How to leverage GenAI, moving away from zero-shot prompting

# What is zero-shot prompting?

* Zero shot prompting is when we provide a single prompt, expecting an LLM to perform a task, which is often complex in nature. For example: write me a 1500 word essay on the benefits of a free floating currency.
* Zero shot prompting is not realistic, and often leads to poor results.
* In real world terms it is like asking someone to write an essay in one go, start to finish, without the use of the delete or backspace key.

# Few-shot prompting

* The primary difference between zero-shot and few-shot prompting is the amount of task-specific data given to guide the model to help it understand and perform the task.
* The results are far superior, and smaller more cost-effective models can outperform more complex models.

# An example of few-shot prompting

## Prompt 1

Acting as a business systems analyst, write detailed user requirements for the following high level business requirements: Create a web application for Document Loading and Retrieval-Augmented Generation (RAG) The application should have a user-friendly interface designed to load documents from a pre-defined folder called “data”, process these documents to extract their content, including text contained within images within documents, and utilize a local LLM to create a Retrieval-Augmented Generation (RAG) solution. The main functionalities of the application include: Document Loading: At launch, the system will check to see if any files in the data folder are new or have changed. If there are files that are new or have changed, they are loaded into the persistent vector DB. Otherwise, the application loads normally. Documents in the data folder include PDF, PDF with images, TXT, CSV, DOC, DOCX, PPT, PPTX, HTM, and HTML. When extracting content from files, text is split into manageable chunks and embedded in a vector store, which allows for efficient similarity searches. The vector store is persisted to avoid reprocessing documents every time the application is restarted. While running, the application also checks for new documents in the folder and updates the vector store hourly. Query Processing: Users can input their queries into the application, and the local LLM generates responses based on the document content stored in the vector store. Results should also include citation to the documents used to generate the response. The LLM interacts with the vector store to retrieve relevant document chunks and generate coherent, contextually accurate responses.

## Result of Prompt 1

**User Requirements Document**

**Project: Document Loading and Retrieval-Augmented Generation (RAG) Web Application**

**Version: 1.0**

**Date: [Current Date]**

**Prepared by: [Your Name]**

**1. Introduction**

This document outlines the detailed user requirements for developing a web application for Document Loading and Retrieval-Augmented Generation (RAG). The application will enable users to load documents from a predefined folder, process these documents to extract content, and use a local LLM to generate responses based on the document content. The main functionalities include Document Loading and Query Processing.

**2. User Requirements**

**2.1. General Requirements**

1. **User-Friendly Interface**
   * The application must have an intuitive and user-friendly interface to facilitate easy interaction for users with varying technical skills.
   * The interface should be responsive and accessible from multiple devices (desktop, tablet, mobile).

**2.2. Document Loading**

1. **Initial Document Loading**
   * Upon launching the application, it must check for new or modified files in the predefined "data" folder.
   * Supported document formats: PDF, PDF with images, TXT, CSV, DOC, DOCX, PPT, PPTX, HTM, HTML.
   * New or changed documents should be loaded into a persistent vector database (vector DB).
2. **Content Extraction**
   * The application must extract text from all supported document types.
   * For documents containing images, the application must use OCR (Optical Character Recognition) to extract text from images.
3. **Text Chunking and Embedding**
   * Extracted text should be split into manageable chunks.
   * Each chunk must be embedded into the vector store to allow for efficient similarity searches.
4. **Persistence**
   * The vector store must be persisted to avoid reprocessing documents upon each application restart.
   * The application should automatically check for new documents in the "data" folder and update the vector store every hour.

**2.3. Query Processing**

1. **User Query Input**
   * Users must be able to input their queries through a user-friendly query interface.
   * The query interface should support natural language input.
2. **Response Generation**
   * The local LLM should interact with the vector store to retrieve relevant document chunks based on the user query.
   * The LLM must generate coherent, contextually accurate responses.
3. **Citations**
   * The application must provide citations for the documents used to generate each response.
   * Citations should include document titles, authors (if available), and relevant page numbers or sections.

**2.4. Performance and Reliability**

1. **Performance**
   * The application should load and process documents efficiently, minimizing the delay for users.
   * Query processing and response generation should occur in a timely manner, providing results within a reasonable timeframe.
2. **Reliability**
   * The system must be robust, handling large volumes of documents and queries without crashing or significant performance degradation.
   * Regular updates to the vector store must not interrupt the normal operation of the application.

