# Building a Diet and Wellness AI Agent: A Comprehensive Guide

## Introduction

In today's fast-paced world, maintaining a healthy lifestyle through proper diet and wellness is more important than ever. With an overwhelming amount of information available, it can be challenging to find accurate and personalized advice. This guide aims to build an AI agent that can answer queries related to fitness, health, calories, and more, leveraging reasoning capabilities to provide accurate and contextual responses.



## Problem Statement

Responding to health-related questions often requires more than just retrieving information; it necessitates reasoning and contextual understanding. For example, determining whether a particular diet is suitable for someone with specific health conditions involves evaluating multiple factors. Therefore, an AI agent with robust reasoning capabilities is essential to provide meaningful and accurate advice in the domain of diet and wellness.

## Code and Guide

### Code

https://github.com/heathbrew/Building-a-Diet-and-Wellness-AI-Agent-A-Comprehensive-Guide

Git clone this repo [1] and follow along for the set-up.

### Dataset

For this guide, we will use the PUBHEALTH dataset. This comprehensive dataset is designed for explainable automated fact-checking of public health claims, making it ideal for training our AI agent. Each instance in the dataset includes a veracity label (true, false, unproven, mixture) and an explanation text field that justifies the claim's veracity label [2].

### Dataset Summary

**Dataset Card for PUBHEALTH**:

* **Dataset Summary**: PUBHEALTH is a comprehensive dataset for explainable automated fact-checking of public health claims. Each instance has a veracity label and an explanation text field.
* **Languages**: The text in the dataset is in English.

### Dataset Creation & Curation Rationale

The dataset was created to explore fact-checking of difficult to verify claims i.e., those which require expertise from outside of the journalistics domain, in this case biomedical and public health expertise.

It was also created in response to the lack of fact-checking datasets which provide gold standard natural language explanations for verdicts/labels.

<https://huggingface.co/datasets/health_fact>

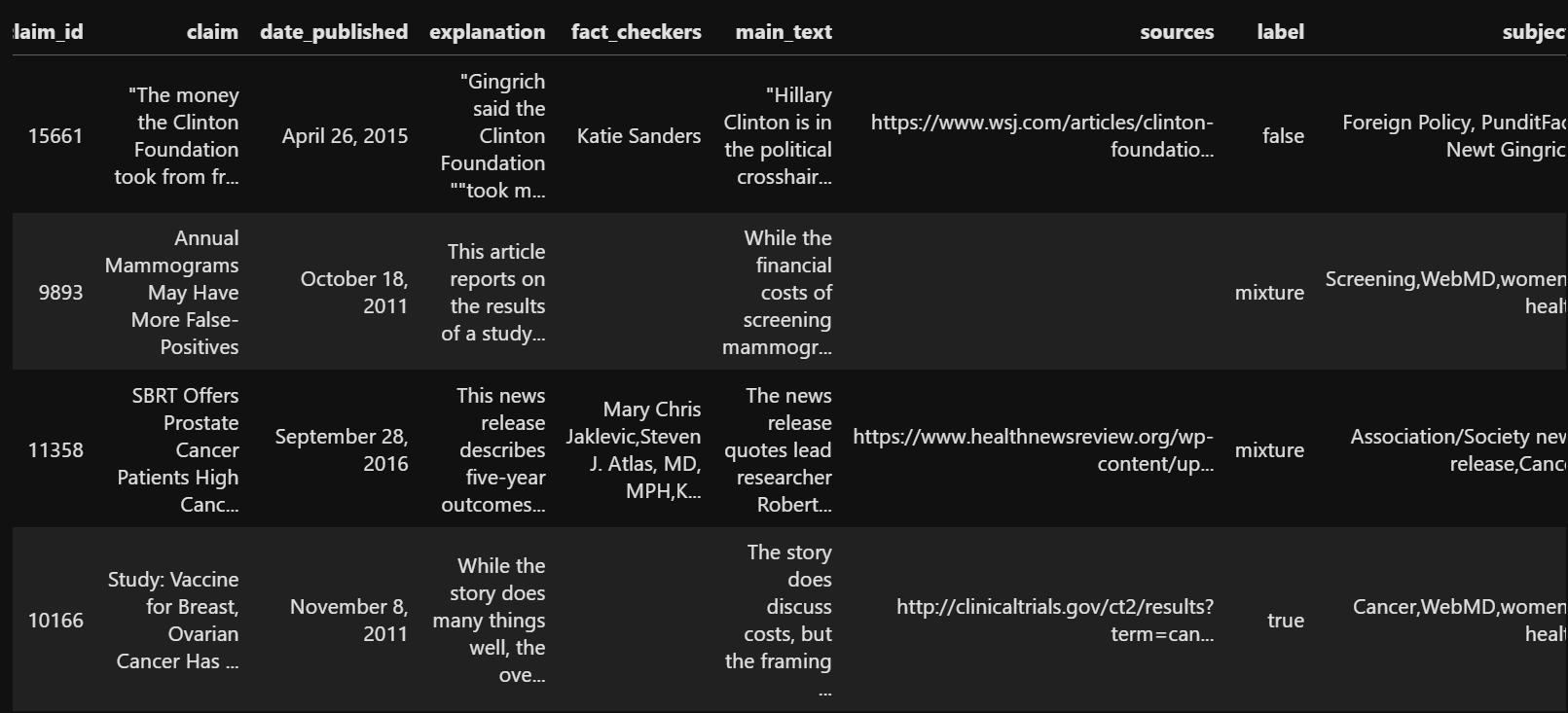
import pandas as pd

# Read the TSV file into a pandas DataFrame

df = pd.read\_csv("Dataset/train.tsv", sep='\t')

