------------------------------------------------------------------
 Revenue and operating income included an unfavorable foreign currency impact of 2% and 3%, respectively.
 More Personal Computing
 Revenue increased $5.6 billion or 10%.

•   Windows revenue increased $2.3 billion or 10% driven by growth in Windows OEM and Windows Commercial...
(In millions)
Year Ended June 30,                  2023               2022               2021
Server products and cloud services             $79,970            $67,350            $52,589
Office products and cloud services              48,728             44,862             39,872
Windows                                         21,507             24,732             22,488
Gaming             ...
```

```
if qdrant_db:
    logger.info("Vector database created in memory.")
    results = ingester.search_similar_chunks("What was Microsoft's revenue in 2023?", top_k=5)

    print("\nTop 5 Results:\n")
    for i, doc in enumerate(results, 1):
        print(f"{i}. {doc.metadata.get('source', 'Unknown')} - {doc.page_content[:500]}...\n")
```

```
Top 5 Results:

1. sec_filings/MSFT_2024_10K.txt - We have recast certain prior period amounts to conform to the way we internally manag
Our Microsoft Cloud revenue, which includes Azure and other cloud services, Office 365 Commercial, the commercial portio

2. sec_filings/MSFT_2022_10K.txt - •   Operating expenses increased $2.8 billion or 16% driven by investments in Azure a
----------------------------------------------------------------------------------------------------
 Revenue and operating income included an unfavorable foreign currency impact of 2% and 3%, respectively.
 More Personal Computing
 Revenue increased $5.6 billion or 10%.

•   Windows revenue increased $2.3 billion or 10% driven by growth in Windows OEM and Windows Commercial...

3. sec_filings/MSFT_2023_10K.txt - Our Microsoft Cloud revenue, which includes Azure and other cloud services, Office 36

4. sec_filings/MSFT_2022_10K.txt - •   Operating expenses increased $1.5 billion or 14% driven by investments in Gaming,
----------------------------------------------------------------------------------------------------
 45...

5. sec_filings/MSFT_2024_10K.txt - Operating income increased $11.7 billion or 31%.
 •Gross margin increased $11.6 billion or 19% driven by growth in Azure. Gross margin percentage decreased slightly. Exc
 •Operating expenses decreased slightly primarily driven by the prior year Q2 charge, offset in part b...
```

Query Engine

```
!pip install langchain_google_genai
```

Show hidden output

```
import logging
from typing import List
from langchain.schema import Document

logger = logging.getLogger(__name__)

class QueryEngineAgent:
    def __init__(self, qdrant_db):
        self.qdrant_db = qdrant_db

    def search_similar_chunks(self, query: str, top_k: int = 5) -> List[Document]:
        """
        Perform semantic search over the vector database to retrieve top-k relevant chunks
```

```
        Perform semantic search over the vector database to retrieve top-k relevant chunks.

        Args:
            query: Natural language query
            top_k: Number of top relevant chunks to return

        Returns:
            List of top-k Document objects
        """
        if self.qdrant_db is None:
            logger.error("Qdrant DB not provided.")
            return []

        try:
            logger.info(f"Performing similarity search for query: '{query}' (top {top_k})")
            results = self.qdrant_db.similarity_search(query=query, k=top_k)
            logger.info(f"Retrieved {len(results)} chunks.")
            return results
        except Exception as e:
            logger.error(f"Failed to perform similarity search: {e}")
            return []

    def decompose_query(self, complex_query: str) -> List[str]:
        """
        Decompose a complex financial query into simpler sub-queries.

        Args:
            complex_query: The original complex query

        Returns:
            List of sub-queries
        """
        import re
        from langchain_google_genai import ChatGoogleGenerativeAI
        from langchain.chains import LLMChain
        from langchain.prompts import PromptTemplate
        os.environ["GOOGLE_API_KEY"] = "Key"
        llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0.2)

        decompose_prompt = PromptTemplate(
            input_variables=["question"],
            template="""Decompose the following complex financial question into simpler, logically ordered sub-queries:

Question: {question}
Sub-queries:
1."""
        )
        decompose_chain = LLMChain(llm=llm, prompt=decompose_prompt)
        decomposition = decompose_chain.run(complex_query)

        subqueries = re.findall(r"\d+\.\s*(.*?)\n", decomposition + "\n")
        if not subqueries:
            subqueries = [complex_query]
        return subqueries

    def multi_step_retrieve(self, subqueries: List[str], top_k: int = 5) -> List[Document]:
        """
        Execute multiple searches and combine results.

        Args:
            subqueries: List of decomposed sub-queries
            top_k: Top chunks per subquery

        Returns:
            Combined list of relevant documents
        """
        all_docs = []
        for subquery in subqueries:
            docs = self.search_similar_chunks(subquery, top_k=top_k)
            all_docs.extend(docs)
        return all_docs

    def synthesize_answer(self, question: str, context_chunks: List[Document]) -> str:
        """
        Generate final answer from relevant document chunks using LLM.

        Args:
            question: Original user question
            context_chunks: Retrieved chunks

        Returns:
            Final synthesized answer
        """
```

```
            from langchain_google_genai import ChatGoogleGenerativeAI
            from langchain.chains import LLMChain
            from langchain.prompts import PromptTemplate

            llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0.2)

            context = "\n\n".join([doc.page_content for doc in context_chunks])

            synth_prompt = PromptTemplate(
                input_variables=["question", "context"],
                template="""You are a financial analysis assistant. Given the context extracted from 10-K filings, answer the fol

    Context:
    {context}

    Question:
    {question}

    Answer:
    """
            )
            synthesis_chain = LLMChain(llm=llm, prompt=synth_prompt)
            final_answer = synthesis_chain.run({"question": question, "context": context})
            return final_answer

        def run_query_pipeline(self, user_query: str, top_k: int = 5) -> str:
            """
            Complete agent pipeline: decompose → retrieve → synthesize.

            Args:
                user_query: Original user question
                top_k: Top results per sub-query

            Returns:
                Final answer string
            """
            subqueries = self.decompose_query(user_query)
            docs = self.multi_step_retrieve(subqueries, top_k=top_k)
            answer = self.synthesize_answer(user_query, docs)
            return answer


    agent = QueryEngineAgent(qdrant_db=qdrant_db)


    query = "How did NVIDIA’s data center revenue grow from 2022 to 2023?"
    response = agent.run_query_pipeline(user_query=query, top_k=10)
    print("\nFinal Answer:\n")
    response
```

```
/tmp/ipython-input-10-63445993.py:60: LangChainDeprecationWarning: The class `LLMChain` was deprecated in LangChain 0.1.
  decompose_chain = LLMChain(llm=llm, prompt=decompose_prompt)
/tmp/ipython-input-10-63445993.py:61: LangChainDeprecationWarning: The method `Chain.run` was deprecated in langchain 0.
  decomposition = decompose_chain.run(complex_query)

Final Answer:

'NVIDIA's Data Center revenue increased by 41% from fiscal year 2022 to fiscal year 2023. Data Center revenue for fisca
l year 2023 was $15.01 billion, up from $10.613 billion in fiscal year 2022.'
```

```
# can do formating but not giving out proper answer for Complex queries


Start coding or generate with AI.
```