**2.5. Security**

1. **Data Security**
   * All documents and extracted content must be stored securely.
   * Access to the application and its features should be restricted to authorized users.
2. **User Authentication**
   * The application should support user authentication mechanisms to ensure only authorized users can load documents and generate queries.

**2.6. User Support**

1. **Help and Documentation**
   * The application should provide comprehensive help and documentation to assist users in understanding and using the system effectively.
   * Tooltips and contextual help should be available within the application interface.

**3. Functional Requirements**

The functional requirements will detail the specific capabilities and behaviors the application must exhibit. These requirements will be outlined in subsequent sections.

**Approval**

Prepared by: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Approved by: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This document will be the basis for the design, development, and testing of the Document Loading and Retrieval-Augmented Generation (RAG) web application. Further details and modifications may be made as the project progresses.

## Prompt 2

Acting as a technical analyst, write a technical specifications document for these requirements.

## Result of Prompt 2

**Technical Specifications Document**

**Project: Document Loading and Retrieval-Augmented Generation (RAG) Web Application**

**Version: 1.0**

**Date: [Current Date]**

**Prepared by: [Your Name]**

**1. Introduction**

This technical specifications document outlines the architecture, components, and implementation details required to develop a web application for Document Loading and Retrieval-Augmented Generation (RAG). The application will facilitate document loading from a predefined folder, process the documents, and use a local LLM to generate responses based on the document content.

**2. System Architecture**

**2.1. Overview**

The system architecture consists of the following key components:

* User Interface (UI)
* Document Loader
* Content Extractor
* Vector Store
* Local LLM (Language Model)
* Query Processor
* Persistent Storage
* Scheduler

**2.2. Component Diagram**

**3. Technical Specifications**

**3.1. User Interface (UI)**

* **Framework:** React.js
* **Features:**
  + Responsive design for multi-device compatibility
  + Query input field with natural language support
  + Display of response with citations
  + Real-time status updates for document loading

**3.2. Document Loader**

* **Language:** Python
* **Libraries:** Watchdog, SQLAlchemy
* **Features:**
  + Monitors the "data" folder for new or modified files
  + Triggers content extraction and updates the vector store
  + Runs at application startup and checks for new documents hourly

**3.3. Content Extractor**

* **Language:** Python
* **Libraries:** PyPDF2, Tesseract (OCR), Python-docx, Pandas, BeautifulSoup
* **Features:**
  + Extracts text from various document formats (PDF, DOCX, TXT, CSV, PPTX, HTML)
  + Uses OCR for extracting text from images within documents
  + Splits extracted text into manageable chunks

**3.4. Vector Store**

* **Library:** FAISS (Facebook AI Similarity Search)
* **Features:**
  + Stores embedded text chunks for efficient similarity searches
  + Supports incremental updates with new document chunks
  + Persisted using SQLAlchemy for data integrity and persistence

**3.5. Local LLM (Language Model)**

* **Model:** OpenAI GPT-4
* **Integration:** Python API
* **Features:**
  + Generates responses based on retrieved document chunks
  + Ensures contextually accurate and coherent responses

**3.6. Query Processor**

* **Language:** Python
* **Libraries:** Flask (for API), Transformers
* **Features:**
  + Accepts user queries via the UI
  + Interacts with the vector store to retrieve relevant document chunks
  + Utilizes the local LLM to generate and return responses
  + Includes citations for the documents used in the response

**3.7. Persistent Storage**

* **Database:** SQLite or PostgreSQL (depending on scale)
* **Libraries:** SQLAlchemy
* **Features:**
  + Stores vector embeddings and metadata for documents
  + Ensures data is persisted between application restarts

**3.8. Scheduler**

* **Library:** APScheduler
* **Features:**
  + Schedules hourly checks for new documents in the "data" folder
  + Triggers the Document Loader for processing and updating the vector store

**4. Implementation Details**

**4.1. Document Loading Workflow**

1. **File Monitoring:**
   * Use Watchdog to monitor the "data" folder for any new or modified files.
2. **Content Extraction:**
   * Depending on the file type, use appropriate libraries to extract text.
   * For PDFs and images, use PyPDF2 and Tesseract for OCR.
3. **Text Chunking:**
   * Split the extracted text into chunks of manageable size (e.g., 500 words).
4. **Embedding:**
   * Use a pre-trained language model to embed text chunks into vectors.
5. **Storage:**
   * Store the vectors in FAISS and metadata in a SQLite/PostgreSQL database.