# Now you can work with the DataFrame as usual

df.head()



As we are using react agent so will consider claim and explanation from the dataset

# Fill NaN values with empty strings

df['claim'] = df['claim'].fillna('')

df['explanation'] = df['explanation'].fillna('')

df['merged\_data'] = df.apply(lambda row: row['claim'] + ' ' + row['explanation'], axis=1)

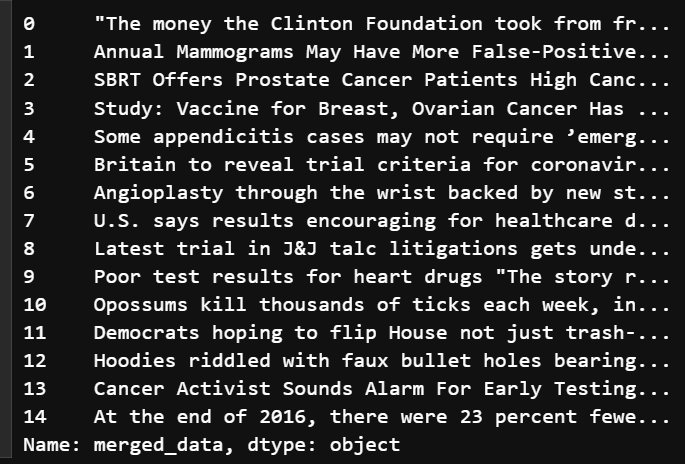
# Create dfcopy with 15 rows

dfcopy = df.head(15).copy()

# Display the first few rows

dfcopy.head()

dfcopy['merged\_data']



## **Creating a Qdrant Database**

In the last project [4], I used [Qdrant](https://qdrant.tech/) which is a locally supported vector store for RAG.

<https://medium.com/@AyushmanPranav/anomaly-detection-using-vector-search-c3f48f8e61eb>

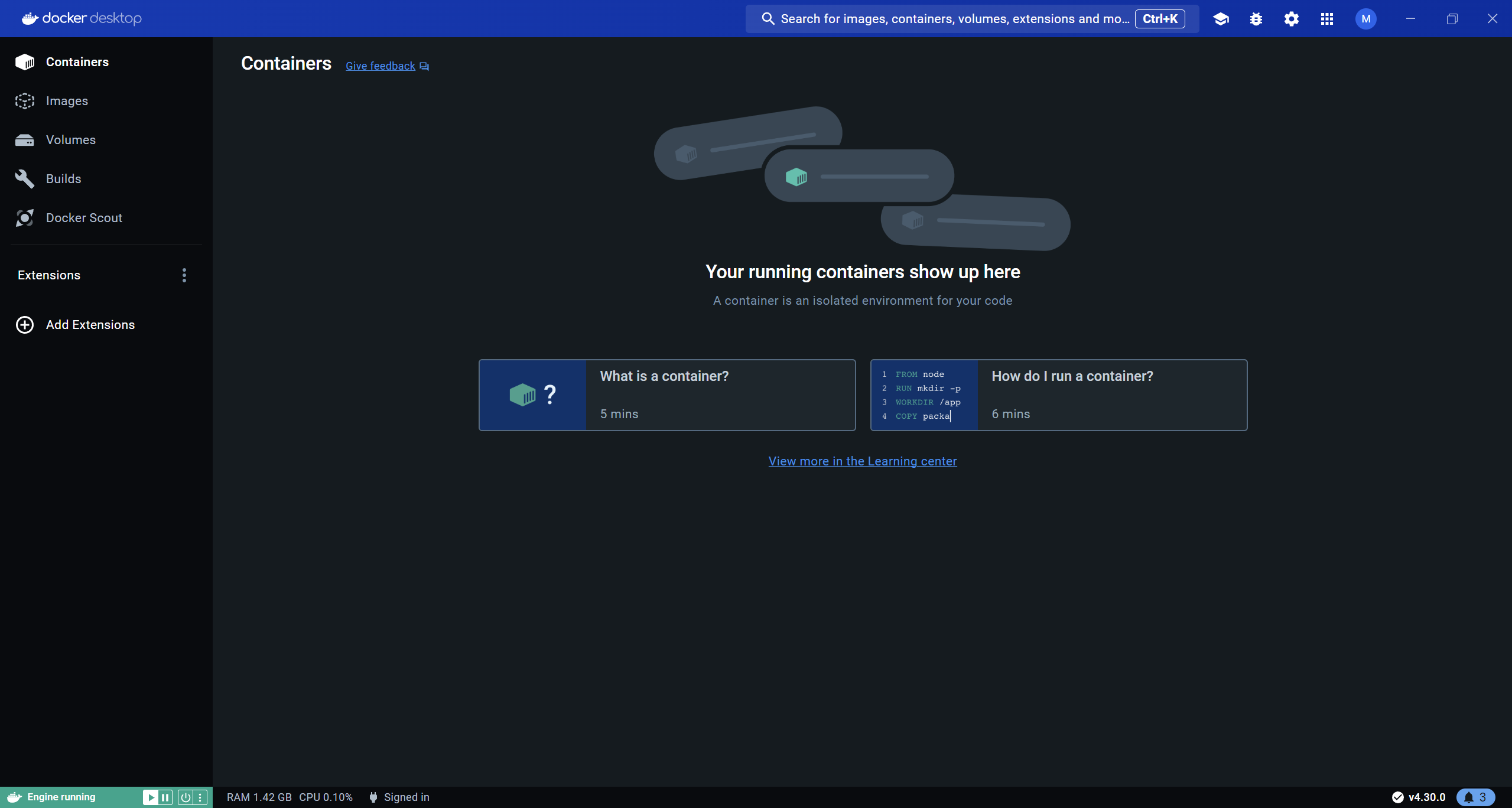
For this project, I will once again use Qdrant, but this time for financial data, to create a vector store.