**4.2. Query Processing Workflow**

1. **Query Input:**
   * User inputs a query through the UI.
2. **Vector Retrieval:**
   * Query is transformed into a vector and matched against stored vectors in FAISS.
3. **Response Generation:**
   * Retrieve relevant document chunks and pass them to the local LLM.
   * Generate a response and include citations from the metadata.
4. **Response Display:**
   * Display the response and citations in the UI.

**5. Performance Considerations**

* **Scalability:**
  + Use PostgreSQL for larger datasets and multiple concurrent users.
* **Optimization:**
  + Optimize vector searches using FAISS indexing.
* **Caching:**
  + Implement caching for frequently accessed documents and queries.

**6. Security Considerations**

* **Authentication:**
  + Implement user authentication using OAuth or JWT.
* **Data Security:**
  + Encrypt sensitive data in storage and during transmission.
* **Access Control:**
  + Restrict access to document loading and query processing based on user roles.

**7. Testing and Validation**

* **Unit Testing:**
  + Write unit tests for each component using pytest.
* **Integration Testing:**
  + Test the end-to-end workflow from document loading to query response.
* **Performance Testing:**
  + Measure response times and scalability under load.

**8. Deployment**

* **Environment:**
  + Use Docker for containerized deployment.
* **CI/CD:**
  + Set up continuous integration and deployment pipelines using GitHub Actions or Jenkins.

**9. Documentation**

* **User Guide:**
  + Provide a comprehensive user guide covering all features.
* **Technical Documentation:**
  + Document the codebase, APIs, and deployment process.

**Approval**

Prepared by: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Approved by: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This technical specifications document serves as a detailed blueprint for the development, implementation, and deployment of the Document Loading and Retrieval-Augmented Generation (RAG) web application. Further adjustments and refinements may be made during the development process to ensure optimal functionality and performance.

## Prompt 3

Acting as a software developer, write the code for this application.

## Result of prompt 3

Given the comprehensive nature of the application, we can break down the development into several key parts:

1. **Monitoring the "data" folder for new or modified files**
2. **Extracting content from various document formats**
3. **Processing and embedding text into vectors**
4. **Storing vectors in a persistent vector database**
5. **Query processing and response generation**
6. **Building the user interface**

We'll start by setting up the project structure and then implement each component step by step. Here is the overall project structure:

lua

Copy code

rag\_web\_app/

|-- data/

|-- app/

| |-- \_\_init\_\_.py

| |-- main.py

| |-- document\_loader.py

| |-- content\_extractor.py

| |-- vector\_store.py

| |-- query\_processor.py

|-- static/

|-- templates/

|-- Dockerfile

|-- requirements.txt

|-- run.py

### 1. Setting up the Project

First, let's set up the project dependencies in requirements.txt:

txt

Copy code

Flask==2.1.2

watchdog==2.1.9

PyPDF2==1.26.0

pytesseract==0.3.8

python-docx==0.8.11

pandas==1.3.5

beautifulsoup4==4.10.0

faiss-cpu==1.7.1

SQLAlchemy==1.4.32

transformers==4.19.4

torch==1.11.0

APScheduler==3.7.0

openai==0.18.1

### 2. Implementing the Document Loader

app/document\_loader.py:

python

Copy code

import os

from watchdog.observers import Observer

from watchdog.events import FileSystemEventHandler

from app.content\_extractor import ContentExtractor

from app.vector\_store import VectorStore

class DocumentLoader(FileSystemEventHandler):

def \_\_init\_\_(self, data\_folder, vector\_store):

self.data\_folder = data\_folder

self.vector\_store = vector\_store

self.content\_extractor = ContentExtractor()

def on\_modified(self, event):

if event.is\_directory:

return

self.process\_file(event.src\_path)

def on\_created(self, event):

if event.is\_directory:

return

self.process\_file(event.src\_path)

def process\_file(self, file\_path):

print(f"Processing file: {file\_path}")

text\_chunks = self.content\_extractor.extract\_content(file\_path)

if text\_chunks:

self.vector\_store.store\_vectors(text\_chunks)

def monitor\_folder(data\_folder, vector\_store):

event\_handler = DocumentLoader(data\_folder, vector\_store)

observer = Observer()

observer.schedule(event\_handler, data\_folder, recursive=True)

observer.start()

try:

while True:

time.sleep(1)

except KeyboardInterrupt:

observer.stop()

observer.join()

### 3. Implementing the Content Extractor

app/content\_extractor.py:

python

Copy code

import os

import pytesseract

from PyPDF2 import PdfFileReader

from docx import Document

from bs4 import BeautifulSoup

class ContentExtractor:

def extract\_content(self, file\_path):

extension = os.path.splitext(file\_path)[1].lower()

if extension in ['.pdf']:

return self.extract\_from\_pdf(file\_path)

elif extension in ['.txt']:

return self.extract\_from\_txt(file\_path)

elif extension in ['.doc', '.docx']:

return self.extract\_from\_docx(file\_path)

elif extension in ['.htm', '.html']:

return self.extract\_from\_html(file\_path)

# Add more formats as needed

return []

def extract\_from\_pdf(self, file\_path):

text\_chunks = []

pdf\_reader = PdfFileReader(file\_path)

for page\_num in range(pdf\_reader.getNumPages()):

page = pdf\_reader.getPage(page\_num)

text = page.extract\_text()

if text:

text\_chunks.append(text)

return text\_chunks

def extract\_from\_txt(self, file\_path):

with open(file\_path, 'r', encoding='utf-8') as file:

text = file.read()

return [text]

def extract\_from\_docx(self, file\_path):

doc = Document(file\_path)

text\_chunks = [p.text for p in doc.paragraphs if p.text]

return text\_chunks

def extract\_from\_html(self, file\_path):

with open(file\_path, 'r', encoding='utf-8') as file:

soup = BeautifulSoup(file, 'html.parser')

text = soup.get\_text()

return [text]

def extract\_text\_from\_image(self, image):

return pytesseract.image\_to\_string(image)

### 4. Implementing the Vector Store

app/vector\_store.py:

python

Copy code

import faiss

import numpy as np

import pickle

from sqlalchemy import create\_engine, Column, Integer, String, LargeBinary

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

Base = declarative\_base()

class VectorData(Base):

\_\_tablename\_\_ = 'vectors'

id = Column(Integer, primary\_key=True)

text = Column(String)

vector = Column(LargeBinary)

class VectorStore:

def \_\_init\_\_(self, db\_url):

self.engine = create\_engine(db\_url)

Base.metadata.create\_all(self.engine)

self.Session = sessionmaker(bind=self.engine)

self.index = faiss.IndexFlatL2(768) # Assuming embedding size of 768

def store\_vectors(self, text\_chunks):

session = self.Session()

for chunk in text\_chunks:

vector = self.embed\_text(chunk)

self.index.add(np.array([vector]))

vector\_data = VectorData(text=chunk, vector=pickle.dumps(vector))

session.add(vector\_data)

session.commit()

def embed\_text(self, text):

# Placeholder for embedding function

return np.random.rand(768)

def search\_vectors(self, query\_vector, top\_k=5):

D, I = self.index.search(np.array([query\_vector]), top\_k)

session = self.Session()

results = []

for i in I[0]:

if i == -1:

continue

vector\_data = session.query(VectorData).get(i + 1)

results.append(vector\_data.text)

return results

### 5. Implementing the Query Processor

app/query\_processor.py:

python

Copy code

from flask import Flask, request, jsonify

from transformers import pipeline

from app.vector\_store import VectorStore

app = Flask(\_\_name\_\_)

vector\_store = VectorStore('sqlite:///vectors.db')

qa\_pipeline = pipeline('question-answering')

@app.route('/query', methods=['POST'])

def query():

data = request.json

query = data.get('query')

query\_vector = vector\_store.embed\_text(query)

relevant\_chunks = vector\_store.search\_vectors(query\_vector)

context = " ".join(relevant\_chunks)

answer = qa\_pipeline(question=query, context=context)

return jsonify(answer)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

### 6. Running the Application

run.py:

python

Copy code

import threading

from app.main import app

from app.vector\_store import VectorStore

from app.document\_loader import monitor\_folder

def start\_monitoring():

vector\_store = VectorStore('sqlite:///vectors.db')

monitor\_folder('data', vector\_store)

if \_\_name\_\_ == '\_\_main\_\_':

monitor\_thread = threading.Thread(target=start\_monitoring)

monitor\_thread.start()

app.run(debug=True)

### 7. Dockerfile

dockerfile

Copy code

FROM python:3.8-slim

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "run.py"]

### 8. Templates and Static Files

You can create HTML templates and static files (CSS, JavaScript) as needed for the UI in the templates/ and static/directories.

This code provides a basic framework for the Document Loading and Retrieval-Augmented Generation (RAG) web application. Each component can be further developed and refined to meet specific requirements and enhance functionality.

## Prompt 4

Acting as a code reviewer, please review the code to find areas of improvement.

## Result of prompt 4