Docker Desktop must be installed.

In [1] I have included a file named Qdrant.ps1.

You can use this to pull the Qdrant image and then run it on port 6333. You will see the local UI, which displays the vector stores.

For this ensure you have docker desktop installed



**Implementation**

We will use the MiniLM 12 model for embedding and Qdrant for vector storage and retrieval. Here’s how you can set it up:

1. **Loading and Preprocessing the Dataset**:

python

Copy code

import pandas as pd from datasets import load\_dataset dataset = load\_dataset('pubhealth') df = pd.DataFrame(dataset['train'])

1. **Embedding the Data**:

python

Copy code

from sentence\_transformers import SentenceTransformer model = SentenceTransformer('sentence-transformers/all-MiniLM-L12-v2') df['embedding'] = df['main\_text'].apply(lambda x: model.encode(x))

1. **Storing Embeddings in Qdrant**:

python

Copy code

from qdrant\_client import QdrantClient from qdrant\_client.http.models import PointStruct client = QdrantClient(host="localhost", port=6333) points = [ PointStruct(id=i, vector=embedding, payload={"text": text}) for i, (embedding, text) in enumerate(zip(df['embedding'].tolist(), df['main\_text'].tolist())) ] client.upsert(collection\_name="pubhealth", points=points)

1. **Building the Retrieval-Augmented Generation (RAG) Model**:
   * Define tools for date calculations, searches, and list operations.
   * Implement the agent using LangChain’s ReAct framework for multi-hop reasoning.

**Example Implementation with LangChain ReAct**

1. **Setting Up Tools**:

python

Copy code

from langchain.tools import Tool from datetime import datetime, timedelta def get\_date\_range(past\_months): end\_date = datetime.today() start\_date = end\_date - timedelta(days=past\_months \* 30) return start\_date, end\_date date\_tool = Tool(name="DateTool", func=get\_date\_range, description="Calculates date range for a given number of past months.")

1. **Creating the ReAct Agent**:

python

Copy code

from langchain.agents import create\_react\_agent from langchain.llms import AzureOpenAI from langchain.prompts import PromptTemplate from langchain.chains import RetrievalQA from langchain.document\_loaders import TextLoader from langchain.vectorstores import Qdrant # Define LLM and embedding model llm = AzureOpenAI(model\_name="gpt-4-turbo") # Set up Qdrant as the vector store vector\_store = Qdrant( collection\_name="pubhealth", embedding\_function=model.encode, ) retriever = vector\_store.as\_retriever() # Build the RetrievalQA chain qa\_chain = RetrievalQA(llm=llm, retriever=retriever) # Create the ReAct agent react\_agent = create\_react\_agent(llm=llm, tools=[date\_tool, retriever], prompt=PromptTemplate(template="..."))

1. **Testing the Agent**:

python

Copy code

query = "Is it safe to eat expired pancake mix?" response = react\_agent({"input": query}) print(response)

**Results**

After implementing and testing the AI agent, we can observe its ability to handle complex queries related to diet and wellness by leveraging the PUBHEALTH dataset and reasoning capabilities provided by LangChain’s ReAct framework. The agent can accurately fetch and reason about information, providing detailed and contextual responses.

**Conclusion**

Building a diet and wellness AI agent with reasoning capabilities involves integrating various tools and frameworks. By using the PUBHEALTH dataset, MiniLM embeddings, Qdrant for storage, and LangChain’s ReAct framework, we can create a robust system capable of answering complex health-related queries. This guide provides a comprehensive approach to setting up such an AI agent, highlighting the importance of reasoning in delivering accurate and meaningful advice in the domain of diet and wellness.